

First Prize

Cryptographic Algorithm Using a Multi-Board FPGA Architecture

Institution: Indian Institute of Technology, Chennai

Participants: G. Ananth and U.S. Karthikeyan

Instructor: Dr. V. Kamakoti

Design Introduction

Information security has assumed a significant importance in today's world, especially because minor breaches can lead to major risks in the fields of national security and other e-commerce applications and transactions. This necessitates implementing cryptographic algorithms in hardware to achieve better security and faster response as opposed to any software implementation. A promising solution combining high flexibility with the speed and physical security of traditional hardware is the FPGA.

Implementing cryptographic algorithms requires the generation of random numbers that can be then used in any algorithm to derive the keys for carrying out a secure transmission. Keeping this in mind, a design was created implementing a multi-board architecture using two Altera® boards. One board constantly generates random numbers using a data encryption standard (DES) random bit generator and at the same time keeps polling its input port for requests by another program designed to receive random numbers. The second board contains a design that implements the RSA algorithm and incorporates the reception of random numbers on the fly by means of hardware interrupts. On receiving the random number, the second board sends an acknowledgement back to the first board to continue the process. The designs (implemented as peripherals) on each board make use of a Nios® embedded processor for communicating and exchanging data between the driver program and the peripheral.

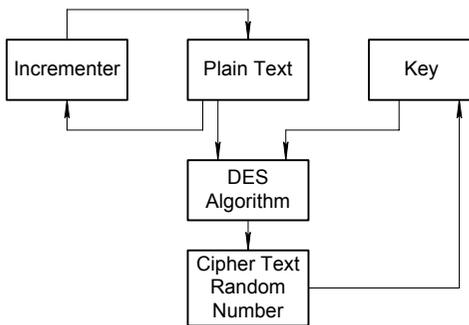
The FPGA device family chosen for implementing the RSA algorithm is Altera's APEX™ 20KE device family. APEX devices are high-density FPGAs that allow complex designs to be implemented on a single device. The target device was an APEX 20K EP200EFC484-2X and the design files were written in Verilog HDL, while compilation, synthesis, fitting, placement, and routing was carried out using the Quartus® II software. The Nios development board provided a hardware platform to immediately start developing embedded systems based on Altera APEX devices. The Nios development board was preloaded with a 32-bit Nios embedded processor system reference design.

The highlight of this project is the efficacious use of interrupts for inter-board communication and the use of numerous custom peripherals for both random number generation and implementing the RSA algorithm and hardware acceleration.

Functional Description

The functional description of this project is depicted through the flow diagram below. It is essentially comprised of two flows. One flow is the generation of the random number using the DES-based random bit generator. See Figure 1.

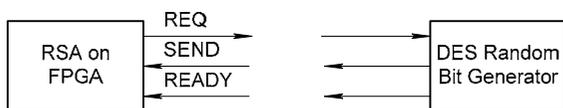
Figure 1. DES-Based Random Bit Generator



The flow diagram for the RSA implementation is as follows:

1. A request is sent from the RSA module to fetch a random byte.
2. On receipt of a request, a random byte is sent by the DES random bit generator that continuously polls a designated port for the request for random bytes.
3. It also signals READY after sending the random byte, and indicates readiness to accept the next request from the device (FPGA running RSA).

Figure 2. Handshaking Between RSA Module & DES Random Bit Generator



Performance Parameters

The performance parameters entail results obtained for the random number generator as well as for the RSA implementation.

Random Number Generation

A comparative overview of the results obtained for all nine designs implemented during the course of this project is tabulated below:

- Resource Requirements
- Session Yield
- Bit Generation Speed/Throughput
- Lines of Verilog HDL Code

Resource Requirements

Number	Item	Logic Elements (LEs)	Pins	Memory Bits	Phase-Locked Loops (PLLs)
1.	Maximum available resources on Altera APEX 20K EP200EFC484-2X	8,320	376	106,496	2
2.	Resources utilized by the standard Nios processor with essential peripherals	2,641	111	26,496	0

Resource Utilization per Design

Number	Design	LEs	Pins	Memory Bits	PLLs
1.	PLL-Based TRBG	4,737	113	26,496	2
2.	Ring Oscillator-Based TRBG	2,772	119	26,496	0
3.	Modified LILI-II PRBG	4,114	111	26,496	0
4.	Nonlinear Combiner Model-Based PRBG	7,419	111	26,496	0
5.	Nonlinear Combiner Model (Enhanced With Memory)	8,312	111	26,496	0
6.	Nonlinear State Filter Model-Based PRBG	8,312	111	26,496	0
7.	DES-Based PRBG	4,969	111	26,496	0
8.	DES-ALFG-Based PRBG	8,043	111	26,496	0
9.	BBS-Based PRBG	6,449	111	26,496	0

