AN ABSTRACT OF THE THESIS OF

Budiyoso Kurniawan for the degree of Master of Science in
Electrical & Computer Engineering presented on March 14, 2002.
Title: ASIC Design and Implementation of A Parallel Exponentiation Algorithm using Optimized Scalable Montgomery Multipliers

Abstract approved: Redacted for Privacy
Alexandre Ferreira Tenca

Modular exponentiation and modular multiplication are the most used operations in current cryptographic systems. Some well-known cryptographic algorithms, such as RSA, Diffie-Hellman key exchange, and DSA, require modular exponentiation operations. This is performed with a series of modular multiplications to the extent of its exponent in a certain fashion depending on the exponentiation algorithm used.

Cryptographic functions are very likely to be applied in current applications that perform information exchange to secure, verify, or authenticate data. Most notable is the use of such applications in Internet based information exchange. Smart cards, hand-helds, cell phones and many other small devices also need to perform information exchange and are likely to apply cryptographic functions.

A hardware solution to perform a cryptographic function is generally faster and more secure than a software solution. Thus, a fast and area efficient modular exponentiation hardware solution would provide a better infrastructure for current cryptographic techniques.

In certain cryptographic algorithms, very large precisions are used. Further, the precision may vary. Most of the hardware designs for modular multiplication and

modular exponentiation are fixed-precision solutions. A scalable Montgomery Multiplier (MM) to perform modular multiplication has been proposed and can operate on input values of any bit-size, but the maximum bit-size should be known and is the limiting factor. The multiplier can calculate any operand size less than the maximal precision. However, this design's parameters should be optimized depending on the operand precision for which the design is used.

A software application was developed in C to find the optimized design for the scalable MM module. It performs area-time trade-off for the most commonly used precisions in order to obtain a fast and area efficient solution for the common case.

A modular exponentiation system is developed using this scalable multiplier design. Since the multiplier can operate on any operand size up to a certain maximum value, the exponentiation system that utilizes the multiplier will inherit the same capability.

This thesis work presents the design and implementation of an exponentiation algorithm in hardware utilizing the optimized scalable Montgomery Multiplier. The design uses a parallel exponentiation algorithm to reduce the total computation time.

The modular exponentiation system experimental results are analyzed and compared with software and other hardware implementations.

ASIC Design and Implementation of A Parallel Exponentiation Algorithm using Optimized Scalable Montgomery Multipliers

by

Budiyoso Kurniawan

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for
the degree of

Master of Science

Presented March 14, 2002

Commencement June 2002

Master of Science thesis of Budiyoso Kurniawan presented on March 14, 2002

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical & Computer Engineering

Redacted for Privacy

Chair of the Department of Electrical & Computer Engineering

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Budiyoso Kurniawan, Author

ACKNOWLEDGEMENTS

I would like to thank Dr. Alexandre Tenca and Dr. Cetin Koc for giving me the opportunity to work on this interesting project. Dr. Tenca's reviews, comments, and discussions on this thesis work were really valuable.

I would like to thank Dr. David Kim for giving me information, comments, and discussion for the optimization portion of the thesis.

I would like to thank all my friends, especially Georgi Todorov, Donald Heer, Jirachai Buddhakulsomsiri, and Shakib Shakeri, for their ideas, comments, and discussions related to the thesis as well as their support.

Finally, I would like to thank my family, who makes it possible for me to explore this degree.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

ASIC DESIGN AND IMPLEMENTATION OF A PARALLEL EXPONENTIATION ALGORITHM USING OPTIMIZED SCALABLE MONTGOMERY MULTIPLIERS

1. INTRODUCTION

Modular exponentiation and modular multiplication are the most used operations in current cryptographic systems. Some well-known cryptographic algorithms are Rivest-Shamir-Adleman (RSA) [1, 3], Diffie-Hellman key exchange [2], Digital Signature (DSA) [4], and elliptic curve cryptography (ECC) [5]. All these algorithms, except for ECC, need to perform modular exponentiation. Modular exponentiation is performed as a series of modular multiplications to the extent of its exponent in a certain fashion depending on the algorithm used.

In current applications, almost all information exchange apply some kind of cryptographic function to secure, verify, or authenticate data. Applications such as Secure Socket Layer (SSL), Internet Key Exchange (IKE), and many others are used to secure information exchange over the Internet. Many small devices where area and power are limited, such as smart cards, hand-helds, and cell phones, are also likely to apply cryptographic function since they perform information exchange.

A hardware solution is generally more secure and faster than software solution in performing a cryptographic function. A fast and area efficient exponentiation hardware solution would provide a better infrastructure for current cryptographic techniques.

For modular multiplication, the Montgomery Multiplication (MM) algorithm [6] provides advantages in a hardware implementation. The main advantage of this algorithm is that instead of performing division by the modulus, it performs simple bit

shifts, which are easier to implement and much less costly in terms of execution time. Another issue is the operand precision. In most cryptographic techniques, very large precision numbers are used. Several scalable Montgomery Multiplier (MM) module designs have been proposed and implemented in [8, 9, 10, 11]. These designs allow the computation on operands of any size over a limited hardware implementation.

Several exponentiation algorithms [1] can be implemented over the scalable MM module [10]. The kernel inside this scalable MM module consists of a number of processing elements configured to receive input at a certain word size. As in any hardware solution, tradeoff issues between area and time are always present. The total computation time of the kernel varies for different precisions. Since the modular multiplication time varies depending on precision, modular exponentiation time, which utilizes modular multiplication, will reflect the variation on a much larger scale.

The objective of this thesis work was to design and implement an exponentiation algorithm in hardware utilizing the optimized scalable Montgomery Multiplier [10]. The design uses a parallel exponentiation algorithm to reduce the total computation time. A software application was developed in C to find the optimized design for the scalable MM module. It performs area-time trade-off for the most commonly used precisions in order to obtain a fast and area efficient solution for the common case.

The next two sections in this chapter introduce the MM algorithm and a general exponentiation algorithm suitable for hardware implementation. Chapter 2 introduces a parallel exponentiation algorithm and a scalable MM hardware implementation. Chapter 3 shows the parallel exponentiation algorithm using MM and the system level architecture for modular exponentiation. Chapter 4 discusses the issues related to the scalability of the MM module and methods to find an optimized design for its implementation. It also describes a software application implementation and its results. Chapter 5 presents the experimental results of the system proposed. Chapter 6 concludes this work and presents some possible future improvements.

1.1. Montgomery Multiplication (MM) Algorithm

The following notation will be used throughout this text. New fields will be added as appropriate.

- M – modulus for modular multiplication;
- X – multiplier operand for modular multiplication;
- X_i – a single bit of X at position i;
- Y – multiplicand operand for modular multiplication;
- N – number of bits in the operands, operand's precision;
- r – a constant, $r = 2^N$;
- S – partial product in the multiplication process, final result of modular multiplication;
- S_i – a single bit of S at position i.

The application of the Montgomery Multiplication (MM) algorithm, given 2 integers X and Y, with a required parameter for N bits of precision is defined as follows:

$$\mathrm{MM(X, Y)} = XYr^{-1} \bmod M,$$

where $r = 2^N$ and M is an integer in the range $2^{N-1} \le M \le 2^N - 1$ such that gcd(r, M) = 1. The Montgomery multiplication algorithm is used to transform an integer in the range [0, M-1] to another integer in the same range called the image or the M-residue of the integer.

To obtain the modular multiplication $C = XY \bmod M$, the following series of MM operations are performed.

- The image of X and Y are calculated as:

$$\overline{X} = MM(X, r^2) = Xr \bmod M$$
$$\overline{Y} = MM(Y, r^2) = Yr \bmod M$$

- The image of C is then calculated as:

$$\overline{C} = MM(\overline{X}, \overline{Y}) = MM(Xr, Yr) = XYr \bmod M$$

- To return from the image to the original integer value the following operation is performed.

$$C = MM(\overline{C},1) = C \bmod M = XY \bmod M$$

This can be done provided that r^2 (mod M) is pre-calculated and saved. The advantage is the low complexity of the MM algorithm.

As can be seen from Figure 1.1, for a single modular multiplication of two integers, four (minimal of two) MM operations are required. However, if multiple multiplications are required, the numbers can be kept in the Montgomery domain and further multiplications can be performed without transformation. The next modular multiplication only requires one MM operation. After all multiplications are performed, a final MM operation is executed to convert back from M-residue to integer field.

In modular exponentiation implementation, this MM is very beneficial since multiple multiplications can be performed prior to transferring the result back to the integer field.

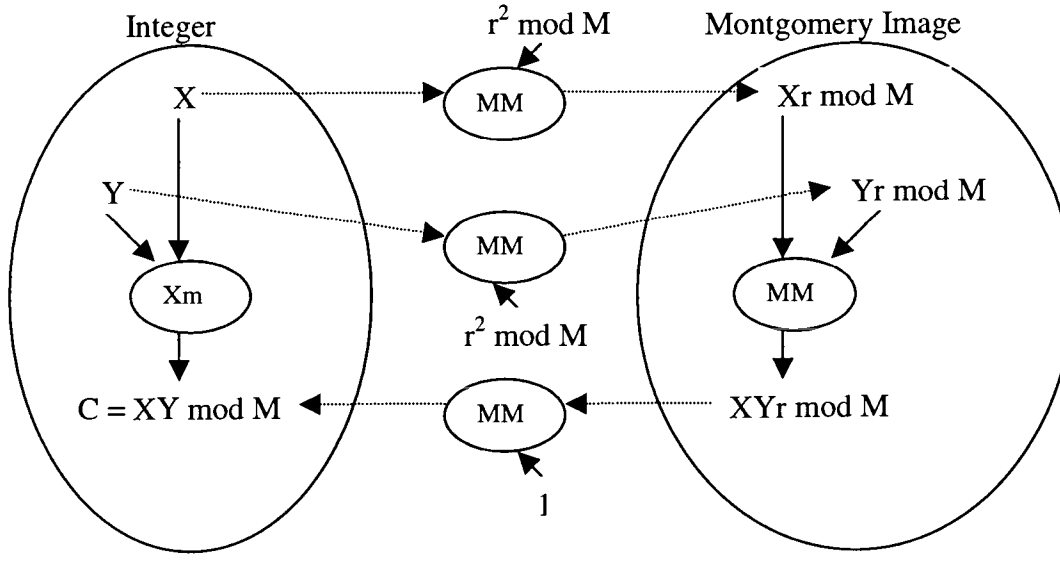


Figure 1.1. Modular Multiplication using MM

The Montgomery multiplication (MM) algorithm has been expanded from its original form [6], which is a fixed-precision implementation in radix 2, to a scalable, word-based implementation on multiple radices [8, 9, 10, 11].

Figure 1.2 shows the radix-2 MM algorithm. Radix-2 implies that X is scanned one bit in each iteration loop. As can be seen from the algorithm (step 2c), the division step for modulus is substituted with a division by 2, which is implemented by a simple shift operation. The algorithm represents a fixed-precision implementation since the S, Y, and M operands are full size. For implementation in hardware, the maximum precision of S, Y, and M operands should dictate the allocation of hardware resources. Even if the actual precision used for computation is smaller; the algorithm will still perform as if S, Y, and M are at the maximum precision. Thus, the multiplier will consume a lot of area, require a high number of pins, have an increased load for gates, and longer wires.

1. $S = 0$
2. For $i = 0$ to $N-1$
   - 2.a. $S = S + X_i * Y$
   - 2.b. $S = S + S_0 * M$
   - 2.c. $S = S/2$
3. If $S \geq M$ then $S = S - M$

Figure 1.2. Radix-2 MM algorithm

In order to make the MM scalable and perform in higher radices, the algorithm is modified to the Multiple-Word High Radix MM algorithm [8, 10, 11]. This algorithm allows the multiplier to run at word size instead of full precision. This allows for the system design as described in Section 2.2.

1.2. Binary Method Algorithm for Exponentiation

The following list extends the notation used in this thesis. New fields will be added as appropriate.

- Z – exponent operand of modular exponentiation;
- Z_i - a single bit of Z at position i;
- C – temporary result of each loop, final result of modular exponentiation;
- K – number of bits in the exponent operand (exponent operand's precision).

A classic algorithm for exponentiation is the binary method [1]. The binary method scans the bits of the exponent either from left (most significant bit) to right (least significant bit) or vice versa. Figure 1.3 describes the binary method algorithm for $C = X^Z \ mod \ M$ calculation, scanning the exponent Z from left to right. At each step, a squaring is performed. A multiplication follows depending on the scanned bit value.

The number of squaring operations (step 2a) is $K-1$, where K is the number of bits (precision) in the exponent. The number of multiplication operations (step 2b) is $H(e) - 1$ where H(e) is the Hamming weight (the number of 1s in the binary expansion of the exponent). Thus, the total number of multiplications is averaged at 3(N-1)/2.

Input: K, X, Z, M
Output: $C = X^Z \ mod \ M$
1. If $Z_{K-1} = 1$ then C = X else C = 1
2. For i = K-2 downto 0
   2.a. $C = C*C \ mod \ M$ (squaring)
   2.b. If $Z_i = 1$ then $C = C*X \ mod \ M$ (multiplication)
3. Return C

Figure 1.3. Binary Method algorithm

An example of the algorithm application is shown in Table 1.1 for Z = 10110110 and K = 8. Since the most significant bit of Z (Z_{N-1}) equals 1, then C = X in step 1.

i	Z_i	Step 2a	Step 2b
7	1	-	-
6	0	$(X)^2 = X^2$	X^2
5	1	$(X^2)^2 = X^4$	$X^4.X = X^5$
4	1	$(X^5)^2 = X^{10}$	$X^{10}.X = X^{11}$
3	0	$(X^{11})^2 = X^{22}$	X^{22}
2	1	$(X^{22})^2 = X^{44}$	$X^{44}.X = X^{45}$
1	1	$(X4^5)^2 = X^{90}$	$X^{90}.X = X^{91}$
0	0	$(X^{91})^2 = X^{182}$	X^{182}

Table 1.1. Binary method implementation example

This method can be called the serial implementation of the binary method. The main issue with this algorithm is the data dependency. The multiplication (step 2b) can only be performed after the result of a squaring operation (step 2a) and the following squaring can only be performed after the result of the previous multiplication. Therefore, the total number of multiplications dictates the total time for exponentiation. This causes longer execution time since every single operation has to be performed serially. In Section 2.1, a modified method that allows faster execution time will be discussed.

1.3. Literature Review

As indicated earlier, modular exponentiation and modular multiplication are common operations used in cryptographic algorithms [2, 3, 4, 5]. Modular exponentiation is performed by a series of modular multiplication operations depending on the exponentiation algorithm used. There are many exponentiation algorithms proposed [1] and most of these algorithms have been implemented in software.

There are also some exponentiation algorithms proposed and implemented in hardware. A proposed implementation is the O(n)-depth circuit algorithm for modular exponentiation [12]. It uses logarithmic-depth circuits for powering/exponentiation. This algorithm utilizes the primes that compose the modulus operand and performs several modular exponentiations in parallel before combining the results.

Another hardware implementation uses the MM algorithm for modular exponentiation on reconfigurable hardware in [13]. The MM algorithm used is very similar to the original MM algorithm by P.L. Montgomery in [6]. The radix-2 algorithm implementation is based on a systolic array of processing elements. For the multiplication of operands X and Y with modulus M, each processing element takes a u-bit word of Y and M operands. The u-bit words presented in the literature [13] are 4, 8, and 16 bits. For a precision of N bits, the number of processing elements required is equivalent to N/u since it needs that number of processing elements for full precision to be reached. In the implementation, the intermediate data are represented in redundant form and are only resolved back to binary representation in the end and in the intermediate result feedback. The use of the MM algorithm for modular multiplication makes it less difficult and time consuming compared to other modular multiplication methods. The hardware design was targeted to FPGA technology so that it can be reconfigured easily for different precisions. Designs for FPGAs have certain advantages giving the flexibility comparable to software implementations as well as good performance since it is a hardware implementation. However, it is also

typically a challenge to fit a large fixed precision modular arithmetic architecture due to the limited resources on FPGA chips. The exponentiation algorithm used in this implementation is the binary method as discussed in Section 1.2. The design also utilizes the F_4 exponent for RSA encryption and CRT [1] for RSA decryption to speed up the execution time.

The MM algorithm [6] has been developed into several different variants from its original form. Further work has been done in [7] to analyze and compare several fixed precision MM algorithms in terms of time and space requirements. A word-based MM algorithm and architecture is proposed in [8], where the operand precision applied to the system may vary instead of being fixed. In [8], the radix-2 word-based MM algorithm is developed and implemented in a pipeline of processing elements. The number of processing elements can be selected arbitrarily unlike the implementation in [13]. A processing element consists of carry save adders and the intermediate data is represented in carry save form. Only in the end of the MM process, is the data resolved back to binary representation by use of a final reduction step.

Further work from [8] is done in [9] to develop a scalable and unified multiplier architecture. The unified architecture refers to the use of the multiplier in both Galois Fields of GF(p) and GF(2^k). The implementation is possible without substantially increasing the required area since in GF(2^k) operation, carries do not need to be propagated.

An ASIC implementation of the radix-2 word-based MM algorithm was developed in [10]. High Radix MM algorithm [10, 11] has been developed for certain maximum precision implementation. In [10, 11], high radices were applied over the word-based MM algorithm, introducing the ASIC design for radix-8 implementation. The result shows an improved execution time compared to that of the radix-2 implementation, especially when a larger number of processing elements are used. The radix-2 ASIC implementation of the scalable MM algorithm is used to develop the scalable modular exponentiation system.

Chiou in [16] proposed a parallel implementation of the RSA public-key cryptosystems. A parallel binary method algorithm is introduced to utilize exponentiation operation in certain parallel hardware implementations. This algorithm has advantages where certain operations can be performed in a parallel instead of serial fashion as in the regular binary method in Section 1.2. This parallel binary method algorithm is also used for the scalable modular exponentiation system in this work.

2. PARALLEL EXPONENTIATION ALGORITHM AND CURRENT SCALABLE MONTGOMERY MULTIPLICATION IMPLEMENTATION

2.1. Parallel Binary Method Algorithm for Exponentiation

The following notion extends the notation used for the exponentiation algorithm: *Sq* – temporary result of squaring operation.

The parallel binary method proposed by Chiou [16] is a modified version of the regular binary method. This algorithm is developed to perform fast exponentiation in a parallel processing environment. Similar to the regular binary method, this method scans the bits of the exponent; however, the scanning is performed from the least significant bits (LSB) to the most significant bits (MSB). The Figure 2.1 describes a modified algorithm from the original parallel binary method of $C = X^Z \bmod M$ as proposed in [16].

Input: K, X, Z, M
Output: $C = X^Z \bmod M$
1. Set C = 1, Sq = X
2. For i = 0 to K-1
   ======== *parallel* ==========
   2.a. $Sq = Sq*Sq \bmod M$ (squaring)
   2.b. If $Z_i = 1$ then $C = Sq*C \bmod M$ (multiplication)
   ======== *parallel* ==========
3. Return C

Figure 2.1. Parallel Binary Method Algorithm

It can be seen that this algorithm performs somewhat differently than the previous algorithm discussed in Section 1.2. In this algorithm, both operations in step 2a (*Sq*Sq mod M*) and step 2b (*C*Sq mod M*) are performed simultaneously on specific parallel hardware. The total number of multiplication and squaring operations is still the same as the regular binary method. However, by performing multiplication and squaring operations simultaneously, the total time for exponentiation corresponds to the time to perform K multiplications. The total time is the same despite the number of 1 bits in the exponent Z. Thus, this algorithm performs a faster exponentiation, provided the existence of parallel hardware. Table 2.1 shows an example of the algorithm implementation for Z = 10110110.

i	Z_i	Step 2a	Step 2b
Init	-	X	1
0	0	$(X)^2 = X^2$	1
1	1	$(X^2)^2 = X^4$	$X^2.1 = X^2$
2	1	$(X^4)^2 = X^8$	$X^4.X^2 = X^6$
3	0	$(X^8)^2 = X^{16}$	X^6
4	1	$(X^{16})^2 = X^{32}$	$X^{16}.X^6 = X^{22}$
5	1	$(X^{32})^2 = X^{64}$	$X^{32}.X^{22} = X^{54}$
6	0	$(X^{64})^2 = X^{128}$	X^{54}
7	1	$(X^{128})^2 = X^{256}$	$X^{128}.X^{54} = X^{182}$

Table 2.1. Parallel binary method implementation example

The LSB to MSB scanning means that it has no way of knowing whether further squaring is necessary before the whole Z is consumed. Consider the case where

Z = 00000001. This will cause the system to continually perform squaring until i = 7, only to find that the result is already available when i = 0. This can be avoided by performing pre-scanning of the Z operand to get the location of the last or most significant 1 bit. Once the location is known, the K is set to that value. In the case where Z = 00000001, then K = 1 instead of K = 8. By doing this, there will be no unnecessary squaring operations performed.

2.2. The Scalable Montgomery Multiplication Hardware Implementation

From the Multiple-Word High Radix Montgomery algorithm, Todorov [10] has developed ASIC implementations of the scalable Montgomery multiplier in radix-2 and radix-8, which will be called MM module from now on. Roger Traylor has implemented the input/output (IO) portion for the MM module in order to make it appropriate for a system design. This IO includes the memory blocks (FIFOs) and controls. The system level datapath architecture can be seen in Figure 2.2.

The system has a d-bit IO bus that can be used to input control signals and operands' data as well as output the system status and result data. The system also has a 4-bit address line as well as 1-bit signal for read (*rd_n*), write (*wr_n*), chip select (*cs_n*), and data ready (*rdy_n*). The address line indicates which register to read or write, controlled by the rd_n or wr_n signal. At the beginning, the operands X, Y, and M data need to be inputted to the FIFO registers inside the system. Although the MM module can perform despite the operand's precision, these FIFOs are the limiting factor since they have to be able to hold the full precision of the data.

After the data load operation is complete, the address line can point to the d-bit control register, where the control information word now provided in the IO bus will be stored. The d-bit control word contains information regarding the operation to be performed, start, operand size, and software reset among others. When a new control word is written to the control register, a *start op pulse generator* (part of control

block) is invoked. This pulse generator will create a one-cycle *start op* signal inside the system. When all the data are loaded, the command requesting to begin an MM operation can be issued as part of the control information.

During the multiplication, individual bits of X must be isolated. A w-bit word of operand X is loaded to the *funnel* to feed the correct k-bit word of X to the MM module depending on the radix implementation. Following the Multiple Word High Radix MM algorithm, k bits of X have to be provided to the MM module, where k = 1 for radix-2 and k = 3 for radix-8 implementation. The operands Y and M are read and transferred at w-bit words to the MM module.

