ALU FOR MBEDTLS DIFFIE-HELLMAN PARAMETERS GENERATOR ON FPGA EMBEDED PROCESSOR SYSTEM

An Undergraduate Research Scholars Thesis

by

CHANGNING CHEN and BRIAN DEMPSEY

Submitted to the Undergraduate Research Scholars program
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Research Advisor: Dr. Samuel Palermo

May 2016

Major: Electrical Engineering

TABLE OF CONTENTS

ABSTRACT

ALU for mbedTLS Diffie-Hellman Parameters Generator on FPGA Embedded Processor System

Changning Chen and Brian Dempsey
Department of Electrical & Computer Engineering
Texas A&M University

Research Advisor: Dr. Samuel Palermo
Department of Electrical & Computer Engineering

Safe prime is a unique subset of the general prime number where both p and $\frac{p-1}{2}$ are primes. Commonly used Public Key encryption scheme Diffie-Hellman key exchange algorithm utilizes ultra large safe primes as the private key. In practice, crypto software libraries implement a specific Diffie-Hellman parameters generator that searches for safe primes with Rabin-Miller probabilistic primality test algorithm. Without any proven theory to predict their occurrences among natural numbers, generator programs generally start at a randomly seeded odd positive integer of a predetermined size; and perform primality tests in iterations over incrementing candidates until success. The staggeringly low density of safe primes causes a prohibitive amount of computing resources to be dedicated in the generation process. As the result, power conscious mobile and embedded devices can no longer compute the standard 2048-bit safe primes without causing prolonged disruption to the overall system performance. Based on the hot path analysis of the generator program, a parallelized and pipelined ALU is proposed and implemented on the FPGA embedded processor system. Utilizing merely 3% of LUT (584/17600) and 20% of DSP (16/80) available from the Xilinx Zynq 7010 All Programmable SoC, the suggested design is theoretically capable of offsetting more than 90% of CPU utilizations needed for the entire safe prime generation process. Such results demonstrate the deficiency of today's general purpose CPU in

handling certain complex and resource intensive computations. Such scenarios greatly incentivize the integration of programmable hardware with fixed design CPU. Additional research is suggested to focus in the area of automating the processes of locating the specific CPU intensive task, translating such task onto programmable hardware, and providing software accessible interface to enable fast development and deployment of the hot function based programmable hardware design. From there, programmable hardware assisted computing platforms can be further enhanced to dynamically program hardware modules based on real-time utilizations to achieve even greater overall system performance. A new system design paradigm can potentially be introduced as the result.

DEDICATION

We dedicate this thesis to our family, friends, and mentors who dedicated their time and patience to ensure we succeeded. A special gratitude goes to Mrs. Xi Li, wife of Changning Chen and Mrs. Bridget Dempsey, wife of Brian Dempsey, whose kindness and devotion has endured the extent of this research. The time and effort put into this thesis would not have been possible without their continuous love and support.

To Ms. Xixian Tong, mother of Changning Chen, for those lessons in resilience and perseverance that empowered the unrelenting driving forward. They shall be remembered and cherished forever.

To Lynn and Keith Dempsey, parents of Brian Dempsey, whose words of encouragement and reassurance have allowed him to succeed as an individual, and as a student. He is forever indebted to them for the drive and determination they instilled to follow my dreams.

Finally, to our children, who have suffered countless time away from their fathers in the name of research. Cohen Dempsey, Harper Dempsey, Scarlett Dempsey, and Yuntao Chen, we love you and hope one day you will recognize the value of education and teamwork as we have.

ACKNOWLEDGMENTS

We would like to give our most sincerely thank to the project faculty advisor Dr. Sam Palermo of the Electrical and Computer Engineering Department at the Texas A&M University for being more than generous with the firm support of his knowledge, expertise, patience and time. Additionally, we want to thank Dr. Paul Gratz, also a professor at the TAMU ECEN Department for providing some invaluable directional advices and expert opinions. Finally, we would like to acknowledge and thank Dr. Jean-Francois Chamberland-Tremblay of the TAMU ECEN Department and Dr. Peter Stiller of the TAMU Math Department for their constructive inputs. Finally, we would like to thank the entire Texas A&M Undergraduate Research Scholar program staff for this precious research opportunity and all the generous assistance they have provided.

NOMENCLATURE

ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IP	Intellectual Property
ISE	Integrated Synthesized Environment
ISIM	Integrated Synthesized Environment Simulator
MPI	Multiple Precision Integer
LUT	Look Up Table
RSA	Rivest-Shamir-Adleman
RPi	Raspberry Pi
VLSI	Very Large Scale Integrated

CHAPTER I

INTRODUCTION

Comparable to the printing press and the telegraph, the rise of the internet has revolutionarily transformed almost every aspect of our society [1]. Over the past few decade, observed as Moore's Law, the semiconductor industry maintained steady logarithmic scale expansion of the VLSI circuit transistor-count. This allowed the continuous development of faster, smaller, more power efficient, yet exceptionally more affordable computing devices. Today, any sub $300 laptop can outperform a $20 million supercomputer from the 80's with ease [2]. Additionally, the leap forward in wireless communication technologies amplified by the uptrend in performance per watt capabilities of general purpose Central Processing Units (CPUs) and the downtrend in cost per gigabyte on memory/storage devices fostered an increasingly broad new platform of mobile communication and cloud computing [3]. The wireless internet age dawns.

Internet communications transmit in open medium, meaning the messages can be easily intercepted. Cryptographic schemes have been implemented for security purposes. Among them, public key encryption schemes such as the RSA and Diffie-Hellman key exchange algorithm are well regarded as most influential. Truly, the internet owes part of its success to them because secure communications through open transmission medium between two unknown parties was not considered possible prior to their inception.

Interestingly, based on number theories and computer science practices, public key algorithms are designed around computational hardness assumptions [4]. Therefore, if such assumptions are

nullified, either by new discoveries in mathematics or through brute force attacks utilizing the improved computing power brought by the newer generation technology, modern cryptographic algorithms will collapse. Exposing vital secure communications at a global scale can have devastating repercussions to the modern society.

Key, key exchange, and the prime number

Cryptanalysis assumes the method of encryption/decryption is known by the adversary, thus the only thing that stops the adversary from knowing your plaintext message is a securely transmitted crypto key. However, transmitting the key insecurely would defeat the purpose of using cryptography. This is the Key Exchange Problem [5]. Additionally, modern public key crypto-algorithms are asymmetrical by design, meaning computational heaviness is unevenly distributed between the corresponding encryption and decryption algorithms. Such characteristics make public key algorithms unattractive for maintaining continuous communication. Thus, in practice, secure transmissions are initiated by both unmet parties generating a secure key using one form of public key crypto-algorithms, such as the commonly used Diffie-Hellman key exchange algorithm. The remaining transmission is then protected by a faster symmetrical crypto-algorithm using the secure key, such as the Advance Encryption Standard.

Public key crypto-algorithms heavily depend on the interesting properties of prime numbers. First, since there is no proven theory to govern their occurrences among natural numbers, entropy can be extracted based solely on their existence. Second, prime numbers obey certain mathematically laws with stunning regularity and extreme precision [5], of which, instills a rigid mathematical integrity. Capitalizing fully on those properties, Diffie-Hellman key exchange algorithm relies on

its ultra large prime modulus for security, more specifically, an ultra large safe prime where both modulus p and $\frac{p-1}{2}$ are prime numbers. However, such inseparable dependency along with the periodic leap forward in computing power have produced a continuous loop where an ever-larger safe prime number modulus is required to offset the increase in computing power. As a result, the default size of the safe prime modulus has steadily risen to 2048 bit over the years. Unfortunately, along with the increase in computing power, so raises the difficulty in finding ultra large safe prime numbers.