Session Yield

Number	Design	Yield per Session
1.	PLL-Based TRBG	362 bits
2.	Ring Oscillator-Based TRBG	> 246 Kbits
3.	Modified LILI-II PRBG	> 40 Kbits
4.	Nonlinear Combiner Model-Based PRBG	> 2 Mbits
5.	Nonlinear Combiner Model (enhanced with memory)-Based PRBG	> 2 Mbits
6.	Nonlinear State Filter Model-Based PRBG	> 40 Kbits
7.	DES-Based PRBG	> 526 Kbits
8.	DES-ALFG-Based PRBG	> 1.87 Mbits
9.	BBS-Based PRBG	> 40 Kbits

Although the session yield of the true random bit generator is based on the PLL and implemented as the Nios peripheral appears to be low, it is conjectured that it will perform far better as a stand-alone device. Moreover, 362 bits per session may be considered adequate for session key initialization vector requirements.

Bit Generation Speed/Throughput

The bit generation speed/throughput shown below has been worked out in terms of the clock cycles taken to generate one random bit, based on the implemented algorithm. Its translation into throughput has been done for a clock speed of 33.3 MHz.

Number	Design	Clock Cycles per Random Bit	Throughput for a Clock at 33.3 MHz
1.	PLL-Based TRBG	< 3	> 11.1 Mbps
2.	Ring Oscillator-Based TRBG	-	0.59 Mbps
3.	Modified LILI-II PRBG	5	6.66 Mbps
4.	Nonlinear Combiner Model-Based PRBG	3	11.1 Mbps
5.	Nonlinear Combiner Model (Enhanced with Memory)-Based PRBG	3	11.1 Mbps
6.	Nonlinear State Filter Model-Based PRBG	4	8.325 Mbps
7.	DES-Based PRBG	140	7.611 Mbps
8.	DES-ALFG-Based PRBG	Initially 140 subsequently amortized to < 140	> 7.611 Mbps
9.	BBS-Based PRBG	520	64.038 Kbps

Lines of Verilog HDL Code

Number	Design	Lines of Verilog HDL Code
1.	PLL-Based TRBG	275
2.	Ring Oscillator-Based TRBG	275
3.	Modified LILI-II PRBG	177
4.	Nonlinear Combiner Model-Based PRBG	189
5.	Nonlinear Combiner Model (Enhanced with Memory)-Based PRBG	322
6.	Nonlinear State Filter Model-Based PRBG	324
7.	DES-Based PRBG	1,143
8.	DES-ALFG-Based PRBG	1,191
9.	BBS-Based PRBG	601

The RSA algorithm was implemented as separate peripherals performing the following operations:

- Random number receiver
- Multiplicative inverse calculator
- Modular exponentiation calculator

After implementing these peripherals, all were combined to form a RSA integrated design working through a C driver program, which passed inputs and outputs between the various peripherals, in order. Due to the paucity of the space on the FPGA in terms of the number of LEs, only a 32-bit RSA integrated algorithm was implemented. Space on FPGA (number of LEs) permitting, this design can easily be scaled up.

Random Number Receiver

The random number receiver was implemented to receive one byte of random number through the external pins on the board. The peripheral consumed the following resources.

Family	APEX 20KE
Device	APEX 20K EP200EFC484-2X
Total LEs	2,783/8,320 (33%)
Total Pins	121/376 (32%)
Total Memory Bits	26,496/106,496 (24%)
Total PLLs	0/2 (0%)

The total time taken for compilation, synthesis, fitting, placement, and routing of this peripheral was 4 minutes and 42 seconds.

Multiplicative Inverse

This peripheral was implemented to compute the secret key using an extended Euclidean algorithm. Since the algorithm implemented required division operations to compute the remainder and quotient at every step, it consumed a lot of resources. In simulation, this algorithm was tried and tested up to 128 bits, but in hardware, it could be implemented only up to 48 bits. Total time taken for compilation and synthesis, fitting, placement, and routing was 12 minutes, 31 seconds. The compilation report for this peripheral was:

Family	APEX 20KE
Device	APEX 20K EP200EFC484-2X
Total LEs	6,524/8,320 (78%)
Total Pins	111/376 (29%)
Total Memory Bits	26,496/106,496 (24%)
Total PLLs	0/2 (0%)

As can be seen from the compilation report, a 48-bit implementation itself consumes 6,524 LEs. Hence, if used along with other peripherals such as the exponentiator and the random number receiver, no other peripheral would be able to fit on the FPGA. Therefore, only a 32-bit implementation was used in the RSA integrated implementation.

