AN ABSTRACT OF THE THESIS OF

Akekalak Chaitheerayanon for the degree of Master of Science in Electrical & Computer Engineering presented on December 16, 2003.

Title: Investigating New Design Alternatives for a Radix-2 Modular Multiplier Kernel and I/O Subsystem..

Abstract approved: _____

Alexandre Ferreira Tenca

A main arithmetic operation for cryptographic systems is modular exponentiation. Exponentiation is computed by a long sequence of modular multiplications. Modular multiplication can be implemented in a general-purpose processor or a dedicated hardware, but dedicated hardware tends to be faster than a processor. Modular multiplication is a time-consuming operation, and therefore it requires a fast and efficient algorithm that can be suitably implemented in hardware. Montgomery multiplication algorithm is one of these efficient algorithms.

There are several designs that are implemented based on the Montgomery algorithm. Most of them are fixed-precision implementations which means that the system can not perform the multiplication if the operand size is larger than the precision of the system datapath. Its lack of flexibility has led to a new design that can perform the multiplication for operands of any size — a scalable architecture.

The hardware implementation of the scalable Montgomery Multiplier (MM) is composed of the kernel and the I/O interface. The main function of the I/O is to interact with both the host system and the kernel. Multiplication operands are loaded from software running on the host system and then stored inside the I/O. These operands are transferred to the kernel when they are needed during

the computation. Moreover, the temporary results from the kernel are also stored inside the I/O.

All major goals of this thesis work involve the investigation of design alternatives for the MM architecture. The first goal is to investigate an alternate design for the kernel. The second one is to develop a new design for the I/O subsystem. As a final goal, the generated VHDL code must be ported to a Field Programmable Gate Array (FPGA). That will take advantages of the Xilinx technology, showing the flexibility of the scalable MM design in a FPGA chip.

Investigating New Design Alternatives for a Radix-2 Modular Multiplier Kernel and I/O Subsystem.

by

Akekalak Chaitheerayanon

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 16, 2003
Commencement June 2004

Master of Science thesis of Akekalak Chaitheerayanon presented on December 16, 2003

APPROVED:

Major Professor, representing Electrical & Computer Engineering

---

Director of the School of Electrical Engineering and 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.

Akekalak Chaitheerayanon, Author

ACKNOWLEDGMENT

First of all, my family, father, mother, brother, and sisters are always with me when I needed them the most. Their encouragement helped me to continue study and finished this work.

This thesis work would not be completed if there was no support from a number of people, especially Dr. Alexandre F. Tenca, my advisor, who has been considerably helpful. My first class at Oregon State University was his, and he gave me a chance to do a research with him since. All his advice and comments are so invaluable. It has been good experiences working with him.

Last but not least, I would like to thank to all Thai friends here at Oregon State University, especially Suchittra Chatpimolkul who has been more than a friend to me since we met.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF APPENDIX FIGURES

INVESTIGATING NEW DESIGN ALTERNATIVES FOR A RADIX-2 MODULAR MULTIPLIER KERNEL AND I/O SUBSYSTEM.

1. INTRODUCTION

As an electronic communication becomes more and more popular, people need a secure channel when communicating among themselves. Thus, the issue of security has become more attractive, and several approaches have been developed. RSA [1] is one of the most reliable cryptographic systems found inside many systems. A main arithmetic operation for RSA cryptography is modular exponentiation. This type of exponentiation can be accomplished by performing a long sequence of modular multiplications. Modular multiplication is a very important operation in other cryptographic systems as well.

Modular multiplication can be implemented in a general-purpose processor or a dedicated hardware. Both implementations have advantages and disadvantages. Dedicated hardware tends to be faster than the general-purpose processor. One way to illustrate this point is that designers can optimize their designs to have as many arithmetic units as possible under the area constraint and the satisfaction of the performance. However, the general-purpose processor has a limitation to perform arithmetic operations. On the other hand, the software running on the general-purpose processor has more flexibility when the design or implementation needs to be modified. Also, when changes are needed, modification to the dedicated hardware can be costly.

Modular multiplication is a time-consuming operation especially in an effective and reliable RSA which uses operand sizes in a range of 512 to 2048 bits. It needs

a fast and efficient algorithm that also can be suitably implemented in hardware. The Montgomery multiplication algorithm [2] is very efficient. It replaces a regular division operation that is needed in modular multiplication by a series of shift operations and additions. These shift and addition operations are fast and easy to implement in hardware. As the multiplication is executed, the algorithm uses right shift operations (division by 2) to reduce the result and keep it bounded.

There are several implementation approaches of the Montgomery algorithm. Most of them are fixed-precision implementations. This means the system cannot perform the multiplication if the operand size is larger than the precision of the system datapath. Its lack of flexibility has led to a new design that can perform the multiplication for operands of any size.

Tenca and Koç [3] have developed a modified version of the Montgomery algorithm. This algorithm is called *a Multiple-Word Radix-2 Montgomery Multiplication Algorithm (MWR2MM)*. The algorithm has an advantage over others because it works with a group of bits (words) of the operands instead of handling whole operands at once. Operations on words are used to reach any required precision. This algorithm leads to a scalable design. This scalable architecture is also flexible in hardware implementation because it can be reconfigured to fit in a chip that has a limited area. Thus it is an efficient hardware solution for modular multiplication.

1.1. Montgomery Multiplication Algorithm

The Montgomery multiplication algorithm performs modular multiplication of two given numbers, A and B; they both are n-bit long. The operation of the Montgomery multiplication algorithm can be written as $\mathrm{MM}(A, B)$. As a result, it gives $(ABr^{-1} \bmod M)$, where $r = 2^n$; and M is an integer in the range of

$2^{n-1} < M < 2^n$. r and M have to be selected such that they are relatively prime to each other. Since $r = 2^n$ is an even integer, M can be selected as any odd integers. Normally, M is selected as a prime, except for 2, or a product of two primes. This is enough to satisfy the relatively prime condition.

The Montgomery multiplication algorithm replaces regular division by performing only shift operations and additions. All values are residues in modulo M. The Montgomery image of A is $\overline{A} = (Ar \bmod M)$. $\overline{A}$ can be computed by performing $\mathrm{MM}(A, r^2)$. On the other hand, the transformation back from the image to the integer domain is $A = \mathrm{MM}(\overline{A}, 1) = (Arr^{-1} \bmod M)$.

If C is defined as the outcome from modular multiplication of two integers A and B, which means $C = (AB \bmod M)$, then Montgomery multiplication of the images of A and B is the image of C. That is:

$$\begin{aligned} \overline{C} &= \mathrm{MM}(\overline{A}, \overline{B}) \\ &= \overline{A}\,\overline{B}r^{-1} \pmod M \\ &= ArBrr^{-1} \pmod M \\ &= ABr \pmod M \\ &= Cr \pmod M \end{aligned}$$

Figure 1.1 shows a radix-2 Montgomery algorithm that is stated in [3]. It is suitable for both software and hardware implementations.

The step described in line 8 of the algorithm is called *final reduction*. We need this step because the value of P after the loop is in a range $[0, 2M-1]$. Thus, the conversion of the result back to the range $[0, M-1]$ is done by subtracting M out of P when $P \geq M$.

```
1        P = 0
2        for i = 0 to n − 1
3            if (P + a_i B) is even
4                then P := (P + a_i B)/2
5                else P := (P + a_i B + M)/2
6            end if
7        end for
8        if P ≥ M then P := P − M
9        end if
```

FIGURE 1.1: Radix-2 Montgomery Multiplication Algorithm

1.2. Motivation

The hardware implementation of the scalable Montgomery Multiplier (MM) proposed in [3] is composed of the kernel and the I/O interface. The motivation for this thesis work is to investigate design alternatives for the MM architecture. The first goal is to investigate the alternate design for the kernel. The second one is to develop a new design for the I/O subsystem which replaces an old version of the subsystem designed by the research group at Oregon State University. The old version imposed some constraints to scalability and portability. As a final goal, the code is ported to a Field Programmable Gate Array (FPGA) taking advantages of the Xilinx technology, and showing the flexibility of the scalable MM design in a small FPGA chip.

1.2.1. Radix-2 Architectural Approaches for Scalability

We have investigated some previous hardware designs that are implemented based upon the Montgomery multiplication algorithm and tried to find alternative designs for the kernel in [3].

The solution that could more closely compete with the design in [3] is a systolic architecture called *Array-C* [4]. A hardware implementation realizes the Montgomery algorithm presented in [5]. The algorithm is slightly different from the one in Figure 1.1. The architecture is a one-dimensional systolic array that works with a full-precision of the operands. To be comparable to the design in [3], we investigated the option of applying a partitioning and mapping algorithms for fixed-size systolic arrays as proposed in [6]. The partitioning method allows the system to perform the operation on chunks (words) of the input operands allowing it to work with operand sizes that are larger than the size of the system data path.

1.2.2. I/O Subsystem

The main function of the I/O is to interact with both the host system and the kernel. The operands are loaded from a software running on the host system and then stored inside the I/O. These operands are transferred to the kernel when they are needed. Moreover, the temporary results from the kernel are also stored inside the I/O.

The first version of the I/O subsystem was targeted to be used with the radix-8 kernel [7]. Since there is an on-going investigation of other versions of the kernel that operate in different radices, the best way is to design an I/O block that is reconfigurable. That means, it will be able to work with various kernel

configurations, e.g. radices, word sizes, and number of words of operands. This is one of the goals in this work.

In the first version of the I/O subsystem, FIFOs used in operand registers were implemented by using flip-flops. Some of these operands are reused during the computation. To make the information inside the FIFOs reusable, the first version uses a data structure that does not destroy the information when data is read from the FIFO. When the information is requested again, the pointer is controlled to point back to the location of that data; this method is call *wrapping*. As a consequence, FIFO pointers are moved back and forth during the computation. It is very unconventional. Therefore, in the new design, a FIFO with rotation capability is used, instead of the wrapping idea. An advantage of the new version is that a FIFO is designed to be easily modified for different storage spaces.

Since the design of the I/O subsystem has to support the kernel in different configurations, reconfigurable devices are of interest to us. Therefore, the target technology for this design is Field Programmable Gate Array (FPGA).

1.3. Organization of this Thesis

We describe the investigation of an alternative design for the Montgomery multiplier kernel in chapter 2. In chapter 3, we describe a flexible design of the I/O subsystem to be used in the scalable Montgomery multiplier system. Next, in chapter 4, the experimental results are discussed. Lastly, we conclude this work and give suggestions for future work.

2. ALTERNATE DESIGN FOR RADIX-2 MONTGOMERY MULTIPLIER KERNEL.

In this chapter, the systolic array for Montgomery algorithm proposed in [4] is investigated. This Montgomery multiplier design is considered to be a competitor to the scalable architecture presented in [3], and both design alternatives are compared to each other.

Fundamentally, a systolic array architecture is composed of many processing elements (PEs). Normally, all PEs are structured in the same way. The PEs are connected to one another to form a linear array or a tree [8]. The word *systolic* in this context means that the data is regularly pumped out of the output of one PE to the input of the next PE. The data are transferred and used in further PEs in the array.

There are many advantages when hardware is implemented by a systolic architecture [8]. For instance, it improves the throughput of the whole system since once I/O is accessed, the data can be repeatedly used by PEs in the array. Also, PEs used in the systolic architecture are modular elements; therefore, the system has regularity in data and control flows, making it very appropriate for VLSI designs.

2.1. Systolic Array for Montgomery Algorithm

As described in the last chapter, Montgomery multiplication of two input operands A and B can be expressed as

$$\mathrm{MM}(A, B) = C = ABr^{-1} \bmod M$$

where A and B are two given integers, M is an odd number in the range of $2^{n-1} < M < 2^n$, and $r = 2^n$.

Operands A, B, M, and C are divided into groups of bits (words). A bit vector A is divided into groups of v bits. The number of v-bit words for A is d. This means, if A is n bits long, then $n = v \cdot d$. The radix of the multiplier is a function of v as $V = 2^v$. For other operands, they are divided into groups of w bits *(word size)*, and there are e words for each n-bit operand. The number representation for operands can be shown below.

$$\begin{aligned} A &= (A^{(d-1)}, \ldots, A^{(1)}, A^{(0)}), \\ B &= (B^{(e-1)}, \ldots, B^{(1)}, B^{(0)}), \\ M &= (M^{(e-1)}, \ldots, M^{(1)}, M^{(0)}), \\ C &= (C^{(e-1)}, \ldots, C^{(1)}, C^{(0)}) \end{aligned}$$

The systolic algorithm presented by Dussé and Kaliski in [5] is considered. It is also used in [4]. The full precision version of the algorithm is:.

$$C[i] = (C[i-1] + A^{(i)} \cdot B \cdot V + Q[i-1] \cdot M)/V \tag{2.1}$$

where $Q[i-1] = ((C[i-1] \bmod V) \cdot M'^{(0)}) \bmod V$

$C[-1] = 0$

$M'^{(0)} = -(M^{(0)})^{-1} \bmod V = -(M \bmod V)^{-1} \bmod V$

To complete the operation, this algorithm is performed for i in the range of $[0, d]$, which means the expression shown above performs for $d + 1$ times.

From the generic algorithm shown in Eq. (2.1), let us consider its use in radix 2. Literally, it means A is divided into groups of 1 bit for each word, and this gives $v = 1$, and consequently $V = 2$. Hence, the radix-2 algorithm is:

$$C[i] = (C[i-1] + 2 \cdot a_i \cdot B + Q[i-1] \cdot M)/2 \tag{2.2}$$

where $Q[i-1] = c_0[i-1] = C[i-1] \bmod 2$

$C[-1] = 0$

The expression in Eq. (2.2) can be written into an algorithm called *Algorithm A* as follows:

Algorithm A

1 $C[-1] = 0, Q[-1] = 0$

2 for $i = 0$ to n

3 $\quad C[i] = (C[i-1] + 2 \cdot a_i \cdot B + Q[i-1] \cdot M)/2$

4 $\quad Q[i] = C[i] \bmod 2$

5 end for

6 if $C[n] \geq M$ then $C := C[n] - M$

7 end if

FIGURE 2.1: Radix-2 Montgomery Algorithm for Systolic Array of Eq. 2.2

In this case, operand A is represented by a n-bit vector $a_{n-1}, \ldots, a_1, a_0$. In radix 2, $M'^{(0)}$ can be eliminated from the algorithm because its value is always 1. Since M is always odd, the least significant bit (LSB) of M which is m_0 will always be 1. As a result, the modular inverse of 1 when the modulus is 2 is also 1.

Note that when A is an n-bit precision number, the expression in Eq. (2.2) has to be executed $n + 1$ times to complete the operation.

2.1.1. Montgomery Multiplication Algorithm Comparison

In this section, two Montgomery multiplication algorithms are compared. The first algorithm is the one shown in Figure 2.2 (also shown in the previous chapter), which is referred as *Algorithm B*. The second algorithm (Figure 2.1) is *Algorithm A* presented in the prior section.

Algorithm B

```
1        P = 0
2        for i = 0 to n − 1
3            if (P + a_i B) is even
4                then P := (P + a_i B)/2
5                else P := (P + a_i B + M)/2
6            end if
7        end for
8        if P ≥ M then P := P − M
9        end if
```

FIGURE 2.2: Radix-2 Montgomery Multiplication Algorithm

Both algorithms look very similar to each other. In the implementation point of view, both algorithms have about the same level of complexity. *Algorithm B* needs an addition and a division by 2, which is easily accomplished by using a right-shift operation. On the other hand, *Algorithm A* has an addition, a multiplication by 2, and a division by 2. The multiplication and division by 2 can be implemented by a left-shift and right-shift operation, respectively.

Algorithm A takes one iteration longer than *Algorithm B* does. However, n is usually much larger than 1 ($n >> 1$); therefore, the execution time will not be much different between the two algorithms ($n + 1 \approx n$ when $n >> 1$).

2.1.2. Examples of Montgomery Multiplication Algorithms

The following examples show that both algorithms give the same result, but *Algorithm A* takes one more iteration to finish the computation. The example shows the multiplication of $A = 011$ and $B = 101$ where $M = 111$. Moreover, let $v = w = 1$; therefore, $d = e = 3$. Note that for *Algorithm A*, $C[i]$ is replaced with $P[i]$ for comparison purposes.

- The following steps perform the multiplication of the Algorithm A

1	$i = 0;$	$P[0]$	$= (P[-1] + 2a_0B + Q[-1]M)/2$	
2			$= (0 + 1010 + 0)/2 = 10\underline{1};$	$Q[0] = 1$
3	$i = 1;$	$P[1]$	$= (P[0] + 2a_1B + Q[0]M)/2$	
4			$= (101 + 1010 + 111)/2 = 101\underline{1};$	$Q[1] = 1$
5	$i = 2;$	$P[2]$	$= (P[1] + 2a_2B + Q[1]M)/2$	
6			$= (1011 + 0 + 111)/2 = 100\underline{1};$	$Q[2] = 1$
7	$i = 3;$	$P[3]$	$= (P[2] + 2 \cdot 0 \cdot B + Q[2]M)/2$	
8			$= (1001 + 0 + 111)/2 = 1000$	
9	test: $P[3] \geq M$ is true: so, $P = P[3] - M = 1000 - 111 = \underline{\underline{001}}$			

  The underlined least significant bit indicates the value of $Q[i]$ of each iteration.

