THE DESIGN SPACE OF ULTRA-LOW ENERGY ASYMMETRIC CRYPTOGRAPHY

A Dissertation

by

ANDREW DAVID TARGHETTA

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Paul V. Gratz
Committee Members,	Sunil P. Khatri
	Harlan R. Harris
	Maury H. Rahe
Head of Department,	Miroslav M. Begovic

May 2015

Major Subject: Computer Engineering

ABSTRACT

The energy cost of asymmetric cryptography, a vital component of modern secure communications, inhibits its wide spread adoption within the ultra-low energy regimes such as Implantable Medical Devices (IMDs), Wireless Sensor Networks (WSNs), and Radio Frequency Identification tags (RFIDs). In literature, a plethora of hardware and software acceleration techniques exists for improving the performance of asymmetric cryptography. However, very little attention has been focused on the energy efficiency. Therefore, in this dissertation, I explore the design space thoroughly, evaluating proposed hardware acceleration techniques in terms of energy cost and showing how effective they are at reducing the energy per cryptographic operation. To do so, I estimate the energy consumption for six different hardware/software configurations across five levels of security, including both $GF(p)$ and $GF(2^m)$ computation. First, we design and evaluate an efficient baseline architecture for pure software-based cryptography, which is centered around a pipelined RISC processor with 256KB of program ROM and 16KB of RAM. Then, we augment our processor design with simple, yet beneficial instruction set extensions for $GF(p)$ computation and evaluate the improvement in terms of energy per cryptographic operation compared to the baseline microarchitecture. While examining the energy breakdown of the system, it became clear that fetching instructions from program memory was contributing significantly to the overall energy consumption. Thus, we implement a parameterizable instruction cache and simulate various configurations. We determine that for our working set, the energy-optimal instruction cache is 4KB, providing a 25% energy improvement over the baseline architecture for a 192-bit key-size. Next, we introduce a reconfigurable $GF(p)$ accelerator to our microarchitecture and mea-

sure the energy per operation against the baseline and the ISA extensions. For ISA extensions, we show between 1.32 and 1.45 factor improvement in energy efficiency over baseline, while for full acceleration we demonstrate a 5.17 to 6.34 factor improvement. Continuing towards greater efficiency, we investigate the energy efficiency of different arithmetic by first adding $GF(2^m)$ instruction set extensions to our processor architecture and comparing them to their $GF(p)$ counterpart. Finally, we design a non-configurable 163-bit $GF(2^m)$ accelerator and perform some initial energy estimates, comparing them with our prior work. In the end, we discuss our ongoing research and make suggestions for future work. The work presented here, along with proposed future work, will aid in bringing asymmetric cryptography within reach of ultra-low energy devices.

ACKNOWLEDGEMENTS

I would like to thank Don Owen, Allen Luettgen and Francis Israel for their contributions to this work. Without their invaluable guidance and efforts, none of this work would have been possible.

TABLE OF CONTENTS

Page

ABSTRACT ii

ACKNOWLEDGEMENTS iv

TABLE OF CONTENTS v

LIST OF FIGURES vii

LIST OF TABLES ix

1. INTRODUCTION 1

1.1 Asymmetric Cryptography 1
1.2 Thesis Statement 4
1.3 Contributions 4

2. BACKGROUND 6

2.1 Asymmetric Cryptography 6
2.1.1 Confidentiality, Authenticity, and Integrity 7
2.1.2 The One-Way Function and Finite Fields 9
2.1.3 Prime Fields and Modular Exponentiation 11
2.1.4 Binary Fields 13
2.1.5 Elliptic Curve Cryptography 16
2.2 HW/SW Codesign and Computer Architecture 21
2.3 Energy Consumption in Digital Circuits 30

3. RELATED WORK 32

4. ALGORITHMS AND SOFTWARE 36

4.1 ECDSA 36
4.2 Multi-precision Routines 38
4.2.1 Prime Field Multiplication 39
4.2.2 Binary Field Multiplication 45
4.2.3 Binary Squaring 47
4.2.4 Field Addition/Subtraction and Inversion 48

4.3 Software Build/Run-time Environment 49

5. MICROARCHITECTURES 50

5.1 Baseline 50
5.1.1 Statically Scheduled Multiply 51
5.1.2 Karatsuba Multiplier Implementation 53
5.2 ISA Extensions 54
5.2.1 Prime Fields 55
5.2.2 Binary Fields 56
5.3 Instruction Cache 61
5.3.1 Cache Implementation 61
5.3.2 System Integration 63
5.3.3 Prefetching 64
5.4 Prime-field Accelerator 65
5.4.1 Coprocessor Interface 66
5.4.2 Prime-field Arithmetic Unit 69
5.5 Binary-field Accelerator 78
5.5.1 Coprocessor Instructions 79
5.5.2 Microarchitecture 80
5.5.3 $GF(2^m)$ Arithmetic Units 82

6. METHODOLOGY 85

7. EVALUATION 86

7.1 Prime Fields 86
7.2 Binary Fields 92
7.3 Prime Fields vs. Binary Fields 93
7.4 Power Consumption 96
7.5 Evaluation with Instruction Cache 98
7.6 Performance Evaluation 102
7.7 Double Buffer Evaluation 105
7.8 Baseline Validation 106
7.9 FFAU Evaluation 107

8. CONCLUSIONS AND FUTURE WORK 111

REFERENCES 114

LIST OF FIGURES

FIGURE Page

1.1 The hardware acceleration trade-off. 4

2.1 Basic cryptography 7

2.2 Asymmetric cryptography for confidentiality or authenticity 8

2.3 Elliptic Curve point addition and doubling on $E(\mathbb{R})$ 17

2.4 The microarchitecture of a 5-stage pipeline processor 26

2.5 A direct-mapped cache with a block size of 16 bytes and a 32-bit word width. 29

4.1 Elliptic Curve Digital Signature Algorithm computation hierarchy . . 36

5.1 Baseline: RISC Processor w/ ROM and RAM 51

5.2 The Karatsuba Multiply Unit within the baseline architecture. 54

5.3 The Karatsuba Multiply-Accumulate Unit including support for prime-field ISA extensions. 58

5.4 The Karatsuba Multiply-Accumulate Unit including support for prime- and binary-field ISA extensions. 59

5.5 The implementation of a direct-mapped instruction cache. 60

5.6 Pete with an instruction cache. 63

5.7 The prime field accelerated architecture, “Pete with Monte.” 66

5.8 The Finite-Field Arithmetic Unit at the center of “Monte” 70

5.9 Top Level Architecture of the FFAU 72

5.10 The Control Unit within the FFAU 75

5.11 The binary-field accelerated architecture, “Pete with Billie” 78

5.12 Billie's coprocessor architecture. 80

5.13 Binary-field squaring unit 84

7.1 Energy per Sign + Verify vs. key size and microarchitecture for prime fields. 87

7.2 Breakdown of energy per Sign + Verify for 192 and 256-bit key sizes into various sub-components. 88

7.3 Energy per Sign + Verify vs. key size for our baseline with no hardware acceleration. 89

7.4 Energy per Sign + Verify vs. key size for the ISA extended microarchitecture and the architecture accelerated with Monte. 91

7.5 Energy per Sign + Verify vs. key size for binary fields. 92

7.6 Energy per Sign + Verify vs. key size for binary ISA extensions. . . . 93

7.7 Energy per Sign + Verify vs. key size, comparing prime and binary fields of equivalent security. 94

7.8 Energy per Sign + Verify vs. key size for Monte and Billie 95

7.9 Breakdown of energy per Sign + Verify for 192/163- and 256/283-bit key sizes into various sub-components. 97

7.10 Static and dynamic power of evaluated microarchitectures. 98

7.11 Energy improvement with ideal instruction cache vs. key size. 99

7.12 Energy per 192-bit Sign + Verify with real instruction cache for various cache configurations 100

7.13 Energy per Sign + Verify vs. key size for prime ISA extended microarchitecture with 4KB instruction cache. 102

7.14 Performance for 163-bit scalar point multiply comparing Billie to prior work 105

7.15 Energy per Montgomery multiplication vs. datapath width 109

LIST OF TABLES

TABLE Page

5.1 Instruction set extensions for prime fields 55

5.2 Instruction set extensions for binary fields 58

5.3 Coprocessor 2 Instructions used to control Monte 66

5.4 Arithmetic Core Computational Capabilities 73

5.5 Index Register Control Codes 75

5.6 Coprocessor 2 Instructions used to control Billie 79

7.1 Latency per operation for prime-field microarchitectures 103

7.2 Latency per operation for binary-field microarchitectures 104

7.3 Area utilization, static power, and dynamic power vs. datapath width. 108

7.4 Average power, execution time, and energy per Montgomery multiplication vs. datapath width 110

7.5 Average power and energy per modular multiplication vs. key size for the ARM Cortex-M3 110

1. INTRODUCTION

Since the advent of the microprocessor in the early 1970s, the number of components that fit on a single Integrated Circuit (IC) has continued to climb. This increase has primarily been due to advances in IC fabrication techniques, leading to trends in device scaling first described by Gordon Moore in 1964 [1]. "Moore's Law," the name given to the rapid growth in integrated circuit density, has given rise to the System on a Chip (SoC), which has allowed miniature computer systems to be embedded in everything from microwaves to the human body.

As SoCs become more ubiquitous, the desire to communicate with them escalates. For example, many programmable thermostats now have built-in wireless capabilities. Moreover, these devices are being trusted to communicate increasingly sensitive data, while concerns for privacy grow stronger. Therefore, embedded devices need to be equipped with algorithms such as asymmetric cryptography in order to securely communicate.

1.1 Asymmetric Cryptography

Asymmetric cryptography, also known as public key cryptography, has become an essential component in modern, secure communications. Unlike its symmetric counterpart, asymmetric cryptography requires separate keys for encryption and decryption, allowing it to solve a host of security challenges not possible with symmetric cryptography alone. Uses for asymmetric cryptography range from session key establishment for secure communications to digital signatures for message authenticity and non-repudiation. While symmetric cryptography is based on data shifts and permutations, asymmetric cryptography is built upon a foundation of mathematically hard problems. As a result, the computational requirements for asymmetric

cryptography are far greater than that of symmetric cryptography [2].

Employing asymmetric cryptography on ultra-low energy devices, such as Implantable Medical Devices (IMDs) [3, 4], Wireless Sensor Networks (WSNs) [5], and Radio Frequency Identification (RFID) tags [6, 7], can be especially challenging. In this class of applications, the energy cost of each operation is paramount to the device's utility. For example, in a typical IMD, each extra Joule expended in computation reduces the life of the device, and each surgical replacement of the device endangers the life of the patient. Security in this application is of critical importance; unauthorized access to an implanted cardiac defibrillator's programming interface poses an unambiguous threat to the patent's health and privacy.

Despite the obvious need for security in this domain, relatively few designs have incorporated encryption; among these, most employ symmetric (shared-key) encryption techniques [3]. More secure schemes for communication exist that involve asymmetric cryptography. However, the high computational cost of asymmetric cryptography has put these schemes out of reach for ultra-low energy applications. In the WSN domain, Wander *et al.* found that even weak asymmetric cryptography (160-bit ECC, equivalent to 1024-bit RSA) consumes approximately 72% of the energy allotted for communication handshaking. Moreover, they assume that only 5% to 10% of a WSN's energy budget is available for handshakes [8]. Pabbuleti *et al.* show that asymmetric cryptography reduces the energy cost of transmitting the signature compared to hash-based authentication protocols; however, the energy cost of computation rapidly exceeds the cost of signature transmission when considering 128-bit security levels [9]. For RFID tags, it is difficult to quantify the energy budget for encryption; however, because most tags are passive energy harvesters, the budget is significantly less than that of a WSN node.

To alleviate this computational burden, special purpose hardware can be designed

into an embedded system to accelerate portions of the cryptographic algorithms. Hardware designed to perform specific computations will typically do so more efficiently compared to hardware designed for general purpose computation. For the end user, hardware acceleration yields an overall design that is not only more responsive but is also much more energy efficient. Whereas past work has extensively evaluated the performance gains associated with hardware acceleration, this work focuses primarily on the energy benefit. In other words, this work attempts to comprehensively quantify energy improvements available through the hardware acceleration of asymmetric cryptography.

In the ultra-low energy domain, a spectrum of hardware/software acceleration techniques exists, in which an increase in hardware acceleration will lower reconfigurability in exchange for energy efficiency. Figure 1.1 depicts this trade-off with compiled software executing on a power-conscious processor on one side and a fully dedicated cryptographic processor on the other. The more interesting research lies in the middle, where some degree of reconfigurability is maintained while the energy consumed per operation is much less than that of a pure software implementation. This area is precisely the portion of the spectrum our work attempts to capture.

Understanding the energy design space specific to asymmetric cryptography is important in order to ensure the correct trade-offs are made prior to device fabrication. For example, a lack of reconfigurability could render the device obsolete sooner, as security requirements change, while too little hardware acceleration could render the device inoperable under assumed energy budgets. Furthermore, too much hardware acceleration could unnecessarily increase the cost of design validation and device fabrication. Thus, we compare different points on the spectrum and let the system designer choose which level of acceleration is appropriate.

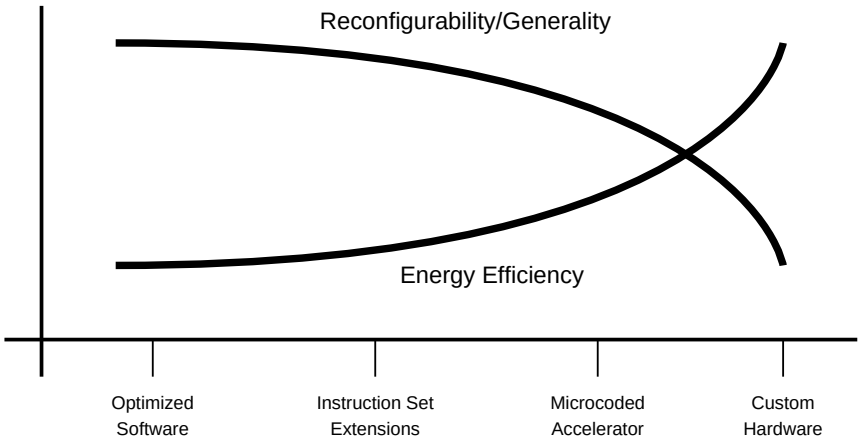


Figure 1.1: The hardware acceleration trade-off.

1.2 Thesis Statement

In this dissertation, I provide a thorough evaluation of the design space of energy-efficient asymmetric cryptography. In doing so, I describe the steps taken to design and accurately model our embedded system, which includes the development of an embedded processor with application specific extensions as well as two custom arithmetic accelerators. This dissertation showcases the energy efficiency of our custom arithmetic logic, making a strong argument for hardware acceleration of asymmetric cryptography.

1.3 Contributions

For comparison, we start by evaluating a baseline architecture in terms of energy cost per signature and verification operation, as defined by the Elliptic Curve Digital Signature Algorithm (ECDSA) [10, 11]. Our baseline represents the left-most side of Figure 1.1 and consists of a low-power RISC processor without an instruction cache and a minimal memory layout, typical of an embedded microcontroller. Moving to the right within Figure 1.1, we add simple yet effective instruction set extensions

to our baseline architecture and evaluate the improvement in terms of energy cost per operation. Next, we evaluate our system with a reconfigurable, microcoded accelerator that we designed for prime finite-field arithmetic. As a comparison, we evaluate the energy benefit of a non-configurable, accelerator that we designed for binary finite-field arithmetic. Although the non-configurable aspect implies that the level of security is fixed after device fabrication, this configuration yields the highest degree of energy efficiency. Finally, we include an instruction cache in our design and measure the energy improvement that it provides for the ISA extended architectures. The contributions of this work are summarized as follows:

- Detailed power, energy and performance analysis of ultra-low energy asymmetric cryptography for several different hardware/software configurations within the same technology node, using the same experimental techniques
- Design space exploration across a range of Elliptic Curve Cryptography (ECC) key-sizes that includes up to 521-bit prime and 571-bit binary, providing insight into current and future secure data exchange for embedded systems
- Development of a microcoded, prime-finite field accelerator that maintains reconfigurability via microcode programming while decreasing the energy per digital signature
- Development of a binary-field accelerator that further reduces the energy of asymmetric cryptography while outperforming prior work
- Evaluation of the energy benefit of an instruction cache in the context of asymmetric cryptography
- Detailed hardware models and recommendations for future energy exploration within the ultra-low energy domain

2. BACKGROUND

In this chapter, we refresh the reader's understanding of the relevant background topics for this study. We start by reviewing basic cryptographic concepts and introducing the mathematics that underpin all asymmetric cryptosystems. Then we provide a brief primer on computer architecture in order to explain some of the terminology referenced throughout this work. Finally, we review how energy is consumed in digital circuits and discuss the relationship between power and energy. A reader already familiar with these topics may skip this chapter.

2.1 Asymmetric Cryptography

The field of cryptography encompasses the techniques and mechanisms used to communicate securely over an insecure channel. The primary objective is to encrypt data prior to communication in such a way that it can only be decrypted by the intended recipient. Consider the textbook scenario, depicted in Figure 2.1, where Alice encrypts a plaintext message using her encryption key and sends the encrypted data, also know as the ciphertext, to Bob over a public channel. Bob uses his decryption key to translate the ciphertext back into plaintext. Along the way, the data is intercepted by Eve, an eavesdropper; however, Eve is unable to recover the plaintext message without Bob's decryption key.

In cryptography, there are two distinct categories: symmetric and asymmetric. Symmetric cryptography uses the same key for encrypting and decrypting data, whereas asymmetric cryptography uses one key for encrypting and a separate key for decrypting. By keeping one key private and making the other key publicly available, asymmetric cryptography (a.k.a public-key cryptography) can solve a host of problems not possible with symmetric cryptography alone[12]. Classic schemes, such

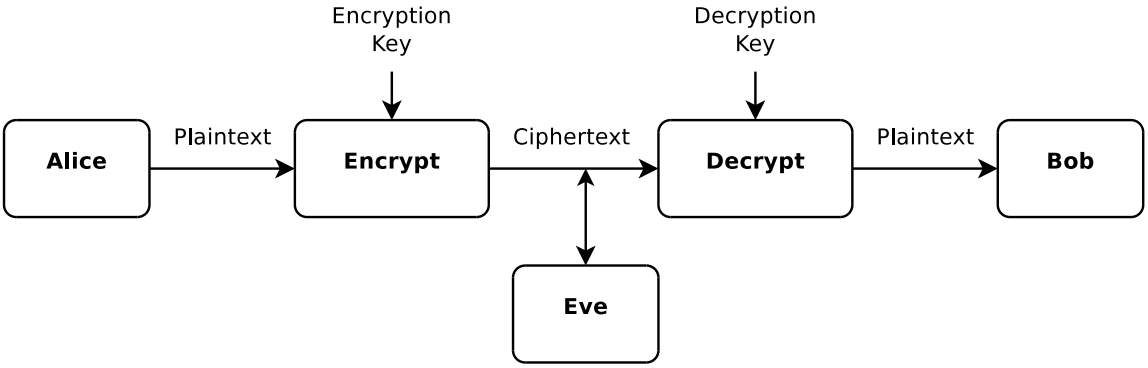


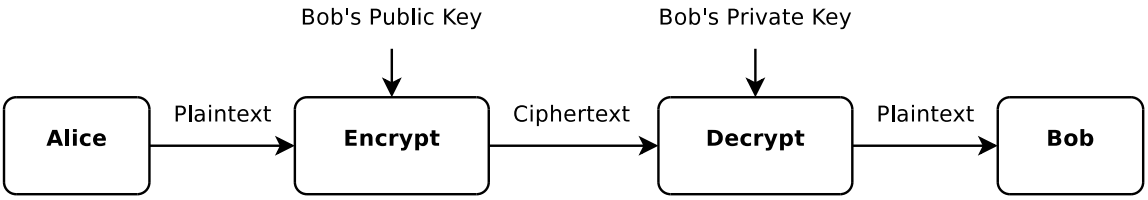
Figure 2.1: Basic cryptography

as substitution ciphers, along with some modern encryption algorithms, such as DES and AES, fall into the symmetric cryptography category. RSA and Diffie-Hellman key exchange are examples of early asymmetric cryptography, while modern schemes employ Elliptic Curve Cryptography, such as Elliptic Curve Diffie-Hellman (ECDH) key exchange and the Elliptic Curve Digital Signature (ECDSA).

