Low Power Multiplication through the Urdhva Tiryagbhyam Vedic Algorithm

Low Power Digital Systems

Zack Fravel

010646947

May 2, 2017

Abstract

This paper proposes a possible solution to reducing power consumption on CMOS multiplier circuits. Specifically, this paper looks into one of sixteen of the ancient sutras in Vedic Mathematics, Urdhva Tiryagbhyam, and its application in multiplier circuits. The main concept that allows the Vedic Multiplier to be more efficient than a standard Shift-and-Add multiplier is its use of parallelism and hardware reduction in order to reduce area and thereby reduce power consumption. I designed two 16x16 multipliers, one standard shift-and-add multiplier using the "shift-and-add" approach and one Urdhva Tiryagbhyam multiplier. My expectations were unfortunately unconfirmed with my designs as I wasn't able to show that the Vedic multiplier consumes less power than an shift-and-add multiplier.

Introduction

Multipliers are used in most computing applications with many different possible algorithms and methods to go about computation. The main factor that is desirable to reduce in a multiplier is computation time. There are two different categories of multipliers, serial and parallel. Parallel multipliers reduce power by reducing the time the circuit it running per computation. Below I give the equation for power consumption in CMOS circuits.

$$P = \alpha C_L V_{DD}^2 f + \alpha t_{sc} V_{DD} I_{peak} f + V_{DD} I_{leakage}$$

The main aspect of the equation this paper's designs focus on are the reduction of power through the reduction of circuit area and therefore, C_L, or load capacitance. The load capacitance is a function of circuit area (wire length, fanout, etc). There are many different kinds of parallel

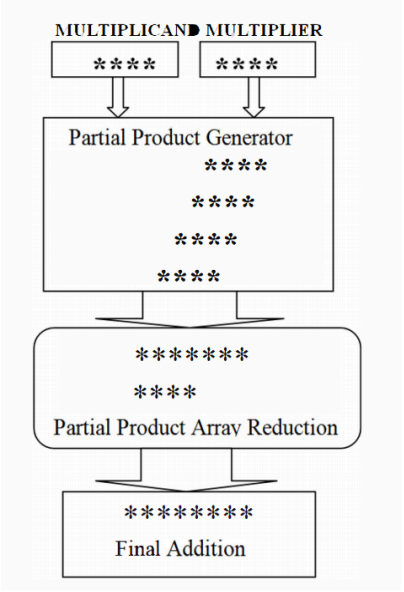
multipliers, we will be focusing on the Shift-and-add and the Vedic Urdhva Tiryagbhyam multipliers.

A total of 16 sutras (algorithms) were discovered through research of the ancient Vedas. These Vedic mathematic algorithms were meant for all kinds of computation (addition, subtraction, multiplication, and division), including being able to handle complex numbers. The algorithms all share a common trait in that they speed up the calculation through the reduction of complexity and hierarchical design. The hierarchical nature of the design makes it very easy to design a basic 2x2 bit design and from there obtain an NxN bit design.

Background

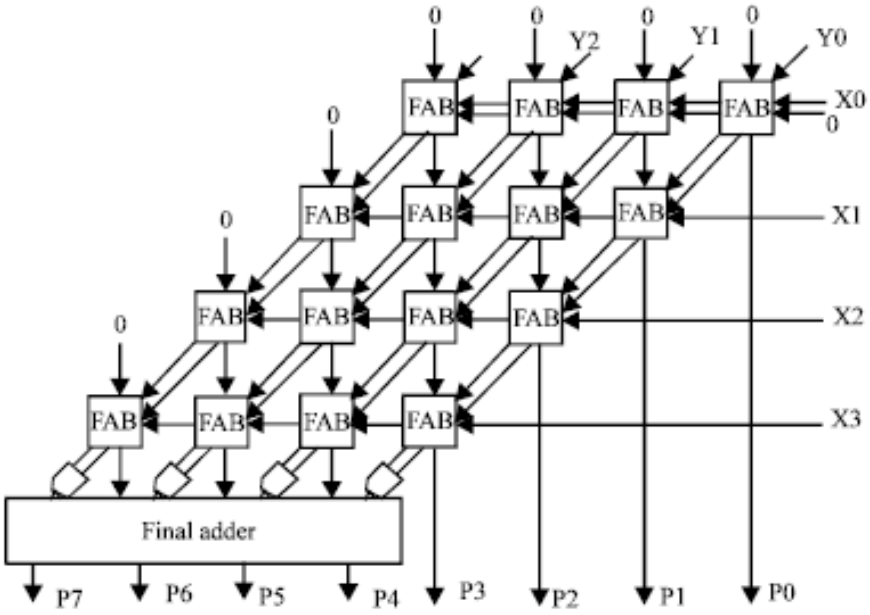
The concept of multiplication, when broken down, is just repeated addition N number of times. It would not be unreasonable to suggest that a circuit design that simply adds two numbers N number of times would not be the most efficient way of performing this sort of calculation. This would be a 'serial' calculation. A 'Parallel' calculation involves manipulating your inputs such that fewer partial products need to be generated and subsequently added together. This allows the calculation to be complete faster, reducing the time the circuit is under load, thereby reducing power.

