FPGA based Communication Security for Wireless Sensor Networks

Engr. Junaid Majeed
Lecturer, FEST Hamdard University

Abstract

Networks of wireless microsensors for monitoring physical environments have emerged as an important new application area for wireless technology. Key attributes of these new types of networked systems are the severely constrained computational and energy resources and an ad hoc operational environment. Information security is increasingly becoming very important. Encryption and Decryption are very likely to be in many systems that exchange information to secure, verify, or authenticate data. Many systems, like the internet, cellular phones, handheld devices, and E-commerce, involve private and important information exchange and they need cryptography to make it secure. This paper is a study of the communication security aspects of these networks. Resource limitations and specific architecture of sensor networks call for customized security mechanisms. There are three possible solutions to accomplish the cryptographic computation: Software, hardware using application-specific integrated circuits (ASICs), and Hardware using field-programmable gate arrays (FPGAs). The software solution is the cheapest and most flexible one. But, it is the slowest. The ASICs solution is the fastest. But, it is inflexible, very expensive, and needs long development time. The FPGAs solution is flexible, fast, and needs shorter development time.

Elliptic curve cryptography (ECC) needs modular multiplication. Montgomery multiplication algorithm is a very smart and efficient algorithm for calculating the modular multiplication. It replaces the division by a shift and modulus-addition (if needed) operation, which are much faster. The algorithm is also very suitable for a hardware implementation. Many designs have been proposed for fixed precision operands. This scalable Montgomery multiplier can be configured to meet the design area-time tradeoff. Also, it can work for any operand precision up to maximum design memory capability.

1. Introduction

Wireless sensor networks, applied to monitoring physical environments, have recently emerged as an important application resulting from the fusion of wireless communications and embedded computing technologies [1][3][7][10][11]. Sensor networks consist of hundred or thousands of sensor nodes, low power devices equipped with one or more sensors. Besides sensors, a sensor node typically contains signal processing circuits, microcontrollers, and a wireless transmitter/receiver. By feeding information about the physical world into the existing information infrastructure, these networks are expected to lead to a future where computing is closely coupled with the physical world and is even used to affect the physical world via actuators. Potential applications include monitoring remote or inhospitable locations, target tracking in battlefields, disaster relief networks, early fire detection in forests, and environmental monitoring. While recent research has focused on energy efficiency [8], network protocols [4], and distributed databases, there is much less attention given to security. The only work that we are aware of is [5]. However, in many applications the security aspects are as important as performance and low energy consumption. Besides the battlefield applications, security is critical in premise security and surveillance and in sensors in critical systems such as airports, hospitals, etc. Sensor networks have distinctive features, the most important ones being constrained energy and computational resources. To accommodate those differences existing security mechanisms must be adapted or new ones created.

Our approach to communication security in sensor networks is based on a principle stated in [6] that says that data items must be protected to a degree consistent with their value.

Our main goal is to minimize security related energy consumption. By offering a range of security levels we ensure that the scarce resources of sensor nodes are used accordingly to required protection levels. There are many other important issues for security in sensor networks, e.g. physical protection of the sensitive data in sensor nodes, and the system-level security. However, those topics are outside of the scope of this paper. The complexity of building tamper-proof circuits that could protect sensitive information held in a node is described in [2].

1.1 Security Threats

Wireless networks, in general, are more vulnerable to security attacks than wired networks, due to the broadcast nature of the transmission medium. Furthermore, wireless sensor networks have an additional vulnerability because nodes are often placed in a hostile or dangerous environment where they are not physically protected.

We list the possible threats to a network if communication security is compromised:

1. Insertion of malicious code is the most dangerous attack that can occur. Malicious code injected in the network could spread to all nodes, potentially destroying the whole network, or even worse, taking over the network on behalf of an adversary. A seized sensor network can either send false observations about the environment to a legitimate user or send observations about the monitored area to a malicious user.
2. Interception of the messages containing the physical locations of sensor nodes allows an attacker to locate the nodes and destroy them. The significance of hiding the location information from an attacker lies in the fact that the sensor nodes have small dimensions and their location cannot be trivially traced. Thus, it is important to hide the locations of the nodes. In the case of static nodes, the location information does not age and must be protected through the lifetime of the network.
3. Besides the locations of sensor nodes, an adversary can observe the application specific content of messages including message IDs, timestamps and other fields. Confidentiality of those fields in our example application is less important then confidentiality of location information, because the application specific

data does not contain sensitive information, and the lifetime of such data is significantly shorter.

4. An adversary can inject false messages that give incorrect information about the environment to the user. Such messages also consume the scarce energy resources of the nodes. This type of attack is called *sleep deprivation torture* in [9].

1.2 Modular Multiplier

Modular Multiplication is a time-consuming arithmetic operation because it involves multiplication as well as division. Modular exponentiation can be performed as a sequence of modular multiplications. Speeding the modular multiplication will have a great impact on the speed of modular exponentiation. Modular exponentiation and modular multiplication are heavily used in current cryptographic systems. Well known cryptographic algorithms, such as RSA [16] and Diffie-Hellman key exchange [15], require modular exponentiation operations. Digital Signature Standard (DSS) cryptography [17] as well as Elliptic curve cryptography (ECC) [18] need modular multiplication.