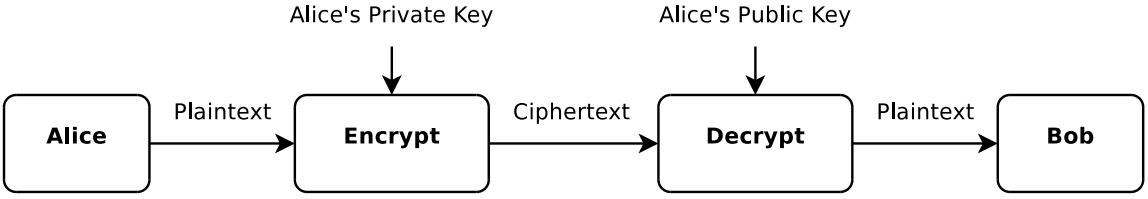
2.1.1 Confidentiality, Authenticity, and Integrity

Confidentiality refers to the protection of a message from eavesdropping, while authenticity refers to trust in the origin of the message. Data integrity ensures the message has not been modified, whether accidental or malicious. In an asymmetric cryptosystem, each communicating entity has its own private/public key pair. Then, depending on how the keys are used, asymmetric cryptography can provide data confidentiality or authenticity/integrity. Figure 2.2a demonstrates the use of asymmetric cryptography for confidentiality, while Figure 2.2b demonstrates its use for authenticity/integrity. For confidentiality, Alice uses Bob's public key to encrypt a message and sends the resulting ciphertext to Bob who uses his private key to decrypt the ciphertext. In this scenario, only Bob's private key can be used to recover

a message encrypted with his public key. Furthermore, Bob's private key cannot be derived from his public key. Even though the message is sent via an unsecured channel, it is still protected from unauthorized access.



(a) Confidentiality



(b) Authenticity

Figure 2.2: Asymmetric cryptography for confidentiality or authenticity

For authenticity, Alice uses her private key to encrypt a message before she sends it to Bob, who in turn uses her public key to decrypt the ciphertext. If the decryption process yields an intelligible message, then Bob has high degree of confidence that the message originated from Alice. Remember that only Alice's private key could have been used to encrypt a message that can be decrypted with her public-key.[1] It should be noted that tampering with the ciphertext will yield an unreadable message after decryption, so data integrity is ensured as well.

Both confidentiality and authenticity can be achieved by encrypting first for confidentiality and then again for authenticity. In which case, two encryption operations

[1]The underlying assumption here is that Bob is able to somehow authenticate Alice's public key. This is where certificates and the public-key infrastructure come into play.

will be required on the sender's side as well as two decryption operations on the receiver's side. Likewise, both key pairs from Alice and Bob are required. In this scenario, an adversary who stands between Alice and Bob can neither decipher their communication nor successfully impersonate one or the other.

With the properties of confidentiality, authenticity and data integrity, asymmetric cryptography can solve secret-key distribution problems. In order for two entities to communicate securely using a symmetric cipher, they must somehow securely exchange a shared secret key. Without asymmetric cryptography, this would require an additional medium that can guarantee privacy; otherwise, an adversary could discover the shared key and easily decrypt future communication. Moreover, large key rings are required if a number of devices need to communicate securely. For instance, n devices would require a total of $\frac{n(n-1)}{2}$ different secret keys, where each device must store $n-1$ keys [2].

With asymmetric cryptography, any two entities can easily and securely exchange a temporary secret-key and then use symmetric cryptography to encrypt data traffic for the remainder of the communication session. It should be noted that asymmetric cryptography is ill-suited for bulk data encryption due to its high computational cost. Thus, it is more energy efficient to amortize a key-exchange across a lengthy communication session [13]. We will talk more about the protocols developed for secure key exchange after delving into the mathematics.

2.1.2 The One-Way Function and Finite Fields

Improving the energy efficiency of an asymmetric cryptosystem requires an understanding of the underlying mathematics. Therefore, the following section will briefly review the necessary mathematical concepts. At the core of asymmetric cryptography is the mathematical one-way function with a trapdoor. A one-way function

has a forward operation that is easy to compute but an inverse operation that is considered computationally infeasible to compute. When a one-way function has a trapdoor, certain knowledge can make the inverse operation also easy to compute [14]. One-way functions for asymmetric cryptography are constructed using finite fields, which are part of a division of mathematics known as abstract algebra. In order to understand *finite fields*, we must first understand *groups* and *rings*.

A *group*, $\{G, \bullet\}$, is a set (G) with a binary operation $(\bullet)$ and the following properties:

- *Closure*, if $a, b \in G$, then $a \bullet b \in G$
- *Associative*, $a \bullet (b \bullet c) = (a \bullet b) \bullet c$ if $a, b, c \in G$
- *Unit Element*, there exists an element, e, such that $a \bullet e = e \bullet a = a$ for all $a \in G$
- *Inverse Element*, for all $a \in G$ there exists an element a' such that $a \bullet a' = a' \bullet a = e$

If the *commutative* property, $a \bullet b = b \bullet a$ for all $a, b \in G$, holds true, then the group is an *Abelian group*.

A *ring*, $\{R, +, \times\}$, is a set (R) with two binary operations $(+, \times)$ and the following properties:

- R is an Abelian group with respect to $+$
- *Closure* over $\times$
- *Associative* over $\times$
- *Unit Element* with respect to $\times$

- *Distributive*, $a \times (b + c) = a \times b + a \times c$ and $(b + c) \times a = b \times a + c \times a$

If the *commutative* property holds true for $\times$, then the ring is commutative. Note that the $+$ and $\times$ operations are commonly referred to as addition and multiplication, respectively.

A *Field*, $\{F, +, \times\}$, is a commutative ring such that all elements *except* the additive identity element (*i.e.*, zero) in F have a multiplicative inverse element. For multiplication, the inverse element of a is denoted by a^{-1}. The inverse of a with respect to addition is denoted by $-a$. In a field, the subtraction and division operations are derived from addition and multiplication by utilizing the inverse element of the second operand, so the following holds true:

- $a - b = a + (-b)$
- $a/b = a \times (b^{-1})$

In other words, a field is a set of elements over which we can perform addition, subtraction, multiplication and division; however, division by zero is not allowed. If F is *finite*, then the field is referred to as a *finite field*.

2.1.3 Prime Fields and Modular Exponentiation

The modulo operation, *a modulo p* where a and p are integers, is equal to r such that $a = q * p + r$ for some value of q. The integers from 0 to $p - 1$ are known as the set of residues *modulo p*. If p is prime and all arithmetic computations on the set of residues are performed *modulo p*, the result is a *prime field*, denoted by GF(p).[2] The unit element with respect to addition for prime fields is 0, while the unit element for multiplication is 1. The following are examples of GF(7) computation:

- *Addition:* $2 + 5$ *modulo* $7 = 0$

[2]GF stands for Galois Field and is named after the French mathematician, Evariste Galois.

- *Subtraction:* $3 - 6$ *modulo* $7 = 4$
- *Multiplication:* 5×4 *modulo* $7 = 6$
- *Division:* $2 \div 4$ *modulo* $7 = 4$

For division, if $a, b, c \in GF(p)$, $c = a \div b$ modulo p such that $c \times b \equiv a$ modulo p and is found by first solving for b^{-1} modulo p then computing $a \times b^{-1}$ modulo p. It should be noted that big integer division is extremely costly in terms of computation. Thus, more efficient methods exist to perform the reduction operation (*modulo* p) and compute the inverse (a^{-1} *modulo* p). We will discuss these methods in more detail when we talk about the specific algorithms used in this study.

Traditional public-key cryptosystems such as RSA, Diffie-Hellman, and the Digital Signature Algorithm (DSA) utilize modular exponentiation ($y = g^x \bmod p$) as the one-way function [11, 15, 12]. The brute-force method for computing modular exponentiation is to multiply g by itself x times, but far more efficient techniques exist, such as the suite of *repeated square-and-multiply algorithms* [2]. Each square or multiply in modular exponentiation is an operation performed over a finite field.[3] Assuming a 4096-bit RSA algorithm, on the order of $1.5 * 4096$ field multiplications, each of size 4096 bits, must be performed for each modular exponentiation. The reverse operation, compute x given y, g, p, is referred to as the Discrete Logarithm Problem (DLP) and is considered intractable as the size of the modulus increases. Methods considerably more efficient than brute force exist for computing the DLP. Thus, very large integers must be used to ensure security with traditional public-key cryptosystems based upon modular exponentiation. As we will see shortly, more efficient one-way functions exist, which allow computation over smaller fields.

[3]To be pedantic, the operations are over a *multiplicative Abelian group* because modular exponentiation only uses multiplication.

2.1.4 Binary Fields

Prime fields are commonly used for asymmetric cryptography, but when considering elliptic curves, other types of fields may be used as well. For finite-field computations, the order does not necessarily have to be prime but must be a power of a prime, *e.g.*, GF(p^m) where m is an integer such that $m > 0$, and p is the *characteristic* of the finite field. If $m > 1$, polynomial arithmetic, such that the coefficients are computed *modulo* p can be used.

Finite fields with a characteristic of 2, referred to as *binary fields* or $GF(2^m)$, are especially attractive for custom hardware implementations because addition is simply a bitwise XOR operation. Since multiplication is derived from addition, the partial-product accumulation within multiplication is similar to that of integer multiplication but without the carry logic. For this reason, binary-field arithmetic is often called "carry-less" arithmetic.

Because we use polynomial arithmetic for binary-field computation, we borrow the polynomial representation. As such, a $GF(2^m)$ field is denoted in the following way: $a(x) = a_{m-1}x^{m-1} + \cdots + a_2x^2 + a_1x + a_0$ where x is the indeterminate of the polynomial, and the coefficients, $a_{m-1}, \cdots, a_2, a_1, a_0 \in [0, 1]$. In a computer system, a binary-field element is stored as an m-bit binary vector, $(a_{m-1}, \cdots, a_2, a_1, a_0)$. As with prime fields, the result of a binary-field multiplication needs to be reduced. Binary-field reduction is performed modulo an irreducible polynomial, $f(x)$. Note that unlike prime fields, binary-field addition and subtraction do not require reduction because there are no arithmetic carries.

The following are examples of $GF(2^7)$ computation assuming $f(x) = x^7 + x + 1$:

- *Addition:* $(x^6 + x^4 + x^3 + 1) + (x^5 + x^4 + x^2 + 1) = x^6 + x^5 + x^3 + x^2$
- *Subtraction:* $(x^6 + x^4 + x^3 + 1) - (x^5 + x^4 + x^2 + 1) = x^6 + x^5 + x^3 + x^2$

- *Multiplication:* $(x^6 + x^3 + x) \times (x^6 + x^2 + 1)$ *modulo* $f(x) = x^3 + x + 1$
- *Squaring:* $(x^6 + x^3 + 1)^2$ *modulo* $f(x) = x^5 + 1$

Let us take a closer look closer at multiplication. First, we must perform polynomial multiplication, such that the coefficients are computed *modulo* 2, as shown below:

$$\begin{aligned}(x^6 + x^3 + x)(x^6 + x^2 + 1) &= x^{12} + x^8 + x^6 + x^9 + x^5 + x^3 + x^7 + x^3 + x \\ &= x^{12} + x^9 + x^8 + x^7 + x^6 + x^5 + x\end{aligned}$$

Then we reduce the result of the polynomial multiplication by dividing it by $f(x)$ and taking the remainder, *i.e.*, compute *modulo* $f(x)$. The necessary polynomial division is shown below:

$$\begin{array}{rrrrrrrrrrrrrr}
 & & & & & & & x^5 & & & +x^2 & +x & +1 \\
\cline{8-14}
x^7+x+1\big) & x^{12} & & +x^9 & +x^8 & +x^7 & +x^6 & +x^5 & & & & +x & \\
- & x^{12} & & & & & +x^6 & +x^5 & & & & & \\
\cline{2-8}
 & & & x^9 & +x^8 & +x^7 & & & & & & & \\
 & & - & x^9 & & & & & +x^3 & +x^2 & & & \\
\cline{4-11}
 & & & & x^8 & +x^7 & & & +x^3 & +x^2 & & & \\
 & & & - & x^8 & & & & & +x^2 & +x & & \\
\cline{5-12}
 & & & & & x^7 & & & +x^3 & & & & \\
 & & & & - & x^7 & & & & & +x & +1 & \\
\cline{6-14}
 & & & & & & & & x^3 & & +x & +1 &
\end{array}$$

It should be noted that polynomial division is extremely costly in terms of computation. Thus, more efficient algorithms exist to perform the reduction operation, and we will elaborate on those algorithms in Section 4.2.

For the squaring example, we have the following:

$$\begin{aligned}(x^6 + x^3 + 1)(x^6 + x^3 + 1) &= x^{12} + x^9 + x^6 + x^9 + x^6 + x^3 + x^6 + x^3 + 1 \\ &= x^{12} + x^6 + 1 \\ &= (x^5)(x^7) + x^6 + 1 \\ &= (x^5)(x + 1) + x^6 + 1 \\ &= x^6 + x^5 + x^6 + 1 \\ &= x^5 + 1\end{aligned}$$

One thing to note is that most of the terms generated in the first step cancel out. Mathematically, this can be explained by observing that

$$(a + b)(a + b) = a^2 + (ab + ab) + b^2$$

where $(ab + ab)$ equals zero because addition is an exclusive OR operation. This concept can be extrapolated to more terms as shown below:

$$(a + b + c)(a + b + c) = a^2 + b^2 + c^2$$

For this reason, binary-field squaring is much less computationally expensive compared to binary-field multiplication.

Although the squaring algorithm for binary fields is fast, a reduction operation must still be performed. Thus, the example above includes a glimpse of the aforementioned fast-reduction techniques. In the fourth step, we are able to substitute $x + 1$ for x^7 because of modular congruency shown below:

$$x^7 + x + 1 \equiv 0 \; modulo \; f(x)$$
$$x^7 \equiv x + 1 \; modulo \; f(x)$$

Finally, it should be noted that although binary-field arithmetic can be efficiently realized in hardware, some protocols, such as the Elliptic Curve Digital Signature Algorithm (ECDSA) still require prime-field mathematics [16]. In the next section, we will introduce a more efficient one-way function based on elliptic curves.

2.1.5 Elliptic Curve Cryptography

The Elliptic Curve Cryptography (ECC) analog of modular exponentiation is scalar point multiplication, which involves *repeated addition-and-doubling* of points on an elliptic curve defined over a finite field. As with modular exponentiation, the reverse operation, known as the Elliptic Curve Discrete Logarithm Problem (ECDLP), is considered intractable.

Elliptic curves are defined by a form of the Weierstraß equation. When prime fields[4] are used as the *underlying field*, K, the elliptic curve equation can be simplified to the following:

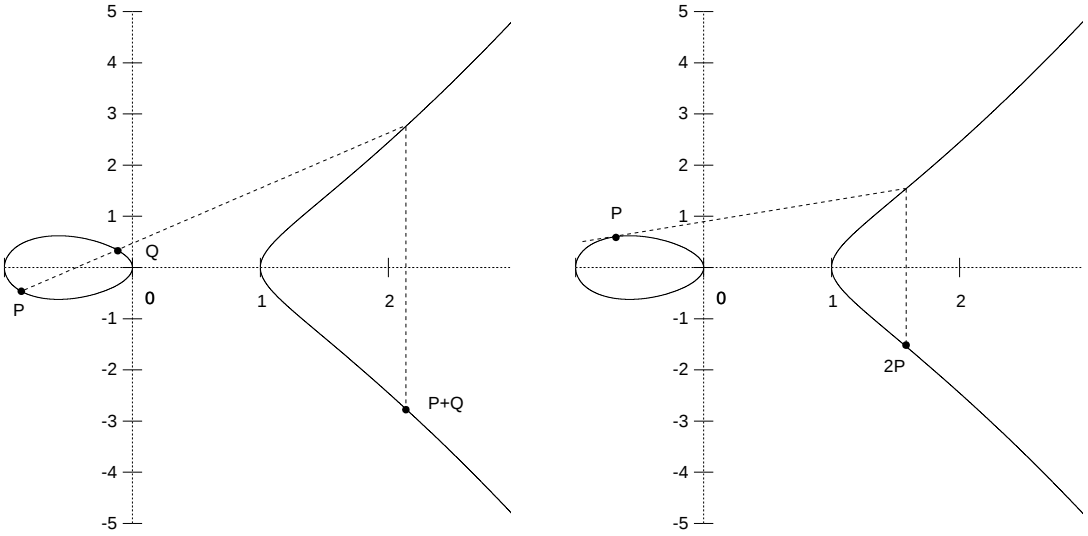
$$E : y^2 = x^3 + ax + b \tag{2.1}$$

where $a, b \in K$ and the discriminant, $\triangle = -16(4a^3 + 27b^2) \neq 0$. For binary fields, the simplified Weierstraß equation is given by:

$$E : y^2 + xy = x^3 + ax^2 + b \tag{2.2}$$

For cryptography, K is a finite field; however, for pedagogical purposes, it is useful to view elliptic curves defined over the set of real numbers, $K = \mathbb{R}$, as shown in Figure 2.3. A graphical representations of point addition on an elliptic curve,

[4] A prime field has a characteristic $\neq 2, 3$.



(a) P+Q (b) 2P

Figure 2.3: Elliptic Curve point addition and doubling on $E(\mathbb{R})$

$E(\mathbb{R})$, is depicted in Figure 2.3a. As shown, P and Q are added by first drawing a straight line through the two points and then locating the third point of intersection between the line, PQ, and the elliptic curve. The point, $P + Q$, is then the x-axis reflection of the third point of intersection. For point doubling, shown in Figure 2.3b, the x-axis reflection of $2P$ is the second point of intersection between the tangent line of point P and the elliptic curve.

The set of points defined on the elliptic curve along with the point addition operation form an *Abelian group*. The unity element for this Abelian group is the point at infinity, ∞. When the line between two points is vertical, the third point of intersection is said to be at infinity. Thus, $P - P = \infty$ where $-P$ is the x-axis reflection of P.

