Hardware Implementation of an Additive Bit-Serial Algorithm for the Discrete Logarithm Modulo 2^k

L. Li*, Alex Fit-Florea, M. A. Thornton*, D. W. Matula**
Department of Computer Science and Engineering
Southern Methodist University, Dallas, TX 75275, USA
lli,alex,mitch,matula@engr.smu.edu

Abstract

We describe the hardware implementation of a novel algorithm for computing the discrete logarithm modulo 2^k. The circuit has a total latency of less than k table-lookup-determined shift-and-add modulo 2^k operations. We introduce a one-to-one mapping between k-bit binary integers and k-bit encodings of a factorization of the integers employing the discrete logarithm. We compare the physical layout result for the circuit when k=8, 16, 32, and 64.

1. Introduction

The hardware support of floating point arithmetic for high precision scientific computation has greatly advanced since the introduction of the IEEE floating point standard in the early 80's. However, the hardware capabilities for integer arithmetic have been enhanced primarily only at low precision such as where 64 and/or 128 bit words may be partitioned to support parallel addition/multiplication of four 16/32-bit precision operands.

The focus of this paper is to explore hardware implementation of the discrete logarithm modulo 2^k. Following [4] we employ modular function notation using $|n|_m = j$ to denote the congruence relation $n \equiv j \pmod{m}$ with the further restriction that j is the standard residue for modulus m $\geq$ 2 satisfying $0 \leq j \leq m-1$. Herein we restrict our investigation exclusively to moduli $m = 2^k$ for $k \geq 3$, with particular interest in time and space feasible algorithms for hardware implementation where k = 64, 128, 256, 512 or 1024. We utilize the value 3 as the logarithmic base for our discrete logarithm as it has useful properties in combination with the modulus 2^k. The discrete logarithm problem asks for determination of the minimum exponent e (when it exists) for given $0 \leq j \leq 2^k - 1$ and $k \geq 3$ satisfying $\left|3^e\right|_{2^k} = j$, and the corresponding *discrete logarithm modulo* 2^k function is denoted by $dlg(j) = e$.

Extending the hardware support to include the discrete logarithm modular 2^k will be beneficial for number systems such as the one introduced by Benschop in [3]. There, every k-bit integer N is shown to be representable by a factorization tuple (s, p, e), with $N = \left|(-1)^s 2^p 3^e\right|_{2^k}$. This makes it possible to perform multiplications as much faster and inexpensive modular additions with $N_1 \times N_2$ represented by the tuple $((-1)^{s_1+s_2}, \left|p_1 + p_2\right|_k, \left|e_1 + e_2\right|_{2^{k-2}})$. An algorithmic conversion from the standard binary representation to the number system presented in [3] is needed. The operations corresponding to conversions to-and-from are the discrete logarithm and the exponential residue operation.

Known algorithms for determination of the discrete logarithm for an arbitrary modulus are very complex. Existing algorithms such as: Pollard's ρ and λ Algorithms, the Pohlig-Hellman Algorithm, the index calculus method, and Shanks baby step - giant step Algorithm have *super polynomial running times* that are at best sub-exponential. Only Shanks baby step - giant step Algorithm is deterministic. In the recent papers [1][2], two discrete logarithm algorithms for the special case of modulo 2^k were introduced. The algorithm described in [1] uses binary arithmetic with 3 as the logarithmic base and has a critical path containing one modulo 2^k multiplication operation for each of its k iterations. Extensions of the algorithm to other logarithmic bases and computations using digits in a higher radix 2^r are also described. The algorithm in [1] was improved in [2] by replacing k modulo 2^k

* This work was supported in part by the Texas Advanced Technology Program under grant 003613-0029-2003,

** This work was supported in part by the Semiconductor Research Corporation under contract RID 1289,

Table 1. Conversion Table from the 5-bit Discrete Log Modular System to the 5-bit Integers [0,31]