- The following steps perform the multiplication of the Algorithm B:

  1 $P = 0$

  2 $i = 0$; test: $P + a_0 B = 0 + 101$ is odd

  3 $P = (P + a_0 B + M)/2 = (0 + 101 + 111)/2 = 110$

  4 $i = 1$; test: $P + a_1 B = 110 + 101$ is odd

  5 $P = (P + a_1 B + M)/2 = (110 + 101 + 111)/2 = 1001$

  6 $i = 2$; test: $P + a_2 B = 1001 + 0$ is odd

  7 $P = (P + a_2 B + M)/2 = (1001 + 0 + 111)/2 = 1000$

  8 test: $P \geq M$ is true: so, $P = P - M = 1000 - 111 = \underline{\underline{001}}$

2.2. Hardware Implementation for Systolic Array Algorithm

As proposed in [4], the implementation of *Algorithm A* into a systolic array is done by connecting several processing elements (PEs) in a linear array, and then performing the operation of Eq. (2.2). Figure 2.3 shows the data dependence graph for the systolic architecture. A similar figure is also shown in [4].

Figure 2.3(b) represents the input-output relation. Each circle represents one PE. The value of i represents the position where the i-th PE (PE $\#i$) is located, and the value of j is the time domain (clock). The data of each cell, which traverse along the i axis, are b_j and m_j. And, the data of each cell traversing along the j axis are a_i and $Q[i-1]$. Each PE performs the operation $Z[i][j]$ which is shown in Figure 2.3(a). We take the lower v bits of $Z[i][j]$ to be $P[i][j]$. Then, the rest of the bits of $Z[i][j]$ becomes $Carry[i][j]$.

The radix-2 bit-level algorithm for the data dependence graph is shown in Figure 2.4. Although the algorithm is claimed to function properly for any value of w, we only succeeded in working with $w = 1$.

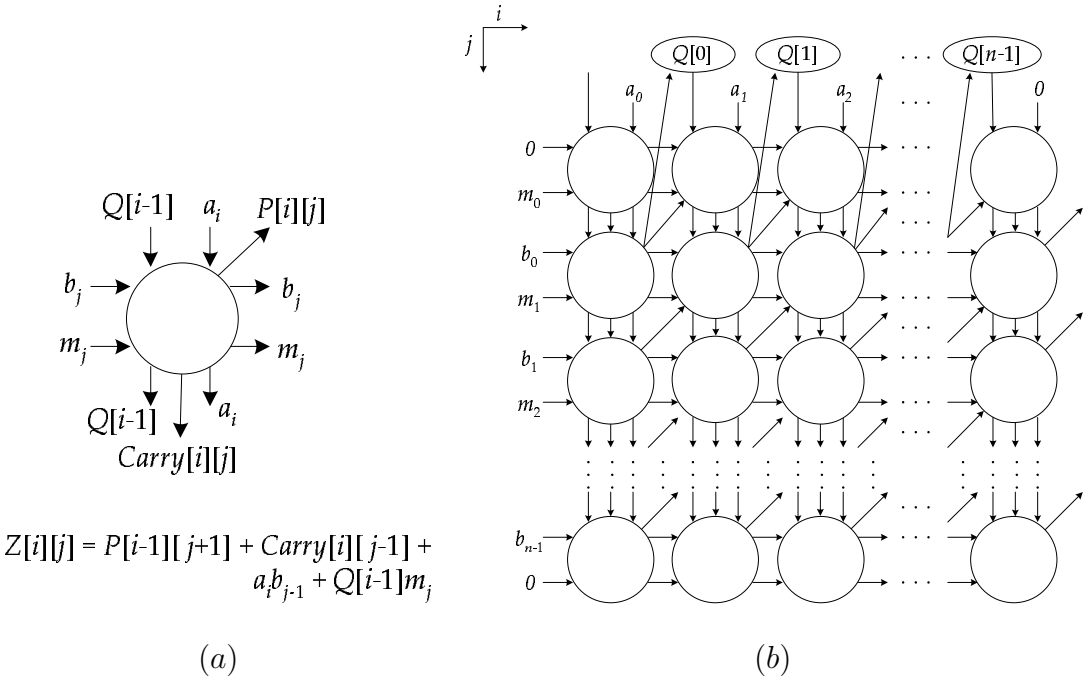


FIGURE 2.3: Data dependence graph for systolic array algorithm

Algorithm C

$P[-1] = 0, b_{-1} = 0$

FOR $i = 0$ TO n

$Q[i-1] = P[i-1][1]$

FOR $j = 0$ TO n

$$\text{PE} \begin{cases} Z[i][j] &= P[i-1][j+1] + Carry[i][j-1] + \\ & \quad a_i \cdot b_{j-1} + Q[i-1] \cdot m_j \\ P[i][j] &= (Z[i][j] \bmod 2) \\ Carry[i][j] &= Z[i][j]/2 \end{cases}$$

NEXT

NEXT

FIGURE 2.4: Radix-2 bit-level Algorithm for Systolic Array

In this case, where $v = w = 1$, the length of Z is 3 bits. The intermediate result bit P is the LSB of Z, and the upper two bits of Z are assigned to $Carry$.

A block diagram of the systolic array is shown in Figure 2.5, and each PE used in the array is constructed as shown in Figure 2.6.

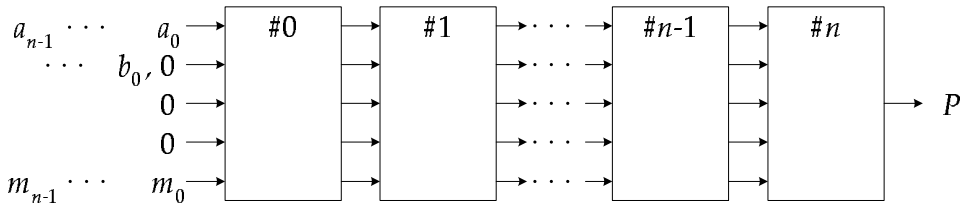


FIGURE 2.5: A systolic array structure

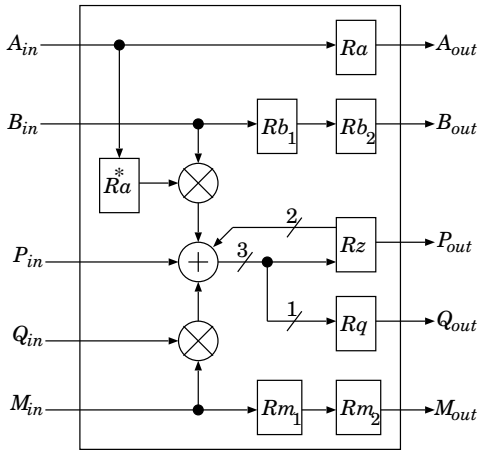


FIGURE 2.6: Processing element for a systolic array

The systolic array is composed of n+1 PEs. The $A_{out}, B_{out}, P_{out}, Q_{out}$, and M_{out} pins of one PE are connected to the corresponding $A_{in}, B_{in}, P_{in}, Q_{in}$, and M_{in} pins of the next PE, respectively. On the first PE (PE #0), for each clock cycle,

A_{in} is fed by a bit a_i of operand A. Similarly, B_{in} and M_{in} are fed by b_i and m_i, respectively. However, due to the term $2 \cdot a_i \cdot B$, operand B is shifted one bit to the left. Thus, a_i, b_i, and m_i are synchronously fed to the first PE (the least significant bit is taken first), but b_i is relatively inputted one clock cycle after a_i and m_i. For instance, at the time we insert a_2 and m_2 to the PE #0, b_1 is taken to the B_{in}. For P_{in} and Q_{in} of the PE #0, we insert a zero bit to both pins as their initial values.

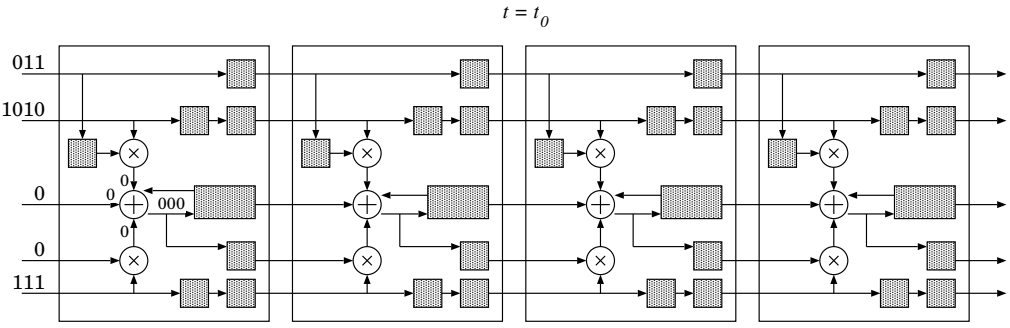
The PE has a simple structure; it mainly consists of registers and arithmetic elements. In Figure 2.6, all registers are one-bit registers except for Rz, which is a 3-bit register. Ra^* is used to store a value of a_i where i is the corresponding PE position. That means Ra^* of the PE #0 stores a_0 and will not change. Similarly, the register Rq stores a value of $Q[i]$ where i is the PE position, and it keeps the same value until the operation is finished. The symbol $\otimes$ is a bit-by-bit multiplier, which can be realized by an AND gate. There are two multipliers of this type in the PE. We call the first one on the top *mul1* and the other one *mul2*. The symbol $\oplus$ represents a 4-input adder that takes operands from the results of both multipliers, the intermediate result from the previous PE, and the carry. We will refer to this adder as *add*. The four input terminals of *add* are *in1*, *in2*, *in3*, and *in4*.

During the operation, operand A is delayed by one clock cycle, and B and M are delayed by two clock cycles. The delay on B is enough for a_i to be set and stored in Ra^* just in time when b_0 arrives at the PE $\#i$ to execute the multiplication. Then, the result of the adder has three bits and is registered in Rz. We take the least significant bit of Rz as P and insert it to the next PE. The rest of the bits of Rz are fed back to the adder of the same PE as a carry of last addition. $Q[i]$ is set to the value of the first computed P of that PE. The value of P, which comes out of the last PE in the array, is the final result for this Montgomery multiplier.

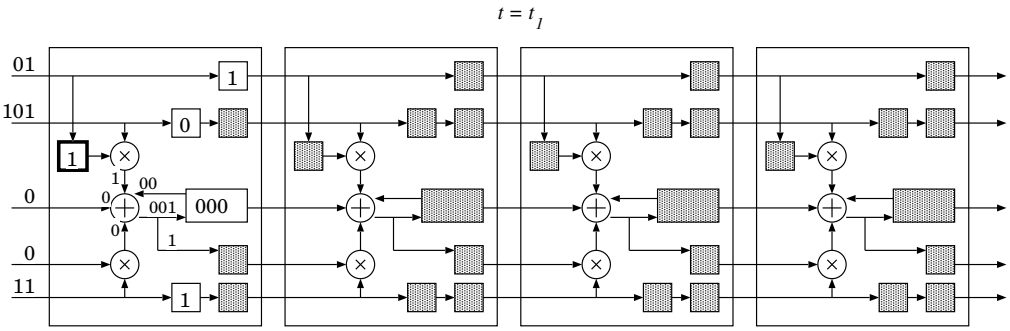
2.3. Operation Illustration

We show, in this section, the graphical representation of the Montgomery multiplier using the systolic array just presented. In this example, $A = 011, B = 101$, and $M = 111$. We define t_i (where $i = 0, 1, 2, \ldots$) as a clock cycle.

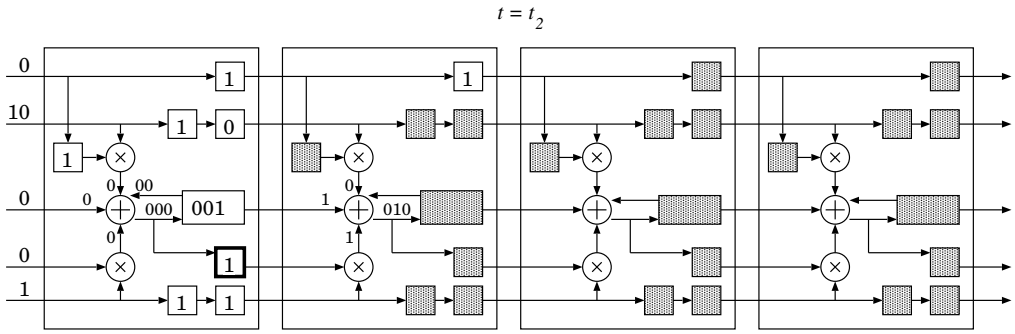
$t = t_0$; Vectors of operands line up at the input terminal of PE#0. $011, 1010$, and 111 are shown up at A_{in}, B_{in}, and M_{in} respectively. For P_{in} and Q_{in} of PE#0, a bit 0 is always taken here.



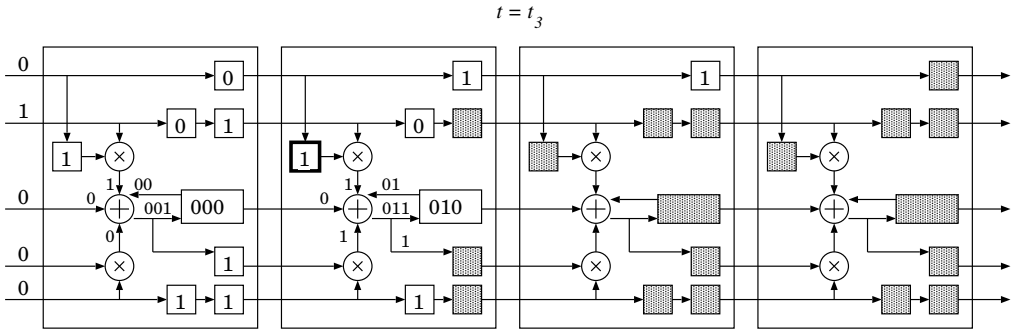
$t = t_1$; a_0 which is 1 is registered to Ra^* and Ra. The value in Ra^* will not be changed; however, the value in Ra will be altered as the operation goes on since it merely acts like a buffer for the pipeline. At *mul1*, a_0 and b_0 are multiplied, and the result is sent to *in1* of *add*. All other inputs of *add* have 0; therefore, the result is 001. This value will be registered to Rz in the next clock cycle, and its LSB will be stored in Rq as $Q[0]$.



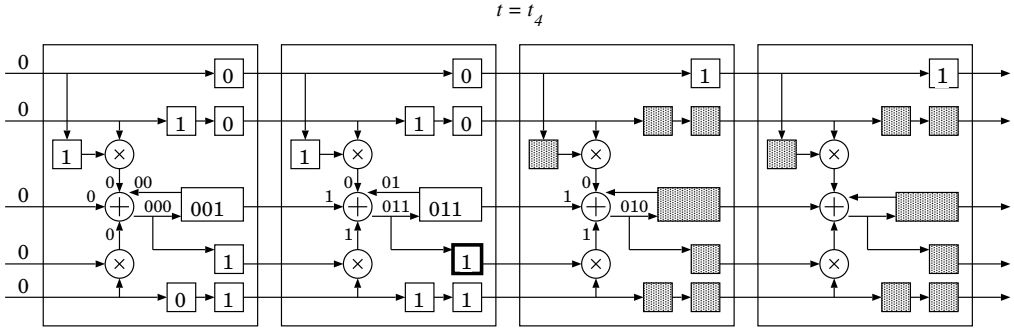
$t = t_2$; At PE#0, bit vectors of A, B, and M are still moving along their corresponding registers. Rz now has a value of 001, and Rq is registered with 1 which is a value of $Q[0]$. *mul1* now takes a_0 from Ra^* and b_1 as its inputs. The result from *add* for this cycle is 000. Again, this value will be registered to Rz in the next cycle, but Rq will not further change the value. At PE#1, a_0, which is passed on from Ra of PE#0 in the previous cycle, is now buffered in Ra of PE#1. We now take a look at *add*. *in1* has 0; *in2* has $P[0][1]$ which is 1; *in3* has a multiplication of m_0 and $Q[0]$, and it is 1; lastly, *in4* is 00. The result of *add* which is 010 will be registered to Rz in the next clock cycle.



$t = t_3$; At PE#0, the same operation occurs as it does in the previous cycle. At PE#1, this is very similar to what happens to PE#0 at t_1. Ra^* is registered and not changed with a_1. *mul1* takes a_1 an b_0 as the inputs and gives the result to *add*. *mul2* has m_1 and $Q[0]$; it sends the result to *in3* of *add*. Besides those two inputs, the LSB of Rz from PE#0 and feeded-back carry bits of its own Rz are the other two inputs for *add*. The result of *add*, 011, is sent to Rz to be registered in the next clock. And the LSB will be kept in Rq to be $Q[1]$.



$t = t_4$; At PE#1, Rz is now 011 which is the result from *add* of the previous cycle. also, Rq is registered with 1, and this is used as $Q[1]$ for the operation. *in1* of *add* now has a_1b_1; *in2* has 1 from PE#0's Rz; *in3* has $Q[0] \cdot m_2$; and *in4* has carry bits, 01. Then, the result is 011. At PE#2, there is the operation at *add* by adding $Q[1] \cdot m_0$ and the LSB from PE#1's Rz.