Mathematically, point addition and doubling for prime fields can be described by the equations below such that $A = (x_a, y_a)$, $B = (x_b, y_b)$, and $C = (x_c, y_c)$.

- *Addition:*

$$x_c = \left(\frac{y_b - y_a}{x_b - x_a}\right)^2 - x_a - x_b \tag{2.3}$$

$$y_c = \left(\frac{y_b - y_a}{x_b - x_a}\right)(x_a - x_c) - y_a \tag{2.4}$$

- *Doubling:*

$$x_c = \left(\frac{3x_a^2 + a}{2y_a}\right)^2 - 2x_a \tag{2.5}$$

$$y_c = \left(\frac{3x_a^2 + a}{2y_a}\right)(x_a - x_c) - y_a \tag{2.6}$$

Note that from Eq. (2.2), we can develop slightly different point addition and doubling expressions for binary fields.

Just like modular multiplication and squaring are used by the modular exponentiation algorithm, point addition and doubling are used for the scalar point multiplication algorithm, which is the elliptic curve one-way function. Scalar point multiplication, $Q = xP$, can be computed via the *repeated point add-and-double* method such as *right-to-left binary point multiplication* described in Algorithm 1. The right-to-left binary point multiplication algorithm is nearly identical to the algorithms used for modular exponentiation, except square and multiply have been replaced with double and add, respectively. It should be noted that Algorithm 1 is shown here purely for example sake. Due to its simplicity, it is relatively inefficiently and susceptible to side-channel attacks. In practice, more efficient sliding-window algorithms are utilized, in which more than one bit of the multiplier is scanned at once [16].

As we can see from Eq. (2.3) to Eq. (2.6), each point operation requires a field inversion, which can be up to two orders of magnitude more costly than a field

Algorithm 1 Calculate $Q = xP$ [16]

Input: $P \in E(F_q)$ and integer $x \geq 1$

```
Q ⇐ ∞
while x ≠ 0 do
  if x is odd then
    Q ⇐ Q + P
  end if
  x ⇐ ⌊x/2⌋
  if x ≠ 0 then
    P ⇐ 2P
  end if
end while
return Q
```

multiplication. Rather than using 2-dimensional, *affine* coordinates, we can use 3-dimensional, *projective* coordinates. When using projective coordinates, intermediate field inversions are not necessary. Instead, a conversion into projective coordinates is performed at the beginning of a scalar point multiplication. Then throughout the scalar point multiplication, projective coordinates are used, requiring no inversion operations. Finally at the end of the scalar point multiplication, one inversion is required to map the result back into affine coordinates.

Various projective coordinates have been proposed in literature; however, for prime fields, *Jacobian* coordinates stand out as being the most computationally efficient [16]. The mapping between Jacobian and affine points is $(X, Y, Z) \rightarrow (X/Z^2, Y/Z^3)$ such that $Z \neq 0$, and the point at infinity is represented as $(1, 1, 0)$. To project an affine point onto a Jacobian point, we simply set $Z = 1$. However, to convert a Jacobian point into an affine point, we must perform one field inversion to calculate Z^{-1}. For prime fields, the negative of a Jacobian point, (X, Y, Z), is simply $(X, -Y, Z)$. We can further improve the computational efficiency by using a

mixture of Jacobian and affine points. In particular, we use Jacobian coordinates for the point double operation, but when we perform a point addition, we actually add an affine point to a Jacobian point.

For binary fields, Lopez and Dahab introduced a more efficient coordinate system with the projective mapping of $(X, Y, Z) \rightarrow (X/Z, Y/Z^2)$ and the point at infinity being represented as $(1, 0, 0)$ [17]. Because the additive inverse of an element in a binary field is the element itself, the negative of a point is represented differently compared to the prime counterpart. Specifically, in the Loped-Dahab (LD) coordinate system, the negative of the point (X, Y, Z) is $(X, X + Y, Z)$.

Determining the number of finite-field operations for ECC is not as straightforward as it is for modular exponentiation because each ECC addition and doubling encompasses potentially dozens of finite-field operations. Given the same key size, there is an order of magnitude more field operations for a typical ECC scalar point multiplication compared to an RSA modular exponentiation, but the advantage of elliptic curves over modular exponentiation for asymmetric cryptography is that the ECDLP is considered to be computationally harder than the DLP. Consequently, the size of integers used for ECC is much smaller than that of modular exponentiation-based schemes of equivalent security. *For this reason, ECC is substantially more energy efficient than modular exponentiation schemes for the same level of security and is the only asymmetric cryptosystem evaluated in this study* [8, 13]. Given existing computational capabilities, integer computation in the range of 192-bits to 384-bits maintains adequate security for ECC. To provide similar levels of security, RSA would need 1024-bit to 15360-bit computations [18].

The discussion here on elliptic curve cryptography is in no way intended to be comprehensive. For a more in-depth treatment, we recommend two excellent books dedicated to the subject, the "Guide to Elliptic Curve Cryptography" and the "Hand-

book of Elliptic and Hyperelliptic Curve Cryptography" [16, 19].

2.2 HW/SW Codesign and Computer Architecture

The work presented here investigates the energy efficiency of asymmetric cryptography on an ultra-low power embedded system. The term *embedded system* refers to a special-purpose, System on a Chip (SoC), *i.e.*, a small self-contained computer system. An embedded system is comprised of two major design components, namely hardware and software. The *hardware* is the tangible part, typically containing digital logic gates fabricated on a silicon substrate, while the *software* is the program that orchestrates computation on the hardware. Often, these two components are designed separately; however, hardware/software co-design can yield far more efficient systems.

At the heart of any modern computer system is the *processor core*. Although higher performance systems contain processors with multiple cores, we focus here on a system with a single core. A typical processor core follows the stored-program model, in which it sequentially fetches *instructions* from memory and performs arithmetic and logic operations on data accordingly. The unit of data that is processed by a given instruction is referred to as a *word*. Common word widths for modern processors include 32-bits and 64-bits; however, some embedded systems use 8-bit and 16-bit words. Data words can be stored in either memory or registers, the latter having a much faster access time than the former. A *register* is nothing more than a grouping of logic storage elements knowns as flip-flops. From the programmer's point of view, data and instructions are stored in the same memory but usually in different locations, *i.e.*, a *Von Neumann architecture*.

The *Instruction Set Architecture* (ISA) is the programmers interface to the processor and essentially defines the hardware/software boundary. We can broadly cate-

gorize an ISA into one of two categories: *RISC* or *CISC*. In a Reduced Instruction Set Computer (RISC) architecture, instructions only perform operations on data within registers. Therefore, data is loaded from memory into registers prior to computation and then stored back out to memory after computation. For this reason, RISC architectures are synonymous with *load-store* architectures. A Complex Instruction Set Computer (CISC) architecture, on the other hand, is a *register-memory* architecture, where instructions can operate on data in memory as well as in registers. Common examples of a RISC architecture include MIPS, SPARC, PowerPC, and the ubiquitous ARM, while well-known CISC architectures include Intel's x86 and Motorola's 68000.

One of the primary advantages of a RISC architecture is that the instructions have a simpler, fixed-width format and are therefore easier to decode in hardware. Coupled with the load-store concept, a RISC machine is also easier to pipeline and hence a good choice for an embedded system. This is especially true if code compatibility is not a requirement, which means the designer is not forced to use a legacy CISC ISA. For our work, we chose a subset of the MIPS ISA due mostly to its popularity in the academic community but also partially because of being well supported in the GNU toolchain.

The implementation of an ISA is commonly called the *microarchitecture*. An ISA can have various microarchitectures depending on the generation of the machine and the level of expected performance. For example, the R2000 and R3000 both implement the MIPS-I ISA; however the R3000 is an improvement over the R2000 [20]. Both machines are binary compatible, which implies a program does not have to be modified to run on either machine.[5] The field of computer architecture encompasses

[5]Binary compatibility does not necessary mean a program written for one machine will run well on the other as a certain amount of machine-level software optimization might be present.

microarchitecture design as well as ISA design. Although very little research today is being put towards full ISA design, a hardware/software codesign project such as this will often include both aspects of computer architecture by extending or enhancing an existing ISA.

The microarchitecture of our research processor is based on the classic five-stage pipelined processor taught in many computer architecture classes [21]. In such a design, instruction execution consists of five stages, each normally requiring a single clock cycle. The five stages are described below:

1. *Fetch:* An instruction is read from memory (or an instruction cache, which will be discussed later), and the *Program Counter* (PC), which keeps track of the processor's place in the instruction sequence, is updated.

2. *Decode:* The instruction is decoded, creating control signals that will flow down the pipeline, and the register file is read. Also, hazards (briefly discussed later) are detected and handled.

3. *Execute:* This stage contains the *Arithmetic-Logic Unit* (ALU), which performs an arithmetic or logic operation on the data read from the register file. If the instruction is a load or store instruction, the memory address is calculated. If the instruction is a branch instruction, which changes the control flow of the program, the branch address is determined.

4. *Memory:* If the instruction is a load or store instruction, the memory (or the data cache) is accessed. Instruction exceptions are also handled in this stage. An exception interrupts normal program execution and can be caused by a number things, including an unrecognized instruction, an arithmetic overflow, or even an external notification.

5. *Write-Back:* The register file is updated in this last and final stage. If the instruction was a load, the value read from memory is written into the appropriate register. If the instruction was an arithmetic-logic instruction, the destination register is written.

In an *in-order* microarchitecture, instructions flow through the pipeline in program order, ideally progressing to the next stage every clock cycle. This implies that the ideal throughput of such a machine is one instruction per clock cycle ($IPC = 1$).[6] Real processors, however, do not achieve ideal IPC due to hazards in the pipeline. A *hazard* occurs when an instruction must stall because it depends on the results of an instruction ahead of it in the pipeline. Hazards are a negative effect of instruction execution overlap and must be handled properly to ensure correct execution. We will now briefly describe the hazards possible in our 5-stage pipelined processor and discuss how they are handled.

Three types of hazards exist in a traditional processor: data, control and structural. A *data hazard* exists when an instruction needs the result of another instruction ahead of it in the pipeline. In computer architecture terminology, this presents a *Read-After-Write* (RAW) data hazard.[7] Forwarding logic allows the pipeline to continue without stalling in the case of back-to-back arithmetic-logic instructions so long as each instruction only requires one clock cycle in the execute stage. However, when an arithmetic-logic instruction immediately following a load instruction needs the data being loaded, the pipeline must stall until the load is complete.[8]

A *control hazard* is caused by a branch instruction, which modifies the control flow of the program. The problem is that the processor must update the PC in

[6]More aggressive microarchitectures fetch multiple instructions per clock cycle and execute them *out of order*. These designs can achieve an IPC greater than one.

[7]Other types of data hazards exist but are not relevant for an in-order processor.

[8]Note that some literature uses the term *interlocking* to refer to pipeline stalling.

the fetch stage, but the branch address is not determined until the execute stage. Various techniques have been developed to reduce pipeline stalls caused by control hazards, such as delaying the branch decision and predicting the branch outcome early in the pipeline. The latter technique has proven useful in many microarchitectures, while the former actually complicates the design of modern, more aggressive microarchitectures.

In MIPS, the branch decision is delayed one clock cycle with a branch delay slot. In other words, the instruction immediately following the branch in program order is always executed, regardless of the outcome of the branch. To further reduce the effects of control hazards, we use a simple branch predictor to predict the branch outcome in the decode stage and then verify the prediction in the execute stage. If the prediction was incorrect, the instruction that was speculatively fetched is invalidated, and instruction fetch resumes at the correct branch target address. Of course, if the prediction was correct, the processor simply continues execution uninterrupted.

A *structural hazard* exists when two or more instructions require the same hardware resource in a given clock cycle. In an in-order, pipelined processor such as ours, structural hazards only exist when a hardware resource is needed in two or more pipeline stages. For instance, memory is accessed in the fetch and memory stage. Similarly, the register file is read in the decode stage and written in the write-back stage.

A simple solution to avoid memory structural hazards in a pipelined processor is to store instructions and data in separate memories. This type of architecture, commonly referred to as a Harvard architecture, is in contrast with the Von Neumann architecture. Another solution is to provide multiple ports to the same memory. A memory with two read/write ports is typically called a *dual-port* memory. One disadvantage to such an approach is that the density of the memory goes down as

the number of ports goes up (*e.g.*, a single-port memory requires six transistors per bit, while a dual-port memory requires eight). A third solution is to use separate caches for data and instructions. While many embedded systems use a hybrid of the aforementioned solutions, most computer system today use caches to solve memory structural hazards. Structural hazards caused by the register file are usually avoided with multiple ports. In particular, the register file in our processor has two read ports and one write port.

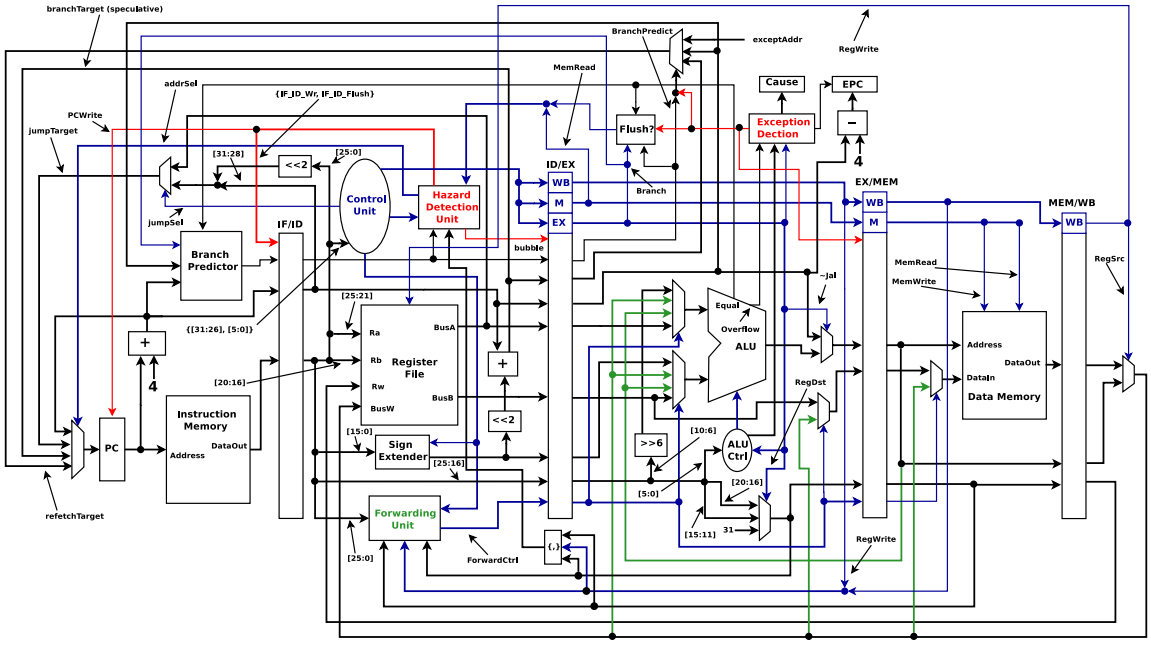


Figure 2.4: The microarchitecture of a 5-stage pipeline processor

Putting all these concepts together, Figure 2.4 depicts the 5-stage pipelined processor design used in our research. The forwarding paths for avoiding data hazards are highlighted in green, while the signals for stalling and invalidating (*i.e.*, *flushing*) instructions are highlighted in red. The control signals that flow through the

pipeline are colored blue and all other components of the datapath are colored black. Note that the term *datapath* describes the data buses, registers, and arithmetic-logic components within the microarchitecture.

For asymmetric cryptography, the width of the datapath (*i.e.*, the word width), specified by the ISA, can have a significant effect on the computational efficiency. As seen in Section 7.9, larger datapaths prove to be beneficial in our accelerator architecture. For MIPS-I and II, the datapath is 32-bits wide, while for MIPS-III and above, the datapath increases to 64-bits. We currently utilize a 32-bit datapath for our processor, but for future work, we would like to investigate the energy benefit of using a 64-bit processor.

As will be discussed in Section 5.3, we investigate the energy impact of an instruction cache in our system. Thus, a brief description of caches is in order. We already mentioned that caches can help eliminate structural hazards due to memory access, but it is interesting to note that caches were developed primarily because the access time of main memory was not keeping up with the speed of the processor. Starting in the 1980s and continuing until about 2005, processor and main memory speed have diverged exponentially [21]. Processor performance has increased drastically due to advances in VLSI and computer architecture. Meanwhile, main memory speed has been growing at a much slower rate as commodity DRAM capacity increases and cost decreases.

To bridge this performance gap, computer architects began placing caches between main memory and the processor. *Cache* is simply smaller, faster memory that leverages the principals of temporal and spacial locality. *Temporal locality* is the observation that when an instruction or data word is accessed in memory, it is very likely to be accessed again in the near future. Similarly, *spacial locality* observes that when a word is accessed in memory, words within close proximity will likely be

accessed as well.

By using faster SRAM technology, rather than slower, more dense DRAM, and keeping the memory small, the access time for a cache can be orders of magnitude less than that of main memory. Initially, processors only had a single level of cache; however as the processor-memory performance gap continued to increase, more levels of cache were added to the memory hierarchy. The access time of the lowest level of cache, *L1*, must be matched with the speed of the processor, requiring only one or two processor clock cycles. Moving up in the memory hierarchy, each level of cache is larger and slower than the level below it. The goal of a well designed memory hierarchy is to give the programmer the illusion of a single memory that is large but also fast. While a typical computer system today will have up to three levels of cache, many embedded systems only use one level of cache or no cache at all.

The simplest of all cache designs, illustrated in Figure 2.5, is the *direct-mapped* cache.[9] In such a cache, a block of data in main memory maps to only one location in the cache. Because the cache is much smaller than main memory, many blocks in memory will map to the same block in the cache. Consequently, a *tag* must be stored with a given cache block in order to uniquely identify that block in memory. In addition to the tag, other bits must be stored to keep track of the state of each cache block. In our simple architecture, we only need to know if the cache block contains valid data; thus a single *valid bit* is sufficient.

One of the advantages of a direct-mapped cache is simplicity. As shown in Figure 2.5, the word address from the processor is broken up into three components, namely tag, index and block offset. The index is used to select a cache block from the cache. Due to spacial locality, a cache block will usually contain multiple words

[9]Modern computer systems use more advanced caching techniques, which lie outside the scope of this work.