Before going into the theory of the Vedic multiplier design, it is best to be familiar with how binary multiplication works. To the right is a diagram showing an example of binary number multiplication. These partial products are generated using the AND operator and then summed together to generate a final result. The first design we will look at is the most simple, performing 15 shifts on the multiplicand and adding up the 15 partials.



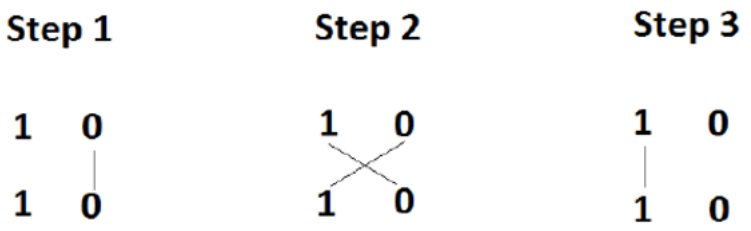
Theory

The Shift-And-Add Multiplier works by simultaneously running AND comparisons of the multiplicand's bits with all the shifted multiplier bits and adds them all together at the end of the circuit. Below is a diagram of what the circuit is doing to generate a result for a 4x4 bit multiplication. This general structure stays the same for N bits. All that is needed to expand this

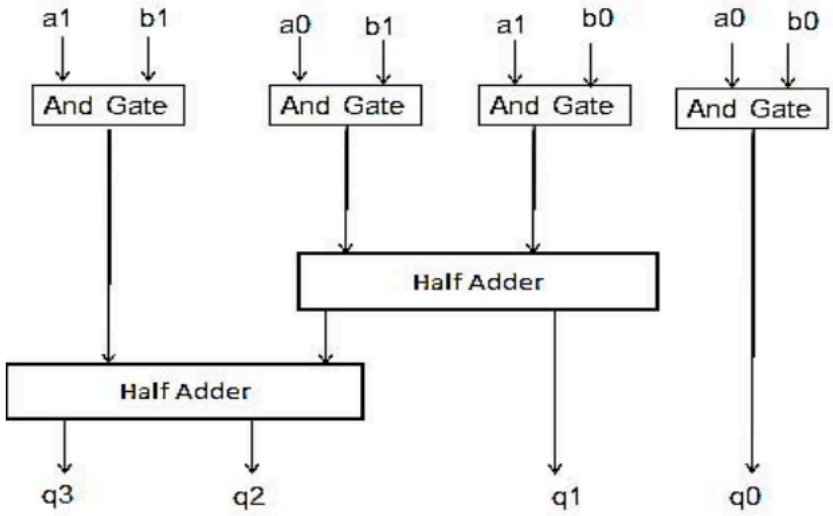


design and make larger is to increase the number of rows and columns of adders/AND gates to the number of bits you wish to multiply and connect accordingly.