The systolic array has a similar routine to the previous clock cycles. We note here only important events which are:

- a_2 is stored in Ra^* of PE#2 at t_5,
- $Q[2]$ is stored in Rq of PE#2 at t_6,
- an appended 0 bit is stored in Ra^* of PE#3 at t_7,
- the final result p_0, p_1, p_2, and p_3 are coming out of Rz of PE#3 at t_8, t_9, t_{10}, and t_{11} respectively. The result of the operation is in the range of $[0, 2M-1]$. Therefore, we need a hardware to do the final reduction step.

2.4. Partitioning the Systolic Array

As described in the previous chapter, there are several advantages of the scalable design over the fixed hardware. In this section, we discuss the possibility of making the systolic array, presented in the prior section, *scalable*. To accomplish this, we apply the partitioning method proposed in [6] to the systolic array.

For the algorithm shown in Figure 2.4, we can partition the outer loop (loop i) into smaller computations. That means we divide n+1 computations of the outter loop to the small number of computations and perform these tasks several times. If the number of computations is p, then the number of iterations (performing these p computations) k has to be such that $k = \lceil \frac{n+1}{p} \rceil$.

The hardware mapping for the partitioned algorithm can be realized as a linear array of p PEs, and we call this array a *pipeline*. Since there are intermediate data communicating between each PE, there have to be a storage keeping the communications between the last PE of the pipeline and the first PE of the next round (*pipeline cycle*). These communications during the pipeline cycle are the intermediate result P and Q. The storage for P is designed as a FIFO. The size of this FIFO have to be large enough to collect all bits that will be passed through the pipeline. And, it is $n + 1$. For Q, a one-bit register is enough for this purpose. The data path infrastructure for this hardware is shown in Figure 2.7

Since, in the full precision array where $r = 2^n$, the systolic hardware needs $n + 1$ PEs to finish the operation, r has to be carefully selected when the array is partitioned into the pipeline. As discussed that the p-PE pipeline needs k pipeline cycles to finish the operation, the equivalent number of PEs is kp. Thus, r for this case should be a constant $r = 2^{kp-1}$.

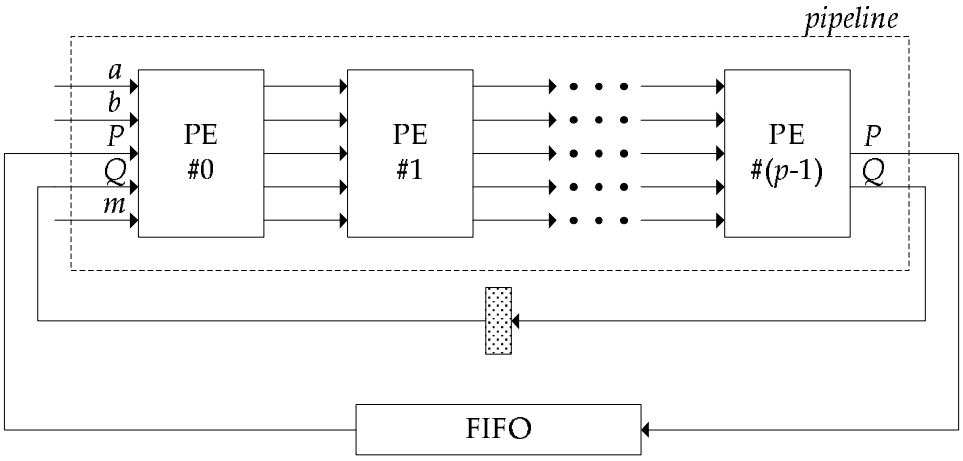


FIGURE 2.7: Data Path Infrastructure for the Partitioned Systolic Array

2.5. Comparison of Scalable Architecture

Let us compare the design of scalable systolic array described in the previous section (*design-C*) with the design by Tenca and Koç, proposed in [3] (*design-TK*). Since *design-C* was not synthesized for any technology, we do not compare the area or clock period of the design. We compare both designs by looking at the design perspectives and their operation.

The PE of both designs have very similar arithmetic units. The PE consists of two vector-by-bit multipliers which can be realized by arrays of AND gates. One 4-input adder is employed in each PE. In *design-C* the adder is designed by using Carry Ripple Adder (CPA) whereas, in *design-TK*, Carry Save Adder (CSA) is used.Now, we might conclude that the infrastructure (pipeline plus memory) of *design-TK* requires more area than the other because we need extra storage for data in a carry save form.

The computation time for both designs can also be compared. When there are p PEs in the pipeline, *design-TK* takes $\lceil \frac{n}{p} \rceil$ pipeline cycles while *design-C* needs $\lceil \frac{n+1}{p} \rceil$ to finish the computation. Most of the time the two numbers are the same.

However, when n is a multiple of p, *design-C* will take one more pipeline cycle than *design-TK*.

design-C has a critical disadvantage when compared to *design-TK*. The PE used in *design-C* is designed based on the description of the hardware implementation presented in [4]. However, *design-C* is found to work only when $w = 1$. Although it is claimed to be work with any word size, the design for those cases are not shown in the paper, only stated that the computation can be done by using a similar method. It is not possible to derive information from the paper. Being able to work only with $w = 1$ is a huge limitation in terms of flexibility.

It is a disadvantage of a scalable architecture to work only with $w = 1$. First of all, the number of clock cycles taken to finish the multiplication is larger than that with a bigger word size when working with the same number of stages. The total execution time for the system would be much longer.

3. I/O SUBSYSTEM DESIGN.

In this chapter, we describe the design aspect of the I/O subsystem used for Montgomery multiplier hardware. The overview of the multiplier hardware structure will also be shown in this section. And, we explain the problems of the previous version of the I/O as well as the difficulties in I/O designing. Then, the new version of the design is described with details for all components used to build up the I/O block.

3.1. Montgomery Multiplier Hardware Overview

Montgomery Multiplier Hardware(MMHW) consists of two main components: the I/O subsystem and the kernel. The block diagram of the top level of this hardware is shown in Figure 3.1 below.

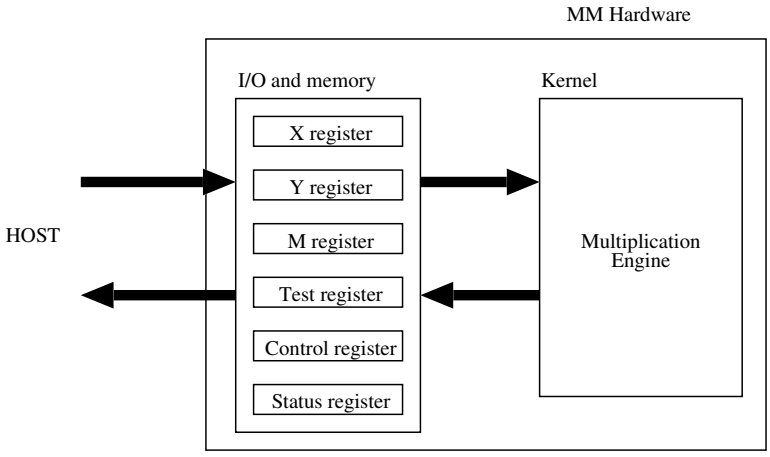


FIGURE 3.1: Block Diagram of MMHW

In the Figure 3.1, only the registers that are visible to the host system (or application software) are shown. The kernel is described in detail in [7]. X, Y, and M registers are used to store (from the host) and distribute (to the kernel) the values of X, Y, and M operands. The Test register is offered for testing purposes. It is used to read and write various internal locations. The Control register is used to hold the commands and configurations for MMHW operations. The Result register stores the result coming out of the kernel.

Inside the I/O subsystem, operands are divided into groups of bits (words) and send out to the kernel one word per clock cycle. The word size for X, Y and M operands is based on the kernel configuration. The word size for X is an integer number depending on the radix of the kernel. For example, the radix-2 version of the kernel would need one bit from X each time a new PE starts a computation; therefore, the word size for X is 1. For Y and M, the word size is variable and could be configured as 1, 2, 4, 8, 16, or 32. The size of 32 bits is the maximum value allowed in this version of the I/O subsystem design.

3.2. Previous Work on the I/O Subsystem

The first version of the I/O subsystem was developed by the research group in the *Information and Security Laboratory* at Oregon State University. It was designed to support a radix-8 version of the kernel [7]. There are three critical limitations in this first design:

- It was designed to work only with the radix-8 kernel; therefore, each word that the X register distributes to the kernel has three bits.
- Since data in Y and M registers are reused by the kernel in several pipeline cycles, we need a method to keep those data in the registers. The *wrapping*

method, which is used in this design, is not conventional. Even though it uses the idea of the rotating data, it does not actually write the data back to the register after the data is read out. Instead, these data are never destroyed; they are still located at the same position inside the register. The first word of data is located at a so-called *base address*, and the read address is initialized to this value. During the computation, the read address is increased one position after one word is read out from the register. When the next computation cycle starts, the read address is assigned with the value of the base address in order to move the pointer back to where the first word is located. This process continues until the computation is finished.

- The size of the memory modules used for X, Y, and M registers are fixed to 2,048 bytes, which make the synthesis of the multiplier impractical for small devices (small area). That is a limitation to flexibility.

This thesis presents an I/O design that is reconfigurable. This means the number of bits coming out of the X register can be any integer number as explained previously. The size of the queue is also adjustable by using the memory that can be configured both in depth and width.

3.3. I/O Subsystem Architecture

Our goal in redesigning the I/O subsystem used in the MMHW was to create a generic and parameterizable solution that uses basic digital components as much as possible. In this section, we describe the design of all of its components. The block diagram illustrating the structure of the I/O is shown in Figure 3.2.

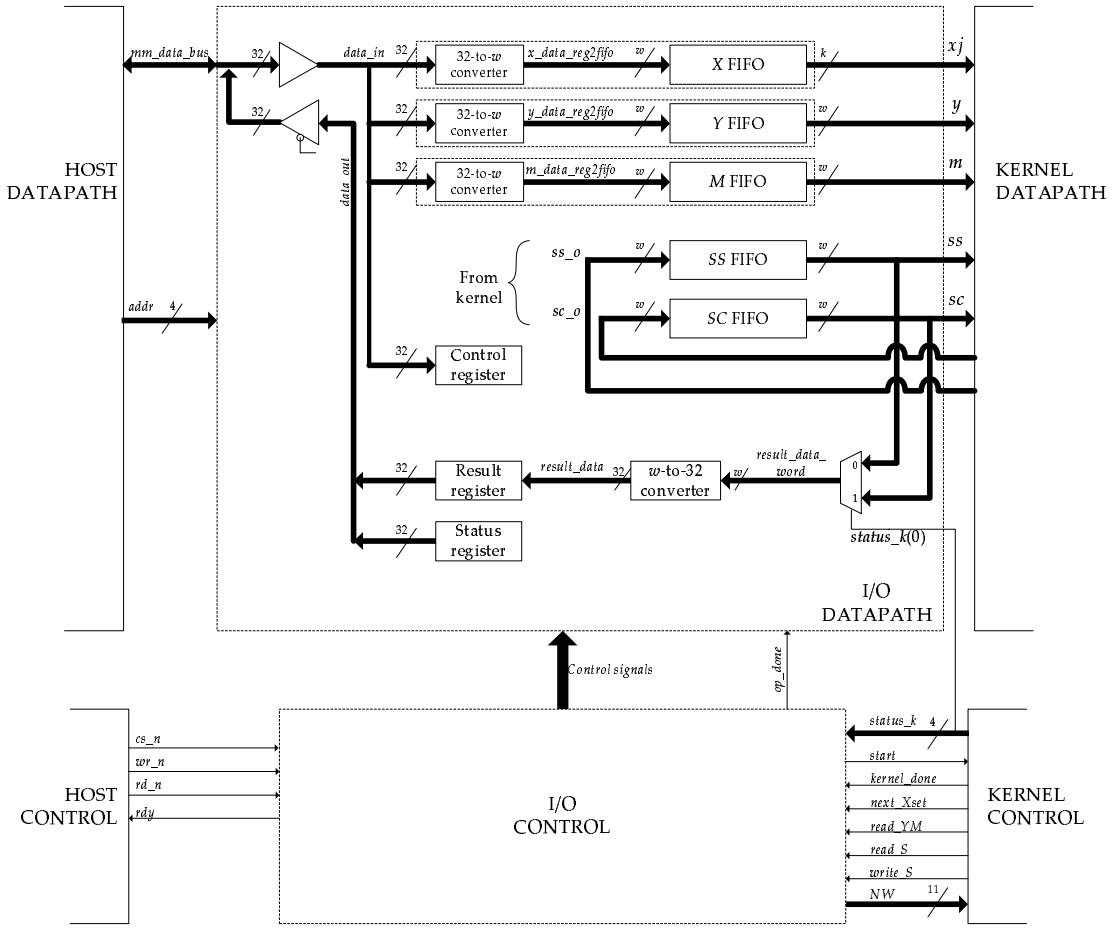


FIGURE 3.2: Block diagram of the I/O subsystem

3.3.1. MMHW Operation Overview

The user loads data into the MM, activates the operation, and reads the result through a sequence of read/write cycles on the MM registers.

First of all, the user writes the operand size to the control register. Next, the user loads the X, Y, and M operands. They are loaded with several 32-bit words. They could be loaded in any order, i.e. we can load either X, Y, or M

first. However, for each operand, words must be loaded in sequence, from the least-significant word (LS Word) to the most-significant word (MS word).

For example, if the operand size is 96 bits, and the user wants to load $Y = (y_{95}y_{94}\ldots y_1y_0)$. The user has to start with the LS word $(y_{31}\ldots y_0)$ first. Then, the next word to be written is $(y_{63}\ldots y_{32})$. And, the last word of Y to be sent to the I/O is $(y_{95}\ldots y_{64})$.

After loading all operands, the user tells the hardware to start the operation by writing a command to the control register. After while, the kernel finishes the computation and notifies the I/O to update the status register as the multiplication was completed. The user obtains this information by reading the status register. As of now, the result is stored inside the S FIFOs. Then, the user can read 32-bit words from the result register in several consecutive cycles. The status register is updated again when all bits of the result are acquired.

3.3.2. Host Interfacing Pins to MMHW

There are many signals used to connect the MMHW to the host system. Table 3.1 shows a type and describes the function for each of those ports.

MMHW is selected by the host system when *cs_n* is asserted. This signal is generated by an address decoder outside the MMHW. To access the registers inside the MMHW I/O block, the 4-bit *addr* signal is used to specify those register addresses. The 32-bit *mm_data_bus* is the bus carrying data between the host system and the MMHW during read or write cycles. The *rd_n* is asserted when the host system performs a read cycle. Likewise, the *wr_n* is asserted when a write operation is executed. The MMHW can be globally reset by the assertion of *reset_n*. The *rdy* signal is used to synchronize the read and write operations when the MMHW is not ready to accept new data or deliver a requested result.

Pin(s)	Function	#	Type	Description
cs_n	chip select	1	in	Validates the address applied to the MMHW. It is generated from an external address decoder.
addr	address	4	in	Address for accessing data in the MMHW.
mm_data_bus	data	32	in/out	A bidirectional data bus to/from the MMHW.
rd_n	read strobe	1	in	With *cs_n*, validates an address for reading data from the MMHW.
wr_n	write strobe	1	in	With *cs_n*, validates an address for writing data to the MMHW.
clock	clock	1	in	Clock source for MMHW.
reset_n	reset	1	in	Reset signal to MMHW. Resets all internal state storages to zero.
rdy	ready	1	out	Data ready signal. The MMHW will pull *rdy* low in a case where the host attempts to write to the operand FIFO in the MMHW, but the previous write request has not finished loading data to the FIFO. Or, the host attempts to read the result from the result register, but the value in the register is not ready to be read.

TABLE 3.1: Pin Description (reproduced from [12])

3.3.3. Accessing to I/O

To access the registers inside the I/O subsystem, the host has to provide specific address bits (*addr*) for those registers. The following table shows the addresses associated with the I/O registers.

Address Pins	Register
0111	result register
0110	Y operand register
0101	X operand register
0100	M operand register
0011	reserved
0010	test register
0001	control register
0000	status register

TABLE 3.2: I/O registers and their addresses

There are four registers in the I/O to which the user can write: Control, X, Y, and M registers. When the host writes to one of these registers, it asserts a request (*wr_n* signal) and provides the register address to the I/O.

During the reading operation, there are two registers from which the host can read: status and result registers. Similar to writing operation, when the host reads from the I/O, it sends *cs_n*, *rd_n* signals, and the target address.

