AN ABSTRACT OF THE THESIS OF

Khaldoon Mhaidat for the degree of Master of Science in
Electrical & Computer Engineering presented on July 23, 2002.
Title: Prototyping A Scalable Montgomery Multiplier Using Field Programmable Gate Arrays (FPGAs)

Abstract approved:

Alexandre F. Tenca

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 increases the speed of modular exponentiation. Modular exponentiation and modular multiplication are heavily used in current cryptographic systems. Well-known cryptographic algorithms, such as RSA and Diffie-Hellman key exchange, require modular exponentiation operations. Elliptic curve cryptography (ECC) needs modular multiplication.

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, hand-held devices, and E-commerce, involve private and important information exchange and they need cryptography to make it secure.

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 ASIC

solution is the fastest. But, it is inflexible, very expensive, and needs long development time. The FPGA solution is flexible, reasonably fast, and needs shorter development time.

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) operations, which are much faster than regular division. The algorithm is also very suitable for a hardware implementation. Many designs have been proposed for fixed precision operands. A word-based algorithm and the scalable Montgomery multiplier based on this algorithm have been proposed later. The scalable multiplier can be configured to meet the design area-time tradeoff. Also, it can work for any operand precision up to the memory capacity.

In this thesis, we develop a prototyping environment that can be used to verify the functionality of the scalable Montgomery multiplier on the circuit level. All the software, hardware, and firmware components of this environment will be described. Also, we will discuss how this environment can be used to develop cryptographic applications or test procedures on top of it.

We also present two FPGA designs of the processing unit of the scalable Montgomery multiplier. The FPGA design techniques that have been used to optimize these designs are described. The implementation results are analyzed and the designs are compared against each other. The FPGA implementation of the first design is also compared against its ASIC implementation.

Prototyping A Scalable Montgomery Multiplier Using Field Programmable Gate Arrays (FPGAs)

by

Khaldoon Mhaidat

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for
the degree of

Master of Science

Presented July 23, 2002

Commencement June 2003

Master of Science thesis of Khaldoon Mhaidat presented on July 23, 2002

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical & Computer Engineering

Redacted for Privacy

Chair of the Department of Electrical & Computer Engineering

Redacted for Privacy

Dean of Graduate School

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

Redacted for Privacy

Khaldoon Mhaidat, Author

ACKNOWLEDGEMENTS

I would like to thank Dr. Alexandre Tenca for giving me the opportunity to work with him and gain from his experience. Dr. Tenca's reviews, comments, and discussions on this thesis work have been really invaluable.

I would like to thank my lovely wife, Tamara, who have supported and helped me during the last two years. She has always inspired me to do the best.

I would like to thank my dear mother, father, and family, who always wanted me to be excellent student and get my masters and doctoral degrees.

Last but most, I would like to thank the GOD, who created me and gave me the ability to live, sense, think, and search for the truth.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

Figure Page

LIST OF TABLES

PROTOTYPING A SCALABLE MONTGOMERY MULTIPLIER USING FIELD PROGRAMMABLE GATE ARRAYS (FPGAS)

1. INTRODUCTION

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 increases the speed of modular exponentiation. Modular exponentiation and modular multiplication are heavily used in current cryptographic systems. Well-known cryptographic algorithms, such as RSA [5] and Diffie-Hellman key exchange [4], require modular exponentiation operations. Digital Signature Standard (DSS) cryptography [7] as well as Elliptic curve cryptography (ECC) [8] need modular multiplication.

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, hand-held devices, and E-commerce systems, involve private and important information exchange and they need cryptography to make them secure.

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 since it needs a general-purpose processor for its execution. The ASIC solution is the fastest. But, it is inflexible, very expensive, and needs long development time. The FPGA solution is flexible, reasonably fast, and needs shorter development time.

Montgomery multiplication algorithm [1] 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 than regular division. 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 [2] and the scalable Montgomery multiplier based on this algorithm have been proposed later. This multiplier can be configured to meet the design area-time tradeoff. Also, it can work for any operand precision up to the memory capacity.

In this Chapter, we will describe the Montgomery multiplication algorithm and give a brief introduction to FPGAs. Also, we will review some relevant literature in the area of Montgomery multiplication and its implementations in software, ASICs, and FPGAs. In Chapter 2, We describe a prototyping environment that can be used to verify the functionality of the scalable Montgomery multiplier at the circuit level. All the software, hardware, and firmware components of this environment are described. Also, we discuss how this environment can be used to develop cryptographic applications and test procedures on top of it. In Chapter 3, we present two FPGA designs of the multiplication unit of the scalable Montgomery multiplier. The FPGA design techniques that have been used to optimize these designs will also be described. In Chapter 4, The FPGA implementation results of these two designs are analyzed and compared against each other. In the last Chapter, we provide conclusions on what we have done during this work and suggest possible future work.

1.1. 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

- X_i 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
- S_i 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 $2^{N-1} \leq M \leq 2^N-1$ such that gcd(r, M) = 1. The algorithm may be used to transform an integer in the range [0, M-1] to another integer in the same range called the image 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

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

2. image of C is calculated as

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

3. modular multiplication is calculated as

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

Figure 1.1 shows the three steps mentioned above.

Figure 1.2 shows the radix-2 MM algorithm. The algorithm replaces the division operation required in modular multiplication by simple 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 result because it is the modulus and *(S+M mod M)* is equal to *(S mod M)*. 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]. 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.

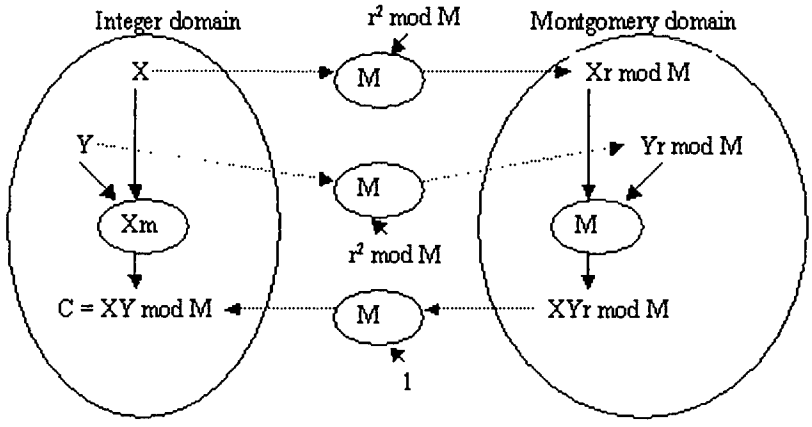

Figure 1.1. Modular Multiplication using MM

Step

1. $S = 0$
2. for $i = 0$ to $N-1$
   $S = S + X_i * Y$
   if S is ODD then $S = S + M$
   $S = S/2$
3. If $S \geq M$ then $S = S - M$

Figure 1.2. Radix-2 MM algorithm

1.2. Field Programmable Gate Arrays (FPGAs)

Field programmable gate arrays (FPGAs) are programmable chips. Figure 1.3 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 are 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. Description of these FPGAs is provided in Chapter 3.

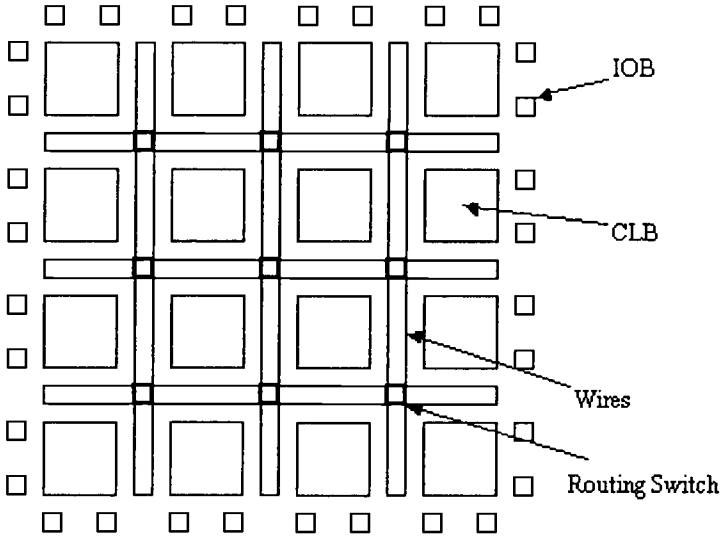

Figure 1.3. FPGA general structure

1.3. Literature Review for Montgomery Multiplication

A systolic array design for modular multiplication base on Montgomery algorithm has been presented in [20]. 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 [2]. 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.

ASIC designs, implementations and analysis of one radix-2 and two radix-8 scalable Montgomery multipliers have been presented in [10]. The radix-2 design has been based on the algorithm presented in [2]. The radix-8 designs have been based on high radix word-based Montgomery multiplication algorithm developed also in [10] based on the algorithm in [2].

Some FPGA architectures were investigated in [19] to find the key architectural criteria for implementing high performance wide-operand addition. An FPGA architecture for high performance wide-operand modular multiplication has been also proposed.

A modular exponentiation architecture that combines high-radix Montgomery multiplication with systolic array was derived in [13]. 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. THE SCALABLE MONTGOMERY MULTIPLIER PROTOTYPING ENVIRONMENT

In this Chapter, we will present the environment that can be used to prototype the scalable Montgomery multiplier. It is very important to have such an environment because it allows us to verify a design at the circuit level. Moreover, testing the functionality of the design using a prototyping environment is much faster than using CAD simulation tools. Figure 2.1 shows the scalable Montgomery prototyping environment components. Some of the components are pure software, some are pure

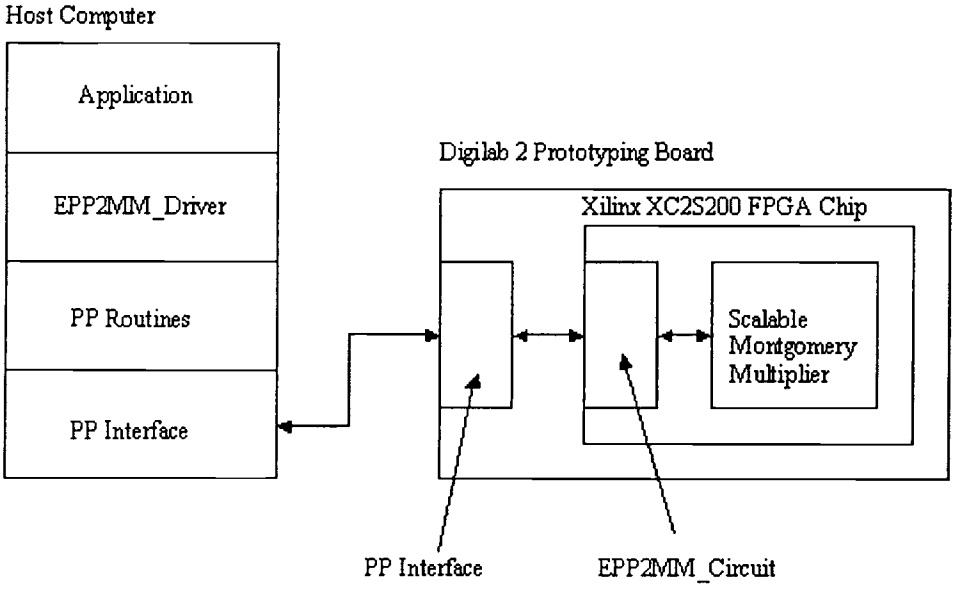

Figure 2.1. The prototyping environment

hardware, and some are firmware (software that deals with hardware). The user of this environment can develop applications that use the EPP2MM driver. The EPP2MM driver uses parallel port routines to appropriately operate the EPP2MM circuit, which

in turn makes the interface between the Enhanced Parallel Port (EPP) [37] and the Montgomery multiplier (MM). The MM will be running on the Spartan-II FPGA chip [35] residing on the Digilab 2 board [38]. In the following sections, we will be describing each component and how each of them serves us to develop and integrate the prototyping environment.

2.1. The Digilab2 Prototyping Board

Digilab 2 (D2) board has been designed by Digilent, Inc. This board provides the hardware necessary to prototype digital designs of many types using Xilinx Spartan-II FPGAs. Figure 2.2 is a block diagram of the D2 board. D2 has Xilinx XC2S200 FPGA which is the largest member in the Spartan-II family (as of the time of writing this thesis). The resources available in the Spartan-II are described in Chapter 3.

Although there are not many user input buttons and output LEDs on the board itself, the board is flexible enough to allow input/output by other ways. The board has six expansion connectors to which the user can connect other wired or printed boards. Also, the user can send the inputs to its design and receive the outputs from it through the parallel port using the Enhanced Parallel Port (EPP) protocol. The EPP will be described in the next section.

D2 uses the parallel port for programming the FPGA chip using the JTAG programming protocol as well as transferring the user data using the EPP protocol. Tri-state buffers are controlled by a port/programming switch to make this possible. The user first configures the FPGA chip with a digital circuit that understands the EPP protocol using Xilinx programming tools. Then, with the switch in the port mode the FPGA chip is able to transfer data using the EPP protocol. This is the way we have adopted to develop the prototyping environment. In the following sections, we will describe the EPP interface and the scalable Montgomery multiplier interface. Also, we

will present the interface circuit and the driver that we have developed to be able to communicate with the Montgomery multiplier hardware through the EPP port.

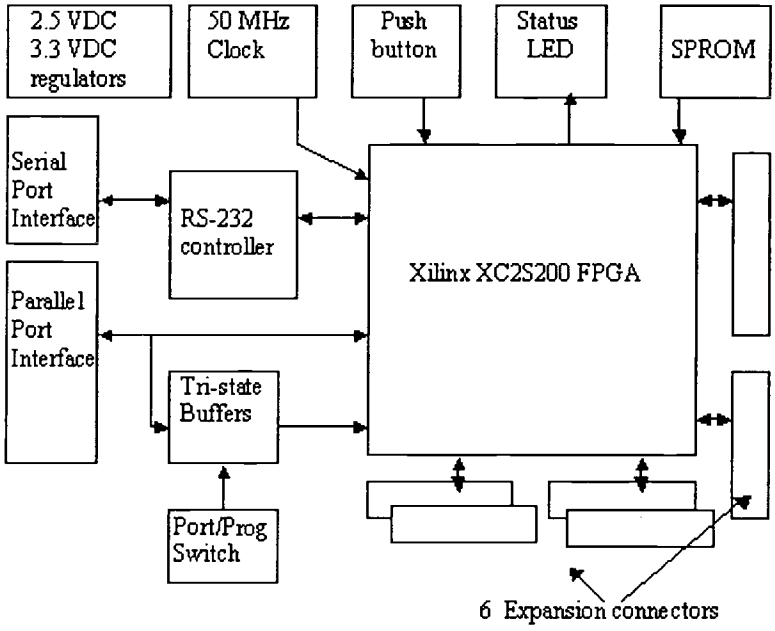

Figure 2.2. Digilab 2 circuit block diagram

2.2. The Enhanced Parallel Port (EPP)

The EPP protocol (IEEE 1284) is a handshaking communication protocol used to transfer the data between a host computer and a peripheral. There are four possible types of data transfer cycles: *data-write cycle*, *data-read cycle*, *address-write cycle*, and *address-read cycle*. Data cycles are intended for data transfer. Address cycles are intended for address or control information transfer. It is up to the designer to interpret data and address cycles in the way that makes sense to the design. We will see in Section 2.5, which describes the interface between the EPP and the MM, how these cycles where useful for transferring data, address, and control information. Table 2.1 shows the EPP signals and their use.

Figure 2.3 is an example of how a *data-write cycle* can be performed. The host asserts the *WRITE_n* line to low (to indicate that this a writing operation) and outputs the data to the parallel port. The *DATASTB_n* is then asserted to indicate that this is a data cycle. The peripheral then sends the *WAIT_n* signal high to inform the host that it has recognized the cycle and the host can end it. So, the host ends the cycle by deasserting the data strobe. After this, the *WAIT_n* is asserted low by the peripheral to indicate that the next cycle may begin. The *address-write cycle* is very similar to the *data-write cycle* except for the use of the *ADDRSTB_n* instead of *DATASTB_n* to indicate that it is an address cycle.

