2024
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
JAPAN

Impact of a 64-bit Vedic Multiplier on Processor, Multi-Core, and DSP Performance
64 ビット ヴェーダ乗算器がプロセッサ、マルチコア、DSP パフォーマンスに与える影響

Lakshya Miglani & Gopi Srinivas Deepala

Silicon Interfaces®
info@siliconinterfaces.com
www.siliconinterfaces.com

accellera
SYSTEMS INITIATIVE

Kon'nichiwa & Introduction

- The 64-bit binary Vedic Multiplier uses Urdhva Tiryakbhyam sutra (algorithm) discovered by Scientist Bharati Krishna Tirtha (1884-1960)
- 64ビットバイナリヴェーダ乗算器は、科学者バーラティ・クリシュナ・ティルタ（1884-1960）によって発見されたウルドヴァ・ティリヤクビヤム・スートラ（アルゴリズム）を使用します。

- A 64-bit multiplier using this algorithm shows improved efficiency and performance, crucial for Arithmetic Logic Units in processors, multi-core systems, and Digital Signal Processors (DSPs) and significantly reduces computational steps, enhancing Power, Timing, and Area (PTA) metrics

Continued…

- The sutra introduces a fixed pattern novel “vertically and crosswise” “and” and then concatenation operation to calculate partial products and add all those vector values with padded zeros to get final product, thereby simplifying multiplication and reducing computational complexity.

- 経典では、固定パターンの新しい「縦横」の「そして」と連結演算を導入し、部分積を計算し、すべてのベクトル値にゼロを埋め込んで最終積を得ることで、乗算を簡素化し、計算の複雑さを軽減しています。

Motivation

- Vedic Multipliers offer a novel, precise alternative for computation in design frameworks, addressing limitations and reducing critical errors in conventional methods.
- Most Vedic Multipliers in industry develop base components for 4-bit, 8-bit, 16-bit Vedic Multiplier then 32-bit and 64-bit Vedic Multipliers are used using these component as libraries. 業界のほとんどのヴェーダ乗算器は、4ビット、8ビット、16ビットのヴェーダ乗算器の基本コンポーネントを開発し、これらのコンポーネントをライブラリとして使用して32ビットおよび64ビットのヴェーダ乗算器が使用されます。
- This leads to more power usage, higher area and lesser speeds.
- The industry needs a more fundamental Vedic Multiplier with pure and native implementation of the algorithm (アルゴリズムのネイティブ実装) for scalable to larger bit operations as well as save power, area and increases speed.

Algorithm for Partial Product (次は日本語…)

- The steps shown in Figure 1 is used to calculate partial product for Vedic Multiplication, which is same for binary number or decimal number multiplication.
- Further we will demonstrate this for 4-bit Vedic Multiplication and using the same logic we can go to 64-bit Vedic Multiplication.