3.3.3.1. Writing the Control Register

The control register can be written when *wr_n* is asserted with $addr = 0001$. This 32-bit register contains command and parameters for the MMHW hardware. The command and parameters are specified in several fields which are described in detail in [9]. Figure 3.3 shows the 32-bit diagram consisting of fields in the control register. Since the modification is made to the I/O design, only new and modified fields will be described below:

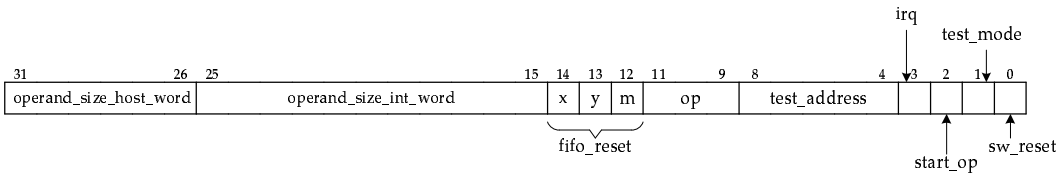


FIGURE 3.3: The diagram of the fields in the control register

- operand_size_host_word - control_reg(31:26)[6 bits]: This field specifies the X, Y, and M operand size in host words. The host word size is 32. Note that the length of the X, Y, and M operands must be an integer number and a multiple of 32. The MMHW supports operands up to 2048 bits precision which is (2048/32 = 64 host words).

- operand_size_int_word - control_reg(25:15)[11 bits]: This field specifies the X, Y, and M operand size in (internal) kernel words. The number in this field corresponds to the number in operand_size_host_word field. For example, if the operand is 128 bits precision (which is 4 host words) and the word size is 8, the number of kernel words is (128/8) = 16.

- x_fifo_reset - control_reg(14),y_fifo_reset - control_reg(13), m_fifo_reset - control_reg(12): These bits are used to reset the operand FIFOs. Each FIFO can be reset individually by setting its corresponding bit to 1.

3.3.3.2. Writing Operand Registers

The operand registers can be written when *cs_n* and *wr_n* are asserted with appropriate operand addresses. Additionally, the selected operand register has to be "ready". Since the operand register takes 32-bit input at each clock cycle, but it internally works with w-bit output, it takes more than one clock cycle to transfer data from the operand register to its permanent location in the FIFO when w is less than 32. Then, the host cannot write the next 32 bits to the operand register when it is in the transfer period (the register is not ready).

Figure 3.4 shows a timing diagram when the user loads data to the X FIFO. In this example, w is 8 bits, so there are 4 words in the X register to be transferred to the FIFO. The operation starts when the host sends to the I/O block a write request, the address 0101 used to specify the X register, and 32-bit data. Then, at the next rising edge clock, the host data are written to the register. A state machine starts sending all 4 words to the FIFO. However, in this example, the host requests to write again to the register while there are still two words of the current 32-bit word left in it. Therefore, the new data cannot be written to the register at this time, and the I/O control pulls the *rdy* signal down, forcing the host to wait. The write cycle is accomplished when the *rdy* signal becomes high.

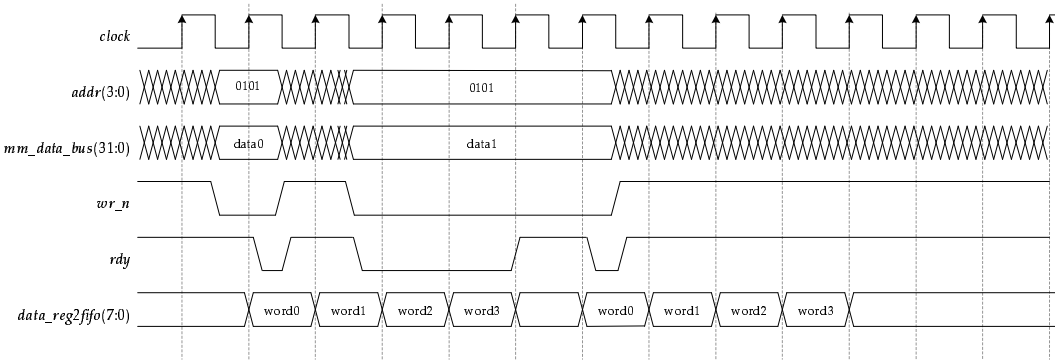


FIGURE 3.4: Example of the timing diagram for writing X operand

3.3.3.3. Reading the Status Register

The status register can be read by the host when *rd_n* is asserted with the address pins (*addr*) are 0000. The host can monitor the status of the MMHW by looking at the contents inside the status register. The status of the MMHW is defined in many fields. Figure 3.5 shows the diagram of these fields in the status register. Some of the status fields is already described in [9]. Only new fields are described in the following list:

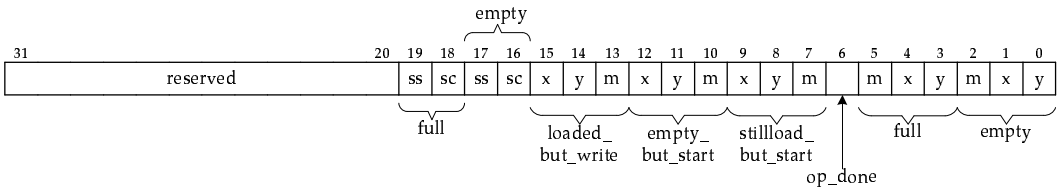


FIGURE 3.5: The diagram of the fields in the status register

- m_stillload_but_start - status_reg(7), y_stillload_but_start - status_reg(8), x_stillload_but_start - status_reg(9): These bits indicate errors occurring to

the corresponding operands. This error generated when the host started the multiplication before all operand words have been loaded.

- m_empty_but_start - status_reg(10), y_empty_but_start - status_reg(11), x_empty_but_start - status_reg(12): These bits indicate an error generated because the operand FIFO has not been loaded with any data (empty), and the user starts a multiplication.

- m_loaded_but_write - status_reg(13), y_loaded_but_write - status_reg(14), x_loaded_but_write - status_reg(15): These bits indicate the error generated because the operand FIFO has already been loaded with all operand bits, but the user writes more data to the FIFO.

- sc_fifo_empty - status_reg(16), ss_fifo_empty - status_reg(17): If set to one, these bits indicate that the corresponding FIFO is empty.

- sc_fifo_full - status_reg(18), ss_fifo_full - status_reg(19): If set to one, these bits indicate that the corresponding FIFO is full.

- reserved - status_reg(31:20)[12 bits]: Reserved for future use.

3.3.3.4. Reading the Result Register

Reading from the result register is controlled by the assertion of *cs_n* and *rd_n* with *addr* is 0111. Also, the result register has to be "ready". Since the size of the register is 32 bits wide, but it only gathers one w-bit word from the S FIFOs in each clock cycle, it takes more than one cycle for the register to have a 32-bit result word ready when w is less than 32. If the host requests to read when the result register is in the gathering period (it is not ready), then the host cannot read from the register.

Figure 3.6 shows a timing diagram when the I/O builds the 32-bit result word and keeps it in the result register, ready for the host to read. Whenever the *op_done* signal is set, the state machine starts gathering words from the S FIFOs to fill in the result register. In this example, w is 8 bits; therefore, there are 4 words to be read from the S FIFOs. The I/O takes at least 5 cycles to complete the filling the result register. If the host reads from the result register during these cycles, the I/O will pull the *rdy* signal to 0, so the host has to wait. The read is successful when the *rdy* signal is set back to 1, and then the 32-bit result is shown up at the data bus.

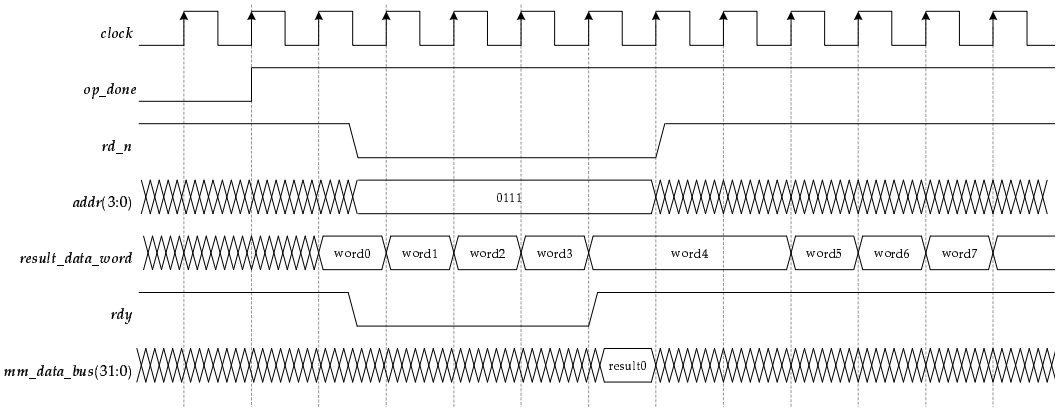


FIGURE 3.6: Example of the timing diagram for reading the result register

3.4. Description of Building Blocks

3.4.1. 32-to-w Converter

This module is constructed by a normal parallel-in/serial-out shift register and its control unit as shown in Figure 3.7. It takes 32-bit input at a time. The

serial output bits are shifted out w bits every cycle where w is a word size defined by the kernel design. w can be described as the number of bits in each *kernel word*.

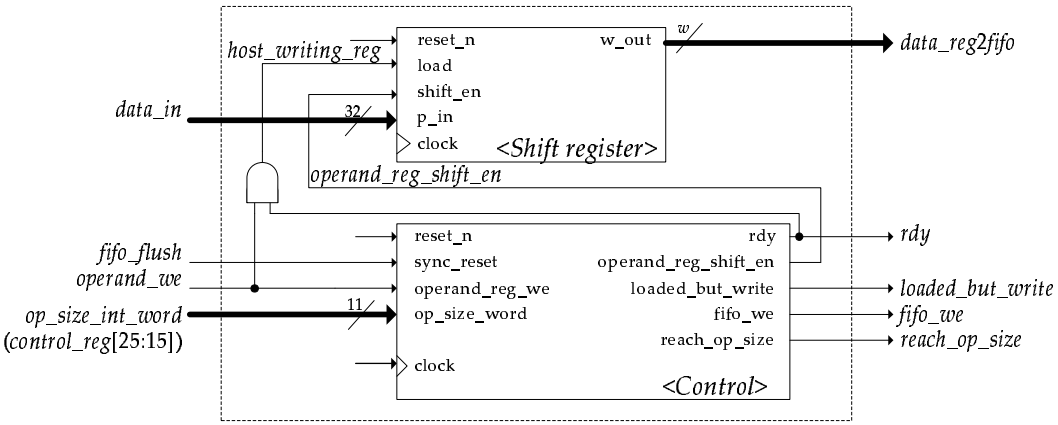


FIGURE 3.7: System level of 32-to-w converter

The I/O has three copies of this type of the converter for each one of the operands' FIFOs. It is mainly used to interface between the 32-bit host bus and the input port of the FIFO. Since the FIFO is constructed by a random access memory (RAM) and the width of the RAM is defined by w, we need a temporary storage that is able to hold 32-bit data from the host, and then transfer these data to the FIFO, w bits at a time.

It also has a control part that maintains the proper functionality of the I/O. Firstly, it has to be able to correctly pass on 32-bit data from the host to its corresponding operand's FIFO. So, whenever there is a request from the host to write data to the converter, in the next clock cycle, the *fifo_we* signal (a write request signal) is sent to the FIFO, enabling it for the write operation.

The *rdy* signal, which is used to control the host writing the FIFO, is also implemented in this unit. As explained in section 3.3.3.2. on page 30, the *rdy*

signal is normally low, and it will be set high when the host attempts to write new data into a FIFO that is still being updated (written).

The *reach_op_size* signal indicates that all operand bits have been loaded to the FIFO. A counter keeps track of the number of operand words loaded to the FIFO. If the value of this counter is equal to the operand size (portion of the control register), the *reach_op_size* will be set high.

The converter generates two signals indicating that there is an error in the operation: *loaded_but_write* and *stillload_but_start*. The *loaded_but_write* error indicates that the FIFO has already been loaded with all the bits of the operand, but the user wrote more data to the FIFO. The exact size of the operand is needed for the MMHW. If the user writes more data to the FIFO, the result from MMHW will be incorrect. The *stillload_but_start* error indicates that the size of data in the FIFO is less than the size of the operand, which means the FIFO expects more data to come; however, the user started the MMHW.

3.4.2. *S* FIFOs

Regular FIFOs are used to store word vectors for both *SS* and *SC*. The interface of the FIFO is shown in Figure 3.8. *data_in* is a w-bit input port. It is connected to one of the kernel ports where words of S are sent out. The data presenting at this port is written to the FIFO when there is a write request via a *fifo_we* pin. *data_out* is a w-bit output port. This ports sends out a word of S to the kernel when it is requested via a *fifo_re* pin. *full* and *empty* are output pins indicating the status of the FIFO. The FIFO can be reset (set to empty) manually via either *reset_n* (global system reset) or *fifo_flush* pin.

Figure 3.9 shows a block diagram of the FIFO design. The FIFO uses RAM as its storage. Since the target technology for our design is Spartan II FPGA

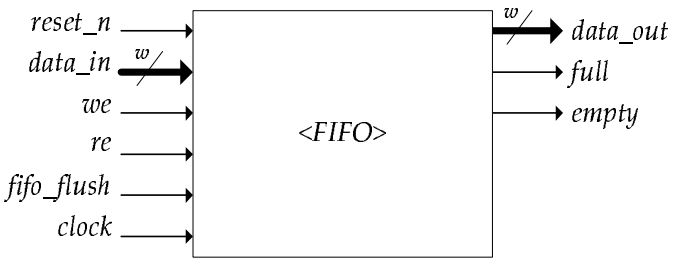


FIGURE 3.8: The interface of the FIFO used for S

chip [10], we use on-chip dual-port Block RAM [11] to be a memory element in this design rather than a RAM module described in VHDL.

The dual-port Block RAM has two independent ports: A and B. Both ports have an access to the same memory space, but they are independent to each other which means each memory port has its own interface, e.g. address bus, data in/out bus, clock, etc. We configure port A for read. The interfaces for port A are: read address(ADDRA), clock(CLKA), and data out(DOUTA). On the other hand, port B is configured for write, and its interfaces are: write address(ADDRB), data in(DINB), write enable(WEB), and clock(CLKB). Note that the bus for data in and address are separate lines.

Moreover, the RAM has a registered output, which implies that the content of the memory, when read, is registered before showing at DOUTA pins. So, whenever the address at ADDRA changes, the content of that address will show up at DOUTA pin at the next clock cycle.

The FIFO configuration is parameterizable. This can be done by changing the width and depth of the RAM. w represents the width of the memory. The depth is configured by d which is the width of the address bus. Then, the depth of the RAM is 2^d. For instance, if w is 8 and d is 5, this gives a memory size of 8×32 which is 256 bits.

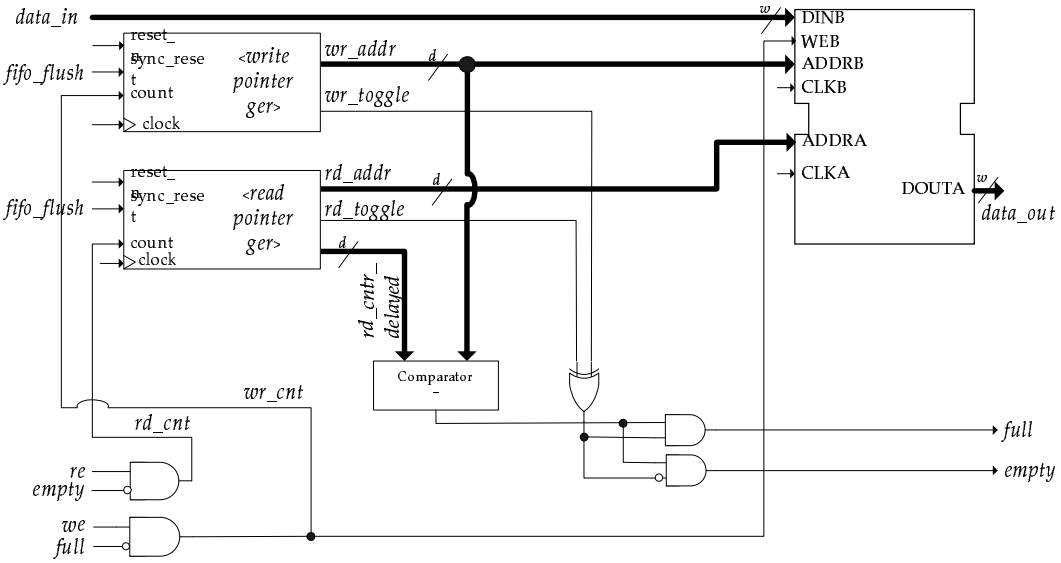
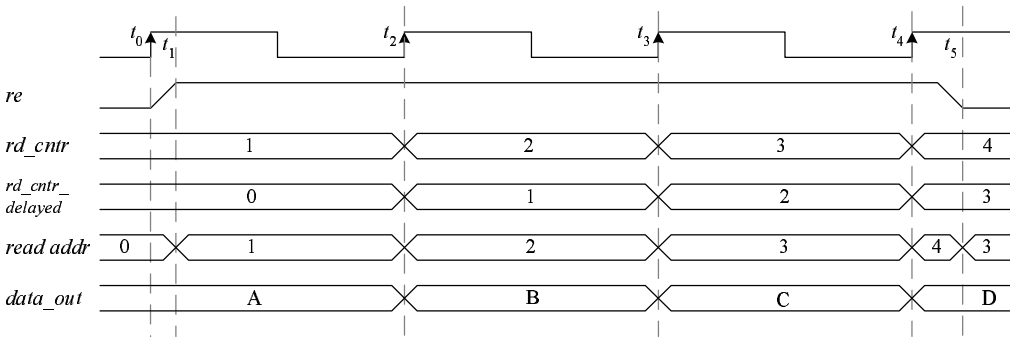


FIGURE 3.9: Block diagram of the FIFO

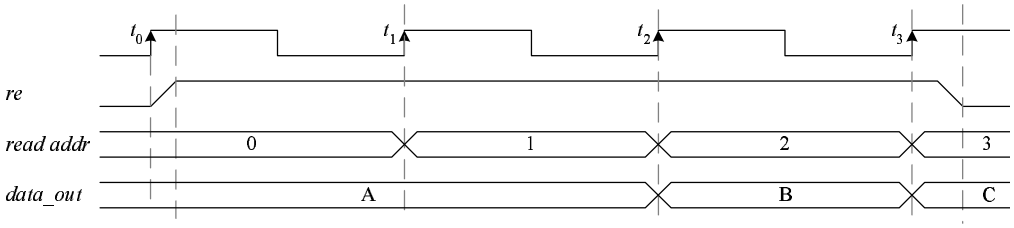
The write pointer is generated by the use of a regular counter. However, it is a different story for generating the read pointer. Since the RAM has a registered output as explained previously, the data is delayed by one clock cycle. Therefore, when the data are read out from the FIFO, the sequence of data is wrong. Figure 3.10 shows a comparison of the sequence of data read from the queue along the clock. Figure 3.10(a) is the one that we expect for the FIFO; Figure 3.10(b) is the one that happens when we use just a counter to generate the read pointer.

