A Digit Serial Algorithm for the Integer Power Operation

Lun Li
Southern Methodist University
Dept. of CSE
Dallas, TX 75275
1-214-768-1697
lli@engr.smu.edu

Mitch Thornton
Southern Methodist University
Dept. of CSE
Dallas, TX 75275
1-214-768-1371
mitch@engr.smu.edu

David W. Matula
Southern Methodist University
Dept. of CSE
Dallas, TX 75275
1-214-768-3089
matula@engr.smu.edu

ABSTRACT
We introduce a right-to-left digit serial algorithm for the integer power operation x^y where x and y are positive integers. For n-bit words the algorithm utilizes $O(n)$ additions and does not require use of a multiplier. We describe a hardware implementation and evaluate the effectiveness employing a Synopsys tool set with a standard cell implementation. Out digit serial algorithm compares favorably with a popular iterative square and multiply algorithm implemented with the same tool set.

Categories and Subject Descriptors
B.2.4 [**Arithmetic and Logic Structures**]: High Speed Arithmetic – *Algorithms, Cost/performance*

General Terms
Algorithms, Performance, Design.

Keywords
Power Operation, Standard Cell Implementation, Discrete Log, Exponential.

1. INTRODUCTION
Algorithms for computing the operation $z = x^y$ where y is a positive integer have been the subject of considerable research. The binary squaring method determines $x, x^2, x^4, x^8, \ldots$ and processes the bits of y right-to-left to multiply by the appropriate binary powers of x to determine x^y. This algorithm has been described in many popular texts [9][10][11]. Knuth [11] traces this "fast" algorithm back to al-Kashi in the 15th century.

We are interested in the particular case where x, y and the result z are all non-negative *k*-bit integers. For typical word sizes such as k = 8, 16, 32, 64, 128,…, this integer valued powering operation is proposed to supplement the integer addition and multiplication operations. The squaring algorithm may be implemented in hardware with microcode and a fast multiplier much like the floating-point transcendental operations in the Pentium and Athlon processors.

For implementation in hardware there is a need for a simpler algorithm that avoids the use of a large multiplier. There is a further need for a right-to-left digit serial algorithm that requires less time for lower precision operations when a family of precision levels is implemented in hardware.

In this paper, we introduce a novel digit serial algorithm for evaluation of the integer power operation x^y that does not require a multiplier. The algorithm employs conversion of x to a discrete log format [2], bit serial multiplication with the discrete log value providing the "recoded multiplier bits" and the exponent y being the multiplicand, and bit serial deconversion of x^y [6] to provide the result z.

The paper is organized as follows. We present some number theoretic background material on the integer power operation and review the foundations for the algorithms in Section 2. In Section 3, we present our digit serial integer power algorithm. Section 4 contains a description of the hardware implementation of our proposed algorithm and Section 5 provides area and delay estimates from the standard cell synthesis procedure. Conclusions are presented in Section 6.

2. A DIGITAL SERIAL INTEGER POWER ALGORITHM
The inheritance principle [8] for integer operations on binary operands informally states that the *k* low-order bits of the result depend only on the *k* low-order bits of the operands for all $k \geq 1$. This principle provides the basis for right-to-left digit serial integer operations. Specifically, if we assume we have determined the low order (*k*-1)-bits of the result from the (*k*-1) low order operand input bits, incorporating the *k*-th bits of the operands, the *k*-th result bit can then be determined with the (*k*-1) lower order result bits "inherited" from the preceding serial computation.

GLSVLSI'06, April 30–May 2, 2006, Philadelphia, Pennsylvania, USA.

For a binary integer operand $x = b_{n-1}b_{n-2}..b_2b_1b_0$, the modular notation $|x|_{2^k} = b_{k-1}b_{k-2}..b_2b_1b_0$ is employed [4] herein to denote the value of the standard low order *k*- bit string for all $1 \le k \le n$, and the inheritance principle can then be stated more formally.