EPP signal name	Signal direction	EPP signal description
WRITE_n	OUT	Active low. Means writing operation when asserted
DATASTB_n	OUT	Active low. Means data cycle when asserted.
ADDRSTB_n	OUT	Active low. Means address cycle when asserted.
RESET_n	OUT	Active low. Resets the Peripheral when asserted by the host.
INTR_n	IN	Active low. Interrupts the host when asserted by the peripheral.
WAIT_n	IN	Handshaking signal. Low means the host can start a new cycle. High means the host can end the current cycle.
AD[7:0]	INOUT	data/address lines.

Table 2.1. EPP signals

Figure 2.4 is an example of how a *data-read cycle* can be performed. It is similar to the *data-write cycle* except for the use of the *WRITE_n* signal. It should stay high during the cycle to indicate that it is a reading operation. The *address-read cycle* is similar to the *data-read cycle* except for the use of the *ADDRSTB_n* instead of *DATASTB_n* to indicate that it is an address cycle.

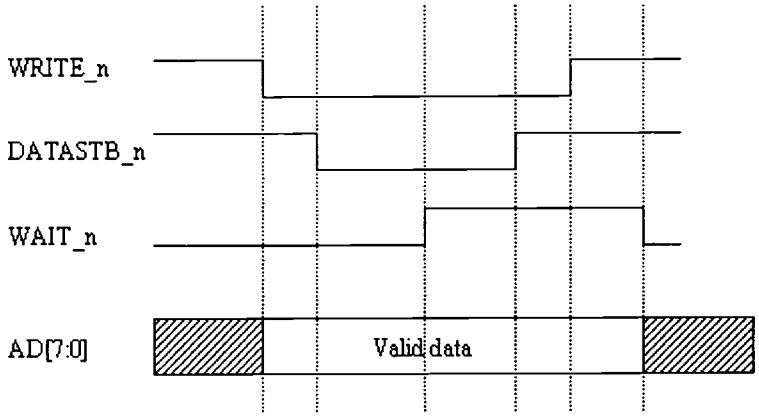

Figure 2.3. EPP data-write cycle

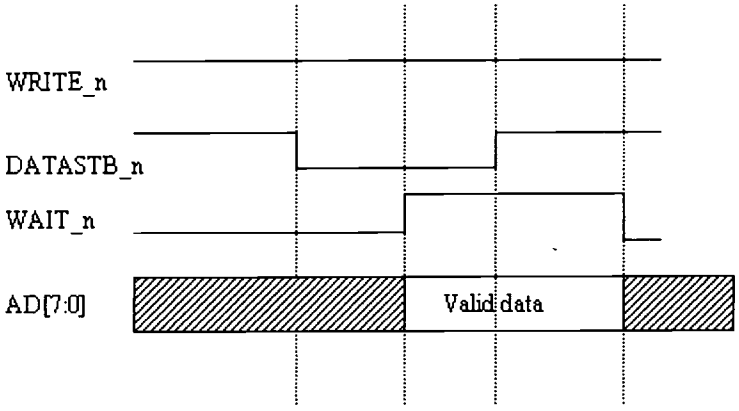

Figure 2.4. EPP data-read cycle

2.3. The Scalable Montgomery Multiplier Interfaces

In this section, we will describe the hardware and software interfaces of the scalable Montgomery multiplier (MM). Figure 2.5 is a block diagram of MM.

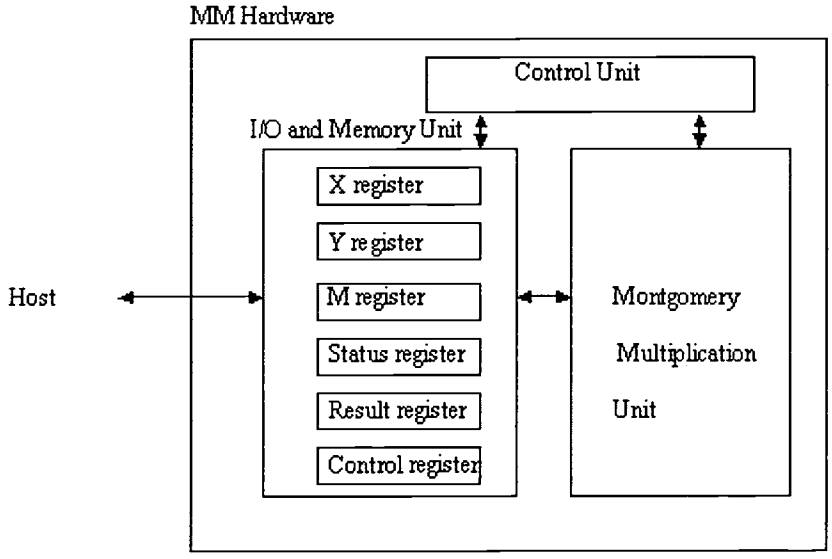

Figure 2.5. Block diagram of MM hardware

2.3.1. MM Hardware Interface

Table 2.2 shows the MM hardware interface pins and their functions. MM interface has been defined so that different host computers will be able to use its hardware.

The *cs_n* line is used to select the MM chip when asserted. Four address lines are used to specify up to sixteen locations. These locations can be mapped to either the host memory space or to its I/O space. The 32-bit registers that can be accessed in these locations are described in the next section. The 32 data lines are used to carry the 32-bit data to and from these registers. The *rd_n* line indicates a register reading

operation. The *wr_n* line indicates a register writing operation when asserted. The *reset_n* line can be used to reset the memory elements that are connected to it. The MM uses the *irq* line to interrupt the host when the job is done.

Pin(s)	Function	I/O
cs_n	chip select	In
addr(3:0)	Address	In
data(31:0)	Data	in/out
rd_n	read strobe	In
wr_n	write strobe	In
clock	Clock	In
reset_n	Reset	In
irq	interrupt	Out

Table 2.2. MM pin description

2.3.2. MM Software Interface

From a programmer point of view, the MM looks like a register file that has the registers shown in Figure 2.5. The programmer can write and read 32-bit data to these registers. The data then will be interpreted by the hardware according to the register from or to which they are being read or written. Table 2.3 shows the registers currently accessible by the programmer and their designated locations.

The X, Y, and M are 32-bit registers used to write the operands to the MM. Since these operands are most likely to be much longer than 32 bits there should be sufficient FIFO memory for these operands. These operands can be written to these big FIFOs by successive write cycles to their registers. The result also will be

available in a FIFO. Thus, it can be retrieved by successive read cycles from the result register.

Address	Register
x...xx0111	result reg
x...xx0110	Y operand reg
x...xx0101	X operand reg
x...xx0100	M operand reg
x...xx0011	reserved
x...xx0010	reserved
x...xx0001	control reg
x...xx0000	status reg

Table 2.3. MM registers

The control register has fields that can be set to specific values to control the MM operation. These values should be written to the control register before the operation can start. By writing the control register appropriately, the user can do the following things:

- specify the maximum operand size
- specify the wrap count, which determines how many times the multiplication engine will scan the Y operand words
- specify the operation to be done. The main operation is the multiplication. Other operation might be needed in the future, such as copying registers.
- clear the interrupt request
- reset the MM by software

The status register keeps information about the current status of the MM hardware, such as weather the FIFOs are empty or full and weather the operation is in progress or was completed. This register can be read by the host to check the MM status when needed.

2.3.3. Using The MM Hardware

In order to perform multiplication on the scalable MM hardware, the user must perform a sequence of steps as follows:

1. load operand size, wrap count and operation requested to the control register.
2. load x, y and m operands. This can be done, as we said, by successive writing to the x, y, and M registers. The writing operation can be done in any order.
3. ask the MM to start the operation by setting the start bit in the control register.
4. wait until the operation is done. This can be known by two ways: checking the done bit in the status register or waiting for interruption, which is activated when the MM hardware asserts the *irq* line once the operation is done.
5. retrieve the result by successive read cycles from the result register.

The application software must perform these tasks in order to execute the multiplication. The EPP2MM driver provides the required functionality to allow the application to read from and write into the appropriate MM registers.

2.4. EPP Versus MM

As we mentioned before, we intend to use the Digilab 2 board to prototype the Montgomery Multiplier. But, we need to communicate with the FPGA chip in which the MM will reside through the parallel port. It is obvious after reading Sections 2.2

and 2.3 that EPP and MM differ a lot. Table 2.4 summarizes the main differences between them.

Characteristic	EPP	MM
Communicatio n protocol	Asynchronous (with handshaking)	Synchronous
Clock	None	Clocked
Data bus	8 bits	32 bits
Address bus	N/A	4 bits
Speed	Slower; Data rate = (0 Mbytes per second - 2 Mbytes per second)	Faster; Data rate = (2816 Mbps – 4288 Mbps) for the implementation results in Chapter 4.

Table 2.4. Main differences between EPP and MM interfaces

As you can see from the table, EPP uses Asynchronous communication protocol with handshaking for transferring the data into and out of its port. On the other hand, the data is read and written in and out of the MM registers synchronously (depending on the clock and other read and write control signals). The EPP has no clock signal. The data bus of the EPP is 8-bit wide, whereas it is 32-bit wide for the MM. There is no dedicated address bus for the EPP port unlike the MM. The fastest EPP data rate can approach 2 MHz while the MM can run at clock frequency that ranges from 134 MHz to 88 MHz for some of its configurations implemented for the Xilinx XC2S200 FPGA, as we'll see in Chapter 4. Thus, the reading and writing operations in the EPP are slower than and they have different timing from these of the MM.

For all these reasons, we need an EPP-MM interface circuit, which is able to transfer the data correctly between the EPP and the MM. Also, we need a driver (software program) that can operate this circuit. In the next two subsections, we will describe the design of circuit that resides on the FPGA chip as well as the software driver in the host, used to bridge the access between the host and the MM hardware in the FPGA chip.

2.5. EPP2MM Interface Circuit

Figure 2.6 shows the EPP2MM interface circuit. The purpose of this circuit is to allow the communication between the host using the EPP and the MM hardware inside the FPGA. This circuit understands the protocols used by the two sides. It enables the driver running on the host computer to control the data transfer. It also correctly manipulates their signals in each interface (EPP and MM) in terms of value and timing.

The operations supported by this circuit are:

- host writes a 32-bit word to a temporary write buffer (4-byte register) in chunks of 8 bits at a time.
- host instructs the interface circuit to write the temporary write buffer contents to specific MM register.
- host instructs the interface circuit to read an MM register and store the value into a temporary read buffer (4-byte register).
- host reads a 32-bit word from the temporary read buffer in chunks of 8 bits at a time.

Before we explain how the circuit performs the operations above, we need to explain the purpose of the address and control byte shown in Figure 2.6. Table 2.5 shows the address and control byte fields and their use. The least significant four bits of the address and control byte specify the address of the MM register. The most

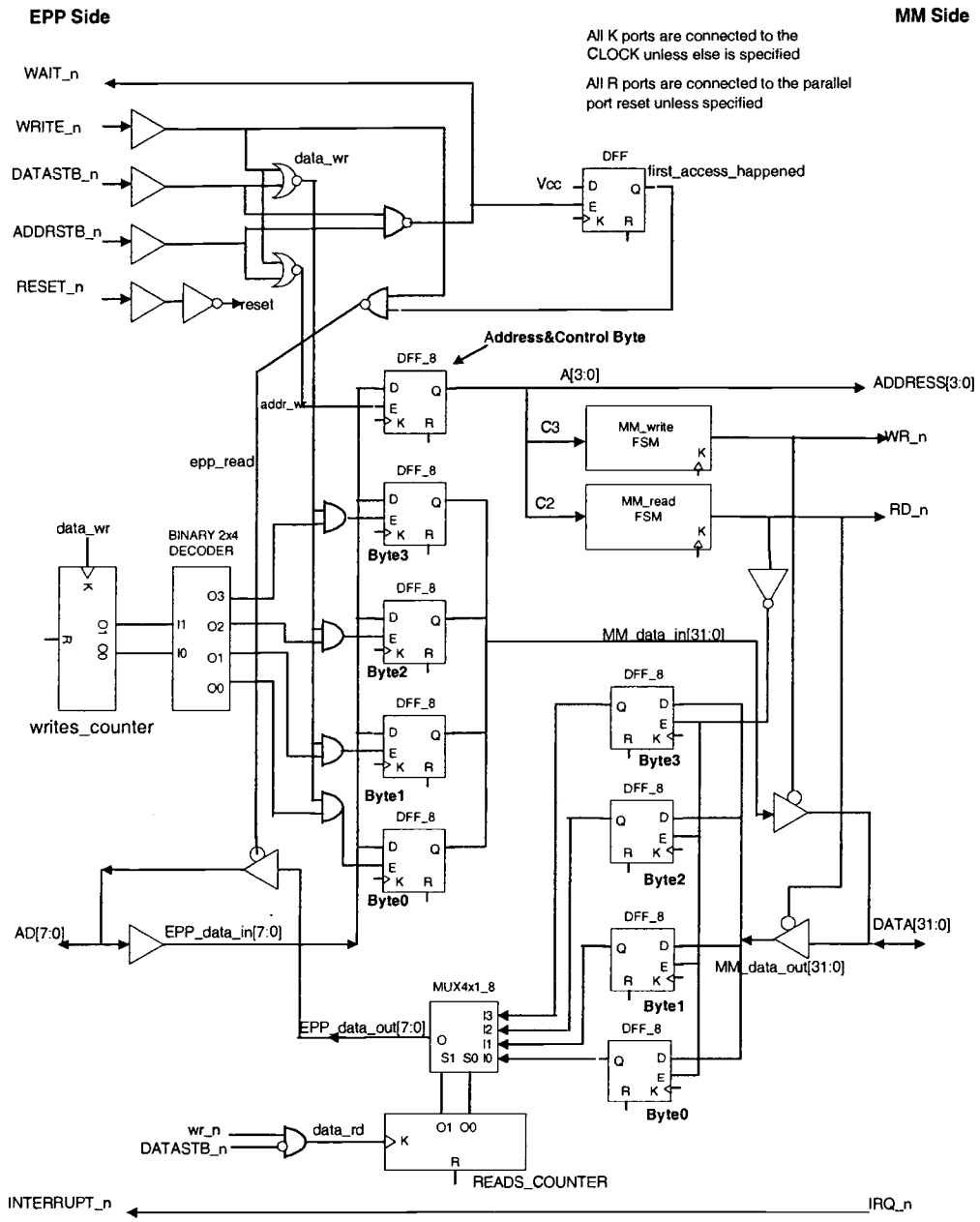

Figure 2.6. EPP2MM Interface circuit

significant four bits of the address and control byte are used as a control code. The control code and the current state of the control part of the circuit will determine which operation (read or write) should be implemented and for how long, i.e., they are used to control the MM reading and writing operations.

2.5.1. Writing A 32-bit Word From The Host To The Write Buffer In The EPP2MM Circuit

For writing a 32-bit data word to the MM, the circuit firstly assembles them from the EPP in its four data bytes (write buffer). A 2-bit write counter is used to keep the number of the data byte to be written. Initially, it has 0 output then it is incremented each time an EPP *data-write cycle* is initiated. The output of this counter is fed into the input of a binary 2x4 decoder. If the *data_wr* signal is high, this means a *data-write cycle*, and the decoder output will enable a byte to be written in the write buffer. The 2-bit write counter is triggered by the *data_wr* signal. This signal goes from low to high when the EPP we_n and *DATASTB_n* are both asserted to indicate a data write cycle.

2.5.2. Writing A 32-bit Word From The Write Buffer In the EPP2MM Circuit To The MM Register

After writing a 32-bit word from the EPP into the write buffer, it is possible to write this word to one of the MM registers. To do so, we write the value (10xx $A_3A_2A_1A_0$) then the value (00xx $A_3A_2A_1A_0$) to the control and address byte. $A_3A_2A_1A_0$ specifies the address. x means don't care bit input. The most significant bit (C3), of the address and control byte, is used to trigger the operation of the MM_write Finite State Machine (*MM_write FSM*). *MM_write FSM* is shown in Figure 2.7. This FSM is necessary to assert the write enable line (*wr_n*) of the MM hardware only for

one cycle. Such a behavior is needed to avoid writing the word more than once to the MM FIFO for example.