Computing the ultra large safe prime modulus

The common method of conventional prime number generation is to start at a random odd number, s, of the desired bit-width from a high entropy pseudo-random source. From there, a set of candidates is established that includes $\{s, s+n, s+2\cdot n, \dots, s+k\cdot n\}\ \ k \in \mathbb{Z}^+$, with n being a small even number. Then, the probabilistic Rabin-Miller primality tests iteratively progresses through the set until the one candidate passes the test and returns as the prime number. By definition, the Rabin-Miller primality test assesses number n using m randomly chosen values of $b < n$. If n is composite, the probability that it is a strong pseudo prime for one b is at most $\frac{1}{4}$, so the probability that it passes all m tests is at most $\left(\frac{1}{4}\right)^m$. Therefore, if n passes all m tests, then n is prime with a probability at $1-\left(\frac{1}{4}\right)^m$ [5]. In practices, after 5 successful pass of the Rabin-Miller test using different b values, the probability of n being prime becomes$1-\left(\frac{1}{4}\right)^5 = 0.999023$, or 99.9023%, already a reasonably high probability to suggest confidence in the state of primality. Surely, additional tests can be performed to increase such probability further; however, the return per test diminishes remarkably quickly.

Even though stipulated by the prime number theorem, an infinite number of primes do exist; they are still an extremely rare breed among natural numbers. The average gap between consecutive prime numbers less than n is roughly $\log(n)$ [5], meaning the larger the number, the larger the gap between successive primes. As a unique but small subset of the regular primes, the safe prime's staggeringly low density crucially increases the number of candidates in between two successive safe primes. Furthermore, since both p and $\frac{p-1}{2}$ must pass the Rabin-Miller primality test, the amount of computation needed doubles at each candidate. Collectively, depending on the initial randomly seeded starting position, extremely to prohibitively costly amount of computing resource must be dedicated to the generation process while the operating system and other potentially critical programs may suffer prolonged and continuous deficiency in available resources.

Unfortunately, even though the total computing power provided by the cloud computing platforms have skyrocketed, as the essential parameter of the Diffie-Hellman key exchange algorithm, safe prime number modulus must be computed privately by where only the limited local computing resource is available. This causes significant issues on the power conscious mobile and embedded platforms because of their less powerful CPUs. Thus, to avoid the heavy computations involved in generating a large arbitrary precision prime, they are not typically generated from scratch. Rather, they are reused from previous work or taken from recommendations in established standards [6], making such practices a major security vulnerability for the Diffie-Hellman key exchange [7].

Capitalizing on the combined iterative nature of the multi-candidate style generation procedure and the successive Rabin-Miller primality tests based individual candidate checking method, this

project proposes a dedicated Field Programmable Gate Array synthesized ALU module to alleviate the overall CPU resource utilization of the safe prime generation process.

Organization

To provide a better understanding of today's hardware and software environment, Chapter 2 details the various common hardware platforms selected for testing and the profiling software tools used. Additionally, the methods of analysis and implementation are presented along with the analytical software used to provide an overview of the project direction.

Chapter 3 focuses on the analyses of the aggregated testing results obtained from different hardware platforms. The hot function of the generator program is pinpointed and studied extensively. From there, timing, data paths, and other design metrics are extracted to implement a graphical representation of the actual algorithm, which the proposed hardware will be based upon.

Chapter 4 describes the procedures taken to actuate the implementation on the specific design platform. With the careful evaluation of those newly added system constraints, the proposed implementation of the ALU is presented. Last, to ensure design correctness, the rigorous testing of the system is detailed.

Finally, Chapter 5 concludes the thesis with the proposed hardware design summary and possible future research areas.

CHAPTER II

METHODS

Intel CPUs are based on the influential x86 Complex Instruction Set Computer (CISC) style microarchitecture. ARM processor is on the other end of the spectrum with the ARMv7 Reduced Instruction Set Computer (RISC) style microarchitecture. Their combined dominance in the general purpose CPU market propels this project to select three particular types mobile platform CPUs: Lenovo ThinkPad Yoga 12 with Intel 4th Generation "Haswell" Core i7-4500U representing the high-end Ultrabook laptop market segment, Foxconn Kangaroo with Intel Cherrytrail Atom X5-Z8500 representing the MINI-PC and tablet segment, and Raspberry Pi 2b with ARM Cortex-A7 representing the smartphone and embedded device segment. In terms of the operating systems, Ubuntu 15.10 64-bit Desktop Edition and Microsoft Windows 10 Pro 64-bit are installed on the x86 platforms. Due to the lack of support from Microsoft over the ARMv7 platform, only Debian based Raspbian version Jessie is installed on the RPi platform.

CPUs of the aforementioned platforms are designed with only fixed-precision Arithmetic Logic Unit (ALU) instructions using either 32 or 64-bit wide registers, based on their memory addressing capabilities. However, in order to perform modular arithmetic operations on 2048-bit ultra large numbers, multiple precision arithmetic functions are implemented collectively as software libraries, commonly written in C. Among them, mbedTLS, previously known as PolarSSL, a liberal Apache licensed Free Open Source Software crypto-library, which is also notably maintained by the prominent CPU design firm ARM, became the library of choice for two reasons. First, as its name may suggest, the mbedTLS library differentiates itself as an embedded platforms

solution, where the different cryptographically algorithms and protocols are loosely coupled. The included Diffie-Hellman parameter generator program is an isolated, ready-to-use program that can be compiled by both the Windows based Microsoft Visual Studio and Linux based Make tool. Such isolation helps the project to obtain testing results not skewed by non-essential components such as memory debugging.

Software analysis platform

With the hardware and software firmly in place, profiling software is introduced to gain insights with respect to the overall generator program behavior in terms of performance metrics and resource utilization statistics. Profilers run parallel to the target program, which allows the continuous analysis of the target program's detailed statistics. Among a handful of choices, two of the most popular profiling programs are utilized: Windows operating system based Microsoft Visual Studio and Linux operating system based gprof. Visual Studio Enterprise 2015 offers an in-depth diagnostic profiling tool over the Intel x86 microarchitecture under the Microsoft Windows environment. However, because Windows operating system's scheduler does not allow full system utilizations to the Visual Studio compiled Diffie-Hellman parameter generator program, a win32 console style executable, only the resource utilization distribution statistics portion of the Visual Studio profiling result can be treated as definitive data. Conversely, the Linux operating system allows full system utilization for its terminal run software. As a result, Linux based gprof is also used, even though it is much less sophisticated in comparison to other profilers. However, this is much desired as gprof introduces very little overhead for the actual profiling operations. It makes gprof output data much more authoritative in performance data, as it exposes the maximum amount of system resource to the program under profiling. Once data gathering and aggregation

completes, the project shifts the focus to the understanding of the dictating algorithms and the method of their implementation within the generator program. Any meaningful results are then forwarded downstream as potential design metrics.

Design implementation platform

During the designing of the proposed hardware, the main objective is to perform a weight based assessment of all the design metrics to determine their relative significance; then develop a hardware design based on the previous evaluation results. National Instrument's myRIO device is introduced as a hardware development platform because it is powered by the Field Programmable Gate Array (FPGA) device market leading manufacturer Xilinx's Zynq 7010 series All Programmable System-on-Chip (SoC) processor, where FPGA device is integrated with a "hard" dual-core ARM Cortex-A9 CPU on the same die. This duality style design enables the project to deploy the safe prime generator program to run under the Linux Real-Time operating system on the Cortex-A9 CPU while allowing the project to deploy the proposed hardware module and modify the mbedTLS program to utilize such module. National Instruments myRIO development platform's specifications need to be taken into special consideration, as they place significant system constraints to the overall design in the area such as the clock speed, IO capabilities between the ARM core and the FPGA, as well as the method of development. From there, a design is draft by considering all potential design metrics collectively. Before entering the next stage, such a proposal is reevaluated against all metrics to ensure conformity to the weighted design metrics.