Inheritance Principle: The integer operation $z = x * y$ has the inheritance property if for all non negative integers x, y,

$$|z|_{2^k} = \left| |x|_{2^k} * |y|_{2^k} \right|_{2^k} \text{ for all } k \ge 1. \qquad (1)$$

Integer addition and multiplication clearly have the inheritance property, as evidenced in the traditional right-to-left "carry ripple" algorithms. To establish a foundation for our digit serial integer power algorithm, we note here without proof that the operation x^y satisfies the inheritance property given appropriate exception handling for zero valued operands as indicated in the following:

Lemma: Let x, y be integers with $x \ge 0, y \ge 1$. Then for all $k \ge 1$,

$$\left|x^y\right|_{2^k} = \begin{cases} \left| (|x|_{2^k})^{|y|_{2^k}} \right|_{2^k} & \text{for } |y|_{2^k} \ne 0, \\ \left| (|x|_{2^k})^{2^k} \right|_{2^k} & \text{for } |y|_{2^k} = 0. \end{cases} \qquad (2)$$

Note that x has a unique factorization into odd and even terms $x = 2^p n$ with *n* odd. It is straightforward to show [1,3] that the 2^{k-2} members of the sequence $3^0, 3^1, 3^2, ..., 3^{2^{k-2}-1}$ reduce modulo 2^k to a sequence of distinct odd numbers covering half the odd numbers in $[1, 2^k - 1]$. The complementary values $\left|(-1)^s 3^e\right|_{2^k}$ for $s \in \{0,1\}$, $0 \le e \le 2^{k-2} - 1$ cover the other half of the odd numbers. For example for $k = 5$, the reduced sequence is 1,3,9,27,17,19,25,11, and the complementary sequence is 31,29,23,5,15,13,7,21. Thus every odd k-bit integer is uniquely represented by the exponent pair (s, e) termed the Discrete Logarithmic System (DLS) representation.

Efficient solutions of the integer-to-discrete-log *conversion* and *deconversion* operations are presented at the algorithmic level in [1,2,6]. These methods employ k sequential steps of a table lookup exponent recoding operation interleaved with a shift-and-add modulo 2^k operation.

Binary-to-discrete log conversion refers to determining the pair (s, e) given the $k-$bit odd integer n, and deconversion refers to determining n given the pair (s, e), where n, s, e satisfy $n = \left|(-1)^s 3^e\right|_{2^k}$. For conversion, s is determined by conditional complementation to obtain a normalized n congruent to 1 or 3 (mod 8). This reduces the conversion operation to the determination of the discrete log $e = \mathrm{dlg}(n)$ for n congruent to 1 or 3 (mod 8), with $0 \le e \le 2^{k-2} - 1$. The deconversion problem reduces to evaluating the exponential residue operation determining n where $n = \left| 3^e \right|_{2^k}$. For completeness, we review algorithms from [2,6] demonstrating that both the exponential residue operation (determining n given e) and the discrete log operation (determining e given n) can be performed by a series of less than k table-assisted shift-and-add operations employing *exponent recoding*.

2.1 Existing "Fast" Binary Squaring Algorithm

The existing fast algorithm is based on the fact that $y = \sum_{i=0}^{k-1} y_i 2^i$. So that we can get the formula

$$z = \left|x^y\right|_{2^k} = \left| x^{\sum_{i=0}^{k-1} y_i 2^i} \right|_{2^k} = \left| x^{y_0 2^0} \times x^{y_1 2^1} \times ... \times x^{y_{k-1} 2^{k-1}} \right|_{2^k}$$

e.g. $3^{10} = 3^{0\times2^0 + 1\times2^1 + 0\times2^2 + 1\times2^3} = 3^2 \times 3^8$, the algorithm is given in the following.