Montgomery multiplication algorithm [12] is a very smart and efficient algorithm for calculating the modular multiplication. It replaces the division by a shift and modulus-addition (if needed) operation, which are much faster. The algorithm is also very suitable for a hardware implementation. Many designs have been proposed for fixed –precision operands. A word-based Montgomery multiplication algorithm [13] has been proposed later and the scalable Montgomery multiplier is based on this modified algorithm. This multiplier can be configured to meet the design area-time tradeoff. Also, it can work for any operand precision up to the maximum design memory capability.

1.2.1. Montgomery Multiplication (MM) Algorithm

Before we describe the Montgomery multiplication algorithm, we will introduce the following definitions and notations.

- M is the modulus of the modular multiplication
- X is the multiplier of the modular multiplication
- Xi is a single bit of X at position i
- Y is the multiplicand of the modular multiplication
- N is the number of bits in each operand
- r is a constant equal to 2^N
- S is the partial product of the modular multiplication
- Si is a single bit of S at position i

The Montgomery multiplication algorithm calculates the following result

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

Where $r = 2^N$ and M is an integer in the range $2^{N-1} \le M \le 2^N-1$ such that gcd(r, M) = 1.

The algorithm transforms an integer in the range [0, M-1] to another integer in the same range called the image or the M-residue of the integer.

The following steps show how Modular multiplication can be calculated using a series of Montgomery multiplication

1. Images of X and Y are calculated as

$\bar{X} = MM(x, r^2) = Xr \bmod M$

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

2. Image of C is calculated as

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

3. Modular multiplication is calculated as

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

Figure 1.1 shows the three steps mentioned above.

Algo1.1 shows the radix-2 MM algorithm. The algorithm –as we said before- is very smart since it replaces the division by shifting and addition, which are faster and more efficient. The partial product S is initialized to zero. Then, for each iteration in the algorithm, a bit of the multiplier X is multiplied by the multiplicand Y.

The X bits are scanned one bit at a time starting from the least significant bit. If the partial sum S is odd then M needs to be added. This makes the partial sum even because M is odd since gcd(r, M) = 1. Adding M will not affect the modular multiplication results because it is the modulus. S is then shifted one bit position to the right. After all the iterations are executed, S is compared with M to see if it is outside the range [0,M-1] (i.e., if it greater than M). If so then M is subtracted from S to have it in the range. By the end, S will hold the Montgomery multiplication result of X and Y.

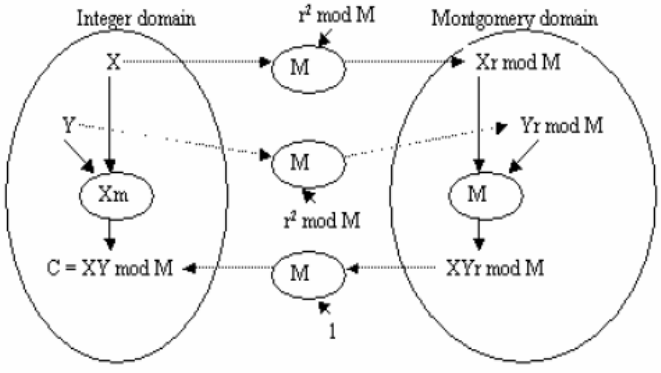

Figure1.1. Modular Multiplication using MM

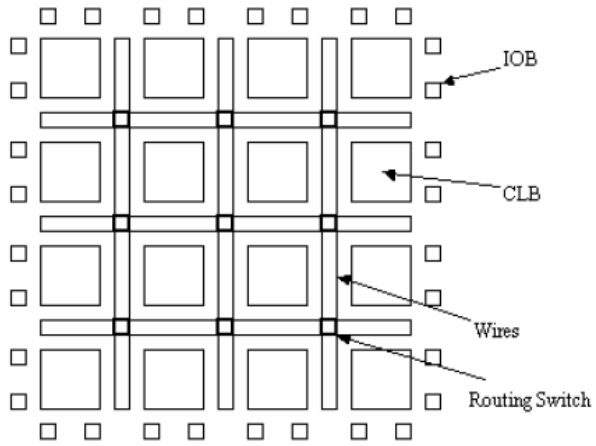

Figure 1.2. FPGA general structure

Step

1. S = 0
2. for i = 0 to N-1

S = S + Xi*Y

if S is ODD then S = S + M

S = S/2

3. If S ≥ M then S = S – M

Algo. 1.1. Radix-2 MM algorithm

1.3. Field Programmable Gate Arrays (FPGAs)

Field programmable gate arrays (FPGAs) are programmable chips. Figure 1.2 shows a general structure of an FPGA chip. The

FPGA is an array of configurable logic blocks (CLBs). It has also input/output blocks to provide the interface between the chip pins and the internal signals. The signals from all blocks are connected to each other using wires, which in turn connected to each other by programmable routing switches. The CLBs have the logic resources that are necessary to implement various combinational and sequential logic functions. Normally, a CLB has look-up tables (LUTs), multiplexers, and flip-flops. The programming of all resources (CLBs, IOBs, and routing switches) is done using RAM, EPROM, EEPROM, or Anti-fuse technologies.
For our work, we are using Xilinx Spartan-II FPGAs, which are programmed using RAM technology.