Once the design is proposed, the focus is shifted to the construction a functioning prototype. Xilinx's FPGA is usually programmed using Xilinx Integrated Synthesize Environment with

Hardware Description Language's such as Verilog. Even though National Instruments myRIO device only natively supports the NI LabVIEW graphical programming language, the LabVIEW FPGA program environment does support the importation of Xilinx Component Level IP block which can be generated from the Xilinx ISE based HDL project. As the result, this project utilizes the Xilinx ISE as the main development platform, as it offers an extensive suite of analytic and testing tools for synthesizing FPGA designs. Finally, the implemented prototype undergoes extensive testing within Xilinx ISE using custom test bench code to ensure the logic correctness of the proposed design.

CHAPTER III

ANALYSIS

Based on mbedTLS version 2.2.1, nine hundred profiles of 2048-bit safe prime computations are automated using gprof across all three hardware platforms with Linux shell script. Without user interference, the gprof generated data is more accurate as full system resources dedicated uninterruptedly to the generator program. Such data is aggregated and processed to demonstrate the relationship between the entire computation run time and the individual function run time versus the hardware platforms.

Diffie-Hellman safe prime generator profiler results

With the x-axis showing the number of profile iteration linearly and the y-axis showing the total computation time in logarithmic scale, Fig. 1. depicts the negative proportional relationship between the CPU performance and the total time spent for single prime generation where a more powerful Core i7 is taking considerably less computation time than the ARM Cortex-A7. From 5 seconds to 7 hours, the gargantuan difference in total computation time is considered to be contributed by the randomly seeded starting position. If such position is relatively close to a safe prime, then relative insignificant amount of computation is required. However, when the initial position is rather far away from the next safe prime, compounded by the slower CPU performance, a prohibitive hour long computation could be required at 100% CPU utilization. Overall, this graph demonstrates the significance in the total amount of CPU resource needed for a single 2048-bit safe prime.

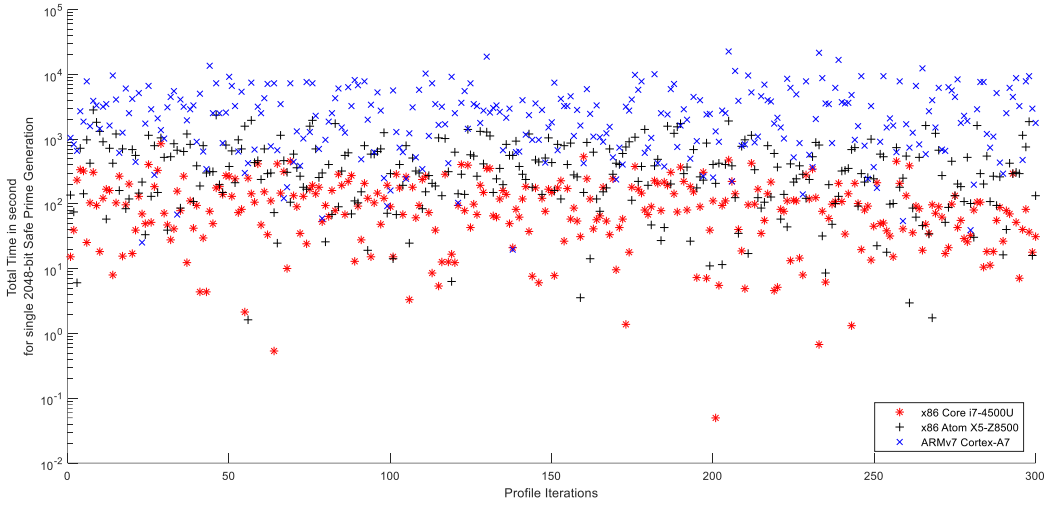


Fig. 1. 2048-bit Safe Prime Generator Run Time per Iteration

The most important result of the gprof profiler data is the hot function of the Diffie-Hellman safe prime generator program. Represented in Fig. 2., with the consistently dominating over 90% overall CPU time utilization, the project turns focus to the mpi_mul_hlp() function. Additionally, the similar behavior in x86 based Core and Atom CPUs with the minor ±4% gap in utilization between the x86 group versus the ARMv7 based Cortex-A7 CPU demonstrates the microarchitecture based differentiation in actual performance metrics.

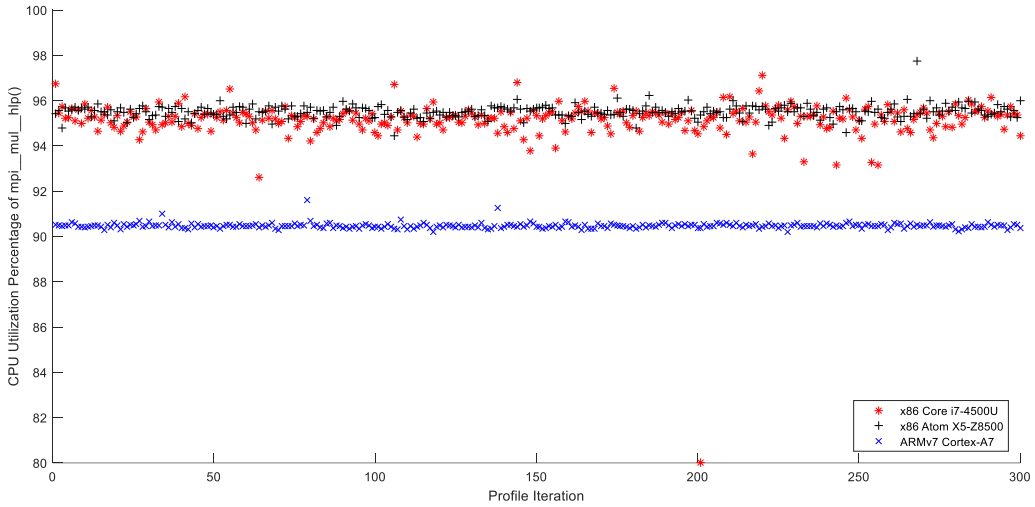


Fig. 2. 2048-bit Safe Prime Generator Hot Function CPU Utilization per Iteration

Using the same low overhead type of CPU sampling method, the project performs generator program profiling on the x86 machines using Windows based Microsoft Visual Studio 2015 Enterprise Integrated Development Environment. The Visual Studio results perfectly complement the gprof profiler's data. Taken directly from Visual Studio, Fig. 3. below demonstrates the function call hierarchy of the generator program dh_genprime. The overall 97.18% inclusive samples by the mpi_miller_rabin confirms the iterative nature of the Rabin-Miller primality test algorithm. From there, mpi_mul_hlp() function is presented as the hot function with 94.25% exclusive CPU time usage with over 3 million exclusive samples taken. Additionally, the dependency tree suggests that mpi_mul_hlp() function is a multiplication helper function in performing the Montgomery multiplication algorithm.