Algorithm 1 Binary Squaring Powering (x^y)
Stimulus: $k, x = x_{k-1}x_{k-2}..x_2x_1 1$, $y = y_{k-1}y_{k-2}..y_2y_1y_0$
Response: $z = | x^y |_{2^k}$.
*L*1: $z := 1$; $q := x$;
*L*2: for $i := 0$ to $k-1$ do
*L*3: if bit $(i, y) = 1$ then
*L*4 $z := | z \times q |_{2^k}$
*L*5 end
*L*6: $q := | q \times q |_{2^k}$
*L*7:end

2.2 Additive Based Exponentiation Modulo 2^k

$| 3^e |_{2^k}$ can be computed using square-and-multiply method. This entails computing $| 3^2 |_{2^k}$, $| 3^4 |_{2^k}$, …, $| 3^{2^k} |_{2^k}$, by successive squaring. We note that similar methods lead to the correct result when the exponent e is recoded as a sum of elements $e = | \sum \alpha_i |_{2^{k-2}}$ [6]. In this case $| 3^e |_{2^k}$ can be computed as $| 3^e |_{2^k} = | 3^{\sum \alpha_i} |_{2^k}$. Of course, this presents an advantage if the α_i and/or corresponding powers $\{3^{\alpha_i}\}$ are precomputed and available by table lookup. In [6] it is shown that any exponent e can be expressed as a sum of $\mathrm{dlg}(2^i + 1)$'s termed the two-ones discrete logs. Since $3^{\mathrm{dlg}(2^i+1)} = 2^i + 1$, it follows that the corresponding multiplications can be performed as a series of

shift-and-add operations. This works if the two-ones dlg's are pre-computed and stored in a table (as shown in Table 1).

Table 1: Two Ones Discrete Log Table for k = 8

i	2^i+1		$\mathrm{dlg}(2^i+1)$		**check**
	Bin.	**Dec.**	**Bin.**	**Dec.**	
0	0000 0001	1	0000 0000	0	$\left\| 3^0 \right\|_{2^{16}} = 1$
1	0000 0011	3	0000 0001	1	$\left\| 3^1 \right\|_{2^{16}} = 3$
2	0000 0101	5	N/A	N/A	N/A
3	0000 1001	9	0000 0010	2	$\left\| 3^2 \right\|_{2^{16}} = 9$
4	0001 0001	17	1011 0100	7604	$\left\| 3^{7604} \right\|_{2^{16}} = 17$
5	0010 0001	33	0010 1000	15912	$\left\| 3^{15912} \right\|_{2^{16}} = 33$
6	0100 0001	65	0101 0000	10064	$\left\| 3^{10064} \right\|_{2^{16}} = 65$
7	1000 0001	129	1010 0000	15008	$\left\| 3^{15008} \right\|_{2^{16}} = 129$

Algorithm 2 determines the unique set of $\mathrm{dlg}(2^i+1)$'s whose sum modulo 2^{k-2} equals e. It thus allows efficient conversion from DLS-to-binary. In the algorithmic description that follows, index notation is used for the corresponding bit of the standard binary representation.

Algorithm 2 Deconversion Algorithm (EXP)
Stimulus: $k, e = e_{k-3}e_{k-4}..e_2e_1e_0$
Response: $\left|3^e\right|_{2^k}$
Method: *L*1: if bit($0, e$)=1 then $y := 11$; $q := e-1$
*L*2: else $z := 1$; $q := e$
*L*3: end
*L*4: for $i := 1$ to $k-3$ do
*L*5: if bit(i, q)=1 then
*L*6: $z := \left|z + z << (i+2)\right|_{2^k}$
$q := q - \mathrm{dlg}(2^{i+2}+1)$
*L*7: end
*L*8: end
*L*9: return *z*