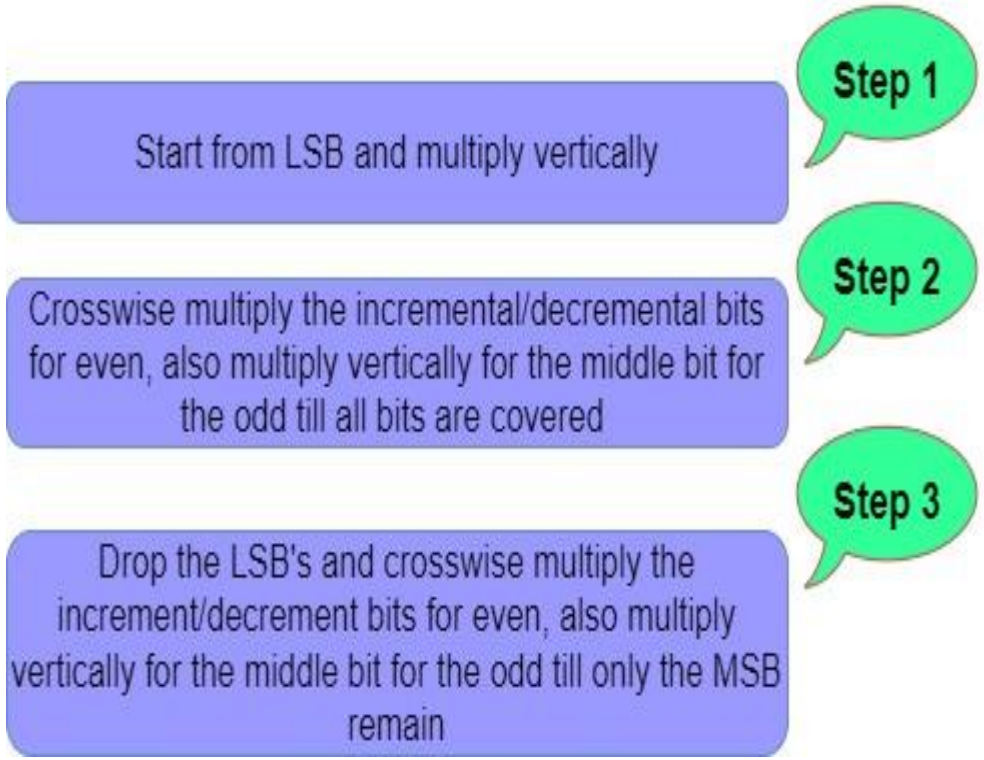

Figure 1: Steps in Vedic Multiplication to calculate partial product

部分積のアルゴリズム

ステップ1: LSBから始めて垂直に掛け算する

ステップ2: 偶数の場合は増分/減分を交差的に乗算し、奇数の場合は中央のビットを乗算してすべてのビットがカバーされるまで乗算します

ステップ3: LSBを削除し、偶数ビットの増分/減分ビットを横方向に掛け合わせ、奇数ビットの中央ビットを縦方向に掛け合わせてMSBだけが残るまで掛け合わせます。

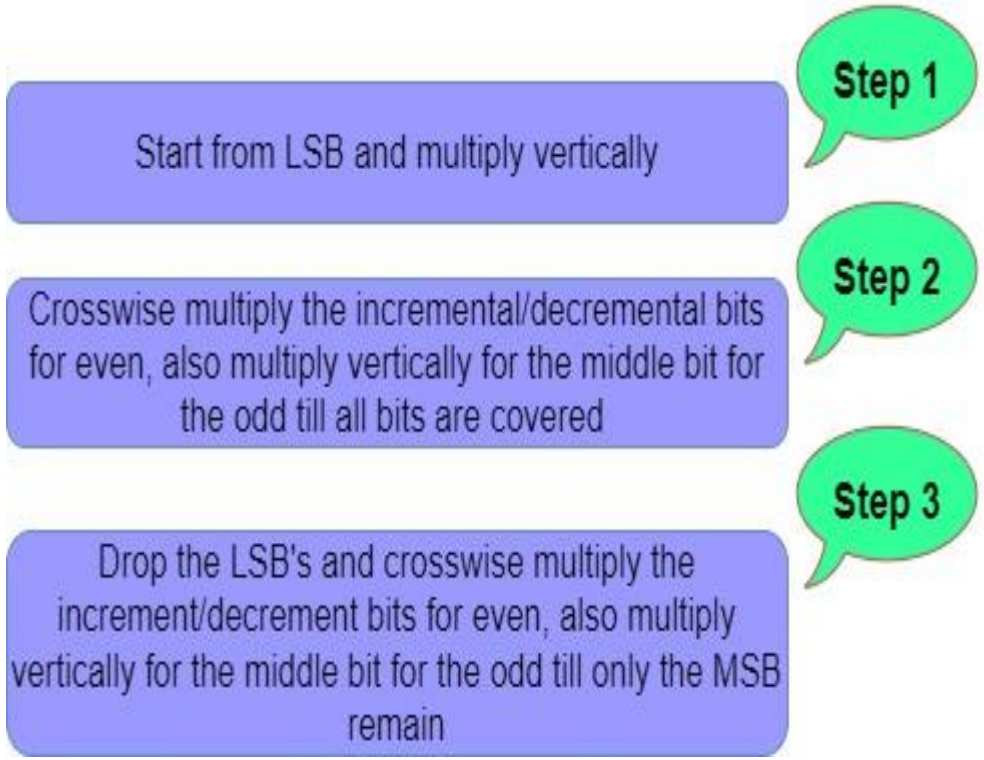

Figure 1: Steps in Vedic Multiplication to calculate partial product

Vedic Multiplication for 4-bit

1. **Partial Product**
   - The steps involved in Figure 1 for partial product which is for both decimal and binary numbers are now illustrated by a line diagram for 4-bit binary multiplication and is shown in Figure 2 to calculate partial product which is denoted by ‘k’ vectors.