Bit position	Bit name	Bit use
3-0	A3-A0	4-bit MM address
5-4	C1-C0	Future use
6	C2	Controls the MM_read FSM
7	C3	Controls the MM_write FSM

Table 2.5. Address and control byte fields

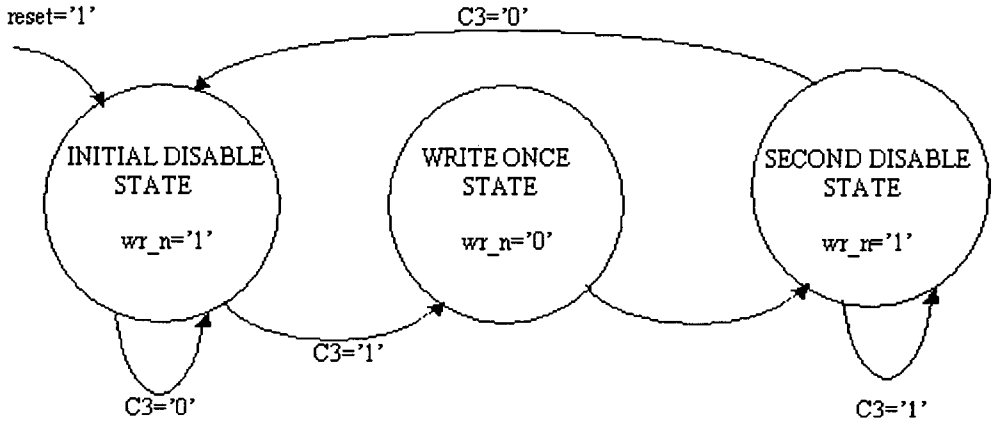

Figure 2.7. MM_write FSM

2.5.3. Reading A 32-bit Word From The MM Register To The Read Buffer In The EPP2MM Circuit

To read a 32-bit word from the MM into the read buffer in the EPP2MM circuit, the host performs the following operations. It writes the address and control byte first and then it reads the data one byte at a time. More specifically, the value

(01xx $A_3A_2A_1A_0$) followed by the value (00xx $A_3A_2A_1A_0$) are written to the address and byte. $A_3A_2A_1A_0$ specifies the MM register address. x represents don't care bit. The C2 bit is used to trigger the operation of the *MM_read FSM*. The *MM_read FSM* is shown in Figure 2.8. This FSM is necessary to assert the read enable line (*rd_n*) of the MM for only one cycle so that only one word is read from the MM FIFO. Otherwise, too many words will be popped out of the FIFO, which is of course undesired.

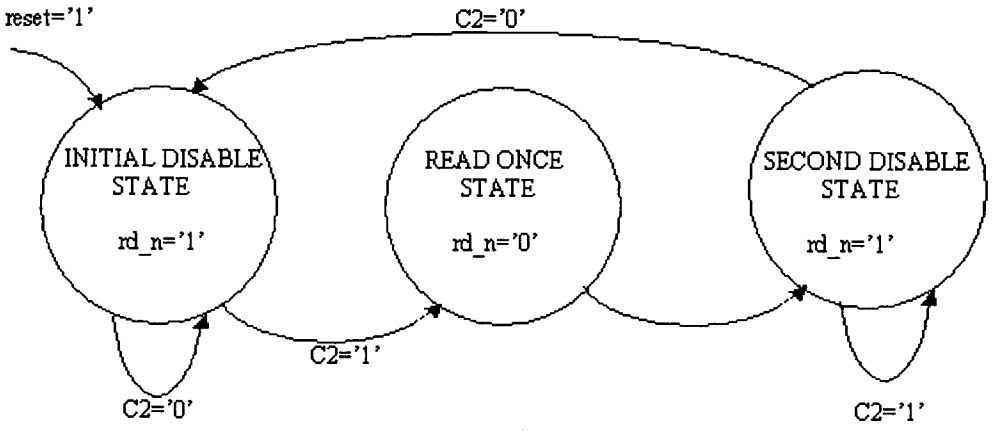

Figure 2.8. MM_read FSM

2.5.4. Reading A 32-bit Word From The Read Buffer In The EPP2MM Circuit To The Host

After reading the 32-bit word from the MM register to the read buffer, we would like to transfer this word through the parallel port to the host (one byte at a time). An 8-bit 4x1 multiplexer is used for this purpose. The selection is based on the value of a 2-bit *reads counter* that keeps track of the byte to be read. This *reads counter* is triggered by the *data_rd* signal. The *data_rd* signal is asserted when both the EPP *WRITE_n* is unasserted and *DATASTB_n* is asserted meaning there is a *data-read cycle*. Thus, the host makes four read accesses to the EPP2MM hardware in order to read the 32-bit word in the read buffer.

2.5.5. Implementation Aspects Of The EPP2MM Circuit

Typically, the data strobe and the address strobe (*DATASTB_n* and *ADDRSTB_n*) should be used to clock the flip-flops of the data write bytes and the address and control byte, respectively. However, we were forced to use an external fast clock to sample the data lines while the *data_wr* or *addr_wr* line is asserted. The reason is that Digilent connected the *DATASTB_n* and *ADDRSTB_n* to input pins that cannot be connected to a global clock buffer, directly. Xilinx ISE 4.1 tools map only dedicated pins in the FPGA to these global clock buffers. A gated clock is allowed but it is generally not a good design practice. A designer should avoid gated clocks unless there is no other way. The tool will give a warning message for this kind of clock because it prefers to use the global clock buffer paths dedicated to the clock signals coming from the clock pins. Such gated clocks were unavoidable in this case. Two gated clocks are needed: *data_wr* and *data_rd.* They are necessary to clock the *writes_counter* and the *reads_counter*, respectively.

A fast clock is used to sample the byte being written into the write buffer, read buffer, or address and control byte while they are enabled. The EPP speed depends on three factors:

1. The EPP interface of the system to which the peripheral is attached.
2. The peripheral itself (in this case, the Digilab2 board with EPP2MM circuit on it.)
3. The driver that operates the peripheral in the EPP mode (in this case, the EPP2MM driver presented in the next Section.)

The following times have been measured for our prototyping environment (PC, EPP2MM circuit, and EPP2MM drive) using a logic analyzer:

- $T_{EPP\_read} = T_{EPP\_write}$ = EPP cycle time = 2 μ sec

- $T_{DATASTB\_n}$ = data strobe assertion time = 200 ns
- $T_{ADDRSTB\_n}$ = address strobe assertion time = 400 ns
- T_{WR_n} = write strobe assertion time = 700 ns
- T_{DATA} = valid data time = 740 ns
- T_{data_wr} = *data_wr* assertion time = 200 ns
- T_{data_rd} = *data_rd* assertion time = 200 ns
- T_{addr_wr} = addr_*wr* assertion time = 400 ns

Some of these times are also shown on Figure 2.9. The sampling period attained from the clock available on the board is equal to

$$T_s = 1/ F_s = 1/ (50 \text{ MHz}) = 20 \text{ ns}$$

This ensures that the write and read buffers will be clocked at least 9 times and the address and control byte will be clocked at least 19 times while their inputs are valid. In general, we need two samples to occur while the enable line is asserted. The EPP and the clock are not synchronized. So, one sample is not enough because it might not meet the setup and hold time requirements of the FFs. This means that the sampling clock must have a cycle time which is less than half the minimum enable assertion time. In this case, we need

$$T_{EPP2MM} < T_s < 100 \text{ ns}$$
$$10 \text{ MHz} < F_s < F_{EPP2MM}$$

Where T_{EPP2MM} and F_{EPP2MM} are the maximum clock frequency and the minimum cycle time of the EPP2MM circuit. Since each EPP cycle takes about 2 μ sec then the achieved data rate is 500 Kbps. But, since 4 out of 6 cycles are used for data transfer because 2 cycles are used for address and control transfers then the effective data rate is 333.3 Kbps.

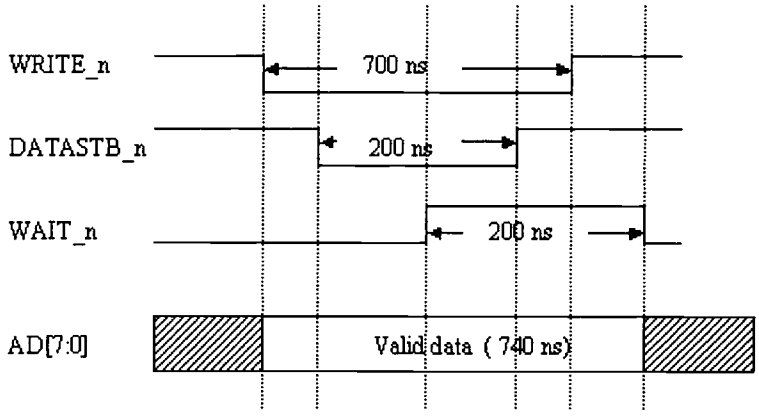

Figure 2.9. The prototyping environment EPP timing

2.6. EPP2MM Driver

The EPP2MM driver (*EPP2MM.cpp*) is the program that controls the EPP2MM interface circuit. The driver code can be found in appendix A. This program has two procedures (C++ functions). The first one writes a 32-bit word to the MM at the specified address. The second one reads a 32-bit word from the MM at the specified address.

The writing procedure follows a sequence that allows the EPP2MM circuit to do the writing in a correct manner. The writing sequence is as follows:

1. Write the first data byte to the EPP data port
2. Write the second data byte to the EPP data port
3. Write the third data byte to the EPP data port
4. Write the fourth data byte to the EPP data port
5. Write the control and address byte (10xx $A_3A_2A_1A_0$) to the EPP address port
6. Write the control and address byte (00xx $A_3A_2A_1A_0$) to the EPP address port

The reading procedure follows a sequence that let the EPP2MM circuit does the reading in a correct manner. The reading sequence is as follows:

1. Write the control and address byte (01xx $A_3A_2A_1A_0$) to the EPP address port
2. Write the control and address byte (00xx $A_3A_2A_1A_0$) to the EPP address port
3. Read the first data byte from the EPP data port
4. Read the second data byte from the EPP data port
5. Read the third data byte from the EPP data port
6. Read the fourth data byte from the EPP data port

Detailed description of the driver functions is included in the source code in the Appendix.

3. DESIGNING FOR FPGAS

In this Chapter, two FPGA implementations of radix-2 scalable Montgomery multiplier are presented. These two implementations are based on the MWR2MM algorithm and architecture presented in [3]. The algorithm and architecture will be described in Section 3.3.

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.

In the work presented in this thesis, we are using Xilinx ISE 4.1.03 design development tools. VHDL has been used for design description. Xilinx synthesis technology (XST) has been used for synthesis. Our target technology is Xilinx Spartan-II FPGAs since our prototyping board has a XC2S200 FPGA, which belongs to the Spartan-II family. We will leave the presentation of the results until Chapter 4. But, in this Chapter we will describe the Spartan-II FPGAs. Also, Some important notes while writing VHDL for the FPGA synthesis tool will be discussed.

3.1. 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 3.1.

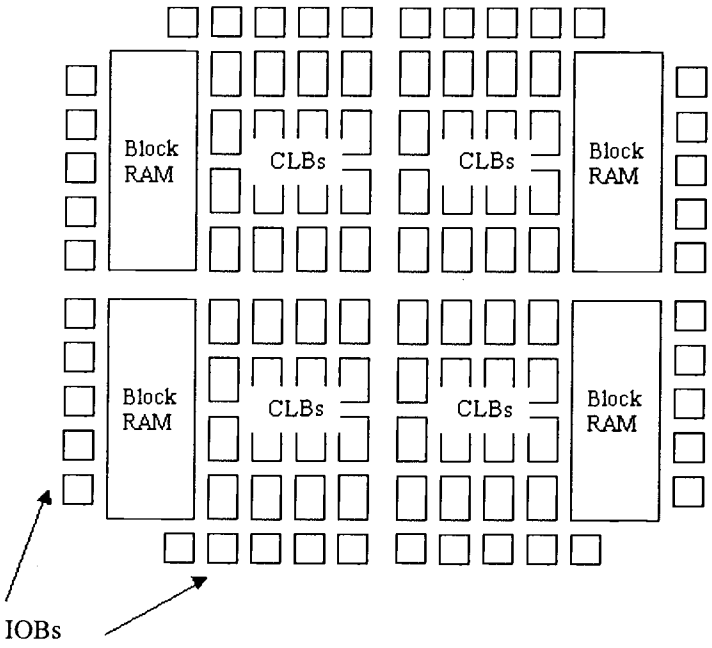

Figure 3.1. Spartan-II FPGA block diagram

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 [33] and [34]. 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.

One kind of the resource of special interest to us is the CLB because the logic of the design is realized using the CLBs. A Spartan-II FPGA contains an RxC array of CLBs. The height and width of the array depends on how big the chip is. Each CLB has two slices. Figure 3.2 shows the basic slice structure. Each slice has the following logic elements: two look-up tables (LUTs), two storage elements, one multiplexer (F5MUX), carry and control logic. Each LUT is a 16x1 RAM that can be used as a logic function generator, 16x1 synchronous RAM, or 16-bit shift register. The two LUTs can be combined to make a 32x1 or 16x2 synchronous RAM, or 16x1 dual-port synchronous RAM. The F5MUX can be used to combine the output of both LUTs. By this combination it is possible to implement a 4-to-1 multiplexer, any 5-input logic function, or some 9-input functions. Each CLB has also an F6MUX. This multiplexer combines the outputs of the two slices. This combination of two slices can implement an 8-to-1 multiplexer, any 6-input functions, or some 19-input functions. The two storage elements provide the support for implementing sequential logic functions. They can be configured to be D flip-flops or D latches. The dedicated carry logic inside each slice provides arithmetic carry chain. A slice has one chain. The chain is 2-bit in length. Later in this Chapter, we will describe how fast carry propagate adders (CPAs) can be used instead of the redundant carry-save adders. In Chapter 4, the experimental results will show us impact of using these adders. We also identify some situations in which these adders might become the best solution.

To be more specific, the XC2S200 FPGA that we use in this work (the Digilab2 board) is presently the biggest in the Spartan-II family. It has 28x42=1176 CLBs, 284 user I/O pins, and 56 K bits of block RAM. This provides a lot of resources that should be carefully utilized. The same design might need more than 100% of some kind of resource. But, it might take much less than that if the design is described in another way that take advantage of the availability of other kinds of resources. Detailed information about Spartan-II FPGAs can be found in [29].

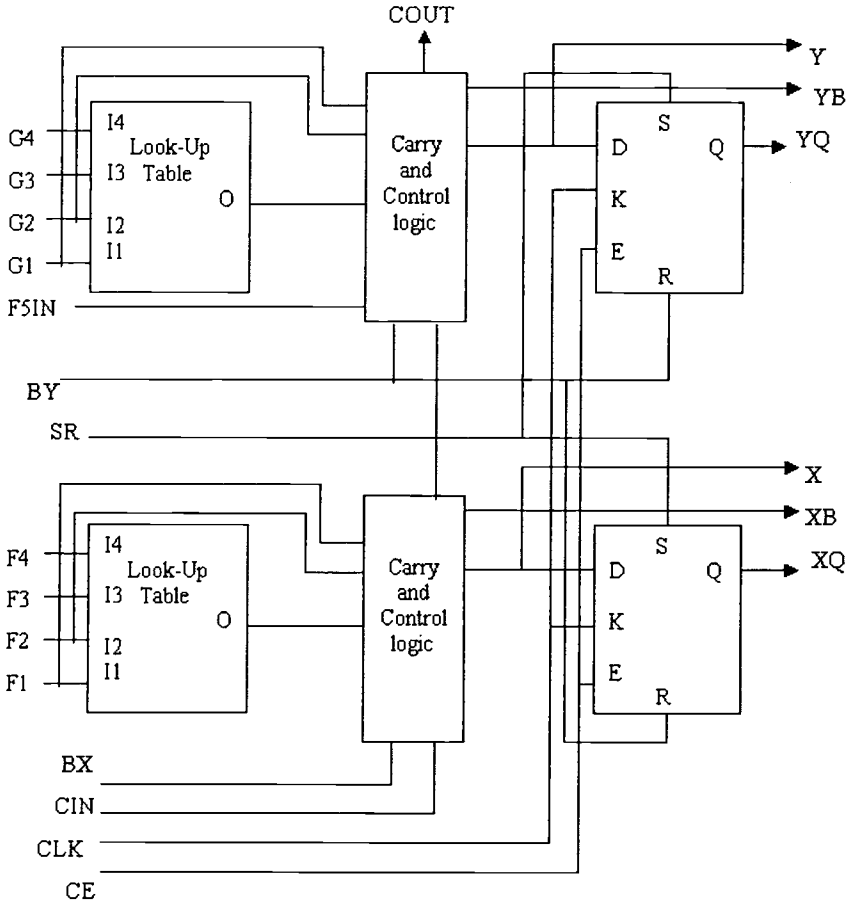

Figure 3.2. Spartan-II CLB slice

3.2. 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.