1.4. Literature Review for Montgomery Multiplication

A systolic array design for modular multiplication base on Montgomery algorithm has been presented in [22]. The design can generate one modular multiplication every clock cycle with latency equal to 2N+2 cycles, where N is the operand size. This design is useful when consecutive modular multiplications are needed like in RSA cryptography.
A radix-2 word-based Montgomery multiplication algorithm and a scalable Montgomery multiplication architecture based on it have been presented in [13]. The scalable Montgomery multiplier has no limit on the operand size since it processes the operands word by word. Also, it exploits the parallelism in the algorithm by using multiple processing elements in a pipelined fashion. The scalable multiplier can be configured by selecting the word size and the number of processing elements in the pipeline that best meet the area and time requirements of the system. ASICs designs, implementations and analysis of one radix-2 and two radix-8 scalable Montgomery multipliers have been presented in [19]. The radix-2 design has been based on the algorithm presented in [13]. The radix-8 designs have been based on high radix word-based Montgomery multiplication algorithm developed also in [19] based on the algorithm in [13]. The study shows that after some number of processing elements, adding more elements will increase the total execution time. The study also was trying to optimize for two operand sizes at the same time.
FPGAs that are in-system programmable were investigated in [21] to find the key architectural criteria for implementing high performance wide-operand addition. FPGA architecture for high performance wide-operand modular multiplication has been also proposed. This architecture is intended for high performance cryptography.
A modular exponentiation architecture that combines high radix Montgomery multiplication with systolic array was derived in [20]. This design performs 1024-bit RSA operation in 3.1 ms using 45.6 MHz clock frequency. This design was implemented on one Xilinx XC40250XV FPGA.

2. Design

To implement any design on an FPGA chip, the designer should be aware of the design development tools (i.e., the CAD tools) and the target FPGA technology. An ASIC design that is efficient in terms of area and/or speed for some ASIC tools and technology is not necessarily efficient for some FPGA tools and technology. Same thing applies when considering tools and technologies from different vendors. What is efficient for Xilinx FPGAs might not be efficient for Altera FPGAs. Even this applies to different tools and technologies from the same vendor. For example, a design that is implemented using Foundation 2.1i tools from Xilinx and efficient for the XC4000 FPGAs might not be efficient when using Xilinx ISE4.1 tools and Spartan-II FPGAs as the target technology. So, the key is to understand how to let the tools interpret the design description efficiently and optimize it as much as possible. Also, to understand the target FPGA chip and make good use of its resources.

2.1. Xilinx Spartan-II FPGAs

Spartan-II FPGA is made mainly of five kinds of elements: Input/Output blocks (IOBs), Configurable logic blocks (CLBs), block random-access memories (Block RAMs), Delay-locked loops (DLLs), and versatile multi-level interconnect structure. A block diagram of Spartan-II FPGA is shown in Figure 2.1.

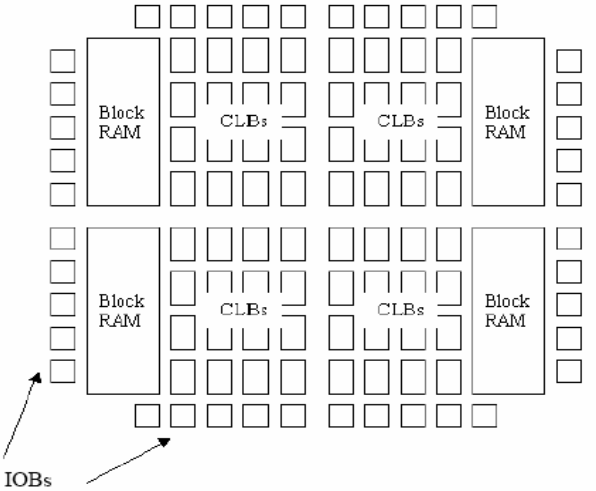

Figure 2.1. Spartan-II FPGA block diagram

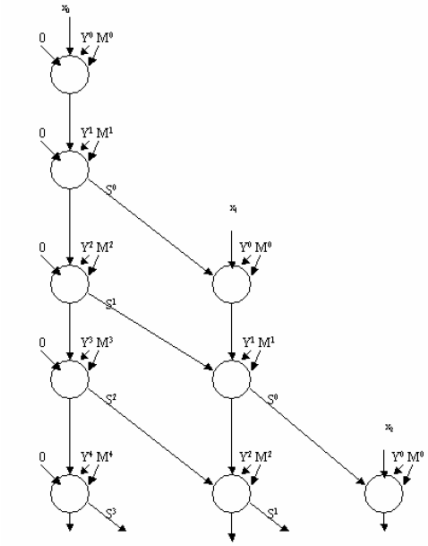
Figure 2.2. Data dependencies in the MWR2MM

The CLBs can be configured to realize the logic functions. On the left and the right sides of the chip there are block RAMs that can be configured to realize RAMs or FIFOs as explained in [23] and [24]. For each four rows of CLBs, there are two block RAMs: one on the left side and one on the right side. Each block RAM is 4 Kbits. The IOBs surround the CLBs and the block RAMs to provide the interface between the package pins and the internal signals. The versatile multi-level interconnect structure is configured to provide the necessary interconnection and routing

among the various blocks as well as among the cells inside the blocks themselves. The DLLs provide multiple minimal-skew clock signals. The programming (i.e., the FPGA configuration) of all elements is done by SRAM. This means that a Spartan-II needs to be reprogrammed every time the power is off. This is not so bad as you might think because it does not take more than 10 seconds to program the largest chip of this family, which is the same one we have on our prototyping board.

