國立臺灣大學電機資訊學院電機工程學系

碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

ARM 處理器上準梅森質數體的快速運算程式碼產生器

Code Generation for Fast Pseudo-Mersenne Prime Field Arithmetic on ARM Processors

楊上逸

Shang-Yi Yang

指導教授：鄭振牟 博士

Advisor: Chen-Mou Cheng, Ph.D.

中華民國 102 年 7 月

July, 2013

誌謝

我覺得研究生的生涯就像一本哲學書，除了一直反覆思考研究項目之外，也曾體會這短短的兩年研究生生活，我的研究項目與生活從不曾分開過。有時候被一項研究困住，完全解不開；有時候會被這毫無生趣的研究生生活逼著去思考到底研究的意義在哪裡？會迷失反向，會失去自我，然而，總有一縷陽光會眷顧準備好的人。他們出現在我最困難的時候，給我方向，好讓我繼續尋找答案。這些人有我最尊敬的指導老師，也有相關領域的中研院教授，當然不能少了研究室的夥伴，還有我的家人朋友們。

飲其流時思其源，成吾學時念吾師。首先，我衷心感謝我的指導老師鄭振牟教授。鄭振牟老師對教學十分認真，從不馬虎。感謝您除了教授專業的知識給我之外，同時也在我困惑的時候給予我許多提示與幫助。再者，我要感謝的是中研院楊柏因老師，在我研究項目中給予許多寶貴的意見與見解。衷心感謝兩位老師的協助與指導，使我獲益良多，順利完成論文，讓我感受到：師者，所以傳道、授業、解惑也。

接下來要感謝的是研究室的朋友們：大明星、郭大師、瑋哥。感謝你們在我這兩年的研究生生涯中帶給我無數的快樂與關懷。特別是大明星，每次都偷吃我的雞塊，讓我餓到不知所措；還有郭大師，當你找我去登山解壓的時候，我總是婉拒，因爲體力眞沒你那麼好；瑋哥，感謝你總是略帶幽默的對話，讓我的內心世界充滿歡笑，還有其他的同仁，像是榜哥、安安，由於篇幅的關係，就不一一介紹了！衷心地感謝你們，讓我在這兩年研究光陰裡面充滿了無數地快樂與美好。

感謝家人的精神支持與鼓勵！每當回家時，總是最幸福的時候，除了享受每天睡到自然醒之外，還可以吃到母親精湛的廚藝美食，還有父親沉默無聲的獨特關懷，兩個妹妹經常的問候與鼓勵。讓我感到家人的支持與關愛是這次順利完成論文的重要關鍵。還要感謝好朋友Joseph，除了經常帶我去吃美食紓壓之外，還經常把歡笑與快樂帶進我的世界裡。

最後，要感謝的是打開這本論文的您，希望可以帶領您走進一個精彩而又有趣的世界。衷心感謝您們！

摘要

近期高速密碼學研究中，往往透過電腦指令的排列組合來提升運算效率，但如果少了自動化工具，則需要耗費相當大的人力。

使用我們提出的工具，只需要準梅森質數作爲輸入，就能透過窮舉找出在ARM11上最高效率的模乘法程式。窮舉的參數包含大數的表示方示及程式碼產生器參數，而提出的模乘法演算法則混合了乘法與模餘兩部份，特別適合提升準梅森質數體上的計算效率。

使用提出的演算法，自動產生出的高質量程式碼運行時間較GCC編譯器的結果快16.4%，且爲GMP模乘法的4至8倍。

關鍵字: 多精度乘法、準梅森質數、快速密碼學、*ARM11*。

Abstract

Recent research on high-speed cryptography has been striving for performance by twiddling with instructions, but without an automated tool, writing fast software takes much precious labor effort.

We present a tool with a simple interface for crypto developers to generate fast modular multiplication routines in a few keystrokes: you provide the prime as the modulus and it produces several candidate results or enumerates them all for benchmark. Specifically, we automatized the choice of number representation and the code generation for multiplication modulo a pseudo-Mesenne prime on ARM11, using the proposed convolved multiplication method, which interleaves multiplication and modular reduction.

The high-quality code generated runs up to 16.4% faster than the convolved multiplication compiled by defacto-standard compilers such as GCC, and is 4 to 8 times faster than the GMP modular multiplication.

Keywords: *multi-precision multiplication, modular multiplication, pseudo-Mersenne primes, high-speed cryptography, ARM11, convolved multiplication.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multi-precision arithmetic is essential in public-key cryptography, such as RSA and elliptic Curve Cryptography (ECC); even some hash functions and message authentication codes are based on multi-precision modular arithmetic (MASH-1, Poly1305 [Ber05]). They all require efficient computations in $\mathbf{F}_p$.

Why is software performance so important in cryptography? Imagine a cloud server cluster that encrypts transmitted data. Most of the time, these computers do nothing but perform cryptographic operations. Even a 10% speedup would save 10% less power and time, and leaves more processor resources for other operations.

1.1 Motivation

Recent research on high-speed cryptography has been striving for performance by exploiting hardware limitations and twiddling with instructions, but the precious labor effort is hardly regarded. Without an automated tool for producing and autotuning code for best performance, implementers would struggle with assembly to come up with programs that outperform a compiler-optimized C counterpart.

Although the literature exhibits a number of multiplication and reduction techniques, translating each method into assembly does not automate the overall process. A good choice of number representation, a proper application of algorithms and a thorough knowledge of variant architectural irregularities and specialties, all of the three intervene the process, and make it even more complicated.

In the thesis, we investigate the viability of automation by starting with one of the most fundamental cryptographic primitives: multi-precision modular multiplication. Specifically, we automatized the choice of number representation and the code generation for multiplication modulo a pseudo-Mesenne prime. We chose to benchmark on the ARM11 processor family because these RISC processors are less complex than other CISC variants as x86, and sufficiently serve as our primary experimental target.

1.2 Problem Statement

We formally define the problem as follows.

Problem Given $p = 2^m - k$, a pseudo-Mersenne prime with small positive k, produce efficient routines for the function $f(x, y) = xy$, where $x, y \in \mathbf{F}_p$.

By efficient, we mean the lower the number of clock cycles on the platform in question, the better.

1.3 Contributions

Our contributions include:

- We present a tool with a simple interface for crypto developers to generate fast modular multiplication routines in a few keystrokes: you provide the prime as the modulus and it throws out several candidate results or enumerates them all.

