HARDWARE ACCELERATION OF WINDOW BIG-DIGIT (WBD) MULTIPLICATION FOR EMBEDDED APPLICATIONS

LIM EE WAH

UNIVERSITI SAINS MALAYSIA

2015

HARDWARE ACCELERATION OF WINDOW BIG-DIGIT (WBD) MULTIPLICATION FOR EMBEDDED APPLICATIONS

by

LIM EE WAH

A Dissertation submitted for partial fulfilment of the requirement for the degree of Master of Science (Electronic Systems Design Engineering)

August 2015

ACKNOWLEDGEMENTS

I would like to acknowledge all those who have made this dissertation a success. With great pleasure, I would like to extend my sincere thanks and gratitude to my dissertation adviser Prof. Dr. Rizal for his patience, continuing support and guidance throughout the dissertation.

I extend my heartfelt thanks and gratitude to my fellow Intel colleague Suhaini Annuar and Dr. Vishnu Paramasivam, who gave their valuable time and guidance in making this thesis a success.

I would also like to thanks Shahram Jahani and Hani Al-Mimi on their contribution on inventing the ZOT-Binary and window Big Digits (wBD) algorithm.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ABBREVIATIONS

FPGA Field Programmable Gate Array

SoC System on Chip

AXI Advanced eXtensible Interface

ZOT Zero-One-Two

ECC Elliptic Curve Crypto

BD Big Digit

wBD Window Big Digit

NAF Non Adjacent Form

MLUT Multiplication Look-up Table

PP Partial Product

ALM Adaptive Logic Modules

PERKAKASAN PECUTAN PENDARABAN DIGIT BESAR TETINGKAT (WBD) UNTUK SISTEM TERBENAM

ABSTRAK

Satu algoritma pendaraban baru yang bernama Digit Besar tetingkap (wBD) telah dicadangkan kebelakangan ini. Algoritma ini berasaskan sistem penomboran Digit Besar (BD) dan bersasarkan nombor besar yang beribu-ribu bit. Berat Hamming bagi sistem pernomboran wBD hanya $\frac{n}{4.6}$ berbanding dengan $\frac{n}{2}$ bagi sistem binari, $\frac{n}{3}$ bagi nombor tanpa bersebelahan (NAF) dan $\frac{n}{w+1}$ bagi NAF tetingkap (wNAF). Berat Hamming yang rendah bermaksud bilangan daraban separa yang rendah, yang mana akan mengurangkan bilangan langkah yang diperlukan dalam sesuate operasi pendaraban. Oleh itu, sistem pernomboran wBD dapat mempercepatkan proses pendaraban secara keseluruhan. Algoritma wBD telah dianalisis dan dibandingkan dengan kaedah pendaraban yang lain di tahap algoritma. Namun, belum ada karya yang mengenai pelaksanaan in peringkatan perkakasan diterbit. Untuk membolehkan penggunaan sistem penomboran wBD secara meluas dalam sistem terbenam, sesuatu rekabentuk sistem pengecutan pendarab wBD yang optimum telah diperkenalkan dalam kerja ini. Dalam kajian ini, satu sistem pengecutan pendarab wBD telah direka dengan menggunakan Verilog dan diprototaipkan dalam platform FPGA. Sistem pengecutan ini dilengkapi dengan port AXI dan disepadukan ke dalam sistem SoC yang berasaskan pemproses ARM bagi tujuan perbandingan. Pendaraban 256-bit dengan menggunakan sistem pengecut ini didapati adalah 340 kali ganda lebih cepat daripada pendaraban klasik yang dijalankan melalui teknik perisian. Ini menunjukkan bahawa sistem pendaraban wBD boleh dilaksanakan dengan optimumnya dalam peringkat perkakasan dan dapat mempercepatkan operasi pendaraban yang cuma berdasarkan perisian.