Please note that lines *L*1 − *L*3 correspond to initialization. The product *z* is set to either 1 or binary 11 (corresponding to $e_0 = 1$ or $e_0 = 0$). The working variable exponent q is always set in such a way that *z* corresponds – for each iteration – to 3 raised to the exponent ($e-q$) and the least significant *i* bits of q are all 0s. The algorithmic step of lines *L*4 − *L*8 represents updating q by subtracting $\mathrm{dlg}(2^{i+2}+1)$, which simply represents the exponent of 3 that reduces to $2^{i+2}+1$. This is followed by updating the product z to reflect the changes in exponent, $z := \left|z \times (2^{i+2}+1)\right|_{2^k}$. Eventually, after ($k-2$) steps, q becomes 0 and the "product" z corresponds to $\left|3^{e-0}\right|_{2^k} = \left|3^e\right|_{2^k}$. The values $\mathrm{dlg}(2^{i+2}+1)$ can be stored in a lookup table and this method is practical for large $k = 64, 128, ...,$ since the table has only *k* entries.

2.3 Additive Based Discrete Logarithm Modulo 2^k

Computing the discrete logarithm for certain $k-$bit odd integers x can be accomplished using a method [2] that is essentially the dual of the exponentiation method of Section 2.2. The key idea is to express x, if possible, as a product of two-ones residues: $x = \left| \prod (2^i+1) \right|_{2^k}$ for selected i's. Once this is done, the discrete logarithm can be computed as the corresponding sum: $\mathrm{dlg}(x) = \mathrm{dlg}(\prod (2^i+1)) = \left| \sum \mathrm{dlg}(2^i+1) \right|_{2^{k-2}}$ [2]. The solution involves identifying the cases when x can be expressed as such a product and finding the corresponding unique set of two-ones residues. It is shown in [2] that x can be expressed as a two-ones residue product as long as x is congruent with 1 or 3 modulo 8. Note that for the remaining odd residues, corresponding to x congruent with 5 or 7 modulo 8, their additive inverses $|-x|_{2^k}$ are congruent with 1 or 3 modulo 8. The method in [2] identifies the set of two-ones residues and thus it is the core of a digit serial conversion method from binary to DLS.

Algorithm 3 Binary to DLS Conversion Algorithm (DLG)
Stimulus: $k, x = x_{k-1}x_{k-2}..x_2x_1x_0$ with $x_0 = 1$
Response: discrete log of x, expressed as an (*s*, *e*) pair: $x = \left| (-1)^s 3^e \right|_{2^k}$.
Method: *L*1: if $\left|x\right|_8 \in \{1, 3\}$ then $s := 0$;
*L*2: else $s := 1$; $x := 2^k - x$
*L*3: end
*L*4: $p := 1$; $e := 0$
*L*5: for $i := 1, 3$ to $k-1$ do
*L*6: if bit(i, x)=bit(i, p) then
*L*7: $p := \left|p + p << i\right|_{2^k}$;
$e := e + \mathrm{dlg}(2^i+1)$
*L*8: end
*L*9:end
*L*10: Result: (*s*, *e*).

The initialization stage is performed in lines *L*1 − *L*4. If x is not congruent with 1 or 3 modulo 8, the arithmetic sign is considered (i.e. *s* := 1 in *L*2) and the algorithm determines the dlg of the

complement $\left|2^k - x\right|_{2^k}$ (i.e. $x := 2^k - x$ in *L2*).

The second stage contains the main iteration step and is represented by lines *L*5 − *L*9, where both p and the exponent e are updated. p is conceptually updated as $p = p \times (2^i + 1)$, while the exponent e is updated by subtracting the corresponding values $\mathrm{dlg}(2^i + 1)$, looked up from a table.

The final result is computed in line *L*10 as the sign s and the exponent e. The updating of e and p in lines *L*7 can be performed concurrently. As can be seen by inspection of Algorithm 2, the time complexity is essentially k dependent shift-and-add modulo 2^k operations.