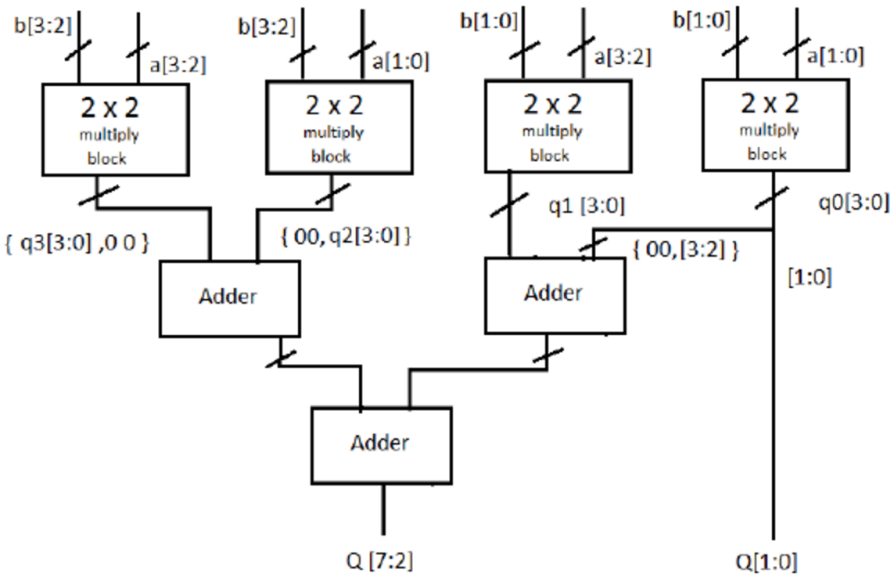
Of the two different Vedic multiplication algorithms, this paper focuses on the Urdhva Tiryagbhyam algorithm, which means literally "vertically and crosswise." This sutra simplifies multiplication by generating partial products and summation in a single iterative step. The sutra works by performing a simple comparison of two bits and then being able to use that same process to build larger and larger multiplier blocks to multiply bigger numbers. Since the actual logical comparisons are now very small, this helps to reduce the area and delay, and therefore power. On the beginning of the next page is a diagram displaying how this algorithm works.



All that is needed for this is two half adders and four AND gates, circuit diagram given below.



From this 2x2 bit Vedic multiplier, a hierarchical design can be formed to construct a 4x4 bit Vedic multiplier, as shown below.



This is the structure from which the rest of the multipliers will take form. To build an 8x8 bit you use the same exact structure, just replacing the 2x2 multiply blocks with 4x4 and adjusting the size of the partial products accordingly to accommodate 8 bits. The same thing is done to be able and perform 16x16 bit multiplication. It can be seen in the design of the Vedic multiplier that the way it splits up the partial product production allows each signal path to be the same length, the result all arrives at the same time whereas on other multiplier designs there is an adder chain that causes a delay in producing the final result. This concludes the theory discussion behind the designs, next is the implementation in Verilog.