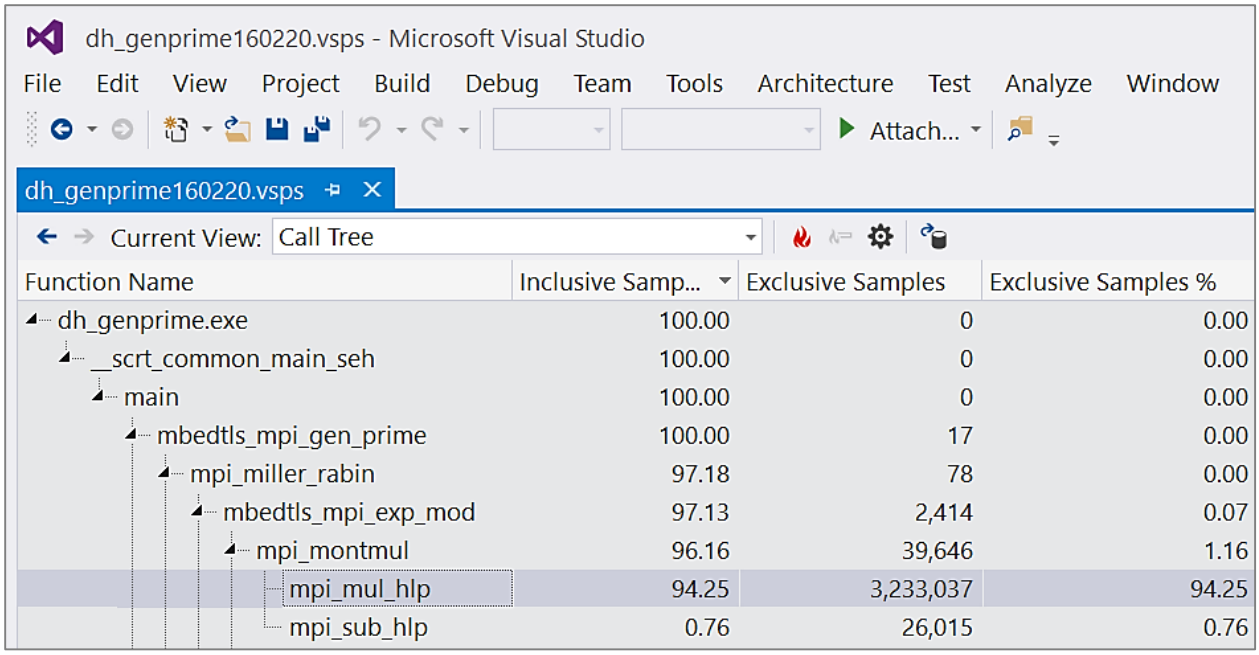


Fig. 3. Diffie-Hellman Safe Prime Generator Program Function Call Tree

Further examination of the Visual Studio profile result shown on Fig. 4. confirms the previous proposition. The mbedTLS implementation of Montgomery multiplication algorithm is showing with a left column populated by the Visual Studio profiler results, of which details the CPU time utilization of each line. Towards the bottom, two instances of the hot function mpi_mul_hlp() are apparent with the near identical 47.1% and 47.9% exclusive CPU utilization. Evidently, the function comments atop suggest that this implementation of the Montgomery multiplication algorithm is based on algorithm 14.36 from the Handbook of Applied Cryptography.

```
          /*
           * Montgomery multiplication: A = A * B * R^-1 mod N  (HAC 14.36)
           */
          static void mpi_montmul( mbedtls_mpi *A, const mbedtls_mpi *B, const mbedtls_mpi *N,
                                   const mbedtls_mpi *T )
< 0.1 %   {
              size_t i, n, m;
              mbedtls_mpi_uint u0, u1, *d;

< 0.1 %       memset( T->p, 0, T->n * ciL );

              d = T->p;
< 0.1 %       n = N->n;
< 0.1 %       m = ( B->n < n ) ? B->n : n;

< 0.1 %       for( i = 0; i < n; i++ )
              {
                  /*
                   * T = (T + u0*B + u1*N) / 2^biL
                   */
                  u0 = A->p[i];
  0.2 %           u1 = ( d[0] + u0 * B->p[0] ) * mm;

 47.1 %           mpi_mul_hlp( m, B->p, d, u0 );
 47.9 %           mpi_mul_hlp( n, N->p, d, u1 );
```

Fig. 4. mbedTLS mpi_montmul() Code with Lined Based CPU Usage

Montgomery multiplication algorithm and the helper function

Montgomery multiplication algorithm accelerates the solving of $x \equiv a \cdot b \ mod\, n$, where both operands a, b and modulus n are ultra large integers. The classic method incurs heavy computations attained from the complex multiplication of two ultra large numbers and the excessive number of divisions required for the final modulo operation. However, based on the Montgomery reduction algorithm, Montgomery multiplication algorithm uses easy-to-realize shift operations in place of the more difficult division operations. By introducing one extra multiplication, one addition, and a conditional subtraction [8], Montgomery multiplication algorithm removes the computational intensive modulo operation entirely. While the overhead associated with the added operations can be over exorbitant for small integers, once entering the

Multiple Precision Integer (MPI) space, it becomes relatively insignificant. With the removal of the modulo operation, shown on Fig. 3., the 96.16% inclusive CPU utilization of the mpi_mont_mul() function is predominantly contributed by the multiplication computations.

14.36 Algorithm Montgomery multiplication

INPUT: integers $m = (m_{n-1} \cdots m_1 m_0)_b$, $x = (x_{n-1} \cdots x_1 x_0)_b$, $y = (y_{n-1} \cdots y_1 y_0)_b$ with $0 \leq x, y < m$, $R = b^n$ with $\gcd(m, b) = 1$, and $m' = -m^{-1} \bmod b$.
OUTPUT: $xyR^{-1} \bmod m$.

1. $A \leftarrow 0$. (Notation: $A = (a_n a_{n-1} \cdots a_1 a_0)_b$.)
2. For i from 0 to $(n-1)$ do the following:
   2.1 $u_i \leftarrow (a_0 + x_i y_0) m' \bmod b$.
   2.2 $A \leftarrow (A + x_i y + u_i m)/b$.
3. If $A \geq m$ then $A \leftarrow A - m$.
4. Return(A).

Fig. 5. Algorithm 14.36 from the Handbook of Applied Cryptography [9]

Fig. 5. shows the computer science implementation details of the actual Montgomery multiplication. Knowing mbedTLS' mpi_mont() function is based on such implementation, the project found that with some simple manipulation to the Eq. 1. from Fig. 5., it can be separate in three consecutively performed operations presented by Eq. 2.1 - 2.3. Clearly, the Eq. 2.1 and Eq. 2.2 not only carry identical format but also the same serial operation style as the mpi_mul_hlp() function. To speed up the process, mpi_mul_hlp() function is optimized in its entirety using C macro based CPU microarchitecture specific assembly codes to avoid compiler interference. With $b = 2^{32}$ in the mbedTLS implementation, the final division presented by Eq. 2.3. can be translated to a simple shift operation where the least significant 32-bit of S or its equvalent $(A + X_i \cdot Y + U_i \cdot M)$ is simply discarded.

$$A \leftarrow (A\ + X_i \cdot Y\ + U_i \cdot M)/b \tag{1}$$

$$R \leftarrow\ (A\ + X_i \cdot Y\ ) \tag{2.1}$$

$$S\ \leftarrow (R + U_i \cdot M) \tag{2.2}$$

$$A \leftarrow S/b \tag{2.3}$$

Closer examination of Eq. 1. reveals that parallelization can be applied both internally and externally. Within each multiplication, because X_i is a 32-bit unsigned integer and Y is an equal bit-sized unsigned integer as the safe prime number in generation of n-bit, Y can be rewritten as Eq. 3. where $Y_0, Y_1, \dots, Y_n$ are 32-bit unsigned integers with Y_0 being the least significant 32-bits. Those 32-bit pieces are commonly referred as the limbs of an MPI.

$$Y = Y_0 \cdot 2^{32 \cdot 0} + Y_1 \cdot 2^{32 \cdot 1} + Y_2 \cdot 2^{32 \cdot 2} + \cdots + Y_n \cdot 2^{32 \cdot n} \tag{3}$$

$X_i \cdot Y$ can be readily expanded using the distributive property of multiplication to arrive at Eq. 4.

$$X_i \cdot Y = X_i \cdot Y_0 \cdot 2^{32 \cdot 0} + X_i \cdot Y_1 \cdot 2^{32 \cdot 1} + X_i \cdot Y_2 \cdot 2^{32 \cdot 2} + \cdots + X_i \cdot Y_n \cdot 2^{32 \cdot n} \tag{4}$$

Clearly, the same transformation applies to the multiplication of $U_i \cdot M$. It is apparent both multiplications can be individually parallelized by n-threads of smaller 32-bit-by-32-bit multiplications. Both special case multiplications are serially implemented with separate helper functions; however, since the none of the operands and results of both multiplications are dependent of one another, two multiplications can be performed concurrently. Collectively, a notable amount of computations from Eq. 1. can be executed in parallel to reduce overall run time.