3. PROPOSED FEEDBACK SHIFT ADD (FSA) ALGORITHM

Based on previous work, we know that any number can be converted to a triple (s, p, e) where $x = 2^p n$ with n odd [2,6]. So that

$$z = \left|x^y\right|_{2^k} = \left|((-1)^s 2^p 3^e)^y\right|_{2^k} = \left|(-1)^{sy} 2^{py} 3^{ey}\right|_{2^k}$$

In the above formula, $(-1)^{sy}$ determines the sign. 2^{py} determines the number (py) of least significant zeros. Without loss of generality, we focus on odd numbers in the following discussion. For odd numbers, we need to calculate $e \times y$ for term 3^{ey}. Then we can convert the $\left|(-1)^{sy} 3^{ey}\right|_{2^k}$ back to binary to get z. The block diagram for such an approach is shown in Figure 1.

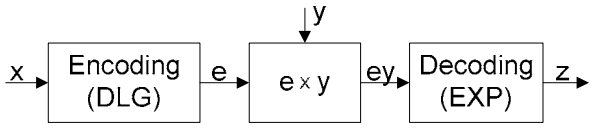

Figure 1. A Serial Version of Proposed algorithm

In Figure 1, we calculate the y^{th} power of operand x in a serial fashion. That is we start multiplication and decoding after we obtain the entire value of e. A better technique is a pipelined arrangement of the sub-operations in which multiplication and decoding starts when the first bit of e is available. For every available bit of e, a bit of the intermediate product is generated followed by a bit of z being produced. This method is referred to as the pipelined algorithm and is described in the following algorithm.

Algorithm 4: Additive Digit Serial Powering (x^y)
Stimulus: $k, x = x_{k-1}x_{k-2}..x_2x_1 1$, $y = y_{k-1}y_{k-2}..y_2y_1y_0$
Response: $z = \left| x^y \right|_{2^k}$.
Method
*L*1: if $\left|x\right|_8 \in \{1,3\}$ then $s := 0$;
*L*2: else $s := 1$; $x := 2^k - x$
*L*3: end
*L*4: $p := 1$; $e := 0$; $z := 1$; $q := 0$; $t = e$;
*L*5. if bit($1, x$)=bit($1, p$) then
*L*6: $p := \left|p + p << 1\right|_{2^k}$; $e := e + \mathrm{dlg}(3)$
*L*7: if bit($0, y$)=1 then
*L*8: $z := \left|z + z << 1\right|_{2^k}$;
*L*9: end
*L*10: end
*L*11: for $i := 3$ to $k - 1$ do
*L*12: if bit(i, x)=bit(i, p) then //update for DLG
*L*13: $p := \left|p + p << i\right|_{2^k}$; $e := e + \mathrm{dlg}(2^i + 1)$
*L*14: end
L15: $t = t << 1$;
L16: if(bit($i - 2, e$)=1)
*L*17: $m = m + t$ //accumulator.
L18 end
*L*19: if bit($i - 2, m$)=1 then
*L*20: $q = q + 2 << (i - 2)$
*L*21: if bit(i, q)=1 then //update for EXP
*L*22: $y := \left|y + y << i\right|_{2^k}$;
*L*23: $q := q - \mathrm{dlg}(2^i + 1)$
*L*24: end
L25:end

The initialization stage is performed in lines *L*1 − *L*4. All the required initialization for both stages of the algorithm is performed here. The second stage (*L*5 − *L*10) actually performs the computation for $i = 1$. The third stage contains the main iteration step and is represented by lines *L*11 − *L*24. The third stage can be separated into 3 sub-stages. Both p and the exponent e are updated (i.e. *L*12 − *L*14) which generates one bit of d according to the DLG algorithm. The second sub-stage (i.e. *L*15 − *L*19) corresponds to the accumulator used to implement $e \times y$. The third stage updates z according to EXP algorithm (i.e. *L*19 − *L*24). The final result is obtained at line *L*25. As can be seen by inspection of the algorithm, the time complexity is essentially k dependent shift-and-add modulo 2^k operations.