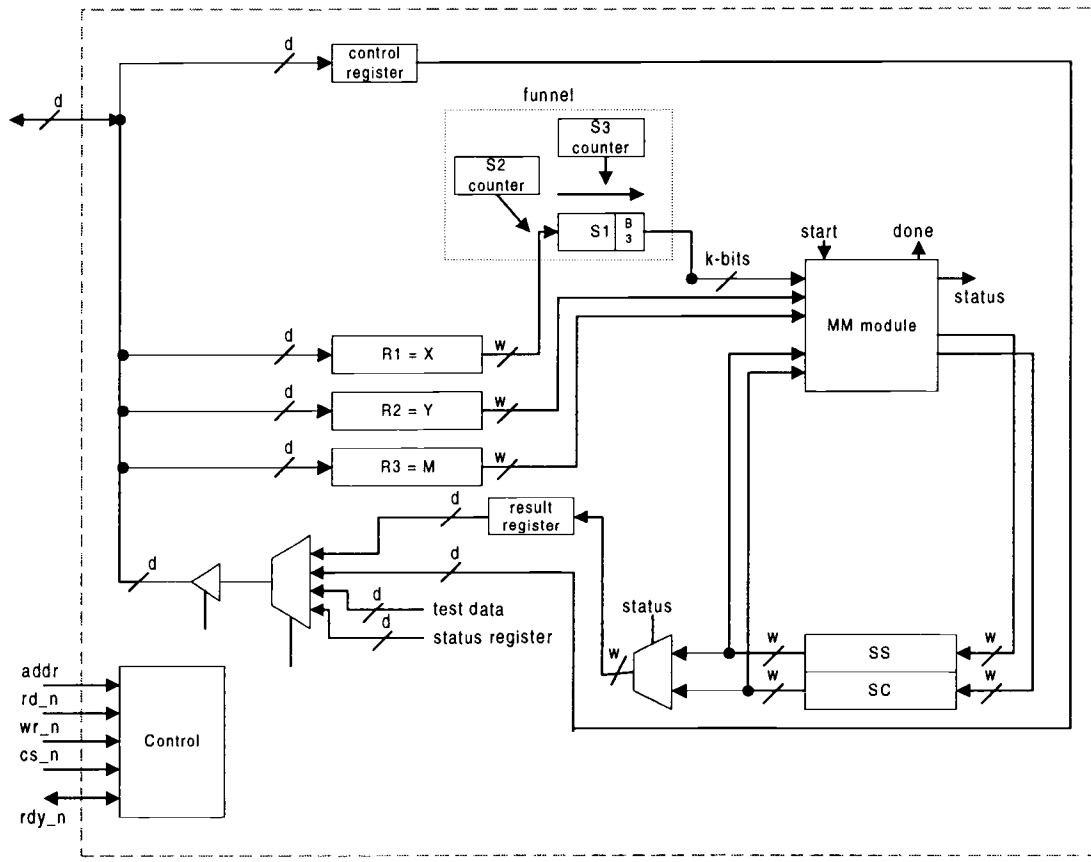


Figure 2.2. System level architecture for MM hardware (datapath only)

The MM module sends out w-bit words of the partial results to SS and SC registers since it produces partial results in redundant (Carry Save) form. The SS and SC partial results are fed back to the MM module repeatedly until all the bits of X are processed. In the final step of the MM module's operation, either the SS or SC register will hold the correct final result of the multiplication.

At any time, the system status can be read by setting the address line to the status register. The status register reports information such as MM operation done, and empty/full status of FIFO registers. The FIFOs information reported are that of X, Y, M, SS, and SC registers.

When the MM module completes its operation, it issues a *done* signal that is forwarded to the status register and is reflected on *rdy_n* line. After the rdy_n signal is issued, the user can check for status and pick up the result data from the system. The result data is stored in either the SS or SC register as indicated earlier. The MM module informs the location of the correct result through its *status* signal. The result can be picked up at the IO bus by setting the address line to point to the result register. The result register is used to recompose the w-bit data from SS or SC to d-bit suitable for the IO bus. This re-composition will be referred to as data reformatting. In case d and w are of the same size, data reformatting and result register are not necessary.

In general, there are two main functional blocks inside the MM module: *Kernel* and *Final reduction block.* The Kernel is the block where the computation takes place. The final reduction block conditionally subtracts M from the Kernel output in order to keep the final result within the modulus M. This can be seen in the Figure 2.3.

The *datapath control* coordinates the operation of the kernel block. As indicated in the system level description, k-bit words of X and w-bit words of Y and M are fed to the MM module. The words of X are held in the REGF register for 2 cycles following the operation of the kernel block, which will be discussed later. The Y* and M* indicate w-bit words of Y and M operands. They are held in DFFR (D-FlipFlop with Reset) register for 1 cycle only. In the beginning, SS and SC registers are filled with zeros, indicating that there are no previous partial results. Again SS*

and SC* refer to w-bit words of SS and SC. The kernel is fed with a new word of X every 2 clock cycles and fed with Y*, M*, SS*, and SC* every clock cycle.

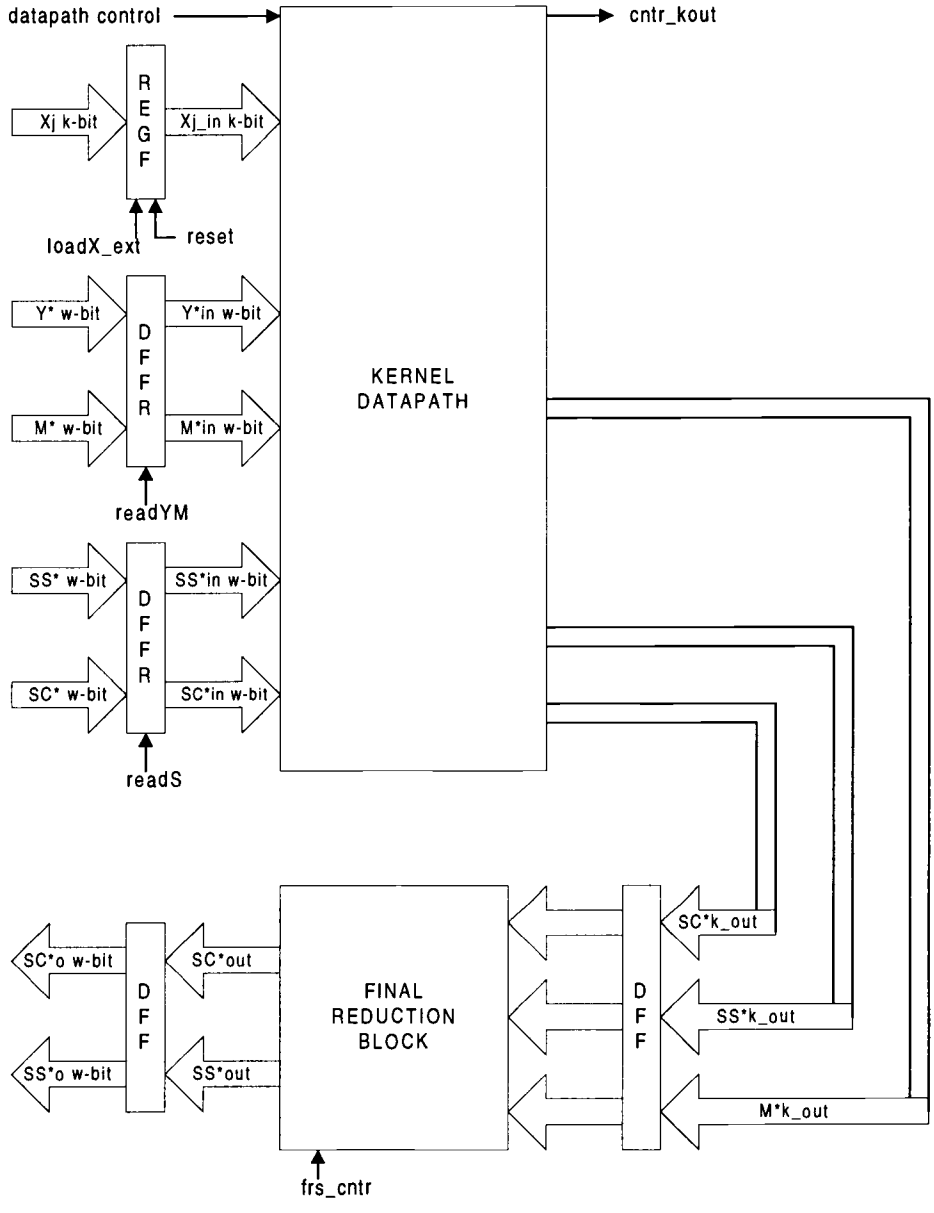


Figure 2.3. System Level Diagram of MM Module (datapath only)

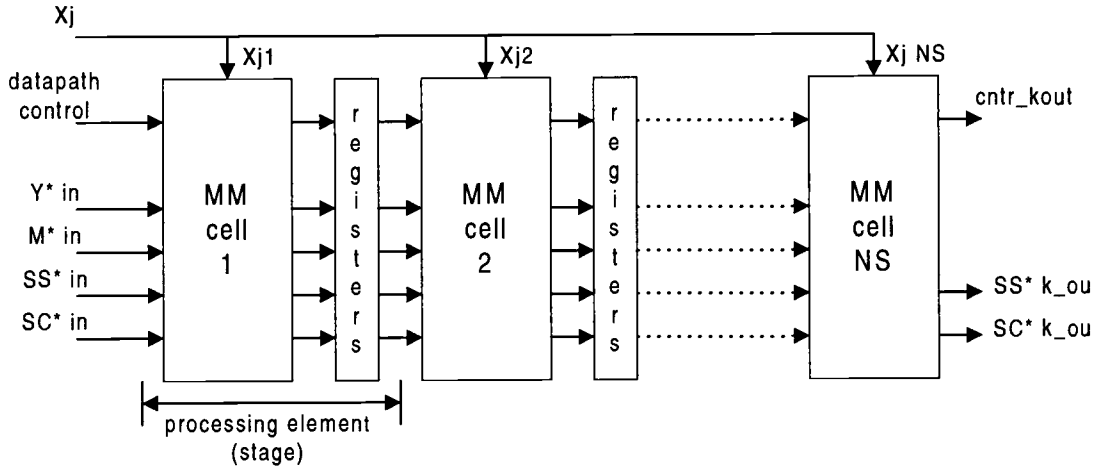


Figure 2.4. System Level Diagram of Kernel (datapath only)

The kernel is composed as a pipeline of MM cells separated by registers as can be seen in Figure 2.4. An MM cell and the preceding register are referred to as a *processing element* or *stage*. The unit receives w-bit words of Y, M, SS, and SC each clock cycle. Over 2* NS clock periods, (NS*k) bits of X are fed to the kernel. At every second clock period, k bits of X are loaded in a stage. The MM cell propagates the Y* and M* as well as the partial results in redundant (CS) form referred as SS* and SC* to the next MM cell, which performs the next iteration of the Montgomery Multiplication algorithm. In turn, the following MM cell propagates Y*, M* and the newly computed SS* and SC* to the next MM cell. These results are kept in DFF (D-Flip-Flop register) for 1 cycle before being forwarded to the final reduction block.

The final reduction block performs addition of SS and SC as well as reducing the addition result to the range [0, M-1]. This is being performed after the last iteration of the loop scanning the bits of X. During the intermediate iterations, the final reduction block only propagates the results and data from the kernel datapath without operating on them. This block basically performs the final step of the Montgomery Multiplication algorithm where $S \geq M$ is being checked. Since this condition is not

known until the final pair of words result are coming out of the kernel datapath, this block implements the computation of both conditions whether $S \geq M$ is true or not. The final reduction block takes advantage of the word size output of the kernel datapath, where the final reduction is performed serially as words are coming out of the kernel. Both results are sent to the DFF registers where they will be stored for 1 cycle before being stored in SS and SC registers. When the last pair of words of S is processed, a status flag is set to indicate which register contains the correct result.

Todorov [10] has proposed several different implementations of the MM module. The simplest design is a radix-2 implementation, which has also been referred to in previous works [8, 9]. Further high radix implementation of the MM module is discussed in [10, 11].

3. PARALLEL EXPONENTIATION ALGORITHM IMPLEMENTATION USING THE SCALABLE MONTGOMERY MULTIPLIER

3.1. Parallel Binary Method Implementation

Modular exponentiation is performed by a series of modular multiplications. Montgomery multiplication provides the most benefit if multiple multiplications are executed instead of just a single one. It also simplifies modular multiplication by comparison to regular modular multiplication method. Thus, the usage of Montgomery multiplier is adequate to implement the parallel binary method for exponentiation.

The parallel binary method uses regular integer field multiplication to perform exponentiation. Since the Montgomery multiplication requires multiplication to be performed in the M-residue field, the parallel binary method has to be modified to work with this multiplier. Figure 3.1 describes the algorithm using MM.

Input: N, X, Z, M
Output: $C = X^Z \bmod M$
1. Set C = 1, Sq = X
2. $Sq = MM(X,r^2) = Xr \bmod M$ (transformation)
3. For i = 0 to K-1
   ======== *parallel* ==========
   3.a. $Sq = MM(Sq,Sq)$ (squaring)
   3.b. If $Z_i = 1$ then $C = MM(Sq,C)$ (multiplication)
   ======== *parallel* ==========
4. Return C

Figure 3.1. Parallel Binary Method Algorithm using MM

The transformation from integer field to M-residue field is performed only once in the 2nd step of the algorithm. The squaring operation is actually a multiplication operation with the special case that both variables being multiplied have the same value. In exponentiation, there are 3 variables - X, Z, and M; however, only X and M will be used as inputs for multiplication (including squaring) operations. Since C is equal to 1 (one) in the beginning, the first multiplication (step 3b) will be between the previous value of Sq and one. This means that the C value will already be in the integer field and recursively so until the last Z bit. The final result of the algorithm will be in C; therefore, only one X integer-to-image transformation is required (step 2) when compared to exponentiation using regular multiplication.

An example of the algorithm execution can be seen in Table 3.1 for Z = 10110110. The X transformation is performed earlier to produce Xr.

i	Z_i	Step 3a	Step 3b
Init	-	Xr	1
0	0	$MM(Xr, Xr) = X^2r$	1
1	1	$MM(X^2r, X^2r) = X^4r$	$MM(X^2r, 1) = X^2$
2	1	$MM(X^4r, X^4r) = X^8r$	$MM(X^4r, X^2) = X^6$
3	0	$MM(X^8r, X^8r) = X^{16}r$	X^6
4	1	$MM(X^{16}r, X^{16}r) = X^{32}r$	$MM(X^{16}r, X^6) = X^{22}$
5	1	$MM(X^{32}r, X^{32}r) = X^{64}r$	$MM(X^{32}r, X^{22}) = X^{54}$
6	0	$MM(X^{64}r, X^{64}r) = X^{128}r$	X^{54}
7	1	$MM(X^{128}r, X^{128}r) = X^{256}r$	$MM(X^{128}r, X^{54}) = X^{182}$

Table 3.1. Parallel binary method implementation using MM

3.2. Modular Exponentiation in a System Implementation

Based on the Montgomery Multiplier system implementation discussed in Section 2.2, an exponentiation system implementation was developed. It has been established that modular exponentiation can be performed by a series of modular multiplications. In the previous section, it was shown that a parallel binary method could be performed by using MM. However, two MM modules are required to explore the parallelism. The system basically should be able to perform both MM and modular exponentiation operations since an image transformation is always necessary in the beginning. The control word loaded by the user or any interface attached to the system would determine what it should do.

3.2.1. Motivations for using parallel binary method in the modular exponentiation system

The main consideration of using the parallel binary method is actually two fold. First, the parallel binary method allows for fast exponentiation execution time as discussed in Section 3.1. Each MM operation is so time consuming that even reducing one MM operation from a series required for exponentiation is a significant reduction in total exponentiation time. Comparing the regular binary method from Section 1.2 and the parallel binary method in Section 2.1, the difference is quite significant in terms of total computation time. Second, the system does not require to double its internal components to run both series of MM operations in parallel. This saving in area consumption while achieving significant reduction in computation time is a benefit in itself.

For a single MM, there are five registers required, namely X, Y, M, SS, and SC. The X register data is fed to the *funnel* that forwards k-bit words to the MM module. The Y and M registers' data are fed to the MM module at w-bit word. The outputs of the MM module are fed to the SS and SC registers at w-bit word. The

exponentiation system will implement the parallel binary method, which needs two MM modules namely MM0 and MM1.

Looking at the parallel binary method example in Table 3.1, it can be seen that step 3b may or may not perform MM while step 3a always perform MM for squaring purposes. However, whenever step 3b performs an MM, one of the input operands is the same with the corresponding operand in step 3a that is running parallel with step 3b. Thus, this multiplicand can be stored in a single register and is fed to both MM modules simultaneously instead of having two separate registers. Further, by having this operand stored in X instead of Y, more saving can be achieved by removing the need of an extra funnel.

Having the first multiplier operand in X means that the second operand is stored in Y. In this case, for step 3a and step 3b, the second operand is not the same. For step 3a, which is performed using MM0, the second operand is the same as the first since it performs squaring. For step 3b, which is performed using MM1, the second operand is the value from the previous step 3b calculation. Thus, each of these MM modules requires separate Y registers. The Y register is now named Y0 register purposely to feed the second operand to MM0. A new Y1 register is added to store the result and feed the second operand to MM1.

The modulus M value is stored in the M register. This register can also feed both MM modules w-bit at the same time. Both MM modules require the w-bit data at the same time. The SS and SC for both MM modules need to be separated since the outputs will be different. Thus, for MM0, the original SS and SC registers will be used and renamed as SS0 and SC0 registers. For MM1, new SS1 and SC1 registers are added. Both SS and SC registers will be fed back to the MM module in mid operations and the timing will be the same for both MM modules. Therefore, the control can be done by one of the modules instead of having replication of the control unit for each MM module.

An extra register Z is required to hold the exponent value. Following the parallel binary method implementation, 1 bit of Z is needed to determine whether MM1 needs to execute. Thus, a 1-bit funnel is required at the output of the Z register.

Since the system should be able to perform both MM and modular exponentiation, there has to be an extra block that will control the internal system operations. This extra block is called *op control logic* block from now on. The op control logic block is composed of state machines that will send out control signals to the rest of the system depending on its state and the input signals it receives. The details of the op control logic implementation will be discussed later in Section 3.2.5.

3.2.2. System level description of the system

The basic hardware design of the system is similar to the MM system implementation discussed in Section 2.2. The system level block diagram can be seen in Figure 3.2. There are a few additions to the original design. The number of full size registers (FIFOs) is increased by four (Z, Y1, SS1, and SC1). An extra one-bit funnel, op control logic block, and MM module are also needed. There are several small additions such as multiplexers (MUXes) and non-full size registers such as result registers for data reformatting. The result registers' outputs are d-bit wide following the size of the IO bus.

The X, Y0, Y1, M, and Z registers are full precision registers with d-bit input and w-bit output. The input is connected to the d-bit IO bus. In exponentiation, it is sometimes necessary that the MM results be copied back to some of the registers; namely to X, Y0 and Y1 registers. MUXes are used to allow selections of inputs for these registers. The details of this requirement will be explained further in Section 3.2.4.

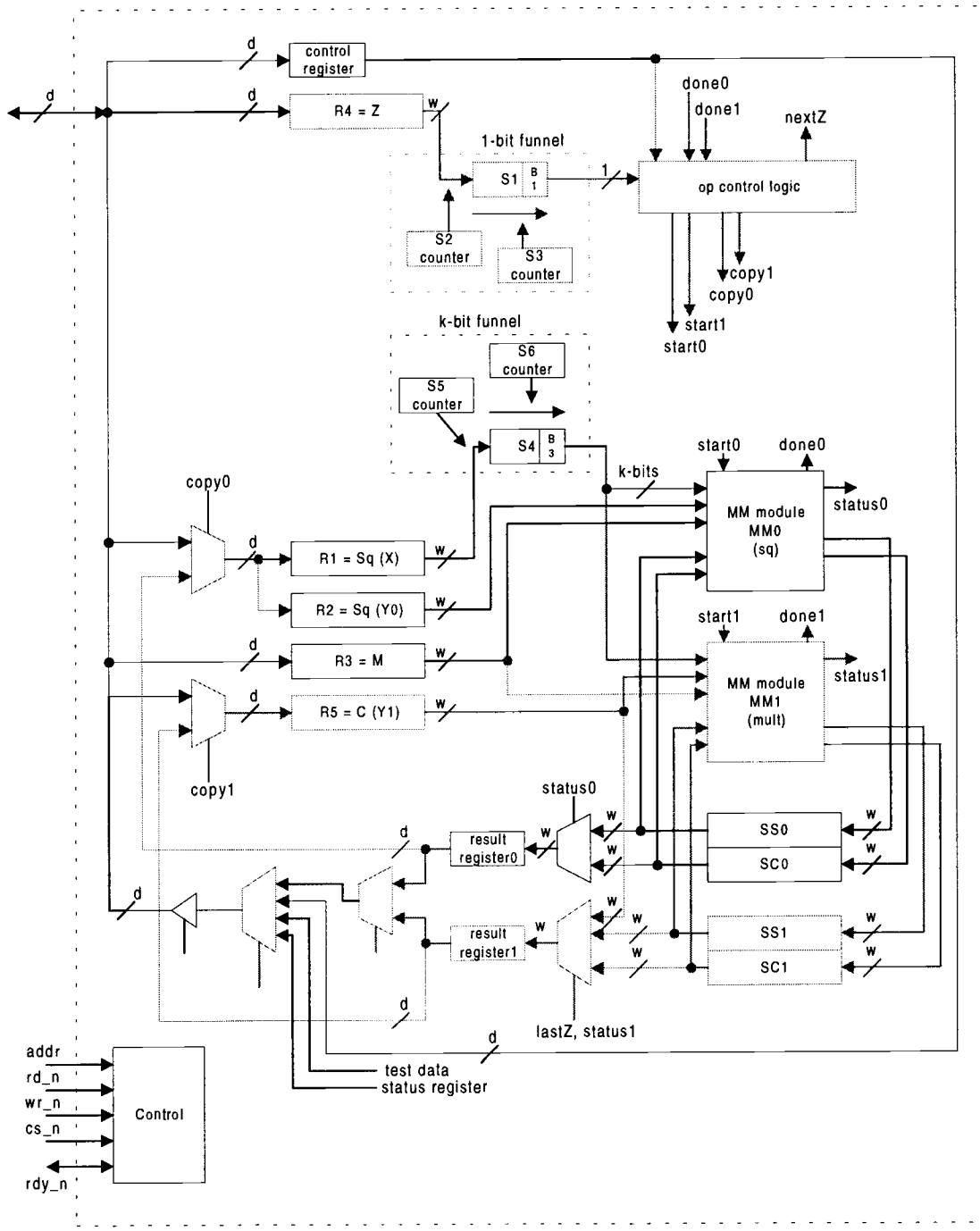


Figure 3.2. System level block diagram (datapath only)

The Z register output is fed to a one-bit funnel. The funnel consists of w-bit shift register and two counters. The purpose of the funnel is to change the input word size to a different word size for output. In this case, the data from Z comes at w-bit words, but the op control logic block needs only one bit of Z each time. Thus, the funnel stores the w-bit data and performs right shifts each time a new Z bit is requested. The rightmost 1-bit of the w-bit register in the funnel is the 1-bit output fed to the op control logic block. This will determine whether only one or both MM modules should be activated.