Exponentiator

This peripheral was implemented to carry out the following tasks:

- Primality check using Fermat's Theorem
- Encryption
- Decryption

The algorithm implemented was the Montgomery exponentiation algorithm, which in turn uses the Montgomery multiplication algorithm for the intermediate steps. The modular multiplication was implemented using the systolic array architecture, which is quite resource efficient. In simulation, a 512-bit exponentiation was implemented, however, in hardware only a 128-bit exponentiation was possible. The total time taken for compilation, synthesis, fitting, placement, and routing was 11 minutes, 36 seconds. The compilation report for this peripheral was:

Family	APEX 20KE
Device	APEX 20K EP200EFC484-2X
Total LEs	6,971/8,320 (83%)
Total Pins	111/376 (29%)
Total Memory Bits	26,496/106,496 (24%)
Total PLLs	0/2 (0%)

The peripheral consumed 6,971 LEs, hence a higher implementation such as 256- or 512-bit exponentiation was not possible, despite a resource-efficient architecture. The 256-bit exponentiator

itself required 10,277 LEs, while a 512-bit exponentiator required 17,459 LEs. In the RSA integrated implementation, only a 32-bit exponentiator was included, since two other peripherals, the random number receiver and the multiplicative inverse, were also required to be fitted on the same chip.

RSA Integrated

The RSA integrated peripheral implements the complete RSA algorithm primitive, which includes the following operations:

- Receiving random numbers.
- Primality checking
- Computation of multiplicative inverse.
- Computation of modular exponentiation.

All of the above operations were implemented as separate peripherals and fitted on the same chip. A C driver program then interacts with all the peripherals and passes appropriate values between them. This requires that all the peripherals are instantiated correctly in the C program. The total time taken for compilation, synthesis, fitting, placement, and routing was 13 minutes, 8 seconds. The compilation report for this integrated design was:

Family	APEX 20KE
Device	APEX 20K EP200EFC484-2X
Total LEs	6,984/8,320 (84%)
Total Pins	121/376 (32%)
Total Memory Bits	26,496/106,496 (24%)
Total PLLs	0/2 (0%)

The RSA integrated implementation of 48 bits, excluding the random number receiver and the primality checker, consumed 8239 LEs, which is almost 99% of the total available LEs on the board. Hence the final implementation was scaled down to 32 bits to accommodate the random number and the primality check peripherals.

Execution Time & Throughput

The RSA algorithm has been implemented with a modulus of 32 bits, with a multi-board architecture also included to receive the random numbers on the fly. However, this makes the measurement of the execution time difficult since it involves an interrupt-driven mechanism. By simulation, the execution time and the throughput for only the encryption/decryption can be approximated for a clock speed of 33 MHz. In the case of RSA, the encryption and decryption is carried out by modular exponentiation, and for a modulus of 32 bits, it took 1,555 clock cycles, which gave a throughput of 0.68 Mbps.

Design Architecture

The system architecture entails two parts, namely:

- Generation of random numbers using the DES random bit generator

- Implementation of RSA using the random number generated by the above method

PRBG Based on Block Cipher Techniques

This section describes how the random numbers were generated.

DES Random Bit Generator

The data encryption standard (DES) was developed by an IBM team around 1974 and adopted as a U.S. national standard in 1977. Since that time, many cryptanalysts have attempted to find shortcuts for breaking the system. It is defined by the American standard FIPS 46-2. We wish to encipher a 64-bit plaintext message block under the 56-bit key, to produce a 64-bit ciphertext message block $c = E_k(m)$. Decipherment, or recovering plaintext from ciphertext, is denoted $m = D_k(c)$. The plaintext message block m is subjected to an initial permutation P , and the result is broken into two 32-bit message halves, m_0 and m_1 . Intermediate message halves, m_2, \dots, m_{17} are then created in sixteen rounds according to the procedure described below. Finally, the 64-bit ciphertext c is generated by applying the inverse permutation IP^{-1} to the two message halves m_{17} and m_{16} .)

The plain text and intermediate message halves $m_{\{0\}}, m_{\{1\}}, m_{\{1\}}, \dots, m_{\{17\}}$ are related as follows:

$$m_{\{i+1\}} = m_{\{i-1\}} \text{ XOR } f(k_{\{i\}}, m_{\{i\}}), \quad i = 1, 2, \dots, 16$$

Here k is the secret 56-bit key and i is the number of the round (from 1 through 16). Also, $k_{\{i\}}$ (round key) is a selection of 48 bits from the 56 bits of k . This selection, or key schedule, depends on the round number i . Now we describe the substitution function f . There are eight S-boxes, $S_{\{1\}}, \dots, S_{\{8\}}$ described in the standard. Each S-box is a table lookup, using six bits as input and providing four bits as output. For each S-box, say $S_{\{j\}}$, six consecutive bits are selected from the 48 bits of $k_{\{i\}}$, namely bits $6j - 5, 6j - 4, \dots, 6j$. Also, six consecutive bits are selected from $m_{\{i\}}$, namely bits $(4j - 4, 4j - 3, \dots, 4j + 1) \bmod 32$. The mod 32 is shorthand for the convention that for $j = 1$, the bits are 32, 1, 2, 3, 4, 5, and for $j = 8$ the bits are 28, 29, 30, 31, 32, 1. Two adjacent S-boxes share two message bits. For instance, $S_{\{1\}}$ uses message bits 32, 1, 2, 3, 4, 5, while $S_{\{2\}}$, uses message bits 4, 5, 6, 7, 8, 9, and they share bits 4 and 5. (Key bits are not shared among S-boxes on one round.) $S_{\{8\}}$, and $S_{\{1\}}$, are considered to be adjacent because they share message bits 32 and 1. The six key bits and the six message bits are XORed together bitwise, and the resulting six bits are used as input for a table lookup.