Multiplexers in XST can be described in VHDL using if-then-else, case, or when structures. A designer must be careful when describing the mux to XST. For example, if some of the selection values are not specified in any of these VHDL structures then XST will infer latches instead of muxes for the unspecified selection values. For example, this situation happens when using the following code:

```
entity mux is
     port (in0, in1, in2, in3 : in std_logic;
     sel : in std_logic_vector (1 downto 0);
     o : out std_logic);
end mux;

architecture behavioral of mux is
begin
     process (in0, in1, in2, in3, sel)
     begin
          if (sel = "00") then o <= in0;
          elsif (sel = "01") then o <= in1;
          elsif (sel = "10") then o <= in2;
          end if;
     end process;
```

```
end behavioral;
```

In this example, the output is not assigned any value when the selection value is "11". XST assumes that it should keep its old value. Thus, it infers a latch for this purpose.

XST supports various types of latches and flip-flops. It can recognize positive/negative edge-triggered flip-flops with/without clock enable and synchronous/asynchronous set/clear control signals. It also recognizes latches with positive/negative enable and synchronous/asynchronous set/clear control signals. In this context, an important notice should be mentioned. As we mentioned earlier, the same design that is efficient for some FPGA family using some CAD tool can be inefficient for another family using another tool. In our design there are some registers that don't need to be reset, i.e., we can use flip-flops that don't have a clear signal. Since XST supports this and Spartan-II have this kind of resource, the design that uses flip-flops without clear signal was found to be 1-2% faster and needs 1-2% less area than the one that uses flip-flops with clear signal. We will comment on this in Chapter 4. On the other hand, experiments with XC4000 FPGAs and Xilinx Foundation 2.1i tools show the opposite. XC4000 FPGAs have only flip-flops with asynchronous clear. VHDL codes that describe a flip-flop with no reset signal will cause a tool to generate a warning message because this leads to inefficient implementation. In [12], it is noted that designs that don't make use of the global reset available paths in the chip will be 10-20% slower.

As mentioned before, Spartan-II FPGAs have dedicated fast carry logic (FCL). This FCL can be used to implement adders, subtractors, and comparators. Adders can have carry-in and carry-out. Also, subtractors can have borrow-in and borrow-out. The VHDL writer can let the XST infer adders, subtractors, or comparators by simply applying the desired arithmetic operation. For example, an adder with carry-in can be inferred by writing:

```
C <= A+B+ cin ;
```

Whereas, an adder with carry-in and carry-out can be inferred by writing:

```
ext_A <= '0' & A;
ext_B <= '0' & B;
C <= ext_A + ext_B +  cin;
cout <= C(n-1);
```

The operands should be zero-extended one more bit so that the most significant bit of the summation will indicate the carry-out value. The same thing applies for subtraction. For comparison, a less-than comparator, for example, can be inferred by writing:

```
(A < B)
```

The adders implemented using the FCL are much faster than any carry-propagate adders the designer might design without making use of the FCL available in the chip. The reason is that when using the FCL the adder will be implemented in adjacent slices. The routing delays of the carry signals between the two adjacent carry chains are minimal (almost zero) compared to the routing delays of the carry signals generated by the function generators (LUTs). Redundant adders like carry-save adders are faster though, because they don't have carry-propagation at all. But, adders that use the FCL are comparable to the carry-save adders in terms of speed. The two design presented in this Chapter are examined in respect to the effects of using carry-save adders (CSAs) vs. carry-propagate adders (CPAs) that uses the FCL available in Spartan-II FPGAs, on the speed and area. The implementation results of these two designs are presented and analyzed in Chapter 4.

Synthesis and implementation constraints are additional information to help the tools optimize the design. For example, to reduce the delay caused by high fan-out signals, a synthesis constraint that specifies the maximum allowed fan-out can be set.

Also, speed can be improved by allowing register replication. Register replication will reduce the delay because many copies of the same register can be placed in different chip locations to minimize the routing delay. Of course, this strategy increases the required area. Resource sharing can be enabled at synthesis time to reduce the needed area. Resource sharing is possible whenever two or more tasks can be implemented using the same resource and they don't need it at the same time. For implementation, timing constraints are useful in optimizing the design. They affect the placement and routing process. For example, an initial target minimum clock cycle time can be set. Then, the tool will try to achieve it. Then, we can tighten the constraint more and more until the tool is no longer able to achieve it.

3.3. Multiple Word Radix-2 Montgomery Multiplication (MWR2MM) Algorithm And Architecture

This algorithm and its general architecture were proposed in [3]. The algorithm is derived from the original Montgomery algorithm proposed in [2]. 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 Figure 3.3 is equivalent to the MWR2MM algorithm proposed in [3]. 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

$$m = \lceil n / w \rceil$$

Step

1 $S = 0$
2 for i = 0 to n-1
3 $(C, S^0) := x_i Y^0 + S^0$
4 if S^0 is odd then
5 $(C, S^0) := (C, S^0) + M^0$
6 for j = 1 to m-1
7 $(C, S^j) := C + x_i Y^j + M^j + S^j$
8 $S^{j-1} := (S^j_0 , S^{j-1}_{w-1\ldots1})$
9 end for
10 $S^{m-1} := (C , S^{m-1}_{w-1\ldots1})$
11 else
12 for j = 1 to m-1
13 $(C, S^j) := C + x_i Y^j + S^j$
14 $S^{j-1} := (S^j_0 , S^{j-1}_{w-1\ldots1})$
15 end for
16 $S^{m-1} := (C , S^{m-1}_{w-1\ldots1})$
17 end if
18 end for

Figure 3.3. MWR2MM algorithm

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 Y^i is word number i of Y

$M = M^m \ldots M^1 M^0$ where M^i is word number i of M

$S = S^m \ldots S^1 S^0$ where S^i is word number i of S

A range of bits in a word is represented, for example, as

$S^i_{w-1\ldots1}$ 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

$(S^j_0 , S^{j-1}_{w-1\ldots1})$

The algorithm differs from the original Montgomery multiplication algorithm in the sense that the operands are processed word by word. It is shown in [3], that the carry variable C must be in the set {0,1,2} because its maximum C_{max} needs to satisfy the following containment condition

$3(2^w - 1) + C_{max} \leq C_{max}2^w + 2^w - 1$

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

3.3.1. Parallelism In The MWR2MM Algorithm

From the algorithm shown in Figure 3.3, 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 3.4 shows the data dependencies of the algorithm and their timing.

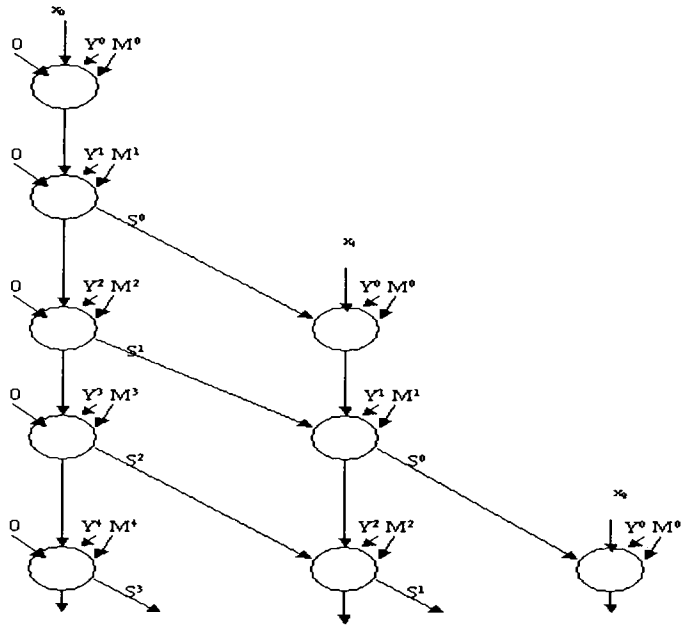

Figure 3.4. Data dependencies in the MWR2MM

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 Figure 3.3. 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.

3.3.2. The Scalable Architecture

The scalable architecture that implements the MWR2MM algorithm was also proposed in [3]. 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 3.5 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 can might not be useful as one might think. This was discussed in [10]. 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 [10] that this multiplier will take the following number of cycles to execute the MWR2MM algorithm.

$$C = \begin{cases} 2 * \lceil n/K \rceil * K + \lceil n/w \rceil + 1 & , \text{if } (\lceil n/w \rceil + 1) \leq 2 * K \\ \lceil n/K \rceil * (\lceil n/w \rceil + 1) + 2 * (K-1) & , \text{if } (\lceil n/w \rceil + 1) > 2 * K \end{cases}$$

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.

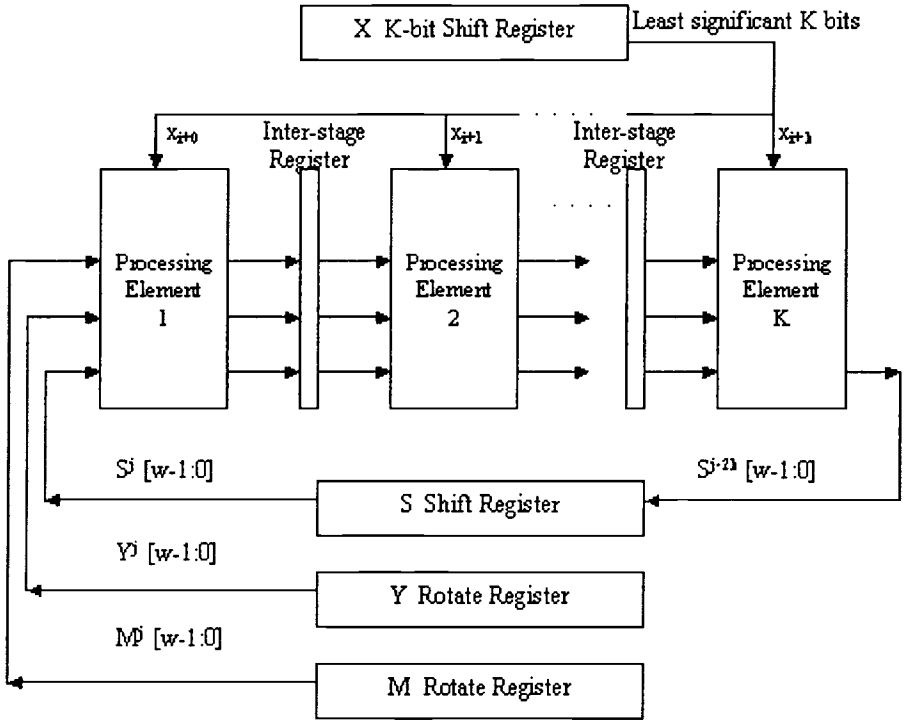

Figure 3.5. Radix-2 Scalable Montgomery Multiplier

3.4. Design Of The Processing Element (Version 1)

Figure 3.6 shows the first design of the processing element that can be used to compose the pipeline shown in Figure 3.5. This design uses carry-save adders (CSAs) to perform the addition. This has the advantage of no carry propagation delay. The carry propagation delay is one of the major sources of delay in the designs require addition. By using CSAs, the addition delay is no longer dependent on the word size. Because we are using CSAs, S is represented should the redundant CSA form. As you see in the diagram, it is represented by SS and SC for the input S and by OS an OC for the output S.

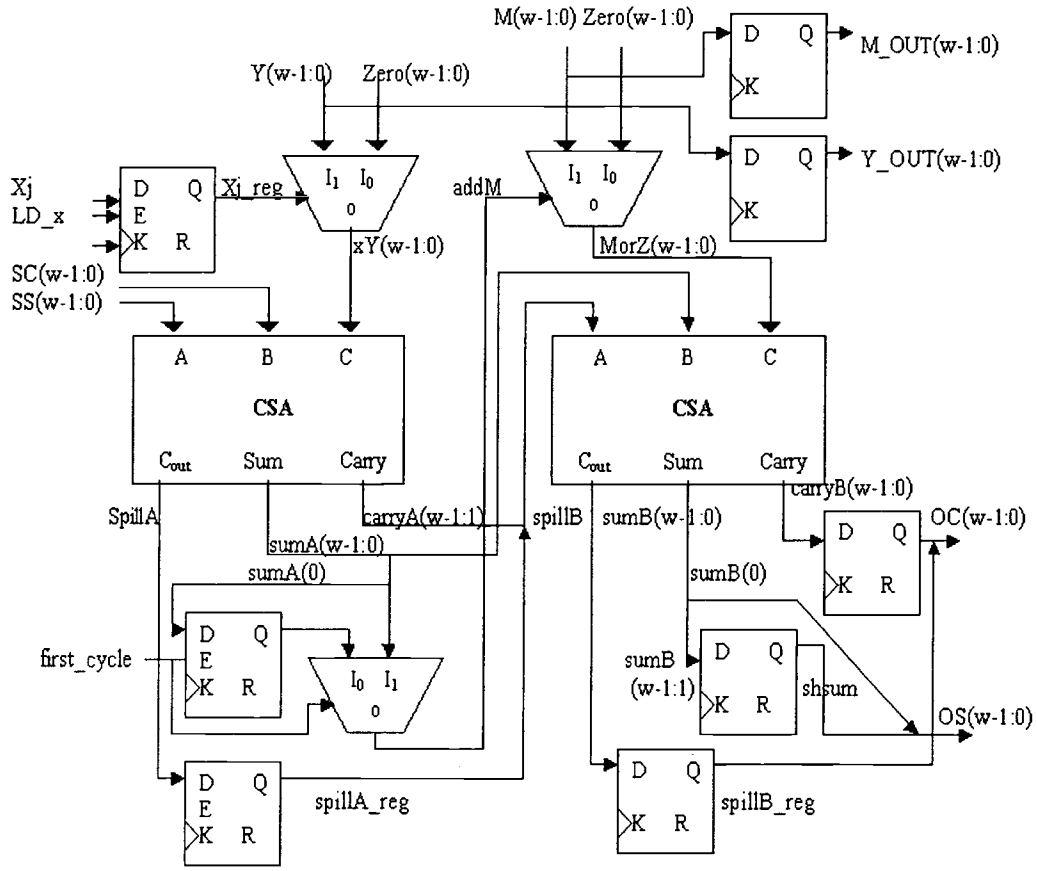

Figure 3.6. Design of MWR2MM processing element (version 1)

The term $x_i * Y^j$ is calculated using a w-bit 2-to-1 mux. The first CSA in the left performs the addition $(C, S^j) := x_i Y^j + S^j$. A flip-flop and a 2-to-1 mux implement the odd-or-even test. The flip-flop is enabled only in the first cycle to keep the least significant bit (LSB) of the result until the end of the computation cycle (one i-iteration). The mux is to select between the LSB of the sum of the first CSA or the value stored in the flip-flop. The mux will select the LSB of the sum of the first CSA only in the first cycle. After that, it will select the value kept in the flip-flop until the end of the iteration. The mux controlled by addM implements the cases weather to add M or not by selecting M^j or Zero. The second CSA on the right will either perform the

addition $(C, S^j) := C + x_i Y^j + M^j + S^j$ or $(C, S^j) := C + x_i Y^j + S^j$ depending on the result of the M^j-or-Zero mux. The shifting is done by taking the LSB of the second CSA sum as the MSB of the OS output and taking the spillB_reg as the LSB of the OC output while saving other sum and carry bits to be output in the next cycle. X_i will be loaded only in the beginning of a computational cycle (i-indexed iteration). Y^j, M^j, and S^j are loaded to the first processing element in the pipeline each cycle. They will be propagated through registers inside the processing element itself and through inter-stage registers to the next processing elements. As we explained before, each processing unit will need to wait two cycles to start its i-indexed iteration after the one before it in the pipeline starts its own i-indexed iteration.

The delay and area results of the implementation of this design on Spartan-II FPGAs are presented and analyzed in Chapter 4.

3.5. Design of the Processing Element (Version 2)

This design of the processing element, shown in Figure 3.7 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 shifting is done by taking the LSB of the second CPA sum as the MSB of the S_out output while saving other sum bits to be output in the next cycle. 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). As said before in Section 3.2, 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 Chapter 4.