The X register output is fed to a k-bit funnel. In this case, the w-bit input of the funnel is outputted as k-bit words each two cycles. If radix-2 MM modules are used, then k = 1, which it is the same as the one-bit funnel. The output of the funnel is connected to both MM modules as their first operand input. The Y0 and Y1 registers feed w-bit words to the MM0 and MM1 modules. The M register feeds w-bit word each time to both MM modules simultaneously.

The system designed is restricted to have d less than or equal to w. The full size register has d-bit word input and w-bit word output. Both d and w are at 2^h, where each h is a positive integer and the h for d is larger than the h for w. This way, the w is always a factor of d. The funnel has w-bit word input and k-bit word output. This is inherently different than the full size register such that the k-bit output is arbitrary in size and not necessarily a factor of w. The funnel has a w-bit register to perform k-bit shifting. This would be very beneficial compared to performing shift for large N at the full size register. Further, this way, the full size registers can be general for all rather than specifically designed for a particular operand.

The outputs of the MM module are stored in SS and SC registers. MM0 outputs are stored in SS0 and SC0 registers, while MM1 outputs are stored in SS1 and SC1 registers. The SS0, SS1, SC0, and SC1 registers' input and output are w-bit wide. Most of the time, these registers will hold partial results of the MM modules. When the MM modules are done; either the SS0 or SC0 register will hold the final MM0 result and either the SS1 or SC1 register will hold the final MM1 result.

In case the correct results of MM0 and MM1 need to be copied from corresponding SS and SC registers to other registers with different input size, data reformatting is required. Again, this is where the result register0 and result register1 are used to compose several of the w-bit words to d-bit word. The result register itself is a d-bit register with w-bit input and d-bit output. The MUXes prior to the result registers are used to choose the location of the correct final results, which may be held in either the SS or SC. If MM operation is performed, the final result will be composed in result register0. However, if modular exponentiation operation is performed, the final result will be composed in result register1. Thus, an extra 2-to-1 MUX is placed after these result registers in order to account for these two options.

As indicated before, the final result of the MM operation may be held in SS0 or SC0 register. A 2-to-1 MUX is required in order to forward the correct final result to the result register0. The exponentiation final result may be held in SS1 or SC1 register if the last bit of Z equals to 1 or held in Y1 register if the last bit of Z is 0. Thus, the MUX before result register1 has to be a 3-to-1 MUX.

3.2.3. Control and status signal implementation

As indicated earlier, the control and status signals are composed to be d-bit words stored in d-bit registers in the system. The contents of the control and status words can be seen in Figure 3.3 and Figure 3.4.

The *software-reset* field in the control word is used to reset the system by means of software for warm boot purposes. The *start* field is used to start execution of a certain operation (multiplication or exponentiation, for example). The *reuse register* field is used to retain data in the registers. *Test features* field signal is used when the system needs to be tested. The *operation (op)* field is used to identify operation to perform. *Reserved* field is reserved for possible future need. The *Z operand size word* and *other operands size word* fields indicate the number of words in the Z operand (representing K) and other operands (representing N).

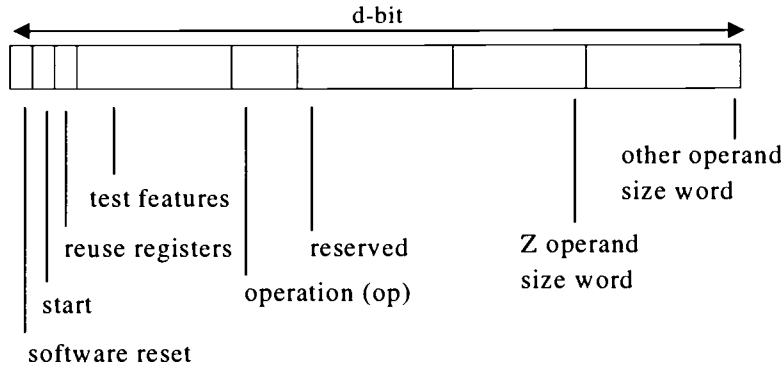


Figure 3.3. Control word description

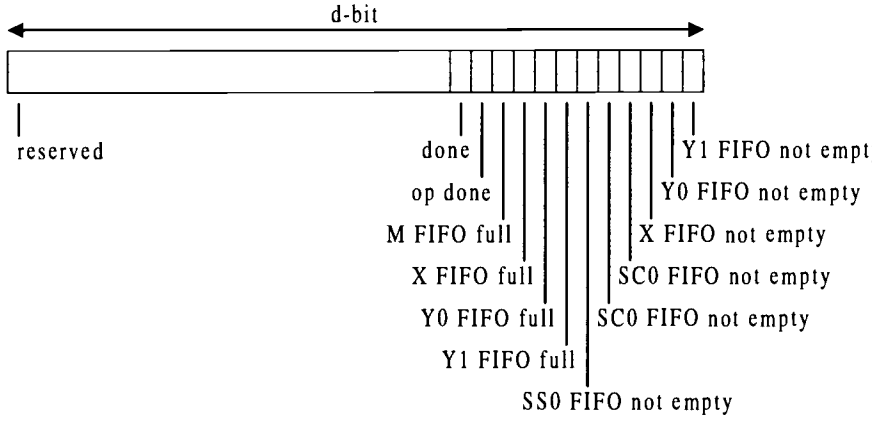


Figure 3.4. Status word description

The status word consists of multiple internal signals regarding the condition of the system. Only a small portion of the status word is currently used while the rest is reserved. The *done* and *op done* indicates whether the operation has finish execution. The *FIFO full* signals indicate whether certain FIFOs are full. The *FIFO not empty* signals are used to inform whether the FIFOs have data inside.

3.2.4. System functionality in performing MM operation

The system performs MM operation in the same way as the MM system originally designed. The multiplicand and modulus operands are loaded to X, Y0, and M registers. The system then receives the control signal with information regarding the MM operation, operand size, and others. This is stored in the d-bit control register and forwarded to the op control logic block. Every time a new d-bit control word is written to the control register with the *start* signal set high, the one-cycle pulse signal will be issued as the *start op* signal.

When the op control logic receives the start op pulse signal, it will issue *start* signal for MM0 to begin performing. The X register data is loaded one bit each time by using the one-bit funnel. The operands Y0 and M are loaded at w-bit to the MM0 module. The Y0 and M operands are recursively used until the X operand is totally consumed. This requires the Y0 and M registers to be able to repeatedly provide the data until X is consumed. Conceptually, this is done as in Figure 3.5. The data outputted by the register is also fed back to the FIFO register's input as long as the *reuse* signal is high. In case, the output size is not the same to the input size, data reformatting is performed in the feedback loop.

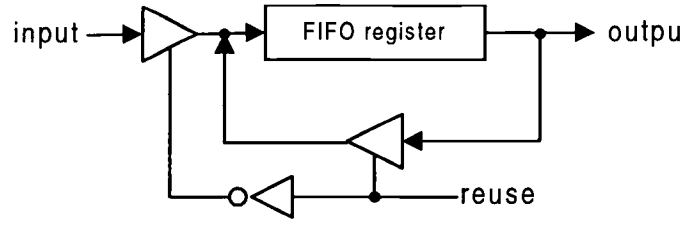


Figure 3.5. Repeating the same data to be outputted by the register

The MM0 module outputs w-bit data of *temporary* results, which are kept in the SS0 and SC0 registers. The SS0 and SC0 registers' input and output are w-bit

wide. Thus, no data reformatting is required. The SS0 and SC0 outputs are fed back to the MM0 module following the MM operation while X is not consumed. When the MM0 module is done, it will issue a *done* signal and the final MM result will be in either the SS0 or SC0 register. The MM0 module will also send out a *status0* signal indicating where the correct result is located. This signal is used by the MUX following SS0 and SC0 such that the correct final result can be reformatted to d-bit word and outputted to the IO bus where it can be picked up.

3.2.5. System functionality in performing modular exponentiation operation

The system performs modular exponentiation operation following the parallel binary method implementation in Section 3.1. This modular exponentiation operation is performed with a series of MM operations running in parallel on both MM0 and MM1 modules. The operand to be exponentiated is first transferred to its image by performing MM of the operand X and the r^2 mod M. The X operand is loaded into the X register and the r^2 mod M is loaded into Y0 register. The MM operation is performed and the result is taken out from the system and stored by user.

This Xr data is reloaded back to X and Y0 registers. The Y1 register is loaded with a value of 1. The Z register is loaded with the exponent data. At this point, the modular exponentiation is ready to start. The control word is applied to the system and stored in the d-bit control register. Knowing that modular exponentiation operation has to be performed, the op control logic block will need to analyze the Z operand bit by bit to determine whether MM1 needs to be activated. MM0 will always be activated since it will perform squaring. The synchronization of both MM modules during their simultaneous execution is crucial since both are fed by the X and M registers.

Both MM modules will output w-bit words at the same time to the corresponding SS and SC registers. When both SS and SC hold the temporary results, those will be fed back to the MM modules at the same time also. Further, when the

corresponding SS or SC hold the final results of each module, they are received and stored at the same time. Thus, the control does not need to be duplicated for each module's operation. When the MM1 is not executing, it doesn't really matter what kind of control signals are received since they will just be ignored by MM1.

	Registers data before			MM execution		Copy function		Registers data after		
Z	X	Y0	Y1	MM0	MM1	copy0	copy1	X	Y0	Y1
0	X_r		1	v	-	v	-	X^2_r		1
1	X^2_r		1	v	v	v	v	X^4_r		X^2
1	X^4_r		X^2	v	v	v	v	X^8_r		X^6
0	X^8_r		X^6	v	-	v	-	X^{16}_r		X^6
1	X^{16}_r		X^6	v	v	v	v	X^{32}_r		X^{22}
1	X^{32}_r		X^{22}	v	v	v	v	X^{64}_r		X^{54}
0	X^{64}_r		X^{54}	v	-	v	-	X^{128}_r		X^{54}
1	X^{128}_r		X^{182}	v	v	v	v	X^{256}_r		X^{182}

Table 3.2. Parallel binary method implementation in the system

A series of MM operations need to be performed for exponentiation and the final result of each MM operation is in either the SS or SC. During the intermediate MM operation, these temporary final results need to be copied back to the registers that will feed the MM module input for the next run. In MM0 case, the result will always need to be copied back to both X and Y0 registers. In MM1 case, the Z bit will determine whether this is necessary. Y1 may just need to retain its previous result or

copy the result from SS1 or SC1. Using the same example as in Section 3.1, Z = 10110110 is used to compose Table 3.2.

The table shows what the registers are holding before and after MM operations and copying are performed. As indicated, MM1 and copy1 may or may not be performed depending on the value of Z. The 'v' sign under MM and copy indicate whether MM and copy operations need to be performed depending on Z. It can be seen that if MM0 is executed then copy0 is executed and if MM1 is executed then copy1 is executed. This happens at different times since copy can only be performed after the MM operation is done.

As can be seen from Table 3.2, in the end, the final exponentiation result will always be in the Y1 register. If the last bit of Z is 0, the final exponentiation result will already be at Y1 register. The Y1 register output is w-bit wide; therefore, data reformatting is necessary before it is forwarded to the d-bit IO bus. If the last bit of Z is 1 then the correct final result can be directly taken from either SS1 or SC1 after data reformatting as in regular single MM operation. Overall, the final exponentiation result may come from Y1, SS1 or SC1. Thus, a 3-to-1 MUX is used to select the input stream for the data reformatting module. The MUX selection is done by combining the last Z bit and the *status1* signal from the MM1. After data reformatting, the final result can be taken out of the system using the d-bit IO bus.

3.2.6. Op control logic block functionality

The op control logic block is responsible for generating the system's internal control signals. Its operation depends on the *op* field, which is part of the d-bit control word given to the system.

When the op is a *"no op"*, the op control logic block remains idle. As discussed earlier, aside to the control word, the system also has input signals for rd_n, wr_n, cs_nt, and address. When these signals are issued, read and write to and from certain registers can be done to the system at this time depending on the address line

signal. This operation has to be done before the system can perform any other operations.

When the op is an *"MM op"*, the op control logic will send out internal control signals for the system to perform MM operation. The state diagram can be seen as Figure 3.6. This operation should only be issued after the operands are loaded into the registers during no op. When start op signal is high, the op control logic will issue *start0* signal to the MM0 module and wait for it to finish execution. MM0 will issue a *done0* signal indicating that it is done. The final MM0 result will be located in either SS0 or SC0 register. The op control logic block will issue a *done* signal that will be visible through the status register. At this point, the data is ready to be reformatted and delivered to the user.

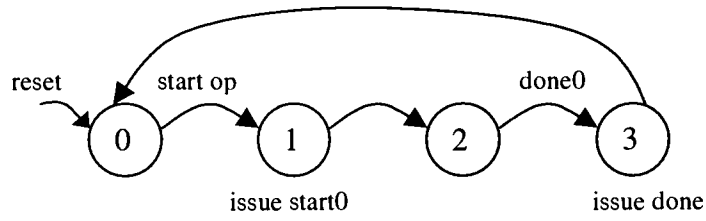


Figure 3.6. Op control logic state diagram performing MM

When the op is *"modular exponentiation op"* and start op pulse signal is high, the op control logic block will perform a more complex task. The state diagram of the op control logic performing modular exponentiation can be seen in Figure 3.7. It basically performs the parallel binary method. Again, this operation should only be performed after the operands are loaded to the registers during no op. The op control logic first checks the bit of Z. If the bit of Z is 1, it will issue *start0* and *start1* signals to MM0 and MM1 modules, which would perform step 3a and 3b. If the bit of Z is 0, it will issue *start0* signal only to the MM0 module, which would perform step 3a

(squaring) only. The op control logic then waits for the *done0* signal from MM0. Only done0 signal from MM0 needs to be checked because both MM modules will always finish at the same time since both operands are of the same precision. Upon receiving the done0 signal, the op control logic issues *clear register* signals to either X and Y0 registers only or X, Y0, and Y1 registers. This is necessary to erase any previous data in the corresponding registers.

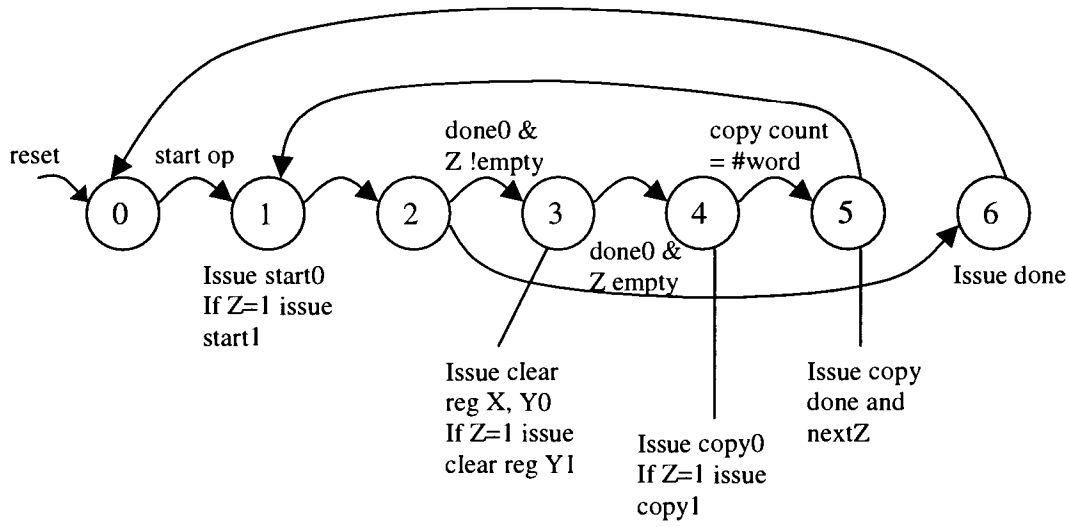


Figure 3.7. Op control logic state diagram performing modular exponentiation

On the next cycle, the op control logic will issue a copy signal to the MUXes and write signals to the corresponding FIFO registers. The *copy0* signal to copy data to X and Y0 registers is always issued. The *copy1* signal to copy data to Y1 register will be issued based on Z = 1. The op control logic counts the number of copies until it fits the operands' number of words described in the d-bit control word. Once the number of copies is satisfied, it will issue *nextZ* signal to request for the next Z bit value followed by a *copy_done* signal that will be used to reset the kernel, SS, and SC

registers. This reset phase is required since the kernel has information regarding its previous state and the SS and SC registers may have a portion of their previous data in them. This functionality is repeated until all Z bits are consumed. At the last Z bit, after MM0 issues the done0 signal, the op control logic will issue a done signal, indicating that the operation is completed. This signal is visible from the status register as in the MM operation. The final exponentiation result can be read from the system at the d-bit IO bus at this point.

4. OPTIMIZATION OF THE SCALABLE MONTGOMERY MULTIPLIER

4.1. Problem Description

The scalable Montgomery Multiplication kernel proposed in [8, 9, 10, 11] consists of a number of processing elements (p) organized in a pipeline. As more processing elements are added, larger area is required for the kernel and ideally, the computation time would be reduced.

Each processing element operates on w-bit words (w) and all processing elements in the same kernel should be of the same word size. Each processing element's area is determined by its w-bit word. The larger the w, the more area is required for each processing element. The total area of the kernel is determined by the values of p and w. Area usage of the kernel (A_{kernel}) in number of gates for radix-2 implementation acquired from [8, 10, 11] is approximated as:

$$A_{kernel} = 59.65pw + 51.44p - 31w - 35.52$$

The kernel computation time is determined by p, w, and the precision (N) in bits of the input operands. The kernel may be used for any N. However, as indicated earlier, ideally a kernel with more processing elements would require less computation time for the same N. If N is of higher precision (larger number of bits) and p is the same, then the computation time increases. The total computation time (T) in clock cycles for radix 2 implementation also acquired from [8, 10, 11] are as follows:

$$T = \begin{cases} 2kp + e - 1 & \text{if } (e+1) \leq 2p \\ k(e+1) + 2(p-1) & \text{otherwise} \end{cases}$$

where $e = \left\lceil \frac{N}{w} \right\rceil$ and $k = \left\lceil \frac{N}{p} \right\rceil$

Total computation time is $T * RC\_delay$, where RC_delay is the time required for each clock cycle in nanosecond (ns). The RC delay value in this work was obtained from synthesis of the kernel using Leonardo software (Mentor Graphics tool) based on 0.5 μm technology and further refined using IC station software. Ideally, the delay should be the same for any number of processing elements in the pipeline provided that the processing elements have the same w and each processing element is isolated from the other by interstate registers. However, the experimental results in [10] were obtained using flattened design, where processing elements are merged together instead of being treated as blocks; the delay increases as p increases.

	N = 256											
w	**8**			**16**			**32**			**64**		
p	Area (gates)	Time (cycles)	Time (us)	Area (gates)	Time (cycles)	Time (us)	Area (gates)	Time (cycles)	Time (us)	Area (gates)	Time (cycles)	Time (us)
1	246	8448	56.7	475	4352	32.3	933	2304	20.6	1,850	1280	13.7
2	774	4226	31.3	1,481	2178	18.3	2,893	1154	12.1	5,719	642	8.1
5	2,360	1698	13.5	4,498	879	8.7	8,774	527	6.8	17,326	523	8.8
6	2,889	1418	11.4	5,504	736	7.3	10,734	523	7.1	21,195	519	9.2
7	3,417	1219	10	6,510	634	6.3	12,695	525	7.5	25,064	521	9.8
8	3,946	1070	8.9	7,516	558	5.5	14,655	519	7.8			
9	4,475	955	8.2	8,522	537	5.4	16,615	529	8.1			
10	5,003	863	7.5	9,527	535	5.5	18,575	527	8.3			
11	5,532	788	6.8	10,533	543	5.7	20,536	535	8.6			
12	6,061	726	6.3	11,539	543	5.8	22,496	535	8.6			
13	6,589	674	5.8	12,545	535	5.8	24,456	527	8.6			
14	7,118	630	5.5	13,551	547	6	26,416	539	8.8			
15	7,647	592	5.1	14,557	555	6.2						
16	8,175	558	4.9	15,562	527	6						
20	10,290	551	5	19,586	535	6.7						
26	13,462	551	4.8	25,621	535	8.3						
35	18,219	591	5.1									
50	26,149	631	6.1									

Table 4.1. Area, time (clock cycles), and time (μs) for each w and p with N = 256 bits

From the two formulas, the area and computation time can be calculated given p, w, and N. For an N of 256 bits, the area and time for possible kernel configuration with area less than 26,500 gates can be seen in Table 4.1. Not all data for all p is shown due to space limitation, but in actual implementation all values of p have to be considered. The reason that 26,500 gates is used as the limit is the available RC delay information obtained in [10].

The information in Table 4.1 can also be seen in graphical form in Figure 4.1. Each line represents area versus time information of an increasing number of processing elements (p) with certain word size w. The graphical form allows for easier manual tracing; just by looking at the graphic, it can be seen where the kernel configurations are located.