If we take a look at Figure 3.10(b), we can see that the data are delayed. The first word of the queue shows up at the output port for two clock cycles. Therefore, we need an extra control to correct this problem by advancing the read pointer one position in order to have the next word ready when it is needed.

The control part for generating the read pointer is shown in Figure 3.11. It consists of a regular counter, an array of flip-flops with an enable input, and a multiplexer. The counter has 1 as its initial value which means when the system is reset, the counter starts counting from 1. The read enable signal is the signal



(a)



(b)

FIGURE 3.10: Comparison of the sequence of data read from the FIFO

that enables the counter to operate. Then, the value of the counter is sent out to the flip-flops and the multiplexer. The purpose of those flip-flops is to delay the output from the counter. Thus, the value that comes out of the flip-flops is always one less than the value of the current counter. Each flip-flop has the initial value of 0; therefore, at the initial state, the flip-flops give 0, but the counter gives 1. Then, the multiplexer selects the value of the read pointer from either the flip-flops or the counter by using the read enable for a select signal.

The FIFO can be re-initialized by the user via the *fifo_flush* signal. This will only move the write and read pointers of the FIFO to their initial position. Thus,

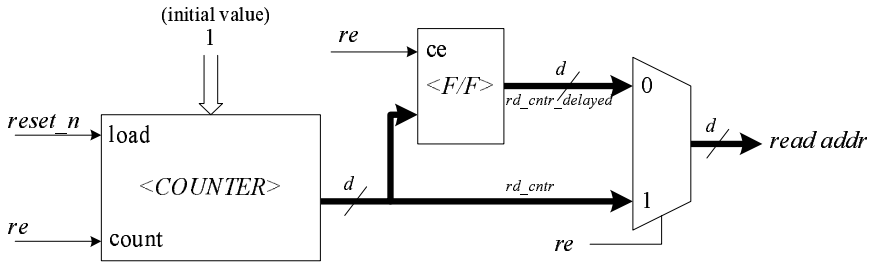


FIGURE 3.11: The structure of a read pointer generator

the FIFO is empty. But, the data in the FIFO need not to be cleared; they can be left in the queue.

3.4.3. Y and M FIFOs

The FIFOs, used for both Y and M operands, have a rotate capability. This means they can be selected to be regular queues or rotators. When the FIFO is acting as a rotator, the data coming out of the FIFO are written back into it.

To make the FIFO feasible to function as a rotator, the regular FIFO used for S operand is upgraded. Figure 3.12 shows the modification to the FIFO.

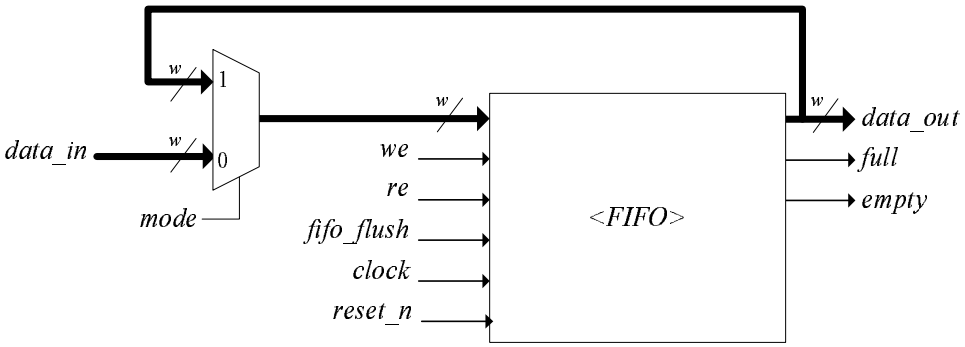


FIGURE 3.12: Block diagram of the rotator design

The multiplexer is added to the data path of the FIFO. Then, the data attempted for the FIFO is selected between an external data or the data out from its own output port. *mode* signal is used for the selection.

The signals that activate the increment for both pointer generators are also modified. In a rotator mode, both generators increase their value according to the read operation. Therefore, the multiplexer is added to the control section as shown in Figure 3.13. This logic circuit is designed based on the function of this FIFO version as shown in Table 3.3

Main function	Signals					FIFO Operation
	mode	*we*	*re*	*wr_cnt*	*rd_cnt*	
NOP	X	X	X	0	0	NOP
Queue	0	0	0	0	0	NOP
	0	0	1	0	1	read
	0	1	0	1	0	write
	0	1	1	1	1	write and read
Rotator	1	0	0	0	0	NOP
	1	0	1	1	1	rotate
	1	1	0	0	0	NOP
	1	1	1	1	1	rotate

Note: NOP - No Operation, X - Do not care

TABLE 3.3: The truth table signals controlling FIFO read/write

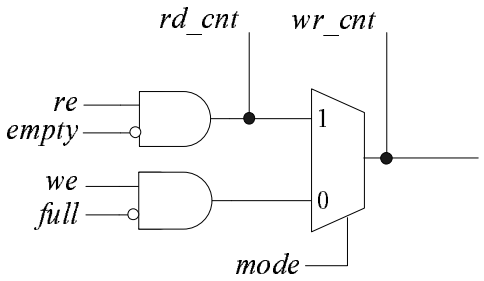


FIGURE 3.13: Logic for pointer generators' count signals

3.4.4. X FIFO

Unlike the other two operands, the operand X is not reused during the MMHW operation. Its value can be discarded once it is loaded into the kernel. We do not have to write it back to the queue. Therefore, the design of the storage for X should be very simple. However, the number of bits that is taken from X is not w. This number varies depending on the radix used in the kernel design. For instance, in radix-8 kernel, 3 bits are needed each time X is read. This means that the design of a storage for X will be very complex if we would like to construct it with the RAM-based FIFO. However, we will still refer to this module as an X FIFO.

Figure 3.14 shows our design for X FIFO. We design the module as a series of the parallel-in/serial-in/serial-out k-shift registers. The registers are connected to one another in a linear form. Each register takes w bits parallel or k bits serial input. At the output, it shifts out serial k bits. When data is loaded into the FIFO, the control sends an enable signal to one of the registers, and then write to that register. The serial output port of one register is connected to the serial input port of another register on its right.

For X operand, we only write to the FIFO until the whole data of X is kept inside. Then, we start to read out from it; the read operation continues until

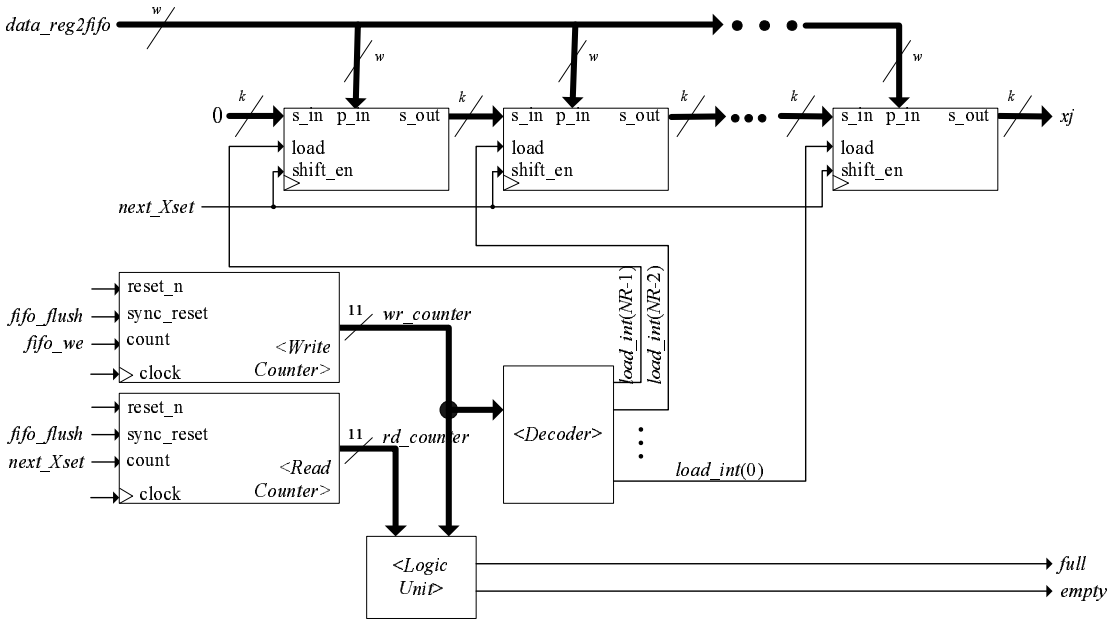


FIGURE 3.14: The structure of X FIFO

the MMHW finishes its operation. In the control section, there is a counter that counts every time when there is a write request for the FIFO. The value of the counter is corresponding to the position of the register in the series. So, the value of the counter is sent to a decoder, which , then, sends an enable signal to the corresponding register. Thus, only the chosen register is written with w-bit word sent from the X's 32-to-w converter. When the FIFO is read out, k bits are shifted out at the output port of each register. These k bits are sent to the serial input port of the adjacent register. The k bits output of the right most register is send to the output port of the FIFO which is connected to the kernel.

3.4.5. w-to-32 Converter

After the kernel has finished the multiplication, it sends the *kernel_done* signal to the I/O. At this time, the result is stored inside either the SC or SS

FIFOs. Since the result register is 32 bits wide, but the output of the FIFOs has only w bits, we need a hardware to convert multiple words of w-bit data to a 32-bit value. We call this circuit a w-to-32 converter. The value of w is a power of 2 and less than or equal to 32; therefore, we need $NW\_in\_32 = \frac{32}{w}$ words in order to construct the 32-bit data and put them into the register. The diagram of w-to-32 converter is shown in Figure 3.15 below.

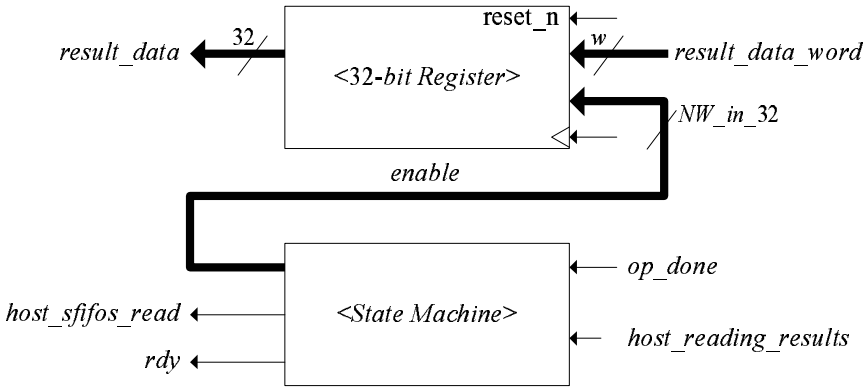


FIGURE 3.15: The structure of w-to-32 converter

When the *op_done* signal is set, the state machine sends $host\_sfifos\_read$ to both of S FIFOs requesting the first w-bit word from them. The $result\_data\_word$ signal is the w-bit data from either SC or SS FIFOs (selected by the $status\_k(0)$ signal from the kernel). This first word is the least significant word of the result; therefore, when we write w bits to the 32-bit result register, the least significant position is written first.

In the state machine, there is a counter that keeps track of the number of words of the result read from the S FIFOs. It is used to make sure that the exact number of NW_in_32 words are read from the S FIFOs and written to the result register. Additionally, we use the value of the counter to generate the enable signals

for writing the result to the proper position in the result register. For instance, if w is 4, there are 8 words (NW_in_32 is 8) to read from the S FIFOs and write to the register. We assume that at some point when the value of counter is 3, the enable signals which are 8 bits will be 0000 0100. Therefore, at the next rising edge of the clock, the third word of the result from the S FIFOs will be written from the 8^{th} to 11^{th} position of the register.

When the result register is ready with 32-bit result data stored inside, the host can now send $host\_reading\_results$ signal to the I/O. This process connects 32-bit result register to the host data bus (i.e. mm_data_bus), and takes the result into the host system. Then, the process of taking words from the FIFOs and constructing the result register is repeated until the op_done is cleared, i.e. the host receives all the bits of the result value from the I/O block.

4. EXPERIMENTAL RESULTS AND ANALYSIS

In this chapter, we present the performance evaluation of the MMHW (both radix-2 kernel and I/O subsystem) based on area and time estimation. The discussion of the kernel alone was already presented in [7] and [3] for ASIC, and [12] for FPGA implementation.

4.1. Simulation, Synthesis, and Implementation Environment

The design was simulated in Modelsim for testing and verifying its functions. It was implemented and optimized for FPGA. Xilinx Synthesis Technology (XST), embedded in Xilinx ISE WebPack version 5.1i, was used in the project as a synthesis tool. Then, the implementation results were generated and presented in this chapter.

The main target technology is a Xilinx FPGAs. One of them is the Spartan II XC2S200 [10]. This chip has 28×42 CLB arrays which is 1176 CLBs and equal to 2352 slices. It has 14 blocks of RAM with the toltal size of 56 Kbits. The design was also synthesized for Virtex XCV200 [13]. This chip has the same number of slices and block RAMs as the other one. However, the implementation results are very similar for both families. Therefore, we will only illustrate the results for Spartan II XC2S200.

During the experiment, many design configurations were tried by varying the values of operand precision (n), word size (w), number of PEs in the pipeline (NS), and size of the registers. For this work, n was set to 256, 512, 1024, or 2048 bits; w was considered as 8, 16, or 32 bits; NS was an integer number. The size

of registers was set to store 2048-bit operands for each case of w. Therefore, the size of RAM were 8x256, 16x128, and 32x64 for w as 8, 16, and 32, respectively.

The data were collected after the Place and Route (PAR) step because the post-PAR results are more accurate than those of synthesis step. During PAR step, the tool applied the constraints to device resources (e.g. number of slices, I/O pins, etc.). The constraint of the number of slices created the limitation of NS when the design was implemented with different configurations.

4.2. Area Estimation for MMHW

The area of the MMHW depends on word size (w) and number of stages (NS). The dependence of w and NS over the kernel area was discussed in [7]. And, as mentioned eariler, the operand register size is set to store 2048 bits which is the maximum operand size; thus, the I/O can be used with the scalable kernel to work with any operand size upto its maximum. Hence, the only parameter that has an effect on the I/O area is w. For example, the I/O area for $w = 8$ is 1517 slices which is about 65% of the chip. Figure 4.1 shows the area of MMHW configurations against the number of PEs for all three word sizes.

The figure shows that the area of MMHW linearly increases for each addition of NS value. However, after some value of NS, the chip gets crowded, and the area stays constant at 2,350 slices which is almost 100% of the chip area.

As stated previously, only the I/O alone already takes about 65% of the chip area. The main reason for this is because the X FIFO is designed by using a series of shift registers that has the total size of 2,048 bits. This large area limits the configurations for NS to 15, 7, and 4 PEs for word size 8, 16, and 32, respectively. Hence, the exploration of the area/time tradeoff is also limited.