Discrete Log Modular Bit String	Paritioned String			Integer Value	Standard Binary	Integer Parity
	e	e_0 xor s	2^p	$\left\|(-1)^s 2^p 3^e\right\|_{32}$		Parity
00001	000	**0**	1	1	00001	Odd
00011	000	**1**	1	31	11111	
00101	001	**0**	1	29	11101	
00111	001	**1**	1	3	00011	
01001	010	**0**	1	9	01001	
01011	010	**1**	1	23	10111	
01101	011	**0**	1	5	00101	
01111	011	**1**	1	27	11011	
10001	100	**0**	1	17	10001	
10011	100	**1**	1	15	01111	
10101	101	**0**	1	13	01101	
10111	101	**1**	1	19	10011	
11001	110	**0**	1	25	11001	
11011	110	**1**	1	7	00111	
11101	111	**0**	1	21	10101	
11111	111	**1**	1	11	01011	
00010	00	**0**	10	2	00010	Singly Even
00110	00	**1**	10	30	11110	
01010	01	**0**	10	26	11010	
01110	01	**1**	10	6	00110	
10010	10	**0**	10	18	10010	
10110	10	**1**	10	14	01110	
11010	11	**0**	10	10	01010	
11110	11	**1**	10	22	10110	
00100	0	**0**	100	4	00100	Doubly Even
01100	0	**1**	100	28	11100	
10100	1	**0**	100	20	10100	
11100	1	**1**	100	12	01100	
01000		**0**	1000	8	01000	Triply Even
11000		**1**	1000	24	11000	
10000			10000	16	10000	Quadruply Even
00000			00000	0	00000	Zero

multiplication operations with k table lookup determined shift-and-add modulo 2^k operations. The algorithm described in [2] is well suited for implementation in special purpose hardware.

The paper is organized as follows. In Section 2, we introduce a new one-to-one bit string encoding between Benschop's modular factors and standard binary. To support the determination of the encoding, we review background material, which is also a condensed version of the algorithm described in [2]. We summarize it here to make this paper self-contained. Section 3 describes our implementation. Section 4 describes experimental results and cell library information and conclusions are presented in Section 5.

2. Preliminaries and Algorithm

It is readily shown (see [1]) that the set of values $\left|3^e\right|_{2^k}$ covers half of the odd integers (1/4 of all integers) in the range [0, 2^k-1] and the complements $\left|(-1)3^e\right|_{2^k}$ yield the other half of the odd values. In particular the k-2 bit values $0 \le e \le 2^{k-2}-1$ are sufficient to uniquely represent the 2^{k-1} odd k-bit values by modular products

$\left|(-1)3^e\right|_{2^k}$ with $s \in \{0,1\}$. Note the "sign factor" $(-1)^s$ simply serves as denoting the complement since the results are the standard positive odd residues modulo 2^k. Benschop [3] used this fact to represent all integers in the range [0, 2^k-1] by factoring each integer $0 \le j \le 2^k - 1$ into even and odd parts, $j = 2^p j'$ (j' is odd). To cover all of the *k*-bit integers, the complements are included with $\left|(-1)^s 2^p 3^e\right|_{2^k}$ for $s \in \{0,1\}$, $0 \le p \le k-1$, and $0 \le e \le 2^{k-2} - 1$, providing (*s*, *p*, *e*) triples that represent each integer in the range $0 \le j \le 2^k - 1$.

Based on Benschop's modular factorization $j = \left|(-1)^s 2^p 3^e\right|_{2^k}$, we introduce here a new binary representation system that provides a one-to-one mapping between *k*-bit "discrete log" strings and standard binary *k*-bit strings. Our encoding further contains a technical detail to insure that the one-to-one mapping also satisfies an "inheritance" property. More information on the encoding and the practical value of the inheritance property in reducing the size of associated lookup tables is given in [7] and [8]. Herein we simply utilize an example conversion table to illustrate the encoding and some of its significant properties.

The one-to-one mapping between 5-bit discrete log strings and binary strings is given by the conversion table illustrated in Table 1. The string is partitioned as follows to determine the three exponents *s*, *p* and *e*. Consider the line in the table for string 10110 which yields binary 01110B=14D.

The parsing begins from the right hand side, by reading until the first unit bit is encountered, determining the variable length field identifying $10B = 2^p = 2^1$. The next bit is a separation bit providing the logical value s xor e_0. The remaining leading bits are the $3 - p$ bits of the exponent $0 \le e \le 2^{3-p} - 1$ sufficient to determine the odd part $j' = \left|(-1)^s 3^e\right|_{2^k}$ where $j = 2^p j'$ is the integer represented with $0 \le j \le 2^5 - 1$ uniquely determined. In this example, *e*=10B=2D, and then *s*=1 is determined from the encoded bits $e_0 = 0$ and s xor $e_0 = 1$, yielding $\left|(-1)^1 2^1 3^1\right|_{32} = \left|-18\right|_{32} = 14$.