2.2. VHDL Coding for the FPGA Synthesis Tool

In this work, we are using Xilinx synthesis technology (XST). Designs are constructed of combinatorial logic and macros. XST has a set of predefined macros like multiplexers, adders, latches, flip-flops, counters, finite state machines (FSMs), and RAMs. Macros can greatly help the tool optimize the design. So, it is important that the generated VHDL code describe the design in such a way that the tool infers the appropriate macros. XST passes through two phases while synthesizing the VHDL code. In the first phase, it tries to infer as many macros as possible. In the second phase, it tries to low level optimize the design by either preserving the macros (inferred in the first phase) as separate blocks or merge them with the surrounding logic. For example, a 2-to-1 mux might be merged with other combinational logic to get better synthesis results. However, the designer can force XST to preserve a macro by setting synthesis constraints.

2.3. Multiple Word Radix-2 Montgomery Multiplication (MWR2MM)

Algorithm and Architecture

This algorithm and its general architecture were proposed in [14]. The algorithm is derived from the original Montgomery algorithm proposed in [13]. It deals with the input operands and the result of the multiplication as group of bits (words) instead of handling them at once. This makes it easier and more efficient for both software and hardware implementation. As we know, numbers in cryptography are very long. For RSA cryptography, 1024-bit numbers are used. This is expected to increase in the future because the computing power is increasing and it might be possible to crack the code in a reasonable amount of time. The algorithm shown in Algo. 2.1 is equivalent to the MWR2MM algorithm proposed in [14]. Another reason behind its suitability for hardware is that one processing unit can be reused in an iterative manner until all the whole operands are processed. This is particularly useful where the area available in the chip for such operation is limited.

The notation used in this algorithm follows the following rules. Subscripts are used to index bits and superscripts to index words. Higher index indicates a more significant bit or word. Let n be the operand size in bits, m the operand size in words, and w the word size. This means that

$m = \lceil n / w \rceil$

The operand X is scanned bit-by-bit so it is represented as

$X = x_{n-1} \ldots x_1 x_0$

Y, M, and S are scanned word-by-word. So they are represented as

$Y = Y_m \ldots Y_1 Y_0$ where Yi is word number i of Y

$M = M_m \ldots M_1 M_0$ where Mi is word number i of M

$S = S_m \ldots S_1 S_0$ where Si is word number i of S

A range of bits in a word is represented, for example, as S^i w-1…1 where this represents the bits from w-1 down to 0 of the word i of S.

Concatenation of groups of bits is performed, for example, as $(S^j0 , S^{j-1}w\text{-}1 \ldots 1)$

Step

1 S = 0
2 for i = 0 to n-1
3 $(C, S^0) := xiY^0 + S^0$
4 if S0 is odd then
5 $(C, S^0) := (C, S^0) + M^0$
6 for j = 1 to m-1
7 $(C, S^j) := C + xiY^j + M^j + S^j$
8 $S^{j-1} := (S^j0 , S^{j-1}w\text{-}1 \ldots 1)$
9 end for
10 $S^{m-1} := (C , S^{m-1}w\text{-}1 \ldots 1)$
11 else
12 for j = 1 to m-1
13 $(C, S^j) := C + xiY^j + S^j$
14 Sj-1 := $(S^j0 , S^{j-1}w\text{-}1 \ldots 1)$
15 end for
16 $S^{m-1} := (C , S^{m-1}w\text{-}1 \ldots 1)$
17 end if
18 end for

Algo 2.1. MWR2MM algorithm

The algorithm differs from the original Montgomery multiplication algorithm in the sense that the operands are processed word by word. It is shown in [14], that the carry variable C must be in the set {0,1,2} because its maximum Cmax needs to satisfy the following containment condition

$3(2^w - 1) + Cmax \leq Cmax 2^w + 2^w - 1$

This results in $C_{max} \geq 2$. Thus, Cmax =2 satisfies the condition.

2.3.1. Parallelism in the MWR2MM Algorithm

From the algorithm shown in Algo 2.1, we can see that there is data dependency in the steps performed among the j-indexed loops. This is because the previous word of S is not generated until the least significant bit of the current word of S is known. So, it is not possible to do parallel processing on them. They need to be executed serially but for the i-indexed loop, it is possible to start the next loop once the least significant word of S (S^0) of the current i-iteration is generated. But, we should note that S^0 of the current i-iteration is generated when the least significant bits of S^1 are generated, for the reason just mentioned above. This causes two-cycle delay until we can feed S^0 to the next i-iteration. But, we can still start it though and thus parallelism is possible among the i-indexed loops. Figure 2.2 shows the data dependencies of the algorithm and their timing.

Each i-indexed iteration can be executed using one processing element. The processing element is capable of performing the operations in steps 3 through 17 in Algo.2.1. These operations include checking weather M should be added or not to the result. This information is kept until the end of the iteration, which basically consists in adding the operands and the carry word-by-word in a serial manner.

2.3.2. The Scalable Architecture

The scalable architecture that implements the MWR2MM algorithm was also proposed in [14]. It is scalable in the sense that the word size (w) and the number of processing elements can be chosen by the designer, i.e., these hardware parameters can be chosen to meet the area, speed, and power requirements within the resources available to the design. Figure 2.3 shows a general organization that uses pipelining to exploit the parallelism in the MWR2MM algorithm. Virtually, we would like to make the

word size (w) as large as possible. But, this might cause several problems. One of them is the speed degradation because of high fan-out signals, long wires, and big area required for the processing unit. The area given to the multiplier might be small and limited. This kind of multiplier allows us to investigate the trade-off between the area and the speed.