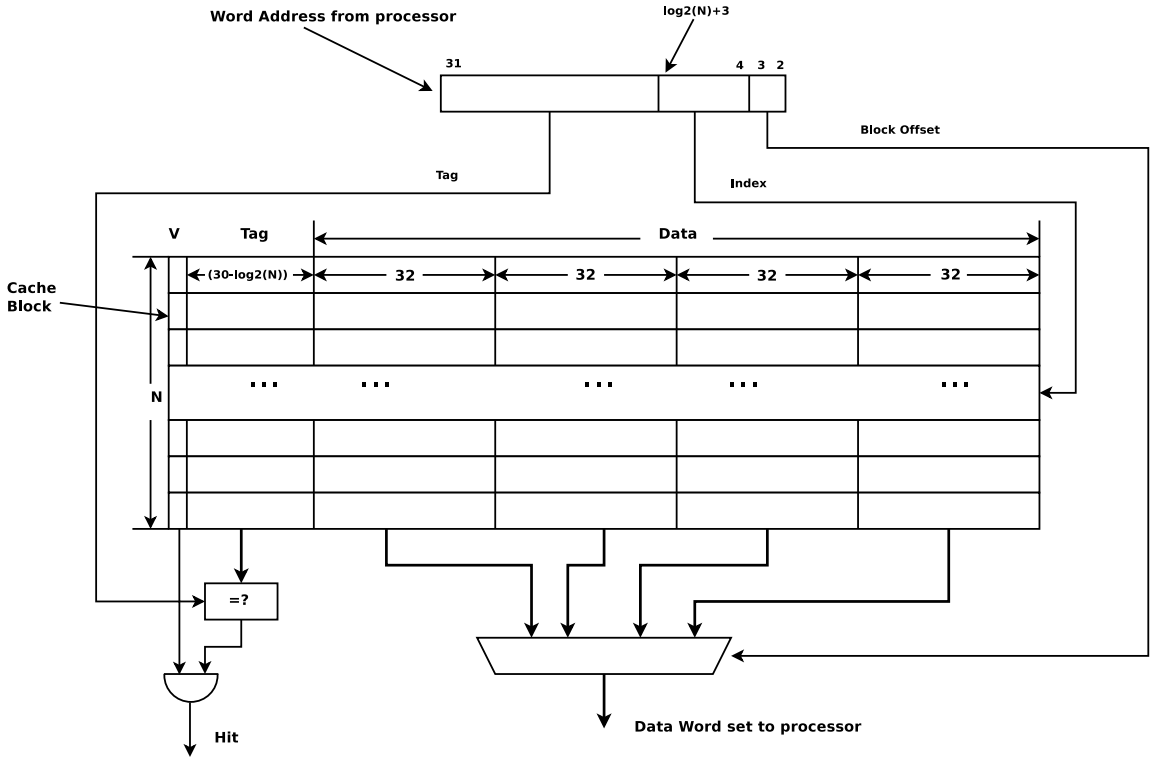


Figure 2.5: A direct-mapped cache with a block size of 16 bytes and a 32-bit word width.

(a power of two for hardware simplicity). Thus, the block offset selects a particular word within a cache block. To ensure the cache block referenced by the index is the exact block in memory the processor is addressing, the tag stored in the cache is compared with the tag portion of the address. Also, the valid bit is checked. If the tags match and the valid bit is set, a cache *hit* is detected, and the appropriate word is forwarded to the processor; otherwise, a cache *miss* is detected, causing the cache controller to access the next level in the memory hierarchy. As a miss is being handled, an in-order processor such as ours, must stall, waiting for the correct cache block to be returned from the memory system. This wait time is commonly referred to as the *miss penalty*. More details about the cache design used in this study can

be found in Section 5.3.

2.3 Energy Consumption in Digital Circuits

Understanding how energy is consumed in CMOS logic is key to creating energy efficient designs. The general equation for energy is given by

$$Energy = Power * \Delta Time \tag{2.7}$$

such that $Power$ is the average computation power, and $\Delta Time$ is the time per operation. While $\Delta Time$ is dependent upon the computation time, $Power$ is dependent upon the CMOS implementation and usage.

CMOS circuits dissipate power in three different ways. First, there is static power dissipation, which can be described by the formula below:

$$P_{static} = V * I_{leak} \tag{2.8}$$

where V is the source voltage and I_{leak} is source to drain current when the transistor is turned off, referred to as leakage current. The second type of energy consumption is switching power, given by the following formula:

$$P_{switching} = (1/2) * \alpha * C * f * V^2 \tag{2.9}$$

C is the capacitance the transistors must drive and is made up of wire and gate capacitance. The clock frequency, f, and the switching activity factor, α, capture the rate at which the transistors switch. The third component of power is short

circuit power and is given by the following formula:

$$P_{short} = V * I_{sc} \tag{2.10}$$

I_{sc} is the short circuit current which exists between the type N and P transistors during a logic state transition [22].

In computing, we can reduce energy per operation by either reducing the power consumed in the computation logic or by reducing the amount of time required per operation. Often, a small increase in power can be traded for a significant reduction in execution time such that there is an overall benefit in energy conservation [23]. Conversely, an increase in execution time might be traded for a significant reduction in power as seen with Dynamic Voltage Frequency Scaling (DVFS) [24].

3. RELATED WORK

Researchers have dedicated much effort to achieving significant acceleration using hardware in FPGA and ASIC designs; however, only a few publications seem to investigate the energy consumption aspect of public-key cryptography for embedded devices. In order for public-key cryptography to be viable in energy-constrained applications, a better understanding of the energy cost associated with asymmetric encryption in both hardware and software is necessary.

Wander *et al.* compared the energy cost of 1024-bit RSA with that of 160-bit ECC to show that 160-bit ECC significantly reduces energy consumption when executed on an 8-bit Atmel ATmega128L microprocessor [8]. The results provide a very compelling argument for ECC, showing that, based on an assumed battery life, the device using ECC could execute 4.2 times the number of key exchange operations. While their work looked at the energy cost for asymmetric cryptography on the far left side of the range shown in Figure 1.1, our work examines its cost for additional points on the spectrum.

Also on the far left of the spectrum, Potlapally *et al.* investigated the energy requirements of OpenSSL on an Intel SA-1110 StrongARM processor [13]. To do so, they devised a LabVIEW based testbed that measures, in real-time, the power consumption of a handheld device with the SA-1110 processor. Their experimental results motivate further research by showing that for 1KB data transfers, asymmetric cryptography consumes greater than 90% of the total energy spent on cryptographic processing. This equates to 56% of the total energy expended during the data transfer. Additionally, they show that 163-bit ECC requires less energy than 1024-bit RSA when client authentication is used. Their work is particularly relevant considering

the SA-1110 is comparable in size to the processor evaluated in our study.

Pabbuleti *et al.* evaluated the energy cost of several public-key authentication schemes based on ECC (ECDSA) and one-time hashes (LD-OTS and W-OTS) [9]. For their experiments, the authors used a CC2500 low-power RF transceiver paired with an MSP430 microcontroller, 256 KB of flash and 16 KB of RAM. To facilitate independent energy measurements, the authors used separate shunt resistors to measure the current independently through the microcontroller and the transceiver. Thus, their evaluation included not only the cost of computing the signature, but also the cost of transmitting the signature. While ECC based protocols require much more energy for computation compared to the hash-based schemes, the hash-based schemes require more energy for transmission because of longer signatures. Unfortunately, the energy cost of computation for ECDSA does not scale well to greater security levels. As a result, 160-bit ECDSA was shown to be more energy efficient at the 80-bit security level compared to LD-OTS and W-OTS, but 256-bit ECDSA was much less efficient at the 128-bit security level.

Keller *et al.* examined public-key energy consumption for FPGAs [25]. First, the design of an entire asymmetric cryptographic processor is explained. Then, the design is implemented on an Xilinx Spartan 3E FPGA and characterized in terms of energy consumption. The processor is capable of utilizing binary or prime finite-fields. For prime-field mathematics, the authors used 192-bit integers, while for binary-field mathematics, the authors used 163-bit polynomials. For energy consumption characterization, the authors kept the bit lengths the same but made various algorithmic changes. They found that the power consumption of the FPGA remained quite constant throughout their experimentation, and thus, the fastest system configuration was also the most energy efficient. In the design by Keller *et al.*, the field size was fixed at synthesis time, placing it on the far right of the spectrum of Figure 1.1. By

contrast, the prime-field accelerator presented here is run-time configurable for up to 521-bit ECC. Furthermore, our work evaluates the energy cost for ASIC technology as opposed to FPGA logic, which presents a significantly different power-performance profile.

Goodman *et al.* compared public-key cryptography on a Domain-Specific Reconfigurable Cryptographic Processor (DSRCP) with previously reported FPGA implementations and a software only implementation on a strongARM [26]. The DSRCP was implemented in a 0.25 μm process technology, and the energy consumption numbers were true measurements. The authors report orders of magnitude lower energy consumption for the DSRCP compared to software and FPGA implementations. For public-key cryptographic algorithms, reconfigurability of the DSRCP is possible, while the energy consumed by the DSRCP is half that of previously reported non-reconfigurable hardware solutions. Because the DSRCP can only perform public-key encryption, it lies on the right side of the diagram in Figure 1.1. Our work investigates more reconfigurable points to the left on the diagram.

Wenger *et al.* compared 192-bit prime- and 191-bit binary-field implementations of ECC in terms of energy consumption on a custom cryptographic processor[27]. Their results show that binary-field computation provides a 2.82 factor improvement in energy efficiency for an ECDSA signature. Specifically, their custom processor, "Neptun," requires only 19.53 μJ for a 191-bit binary-field signature compared to 55.10 μJ for a 192-bit prime-field signature. Results were reported assuming a 1 MHz clock rate and a low-power 130 nm technology node. Even though each architecture was tuned for a particular field, the difference in power consumption between the two is insignificant. Thus, the majority of the energy savings due to binary-field support comes from a reduction in execution time. The authors attribute fast field squaring and a 50% reduction in the number of field multiplications as the primary

benefits of binary-fields. For an ECDSA verification, however, binary-field computation only provides a 1.49 factor improvement in energy efficiency because the twin multiplication algorithm for verification is not as efficient for binary-fields. As with the DSRCP by Goodman *et al.*, the Neptun processor is designed specifically for ECC but maintains a certain degree of reconfigurability. Our work not only investigates more reconfigurable architectures but also covers a larger portion of the design space by evaluating greater security levels.

For symmetric encryption, Wu *et al.* show a 2.25x performance improvement over pure SW with CryptoManiac, which requires $1/100^{th}$ of the area of an Alpha 21264. Although the authors did not investigate energy, we acknowledge that this design would yield a significant reduction in energy per symmetric cryptographic operation. *It should be noted that this work is complementary to ours because symmetric and asymmetric cryptography are used cooperatively.*

4. ALGORITHMS AND SOFTWARE

4.1 ECDSA

In this study, we examine the energy cost for the Elliptic Curve Digital Signature Algorithm (ECDSA), which is a variant of the Digital Signature Algorithm (DSA) that utilizes elliptic curve scalar point multiplication in place of modular exponentiation [10]. We chose ECDSA as our benchmark because it is a standardized elliptic curve-based algorithm found in many protocol implementations, including OpenSSL [18]. Figure 4.1 depicts the computational hierarchy of ECDSA with finite-field arithmetic at the foundation.

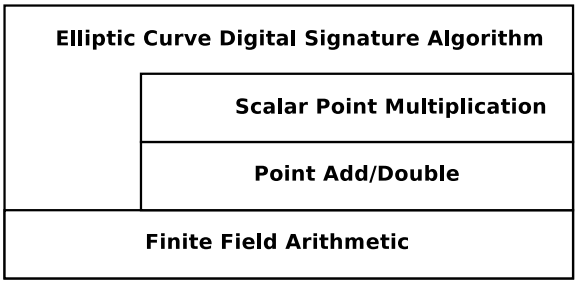


Figure 4.1: Elliptic Curve Digital Signature Algorithm computation hierarchy

Finite-field arithmetic is essentially addition, subtraction, multiplication, and inversion on a finite set of elements. In terms of clock cycles per operation, field inversion is the most costly, with multiplication coming in second. The number of field inversions required is kept to a minimum, however, making multiplication the most costly operation overall. Significantly, when we accelerate ECC, the finite-field

arithmetic is the portion of the algorithm that gets mapped into hardware, while the rest remains in software and is consequently reconfigurable.

Next in the computational hierarchy are the point addition and doubling algorithms that perform mathematical operations on an elliptic curve over a finite field. The underlying field can be either prime – $GF(p)$ – or binary – $GF(2^m)$ – both of which are endorsed by the National Institute of Standards and Technology (NIST). Mathematically speaking, the point double and add operations constitute an Abelian group with the points on the curve and a point at infinity (*i.e.*, the identify element). Although an elliptic curve is described in two dimensions with the Weierstraß equation, practical implementations use a three-dimensional coordinate system to avoid costly field inversions. For our $GF(p)$ implementations, we use mixed Jacobian-Affine coordinates, while for $GF(2^m)$, we use mixed Lopez-Dahab-Affine. These coordinate systems are optimal in that they require the least amount of field operations for their respective curves [16].

Continuing up the hierarchy, we have the scalar point multiplication algorithms. ECDSA defines an operation for signing a message and another operation for verifying the signature of a message. Our study examines the energy cost of both in order to understand the cost of an SSL handshake. An ECDSA signature requires a single scalar point multiplication ($X = kP$), while a verification requires a twin scalar point multiplication ($X = u_1P + u_2Q$). For a single scalar point multiplication, we use a sliding-window algorithm that uses two pre-computed points ($3P$ and $5P$) and takes advantage of the fact that point subtraction is only marginally more costly than addition. For the twin scalar point multiplication, we use an algorithm that pre-computes $P - Q$ and $P + Q$ and then simultaneously scans both multipliers (u_1 and u_2). In such a case, the cost of a twin scalar point multiplication is less than two single scalar point multiplication [28]. We evaluated Montgomery scalar point

multiplication for use with our binary-field coprocessor but found the algorithm to be more costly in terms of performance and energy compared to the sliding-window algorithm [17].

Encompassed within ECDSA is also arithmetic performed modulo the order of the base point of the specified curve and is done in addition to the scalar point multiplications to complete either a signature or a verification operation. For most implementations, the scalar point multiplication constitutes the majority of the ECDSA computation, but as will be shown later, this is not always the case with hardware acceleration. For inversion modulo the group order, we implement the extended Euclidean algorithm on Pete for all hardware/software configurations.

4.2 Multi-precision Routines

Because asymmetric cryptography involves computation on integers typically much larger than the word width of the machine with which they are computed, multi-precision routines are necessary to perform the finite-field arithmetic essential for ECDSA. With multi-precision computation, large integers are stored in memory as arrays of w-bit words, where w is the width of the computational datapath. Multi-precision computation then proceeds one word at time. For the architectures evaluated in this study, $w = 32$. The size of the array necessary to store an n-bit integer is given by $k = \lceil n/w \rceil$. The computational complexity for the multi-precision addition routines are $O(k)$. In other words, the addition algorithm run time is linearly related to the number of words required to represent the field. For multiplication, the computational complexity is $O(k^2)$.

Of the multi-precision routines, inversion and multiplication have the highest computational complexities; however, software acceleration techniques, such as the use of three-dimensional coordinate systems, reduce the number of required inver-

sions [29]. In terms of energy, multiplication is the most costly multi-precision routine. Therefore, we will begin by briefly reviewing the specific multi-precision multiplication algorithms used in this study. Because we evaluated the use of prime and binary fields, our discussion will include both types of computation. For a more in-depth coverage of the material presented in this section, consult Hankerson *et al.* [16].

Algorithm 2 Operand scanning multiplication [30]

Input: $A = (a_{k-1}, ..., a_1, a_0)$, $B = (b_{k-1}, ..., b_1, b_0)$
Output: $P = A * B = (p_{2k-1}, ..., p_1, p_0)$

1: $P \leftarrow 0$
2: **for** i from 0 to $k-1$ **do**
3: $\quad u \leftarrow 0$
4: $\quad$ **for** j from 0 to $k-1$ **do**
5: $\quad\quad (u, v) \leftarrow a_j * b_i + p_{i+j} + u$
6: $\quad\quad p_{i+j} \leftarrow v$
7: $\quad$ **end for**
8: $\quad p_{i+k} \leftarrow u$
9: **end for**

4.2.1 Prime Field Multiplication

Prime-field multiplication requires a multi-precision integer multiplication followed by a reduction operation to map the multiplication result back into the finite field. Multi-precision integer multiplication can be broadly divided into two categories: product scanning and operand scanning. Operand scanning, described in Algorithm 2, is the traditional "school-book" technique, also known as "pencil-and-paper" multiplication. When implemented in software, operand scanning requires a nested for-loop with the inner-loop iterating over the multiplicand and the outer-

loop iterating over the multiplier. Within the inner-loop, the primary arithmetic computation is given by

$$(u, v) = a_j * b_i + p_{i+j} + u \tag{4.1}$$

assuming $P = A * B$. In other words, operand scanning requires a succession of multiply-add operations.

Algorithm 3 Product scanning multiplication [30]

Input: $A = (a_{k-1}, ..., a_1, a_0)$, $B = (b_{k-1}, ..., b_1, b_0)$
Output: $P = A * B = (p_{2k-1}, ..., p_1, p_0)$

1: $(t, u, v) \leftarrow 0$
2: **for** i from 0 to $k - 1$ **do**
3: **for** j from 0 to i **do**
4: $(t, u, v) \leftarrow a_j * b_{i-j} + (t, u, v)$
5: **end for**
6: $p_i \leftarrow v$
7: $v \leftarrow u, u \leftarrow t, t \leftarrow 0$
8: **end for**
9: **for** i from k to $2s - 2$ **do**
10: **for** j from $i - k + 1$ to $k - 1$ **do**
11: $(t, u, v) \leftarrow a_j * b_{i-j} + (t, u, v)$
12: **end for**
13: $p_i \leftarrow v$
14: $v \leftarrow u, u \leftarrow t, t \leftarrow 0$
15: **end for**
16: $p_{2k-1} \leftarrow v$

Product scanning, like operand-scanning, encompasses a nested-loop structure; however, it iterates over the result array in the outer-loop and accumulates the product terms within the inner-loop. For product scanning, described in Algorithm 3,

the inner-loop computation is given by

$$(t, u, v) = (t, u, v) + a_j * b_{i-j} \tag{4.2}$$

such that (t, u, v) is the accumulator register set. In other words, product scanning requires a succession of multiply-accumulate operations. Operand scanning and product scanning require the same number of multiplications; however, when a multiply-accumulate instruction is available, product scanning requires fewer adds and stores to memory. If a multiply-accumulate instruction does not exist in the target architecture, the multiply-accumulate operation must be emulated with multiplies and adds and uses additional registers, thereby diminishing the overall benefit. For our baseline architecture, we found operand scanning to perform marginally better than product scanning. For that reason, we used product scanning only in the case of instruction set extensions.