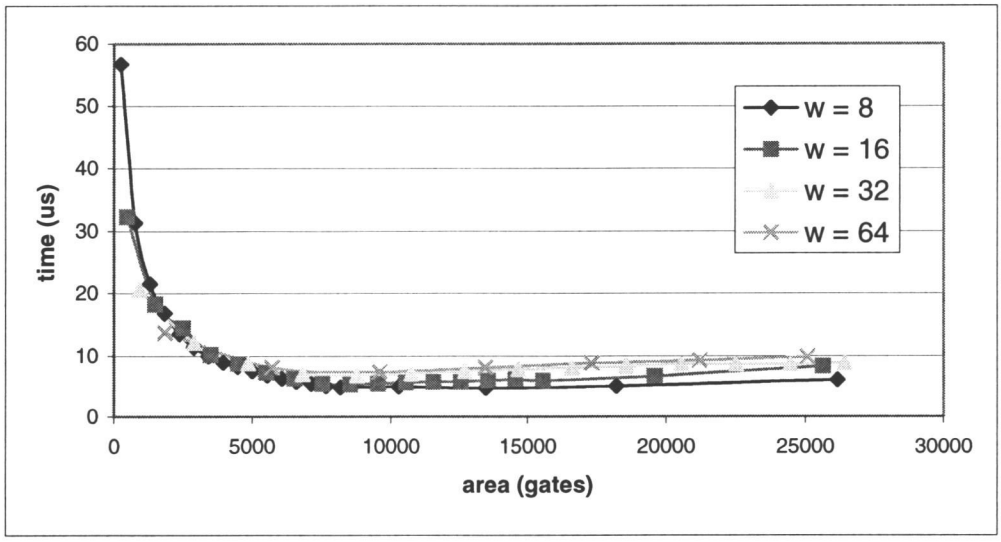


Figure 4.1. Area to time comparison graph for N = 256 bits

In this case of a single given N, a simple lookup or trace can be performed on the data in the table for each w. If an absolute minimal area is the objective, a p of one

is always the result for each w. Table 4.2 shows the absolute minimal area for some values of w. If absolute minimal time is the objective, simply by going through the computational time from the Table 4.1, a minimal value can be found. Hence, a p value that provides the minimal computation time can be found for each w. Ideally in a scalable design, increasing p should yield faster computation; however, as the data indicate, this is not the case. The computation time may increase or decrease for increasing values of p. Table 4.3 shows the absolute minimal computation time for some values of w.

These absolute minimal area and minimal time solutions for each w are extreme approaches and the p value for each w can be easily found. Looking at the data in Table 4.1 more closely and modifying the objective slightly, what would be a "reasonable" p to use instead of going for the absolute minimal area or minimal computation time? Even before that, what is "reasonable"?

N = 256				
w	**Area (gates)**	**Time (cycle)**	**Time (us)**	**p**
8	246	8,448	56.7	1
16	475	4,352	32.3	1
32	933	2,304	20.6	1
64	1,850	1,280	13.7	1

Table 4.2. Minimal area for each w with N = 256

From Table 4.1, it can be seen that for w = 8 and p = 16, the computational time is 4.9 μs with a total area of 8,175 gates. Going further, for w = 8 and p = 26, it yields the minimum computational time of 4.8 μs with a total area of 13,462 gates consumed. Thus, is it "reasonable" to choose a computational time of 4.8 μs instead of 4.9 μs while sacrificing extra 5,287 gates? A comparable question arises when looking

at w = 16, comparing p of 8 and 9, with only 0.1 μs improvements in computational time while sacrificing an extra 1,006 gates. Are any of these worthy area-time trade offs?

N = 256				
w	**Time (us)**	**Time (cycle)**	**Area (gates)**	**p**
8	4.8	551	13,462	26
16	5.4	537	8,522	9
32	6.8	527	8,774	5
64	7.4	519	9,588	3

Table 4.3. Minimal time for each w with N = 256

If a designer is certain on N and w, finding the minimal area or computation time is straightforward. Even if the designer is uncertain on which w to use, the task is still straightforward. Intuitively, w = 8 and p = 1 is always the absolute minimal area for the kernel, provided that total computation time is irrelevant. Also, the absolute minimal time is obtained at w = 8 and p = 26, yielding a total computation time of 4.8 μs, provided area is not relevant. However, given the previous modified question regarding a "reasonable" p to use, which w would come up to be reasonable in the end? Looking at Table 4.2, would w = 8 with an area of 246 gates and computation time of 56.7 μs be better than w = 64 with an area of 1,850 gates and computation time of 13.7 μs? Or, looking at Table 4.3, would w = 8 with an area of 13,462 gates and computation time of 4.8 μs be better than w = 16 with an area of 8,522 gates and computation time of 5.4 μs?

As mentioned in the beginning, the kernel may be used for any value of N. In the previous setup, a single N = 256 is used. However, in the actual usage of the kernel, N may vary. These extra degree of freedom raises another question: which p at

each w should be used that would perform reasonably well for all the values of N being used? Further, if the designer is uncertain about the w value to use, which p and w should be used in order to perform reasonably well for all N values of interest? If these questions are answered, the p and w values will be the parameters for the optimized kernel design for the corresponding N sizes.

In summary, the first question to be solved is the trade off analysis method that would yield the "reasonable" p given fixed w and N. Increasing the complexity, the second question is to find the "reasonable" p and w for a single fixed N. The third and fourth questions deal in finding the solutions for a set of N sizes.

As discussed earlier, modular exponentiation is performed with a series of modular multiplications. Thus, an effective choice of w and p for kernel implementation would provide an efficient solution in terms of time and area for the kernel in the exponentiation system implementation.

4.2. Considerations and Methods Discussion

Given that the proposed design's problem increases in complexity at each level, it makes sense to try to solve the lower level problem first and proceed from there.

4.2.1. Finding p for particular w and N

The first problem is to find p for each w at a particular N that would consume "reasonable" area while performing at a "reasonable" computation time. The term *reasonable* is relative, thus it needs to be defined in order to find the correct approach in solving the problem. Reasonable in this case means that given p, the design would perform relatively fast while it consumes as minimal area as possible. This is inherently different than going for the absolute minimal time where area usage is not

an issue or going for the absolute minimal area where the computation time is not an issue.

Since the best solution for this first problem is to reach almost minimum computation time while consuming almost minimal area, there has to be an evaluation method that considers all possible trade offs and determines if a design (with certain p) is good enough. It is difficult to say that an addition of 1,000 gates is a worthy trade off for a gain of 1 μs in computation time. Clearly, it is not possible to simply compare gate unit and μs unit for trade off analysis purposes.

A common way to evaluate such trade off is by using a gain/cost estimate method. This method quantifies the cost of each additional gate and the gain per reduced execution time unit. If the total of the two is positive, meaning there are more gain than cost, it is a worthy trade off. Instead, if the total is negative, then it is not a worthy trade off. However, while it is possible to obtain a cost estimate for each additional gate, it is relatively more difficult to obtain the gain value for each reduced time unit.

Looking at the cost estimate method more closely, it is clear that the objective is to scale area (in number of gates) and time (in μs) to a common domain for comparison. The comparison process itself is relatively easy and straightforward. Consider three points in Cartesian coordinate system: A(2,7), B(4,4), and C(6,3). The goal is to find the point with maximum gain in comparison to the other points. To compare one point to the other, the differences in x and y values are calculated. Both differences are then added, if the result is positive, the latter point is chosen. If the result is negative, the prior point is retained to be the best point. If the result is equal to zero, both points are the same and the best point can be chosen arbitrarily. The total gain/cost formula and comparison procedure example is as follows:

$$Total\ gain/cost = (X_1 - X_2) + (Y_1 - Y_2)$$

Point A is assumed to be the best point in the beginning.

Comparing point A and B: (2-4) + (7-4) = 1 → B is chosen to be a best point

Comparing point B and C: (4-6) + (4-3) = -1 → B is still the best point

Thus, point B is the best point amongst the three points.

While the cost estimate method cannot be implemented as is, there may be another way to scale area and time into a single domain. If time and area can be transformed into a common domain, then comparison can be implemented similar to the previous example.

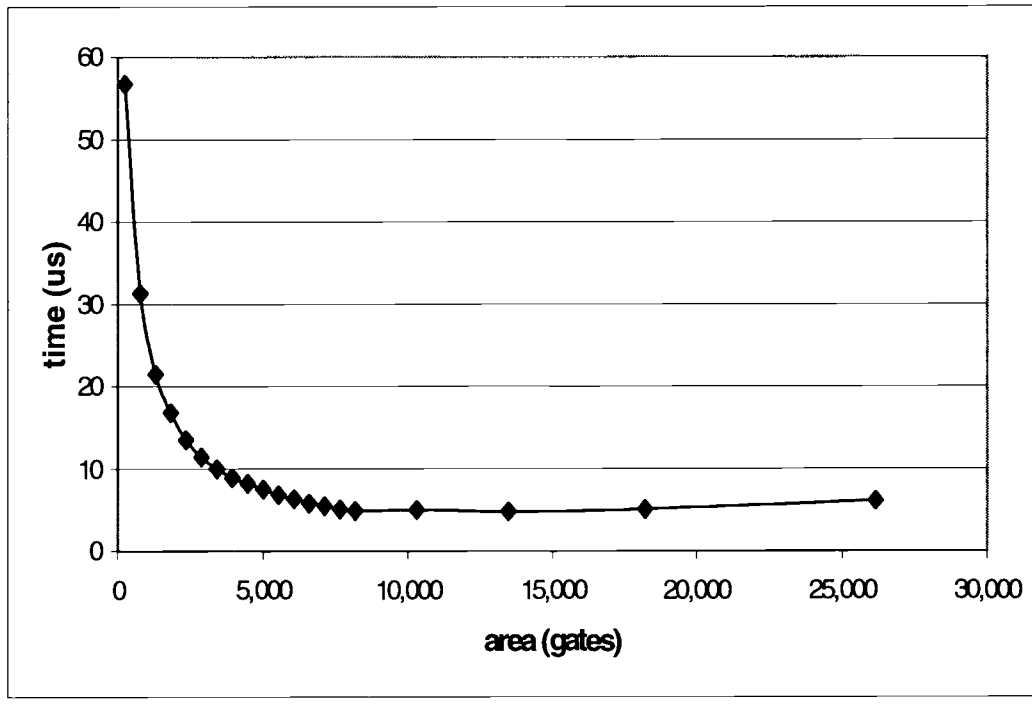


Figure 4.2. Area to time comparison graph for N = 256 with w = 8

In any hardware implementation, it is not possible to have unlimited area for hardware. The designer has to know the area limitation for hardware implementation. From this information, the maximum and minimum area for each value of w can be obtained. The maximum area (*max*_area) for the kernel implementation should be less than the given area limitation. The minimum area (*min*_area) for each w is found when

p is one. These maximum and minimum areas define the area range of the kernel. Within this area range, it is also possible to find the maximum and minimal computation time. The maximum time (*max_time*) for each w is found when p is one where the least number of processing elements is used. The minimal time (*min_time*) may vary depending on N used. For the area limit of 26,500 gates, the graph that compares area and time for N = 256 and w = 8 can be seen in Figure 4.2.

The main issue now is to transfer the area and time into a common domain in order to evaluate trade offs and find the best design. It can be found that the min_area is 246 gates and the max_time is 56.7 μs, which is obtained at p = 1. The min_time is 4.8 μs with an area of 13,462 gates, which is obtained at p = 26. The max_area of 26,149 gates is reached at p = 50. The difference of max_area and min_area as well as max_time and min_time can be calculated. The difference between max_area and min_area is called *area_range*, while the difference between max_time and min_time is called *time_range*.

Given this information, it is possible to present the data in magnitudes of time and area domains into percentage changes of time and area domains. The way to do this is by scaling time according to time_range and scaling area according to area_range. The percentage changes represent the difference of each design relative to time_range and area_range. The two originally different domains of time and area are now represented in the same percentage change as can be seen in Figure 4.3. The formulas to perform these scaling is defined as follows:

$$scaled\_time = \frac{time}{time\_range}$$

$$scaled\_area = \frac{area}{area\_range}$$

where $time\_range = max\_time - min\_time$

$area\_range = max\_area - min\_area$

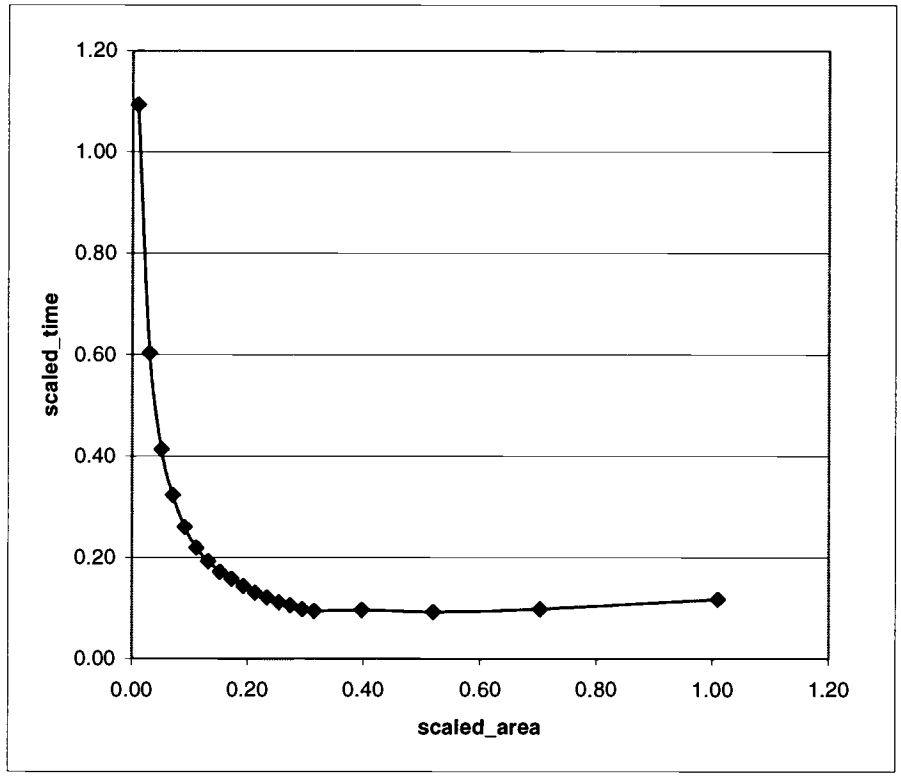


Figure 4.3. Scaled_area to scaled_time comparison graph for N = 256 with w = 8

The previously discussed scaling requires calculation on both area and time data into percentage changes of area and time. It is possible to modify the formula slightly so computation is performed on only one domain of the data. Since area has a limitation as given by the designer, time is selected to be scaled into area. Since time is scaled to area, the area does not need to be scaled. The modified formula can be seen as follows:

$$scaled\_time = time * area\_to\_time\_ratio$$

where $area\_to\_time\_ratio = \frac{area\_range}{time\_range}$

It can be seen that the scaled_time represents the percentage time changes in terms of area and the other calculation is not needed. The originally two scaling calculation can be reduced to one calculation for scaled_time only. The magnitude of time and area in their separate domains can now be compared in a common percentage changes in area domain. The result of the scaling for w = 8 and N = 256 can be seen in Figure 4.4.

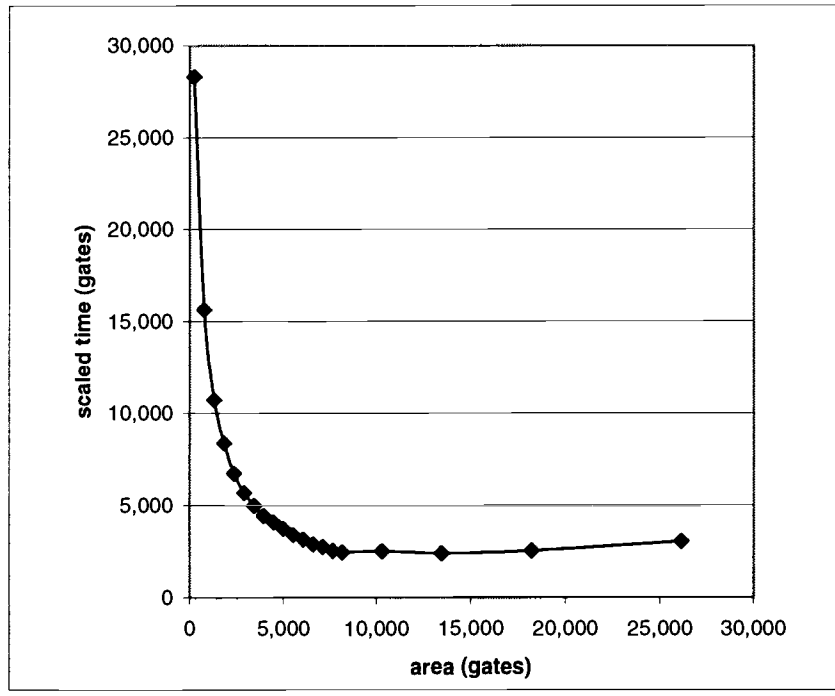


Figure 4.4. Area to scaled_time comparison graph for N = 256 with w = 8

It should be noted that the time is not actually converted to area in actual physical term. The scaled_time just represents the percentage changes of time in terms of number of gates for trade off analysis execution.

N = 256 w = 8					
p	Area (gates)	Actual time (μs)	Scaled time (gates)	Total gain/cost	Best p
1	246	56.7	28,299	-	1
2	774	31.3	15,622	6,075	2
3	1,303	21.5	10,731	2,181	3
4	1,832	16.8	8,385	909	4
5	2,360	13.5	6,738	560	5
6	2,889	11.4	5,690	260	6
7	3,417	10	4,991	86	7
8	3,946	8.9	4,442	10	8
9	4,475	8.2	4,093	-90	8
10	5,003	7.5	3,743	-89	8
11	5,532	6.8	3,394	-90	8
12	6,061	6.3	3,144	-140	8
13	6,589	5.8	2,895	-140	8
14	7,118	5.5	2,745	-190	8
15	7,647	5.1	2,545	-165	8
16	8,175	4.9	2,446	-215	8
20	10,290	5	2,495	-1,082	8
26	13,462	4.8	2,396	-1,537	8
35	18,219	5.1	2,545	-2,453	8
50	26,149	6.1	3,044	-4,215	8

Table 4.4. Trade off between area and time with equal importance

Having time scaled into area, the previously two different domains are currently in a common area domain. Thus, trade off analysis can be performed as discussed earlier. If the trade off analysis were performed as is, this would imply both area and time have the same importance. It can be assumed that the total importance of both area and time is 100%. If both area and time are of the same importance, this means that area is 50% important and time is 50% important. For the case where absolute minimal time is the goal, it can be said that time is of utmost importance, thus the time importance is 100%. On the other hand, if absolute minimal area is the goal, the area importance is 100%. This importance ratio is to be determined by the designer

and can be implemented over the trade off analysis method. The total gain/cost formula can be modified slightly to implement the importance ratio as follows:

$$Total\ importance = R_x + R_y = 100\%$$

$$Total\ gain/cost = R_x\,(X_1\text{-}X_2) + R_y\,(Y_1\text{-}Y_2)$$

where R_x is the X importance and R_y is the Y importance.

In terms of area and time consideration, the X domain is area in number of gates and the Y domain is time (scaled time) in number of gate. R_x is the area importance while R_y is the time importance.

Table 4.4 represents the trade off analysis when area and time are equally important (50% importance each). It can be seen that for w = 8 and N = 256, the p = 8 is the best amongst all possible configurations with area less than 26,500 gates given as the limitation.

4.2.2. Finding p and w for a particular N

The second problem is to find the best p and w to use at particular N, assuming that the designer is not sure of what w to use. The kernel still has to perform in a reasonable computation time while consuming reasonable area. The previous solution can be used to find the best p for a given w and N. The best p of each w can be found using the trade off analysis method. A simple example would be by looking at Table 4.5 for N = 256 and time importance of 50%. However, these results cannot be compared between one to another since they are obtained separately for each w. The separation implies different area_to_time_ratios and thus, the designs are scaled differently at each w. For fairness, all configurations have to be scaled in the same way to insure equal comparison.

Instead of separating the data for each w, the data can be looked at differently. A way to do this is to look only at the area consumption of each configuration. The area and computation time for all the possible designs for all values of w are first

calculated. From here, the information of w is retained, but not used to categorize or separate the data as in Table 4.1. The data is re-sorted according to the area consumption. The result can be seen in Figure 4.5.

N = 256			
w	**Time (μs)**	**Area (gates)**	**p**
8	8.9	3,946	8
16	7.2	5,504	6
32	6.9	6,814	4
64	8.1	5,719	2

Table 4.5. Results for each w with time importance of 50% at N = 256

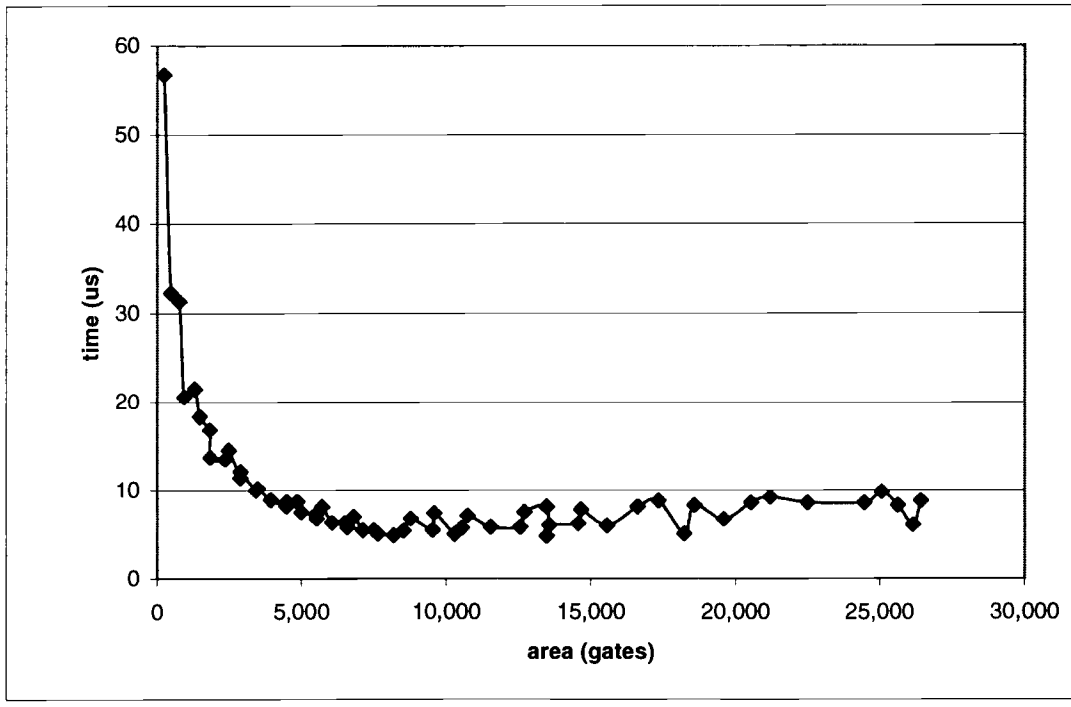


Figure 4.5. Resorted data according to area versus time for N = 256

The computation time in Figure 4.5 can be scaled to number of gates as in the previous section implementation. In the same way, the maximum and minimum area and time information are obtained. The differences between the maximum and minimum are used to find the time-to-area ratio. The area versus scaled time can be seen in Figure 4.6.