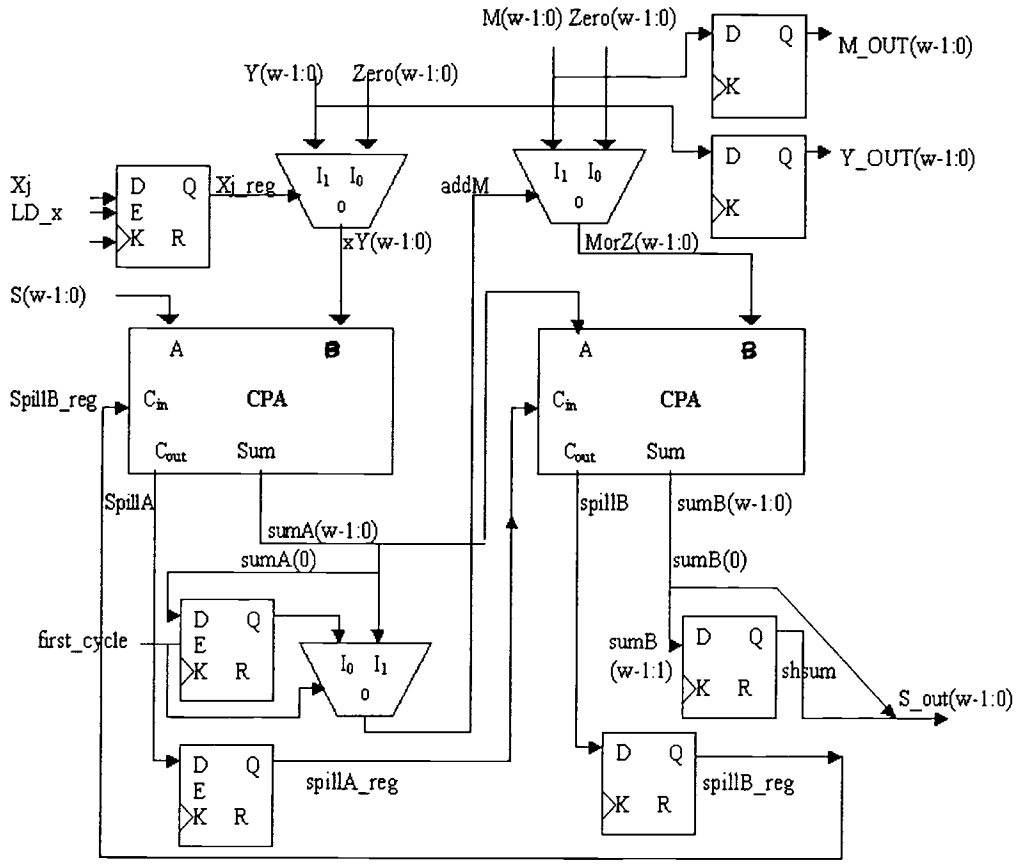

Figure 3.7. Design of MWR2MM processing element (version 2)

4. EXPERIMENTAL RESULTS AND ANALYSIS

In this Chapter, the implementation results for the two designs presented in Chapter 3 are shown and analyzed. We will the area and speed of the scalable radix-2 Montgomery multiplier pipeline designs as we change their configuration. As mentioned in Chapter 3, the pipeline configuration can be changed by changing the word size and the number of stages. Also, we identify the best design configuration for some selected operand sizes. We will compare the two designs based on their implementation results.

The results presented in this thesis are generated using Xilinx ISE4.1.03 design tools. VHDL has been used for design description. Xilinx Synthesis Technology (XST) has been used for synthesizing the VHDL designs. This synthesis tool is included in the ISE4.1.03. Also, ISE4.1.03 has the mapping, placement, routing, and configuration tools. Synthesis and implementation constraints have been provided to the design tools to help them optimize the design.

The target FPGA technology is Xilinx Spartan-II FPGAs that were described in Chapter 3. We are particularly interested in the XC2S200 FPGA chip because it is the one we have on the Digilab2 board. It has 28x42=1176 CLBs = 2352 slices, 284 user I/O pins, and 56 K bits of block RAM.

Most of the results presented here are those generated after placement and routing (PAR) phases are completed. The after-PAR results are more accurate and realistic than the results collected after synthesis only. However, other than the accuracy, we may be interested in examining the use of resources in the FPGA chip. If the design does not fit into the FPGA for lack of any resource like slices or I/O pins then it will not be implemented. Thus, no implementation results will be available, but the synthesis tool can still generate valid synthesis results. These synthesis results are generated to help the designer understand the requirements (i.e., the quantity and the type of FPGA resources) the design needs.

4.1. Scalable Radix-2 Montgomery Multiplier Pipeline (Version 1)

This scalable radix-2 Montgomery multiplier pipeline is composed out of processing elements from the first version presented in Chapter 3. 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 as we change its configuration.

4.1.1. Area

Figure 4.1 shows the area of the design versus the word size. The area is measured in slices. The word size is changed from 4 to 16 bits in steps of four. We stop at 16 bits is because there are not enough I/O pins and thus the pipeline design cannot be implemented. We would like to rely on implementation results not only synthesis. The number of stages is fixed to 28 so that we can see how the area of the design changes as we change the word size. This number of stages has been selected so that the design will take as many slices as possible when we reach to the last configuration. In fact, the configuration of 16-bit word and 28 stages takes 2350 slices, which is about 100% of the slices in the FPGA chip. The same approach will be used for the second design. This allows us to test the pipeline configurations over a wide range (i.e., small designs to big designs).

From Figure 4.1, we can see that the area increases almost linearly as we increase the word size. For each additional bit, the design needs about 6.3% of the slices (148 slices) on the average. It is expected since the length of the multiplexers, CSAs, and words registers is linearly dependent on the word size. Moreover, The optimization goal imposed to the tool was the speed, not the area.

Figure 4.2 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 us also that the area

increases almost linearly as the number of stages increases. For each additional stage, the design needs about 3.7% of the slices (87 slices) on the average.

4.1.2. Clock Cycle Time (CCT)

Figure 4.3 shows the clock cycle time versus the word size. For the first three configurations, the CCT is below 8 ns, i.e., the design can run on a frequency a little bit higher than 125 MHz. But for the 16-bit word size, it is 8% slower. We should recall that this last configuration belongs to the case when the design takes the whole chip. The placement and routing (PAR) tool optimizes the speed by trying many placement and routing configurations and then going for the one that gives the best performance. We should note that when the chip gets very crowded (i.e., the design become very large), it becomes more difficult for the tool to optimize the speed.

It is important to note that the 8% slowdown in the last configuration is not caused by the 16-bit word size but because the FPGA becomes very crowded. In fact, the 16-bit word size shows excellent performance (like those less than it) for 24 stages and less, as shown in Figure 4.4. From Figure 4.3, we can conclude that this design of 28 stages can run at frequencies little bit higher than 125 MHz for word size less than 14 bits. Also, speed is almost the same for these configurations. There are several reasons behind this. Before explaining them, we will explain how the design gets implemented in the FPGA. Any delay has two components: logic and Routing. The longest path of this design has three logic levels and two routing delays (LUT-routing-MUXF5-routing-LUT). In the first logic level, the logics of the $x_i Y^j$ product mux and the CSAs are merged into one combinational logic function and implemented in LUTs. The tool does this in the second phase of the synthesis (the low-level optimization phase). The too l can do this for some macros like 2-to-1 muxes if the

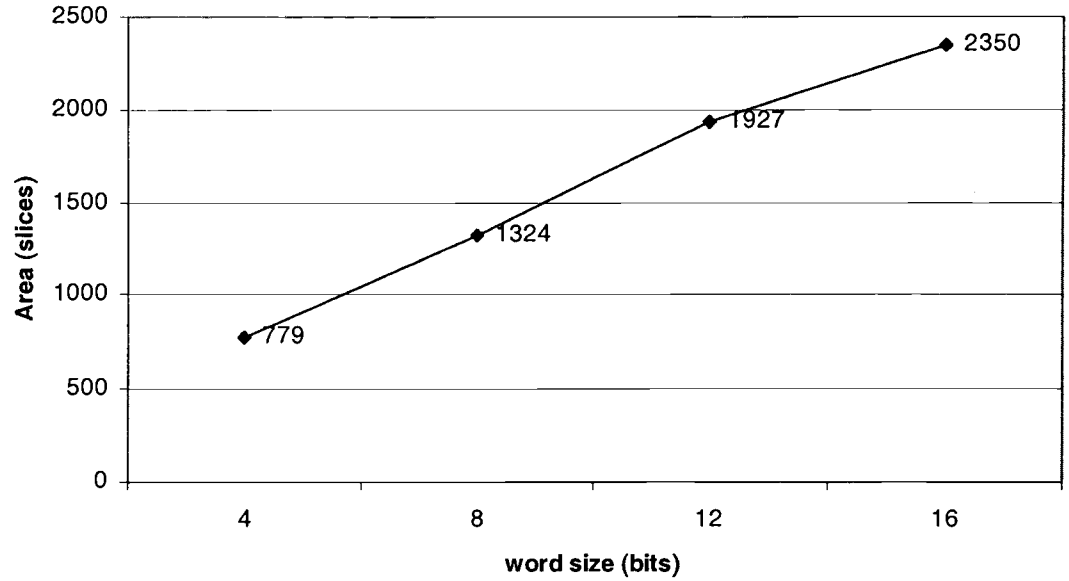

Figure 4.1. Version 1: Area vs. word size, number of stages = 28

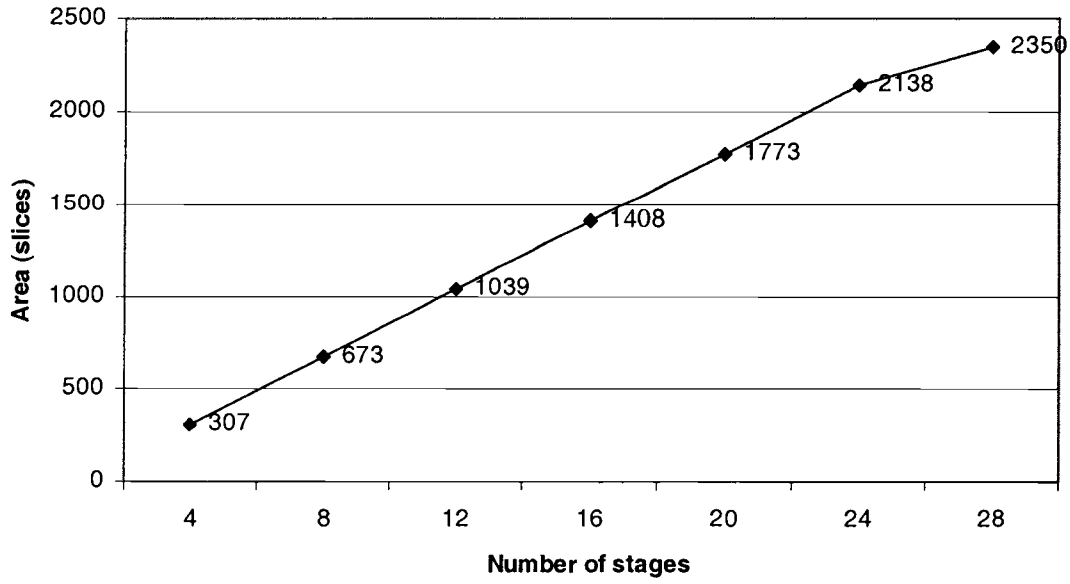

Figure 4.2. Version 1: Area vs. number of stages, word size = 16 bit

designer does not ask the synthesis tool to preserve the macros hierarchy when setting the synthesis constraints. In the second logic level, MUXF5 is used to implement the ODD test 2-to-1 mux. In the third level, similar to what happened in the first level happens here. The logic of the M^j-or-Zero 2-to-1 mux and the second group of CSAs are merged into one combinational logic function and implemented in LUTs. So, for this design the number of logic levels are fixed and don't change by changing the word size or the number of stages (as it will be shown).

As mentioned above, the other component of the delay is the routing. This can be improved by the tools in the synthesis and the PAR phases. More routing delay come from signals that need to be routed to far places in the chip, i.e., the shorter the route is the less the routing delay. To help the tools reduce the routing delay, the following techniques have been used:

- Setting the MAX_FANOUT synthesis constraints to 16. This will limit the delay caused by the high fan-out signals (signals that drive too many inputs like x_reg and addM. See the first design in Chapter 3).
- Enabling the REGISTER REPLICATION synthesis option. The synthesis tool can make many copies of the same register if needed to improve the speed.
- Asking the synthesis tool to optimize for speed.
- Setting the timing constraints. The PAR tool will try harder and harder until it meets our timing constraints, if it can.

In fact, the routing delay makes a major source of delay to this design. It is about 60% of the total delay while the logic delay is fixed at about 40%.

Figure 4.4 shows the clock cycle time versus the number of stages. This design shows almost the same performance for number of stages less than 26 (i.e., designs that take less than 92% of the total slices). For these configurations, the design can run at frequencies little bit higher than 125 MHz.

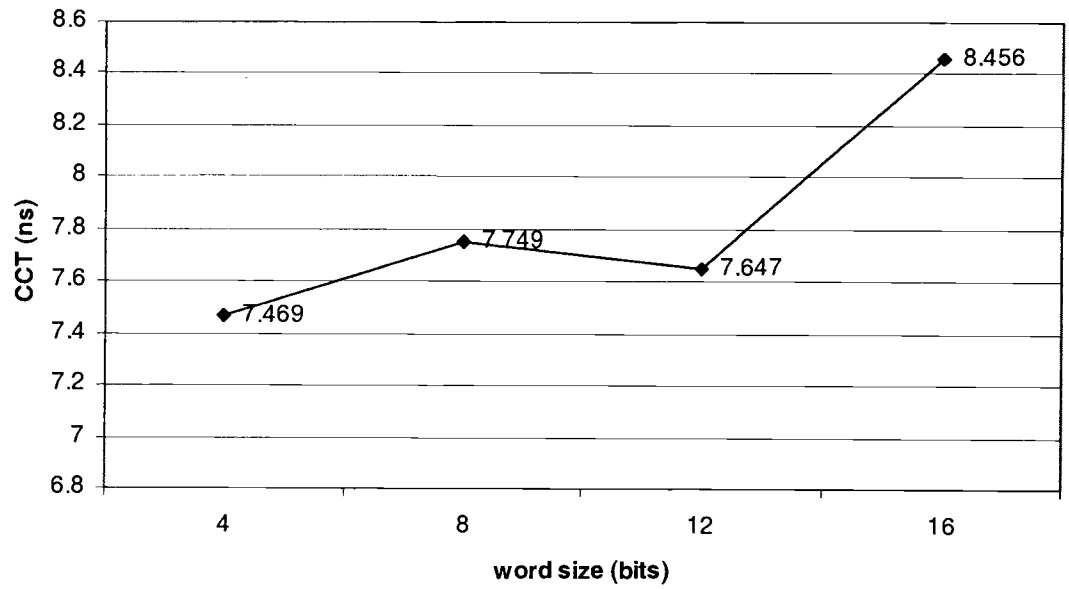

Figure 4.3. Version 1: Clock cycle time vs. word size, stages = 28

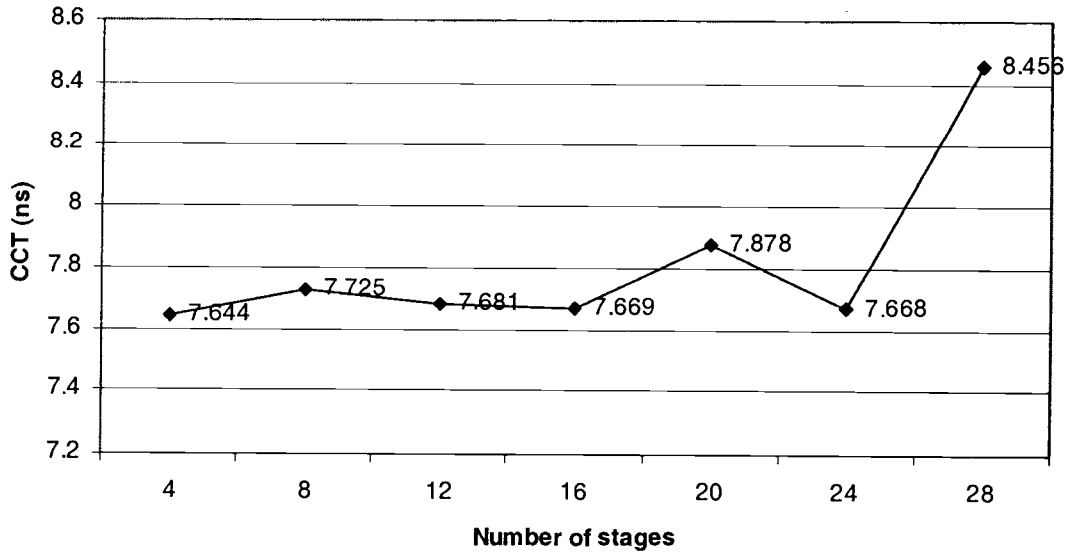

Figure 4.4. Version 1: Clock cycle time vs. number of stages, word size = 16 bit

4.1.3. Total Execution Time (TET)

It is important when trying to select the fastest configuration to remember that our objective should be the total execution time (TET). The TET is the time the pipeline needs to finish the MWR2MM algorithm presented in Chapter 3. The total execution time is calculated as

TET= C x CCT where C is the cycles count

It is shown in [10] that the cycle count (C) can be calculated as

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

More discussion on this equation was provided in [10]

The TET values for four operand sizes: 128, 256, 1024, and 2048 as we change the number of stages are shown in Table 4.1. the word size is set to 16 bits.