Graphical representation of parallelized $\boldsymbol{A \leftarrow (A\ + X_i \cdot Y\ + U_i \cdot M)/b}$

Following the size requirements set forth by Fig. 5., this project presents an example based on Eq. 5. External and internal parallelization capabilities of the Eq. 1. are demonstrated by Fig. 6. and Fig. 7., respectively.

$$A[0:159]\ \leftarrow (A[0:159]\ + X_i[0:31] \cdot Y[0:127]\ + U_i[0:31] \cdot M[0:127])/b[0:31] \qquad (5)$$

From Fig. 6. each square is a memory element of 32-bit or 4-bit in size. Each horizontal box of 2 memory elements represents an instance of 32-bit by 32-bit multiplication operation that can be parallelized, while its 64-bit result is evenly separated and stored into two 32-bit sized memory element. This level of parallelization is the most influential determinant to the overall circuit speed because the critical path is determined by the position of the slower multipliers. The descending vertical arrows performs the column based operation of $f \leftarrow\ (e_1 + \cdots + e_j)$, where $e_1, \ldots, e_j$ are all the memory elements aligned directly above the f element at the base. The ascending arrows are simple forwarding operations. From observation, the first column has 4 elements with $j = 3$; the last column has 2 elements with $j = 0$, the second to the last column has 5 elements with $j = 4$, and all the remaining columns have 7 elements with $j_{max} = 6$. In other words, the majority of the return value's limbs require the maximum 6 additions; while less additions are needed for the initialization and finalization stage. Based on extended testing, this project found such result is preserved for all As with size greater than 64-bit, or equivalently as having more than 2 limbs. This establishes the basic internal data path control logic. Centering on the majority 6-addition columns, Using the second column of Fig. 6. as an example, Fig. 7. depicts the internal parallelization by

rearranges 3 out of the 6 linear additions to 3 parallelized additions represented by the 3 vertical boxes. Six stages of additions are reduced down to 4 stages as the direct result.