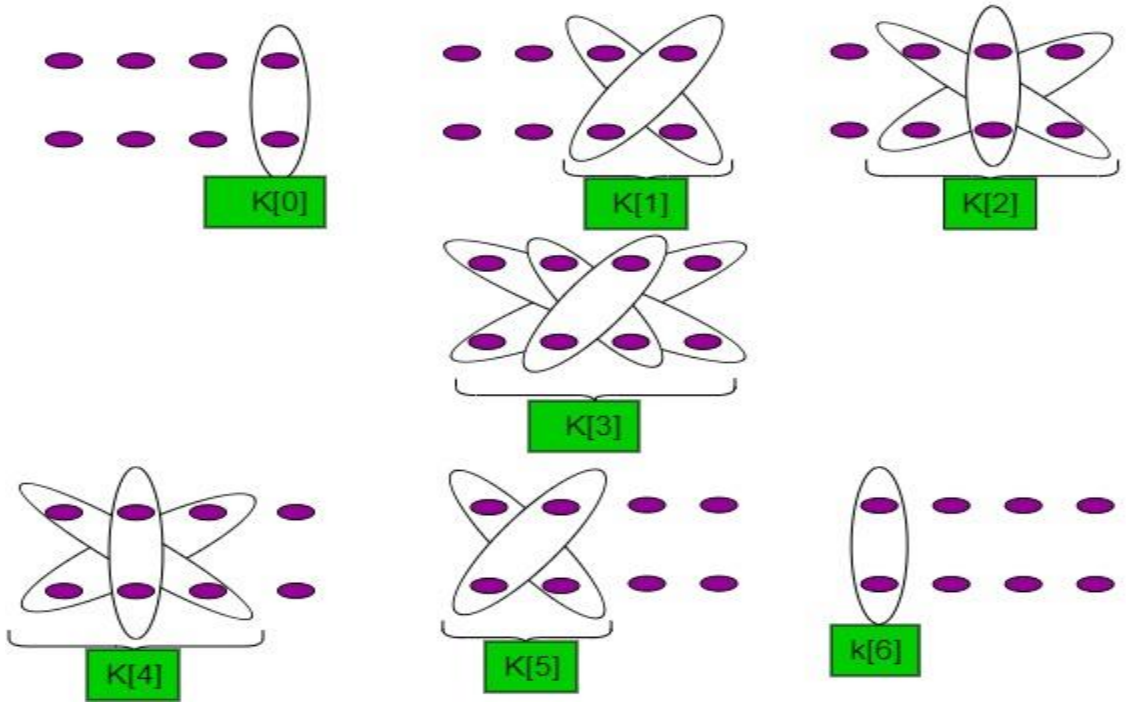

Figure 2: Line diagram for the native Urdhva Tiryagbhyam 4-bit algorithm.

Continued…

After crisscross and vertical “and operations” on both of the 4-bit binary numbers i.e. ‘a’ and ‘b’, every combination is placed using concatenation operation in different “k” values (異なる「k」値での連結演算) according to the Urdhva Tiryakbhyam sutra

```
k[0] = a[0] & b[0];

k[1] = {(a[0] & b[1]) , (a[1] & b[0])};

k[2] = {(a[0] & b[2]) , (a[1] & b[1]) , (a[2] & b[0])};

k[3] = {(a[0] & b[3]) , (a[1] & b[2]) , (a[2] & b[1]) , (a[3] & b[0])};

k[4] = {(a[1] & b[3]) , (a[2] & b[2]) , (a[3] & b[1])};

k[5] = {(a[2] & b[3]) , (a[3] & b[2])};

k[6] = {(a[3] & b[3])};
```