Increasing the number of pipeline stages as much as we might not be useful as one might think. This was discussed in [19]. This is because the number of clock cycles the multiplier needs to execute the MWR2MM algorithm depends not only on the number of stages but also on the operand size and the word size. It is shown in [19] that this multiplier will take the following number of cycles to execute the MWR2MM algorithm.

$$C = \begin{cases} 2*\lceil n/K \rceil * K + \lceil n/w \rceil + 1 \\ \lceil n/K \rceil * (\lceil n/w \rceil + 1) + 2*(K-1) \end{cases} \begin{array}{l} \text{if } (\lceil n/W \rceil + 1) \le 2*K \\ \text{if } (\lceil n/W \rceil + 1) > 2*K \end{array}$$

Where K is the number of processing element in the pipeline, n is the operand precision, w is the word size, and $\lceil x \rceil$ is the ceiling integer of x.

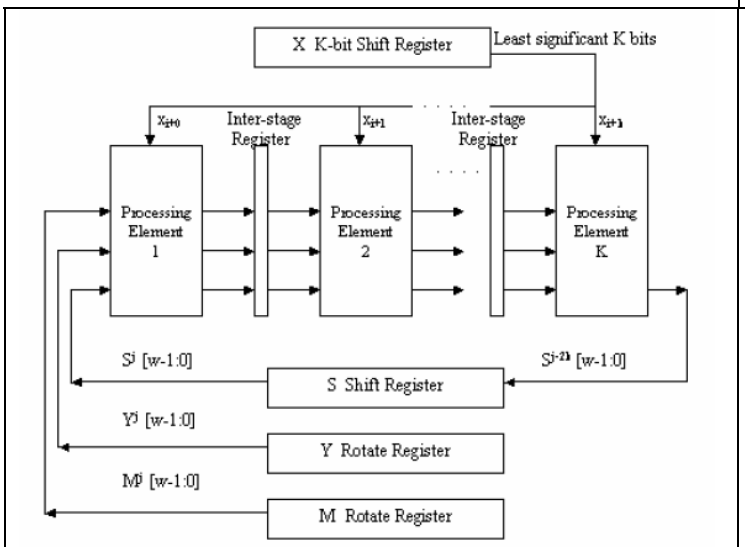

Figure 2.3. Radix-2 Scalable Montgomery Multiplier

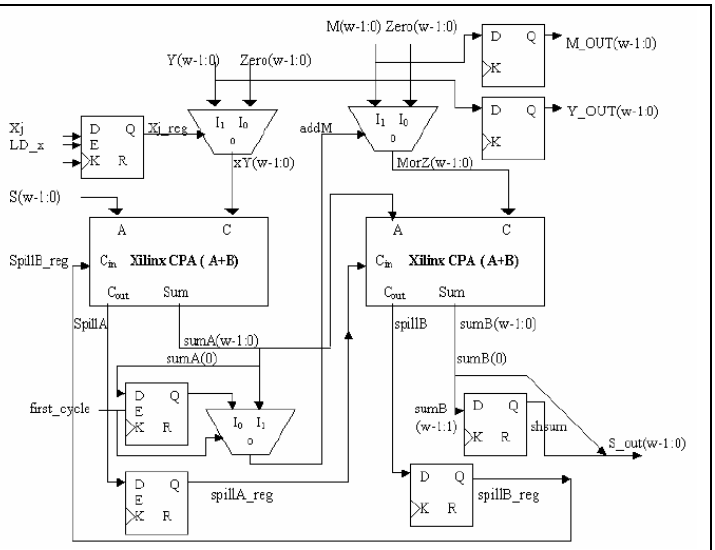

Figure 2.4. Design of MWR2MM processing element

2.4. Design of the Processing Element

This design of the processing element, shown in Figure 2.4 is very similar in functionality to the first one. However, the major difference is that instead of using carry-save adders (CSAs) Xilinx carry-propagate adders (CPAs) are used. The objective of this other design is to examine the effect of using the dedicated fast carry logic (FCL) inside the Spartan-II FPGA chip on the speed of the (CPAs).

CPAs that make use of the FCL show much better performance than those that don't use it. By using CPAs instead of CSAs, we expect to reduce the area of the processing element because we don't need to have S in two registers (as in the CSA form). This also, will reduce the size of the inter-stage register. So, we expect a reduction in the overall area needed for the pipeline because of these two expected reductions. Also, we expect the speed of the design to be comparable to the speed of the design that uses CSAs. The impact in the area and speed of the processing element is presented and analyzed in next Section.

3. Experimental Results and Analysis

This scalable radix-2 Montgomery multiplier pipeline is composed out of processing elements. In the following three subsections, we will present the implementation results and study how its area, clock cycle time (CCT), and total execution time (TET) change for each configuration.

3.1. Area

Figure 3.1 shows the area of the design versus the word size. We will stop at 16-bit word size because of the lack of I/O pins. The number of stages is fixed to 28 so that we can later compare it with the first design. The last configuration of 16-bit word and 28 stages takes 1782 slices, which is about 76% of the slices in the FPGA chip. This allows us to test a wide range of pipeline configurations as we were trying for the first design.