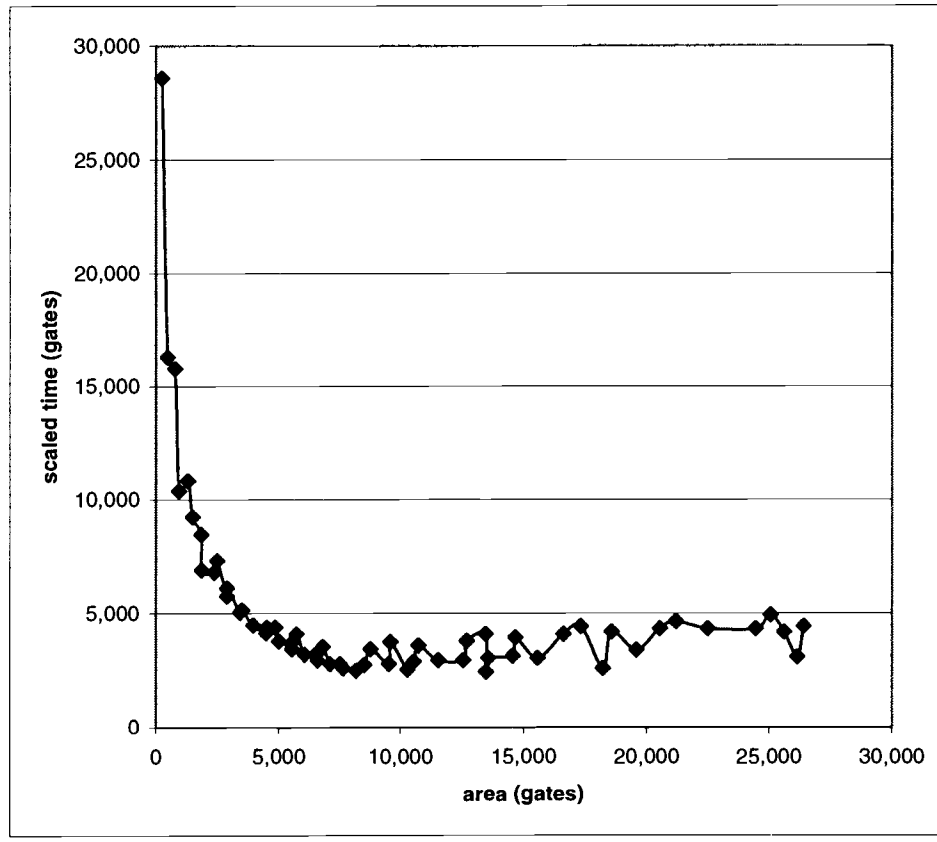


Figure 4.6. Resorted data according to area versus scaled time for N = 256

Again having the originally different domains into a single one, comparison between designs can be analyzed for trade offs. Similar importance ratio can also be

implemented to allow the designer specification in the trade off analysis. A best design can be determined without the data being separated to each w. With this implementation, all the configurations can be compared fairly since all of them are compared equally in the same way. Thus, all the possible designs are compared to each other to find the better design despite its w.

4.2.3. Finding p for particular w and multiple N

As indicated earlier, the kernel can be used for any N. The commonly used N sizes should be considered since the kernel is likely to perform computations for common cryptographic algorithms. For example, the RSA algorithm uses N sizes of 1024, 2048, and 4096 bits. The values may increase in time because the higher the N, the more secure the encryption will be. Thus, if the designer knows which algorithms will be used, he/she would know the sizes of N that the kernel will be performing at.

In high performance computer architecture, where the processor performs many different operations, it is important to make the commonly used operations fast. This is known as Amdahl law. The same principle applies in this case of kernel design, which is to make the execution time on commonly used N fast. The quantitative approach in regular multi operations processor design uses the formula as follows:

$$Expected\_time = \sum_{i=1}^{v} T_i * usage_i \text{ , where } \sum_{all.i} usage_i = 1$$

where v represents the number of operations, T_i is the execution time of operation i, and $usage_i$ is the frequency of operation i.

From the formula, the expected time that the processor needs to perform all the operations can be known. Reducing the time of the operation, which affects the expected time the most, would make the processor finish faster.

In this case, the kernel has to work in a similar way for multiple N values. The designer should know approximately the usage distribution of the corresponding N

values. In case where the kernel is only used for a single N, the expected time will reflect the kernel computation time since the kernel will 100% of the time perform at this N. If the kernel needs to perform modular exponentiation at N of 1024 bits 50% of the time, 2048 bits 25% of the time, and 4096 bits the rest of the time, then this information may be used as a profile to optimize the kernel design. The method will focus on the N values that affect the expected computation time the most, in order to find the best kernel configuration. The formula for the expected time stays the same except for the change in notation as follows:

$$Expected\_time = \sum_{i=1}^{v} T_i * usage_i$$

where v represents the number of precision values the kernel will be used at, T_i is the time for kernel to finish computation with precision N_i, and $usage_i$ is the frequency usage of operators of size i bits.

Figure 4.7 shows the area to time comparison at w = 8 for two values of N: 256 and 512 bits. It can be seen that area increases of designs is the same for both N, indicating the same kernel configuration. However, the performance of a kernel configuration is not the same at different N. Using the expected time formula, if N = 256 is used 50% of the time and N = 512 is used for the rest of the time, the two different lines can be merged into one single line as shown in Figure 4.8. The y-axis now indicates the expected time of kernel execution instead of the actual time. The same method can be used for multiple values of N as long as the usage distribution data is available. Figure 4.9 is constructed for N of 256, 512, 768, and 1024 equally distributed at 25% usage each at w of 8, 16, 32, 64, and 128.

Reaching this stage, the expected time to area scaling can be performed for each w of interest using merged N in the same way as discussed in Section 4.2.1. Trade off analysis can then be performed after scaling expected time into area. The best p for each w can be determined for all N values used in the kernel will be used at, given the usage distribution and importance ratio.

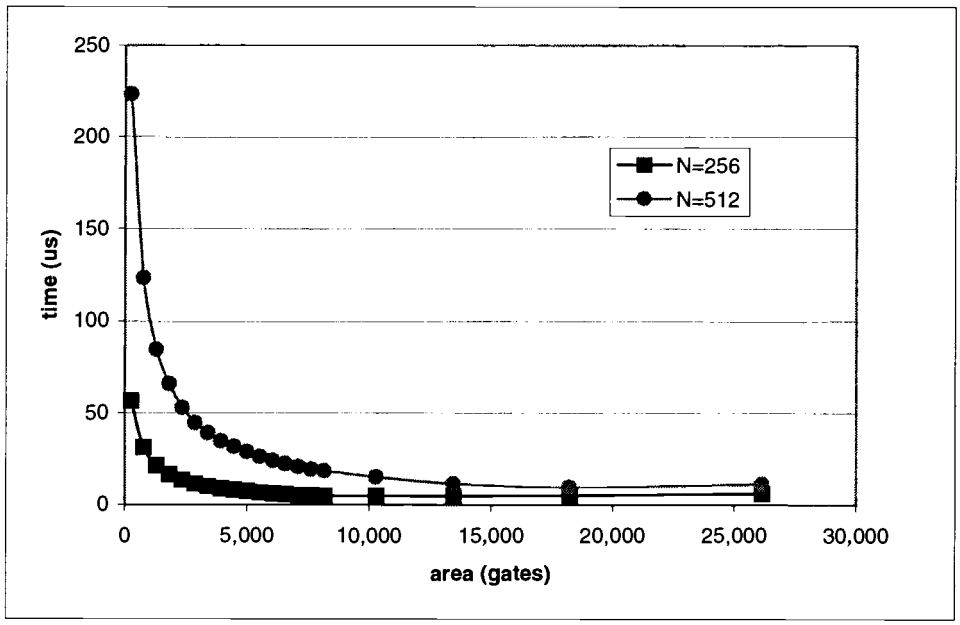


Figure 4.7. Area to time comparison at w = 8 for N = 256 and N = 512

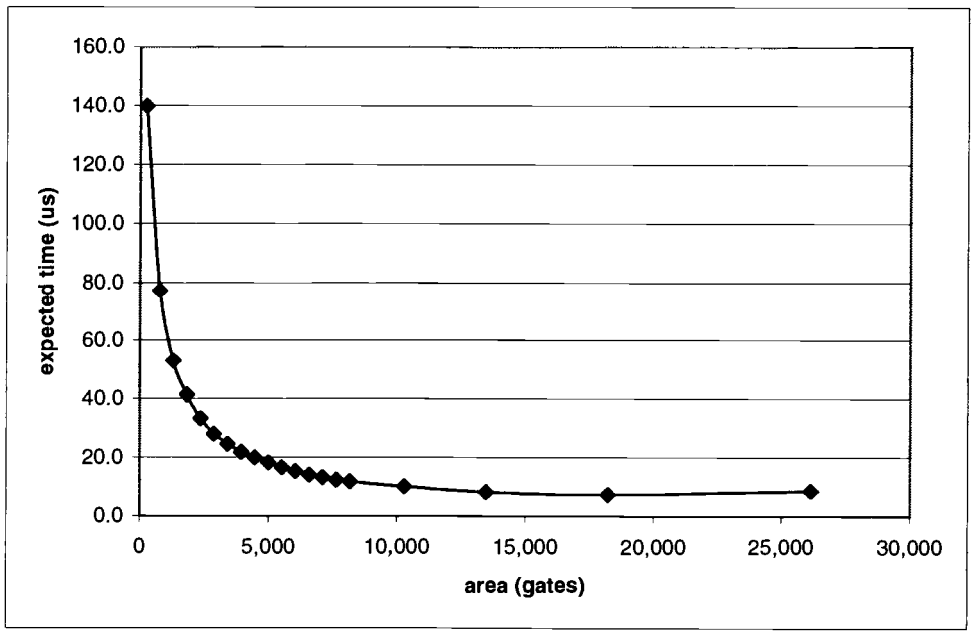


Figure 4.8. Area to expected time comparison at w = 8 for N = 256 and N = 512

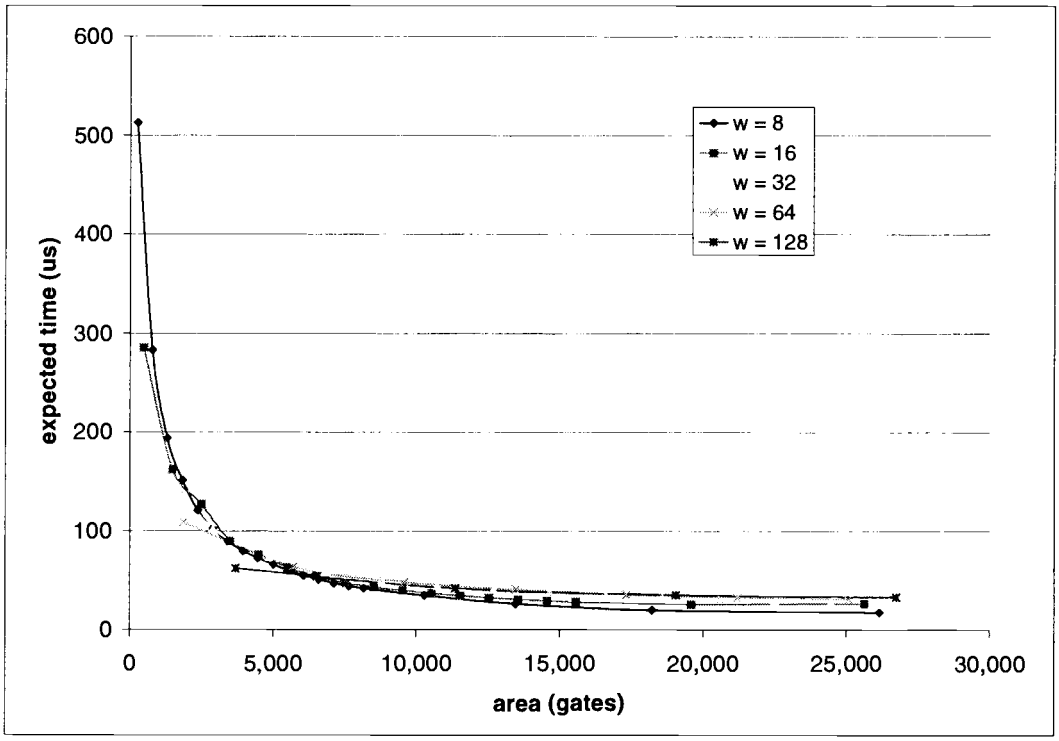


Figure 4.9. Area to expected time considering multiple values of w and N

4.2.4. Finding p and w at multiple N

The expected time method implemented in Section 4.2.3 can again be applied here for the problem of finding the best p and w values when the kernel is used for multiple N. In Section 4.2.2, sorting according to area is implemented over all the designs despite their different w for fairness of comparison. This is performed in case the designer is not certain about which w will be used for the kernel implementation. The same methods can be applied here since it faces similar problems regarding the use of multiple N values and the separation of possible designs according to their w values.

The expected time is first calculated for each p and w. The result will be similar to Figure 4.9, however, instead of separating the possible designs according to

w, the possible designs are sorted according to area. The result can be seen in Figure 4.10.

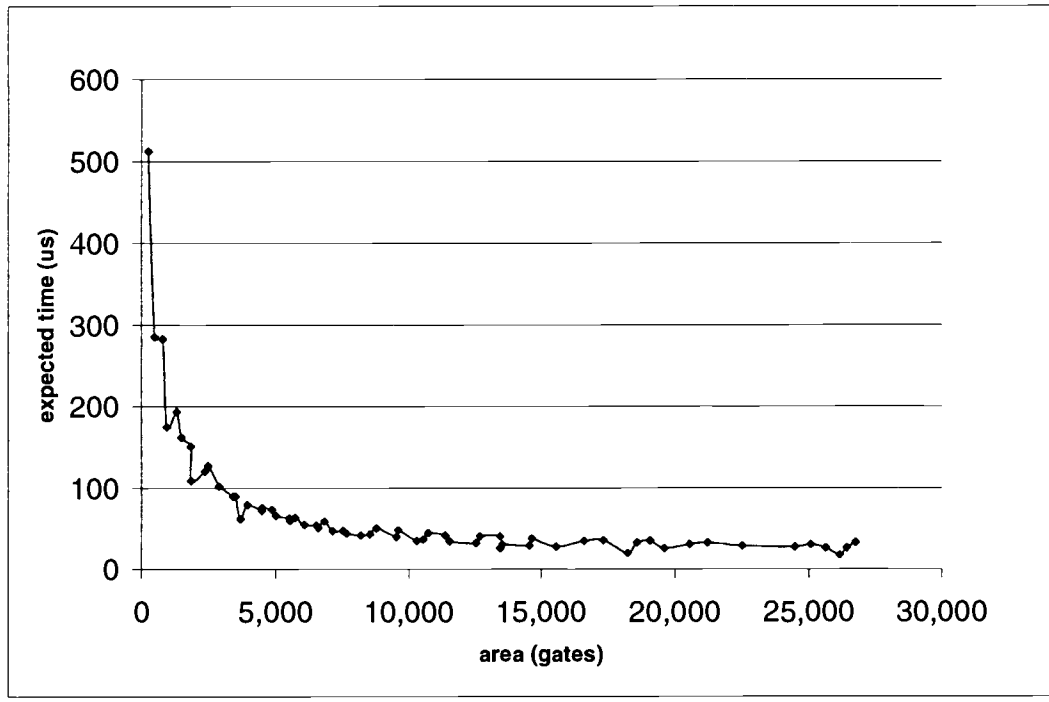


Figure 4.10. Resorted area to expected time comparison at multiple values of w and N

At this stage, again scaling can be performed. This implementation is similar to the implementation in Section 4.2.7 with the exception that the possible designs are not categorized according to their w. After scaling, the best design can be determined.

4.3. Kernel Design Tool

A software application was developed in C to implement the discussed methods and find the reasonable configurations for the kernel given the design constraints. This software application is called *kernel design tool (KDT)* from now on.

4.3.1. KDT input parameters

In the KDT implementation, there are certain parameters that are considered to be fixed information. Some information is fixed inside the KDT code to remove the necessity of inputting them every time. This fixed information are:

- The w for the kernel implementation (8, 16, 32, 64, and 128)
- The RC delay information, which was gathered with other simulation tool

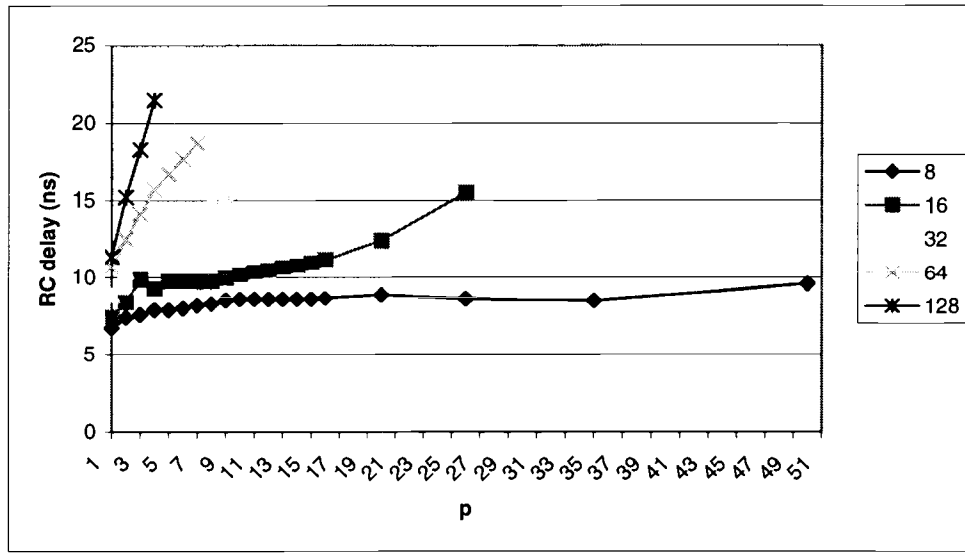


Figure 4.11. RC delay for each w by p (from flattened design results)

As indicated earlier in Section 4.1, the RC delay information is gathered from Leonardo software synthesis. This represents the delay for each clock cycle (maximum frequency) needed by the kernel at particular design and any larger delay (lower frequency) can be used. The RC delay data available is limited since only data until area up to approximately 26,500 gates are available for each w. Further, this information does not cover all existing values of p and w. Thus, interpolation method needs to be implemented to estimate the RC delay values for the configurations that are not available.

As mentioned before, the simulation to acquire RC delay was using flattened design approach. When w is constant and p increases, the number of gates used will increase. This also means higher fan out requirements. Both contributes to the increase of the RC delay in flatten design synthesis. When w increases and p is constant, the similar case as above also occurs. If each processing element is treated as a block and flattened design is not used, only the increase of w would increase the RC delay. However, since flattened design is used, this is not the case. The RC delay increases for each w and p as shown in Figure 4.11.

Instead of keeping all the RC delay of each kernel configuration, which is not available, RC delays of some configurations are collected. At the beginning of the section, it was mentioned that there are only the values of 8, 16, 32, 64, and 128 used as w. The data points were for p values of 1, 3, 9, 33, 65, 129, etc. The last data point of each w should be at the p value, where the kernel's area has not exceeded the maximum area limit or the RC delay available. RC delay can be estimated for any p and w until either case is reached.

In Cartesian coordinate system, one way to find a point in between two points is by *linear interpolation*. The RC delay between two others of consecutive p with the same w can be found through the same method with the following formula:

$$d3 = d1 + \left(\frac{d2 - d1}{p2 - p1}\right)p3 \quad \text{for any given p3 while } \; p1 < p3 < p2$$

The graphical implementation of the formula can be seen in Figure 4.12.

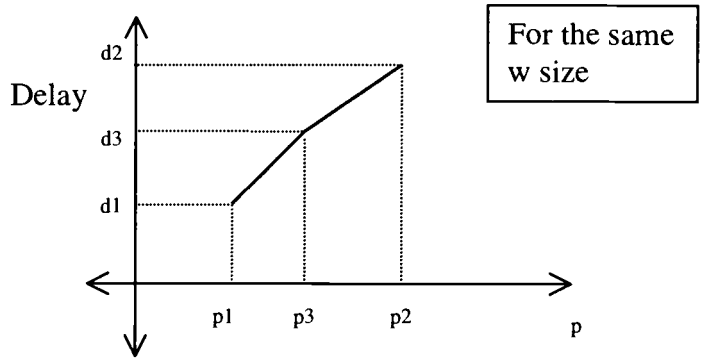


Figure 4.12. RC delay extraction by using the linear interpolation method

As discussed earlier, there are certain inputs that the designer should provide to the KDT in order to find a reasonable design. The designer knows how the kernel will be used, and thus, can provide this information. The result quality of the KDT will be determined on how much the designer knows regarding its kernel use. The necessary inputs are:

- The maximum area limit for the kernel, which at present the maximum equals to 26,500 gates due to the available RC delay data. This is easy to change if further data is available in the future;
- The values of N that will be used and their usage distribution;
- The area to time importance ratio;
- The core clock frequency, if any, that the designer would like the kernel to perform at. This additional information is used in case the designer knows that the kernel needs to perform at a certain clock frequency. This is necessary when the kernel shares the same clock with other components in an embedded system implementation;

There are two types of usage distribution methods available to the designer:

- Even distribution, when all values of N are used at about the same ratio. The *sample range* here is useful to determine how many N values should be taken into

account. In this case, the designer has no information regarding the N usage distribution. It will assume the N values are equally distributed to the sample size. The higher the sample size, the closer to real optimal the result will be.

- Discrete distribution, when the designer knows exactly the N values used and their probability of use. The designer has to enter each N value and provide the information regarding the usage of each N. This method would yield a better result than the even distribution method since it is more precise.

4.3.2. KDT Implementation

In programming the KDT application, a conceptual design is developed as a guide. The conceptual design of the KDT can be seen in Figure 4.13.