- We extend the principle of the hybrid multiplication method to put multi-precision multiplication and modulo reduction together, and incorporate them to work with the mixed-radix representation.

- Even without auto-tuning and enumerating, the tool produces high-quality code comparable to the fastest one enumerated. The produced code runs up to 16.4% faster than the convolved multiplication compiled by defacto-standard compilers such as gcc, and is 4 to 8 times faster than the GMP modular multiplication.

國立臺灣大學
愛國・愛學・勤品敦

Chapter 2

Preliminaries

This section summarizes necessary information to understand the approach we take. We first introduce the pseudo-Mersenne primes. Next, we analyze two different representations for large integers. Finally, we sketch common multiplication techniques.

2.1 Pseudo-Mersenne Primes

Definition 1. *A pseudo-Mersenne prime is a prime of the form* $p = 2^m - k$ *with* $k \ll 2^m$*.*

Some such primes attract attention because their modular reduction requires only additions and multiplications by k. In FIPS 186-2 [Nat00], NIST recommended prime fields for elliptic curves with pseudo-Mersenne moduli:

$$\begin{aligned}
p_{192} &= 2^{192} - 2^{64} - 1 \\
p_{224} &= 2^{224} - 2^{96} + 1 \\
p_{256} &= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \\
p_{384} &= 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 \\
p_{521} &= 2^{521} - 1.
\end{aligned}$$

Except p_{512}, the form as the sum or difference of powers of 2^{32} makes fast reduction routines possible on 32-bit processors.

In the thesis, we confine the primes with rather small k, for example $k < 32$. Several cryptography schemes and libraries intended at high speed, such as Ed25519 [BDL+12] and Poly1305 [Ber05] in the library NaCl [HS13], has chosen primes $2^{255} - 19$ and $2^{130} - 5$, for their simplicity of modular reduction.

2.2 Radix Representations

Let W be the word size of the processor (e.g. 8, 16, 32 or 64 bits) and p the m-bit prime modulo for $\mathbf{F}_p$.

Implementations usually decompose an m-bit number x as n unsigned integers $(x_0, x_1, \ldots, x_{n-1})$ with $x = \sum_{i=0}^{n-1} x_i 2^{Wi}$, where $n = \lceil m/W \rceil$ and each $x_i \in [0, 2^W - 1]$.

Definition 2. *The unique representation of x as a sum of multiples of powers of 2^W, as given above, is called the radix-2^W representation of x.*

Algorithm 1 shows the standard way to add two numbers on $\mathbf{F}_p$. Although on most processors with the add-with-carry instruction, overflow checks in the loop can be implicit, these consecutive word-wide additions still hinder possible parallelization. Moreover, a costly comparison to p and a reduction modulo p may occur unpredictably within each addition, which is invulnerable to timing attacks.

Algorithm 1 Addition in $\mathbf{F}_p$ using radix-2^W representation.

```
Require: $a, b \in [0, \ldots, 2^{Wn} - 1]$
Ensure: $c = a + b \mod p$
  $(\varepsilon, c_0) \leftarrow a_0 + b_0$                      ▷ $\varepsilon$ is the carry bit.
  for $i = 1 \to n - 1$ do
      $(\varepsilon, c_i) \leftarrow a_i + b_i + \epsilon$
  if $\varepsilon = 1$ or $c \geq p$ then                ▷ Reduction modulo $p$.
      $c \leftarrow c - p$
  return c
```

Fortunately, the lazy-reduction technique gets rid of all these downfalls. By trading off the number of limbs and not using each word to the full extent, these excessive reductions can be shrunk into one and postponed until after several add operations. The carry chain can also be eliminated to empower parallelism.

Definition 3. *Let B be an integer smaller than W. We represent x as an $n = \lceil m/B \rceil$-tuple $(x_0, x_1, \ldots, x_{n-1})$ with each $x_i \in [0, 2^W - 1]$ and $x = \sum_{i=0}^{n-1} x_i 2^{Bi}$. This non-unqiue representation of x is called the radix-2^B (redundant) representation of x.*

This a *non*-unique redundant representation does not enforce the bounds $x_i \leq 2^B - 1$ on each limbs. An integer x is *fully-reduced* if each x_i is below 2^B.[1] For example, on a 32-bit processor, a 5-limb radix-2^{32} representation splits a 130-bit number into five 32-bit parts. But with a more cost-effective 5-limb radix-2^{26} representation, limb-wisely adding together 64 fully-reduced integers still conform to the redundant representation since $64 \times (2^{26} - 1) < 2^{32}$. Only after then a reduction is needed to allow another addition.

2.3 Multi-precision Multiplication Techniques

In the following subsections, we sketch common multiplication techniques: the row-wise method, the column-wise method and the hybrid method. They all assumes the radix-2^W representation and need to cope with carry propagation. Their main ideas differ in how they cache operands to use the register file efficiently. In order words, they differs in how they reduce the number of reduplicate loads and stores.

We adopted the rhombus form [HW11] to illustrate the structure of each method. Also we rule out complex methods, such as the Karatsuba algorithm or FFT methods, as they generally lead to high overhead on resource-limited processors.

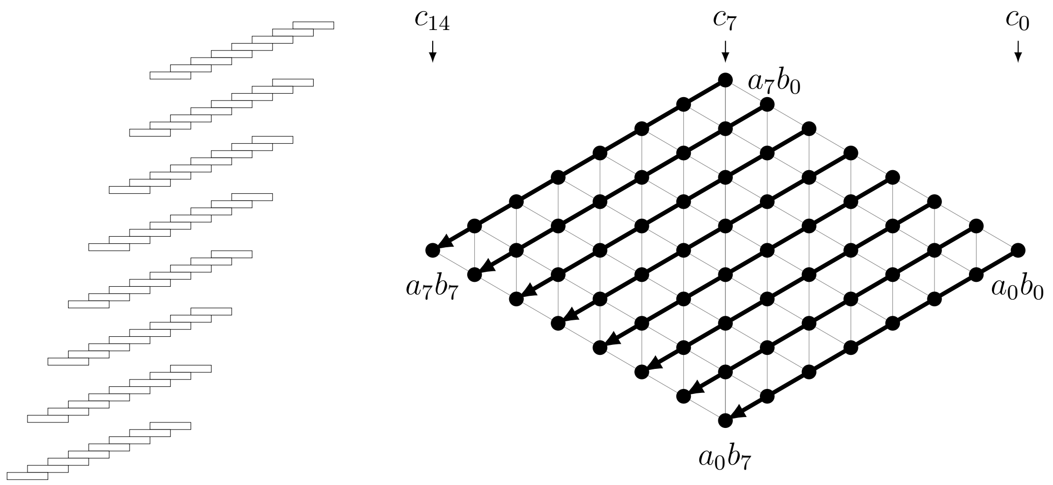


Figure 2.1: Row-wise method for 8-word numbers. Each point a partial product $a_i \times b_j$.