That is, the six inputs to S-box $S_{\{j\}}$ at round $\{i\}$ are:

$$m_{\{i\}}[4j - 4] \text{ XOR } k_{\{i\}}[6j - 5],$$

$$m_{\{i\}}[4j - 3] \text{ XOR } k_{\{i\}}[6j - 4],$$

$$m_{\{i\}}[4j + 1] \text{ XOR } k_{\{i\}}[6j]$$

or, written another way,

$$[4j - 4, 4j - 3, 4j - 2, 4j - 1, 4j, 4j + 1] \text{ XOR } k_{\{i\}}[6j - 5, 6j - 4, 6j - 3, 6j - 2, 6j - 1, 6j].$$

Each of the eight S-boxes implements a different table, each with 26 entries of four bits each. These tables are described in the standard. The eight S-boxes together put out $8 \times 4 = 32$ bits. These bits are permuted according to a permutation P that is fixed for all rounds i . The resulting 32-bit quantity is the value of $f(k_{\{i\}}, m_{\{i\}})$.

In summary, the 64-bit message undergoes a permutation IP to produce two 32-bit message halves $m\{0\}$ and $m\{1\}$. Then we compute the 32-bit quantity $f(k\{1\}, m\{1\})$ and XOR that quantity with $m\{0\}$, to produce $m\{2\}$. We use this new quantity $m\{2\}$ to compute $f(k\{2\}, m\{2\})$ and XOR that quantity with $m\{1\}$, to produce $m\{3\}$. We continue in a like fashion until $m\{16\}$ and $m\{17\}$ have been computed. These two message halves are interchanged and then subjected to the permutation IP^{-1} , to produce the ciphertext c . Decryption is easily accomplished by a user in possession of the same key k . First, one applies the permutation IP to c , to produce the message halves $m\{17\}$ and $m\{16\}$. Next, one computes $f(k\{16\}, m\{16\})$ and XORs that quantity with $m\{17\}$ to recover $m\{15\}$. Recalling that $m\{17\} = m\{15\} \text{ XOR } f(k\{16\}, m\{16\})$, we have $m\{17\} \text{ XOR } f(k\{16\}, m\{16\}) = [m\{15\} \text{ XOR } f(k\{16\}, m\{16\}) \text{ XOR } f(k\{16\}, m\{16\})] = m\{15\}$, because of the identity $(A \text{ XOR } B) \text{ XOR } B = A$. Similarly, one computes $m\{14\} = m\{16\} \text{ XOR } f(k\{15\}, m\{15\})$, and continues in like fashion until one has computed $m\{1\}$ and $m\{0\}$. Applying IP^{-1} to the pair $(m\{0\}, m\{1\})$, one recovers the plaintext message m .

DES RBG Design

DES was implemented in ECB mode for any arbitrarily selected IV, using a secret key. The ciphertext emerging after each round of encryption was thereafter used as the key for the next round of encryption, while simultaneously incrementing the plaintext once in counter mode. This fundamental operation is iteratively executed for the desired number of times, with an interrupt being raised after each execution. The random bits are read back by the driver as 32-bit words, and the next iteration by the hardware is triggered as per the interrupt service routine.

Implementation Details

The design entry was created in Verilog HDL. Quartus II software was used for the compilation, analysis, synthesis, fitting, assembling, and timing analysis. The random bit generator was designed as a peripheral device to the embedded Nios processor. A device driver written in C was used to control the peripheral device. Further, the DES RBG was adapted to serve as the random bit generator for the RSA implementation created by a colleague. The DES RBG continuously polls a designated port for the request for random bytes. On receipt of the same, it generates a 64-bit word of random bits and sends the lower order byte to the requesting device. It signals "ready" after doing so, indicating readiness to accept the next request. The implementation performed as a multi-board design and the checking for primality of the generated random number is done at the distant end.

The statistical performance of both generators fails to impress. The DES-ALFG generator is an absolute flop, while the DES generator is scarcely much better against this benchmark. At a pinch, the DES generator could be used—in spite of a little bias in its output, it exhibits no periodicity—but the ALFG as a primitive for cryptographic random number generation does not pass any statistical test other than block-frequency. In summary, the block cipher-based approach, for the primitives selected, has yielded disappointing results.

RSA Design

The design for RSA includes the design for random number generation, multiplicative inverse, and modular exponentiation.