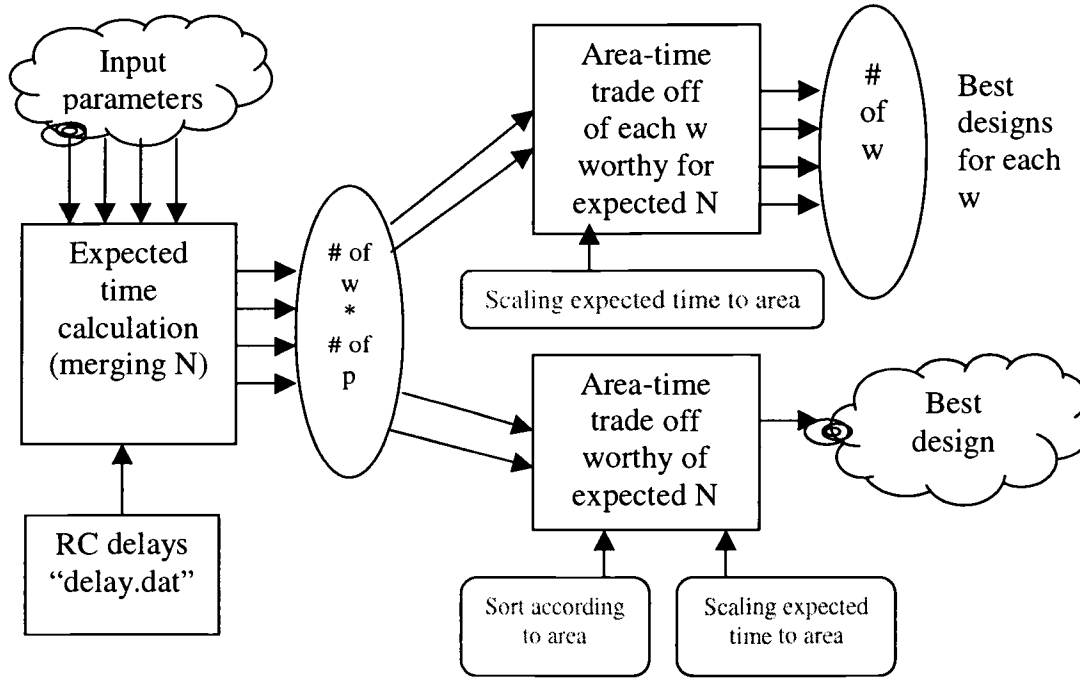


Figure 4.13. Conceptual design for KDT implementation

The KDT starts by loading the delay data value from RC delay extracted as discussed in the previous section. The extracted RC delay data for some kernel configurations are placed inside a text file called 'delay.dat', which contains information about w, p, and delay (c) separated by a single space. If the RC delays information is changed or expanded, this delay file can be updated or replaced.

The KDT computes all the possible designs' area and computation time for each N value provided as inputs, applying the RC delay from linear interpolation by using the data from delay file. The next step is to calculate the expected time by merging all computation time values for each N of each design configuration using their usage distribution. After these steps, the data will be similar to the data available for Figure 4.9. From this point on, the KDT will perform the methods previously discussed in previous sections.

Using the data, the KDT splits two ways at this point. The upper part of Figure 4.13 will assume the designer is uncertain regarding the w to use and would like to find the better design for each w. The KDT tries to find the best p for each w in order to make the kernel perform well at all used N sizes. Before the KDT can find these p values for each w, it has to perform expected time to area scaling first. After this is being performed, trade off analysis can be executed under each w. Thus, the better p for each w can be found. This better p for each w is called the *optimal designs for each w* to perform at certain used N.

The lower part of Figure 4.13 will assume the designer is uncertain about the w to use and would like to find the better design despite the w. The KDT tries to find the better design amongst all available p and w that will perform well at all used N. As prescribed earlier, sorting the possible designs by their area is first performed. The table is reconstructed after the sorting and if represented in figure, it will look like Figure 4.10. Following the same construct, expected time is scaled to area to allow trade off analysis. After trade off analysis is performed, a better design for all used N will come out as the *optimal design* to perform at certain used N.

4.4. Optimization Results for Particular Case

This section provides optimized designs for two particular N sizes of 160 and 1024 bits. In this case, the designer would like the best design considering all possible w and p. The execution time of each optimized design at its precision is also calculated. This execution time result is then compared to that at the other precision. Slowdowns are then calculated from these results. Slowdown is used since it is likely that an optimized design for specific N will not perform as well on different N. However, as previously indicated, the more p, the better the execution time would be.

Equalized design is also provided as an alternative to optimizing the model for one particular precision only. The equalized model assumes that each precision is used equally. Overall, this equalized model should perform better than the models optimized only for particular precision.

4.4.1. Analysis for kernel design for N = 160 bits and max area of 26,500 gates

Figure 4.14 describes all the possible designs given that the kernel is designed to obtain the best performance for N = 160 bits. These data used in the figure is gathered after sorting the possible designs according to their area, but before converting expected time to area. It is easy to see that not all the designs are good to implement.

The proceedings are the KDT results with different ratio of time importance. It also shows how the kernel performance at different precision. The column labeled "time using 160 bit optimized design" indicates the computation time of the kernel at precision N using the optimized kernel configuration for N = 160 bits. The "time using N bit optimized design" indicates the computation time of the kernel at precision N using the optimal design for N itself. Since a design optimized for 160 bits is not

going to perform well at N = 1024 compared to another designed optimized for 1024 bits, the slowdown can be seen in the following tables.

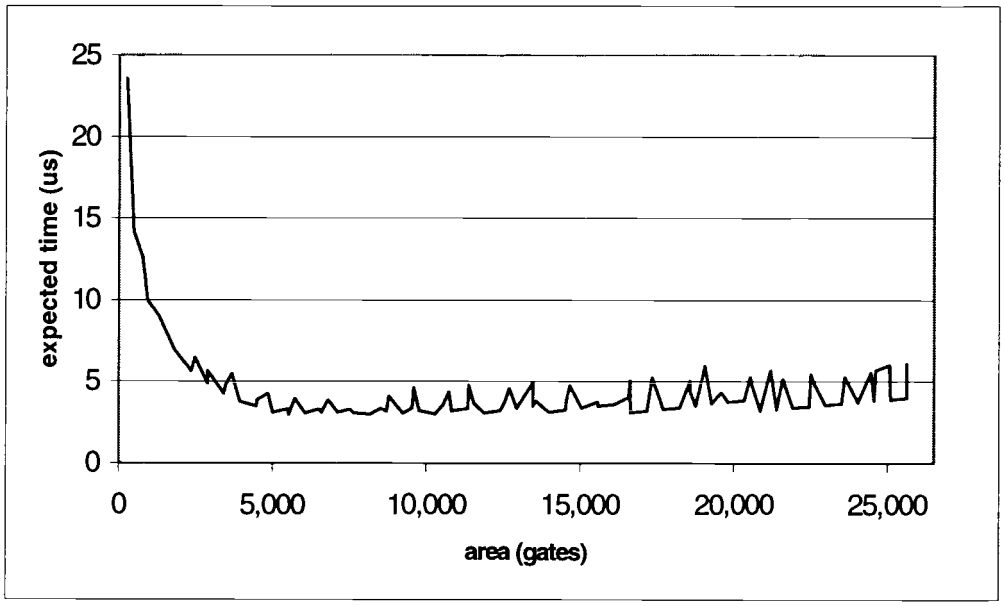


Figure 4.14. Kernel designs for N = 160 bits up to 26,500 gates

Time Importance = 100% result: w = 8, p = 16, area consumed = 8,175 gates
Max frequency = 114 MHz
Running at different precisions:

N	Time using 160 bit optimized design	Time using N bit optimized design	Slowdown
160	2.979	2.979	0%
1024	73.167	27.271	63%

Time Importance = 95% result: w = 8, p = 11, area consumed = 5,532 gates

Max frequency = 116 MHz

Running at different precisions:

N	Time using 160 bit optimized design	Time using N bit optimized design	Slowdown
160	3.001	3.001	0%
1024	104.958	38.737	63%

Time Importance = 90% result: w = 8, p = 11, area consumed = 5,532 gates

Max frequency = 116 MHz

Running at different precisions:

N	Time using 160 bit optimized design	Time using N bit optimized design	Slowdown
160	3.001	3.001	0%
1024	104.958	48.402	54%

4.4.2. Analysis for kernel design for N = 1024 bits and max area of 26,500 gates

The following Figure 4.15 describes all the possible designs given that the kernel is designed specifically for 1024 bits. Similar to the previous section, this data is gathered after sorting the possible designs according to their area, but before converting expected time to area. Here however, it is not as easy to see that not all the designs are good to implement.

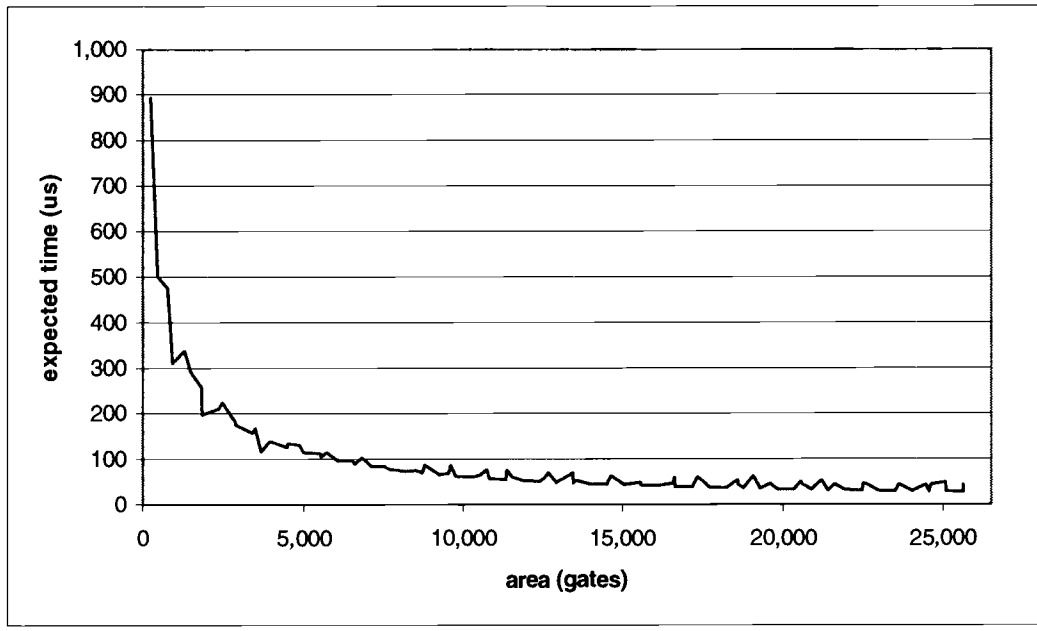


Figure 4.15. Kernel designs for N = 1024 bits up to 26,500 gates

The KDT results of different time importance and the kernel configurations result comparison can be seen as follows:

Time Importance = 100% best design: w = 8 and p = 49

Area consumed = 25,620 gates and max frequency = 103 MHz

Running at different precisions:

N	Time using 1024 bit optimized design	Time using N bit optimized design	Slowdown
160	3.976	2.979	25%
1024	27.271	27.271	0%

Time Importance = 95% best design: w = 8 and p = 32

Area consumed = 16,633 gates and max frequency = 108 MHz

Running at different precisions:

N	Time using 1024 bit optimized design	Time using N bit optimized design	Slowdown
160	3.119	3.001	4%
1024	38.737	38.737	0%

Time Importance = 90% best design: w = 8 and p = 25

Area consumed = 12,933 gates and max frequency = 111 MHz

Running at different precisions:

N	Time using 1024 bit optimized design	Time using N bit optimized design	Slowdown
160	3.330	3.001	10%
1024	48.402	48.402	0%

4.4.3. Analysis for equalized optimal design for max area of 26,500 gates

This equalized method takes into consideration the usage distribution of the related precisions (160, 256, 512, 1024, and 2048 bits). In this case, equalized distribution is assumed where each N is used 20% of the time.

The Figure 4.16 describes all the possible designs given that the kernel is designed specifically for the five mentioned N. The expected time plays a large role in determining which design would come out to be the optimal result. Further, given the combined effect of N on each particular possible kernel configuration, it is much harder to see which design would come out to be the better design.

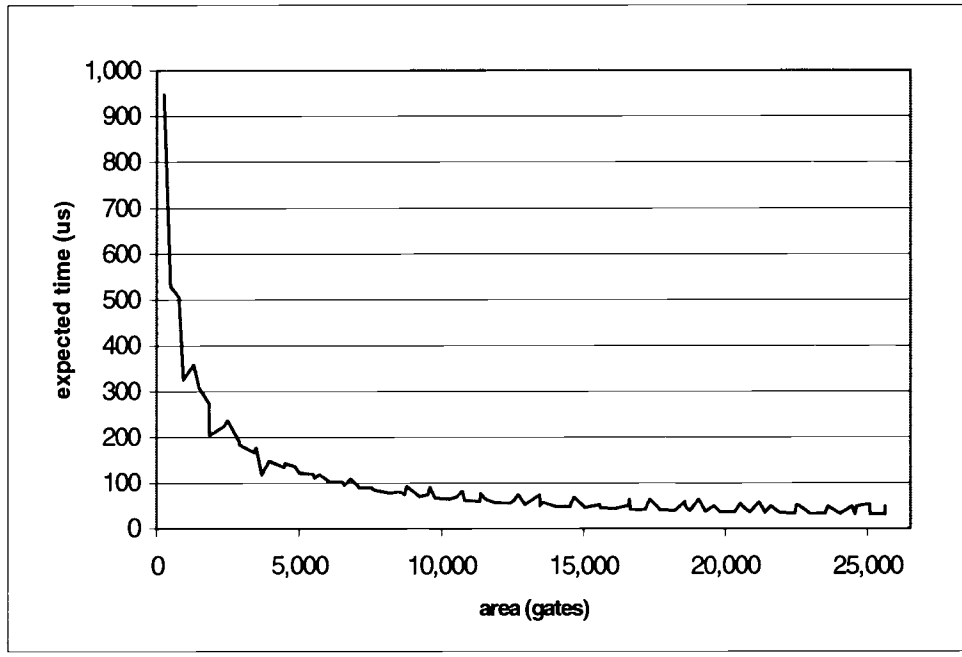


Figure 4.16. Equalized designs up to 26,500 gates

Time Importance = 100% best design: w = 8 and p = 49

Area consumed = 25,620 gates and max frequency = 103 MHz

Running at different precisions:

N	Time using equalized optimized design	Time using n bit optimized design	Slowdown
160	3.976	2.979	25%
256	5.983	4.899	18%
512	11.020	10.244	7%
1024	27.271	27.271	0%
2048	105.494	105.494	0%

Time Importance = 95% best design: w = 8 and p = 32

Area consumed = 16,633 gates and max frequency =108 MHz

Running at different precisions:

N	Time using equalized optimized design	Time using N bit optimized design	Slowdown
160	3.119	3.001	4%
256	4.991	5.030	-1%
512	10.258	10.258	0%
1024	38.737	38.737	0%
2048	152.066	152.066	0%

Time Importance = 90% result: w = 8 and p = 25

Area consumed = 12,933 gates and max frequency = 111 MHz

Running at different precisions:

N	Time using equalized optimized design	Time using N bit optimized design	Slowdown
160	3.330	3.001	10%
256	5.238	5.030	4%
512	12.906	11.819	8%
1024	48.402	48.402	0%
2048	190.836	190.836	0%

5. EXPERIMENTAL RESULTS AND ANALYSIS

5.1. Total Area Approximation for the Proposed System Implementation

In calculating the total area, each block's area of the proposed system's internals is first determined. Straightforward approximation is used for common implementation such as MUXes and FIFOs. The MM module's area can be determined using the formula in Section 4.1. The op control logic's and counters' areas are determined from synthesis using Leonardo software. If d and w are of the same width, the result registers are not needed since there is no data reformatting required.

The proposed system's blocks and area consumption are as follows:

- Memory
    - 9 full size FIFOs for X, Y0, Y1, M, Z, SS0, SC0, SS1, and SC1 registers
      Total area for each N-bit FIFO = 6N

  Total area for memory = 54N
- Control and data reformatting
    - 4 d-bit size FIFOs for control, status, result0, and result1 registers
      Total area for each d-bit FIFO = 6d
    - MUXes
        - 1 4-to-1 d-bit MUX = 8d
        - 3 2-to-1 d-bit MUXes = 4d each
        - 1 2-to-1 w-bit MUX = 4w
        - 1 3-to-1 w-bit MUX = 7w
    - 2 funnels
        - 1 w-bit FIFOs = 6w
        - Shift control = w

- 1 e-bit counter ($e = \sqrt[2]{w}$) = r, which is acquired from synthesis.
- 1 f-bit counter ($f = \sqrt[2]{N}$) = t, where N = 8192 (acquired from synthesis)

Total funnel area = 7w + (r + t)

o Op control logic block

Total area = 350 gates, which is fixed, acquired from synthesis.

If $d \neq w$, then total area for control and data reformatting= 44d + 24w + 2(r + t) + 350

If d = w, then total area for control and data reformatting= 32d + 24w + 2(r + t) + 350

- 2 MM modules

Total area for MM modules = 2A, where A is acquired from the area formula in Section 4.1 after p and w is selected by the KDT.

For examples, the d-bit IO bus is set at 32-bit following current common computer bus size. The area for MM modules can be substituted using the area formula in Section 4.1 after p and w is known from the KDT. As an example, Table 5.1 shows the area consumption of several selected MM module configurations.

w	p	Control & data reformatting	MM module (each)	Total area
8	8	2,334	3,946	10,226
8	16	2,334	8,175	18,684
8	25	2,334	12,932	28,198
8	32	2,334	16,632	35,598
16	8	2,550	7,515	17,580
16	16	2,550	15,562	33,674
32	8	2,574	14,655	31,884

Table 5.1. Area consumption of several configurations with d = 32 bits

5.2. Experimental Result and Analysis of the Proposed System Operating Montgomery Multiplication (MM)

For testing purposes, certain initial setup is determined for the system. The d-bit IO bus is set at 32-bit. For the first experiment series, the MM module is set to use w = 8 and p = 8. The N used in the preceding experiment is of 96, 256, and 1024 bits wide. For the second experiment series, the MM module is set to use w = 8 and p = 16. The same series of N are used for this experiment as well. These actual experimental results are compared to the calculated result from the time formula in Section 4.1. Time is represented in terms of clock cycles instead of actual seconds since the RC delays may vary for different implementation. The results can be seen in Table 5.2.

Initial setup			Actual time (clock cycles)				Calculated time (clock cycles)
w	p	N	Control delay	MM execution	Control and data reformatting	Total	MM execution
8	8	96	2	266	23	291	203
8	8	256	2	1,078	53	1,133	1,070
8	8	512	2	4,182	101	4,285	4,174
8	8	1024	2	16,534	197	16,733	16,526
8	16	96	2	236	23	261	203
8	16	256	2	626	53	681	558
8	16	512	2	2,118	101	2,221	2,110
8	16	1024	2	8,294	197	8,493	8,286

Table 5.2. Timing result of the system performing MM operation

The *control delay* time is always fixed at 2 cycles. These 2 cycles are used to decode operation op and issue start op signal.

The *control and data reformatting* time consists of a fixed 5 cycles for control and a variable number of cycles for data reformatting. The variable number of cycles depends on the size of the N, d, and w. In both proposed systems case where d = 32, w = 8, and N = 96, data need to be reformatted from 8 bits to 32 bits first, which will take 4 cycles each. After reformatting, the composed data can be picked up and cleared, which takes 2 cycles. A total of 96 bits data means this have to be done three times. The total time for control and data reformatting (T_d) is:

$$T_d = 5 + \frac{N}{d}\left[\frac{d}{w} + 2\right]$$

The MM execution time is quite different between the actual and calculated results. The actual MM execution time includes 1 cycle for start signal and 1 cycle for done signal, which are not included in the original formula. There is also 1 extra cycle to process one extra word in the final reduction block. These add up to a fixed 3 cycles in the MM execution.

Further, the original formula in Section 4.1 is based on the ideal model of the MM, while the experimental time result is based on actual implementation. In the actual implementation, there are 5 interstate registers that delay the data by 5 clock cycles each time data goes through the processing elements pipeline. The formula should be modified (without including the fixed 3 clock cycles) as follows:

$$T = \begin{cases} k(2p+5) + e - 1 & \text{if } (e+1) \pounds (2p+5) \\ k(e+1) + (2p+5-2) & \text{otherwise} \end{cases}$$

where $e = \left\lceil \frac{N}{w} \right\rceil$ and $k = \left\lceil \frac{N}{p} \right\rceil$

5.3. Experimental Result and Analysis of the Proposed System Operating Modular Exponentiation

For the first test, similar experimental scenario as in Section 5.2 is implemented. The d-bit IO bus is set at 32 bits. The MM module is set to use w = 8. The p = 8 and p = 16 are used. The N used in this particular experiment is of 96, 256, 512, and 1024 bits wide. The Z operand is set at precision N. Execution time is presented in terms of clock cycle and the results can be seen in Table 5.3.

Initial setup			Actual time (cycle)					
			Control delay		MM execution	Control, data reformatting, and copy		Total
w	p	N	First MM	Intermediate & last MM	All MM	First & intermediate MM	Last MM	
8	8	96	3	2	266	23	23	27,937
8	8	256	3	2	1,078	58	53	291,324
8	8	512	3	2	4,182	114	101	2,201,075
8	8	1024	3	2	16,534	226	197	17,164,260
8	16	96	3	2	236	23	23	25,057
8	16	256	3	2	626	58	53	175,612
8	16	512	3	2	2,118	114	101	1,144,307
8	16	1024	3	2	8,294	226	197	8,726,500

Table 5.3. Timing result of the system performing modular exponentiation operation

As indicated before, modular exponentiation is performed with a series of MM operations. The timing results are separated according to *control delay*, *MM execution*, and *control, data reformatting, and copy*. It can be seen that the control delay on the

first MM is one cycle longer than that at the remaining MM. This is needed since the Z bit needs to be scanned before issuing start signals to the MM modules. The MM execution itself takes the same amount of time for all MM operations. The control, data reformatting, and copy takes the same number of cycles in any MM operation except for the last. In the last MM, the time used is the same as that when performing a single MM operation. Only in the last MM operation, the system performs data reformatting for output purposes. In the first and intermediate MM, it typically requires more time because it needs to perform clear registers and copying. The time (T_d) required for control, data reformatting, and copy at the first and intermediate MM is as follows:

$$T_d = 2 + \frac{N}{d}\left[\frac{d}{w} + 3\right]$$

The next test will compare the timing results of the two configured proposed hardware system to those of software implementations. The software implementations selected for performance comparison are Fast Machine Code [18] and Crypto++ version 4.0 [19]. In these two software implementations, performance for RSA encryption and decryption are available. RSA encryption and decryption use the basic modular exponentiation operation and therefore, can be compared directly to the system performing modular exponentiation. The main difference between RSA encryption and decryption are the key size or exponent (Z) used for modular exponentiation. In RSA decryption, the Z operand size should be close to the modulus operand size. Thus, these are the software performances to which the system performance should be compared.