2.3.1 Row-Wise Method

Also called the schoolbook or the operand-scanning method, the row-wise method corresponds to how a primary school pupil would multiply a and b — keep a word a_i, loop through b and accumulate $a_i \times b_j$ to c_{i+j} (Figure 2.1). Whenever a partial product $a_i \times b_j$ overflows, the result is carried into the next partial product c_{i+j+1}.

The pitfall of the row-wise method is that each word of b is reloaded every time as the outer loop walks through a. This is unclever if the processor owns a large bank of registers to store these value for later use. We can do better.

2.3.2 Column-Wise Method

The column-wise method is also called the Comba [Com90] or the product-scan method because it walks through each accumulator and calculates all partial products in the same column (Figure. 2.2). The register usage is immediately visible. In each column a bunch of 2-word partial products are added together and carried to a third word, so totally 3 words of accumulator are required, but only 2 of them are stored.

[1]We do not always reduce x_i below 2^B. In this case, we say x is *reduced*, depending on the context.

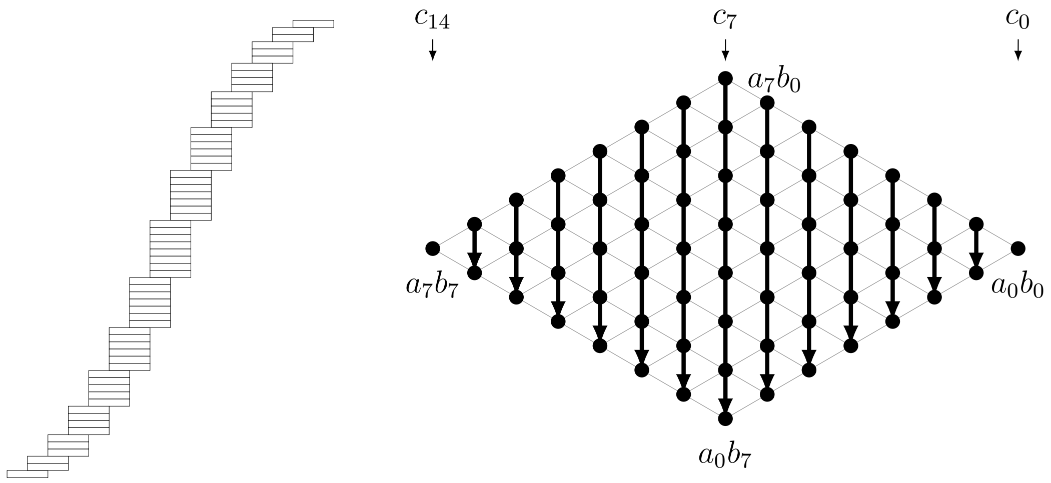
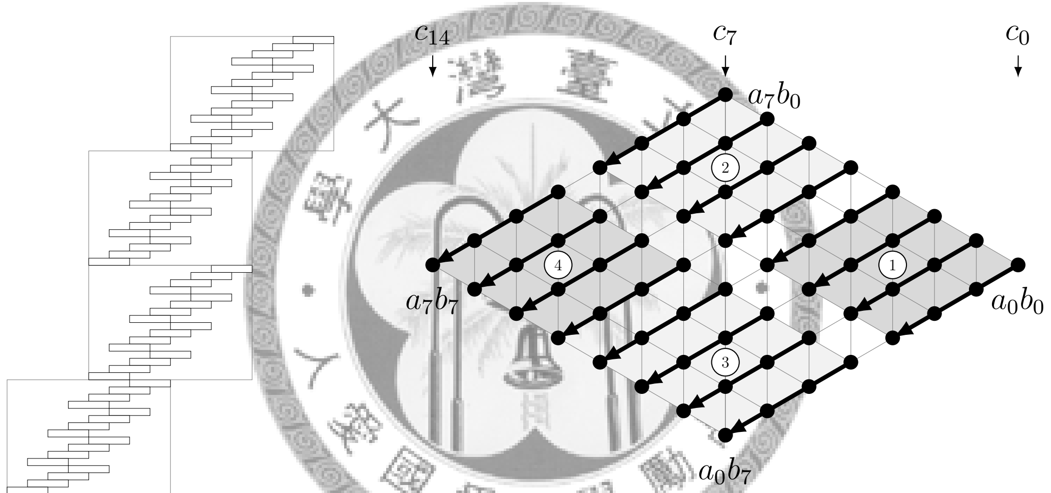


Figure 2.2: Column-wise method for 8-word numbers.



國立臺灣大學 敦品勵學 愛國愛人

Figure 2.3: Hybrid method for 8-word numbers. $(d = 4)$

2.3.3 Hybrid Method

The hybrid method [GPW$^+$04] combines the advantages of both the row-wise and column-wise methods. The basic idea is simple — perform the multiplication as if the word size is actually Wd and do the inner large partial product, which consists of d^2 word-wise multiplications, using the row-wise method (Figure 2.3). Often we choose a proper d so that in each block, the $2d + 1$ accumulators are all maintained in the register file. We will later adopt the idea of putting all accumulators in the register file when discussing how we extend the hybrid method to deal with the convolved structure.

Chapter 3

Convolved Multiplication

3.1 Two Examples

Usually, modular reduction comes only after a full multiplication; they do not interleave to maximize throughput. We adopted the ideas from Bernstein, who first used floating points to speed up operations modulo $2^{127} - 1$ [Ber00] and modulo $2^{130} - 5$ [Ber05], and later adopted by Schwabe to accelerate arithmetic modulo $2^{255} - 19$ on a variety of platforms [Sch11, BDL+12].

The first subsection exemplifies how the fix-sized redundant representation on $\mathbf{F}_{2^{130}-5}$ puts multiplication and reduction together; the second one shows that it is not a cure-all — the mixed representation outworks on $\mathbf{F}_{2^{127}-25}$. This is because the bit length of the prime $2^{127} - 25$ is not a multiple of any integer, but 130 is a multiple of 26.