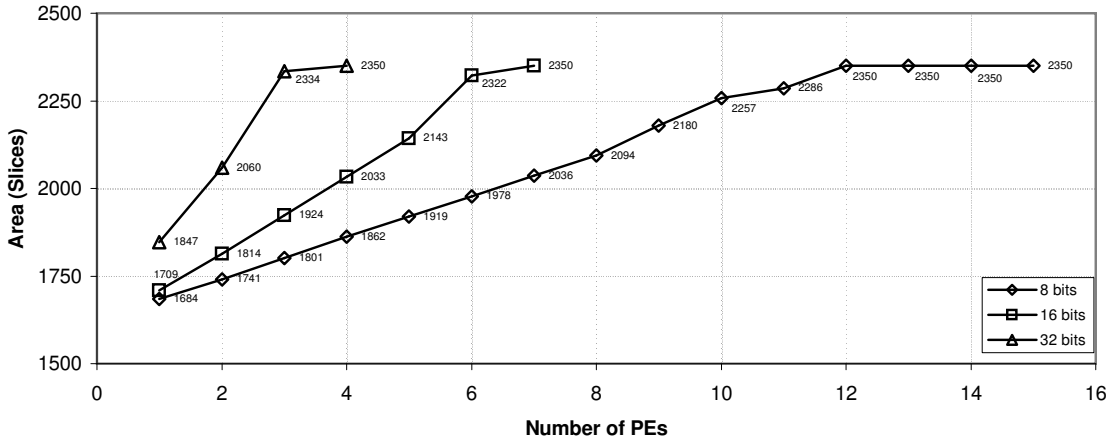


FIGURE 4.1: Area of MMHW for different word size configurations on Xilinx chip

In order to have more number of PEs in the pipeline, the size of the X operand register is configured to be exactly the same as the size of the operand. For example, while working with $w = 8$ and $n = 256$, the number of 8-bit shift registers used inside the X register is only 32, but it was configured to have 256 shift registers; therefore, there are 224 of those that are never used. Trimming off those unused shift registers results in giving up area from the I/O; therefore, the MMHW is able to be configured with more PEs. Note that the size of Y and M registers is set to 2,048 bits via a RAM configuration. Since block RAMs are primitive components of the FPGA chip, changing their size from 256 to 2,048 bits only take few more slices occupied by the FIFO control unit.

Further results of this work from this point are based on a setup where the X register size is set to the same size of the operand.

Figure 4.2 shows the MMHW area for different word size configurations versus the number of PEs. There are four lines in each figure. Each line represents the area for different operand precision. The larger the precision, the smaller the

maximum number of stages. For example, in $w = 8$ configuration, the maximum number of stages is 27 for 256-bit precision, but it is only 11 for 2048-bit precision.

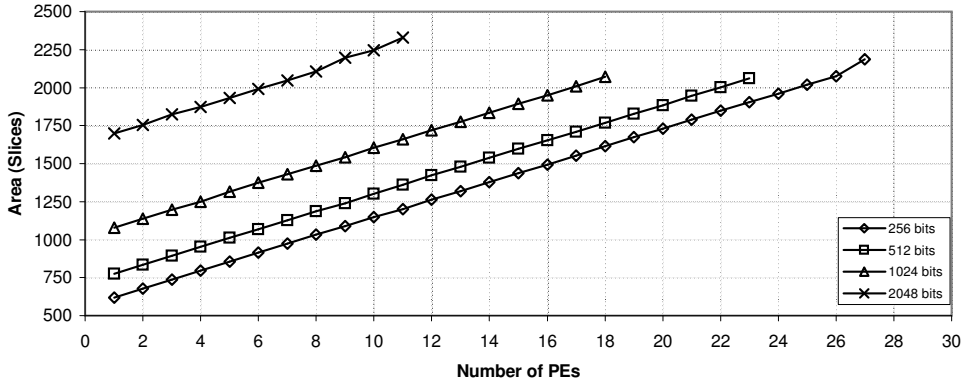
The area where $w = 8$ and $w = 16$ is increased linearly as the number of stages increases as shown in 4.2(a) and 4.2(b). However, the area for $w = 32$ is linear to the point where the number of stages is 4 for operand size less than 2048 bits. As the number of stages increases from that point, the area slowly increases and stays constant at 2350 slices (about 99% of the area).

4.3. Time Estimation for MMHW

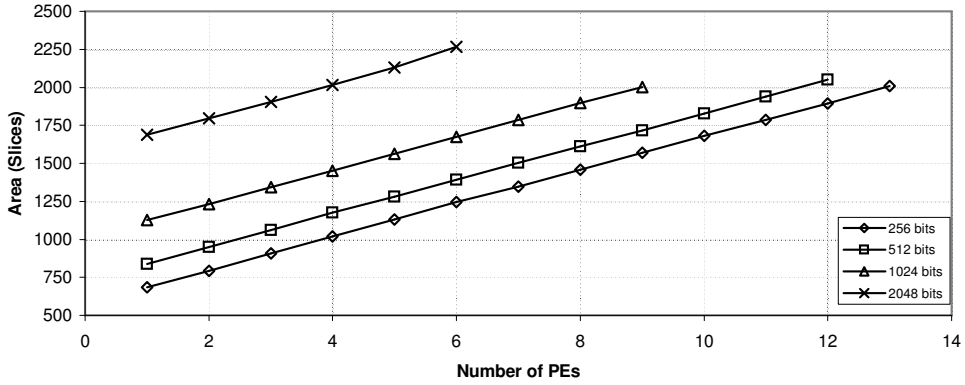
The critical path delay for the MMHW was found to be in the path between the block RAM and one of the address generators (write or read). The example of implementation results of the critical path delay for 1,024-bit precision for different kernel configurations is shown in Table 4.1. The values in the table are very closed to one another because the timing constraint was set and applied to the design during the PAR process. The constraints were set to: 13.75ns for $w = 8$ and 16; 13.9ns for $w = 32$.

Figure 4.3 shows the critical path delay of different word size configurations versus the number of PEs. Each graph has four lines that are the delay of different operand precisions. The delay is quite a constant as the timing constraint was applied when the design was implemented. However, as the area increases, the value of timing constraints has to be increased.

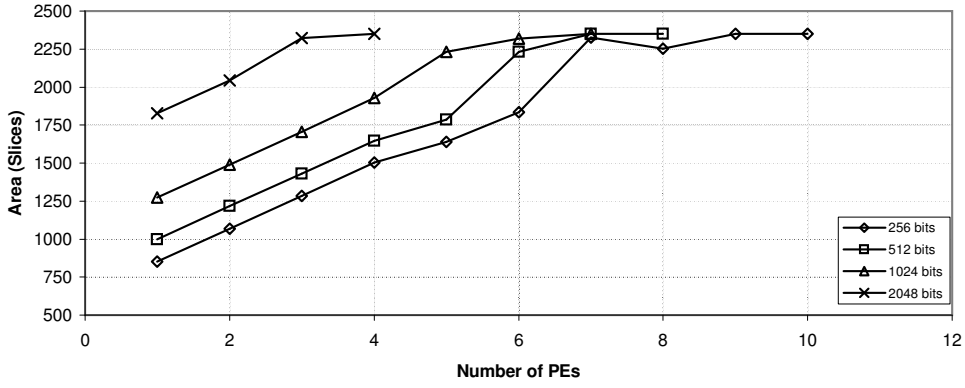
The MMHW can run at clock frequency about 72.5 MHz for $w = 8$ and $w = 16$. However, for $w = 16$ and $n = 256$, when $NS = 13$, the delay is uncharacteristically higher than the other values of NS. For $w = 32$, the MMHW can operate with the clock about 71 MHz. But, for $n = 256$ and $NS = 7$, the clock is about 67 MHz.



(a)



(b)

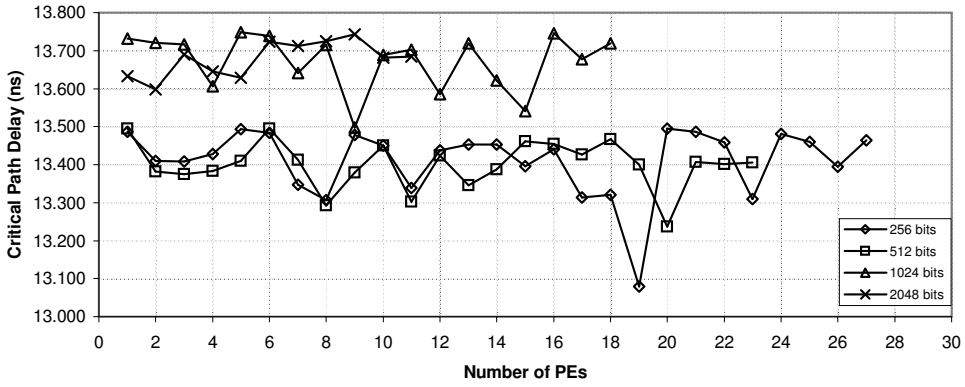


(c)

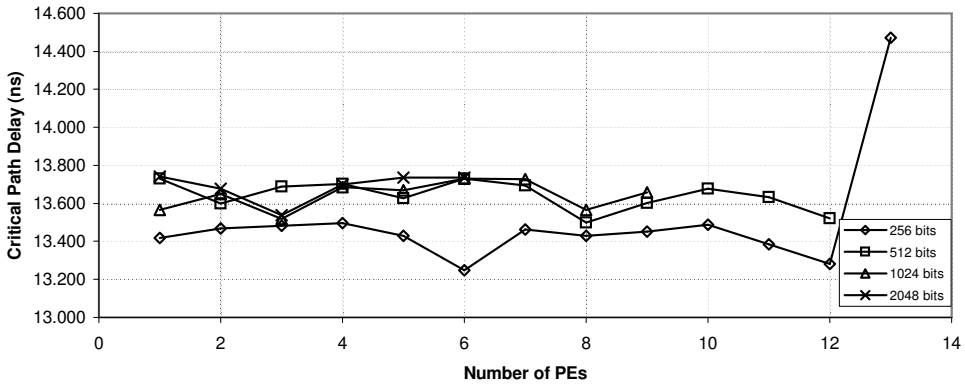
FIGURE 4.2: Area of MMHW where (a) $w = 8$, (b) $w = 16$, (c) $w = 32$

Clock period (ns)			
Operand's precision - n = 1,024 bits			
NS	Word size - w		
	8	16	32
1	13.732	13.565	13.841
2	13.720	13.649	13.894
3	13.717	13.518	13.821
4	13.606	13.686	13.900
5	13.748	13.668	13.776
6	13.739	13.731	13.860
7	13.641	13.728	13.769
8	13.715	13.567	
9	13.498	13.658	
10	13.688		
11	13.702		
12	13.586		
13	13.719		
14	13.621		
15	13.541		
16	13.745		
17	13.677		
18	13.719		

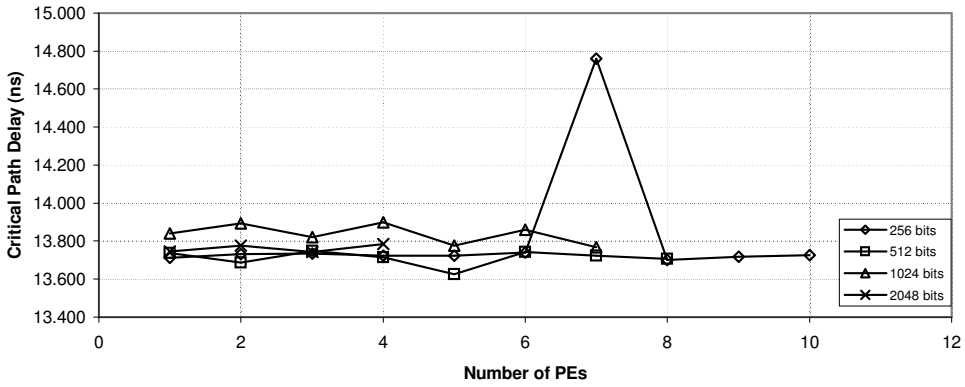
TABLE 4.1: Clock period for different configurations, n = 1,024 bits



(a)



(b)



(c)

FIGURE 4.3: Critical path delay of MMHW where (a) $w = 8$, (b) $w = 16$, (c) $w = 32$

4.4. Total Operation Time

The total operation time T_{op} is the product of the total number of clock cycles to operate MMHW and the clock period. The total number of clock cycles is measured in three parts: *pre-computation*, *computation*, and *post-computation*. In the first part, the number of clock cycles depends on n and w. It includes cycles needed for writing parameters to the control register, loading X, Y, and M operands, and starting the kernel. For writing the control register and starting the kernel, each task takes one cycle.

For loading operands, the number of clock cycles taken depends on how the task is performed. The scenario for this task is to interleave the write cycles into X, Y, and M. That means the 32-bit words of different operands are loaded in three consecutive cycles. After one word of each operand is loaded to its register which takes one clock cycle, it takes $\frac{32}{w}$ cycles to write 32-bit word to its corresponding FIFO. Then, the next 32-bit word can be loaded. The process repeats until all operand bits are written in each FIFO. We assume that the first 32-bit word of X is loaded first, then Y, and M is the last operand. Therefore, it takes $(\frac{32}{w}+1)\cdot\lceil\frac{n}{32}\rceil$ clock cycles between loading the first 32-bit word of X to its register and writing the last w-bit word to its FIFO. For the M operand, the task finishes two cycles after. Hence, the number of clock cycles taken before starting the MMHW (C_1) is:

$$\begin{aligned} C_1 &= C_{\text{write control register}} + \left(\frac{32}{w}+1\right)\left\lceil\frac{n}{32}\right\rceil + 2 + C_{\text{start kernel}} \\ &= \left(\frac{32}{w}+1\right)\left\lceil\frac{n}{32}\right\rceil + 4 \end{aligned}$$

For the second part, the number of clock cycles the kernel takes to finish the computation (C_2) is discussed in [7] and [3]. It is:

$$C_2 = \begin{cases} 2kp + e - 1 & \text{if } (e+1) \leq 2p \\ k(e+1) + 2(p-1) & \text{otherwise.} \end{cases}$$

where $k = \left\lceil \frac{n}{p} \right\rceil$ and $e = \left\lceil \frac{n}{w} \right\rceil$.

The number of clock cycles in the post-computation is the cycles needed for reading the final result from the result register. After the kernel finishes the computation, the 32-bit result register is filling by w-bit words from the S FIFOs in the next clock. This filling period takes $\frac{32}{w}$ cycles, and the 32-bit result word is now ready. When the word is read by the host (one cycle is taken), the filling period of the next 32-bit result word starts. The process repeats until the host reads the last 32-bit result word from the MMHW. Assuming the best case scenario, the number of clock cycle in this part is:

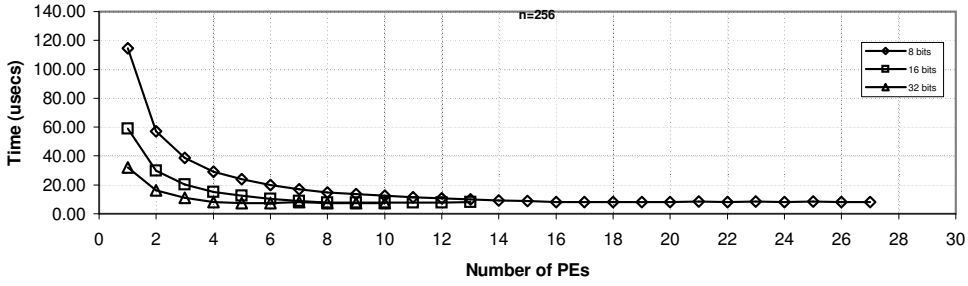
$$C_3 = \left(\frac{32}{w} + 1\right) \left\lceil \frac{n}{32} \right\rceil + 1$$

Hence, the total number of clock cycles to operate MMHW is $C_1 + C_2 + C_3$. The number of clock cycles for pre- and post-computation for each configuration is shown in Table 4.2. Table 4.3 shows the total number of clock cycles, and the total operation time in microsecond when $w = 32$ and $n = 1{,}024$.

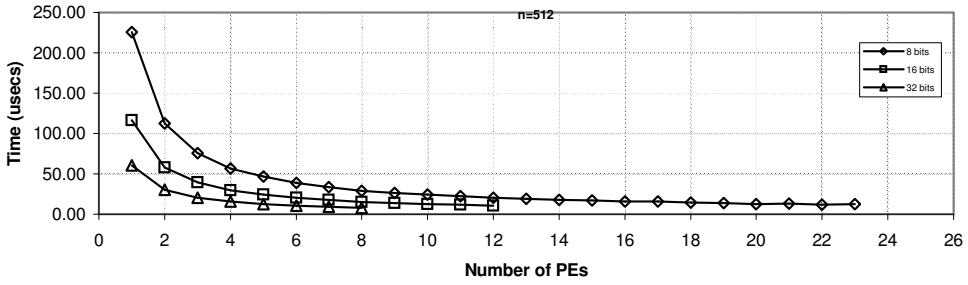
The total operation time T_{op} versus the number of PEs is shown in Figure 4.4. Each figure presents the time in microseconds for one operand precision with different word sizes.