HARDWARE ACCELERATION OF WINDOW BIG-DIGIT (WBD) MULTIPLICATION FOR EMBEDDED APPLICATIONS

ABSTRACT

Window Big-Digit (wBD) is a recently proposed multiplication algorithm. This algorithm relies on Big-Digit (BD) numbering system and is targeting big integer with thousands bits. The hamming weight of wBD representation is only $\frac{n}{4.6}$ compared to $\frac{n}{2}$ for binary, $\frac{n}{3}$ for Non-adjacent form (NAF) and to $\frac{n}{w+1}$ for window-NAF (wNAF). Low hamming weight of multiplicand proportionately reduces the number of immediate partial products, which in turn will reduce the number of steps required in a multiplication function. Hence, wBD number system could be an excellent candidate to speed up overall multiplication process. The wBD algorithm has been analyzed and benchmarked against other multiplication methods in algorithmic level. However, there is no published works regarding hardware implementation of the algorithm yet. In order to enable boarder adoption of the wBD numbering system in resource constrained embedded systems, an optimized hardware accelerator design is introduced in this work. In this study, the hardware implementation of wBD multiplier is designed using Verilog and prototyped in FPGA platform. The accelerator is equipped with AXI interface and integrated into an ARM-based SoC system for benchmarking purpose. The test programs are The hardware-accelerated 256-bits multiplication is found that to be 340 fold faster than pure software implementation of classical multiplication. This shows that wBD algorithm can be optimally implemented in hardware and demonstrates excellent speed gain over pure software implementation.

CHAPTER 1

INTRODUCTION

Window big-digit (wBD) is a new window-based translation method that relied on big-digit (BD) recoding method, which is originally based on ZOT-binary number system. The hamming weight of ZOT-binary numbers is lowest among some popular recoding methods, such as Non-adjacent Form (NAF) and window NAF (wNAF) (Jahani and Samsudin, 2013b). Low hamming weight of multiplicand will proportionately reduces the number of immediate partial products, which in turn will reduce the number of steps required in the multiplication operation. Thus, wBD number system could be an excellent candidate to speed up systems with high occurrences of large integer multiplication, such as Elliptic Curve Crypto (ECC) system.

1.1 Problem Statement

Hani Al-Mimi et al. (Mini et al., 2013) has implemented and tested the wBD multiplication algorithm with MIRACL library, which is a C/C++ based library . But, the manipulation on integers is not native to most of the existing computer architecture which rely on binary numbering structure of 8, 16, 32 or 64 bits. Thus, the pure software implementation may not be a viable solution especially for embedded systems that has limited resource and power constrained. Besides the register size, single threaded processors tend to do the computationally intensive processing in sequential nature and unable to take advantage of the parallelism and optimization that exist in some operations. Therefore, in order to enable boarder adoption of the wBD numbering system in resource constrained embedded systems, an optimized hardware accelerator design will be introduced.

1.2 Research Objective

The main objective of this research is to build an efficient hardware accelerator prototype of wBD-based multiplication. Followings are the sub-objectives:

1. The architectural implementation is targeting efficiency as well as provides significant improvement over equivalent pure software implementation.
2. Prototype of the hardware accelerator will be emulated in FPGA development board.
3. The speed improvement of the hardware accelerator is to be benchmarked against the pure software implementation of classical multiplication under a same platform.

1.3 Thesis Overview

This thesis is organized into five main chapters. Chapter 2 discusses the background and hardware implementation of several multiplication algorithms reported in literature. In addition, topics related to hardware prototyping such as ARM architecture and Hardware Description Language (HDL) are also discussed. In Chapter 3, the development methodology of software and hardware implementation are elaborated. Detailed block diagrams and data flows are also discussed in the chapter. Result and discussion are listed down in Chapter 4. Finally, the conclusion and future works are discussed in Chapter 5.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

Multiplication between large numbers is crucial in wide range of applications, such as cryptography and scientific calculations. Thus, there have been many research efforts in enhancing the multiplication speed (Khan et al., 2015). Generally, there are two ways to speed up the multiplication process, which is using new algorithm or by decoding the operands of the multiplication into different numbering system.

2.2 Multiplication Algorithms

Among the algorithms for large number multiplication are Classical, Karatsuba (Karatsuba and Ofman, 1963), Toom-Cook (Toom, 1963) and Schönhage-Strassen (Schönhage and Strassen, 1971). The first two algorithms are especially popular among others in cryptography applications.