The following examples assumes a 32-bit processor with a 32×32 to 64-bit multiplication instruction. The methods described in the section do not necessarily require prime moduli, but the examples are given with prime moduli because these are practical in cryptography. After each example we provides several points to consider on choosing limb sizes so it facilitates even more efficient calculations.

3.1.1 First Example

The field operations on $\mathbf{F}_{2^{130}-5}$ profits from the radix-2^{26} representation. Multiplying without reduction two 5-limb numbers a and b yields a wedge-shaped 9-limb c, with each c_i a 64-bit word:

$$\begin{aligned}
c_0 &= a_0b_0 \\
c_1 &= a_0b_1 \ + a_1b_0 \\
c_2 &= a_0b_2 \ + a_1b_1 + a_2b_0 \\
c_3 &= a_0b_3 \ + a_1b_2 + a_2b_1 + a_3b_0 \\
c_4 &= a_0b_4 \ + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 \\
c_5 &= a_1b_4 \ + a_2b_3 + a_3b_2 + a_4b_1 \\
c_6 &= a_2b_4 \ + a_3b_3 + a_4b_2 \\
c_7 &= a_3b_4 \ + a_4b_3 \\
c_8 &= a_4b_4.
\end{aligned}$$

We then eliminate the coefficients $c_5, \ldots, c_8$ by reduction modulo $p = 2^{130} - 5$, which suggests $2^{130} \equiv 5 \mod p$. Therefore, $5c_5$ is added to c_0, $5c_6$ is added to c_1, and so on, yielding the *convolved structure* before doing the carry chain:

$$\begin{aligned}
c_0 &= a_0b_0 + 5a_1b_4 + 5a_2b_3 + 5a_3b_2 + 5a_4b_1 \\
c_1 &= a_0b_1 + \ \ a_1b_0 + 5a_2b_4 + 5a_3b_3 + 5a_4b_2 \\
c_2 &= a_0b_2 + \ \ a_1b_1 + \ \ a_2b_0 + 5a_3b_4 + 5a_4b_3 \\
c_3 &= a_0b_3 + \ \ a_1b_2 + \ \ a_2b_1 + \ \ a_3b_0 + 5a_4b_4 \\
c_4 &= a_0b_4 + \ \ a_1b_3 + \ \ a_2b_2 + \ \ a_3b_1 + \ \ a_4b_0
\end{aligned}$$

Finally, rewriting $5a_1b_4$ as $a_1 \times 5b_4$ and precalculating $5b_4$ avoids recalculation and better uses the 32×32 to 64 multiplication instruction.

To summarize, the overall *convolved multiplication* involves:

1. Precalculate $5b_1$ to $5b_4$.

2. Carry out the multiplication in the coefficients c_0 to c_4, using the method described in Section 4.1.

3. Reduce the coefficients c_0 to c_4, as in Section 3.1.3, so that the result can be fed into another multiplication.

We now analyze possible overflow conditions. Only if the constraints listed below are all true for all inputs a and b, the convolved multiplication can be carried out.

- If each sum $c_0, \ldots, c_4$ exceeds 64-bit, it will takes three 32-bit registers to store. That said, the limbs $c_0 \ldots c_4$ must fit into 64-bit registers. In other words, the sum $a_0b_0 + 5a_1b_4 + 5a_2b_3 + 5a_3b_2 + 5a_4b_1$ (and thus its partial products) must not overflow over 64-bit.

  Suppose a and b are not fully-reduced and of which $a_i, b_j \leq R$ for some R. Then c_0 is the one the most possible to overflow. We have $c_0 \leq (1+5+5+5+5)R^2 \leq 2^{64} - 1$, or $R \leq 1.75 \times 2^{29}$.[1]

- We rewrite $5a_1b_4$ as $a_1 \times 5b_4$, so $5b_4$ must not overflow over 32-bit, or $b_i \leq (2^{32} - 1)/5 \leq 1.6 \times 2^{29}$.

3.1.2 Second Example

We now show that the radix-2^{26} representation is inappropriate for the field $\mathbf{F}_{2^{127}-25}$.

Since $2^{130} \equiv 2^3 \cdot 25 \equiv 200$, we have:

$$\begin{aligned}
c_0 &= a_0b_0 + 200a_1b_4 + 200a_2b_3 + 200a_3b_2 + 200a_4b_1 \\
c_1 &= a_0b_1 + a_1b_0 + 200a_2b_4 + 200a_3b_3 + 200a_4b_2 \\
c_2 &= a_0b_2 + a_1b_1 + a_2b_0 + 200a_3b_4 + 200a_4b_3 \\
c_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + 200a_4b_4 \\
c_4 &= a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0
\end{aligned}$$

[1]The upper bound for c_0 is actually smaller than $2^{64} - 1$, because we have not considered the carry chain yet. We will still stick with this bound in the examples. See Section 3.3.

Similarly, we observe two limits on the limb size:

- If a and b are not fully-reduced, of which each limb is less then R, then $c_0 \leq 801R^2 \leq 2^{64} - 1$, or $R \leq 1.13 \times 2^{27}$.
- We can rewrite $200a_1b_4$ as $8a_1 \times 25b_4$, or $10a_1 \times 20b_4$. Either case, these two quantities must not cause a 32-bit overflow.

The large coefficient 200 is due to the ineffective use of limb sizes. They sum up to 130 bits but the modulus $2^{127} - 25$ is only 127-bit long. Instead, we split 127 into 5 parts and let the i-th limb take $\lceil 127/5 \times i \rceil = \lceil 25.4i \rceil$ bits. This is more cost-effective.

Specifically, an element a of $\mathbf{F}_{2^{127}-25}$ is represented as a tuple $(a_0, \ldots, a_4)$ where

$$a = \sum_{i=0}^{4} a_i 2^{\lceil 25.4i \rceil}.$$

Now the new formula set is:

$$\begin{aligned}
c_0 &= a_0b_0 + 50a_1b_4 + 50a_2b_3 + 50a_3b_2 + 50a_4b_1 \\
c_1 &= a_0b_1 + a_1b_0 + 25a_2b_4 + 50a_3b_3 + 25a_4b_2 \\
c_2 &= a_0b_2 + 2a_1b_1 + a_2b_0 + 50a_3b_4 + 50a_4b_3 \\
c_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + 25a_4b_4 \\
c_4 &= a_0b_4 + 2a_1b_3 + a_2b_2 + 2a_3b_1 + a_4b_0
\end{aligned} \tag{3.1}$$

But with looser limits:

- $c_0 \leq 201R^2 \leq 2^{64} - 1$, or $R \leq 1.12 \times 2^{28}$.
- A possible approach is to rewrite $50a_ib_j$ as $2a_i \times 25b_j$, $2a_ib_j$ as $2a_i \times b_j$ and $25a_ib_j$ as $a_i \times 25b_j$, so we could reuse the values $2a_i$ and $25b_j$. These quantities must not overflow either.