4.5. Optimal Design

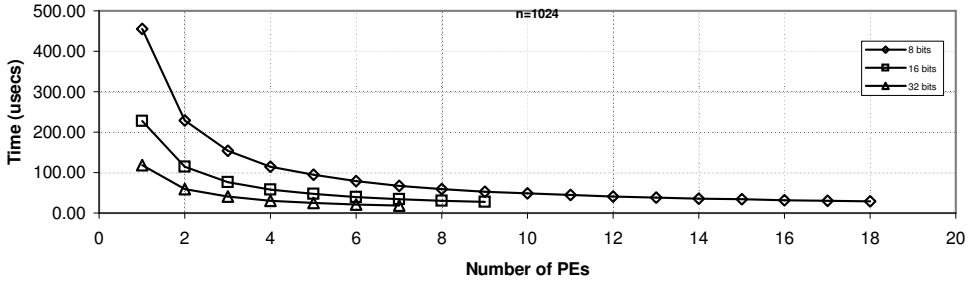
As the number of slices of the chip is limited, it can be considered as a design constraint. Therefore, the goal is to identify the configuration of the MMHW



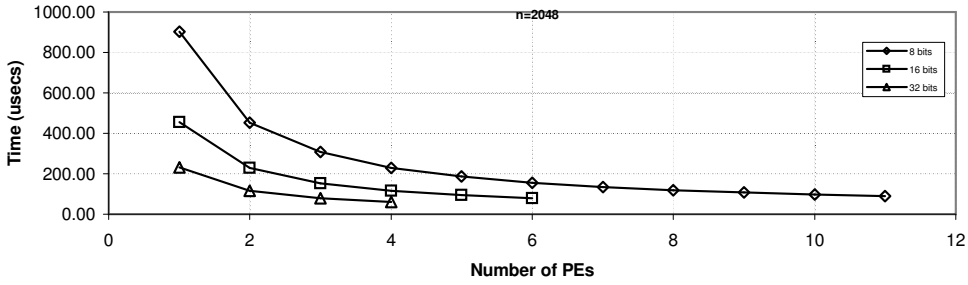
(a)



(b)



(c)



(d)

FIGURE 4.4: Total operation time of MMHW for (a) $n = 256$, (b) $n = 512$, (c) $n = 1024$, (d) $n = 2048$

C_1, C_3	word size (bits)	Operand size (bits)			
		256	512	1024	2048
C_1	8	44	84	164	324
	16	28	52	100	196
	32	20	36	68	132
C_3	8	41	81	161	321
	16	25	49	97	193
	32	17	33	65	129

TABLE 4.2: The number of clock cycles needed in pre- and post-computation for each configuration of w and n

NS	Clock cycles	Operation time (μs)
1	8581	118.77
2	4359	60.56
3	2975	41.12
4	2251	31.29
5	1857	25.58
6	1562	21.65
7	1366	18.81

TABLE 4.3: The total number of clock cycles and total operation time for $w = 32$; n=1,024 on Xilinx Spartan II XC2S200 chip

(w and NS) that satisfies the constraint while yielding the best performance — shortest total operation time.

Since the number of clock cycles for the pre- and post-computation parts are taken into account, the effect of the word size configuration on the total operation time is more significant than that of the computation part alone — only kernel is considered for the operation time. C_1 and C_3 are notably small when $w = 32$. As a result, the configuration for the most efficient design is based on a design with $w = 32$. The optimal designs for each operand size are shown in Table 4.4.

n	w	NS	Area	T_{op} (μs)
256	32	8	2252	7.62
512	32	8	2350	8.59
1024	32	7	2350	18.81
2048	32	4	2350	61.02

TABLE 4.4: The configurations of the optimal design for each operand size

Table 4.5 shows the performance of the MMHW comparing with that of the ARM system (running at 80 MHz) and software algorithms [14]. The configurations for MMHW are selected with $w = 32$ and $NS = 8$ and 7 for 256-bit and 1024-bit precision, respectively. The MMHW is selected to operate at clock period of $20ns$ (50MHz) that yields the total operation time as $11.1\mu s$ and $27.3\mu s$ for operand precision 256 and 1,024 bits, respectively. The speedup is very significant even though the operation time of the MMHW is already included the time to access the I/O, and it operates at a slower clock frequency.

n	MMHW (μs)	Software on ARM system (μs)	Speedup
256	11.1	42.3	3.81
1024	27.3	570	20.88

TABLE 4.5: Comparison of a performance between MMHW and ARM system with 80MHz clock

4.6. Operation Bandwidth

The comparison of the bandwidth of the I/O and the kernel is necessary. This is because the MMHW is operated several times in the execution of the modular exponentiation; thus, the I/O is also accessed for each MMHW operation, i.e. the result read from the I/O is used as operands in the next MMHW operation. Therefore, several cycles are taken to finish reading result and writing that result to the operand registers before the following MMHW operation can start. These cycles are considered to be the I/O bandwidth.

The I/O bandwidth should be small when compared to that of the kernel because, most of the time, the hardware is expected to do the computation rather than spending too much time just for transferring data through the I/O. As the configurations are already selected, the proportion of the number of clock cycles is inspected for those configurations. Figure 4.5 shows a comparison of the number of clock cycles needed to perform each task in the MMHW operation.

As shown in the figure, the proportion of time used by the I/O is around 6 - 11% of the total number clock cycles. That means about 94 - 89% of the total number of clock cycles is used by the hardware to perform the actual computation.

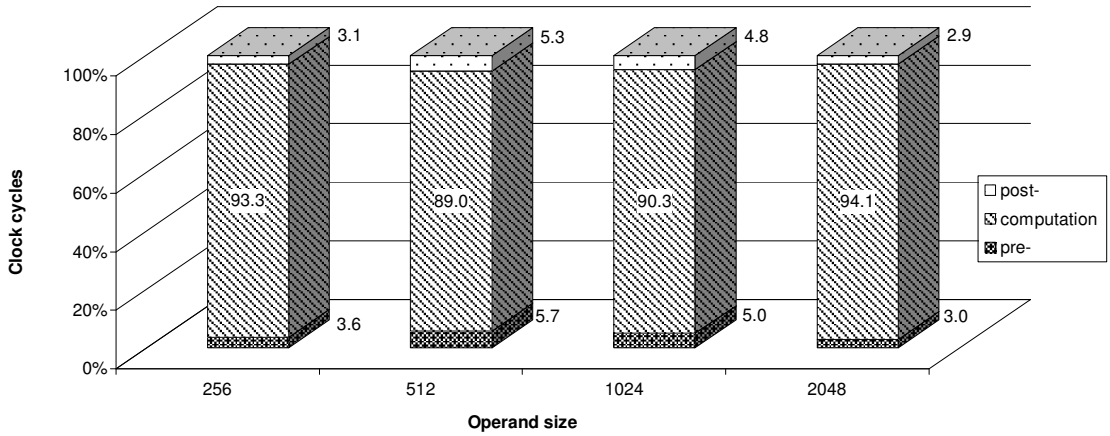


FIGURE 4.5: Clock cycles performing each MMHW task in percentage proportions

5. CONCLUSION AND FUTURE WORK

5.1. Conclusion

An alternative design of the kernel is investigated. The design is a systolic array based on the Montgomery multiplication algorithm. The algorithm for this design needs one more iteration than the kernel in [3] does to finish the computation. Therefore, the multiplication of n-bit operands requires $(n + 1)$ PEs in the full precision array. The only word size that works in the systolic array is 1. Although it is claimed to be able to work with any word size, only simple cases (the number of bits for all operands are the same) were shown in [4]. For the cases where the word size of X is different from that of Y and M, the PE design is not shown in [4], and it was not possible to derive the information from the paper. The systolic array can be scalable when the partitioning method is applied to it. The infrastructure for a scalable architecture is composed of the pipeline with some number of PEs and the temporary memory element used to store the intermediate communications between pipeline cycles.

The architecture of the scalable architecture using the PEs proposed in [4] would be similar to the one in [3], very similar performance, but the inability to make the word size bigger than 1 made us abandon further comparison of this alternative with the design in [3].

The new version of the I/O subsystem is presented in this work. It is scalable and parameterizable. It can be configured to work with the operand size of up to 2048 bits, and the word size is a number that is a power of 2 and less than 32. Regular FIFOs are used to store the intermediate data between pipeline cycles. Rotators needed for some operands are designed using the regular FIFO with a small modification to support three types of operations: writing into the FIFO,

reading from the FIFO, and writing back the information into the FIFO as it is read (rotation).

Several configurations of the MMHW support the analysis of many area and time tradeoffs. The best option to configure the MMHW depends on the operand precision. The design is optimized and implemented for FPGA. In last chapter, the optimal configuration was analyzed for each precision case. The result shows that the biggest word size is the best choice as the I/O bandwidth is smaller.

5.2. Future Work

The I/O subsystem described in Chapter 3 was only tested when connected to the radix-2 version of the kernel. Testing its functionality with other radices is needed to be done since it is intended to be generic.

Moreover, the implementation results presented in Chapter 4 are limited for the large word size as well as the large operand size. It would be interesting to observe the performance of the MMHW with more than 4 PEs in the pipeline when word size is 32 bits and operand size is 2048 bits, for example. The limited results are caused by the area constraint. There are two possible solutions: change the device technology and change the design of the X FIFO. Our target device in this work is Spartan II XC2S200 which has 2352 slices. We can change to other technologies that have more area.

The X FIFO was designed using registers instead of RAMs. Since most FPGA devices including Spartan II family have block RAMs as their primitive components. When block RAMs are used in the design, only few CLBs are occupied by the the control unit; thus, the number of CLBs used reduces significantly. However, designing the X FIFO using RAMs is not straightforward. Because the number of bits of X required by the kernel is an integer number, e.g. 3 bits for

radix-8, the data coming out of the RAM, whose size normally is the power of two, needs to be manipulated before they can be used in the kernel. The circuitry for the bit size manipulation (*funnel*) would be very complicated to design, but the savings in area would be significant.

The funnel design for the case where k divides w can be done by using a shift register with some control. A w-bit word is read from the RAM and stored in the w-bit shift register. When bits of X are requested from the kernel, k bits are shifted out in each read cycle of X. As the shift register gets empty, the next w-bit word is requested from the RAM and loaded into the shift register. The timing is satisfied because k bits of X is requested every two clock cycles; therefore, after the shift register is empty, the next w-bit word is loaded to it in the next cycle. Then, the first k bits from the new word will be ready in the following cycle.

The design for other cases has to be specific for the value of k. The possible solution when $k = 3$ (radix-8 kernel) and $w = 8$ is shown in Figure 5.1.

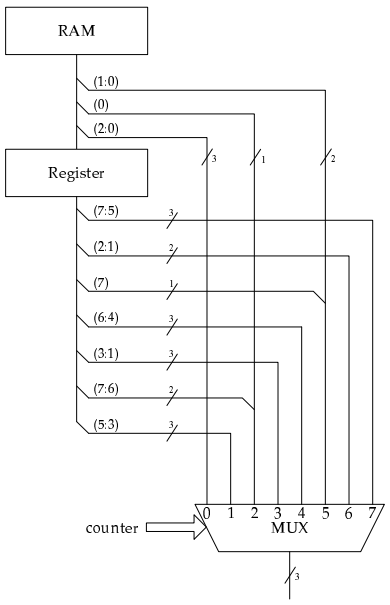


FIGURE 5.1: Funnel design for $k = 3$ and $w = 8$

The diagram in Figure 5.2 illustrates how three 8-bit words from the RAM are divided into eight 3-bit words of X ($X^{(0)} \ldots X^{(7)}$). Most X words are taken from three adjacent bits inside the same RAM word, except for $X^{(2)}$ and $X^{(5)}$. The mechanism that concatenates bits between two RAM words in order to produce $X^{(2)}$, for example, is to use a register holding a content of the first RAM word; then, two MS bits from the register are concatenated with the LS bit from the RAM. Table 5.1 shows an example of nine clock cycles used to produce $X^{(0)} \ldots X^{(7)}$. Note that the read address for the RAM is not changed every cycle; it is increased only when a new word from the RAM is needed. Also, since the RAM has a registered output, the word coming out from the output port shows one cycle after the address is changed.

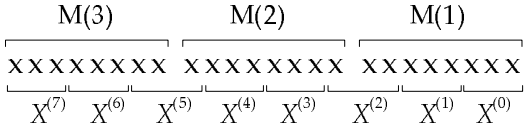


FIGURE 5.2: Division of 3 8-bit words into 8 3-bit words for X

Time	1	2	3	4	5	6	7	8	9
Address	1	1	2	2	2	3	3	3	4
Data(RAM)		M(1)	M(1)	M(2)	M(2)	M(2)	M(3)	M(3)	M(3)
Data(register)			M(1)	M(1)	M(2)	M(2)	M(2)	M(3)	M(3)
$X^{(j)}$		$X^{(0)}$	$X^{(1)}$	$X^{(2)}$	$X^{(3)}$	$X^{(4)}$	$X^{(5)}$	$X^{(6)}$	$X^{(7)}$

where M(i) — Memory content of address i.

TABLE 5.1: Timing of the funnel operation with contents from RAM and a register shown

BIBLIOGRAPHY

1. L. Adleman R.L. Rivest, A. Shamir, "A method for obtaining digital signature and public-key cryptosystems," *Comm. ACM*, vol. 21, no. 2, pp. 120–126, February 1978.

2. P.L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, April 1985.

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

4. T. Matsumoto K. Iwamura and H. Imai, "Montgomery modular-multiplication method and systolic arrays suitable for modular exponentiation," in *Electronics and communications in Japan. Part 3, Fundamental electronic science*, March 1994, vol. 77, pp. 40–51.

5. S. R. Dussé and Jr. B. S. Kaliski, "A cryptographic library for motorola dsp56000," *Advances in Cryptology—EUROCRYPT'90*, Spring-Verlag, pp. 230–244, 1990.

6. D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Transactions on Computers*, vol. C-35, pp. 1–11, January 1986.

7. G. Todorov, "Asic design, implementation, and analysis of a scalable high-radix montgomery multiplier," *MS thesis*, December 2000.

8. H. T. Kung, "Why systolic architectures?," *Computer*, vol. 15, no. 1, pp. 37–46, January 1982.

9. R. Traylor, "Scalable MM - Hardware interfaces," Tech. Rep., ISL Research Group, Oregon State University, January 2001.

10. Xilinx, "Spartan-II 2.5v FPGA family: Complete data sheet," *Product Datasheet, DS001*, September 2003.

11. Xilinx, "Dual-port block memory for Virtex, Virtex-e, Virtex-II, Virtex-II Pro, Spartan-II, and Spartan-IIe v4.0," *LogiCORE Datasheet (http://www.xilinx.com)*, November 2001.

12. M. Khaldoon, "Prototyping of scalable montgomery multiplier using field programmable gate arrays (FPGAs)," *MS thesis, Oregon State University*, July 2002.

13. Xilinx, "Virtex 2.5v Field Programmable Gate Arrays," *Product Datasheet, DS003-1 (v2.5)*, April 2001.

14. T. Acar Ç. K. Koç and Jr. B. S. Kaliski, "Analyzing and comparing montgomery multiplication algorithms," in *IEEE Transactions on Micro*, June 1996, vol. 16, pp. 26–33.

APPENDICES

A. KERNEL-I/O INTERFACE

This appendix shows the updated timing diagram of signals interfacing between the kernel and the I/O subsystem. The original description of signals is presented in [7].