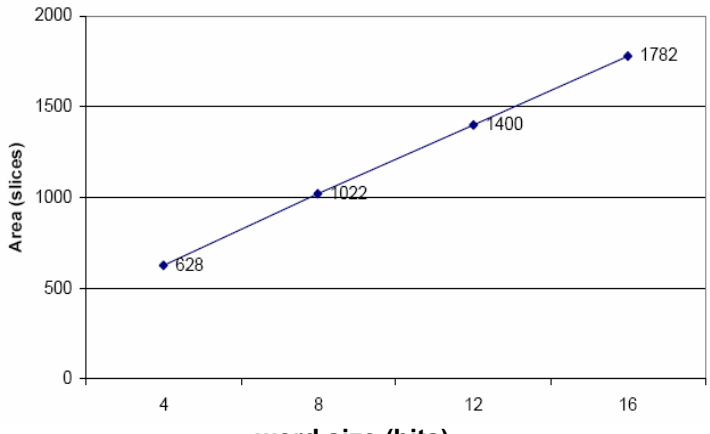

Figure 3.1. Area vs. word size, number of stages = 28

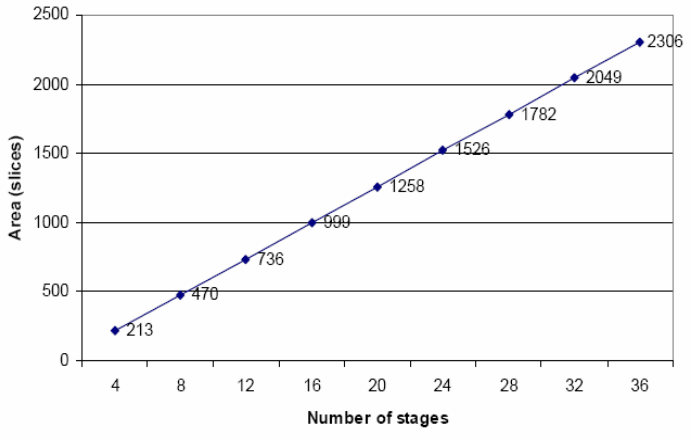

Figure 3.2. Area vs. number of stages, word size = 16 bit

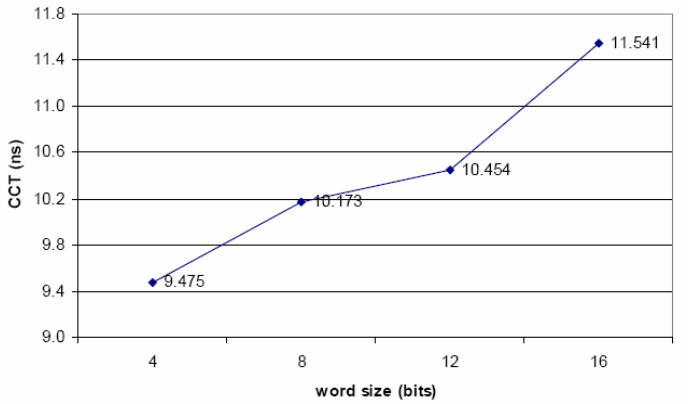

Figure 3.3. Clock cycle time vs. word size, number of stages = 28

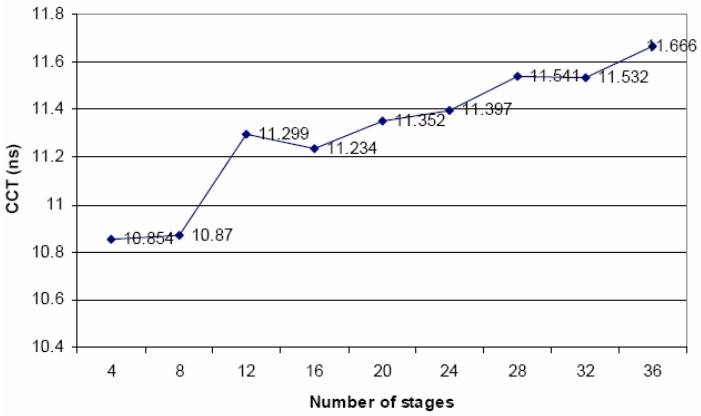

Figure 3.4: Clock cycle time vs. number of stages, word size = 16 bit

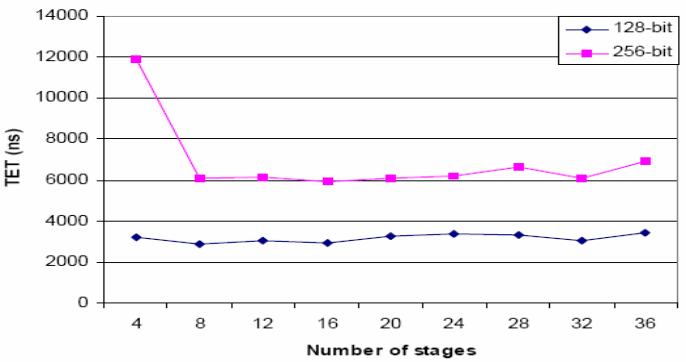

Figure 3.5 Total execution time vs. number of stages, word size = 16 bit

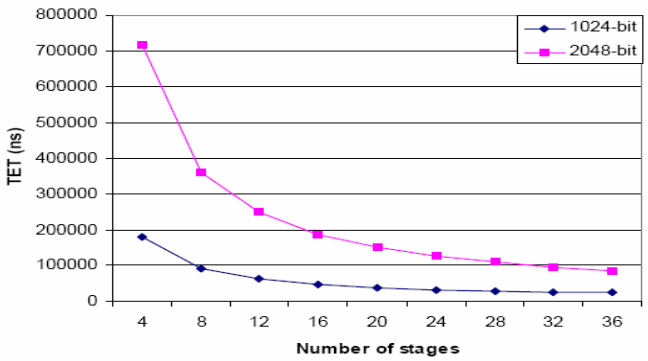

Figure 3.6. Total execution time vs. number of stages, word size = 16 bit

From Figure 3.2, we can see that the area increases almost linearly as we increase the word size. For each additional bit, the design needs about 4.7% of the slices (110 slices) on the average. As before, the length of the multiplexers, CPAs, and word registers is linearly dependent on the word size. Moreover, the optimization goal was again for speed.

Figure 3.2 shows the area versus the number of stages. The word size is fixed to 16 bits for the same reason mentioned above. The Figure shows that the area increases almost linearly as the number of stages increases. For each additional stage, the design needs about 2.7% of the slices (63 slices) on the average.

3.2. Clock Cycle Time (CCT)

These CPAs are implemented using fast carry logic (FCL) chains that require their slices be adjacent to each other in the chip. So, this puts limitation on the power of the PAR tool in optimizing design because it cannot easily move things around inside the chip. It has to keep the slices containing the carry chains adjacent. This is how Spartan-II architecture is designed.

Figure 3.3 shows the clock cycle time versus the word size. It shows that the CCT increases when the word size increases. The fastest configuration among the results is the one with 4-bit word size. It can run at 105MHz maximum frequency. The slowest is the 16-bit. It can run on 87MHz maximum frequency. The first reason why this happens is that as we increase the word size the carry chains become longer. This means the number of logic levels increases causing more logic delay. The second main reason is that there is more limitation on the PAR tool because it

has to place these carry chains in adjacent slices. This causes less efficient placement and routing and thus more routing delay. For this design, more delay comes from the logic than the routing.
The logic delay is about 60% of the total delay while the routing is about 40%. Figure 3.4 shows the clock cycle time versus the number of stages. Because this design takes less area than the first one we are able to fit designs with up to 36 stages. The Figure shows that increasing the number of stages increases the CCT. This is mainly because of the second reason mentioned above. As the design gets bigger, the chip gets more crowded. Thus, the PAR tool will have less placement and routing options as it is trying to keep the carry chains in adjacent slices. Even though there is an increase in the CCT as we increase the number of stages, the tool is still doing good and satisfactory optimization. The 4-stage design, which takes 9% of the slices and runs on 92MHz, is only about 7% faster than the 36-stage design, which takes 98% of the slices and runs 85MHz. The optimization techniques were also used here and helped the tools during the optimization process.