Conversion from this discrete logarithm number system to a standard binary representation is possible by this procedure using the integer multiplier to evaluate the product $(-1)^s 2^p 3^e$. As discovered by Benschop, the discrete logarithm format allows integer multiplications to be handled by addition (and left shifts to accumulate the factors of 2). In [1] and [2] the authors described an efficient algorithm for determining the discrete logarithm of an odd integer which then supports the necessary conversion from binary into the discrete logarithm number system.

In this section we point out the essential mathematical properties that make the algorithm we implemented feasible. For the remainder of this paper we use $|A|_M$ as shorthand for (A mod M), a_j for the j^{th} digit of A and a_j^i for the j^{th} digit of A_i. Also, ($dlg(A)$) and ($\log_{(\beta,M)} A$) denote the discrete logarithm modulo 2^k of A with logarithmic base 3 and, respectively, the discrete logarithm modulo M with logarithmic base β of A. Let $\left|B\right|_{2^k} = \left|A^{-1}\right|_{2^k}$. In [1] it is showed that:

a. $dlg(A) + dlg(B) = 2^{k-2}$, $\forall k \ge 3$;
b. $\left|2^{k-1}+1\right|_{2^k} = \left|3^{2^{k-3}}\right|_{2^k}$, or, equivalently
$dlg(2^{k-1} + 1) = 2^{k-3}$, $\forall k \ge 4$;
c. Either $\log_{(3,2^i)}(A) = \log_{(3,2^{i-1})}(A)$, or
$\log_{(3,2^i)}(A) = \log_{(3,2^{i-1})}(A) + 2^{i-2}$, $\forall k \ge 4$

The correct selection for $\log_{(3,2^i)}(A)$ in (c) looks to be a rather complicated problem for the general case. However, it becomes trivial for the special cases A ≡ (1 mod 2^i) - when $dlg(A) = 0$, and $A \equiv (2^{i-1} + 1) \pmod{2^i}$) - when, as a direct consequence of (b), $dlg(A) = 2^{k-3}$.

Our algorithm can be abstracted as successively updating $P_{i+1} = \left|P_i + B_i\right|_{2^k}$, and it consists of three main stages. The first stage is initializing $P_2 = A$. The second stage is represented by the main iteration step of updating P^{i+1}. Values B^i are to be selected in such a way that after (k–2) steps $P_k \equiv \left|1\right|_{2^k}$ is obtained. Finally, in the third stage, the discrete logarithm modulo 2^k of A is obtained as:

$$dlg(A) = \left| -\sum_{i=2}^{i \le k} dlg(B_i) \right|_{2^{k-2}}$$

The algorithm works as long as $dlg(B_i)$ are all known and $P_k \equiv \left|1\right|_{2^k}$. For example, as suggested in [1]), a lookup table can be used in order to provide feasible values for B_i. The detailed procedure for generating the table is available in [2].

For convenience, a residue that is of the form $\left|2^i + 1\right|_{2^k}$, $3 \le i < k$ is termed a two-ones residue. Whenever the binary digit p_i^i of P_i equals 1, multiplying

P_i with the two-ones residue $\tau_i = (2^i + 1)$ results in a product $P_{i+1} = P_i \times \tau_i$ that is congruent with 1 modulo 2^{i+1}. That is:

$$P_{i+1} = P_i \times \tau_i \equiv |1|_{2^k}$$

When the binary digit p_i^i of P_i equals 0, P^{i+1} can be set to $P_{i+1} = P_i \times 1$ and it is still congruent with $|1|_{2^{i+1}}$.

The two-ones residues set $\{\tau_i : i \in \{3,4,...k-1\}\}$ provides all of the values needed when updating the partial products P_i. We show in Table 2 the 8-bit *dlg* associated with the corresponding two-ones residues. In the table the values of τ_i and $dlg(\tau_i)$ are given in both binary and decimal to illustrate the process.