Figure 3: K values for 4-bit Vedic Multiplier to generate Partial Product

Continued…

2. Concatenation of Partial Products

- Then after calculating all the k values the ‘s’ vector is formed by concatenating specific bits from sequence of partial products ‘k’, ensuring correct alignment and padding.

```
s[0] = {k[6][0],k[5][0],k[4][0],k[3][0],k[2][0],k[1][0],k[0][0]};

s[1] = {1'b0,k[5][1],k[4][1],k[3][1],k[2][1],k[1][1],1'b0};

s[2] = {1'b0,1'b0,k[4][2],k[3][2],k[2][2],1'b0,1'b0};

s[3] = {1'b0,1'b0,1'b0,k[3][3],1'b0,1'b0,1'b0};
```

Figure4. Individual bit concatenation of criss-cross multiplication for 4-bit Vedic Multiplier.

Continued…

- Each ‘s[i]’ corresponds to a distinct bit position in the output, ranging from the least significant bit (‘s[0]’) to the most significant bit (‘s[3]’) for 4-bit Vedic Multiplier.
- The code strategically includes `1'b0` values to maintain proper alignment and padding within the final output structure, optimizing the computational process. (コードには、最終的な出力構造内で適切な配置とパディングを維持し、計算プロセスを最適化できるように、戦略的に `1'b0` 値が含まれています。)
- The concatenation operation is also exclusively used to ensure precise alignment throughout this process.

3. Result (product of ‘a’ and ‘b’) for 4-bit multiplication

= s[0] + s[1] +s[2]+ s[3];

Vedic Multiplication for 64-bit

1. Partial Product

```
k[0] = {a[0] & b[0]};

.............................

k[63] = { (a[0] & b[63]) , (a[1] & b[62]) , (a[2] & b[61]) , (a[3] & b[60]) , (a[4] & b[59]) , (a[5] & b[58]) , (a[6] & b[57]) , (a[7]
& b[56]) , (a[8] & b[55]) , (a[9] & b[54]) , (a[10] & b[53]) , (a[11] & b[52]) , (a[12] & b[51]) , (a[13] & b[50]) , (a[14] & b[49]) ,
(a[15] & b[48]) , (a[16] & b[47]) , (a[17] & b[46]) , (a[18] & b[45]) , (a[19] & b[44]) , (a[20] & b[43]) , (a[21] & b[42]) , (a[22] &
b[41]) , (a[23] & b[40]) , (a[24] & b[39]) , (a[25] & b[38]) , (a[26] & b[37]) , (a[27] & b[36]) , (a[28] & b[35]) , (a[29] & b[34]) ,
(a[30] & b[33]) , (a[31] & b[32]) , (a[32] & b[31]) , (a[33] & b[30]) , (a[34] & b[29]) , (a[35] & b[28]) , (a[36] & b[27]) , (a[37] &
b[26]) , (a[38] & b[25]) , (a[39] & b[24]) , (a[40] & b[23]) , (a[41] & b[22]) , (a[42] & b[21]) , (a[43] & b[20]) , (a[44] & b[19]) ,
(a[45] & b[18]) , (a[46] & b[17]) , (a[47] & b[16]) , (a[48] & b[15]) , (a[49] & b[14]) , (a[50] & b[13]) , (a[51] & b[12]) , (a[52] &
b[11]) , (a[53] & b[10]) , (a[54] & b[9]) , (a[55] & b[8]) , (a[56] & b[7]) , (a[57] & b[6]) , (a[58] & b[5]) , (a[59] & b[4]) , (a
[60] & b[3]) , (a[61] & b[2]) , (a[62] & b[1]) , (a[63] & b[0]) };

.............................

k[126] = { (a[63] & b[63]) };
```

Figure 5: Snippet for range of K values for 64-bit Vedic Multiplier to generate Partial Product

Continued..