Stages	128-bit	256-bit	1024-bit	2048-bit
4	2247	8363	127575	504917
8	2032	4311	64380	255219
12	2082	4170	43106	169604
16	2017	4042	32133	126861
20	2261	4214	26927	104974
24	2262	4164	21785	85422
28	2427	4862	20793	81178

Table 4.1. Version 1: Total execution times (ns), word size= 16 bit

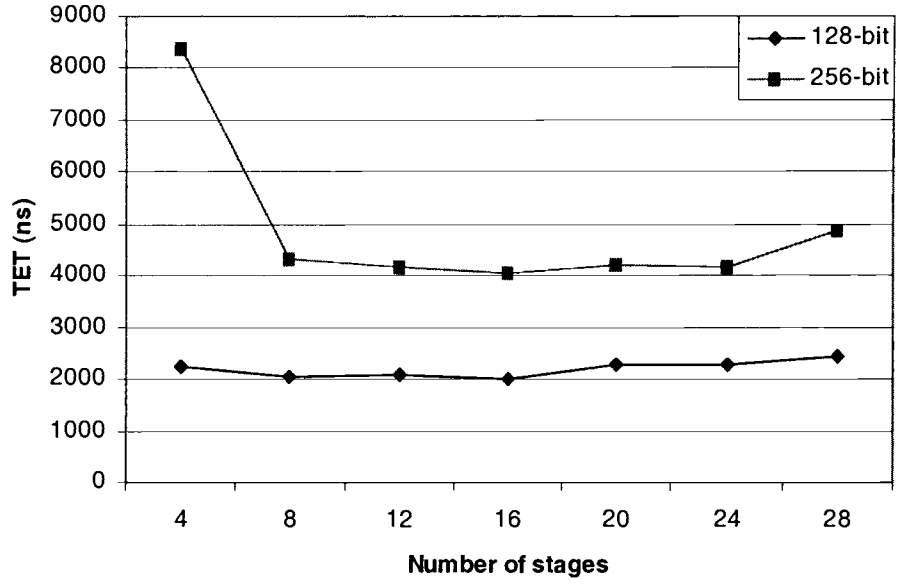

Figure 4.5. Version 1: Total execution time vs. number of stages, word size = 16 bit

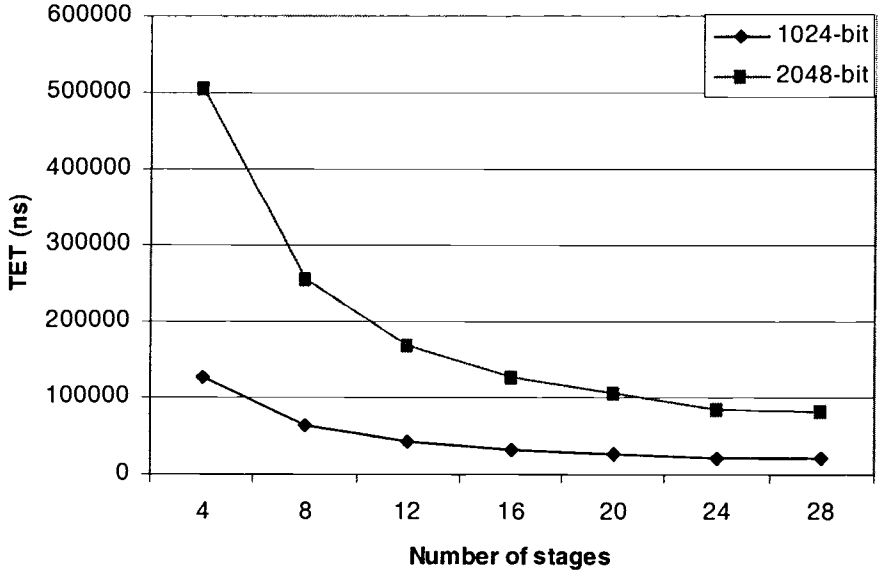

Figure 4.6. Version 1: Total execution time vs. number of stages, word size = 16 bit

Figures 4.5 and 4.6 show total execution time versus the number of stages. For 128-bit operand size, the minimum TET occurs when the number of stages is 16. But, we recommend 8 stages because we will loose only 0.7% in speed and we will gain 31% in area. Also, for 256-bit the minimum is at 16. 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 designs with more than 28 stages can be implemented then we expect that after some number of stages, increasing the number will not help and the design will be worse.

4.2. Scalable Radix-2 Montgomery Multiplier Pipeline (Version 2)

This scalable radix-2 Montgomery multiplier pipeline is composed out of processing elements from the second design version presented in Chapter 3. 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.

4.2.1. Area

Figure 4.7 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.

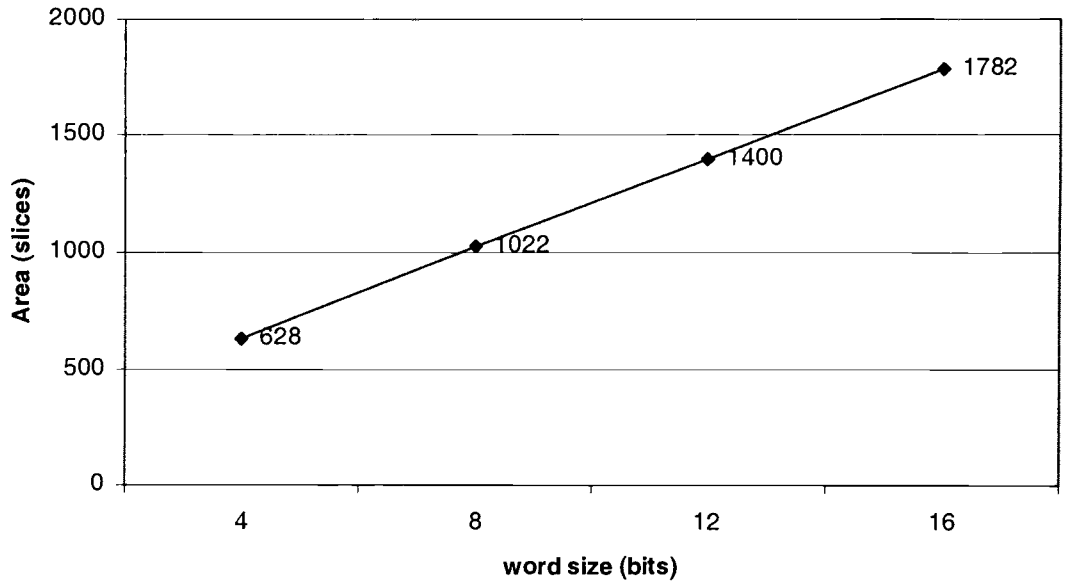

Figure 4.7. Version 2: Area vs. word size, number of stages = 28

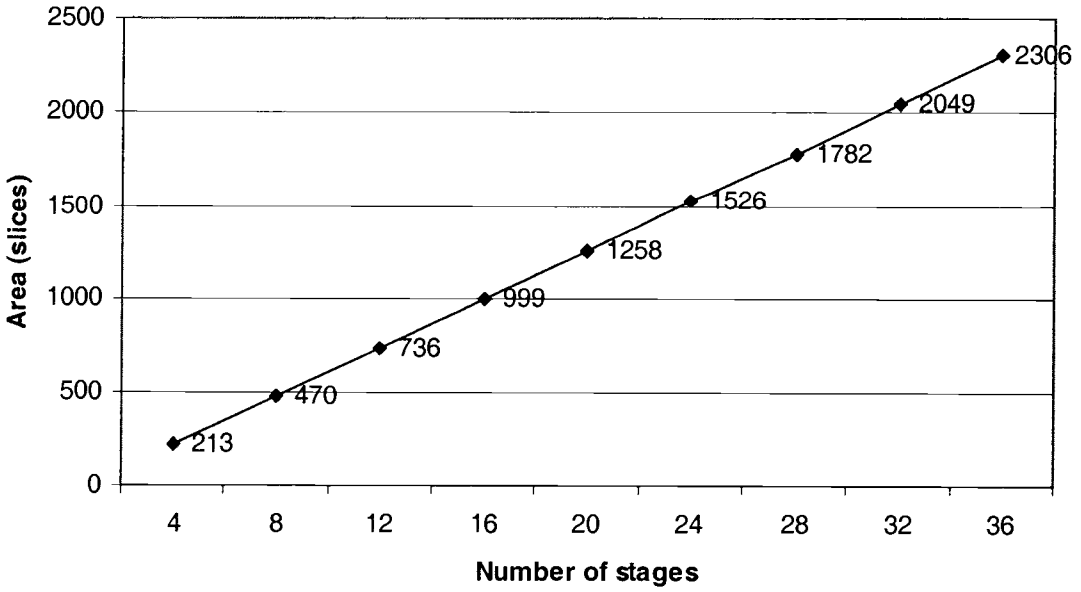

Figure 4.8. Version 2: Area vs. number of stages, word size = 16 bit

From Figure 4.7, 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 4.8 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.

4.2.2. Clock Cycle Time (CCT)

Before we study how the CCT changes for this design, we need to remember from Chapter 3 that this design uses Xilinx CPAs available in the chip. 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 in optimizing the 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 4.9 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

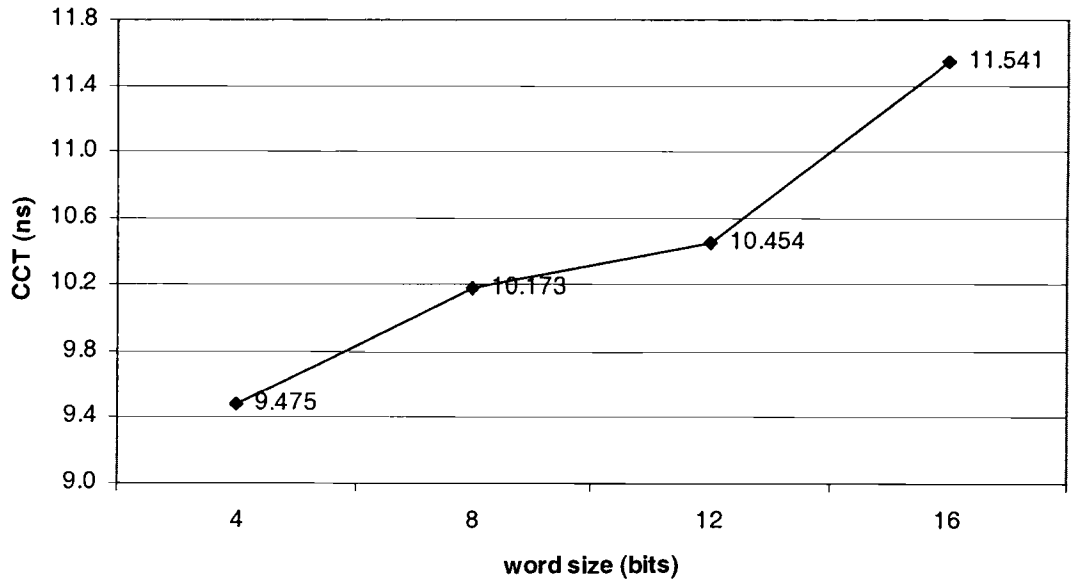

Figure 4.9. Version 2: Clock cycle time vs. word size, number of stages = 28

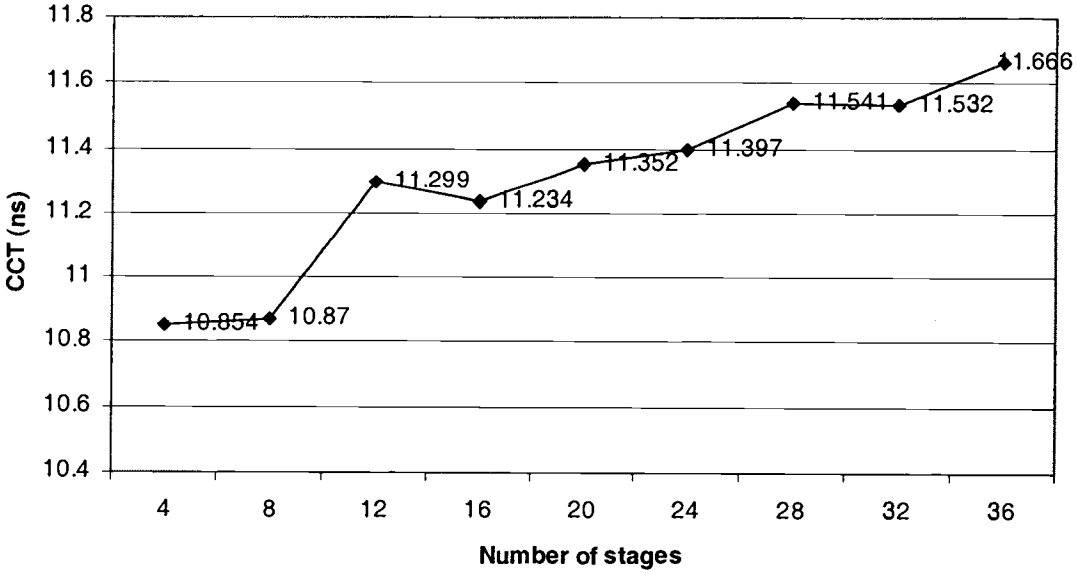

Figure 4.10. Version 2: Clock cycle time vs. number of stages, word size = 16 bit

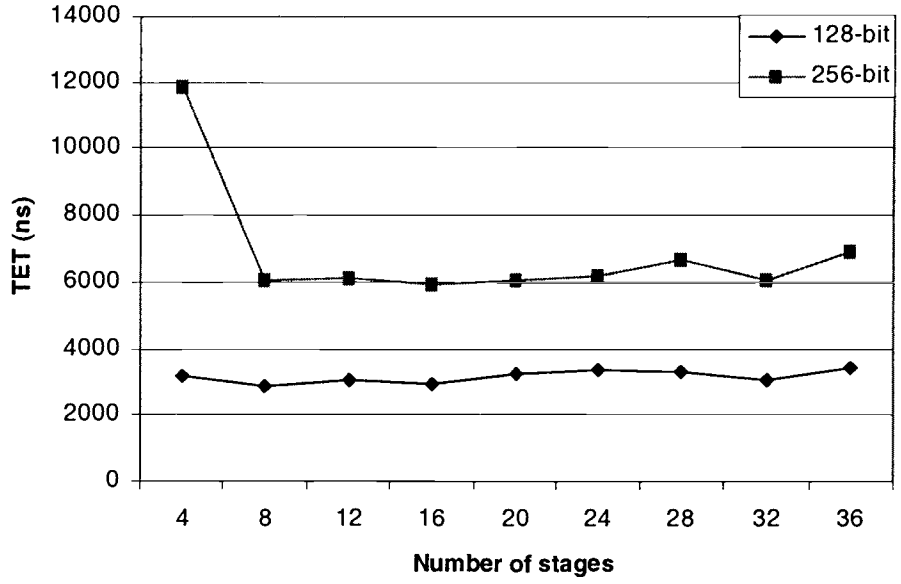

Figure 4.11. Version 2: Total execution time vs. number of stages, word size = 16 bit

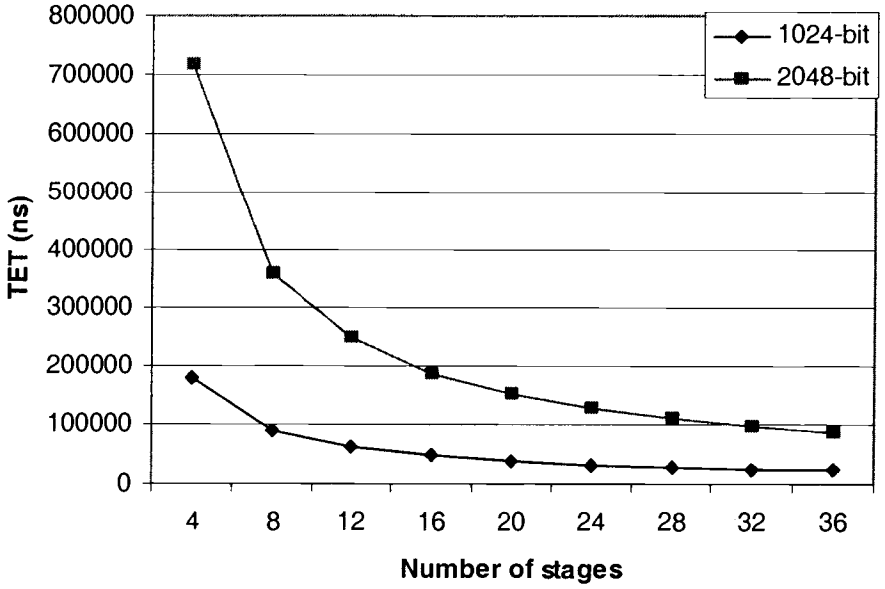

Figure 4.12. Version 2: Total execution time vs. number of stages, word size = 16 bit

routing. The logic delay is about 60% of the total delay while the routing is about 40%.

Figure 4.10 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, mentioned in Section 4.1.2, were also used here and helped the tools during the optimization process.

4.2.3. Total Execution Time (TET)

The fastest configuration is identified by the total execution time (TET), as explained in Section 4.1.3. The TET values for four operand sizes: 128, 256, 1024, and 2048 as we change the number of stages are shown in Table 4.2. The word size is set to 16 bits.

Figures 4.11 and 4.12 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	151265
24	3362	6189	32379	126963
28	3312	6636	28379	110794
32	3033	6077	24702	95923
36	3441	6895	25094	86597

Table 4.2. Version 2: Total execution times (ns), word size = 16 bit

4.3. Comparison Between The Two FPGA Implementations

In the following three subsections, we compare the two FPGA implementations of the scalable radix-2 Montgomery multiplication pipeline. As we said above, the comparison will be regarding the area, clock cycle time (CCT), and the total execution time (TET). The Final judgment between them will be based on the total execution time because this is the one that affects the modular multiplication and exponentiation times.

4.3.1. Area

From Figure 4.13, we can conclude that the area of both designs increase almost linearly as we increase the word size. But, the second design takes less area than the first one. The first design uses carry-save adders (CSAs), which requires more registers for storing the carry vector of the result word (SC^j) in the processing element

itself. Also, it requires more pipeline registers. Whereas, the second design uses CPAs where the result is represented in the conventional non-redundant form, which requires less registers and logic, and thus less area. The gain in area from using the second design ranges from 6% to 24% (about 5% for each 4-bit increase in the word size which is 1.25% per word). Also, we can conclude from Figure 4.14 that areas of both designs increase almost linearly as the number of stages increases. But, the second one increases less. The gain in area from using the second design is about 1% per stage.

This gain can be useful only if we need to make the design as small as possible and we can afford to give up some of the computation speed. We'll talk more about this when we study the total execution time of both designs as a function of area.

4.3.2. Clock Cycle Time (CCT)

We can conclude from Figure 4.15 that the first design is faster than the second design in terms of clock cycle time. The tested configurations of the first design can run on frequencies from 118MHz to 133MHz, whereas the tested configurations of the second design can run on frequencies from 86MHz to 105MHz. Another important conclusion is that the first design is more immune to the change in word size than the second design. The CCT for the first design stays below 8 ns until 14-bit word size. In fact, it stays below 8 ns even for the 16-bit word size but when the number of stages is less than 26 (i.e., the design takes less than 95% of the slices). See Figure 4.16. The main reason behind these two results is that the first uses CSAs and the second uses fast CPAs. Thus, the logic level is fixed for the first design while it is increasing for the second design. Another reason is that the PAR tool does a better job for the first design than for the second design because it has to place the carry chains in

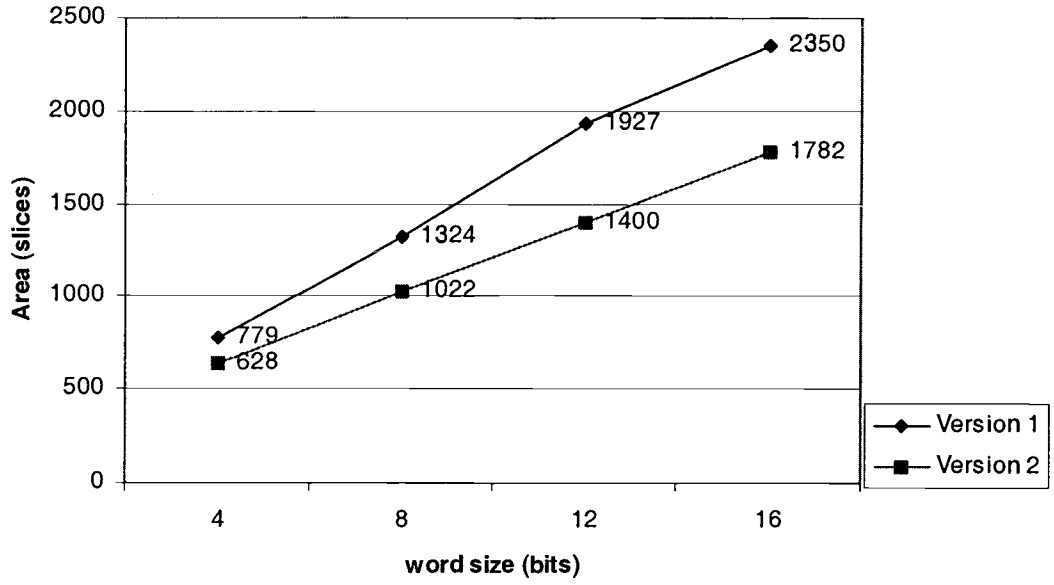

Figure 4.13. Version 1 and Version 2: Area vs. word size, number of stages = 28

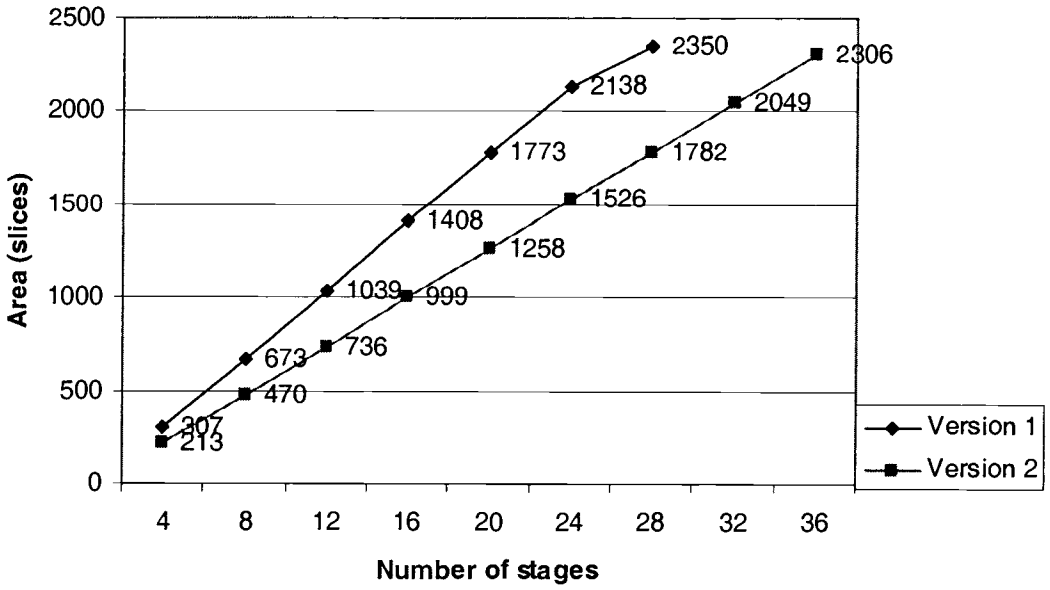

Figure 4.14. Version 1 and Version 2: Area vs. number of stages, word size = 16 bit

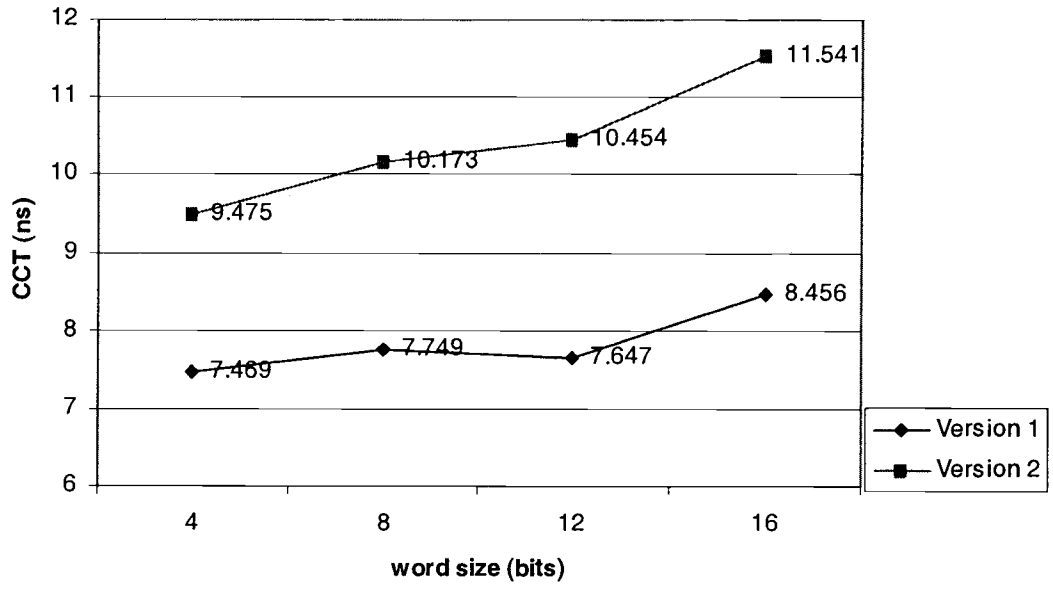

Figure 4.15. Version 1and Version 2: CCT vs. word size, stages = 28

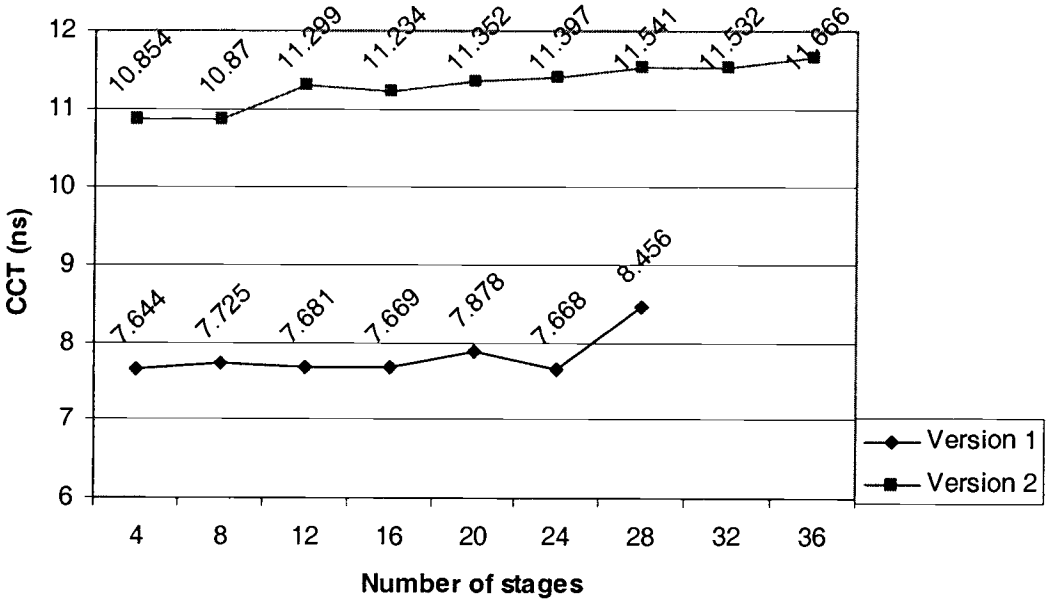

Figure 4.16. Version 1and Version 2: CCT vs. number of stages, word size = 16 bit

adjacent slices in the second one. In the first design, the logic delay is about 40% of the total delay while the routing delay is about 60%. But in the second design, the logic delay is about 60% of the total delay while the routing delay is about 40%. This is also because of the two reasons mentioned here.

From Figure 4.16 we can conclude also that design one is faster and more immune to the increase in the number of stages. The first design can run on frequency a little higher than 125 MH for the configurations that take less than 95% of the slices. On the other hand, the second design CCT is slightly increasing as we increase the number of stages. Although the second design is not as immune as the first one to the increase in word size and number of stages, it is still showing very good immunity. This very good immunity comes from the use of the fast carry logic (FCL) available in the FPGA chip and from the good synthesis and PAR optimization techniques supported by the tools.

4.3.3. Total Execution Time (TET)

Figures 4.17, 4.18, 4.19, and 4.20 show the total execution times versus the number of stages for version 1 and version 2 simultaneously, for the operand sizes 128, 256, 1024, and 2048, respectively. The first design is always faster than the second for the same number of stages. Both of them show similar behavior though.

For 128-bit operand, the first design needs about 2 μs to execute the algorithm when using 8 stages (8 is recommended because the minimum is at 16 and it is only 0.7% faster). The second design has its minimum when using 8 stages and needs about 2.8 μ s. So, it is about 0.8 μ s slower. For both of them, the curve is almost flat between 8 and 16 stages. After 16, it starts to increase. For the recommended configurations, the second design is 29% slower and 9% smaller.

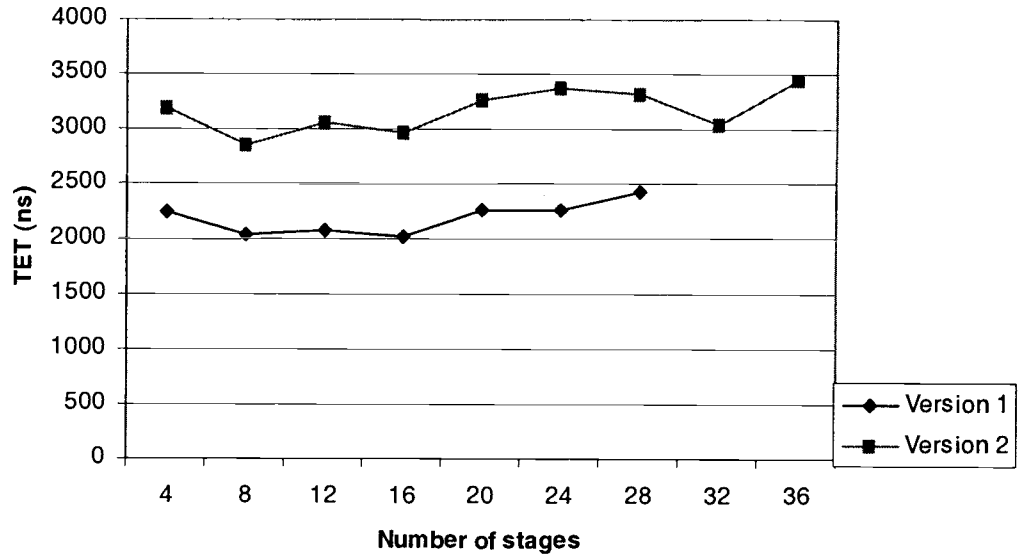

Figure 4.17. Version 1and Version 2: TET vs. number of stages, word size = 16 bit, 128-bit operand

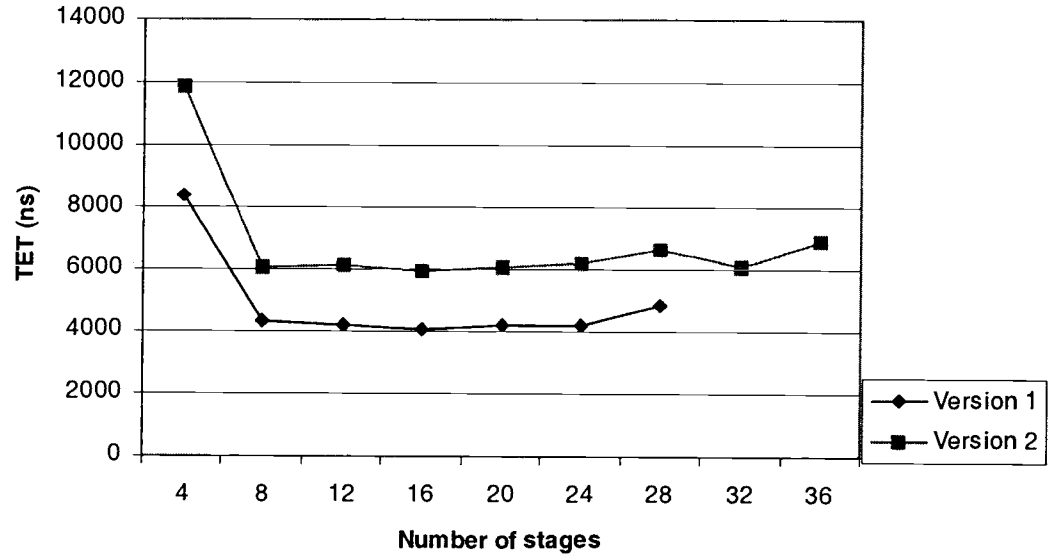

Figure 4.18. Version 1and Version 2: TET vs. number of stages, word size = 16 bit, 256-bit operand

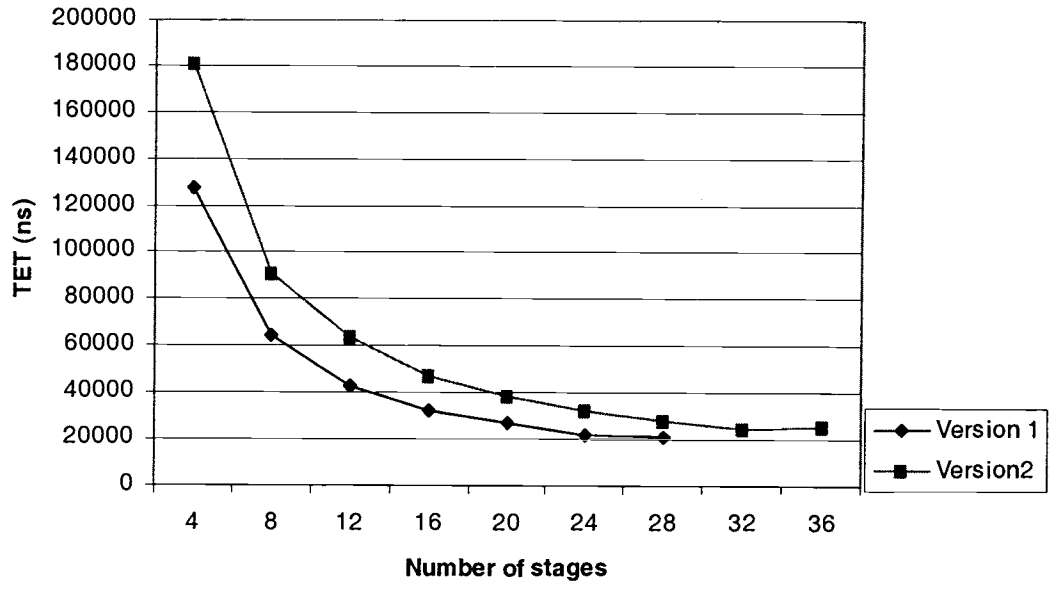

Figure 4.19. Version 1and Version 2: TET vs. number of stages, word size = 16 bit, 1024-bit operand

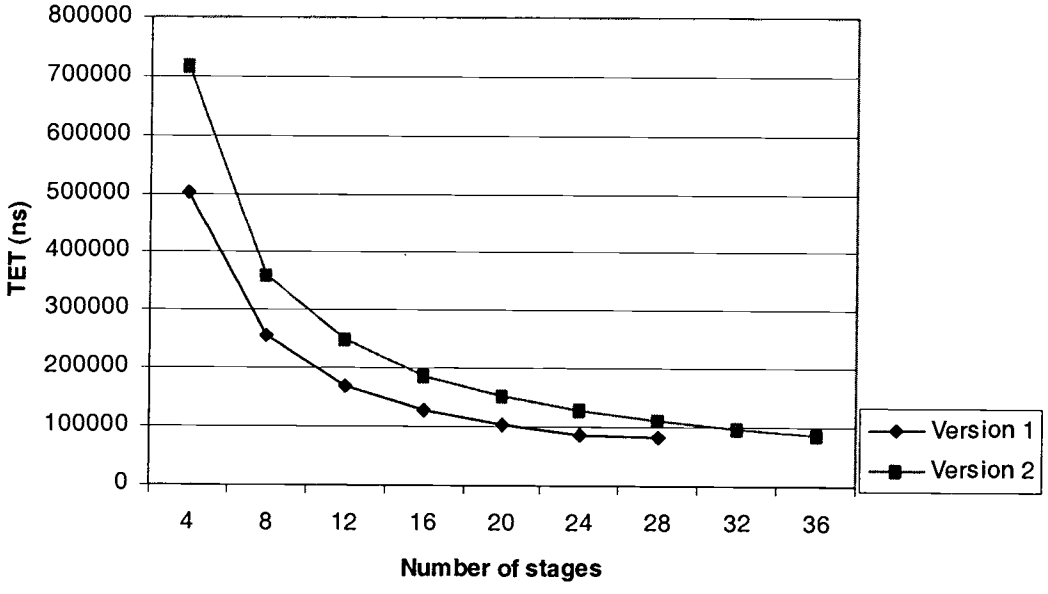

Figure 4.20. Version 1and Version 2: TET vs. number of stages, word size = 16 bit, 2048-bit operand

For 256-bit operand, the first design has its minimum when using 16 stages and needs about 4 μ s to execute the algorithm. The second design has its minimum also when using 8 stages and needs about 5.9 μs. So, it is about 1.9 μ s slower. For both of them, the curve is almost flat between 8 and 24 stages. After 24, it starts to increase. For these best configurations, the second design is 32% slower and 17% smaller.

For 1024-bit operand, the first design has its minimum when using 28 stages and needs about 20.7 μ s to execute the algorithm. The second design has its minimum when using 32 stages and needs about 24.7 μs. So, it is about 4 μ s slower. Increasing the number to 36 stages makes it slower (TET is 25.1 μs). We expect the same thing to happen to the first design if more stages can be implemented. For these best configurations, the second design is 16% slower and 13% smaller.

For 2048-bit operand, both designs have their minimum TET values when using the maximum possible number of stages (28 for the first and 36 for the second). For these large configurations, the first design can execute the algorithm in about 81.2 μ s while the second design can execute it in about 86.6 μ s. For these best configurations, the second design is 6% slower and 2% smaller.

From the previous comparison regarding the algorithm total execution time, we can conclude that the second design should be used only if we have very limited area and we can give up some of the computation speed.

5. CONCLUSIONS AND FUTURE WORK

In this Chapter, we will compare the FPGA implementation of the first design with the ASIC implementation of the same design presented in [10]. Of course, it does not make sense to compare in regard of absolute quantities. Our approach will be to compare them based on their behavior. So, we will concentrate on how they change not how much they change, as we change their configuration. Then, we will compare fixed design of 1024-bit Montgomery multiplier (uses single big processing element) versus the scalable one (uses 32 processing elements of 32-bit word size each). This comparison will be based on post-synthesis results not post-PAR because the scalable cannot be implemented in the largest Spartan-II chip (the one we have). In the end, we will conclude what we have accomplished in this work and suggest some future work based on it.

5.1. Qualitative Comparison With ASIC Implementation

In this section, we will compare our FPGA implementation of the second design against the ASIC implementation of similar design presented in [10]. As we said earlier, we will approach the comparison in qualitative manner not quantitative manner. ASIC and FPGA are two different technology and they have different design methodologies. For the work presented in [10], Mentor Graphics tools has 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.

5.2. 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 caused by 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.

5.3. Concluding Remarks

In this work, we have an FPGA-based prototyping environment that can be used to test the functionality of the Montgomery multiplier (MM) hardware at the circuit level. Software applications running on the host computer can use this environment to operate the MM hardware. 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) for FPGAs. We have come 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, which has been presented in Section 5.2 of this Chapter, 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.

5.4. Future Work

Based on the experiments and the work we have done in this thesis, we suggest the following things to be done and examined in the future.

A cryptographic application should be built on the top of the prototyping environment we have developed in Chapter 2. This application will need to make use of the EPP2MM_driver routines that operate the EPP2MM driver circuit. We've shown in Chapter 2 that 375 Kbps data rate can be achieved using this environment. This application should be used to test the Montgomery multiplier that was presented in Chapter 2 also.

The fixed Montgomery multiplication should be examined and implemented on other FPGA and ASIC technologies, especially in the situations where large area is available to it. The operands can be fed to its registers word-by-word so that it does not need large number of pins and high bandwidth.

Other FPGA technologies that have special resources, which may be useful, should be considered. Implementation of Montgomery multiplication, in particular, and other Modular multiplication, in general, on such FPGA chips should be done. For example, Xilinx Virtex-II Pro FPGAs have IBM PowerPC 405 RISC core embedded inside the chip, on-chip memory controllers, block RAMs, and 18 bit x 18 bit Block multipliers. Such large FPGAs, gives the designer a lot of logic, arithmetic, processing, control, and storage resources.

Much more research and effort should be dedicated to the way the scalable Montgomery multiplier receives and sends the data. The I/O unit should be as generic as possible so that it works for any maximum operand size and any good word size. The reason we are saying "any good" not "any" only is because the pipeline needs to read the words from and write the result words to the I/O memory. If we want the control unit to work for any word size then this will increase the complexity of the I/O memory and the multiplier control circuitry. We believe power-of-two word sizes should make the I/O memory and the multiplier control circuitry simpler.

BIBLIOGRAPHY

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

2. 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.

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

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

5. 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.

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

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

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

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

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

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

12. R. Galli, "Design and Evaluation of On-line Arithmetic Modules and Networks for Signal Processing Applications on FPGAs," MS thesis, Oregon State University, June 2001.

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

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

15. A. Bernal and A. Guyot. Design of a modular multiplier based on Montgomery's algorithm. In 13th Conference on Design of Circuits and Integrated Systems, pages 680{685, Madrid, Spain, November 17{20 1998.

16. D. Naccache and D. M'Raihi. "Cryptographic smart cards". *IEEE Micro*, 16(3):14–24, June 1996.

17. H. Orup. Simplifying quotient determination in high-radix modular multiplication. In S. Knowles and W. H. McAllister, editors, *Proceedings, 12th Symposium on Computer Arithmetic*, pages 193–199, Bath, England, July 19–21 1995. IEEE Computer Society Press, Los Alamitos, CA.

18. N. Takagi, "A radix-4 modular multiplication hardware algorithm for modular exponentiation," IEEE Transactions on Computers, vol. 41, no. 8, pp. 949-956, August 1992.

19. 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.

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

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

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

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

24. A.F. Tenca, M.D. Ercegovac, and T.Lang, "Class notes: ECE577 - computer arithmetic," Oregon State University, 2001.

25. A.F. Tenca and M.D. Ercegovac, "A multiplier design for variable long-precision computations," Proceedings of 31st Asilomar Conference on Signals, Systems and Computers, November 1997.

26. B. Parhami, Computer Arithmetic, Algorithms and Hardware Design, Oxford University Press, 2000.

27. J.R. Armstrong and F.G. Gray, VHDL Design Representation and Synthesis, Prentice Hall, 2000.

28. Xilinx, "XC4000 series field programmable gate arrays," Xilinx Databook (http://www.xilinx.com), May 1999.

29. Xilinx, "Spartan-II series field programmable gate arrays," Xilinx Databook (http://www.xilinx.com), March 2001.

30. Xilinx, "Vertix-II Pro series platform field programmable gate arrays," Xilinx Databook (http://www.xilinx.com), June 2002.

31. R. Pragasam, "Spartan FPGAs - The gate array solution," Xilinx Application Note XAPP120, August 2001.

32. Xilinx, "Using Xilinx Programmable Logic with High-Speed Printers," Xilinx Application Note XAPP142, July 2002.

33. Xilinx, "Using Block SelectRAM+ Memory in Spartan-II FPGAs," Xilinx Application Note XAPP173, December 2000.

34. Xilinx, "High speed FIFOs in Spartan-II FPGAs," Xilinx Application Note XAPP175, November 1999.

35. Xilinx, "Spartan-II FPGA Family Configuration and Readback," Xilinx Application Note XAPP176, December 1999.

36. B. New, "Using the dedicated carry logic in XC4000E," Xilinx Application Note XAPP013, July 1996.

37. Wrap Nine Engineering, "IEEE 1284 Enhanced Parallel Port Protocol (EPP)," (http://www.fapo.com/ieee1284.htm), July 2002.

38. Digilent, "Digilab 2 Reference Manual," (http://www.digilentinc.com), May 2002.

APPENDIX. EPP2MM DRIVER SOURCE CODE

```
////////////////////////////////////////////////////////////////////////
//
//  epp2mm_driver.cpp    --  EPP2MM Driver
//
//    Author:            Khaldoon Mhaidat
//    Copyright 2002, Oregon State University
//
////////////////////////////////////////////////////////////////////////
//
//  Module Description:
//
//  This module contains functions used to operate the epp2mm
//  interface circuit. The writing function is used to write a 32-bit
//  word of data to the specified Montgomery Multiplier register
//  through the EPP port. The reading function reads 32-bit word of
//  data from the specified MM register through the EPP port.
//
////////////////////////////////////////////////////////////////////////


#include <windows.h>
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include "gendefs.h"
#include "epputil.h" // SPP and EPP interface routines
                     // available from Digilent, Inc.


WORD        iopPort=0x378;    // parallel port to use


////////////////////////////////////////////////////////////
// epp2mm_write
// input:
//       address    : determines the writing location
//                    values: [0-15]
//       data       : 32-bit data word to be written to
//                    the MM register
//
// output:
//       none
//
// Description:
//   This is the writing function of the EPP2MM driver
////////////////////////////////////////////////////////////
```

```
void epp2mm_write(unsigned short address, unsigned long data)
{
      unsigned int bytes[4];

      // Parse the unsigned long integer into four integers

      for( int i=0; i<4; i++){

            bytes[i]= data & 0x000000FF;
            data= data>>8;
      }

      // Write the four bytes to the EPP data port

      for(i=0; i<4; i++){

            OutEppData(iopPort, bytes[i]);

      }

      // Write the address and control sequence to the
      // EPP address port

      address = address & 0x000F ;
      address = address | 0x0080 ;
      OutEppAddr(iopPort, address);
      address = address & 0x000F ;
      OutEppAddr(iopPort, address);

}



////////////////////////////////////////////////////////////
// epp2mm_read
// input:
//       address    : determines the reading location
//                    values: [0-15]
//
// output:
//       data       : 32-bit data read from the MM register
//
// Description:
//   This is the reading function of the EPP2MM driver
////////////////////////////////////////////////////////////

unsigned long epp2mm_read(int address)
{

      unsigned short bytes[4];
      unsigned long data = 0;
```

```
        // Write the address and control sequence to the
        // EPP address port

        address = address & 0x000F ;
        address = address | 0x0040 ;
        OutEppAddr(iopPort, address);
        address = address & 0x000F ;
        OutEppAddr(iopPort, address);


        // Read the four bytes from the EPP data port

        for( int i=0; i<4; i++){

              bytes[i] = FbInEppData(iopPort);

        }


        // Assemble the four integers into unsigned long integer

        for( i=0; i<4; i++){

              data = data<<8;
              data = data + bytes[3-i] ;

        }


        return data;
}
```