A number of techniques exist for reducing the result of the multiplication (*i.e.*, the *modulo* operation). The naive approach is to perform a multi-precision division but is far too computationally intense to be considered in practice. Assuming the use of general Mersenne primes selected by NIST, software routines can take advantage of modular congruency in order to reduce a multiplication result using substitutions, additions, and subtractions [31]. For example, consider the prime number used in 192-bit computations:

$$P_{192} = 2^{192} - 2^{64} - 1 \tag{4.3}$$

Due to modular congruency,

$$2^{192} - 2^{64} - 1 \equiv 0 \pmod{P_{192}}$$

$$2^{192} \equiv 2^{64} + 1 \pmod{P_{192}}$$

In this manner, the upper 192-bits of the multiplication result can be folded back into the lower 192-bits. The reduction algorithm for the NIST 192-bit prime is described in Algorithm 4. For completeness, the other NIST primes used in this study are listed below:

$$P_{224} = 2^{224} - 2^{96} + 1 \tag{4.4}$$

$$P_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \tag{4.5}$$

$$P_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 \tag{4.6}$$

$$P_{521} = 2^{521} - 1 \tag{4.7}$$

Note that the terms of the NIST primes have been purposely selected to be multiples of 2^{32}, making the reduction more efficient on a 32-bit processor.[1]

Algorithm 4 NIST fast reduction modulo P_{192} [31]

Input: $C = (c_5, c_4, c_3, c_2, c_1, c_0)$
Output: C modulo P_{192}
1: $s_1 = (c_2, c_1, c_0)$, $s_2 = (0, c_3, c_3)$, $s_3 = (c_4, c_4, 0)$, $s_4 = (c_5, c_5, c_5)$
2: $T \leftarrow s_1 + s_2 + s_3 + s_4$
3: **while** $T \geq P_{192}$ **do**
4: $\quad T \leftarrow T - P_{192}$
5: **end while**
6: **return** T

The drawback to fast reduction is that each field requires a unique NIST reduction algorithm. As a consequence, the NIST reduction techniques are not recommended for hardware implementations. Rather, the preferred method of reduction is Mont-

[1]This is with the exception of P_{521}; however, the limited number of terms keeps the reduction fast.

gomery reduction [32]. When Montgomery reduction is utilized, the reduction steps can be interleaved with the multiplication steps to form Montgomery multiplication. Koç *et al.* provide a comprehensive examination of Montgomery multiplication, in which the Coarsely Integrated Operand Scanning (CIOS) technique, described in Algorithm 5, stands out amongst the rest [33].

The CIOS algorithm uses operand scanning for the multi-precision multiplication and coarsely integrates the Montgomery reduction into the multiplication by performing the reduction on every iteration of the outer loop. The first inner loop (lines 3-7) performs the operand-scanning multiplication, which will create a partial product of length $k + 1$ words, while the second inner loop (lines 8-20) reduces the partial product to k words. Thus, the final result is k words long and congruent to $A * B * R^{-1} \pmod{P}$, where $R = 2^{k*w}$. A final correction step is then necessary to ensure the result is less than the prime, P.

The primary advantage of Montgomery reduction is that the same algorithm can be used for any arbitrary prime; only the algorithm parameters, such as word length, must be configured. For our baseline architecture, we implemented the various multiplication techniques in C++ and evaluated their performance with a 384-bit ECDSA operation. The results showed operand scanning with NIST fast reduction to perform the best with our given HW/SW architecture. We assumed power would remain fairly constant across the various techniques, and therefore selected operand scanning with NIST fast reduction for our baseline software suite.

The instruction set extensions (discussed in Section 5.2) were specifically designed to allow computation with an accumulator, so we compared product scanning with NIST fast reduction to the Finely Integrated Product Scanning (FIPS) Montgomery multiplication using these enhancements. We found that product scanning with NIST fast reduction outperforms FIPS. Thus, our ISA-extended microarchi-

tecture uses product scanning with NIST fast reduction for multiplication. For our fully-accelerated microarchitecture (discussed in Section 5.4), we implemented CIOS Montgomery multiplication in microcode.

Algorithm 5 Calculate $t = MontMult(a, b, n, n'_0)$ (CIOS) [33]

1: $t \Leftarrow 0$
2: **for** i from 0 to $k-1$ **do**
3: $\quad C \Leftarrow 0$
4: $\quad$ **for** j from 0 to $k-1$ **do**
5: $\quad\quad (C, S) \Leftarrow T[j] + A[j] * B[i] + C$
6: $\quad\quad T[j] \Leftarrow S$
7: $\quad$ **end for**
8: $\quad (C, S) \Leftarrow T[k] + C$
9: $\quad T[k] \Leftarrow S$
10: $\quad T[k+1] \Leftarrow C$
11: $\quad C \Leftarrow 0$
12: $\quad m \Leftarrow T[0] * n'_0$ modulo 2^w
13: $\quad (C, S) \Leftarrow T[0] + m * N[0]$
14: $\quad$ **for** j from 1 to $k-1$ **do**
15: $\quad\quad (C, S) \Leftarrow T[j] + m * N[j] + C$
16: $\quad\quad T[j-1] \Leftarrow S$
17: $\quad$ **end for**
18: $\quad (C, S) \Leftarrow T[k] + C$
19: $\quad T[k-1] \Leftarrow S$
20: $\quad T[k] \Leftarrow T[k+1] + C$
21: **end for**
22: **if** $t \geq n$ **then**
23: $\quad$ **return** $t - n$
24: **else**
25: $\quad$ **return** t
26: **end if**

4.2.2 Binary Field Multiplication

Algorithms 2 and 3 assume the target ISA has an integer multiply instruction that performs 32-bit by 32-bit multiplication. Unfortunately for binary field-computation, most processor architectures do not include a carry-less multiplication unit, which drastically reduces the computational efficiency. The naive method for computing binary-field multiplication without hardware support involves bit-serial multiplication such that the multiplier is scanned one bit at a time. In software, this algorithm is impractical due to the cost of shifting and adding the multiplicand. More advanced multiplication algorithms attempt to recover some lost efficiency via precomputation [34]. In other words, some memory space must be traded for increased efficiency.

Algorithm 6 describes the binary-field multiplication used in our software-only evaluation. W here refers the datapath width of the target machine, while w refers to the window width of the algorithm. Window width, w, is the number of bits of the multiplier that are scanned at a time, which dictates the amount of precomputation necessary. We found that $w = 4$ provides a reasonable balance between precomputation storage and performance for an embedded system. Larger window widths would speed up multiplication but require more RAM, thereby increasing power consumption.

As will be discussed in Section 5.2.2, we have extended our target ISA to include carry-less arithmetic instructions such that Algorithm 3 can be efficiently implemented for binary fields. In such case, the need for precomputation is removed and binary-field multiplication has a run time comparable to that of prime-field multiplication. A comparison of the energy per operation when using binary fields as opposed to prime fields can be found in Section 7.3.

As with prime fields, binary fields have a set of NIST recommended fast reduc-

Algorithm 6 Left-to-right comb method with windows of width W [16]

Input: $a(x)$ and $b(x)$ of degree at most $m-1$
Output: $c(x) = a(x) \cdot b(x)$

```
1: Compute B_u = u(x) · b(x) for all polynomials, u(x) of degree at most w − 1.
2: C ← 0
3: for j from (W/w) − 1 to 0 do
4:    for i from 0 to k − 1 do
5:       Add B_u to (C[k], C[k − 1], ..., C[j + 1], C[j]), where u = (u_{w−1}, ..., u_1, u_0)
         such that u_l is bit (wj + l) of A[i]
6:    end for
7:    if j ≠ 0 then
8:       C ← C · x^w
9:    end if
10: end for
11: return C
```

tion algorithms. The follow trinomials and pentanomials are considered for binary reduction polynomials:

$$f(x) = x^{163} + x^7 + x^6 + x^3 + 1 \tag{4.8}$$

$$f(x) = x^{233} + x^{74} + 1 \tag{4.9}$$

$$f(x) = x^{283} + x^{12} + x^7 + x^5 + 1 \tag{4.10}$$

$$f(x) = x^{409} + x^{87} + 1 \tag{4.11}$$

$$f(x) = x^{571} + x^{10} + x^5 + x^2 + 1 \tag{4.12}$$

Unfortunately, the terms in the reduction polynomials do not lie on word boundaries as is the case with the NIST primes. Thus, the reduction algorithms for binary require a significant amount of additional shifting. For example, the reduction algorithm for the 163-bit binary field is described in Algorithm 7. Because binary-field computation does not need to deal with carries, the reduction time for binary is sim-

ilar to that of prime, despite the extra shifting. In our work, we found the reduction for P_{192} to take on average 97 clock cycles, while the reduction for B_{163} takes 100 clock cycles. As a comparison, the run time in clock cycles for Algorithm 3 using ISA extensions is 374 and 376 for P_{192} and B_{163}, respectively. For a listing of the reduction algorithms for the other four binary fields, please see Hankerson *et al.* [34].

Algorithm 7 NIST fast reduction modulo $f(x) = 2^{163} + 2^7 + 2^6 + 2^3 + 1$ [34]

Input: $c(x)$ of degree at most 324
Output: $c(x)$ *modulo* $f(x)$
1: **for** i from 10 downto 6 **do**
2: $T \leftarrow C[i]$
3: $C[i-6] \leftarrow C[i-6] \oplus (T \ll 29)$
4: $C[i-5] \leftarrow C[i-5] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$
5: $C[i-4] \leftarrow C[i-4] \oplus (T \gg 28) \oplus (T \gg 29)$
6: **end for**
7: $T \leftarrow C[5] \gg 3$
8: $C[0] \leftarrow C[0] \oplus (T \ll 7) \oplus (T \ll 6) \oplus (T \ll 3) \oplus T$
9: $C[1] \leftarrow C[1] \oplus (T \gg 25) \oplus (T \gg 26)$
10: $C[5] \leftarrow C[5]$ & 0x7
11: Return$(C[5], C[4], C[3], C[2], C[1], C[0])$

4.2.3 Binary Squaring

One of the more significant advantages that binary-field computation offers is fast squaring. As shown in Section 2.1.4, the result of a binary square operation is simply the result of squaring the individual terms in the input polynomial. In a computer system, this can be accomplished by inserting zeros between the bits of the input operand as illustrated below:

$$(a_{m-1}, \cdots, a_2, a_1, a_0) \rightarrow (a_{m-1}, 0, \cdots, 0, a_2, 0, a_1, 0, a_0)$$

For our software-only system, the binary-field squaring algorithm is accelerated with a precomputed table of 8-bit polynomials and associated 16-bit squares. The algorithm then scans the input operand 8-bits at a time, using each 8-bit window to reference the table. For our ISA-extended system, the carry-less multiplier is used in lieu of a table, allowing for a 32-bit window when squaring.

As with multiplication, the result of a square operation is reduced using the NIST fast reduction algorithms. The computational complexity of these optimized squaring algorithms are $O(k)$ as opposed to $O(k^2)$, making binary-field squaring significantly faster than multiplication.

4.2.4 Field Addition/Subtraction and Inversion

Modular addition and subtraction with prime and binary fields have a computational complexity of $O(k)$, and therefore, have a minimal impact on the efficiency of ECC. Consequently, we will only briefly discuss the algorithms used.

For both prime and binary fields, addition is performed by breaking up the addends into words (*i.e.*, the datapath width) and performing addition at the word level (*i.e.*, multi-precision). Prime-field computation requires integer addition, which means arithmetic carries must be properly handled between word boundaries. Also, arithmetic overflow is possible, so a reduction step is necessary after a prime-field add. The reduction step for addition is simply a condition subtraction of the prime modulus from the result. Prime-field subtraction is similar to addition in which a conditional integer addition of the prime follows a subtraction. For binary-fields, there are no arithmetic carries, which means no reduction step is necessary, and addition and subtraction are the same operation [16].

Field inversion is typically performed with variations of either the extended Euclidean algorithm or Fermat's little theorem. The extended Euclidean algorithm uses

shifts, additions, and subtractions and is $O(k^2)$, while Fermat's little theorem uses modular exponentiation and is $O(k^3)$. Although its computational complexity is lower than Fermat's little theorem, the extended Euclidean algorithm is more challenging to implement in hardware. Thus, we implement field inversion with Fermat's little theorem on our Monte and Billie accelerated architectures and the extended Euclidean algorithm on the rest [16].

4.3 Software Build/Run-time Environment

We used crosstools-ng 1.18.0 to compile our build environment, which includes the GNU Compiler Collection (gcc) 4.7.2 and Binutils 2.23. The executable binaries used for our evaluation were compiled with -O2 and statically linked to Newlib. Unless stated otherwise, the algorithms mentioned here were developed in C++. For the instruction set extensions in Section 5.2 and coprocessor instructions in Section 5.4, we modified the *mips-opc.c* source file to include these supplementary instructions and recompiled Binutils.

The run-time environment for our study was a bare-metal (*i.e.*, no OS) environment representative of a low-power, embedded microcontroller with minimal memory configuration. Instructions and initialization data are read directly out of ROM. A minimal amount of RAM is supplied for stack, heap, and miscellaneous data sections.

5. MICROARCHITECTURES

As discussed in Chapter 3, a significant amount of research and development has been put towards accelerating asymmetric cryptography. As a result, a number of specific microarchitectural enhancements have been proposed in literature. In this study, we evaluate the energy benefit of a few of these enhancements across the spectrum illustrated in Figure 1.1. We will now describe the evaluated microarchitectures in detail, starting with our baseline.

5.1 Baseline

The baseline architecture we modeled, depicted in Figure 5.1, consists of a RISC processor with 256KB of program ROM and 16KB of RAM. The ECDSA software, including the necessary C++ libraries, requires just under 128KB of program memory. Thus, we assume an additional 128KB remains for other embedded system functions. Also, our baseline system does not include an instruction cache. Our initial thoughts were that an instruction cache would be more costly in terms of energy, but as we will see later, we discovered this was not the case.

The RISC processor, from here on referred to as "Pete," is a classic, five-stage, pipelined processor without a Memory Management Unit (MMU). Pete executes a *subset*[1] of the MIPS-II Instruction Set Architecture (ISA) [20]. The program ROM has a 32-bit, dual-port interface to allow simultaneous access for Pete's instruction and data bus. Note that the data bus requires access to the program ROM because the processor must initially copy the data portion of the program into RAM. Moreover, we make use of C++ virtual functions in our software, which require an

[1]The MIPS® unaligned load and store instructions as well as floating point instructions and those related to memory management are not included in Pete's ISA.

occasional table look-up. The RAM in the baseline system is assumed to have only a single 32-bit interface for Pete's data bus because instructions will not be stored in RAM. For both memories, we assume single-cycle access. The clock rate we selected for our evaluation has a period of 3 ns, which is greater than the access time required for all memories, including the program ROM.

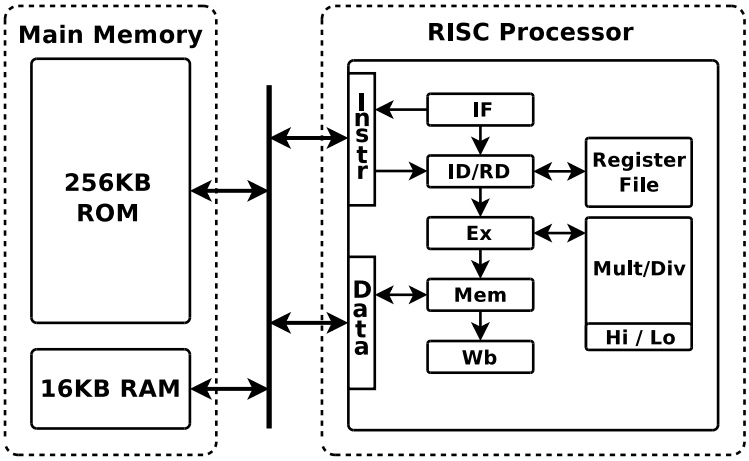


Figure 5.1: Baseline: RISC Processor w/ ROM and RAM

5.1.1 Statically Scheduled Multiply

One particularly unique characteristic of the MIPS ISA is the Hi/Lo register set used for storing multiplication and division results. The use of these registers allows the multiply/divide hardware to lie outside of the *integer* pipeline, as shown in Figure 5.1, and therefore operate in parallel with the integer pipeline. For those unfamiliar with this concept, consider the assembly code below:

```
1 mult $t0, $t1 #initiate t0*t1
  #other independent instructions
3 #may be placed here...
  mflo $t2      #store lower 32-bit result in t2
5 mfhi $t3      #store upper 32-bit result in t3
```

The **mult** instruction initiates the multiplication of $t0$ by $t1$, while the **mflo** and **mfhi** instructions store the lower and upper parts of the 64-bit result, respectively. Therefore, instructions independent of the multiply can be statically scheduled between the **mult** and **mflo/mfhi** instructions. This feature is especially useful for hiding the cost of loop maintenance for tightly-nested arithmetic loops. The multi-precision integer algorithms required for asymmetric cryptography particularly benefit from this light instruction-level parallelism.

Fast, parallel multiplication found on many high-performance RISC cores is costly in terms of area and power [35]. To alleviate the cost of Pete's 32-bit multiplier, we designed a multi-cycle multiplication unit using only a single half-word parallel multiplication block. After examining the assembly output for our multi-precision integer routines, it became clear to us that the compiler effectively schedules instructions to take advantage of the instruction-level parallelism that this architecture can provide. For this reason, we were able to increase the multiplication latency to four clock cycles without significantly affecting the execution time of the multi-precision multiplication routines.

5.1.2 Karatsuba Multiplier Implementation

To further reduce dynamic power, we based our multi-cycle multiplication unit on Karatsuba's divide-and-conquer technique, described by Großschädl *et al.* [36].

$$\begin{aligned} P =& (A_H * B_H) * 2^n \\ &+ [(A_H - A_L) * (B_L - B_H)] * 2^{n/2} \\ &+ (A_L * B_L) \end{aligned} \tag{5.1}$$

Equation 5.1 expresses Karatsuba multiplication mathematically, such that P is the product and A_H, A_L, B_H, B_L represent the input operands, A and B, split into high and low parts. The principal advantage of Karatsuba multiplication is that only three half-word multiplications are needed, as opposed to four with operand or product scanning methods. It should be noted that the term enclosed by square brackets in (5.1) can be less than zero, so Karatsuba multiplication introduces signed arithmetic within an unsigned computation. If the multiplication unit is expected to handle signed as well as unsigned multiplication, which was the case for our work, then this will not require an exorbitant amount of extra logic when compared to other techniques. Figure 5.2 depicts the multi-cycle multiplication unit used in our baseline architecture. As shown, the primary arithmetic components of our Karatsuba multiplier include a 17-bit by 17-bit signed parallel multiplication block, a four-port 49-bit adder, and two 16-bit subtraction units.