4. HARDWARE IMPLEMENTATION

The state diagram of a controller for a hardware implementation is given in Figure 2. There are 6 states defined, **Load**, **Init**, **Loop_DLG, Loop_ACC, Loop_EXP** and **Ready**. The **Load** state is also a reset state. It accepts input when the *load* signal is asserted and also performs all the initialization operations in lines *L*1 − *L*4. The **Init** state corresponds to the second stage (*L*5 − *L*10) in the previous algorithm. The **Loop_DLG, Loop_ACC, Loop_EXP** states correspond to the 3 sub-stages in the previous algorithm. The loop count goes from 3 to k, with a maximum of k-3 iterations. The **Ready** state is the state that outputs the result. The circuit automatically transitions into the **Load** state after **Ready** state.

There are three major components in the circuit, a controller, a ROM lookup table, and a computation datapath. The controller consists of a counter and state control block Finite State Machine

(FSM). The FSM will start and stop the counting procedure. The output of the counter, *count*, is used for purposes such as address generation for the ROM, index production for the bit checker and loop controller and feedback to the FSM for state transition. The ROM is used as lookup table for the *dlg*(τ) function. The major components in the datapath are adders, shifters and units called bit-checkers that are 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 shifters. If the output is false, no operation will be performed, otherwise, registers holding p, e, z, q will be updated by the shifter and adder. The modulo operation given in previous algorithm is handled by limiting the size of p, e, z, q . The size of p, e, z, q are set to k. Thus, while updating p, e, z, q , the result values may be longer than the specified size (or overflow). We can ignore the overflowed bits since this computation is performed modulo 2^k.

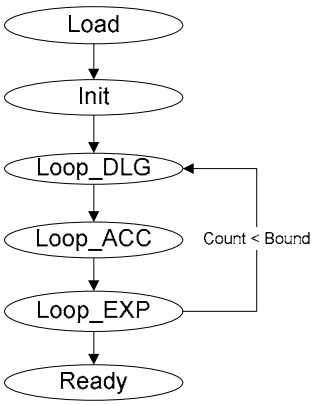

Figure 2. Controller State Transition Diagram

5. EXPERIMENTAL RESULTS

In order to evaluate the effectiveness of our method as compared to the well-known "fast" squaring method, we described each method in Verilog RTL and synthesized the circuits using the Synopsys tool set based on a standard cell library from Synopsys [5] and a standard cell library from Oklahoma State University [7]. Since the results from the two standard cell libraries were similar, we only list the result based on the standard cell library from Synopsys.

Table 2. Comparison of layout result

	speed(ns)		core area	
	ours	fast	ours	fast
8	2.05	2.4	23386.4	8207.48
16	2.41	3.45	40306.7	26076.3
32	2.75	4.55	109135	79409.1
64	3.52	5.55	184725	302942
128	3.8	6.8	371366	1.26E+06

We implemented five designs corresponding to wordsizes of k=8, 16, 32, 64, 128 respectively. We also implemented the existing fast algorithm described in Section 2.3. Table 2 compares the results of our algorithm with the existing fast algorithm for different k values. We also plot the trend of the two algorithms in Figure 3 (speed) and Figure 4 (area). It is seen that for all k values, our algorithm is faster than the existing fast algorithm when each algorithm is synthesized with the standard cell library. Regarding area, our method requires more space for small word sizes but increases slowly compared with the existing fast algorithm. Thus, when $k \geq 64$, our algorithm requires less area. It should be noted that the area values reported here are only the net area required by the total cell area since we did not route the resulting circuits, thus additional area required by routing is not included.