- Using the same Urdhva Tiryagbhyam sutra (algorithm) we have generated partial product values i.e. ‘k’ for 64-bit native Vedic Multiplier as shown in Figure 5 previous slide (図5 前のスライド).
- By following the same three steps that are discussed above in Figure 1 (slides 5/6), we have the ‘k’ vector for 64-bit as:
  - In the Figure 5 starting from k[0] value that is vertically ‘and’ operation on the LSBs (k[0]はLSBに対して垂直に「AND」演算された値である).
  - The middle value k[63] in which we are covering the crisscross ‘and operation’ for all of the 64 bits and then combine them using the concatenation operation (64ビットすべてに対して「AND演算」を交差させ、連結演算を使用して結合する).
  - In the final step that is k[126] value, which is the vertically ‘and’ operation on the MSBs (MSBの垂直方向の「AND」演算).

Continued..

2. Concatenation of Partial Products

```
s[0] = { k[126][0],k[125][0],k[124][0],k[123][0],k[122][0],k[121][0],k[120][0],k[119][0],k[118][0],k[117][0],k[116][0],k[115][0],k[114][0],k[113]
[0],k[112][0],k[111][0],k[110][0],k[109][0],k[108][0],k[107][0],k[106][0],k[105][0],k[104][0],k[103][0],k[102][0],k[101][0],k[100][0],k[99][0],k[98]
[0],k[97][0],k[96][0],k[95][0],k[94][0],k[93][0],k[92][0],k[91][0],k[90][0],k[89][0],k[88][0],k[87][0],k[86][0],k[85][0],k[84][0],k[83][0],k[82]
[0],k[81][0],k[80][0],k[79][0],k[78][0],k[77][0],k[76][0],k[75][0],k[74][0],k[73][0],k[72][0],k[71][0],k[70][0],k[69][0],k[68][0],k[67][0],k[66]
[0],k[65][0],k[64][0],k[63][0],k[62][0],k[61][0],k[60][0],k[59][0],k[58][0],k[57][0],k[56][0],k[55][0],k[54][0],k[53][0],k[52][0],k[51][0],k[50]
[0],k[49][0],k[48][0],k[47][0],k[46][0],k[45][0],k[44][0],k[43][0],k[42][0],k[41][0],k[40][0],k[39][0],k[38][0],k[37][0],k[36][0],k[35][0],k[34]
[0],k[33][0],k[32][0],k[31][0],k[30][0],k[29][0],k[28][0],k[27][0],k[26][0],k[25][0],k[24][0],k[23][0],k[22][0],k[21][0],k[20][0],k[19][0],k[18]
[0],k[17][0],k[16][0],k[15][0],k[14][0],k[13][0],k[12][0],k[11][0],k[10][0],k[9][0],k[8][0],k[7][0],k[6][0],k[5][0],k[4][0],k[3][0],k[2][0],k[1]
[0],k[0][0]};

.........................

s[63] = {1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,
1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0
,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,k[63][63],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,
1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,
1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0 };
```

Figure 6: Extreme values for individual bit concatenation of partial product vector in 64-bit Vedic Multiplier

Continued..

- Concatenation of individual bits of partial product formed using crisscross multiplication is now used to form “s” vector for further process.
- Figure 6 illustrates the snippet showing the initial and last value of ‘s’ vector.
- Each ‘s[i]’ vector is formed by placing ‘k[x][y]’ vectors at a specific position (各「s[i]」ベクトルは、「k[x][y]」ベクトルを特定の位置に配置することによって形成される。) which is defined as:
  - k[x]: [Concatenation of all the ‘k’ values in the descending order and placing ‘i’ number of zeros at MSB and LSB]
  - k[y]: k[i]

Continued..

3. Result (product of ‘a’ and ‘b’) for 64-bit multiplication

- These ‘s’ vectors, collectively contributing to the final product. As shown in figure 7, the final product of two 64-bit numbers ‘a’ and ‘b’ is achieved by adding all the ‘s’ vectors (2つの64ビット数「a」と「b」の最終的な積は、すべての「s」ベクトルを加算することによって得られる。)

```
Product of two 64 bit binary number = s[0] + s[1] + s[2] + s[3] + s[4] + s[5] + s[6] + s[7] + s[8] +
s[9] + s[10] + s[11] + s[12] + s[13] + s[14] + s[15] + s[16] + s[17] + s[18] + s[19] + s[20] + s[21] +
s[22] + s[23] + s[24] + s[25] + s[26] + s[27] + s[28] + s[29] + s[30] + s[31] + s[32] + s[33] + s[34]
+ s[35] + s[36] + s[37] + s[38] + s[39] +  s[40] + s[41] + s[42] + s[43] + s[44] + s[45] + s[46] + s
[47] + s[48] + s[49] + s[50] + s[51] + s[52] + s[53] + s[54] + s[55] + s[56] + s[57] + s[58] + s[59] +
s[60] + s[61]+ s[62] + s[63];
```