The Fast Machine Code [18] uses Knuth's m-ary method with nonzero sliding-windows variation [1]. It also exploits the use of Chinese Remainder Theorem (CRT) [1], where only two half-size Z and M operands calculations are required. The original results are obtained for Fast Machine Code implemented on Pentium 60MHz system. The Crypto++ version 4.0 [19] also exploits the usage of CRT to speed up the computation. Its' original results are obtained from its implementation on Pentium

850MHz system. Both software implementations' results are interpolated to obtain the timing results if Pentium 100MHz system is used.

RSA modulus	Decryption time			
	Fast Machine Code	Crypto++ version 4.0	Proposed system (w = 8 & p = 25)	Proposed system (w = 8 & p = 32)
512	15.00 ms	16.41 ms	7.87 ms	6.58 ms
1024	87.00 ms	86.96 ms	57.07 ms	45.29 ms
2048	624.00 ms	545.11 ms	442.00 ms	347.55 ms

Table 5.4. Decryption time result of software implementations and the two systems

RSA modulus	Fast Machine Code		Crypto++ version 4.0	
	Proposed system (w = 8 & p = 25)	Proposed system (w = 8 & p = 32)	Proposed system (w = 8 & p = 25)	Proposed system (w = 8 & p = 32)
512	90.60%	127.96%	108.51%	149.39%
1024	52.44%	92.10%	52.37%	92.01%
2048	41.18%	79.54%	23.33%	56.84%

Table 5.5. Speedup result of the two systems compared to software implementations

The proposed system is set at two different configurations of p and w with d = 32 for comparison. For the first configuration, the system is set at w = 8 and p = 25. For the second configuration, the system is set at w = 8 and p = 32. These two best configurations are used following the KDT result in Section 4.4.3, where the MM module was optimized for equal usage of N for the values 160, 256, 512, 1024, and 2048 with 90% and 95% time importance. Both systems are set to perform at a clock frequency of 100MHz so they can be fairly compared with the software

implementation on Pentium 100MHz system. It should be noted that the two configured systems perform straight exponentiation (without using CRT) at full precision modulus and exponent operands. The time results comparison between software implementations and the proposed systems can be seen in Table 5.4. The speedup percentages from software implementations can be seen in Table 5.5.

From either table, it can be seen that both proposed hardware configurations perform better than software implementation. The first system with w = 8 and p = 25 requires an area of approximately 28,200 gates (not including area for memory). The second system with w = 8 and p = 32 requires an area of approximately 35,600 gates (not including area for memory). As expected, the second system yields better timing results, but consumes more area, when compared to the first.

5.4. System Performance Comparison to Other Modular Exponentiation Hardware Systems

Blum and Paar in [13] have implemented a Montgomery modular exponentiation system on Xilinx XC4000 FPGA board. The Montgomery multiplier implemented is a fixed-precision solution. FPGA is selected since it can be easily reconfigured for different precision. Thus, their design area increases correspondingly with the precision used. An FPGA implementation cannot really be compared with an ASIC implementation. However, since their work describes the algorithms used in the implementation, it provides some ideas regarding a fixed precision design implementation, which requires reconfiguration when different precisions are used. These results are compared with the results of the proposed system.

The MM algorithm used in [13] is similar to that proposed by Montgomery in [6], but slightly modified for hardware implementation. It is designed as a systolic array of processing element each taking u-bit of Y and M operands. The X operand is taken 1-bit each time. All these operands are represented in redundant form. The exponentiation algorithm used is the binary method as discussed in Section 1.2. For

different precision, this FPGA implementation has to be reconfigured. The larger precision used means more area consumed. The area in FPGA is given in terms of Common Logic Blocks (CLBs) instead of gates as in ASIC implementation.

Two configurations of the Blum and Paar design are selected with u = 4 and 16. The frequencies used by each design vary depending on the u and the modulus size. Total execution time at each configuration best frequency is used for comparison.

	Blum and Paar FPGA design (u = 4)		Blum and Paar FPGA design (u = 16)		Proposed system (w = 8 & p = 32)		Proposed system (w = 8 & p = 25)	
Modulus and exponent size	Area (CLBs)	Time (ms)	Area (CLBs)	Time (ms)	Area (gates)	Time (ms)	Area (gates)	Time (ms)
512	2,555	9.38	2,001	11.56	31.9k	6.58	28.2k	7.87
768	3,745	22.71	2,946	25.68	31.9k	19.74	28.2k	24.84
1024	4,865	40.50	3,786	49.78	31.9k	45.29	28.2k	57.07

Table 5.6. Comparison to Blum and Paar's FPGA implementation

The proposed system is set at two different configurations of p and w with d = 32 for comparison. For the first configuration, the system is set at w = 8 and p = 32. For the second configuration, the system is set at w = 8 and p = 25. These two configurations are used following the optimization result in Section 4.4.3, where the MM module is optimized for equal usage at N of 160, 256, 512, 1024, and 2048 with 95% and 90% time importance. Both proposed systems are set to perform at a frequency of 100MHz. The comparison results can be seen in Table 5.6.

As indicated earlier, it is difficult to compare an FPGA and ASIC implementation. However, the timing results suggest that the proposed ASIC implementation can perform as well as the FPGA implementation by Blum and Paar.

Whereas the ASIC implementation cannot be reconfigured, this proposed word size solution design allows the system to work on any precision so long as the precision does not exceed certain limit size. The Blum and Paar design has its own difficulty where it requires increasing large number of CLB when the precision used is large. Further, it has to be reconfigured each time larger precision is used.

There are several other hardware solutions in performing modular exponentiation available in the market. Most of these hardware solutions typically can perform other modular arithmetic operations (add, sub, multiply) as well. The proposed system has the final reduction block that can actually perform modular addition and modular subtraction; however, it is not intended to run such operations separately at current time. The main objective in this section is to compare modular exponentiation execution time and area consumption of the existing hardware solutions to those of the proposed system.

	Proposed system (w=8 & p=32)	Others		
		SIDSA RSAC2048A	SCI Worx RSA DesignObject™	HIFN 6500
Key or exponent (bits)	Up to 8192	Up to 8196	Up to 8192	1024 – 2048
Area without memory (gates)	35.6k	18k	15.5k	-
Clock (MHz)	100	60	150	-
Tech (µm)	0.5	0.35	0.18	0.18
Time result for 1024-bit modulus	45.29 ms	E – 235 us D – 60 ms	18 ms	5 ms

Table 5.7. Comparison of different exponentiation hardware systems

Three other systems are selected for comparison purposes, namely: SIDSA RSAC2048A [20], SCI Worx RSA DesignObject™ [21], and Hifn 6500 [22]. Information about these systems was gathered from the products' information. Table 5.7 shows the comparison. Further explanation regarding each system follows.

For comparison, the proposed system is configured with d = 32, w = 8, and p = 32. This configuration is used again since it provides the best result in Section 5.3. The clock frequency is set at 100MHz with IC technology of 0.5µm. This system performs straight modular exponentiation (without using CRT). The time result when performing modular exponentiation of 1024 bits on both modulus and exponent operands is 45.29ms.

The closest comparable system is RSAC2048A [20] developed by SIDSA. It uses MM algorithm, but it is not disclosed whether a modified version is used. The exponentiation algorithm used is also undisclosed. The clock frequency of 60MHz is used with IC technology of 0.35µm. It also performs straight modular exponentiation. The data provides two separate results for RSA 1024-bit encryption and decryption. As indicated earlier in Section 5.3, RSA decryption uses modular exponentiation with close to full size (1024-bit) operands and, therefore, takes longer than RSA encryption. This decryption time result is the one that should be compared with the proposed system result since it is unclear the Z size used for the RSA encryption. This system requires 60 ms to perform decryption operation, which is slower than the proposed system. However, it does consume less area than the proposed system.

Another comparable system is RSA ObjectDesign™ [21] developed by SCI Worx. The algorithms used in this system are not disclosed. In the product information, there are 5 available designs to choose from. For comparison purposes, a design with an internal datawidth of 32-bit (with smallest area) is chosen since it performs at a clock frequency closest to that of the proposed system. The clock frequency is 150MHz and the IC technology used is 0.18µs. The execution time presented in the table shows the proposed systems performing straight exponentiation with both modulus and exponent operands at 1024-bit. The exponent operand is

selected to have 50% bit "1" and 50% bit "0" without clarification on how they are distributed. The time result is 18 ms, which is much faster than the proposed system, but it is also unclear if this is an average execution time result. Looking at the data just as is, this system performs faster and consumes less area than the proposed system.

The last comparable system is Hifn 6500 [22]. The algorithms used in this system are not disclosed as well as its' clock frequency and area consumption. It performs RSA decryption using CRT. It uses 0.18μm IC technology and claims to be able to perform 200 RSA 1024-bit private key computations per second. This means that it perform RSA 1024-bit decryption in 5 ms, which is much faster than any system discussed in this section. The closest time result to this system is that of SCI Worx with 64-bit internal datawidth of 64-bit (with fastest execution) running at >200MHz that takes less than 4 ms to perform, which were not mentioned here. Due to the limited information regarding this particular system, it is difficult to conclude its comparison result to the proposed system. Based on the data available, this system uses CRT method to speed up the execution time and outperforms the proposed system by almost ten times.

It is difficult to compare these different systems straightforwardly due to several reasons. First, most of the algorithms used by the other systems are not disclosed. Second, the clock frequency and IC technology used varies. Finally, the execution time results are obtained differently. Therefore, it cannot be concluded which of these systems is the best.

6. CONCLUSIONS AND FUTURE WORK

6.1. Conclusions

This work presented a modular exponentiation ASIC implementation using the radix-2 scalable Montgomery Multiplier by Tenca and Todorov. The parallel binary method by Chiou is used as the exponentiation algorithm. Since the Montgomery Multiplier is flexible enough to perform in any precision, this flexibility is inherited by the proposed modular exponentiation system.

The radix-2 scalable Montgomery Multiplier is optimized for the precisions that the system will be used. A software application is specifically developed to find the best design for the scalable Montgomery Multiplier configuration. The best configuration will vary depending the designer specification and the precisions where it will perform.

The proposed system performance is compared with several selected software implementations. It is performing faster than the software implementations running at the same frequency. The timing results of the proposed system are comparable to the FPGA implementation by Blum and Paar. Although this proposed system is not reconfigurable, it can perform at any precision without modification. This is true as long as the maximum precision does not exceed the memory size. The proposed system is also compared with several other modular exponentiation hardware systems that are available on the market. However, it is very difficult to draw any solid conclusion from the limited information available about them.

6.2. Future Work

Finding the best configuration for the MM module implementation is complex and the result varies significantly depending on the designer criteria. The KDT application introduced in this work is designed to find the best kernel configuration in the MM module for MM operation, which is the basic operation for modular exponentiation. In order to determine the best system design for modular exponentiation operation, the KDT may be improved to include the details area requirements and extra time of the proposed system implementation when performing modular exponentiation. For example, if the d and w are of the same size, data reformatting and result registers will not be necessary. Nevertheless, the design without data reformatting may not be better than the design with it in terms of overall execution time and area consumption. The improved KDT can be developed from experimental result described in Section 5.1, 5.2, and 5.3.

The main focus of this work is to get a reasonably fast modular exponentiation system using the MM hardware. The approach taken in this work consists in analyzing the available exponentiation algorithm and optimizing the MM module particularly used in the system. There are higher radices scalable MM hardware designs proposed in [10] that can also be used. The higher radices MM design can be used by the parallel binary method algorithm and may yield faster execution time.

Improvement over the MM module design itself may improve the overall system execution time in performing modular exponentiation. The MM module was designed for single MM operation each time. In case of modular exponentiation operation, which includes series of MM operation, elimination of the final reduction block maybe possible. Data in carry save form originally located in SS and SC prior to final reduction can be fed back to the system. This requires extra input on the carry save adder and some other modifications. However, the elimination of final reduction block may improve the RC delay of the MM module and reduce total number of clock cycles.

Another potential improvement is by utilizing CRT [1] in the proposed hardware system implementation for certain cryptographic algorithms. This will reduce the execution time significantly. There is no modification required for the proposed system to use CRT. The only thing that needs to be done is to rearrange the input data for the system accordingly with the right operations to perform in the correct sequences.

Further analysis of how the modular exponentiations are used in most cryptographic algorithms may produce other approaches to achieve even faster execution time. For example in RSA implementation, RSA public keys typically only contain a small number of bits while its private keys contain near the full size (N) number of bits. This means that RSA encryption, which uses the small size public key, has small size exponent. Currently, the proposed system handles this in the number of words fashion. Thus, if the last most significant 8-bit word consists of a single 1 in the LSB followed by 0s, the proposed system will waste seven MM operations. By adding a "last 1 bit detector" for the exponent Z operand, the system does not need to perform unnecessary squaring in anticipation of other 1 bit in the Z operand. This detector can simply perform a simple scanning of the Z bit prior to starting the modular exponentiation operation. If the Z is scanned one bit each cycle, it will take N cycles to scan the whole Z operand. However, this is still a small price to pay considering that it takes more than N cycles even for a single MM operation to finish.

Along the line of analyzing how the modular exponentiations are used in cryptographic applications, sometimes in RSA, Diffie-Helman and El Gamal algorithms, the precision size of modulus (M) operand may not be the same with that of exponent (Z) operand. It is likely that the Z operand size is smaller than the M operand size. In current system implementation, these precisions are assumed to be the same since it would be the most secure implementation. Having separate operands' size for both M and Z can save significantly in terms of modular exponentiation execution time. Even if the "last 1 bit detector" is already implemented in the system, this Z operand size information can make the detection finish faster.

The system proposed in this thesis provides a basic underlying implementation and can be enhanced to incorporate most of the future work discussed in this section.

BIBLIOGRAPHY

1. C.K.Koc, "High-Speed RSA Implementation," *RSA Laboratories*, version 2.0, November 1994.

2. M.E. Hellman, W. Diffie, "New Directions on Cryptography," *IEEE transactions on Information Theory*, vol. 22, pp. 644-654, November 1976.

3. L. Adelman, R.L. Rivest, A. Shamir, "A Method for Obtaining Digital Signature and Public-Key Cryptosystems," *Comm. ACM*, vol. 21, no. 2, pp. 120-126, February 1978.

4. National Institute for Standard and Technology, "Digital Signature Standard (DSS)," Tech. Rep., FIPS PUB 186-2, January 2000.

5. N. Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203-209, January 1987.

6. P.L. Montgomery, "Modular Multiplication Without Trial Division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519-521, April 1985.

7. B.S. Kaliski, C.K. Koc, T. Acar, "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26-33, June 1996.

8. A.F. Tenca, C.K. Koc, "A Scalable Architecture for Montgomery Multiplication," in *Cryptographic Hardware and Embedded Systems*, Ed. 1999, number 1717 in Lecture Notes in Computer Science, pp. 94-108, Springer, Berlin, Germany.

9. A.F. Tenca, C.K. Koc, E. Savas, "A Scalable and Unified Multiplier Architecture for Finite Fields *GF(p)* and $GF(2^m)$," in *Cryptographic Hardware and Embedded Systems*, Ed. 2000, Lecture Notes in Computer Science, pp. 94-108, Springer, Berlin, Germany.

10. G. Todorov, "ASIC Design, Implementation, and Analysis of A Scalable High-Radix Montgomery Multiplier," MS thesis, Oregon State University, December 2000.

11. A.F. Tenca, G. Todorov, C.K. Koc, " High Radix Design of a Scalable Modular Multiplier," in *Cryptographic Hardware and Embedded Systems*, Ed. 2001, Lecture Notes in Computer Science, pp. 189-205, Paris, France.

12. Et al. T. Hamano, "O(n)-Depth Circuit Algorithm for Modular Exponentiation," in *IEEE 12th Symposium on Computer Arithmetic*. 1995, pp. 188-192, IEEE Computer Society Press, Los Alamitos, CA.

13. T. Blum, C. Paar, "Montgomery Modular Exponentiation on Reconfigurable Hardware," in *IEEE 14th Symposium on Computer Arithmetic*. 1999, pp. 70-77, IEEE Computer Society Press, Los Alamitos, CA.

14. C.C. Chang, D.C. Lou, "Parallel Computation of The Multi-Exponentiation for Cryptosystems", Computer Mathematics, Institute of Computer Science and Information Engineering, The Gordon and Breach Publishing Group.

15. S.C. Goldstein, R.R. Taylor, "A High-Performance Flexible Architecture for Cryptography," in *Cryptographic Hardware and Embedded Systems*, C. Paar, C.K. Koc, Ed. 1999, number 1717 in Lecture Notes in Computer Science, pp. 231-245, Springer, Berlin, Germany.

16. C.W. Chiou, "Parallel Implementation of The RSA Public-Key Cryptosystem," *International Journal of Mathematics*. 1993, pp. 153-155.

17. C.D. Walter, "Space/Time Trade-Offs for Higher Radix Modular Multiplication using Repeated Addition," *IEEE Transaction on Computers*, vol. 46, no. 2, Feb 1997.

18. M. Scott, "Fast Machine Code for Modular Multiplication," School of Computer Applications, Dublin City University. January 1995.

19. http://www.eskimo.com/~weidai/benchmarks.html, Crypto++ 4.0 Benchmark.

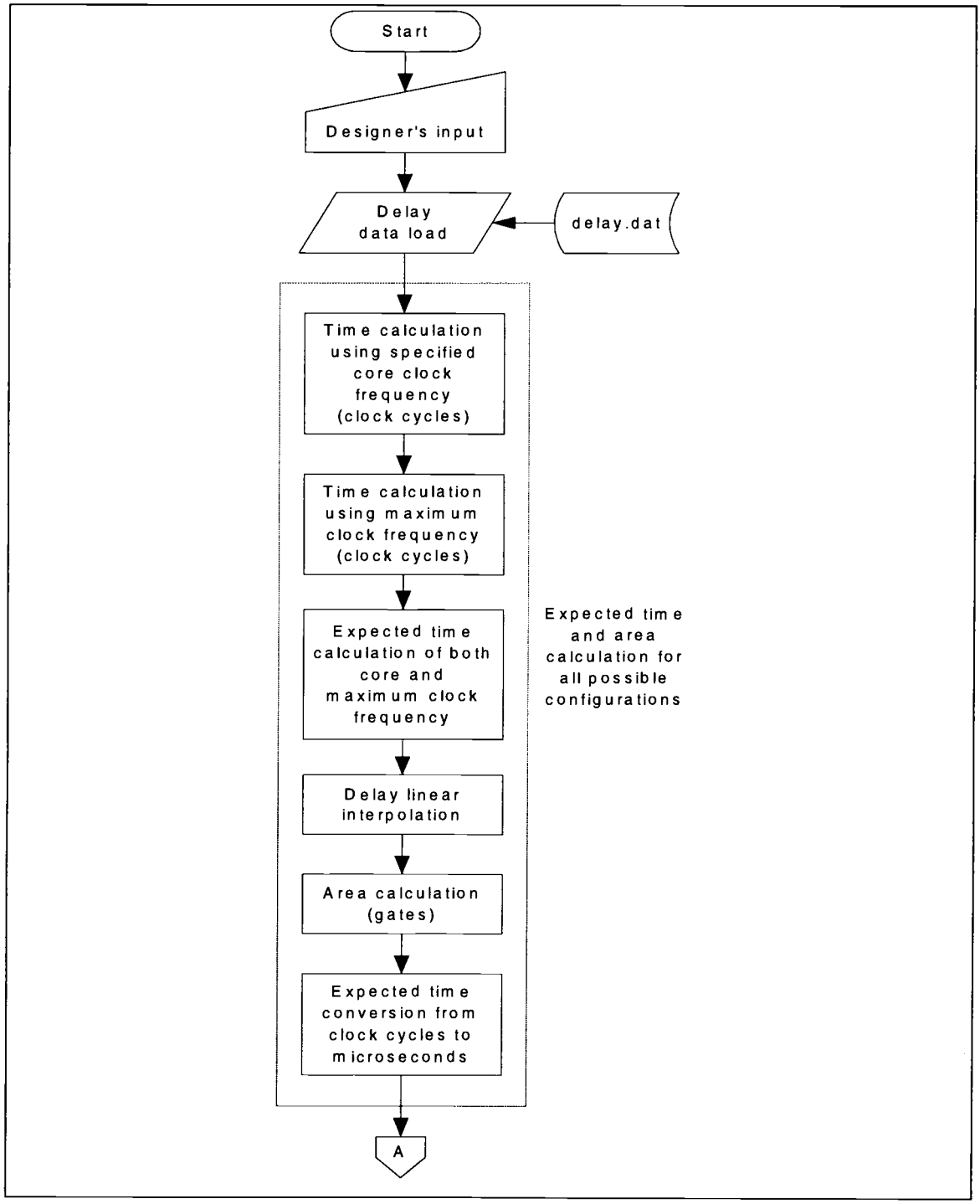
20. http://www.sidsa.com/datasheets/RSA/ds_rsa2048a_short.html, SIDSA RSA2048A RSA Coprocessor IP.