Random Number Generation

The RSA algorithm requires that two random prime numbers of $n/2$ bits be generated, where n is the number of bits in the modulus. These random prime numbers are then tested for primality before they are used in the algorithm proper. Since there is a separate project on `\emph{Random Number Generation}` being implemented, no random numbers were generated as part of this project. However, a peripheral module to receive the random numbers generated by an external program on a different board was implemented and incorporated in the main RSA algorithm.

Architecture of Random Number Receiver

The project has been implemented on Altera's APEX 20K EP200EFC484-2X board, which has a space limitation as far as the number of LEs is concerned. Also, the board has been manufactured in such a way that it does not permit daisy-chaining architecture to overcome the above limitation. Hence, the only method available is to use the external pins on the board, connect those to another board, and exchange data between the two. This, however, has certain limitations, such as the numbers of bits that can be exchanged, the timing issues between the two independent programs, and the requirement of exchanging signals between the boards to facilitate communication as per specific requirements. A multi-board architecture was realized to exchange data between two boards connected through external pins. Due to the limitations mentioned above, a peripheral module for handling random numbers of 16 bits each was implemented. This design is completely scalable and, hardware permitting, can receive any number of bits from another board.

This peripheral module has the following components:

- Random number receiver module
- Driver program, which receives the random numbers from the random number receiver module
- Primality check module, based on Fermat's Theorem and utilizing the exponentiator peripheral

Random Number Receiver Module

This is a module written in Verilog HDL and it resides on the hardware (FPGA). To receive the random numbers and to communicate with another board, 10 external pins have been mapped with this module. On eight of these external pins, the module receives the random numbers, one byte at a time. Of the other two pins, one is used to send a `\emph{start}` signal to the other board and the other to receive the `\emph{done}` signal from it. A common ground is necessary for this type of data exchange. On receiving the `\emph{done}` signal from the second board, this module transfers the byte received on the external pins, first to an internal register and thereafter to the driver program. After sending that byte to the driver program, it is ready to receive the next byte. The number of bytes to be received can be set at the beginning of the data exchange. On completion, it hands over control to the driver program for further processing of these random numbers received.

Random Number Receiver Block Diagram

The random number receiver has been implemented as a peripheral and shown in the figure given below. The block diagram also shows the random number generator peripheral implemented on a different board. Both these peripherals exchange data and signals through the external pins of the Altera board. As explained earlier, these pins have been mapped on to the inputs and outputs of the peripherals in the FPGA.

Driver Program

This program has been written in C and it interacts with the random number receiver module through the Nios processor. With each hardware interrupt, it activates its hardware handler subroutine and captures the byte sent into an array. It then combines two bytes at random and then sends it to the primality check module. If the primality check is positive, this driver program stores that 16-bit random prime number to be used subsequently in the RSA algorithm, else it discards that number. The same process is repeated until it gets at least two prime numbers of 16 bits each. These two prime numbers eventually make p and q for the RSA algorithm. After obtaining p and q , it also computes $n = pq$, which is the modulus, and $\phi = (p-1)(q-1)$, which is $\phi(n)$.

Primality Check Module

This module is based on the Fermat's Theorem, which states that for any integer a , and any prime number n , if n is prime then

$$a^{\{n\}} \bmod n = a$$

If $a^{\{n\}} \bmod n, n \neq a$, then n is not prime. By testing sufficient number of a 's, all composite a 's can be excluded and all primes can be retained. Another variation of Fermat's Theorem that can also be utilized to carry out a primality check is Euler's Theorem. It states that, if a is any integer and p is prime, such that $\gcd(p,a) = 1$, then

$$a^{\{p-1\}} \bmod p = 1$$

This is possible only if p is prime. The existing modular exponentiation architecture can be utilized to carry out the exponentiation required by Fermat's theorem or Euler's theorem to determine whether the number is prime or not. If the number is prime, then, the driver program retains that number to be further handed over to the main RSA driver routine.

Multiplicative Inverse

The multiplicative inverse of a number, over a modulus, is computed based on the Extended Euclidean algorithm. The algorithm needs to do integer division twice for that which the module `calModulus` makes use of. This is by far the most time consuming, as well as resource consuming, operation in RSA. The Altera APEX 20K EP200EFC484-2X board is able to accommodate the algorithm for computing the multiplicative inverse only up to 48 bits. The design incorporates two modules:

- Extended Euclidean module
- Modulus

Extended Euclidean Module

This is the top-level module, which takes as input the value of exponent e and the value of ϕ . Based on the value of e , it goes through the various steps of the Extended Euclidean algorithm. For each step, it sends the dividend and divisor values to the modulus for performing the integer division. The modulus returns the remainder and quotient after the division operation. Finally, the inverse value is returned after ascertaining that the last non-zero remainder is one, and the algorithm is executed for two steps beyond the Euclidean algorithm.