3.3. Total Execution Time (TET)

The fastest configuration is identified by the total execution time (TET). The TET values for four operand sizes: 128, 256, 1024, and 2048 as we change the number of stages are shown in Table 3.1. The word size is set to 16 bits.
Figures 3.5 and 3.6 show total execution time versus the number of stages. For 128-bit operand size, the minimum TET occurs when the number of stages is 8. But, we recommend 8 stages because we will loose 0.7% in speed and we will gain 31% in area. For 256-bit operand size, the minimum TET occurs when the number of stages is 16. But, we recommend 8 stages because we will loose 2.3% in speed and we will gain 22% in area. For both of them we can see that increasing the number of stages (increasing the design size) does not help. On the contrary, it becomes slower. For the large operands, 1024 and 2048 bits, the largest design that can be implemented in the chip gives the best TET. If we assume that designs of more than 36 stages can be implemented then we may find a large number of stages for which the performance (TET) starts to drop (TET increases).

Stages	128-bit	256-bit	1024-bit	2048-bit
4	3191	11874	180676	716950
8	2859	6065	90591	359123
12	3062	6135	63410	249493
16	2954	5920	47070	185833
20	3258	6073	38801	151256
24	3362	6189	32379	126963
28	3312	6636	28379	110794
32	2033	6077	24702	95923
36	3441	6895	25094	86597

Table 3.1: Total Execution times (ns), word size= 16 bit

4. Conclusion

4.1. Comparison with ASIC Implementation: Behavioral Not Quantitative

In this section, we will compare our FPGA implementation of the this design against the ASIC implementation of similar design presented in [19]. As we said earlier, we will approach the comparison in a behavioral manner not quantitative manner. ASIC and FPGA are two different technology and they have different design methodologies. For the work presented in [19], Mentor Graphics tools have been used and the target ASIC technology has been set to AMI05_slow. But for our work Xilinx ISE4.1.03 has been used and the target technology has been set to Spartan-II. Regarding the area, both implementations show that the area increases linearly as we increase the word size and/or the number of stages. Regarding the clock cycle time (CCT), our FPGA implementation shows much better immunity as we increase the word size and/or the number of stages. For 16-bit word size, our FPGA implementation becomes only less than 5% slower (CCT changes from 7.7 ns to 8.1 ns) as we increase the number of stages from 4 to 26. For the same word size, the ASIC implementation becomes 43% slower (CCT changes from 8.1 ns to 14.3 ns) if we increase the number of stages from 4 to 26. For 28 stages, our FPGA implementation becomes about 8% slower (CCT changes from 7.7 ns to 8.4 ns) as we increase the word size from 8 bits to 16 bits. For 26 stages (even less than 28), the ASIC implementation becomes about 48% slower (CCT changes from 7.4 ns to 14.3 ns) as we increase the word size from 8 bits to 16 bits. Regarding the total execution time, both implementation, FPGA and ASIC, show the same kind of curves for small operands and for large operands. Also, both of them show that the total execution time will increase if we increase the number of stages beyond some number.