Although integer division is not necessary for the algorithms used in this study, we included a small divider in our evaluation. We found that it was necessary for debugging and felt it might be necessary in some of the aforementioned application areas. Moreover, Pete's integer divider uses a simple binary restoring technique, so it consumes only a small percentage of the overall logic resources and does not

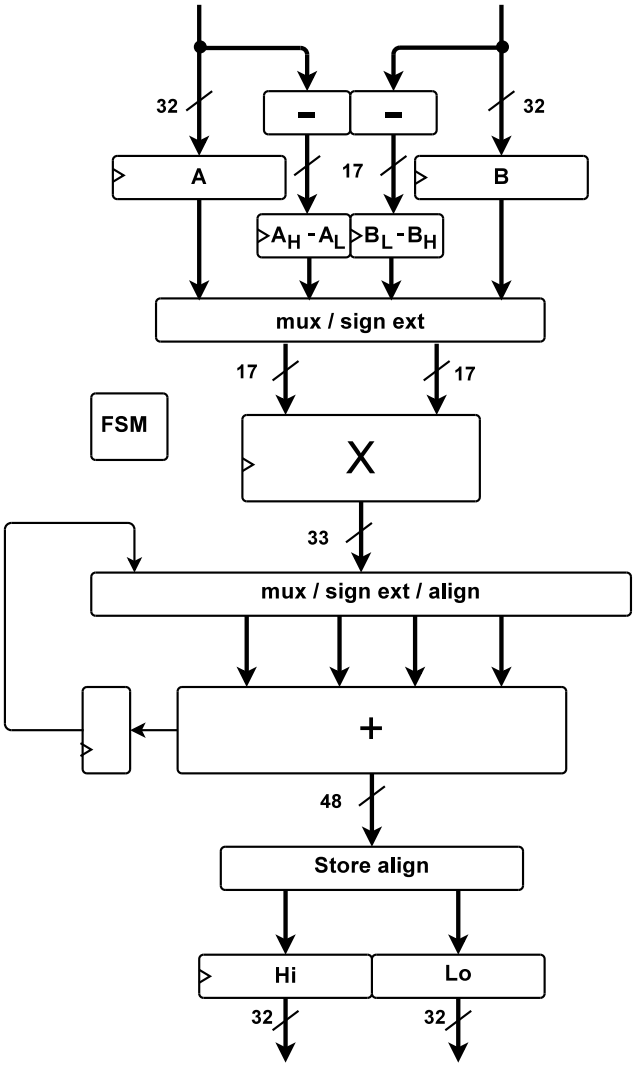


Figure 5.2: The Karatsuba Multiply Unit within the baseline architecture.

significantly impact energy consumption [35].

5.2 ISA Extensions

Instruction set extensions are special purpose instructions built into an existing ISA in order to enhance the execution of a particular algorithm. For many applications, including DSP, communications, and cryptography, these special purpose instructions have shown considerable speedup with very little additional overhead. We

consider instruction set extensions a "middle-of-the-spectrum" acceleration technique and therefore feel they warrant consideration in our comparison study. Großschädl *et al.* extensively explored the use of instruction set extensions for public-key cryptography on various RISC platforms, including MIPS and SPARC V8 [36, 30]. Their research covers both $GF(p)$ (prime finite fields) and $GF(2^m)$ (binary finite fields). In our work, we started with support for only $GF(p)$ and then added $GF(2^m)$ support later [37].

Table 5.1: Instruction set extensions for prime fields. Adapted from the work of Großschädl *et al.* [30].

Format	Operation	
MADDU	rs, rt	(OvFlo, Hi, Lo) ← (OvFlo, Hi, Lo) + rs * rt
M2ADDU	rs, rt	(OvFlo, Hi, Lo) ← (OvFlo, Hi, Lo) + 2 * rs * rt
ADDAU	rs, rt	(OvFlo, Hi, Lo) ← (OvFlo, Hi, Lo) + (rs << 32) + rt
SHA		(OvFlo, Hi, Lo) ← (OvFlo, Hi, Lo) >> 32

5.2.1 Prime Fields

For prime fields, Großschädl *et al.* recommend four supplementary instructions for accelerating all variations of product scanning multiplication (*e.g.*, Comba and FIPS). These instruction set extensions are summarized in Table 5.1. One thing to note is the expansion of the Hi/Lo register set to include a third 32-bit register referred to as the OvFlo register. Those familiar with the MIPS ISA might notice that the **maddu** instruction is actually available in later versions of the MIPS ISA. The difference here is support for higher precision accumulate operations necessary for product scanning multiplication. The M2ADDU instruction is an optimization specifically for squaring, while the **addau** instruction improves the performance of

the FIPS Montgomery multiplication algorithm and potentially the NIST reduction algorithms. The SHA instruction is needed for all variations of the product-scanning algorithm and facilitates access to the OvFlo register [30].

The suggested ISA extensions needed only a minimal amount of modification to our baseline microarchitecture. Aside from extra decode logic in the main pipeline, most of the modifications were concentrated within the Karatsuba multiplication unit. For example, the four-port adder was widened to 50-bits, and extra internal carry bits were added. The multiplexing logic was modified to support extra data paths from the result registers (for accumulate) and the operand registers (for the ADDAU instruction). Additionally, result shifting and stores into the OvFlo register were added. Figure 5.3 depicts Pete's multi-cycle multiply-accumulate unit with the ISA extension modifications highlighted. It should be noted that the multiplication block remained untouched.

5.2.2 Binary Fields

Recall from Section 2.1.4 that binary-field arithmetic is essentially carry-less computation, *i.e.*, add is simply a bitwise XOR. Because most instruction sets include an XOR instruction and carry-less addition does not require a reduction operation, binary-field addition in software is much faster than its prime counterpart. Unfortunately, the same is not true for multiplication because most instruction sets do not include support for a carry-less multiplication. Consequently, computationally inefficient methods such as Algorithm 6 must be utilized for binary-field multiplication. As will be shown in the next chapter, this fact alone renders software only implementations of binary-field ECC impractical for most embedded processors. In such a case, ISA extensions can provide a dramatic improvement.

For binary-field support, Großschädl *et al.* recommend only two additional in-

structions, summarized in Table 5.2. The first instruction, **mulgf2**, is a 32-bit by 32-bit carry-less multiply, *i.e.*, the binary-field equivalent of the **mul** instruction in MIPS. Notice that we represent this operation with $\otimes$. The second instruction, **maddgf2**, is a carry-less multiply-accumulate instructions, *i.e.*, the binary-field equivalent of the **maddu** instruction in Table 5.1. Here we use $\oplus$ to mean binary add.

As with prime ISA extensions, we had to modify Pete's instruction decode unit and the Karatsuba multiply-accumulate unit. The modifications to the Karatsuba multiply-accumulate unit are highlighted in Figure 5.4. The most notable change is the inclusion of a 16-bit by 16-bit carry-less multiplication unit. Rather than overcomplicating the design with a signed multiplication block that also supports carry-less multiply, we chose to multiplex between the two multiplications units depending on the computation mode. This decision was partially influenced by our FPGA prototyping. In an FPGA, integer multiplication is made efficient via hardware multiplication blocks built into the reconfigurable logic. However, an unusual multiplication unit that also supports carry-less multiply would not synthesize to these built-in primitives. We even experimented with various lightweight parallel multiplication techniques in a Virtex-5 but found them to be far too costly in terms of FPGA resources to be practical.

For the four-port addition unit, we designed a dual-mode adder that supports normal addition and carry-less addition. We used a similar design for the 16-bit subtraction units at the top of Figure 5.4. Fortunately, no other modifications to the datapath were required. For the top-level FSM, we had to add a control signal that selects the correct computation mode.

Table 5.2: Instruction set extensions for binary fields. Adapted from the work of Großschädl *et al.* [30].

Format	Operation	
MULGF2	rs, rt	(OvFlo, Hi, Lo) ← rs ⊗ rt
MADDUGF2	rs, rt	(OvFlo, Hi, Lo) ← (OvFlo, Hi, Lo) ⊕ rs ⊗ rt

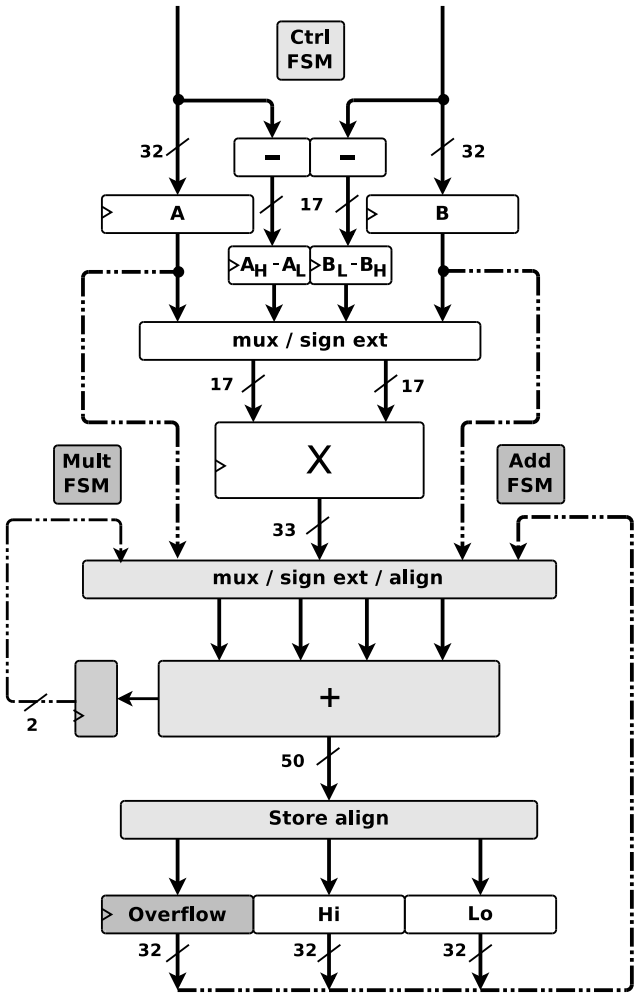


Figure 5.3: The Karatsuba Multiply-Accumulate Unit including support for prime-field ISA extensions. The dashed lines represent data paths that have been added or modified to accommodate these ISA extensions. Lightly shaded boxes signify modified components, while darkly shaded boxes indicate additional components.

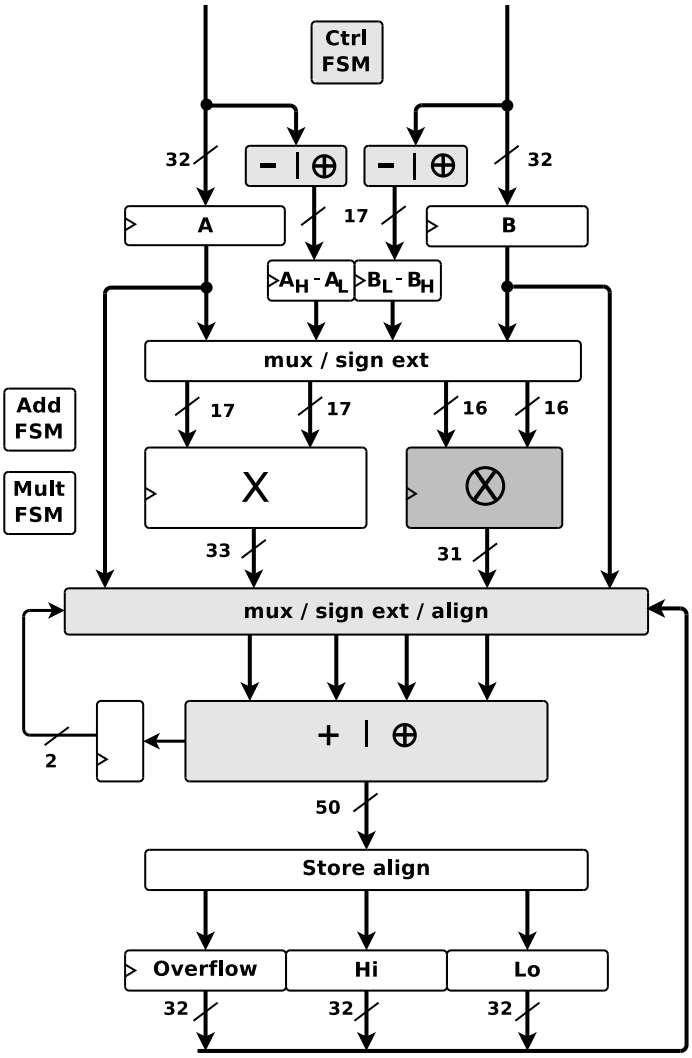


Figure 5.4: The Karatsuba Multiply-Accumulate Unit including support for prime- and binary-field ISA extensions. Lightly shaded boxes signify modified components, while darkly shaded boxes indicate additional components.

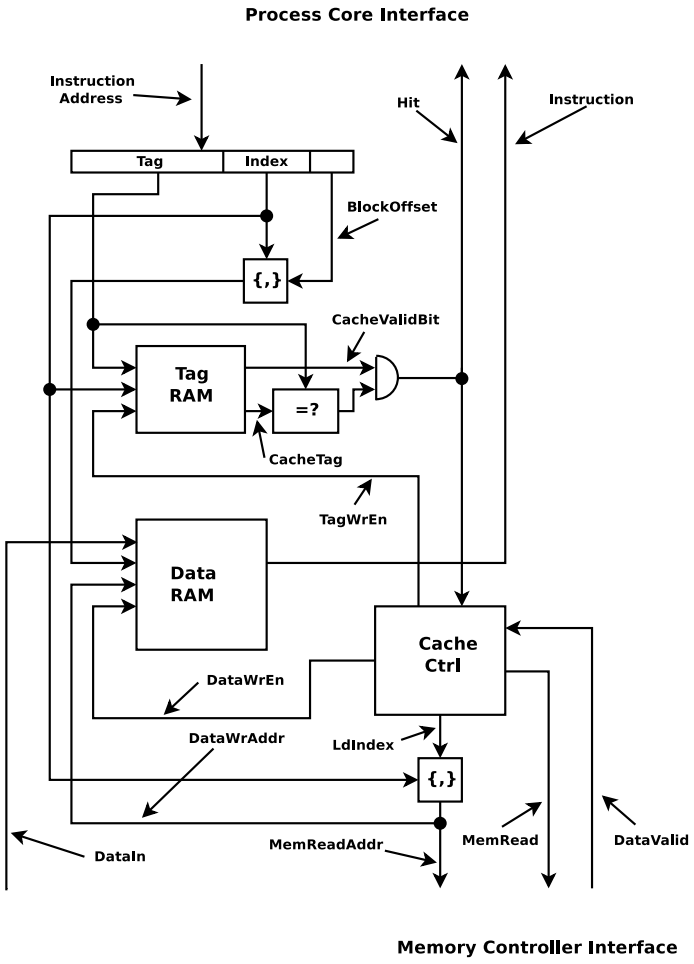


Figure 5.5: The implementation of a direct-mapped instruction cache. Notice the tag and data are stored in different memories. Also, the valid bits are stored with the tag.

5.3 Instruction Cache

A surprising result we encountered while evaluating our baseline microarchitecture is the significant energy cost for instruction fetch from the program ROM. Three factors contribute to the relatively high energy cost of instruction fetch. First, a RISC processor such as Pete fetches an instruction from memory on every clock cycle, causing a large number of reads from program ROM. Second, the energy cost per read of a memory is dependent on the size of memory, so larger memories consume more energy. Finally, compared to the other memory components in the system, the program ROM is the largest by far.

In an effort to reduce this energy cost, we first modeled our system with an ideal 4KB direct-mapped instruction cache, using energy estimates from Cacti, a cache modeling tool developed by HP Labs [38]. In the ideal case, the instruction cache never misses, so the energy cost for instruction fetch only considers reads from the cache. Although this scenario is unrealistic, it provides a simple way of estimating the best case energy benefit we could expect from adding an instruction cache. Our preliminary results, discussed in detail in Section 7.5, showed close to a 50% improvement in overall energy with an ideal instruction cache for the baseline and ISA extended microarchitectures. The Monte accelerated architecture, which will be discussed next, showed far less improvement because instruction fetch is far less dominant in terms of energy.

5.3.1 Cache Implementation

Next, we designed a real instruction cache in Verilog. We chose to implement the simple direct-mapped cache, conceptually described in Figure 2.5. The cache blocks in our design hold four 32-bit words each, *i.e.*, are 16 bytes wide. The number of cache lines, however, is parameterizable, which allows us to change the size of the

cache prior to synthesis. The hardware block diagram for our cache is depicted in Figure 5.5. In the conceptual diagram, an entire cache line, which includes the valid bit, tag and data block, is stored in a single memory. However, in practice, the tag and data components of a cache line are typically stored in separate memories. One advantage of using separate memories for tag and data is that the data memory can also select the appropriate word. In other words, the data memory and word-select multiplexor shown in the conceptual diagram can be combined. Because memories are custom designed, rather than synthesized, this results in a faster, more efficient cache. Another motivation for separating out the tag and data is that in a more complex system, such as a multi-core processor system, the tag must be read more often than the data.[2] In such cases, the tag memory is either dual-ported or duplicated altogether.

In Figure 5.5, we show the processor core interface at the top and the interface to ROM at the bottom. The instruction address from the processor is broken up into three components: tag, index, and block-offset. The data memory is addressed with a concatenation of the index and the block offset, while the tag memory is addressed with just the index. A cache hit is detected by comparing the address tag to the tag read from the tag memory, *i.e.*, the cache line tag. If the tags match and the valid bit is set, a hit is signaled to both the cache controller and the processor. In the case of a hit, the processor will assume the instruction being read from the data memory of the cache is correct and will begin fetching another instruction. Likewise, the cache controller will remain idle. In the case of a miss, the processor will stall the front end of the pipeline (commonly referred to as a pipeline *slip*), and the cache controller will begin access to the program ROM. Once the appropriate memory block has returned from the program ROM, the cache line will be stored in the

[2]More tag reads are required for cache coherency protocols [21].

cache, overwriting the existing cache line. In our implementation, this is carried out with the write-enable signals going from the cache controller to the cache memories. After the store operation is complete, the hit signal will be updated and the processor will continue fetching.

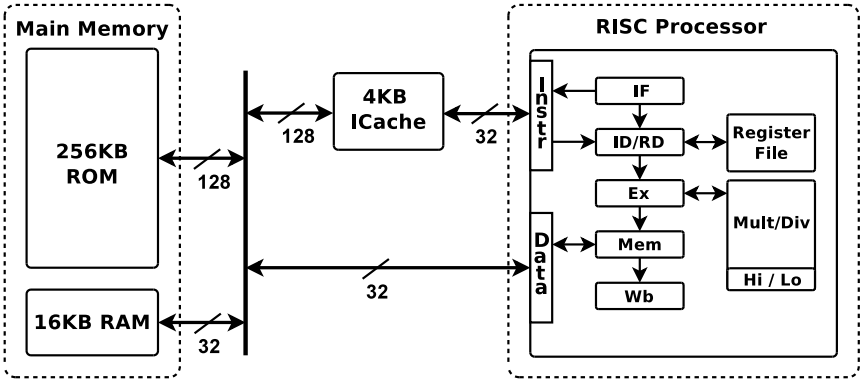


Figure 5.6: Pete with an instruction cache. Notice the interface to the program ROM has been increased to 128-bits.

5.3.2 System Integration

The simplified top-level microarchitecture with the instruction cache is shown in Figure 5.6. Not only did we add an instruction cache, but we also expanded the program ROM port to 128-bits, which allows an entire cache line to be filled at once. In an SoC, the program ROM is fabricated on the same silicon die as the processor logic, making wider ports to memory far less expensive in terms of energy compared to off-chip memory. The primary advantage of a 128-bit program ROM port in our system is a decrease in the miss penalty, which ultimately decreases the energy wasted while Pete is waiting for the correct cache block. To reduce the number of wires and further reduce the cost of ROM access, we made the ROM single-ported.

The changes to the ROM interface and the inclusion of an instruction cache require a slightly more complicated memory system. The data bus from Pete still needs 32-bit access to the program memory. Likewise, we have to support instruction access to non-cached regions in memory. When a processor system is brought out of reset, the cache is in an unknown state and therefore unusable until the processor initializes it. This implies that the reset vector (the location in memory where the processor begins fetching instructions after reset), must be in a non-cached region of memory. At the end of the reset routine, the instruction cache is initialized, which involves invalidating each entry in the cache. From there, the processor jumps to the pre-main routine in the cached address space, where it begins initializing the software environment.