2.2.1 Classical

Classical (also known as Schoolbook) algorithm is a fundamental integer multiplication routine that use the basic operand scanning methods. Figure 2.1 describes the operand scanning form of Classical algorithm (Hankerson et al., 2004).

Input: Integers $A, B \in [0, p-1]$
Output: $C = A \times B$
Set $C[i] \leftarrow 0$ for $0 \leq i \leq t-1$
for i **from** $0 \rightarrow t-1$ **do**
  $(UV) \leftarrow C[i+j] + A[i].B[j] + U$
  $C[i+j] \leftarrow V$
end for
return C

Figure 2.1: Integer Multiplication Algorithm (Operand Scanning form)

In classical algorithm, the operands will be scanned from left to right, and the partial product will be accumulated to generate the final result. Figure 2.2 illustrates a simple classical multiplication of $A = (a_3a_2a_1a_0)_{base}$ (multiplicand) and $B = (b_2b_1b_0)_{base}$ (multiplier). Both numbers are represented in an arbitrary base.

			a_3	a_2	a_1	a_0
$\times$				b_2	b_1	b_0
			a_3b_0	a_3b_0	a_3b_0	a_3b_0
$+$		a_3b_1	a_3b_1	a_3b_1	a_3b_1	
$+$	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
s_6	s_5	s_4	s_3	s_2	s_1	s_0

Figure 2.2: Graphical Representation of Classical Multiplication Algorithm

Most simple digital systems implement the multiplication function according to classical approach and interpreting the operands using radix-2 representation, e.g. binary. The numbering system is straightforward in digital implementation because multiplication with a single bit of binary symbol (0 or 1) can be reduced to zero or replaced with the shifted version of operand.

2.2.2 Karatsuba

The Karatsuba Algorithm was published by Karatsuba and Ofman in 1962 (Karatsuba and Ofman, 1963). It is considered as one of the fastest ways to multiply large numbers. The algorithm adopts the binary splitting scheme over its divide and conquer paradigm. This method requires only four simple additions and three multiplications for each iterations.

To apply the algorithm, both operands are first split into two n/2 digits numbers. In next iteration, the halves are then being split again in half. Since every iteration chop the number by half, the algorithm will be terminated after $t = log_2 n$ steps (Kodali et al., 2014). Figure 2.3 shows the recursive Karatsuba algorithm with assumption that both operands have even number of bits.

Input: Positive integers $A = (a_n, ..., a_0)_r$ and $B = (b_n, ..., b_0)_r$ having $n+1$ base r digits
Output: $C = A \times B$
If $n = 1$ **then return** $C = A \times B$
Split A,B into two equal parts:
$A = A_L \times r^{n/2} + A_R$ and $B = B_L \times r^{n/2} + B_R$
Compute the following:
$d_1 = KA(A_L, B_L); d_0 = KA(A_R, B_R)$ and $d_{0,1} = KA(A_R + A_L, B_R + B_L$
return $C = d_1 \times r^n + (d_{0,1} - d_0 - d_1) \times r^{n/2} + d_0$

Figure 2.3: Recursive Karatsuba Algorithm, $C = KA(A,B)$ (Kodali et al., 2014)

For example, let's examine a case where $n = 2$ and $base = 10$. In this example, the operands could be split into halves, that is, into their decimal digits as in (2.1).

$$\begin{aligned} a &= p \times 10 + q \\ b &= r \times 10 + s \end{aligned} \tag{2.1}$$

Let's say, we have $a = 47$ and $b = 93$ as illustrated in (2.2). Then,

$$\begin{aligned} p &= 4, q = 7 \\ r &= 9, s = 3 \end{aligned} \tag{2.2}$$

The product $a \times b$ can now be written in terms of the digits as in (2.3).

$$\begin{aligned} a \times b &= (p \times 10 + q) \times (r \times 10 + s) \\ &= (p \times r) \times 10^2 + (p \times s + q \times r) \times 10 + q \times s \end{aligned} \tag{2.3}$$