Figure 7. Result for multiplication of 64 bit numbers using ‘s’ vector

RESULT

The multiplication of ‘a’ and ‘b’, with decimal values of 18,446,744,073,394,194,650 and 18,446,744,072,077,428,285 respectively, yields the product ‘re = 340,282,366,885,013,792,836,538,441,420,705,675,250’. Figure 8 confirms the consistency of this result using Vedic multiplication for 64-bit numbers with binary representations of ‘a’ and ‘b’.

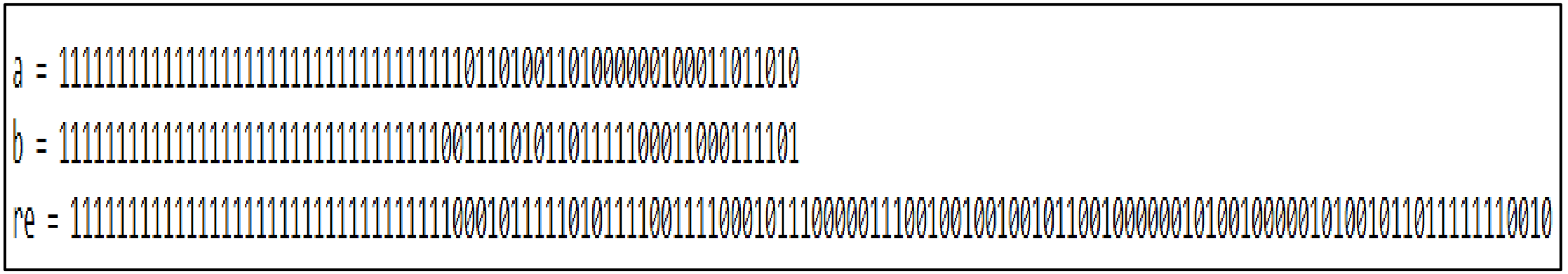

Figure 8: Simulation result Window

RESULT Continued..

- The Vedic multiplier outperforms traditional designs with fewer gates, optimizing logic for scalability and cost-efficiency. It achieves superior area efficiency, maximizing IC space, and advances in power consumption for sustainable computing.

Parameter	Booth encoded parallel multiplier	Modified Booth Encoding Multiplier	Vedic Multiplier
Gate Count	20557	17196	4604
Area (µm2)	102183	85983	23183
Power (mW)	13.7	12.1	3.24

APPLICATION

- **Digital Signal Processing:** Improves signal manipulation speed and accuracy, benefiting telecommunications and multimedia.
- **Image Processing:** Efficiently handles large datasets, and speeds up image transformation in medical imaging and computer vision.
- **Cryptography:** Ensures quick, secure multiplication for stronger financial transaction and communication encryption.
- **Communication Systems:** Facilitates seamless signal and data stream processing, enhancing network transmission efficiency.
- **Overall Impact:** The Vedic multiplier promises significant advancements in computational efficiency and digital system performance.

Conclusion

- Unlike conventional Vedic Multipliers in the industry, which use base components for 4-bit, 8-bit, and 16-bit designs that are scaled up to 32-bit and 64-bit using these components as libraries, our innovative architecture achieves a distinct advantage by reducing propagation delay to just a two-step process (伝播遅延をわずか2段階のプロセスに短縮).
- Physical synthesis results demonstrate that the proposed Vedic multiplier architecture is three times (3回) more efficient than the comparable Booth architecture.
- Our Vedic Multiplication outperforms existing methods by offering lower power consumption, smaller area requirements, and faster operation speeds (lower delay).

Arigatō Gozaimasu