The key advantage of storing the two-ones residues τ_i in the lookup table is that the multiplication operation by τ_i can be performed as a shift-and-add operation:

$$P_{i+1} = P_i + P_i \times 2^i$$

Table 2: Two Ones Discrete Log Table for k = 8.

i	$\tau_i = 2^i + 1$	$dlg(\tau_i)$	notes
1	00000011B/3D	000001B/1D	$\left\|3^1\right\|_{256} = 3$
2	00000101B/5D	---	
3	00001001B/9D	000010B/2D	$\left\|3^2\right\|_{256} = 9$
4	00010001B/17D	110100B/52D	$\left\|3^{52}\right\|_{256} = 17$
5	00100001B/33D	101000B/40D	$\left\|3^{40}\right\|_{256} = 33$
6	01000001B/65D	010000B/16D	$\left\|3^{16}\right\|_{256} = 65$
7	10000001B/129D	100000B/32D	$\left\|3^{32}\right\|_{256} = 129$

The algorithm for *dlg* is given in the following:

Algorithm 1. (Additive DLG)
Response: dlg(A), expressed as an (s, e) pair: $A \equiv ((-1)^s \times 3^e) \pmod{2^k}$.
Method: *L*1: $P := A$; $|e'|_1 := 0$; $s := 0$;
*L*2: if ($a_2 = 1$) then
*L*3: $s := 1$; $P := \left|-P\right|_{2^k}$
*L*4: fi
*L*5: if $\left|P\right|_{2^3} = 011$ then
*L*6: $\left|e'\right|_{2^1} := 1$;
*L*7: $P := \left|P + P \times 2^1\right|_{2^k}$;
*L*8: fi
*L*9: for i from 3 to ($k - 1$) do
*L*10: if ($p_i = 1$) then
*L*11: $e' := e' + dlg(\tau_i)$;
*L*12: $P := \left|P + \left|P \times 2^i\right|_{2^k}\right|_{2^k}$;
*L*13: fi;
*L*14: end;
*L*15: Result: (s, $\left|-e'\right|_{2^{k-2}}$).

The initialization stage is performed in lines *L*1 − *L*8. If A is not congruent with 1 or 3 modulo 8, the arithmetic sign is considered (i.e. $s := 1$ in *L*3) and the algorithm determines the *dlg* of $\left|-A\right|_{2^k}$ (i.e. $P := \left|-P\right|_{2^k}$ in *L*3). The variable e' represents the exponent of 3 that gives $\left|P^{-1}\right|_{2^i} = \left|3^{e'}\right|_{2^i}$ for P of line *L*4. It is set to 0 in *L*1 and this corresponds to $p_2 = 1$. In case $\left|P\right|_{2^3} = 011$, e' is adjusted in line *L*6 to be 1, along with the corresponding update of P (which is equivalent to $P_2 = 3 \times P$ in *L*7).

The second stage contains the main iteration step and is represented by lines *L*9 − *L*14, where both P and the exponent e' are updated. P is conceptually updated as $P_{i+1} = P_i \times \tau_i$, while the exponent is updated with the corresponding values $dlg(\tau_i)$ looked up from a table.

The final result is computed in line *L*15 as the sign s and the exponent $\left|-e'\right|_{2^{k-2}}$. This is because e' really represents $e' = dlg((-1)^s \times A^{-1})$, and, as a direct consequence of (a.), we have that $dlg((-1)^s \times A) = \left|-e'\right|_{2^{k-2}}$.

The updating of e' and P in lines *L*11 and *L*12 can be performed concurrently. As can be seen by inspection from Algorithm 1, the time complexity is essentially k dependent shift-and-add modulo 2^k operations.

3. Hardware Implementation

The state diagram of the hardware implementation is given in the Figure 1. There are 4 states available, **wait**, **init**, **loop** and **ready**. The **wait** state is also a reset state. It accepts input when the *load* signal is *one* and the *busy* signal is *zero*. The **initial** state corresponds to the part of initial checking in the previous algorithm. It will set the initial value for the loop according the third Least Significant Bit (LSB) and the three LSBs. The **loop** state will repeatedly update the P and e'. This will decide the throughput of the whole system. The loop count goes from 3 to k, totally k-3 maximal. The **ready** state is the state that outputs the result. The circuit automatically steps into the waiting state after **ready** state.

There are three major components of the circuit, a controller, ROM and datapath, as shown in Figure 2.

The controller consists of a counter and state control block (FSM). The FSM will start and stop the counting

procedure. The output of the counter, *count*, will be used for 4 purposes: an address for ROM, an index for the bit checker and a shifter controller and feedback to the FSM for state transition. The ROM is used as lookup table for the *dlg*(τ).The major components in the datapath are two adders, one shifter and a unit called bit-checker which is used to check if a certain bit is true or false. The output of the bit-checker will control the operation of the adders and shifter. If the output is false, no operation will be performed, otherwise, the e' value will be updated by one adder and the P value will be updated by the shifter and adder. The modulo operation given in previous algorithm is handled by limiting the size of P and e'. The size of P is set to k while the size of e' is set to be k-2. Thus, while updating P and e', the result values may be longer than the specified size (or overflow). We can simply ignore the overflowed bits since this computation is performed modulo 2^k.