Continuing with the example of $a = 47$ and $b = 93$ as in (2.4).

$$\begin{aligned} 47 \times 93 &= (4 \times 9) \times 10^2 + (4 \times 3 + 7 \times 9) \times 10 + (7 \times 3) \\ &= 4371 \end{aligned} \tag{2.4}$$

By writing the product $a \times b$ of the two-digit numbers a and b as above, it can be shown that the result can be computed using four multiplications of one-digit numbers, followed by additions. This is how the divide-and-conquer technique being adopted in Recursive Karatsuba algorithm. Besides this, the Karatsuba algorithm reduces the operation $a \times b$ from four multiplication steps each iteration to only three steps. These three multiplication steps are used to compute the following three auxiliary products as in (2.5):

$$\begin{aligned} u &= p \times r \\ v &= (q - p) \times (s - r) \\ w &= q \times s \end{aligned} \tag{2.5}$$

The above three auxiliary variables can be related back to the $a \times b$ as been described in (2.6).

$$u+w-v = p\times r+q\times s-(q-p)\times(s-r)$$
$$= p\times s+q\times r \tag{2.6}$$

So, equation (2.7) can then be derived by representing equations (2.6) back into (2.3).

$$a\times b = u\times 10^2+(u+w-v)\times 10+w \tag{2.7}$$

Generally, Karatsuba multiplication algorithm has a better complexity compared to Classical multiplication algorithm (Fang and Li, 2007). However, overhead of Karatsuba multiplication algorithm in lower range numbers (less than 255 digits) has caused designer to combine the Classical and Karatsuba to achieve better overall algorithm efficiency (Jahani and Samsudin, 2013a). However, for software implementation, classical multiplication algorithm generally gives faster result than Karatsuba in numbers that is less than 255 digits due to long and inefficient carry chains on the CPU pipeline. (Fang and Li, 2007).

2.2.3 Toom-Cook Algorithm

Similarly, Toom-Cook algorithm also used the divide-and-conquer paradigm. Same as Karatsuba multiplication, Toom-Cook algorithm also operates by breaking the input numbers into chunks of smaller size. The larger products are then expressed in terms of equations of smaller pieces. Toom-Cook applied recursively on these smaller pieces as well. There is a lot of similarity between this method with the Fast Fourier Transform (FFT), as both are working on the same principles of polynomial multiplication (Crandall and Pomerance, 2006) and (Zuras, 1994).

In Toom-Cook algorithm, the input numbers are split into chunks of given size, and then

the inputs will be written as a polynomial using the chunk size as radix. The polynomials will be evaluated at a set of points and its values will be multiplied together at those points. With the products, the product polynomial can then be determined. Finally, substitution of the radix returns the final answer.

Let's take an example of multiplying 123,456 by 987,654 as in (2.8). In this case, Toom-3 algorithm can be applied. First, each numbers will be split into 2 chunks with 3 digits in length. Then, choose a set of points, for example $\{-1,0,1\}$

$$A \times B = 123,456 \times 987,654 \tag{2.8}$$

By splitting A and B and writing the chunks into polynomials form, we will have:

$$\begin{aligned} A(x) &= 123x + 456 \\ B(x) &= 987x + 654 \end{aligned} \tag{2.9}$$

Then, substitute the x with a set of points to get the values of $A(x), B(x) and A(x) \times B(x)$.

$$\begin{aligned} x = -1 \rightarrow \quad & A(-1) = 333, \quad B(-1) = -333, \quad A(-1) \times B(x) = -110,889 \\ x = 0 \rightarrow \quad & A(0) = 456, \quad B(0) = 654, \quad A(0) \times B(x) = 298,224 \\ x = 1 \rightarrow \quad & A(1) = 579, \quad B(1) = 1641, \quad A(1) \times B(x) = 950,139 \end{aligned}$$