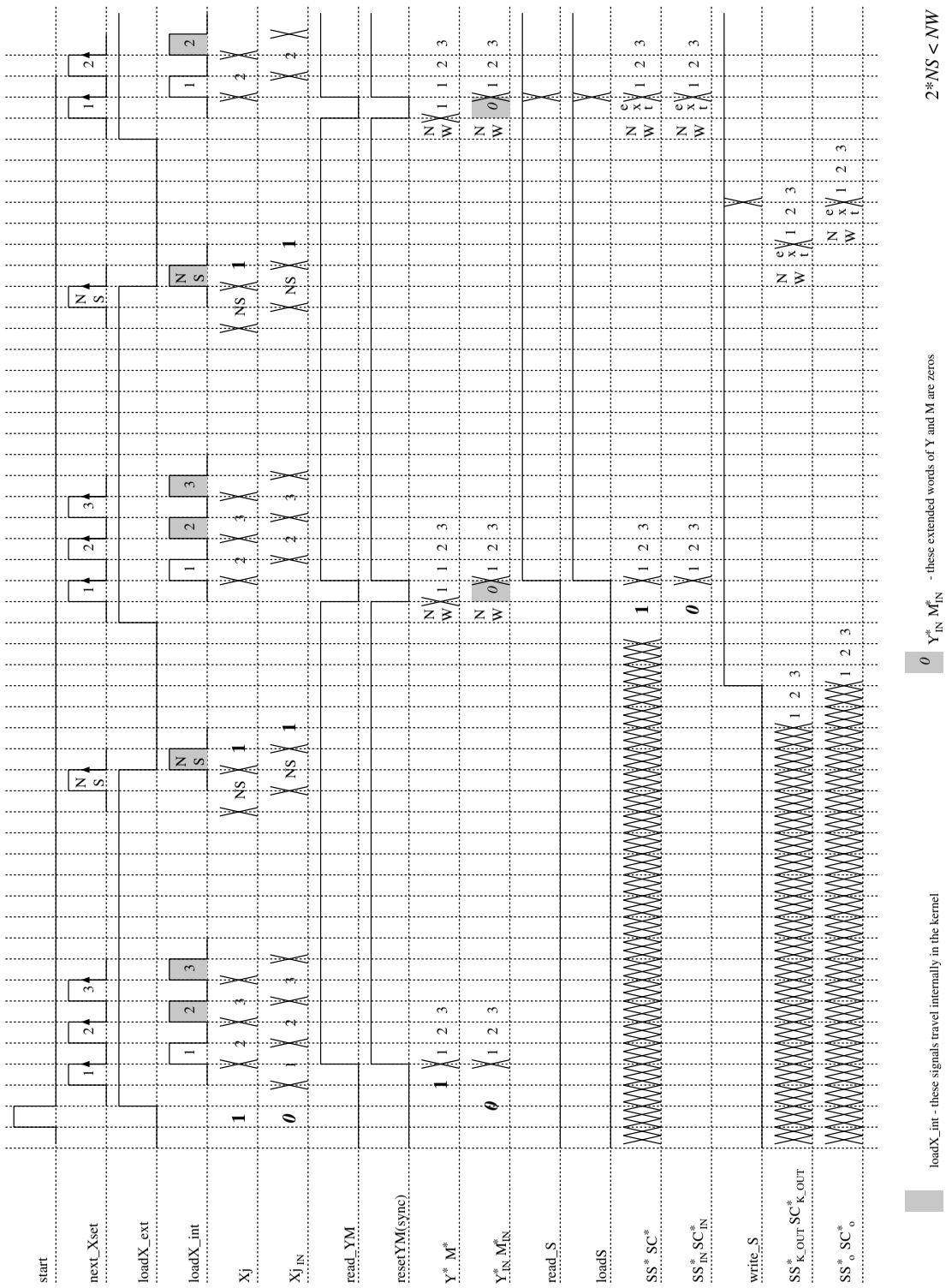


FIGURE A.1: Timing diagram for case $2p < NW$

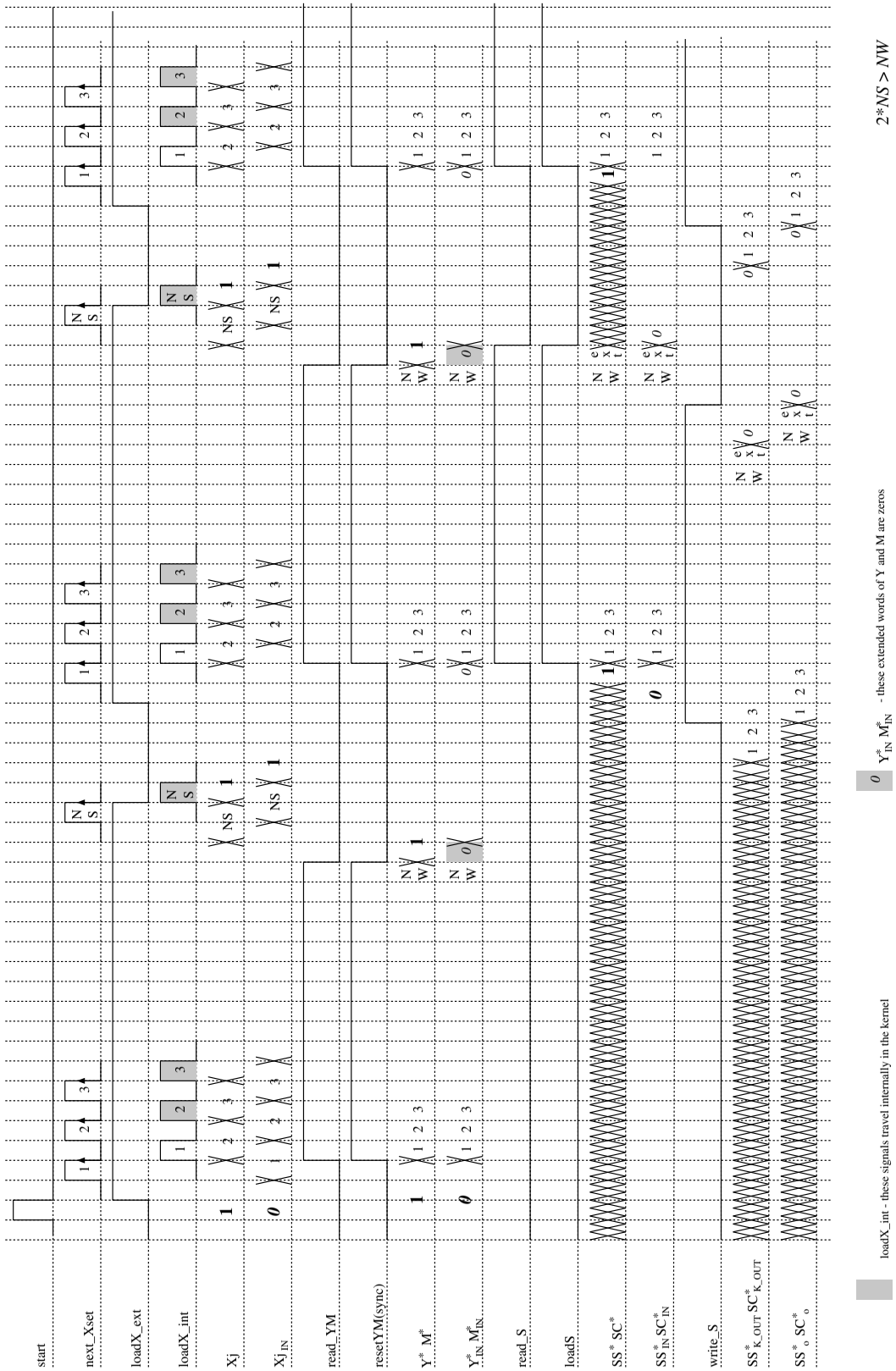


FIGURE A.2: Timing diagram for case $2p > NW$

B. I/O SUBSYSTEM STATE MACHINES DESCRIPTION

Several state machines create signals synchronizing operations of the I/O subsystem. Their state diagrams and functions are described in this section. Note that all state changes happen at the rising edge of the clock.

State machines for the kernel control can be found in [7].

B.1. Top Level Control

The state machine in Figure B.1 generates the *start* signal used to activate the kernel operation.

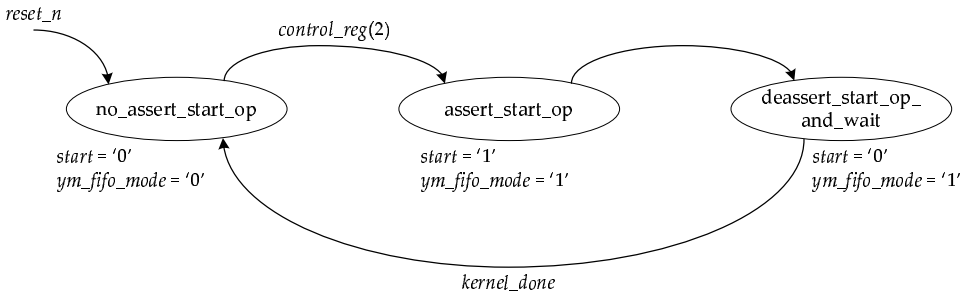


FIGURE B.1: State machine generating *start* signal.

The state machine is initialized in *no_assert_start_op* state. When the host starts the computation (the third bit of the control register is 1), the state is changed to *assert_start_op*. At this state, the *start* signal is asserted to the kernel activating the computation. Also, after the kernel is started, the Y and M FIFOs become rotators (the FIFO mode is 1). The machine is in this state for one cycle;

then, the *deassert_start_op_and_wait* state is reached. At this state, the *start* signal is de-asserted, but the FIFO mode is still 1. The machine stays in this state until the computation in the kernel is done. Then, the it is back to its initial state.

The state machine in Figure B.2 keeps track of the current action of the MMHW. During the initial state *no_operation*, there is no any operation in the kernel. After the kernel starts the operation, the state changes to *op_in_progress*. During this state, the kernel is executing the multiplication. After it finishes, the state changes to *op_done_rd_wait*. In this state, the *op_done* signal is asserted to the seventh bit of the status register. Moreover, w-bit result words are acquired from the S FIFOs (controlled by other state machines) and the 32-bit result word is read from the result register; these two actions repeat several times. the number of w-bit result words read from the S FIFOs is tracked by the counter shown in Figure B.3. When the counter is equal to the number of operand words, the last w-bit word is acquired, and then the *last_word_read* signal is asserted. After the host requests the last 32-bit result word, the machine changes back to its initial state as well as de-asserts the *op_done* signal.

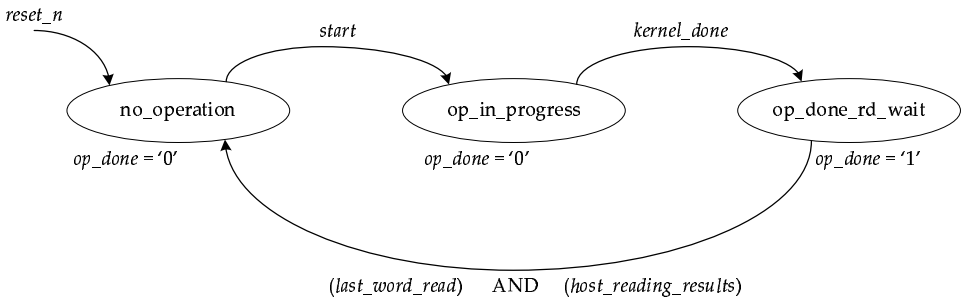


FIGURE B.2: MMHW operation state machine.

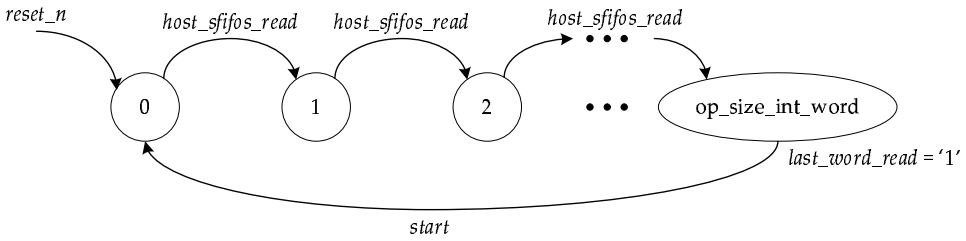


FIGURE B.3: Counter keeping track of the number of result words read.

B.2. 32-to-w Converter Control

The state machine in the 32-to-w converter controls the loading of w-bit word to its associated FIFO. It also keeps track of the number of operand words loaded to the operand FIFO. The state diagram is shown in Figure B.4.

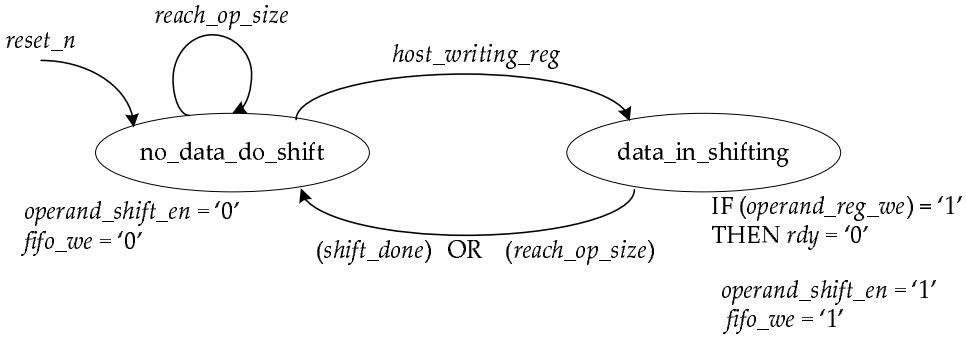


FIGURE B.4: State machine controlling a load operation of w-bit word to the FIFO.

The machine starts in the *no_data_no_shift* state. After the host writes a 32-bit word to the operand register, the machine changes the state to *data_in_shifting*. In this state, the machine enables the FIFO and sends $\frac{32}{w}$ w-bit words to it. If the host writes another 32-bit word to the operand register, the machine de-asserts the

rdy signal indicating the register is not ready to receive a new data and the host must hold the write request. After all 32 bits are sent out, indicated by the signal from the counter keeping track of the number of w-bit word sending to the FIFO, the state is changed back to the initial state. This state change is also activated when all operand words are loaded into the FIFO, the operand size is reached.

Figure B.5 shows the counter keeping track of the number of w-bit operand words loaded to the operand FIFO. Two control signals: *shift_done* and *reach_op_size* are generated based on the value of this counter.

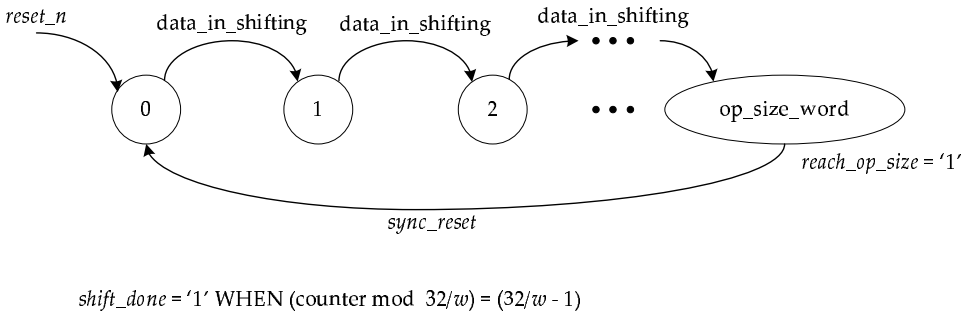


FIGURE B.5: Counter keeping track of the number of w-bit words written to the FIFO.

B.3. w-to-32 Converter Control

There are two state machines inside the w-to-32 converter. Figure B.6 shows the state machine controlling the result word acquiring from the S FIFOs. The initial state is *microp_wait*. When the kernel operation is done, result words are already stored inside the S FIFOs. Thus, the machine changes its state to *rd_word* to start the read from the S FIFOs. The machine changes back to its initial state

when $\frac{32}{w}$ words have been read. This is indicated by the counter keeping track of the number of w-bit words gathered from the S FIFOs shown in Figure B.7. The counter resets itself every $\frac{32}{w}$ counts.

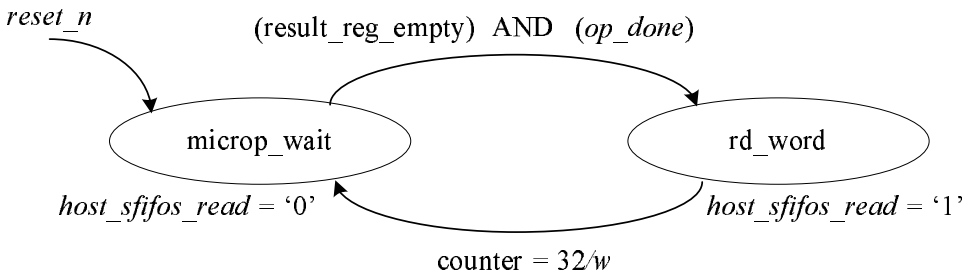


FIGURE B.6: State machine controlling a read operation of w-bit word from the FIFO.

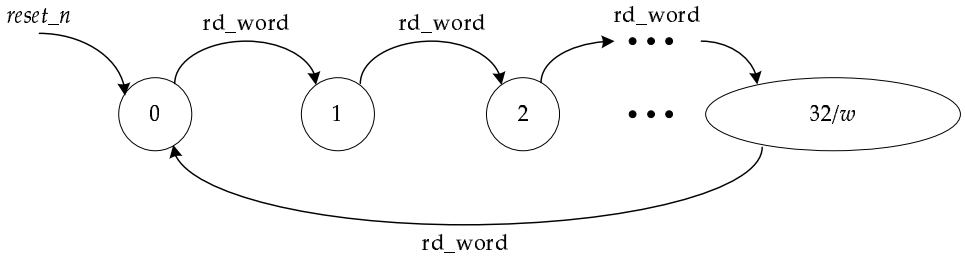


FIGURE B.7: Counter keeping track of the number of w-bit words read from the S FIFO.

Figure B.8 is the state machine keeping track of whether the 32-bit result word is ready. The initial state is *res_reg_empty*. It stays in this state until $\frac{32}{w}$ words fill up the result register. If the host requests the result within this state, the ready signal (*rdy*) is pulled down and prevents the host from reading. After

the result register is ready, the state changes to *res_reg_full*. Within this state, the host is now able to read the result data.

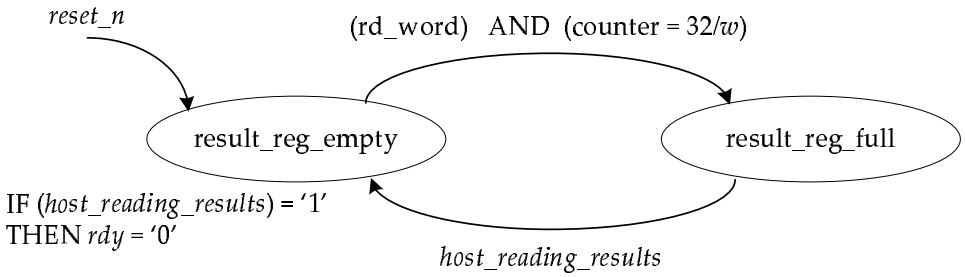


FIGURE B.8: State machine keeping track of the availability of the 32-bit result word.