3.1.3 Reduction and Carry Chains

We continue with the example in Section 3.1.1 to demostrate the reduction on $\mathbf{F}_{2^{130}-5}$ using radix-2^{26} representation.

To reduce a large coefficient c_0, we carry $c_0 \to c_1$, which means replacing (c_0, c_1) with $(c_0 \mod 2^{26}, c_1 + \lfloor c_0/2^{26} \rfloor)$; carry $c_4 \to c_0$ means replacing (c_4, c_0) with $(c_4 \mod 2^{26}, c_0 + 5\lfloor c_4/2^{26} \rfloor)$.

A complete carry chain $c_0 \to c_1 \to c_2 \to c_3 \to c_4 \to c_0 \to c_1$ produces appropriate ranges for each word of c to be fed into another multiplication:

$$\begin{aligned} c_0, c_2, c_3, c_4 &\le 2^{26} - 1, \\ c_1 &\le 2^{26} + 5 \cdot 2^{12}. \end{aligned}$$

Note that each c_1 is reduced but not fully-reduced. We leave the tedious calculation for the upper bound of c_1 until Section 3.3.

3.2 Mixed-Radix Representation

Now we give the formal definition of the mixed-radix representation.

Definition 4. *Let n be an integer and $B = m/n$. Denote x as $(x_0, x_1, \ldots, x_{n-1})$, where $x = \sum_i x_i 2^{\lceil Bi \rceil}$. This is called the radix-2^B mixed-radix representation of x.*

3.3 Formulating Representation Choice Criteria

We formulate the criteria for automating number representation selection, following the same argument as in Section 3.1.1. If these constraints fail, the convolved multiplication will not work properly. Our tool always checks these constraints before generating the program. This section can be omitted without being lost in the roadmap.

For simplicity, we will leave out the case for the mixed-radix representation, and only consider the radix-2^B representation, for which B is a multiple of m, or $m = nB$.

Let $p = 2^m - k$. First, we deal with the possible overflow conditions during reduction (carrying), assuming the carry chain $c_0 \to c_1 \to \ldots \to c_{n-1} \to c_0 \to c_1$.

- Assume that before reduction, each limb of c is bounded by R, i.e. $c_i \leq R$, then the carry $c_0 \to c_1 \to \ldots \to c_{n-1}$ should not overflow:

  $$R + \frac{R}{2^B} + \ldots + + \frac{R}{2^{(n-1)B}} \leq \frac{R}{1 - 1/2^B} \leq 2^{2W} - 1,$$

  or

  $$R \leq (2^{2W} - 1)(1 - \frac{1}{2^B}). \tag{3.2}$$

- The carry $c_{n-1} \to c_0$ should not overflow:

  $$(2^B - 1) + k \cdot \frac{2^{2W} - 1}{2^B} \leq 2^{2W} - 1.$$

  This condition limits the lower bound for limb size B given the prime p.

- Finally, we examine the bound of c_1 after carrying $c_0 \to c_1$.

  $$(2^B - 1) + \frac{1}{2^B}\left[(2^B - 1) + k \cdot \frac{2^{2W} - 1}{2^B}\right] \leq 2^B + k \cdot 2^{2(W-B)}.$$

  In other words, after reduction, the upper bound for c is:

  $$\begin{aligned} c_0, c_2, \ldots, c_4 &\leq 2^B - 1, \\ c_1 &\leq 2^B + k \cdot 2^{2(W-B)}. \end{aligned} \tag{3.3}$$

Second, we deal with the possible overflow during multiplication. A convolved multiplication modulo $p = 2^m - k$ would yield

$$c_0 = a_0b_0 + k \cdot a_1b_{n-1} + k \cdot a_2b_{n-2} + \cdots + k \cdot a_{n-1}b_1.$$

- If each a_i and b_j is bounded by S, then we have

  $$c_0 \leq S^2[1 + k(n-1)] \leq R \leq 2^{2W} - 1,$$

  or

  $$a_i, b_j \leq S = \sqrt{\frac{R}{1 + k(n-1)}},$$

  where R is defined in Equation 3.2.

- Also, a stricter bound is that both ka_i and $kb_i \leq 2^W - 1$, so

  $$a_i, b_j \leq \frac{2^W - 1}{k}.$$

Combining these two inequalities, we have for all a_i and b_j,

$$a_i, b_j \leq \min\left\{\sqrt{\frac{R}{1 + k(n-1)}}, \frac{2^W - 1}{k}\right\}. \tag{3.4}$$

This is a necessary condition for such an representation to work with the convolved multiplication method.

It follows from Equation 3.3 and Equation 3.4 that, the result of summing t numbers without reduction can still be the input of the multiplication, where

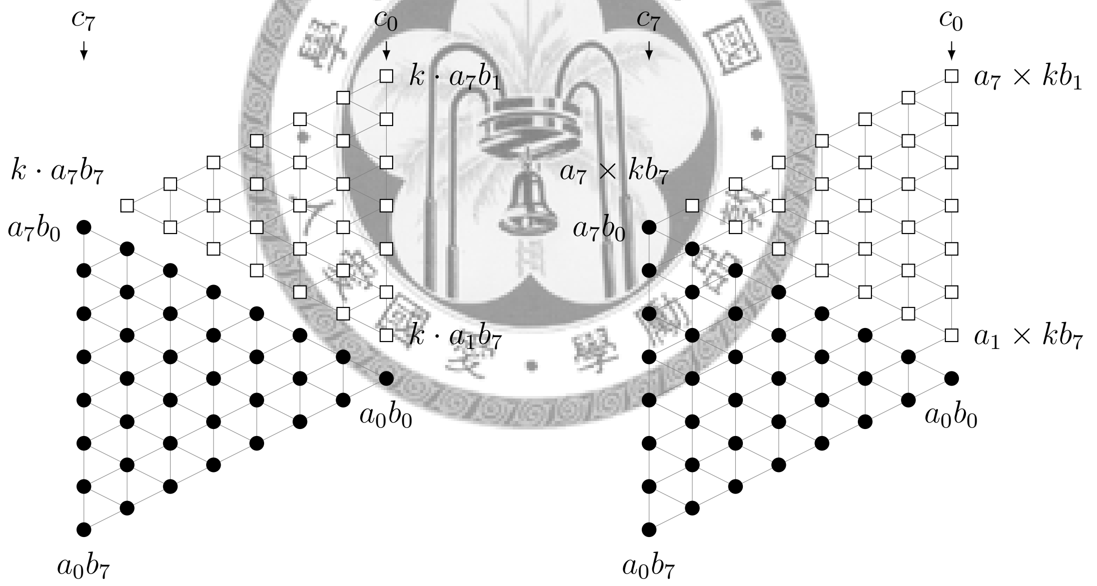
$$t = \min\left\{\sqrt{\frac{R}{1 + k(n-1)}}, \frac{2^W - 1}{k}\right\} \Big/ (2^B + k \cdot 2^{2(W-B)}).$$