Modulus

This module is based on the non-restoring division method of calculating the modulo. It takes two inputs, the dividend and the divisor. After division, it returns the remainder and quotient back to the Extended Euclidean module. The multiplicative inverse computed by this peripheral is based on the value of ϕ generated, as well as the value of exponent e chosen. The value of e chosen is actually the public key and the multiplicative inverse computed is the secret key or d . This value of d is then used during the decryption phase for computing the original plaintext.

The module for computing the multiplicative inverse has been implemented as a peripheral on the FPGA. The driver program sends the exponent value and the ϕ value to this peripheral through the Nios processor. The peripheral computes the secret key or the inverse value of the exponent with respect to ϕ and returns it via the Nios processor to the driver program.

Modular Exponentiation

An architecture for modular exponentiation proposed by Thomas Blum and Christof Paar was chosen for implementation. It is based on the Montgomery exponentiation and Montgomery modular multiplication for radix 2. It is a resource-efficient architecture suitable for implementation in FPGAs. Its design is based on an exponentiator, which handles the exponentiation and feeds values to a systolic array that computes the modular multiplication. The architecture essentially consists of two basic units, the exponentiator and the systolic array.

Exponentiator

This is the top-level module and is based on the Montgomery exponentiation algorithm. It takes as input the following parameters:

- Modulus m
- Message x
- Exponent e
- Number of bits in exponent
- Precomputation factor $R^2 \bmod m$

The precomputation factor and A are fed as inputs so that all values in the intermediate stages of exponentiation are in Montgomery domain carrying a factor of 2^{n+2} , where n is the number of bits in the modulus. This module first feeds the values of x and $R^2 \bmod m$ to the systolic array for computation of \widetilde{x} . Thereafter, it first checks the exponent bit and then feeds appropriate values to the systolic array for multiplication. At the end it feeds the result and value 1 again to the systolic array to obtain the final result, thereby getting rid of the additional factor of 2^{n+2} . The final result so obtained is either the ciphertext or the plaintext depending upon whether it is encryption or decryption. In case of encryption, the exponent used is 65537, while in the case of decryption it is the secret key or d computed as the multiplicative inverse earlier.

Systolic Array

The systolic array computes the modular multiplication based on the Montgomery modular multiplication algorithm. A systolic system comprises a set of interconnected cells, each capable of performing a specified operation. The cells and operations performed by them are usually identical. The time taken for processing by each of the cells is identical. Individual cells are connected only to their nearest neighbors. The flow of data between the cells is rhythmic and regular. Except those at the boundary of the array, the cells do not communicate with the outside world. Systolic architectures are essentially suited for implementing computationally bound operations. The following arithmetic operation is required to be implemented.

$$S_{i+1} = (S_i + q_i M) / 2 + a_i B, \quad q_i, a_i \in \{0, 1\}$$

The above equation can be modified into

$$S_{i+1} = (S_i + q_i M + 2a_i B) / 2, \quad q_i, a_i \in \{0, 1\}$$

Instead of using two adders for computing the addition required in the above step, the sum $2B + M$ is precomputed and stored in a register. A single adder is sufficient to add 0, $2B$, M or $2B + M$ to S_i , depending on the values of a_i and q_i . The same adder can also be used to precompute $2B + M$. The systolic array has the following inputs and outputs:

Inputs:

- `modulus`: Modulus value sent by exponentiator
- `var1`: First variable to be multiplied
- `var2`: Second variable to be multiplied
- `clk`: Clock for synchronization
- `reset`: Reset signal

Outputs:

- `result`: Result of the modular multiplication

All the processing elements get instantiated in this module. The signals and the data exchanged between the processing elements are declared as wires in the systolic array module. During each clock cycle, each processing element computes u bits of $S_{i+1} = (S_i + q_iM + 2a_iB)/2$. Now, during the clock cycle I , the processing element1 computes u bits of S_i . During clock cycle $i+1$ processing element2 computes the next u bits of S_i . To compute u bits of S_{i+1} the processing element1 requires the LS bit of S_i , computed by processing element2. This is on account of the division by two required in step $S_{i+1} = (S_i + q_iM + 2a_iB)/2$. Thus, clock cycle $i+1$ is unproductive for processing element1. Therefore, each unit of the systolic array stays idle in every alternate clock cycle. To achieve parallelism each processing element computations during one cycle and remains idle in the next clock cycle. This alternate clock cycle computation ensures parallelism and a complete utilization of all its units.

To compute $S_{i+1} = (S_i + q_iM + 2a_iB)/2$ where $M = \sum_{i=1}^{n-1} \{m_i2^i\}$, $m_i \{0,1\}$ and $B = \sum_{i=1}^n \{b_i2^i\}$, $b_i \{0,1\}$, $n/u + 1$ units are needed. The unit $n/u + 1$ is used to process the most significant bit of part b and has no part mod inputs. There are two buses each for loading the M and B . The M even bus and B even bus are connected to units processing element1, processing element3,processing element($n/u + 1$). The M odd bus and B odd bus are connected to units processing element2, processing element4, processing element(n/u)

The `s1_out` output of processing element1 is connected back to its input. This is required for subsequent passes through the loop. The carry generated in each addition is also propagated to units in the left through the use of `c_out` and `c_in` pins. The `s0_out` is connected to the `s0_in` in the pin to the right in order to send the LS bit of the left shifted S_i (division by 2). The result bits are pumped backwards to `processing_element1` through the use of the `res_out` and `res_in` pins as only processing element1 is connected to external modules.