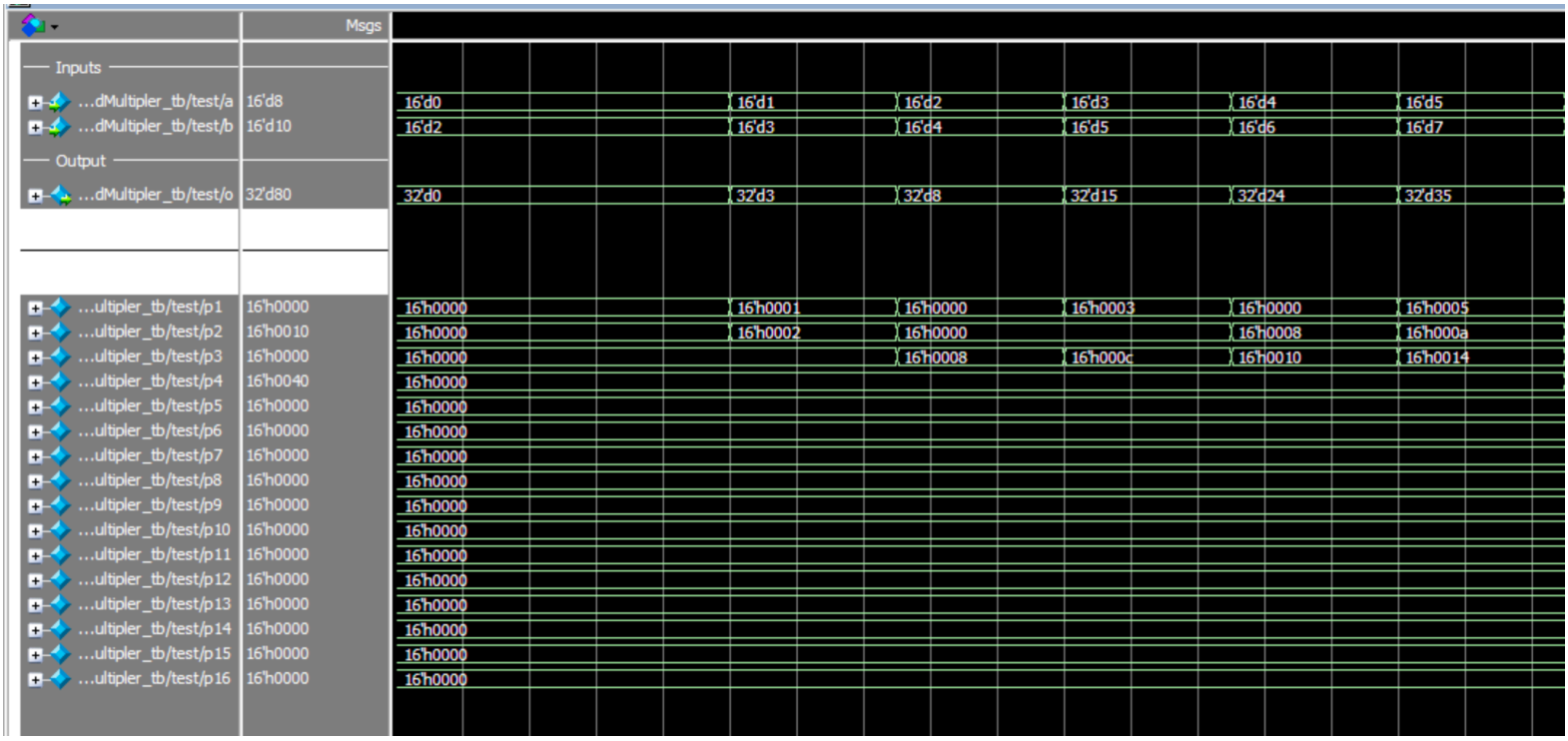
Application

Both multiplier circuits were designed in ModelSim using Verilog as the HDL of choice. First is the shift-add multiplier. Below I have given the code for the shift-add multiplier.

C:/Users/zack/Documents/Low Power Project/ShiftAddMultiplier/ShiftAddMultiplier.v - Default

```
Ln#
 1  // Zack Fravel
 2  // 16x16 Shift/Add Multiplier Verilog Implementation
 3  // Low Power Digital Systems
 4
 5  module ShiftAddMultiplier(a, b, o);
 6      input[15:0] a, b;
 7      output[31:0] o;
 8
 9      wire[15:0] p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16;
10
11      assign p1  = (b[0]==1'b1)  ? {16'h0000, a}                  : 16'b0000000000000000;
12      assign p2  = (b[1]==1'b1)  ? {15'b000000000000000, a, 1'b0} : 16'b0000000000000000;
13      assign p3  = (b[2]==1'b1)  ? {14'b00000000000000, a, 2'b00} : 16'b0000000000000000;
14      assign p4  = (b[3]==1'b1)  ? {13'b0000000000000, a, 3'b000} : 16'b0000000000000000;
15      assign p5  = (b[4]==1'b1)  ? {12'b000000000000, a, 4'b0000} : 16'b0000000000000000;
16      assign p6  = (b[5]==1'b1)  ? {11'b00000000000, a, 5'b00000} : 16'b0000000000000000;
17      assign p7  = (b[6]==1'b1)  ? {10'b0000000000, a, 6'b000000} : 16'b0000000000000000;
18      assign p8  = (b[7]==1'b1)  ? {9'b000000000, a, 7'b0000000} : 16'b0000000000000000;
19      assign p9  = (b[8]==1'b1)  ? {8'b00000000, a, 8'b00000000} : 16'b0000000000000000;
20      assign p10 = (b[9]==1'b1)  ? {7'b0000000, a, 9'b000000000} : 16'b0000000000000000;
21      assign p11 = (b[10]==1'b1) ? {6'b000000, a, 10'b0000000000} : 16'b0000000000000000;
22      assign p12 = (b[11]==1'b1) ? {5'b00000, a, 11'b00000000000} : 16'b0000000000000000;
23      assign p13 = (b[12]==1'b1) ? {4'b0000, a, 12'b000000000000} : 16'b0000000000000000;
24      assign p14 = (b[13]==1'b1) ? {3'b000, a, 13'b0000000000000} : 16'b0000000000000000;
25      assign p15 = (b[14]==1'b1) ? {2'b00, a, 14'b00000000000000} : 16'b0000000000000000;
26      assign p16 = (b[15]==1'b1) ? {1'b0, a, 15'b000000000000000} : 16'b0000000000000000;
27
28      assign o = p1+p2+p3+p4+p5+p6+p7+p8+p9+p10+p11+p12+p13+p14+p15+p16;
29
30  endmodule
31
```

It can be seen that the result is dependent on the sixteen previous conditional comparisons with the multiplicand and the shifted multiplier. I designed a testbench to run through all of the possible input combinations, below is a screenshot of the operating circuit.