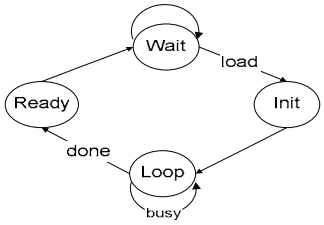

Figure 1. State Transition Diagram

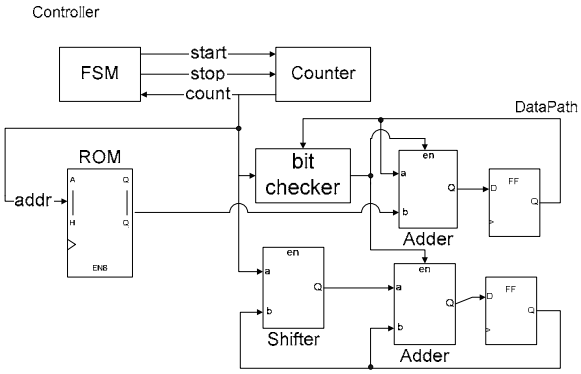

Figure 2 Block Diagram of circuit

4. Experimental Results

We implemented the circuit using the Synopsys tool set based on a standard cell library from the Synopsys tutorial [5]. Table 3 shows the cell delay and transition delay calculated based on the output net total capacitance (cap.) for three types of registers in the library.

Figure 3 shows the schematic view of the circuit after synthesis. Figure 4 (a) shows the physical layout of the design with the standard cell library. Figure 4(b) shows the critical path of the design in the layout for k=8.

Table 3 Technology library parameters

Cell name		denrq1	denrq2	denrq4
Cell delay(ns)	raise	0.379	0.266	0.297
	fall	0.458	0.319	0.357
Total out cap.(pf)		0.051	0.038	0.0386
Size	x	11.89	12.30	13.12
	y	3.69	3.69	3.69

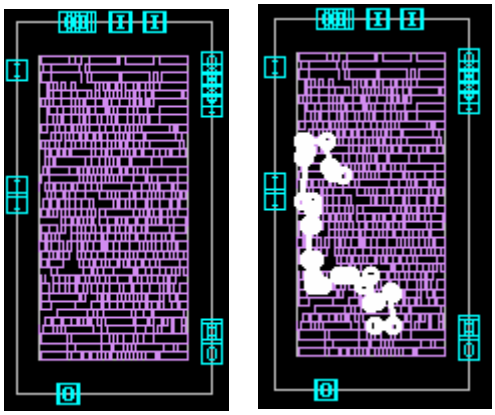

Figure 4. (a) **(b)**

We implemented four designs corresponding to k=8,16,32,64 respectively. Table 4 compares the layout results of the four designs. From the table, it is seen that as k increases, the speed will decrease since the output net total capacitance will increase and the cell delay will also increase. In summary, area, number of cells, and nets all increase since more cells are needed for a larger k value.

From Table 3, it is seen that the cell library is not a fast technology library. Even with this library, we have reached an estimated speed of 500Mhz for k=8. Also, the latency is linear with respect to k. Based on the above data, it is observed that the algorithm provides a very efficient way of calculating the discrete logarithm of a value modulo 2^k.

5. Conclusion and future work

We presented a standard cell hardware implementation of a novel algorithm for computing the discrete logarithm modulo 2^k. The algorithm has a critical path of less than k shift-and-add modulo 2^k operations. We compare the physical standard cell implementations for the algorithm when k=8,16,32,64 respectively. The experimental results confirm that the algorithm is an effective way of calculating the discrete logarithm of a value modulo 2^k.

The exponential residue modulo 2^k operation can also be implemented with a critical path consisting of k lookup table determined shift-and-add modulo 2^k operations [6].

We are currently investigating a hardware extension where components could be shared between the discrete logarithm modulo 2^k and exponential residue modulo 2^k [7]. Based on these two operations, we can implement the multiplication circuit in [3], and provide support for more applications of modular arithmetic in the "hardware friendly" family of binary power moduli 2^k.

Acknowledgement

We would like to thank the Synopsys Corporation for the donation of their tools and use their standard cell library.

6. References

[1] A. Fit-Florea, D. W. Matula, "A Digit-Serial Algorithm for the Discrete Logarithm Modulo 2^k ", *Proc. ASAP, IEEE*, 2004, pp. 236-246.