Operation:

1. Initially, the values M and $2B$ are fed to all the units and saved in registers.
2. The computation of $S_{i+1} = (S_i + q_iM + 2a_iB)/2$ begins by initialization followed by giving the a_i input at the `a_i_in` pin of processing_element1.
3. At the end of computation, the result is pumped across the units to processing_element1.

Processing Element

At the heart of the processing element is a u bit Adder. The result of the adder is latched into S_Reg (including the carry). An extra S_Reg_2 is required to introduce the one clock delay before the result is fed back as input for the next pass through the loop after it is left-shifted. The LS bit of the shifted input comes from the neighboring unit from $S0_In$ pin. Registers B_Reg , M_Reg , and $BMSum_Reg$ are used to save $2B$, M , and $B + M$. Multiplexer Mux_B is used to selectively input B_Reg with the B_In or the S_Reg . Multiplexers Mux_1 and Mux_2 select the appropriate inputs for the adder. Mux_Res selectively outputs the result of the adder or $Result_In$ from the neighbor. $Control_Reg$ and qa_Reg are used to latch the values before passing it along to their neighbors. Each processing element of the systolic array computes u bits of modular multiplication. Each processing element has the following inputs and outputs.

Inputs:

- res_in : Result bits from the (left) neighboring unit
- qa_in : q_i, a_i bits
- c_in : Carry bit from (right) neighbor
- $s0_in$: LS bit of S_I from (left) neighbor required on account of division by 2 of S_i
- $part_b$: u bits of $2B$ fed externally at the start for saving and precomputation of $2B + M$
- $part_mod$: u bits of M fed externally at the start for saving and precomputation of $2B + M$
- clk : Clock signal for synchronization
- $reset$: Reset signal
- $start$: Start signal from exponentiator to begin computation
- $flush$: Flush signal from the exponentiator to flush all the registers before the next multiplication

Outputs:

- res_out : Result bits computed by the unit
- $s0_out$: LS bit of result for the (right) neighbor
- qa_out : q_i, a_i bits for the (left) neighbor
- c_out : Carry bit generated by the addition
- $s1_out$: 2nd LSB required as q_i input for unit 1, can be taken from the LSB of res_out , also
- $next$: Next signal for the neighbor to start computation

Modular Exponentiator Block Diagram

The modular exponentiation has been implemented as a peripheral comprising all of the modules mentioned above. The top-level module in this peripheral receives the exponent value, modulus value,

correction factor, message, and the number of bits in the exponent from the driver program via the Nios processor in 32 bits each. The peripheral then computes the value of the exponentiation and returns it back to the driver program.

Design Methodology

RSA Implementation

Altera's APEX 20KE FPGA family was chosen for implementing the RSA algorithm. APEX devices are high-density FPGAs that allow complex designs to be implemented on a single device. The target device was an EP20K200EFC484-2X. The design files were written in Verilog HDL, while compilation, synthesis, fitting, placement, and routing were carried out using Quartus II software.

Design Flow

The complete implementation of the RSA in FPGA was performed in the following stages:

4. Design entry
5. Compilation and synthesis
6. Fitting, placement, and routing
7. Interaction with the C Driver program

Design Entry

The designs for the project were specified by using the Verilog HDL. The Verilog HDL files are essentially the source files, giving the structural description of each of the sub-units.

Random Receiver

This contains the design file `random_receiver.v`, which receives the random numbers on the output pins, generated on the other board. A total of 10 external pins were used to collect the random numbers one byte at a time. The balance of the two pins was used for synchronization purposes. This Verilog file contains the mechanism of raising hardware interrupts and throwing out the byte received to the driver program for further processing.

Multiplicative Inverse

This contains the following design files:

- *calModulus.v*—This module performs the division operation, given the dividend and divisor, and returns the remainder and quotient after the division operation. The size of the inputs and outputs of this module are parameterized to facilitate easy scalability.
- *topInverse.v*—This module implements the extended Euclidean algorithm for calculating the multiplicative inverse. It instantiates the `calModulus.v` module for performing the division operation. The inputs and outputs of this module are also parameterized.

Modular Exponentiation

This contains the following design files:

- *processing_element.v*—This gives the structural description of the processing element of the systolic array. The word size of the processing element is parameterized and can be altered. Each processing element computes the sum as per the algorithm.
- *systolic_array.v*—This module instantiates a series of processing elements and specifies the interconnections between them in terms of inputs and outputs. It returns the result of a multiplication to the exponentiator module, based on the Montgomery modular multiplication algorithm.
- *monty_expo.v*—This is the top-level module that implements the Montgomery exponentiation algorithm as a series of modular multiplications with the help of the underlying systolic array module.