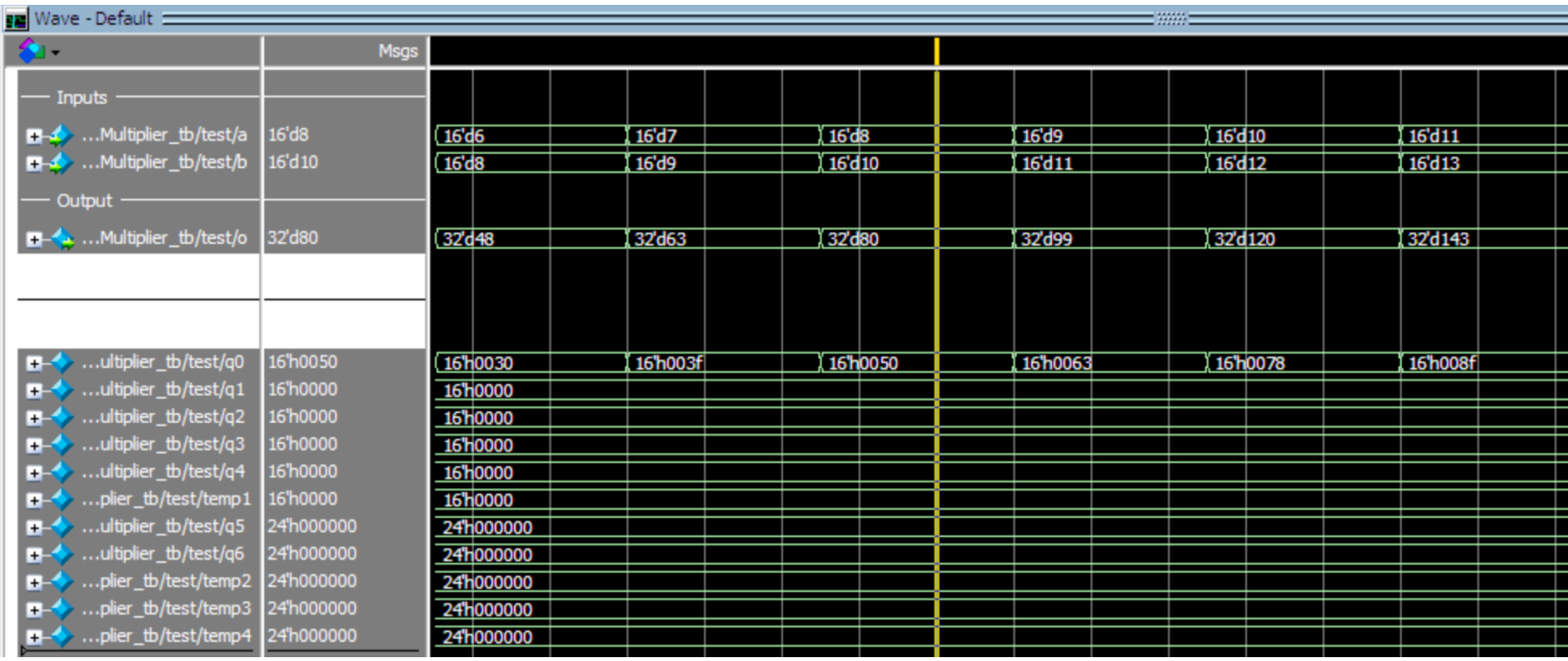
It can be observed on the third row that the output is the resulting multiplication of the two inputs on the first two top rows. The bottom shows the signals of the shifted multiplier. I have also included a snapshot of the synthesis report given by the Xilinx design tool when synthesizing the design. This details the actual blocks used in the design that has the power analysis information.

```
=========================================================================
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 15
 17-bit adder                                          : 1
 18-bit adder                                          : 1
 19-bit adder                                          : 1
 20-bit adder                                          : 1
 21-bit adder                                          : 1
 22-bit adder                                          : 1
 23-bit adder                                          : 1
 24-bit adder                                          : 1
 25-bit adder                                          : 1
 26-bit adder                                          : 1
 27-bit adder                                          : 1
 28-bit adder                                          : 1
 29-bit adder                                          : 1
 30-bit adder                                          : 1
 31-bit adder                                          : 1
# Multiplexers                                         : 16
 32-bit 2-to-1 multiplexer                             : 16

=========================================================================

=========================================================================
*                      Advanced HDL Synthesis                           *
=========================================================================


Synthesizing (advanced) Unit <ShiftAddMultiplier>.
        The following adders/subtractors are grouped into adder tree <Madd_n0186[3
        <Madd_n0144[16:0]> in block <ShiftAddMultiplier>,       <Madd_n0147[17:0]>
Unit <ShiftAddMultiplier> synthesized (advanced).

=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# Adder Trees                                          : 1
 31-bit / 16-inputs adder tree                         : 1
# Multiplexers                                         : 16
 32-bit 2-to-1 multiplexer                             : 16

=========================================================================
```

Next is the implementation of the Urdhva Tiryagbhyam multiplier. Below I have given the Verilog code for the 16x16 bit Vedic Multiplier.

```
// Zack Fravel
// 16x16 Vedic Multiplier Verilog Implementation
// Low Power Digital Systems

module VedicMultiplier(a, b, o);
    input [15:0]a, b;
    output [31:0]o;

    // internal signals
    wire [15:0]q0, q1, q2, q3, q4, temp1;
    wire [23:0]q5, q6, temp2, temp3, temp4;
    wire [31:0]o;

    // Generate Partial Products using 4 Parallel 8x8 Vedic
    multiply8 M1(a[7:0],  b[7:0],  q0[15:0]);
    multiply8 M2(a[15:8], b[7:0],  q1[15:0]);
    multiply8 M3(a[7:0],  b[15:8], q2[15:0]);
    multiply8 M4(a[15:8], b[15:8], q3[15:0]);

    // Parrallel adding up partials
    assign temp1 = {8'h00, q0[15:8]};
    assign q4    = q1[15:0] + temp1;
    assign temp2 = {8'h00, q2[15:0]};
    assign temp3 = {q3[15:0] , 8'h00};
    assign q5    = temp2 + temp3;
    assign temp4 = {8'h00, q4[15:0]};

    // Final partial product addition
    assign q6    = temp4 + q5;

    // Send product to output
    assign o[7:0]  = q0[7:0];
    assign o[31:8] = q6[23:0];
endmodule

module multiply8(a, b, o);
    input [7:0]a, b;
    output [15:0]o;

    // internal signals
    wire [7:0]q4, temp1;
    wire [11:0]q5, q6, temp2, temp3, temp4;
    wire [15:0]q0, q1, q2, q3, o;

   // Generate Partial Product using 4 Parallel 4x4 Vedic M
    multiply4 M1(a[3:0], b[3:0], q0[15:0]);
    multiply4 M2(a[7:4], b[3:0], q1[15:0]);
    multiply4 M3(a[3:0], b[7:4], q2[15:0]);
    multiply4 M4(a[7:4], b[7:4], q3[15:0]);

    // Parallel adding up partials
    assign temp1 = {4'h0, q0[7:4]};
    assign q4    = q1[7:0] + temp1;
    assign temp2 = {4'h0, q2[7:0]};
    assign temp3 = {q3[7:0], 4'h0};
    assign q5    = temp2 + temp3;
    assign temp4 = {4'h0, q4[7:0]};

    // Final addition
    assign q6    = temp4 + q5;

    // Send product to output
    assign o[3:0]  = q0[3:0];
    assign o[15:4] = q6[11:0];
endmodule

module multiply4(a, b, o);
    input [3:0]a, b;
    output [7:0]o;

    // internal signals
    wire [3:0]q0, q1, q2, q3, q4, temp1;
    wire [5:0]q5, q6, temp2, temp3, temp4;
    wire [7:0]o;

    // Generate Partial Product using 4 Parallel 2x2
    multiply2 M1(a[1:0], b[1:0], q0[3:0]);
    multiply2 M2(a[3:2], b[1:0], q1[3:0]);
    multiply2 M3(a[1:0], b[3:2], q2[3:0]);
    multiply2 M$(a[3:2], b[3:2], q3[3:0]);

    // Parallel adding up partials
    assign temp1 = {2'b00, q0[3:2]};
    assign q4    = q1[3:0] + temp1;
    assign temp2 = {2'b00, q2[3:0]};
    assign temp3 = {q3[3:0], 2'b00};
    assign q5    = temp2 + temp3;
    assign temp4 = {2'b00, q4[3:0]};

    // Final addition
    assign q6    = temp4 + q5;

    // Send product to output
    assign o[1:0] = q0[1:0];
    assign o[7:2] = q6[5:0];
endmodule

module multiply2(a, b, o);
    input [1:0]a, b;
    output [3:0]o;

    // internal signals
    wire [3:0]o, temp;

    // AND all input combinations
    assign o[0]    = a[0]&b[0];
    assign temp[0] = a[1]&b[0];
    assign temp[1] = a[0]&b[1];
    assign temp[2] = a[1]&b[1];

    // Generate Product using two Half Adders
    halfAdder HA1(temp[0], temp[1], o[1], temp[3]);
    halfAdder HA2(temp[2], temp[3], o[2], o[3]);
endmodule

module halfAdder(a, b, Sum, cOut);
    input a, b;
    output Sum, cOut;

    assign Sum  = a^b;
    assign cOut = a&b;
endmodule
```