[2] A. Fit-Florea, D. W. Matula, M. Thornton, "Additive Bit-serial Algorithm for the Discrete Logarithm Modulo 2^k ", *IEE Electronics Letters* Jan. 2005, Vol. 41, No. 2, pp: 57-59.

[3] Benschop N. F., "Multiplier for the multiplication of at least two figures in an original format" *US Patent Nr.* 5,923,888, July 13, 1999.

[4] Szabo, N. S., Tanaka, R.I., "Residue arithmetic and its applications to computer technology", McGraw-Hill Book Company, 1967.

[5] Synopsys Design/physical Compiler Student Guide. 2003.

[6] A. Fit-Florea, D. W. Matula, M. Thornton, "Addition-Based Exponentiation Modulo 2^k ", *IEE Electronics Letters*, Jan. 2005, Vol. 41, No. 2, pp: 56-57.

[7] Lun Li, A. Fit-Florea, M. Thornton, D. W. Matula, "A New Binary Integer Number System with Simplified Hardware Support ", submitted to ASAP 2005.

[8] D. W. Matula, A. Fit-Florea, M. Thornton, "Table Loopup Structures for Multiplicative Inverses Modulo 2^k ", *17th Symp. Comp.Arith., IEEE (to appear).*

Table 4. Comparison of layout result

	k=8	*k*=16	*k*=32	*K*=64
speed	2ns	2.46ns	2.73ns	3.41ns
Core area(um^2)	8260.43	17592	36335.3	76014.1
Die area	12522.6	23625.4	44820.5	88123.8
# of cells	618	1185	2831	5723
# of nets	638	1221	2943	5900
latency(cycles)	8	16	32	64
Ave. net length(um)	30.59	34.92	32.17	37.18
Ave. cell width(um)	2.78	2.99	2.95	2.80
Aspect ratio	2.02	2.01	2.00	2.01
Total cell area(site)*	4191	8636	20389	39118
# of rows	35	51	73	106

site units(width 0.41um, height 3.69um)

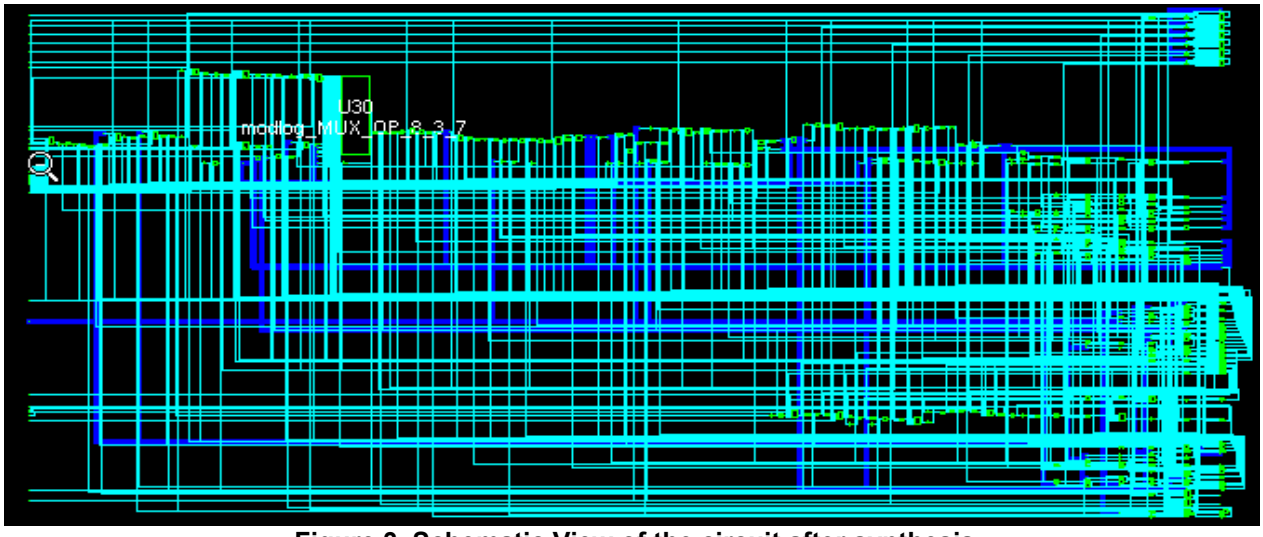

Figure 3. Schematic View of the circuit after synthesis