Compilation & Synthesis

The design files form the input to the compilation and synthesis tool (i.e., Quartus II development software). The design files are first included in the project **standard_32** directory within Quartus II software. Thereafter, a new peripheral is created for each top-level module with the help of the SOPC builder. The SOPC builder is then generated to build the user-defined peripherals along with the design files of the **standard_32** directory. The operating frequency and the target devices are selected at the time of opening a new project. Finally, the whole project is compiled and synthesized.

Fitting, Placement & Routing

Quartus II development software is also used for this purpose. The netlist file generated during the compilation and synthesis forms the input to it. The fitter in Quartus II software assigns each logic function to the best logic cell location for routing and timing. It also selects appropriate interconnection paths and pin assignments. The final output is the **standard.sof** file, which contains the complete routed application.

Interaction with C Driver Program

The design files implemented in the hardware are actually peripherals to the Nios processor and work through the Avalon[®] bus signals. To write/read data to/from the peripheral, a C driver program is used. This C program is loaded in `\cpu_sdk\src` project subdirectory within **standard_32**. The `nios_build` and `nios_run` utilities are then used to compile the C program and run it on the design files already downloaded to the FPGA. The C program includes the `nios.h`, which in turn includes all the header files required for compilation. Also, the peripheral created in the SOPC builder is instantiated in the C program along with its IRQ number. The handler function in the C program then performs the functions mentioned inside the handler in the event of the peripheral raising an interrupt. The data is written to the peripheral through the `writedata` Avalon signal while the reading of data from the peripheral is done through `readdata`. Both `writedata` and `readdata` work for specific addresses that need to be mentioned in the C program.

Implementation Issues

This section describes the implementation issues for this project.

Use of External Pins

For peripherals involving use of external pins, the additional pins used are marked as export, before generation in the SOPC Builder. After generation, physical assignment of each and every pin is carried

out using the assignment editor within Quartus II software. The external pins to be assigned are selected through the Nios development manual. The balance of the operations is similar to that described in earlier sections. This configuration and implementation was carried out for the random_receiver module and peripheral.

16-Bit Implementation

Handling large numbers (1024 bit) makes debugging and functional verification very difficult. Also, the time taken by the software tools, especially the placement and routing (fitter) and simulator, is extremely high. Therefore, a 16-bit exponentiate was built and tested thoroughly as a first step. The exponentiate was then scaled up from 16 bits to 512 bits.

Modular Design

The design of the exponentiate is modular with the processing element and the systolic array being independently implemented and tested. Finally the modules were then integrated together and tested. Same is the case for the other peripherals like multiplicative inverse and the random number receiver. The peripherals have been designed in such a manner that all inputs and outputs are parameterized and can be changed easily without affecting any other part of the module.

Design Scalability

The exponentiator and the multiplicative inverse peripherals scale linearly and therefore require little effort.

Testing & Verification

The test cases for testing were given using the C driver program to the Verilog HDL design file and then reading back the results in the driver program. Initial simulation and testing was carried out using iverilog, being faster. The testing and verification in the hardware takes time owing to time taken for compilation, synthesis, fitting, placement, and routing by the Quartus II software.

Processing Time

An important issue associated with the implementation is the processing time associated with Quartus II software. For the exponentiator, multiplicative inverse, and the random number receiver, the time taken for compilation, synthesis, and fitting is about 12 to 15 minutes.

Software Implementation

A software implementation of modular exponentiation algorithm, multiplicative inverse, modular multiplication, generation of random numbers, and multiplication of large integers was implemented in C and Java to verify the correctness of the results obtained. The Montgomery multiplication algorithm was also implemented to verify the correctness of the intermediate results during exponentiation. This was necessary since the intermediate results carry the additional factor of $2^{\{n+2\}}$ at each stage.

Design Features

The highlights of our design features that we implemented were:

- Interboard communication between two Nios processors using interrupts. This entailed interrupt handling.
- Use of peripherals around the Nios core. This facilitated quick prototyping at the design and trial stage.

- The most significant advantage that is accrued by modeling the design as an Avalon bus peripheral of the Nios processor is that Altera allows control of the input to and the output from the peripherals through a device driver that can be written in C language. This fact allows verification of the cryptographic algorithm once burnt into the hardware, even after simulation is complete.

Conclusion

The entire course of this design was a period of cumulative learning and enrichment of our knowledge regarding the Nios processor and the FPGA. The most satisfying part of this project was the multi-board architecture implemented to make use of two boards simultaneously and realizing an asynchronous system. The use of minimal pins and LEs (as discussed in Part III) to achieve this cryptographic algorithm was one of the achievements of this project. Coupled to this was the fact that a hardware acceleration was achievable, as was hardware reusability.

Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights.