AN ABSTRACT OF THE THESIS OF

Song Jun Park for the degree of Master of Science in
Electrical & Computer Engineering presented on June 6, 2005.
Title: Hardware Design of Scalable and Unified Modular Division and Montgomery Multiplication

Abstract approved: _____

Çetin Kaya Koç

Modular arithmetic is a basic operation for many cryptography applications such as public key cryptography, key exchange algorithms, digital signatures, and elliptic curve cryptography. Therefore, fast and efficient hardware design of modular division and multiplication is proposed for implementation of cryptography system with intent to achieve information security. The hardware unit is capable of computing modular division and Montgomery multiplication in both prime $GF(p)$ and binary extension $GF(2^n)$ fields. Carry-save representation is adopted to create a precision independent hardware and to reduce critical path delay. However, carry-save vectors introduce issues of unsafe arithmetic right shift and ambiguous zero detection, which are investigated and resolved. Additionally, in order to alleviate the increase in area resulting from using carry-save representation, a scalable architecture is developed for the hardware design.

Hardware Design of Scalable and Unified Modular Division and Montgomery Multiplication

by

Song Jun Park

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 6, 2005
Commencement June 2006

Master of Science thesis of Song Jun Park presented on June 6, 2005

APPROVED:

Major Professor, representing Electrical & Computer Engineering

---

Director of the School of Electrical Engineering & Computer Science

---

Dean of the 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.

Song Jun Park, Author

Master of Science thesis of Song Jun Park presented on June 6, 2005

APPROVED:

Major Professor, representing Electrical & Computer Engineering

---

Associate Director of the School of Electrical Engineering & Computer Science

---

Dean of the 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.

Song Jun Park, Author

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

LIST OF TABLES

Hardware Design of Scalable and Unified Modular Division and Montgomery Multiplication

1. INTRODUCTION

Digital systems have become pervasive in modern society and the technology continues to grow and evolve. Similarly, the ubiquitous internet has experienced a tremendous advancement in recent years. In conjunction with the internet, the number of personal computing applications continues to grow with endless possibilities. With such a gain of popularity in digital technology, the issue of information security must be addressed with great importance. Strong security is necessary and crucial element for the functionality of data sensitive applications such as E-commerce, internet banking, and filing documents electronically.

The field of cryptography provides innovative and standardized techniques to achieve information security. Viewing from the highest level of abstraction, the four main objectives in cryptography refers to confidentiality, data integrity, authentication, and non-repudiation [1]. Confidentiality protects the message content, data integrity ensures untempered data, authentication confirms message origin, and non-repudiation proves the ownership. Many cryptographic applications, which attempt to meet the above criteria, involve modular arithmetic or congruences at the core of the operation. Therefore, a unit capable of computing high performance modular arithmetic is an appealing candidate as an implementation for modern cryptography systems.

1.1. Motivation

Modular arithmetic is a fundamental arithmetic operation that applies to wide range of cryptographic application. In particular, RSA, ElGamal, Diffie-Hellman key exchange, digital signatures, and elliptic curve cryptography operate in modular arithmetic [1]. Moreover, RSA and elliptic curve cryptography require inversion computation along with modular multiplication [2]. Coupled with the rapid advancement in computing power and popularity, calls for fast and efficient modular arithmetic core in order to provide strong data security.

The scalable and unified modular division and Montgomery multiplication unit is the proposed hardware as a modular arithmetic core. The design is able to compute both modular division and Montgomery multiplication over the finite fields, $GF(2^n)$ and $GF(p)$. The scalable and unified characteristics of the design offer portability, area flexibility, precision independent delay, and dual field operation.

The internal design of the hardware operates in carry-save representation. Carry-save adder is simply a collection of parallel and independent full adders [3], as shown in Figure 1.1. The basic carry-save adder accepts three input operands and generates two output results, which can be viewed as a 3 to 2 reduction [4]. The logic behind using carry-save or redundant representation is to assure a constant delay time regardless of input vector length since carry bits are not interconnected. Carry-save notation requires a carry vector and a sum vector to represent a single value, effectively doubling the storage requirement. In addition, two full adders are needed to perform an addition of two vectors in carry-save form. In order to alleviate the area overhead introduced by using carry-save

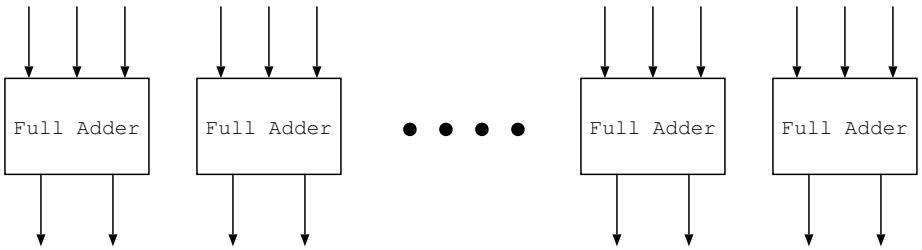

FIGURE 1.1. Carry-Save Adder

vectors, scalable implementation was developed for the hardware design. The logic and description of scalable architecture is described in [5, 6].

1.2. Previous Work

This thesis work is a continuation of research which has been initiated by [7]. A novel algorithm for unified modular multiplication/division was developed and published in [7, 8]. Modular division algorithm has a strong relationship with algorithms for computing greatest common divisor (GCD) [9].

The Euclidean algorithm computes a GCD by employing a recursive integer division [10]. In order to eliminate the integer division in the Euclidean algorithm, binary GCD algorithm is devised in [11]. Binary GCD algorithm is more suitable for binary machines because it solely relies on parity test, subtraction, shift operation, and comparison. To further optimize for binary arithmetic, plus-minus algorithm for computing GCD is presented in [12], which avoids a time consuming comparison process.

By reversing the steps of the Euclidean GCD calculation, modular inverse can be computed. Similarly, it is possible to compute modular inverse by adding auxiliary computations while calculating GCD, known as the extended Euclidean algorithm. As the Euclidean algorithm was extended to compute multiplicative

inverse, algorithm for computing modular division can be derived from modification of a modular inverse algorithm.

1.3. Contribution

A brief overview of the thesis work is described in this section. First, the hardware implementation for the unified modular division and Montgomery multiplication algorithm was designed and written with the aid of Very High Speed Integrated Circuit Hardware Description Language (VHDL). Then, the design was enhanced to support the scalable definition. The scalable architecture offers flexibility to control the trade off between area and delay such that the hardware can be tailored to satisfy wide range of different specifications. The scalable design is able to compute any precision size, limited by available memory space.

Challenges of the design revolved around carry-save notation, a redundant number representation that was adopted for the internal number system. The design issues include unsafe arithmetic right shift and multiple zero detection of carry-save vectors. As the name suggests, several combination of numbers in redundant number system can map to a same number. Therefore, the true value of a number in carry-save format is hidden and difficult to evaluate.

The functional verification of the hardware implementation was rigorously simulated by randomly generated test vectors that consist of over one million different values. Hence the correctness of the design can be validated with confidence. Finally, the Practical Extraction Report Language (PERL) script was used to generate various precision synthesis scripts which were synthesized to obtain area and delay information.

1.4. Thesis Organization

Current chapter provides the introduction and the background information regarding the proposed hardware. Next chapter investigates the problem associated with working in carry-save number system. The chapter 3 elaborates on the hardware implementation of scalable and unified modular division and Montgomery multiplication algorithm with detailed explanations of each individual hardware components. Then, the chapter 4 devotes to simulation and the analysis of synthesized results. Finally, the chapter 5 concludes this thesis presentation.

2. CARRY-SAVE ISSUES

2.1. Right Shift of Carry-Save Vectors

A carry-save adder simply passes a carry signal to the output. This reduces the critical path delay by eliminating the dominant delay of carry propagation, which is apparent in traditional carry-propagate adders. For additions involving high precision, the use of carry-save adder dramatically reduces the delay time. Disadvantages of using carry-save representation would be the increase in storage requirement, a final conversion needed to revert back to the non-redundant representation, and difficulty of sign and magnitude detection [13]. The following chapter discusses the obstacles relating to working with carry-save vectors and presents corresponding solutions.

2.1.1. Problem Definition

Redundant number system was chosen as the internal format for the design to maximize speed and to create a precision independent architecture. Each number is represented in carry-save format, which is composed of a carry vector and a sum vector. To reiterate, a single possible representation of a decimal value can be expressed by many different sets of sum and carry vectors in carry-save domain. For example, a binary value of six or $(0100)_2$ can be written in carry-save notation as $(0010, 0010)_2$ or $(1100, 1000)_2$ where left binary vector refers to a carry vector, right binary vector refers to a sum vector, and the subscripts denote the respective radix. To compute an addition of two numbers represented in a carry-save notation, two full adders are required where the interconnection is shown in Figure 2.1. Two full adders can also be viewed as performing a 4 to 2 reduction.

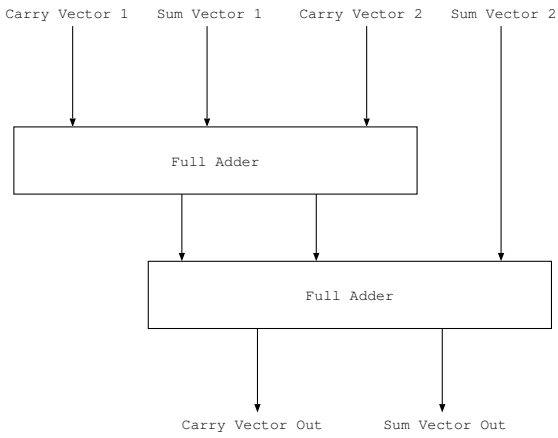

FIGURE 2.1. Carry-Save Addition

Carry-save shift problem is associated with the division portion of the algorithm. In order to comprehend the source of the problem, the flow of the simplified division algorithm must first be understood. The steps of the division algorithm consist of addition of two distinct carry-save vectors followed by an arithmetic right shift. The problem resides on the arithmetic right shift operation of the carry-save vectors. Simply sign extending the resulting carry-save vectors during this right shift operation can lead to an incorrect result.

The arithmetic right shift error was encountered during the debugging process of the design. Sign bits or the most significant bits (MSB) failed to match with the expected result. Upon close examination, it was discovered that the redundant property of carry-save representation imposed a dilemma in determining the MSB that is to replace the current MSB during arithmetic right shift operation. The right shift problem can most easily be demonstrated using an example which is shown in Table 2.1.

As it can be observed from Table 2.1, conventional sign extension fails to comply with the expected answer. This deviation from the correct result portrays

	Original Value	Sign Extend Right Shifted Value	Expected Value	Match
Decimal Value	-4	-2	-2	Yes
Non-redundant Representation				
Binary Vector	1100	1110	1110	Yes
Redundant Representation				
Carry Vector	0110	0011	0011	No
Sum Vector	0110	0011	1011	

TABLE 2.1. Unsafe Carry-Save Right Shift

an interesting characteristic of carry-save representation, where the sign bits of carry and sum vectors fail to provide any indication of whether the resulting non-redundant number is positive or negative. Difficulty of sign detection is due to the intrinsic characteristic of sign and magnitude ambiguity within carry-save vectors. Thus, a right shift procedure based solely on speculation of the MSBs of carry and sum vectors fail to generate a correct answer all the time. To guarantee a correct outcome, extra bit and circuitry logic is needed during the shift operation to monitors MSBs and to determine the correct bit value that is to be shifted. The detailed solution is explained in the next section.

The right shift problem arises amid addition of two carry-save inputs that is followed by a 1-bit right shift operation. In this configuration, carry chains can travel up to 2-bit position. This problem would not arise for systems using only

one carry-save vector, which implies one full adder for performing addition and carry chains propagating up to 1-bit position.

2.1.2. Solution

2.1.2.1. Observation

The arithmetic right shift for a non-redundant binary representation preserves the original sign of a number by duplicating the sign bit while shifting to the right. This provides an important observation that first and the second MSBs of the shifted vector must be identical after the arithmetic right shift. Similarly, the correct right shift operation of the carry-save vectors must mimic a sign extended right shift technique of a non-redundant binary representation. In other words, adding or converting a correctly shifted carry-save vectors should result in a non-redundant binary vector where two MSBs are identical. Figure 2.2 illustrates how the end results for non-redundant binary and carry-save representation should be equivalent.

Following the steps of sign extended shift operation of a non-redundant representation, a proper arithmetic right shift of carry-save vectors must retain the corresponding sign represented by the original vectors. Preserving the original sign includes sign detection and sign duplication, but sign cannot simply be determined by speculation of MSBs. Rather, evaluating the represented sign of a carry-save vectors depend on three parameters, which are the MSB of a carry vector, the MSB of a sum vector, and the propagating carry into the MSB position. Figure 2.3 exemplifies the three required fields for sign detection. The main challenge resides in sign detection, specifically the dependency on propagating carry because obtaining this information insinuates the use of a carry ripple adder. No-

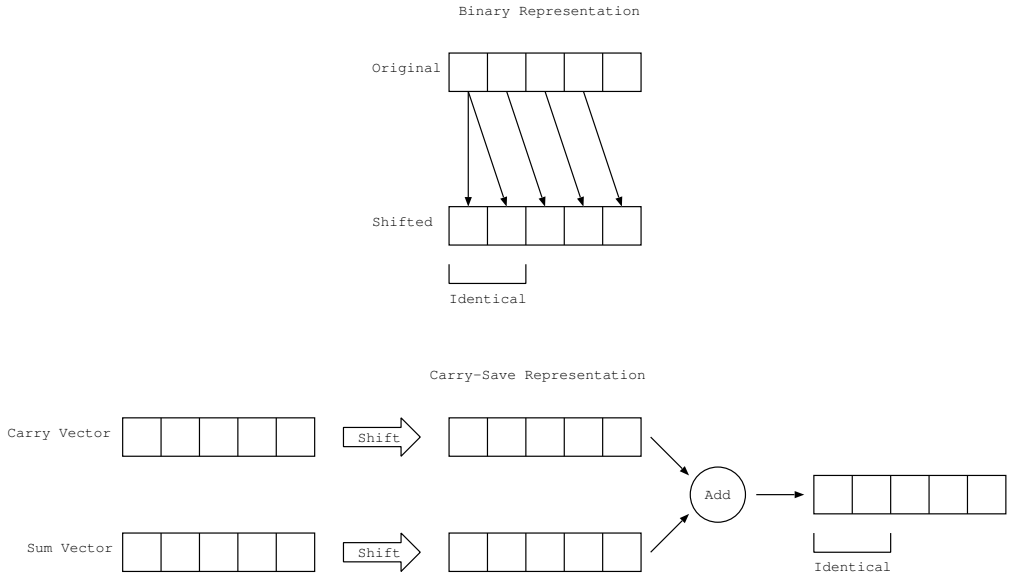

FIGURE 2.2. Mimic Binary Right Shift with Sign Extension

tice that the sign of all the cases in Figure 2.3 depend on the propagating carry.

In search for a more attractive solution that is independent with carry propagation, 2 MSBs are considered under analysis. A table is constructed to observe a pattern, where propagating carry into the second MSB position, instead of the first, was scrutinized. Table 2.2 shows all possible scenarios with before and after bit results of correct arithmetic right shift processes. Again, case (b) and (h) of Table 2.2 indicates that the solution depends on determining the carry that is generated into the second MSB position, which can only be determined by converting the carry-save vector back to the normal representation. Clearly, this method is not a feasible solution since it contradicts the purpose for using the redundant representation.

		With Carry		Without Carry	
		Original	Right Shifted	Original	Right Shifted
(a)	Sum Vector	0010	0001	0010	0001
	Carry Vector	0010	0001	0000	0000
	Binary Vector	0100	0010	0010	0001
(b)	Sum Vector	0010	0001	0010	0001
	Carry Vector	0110	1011	0100	0010
	Binary Vector	1000	1100	0110	0011
(c)	Sum Vector	0110	0011	0110	0011
	Carry Vector	0110	1011	0100	1010
	Binary Vector	1100	1110	1010	1101
(d)	Sum Vector	1010	0101	1010	0101
	Carry Vector	0010	1001	0000	1000
	Binary Vector	1100	1110	1010	1101
(e)	Sum Vector	1110	0111	1110	0111
	Carry Vector	0010	1001	0000	1000
	Binary Vector	0000	0000	1110	1111
(f)	Sum Vector	0110	0011	0110	0011
	Carry Vector	1110	1111	1100	1110
	Binary Vector	0100	0010	0010	0001
(g)	Sum Vector	1010	1101	1010	1101
	Carry Vector	1010	0101	1000	0100
	Binary Vector	0100	0010	0010	0001
(h)	Sum Vector	1110	1111	1110	0111
	Carry Vector	1010	1101	1000	1100
	Binary Vector	1000	1100	0110	0011
(i)	Sum Vector	1110	1111	1110	1111
	Carry Vector	1110	1111	1100	1110
	Binary Vector	1100	1110	1010	1101

TABLE 2.2. Carry-Save Right Shift with 2 MSBs Under Analysis

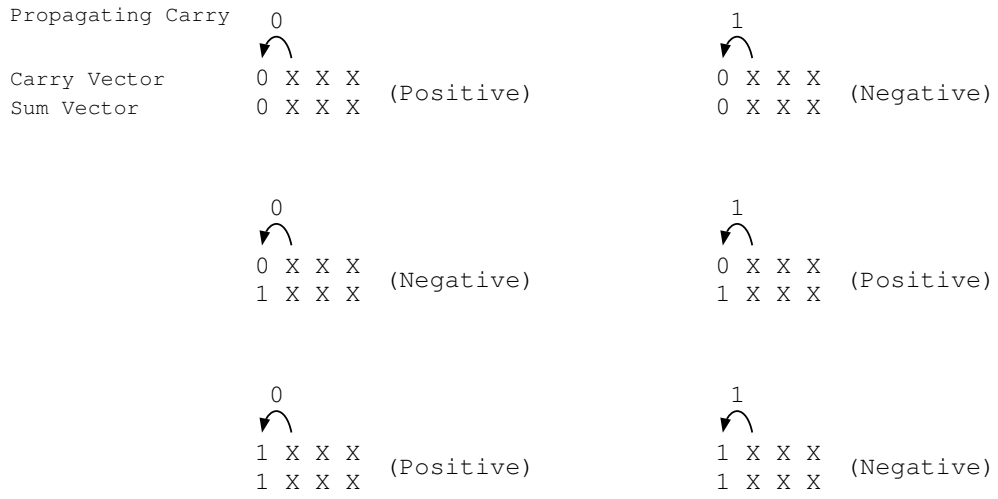

FIGURE 2.3. Dilemma of Sign Detection

2.1.2.2. Extra Sign Duplication of Inputs

Since carry chains can travel up to 2-bit positions during carry-save vector addition, extra sign bit is duplicated for the input operands to retain this information. As a result, first and second MSBs of input vectors represent a sign bit. Following the proposed duplication method results in a smaller output range for carry-save addition, which is expressed by the below compound inequality. Variable n represents the total number of bits, variable d refers to a non-redundant representation of carry-save vector output.

$$-2^{n-2} \leq d \leq 2^{n-2} - 1$$

The range is 2-bit less than total number of bits since 2 MSBs are duplicated sign bit. Table 2.2 can now be reconstructed with excluding cases which fall outside of the range equation. Table 2.3 illustrates the modified table with 'xxxx' indicating cases which violate the range condition. Moreover, Table 2.3 examines

the 2 MSBs of carry and sum vector and carry generated into second MSB with generalization of least significant bits (LSB) that follows, which is indicated by dashes. Now, according to Table 2.3, the outcomes of the shifted value have no dependence on the carry chain information.

2.1.2.3. Solution Logic

The arithmetic right shift of carry-save vectors is performed with the help of logic circuitry that monitors 2 MSBs of sum vector and MSB of carry vector. Table 2.4 is composed of only MSBs of interest, which shows all the possibilities of correct right shift values. Table 2.4 is used as a reference truth table to derive a satisfying solution logic. The Espresso, logic minimization software, was utilized to develop an optimal gate network. Due to several output possibilities for some inputs, educated trial and error approach was adopted and applied to Espresso.

Based on the observation of Table 2.4, the right shift of a sum vector is accomplished by following the conventional sign extension technique. As for a carry vector, logic gates are required to determine the MSB of a shifted vector. Boolean expression is given below where yc, ys, xc, and xs indicate shifted carry vector, shifted sum vector, original carry vector, and original sum vector, respectively with subscripts representing bit positions.

$$yc_{n-1} = xc_{n-1} \cdot xs_{n-2} \; + \; xc_{n-1} \cdot \overline{xs_{n-1}} \; + \; \overline{xs_{n-1}} \cdot xs_{n-2}$$

$$yc_{n-2} = xc_{n-1}$$

$$ys_{n-1} = xs_{n-1}$$

$$ys_{n-2} = xs_{n-1}$$

		With Carry		Without Carry	
		Original	Right Shifted	Original	Right Shifted
(a)	Sum Vector	00–	xxxx	00–	00–
	Carry Vector	00–	xxxx	00–	00–
	Binary Vector	01–	xxxx	00–	00–
(b)	Sum Vector	00–	xxxx	00–	xxxx
	Carry Vector	01–	xxxx	01–	xxxx
	Binary Vector	10–	xxxx	01–	xxxx
(c)	Sum Vector	01–	00–	01–	xxxx
	Carry Vector	01–	10–	01–	xxxx
	Binary Vector	11–	11–	10–	xxxx
(d)	Sum Vector	10–	01–	10–	xxxx
	Carry Vector	00–	10–	00–	xxxx
	Binary Vector	11–	11–	10–	xxxx
(e)	Sum Vector	11–	01–	11–	01–
	Carry Vector	00–	10–	00–	10–
	Binary Vector	00–	00–	11–	11–
(f)	Sum Vector	01–	xxxx	01–	00–
	Carry Vector	11–	xxxx	11–	11–
	Binary Vector	01–	xxxx	00–	00–
(g)	Sum Vector	10–	xxxx	10–	11–
	Carry Vector	10–	xxxx	10–	01–
	Binary Vector	01–	xxxx	00–	00–
(h)	Sum Vector	11–	xxxx	11–	xxxx
	Carry Vector	10–	xxxx	10–	xxxx
	Binary Vector	10–	xxxx	01–	xxxx
(i)	Sum Vector	11–	11–	11–	xxxx
	Carry Vector	11–	11–	11–	xxxx
	Binary Vector	11–	11–	10–	xxxx

TABLE 2.3. Generalized Carry-Save Right Shift with Extra Sign Duplication

2 MSBs ($c_{n-1}c_{n-2}s_{n-1}s_{n-2}$)	Possible Shifted 2 MSBs
0000	0000
0001	xxxx
0010	xxxx
0011	0100, 1000
0100	0101, 1001, 0110, 1010
0101	0101, 1001, 0110, 1010
0110	0101, 1001, 0110, 1010
0111	0101, 1001, 0110, 1010
1000	0101, 1001, 0110, 1010
1001	0101, 1001, 0110, 1010
1010	0101, 1001, 0110, 1010
1011	0101, 1001, 0110, 1010
1100	0111, 1011
1101	xxxx
1101	xxxx
1111	1111

TABLE 2.4. Truth Table for the Shift Logic

Corresponding gate level schematic of the right shift logic is shown in Figure 2.4. Besides maintaining the correct sign, the solution logic gives an insight to the relationship between propagating carry, sign, and shifted value. The propagating carry into the MSB position can be derived by a speculation of shifted results and the fact that first and second MSBs must be identical.

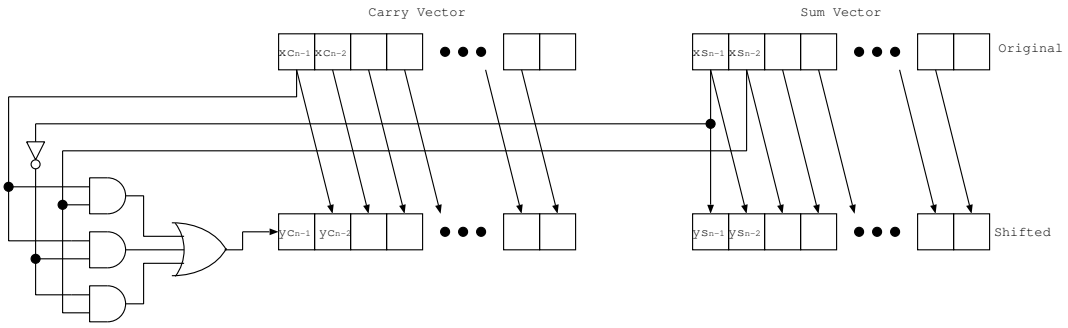

FIGURE 2.4. Gate Level Schematic of Right Shift Logic

2.2. Zero Detection

An examination of the pseudo code given in the Figure 3.1 shows that the modular division algorithm terminates when the value of c becomes zero. Hence, a zero detection unit is required to check for the zero condition and to exit from the iterative loop of the algorithm.

As mentioned previously, carry-save representation exhibits difficulty in performing magnitude tests due to its redundant property. A zero can be expressed in many different combinations of carry and sum vectors. The Table 2.5 provides a few of the possible combinations of carry-save vectors corresponding to a value zero.

	Possible Zero Representations in Carry-save		
Carry Vector	00010000	11000000	11001100
Sum Vector	11110000	01000000	00110100

TABLE 2.5. Possible Zero Vector Combinations

Fortunately, the modular division algorithm follows a unique method for reaching a zero condition, where two input operands must be odd, equal in magnitude, and opposite in sign. According to the definition of 2's complement, odd numbers equal in magnitude and opposite in sign results in pattern of complemented bits in every bit position, excluding the LSB. Following observation is depicted in Table 2.6. Notice, resulting bit patterns for exclusive OR of carry and sum vectors are equivalent for all odd cases. This narrows the search of zero condition to detecting leading ones terminated by a zero. As for even number sets, a uniform pattern does not exist.

	3-3	5-5	7-7	13-13
Carry Vector	00000011	00000101	00001111	00001101
Sum Vector	11111101	11111011	11110001	11110011
XOR	11111110	11111110	11111110	11111110

TABLE 2.6. Zero Vector of Odd Numbers

Recall, addition of two carry-save vectors consists of four input vectors, which requires two full adders. With two full adders, a carry chain could travel 1-bit or 2-bit positions to the left, as shown in Figure 2.5. This translate to variation of the zero vector outcome depending on the addition order of input

vectors. Figure 2.6 shows how addition order of the same input vectors affects the outcome result.

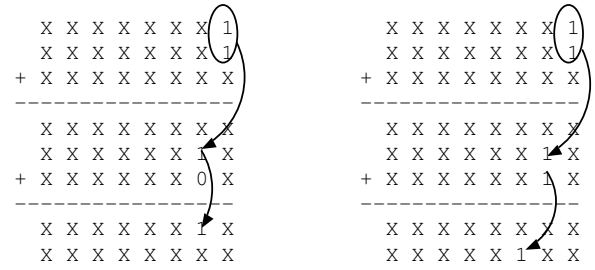

FIGURE 2.5. Carry Chain Possibility

```
Input_1   11110000 >  1
Input_2   00010001 >

Input_3   11110000 > -1
Input_4   00001111 >
```

```
(a)   11110000 (even)
       00010001 (odd)
      +11110000 (even)
      ---------
       00010001
       11100000
      +00001111 (odd)
      ---------
       11111110
       00000010
```

```
(b)   11110000 (even)
       00010001 (odd)
      +00001111 (odd)
      ---------
       11101110
       00100010
      +11110000 (even)
      ---------
       00111100
       11000100
```

```
(c)   11110000 (even)
       11110000 (even)
      +00001111 (odd)
      ---------
       00001111
       11100000
      +00010001 (odd)
      ---------
       11111110
       00000010
```

```
(d)   00010001 (odd)
       11110000 (even)
      +00001111 (odd)
      ---------
       11101110
       00100010
      +11110000 (even)
      ---------
       00111100
       11000100
```

FIGURE 2.6. Addition Order for Input 1 and −1

To accommodate for the variation, resulting output can be added to a value zero which maintains the original value but settles the vectors to a more favorable form. Figure 2.7 shows an example of settling ones. Adding to a value zero reorganizes string of ones into sum vector and left shifts the carry chain in carry vector. For example, (-2,2) becomes (-4,4), where both vectors represent a zero in carry-save form. Accounting for the worse case, testing for the universal value of (-8,8) would detect all zero conditions.

```
  1 0 1 0 1 1 1 1   sum_vector
  0 1 0 1 0 0 0 1   carry_vector
+ 0 0 0 0 0 0 0 0   add to zero
-----------------
  1 1 1 1 1 1 1 1   settled sum_vector
  0 0 0 0 0 0 1 0   settled carry_vector
```

FIGURE 2.7. Settling Ones

2.3. Subtraction

Subtraction in carry-save domain requires performing 2's complement on both carry vector and sum vector. Therefore, carry and sum vectors need to be inverted and each inverted results should be incremented by one. For operations without right shift, adding value of one to each vector can be done with use of LSB of resulting carry vector after addition. This unused free space is shown in Figure 2.8. If right shift is executed, then carry out bit of the bottom adder cannot be utilized since it would be shifted out. Luckily, the algorithm allows right shift operation for odd values only, hence the increment step is integrated with inversion process, where the LSB of the sum vector is forced to a value of

one.

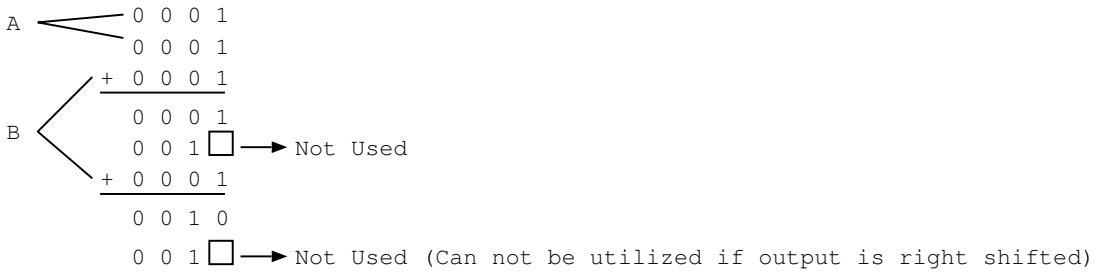

FIGURE 2.8. Unused Space During Addition of Carry-save Vectors $A + B$

3. HARDWARE DESIGN FOR THE SCALABLE ARCHITECTURE

3.1. Unified Modular Division/Multiplication Algorithm

A novel algorithm for unified modular division/multiplication was devised in [8], where the algorithm was written and tested in Maple by [14]. Unified implies the algorithm to be functional in both prime $GF(p)$ and binary extension $GF(2^n)$ fields, also referred to as being dual field. The algorithm extended from the unified division algorithm [7] and ingenuously combines the Montgomery multiplication to the modular division unit by maximizing the reuse of the same datapath. Hence, with small increase in control complexity, modular multiplication was incorporated into the modular division algorithm.

The pseudo code written by [8] is shown in Figure 3.1, which is capable of computing both Montgomery modular multiplication and modular division. Notice that modular inverse calculation is possible by forcing a numerator to a value of one during a modular division operation. This algorithm was translated and mapped into hardware while serving as a corner stone during the design process.

3.2. Scalable Architecture

The idea of scalable architecture was introduced in [5, 6, 15]. The objective for a scalable architecture is to design a flexible hardware that is able to handle various precision sizes by repetitive use of hardware that is of fixed area. Therefore, the scalable approach eliminates the fixed precision constraint imposed on target specific designs. In addition, scalable architecture counteracts the increased area requirement that results from using a carry-save structure. The limit on the

Inputs: $0 \leq X < p$, $0 < Y < p$, $2^{n-1} < p < 2^n$, $Field$, Op, n

Output: $Z = XY2^{-n} \ mod \ p$ when $Op = mult$, $Z = \frac{X}{Y} \ mod \ p$ when $Op = div$

Algorithm:

```
IF Op = mult THEN                          /* Multiplication Mode */
    D = 0, U = 0, W = X, C = Y, δ = n
ELSE                                       /* Division Mode */
    D = p, U = X, W = 0, C = Y, δ = 0
END IF;
WHILE [(C ≠ 0 AND Op = div) OR ( δ ≠ 0 AND Op = mult)]
    IF c0 = 0 THEN
        C := C >> 1
        δ := δ − 1                         /* Integer Operation */
    ELSE
        k = 1
        IF (Op = div) THEN
            IF δ < 0 THEN
                C ⇔ D, U ⇔ W, δ := −δ
            END IF;
            IF((C + D) mod 4 ≠ 0 AND Field = GF(p)) THEN
                k = −1
            ELSE
                δ := δ − 1
            END IF;
        ELSE
            δ := δ − 1
        END IF;
        C := (C + k ∗ D) >> 1,   U := (U + k ∗ W)
    END IF;
    U := (U + u0 ∗ p) >> 1
END WHILE;
IF Op = div THEN
    Z := W
ELSE
    Z := U
END IF;
```

FIGURE 3.1. Unified Modular Division/Multiplication Algorithm

maximum size of the precision allowed for the scalable design is governed by the available memory that stores the inputs and internal results. Given a sufficient amount of memory space, a scalable unit can perform computation on 32-bit or 128-bit without major redesign of hardware.

The scalable architecture represents a generic hardware model which enhances portability across different hardware platforms. For instance, Field Programmable Gate Array (FPGA) with limited amount of area can chose a smaller datapath version of the scalable design to program the whole system into the FPGA. If performance is the primary concern, then larger datapath version can be implemented for speed.

3.2.1. Scalable Issues and Overhead

Complying with the above scalable definition implies the inevitable addition of extra hardware compared to a non-scalable design. Following sections explain encountered issues relating to the implementation of the scalable architecture.

3.2.1.1. Information Exchange

The scalable architecture divides total precision into group of smaller size, referred in this text as words. For example, 1024-bit value can be divided into two 512-bit words. Consequently, working with slices of total precision requires information exchange between consecutive words to follow the correct operation of a complete design. Figure 3.2 illustrates the information exchange needed to achieve a correct scalable operation.

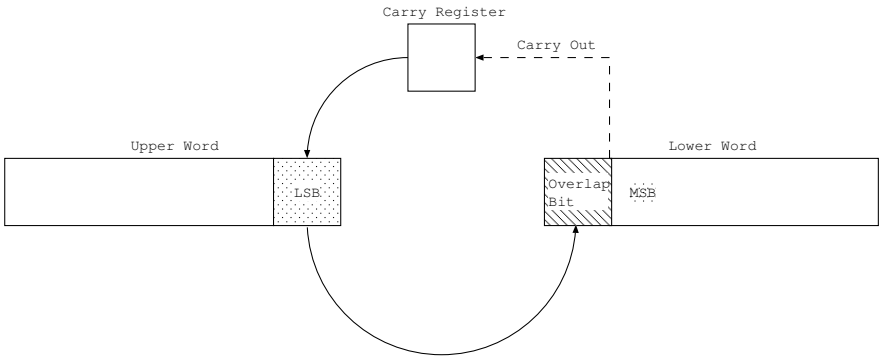

FIGURE 3.2. Information Exchange between Words

First, a carry out bit generated from each word during carry-save addition must be stored and forwarded to the next word. This carry out bit represents an internal propagating carry when viewed from the total precision perspective and cannot be discarded. Transfer of carry bit between words assures that all the carry chains within the original value are preserved and properly processed. As a result, buffer responsible for storing carry out of each word is required inside the data path module which can be implemented as a simple clock edge triggered register.

Secondly, a bit value in position that is 1-bit higher than the MSB of the current word needs to be extracted for a correct right shift operation. This bit can also be viewed as a LSB of the next consecutive word. During arithmetic right shift of individual words, the bit value to be shifted into its MSB position cannot be determined without the information from the connecting higher word. Considering the MSB of a word as a sign bit and shifting accordingly leads to an incorrect result. It is important to realize that the MSB of a word within the scalable design does not necessarily represent the MSB of the total precision.

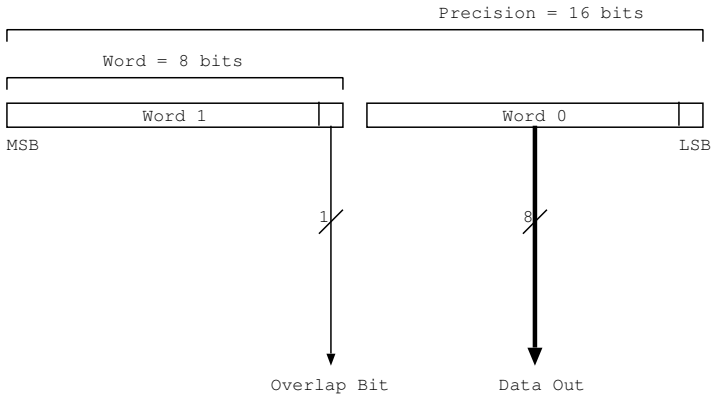

FIGURE 3.3. Overlap Bit

From the reference point of total precision, MSB of a word could map to any bit within the total precision.

This extra bit is referred to as an overlapping bit because the value is read twice in an overlapped fashion. Acquiring overlapped bit occurs inside the dual port register file. The bit tapping register file has an additional output that extracts a LSB from the next consecutive word as illustrated in Figure 3.3. Consequently, the size of the scalable datapath is expanded by 1-bit to include this overlap bit. At the same time, overlap bit satisfies the extra bit requirement of the proposed carry-save arithmetic right shift solution.

3.2.1.2. Algorithm Flow

The flow of the algorithm is determined by the LSBs of total precision, which test for even and divisible by four condition. Following this observation, logical word flow of the scalable architecture should begin with a lowest word and then move on to a higher word consecutively, as shown in Figure 3.4. The true LSBs of the total precision are saved into a register such that the higher words

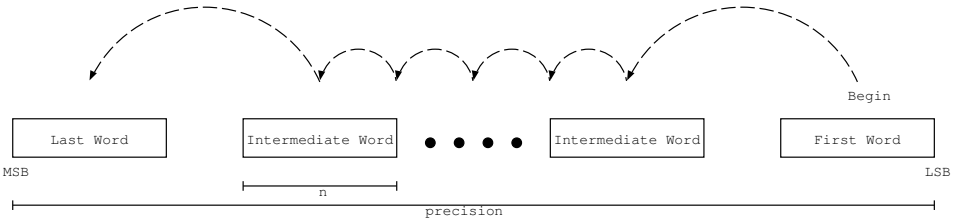

FIGURE 3.4. Word Execution Sequence

can follow the same path of the algorithm. Remember that the LSBs of higher words are meaningless in determining the algorithm's flow since these are not the true LSBs of the total precision.

3.2.1.3. Word Counter

Words inside the scalable design can be categorized into three different classes that are divided into first word, last word, and intermediate words. Different operations are performed for each group as organized below.

- First word
  - Save LSBs
  - Increment required for subtraction
  - Store carry out
  - Extract overlap bit
- Last word
  - Utilize right shift logic
  - Accept carry in
- Intermediate words

- Store carry out
- Extract overlap bit
- Accept carry in

In order to determine the class status of the current word under execution, a word counter is created given *precision* and n parameters. Upon determining the class of a particular word, proper tasks are performed accordingly.

3.2.1.4. Partial Zero Detection

Since the scalable design operates on a reduced portion of the total precision, checking for zero condition is tested in a partial manner to a divided portion. The scalable version of the zero detection unit is composed of a pass or fail register in addition to zero test logic. The zero test logic tests if the given word under execution matches the zero condition of a total precision. The pass or fail register retains the zero test information throughout different word executions.

3.3. Hardware Description

The design of scalable and unified modular division and Montgomery multiplication was coded in VHDL for hardware implementation. Every VHDL code was written with generic parameters defined at the top of the code, such that the size of the architecture can be set to any desired value.

The description of individual components within the complete hardware system is analyzed in this section. The top level structure of the hardware system is composed of register file, datapath, and control. Simplified top level diagram is illustrated in Figure 3.5 to provide a general overview of the design.

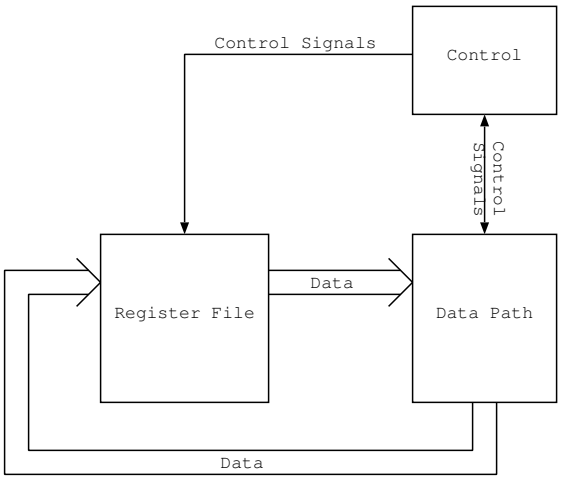

FIGURE 3.5. Top Level Diagram

3.3.1. Register File

The register file serves as a storage space for initial, intermediate, and final data results. The register file consists of one input and two output ports, which can be viewed as a synchronous dual port register file. Dual output ports supply two input values to the datapath for computation and the computed result is written back to the register file. The read and write signals are synchronous with a clock signal.

With the aim to reduce unnecessary register space, the address bit length and the total number of addresses can both be specified. For example, an address bit length of three can allow maximum of $2^3 = 8$ addresses, but if only five addresses are needed then it is possible to create five addresses by initial user setting.

As a requirement of supporting the scalable architecture, register file is capable of reading extra overlap bit in addition to reading a requested data. For

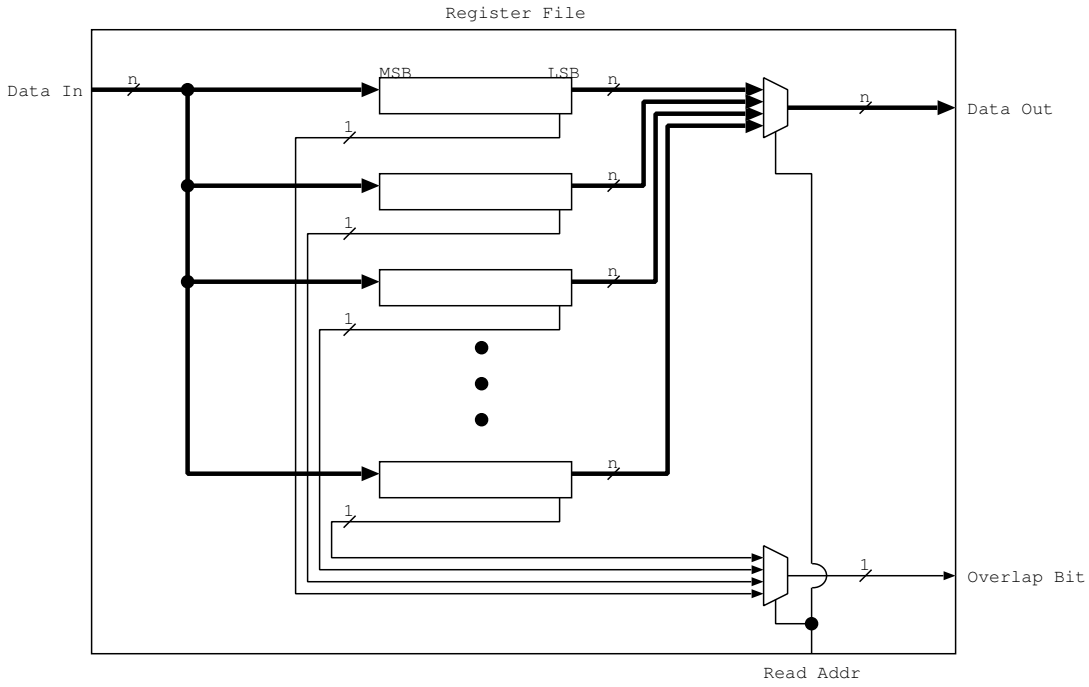

FIGURE 3.6. Register File with Overlap Bit Output

simplicity, architecture for single vector is provided in Figure 3.6 to emphasize the concept of the overlap bit.

3.3.2. Datapath

The datapath is responsible for data transfers and manipulations. Multiplexer direct dataflow and slightly modified full adders manipulate input values. An adder that is capable of computing addition in both finite fields, $GF(p)$ and $GF(2^n)$, is referred to as a dual field adder and satisfies the unified condition. The dual field adder has an input option to propagate or discard a carry bit. Propagating carry would satisfy the addition property in $GF(p)$ and discarding the carry bit would satisfy the addition property in $GF(2^n)$. Extra logic gate inside a full adder in conjunction with a small increase in control unit permits the design

to operate in both type of fields. The critical path delay of the dual field adder is equivalent to a normal full adder [15]. Figure 3.7 displays the layout of the datapath, with scalable overhead included.

3.3.3. Control

The control block generates control signals that are provided to the data path and to the register file. The major component in the control unit is the finite state machine and it follows a hardwired control methodology. Finite state machine within the control unit maintains present and next states to determine necessary actions to be executed. With the intention to design a robust and reliable control unit, the Moore state machine was chosen such that the output signals are only a function of the present states. Since the output signals come directly from a flip flop, the resulting output signals are glitch-free.

The algorithm's swap function is accomplished within control unit to avoid physical swapping of data in hardware. Actual transfer of data for swap function would be costly in terms of time, especially for a system with large precision. The delta counter controls the swapping operation. The delta counter was integrated into the control unit because it controls the flow of the algorithm. The functionality for delta counter include decrementing and negating a count value. With the goal of developing a fast design, a ring counter was chosen and implemented for the delta counter. Theoretically, delay of the ring counters are independent of its size [16], which aligns with the logic behind using carry-save adders.

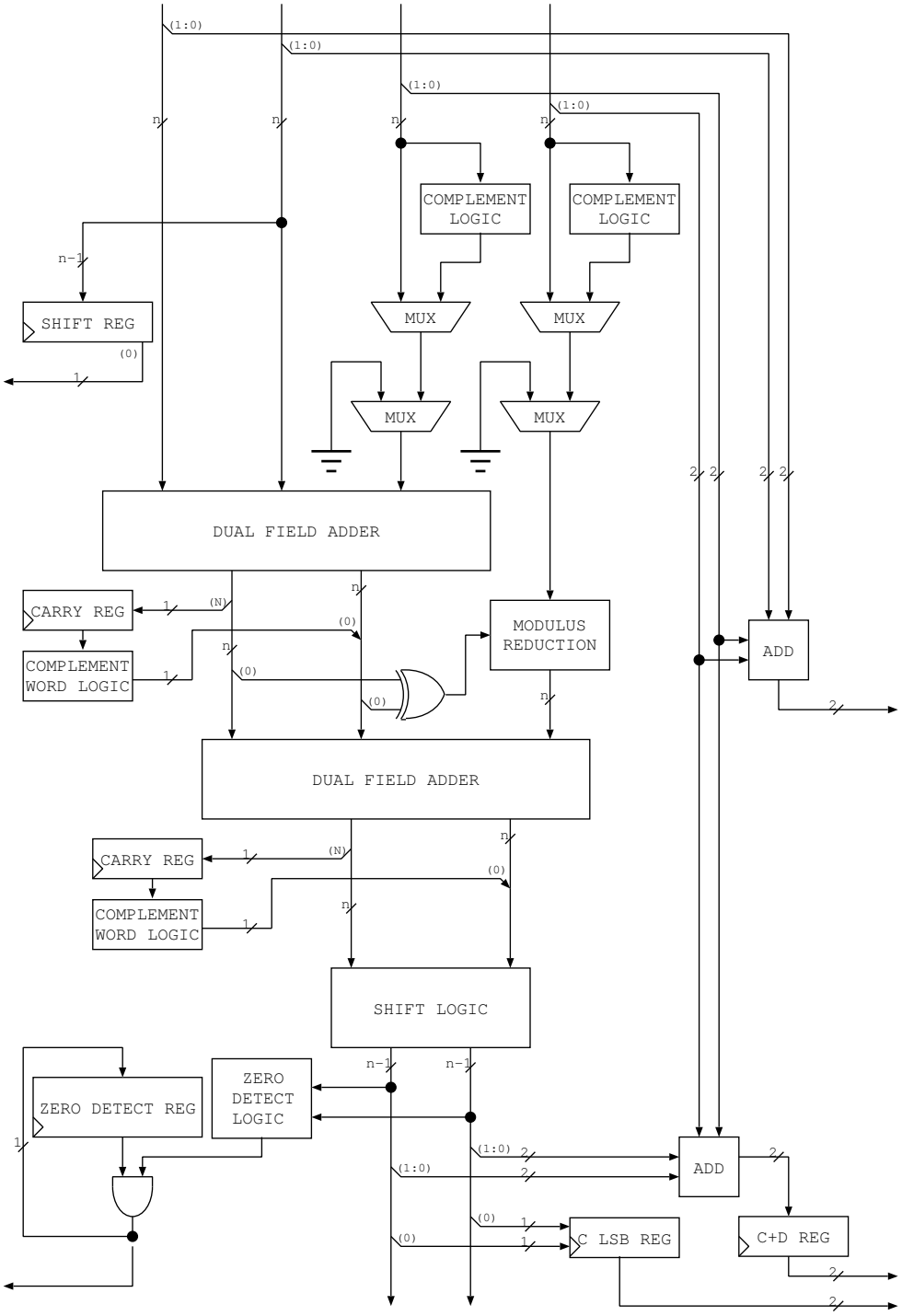

FIGURE 3.7. The Scalable Datapath

4. SIMULATION AND SYNTHESIS

After completion of the hardware design in VHDL, simulation and synthesis was performed for validation and analysis. Simulation step include many cycles of debugging and redesign of the hardware system. Synthesis performs a translation of VHDL code into logic gates. Since all the VHDL codes for the scalable design are written in a generic format, different sizes of the hardware can be simulated and synthesized by entering a desired value into the top level design.

4.1. Simulation

Simulation tests the functionality of the design under ideal characteristics. ModelSim, Mentor Graphics hardware description language simulator, provided the tool environment for testing and verification of the hardware design.

4.1.1. Extra Bits Constraint

For proper execution of the design, extra bits constraint must be satisfied. After deciding on a modulus (m), three extra bits must be added to the size of the datapath. One extra bit corresponds to the sign bit, other two bits bounds the output within $(m-1)+(m-1)+m$, to avoid an overflow. For example, $m = (1101)_2$ requires total datapath size of 7-bits.

4.1.2. Automation

The process of checking and simulating various values to the hardware system is repetitive, error prone, and time consuming. To improve speed and

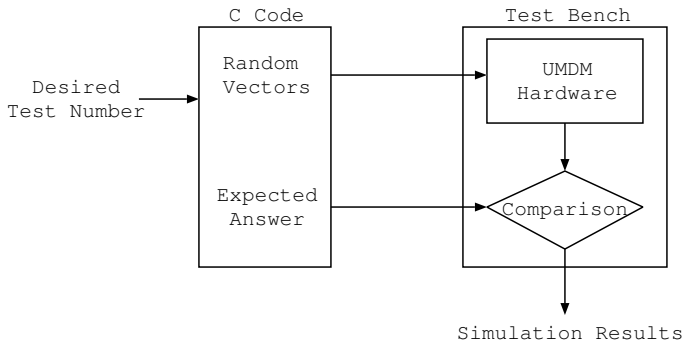

FIGURE 4.1. Overview of Simulation Model

accuracy, vector generation code was written in C language with a goal of automating the simulation process. Vector generation code creates an output file containing random test vectors of specified amount, which can be used as an input for simulation. Moreover, the unified modular division and Montgomery multiplication was translated into C language to obtain a different source of answer for comparison during validation process.

The test bench module was created and combined with the hardware system to complete the automated simulation system. The test bench unit compares the hardware computed value with the C code generated answer. If any discrepancies exist, an error statement is displayed to the main simulator window. In an attempt to verify progress of simulation, the test bench unit records the total number of hardware generated answers and checks if this value matches with the total number of randomly generated vectors. The structure of the simulation model is shown in Figure 4.1.

4.2. Synthesis

After simulation and debugging phase, the hardware design was synthesized into logic gates to obtain area and delay information. Three parameters of interest during synthesis procedure are *precision*, n, and *bits*. Below summarizes the definitions of these parameters.

- *precision* - targeted precision or total intended size
- n - actual datapath size or word length
- *bits* - number of bit for address vector of the register file

The variable *precision* defines the total intended size. The *precision* may represent the actual datapath size if scalable feature is not utilized, otherwise this value is used for control operations. The variable n relates to the scalable feature with range 3 to *precision* and must satisfy $precision \ mod \ n = 0$. The minimum value of three for n is constrained by the zero detection unit. The *bits* corresponds to the size of the register file's address bus such that when $bits = 3$ then maximum of 2^3 or 8 locations of register addresses exist. For unified modular division/multiplication, only five internal variables are required, therefore five registers are allocated instead of creating eight locations. Mathematical formula for calculating *bits* is given below.

$$bits = \left\lceil log_2\left(5 * \frac{precision}{n} - 1\right)\right\rceil$$

4.2.1. PERL Script

Following simulation, synthesis step was automated to ease the inherently redundant and time consuming procedure of synthesizing various sizes of the scal-

able architecture. Practical Extraction Report Language (PERL) was the suitable language for the task of automating synthesis process. The program file written in PERL is able to modify VHDL source codes, generate multiple synthesis script files, and create a master executable file. Modification of VHDL source code was necessary to delete check and verification hardware that is only used for simulation purposes. Multiple synthesis script files are generated with each file containing different sizes of precision and datapath to be synthesized. Master executable file is responsible for starting and exiting between each execution of synthesis script file to avoid log files from overloading the hard drive or causing a bus error.

4.2.2. Leonardo Synthesis Results

Synthesis of the design was performed using Leonardo with target technology of ASIC ADK AMI 0.5 micron fast. Precision starting from 16-bit up to 512-bit with various datapath sizes were chosen and synthesized for space and time analysis. Synthesis results of area and delay are expressed as a tabular form in Table 4.1. Corresponding area and delay graphs are provided in Figure 4.2 and in Figure 4.3, respectively.

Additionally, synthesis was performed with the register file separated from the whole design. The scalable architecture does not apply to the register file because it represents a memory element for inputs, which cannot be reduced. Therefore, treating the register file separately provides a more insightful analysis. Area and delay results for separate register file are organized in Table 4.2. Isolating the register file indicates that there is higher degree of area inconsistency within the register file than in the combination of datapath and control. Graphical

Input Parameters		Output Results	
precision	n	Area	Delay
16	16	4498 gates	20.91 ns
32	16	6513 gates	20.34 ns
32	32	7667 gates	21.53 ns
64	16	11144 gates	21.64 ns
64	32	11744 gates	21.00 ns
64	64	13998 gates	22.84 ns
128	16	20015 gates	22.78 ns
128	32	20281 gates	22.29 ns
128	64	22114 gates	22.06 ns
128	128	27315 gates	23.89 ns
256	16	38442 gates	24.40 ns
256	32	38158 gates	25.03 ns
256	64	39141 gates	23.79 ns
256	128	42723 gates	23.06 ns
256	256	53358 gates	24.50 ns
512	16	78654 gates	28.70 ns
512	32	72405 gates	24.89 ns
512	64	74254 gates	25.04 ns
512	128	76524 gates	24.67 ns
512	256	84008 gates	24.53 ns
512	512	105304 gates	26.40 ns

TABLE 4.1. Synthesis Results of Register File, Datapath, and Control

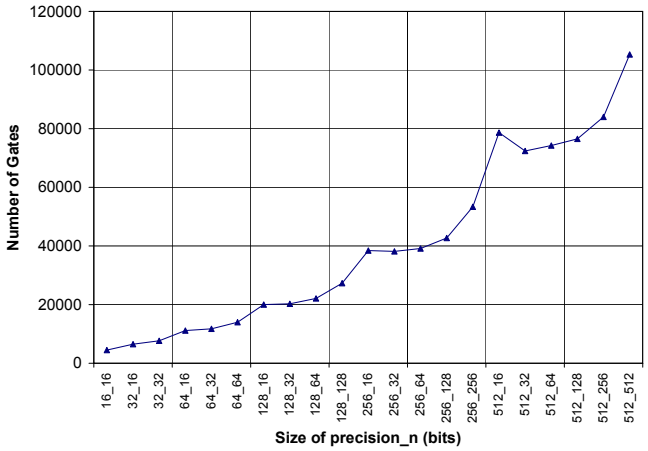

FIGURE 4.2. Area Results for Various *precision* and n

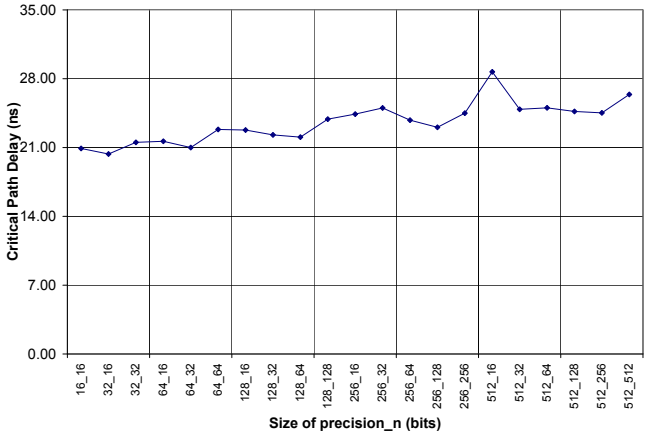

FIGURE 4.3. Delay Results for Various *precision* and n

representation of area comparison is given in Figure 4.4 and delay comparison is given in Figure 4.5.

4.3. Result Analysis

As expected, critical path delays for various precisions remain relatively constant, which fluctuates within the range of 20.34ns to 28.70ns. From Table 4.1 and Figure 4.2, area results appear to deviate slightly from the expectation of increase in area as a function of a datapath size for the case $precision = 512$ and $n = 16$. Figure 4.4 indicates that the register file unit is responsible for the irregular area results. Inconsistent areas for register file could be due to the inherently regular and repetitive structure of the register file. Internal architecture of the register file consist of multiple and identical memory elements. Thus, a synthesis optimization algorithm could favor and be biased toward certain $precision$ and n combination during a translation into logic gates. Overall, the area of the register file remains constant for given precision. As for the delay, Figure 4.5 shows that the register file suffers from a higher delay for scalable architecture, due to higher number of register file addresses.

Input Parameters		Output Results			
		Register File		Datapath & Control	
precision	n	Area	Delay	Area	Delay
16	16	1989 gates	3.73 ns	2416 gates	14.35 ns
32	16	3740 gates	4.68 ns	2618 gates	14.08 ns
32	32	3884 gates	4.08 ns	3612 gates	14.04 ns
64	16	7592 gates	4.94 ns	3271 gates	15.01 ns
64	32	7471 gates	4.81 ns	3987 gates	13.76 ns
64	64	7694 gates	4.54 ns	5980 gates	14.78 ns
128	16	15038 gates	5.17 ns	4332 gates	14.79 ns
128	32	14519 gates	7.44 ns	5225 gates	14.75 ns
128	64	14805 gates	4.88 ns	6771 gates	14.52 ns
128	128	15818 gates	3.66 ns	10723 gates	15.82 ns
256	16	30821 gates	6.95 ns	6380 gates	15.01 ns
256	32	29648 gates	6.84 ns	7302 gates	14.56 ns
256	64	29032 gates	6.41 ns	9117 gates	15.71 ns
256	128	29400 gates	5.50 ns	12283 gates	15.45 ns
256	256	31570 gates	4.38 ns	20258 gates	16.58 ns
512	16	64080 gates	8.85 ns	10495 gates	14.02 ns
512	32	58700 gates	6.77 ns	11384 gates	14.70 ns
512	64	58685 gates	6.56 ns	13227 gates	15.53 ns
512	128	57677 gates	6.66 ns	16912 gates	16.81 ns
512	256	58613 gates	6.12 ns	23352 gates	16.31 ns
512	512	63064 gates	4.38 ns	39195 gates	17.45 ns

TABLE 4.2. Synthesis Results for Separating Register File

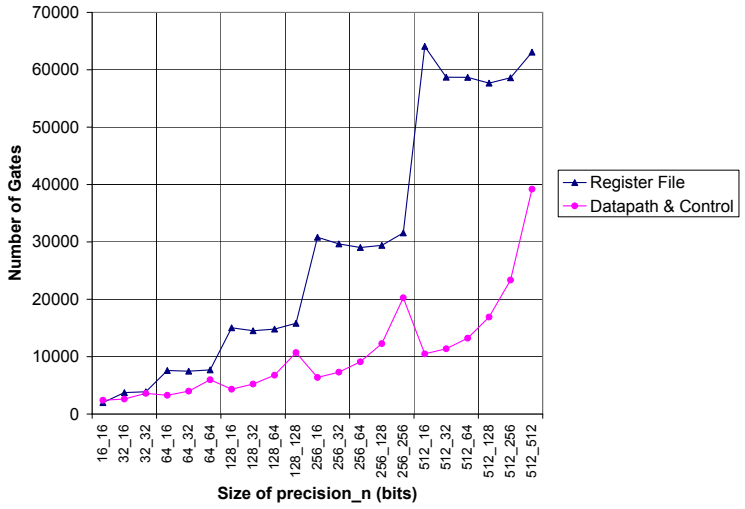

FIGURE 4.4. Area Comparison of Register File vs. Datapath & Control

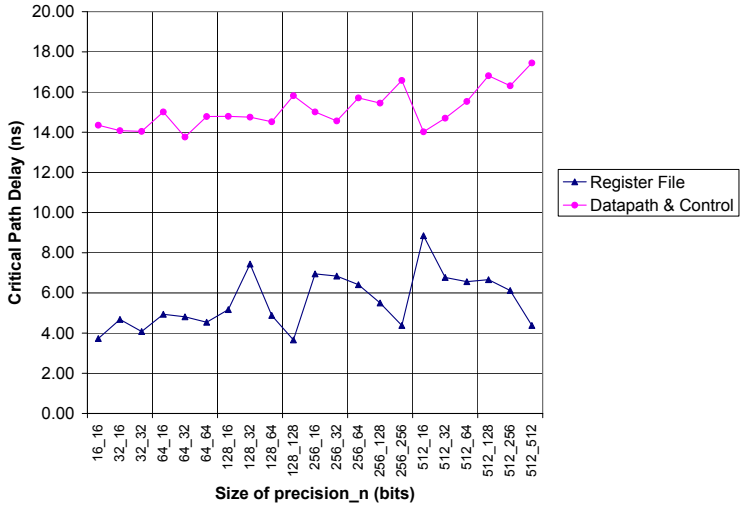

FIGURE 4.5. Delay Comparison of Register File vs. Datapath & Control

5. CONCLUSION

A scalable hardware for the unified modular division and Montgomery multiplication algorithm was designed and implemented with the help of VHDL, a hardware description language. The scalable feature refers to the size flexibility where high precision computations can be achieved by reuse of smaller hardware. It offers a trade-off between area and delay. Unified indicates the hardware's ability to perform modular division and multiplication in Galois fields, $GF(p)$ and $GF(2^n)$. The design is targeted for applications in the field of cryptography.

Carry-save representation was chosen as the hardware's internal number system to develop a fast and precision independent processing core. Due to the redundant nature of carry-save vectors, executing arithmetic right shift and detecting zero condition imposed a challenge. The source of these issues is rooted in the difficulty of magnitude and sign detection for representations in carry-save domain.

Functionality of the hardware was tested with random input vectors for modular division and Montgomery multiplication in both Galois fields. Following the hardware validation, the design was translated into a gate network for time and space analysis. The final product of designed hardware offers area flexibility, dual field operation, and confined critical path delay.

5.1. Future Work

Further research and enhancement can be applied to the delta counter residing inside the control unit. Although ring counters offer precision independent delay time, exponential growth of area consumption with increase in count range is a clear disadvantage compared to binary counters.

Hybrid design with a combination of carry-save and carry-propagating adders would be an attractive architecture. Instead of n-bit carry and sum vectors, carry vector can be reduced to a size smaller than n by allowing partial propagation of carry bits. As a result, amount of area requirement can be saved. Further analysis of partially redundant number system is presented in [17].

The design of unified modular division/multiplication unit can be extended to implement elliptic curve cryptography [14, 2]. Core computations of elliptic curve cryptographic system consist of modular addition, multiplication, and inversion. The scalable UMDM is able to execute all of these modular arithmetic functions. The UMDM would be a competitive hardware design with scalable and precision independent delay characteristics.

BIBLIOGRAPHY

[1] W. Trappe and L. Washington, *Introduction to Cryptography with Coding Theory*, Prentice-Hall, Upper Saddle River, New Jersey, 2002.

[2] A. A. Gutub, *New Hardware Algorithms and Designs for Montgomery Modular Inverse Computation in Galois Fields $GF(p)$ and $GF(2^n)$*, Ph.D. thesis, Oregon State University, June 11 2002.

[3] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, California, 3rd edition, 2003.

[4] I. Koren, *Computer Arithmetic Algorithms*, A K Peters, Ltd, Natick, Massachusetts, 2nd edition, 2004.

[5] A. F. Tenca and Ç. K. Koç, "A scalable architecture for Montgomery multiplication," in *Cryptographic Hardware and Embedded Systems - CHES 1999*, Ç. K. Koç and C. Paar, Eds. August 12–13 1999, vol. 1717 of *First International Workshop, Worcester, MA, USA*, pp. 94–108, Springer-Verlag.

[6] A. F. Tenca and Ç. K. Koç, "A scalable architecture for modular multiplication based on Montgomery's algorithm," *IEEE Transactions on Computers*, vol. 52, pp. 1215–1221, September 2003.

[7] A. F. Tenca and L. A. Tawalbeh, "Algorithm for unified modular division in $GF(p)$ and $GF(2^n)$ suitable for cryptographic hardware," *IEE Electronics Letters*, vol. 40, no. 5, pp. 304–306, March 2004.

[8] A. F. Tenca and L. A. Tawalbeh, "An algorithm and hardware architecture for integrated modular division and multiplication in $GF(p)$ and $GF(2^n)$," in *Application-Specific Systems, Architectures and Processors - ASAP 2004*, September 27–29, 2004, IEEE 15th International Conference on Application-specific Systmes, Architectures and Processors, pp. 247–257.

[9] N. Takagi, "A vlsi algorithm for modular division based on the binary gcd algorithm," *IEICE Trans. Fundamentals*, vol. E81-A, no. 5, pp. 724–728, May 1998.

[10] D. E. Knuth, *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*, Addison-Wesley, Reading, Massachusetts, second edition, 1981.

[11] J. Stein, "Computational problems associated with racah algebra," *Journal of Computational Physics*, vol. 1, pp. 397–405, 1967.

[12] R. P. Brent and H. T. Kung, "Systolic VLSI arrays for linear-time GCD computation," in *VLSI 83*, F. Anceau and E. J. Aas, Eds. 1983, North-Holland, Amsterdam, pp. 145–154, Elsvier Science Publishers.

[13] M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann, San Francisco, California, 2004.

[14] L. A. Tawalbeh, *A Novel Unified Algorithm and Hardware Architecture for Integrated Modular Division and Multiplication in $GF(p)$ and $GF(2^n)$ Suitable for Public-Key Cryptography*, Ph.D. thesis, Oregon State University, October 28 2004.

[15] E. Savas, A. F. Tenca, and Ç. K. Koç, "A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^n)$," in *Cryptographic Hardware and Embedded Systems - CHES 2000*, Ç. K. Koç and C. Paar, Eds. August 17–18 2000, vol. 1965 of *Second International Workshop, Worcester, MA, USA*, pp. 277–292, Springer-Verlag.

[16] M. R. Stan, A. F. Tenca, and M. D. Ercegovac, "Long and fast up/down counters," *IEEE Transactions on Computers*, vol. 47, no. 7, pp. 722–735, July 1998.

[17] H. A. Fahmy and M. J. Flynn, "The case for a redundant format in floating point arithmetic," in *16th IEEE Symposium on Computer Arithmetic*. June 15–18 2003, Santiago de Compostela, Spain, pp. 95–102, IEEE Computer Society.

APPENDICES

APPENDIX A. VHDL Source Codes

A.1. Register File

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY dual_port_mem IS
GENERIC(
       PRECISION : positive := 8;  --entire precision length
       N         : positive := 8;  --datapath bit length
       BITS      : positive := 6   --number of bits for counter, 3 bits implies count in range 0-2^3 or 0-7
       );
PORT(
    clk              : IN std_logic;
    reset            : IN std_logic;
    write            : IN std_logic;
    r_addr_1         : IN std_logic_vector(BITS-1 DOWNTO 0);
    r_addr_2         : IN std_logic_vector(BITS-1 DOWNTO 0);
    w_addr           : IN std_logic_vector(BITS-1 DOWNTO 0);
    data_in          : IN std_logic_vector(2*N-1 DOWNTO 0);
    data_out_1       : BUFFER std_logic_vector(2*N-1 DOWNTO 0);
    data_out_2       : BUFFER std_logic_vector(2*N-1 DOWNTO 0);
    ghost_buffer_1_s : OUT std_logic;
    ghost_buffer_1_c : OUT std_logic;
    ghost_buffer_2_s : OUT std_logic;
    ghost_buffer_2_c : OUT std_logic
    );
END dual_port_mem;
ARCHITECTURE beh OF dual_port_mem IS
CONSTANT SYNTHESIZE : positive := 1;
CONSTANT MAX_REG : positive := (5*PRECISION/N);
-- data registers
TYPE data_array IS ARRAY (0 to MAX_REG-1) OF std_logic_vector(2*N-1 DOWNTO 0);
SIGNAL data : data_array;
BEGIN
internal_mem:
PROCESS (clk, reset, write, data_in, data, r_addr_1, r_addr_2, w_addr)
BEGIN
    IF (reset='1') THEN
        data_out_1 <= (others=>'0');
        data_out_2 <= (others=>'0');
        ghost_buffer_1_s <= '0';
        ghost_buffer_1_c <= '0';
        ghost_buffer_2_s <= '0';
        ghost_buffer_2_c <= '0';
        FOR i IN 0 TO (MAX_REG-1) LOOP
            data(i) <= (others=>'0');
        END LOOP;
    --asynchronous read
    ELSE
```

```
        data_out_1 <= data(conv_integer(r_addr_1));
        data_out_2 <= data(conv_integer(r_addr_2));
        --to avoid indexing too high otherwise 2*N-bits extra memory space is needed
        IF (conv_integer(r_addr_2+1)=MAX_REG OR conv_integer(r_addr_1+1)=MAX_REG) THEN
            ghost_buffer_1_s <= '0';
            ghost_buffer_1_c <= '0';
            ghost_buffer_2_s <= '0';
            ghost_buffer_2_c <= '0';
        ELSE
            ghost_buffer_1_s <= data(conv_integer(r_addr_1+1))(0);  --note data(MAX_REG) is not defined
            ghost_buffer_1_c <= data(conv_integer(r_addr_1+1))(2*N/2);
            ghost_buffer_2_s <= data(conv_integer(r_addr_2+1))(0);
            ghost_buffer_2_c <= data(conv_integer(r_addr_2+1))(2*N/2);
        END IF;
        --synchronous write
        IF (clk'event AND clk='1') THEN
            IF(write='1') THEN
                data(conv_integer(w_addr)) <= data_in;
            END IF;
        END IF;
    END IF;
END PROCESS internal_mem;
END beh;
```

A.2. Datapath

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_vector_arith.ALL;
USE work.ALL;
ENTITY datapath IS
GENERIC(
       N : positive:=4
       );
PORT(
    reset             : IN  std_logic;
    clk               : IN  std_logic;
    f_sel             : IN  std_logic;  --f_sel=0 -> GF(2^n), f_sel=1 -> GF(p)
    op                : IN  std_logic;  --op=0 -> mult, op=1 -> div
    load_shift_reg    : IN  std_logic;
    shift_c           : IN  std_logic;
    complement_sel    : IN  std_logic;
    odd_complement    : IN  std_logic;
    B_zero_sel        : IN  std_logic;
    modulo_reduce_sel : IN  std_logic;
    shifted_out_sel   : IN  std_logic;
    shift_logic       : IN  std_logic;
    first_word        : IN  std_logic;
    final_word        : IN  std_logic;
    reset_zero_reg    : IN  std_logic;
    overlap_sum_A     : IN  std_logic;
```

```
    overlap_carry_A   : IN  std_logic;
    overlap_sum_B     : IN  std_logic;
    overlap_carry_B   : IN  std_logic;
    sum_vector_A      : IN  std_logic_vector(N-1 DOWNTO 0);
    carry_vector_A    : IN  std_logic_vector(N-1 DOWNTO 0);
    sum_vector_B      : IN  std_logic_vector(N-1 DOWNTO 0);
    carry_vector_B    : IN  std_logic_vector(N-1 DOWNTO 0);
    sum_vector_out    : BUFFER std_logic_vector(N-1 DOWNTO 0);
    carry_vector_out  : BUFFER std_logic_vector(N-1 DOWNTO 0);
    c_plus_d_lsb      : OUT std_logic_vector(1 DOWNTO 0);     --from memory
    adder_2_c_plus_d  : OUT std_logic_vector(1 DOWNTO 0);     --C from adder 2 and D from memory
    next_c_plus_d_lsb : BUFFER std_logic_vector(1 DOWNTO 0);  --first word C+D result stored
    next_c_lsb        : OUT std_logic_vector(1 DOWNTO 0);     --stored c lsb for higher word to follow same path
    adder_2_c_lsb     : OUT std_logic_vector(1 DOWNTO 0);     --C LSB from second level CS adder
    shift_reg_lsb     : OUT std_logic_vector(1 DOWNTO 0);
    next_p_or_zero_lsb: OUT std_logic;    --lsb results from first adder which decides to modulo reduce p
    one_detect        : OUT std_logic;    --one detect test
    zero_test         : OUT std_logic     --cs form zero or conventional zero test
    );
END datapath;
ARCHITECTURE beh OF datapath IS
-- component instantiation
COMPONENT dual_field_fa
PORT(
    a     : IN  std_logic;
    b     : IN  std_logic;
    c     : IN  std_logic;
    f_sel : IN  std_logic;
    sum   : OUT std_logic;
    carry : OUT std_logic
    );
END COMPONENT;
CONSTANT zero_vector : std_logic_vector(N DOWNTO 0) := (others => '0');
CONSTANT one_vector : std_logic_vector(N DOWNTO 0) := (others => '1');
-- signal declaration
SIGNAL c_shift_reg : std_logic_vector(N-1 DOWNTO 0);
SIGNAL ov_carry_vector_A, ov_sum_vector_A : std_logic_vector(N DOWNTO 0);
SIGNAL inv_carry_vector_B, inv_sum_vector_B : std_logic_vector(N DOWNTO 0);
SIGNAL zero_carry_vector_B, zero_sum_vector_B : std_logic_vector(N DOWNTO 0);
SIGNAL sum_1, sum_2 : std_logic_vector(N DOWNTO 0);
SIGNAL carry_1, carry_2 : std_logic_vector(N+1 DOWNTO 0);
SIGNAL p_or_zero  : std_logic_vector(N DOWNTO 0);
SIGNAL msb_shift_bit, carry_buffer_1, carry_buffer_2 : std_logic;
SIGNAL shifted_carry_2, shifted_sum_2 : std_logic_vector(N-1 DOWNTO 0);
SIGNAL zero_detect, zero_detect_reg, cs_zero_detect, cs_zero_detect_reg : std_logic;
SIGNAL all_zero_detect, all_cs_zero_detect : std_logic;
BEGIN
-- shift register for 1 cycle multiplication
shift_reg:
PROCESS (clk, reset, shift_c, load_shift_reg, c_shift_reg, sum_vector_A)
BEGIN
    IF (reset='1') THEN
        c_shift_reg <= (others => '0');
```

```
    ELSIF (clk'event AND clk='1') THEN
        IF (load_shift_reg='1') THEN
            c_shift_reg <= sum_vector_A;
        ELSIF (shift_c='1') THEN
            c_shift_reg <= '0' & c_shift_reg(N-1 DOWNTO 1);
        END IF;
    END IF;
END PROCESS;
shift_reg_lsb <= c_shift_reg(1 DOWNTO 0);  --account for delay
-- original input plus the overlap bit
ov_sum_vector_A   <= overlap_sum_A & sum_vector_A WHEN final_word='0' ELSE
                     '0' & sum_vector_A;
ov_carry_vector_A <= overlap_carry_A & carry_vector_A WHEN final_word='0' ELSE
                     '0' & carry_vector_A;
-- mux invert that outputs complemented value when selected
inv_sum_vector_B <= '0' & sum_vector_B WHEN complement_sel='0' AND final_word='1' ELSE
                    overlap_sum_B & sum_vector_B WHEN complement_sel='0' ELSE
                    NOT overlap_sum_B & NOT sum_vector_B(N-1 downto 1) & '1' WHEN odd_complement='1' AND
                                                                                  complement_sel='1' ELSE
                    NOT overlap_sum_B & NOT sum_vector_B;
inv_carry_vector_B <= '0' & carry_vector_B WHEN complement_sel='0' AND final_word='1' ELSE
                      overlap_carry_B & carry_vector_B WHEN complement_sel='0' ELSE
                      NOT overlap_carry_B & NOT carry_vector_B;
-- mux zero that outputs zero when selected
zero_sum_vector_B   <= inv_sum_vector_B WHEN B_zero_sel='0' ELSE
                       zero_vector;
-- c_shift_reg determines whether to add w
zero_carry_vector_B <= inv_carry_vector_B WHEN B_zero_sel='0' AND op='1' ELSE  --division
                       inv_carry_vector_B WHEN c_shift_reg(0)='1' AND op='0' ELSE  --multiplication
                       zero_vector;
-- dual field adder csa network 1
csa_network_1: FOR i IN N DOWNTO 0 GENERATE
    fa_map_1: dual_field_fa
    PORT MAP(
            a     => ov_carry_vector_A(i),
            b     => ov_sum_vector_A(i),
            c     => zero_carry_vector_B(i),  --inverter connected to c which puts INV in parallel to XOR
            f_sel => f_sel,
            sum   => sum_1(i),
            carry => carry_1(i+1)
            );
END GENERATE;
-- passing previous word carry to unused and empty carry space for CS adder
carry_1(0) <= carry_buffer_1 WHEN first_word='0' ELSE
              complement_sel;
-- for U & W where shift is not performed, ignored for C & D since the result is right shifted
carry_2(0) <= carry_buffer_2 WHEN first_word='0' ELSE
              complement_sel;
-- modular reduction depending on LSB of carry_1 and sum_1
p_or_zero <= zero_sum_vector_B WHEN first_word='0' OR modulo_reduce_sel='0' ELSE  --if its not first word pass it
             zero_sum_vector_B WHEN carry_1(0)='0' AND sum_1(0)='1' ELSE  --first word check whether to add p
             zero_sum_vector_B WHEN carry_1(0)='1' AND sum_1(0)='0' ELSE
             zero_vector;
```

```
-- dual field adder csa network 2
csa_network_2: FOR i IN N DOWNTO 0 GENERATE
    fa_map_2: dual_field_fa
    PORT MAP(
            a     => carry_1(i),
            b     => sum_1(i),
            c     => p_or_zero(i),
            f_sel => f_sel,
            sum   => sum_2(i),
            carry => carry_2(i+1)
            );
END GENERATE csa_network_2;
-- bit info for next round except for first word which resets at final word.
c_reg1:
PROCESS (clk, reset, carry_1, final_word)
BEGIN
    IF (reset='1') THEN
        carry_buffer_1 <= '0';
    ELSIF (clk'event AND clk='1') THEN
        IF (final_word='1') THEN  --synchronous reset
            carry_buffer_1 <= '0';
        ELSE
            carry_buffer_1 <= carry_1(N);
        END IF;
    END IF;
END PROCESS;
-- fills the empty carry_2(0) slot, which is only useful in division U+W since no shift is performed
c_reg2:
PROCESS (clk, reset, carry_2, final_word)
BEGIN
    IF (reset='1') THEN
        carry_buffer_2 <= '0';
    ELSIF (clk'event AND clk='1') THEN
        IF (final_word='1') THEN  --synchronous reset
            carry_buffer_2 <= '0';
        ELSE
            carry_buffer_2 <= carry_2(N);
        END IF;
    END IF;
END PROCESS;
-- for C+D mod 4 from memory only used at the beginning of division algorithm
c_plus_d_lsb <= (carry_vector_A(1 DOWNTO 0) + carry_vector_B(1 DOWNTO 0))
               +  (sum_vector_A(1 DOWNTO 0) + sum_vector_B(1 DOWNTO 0)) when reset='0' else
               (others=>'X');
-- C + D lsb with new C or updated C
adder_2_c_plus_d <= ( carry_2(2 DOWNTO 1) + sum_2(2 DOWNTO 1) )
                    + ( carry_vector_B(1 DOWNTO 0) + sum_vector_B(1 DOWNTO 0) ) when reset='0' else
                    (others=>'X');
-- algorithm path gets decided by the first word values of C & D; stored throughout middle and final word
PROCESS (clk, reset, first_word, carry_vector_out, carry_vector_B, sum_vector_out, sum_vector_B)
BEGIN
    IF (reset='1') THEN
        next_c_plus_d_lsb <= "00";
```

```
    ELSIF (clk'event AND clk='1') THEN
        IF (first_word='1') THEN
            next_c_plus_d_lsb <= (carry_vector_out(1 DOWNTO 0) + carry_vector_B(1 DOWNTO 0))
                              + (sum_vector_out(1 DOWNTO 0) + sum_vector_B(1 DOWNTO 0));
        END IF;
    END IF;
END PROCESS;
-- LSB bits out of second CS adder, since c is shifted use second most right bits
adder_2_c_lsb <= carry_2(1) & sum_2(1);
-- LSB for c calculation in deciding algorithms path of first word
PROCESS(clk, reset, first_word, carry_2, sum_2)
BEGIN
    IF (reset='1') THEN
        next_c_lsb <= "00";
    ELSIF (clk'event AND clk='1') THEN
        IF (first_word='1') THEN
            next_c_lsb <= carry_2(1) & sum_2(1);  --unshifted lsb
        END IF;
    END IF;
END PROCESS;
-- LSB for u of sum_1 and carry_1 decide multiplication and division prime reduce of upper words
next_p_or_zero_lsb <= carry_1(0) XOR sum_1(0);

-- determines the bit to shift into the MSB of carry save carry vector
msb_shift_bit <= (NOT sum_2(n-1) AND carry_2(n-2) AND sum_2(n-2)) OR (carry_2(N-1) AND sum_2(N-2)) OR
                 (carry_2(N-1) AND carry_2(N-2)) OR (carry_2(N-1) AND NOT sum_2(N-1));
-- professor's solution which requires one extra bit
--msb_shift_bit <= (carry_2(N-1) AND sum_2(N-2)) OR (NOT sum_2(N-1) AND carry_2(N-1)) OR
                   (NOT sum_2(N-1) AND sum_2(N-2));
-- first solution attempt
--msb_shift_bit <= carry_2(N-1) OR sum_2(N-1);
-- selects normal intermediate right shift or MSB right shift logic function
shifted_carry_2 <= carry_2(N DOWNTO 1) WHEN shift_logic='0' ELSE
                   msb_shift_bit & carry_2(N-1 DOWNTO 1);
shifted_sum_2   <= sum_2(N DOWNTO 1) WHEN shift_logic='0' ELSE
                   sum_2(N-1) & sum_2(N-1 DOWNTO 1);
-- final output mux to select shifted or normal value
carry_vector_out <= carry_2(N-1 DOWNTO 0) WHEN shifted_out_sel='0' ELSE
                    shifted_carry_2;
sum_vector_out   <= sum_2(N-1 DOWNTO 0) WHEN shifted_out_sel='0' ELSE
                    shifted_sum_2;
-- zero test logic for detecting sequence of all zeros used in GF(2^n) field
zero_detect <= '1' WHEN sum_2(N DOWNTO 1)=zero_vector(N-1 DOWNTO 0) AND
                        carry_2(N DOWNTO 1)=zero_vector(N-1 DOWNTO 0) ELSE
               '0';
-- zero test logic
all_zero_detect <= zero_detect AND zero_detect_reg;
-- zero test register which holds previous test results
zero_detect_register:
PROCESS (clk, reset, all_zero_detect, reset_zero_reg)
BEGIN
    IF (reset = '1' OR reset_zero_reg='1') THEN  --asynchronous reset
        zero_detect_reg <= '1';
```

```
    ELSIF (clk'event AND clk='1') THEN
        zero_detect_reg <= all_zero_detect;
    END IF;
END PROCESS;
-- done signal when -4 & 4;  places minimum bit of 3 constraint
cs_zero_detect <= '1' WHEN (sum_2(N DOWNTO 1)=(one_vector(N-1 DOWNTO 3) & "100")) AND
                           (carry_2(N DOWNTO 1)=(zero_vector(N-1 DOWNTO 3) & "100")) AND first_word='1' ELSE
                  '1' WHEN (sum_2(N DOWNTO 1)=one_vector(N-1 DOWNTO 0) AND
                           carry_2(N DOWNTO 1)=zero_vector(N-1 DOWNTO 0)) AND first_word='0' ELSE
                  '0';
all_cs_zero_detect <= cs_zero_detect AND cs_zero_detect_reg;
-- zero test register which holds previous test results
cs_zero_detect_register:
PROCESS (clk, reset, all_cs_zero_detect, reset_zero_reg)
BEGIN
    IF (reset = '1' OR reset_zero_reg='1') THEN
        cs_zero_detect_reg <= '1';
    ELSIF (clk = '1' AND clk'event) THEN
        cs_zero_detect_reg <= all_cs_zero_detect;
    END IF;
END PROCESS;
zero_test <= all_cs_zero_detect OR all_zero_detect;
-- one detect for checking if D=1 for division
one_detect <= '1' WHEN sum_2(1 DOWNTO 0)="01" ELSE
              '0';
END beh;
```

A.3. Control

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_vector_arith.ALL;
ENTITY control IS
GENERIC (
        PRECISION : positive := 4;  --total bit size
        N         : positive := 4;  --datapath size is N+1 bits including overlap bit
        BITS      : positive := 6   --number of bits for the dual port memory space
        );
PORT(
    reset             : IN std_logic;
    clk               : IN std_logic;
    start             : IN std_logic;
    f_sel             : IN std_logic;  --f_sel=0 GF(2^n), f_sel=1 GF(p)
    op                : IN std_logic;  --op=0 mult, op=1 div
    c_plus_d_lsb      : IN std_logic_vector(1 DOWNTO 0);  --from memory
    adder_2_c_plus_d  : IN std_logic_vector(1 DOWNTO 0);  --C from adder 2 and D from memory
    next_c_plus_d_lsb : IN std_logic_vector(1 DOWNTO 0);  --stored lsb results
    c_lsb             : IN std_logic_vector(1 DOWNTO 0);  --from mem
    adder_2_c_lsb     : IN std_logic_vector(1 DOWNTO 0);  --C LSB from second level CS adder
    next_c_lsb        : IN std_logic_vector(1 DOWNTO 0);  --LSB of c vector which is stored
```

```
    shift_reg_lsb     : IN std_logic_vector(1 DOWNTO 0);
    zero_test         : IN std_logic;  --conventional zero test or cs form zeros
    one_detect        : IN std_logic;  --one detect test
    next_p_or_zero_lsb: IN std_logic;  --stored lsb which determines whether to reduce mod p
    r_addr_1          : BUFFER std_logic_vector(BITS-1 DOWNTO 0);
    r_addr_2          : BUFFER std_logic_vector(BITS-1 DOWNTO 0);
    w_addr            : BUFFER std_logic_vector(BITS-1 DOWNTO 0);
    output_addr       : BUFFER std_logic_vector(BITS-1 DOWNTO 0); --for division where the result is stored (W or U)
    write             : OUT std_logic;
    load_shift_reg    : OUT std_logic;
    shift_c           : OUT std_logic;
    complement_sel    : OUT std_logic;
    odd_complement    : OUT std_logic;
    B_zero_sel        : OUT std_logic;
    modulo_reduce_sel : OUT std_logic;
    shifted_out_sel   : OUT std_logic;
    shift_logic       : OUT std_logic;
    first_word        : BUFFER std_logic;
    final_word        : BUFFER std_logic;
    reset_zero_reg    : OUT std_logic;
    done              : OUT std_logic
    );
END control;
ARCHITECTURE beh OF control IS
constant zero_vector : std_logic_vector(PRECISION-1 DOWNTO 0) := (others => '0');
-- constants for memory address; assumed a consecutive memory location like a fifo
constant u : std_logic_vector(BITS-1 DOWNTO 0) := (others=>'0');                            -- U register address
constant w : std_logic_vector(BITS-1 DOWNTO 0) := conv_std_logic_vector(PRECISION/N,BITS);   -- W register address
constant c : std_logic_vector(BITS-1 DOWNTO 0) := conv_std_logic_vector(PRECISION/N*2,BITS); -- C register address
constant d : std_logic_vector(BITS-1 DOWNTO 0) := conv_std_logic_vector(PRECISION/N*3,BITS); -- D register address
constant p : std_logic_vector(BITS-1 DOWNTO 0) := conv_std_logic_vector(PRECISION/N*4,BITS); -- P register address
SIGNAL counter : std_logic_vector(BITS-1 DOWNTO 0);
SIGNAL max_std_logic : std_logic_vector(BITS-1 DOWNTO 0);
-- state declaration
TYPE control_type IS(init, initial_check_c0, finished, swap, swap_back,
                     divide_c, divide_c_m, divide_c_f, u_red_c_even, u_red_c_even_m, u_red_c_even_f,
                     c_plus_d, c_plus_d_m, c_plus_d_f, c_minus_d, c_minus_d_m, c_minus_d_f,
                     u_plus_w, u_plus_w_m, u_plus_w_f, u_minus_w, u_minus_w_m, u_minus_w_f,
                     u_red_pos, u_red_pos_m, u_red_pos_f, u_red_neg, u_red_neg_m, u_red_neg_f,
                     divide_d, divide_d_m, divide_d_f, w_red_d_even, w_red_d_even_m, w_red_d_even_f,
                     d_plus_c, d_plus_c_m, d_plus_c_f, d_minus_c, d_minus_c_m, d_minus_c_f,
                     w_plus_u, w_plus_u_m, w_plus_u_f, w_minus_u, w_minus_u_m, w_minus_u_f,
                     w_red_pos, w_red_pos_m, w_red_pos_f, w_red_neg, w_red_neg_m, w_red_neg_f,
                     load_shift_register, load_shift_register_f, modulo_red_exe, modulo_red_exe_m, modulo_red_exe_f,
                     u_plus_w_exe, u_plus_w_exe_m, u_plus_w_exe_f, p_minus_w, p_minus_w_m, p_minus_w_f,
                     check_d, finished_2);
SIGNAL control_ps, control_ns : control_type;
SIGNAL negate_delta, subtract_delta, sign_of_delta, reset_delta : std_logic;
SIGNAL next_word, last_word, mult_done : std_logic;
SIGNAL delta : std_logic_vector(PRECISION-1 DOWNTO 0);
SIGNAL mult_delta : std_logic_vector(N-1 DOWNTO 0);
SIGNAL load_r_addr, load_w_addr, invert_swap : std_logic;
SIGNAL z_addr, mult_cnt, r_addr_1_temp, r_addr_2_temp, w_addr_temp,
```

```
       swap_addr_w_u, swap_addr_d_c : std_logic_vector(BITS-1 DOWNTO 0);
SIGNAL load_output_addr, load_modulo_red_sel, modulo_red_sel_reg, load_mult_addr, swap_state_reg : std_logic;
BEGIN
-- finite state machine for the main control
control_FSM:
PROCESS (clk, start, reset, control_ns, control_ps, shift_reg_lsb, final_word,
         c_plus_d_lsb, sign_of_delta, f_sel, c_lsb, next_c_lsb, swap_state_reg,
         op, last_word, next_c_plus_d_lsb, mult_done, modulo_red_sel_reg,
         adder_2_c_lsb, adder_2_c_plus_d, zero_test, mult_delta, one_detect,
         swap_addr_d_c, swap_addr_w_u)
BEGIN
    IF(reset='1') THEN
        control_ps <= init;
    ELSIF (clk'event AND clk='1') THEN
        control_ps <= control_ns;
    END IF;
    --default outputs
    load_shift_reg <= '0';
    shift_c <= '0';
    write <= '0';
    B_zero_sel <= '0';
    modulo_reduce_sel <= '0';  --some requires modulo reduction for division
    shifted_out_sel <= '0';
    shift_logic <= '0';
    complement_sel <= '0';
    odd_complement <= '0';
    subtract_delta <= '0';
    negate_delta <= '0';
    done <= '0';
    next_word <= '0';
    first_word <= '0';
    load_r_addr <= '0';
    load_w_addr <= '0';
    r_addr_1_temp <= (others=>'0');
    r_addr_2_temp <= (others=>'0');
    w_addr_temp <= (others=>'0');
    z_addr <= (others=>'0');
    reset_zero_reg <= '0';
    reset_delta <= '0';
    load_modulo_red_sel <= '0';
    load_mult_addr <= '0';
    invert_swap <= '0';
    load_output_addr <= '0';
    CASE control_ps IS
    WHEN init =>
        IF (start='1' AND op='0') THEN
            control_ns <= load_shift_register;
        ELSIF (start='1') THEN
            control_ns <= initial_check_c0;
        ELSE
            control_ns <= init;
        END IF;
--------------------multiplication----------------------------
```

```
 WHEN load_shift_register =>
     load_shift_reg <= '1';
     r_addr_1_temp <= c;
     load_r_addr <= '1';
     reset_delta <= '1';
     IF (c_lsb(0)='0') THEN
         control_ns <= modulo_red_exe;
     ELSE
         control_ns <= u_plus_w_exe;
     END IF;
 WHEN load_shift_register_f =>
     load_shift_reg <= '1';
     r_addr_1_temp <= c;
     load_mult_addr <= '1';
     reset_delta <= '1';
     IF (c_lsb(0)='0') THEN
         control_ns <= modulo_red_exe;
     ELSE
         control_ns <= u_plus_w_exe;
     END IF;
 --first word uses first address, which gets selected by load_r_addr & load_w_addr
 WHEN modulo_red_exe =>
     modulo_reduce_sel <= '1';
     first_word <= '1';
     shifted_out_sel <= '1';
     w_addr_temp <= u;
     load_w_addr <= '1';
     write <= '1';
     r_addr_1_temp <= u;
     r_addr_2_temp <= w;  --w is normal representation
     load_r_addr <= '1';
     next_word <= '1';
     load_modulo_red_sel <= '1';
     shift_c <= final_word;
     subtract_delta <= final_word;  --delta tracks internal word size precision which get reset
     IF (last_word='1') THEN
         control_ns <= modulo_red_exe_f;
     ELSIF (mult_done='1' AND final_word='1') THEN
         control_ns <= finished;
     ELSIF (mult_delta(0)='1' AND final_word='1') THEN --when PRECISION equals N
         control_ns <= load_shift_register_f;
     ELSIF (shift_reg_lsb(1)='0' AND final_word='1') THEN  --for one cycle or one word precision
         control_ns <= modulo_red_exe;
     ELSIF (shift_reg_lsb(1)='1' AND final_word='1') THEN
         control_ns <= u_plus_w_exe;
     ELSE
         control_ns <= modulo_red_exe_m;
     END IF;
--different addresses are used after the first word and same path regardless of current LSB for modulo reduction
 WHEN modulo_red_exe_m =>
     B_zero_sel <= NOT modulo_red_sel_reg;
     shifted_out_sel <= '1';
     w_addr_temp <= u;
```

```
        write <= '1';
        r_addr_1_temp <= u;
        r_addr_2_temp <= w;  --w is normal representation
        next_word <= '1';
        IF (last_word='1') THEN
            control_ns <= modulo_red_exe_f;
        ELSE
            control_ns <= modulo_red_exe_m;
        END IF;
    WHEN modulo_red_exe_f =>
        B_zero_sel <= NOT modulo_red_sel_reg;
        shifted_out_sel <= '1';
        --shift_logic <= '1';
        shift_c <= '1';
        subtract_delta <= '1';  --delta tracks internal word size precision which get reset
        w_addr_temp <= u;
        write <= '1';
        r_addr_1_temp <= u;
        r_addr_2_temp <= w;
        IF (mult_done='1') THEN
            control_ns <= finished;
        ELSIF (mult_delta(0)='1') THEN
            control_ns <= load_shift_register_f;
        ELSIF (shift_reg_lsb(1)='0') THEN
            control_ns <= modulo_red_exe;
        ELSE
            control_ns <= u_plus_w_exe;
        END IF;
    WHEN u_plus_w_exe =>
        modulo_reduce_sel <= '1';
        first_word <= '1';
        shifted_out_sel <= '1';
        w_addr_temp <= u;
        load_w_addr <= '1';
        write <= '1';
        r_addr_1_temp <= u;
        r_addr_2_temp <= w;
        load_r_addr <= '1';
        next_word <= '1';
        load_modulo_red_sel <= '1';
        shift_c <= final_word;
        subtract_delta <= final_word;  --delta tracks internal word size precision which get reset
        IF (last_word='1') THEN
            control_ns <= u_plus_w_exe_f;
        ELSIF (mult_done='1' AND final_word='1') THEN
            control_ns <= finished;
        ELSIF (mult_delta(0)='1' AND final_word='1') THEN   ---------
            control_ns <= load_shift_register_f;
        ELSIF (shift_reg_lsb(1)='0' AND final_word='1') THEN
            control_ns <= modulo_red_exe;
        ELSIF (shift_reg_lsb(1)='1' AND final_word='1') THEN
            control_ns <= u_plus_w_exe;
        ELSE
```

```
            control_ns <= u_plus_w_exe_m;
        END IF;
    WHEN u_plus_w_exe_m =>
        B_zero_sel <= NOT modulo_red_sel_reg;
        shifted_out_sel <= '1';
        w_addr_temp <= u;
        write <= '1';
        r_addr_1_temp <= u;
        r_addr_2_temp <= w;
        next_word <= '1';
        IF (last_word='1') THEN
            control_ns <= u_plus_w_exe_f;
        ELSE
            control_ns <= u_plus_w_exe_m;
        END IF;
    WHEN u_plus_w_exe_f =>
        B_zero_sel <= NOT modulo_red_sel_reg;
        shifted_out_sel <= '1';
        --shift_logic <= '1';
        shift_c <= '1';
        subtract_delta <= '1';
        w_addr_temp <= u;
        write <= '1';
        r_addr_1_temp <= u;
        r_addr_2_temp <= w;
        IF (mult_done='1') THEN
            control_ns <= finished;
        ELSIF (mult_delta(0)='1') THEN
            control_ns <= load_shift_register_f;
        ELSIF (shift_reg_lsb(1)='0') THEN
            control_ns <= modulo_red_exe;
        ELSE
            control_ns <= u_plus_w_exe;
        END IF;
------------------------division---------------------------
    WHEN initial_check_c0 =>  --next state cannot be swap
        B_zero_sel <= '1';    --for checking zero to settle
        r_addr_1_temp <= c;
        r_addr_2_temp <= d;
        load_r_addr <= '1';
        IF ( c_lsb="00" OR c_lsb="11" ) THEN
            control_ns <= u_red_c_even;
        ELSIF (c_plus_d_lsb/="00" AND f_sel='1') THEN
            control_ns <= u_minus_w;
        ELSE
            control_ns <= u_plus_w;
        END IF;
    WHEN swap =>
        negate_delta <= '1';
        invert_swap <= '1';
        IF (next_c_plus_d_lsb/="00" AND f_sel='1') THEN
            control_ns <= w_minus_u;
        ELSE
```

```
        control_ns <= w_plus_u;
    END IF;
WHEN u_red_c_even =>
    reset_zero_reg <= '1';  --enables reset reg to 1 in preparation for testing C for zero
    load_modulo_red_sel <= '1';
    first_word <= '1';
    subtract_delta <= '1';
    next_word <= '1';
    modulo_reduce_sel <= '1';
    shifted_out_sel <= '1';
    shift_logic <= final_word;
    w_addr_temp <= u;
    load_w_addr <= '1';
    write <= '1';
    r_addr_1_temp <= u;
    r_addr_2_temp <= p;
    load_r_addr <= '1';
    IF (last_word='1') THEN  --accounts for 2 word precision skipping middle states
        control_ns <= u_red_c_even_f;
    ELSIF (final_word='1') THEN  --accounts for 1 word precision skipping middle and final states
        control_ns <= divide_c;
    ELSE
        control_ns <= u_red_c_even_m;
    END IF;
WHEN u_red_c_even_m =>
    B_zero_sel <= NOT modulo_red_sel_reg;
    next_word <= '1';
    shifted_out_sel <= '1';
    w_addr_temp <= u;
    write <= '1';
    r_addr_1_temp <= u;
    r_addr_2_temp <= p;
    IF (last_word='1') THEN
        control_ns <= u_red_c_even_f;
    ELSE
        control_ns <= u_red_c_even_m;
    END IF;
WHEN u_red_c_even_f =>
    B_zero_sel <= NOT modulo_red_sel_reg;
    reset_zero_reg <= '1';  --enables reset reg to 1 in preparation for testing C for zero
    shifted_out_sel <= '1';
    shift_logic <= '1';
    w_addr_temp <= u;
    write <= '1';
    r_addr_1_temp <= u;
    r_addr_2_temp <= p;
    control_ns <= divide_c;
WHEN divide_c =>
    first_word <= '1';
    next_word <= '1';
    B_zero_sel <= '1';
    shifted_out_sel <= '1';
    shift_logic <= final_word;
```

```
        r_addr_1_temp <= c;
        r_addr_2_temp <= d;
        load_r_addr <= '1';
        w_addr_temp <= c;
        load_w_addr <= '1';
        write <= '1';
        IF (last_word='1') THEN
            control_ns <= divide_c_f;
        ELSIF (zero_test='1' AND final_word='1') THEN
            control_ns <= check_d;
        ELSIF ( (adder_2_c_lsb="00" OR adder_2_c_lsb="11") AND final_word='1') THEN
            control_ns <= u_red_c_even;
        ELSIF (sign_of_delta='1' AND final_word='1') THEN
            control_ns <= swap;
        ELSIF (final_word='1' AND f_sel='0') THEN
            control_ns <= u_plus_w;
        ELSIF ( (adder_2_c_plus_d/="00" AND f_sel='1') AND final_word='1') THEN
            control_ns <= u_minus_w;
        ELSIF ( (adder_2_c_plus_d="00" AND f_sel='1') AND final_word='1') THEN
            control_ns <= u_plus_w;
        ELSE
            control_ns <= divide_c_m;
        END IF;
    WHEN divide_c_m =>
        next_word <= '1';
        B_zero_sel <= '1';
        shifted_out_sel <= '1';
        r_addr_1_temp <= c;
        r_addr_2_temp <= d;
        w_addr_temp <= c;
        write <= '1';
        IF (last_word='1') THEN
            control_ns <= divide_c_f;
        ELSE
            control_ns <= divide_c_m;
        END IF;
    WHEN divide_c_f =>
        B_zero_sel <= '1';
        shifted_out_sel <= '1';
        shift_logic <= '1';
        r_addr_1_temp <= c;
        r_addr_2_temp <= d;
        w_addr_temp <= c;
        write <= '1';
        IF (zero_test='1') THEN  --last word passes zero test and previous ones
            control_ns <= check_d;
        ELSIF (next_c_lsb="00" OR next_c_lsb="11") THEN
            control_ns <= u_red_c_even;
        ELSIF (sign_of_delta='1') THEN
            control_ns <= swap;
        ELSIF (next_c_plus_d_lsb/="00" AND f_sel='1') THEN
            control_ns <= u_minus_w;
        ELSE
```

```
        control_ns <= u_plus_w;
    END IF;
WHEN u_plus_w =>
    first_word <= '1';
    next_word <= '1';
    r_addr_1_temp <= u;
    r_addr_2_temp <= w;
    load_r_addr <= '1';
    w_addr_temp <= u;
    load_w_addr <= '1';
    write <= '1';
    subtract_delta <= '1';
    IF (last_word='1') THEN
        control_ns <= u_plus_w_f;
    ELSIF (final_word='1') THEN
        control_ns <= u_red_pos;
    ELSE
        control_ns <= u_plus_w_m;
    END IF;
WHEN u_plus_w_m =>
    next_word <= '1';
    r_addr_1_temp <= u;
    r_addr_2_temp <= w;
    w_addr_temp <= u;
    write <= '1';
    IF (last_word='1') THEN
        control_ns <= u_plus_w_f;
    ELSE
        control_ns <= u_plus_w_m;
    END IF;
WHEN u_plus_w_f =>
    r_addr_1_temp <= u;
    r_addr_2_temp <= w;
    w_addr_temp <= u;
    write <= '1';
    control_ns <= u_red_pos;
WHEN u_red_pos =>
    reset_zero_reg <= '1';  --enables reset reg to 1 in preparation for testing C for zero
    load_modulo_red_sel <= '1';
    first_word <= '1';
    next_word <= '1';
    modulo_reduce_sel <= '1';
    shifted_out_sel <= '1';
    shift_logic <= final_word;
    w_addr_temp <= u;
    write <= '1';
    load_w_addr <= '1';
    r_addr_1_temp <= u;
    r_addr_2_temp <= p;
    load_r_addr <= '1';
    IF (last_word='1') THEN
        control_ns <= u_red_pos_f;
    ELSIF (final_word='1') THEN
```

```
            control_ns <= c_plus_d;
        ELSE
            control_ns <= u_red_pos_m;
        END IF;
    WHEN u_red_pos_m =>
        B_zero_sel <= NOT modulo_red_sel_reg;
        next_word <= '1';
        shifted_out_sel <= '1';
        w_addr_temp <= u;
        write <= '1';
        r_addr_1_temp <= u;
        r_addr_2_temp <= p;
        IF (last_word='1') THEN
            control_ns <= u_red_pos_f;
        ELSE
            control_ns <= u_red_pos_m;
        END IF;
    WHEN u_red_pos_f =>
        B_zero_sel <= NOT modulo_red_sel_reg;
        reset_zero_reg <= '1';
        shifted_out_sel <= '1';
        shift_logic <= '1';
        w_addr_temp <= u;
        write <= '1';
        r_addr_1_temp <= u;
        r_addr_2_temp <= p;
        control_ns <= c_plus_d;
    WHEN c_plus_d =>
        first_word <= '1';
        next_word <= '1';
        shifted_out_sel <= '1';
        shift_logic <= final_word;
        w_addr_temp <= c;
        write <= '1';
        load_w_addr <= '1';
        r_addr_1_temp <= c;
        r_addr_2_temp <= d;
        load_r_addr <= '1';
        IF (last_word='1') THEN
            control_ns <= c_plus_d_f;
        ELSIF (zero_test='1' AND final_word='1') THEN
            control_ns <= check_d;
        ELSIF ( (adder_2_c_lsb="00" OR adder_2_c_lsb="11") AND final_word='1') THEN
            control_ns <= u_red_c_even;
        ELSIF (sign_of_delta='1' AND final_word='1') THEN
            control_ns <= swap;
        ELSIF (final_word='1' AND f_sel='0') THEN
            control_ns <= u_plus_w;
        ELSIF ( (adder_2_c_plus_d/="00" AND f_sel='1') AND final_word='1') THEN
            control_ns <= u_minus_w;
        ELSIF ( (adder_2_c_plus_d="00" AND f_sel='1') AND final_word='1') THEN
            control_ns <= u_plus_w;
        ELSE
```

```
        control_ns <= c_plus_d_m;
    END IF;
WHEN c_plus_d_m =>
    next_word <= '1';
    shifted_out_sel <= '1';
    w_addr_temp <= c;
    write <= '1';
    r_addr_1_temp <= c;
    r_addr_2_temp <= d;
    IF (last_word='1') THEN
        control_ns <= c_plus_d_f;
    ELSE
        control_ns <= c_plus_d_m;
    END IF;
WHEN c_plus_d_f =>
    shifted_out_sel <= '1';
    shift_logic <= '1';
    w_addr_temp <= c;
    write <= '1';
    r_addr_1_temp <= c;
    r_addr_2_temp <= d;
    IF (zero_test='1') THEN
        control_ns <= check_d;
    ELSIF (next_c_lsb="00" OR next_c_lsb="11") THEN
        control_ns <= u_red_c_even;
    ELSIF (sign_of_delta='1') THEN
        control_ns <= swap;
    ELSIF (next_c_plus_d_lsb/="00" AND f_sel='1') THEN
        control_ns <= u_minus_w;
    ELSE
        control_ns <= u_plus_w;
    END IF;
WHEN u_minus_w =>
    first_word <= '1';
    next_word <= '1';
    r_addr_1_temp <= u;
    r_addr_2_temp <= w;
    load_r_addr <= '1';
    w_addr_temp <= u;
    write <= '1';
    load_w_addr <= '1';
    complement_sel <= '1';
    IF (last_word='1') THEN
        control_ns <= u_minus_w_f;
    ELSIF (final_word='1') THEN
        control_ns <= u_red_neg;
    ELSE
        control_ns <= u_minus_w_m;
    END IF;
WHEN u_minus_w_m =>
    next_word <= '1';
    r_addr_1_temp <= u;
    r_addr_2_temp <= w;
```

```
        w_addr_temp <= u;
        write <= '1';
        complement_sel <= '1';
        IF (last_word='1') THEN
            control_ns <= u_minus_w_f;
        ELSE
            control_ns <= u_minus_w_m;
        END IF;
    WHEN u_minus_w_f =>
        r_addr_1_temp <= u;
        r_addr_2_temp <= w;
        w_addr_temp <= u;
        write <= '1';
        complement_sel <= '1';
        control_ns <= u_red_neg;
    WHEN u_red_neg =>
        reset_zero_reg <= '1';
        load_modulo_red_sel <= '1';
        first_word <= '1';
        next_word <= '1';
        modulo_reduce_sel <= '1';
        shifted_out_sel <= '1';
        shift_logic <= final_word;
        w_addr_temp <= u;
        write <= '1';
        load_w_addr <= '1';
        r_addr_1_temp <= u;
        r_addr_2_temp <= p;
        load_r_addr <= '1';
        IF (last_word='1') THEN
            control_ns <= u_red_neg_f;
        ELSIF (final_word='1') THEN
            control_ns <= c_minus_d;
        ELSE
            control_ns <= u_red_neg_m;
        END IF;
    WHEN u_red_neg_m =>
        B_zero_sel <= NOT modulo_red_sel_reg;
        next_word <= '1';
        shifted_out_sel <= '1';
        w_addr_temp <= u;
        write <= '1';
        r_addr_1_temp <= u;
        r_addr_2_temp <= p;
        IF (last_word='1') THEN
            control_ns <= u_red_neg_f;
        ELSE
            control_ns <= u_red_neg_m;
        END IF;
    WHEN u_red_neg_f =>
        B_zero_sel <= NOT modulo_red_sel_reg;
        reset_zero_reg <= '1';
        shifted_out_sel <= '1';
```

```
    shift_logic <= '1';
    w_addr_temp <= u;
    write <= '1';
    r_addr_1_temp <= u;
    r_addr_2_temp <= p;
    control_ns <= c_minus_d;
WHEN c_minus_d=>
    first_word <= '1';
    next_word <= '1';
    shifted_out_sel <= '1';
    shift_logic <= final_word;
    w_addr_temp <= c;
    write <= '1';
    load_w_addr <= '1';
    r_addr_1_temp <= c;
    r_addr_2_temp <= d;
    load_r_addr <= '1';
    complement_sel <= '1';
    odd_complement <= '1';  --only applies to the first word
    IF (last_word='1') THEN
        control_ns <= c_minus_d_f;
    ELSIF (zero_test='1' AND final_word='1') THEN
        control_ns <= check_d;
    ELSIF ( (adder_2_c_lsb="00" OR adder_2_c_lsb="11") AND final_word='1') THEN
        control_ns <= u_red_c_even;
    ELSIF (sign_of_delta='1' AND final_word='1') THEN
        control_ns <= swap;
    ELSIF (final_word='1' AND f_sel='0') THEN
        control_ns <= u_plus_w;
    ELSIF ( (adder_2_c_plus_d/="00" AND f_sel='1') AND final_word='1') THEN
        control_ns <= u_minus_w;
    ELSIF ( (adder_2_c_plus_d="00" AND f_sel='1') AND final_word='1') THEN
        control_ns <= u_plus_w;
    ELSE
        control_ns <= c_minus_d_m;
    END IF;
WHEN c_minus_d_m=>
    next_word <= '1';
    shifted_out_sel <= '1';
    w_addr_temp <= c;
    write <= '1';
    r_addr_1_temp <= c;
    r_addr_2_temp <= d;
    complement_sel <= '1';
    IF (last_word='1') THEN
        control_ns <= c_minus_d_f;
    ELSE
        control_ns <= c_minus_d_m;
    END IF;
WHEN c_minus_d_f =>
    shifted_out_sel <= '1';
    shift_logic <= '1';
    w_addr_temp <= c;
```

```
        write <= '1';
        r_addr_1_temp <= c;
        r_addr_2_temp <= d;
        complement_sel <= '1';
        IF (zero_test='1') THEN
            control_ns <= check_d;
        ELSIF (next_c_lsb="00" OR next_c_lsb="11") THEN
            control_ns <= u_red_c_even;
        ELSIF (sign_of_delta='1') THEN
            control_ns <= swap;
        ELSIF (next_c_plus_d_lsb/="00" AND f_sel='1') THEN
            control_ns <= u_minus_w;
        ELSE
            control_ns <= u_plus_w;
        END IF;
    -- swaped control --
    WHEN swap_back =>
        negate_delta <= '1';
        invert_swap <= '1';
        --r_addr_1_temp <= u;
        --r_addr_2_temp <= w;
        IF (next_c_plus_d_lsb/="00" AND f_sel='1') THEN
            control_ns <= u_minus_w;
        ELSE
            control_ns <= u_plus_w;
        END IF;
    WHEN w_red_d_even =>
        reset_zero_reg <= '1';
        load_modulo_red_sel <= '1';
        first_word <= '1';
        subtract_delta <= '1';
        next_word <= '1';
        modulo_reduce_sel <= '1';
        shifted_out_sel <= '1';
        shift_logic <= final_word;
        w_addr_temp <= w;
        write <= '1';
        load_w_addr <= '1';
        r_addr_1_temp <= w;
        r_addr_2_temp <= p;
        load_r_addr <= '1';
        IF (last_word='1') THEN
            control_ns <= w_red_d_even_f;
        ELSIF (final_word='1') THEN
            control_ns <= divide_d;
        ELSE
            control_ns <= w_red_d_even_m;
        END IF;
    WHEN w_red_d_even_m =>
        B_zero_sel <= NOT modulo_red_sel_reg;
        next_word <= '1';
        shifted_out_sel <= '1';
        w_addr_temp <= w;
```

```
        write <= '1';
        r_addr_1_temp <= w;
        r_addr_2_temp <= p;
        IF (last_word='1') THEN
            control_ns <= w_red_d_even_f;
        ELSE
            control_ns <= w_red_d_even_m;
        END IF;
    WHEN w_red_d_even_f =>
        B_zero_sel <= NOT modulo_red_sel_reg;
        reset_zero_reg <= '1';
        shifted_out_sel <= '1';
        shift_logic <= '1';
        w_addr_temp <= w;
        write <= '1';
        r_addr_1_temp <= w;
        r_addr_2_temp <= p;
        control_ns <= divide_d;
    WHEN divide_d =>
        first_word <= '1';
        next_word <= '1';
        B_zero_sel <= '1';
        shifted_out_sel <= '1';
        shift_logic <= final_word;
        r_addr_1_temp <= d;
        r_addr_2_temp <= c;
        load_r_addr <= '1';
        w_addr_temp <= d;
        write <= '1';
        load_w_addr <= '1';
        IF (last_word='1') THEN
            control_ns <= divide_d_f;
        ELSIF (zero_test='1' AND final_word='1') THEN
            control_ns <= check_d;
        ELSIF ( (adder_2_c_lsb="00" OR adder_2_c_lsb="11") AND final_word='1') THEN
            control_ns <= w_red_d_even;
        ELSIF (sign_of_delta='1' AND final_word='1') THEN
            control_ns <= swap_back;
        ELSIF (final_word='1' AND f_sel='0') THEN
            control_ns <= w_plus_u;
        ELSIF ( (adder_2_c_plus_d/="00" AND f_sel='1') AND final_word='1') THEN
            control_ns <= w_minus_u;
        ELSIF ( (adder_2_c_plus_d="00" AND f_sel='1') AND final_word='1') THEN
            control_ns <= w_plus_u;
        ELSE
            control_ns <= divide_d_m;
        END IF;
    WHEN divide_d_m =>
        next_word <= '1';
        B_zero_sel <= '1';
        shifted_out_sel <= '1';
        r_addr_1_temp <= d;
        r_addr_2_temp <= c;
```

```
        w_addr_temp <= d;
        write <= '1';
        IF (last_word='1') THEN
            control_ns <= divide_d_f;
        ELSE
            control_ns <= divide_d_m;
        END IF;
    WHEN divide_d_f =>
        B_zero_sel <= '1';
        shifted_out_sel <= '1';
        shift_logic <= '1';
        r_addr_1_temp <= d;
        r_addr_2_temp <= c;
        w_addr_temp <= d;
        write <= '1';
        IF (zero_test='1') THEN
            control_ns <= check_d;
        ELSIF (next_c_lsb="00" OR next_c_lsb="11") THEN
            control_ns <= w_red_d_even;
        ELSIF (sign_of_delta='1') THEN
            control_ns <= swap_back;
        ELSIF (next_c_plus_d_lsb/="00" AND f_sel='1') THEN
            control_ns <= w_minus_u;
        ELSE
            control_ns <= w_plus_u;
        END IF;
    WHEN w_plus_u =>
        first_word <= '1';
        next_word <= '1';
        r_addr_1_temp <= w;
        r_addr_2_temp <= u;
        load_r_addr <= '1';
        w_addr_temp <= w;
        write <= '1';
        load_w_addr <= '1';
        subtract_delta <= '1';
        IF (last_word='1') THEN
            control_ns <= w_plus_u_f;
        ELSIF (final_word='1') THEN
            control_ns <= w_red_pos;
        ELSE
            control_ns <= w_plus_u_m;
        END IF;
    WHEN w_plus_u_m =>
        next_word <= '1';
        r_addr_1_temp <= w;
        r_addr_2_temp <= u;
        w_addr_temp <= w;
        write <= '1';
        IF (last_word='1') THEN
            control_ns <= w_plus_u_f;
        ELSE
            control_ns <= w_plus_u_m;
```

```
    END IF;
WHEN w_plus_u_f =>
    r_addr_1_temp <= w;
    r_addr_2_temp <= u;
    w_addr_temp <= w;
    write <= '1';
    control_ns <= w_red_pos;
WHEN w_red_pos =>
    reset_zero_reg <= '1';
    load_modulo_red_sel <= '1';
    first_word <= '1';
    next_word <= '1';
    modulo_reduce_sel <= '1';
    shifted_out_sel <= '1';
    shift_logic <= final_word;
    w_addr_temp <= w;
    write <= '1';
    load_w_addr <= '1';
    r_addr_1_temp <= w;
    r_addr_2_temp <= p;
    load_r_addr <= '1';
    IF (last_word='1') THEN
        control_ns <= w_red_pos_f;
    ELSIF (final_word='1') THEN
        control_ns <= d_plus_c;
    ELSE
        control_ns <= w_red_pos_m;
    END IF;
WHEN w_red_pos_m =>
    B_zero_sel <= NOT modulo_red_sel_reg;
    next_word <= '1';
    shifted_out_sel <= '1';
    w_addr_temp <= w;
    write <= '1';
    r_addr_1_temp <= w;
    r_addr_2_temp <= p;
    IF (last_word='1') THEN
        control_ns <= w_red_pos_f;
    ELSE
        control_ns <= w_red_pos_m;
    END IF;
WHEN w_red_pos_f =>
    B_zero_sel <= NOT modulo_red_sel_reg;
    reset_zero_reg <= '1';
    shifted_out_sel <= '1';
    shift_logic <= '1';
    w_addr_temp <= w;
    write <= '1';
    r_addr_1_temp <= w;
    r_addr_2_temp <= p;
    control_ns <= d_plus_c;
WHEN d_plus_c =>
    first_word <= '1';
```

```
        next_word <= '1';
        shifted_out_sel <= '1';
        shift_logic <= final_word;
        w_addr_temp <= d;
        write <= '1';
        load_w_addr <= '1';
        r_addr_1_temp <= d;
        r_addr_2_temp <= c;
        load_r_addr <= '1';
        IF (last_word='1') THEN
            control_ns <= d_plus_c_f;
        ELSIF (zero_test='1' AND final_word='1') THEN
            control_ns <= check_d;
        ELSIF ( (adder_2_c_lsb="00" OR adder_2_c_lsb="11") AND final_word='1') THEN
            control_ns <= w_red_d_even;
        ELSIF (sign_of_delta='1' AND final_word='1') THEN
            control_ns <= swap_back;
        ELSIF (final_word='1' AND f_sel='0') THEN
            control_ns <= w_plus_u;
        ELSIF ( (adder_2_c_plus_d/="00" AND f_sel='1') AND final_word='1') THEN
            control_ns <= w_minus_u;
        ELSIF ( (adder_2_c_plus_d="00" AND f_sel='1') AND final_word='1') THEN
            control_ns <= w_plus_u;
        ELSE
            control_ns <= d_plus_c_m;
        END IF;
    WHEN d_plus_c_m =>
        next_word <= '1';
        shifted_out_sel <= '1';
        w_addr_temp <= d;
        write <= '1';
        r_addr_1_temp <= d;
        r_addr_2_temp <= c;
        IF (last_word='1') THEN
            control_ns <= d_plus_c_f;
        ELSE
            control_ns <= d_plus_c_m;
        END IF;
    WHEN d_plus_c_f =>
        shifted_out_sel <= '1';
        shift_logic <= '1';
        w_addr_temp <= d;
        write <= '1';
        r_addr_1_temp <= d;
        r_addr_2_temp <= c;
        IF (zero_test='1') THEN
            control_ns <= check_d;
        ELSIF (next_c_lsb="00" OR next_c_lsb="11") THEN
            control_ns <= w_red_d_even;
        ELSIF (sign_of_delta='1') THEN
            control_ns <= swap_back;
        ELSIF (next_c_plus_d_lsb/="00" AND f_sel='1') THEN
            control_ns <= w_minus_u;
```

```
    ELSE
        control_ns <= w_plus_u;
    END IF;
WHEN w_minus_u =>
    first_word <= '1';
    next_word <= '1';
    r_addr_1_temp <= w;
    r_addr_2_temp <= u;
    load_r_addr <= '1';
    w_addr_temp <= w;
    write <= '1';
    load_w_addr <= '1';
    complement_sel <= '1';
    IF (last_word='1') THEN
        control_ns <= w_minus_u_f;
    ELSIF (final_word='1') THEN
        control_ns <= w_red_neg;
    ELSE
        control_ns <= w_minus_u_m;
    END IF;
WHEN w_minus_u_m =>
    next_word <= '1';
    r_addr_1_temp <= w;
    r_addr_2_temp <= u;
    w_addr_temp <= w;
    write <= '1';
    complement_sel <= '1';
    IF (last_word='1') THEN
        control_ns <= w_minus_u_f;
    ELSE
        control_ns <= w_minus_u_m;
    END IF;
WHEN w_minus_u_f =>
    r_addr_1_temp <= w;
    r_addr_2_temp <= u;
    w_addr_temp <= w;
    write <= '1';
    complement_sel <= '1';
    control_ns <= w_red_neg;
WHEN w_red_neg =>
    reset_zero_reg <= '1';
    load_modulo_red_sel <= '1';
    first_word <= '1';
    next_word <= '1';
    modulo_reduce_sel <= '1';
    shifted_out_sel <= '1';
    shift_logic <= final_word;
    w_addr_temp <= w;
    write <= '1';
    load_w_addr <= '1';
    r_addr_1_temp <= w;
    r_addr_2_temp <= p;
    load_r_addr <= '1';
```

```
    IF (last_word='1') THEN
        control_ns <= w_red_neg_f;
    ELSIF (final_word='1') THEN
        control_ns <= d_minus_c;
    ELSE
        control_ns <= w_red_neg_m;
    END IF;
WHEN w_red_neg_m =>
    B_zero_sel <= NOT modulo_red_sel_reg;
    next_word <= '1';
    shifted_out_sel <= '1';
    w_addr_temp <= w;
    write <= '1';
    r_addr_1_temp <= w;
    r_addr_2_temp <= p;
    IF (last_word='1') THEN
        control_ns <= w_red_neg_f;
    ELSE
        control_ns <= w_red_neg_m;
    END IF;
WHEN w_red_neg_f =>
    B_zero_sel <= NOT modulo_red_sel_reg;
    reset_zero_reg <= '1';
    shifted_out_sel <= '1';
    shift_logic <= '1';
    w_addr_temp <= w;
    write <= '1';
    r_addr_1_temp <= w;
    r_addr_2_temp <= p;
    control_ns <= d_minus_c;
WHEN d_minus_c =>
    first_word <= '1';
    next_word <= '1';
    shifted_out_sel <= '1';
    shift_logic <= final_word;
    w_addr_temp <= d;
    write <= '1';
    load_w_addr <= '1';
    r_addr_1_temp <= d;
    r_addr_2_temp <= c;
    load_r_addr <= '1';
    complement_sel <= '1';
    odd_complement <= '1';
    IF (last_word='1') THEN
        control_ns <= d_minus_c_f;
    ELSIF (zero_test='1' AND final_word='1') THEN
        control_ns <= check_d;
    ELSIF ( (adder_2_c_lsb="00" OR adder_2_c_lsb="11") AND final_word='1') THEN
        control_ns <= w_red_d_even;
    ELSIF (sign_of_delta='1' AND final_word='1') THEN
        control_ns <= swap_back;
    ELSIF (final_word='1' AND f_sel='0') THEN
        control_ns <= w_plus_u;
```

```
        ELSIF ( (adder_2_c_plus_d/="00" AND f_sel='1') AND final_word='1') THEN
            control_ns <= w_minus_u;
        ELSIF ( (adder_2_c_plus_d="00" AND f_sel='1') AND final_word='1') THEN
            control_ns <= w_plus_u;
        ELSE
            control_ns <= d_minus_c_m;
        END IF;
    WHEN d_minus_c_m =>
        next_word <= '1';
        shifted_out_sel <= '1';
        w_addr_temp <= d;
        write <= '1';
        r_addr_1_temp <= d;
        r_addr_2_temp <= c;
        complement_sel <= '1';
        IF (last_word='1') THEN
            control_ns <= d_minus_c_f;
        ELSE
            control_ns <= d_minus_c_m;
        END IF;
    WHEN d_minus_c_f =>
        shifted_out_sel <= '1';
        shift_logic <= '1';
        w_addr_temp <= d;
        write <= '1';
        r_addr_1_temp <= d;
        r_addr_2_temp <= c;
        complement_sel <= '1';
        IF (zero_test='1') THEN
            control_ns <= check_d;
        ELSIF (next_c_lsb="00" OR next_c_lsb="11") THEN
            control_ns <= w_red_d_even;
        ELSIF (sign_of_delta='1') THEN
            control_ns <= swap_back;
        ELSIF (next_c_plus_d_lsb/="00" AND f_sel='1') THEN
            control_ns <= w_minus_u;
        ELSE
            control_ns <= w_plus_u;
        END IF;
    --check if D=1 OR C=1 when swap_state=1, just the first word required for this check
    WHEN check_d =>
        B_zero_sel <= '1';
        r_addr_1_temp <= swap_addr_d_c;
        load_r_addr <= '1';
        IF(one_detect='1' AND swap_state_reg='1') THEN
            control_ns <= finished;
        ELSIF(one_detect='1') THEN
            control_ns <= finished_2;
        ELSE
            control_ns <= p_minus_w;
        END IF;
    --for division if D!=1 then Z=P-W OR Z=P-U when swap_state=1
    WHEN p_minus_w =>
```

```
        first_word <= '1';
        next_word <= '1';
        r_addr_1_temp <= p;
        r_addr_2_temp <= swap_addr_w_u;
        load_r_addr <= '1';
        w_addr_temp <= swap_addr_w_u;
        write <= '1';
        load_w_addr <= '1';
        complement_sel <= '1';
        IF (last_word='1') THEN
            control_ns <= p_minus_w_f;
        ELSIF (final_word='1' AND swap_state_reg='1') THEN
            control_ns <= finished;
        ELSIF (final_word='1') THEN
            control_ns <= finished_2;
        ELSE
            control_ns <= p_minus_w_m;
        END IF;
    WHEN p_minus_w_m =>
        next_word <= '1';
        r_addr_1_temp <= p;
        r_addr_2_temp <= swap_addr_w_u;
        w_addr_temp <= swap_addr_w_u;
        write <= '1';
        complement_sel <= '1';
        IF (last_word='1') THEN
            control_ns <= p_minus_w_f;
        ELSE
            control_ns <= p_minus_w_m;
        END IF;
    WHEN p_minus_w_f =>
        r_addr_1_temp <= p;
        r_addr_2_temp <= swap_addr_w_u;
        w_addr_temp <= swap_addr_w_u;
        write <= '1';
        complement_sel <= '1';
        IF (swap_state_reg='1') THEN
            control_ns <= finished;
        ELSE
            control_ns <= finished_2;
        END IF;
    --output either in u or w which is stored in output_addr_reg
    WHEN finished =>
        z_addr <= u;
        done <= '1';
        load_output_addr <= '1';
        control_ns <= init;
    WHEN finished_2 =>
        z_addr <= w;
        done <= '1';
        load_output_addr <= '1';
        control_ns <= init;
    END CASE;
```

```
END PROCESS control_FSM;
--swap state address mux
swap_addr_w_u <= w WHEN swap_state_reg='0' ELSE
                 u;
swap_addr_d_c <= d WHEN swap_state_reg='0' ELSE
                 c;
-- delta counter implemented as a shift register with sign bit register (scalable overhead)
delta_counter:
PROCESS(clk, reset, subtract_delta, negate_delta, sign_of_delta,
        delta, op)
BEGIN
    IF(reset='1') THEN
        IF  (op='1') THEN             -- division initialization
            delta <= (others => '0');
            sign_of_delta <= '0';
        ELSE                          -- multiplication initialization
            delta <= '1' & zero_vector(PRECISION-2 downto 0);
            sign_of_delta <= '0';
        END IF;
    ELSIF(clk'event AND clk='1') THEN
        IF(negate_delta='1') THEN
            sign_of_delta <= NOT sign_of_delta;
        ELSIF(subtract_delta='1' AND delta=zero_vector) THEN
            sign_of_delta <= '1';
            delta <= zero_vector(PRECISION-1 DOWNTO 1) & '1';
        ELSIF(subtract_delta='1' AND sign_of_delta='0') THEN
            delta <= '0' & delta(PRECISION-1 DOWNTO 1);
        ELSIF(subtract_delta='1' AND sign_of_delta='1') THEN
            delta <= delta(PRECISION-2 DOWNTO 0) & '0';
        END IF;
    END IF;
END PROCESS delta_counter;
-- N-bit sized counter
word_size_delta:
PROCESS(clk, reset, subtract_delta, mult_delta, reset_delta)
BEGIN
    IF(reset='1' OR reset_delta='1') THEN
            mult_delta <= '1' & zero_vector(N-2 downto 0);
    ELSIF(clk'event AND clk='1') THEN
        IF(subtract_delta='1') THEN
            mult_delta <= '0' & mult_delta(N-1 DOWNTO 1);
        END IF;
    END IF;
END PROCESS word_size_delta;
-- counter with terminal count that indicates the last word, given
-- word size (N) and desired precision size (PRECISION).  Side note:  using
-- 'positive' signal type increases the complexity of the counter greatly.
words_counter:
PROCESS (clk, reset, next_word, counter, max_std_logic)
VARIABLE max : integer := PRECISION/N-1; --cycles required including zero
BEGIN
max_std_logic <= conv_std_logic_vector(max,BITS);
    IF (reset='1') THEN
```

```
        counter <= (others=>'0');
    ELSIF (clk'event AND clk='1') THEN
        IF (counter=max_std_logic ) THEN
            counter <= counter-max_std_logic;
        ELSIF (next_word='1') THEN
            counter <= counter+1;
        ELSE
            counter <= counter;
        END IF;
    END IF;
    --signal to jump to the last word
    IF (reset='1') THEN
        last_word <= '0';
    ELSIF (counter=max_std_logic-1) THEN
        last_word <= '1';
    ELSE
        last_word <= '0';
    END IF;
    --final word signal
    IF (reset='1') THEN
        final_word <= '0';
    ELSIF (counter=max_std_logic) THEN
        final_word <= '1';
    ELSE
        final_word <= '0';
    END IF;
END PROCESS words_counter;
-- done signal for multiplication computation (set in stone)
mult_done_signal:
PROCESS (clk, reset, delta, final_word)
BEGIN
    IF (reset='1') THEN
        mult_done <= '0';
    ELSIF (clk'event AND clk='1') THEN
        IF (delta(1)='1' AND final_word='1') THEN  --done signal generated one cycle early
            mult_done <= '1';
        END IF;
    END IF;
END PROCESS mult_done_signal;
-- c address offset for loading in multiplication
-- C is used bit by bit unlike U or W which are used word by word
c_addr_offset:
PROCESS (clk, reset, mult_delta(0), mult_cnt, load_r_addr)
BEGIN
    IF (reset='1') THEN
        mult_cnt <= (OTHERS=>'0');
    ELSIF (clk'event AND clk='1') THEN
        IF (mult_delta(0)='1' AND load_r_addr='1') THEN  --delta(0)=1 is true for word length cycles
            mult_cnt <= mult_cnt + '1';
        END IF;
    END IF;
END PROCESS c_addr_offset;
-- base addr or offset addr mux, read addr one cycle early
```

```
r_addr_1 <= r_addr_1_temp + mult_cnt WHEN load_mult_addr='1' ELSE  --for mult same C used for N cycle
            r_addr_1_temp + counter WHEN load_r_addr='0' ELSE
            r_addr_1_temp;
r_addr_2 <= r_addr_2_temp + counter WHEN load_r_addr='0' ELSE
            r_addr_2_temp;
w_addr <= w_addr_temp + counter WHEN load_w_addr='0' ELSE
          w_addr_temp;
-- multiplication path storage register and division.  modular reduce of upper words get decided by the first word
modulo_reduction_path:
PROCESS (clk, reset, load_modulo_red_sel, next_p_or_zero_lsb)
BEGIN
    IF (reset='1') THEN
        modulo_red_sel_reg <= '0';
    ELSIF (clk'event AND clk='1') THEN
        IF (load_modulo_red_sel='1') THEN
            modulo_red_sel_reg <= next_p_or_zero_lsb;
        END IF;
    END IF;
END PROCESS modulo_reduction_path;
-- for division result could be in W or U depending on swap state
swap_state:
PROCESS (clk, reset, invert_swap, swap_state_reg)
BEGIN
    IF (reset='1') THEN
        swap_state_reg <= '0';
    ELSIF (clk'event AND clk='1') THEN
        IF (invert_swap='1') THEN
            swap_state_reg <= NOT swap_state_reg;
        END IF;
    END IF;
END PROCESS;
-- stores the memory address of the final output
PROCESS (clk, reset, load_output_addr, z_addr)
BEGIN
    IF (reset='1') THEN
        output_addr <= (others=>'0');
    ELSIF (clk'event AND clk='1') THEN
        IF (load_output_addr='1') THEN
            output_addr <= z_addr;
        END IF;
    END IF;
END PROCESS;
END beh;
```

APPENDIX B. C Source Codes for Random Vector Generation

B.1. Modular Division

```
/*  Generates a do file for SCALABLE DIVISION TESTBENCH simulation over GF(p)
 *
 *  Random values are assigned to X (W) and Y (C).
 *  Function 'erand48()' generates 48-bit random value
 *  in range 0.0 - 1.0, which is scaled to the prime.
 *  Expected value and the result of simulation is
 *  compared.  Expected value is generated by this c code
 *  and simulation result is generated by hardware given
 *  inputs provide by do_file.
 *
 *  Example:
 *     gcc scalable_div.c -lm
 *     a.out #_of_test_values >! test.do
 *     vsim testbench -do test.do
 *  Error messages appears on main ModelSim window if any
 *  discrepency exists.  Ignore first error
 *
 *  X/Y mod P == U/C mod D = W or U;  location of result depends on swap state
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PRECISION 11        // Total number of bits (precision)
#define N 11                // Width of the datapath
#define field 1             // 0 -> GF(2^n)  1 -> GF(p)
//#define p 11              // Prime number under test for PRECISION 7 with 3 extra
#define p 137               // for PRECISION 11 with 3 extra bits
//#define p 5333            // for PRECISION 16
//#define p 4099            //2051/832 mod 4099 in GF(2^n) cause problem
//#define p 5325            //GF(2^n) irreducible polynomial x^12 + x^9 + x^5 + x^2 + 1 NOT a PRIME #
//#define p 2000003         //21-bits
void int2bin (int x, int bin[], int bit_num);
int prime (int candidate);
int main (int argc, char *argv[])
{
    int random_x, random_y;     // integer random number
    double random_float;        // floating point random number generated by erand48
    unsigned short seed[3];     // seed array used for random function
    int binary_array[2*N];      // binary representation of the computed answer
    int i, j, l;                // indices for loops
    int u, c, d;                // input values
    int w;                      // output value
    int x, y;                   // copy of u, c for display
    int test_number;            // number of different values to test fetched from command prompt
    int words;                  // number of divided words
    int u_addr, c_addr;         // memory address
```

```
int d_addr, p_addr;         // memory address
int word_msb;               // Most significant bit of particular word
int delta;                  // counter for division swap
int temp_c, temp_u;         // temporary space for swap function
int k;                      // algorithm variable
int BITS;
// One argument required, else program exits
if (argc < 2) {
    printf ("ERROR: Missing arguments.  Must enter # of samples desired. \n");
    printf ("Correct Argument Example: 'a.out 10' \n");
    exit(-1);
}
// Primality test
if (prime(p)==0 && field==1) {
    printf ("ERROR: p is not a Prime number. \n");
    exit(-1);
}
// Computes minimum BITS given PRECISION & N which is CEILING( LOG2 (5* PRECISION/N))
BITS =  log10(5*PRECISION/N) * 3.32192809489 + 1;
printf ("# division testbench.vhd do file for scalable design \n");
printf ("############################################################\n");
printf ("# PRECISION=%d    N=%d    BITS=%d    ", PRECISION, N, BITS);
if (field==1)
    printf ("FIELD = GF(P)\n");
else
    printf ("FIELD = GF(2^N)\n");
printf ("# vsim testbench -do this_file -gprecision=%d -gn=%d -gbits=%d \n", PRECISION, N, BITS);
printf ("############################################################\n");
printf ("\n");
printf ("# CHECK:");
printf ("# for correct simulation, count signal should be incrementing\n");
printf ("# remember to set N and PRECISION to correct values on the testbench.vhd file\n");
printf ("# make sure u_mem and w_mem is defined in dual_port_mem.vhd file in order to compare results\n\n");
printf ("# X/Y mod P == U/C mod D = W or U\n");
printf ("# Predefined memory addresses order:\n");
printf ("# U,W,C,D,P which are defined in control.vhd\n");
printf ("\n");
printf ("view signals\n");
//disable wave viewer to save disk space which is used for debugging
printf ("#view wave -x 0 -height 600 -width 1000\n");
printf ("#add wave /testbench/done\n");
printf ("#add wave -radix unsigned /testbench/count\n");
printf ("#add wave -radix decimal /testbench/expected\n");
printf ("#add wave -radix decimal /testbench/converted\n");
printf ("#add wave /testbench/umdm_module/control_module/control_ps\n");
printf ("#add wave -r /*\n");
test_number = atoi(argv[1]);
words = PRECISION / N;
printf ("\n");
printf ("force -freeze /testbench/reset 1 0,1 0ns\n");
printf ("force -freeze /testbench/clk 1 0, 0 {50 ns} -r 100ns\n");
printf ("force -freeze /testbench/start 0 0\n");
printf ("# 0->GF(2^n); 1->GF(p)\n");
```

```
printf ("force -freeze /testbench/f_sel %d 0\n", field);
printf ("# 0->multiplicaiton; 1->division\n");
printf ("force -freeze /testbench/op 1 0\n");
printf ("force -freeze /testbench/test_number %d 0\n", test_number+2);
printf ("\n");
// Range or number of c to test
for (j=1; j<=test_number; j++) {
    seed[0]=0;
    seed[1]=0;
    seed[2]=j;
    do {
        random_float = erand48(seed);               //produces 48 bit random number in range 0-1
        random_x = (int) (random_float * p) % p;    //scale within range of the prime
        random_float = erand48(seed);
        random_y = (int) (random_float * p) % p;
    } while(random_x==0 || random_y==0);            //repeat if random number equals zero
    u = x = random_x;
    w = delta = 0;
    c = y = random_y;
    d = p;
    //control.vhd defined memory address values
    u_addr = 0;
    c_addr = PRECISION/N*2;
    d_addr = PRECISION/N*3;
    p_addr = PRECISION/N*4;
    printf ("# u / c mod p  -->  %d / %d mod %d \n", u, c, p);
    printf ("force -freeze /testbench/reset 1 1,0 100ns\n");
    printf ("force -freeze /testbench/load_fifo 1 0\n");
    printf ("# u \n");
    word_msb = N;  //resets word msb value
    for (l=0; l<words; l++) {
        printf ("force -freeze /testbench/w_addr_user ");
        int2bin (u_addr, binary_array, BITS);
        for (i=0; i<BITS; i++)
            printf ("%d", binary_array[BITS-1-i]);
        printf(" 0\n");
        printf ("force -freeze /testbench/input ");
        for (i=0; i<N; i++)
            printf ("0");  //zero padding for the initial carry vector
        int2bin (u, binary_array, PRECISION);
        for (i=0; i<N; i++)
            printf ("%d", binary_array[word_msb-1-i]);
        printf(" 0\n");
        printf ("run 100ns\n");
        word_msb = word_msb + N;
        u_addr = u_addr + 1;
    }
    printf ("# c \n");
    word_msb = N;  //resets word msb value
    for (l=0; l<words; l++) {
        printf ("force -freeze /testbench/w_addr_user ");
        int2bin (c_addr, binary_array, BITS);
        for (i=0; i<BITS; i++)
```

```
        printf ("%d", binary_array[BITS-1-i]);
    printf(" 0\n");
    printf ("force -freeze /testbench/input ");
    for (i=0; i<N; i++)
        printf ("0");  //zero padding for the initial carry vector
    int2bin (c, binary_array, PRECISION);
    for (i=0; i<N; i++)
        printf ("%d", binary_array[word_msb-1-i]);
    printf(" 0\n");
    printf ("run 100ns\n");
    word_msb = word_msb + N;
    c_addr = c_addr + 1;
}
printf ("# d \n");
word_msb = N;  //resets word msb value
for (l=0; l<words; l++) {
    printf ("force -freeze /testbench/w_addr_user ");
    int2bin (d_addr, binary_array, BITS);
    for (i=0; i<BITS; i++)
        printf ("%d", binary_array[BITS-1-i]);
    printf(" 0\n");
    printf ("force -freeze /testbench/input ");
    for (i=0; i<N; i++)
        printf ("0");  //zero padding for the initial carry vector
    int2bin (d, binary_array, PRECISION);
    for (i=0; i<N; i++)
        printf ("%d", binary_array[word_msb-1-i]);
    printf(" 0\n");
    printf ("run 100ns\n");
    word_msb = word_msb + N;
    d_addr = d_addr + 1;
}
printf ("# p \n");
word_msb = N;  //resets word msb value
for (l=0; l<words; l++) {
    printf ("force -freeze /testbench/w_addr_user ");
    int2bin (p_addr, binary_array, BITS);
    for (i=0; i<BITS; i++)
        printf ("%d", binary_array[BITS-1-i]);
    printf(" 0\n");
    printf ("force -freeze /testbench/input ");
    for (i=0; i<N; i++)
        printf ("0");  //zero padding for the initial carry vector
    int2bin (p, binary_array, PRECISION);
    for (i=0; i<N; i++)
        printf ("%d", binary_array[word_msb-1-i]);
    printf(" 0\n");
    printf ("run 100ns\n");
    word_msb = word_msb + N;
    p_addr = p_addr + 1;
}
printf ("force -freeze /testbench/load_fifo 0 0\n");
printf ("force -freeze /testbench/start 1 0\n");
```

```
printf ("run 100ns\n");
printf ("force -freeze /testbench/start 0 0\n");
// Division Algorithm
if (field==1) {    //GF(p)
    while (c != 0) {
        if (c % 2 == 0) {
            c = c/2;
            delta = delta - 1;
        }
        else {
            if (delta<0) {
                temp_c = c;
                c = d;
                d = temp_c;
                temp_u = u;
                u = w;
                w = temp_u;
                delta = - delta;
            }
            k = 1;
            if ((c+d) % 4 != 0)
                k = -1;
            else
                delta = delta - 1;

            c = (c + k*d)/2;
            u = (u + k*w);
        }
        u = (u + (abs(u) % 2)*p)/2;
    }
    if (d!=1)
        w = p - w;
    //printf ("c=%d \nu=%d \nd=%d \nw=%d \ndelta=%d \n", c, u, d, w, delta);
}
else {    //GF(2^N)
    while (c != 0) {
        if (c % 2 == 0) {
            c = c/2;
            delta = delta - 1;
        }
        else {
            if (delta<0) {
                temp_c = c;
                c = d;
                d = temp_c;
                temp_u = u;
                u = w;
                w = temp_u;
                delta = - delta;
            }
            k = 1;
            delta = delta - 1;
            c = (c ^ (k*d))/2;
```

```
                    u = (u ^ (k*w));
                }
                u = (u ^ ((abs(u) % 2)*p))/2;
            }
            if (d!=1)
                w = p ^ w;
            //printf ("c=%d \nu=%d \nd=%d \nw=%d \ndelta=%d \n", c, u, d, w, delta);
        }
        printf ("force -freeze /testbench/expected ");
        int2bin (w, binary_array, PRECISION);
        for (i=0; i<PRECISION; i++)
            printf ("%d", binary_array[PRECISION-1-i]);  //expected value
        printf (" 0\n");
        printf ("run %dus \n", words*PRECISION);  ////need better run time estimation
        printf ("#C code Answer = %d         FOR MAPLE COMPARISON:     %d / %d mod %d = %d \n \n", w, x, y, p, w);
    }
return 0;
}
// integer to binary conversion where WIDTH is the # of bits
void int2bin(int x, int bin[], int bit_num) {
    int i;
    // Positive number converstion
    if (x >= 0) {
        for (i=0; i<bit_num; i++) {
            if((x%2) != 0)
                bin[i] = 1;
            else
                bin[i] = 0;
            x = x / 2;
    }   }
    // Negative number conversion
    else {
        x = x + 1;
        for (i=0; i<bit_num; i++) {
            if((x%2) != 0)
                bin[i] = 0;
            else
                bin[i] = 1;
            x = x / 2;
}   }   }
//check if the number is prime
int prime(int candidate) {
    int i;
    int test = 1;
    for (i=2; i<candidate; i++) {
        if (candidate%i==0) {
            test=0;
            break;
    }   }
    return test;
}
```

B.2. Montgomery Muliplication

```
/*  Generates a do file for SCALABLE UMDM TESTBENCH simulation over GF(p)
 *
 *  Random values are assigned to X (W) and Y (C).
 *  Function 'erand48()' generates 48-bit random value
 *  in range 0.0 - 1.0, which is scaled to the prime.
 *  Expected value and the result of simulation is
 *  compared.  Expected value is generated by this c code
 *  and simulation result is generated by hardware given
 *  inputs provide by do_file.
 *
 *  Example:
 *     gcc scalable_mult.c -lm
 *     a.out #_of_test_values >! test.do
 *     vsim testbench -do test.do
 *  Error messages appears on main ModelSim window if any
 *  discrepency exists.  Ignore first error
 *
 *  X*Y*2^N mod P == W*C mod P = U
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PRECISION 4        // Total number of bits (precision)
#define N 4                 // Width of the datapath
#define field 0             // 0 -> GF(2^n)  1 -> GF(p)
#define p 13                // Prime number under test for PRECISION 4
void int2bin (int x, int bin[], int bit_num);
int prime (int candidate);
int main (int argc, char *argv[])
{
    int random_x, random_y;     // integer random number
    double random_float;        // floating point random number generated by erand48
    unsigned short seed[3];     // seed array used for random function
    int binary_array[2*N];      // binary representation of the computed answer
    int i, j, k;                // indices for loops
    int w, c;                   // input values
    int u;                      // output value
    int y;                      // copy of c for display
    int test_number;            // number of different values to test fetched from command prompt
    int words;                  // number of divided words
    int w_addr;                 // Address of w in memory
    int c_addr;                 // Address of c in memory
    int word_msb;               // Most significant bit of particular word
    int BITS;
    // One argument required, else program exits
    if (argc < 2) {
        printf ("ERROR: Missing arguments.  Must enter # of samples desired. \n");
        printf ("Correct Argument Example: 'a.out 10' \n");
        exit(-1);
    }
    // Primality test
```

```
if (prime(p)==0 && field==1) {
    printf ("ERROR: p is not a Prime number. \n");
    exit(-1);
}
// Computes minimum BITS given PRECISION & N which is CEILING( LOG2 (5* PRECISION/N))
BITS =  log10(5*PRECISION/N) * 3.32192809489 + 1;
printf ("#########################################################\n");
printf ("# PRECISION=%d    N=%d    BITS=%d    ", PRECISION, N, BITS);
if (field==1)
    printf ("FIELD = GF(P)\n");
else
    printf ("FIELD = GF(2^N)\n");
printf ("# vsim testbench -do this_file -gprecision=%d -gn=%d -gbits=%d \n", PRECISION, N, BITS);
printf ("#########################################################\n");
printf ("\n");
printf ("# multiplication testbench.vhd do file for scalable unified montgomery multiplication\n");
printf ("# for correct simulation, count signal should be incrementing\n");
printf ("# remember to set N and PRECISION to correct values on the testbench.vhd file\n");
printf ("# w(upper)=w; w(lower)=p\n");
printf ("# u, w, c, d, p  -->  w * c * 2^-n mod p\n");
printf ("\n");
printf ("view signals\n");
//disable wave viewer to save disk space
printf ("#view wave -x 0 -height 600 -width 1000\n");
printf ("#add wave -radix unsigned /count\n");
printf ("#add wave -radix unsigned /testbench/expected\n");
printf ("#add wave -radix unsigned /testbench/converted\n");
printf ("#add wave -r /*\n");
test_number = atoi(argv[1]);
words = PRECISION / N;
printf ("\n");
printf ("force -freeze /testbench/reset 1 0,1 0ns\n");
printf ("force -freeze /testbench/clk 1 0, 0 {50 ns} -r 100ns\n");
printf ("force -freeze /testbench/start 0 0\n");
printf ("# 0->GF(2^n); 1->GF(p)\n");
printf ("force -freeze /testbench/f_sel %d 0\n", field);
printf ("# 0->multiplicaiton; 1->division\n");
printf ("force -freeze /testbench/op 0 0\n");
printf ("force -freeze /testbench/test_number %d 0\n", test_number+2);
printf ("\n");
// Range or number of c to test
for (j=1; j<=test_number; j++) {
    seed[0]=0;
    seed[1]=0;
    seed[2]=j;
    do {
        random_float = erand48(seed);               //produces 48 bit random number in range 0-1
        random_x = (int) (random_float * p) % p;    //scale within range of the prime
        random_float = erand48(seed);
        random_y = (int) (random_float * p) % p;
    } while(random_x==0 || random_y==0);            //repeat if random number equals zero
    u = 0;
    w = 4;//random_x;
```

```
c = y = 4;//random_y;
//control.vhd defined memory address values
w_addr = PRECISION/N;
c_addr = PRECISION/N*2;
printf ("# w * c * 2^-n mod p  -->  %d * %d mod %d \n", w, c, p);
printf ("force -freeze /testbench/reset 1 1,0 100ns\n");
printf ("force -freeze /testbench/load_fifo 1 0\n");
printf ("# w,p address defined in control.vhd\n");
word_msb = N;
for (k=0; k<words; k++) {
    printf ("force -freeze /testbench/w_addr_user ");
    int2bin (w_addr, binary_array, BITS);
    for (i=0; i<BITS; i++)
        printf ("%d", binary_array[BITS-1-i]);
    printf(" 0\n");
    printf ("force -freeze /testbench/input ");
    int2bin (w, binary_array, PRECISION);
    for (i=0; i<N; i++)
        printf ("%d", binary_array[word_msb-1-i]);
    int2bin (p, binary_array, PRECISION);
    for (i=0; i<N; i++)
        printf ("%d", binary_array[word_msb-1-i]);
    printf (" 0\n");
    printf ("run 100ns\n");
    word_msb = word_msb + N;
    w_addr = w_addr + 1;
}
printf ("# c address\n");
word_msb = N;  //resets word msb value
for (k=0; k<words; k++) {
    printf ("force -freeze /testbench/w_addr_user ");
    int2bin (c_addr, binary_array, BITS);
    for (i=0; i<BITS; i++)
        printf ("%d", binary_array[BITS-1-i]);
    printf(" 0\n");
    printf ("force -freeze /testbench/input ");
    for (i=0; i<N; i++)
        printf ("0");  //zero padding for the initial carry vector
    int2bin (c, binary_array, PRECISION);
    for (i=0; i<N; i++)
        printf ("%d", binary_array[word_msb-1-i]);
    printf(" 0\n");
    printf ("run 100ns\n");
    word_msb = word_msb + N;
    c_addr = c_addr + 1;
}
printf ("force -freeze /testbench/load_fifo 0 0\n");
printf ("force -freeze /testbench/start 1 0\n");
printf ("run 100ns\n");
printf ("force -freeze /testbench/start 0 0\n");
// Montgomery Multiplication Algorithm
//GF(p) field
if(field==1) {
```

```
            for (i=0; i<PRECISION; i++) {
                if (c % 2 == 0)
                    c = c/2;
                else {
                    c = c/2;
                    u = u + w;
                }
                u = (u + (abs(u) % 2)*p)/2;
        }   }
        //GF(2^n) field
        else {
            for (i=0; i<PRECISION; i++) {
                if (c % 2 == 0)
                    c = c/2;
                else {
                    c = c/2;
                    u = u ^ w;
                }
                u = (u ^ ((abs(u) % 2)*p))/2;
        }   }
        printf ("force -freeze /testbench/expected ");
        int2bin (u, binary_array, PRECISION);
        for (i=0; i<PRECISION; i++)
            printf ("%d", binary_array[PRECISION-1-i]);  //expected value
        printf (" 0\n");
        printf ("run %d00ns \n", PRECISION*words + 20 );
        printf ("#C code Answer = %d              MAPLE COMPARISON:     %d * %d mod %d = %d \n \n", u, w, y, p, u);
    }
return 0;
}
```

APPENDIX C. PERL Source Codes for Synthesis

C.1. Register File

```
#!/usr/local/bin/perl
# array containing desired n and bits to synthesize (N, BITS, N, BITS, ......)
@bits = qw (016 016 3
            032 016 4    032 032 3
            064 016 5    064 032 4    064 064 3
            128 016 6    128 032 5    128 064 4    128 128 3
            256 016 7    256 032 6    256 064 5    256 128 4    256 256 3
            512 016 8    512 032 7    512 064 6    512 128 5    512 256 4    512 512 3 );
# user defined constants
$show_result = "show_mem_results";
# small script file which displays results
open (RESULT_FILE, ">$show_result") || die "Can't create $show_result file";
print RESULT_FILE "#script file to list area & delay results\n";
print RESULT_FILE "pwd >! ~/umdm/vhdl/scalable/synth/results/current_dir \n";
print RESULT_FILE "cd ~/umdm/vhdl/scalable/synth/results \n";
print RESULT_FILE "grep -i 'number of gates' * \n";
print RESULT_FILE "echo \"----------------------------------------------------------------\" \n";
print RESULT_FILE "grep -m 1 'data arrival time' * \n";
print RESULT_FILE "cd `cat current_dir` \n";
print RESULT_FILE "rm -f ~/umdm/vhdl/scalable/synth/results/current_dir \n";
close (RESULT_FILE) || die "Couldn't close $show_result file";
chmod(0700,"$show_result");
# writes a master executable script file for the shell
open (SCRIPT_FILE, ">master_mem");
print SCRIPT_FILE "#dual_port_mem.vhd synthesis file\n";
print SCRIPT_FILE "#starts elsyn then synthesizes generated script files with different N and BITS.\n\n";
for($i=0; $i<=$#bits; $i=$i+3) {
    print SCRIPT_FILE "elsyn <<!\n";
    print SCRIPT_FILE "\tsource scripts/dual_port_mem$bits[$i]_$bits[$i+1]_$bits[$i+2].syn\n";
    print SCRIPT_FILE "\texit\n!\n";
}
print SCRIPT_FILE "grep -m 1 'data arrival time' results/*\n";
close (SCRIPT_FILE);
chmod(0700,"master_mem");
    #another way to chage permission.  '|' specifies the shell command
    #open STDOUT, "|chmod 700 master_mem";
    #close STDOUT;
#creates scripts/ directory if not present
mkdir ("scripts", 0700);
# writes synthesis script files
for($i=0; $i<=$#bits; $i=$i+3) {
    open (NEW_FILE, ">scripts/dual_port_mem$bits[$i]_$bits[$i+1]_$bits[$i+2].syn") ||
                                                die "Can't create *.syn file in script/ directory: $!";
    open (FILE, "ref_mem.syn") || die "Can't open ref_mem.syn: $!";  #opens reference.syn for reference
    while (<FILE>) {
        #isolates three lines to be modified
        if (/^elaborate/) {
```

```
            print NEW_FILE "elaborate dual_port_mem -architecture beh -work work -generics
                            {PRECISION=$bits[$i] N=$bits[$i+1] BITS=$bits[$i+2]}\n";
        }
        elsif (/^report_area/) {
            print NEW_FILE "report_area  -cell             ./results/mem_area$bits[$i]_$bits[$i+1]_$bits[$i+2].txt\n";
        }
        elsif (/^report_delay/) {
            print NEW_FILE "report_delay -show_nets -num 1 ./results/mem_delay$bits[$i]_$bits[$i+1]_$bits[$i+2].txt\n";
        }
        else {
            print NEW_FILE "$_";
    }   }
    close(FILE) || die "Couldn't close reference.syn";
    close(NEW_FILE);
}
print "Check if SYNTHESIZE is set to 2.\n";
print "Synthesis script files are written to scripts/ directory.\n";
print "DONE.  Type 'master_mem' to synthesize using elsyn\n";
print "After synthesis type '$show_result' to view area & delay results\n";
# sets SYNTHESIZE to 2 in modified_mem.vhd then moves the file to dual_port_mem.vhd
open (NEW_MEM_FILE, '>../backup/modified_mem.vhd') || die "Can't create modified_mem.vhd file";
open (MEM_FILE, '../backup/dual_port_mem.vhd') || die "Can't open dual_port_mem.vhd file: $!";
while (<MEM_FILE>) {
    if (/^CONSTANT/) {
        s/1/2/;
        print NEW_MEM_FILE "$_";
    }
    else {
        print NEW_MEM_FILE "$_";
}   }
close (MEM_FILE) || die "Couldn't close dual_port_mem.vhd file";
close (NEW_MEM_FILE) || die "Couldn't close modified_mem.vhd file";
open STDOUT, "|rm -rf ../backup/dual_port_mem.vhd; mv ../backup/modified_mem.vhd ../backup/dual_port_mem.vhd";
```

C.2. Datapath and Control

```
#!/usr/local/bin/perl
# array containing desired precision, n, and bits to synthesize (PRECISION, N, BITS, PRECISION, N, BITS, ......)
@bits = qw (016 016 3
            032 016 4    032 032 3
            064 016 5    064 032 4    064 064 3
            128 016 6    128 032 5    128 064 4    128 128 3
            256 016 7    256 032 6    256 064 5    256 128 4    256 256 3
            512 016 8    512 032 7    512 064 6    512 128 5    512 256 4    512 512 3 );
# writes a master executable script file for the shell
open (SCRIPT_FILE, ">master_umdm");
print SCRIPT_FILE "#starts elsyn then synthesizes generated script files with different PRECISION, N, and BITS.\n\n";
for($i=0; $i<=$#bits; $i=$i+3) {
    print SCRIPT_FILE "elsyn <<!\n";
    print SCRIPT_FILE "\tsource scripts/umdm$bits[$i]_$bits[$i+1]_$bits[$i+2].syn\n";
    print SCRIPT_FILE "\texit\n!\n";
```

```
}
print SCRIPT_FILE "grep -m 1 'data arrival time' results/*\n";
close (SCRIPT_FILE);
chmod(0700,"master_umdm");
    #another way to chage permission.  '|' specifies the shell command
    #open STDOUT, "|chmod 700 master_umdm";
    #close STDOUT;
#creates scripts/ directory if not present
mkdir ("scripts", 0700);
# writes synthesis script files
for($i=0; $i<=$#bits; $i=$i+3) {
    open (NEW_FILE, ">scripts/umdm$bits[$i]_$bits[$i+1]_$bits[$i+2].syn") || die "Can't open reference.syn: $!";
    open (FILE, "reference.syn") || die "Can't open reference.syn: $!";  #opens reference.syn for reference
    while (<FILE>) {
        #isolates three lines to be modified
        if (/^elaborate/) {
            print NEW_FILE "elaborate umdm -architecture beh -work work -generics {PRECISION=$bits[$i]
                                                                                N=$bits[$i+1] BITS=$bits[$i+2]}\n";
        }
        elsif (/^report_area/) {
            print NEW_FILE "report_area  -cell              ./results/area$bits[$i]_$bits[$i+1]_$bits[$i+2].txt\n";
        }
        elsif (/^report_delay/) {
            print NEW_FILE "report_delay -show_nets -num 1  ./results/delay$bits[$i]_$bits[$i+1]_$bits[$i+2].txt\n";
        }
        else {
            print NEW_FILE "$_";
    }   }
    close(FILE) || die "Couldn't close reference.syn";
    close(NEW_FILE);
}
print "Synthesis script files are written into scripts/ directory.\n";
print "DONE.  Type 'master_umdm' to synthesize using elsyn\n";
```