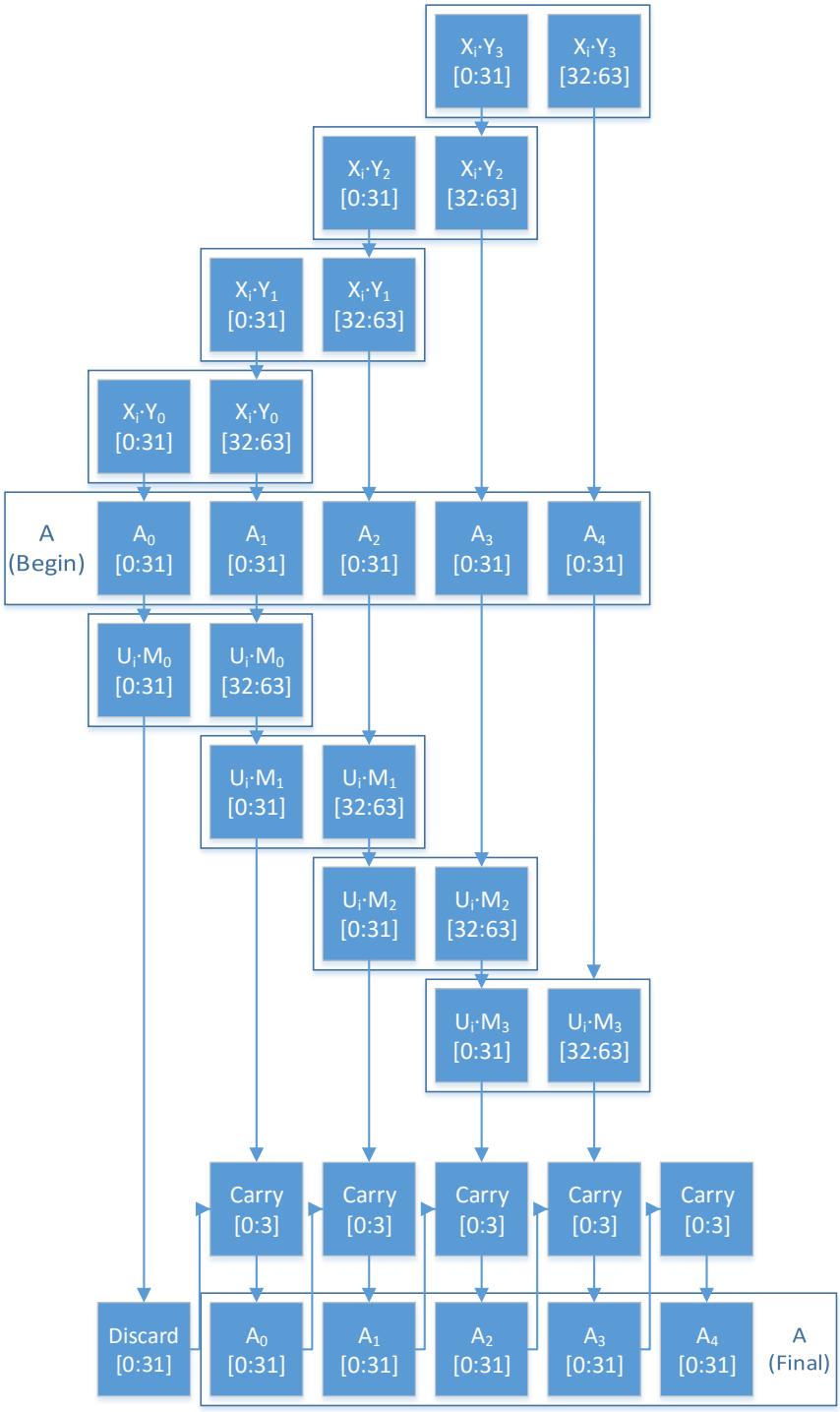


Fig. 6. External Parallelization Diagram

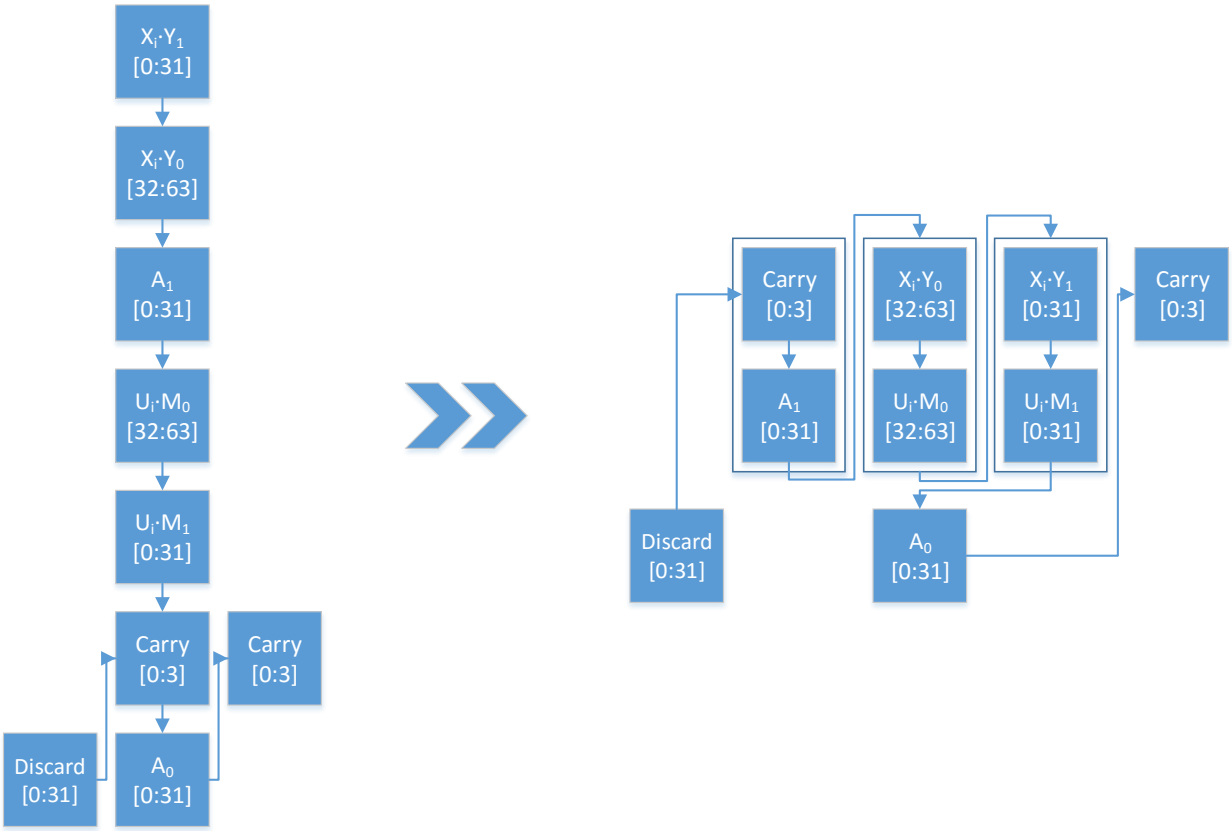


Fig. 7. Internal Parallelization Diagram

CHAPTER IV

IMPLEMENTATION

To successfully create and implement an optimized peripheral hardware form of Eq. 1., it is necessary to analyze the basic system components such as multipliers, adders and registers, as well as the manner in which they behave in unity. This is accomplished through the use of target deployment platform specifications within the Xilinx ISE. This software is used to extract the maximum operating frequencies of the individual components as well as their maximum operating load power consumption. The implemented pipelined acceleration hardware is then developed in stages before being rigorously verified through Verilog-based test benching, in which the results are compared to that of preexisting implementation data.

Component maximum operation frequency extraction

The essence of any hardware is the frequency at which it operates. In order to analyze the maximum frequency at which each component can operate, a Verilog implementation of each building block is instantiated and tested through input and output test arrays. By feeding values into the arithmetic unit, or flip flop in the case of a register, the delay can be measured from the input of the numerical data to the output of the resultant. Through the synthesis of these components, a maximum delay is produced, which yields the maximum operating frequency of any such unit. Table 1. shows the extracted values for each of the respective components of the design. These frequencies dictate the maximum computational speed, as well as the speed at which values are pushed from one stage of the pipeline to the next.

Table 1. Pipelined Component Frequency Analysis Data

Pipelined Design Component	Max Frequency
32 Bit Register	956.023 MHz
64 Bit Register	956.023 MHz
32 Bit Multiplier	223.449 MHz
32 Bit Adder	678.228 MHz
64 Bit Adder	533.532 MHz

For the 32-bit and 64-bit adder, the implementation consists of two inputs of the corresponding adder bit length, along with a clock signal and an output equivalent to the bit quantity + 1. Through Xilinx logic amalgamation, the maximum frequency is found to be 678.228 MHz and 533.532 MHz, respectively. The method is identical for the multiplier with the exception of the output bit width becoming twice as long as the input, 64 bits. The synthesis of the 32-bit multiplier yields a maximum operating frequency of 223.449 MHz, significantly lower due to the complexity involved with the multiplication operation in hardware. The registers, used as intermediary value latches, are tested simply through pushing values of the respective bit length through the simulated design. By extracting the delay from the arrival time of the input value to the time required for output realization, the maximum operating frequency is obtained. It is of importance to note that due to the lack of a substantial path to and from the registers for an individual component, additional internal data manipulation is needed to extract a meaningful clock rate. This synthesis yields a maximum value latching frequency of 956.023 MHz for both bit length registers. This is due to bus data arriving at each respective register input at the same time under ideal conditions. Due to the potential data arrival skewness through the pipeline, the peripheral ALU operation frequency will be throttled 5 Hz below the operating frequency of the slowest component, in this particular case, the multiplier.

Pipeline development

In order to maximize the frequency by which the mpi_mul_hlp() function is able to compute results, a pipelined system is utilized, characterized in Appendix A. This pipeline is partitioned into separate sections, each having a unique arithmetic operation, or computation bit length. The transitional stage between these arithmetic operations contains the registers responsible for holding preceding arithmetic results until the next clock cycle, where the operational resultants is pushed forward to the next stage. Each of the intrinsic functionalities of the arithmetic stages is described below along with specific constraints based on the pipelined design.

The loading stage is the beginning of the mpi_mul_hlp pipelined hardware. In this initialization stage, the values for the first 8 32-bit registers are loaded into their corresponding locations. This is done through the use of input control signals which coincide with a specific register location. This 4-bit signal control first loads the 32-bit limb values X_i and U_i, then the remaining values, which are the upper and lower 32-bits of the input parameters Y and M. Once the final multiplication dependent value is loaded, a flag is set which initiates arithmetic stage 1. As this occurs, the remaining values, A_1 and A_2, are loaded into their respective registers for use in the second stage. Once all values are loaded, the control signal stalls until stage 2 results are latched, before alternately replacing the coupled values $[Y_1\ M_2]$ and $[Y_2\ M_2]$, based on pipeline cycle iteration. For the first arithmetic stage, four 32-bit multiplications are performed in parallel, yielding a 64-bit product for each instantiation. These products are released to an output register feeding into the second stage input latch before an arithmetic stage specific flag is set high, signifying the completion of the multiplication stage of the pipeline. This flag register is wired to the subsequent arithmetic unit instantiation as an always block trigger, allowing the next stage to

perform its respective operation as soon as the previous stage is complete and the results are latched in to the stage two input registers. Once the second stage completes the first set of computations, the values of the products are no longer needed. Consequently, the flag is reset to low, effectively resetting the multiplication module in its entirety.

The second stage consists of cascaded sets of addition modules along with a register used to retain the value of the $s2b$ addition instance. Due to the configuration of the pipelined design, the $s2b$ value must be buffered for a single cycle and added in the subsequent pipeline iteration. A multiplexed supplementary constant value register is used for the first stage, with the buffer register as the logic high selection for the multiplexer. The select bit of the multiplexer is controlled through the return register, in which the latching of a return value sets an internally controlled flag high for the duration of the pipeline cycling. The preliminary arrangement of the cascading adders is set through logic in the module, while the most efficient LUT based mapping and routing is left to the compiler's optimization capabilities.