To make all this work, we added data and instruction buffers to transition from a 128-bit memory port to a 32-bit bus. Furthermore, we included arbitration in our ROM controller in order to multiplex the single port. This means that the data and instruction buses as well as the instruction cache must content for access to the program ROM. Although this presents a structural hazard in our system, it has no noticeable impact on performance once the software system has been initialized.

5.3.3 Prefetching

In addition to different cache sizes, we also considered prefetching in our energy evaluation. In the context of caching, *prefetching* is a technique that attempts to reduce cache misses by speculatively fetching cache lines from main memory just prior to their use. A perfect prefetcher would always fetch the correct cache line early enough so that it is ready for use when the processor needs it and would not fetch caches lines that are not used. Fortunately, instruction access is largely sequential, which means it exhibits a lot of spacial locality and is therefore easy to

predict. Thus, prefetching improves the performance of our instruction cache and has the potential to save energy in the process.

We implemented a simple prefetching technique similar to the *stream buffer* proposed by Norman Jouppi [39]. The stream buffer takes advantage of pipelined memory systems by prefetching a stream of cache lines rather than just one at a time. Our prefetcher is essentially a single-entry stream buffer because we assume non-pipelined access to the program ROM. It works as follows: When the cache detects a miss, the cache controller will read the requested cache block from ROM and place it in the cache as normal. Then it will immediately read the next block in the address space and place it in a prefetch buffer along with a tag that identifies that particular block. For every instruction reference from the processor, the address tag is also compared to the tag in the prefetch buffer. If there is a miss in the cache but a hit in the prefetch buffer, then the prefetch buffer will forward the requested block to the processor and write it into the cache at the same time. The cache controller will simultaneously read the next block from ROM and place it in the prefetch buffer. As long as the prefetch buffer always contains the next need cache block, the processor never has to stall. Also, the prefetch buffer avoids polluting the cache with blocks that may never get used.

5.4 Prime-field Accelerator

Continuing towards the right of the spectrum shown in Figure 1.1, we augmented our microarchitecture with an accelerator designed specifically for $GF(p)$ arithmetic. Figure 5.7 depicts the top-level diagram of our microarchitecture with the $GF(p)$ accelerator, referred to as "Monte," on the left, the memory in the center, and Pete on the right. Similar to work described by Koschuch *et al.* [40], Pete and Monte utilize a shared memory interface in order to reduce any bottlenecks that might be

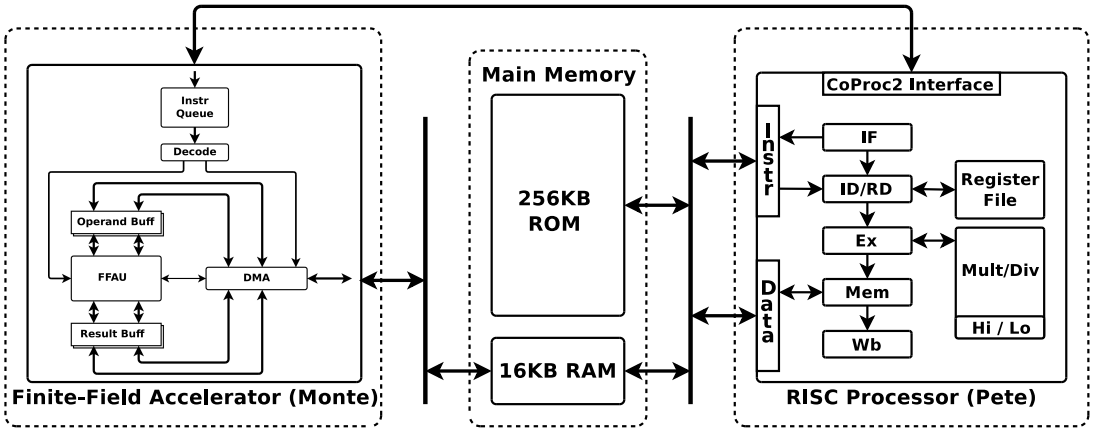


Figure 5.7: The prime field accelerated architecture, "Pete with Monte."

Table 5.3: Coprocessor 2 Instructions used to control Monte

Format	Description	Operation
CTC2 rt, rd	Move to Control Register	cop2CR[rd] ← GPR[rt]
COP2SYNC	CoProcessor 2 Sync	Sync Operation
COP2LDA rt	Load A into Operand Buffer	OpBuff[A] ← MEM[GPR[rt]]
COP2LDB rt	Load B into Operand Buffer	OpBuff[B] ← MEM[GPR[rt]]
COP2LDN rt	Load N into Operand Buffer	OpBuff[N] ← MEM[GPR[rt]]
COP2MUL	Modular Multiply	ResultBuff ← A * B mod N
COP2ADD	Modular Add	Result ← A + B mod N
COP2SUB	Modular Subtract	Result ← A - B mod N
COP2ST	Store Result into Memory	MEM[GPR[rt]] ← ResultBuff

created with a bus interface. Hence, we extended the 16KB of RAM found in our baseline architecture to a true dual-port memory to which both Pete and Monte can read/write.

5.4.1 Coprocessor Interface

To coordinate communication between Pete and Monte, we use the coprocessor interface defined in the MIPS architecture; specifically we modified Pete to include Coprocessor 2 instructions (Table 5.3) for the command and control of Monte. The

first instruction, **ctc2**, allows Pete to initialize the control registers within Monte. As will be discussed later, these control registers allow run-time configuration of the algorithms executing within Monte. The second instruction, **cop2Sync**, facilitates synchronization between Pete and Monte, typical of any parallel processing system.

Instructions **cop2ldA**, **cop2ldB**, and **cop2ldN** initiate Direct Memory Access (DMA) transfers from shared memory to operand buffers within Monte. The start address in shared memory of A, B, and N is contained within Pete's General Purpose Register (GPR) *rt*. Instructions **cop2mul**, **cop2add**, and **cop2sub** initiate modular multiply, add, and subtract, respectively. The result of the above computation instructions is copied back into memory by the **cop2st** instruction, which initiates a DMA transfer from the result buffers within Monte out to shared memory. It should be noted that the above instructions are multi-cycle instructions with latencies dependent on the size of the finite field.

The instructions described in Table 5.3 are fetched and decoded by Pete as regular instructions. Within the decode stage, they are identified as coprocessor instructions and are forwarded to Monte when they reach the execute stage. As shown in Figure 5.7, instructions are placed within an instruction queue once passed to Monte and decoded in instruction order. Similar to out-of-order execution, the coprocessor instructions are dispatched to one of two functional units. The Finite-Field Arithmetic Unit (FFAU), described in detailed shortly, is responsible for modular addition, subtraction, and multiplication, while the DMA handles data movement between the FFAU buffers and shared memory. The loading of operands and storing of results are overlapped with computation via a double buffering scheme. Similarly, operand data is buffered separately from result data to increase buffer bandwidth and avoid unnecessary stalls in the arithmetic logic, while at the same time not demanding more ports per buffer. To reduce the number of reads from shared memory, we have

included forwarding paths from the result buffer to the operand buffer. Consider the snippet of code below to understand how Monte reorders instruction execution:

```
1 #assume monte is initially idle...
  cop2ldA $a1 #load A from address GPR[a1]
3 cop2ldB $a2 #load B from address GPR[a2]
  cop2ldN $a3 #load N from address GPR[a3]
5 cop2mul       #A*B mod N
  cop2st  $a0 #must wait until mul done
7 #instructions below do not depend on previous
  cop2ldA $t0 #can run ahead of store!
9 cop2ldB $t1 #same here
  cop2add       #A+B mod N
11 cop2st  $t3 #must wait until add is done
  cop2ldA $t3 #must be forwarded during store
13 cop2ldB $s0 #can run ahead of store
  cop2sub       #A-B mod N
```

At line 2, the load instruction will be immediately dispatched to the DMA and a transfer will be started during the next clock cycle. Meanwhile, the next instructions will be queued because the instruction at line 3 is not able to dispatch until the current DMA transfer is complete. After instruction 4 dispatches, a DMA transfer will be started and instruction 5 will dispatch to the FFAU. Instruction 5 will *not* issue (*i.e.*, start execution) until the current DMA transfer has completed. Once instruction 4 finishes, Monte will swap operand buffers, and instruction 5 will begin executing. At the same time, instruction 6 will dispatch to the DMA, where it will wait in a reservation register until instruction 5 completes. Note that the DMA functional unit contains a reservation register for stores. Loads, however, are initiated upon dispatch, so a reservation register is not necessary.

Instruction 6 will wait in the store reservation register until instruction 5 completes. In the mean time, instructions 8 and 9 will be processed while instruction 5 continues to execute. The multiply is an expensive operation, so instruction 10 will be held up in the instruction queue until the multiplication completes. When instruction 5 finally finishes, the result buffer will be swapped, and instruction 10 will be dispatched. On the next cycle, instruction 10 will begin because its operands have already been loaded, and instruction 6 will begin storing the multiplication result out to memory. Once instruction 6 has completed, the store instruction at line 11 can dispatch into the reservation register, where it will wait until the add completes. In the meantime, instruction 12 will dispatch. Now, instruction 12 will cause a Read-After-Write (RAW) hazard with instruction 11. Instead of executing instruction 12, a forwarding bit in the DMA unit will be asserted, and instruction 12 will be discarded. Instruction 13 will then dispatch and begin a transfer on the next clock cycle because it does not pose a RAW hazard. Once instruction 10 and 13 complete, the DMA will begin storing the add result out to shared memory, while at the same time, copying the data into operand A. Instruction 14 will dispatch but cannot start until the store has completed the forwarding operation.

5.4.2 Prime-field Arithmetic Unit

For accelerating prime finite-field arithmetic, we designed a microcoded Finite-Field Arithmetic Unit (FFAU). A zoomed in view of the FFAU in Figure 5.8 reveals that the major components of our accelerator include an arithmetic core, multiplexing logic, address logic, and a control unit. The arithmetic core is a flexible, 2-stage pipelined multiply-add unit, which is capable of performing various combinations of adds and multiplies depending on input control bits. Flip-flops within the arithmetic core store intermediate carries to allow for efficient pipelining of the back to back

multiply-adds required by the multi-precision arithmetic. The address logic is nothing more than a few index registers, which generate the operand and result addresses in parallel with the computation. The multiplexing logic provides the FFAU with enough flexibility to compute the CIOS Montgomery multiplication algorithm.

The control unit contains a 64-entry microcode table, along with built-in hardware for nested loop structures and other conditional branches. In an attempt to balance the trade-off between performance and reconfigurability, the control unit contains a set of control registers, programmable by the **ctc2** instruction. Precomputed algorithm parameters as well multi-precision integer width must be preloaded into Monte prior to use. A return address register has been included to allow subroutine calls (leaf functions only). The following sections will describe the design of our FFAU in more detail.

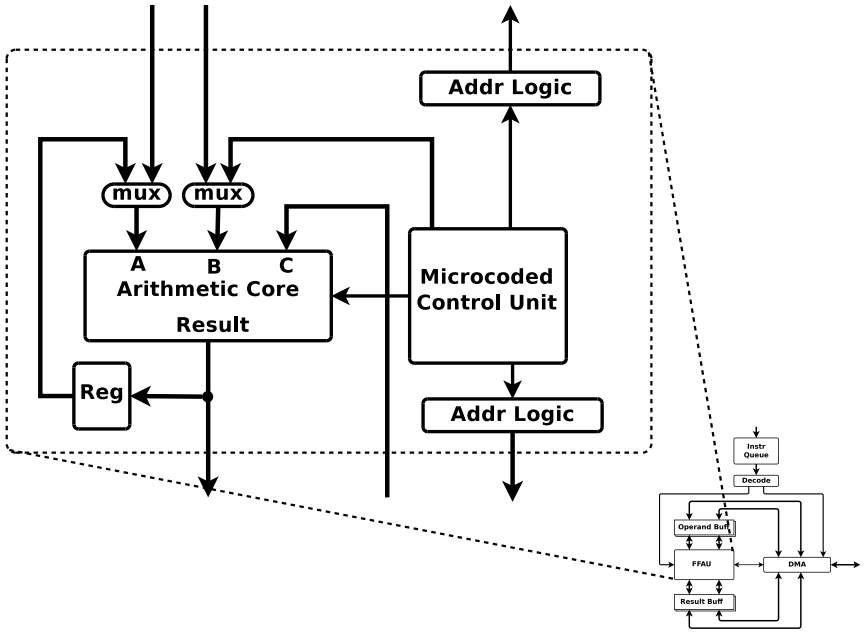


Figure 5.8: The Finite-Field Arithmetic Unit at the center of "Monte"

The intent of the FFAU is to accelerate the underlying mathematics required for ECC. In order to fully explore the design space associated with this sort of acceleration hardware, the HDL code has been written such that the width of the data path and the size of internally addressable memory can be adjusted prior to logic synthesis. Other design parameters such as the size of the microprogram can be adjusted as well. However, for the analysis provided in this study, those design parameters are held constant unless stated otherwise.

5.4.2.1 Hardware Design

The microarchitecture depicted in Figure 5.9 is capable of executing CIOS Montgomery multiplication, described in Algorithm 5, along with modular addition and subtraction. Recall from Section 4.2.1, the CIOS algorithm consists of two nested for-loops. The first loop computes the following:

$$t = t + a * B$$

such that t is an integer with a word length of $k + 2$, a is an integer with a word length of k, and B is of unity word length and part of b, an integer of the same size as a. Note that if l is the bit length of the finite-field elements (*e.g.*, 192, 256, or 384) and w is the bit width of the datapath, then $k = l/w$. For example, if we want to process 192-bit integers (minimal security) with a 32-bit datapath, then each integer, a and b, will be represented by $k = 6$ words. The second inner loop computes the following:

$$t = t + m * n$$

such that t is as previously defined and n is an integer of k words. m is a single word value computed just prior to its use on every iteration of the outer loop. In

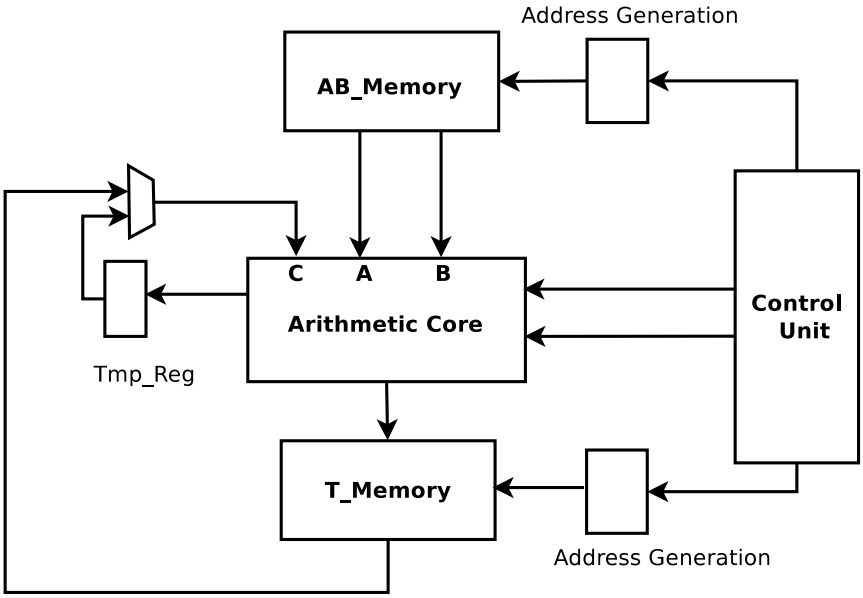


Figure 5.9: Top Level Architecture of the FFAU

short, the computation in each of the inner loops involves a multiplication of a large integer by a single word and the addition of another large integer.

At the center of the FFAU is the arithmetic core. It is capable of clocking in three w-bit operands and clocking out one w-bit result on every clock cycle. For the current design, the arithmetic core has two pipeline stages and uses parallel array multiplication and Carry Save Adder (CSA) row reduction techniques. While it achieves a throughput of one operation per clock cycle, each operation has a latency of three cycles. Table 5.4 reveals a subset of the operations the arithmetic core is capable of performing.

Note that *Result* is the lower w bits of the computation and *Carry* is the remaining upper w bits of the computation. The arithmetic core is self draining, meaning that the control bits from the control unit as well as the store address for the corresponding result propagate through the pipeline, along with the operand data. This greatly simplifies the required control logic.

Table 5.4: Arithmetic Core Computational Capabilities

Multiply-Add
$(Carry, Result) \Leftarrow A * B$
$(Carry, Result) \Leftarrow A * B + C$
$(Carry, Result) \Leftarrow A * B + C + Carry$
Add-Subtract
$(Carry, Result) \Leftarrow A + B$
$(Carry, Result) \Leftarrow A + B + C$
$(Carry, Result) \Leftarrow A + B + C + Carry$
$(Carry, Result) \Leftarrow -A + B$
$(Carry, Result) \Leftarrow -A + B + C$
$(Carry, Result) \Leftarrow -A + B + C + Carry$
Clear Pipe
$(Carry, Result) \Leftarrow C + Carry$
$(Carry, Result) \Leftarrow Carry$

The key to an efficient design is near 100% utilization of the arithmetic core. In order to avoid pipeline stalls, three w-bit operands must be fetched from internal scratchpad RAM, while one w-bit result is stored on every clock cycle. To allow for the use of dual port RAMs, the memory within the FFAU is split into two memory modules. The AB memory holds the a, b, and n integers, while the T memory holds the intermediate result, t. Since the AB memory must hold three k-word integers, the minimum size of the AB memory is $3k$ words. For design simplicity and future expansion, both the AB and T memories were designed to be $4k$ deep. It should be noted that this liberal use of memory will only slightly exaggerate the energy consumption of the FFAU, but for future work, the memory size will be re-examined. The AB memory requires two read ports, and at least one of those ports must support write operations in order to load the input data. The T memory module requires only one read port and one write port for the internal FFAU architecture.

Result data from the arithmetic core can be stored in either the T memory or a temporary result register. Part of the control data that propagates with the computation is the write-enable signal for the T memory module and the load signal for the temporary result register. The A input to the arithmetic core is multiplexed to allow input from either the AB memory module or the temporary result register. The temporary result register is necessary to avoid a structural hazard that would otherwise exist during the reduction step of the CIOS algorithm when computing $t = t + m * n$. Thus, m is stored in the temporary result register during reduction, thereby allowing the architecture to simultaneously access m and t. As with the A input, the B input of the arithmetic core is multiplexed, enabling multiplication by a value from an 8-entry, microcode-selectable RAM module within the control unit. For the calculation of m, a constant must be pre-loaded into the constant RAM.

The address generation logic is responsible for addressing the read ports for both memory modules. An index register is dedicated to each read port and can be independently controlled using the binary codes found in Table 5.5. The width of the index registers is determined by the depth of the RAM modules, $log_2(4k)$, and is automatically set prior to logic synthesis. The constant bus referenced in the table is fed by the constant RAM module within the control unit. The write port on the T memory module is addressable only by the store address pipeline within the arithmetic core. The store address along with the control signals latched into the arithmetic core on every clock cycle is supplied by the control unit.