It can be seen that even though the code is kind of long, the algorithm is very simple. It all starts at the bottom with the halfAdder module. Above it, it can be seen that the multiple two takes the inputs and runs four AND comparisons using the Vedic algorithm and adds those partials. If you follow the hierarchy up, it can be seen that the multiply4 module sets the design pattern trend for the parent modules, performing four parallel multiplications and following it up with parallel appropriate additions to generate the partials with one final addition producing the result at the end. It can be seen that the multiply8 and VedicMultiplier modules follow this exact same pattern, however the only difference being adjusting the signal sizes and inputs/outputs to accommodate for larger data. Below I have included the Xilinx synthesis report as well as a screenshot of the simulation given by ModelSim.

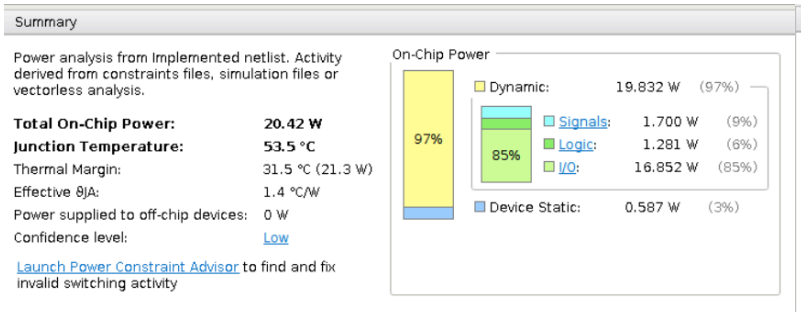


```
================================================================================
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 63
 12-bit adder                                          : 8
 16-bit adder                                          : 1
 24-bit adder                                          : 2
 4-bit adder                                           : 16
 6-bit adder                                           : 32
 8-bit adder                                           : 4
# Xors                                                 : 128
 1-bit xor2                                            : 128

================================================================================

================================================================================
*                      Advanced HDL Synthesis                                  *
================================================================================


================================================================================
Advanced HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 63
 12-bit adder                                          : 8
 16-bit adder                                          : 1
 24-bit adder                                          : 2
 4-bit adder                                           : 16
 6-bit adder                                           : 32
 8-bit adder                                           : 4
# Xors                                                 : 128
 1-bit xor2                                            : 128

================================================================================
```

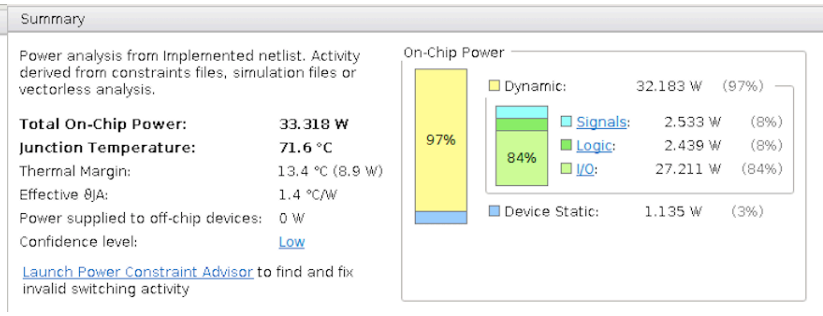
Results and Analysis

Once the designs were synthesized, the only thing left to do was to run a power analysis. I used Xilinx Vivado design suite to analyze the power of my designs. Below are the results when I ran a power analysis for the circuits.