Once the finalized sum of the second stage is complete, the values are pushed into an intermediary register which, as previously mentioned, releases the stalled $s2b$ register value into the second stage for the next iteration. The values are then shifted into 2 separate 32-bit return registers, R_1 and R_2. Due to the addition of propagating carry bits, the R_2 register has a 4-bit extension which is separated from the final return and placed into a stage 2 register for the next iteration. After the pipelined hardware is implemented the system undergoes an initial analysis which yields 584 used Look Up Tables (LUTs) out of 17600 and 20 used Digital Signal Processors (DSPs) out of 80.

Simulation-based verification

In order to verify the functionality of the optimized pipeline hardware implementation, Xilinx ISim is utilized along with a test bench file, used for the control of inputs to the pipeline as well as the output extraction from the return registers. The testing frequency is set at 220 MHz, slightly below the maximum tested frequency of the multiplier modules.

Fig. 8. represents the pipelined hardware simulation results for a single iteration based on a 0xFFFFFFFF value for all input bus registers with the exception of registers A_1 and A_2 , whose value is set to 0. For the initial iteration of the hardware, the first return register yields an output value of 0x02 while the second return register yields a value of 0xFFFFFFFE. The carry from the lower return register is then returned to stage 2 to be included in the subsequent cycle's arithmetic. This cycle also signifies the release of the buffered s2b register value into the data path of the pipeline.

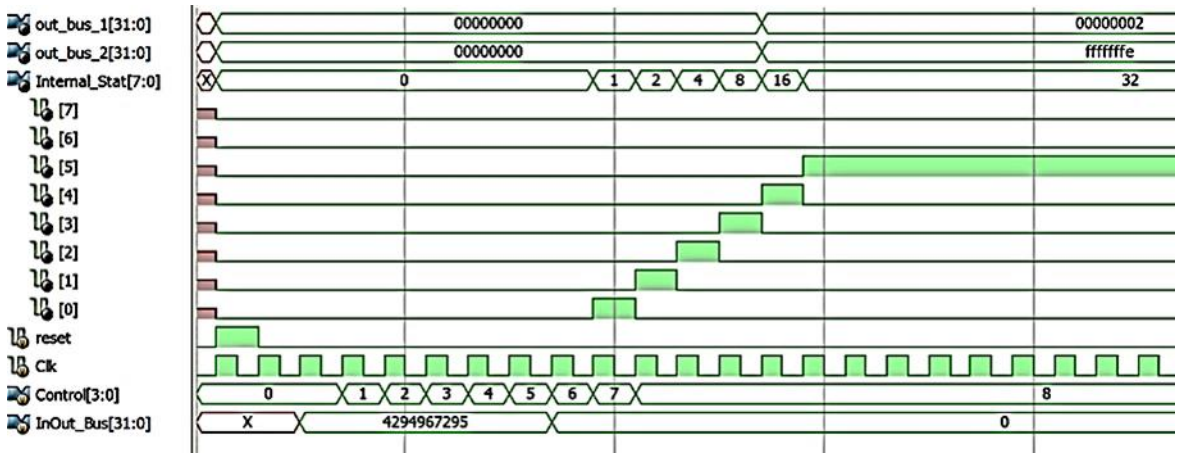


Fig. 8. 1-Cycle Pipeline Simulation Results

Once the operation of the single cycle pipelined helper function is verified to correctly compute accurate intermediate results, the pipeline's cyclical behavior is tested for accuracy. Fig. 9. represents the results of a 2-cycle simulation, in which the carry from the first return set is added to the data path in addition to the stalled value located in the buffer register. Clearly, the 0xFFFFFFFC value from the buffer register, through the cascading adder stage, is combined with the carry bit from the previous cycle to produce the new result. Because this is a two stage simulation, no other values are fed into the test bench, essentially setting most of the remaining arithmetic input and output values to 0. This indicates the 0xFFFFFFFD value is precise for the output of a secondary pipeline cycle in which no further inputs are supplemented. The increase in active stage length is due to the continuous flow of data through the pipelined hardware, which is throttled computationally based on the input clock signal. This successful application of the pipelined Eq. 1. function hardware substantiates the ability to combine both instances of the helper function in a parallelized manner, providing an optimized hardware-based solution that reduces CPU resource dependencies during the safe prime number generation.

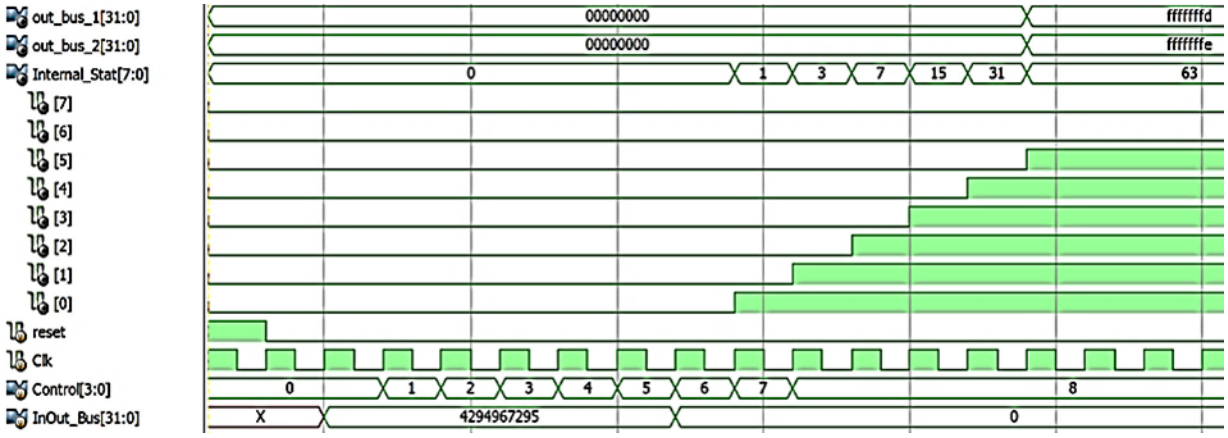


Fig. 9. 2-Cycle Pipeline Simulation Results

CHAPTER V

CONCLUSION

This project proposes a minimalistic implementation of a parallelized and pipelined ALU module for handling the heavy computations of the Montgomery multiplication algorithm on the Xilinx Zynq 7010 FPGA embedded processor system. With more than 90% CPU bound arithmetic offloaded, the remaining residual CPU utilization of the mbedTLS based safe prime generation does not instigate significant impact on the overall system performance. This design enables more frequent random safe primes generation because the cost in CPU resource is significantly lowered. In turn, it may help to patch the logjam security vulnerability of the Diffie-Hellman key exchange protocol. More important, the gain in security is at a marginally minimal cost in both hardware and software. Utilizing merely 3% of LUTs and 20% of DPS of the Zynq 7010 SoC, even with the consideration of a platform-specific IO logic block, the proposed implementation does not introduce heavy expenditures in hardware realization. At the same time, software crypto-libraries such as mbedTLS can be modified to replace its existing Montgomery multiplication helper function with relative ease.

Evidently, ultra large MPI modular exponentiation is the foundation for many other prominent modern cryptographic algorithms such as RSA and Elliptic Curve Cryptography. As suggested by Fig. 3., Montgomery multiplication algorithm function is also the hot function for the modular exponentiation algorithm, which is represented as the mbedlts_exp_mod() function. Thus, following this relationship, the proposed design is essentially a dedicated ALU module for the modular exponentiation algorithm as well.