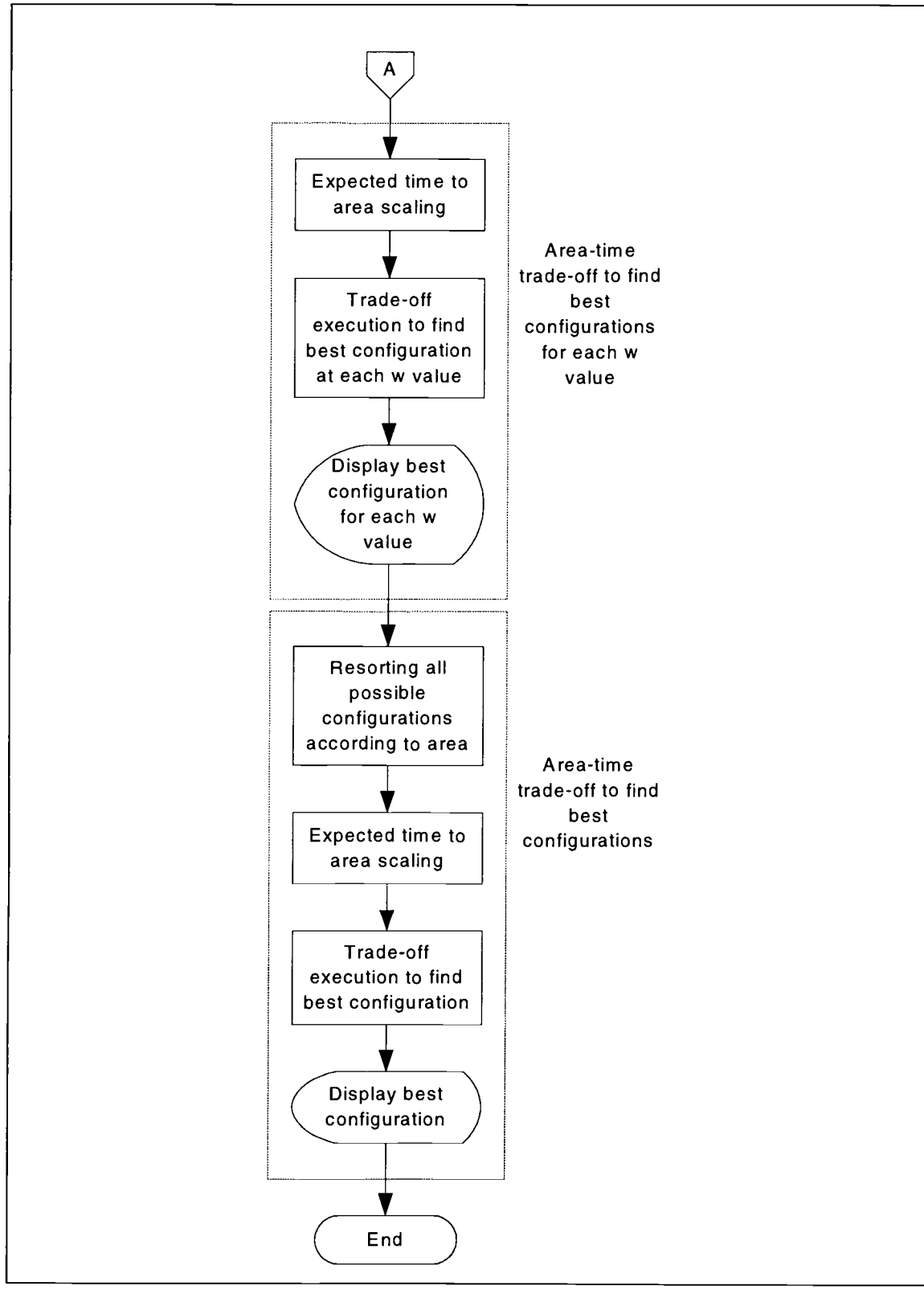
21. http://www.sci-worx.com/internet/homepage_internet_e.html, Sci-Worx RSA Design Object™.

22. http://www.hifn.com/products/6500.html, Hifn 6500 Public Key Processor.

APPENDICES

Appendix A. Kernel Design Tool Flowchart




A
Expected time to area scaling
Trade-off execution to find best configuration at each w value
Display best configuration for each w value
Area-time trade-off to find best configurations for each w value
Resorting all possible configurations according to area
Expected time to area scaling
Trade-off execution to find best configuration
Display best configuration
Area-time trade-off to find best configurations
End

Appendix B. Delay Data File Arrangement

The format of the "delay.dat" data file can be seen as follows. It also shows the actual data used for the KDT operation in this thesis work. Respectively according to the set of numbers, it represents the w, p and RC delay separated by a single space.

```
0008 0001 0006.7
0008 0003 0007.6
0008 0009 0008.5
0008 0017 0008.8
0008 0033 0009.2
0008 0065 0010.1
0016 0001 0007.4
0016 0003 0009.9
0016 0009 0010.0
0016 0017 0011.5
0016 0033 0020.0
0032 0001 0008.9
0032 0003 0011.2
0032 0009 0015.3
0032 0017 0017.0
0064 0001 0010.7
0064 0003 0014.1
0064 0009 0020.7
0128 0001 0011.4
0128 0003 0018.3
```

Appendix C. Kernel Design Tool Source Code

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* Kernel Design Tool (KDT) version 1.0     */
/*                                          */
/* 07/15/2001                               */
/* Developed by Budiyoso Kurniawan          */
/*                                          */

/* This KDT is designed to find best design configuration for  */
/* radix-2 Montgomery Multiplier kernel. The time and area     */
/* formulas used in this implementation is based on            */
/* "A Scalable Architecture for Montgomery Multiplication"     */
/*  by A.F.Tenca and C.K.Koc.                                  */

/* The w used are set at 8, 16, 32, 64, and 128.               */
/* The "delay.dat" data file containing RC delay data is       */
/* required.                                                   */

/* data array used to hold all the possible kernel configurations */
/* w = number of bits in each processing element word             */
/* p = number of processing elements                              */
/* ettime = estimated time                                        */
/* fmax = maximum clock frequency for the configuration           */
/* ettimef = estimated time at maximum clock frequency            */
struct result01 {
   int w, p;
   float area, fuse, ettime, fmax, ettimef;
    } cand[5][200];

/* data array used to hold the best kernel configurations of each w*/
struct result02 {
   int w, p;
   float area, fuse, ettime, fmax, ettimef;
    } bcand[5];

/* data array used to hold all the sorted possible kernel */
/* configurations */
struct result03 {
   int w, p;
   float area, fuse, ettime, fmax, ettimef;
    } mcand[1000];
```

```
/* data array used to hold the best kernel configuration for all w */
/* and N */
struct result04 {
   int w, p;
   float area, fuse, ettime, fmax, ettimef;
    } bdesign;
char tempd[19];


/* data array used to hold the extracted RC delay and kernel */
/* configuration from "delay.dat" file                       */
struct data01 {
      int w, p;
      float c;
      } delay[80];

struct u1 {
   int n;
   float r;
   } usage[100];

float E[100];


/* Procedural call to execute power calculation */
int power(int a, int b) {
      int i, c=1;
      if(b==0) {c=1;}
            else {for(i=1;i<=b;i++) {c = c*a;}}
      return c;
}


/* Procedural call to calculate the area given p and w      */
/* Radix-2 MM area formula is used in this implementation   */
float area(int w, int p) {
   float area;
   area =  59.65*p*w + 51.44*p - 31*w - 35.52;
   return area;
   }


/* Procedural call to calculate the time in clock ticks given p */
/* and w                                                        */
/* Radix-2 MM area formula is used in this implementation       */
int tticks(int n, int w, int p) {
   int k, e, T;
   double n1,p1,w1,e1;
   (double)n1 = (int)n;
   (double)p1 = (int)p;
   (double)w1 = (int)w;
   k = ceil(n1/p1);
   e1 = (n1+1)/w1;
```

```
   e = ceil(e1);
   if (e+1<=2*p) {T = (2*k*p)+e-1;}
      else {T = k*(e+1) + 2*(p-1);}
   return T;
   }


/* Procedural call to calculate RC delay using linear interpolation*/
/* given p and w as well as preloaded delay[x] data array from the */
/* "delay.dat" data file (using delayinit() procedure)             */
float delaycalc(int w, int p) {
   float dly;
   int l;
   for(l=0;delay[l].w<2*w;l++) {
      if (w==delay[l].w) {
        if (p>delay[l].p && p<delay[l+1].p && delay[l].w ==
delay[l+1].w) {
         dly = delay[l].c + (p-delay[l].p) * (delay[l+1].c-
delay[l].c)/(delay[l+1].p-delay[l].p);
         }
        else if (p == delay[l].p) {
         dly = delay[l].c;
         }
        }
      }
   return dly;
   }


/* Procedural call to change the time from clock ticks to  */
/* microsecond  given n (precision), w, p, and dly         */
/* (RC delay)                                              */
float ttime(int n, int w, int p, float dly) {
   float tT;
   tT = tticks(n,w,p)*dly/1000;
   return tT;
   }


/* Procedural call to calculate the expected time (evalue) for each*/
/* configuration by merging computation time at each precision n   */
/* using the usage distribution. The result is stored in cand[x][y]*/
/* data array.                                                     */
void evalue(float maxarea, int maxn, int minn, int cf, float duse,
int sample, int maxw, char dist) {
int i, j, w;
float temptime, maxftime;
int n, nn, x;
float dly;

for(i=3;i<=maxw;i++) {
   w = power(2,i);
   for(j=1; area(w,j)<=maxarea; j++) {
```

```
        temptime = 0;
        maxftime = 0;
        dly = delaycalc(w,j);
        if (cf == 0) {duse = dly;}
        if (dly <= duse) {
           if (maxn == minn) {
              x =  maxn;
              temptime = ttime(x,w,j,duse);
              maxftime = ttime(x,w,j,dly);
              } else {
              nn = 0;
              for(n=minn;n<=maxn;n=n+((maxn-minn)/(sample-1))) {
                 if (dist == 'd') {x=usage[nn].n;} else {x=n;}
                 temptime = temptime + (ttime(x,w,j,duse) *
usage[nn].r);
                 maxftime = maxftime + (ttime(x,w,j,dly) *
usage[nn].r);
                 nn++;
                 }
              }
           cand[i-3][j].ettimef = maxftime;
           cand[i-3][j].fmax = 1000/dly;
           cand[i-3][j].ettime = temptime;
           cand[i-3][j].fuse = 1000/duse;
           cand[i-3][j].area = area(w,j);
           cand[i-3][j].p = j;
           cand[i-3][j].w = w;
           } else {
           cand[i-3][j].ettimef = 1000000;
           cand[i-3][j].fmax = 1000/dly;
           cand[i-3][j].ettime = 1000000;
           cand[i-3][j].fuse = 1000/duse;
           cand[i-3][j].area = 1000000;
           cand[i-3][j].p = j;
           cand[i-3][j].w = w;
           }
        }
    }
}


/* Procedural call to execute trade off and find the best design   */
/* configuration at each w. The result is stored in bcand[x].      */
void attcsearch(float maxarea, int maxw, float areap, float timep) {
int i, j, w;
float areadiff, timediff, diff;
float maxttime, minttime, maxdarea, mindarea;
float areavalue[5];

/* scaling time to area portion                                    */
// printf("maxdarea     mindarea   maxttime    minttime\n");
for(i=3;i<=maxw;i++) {
    w = power(2,i);
```

```
    minttime = 1000000;
    maxttime = 0;
    mindarea = 1000000;
    maxdarea = 0;
    for(j=1; (area(w,j)<=maxarea && cand[i-3][j].fuse <= cand[i-
3][j].fmax); j++) {
       if (mindarea > cand[i-3][j].area) {mindarea = cand[i-
3][j].area;}
       if (maxdarea < cand[i-3][j].area) {maxdarea = cand[i-
3][j].area;}
       if (minttime > cand[i-3][j].ettime) {minttime = cand[i-
3][j].ettime;}
if (maxttime < cand[i-3][j].ettime) {maxttime = cand[i-3][j].ettime;}
       }

   if (maxdarea != mindarea) {
       areavalue[i-3] = (maxdarea-mindarea)/(maxttime-minttime);
       } else {
       areavalue[i-3] = 1;
       }
    }

/* trade-off execution portion                                        */
for(i=3;i<=maxw;i++) {
    w = power(2,i);
    bcand[i-3].ettime = cand[i-3][1].ettime;
    bcand[i-3].area = cand[i-3][1].area;
    for(j=1; (area(w,j)<=maxarea && cand[i-3][j].fuse <= cand[i-
3][j].fmax); j++) {
        if (areavalue[i-3] == 1) {
           diff = 1;
           } else {
           areadiff = (bcand[i-3].area - cand[i-3][j].area);
           timediff = (bcand[i-3].ettime - cand[i-
3][j].ettime)*areavalue[i-3];
           diff = areap*areadiff + timep*timediff;
           }
        if (diff >= 0) {
           bcand[i-3].ettimef = cand[i-3][j].ettimef;
           bcand[i-3].fmax = cand[i-3][j].fmax;
           bcand[i-3].ettime = cand[i-3][j].ettime;
           bcand[i-3].fuse = cand[i-3][j].fuse;
           bcand[i-3].area = cand[i-3][j].area;
           bcand[i-3].p = cand[i-3][j].p;
           bcand[i-3].w = cand[i-3][j].w;
           }
        }
    }
printf("\nBest Candidates:\n");
printf("   w     p   area   fuse(MHz)  ettime(ns)  fmax(MHz)
ettimef(ns)\n");
for(i=3;i<=maxw;i++) {
```

```
   printf("%4d %4d %6.3f %6.3f %6.3f %6.3f %6.3f\n", bcand[i-3].w,
bcand[i-3].p, bcand[i-3].area, bcand[i-3].fuse, bcand[i-3].ettime,

bcand[i-3].fmax, bcand[i-3].ettimef);
   }
}


/* Procedural call to perform sorting of possible design      */
/* configuration according to area increment ignoring the     */
/* different w. The results are stored in mcand[x] data       */
/* array.                                                     */
int sortdesignbyarea(float maxarea, int maxw) {
float maxdarea;
int w, td, i, j, k, l;
td = -1;
for(i=3;i<=maxw;i++) {
   w = power(2,i);
   if(i==3) {
      for(j=1; area(w,j)<=maxarea; j++) {
         if(cand[i-3][j].area != 1000000) {
             td = td + 1;
             mcand[td].ettimef = cand[i-3][j].ettimef;
             mcand[td].fmax = cand[i-3][j].fmax;
             mcand[td].ettime = cand[i-3][j].ettime;
             mcand[td].fuse = cand[i-3][j].fuse;
             mcand[td].area = cand[i-3][j].area;
             mcand[td].p = cand[i-3][j].p;
             mcand[td].w = cand[i-3][j].w;
             }
         }
      maxdarea = mcand[td].area;
      } else {
      for(j=1; area(w,j)<=maxarea; j++) {
         if(cand[i-3][j].area != 1000000) {
            if (cand[i-3][j].area<maxdarea) {
               for(k=0; k<td; k++) {
                  if(cand[i-3][j].area>mcand[k].area && cand[i-
3][j].area<mcand[k+1].area) {
mcand[k+1].area, cand[i-3][j].area);
                     td = td + 1;
                     for(l=td; l>k+1; l--) {
                        mcand[l].ettimef = mcand[l-1].ettimef;
                        mcand[l].fmax = mcand[l-1].fmax;
                        mcand[l].ettime = mcand[l-1].ettime;
                        mcand[l].fuse = mcand[l-1].fuse;
                        mcand[l].area = mcand[l-1].area;
                        mcand[l].p = mcand[l-1].p;
                        mcand[l].w = mcand[l-1].w;
                        } // shifting
                     mcand[k+1].ettimef = cand[i-3][j].ettimef;
                     mcand[k+1].fmax = cand[i-3][j].fmax;
                     mcand[k+1].ettime = cand[i-3][j].ettime;
```

```
                    mcand[k+1].fuse = cand[i-3][j].fuse;
                    mcand[k+1].area = cand[i-3][j].area;
                    mcand[k+1].p = cand[i-3][j].p;
                    mcand[k+1].w = cand[i-3][j].w;
                    } // check if in between and insert
                 } // next between
              } else {
              td = td + 1;
              mcand[td].ettimef = cand[i-3][j].ettimef;
              mcand[td].fmax = cand[i-3][j].fmax;
              mcand[td].ettime = cand[i-3][j].ettime;
              mcand[td].fuse = cand[i-3][j].fuse;
              mcand[td].area = cand[i-3][j].area;
              mcand[td].p = cand[i-3][j].p;
              mcand[td].w = cand[i-3][j].w;
              } // insert or add to the last
           } // check if 0
        }
     }
  }
return td;
}


/* Procedural call to execute trade off and find the best design  */
/* configuration for all w and N. The result is stored in bcand.  */
void btatcsearch(float areap, float timep, float maxarea, int maxw) {
int i, td;
float areadiff, timediff, diff;
float maxttime, minttime, maxdarea, mindarea;
float areavalue;

td = sortdesignbyarea(maxarea, maxw);

/* scaling time to area portion                    */
minttime = 1000000;
maxttime = 0;
mindarea = mcand[0].area;
maxdarea = mcand[td].area;
for(i=0;i<=td;i++) {
   if (minttime > mcand[i].ettime) {minttime = mcand[i].ettime;}
   if (maxttime < mcand[i].ettime) {maxttime = mcand[i].ettime;}
   }
   printf("%f %f %f %f\n", maxdarea, mindarea, maxttime, minttime);
   printf("total data = %d\n", td);
   if (maxdarea != mindarea) {
      areavalue = (maxdarea-mindarea)/(maxttime-minttime);
      } else {
      areavalue = 1;
      }
```

```
/* trade-off execution portion                          */
bdesign.ettime = mcand[0].ettime;
bdesign.area = mcand[0].area;
for(i=0;i<=td;i++) {
    if (areavalue == 1) {
       diff = 1;
       } else {
       areadiff = (bdesign.area - mcand[i].area);
       timediff = (bdesign.ettime - mcand[i].ettime)*areavalue;
       diff = areap*areadiff + timep*timediff;
       }
    if (diff >= 0) {
       bdesign.ettimef = mcand[i].ettimef;
       bdesign.fmax = mcand[i].fmax;
       bdesign.ettime = mcand[i].ettime;
       bdesign.fuse = mcand[i].fuse;
       bdesign.area = mcand[i].area;
       bdesign.p = mcand[i].p;
       bdesign.w = mcand[i].w;
       }
    }
printf("\nBest design for area-time tradeoff:\n");
printf("time importance: %2.5f   area importance:
%2.5f\n",timep,areap);
printf("word size= %4d   pipe stages= %4d   area consumed= %f\n",
bdesign.w, bdesign.p, bdesign.area);
printf("frequency use= %6.3f    max frequency=%6.3f\n", bdesign.fuse,
bdesign.fmax);
}


/* Procedural call to calculate and display MM time result */
/* at each used precision                                  */
void timeit(int maxn, int minn, int sample, char dist) {
float dly, ftime, maxftime;
int n, nn, x;
printf("  n   time-fuse  time-fmax\n");
if (maxn == minn) {
   ftime = ttime(maxn, bdesign.w, bdesign.p, 1000/bdesign.fuse);
   maxftime = ttime(maxn, bdesign.w, bdesign.p, 1000/bdesign.fmax);
   printf("%5d    %6.3f   %6.3f\n", maxn, ftime, maxftime);
   } else {
   nn = 0;
   for(n=minn;n<=maxn;n=n+(maxn-minn)/(sample-1)) {
       if (dist == 'd') {x=usage[nn].n;} else {x=n;}
       ftime = ttime(x, bdesign.w, bdesign.p, 1000/bdesign.fuse);
       maxftime = ttime(x, bdesign.w, bdesign.p, 1000/bdesign.fmax);
       printf("%5d    %6.3f   %6.3f\n", x, ftime, maxftime);
       nn++;
       }
   }
}
```

```
/* Procedural call to open the "delay.dat" data file */
/* and load the data to the delay[x] data array      */
void delayinit() {
FILE *fp;
char *filename = "delay.dat";
int i, imax;
char tmp;

if ((fp = fopen(filename, "r")) == NULL) {
      printf("File can not be opened.\n");
      exit(1);
      }

i=0;
while(!feof(fp)) {
   fgets(tempd, sizeof(tempd), fp);
   i++;
   delay[i].w = atoi(strtok(tempd, " "));
   delay[i].p = atoi(strtok(NULL, " "));
   delay[i].c = strtod(strtok(NULL, " \n\0"), NULL);
   }
fclose(fp);
}


/* The main code                                          */
main() {
int i, j, w;
int maxw = 7;
float maxarea = 26500;
int maxn = 2048;
int minn = 256;
int sample = 8;
int n, nn;
float areap = .5;
float timep = .5;
char dist;
int cf;
float fuse, duse;

/* Designer's inputs portion                              */
printf("Maximum area = "); scanf("%f", &maxarea);
printf("Maximum precision = "); scanf("%d", &maxn);
printf("Minimum precision = "); scanf("%d", &minn);
printf("Frequency used (MHz or 0 for none) = "); scanf("%f", &fuse);
printf("Time importance (max importance 1.0) = "); scanf("%f",
&timep);
printf("Sampling size = "); scanf("%d", &sample);
printf("Distribution [(n)ormal, (e)ven, (d)iscrete] = ");
scanf(" %c%*c", &dist);
tolower(dist);
```

```
/* selection of usage distribution type                                */
switch(dist) {
   case 'e':
      for(i=0;i<sample;i++) {
         usage[i].r = 1.0/sample;
         }
      break;
   case 'd':
      printf("Please make sure the total usage = 1.0 (100%)\n");
      for(i=0;i<sample;i++) {
         printf("Enter precision bits: ");scanf("%d", &usage[i].n);
         printf("Enter usage ratio: ");scanf("%f", &usage[i].r);
         }
      break;
   default:
      printf("Defaulted to even distribution\n");
      for(i=0;i<sample;i++) {
         usage[i].r = 1.0/sample;
         }
   }

/* calculating the RC delay for given core clock frequency        */
if (fuse != 0) {duse = 1000/fuse; cf=1;} else {cf=0;};
printf("duse = %f\n", duse);

/* calculating the area importance from the given time importance */
areap = 1 - timep;

delayinit();

evalue(maxarea, maxn, minn, cf, duse, sample, maxw, dist);
attcsearch(maxarea, maxw, areap, timep);

sortdesignbyarea(maxarea, maxw);
btatcsearch(areap, timep, maxarea, maxw);

timeit(maxn, minn, sample, dist);
}
```

Appendix D. Kernel Design Tool Input and Output Example

```
INPUT
Maximum area = 26500
Maximum precision = 2048
Minimum precision = 160
Frequency used (MHz or 0 for none) = 0
Time importance (max importance 1.0) = .95
Sampling size = 5
Distribution [(n)ormal, (e)ven, (d)iscrete] = d
Please make sure the total usage = 1.0 (100)
Enter precision bits: 160
Enter usage ratio: .2
Enter precision bits: 256
Enter usage ratio: .2
Enter precision bits: 512
Enter usage ratio: .2
Enter precision bits: 1024
Enter usage ratio: .2
Enter precision bits: 2048
Enter usage ratio: .2


OUTPUT
Best Candidates:
   w    p     area      fuse(MHz)   ettime(us)   fmax(MHz)   ettimef(us)
   8   32   16632.961   108.992      41.834       108.992      41.834
  16   17   16567.760    86.957      50.177        86.957      50.177
  32   14   26415.840    61.115      46.479        61.115      46.479
  64    7   25063.760    54.054      53.731        54.054      53.731
 128    3   19056.400    54.645      64.398        54.645      64.398

Best design for area-time tradeoff:
time importance: 0.95000   area importance: 0.05000
word size = 8   pipe stages = 32   area consumed = 16632.960938
frequency use = 108.992    max frequency = 108.992
  n     time-fuse   time-fmax
  160      3.119       3.119
  256      4.991       4.991
  512     10.258      10.258
 1024     38.737      38.737
 2048    152.066     152.066
```