In next step, the polynomial $P(x) = A(x)B(x)$ can be solved based on the values at the three points. The solution is of the form as in (2.10):

$$P(x) = p_2 x^2 + p_1 x + p_0 \tag{2.10}$$

The values of x can be substituted into the (2.10) with the points $\{-1,0,1\}$ to get a set of simultaneous equations that are linear in the unknowns p_i as in (2.11):

$$P(-1) = 1 * p_2 - 1 * p_1 + p_0 = -110,889$$

$$P(0) = 0 * p_2 + 0 * p_0 + p_0 = 298,224$$

$$P(1) = 1 * p_2 + 1 * p_1 + p_0 = 950,139 \tag{2.11}$$

By solving following matrix as in (2.12), the values for the unknowns p_i as shown in (2.13) can be obtained.

$$\begin{pmatrix} 1 & -1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} p_2 \\ p_1 \\ p_0 \end{pmatrix} = \begin{pmatrix} -110,889 \\ 298,224 \\ 950,139 \end{pmatrix} \tag{2.12}$$

$$p_2 = 121,401$$

$$p_1 = 520,514$$

$$p_0 = 298,224 \tag{2.13}$$

With values in equation (2.13), the final multiplication result can be calculated as shown in Figure 2.4.

$$\begin{array}{rr} & \textbf{121,401}\textit{,000,000} \\ + & \textbf{520,514}\textit{,000} \\ + & \textbf{298,224} \\ \hline & \textbf{121,981,812,224} \\ \hline \end{array}$$

Figure 2.4: Final multiplication based on values in Toom3 equations

Steps such as evaluating the values and solving the equation seems difficult, but it turns out

to be relatively straight forward for a computer to execute. Toom-Cook algorithm only requires some simple operations such as addition, subtraction, shifts, division by small constants and multiplication with one-third of the original inputs.

2.3 Numbering Systems

Besides optimizing the multiplication algorithm, the other solution to improve the arithmetic function is by representing the operands using other numbering system. In this regards, several numbering systems have been introduced.

2.3.1 Booth

Booth proposed an algorithm to speed up the Classical multiplication computation in 1951 (Booth, 1951). The implementation of Booth algorithm requires repeatedly addition of either A or S to the partial product P, which the A and S can be calculated from steps mentioned. The resultant P will then be rightward shifted. Also, note that the addition of A or S with P can be done by using ordinary unsigned binary addition.

As described are the steps for Booth algorithm (Booth, 1951). Let m be the multiplicand and r be the multiplier; and x and y represent the number of bits in m and r respectively.

1. First the values of A, S and initial value of P have to be determined. These numbers should have a length equal to $(x+y+1)$ of bits.

    - **A:** Pad the most significant (leftmost) bits with the value of m. Fill the remaining $(y+1)$ bits with zeros.
    - **S:** Pad the most significant bits with the value of âĹŠm in two's complement notation. Fill the remaining $(y+1)$ bits with zeros.

- **P:** Pad the most significant x bits with zeros. Then append the value of r to the right of this. Pad the least significant (rightmost) bit with a zero.

2. Then, the two least significant (rightmost) bits of P are to be determined.
   - If they equal to 01, then find the value of $P+A$ and ignore any overflow.
   - If they equal to 10, then find the value of $P+S$ and ignore any overflow.
   - If they equal to 00, do nothing and reuse the same P directly in the next iteration.
   - If they equal to 11, do nothing and reuse the same P directly in the next iteration.
3. Arithmetically shift the value calculated in the 2nd step by a single bit to the right. Let set P to this new value.
4. Repeat steps 2 and 3 for y times.
5. Finally, drop the least significant bit from P. This is the product of m and r.

As an example, to find the multiplication result of 3 and -4, we will have following variables to start with: m = 3, r = -4, x = 4 and y = 4 as in (2.14).

$$m = 0011_2, -m = 1101_2, r = 1100_2 \tag{2.14}$$

Then, the A, S and P can be calculated by applying the algorithm listed in step 1 as in 2.15,

$$\begin{aligned} A &= 001100000_2 \\ S &= 110100000_2 \\ P &= 000011000_2 \end{aligned} \tag{2.15}$$

Now, start the Booth multiplication by performing following loop for four iterations:

1. $P = 000011000_2$. The last two bits are 00_2.