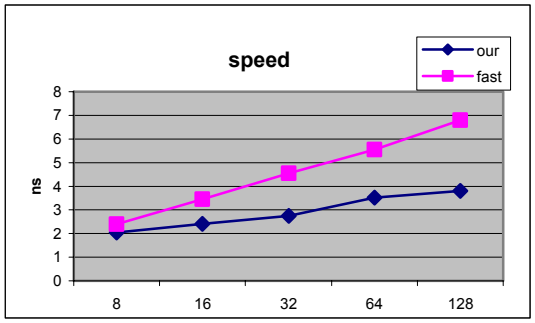

Figure 3. Speed trend of two algorithms

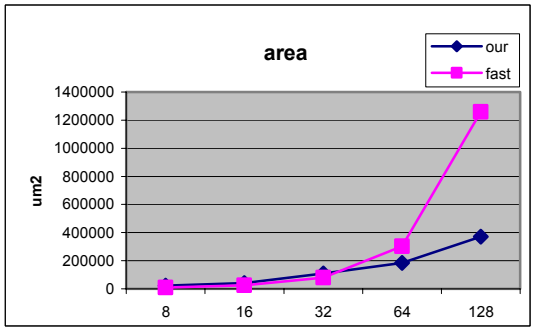

Figure 4. Area trend of two algorithms

6. CONCLUSION AND FUTURE WORK

We presented a novel algorithm for computing the powering operation modulo 2^k. The algorithm has a critical path of less than k shift-and-add modulo 2^k operations. To evaluate the effectiveness of the method, we compare the standard cell synthesis results for the algorithm for wordsizes of k=8, 16, 32, 64, 128 respectively. The experimental results confirm that the algorithm is an effective way of calculating the discrete logarithm of a value modulo 2^k.

The bottleneck in the new digit serial algorithm is the use of repetitive large shifts to implement the compound product $\left|\prod (2^i + 1)\right|_{2^k}$ for selected $i's$ using at most $k-1$ additions. Further work in our laboratory is addressed to reducing the shift penalty for realizing this product to further improve our synthesis results.

One observation we have made can significantly reduce the long shift when the iteration reaches the middle point of the result at

the $k/2$ bit. At this middle-point, the lower half $k/2$ bits of the compound product will not change while the upper half $k/2$ bits are still being accumulated. However, the length of shift after reaching the middle-point is larger than $k/2$, which means that the upper $k/2$ bits will be shifted to a position that is larger than the final product size k. Thus, it will not affect the compound product any more. Based upon this observation, we can conclude that when the iteration reaches the middle-point, we only need to record the lower half $k/2$ bits and shift the recorded data one bit left each time for the next $k/2$ computations.

7. ACKNOWLEDGMENTS

We would like to thank the Synopsys Corporation for the donation of their tools and the use of their standard cell library. We would also like to thank Dr. James Stine from Oklahoma State University who provided us with a standard cell library developed in his laboratory.

8. 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.A. 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.A. Thornton, "Addition-Based Exponentiation Modulo 2^k ", *IEE Electronics Letters*, Jan. 2005, Vol. 41, No. 2, pp: 56-57.

[7] J.E. Stine, J. Grad, I. Castellanos, J. Blank, V. Dave, M. Prakash, N. Illiev, N. Jachimiec: A Framework for High-Level Synthesis of System-on-Chip Designs, Proceedings. 2005 IEEE International Conference on Microelectronic Systems Education, 2005. 12-13 June 2005, pages 67-68.

[8] D.W. Matula, A. Fit-Florea, M.A. Thornton, "Table Loopup Structures for Multiplicative Inverses Modulo 2^k ", *17th Symp. Comp.Arith.,* June 27-29, 2005, pp. 130-135,.

[9] T. Cormen, C. Leiserson, R. Rivest, C. Stein, "Introduction to Algorithms", 2nd edition, The MIT Press, 2001, pp. 879-880.

[10] B. Parhami, "Computer Arithmetic Algorithms and Hardware Designs", Oxford University Press, 2000, pp. 383-384.

[11] D. Knuth, "The Art of Computer Programming: Seminumerical Algorithms" Addison Wesley, Vol. 2, 2nd Edition, 1981, pp: 441-466.