This result is significant for ECC point addition and doubling, because the result of several big integer additions are often the input of a multiplication.

Similar arguments can be carried out for mixed-representation, but it is really complicated and sometimes verification by hand is even faster and less prone to error. We leave it for future work.

Chapter 4

Multiplication on Convolved Structures



(a) No data dependency in-between. (b) Operand reuse of a after rewriting.

Figure 4.1: Convolved structure. Reduced coefficients in rectangular markers.

Assume the radix-2^B representation, where B is an integer, as Example 1 in Section 3.1.1. We illustrate the convolved structure by shifting the left-hand half of the rhombus to the top-right corner, aligning each column with the same accumulator c_i. The reduced partial products to be multiplied by k are designated with rectan-

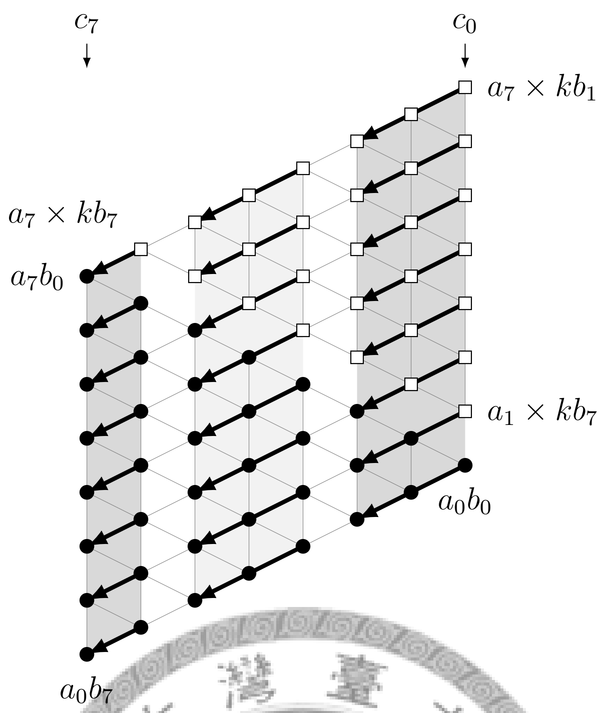


Figure 4.2: Convolved method for 8-word numbers, split into 3 parts. ($d = 3$)

gular markers (Figure 4.1). In Figure 4.1a, the two triangular grids are disconnected because they expose little or no possible operand caching, whereas in Figure 4.1b, rewriting $k \cdot a_ib_i$ as $a_i \times kb_i$ suggests the possibility of reusing words of a (but not b) between the two parts, shown by the lines connecting in-between.

4.1 Convolved Multiplication

We follow the principle that hybrid method uses – keep all the accumulators in a processed block in the register. The parameter d defines the number of columns within a processed block, thus $2d$ registers are needed for accumulation. Figure 4.2 shows the structure for $d = 3$, which is split into $3 = \lceil 8/3 \rceil$ parts, and each part is carried out using the row-wise method. Note that with the redundant representation, there is no carry propagation among each column. Partial products of each column could be summed simultaneously, which makes parallelism such as SIMD possible.

The above method applies to Example 1. For alternating mixed representa-

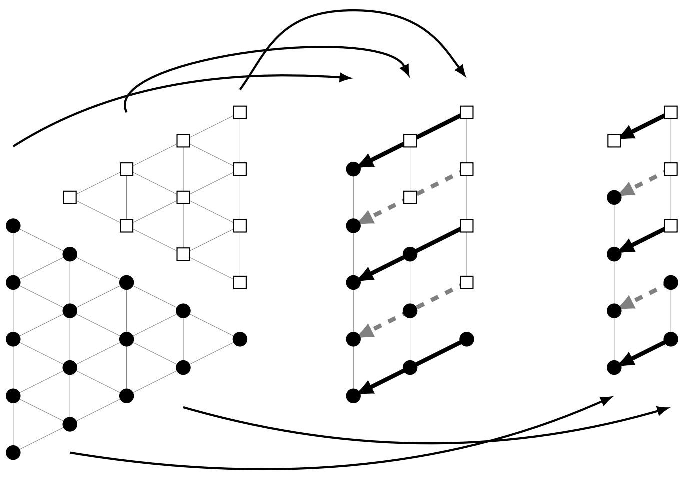


Figure 4.3: Convolved method with mixed-radix representation. Two parts are done with the row-wise method separately. Black arrows first, then dashed arrows.

tions as in Example 2, we observe that the same method is better carried out on even columns and on odd columns, as in Figure 4.3. The two parts are still processed using the row-wise method, but with the odd rows first, then the even rows. Correspondence between Figure 4.3 and Equation 3.1 show that this reuses loaded operands more effectively.

In the tools developed, we can choose use the default setting (particularly $d = 4$ on ARM11) as described above to produce a satisfying result. To achieve better performance, we can also facilitate the auto-tuning mechanism to enumerate different values for d or even split the convolved structure into more irregular parts.

Chapter 5

Implementation

5.1 System Overview

The tool we developed is implemented in Haskell with a little more than 1000 lines of code. We summarize the pipeline as follows:

Given an m-bit prime $p = 2^m - k$ as the input, the system first searches for a viable number representation for an m-bit number. For each possible case, the corresponding formula set for multiplication and reduction is generated to check overflow conditions, as in the two examples of Chapter 3 or in Section 3.3.