4.2. Scalable Versus Fixed

In this section, we rely on a very interesting experiment we have done. We tried to synthesize a pipeline of only one single big processing element that has 1024- bit word size (can handle 1024-bit operands). Then, we tried a pipeline of 32 processing elements (each one is 32-bit word size). The processing elements are from the first version. The synthesis tool took very long time to synthesize them (one night for each). The synthesis result show that the pipeline of the single processing element needs 2231 slices (94% of the total slices) and has 9.576 ns clock cycle time. So, it can be fit in the FPGA we have and it runs on 104 MHz. Whereas, the other pipeline needs 5199 slices (2847 slices more than available in the chip) and has 9.376 ns (can run on 106 MHz). This large area requirement is because of the pipeline registers. If these two pipelines can be implemented, we expect the PAR tools to enhance the performance by 10% as it was doing for the large designs that can be implemented. This gives us a strong indication that a fixed design of Montgomery multiplier is much smaller than the scalable one if they both have process the same number of bits. It also can run at very close speed. These conclusions are correct in our case and we think they are correct for other FPGA and ASIC technologies and other design tools if the designer applies good optimization techniques and if the tools are good enough to support such techniques. However, the scalable design is very useful when we have very limited area because even one small processing element can still execute the algorithm but in longer time.

4.3. Concluding Remarks

In this work, we have FPGA-based prototyping environment that can be used to test the functionality of the Montgomery multiplier (MM) hardware at the circuit level. The MM hardware can also be reconfigured using this environment by loading to the FPGA chip the best design configuration.
In this work, we have also discovered the advantages and disadvantages of using redundant carry-save adders (CSAs) and non-redundant fast carry-propagate adders (CPAs). We have reached to the conclusion that for high-end speed, the CSAs are

better. But for limited chip area, the CPAs are better. We have also proven that fast CPAs using FCL available in some FPGA technologies can significantly improve the performance. We have also explored the FPGA design techniques that improve the design performance.

The experiment indicates that the fixed design is better than the scalable design when a lot of chip area and bandwidth are available. But in applications where area and bandwidth are very limited, the scalable design is better.

5.Bibliography

[1] H. Abelson et. al., "Amorphous Computing", Communication of ACM, vol.43, vol. 5, May 2000, pp. 74-82.

[2] R. Anderson, M. Kuhn, "Tamper resistance—a Cautionary Note", In Proceedings of the Second USENIX Workshop on Electronic Commerce, 1996.

[3] G. Borriello, R. Want, "Embedding the Internet: Embedded Computation Meets the World Wide Web", Communication of ACM, vol.43, no.5, May 2000, pp. 59-66.

[4] D. Estrin, R. Govindan, J. Heidemann, "Embedding the Internet: Introduction", Communications of the ACM, vol.43, no.5, May 2000, pp. 38-41.

[5] A. Perrig, R. Szewczyk, V. Wen, D. Culler, J. D. Tygar, "SPINS: Security Protocols for Sensor Networks", MOBICOM 2001, Rome, Italy, June 2001.

[6] C. P. Pfleeger, "Security in Computing", Second Edition, Prentice Hall, 1997.

[7] G. J. Pottie, W. J. Kaiser, "Embedding the Internet: Wireless Integrated Network Sensors", Communications of ACM, vol.43, no.5, May 2000, pp.51-58.

[8] J. Rabaey, J. Ammer, J. L. da Silva, D. Patel, "PicoRadio: Adhoc Wireless Networking of Ubiquitous Low-Energy Sensor/Monitor Nodes", Workshop on VLSI, April 2000.

[9] F. Stajano, R. Anderson, "The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks", 3rd AT&T Software Symposium, Middletown, NJ, October 1999.

[10] G. S. Sukhatme, M. J. Mataric, "Embedding the Internet: Embedding Robots into the Internet", Communication of ACM, vol.43, vol.5, May 2000, pp.67-73.

[11] D. Tennenhouse, "Embedding the Internet: Proactive Computing", Comm. of ACM vol.43, no.5, May 2000, pp. 43-50.

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

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

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

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

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

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

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

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

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

[21] A.J. Elbirt and C. Paar, "Towards an FPGA Architecture Optimized for Public-Key Algorithms," SPIE's Symposium on Voice, Video, and Communications, September 1999.

[22] C.D. Walter, "Systolic Modular Multiplication,", IEEE Transactions on Computers, vol42, no.3, pp. 376-378, 1993.

[23] Xilinx, **"**Using Block SelectRAM+ Memory in Spartan-II FPGAs**,"** Xilinx Application Note XAPP173, December 2000.

[24] Xilinx, **"**High speed FIFOs in Spartan-II FPGAs**,"** Xilinx Application Note XAPP175, November 1999.