The combined benefits of the low cost in implementation and wide areas of application support one unique system design methodology: FPGA embedded processor system, where programmable hardware is integrated alongside fixed-design CPU to assist and to accelerate certain tasks that are excessively resource intensive for CPU only systems [10]. This project demonstrates that in the case with the mbedTLS Diffie-Hellman parameters generator, the proposed implementation is capable to provide significant results. Based on such findings, additional research is suggested to focus in the area of automating the processes of locating the specific CPU intensive task, translating such task onto programmable hardware, and providing software accessible interface to enable fast development and deployment of the hot function based programmable hardware design. From there, programmable hardware assisted computing platforms can be further enhanced to dynamically program hardware modules based on real-time utilizations to achieve even greater overall system performance. A new system design paradigm can potentially be introduced as the result.

REFERENCES

[1] S. Gustin. (2013) The Internet Doesn't Hurt People- People Do: The New Digital Age. *Time Magazine* [Online]. Available: http://business.time.com/2013/04/26/the-new-digital-age-promise-and-peril-ahead-for-the-global-internet/

[2] J. Sheesley. (2008, September 7, 2015). The 80's supercomputer that's sitting in your lap. *TechRepublic* [Online]. Available: http://www.techrepublic.com/blog/classics-rock/the-80s-supercomputer-thats-sitting-in-your-lap/

[3] J. B. J. Hagel, T. Samoylova, M. Lui, "From exponential technologies to exponential innovation," Online 2, 2013.

[4] R. R. B.G. Aswathy, "Modified RSA public key algorithm," presented at the 2014 First International Conference on Computational Systems and Communications (ICCSC), Trivandrum, 2014.

[5] J. S. Kraft and L. C. Washington, *An introduction to number theory with cryptography*: Boca Raton, Florida : CRC Press, [2014], 2014.

[6] K. B. David Adrian, Zakir Durumeric, Pierrick Gaudry , Matthew Gree , J. Alex Halderma , Nadia Heninger, Drew Springall, Emmanuel Thome , Luke Valent , Benjamin VanderSloot, Eric Wustrow, Santiago, Zanella-Beguelin, Paul Zimmermann, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," presented at the 22nd ACM Conference on Computer and Communications Security, Denver, Colorado, 2015.

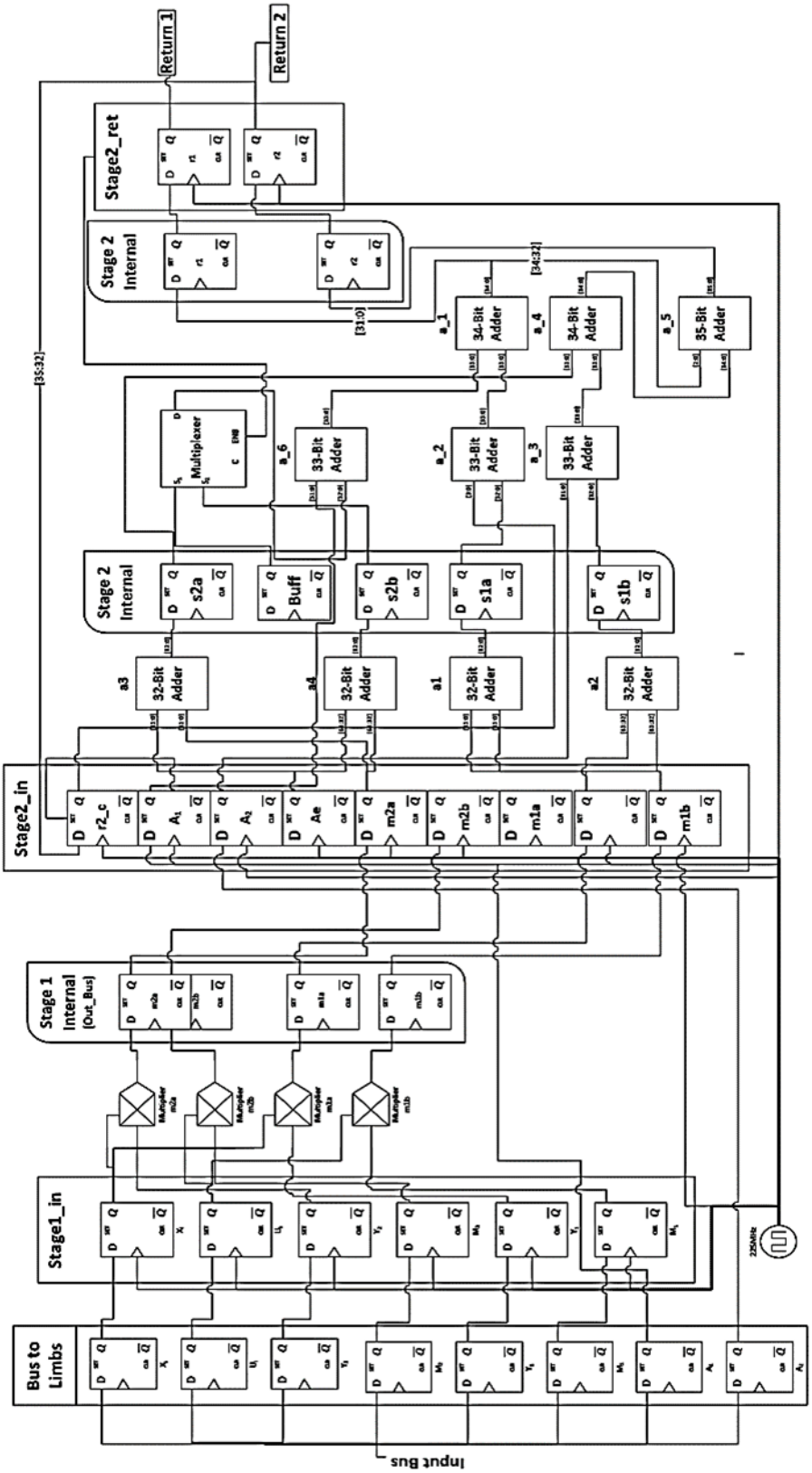
[7] M. Mimoso. (2015, October 5, 2015). *Prime Diffie-Hellman Weakness May Be Key To Breaking Crypto*. Available: https://threatpost.com/prime-diffie-hellman-weakness-may-be-key-to-breaking-crypto/115069/

[8] Y. Gong and S. Li, "High-Throughput FPGA Implementation of 256-bit Montgomery Modular Multiplier," in *2010 Second International Workshop on Education Technology and Computer Science (ETCS)*, 2010, pp. 173-176.

[9] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography. Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone*: Boca Raton : CRC Press, [1997], 1997.

[10] T. P. Morgan. (2015, 2016-04-10 22:50:31). *Why Hyperscalers And Clouds Are Pushing Intel Into FPGAs*. Available: http://www.nextplatform.com/2015/07/29/why-hyperscalers-and-clouds-are-pushing-intel-into-fpgas/

APPENDIX A


Return 1
Return 2
Stage2_ret
Stage 2 Internal
[34:32]
[31:0]
[35:32]
a_1
34-Bit Adder
a_4
34-Bit Adder
a_5
35-Bit Adder
Multiplexer
a_6
33-Bit Adder
a_2
33-Bit Adder
a_3
33-Bit Adder
Stage 2 Internal
s2a
Buff
s2b
s1a
s1b
a3
32-Bit Adder
a4
32-Bit Adder
a1
32-Bit Adder
a2
32-Bit Adder
Stage2_in
r2_c
A1
A2
Ae
m2a
m2b
m1a
m1b
Stage 1 Internal (Out_Bus)
Multiplier m2a
Multiplier m2b
Multiplier m1a
Multiplier m1b
Stage1_in
X
U1
Y2
M2
Y1
M1
225MHz
Bus to Limbs
X1
U1
Y2
M2
Y1
M1
A1
A2
Input Bus