After the appropriate number representation is chosen, a number of intermediate programs will be generated using the proposed multiplication method, either by enumerating the parameter d or by choosing the default setting. These intermediate programs are actually assembly programs on the target platform, but with a view of infinite registers. Each of these intermediate programs is then converted to a form similar to SSA to apply a simplified version of the optimized linear scan register allocation. Some parameters of the register allocation algorithm is also tunable to probe the minimum number of spills.

Finally, a bunch of runnable programs is ready for benchmark on the target platfrom.

5.2 The ARM11 Processor Family

The ARM11 processor family was introduced by ARM in 2002 as the only implementation of the ARMv6 architecture. The most widely used processor from this family is the ARM1136, others the ARM1156 and the ARM1176. We developed and benchmarked the software described in this thesis on an ARM1136 processor, more specifically on a Samsung GT i7500 Galaxy smartphone containing a Qualcomm MSM7200A chip released in 2007. For details, please refer to the ARM1136 technical reference manuals. [ARM09]. In the following, we summarize the features most relevant to the implementations described in the thesis.

ARM11 processors have a 32-bit instruction set and 16 architectural 32-bit integer registers. One register is used as the stack pointer, one as the program counter, so 14 registers are freely usable.

Instructions are issued in order, one instruction per cycle. Except for multiplication, the arithmetic instructions relevant to the implementations in the thesis, have a latency of 1 cycle. Multiplication instructions takes 2 cycles, but their 2 word output have a latency of 4 and 5 cycles. Loads from cache have a latency of 3 cycles.

The instruction set is a standard RISC load-store instruction set except for two features: free shifts and rotates and loads and stores of more than 32 bits. The later yields better performance only in very special cases that we were not able to exploit in our implementations.

5.2.1 Free shifts and rotates

All arithmetic instructions have three operands and the output does not necessarily overwrite one of the inputs. Additionally, the second input operand can be shifted or rotated by arbitrary distances provided as immediate value or through a register. We use these features to speed up multiplication by k in precalculation. These shifts or rotates as part of arithmetic instructions do not decrease throughput or increase latency of the instruction, they are essentially for free. However, the shifted or rotated input value is required one stage earlier in the pipeline than a non-shifted input. Therefore, using the output of one instruction as shifted or rotated input to the next instruction imposes a penalty of one cycle.

5.2.2 Accessing the cycle counter.

Access to the 32-bit cycle counter is only possible from kernel mode, for example using the following code:

```
unsigned int c;
asm volatile("mrc p15, 0, %0, c15, c12, 1" : "=r"(c));
```

In a posting to the eBATS mailing list `ebats@list.cr.yp.to` from August 12, 2010, Bernstein publicized code for a kernel module that gives access to the cycle counter on ARM11 devices through the Linux device file `/dev/cpucycles4ns`.

5.3 Linear Register Allocation

Reducing register spilling is crucial to reducing memory accesses. Most recent research on multiplication techniques precisely specifies when to load operands and store intermediate results. To make the tool more general, we instead decided to allocate and spill registers automatically.

For this work, we chose to use the linear scan register allocation [PS99] for several reasons:

- The experimental results appeared satisfying. We may obtain better results using more sophisticated methods, such as graph coloring or integer programming. Traditional graph coloring is NP-complete, so is register allocation.

- It is much easier to implement and runs faster. The overall enumeration time can be reduced.

Specifically, we use an improved version of linear register scan algorithm with optimized interval splitting (for details, refer to [WM05]). In a nutshell, it works as follows: first the use positions and lifetime interval of each variable are identified through liveness analysis. Then it uses a heuristic based on the fixed interval information and the use positions of the active and inactive intervals: the interval that is not used for the longest time is spilled.

Since the generated code has no control flow, we can transform our programs into a form similar to SSA [CFR+91] (we call it the pseudo-SSA form) to further simply the algorithm. Also, to deal with the ARM11 architectural irregularity, we allocate registers as if each variable were used 1 cycle earlier, which reduces the load latency, and would be used a few cycles later, which reduces the multiplication latency.

5.4 Auto-Tuning

Auto-tuning is one of the core part of our tool. Here we list several parameters that can be tuned or enumerated:

1. The number representation and thus the corresponding limb size. Actually, it always chooses a most effective one that passed all overflow checks.

2. The parameter d of the convolved multiplication, or more precisely, how the convolved structure is split into several parts. We enumerate all possible combinations, of which each part contains no more than 5 columns. With 10 accumulators in the register file, there are still 4 registers free for use. The default setting is as given in the two examples in Chapter 4.

3. Register allocation. Ordinary arithmetic instructions has no latency, but multiplication instructions have a result latency of 5 cycles. For simplicity, we

allocate registers as if each variable would be used 1 or 2 cycles later. The default value is 1 cycle.

國立臺灣大學 · 敦品勵學 · 愛國愛人 ·

Chapter 6

Results and Discussion

6.1 Convolved Method on ARM11

Table 6.1: Convolved method and GMP multiplication modulo $2^{255} - 19$ on ARM11.

		GMP		
Prime	**Convolved**	Total	Multiply	Reduce
$2^{130} - 5$	235.3	1898.2	435.6	1382.3
$2^{255} - 19$	662.8	3054.5	981.1	1925.8

Table 6.1 shows the modular multiplication time using the convolved method and the GMP library 3.5.2 on ARM11. The convolved method is up to 5 times faster than the GMP counterpart, and is even faster than only a multiplication of the GMP library. The GMP modular reduction is slower because it assumes no particular form of the modulus.

Table 6.2 and 6.3 summarize the instruction counts for the convolved multiplication on $\mathbf{F}_{2^{255}-19}$ and on $\mathbf{F}_{2^{130}-5}$, using different number representations. In each table, we compare the fastest one generated by our tool and the equivalent C code compiled by GCC. They both follow the procedure of precalculation, multiplication and reduction. The complex pipeline of GCC may interleave these three stages for better performance.

Table 6.2: Convolved method modulo $2^{255} - 19$ on ARM11. Radix-$2^{25.5}$.

	Instructions						
Case	Mem32	Mem64	MUL	ADD	Others	Total	Cycles
Best Enumerated	172	0	100	84	15	371	662.84
Default Setting	165	0	100	84	15	364	666.88
GCC	235	29	111	22	59	456	792.83

Table 6.3: Convolved method modulo $2^{130} - 5$ on ARM11. Radix-2^{26}.