Shift-Add



Vedic



It can be seen that my results were not actually the results I expected from my designs. The Vivado power analysis shows that my Urdhva Tiryagbhyam multiplier consumes 32.183 W of Dynamic power, which is what we're concerned with. The Shift-Add only consuming 19.832 W. The equation for dynamic power is given below.

$$P = \alpha C_L V_{DD}^2 f$$

To restate, this equation tells us that the total load capacitance is one of the primary variables of dynamic power. Earlier I discussed that the load capacitance is itself a function of the area of the circuit (larger circuit, larger capacitance). Taking a look back at the synthesis results, it can be seen that the Urdhva Tiryagbhyam design uses a good amount more hardware than the shift-add, which is the opposite from what we expect to see. There are many things that could potentially cause this, however I think what happened was the synthesis tool ended up designing a circuit that was slightly different from what was intended in the verilog code.

Conclusion

In summary, the results gathered were not exactly the expected results from the research gathered. Again, this is likely due to a misstep the synthesis tool took in producing a 'more efficient' design in the tool's eye where we were looking for a different design. To restate the original problem, this has been a study of the Urdhva Tiryagbhyam algorithm with another parallel multiplier design in an effort to achieve lower power consumption. The application of ancient Vedic algorithms to CMOS circuits in an effort to achieve more efficient designs gives designers a possible option when looking to reduce power consumption.

References

[1] C. Selvakumari, M. Jeyaprakash and A. Kavitha, "Transistor level implementation of a 8 bit multiplier using vedic mathematics in 180nm technology," *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, New Delhi, 2016, pp. 1514-1520.

[2] E. Masurkar and P. Dakhole, "Implementation of optimized vedic multiplier using CMOS technology," *2016 International Conference on Communication and Signal Processing (ICCSP)*, Melmaruvathur, 2016, pp. 0840-0844.

[3] G. S. Lakshmi, D. Fatima and B. K. Madhavi, "Compressor based 8×8 BIT vedic multiplier using reversible logic," *2016 3rd International Conference on Devices, Circuits and Systems (ICDCS)*, Coimbatore, 2016, pp. 174-178.

[4] H. Sangani, T. M. Modi and V. S. Kanchana Bhaaskaran, "Low power vedic multiplier using energy recovery logic," *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, New Delhi, 2014, pp. 640-644.

[5] H. V. R. Aradhya, H. R. Madan, M. S. Suraj, M. T. Mahadikar, R. Muniraj and M. Moiz, "Design and performance comparison of adiabatic 8-bit multipliers," *2016 IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, Mangalore, 2016, pp. 141-147.

[6] N. Fatima, B. Shrman and L. Jain, "Implementation for minimization of computation time of partial products for designing multipliers," *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*, Nagercoil, 2016, pp. 1-7.

[7] R. Kumari and R. Mehra, "Power and delay analysis of CMOS multipliers using Vedic algorithm," 2016 IEEE 1st International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES), Delhi, 2016, pp. 1-6.

[8] S. Patil, D. V. Manjunatha and D. Kiran, "Design of speed and power efficient multipliers using vedic mathematics with VLSI implementation," *2014 International Conference on Advances in Electronics Computers and Communications*, Bangalore, 2014, pp. 1-6.

[9] S. Singh and T. N. Sasamal, "Design of vedic multiplier using adiabatic logic," 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), Noida, 2015, pp. 438-441.

[10] S. Tripathy, L. B. Omprakash, S. K. Mandal and B. S. Patro, "Low power multiplier architectures using vedic mathematics in 45nm technology for high speed computing," *2015 International Conference on Communication, Information & Computing Technology (ICCICT)*, Mumbai, 2015, pp. 1-6.

[11] V. Jayaprakasan, S. Vijayakumar and V. S. Kanchana Bhaaskaran, "Evaluation of the Conventional vs. Ancient Computation Methodology for Energy Efficient Arithmetic Architecture," *2011 International Conference on Process Automation, Control and Computing*, Coimbatore, 2011, pp. 1-4.

[12] Y. Deodhe, S. Kakde and R. Deshmukh, "Design and Implementation of 8-Bit Vedic Multiplier Using CMOS Logic," *2013 International Conference on Machine Intelligence and Research Advancement*, Katra, 2013, pp. 340-344.