	Instructions						
Case	Mem32	Mem64	MUL	ADD	Others	Total	Cycles
Best Enumerated	40	0	25	37	13	115	235.33
GCC	43	8	25	28	29	133	241.41

For the case of $\mathbf{F}_{2^{255}-19}$ in Table 6.2, our method requires much less memory access, and runs 16% faster then the GCC counterpart. Even with the default setting, the result is comparable with the fastest enumerated one. Note that GCC is equipped with more advanced instruction scheduling and register allocation mechanisms. It also uses double-word memory access instructions supported by ARM11.

For the case of $\mathbf{F}_{2^{130}-5}$ in Table 6.3, the generated code also contains less instruction, but the cycle count is not as significant. We conclude that the convolved method outperforms more on larger moduli.

We expect to reduce the running time even more by interleaving the three procedures, as in the complex pipeline of GCC.

6.2 Drawbacks of Traditional Methods on ARM11

In this section, we discuss why traditional multiplication techniques work poorly on ARM11. Specifically, we estimate the theoretical speed lower bound for the operand-caching multiplication, and show that it can run slower than a modular multiplication using the convolved multiplication.

As one of the fastest multiplication methods present by 2011, the operand-caching method [HW11] aims at reducing the number of loads on embedded platforms, such as an 8-bit ATmega128 microcontroller. In Table 6.4, we list the lowest number of memory accesses, multiplications and additions described in the original paper, and estimate the cycles needed to carry out one multiplication on ARM11. We compare this with the automatically generated 255-bit convolved method, of which the cycle count is actually measured on a ARM11 processor.

Table 6.4: Lower cycle bound for 256-bit operand-caching multiplication on ARM11. Theoretical minimum values in italic, can only be larger.

		Instructions				
Method	Radix	LDR	STR	MUL	ADD	Cycles
Operand-Caching	2^{32}, 8 limbs	*34*	*21*	*64*	*192*	*665*
Convolved (mod $2^{255}-19$)	$2^{25.5}$, 10 limbs	110	62	100	84	662.84

Interestingly, the result suggests that a modular multiplication can even run faster then the operand-caching method without reduction, although the latter has fewer instructions. Recall from Section 5.2 that on ARM11, each 32×32-bit multiplication takes 2 cycles. Although the result has a latency of 5 cycles, we can carefully schedule the instructions to hide these 3-cycle latencies. Traditional multiplication techniques such as the operand-caching method, however, requires each multiplication to set carry flags, which takes 6 cycles. Another source of latency is that the algorithm assumes use the result of a load in the next instruction, which causes a penalty latency of 1 cycle. The total cycle count is therefore at least $34 \times 2 + 21 + 64 \times 6 + 192 = 665$.

Chapter 7

Conclusion

We presented a domain specific compiler framework that automates the cumbersome process of tuning and coding for fast modular multiplication on ARM11. Specifically, we choose to use the convolved multiplication on pseudo-Mersenne prime fields, and proposed a multiplication method on the convolved structure. The generated code may run faster than a hand-craft counterpart. We also showed that multiplication using radix-2^W representation may cause a serious bottleneck on ARM11.

Future Works. We have four primary goals for the tool:

- Integration of domain specific languages. Extending the compiler to handle elliptic curve point multiplication routines is more practical but checking overflow conditions is more completed for complex formulae.
- Early pruning for enumeration. A good pruning heuristic, such as the length of the program, may decrease the benchmark time for running all programs.
- SIMD. The convolved method has a structure that can benefit from SIMD improvements.
- Using Hoopl. Hoopl [RDPJ10] is a Haskell library for dataflow analysis and code transformation for better code quality and performance.

Subsidiary goals include more advanced multiplication techniques and more prime forms. For example, the Karatsuba multiplication may speed up multiplication only a little, but requires more complicated overflow checks. In such cases, possible erroneous inductions or programs may be subject to formal verification. The convolved method may also be used on generalized pseudo-Mersenne primes. We also consider adopt signed number representations, as in [JG02, BDL+12].

國立臺灣大學
愛國・愛人・敦品・勵學

Bibliography

[ARM09] ARM Limited. *ARM1136JF-S and ARM1136J-S Technical Reference Manual, Revision: r1p5*, 2009. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0211k/DDI0211K_arm1136_r1p5_trm.pdf`.

[BDL$^+$12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. Document-ID: a1a62a2f76d23f65d622484ddd09caf8, `http://cryptojedi.org/papers/#ed25519`.

[Ber00] Daniel J. Bernstein. Floating-point arithmetic and message authentication, 2000.

[Ber05] Daniel J. Bernstein. The poly1305-aes message-authentication code. In *In Proc. FSE*, pages 32–49, 2005.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[Com90] P. G. Comba. Exponentiation cryptosystems on the ibm pc. *IBM Syst. J.*, 29(4):526–538, October 1990.

[GPW$^+$04] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In *CHES'04*, pages 119–132, 2004.

[HS13] Michael Hutter and Peter Schwabe. Nacl on 8-bit avr microcontrollers. In Amr Youssef and Abderrahmane Nitaj, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer-Verlag Berlin Heidelberg, 2013. Document ID: cd4aad485407c33ece17e509622eb554, `http://cryptojedi.org/papers/#avrnacl`.

[HW11] Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In *Proceedings of the 13th international conference on Cryptographic hardware and embedded systems*, CHES'11, pages 459–474, Berlin, Heidelberg, 2011. Springer-Verlag.

[JG02] Ghassem Jaberipur and Mohammad Ghodsi. High radix signed digit number systems: Representation paradigms. *Scientia Iranica*, 10:383–391, 2002.

[Nat00] National Institute of Standards and Technology. *FIPS PUB 186-2: Digital Signature Standard (DSS).* January 2000.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, September 1999.

[RDPJ10] Norman Ramsey, João Dias, and Simon Peyton Jones. Hoopl: a modular, reusable library for dataflow analysis and transformation. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 121–134, New York, NY, USA, 2010. ACM.

[Sch11] Peter Schwabe. *High-Speed Cryptography and Cryptanalysis*. PhD thesis, Eindhoven University of Technology, 2011. `http://cryptojedi.org/users/peter/thesis/`.

[WM05] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 132–141, New York, NY, USA, 2005. ACM.

國立臺灣大學 · 敦品勵學 · 愛國愛人