The FFAU control unit, depicted in Figure 5.10, is a microcoded state machine. It has two additional index registers for handling nested loops, a small RAM for holding constants, a return address register for simple subroutines, and a command decoder for supporting multiple operations. The control unit is also capable of making branch decisions within a microprogram based on signals from the datapath. As

Table 5.5: Index Register Control Codes

Code	Operation	Description
00	Hold	no change to value
01	[reg]←const_bus	load register with value on constant bus
10	[reg]← 0	clear register value
11	[reg]← [reg]+1	increment register value

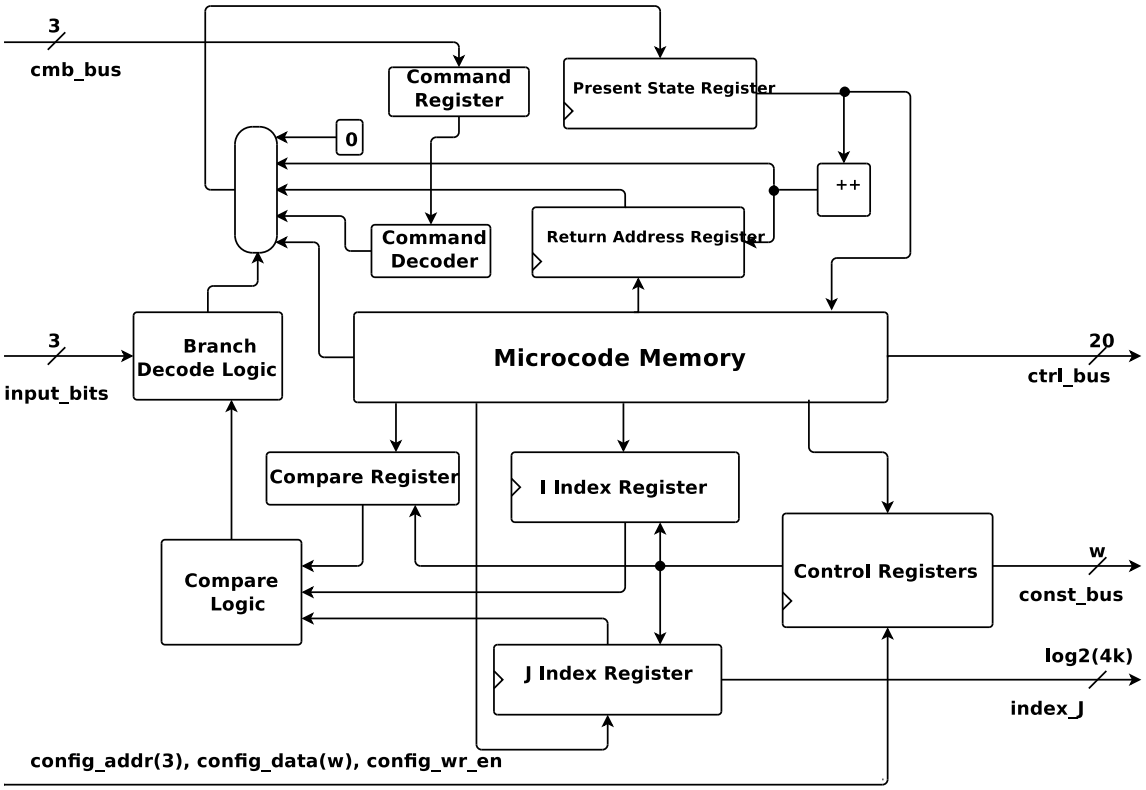


Figure 5.10: The Control Unit within the FFAU

seen in Figure 5.10, the CIOS algorithm requires a minimal amount of decision logic. Currently, the microcode ROM is 64 entries deep, which was more than enough to implement the CIOS algorithm, along with modular addition and subtraction.

5.4.2.2 Design Trade-offs

Even though this study was tailored for energy efficient finite-field arithmetic, many of the lessons learned here can be applied to the acceleration of other algorithms. During the design of the FFAU, a number trade-offs were encountered, such as the trade-off between reconfigurability and efficiency. As more logic is added to the design to support a wider variety of algorithms, the amount of logic being effectively utilized for a given algorithm decreases. If the accelerator is being tuned for a specific algorithm, one would want to provide just enough reconfigurability in the design to allow for a certain amount of scalability and not much more. After all, if reconfigurability is the primary design objective, the use of a general purpose processor should be considered.

To accommodate expansion and dynamic configuration of key size, *i.e.*, the size of the underlying finite field, the FFAU pulls array bounds from the constant RAM within the control unit. For this design, the use of microprogramming over hard-coding the control unit is preferred in order to improve reconfigurability and reduce control unit complexity. In this case, the control complexity is moved into the microprogram; however, a good microcode assembler can help improve the situation. It should be noted that combining a microcoded control unit with a constant RAM allows for two levels of reconfigurability and reduces the cost associated with the control store.

Scalability versus efficiency is another trade-off encountered in this study. Consequently, the FFAU is only scalable up to a certain point determined by the size of scratch memories. The approach for determining memory size taken here is to look at the largest practical problem size for which this device might be used. Unfortunately for cryptographic applications, the problem size grows as new attack algorithms are

developed. A complexity versus performance trade-off exists when considering the use of the temporary result register. At the cost of additional multiplexing logic and a control signal, a structural hazard is avoided, thereby reducing potential pipeline stalls. It should be noted that another solution to the aforementioned structural hazard is to add a third port to the AB memory; however, this could negatively effect the scalability of the design.

The aforementioned trade-offs were discussed somewhat independent of the algorithm complexity, but when considering area versus performance, it is beneficial to examine the computation time in terms of input size. For example, the number of clock cycles required to complete a CIOS operation on the FFAU is as follows:

$$cc = 2k^2 + 6k + (k+1)p + 22 \tag{5.2}$$

where k is the word length of the field, and p is the latency of an arithmetic core operation and is directly related to the depth of the pipeline. Optimizations that reduce the coefficients of higher ordered terms in the complexity equation should be prioritized for both energy efficiency and performance. Hence to reduce the coefficient of the quadratic term, the FFAU has logic for scratchpad address generation that is separate from that of the actual computation. Likewise, the memories are organized such that three operands can be read at once, rather than stalling while waiting for a third operand to be read from memory.

Another interesting trade-off that is quantified in (5.2) is clock rate versus pipeline depth. In this preliminary study, this trade-off was not really considered; instead, a depth which provided a reasonable clock rate with minimal logic optimization was chosen. However, assuming an ideal increase in clock rate due to pipelining, it is fairly straightforward to calculate an optimal pipeline depth for the FFAU using

(5.2). Obviously, the effect that p has on performance will be algorithm specific. In this situation, there is a data dependency within the outer loop of the CIOS algorithm that requires a pipeline stall, hence the coefficient, k. The +1 comes from the fact that the pipeline must drain before the final result is available. It should be noted that the data dependency could be removed at the cost of microcode complexity.

5.5 Binary-field Accelerator

As previously discussed, binary fields, $GF(2^m)$, are advantageous in the fact that addition does not require carry propagation. Thus, custom hardware implementations can perform addition over the entire length of a field element without a significant impact on clock rate. This lends itself to computationally efficient digit-serial multiplication with field-specific reduction [41]. Furthermore, binary-field squaring can be performed simply with a handful of XOR gates when the binary field is fixed [16]. Therefore, we designed and evaluated a non-configurable, $GF(2^m)$ accelerator for further energy efficiency. Figure 5.11 shows the top-level diagram of "Billie," the binary accelerator, with Pete.

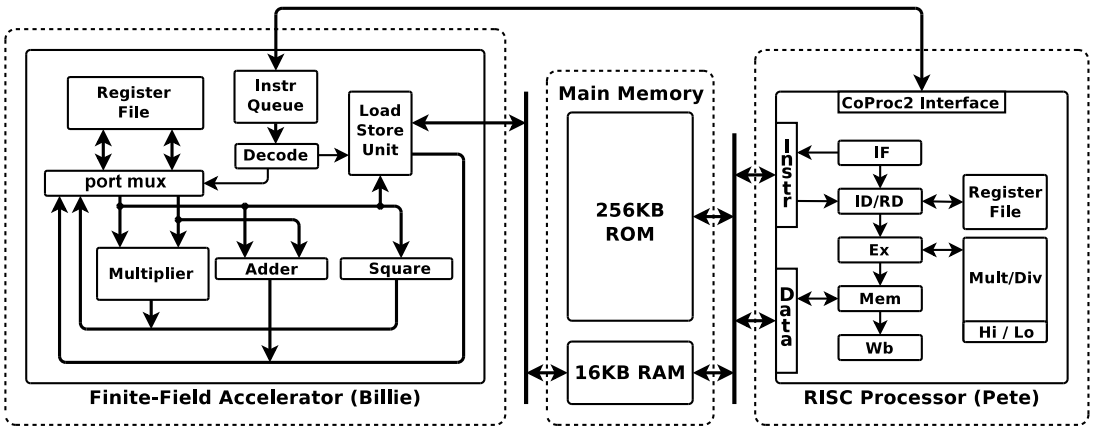


Figure 5.11: The binary-field accelerated architecture, "Pete with Billie"

Table 5.6: Coprocessor 2 Instructions used to control Billie

Format	Description	Operation
COP2SYNC	CoProcessor 2 Sync	Sync Operation
COP2LD rt, fs	Load into Billie Register	BR[fs] $\leftarrow$ MEM[GPR[rt]]
COP2ST rt, fs	Store into Memory	MEM[GPR[rt]] $\leftarrow$ BR[fs]
COP2MUL fd, fs, ft	Modular Multiply	BR[fd] $\leftarrow$ BR[fs] $\times$ BR[ft]
COP2SQR fd, ft	Modular Square	BR[fd] $\leftarrow$ BR[ft]^2
COP2ADD fd, fs, ft	Modular Add	BR[fd] $\leftarrow$ BR[fs] + BR[ft]

5.5.1 Coprocessor Instructions

Prior research has suggested that the control of the binary-field accelerator can significantly limit performance [42]. Thus, Billie utilizes the MIPS coprocessor interface for command and control to reduce this potential bottleneck. In such cases, Pete fetches binary-field instructions and feeds them directly to Billie at a high rate. Similar to the configuration with Monte, Pete and Billie share the dual-port RAM to eliminate inefficiencies caused by processor-to-accelerator data transfers. Table 5.6 lists the instructions added to Pete's ISA in support of the binary-field coprocessor. The **cop2sync** instruction was previously discussed in Section 5.4 but listed here for completeness.

Billie is a load-store architecture, so **cop2ld** and **cop2st** are used to move data to and from her 16 entry register file. Specifically, **cop2ld** loads a multi-precision field element from memory starting at the address referenced by the *rt* GPR into the Billie Register (BR) specified by *fs*.[3] Conversely, **cop2st** stores a field element from the *fs* BR into memory starting at the address referenced by the *rt* GPR. Continuing the load-store concept, the binary-field arithmetic instructions pull input data from and write results back into Billie's register file. For multiplication and addition,

[3]The General Purpose Registers (GPRs) are part of Pete's register file.

cop2mul and **cop2add** follow the three-operand instruction format such that *fs* and *ft* are the input operands, and *fd* is the result operand. Because squaring is a unary operation, the **cop2sqr** instruction only requires a two-operand format such that *fs* is the input operand, and *fd* is the result operand.

5.5.2 Microarchitecture

For a preliminary evaluation, we based Billie on the NIST 163-bit binary field. A zoomed in view of the microarchitecture is illustrated in Figure 5.12. From a high level, our design for Billie takes a similar approach to the original IBM 360 floating point unit [43]. Notable features include an instruction queue, register file, load/store unit and separate functional units for multiplication, squaring and addition.

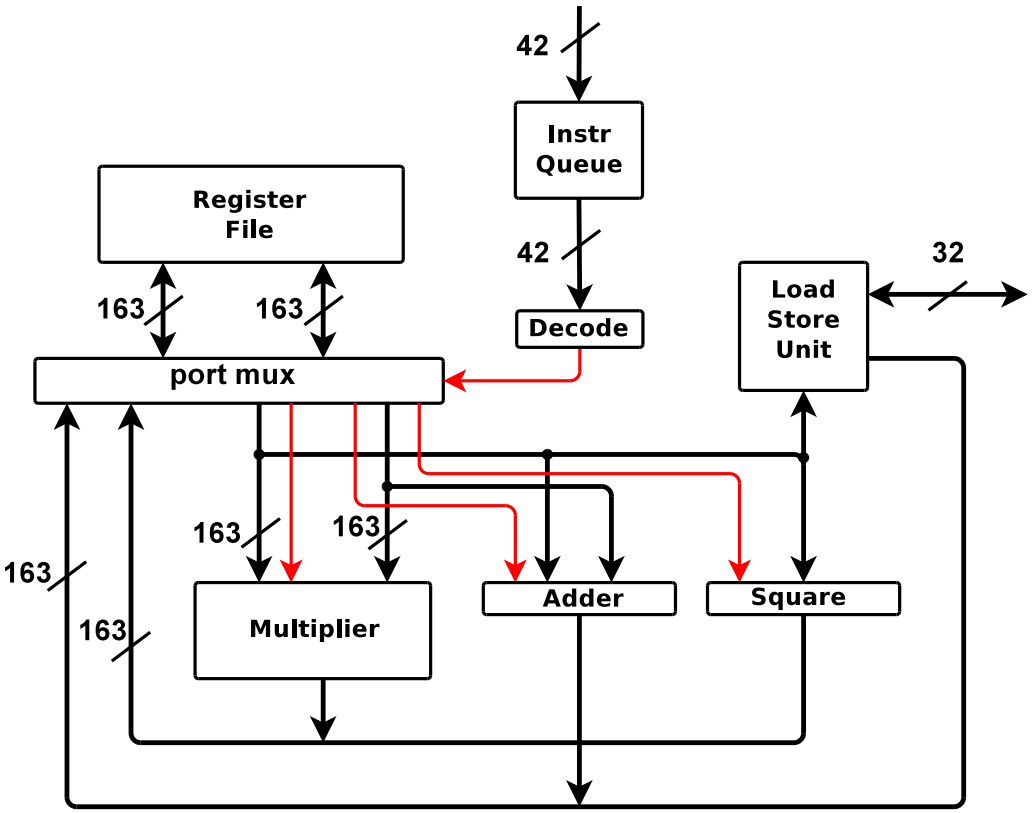


Figure 5.12: Billie's coprocessor architecture.

Coprocessor instructions fetched by Pete are first buffered in Billie's four-entry instruction queue. This avoids stalling Pete while the longer-latency binary-field instructions execute. When an instruction is at the head of the queue, the logic decodes it and checks for structural and data hazards. A structural hazard exists when the appropriate functional unit is currently busy, while a data hazard exists when the input operands have not yet been stored in the register file. If no hazards exist, the operands will be read from the register file, and the instruction will dispatch to the corresponding functional unit. On the next clock cycle, the instruction will begin executing and once complete, will remain in the functional unit until the result has been written back into the register file. In this architecture, reads from the register file are prioritized over writes. Thus, write-back of the result will occur when an instruction is not being dispatched.

To reduce structural hazards, the register file has two read/write ports (*i.e.*, true dual port). The data paths between the register file and the functional units are 163 bits wide. Multiple functional units require write access to the register file, so the port multiplexor must also perform arbitration. We chose a simple scheme in which each functional unit is statically assigned a port for writing into the register file. For instance, the multiplier and squaring unit both share a port, while the adder and load/store unit share another. If both functional units assigned to the same port are ready to write during a given clock cycle, the arbiter will allow one to write and stall the operation of the other. For simplicity, the priorities of each functional unit are fixed. In our design, the multiplier and adder have higher priority over the squaring and load/store units, respectively.

The register file contains sixteen, 163-bit registers to accommodate all intermediate computations for a scalar-point multiplication. Because we use sliding-window algorithms that leverage some precomputation, we require twice the number of reg-

isters as compared to Guo *et al.* [42]. However, as will be shown in Section 7.6, the extra registers yield a significant performance advantage and have the potential to save energy. The load/store unit is responsible for transferring binary-field elements between Billie's register file and shared memory. The interface to shared memory is 32-bits wide, while the interface to the register file is a field width (*e.g.*, 163-bits for this particular configuration). Thus, the load/store unit serves as a buffer between these two mismatched ports and is analogous to Monte's DMA unit.

5.5.3 $GF(2^m)$ Arithmetic Units

For $GF(p)$ computation, the propagation of arithmetic carries from the least significant bit position to the most within multiplication and addition typically becomes the clock-rate limiting critical path. From an implementation stand point, full field-width $GF(p)$ computation is impractical. Thus, field elements are broken up into smaller words, and computation proceeds at that granularity (*i.e.*, multi-precision). For $GF(2^m)$ computation, carry propagation does not exist, so full field-width addition is possible and advantageous. Compared to multi-precision computation, full field-width $GF(2^m)$ arithmetic requires less complex logic and scales more easily to increasing field widths. The hardware scalability is a consequence of data-level parallelism afforded by the carry-less computation.

The addition and multiplication units within Billie take advantage of this parallelism by performing addition over an entire m-bit binary polynomial in a single clock cycle. Because addition is fast, we employ digit-serial multiplication that iterates over the multiplier, shifting and adding the multiplicand into an accumulator, accordingly [35]. Specifically, Algorithm 8 describes the multiplication operation in detail, where $a(x)$ is the multiplicand, $b(x)$ is the multiplier, $c(x)$ is the accumulator, and D is the digit width. As shown, Step 1 zeros out the accumulator. Initially,

Step 3 multiplies the least significant digit of the multiplier (B_0) by the multiplicand and adds the result to the accumulator. Concurrently, Step 4 shifts the multiplicand D bits to the left and reduces the result *modulo* $f(x)$, the irreducible polynomial. Note that this algorithm integrates the polynomial reduction into the multiplication. Steps 3 and 4 repeat with the next significant digit of the multiplier until the multiplication is complete. The final step reduces the $m + D - 1$ result to m-bits with $f(x)$.

Algorithm 8 Digit-serial $GF(2^m)$ multiplication [41]

Input: $a(x) = \sum_{i=0}^{m-1} a_i x^i$, $b(x) = \sum_{i=0}^{\lceil \frac{m}{D} \rceil - 1} B_i x^{D \cdot i}$
Output: $c(x) = a(x) \cdot b(x)\ mod\ f(x) = \sum_{i=0}^{m-1} c_i x^i$
1: $c(x) \leftarrow 0$
2: **for** i from 0 to $\lceil \frac{m}{D} \rceil - 1$ **do**
3: $\quad c(x) \leftarrow B_i \cdot a(x) + c(x)$
4: $\quad a(x) \leftarrow a(x) \cdot x^D\ mod\ f(x)$
5: **end for**
6: **return** $c(x)\ mod\ f(x)$

For squaring, we employ field-specific hardware that takes advantage of fast binary squaring discussed in Section 2.1.4 and incorporates the reduction operation. As an example, Figure 5.13 depicts a hardwired squaring unit for $GF(2^7)$ assuming $f(x) = x^7 + x + 1$.

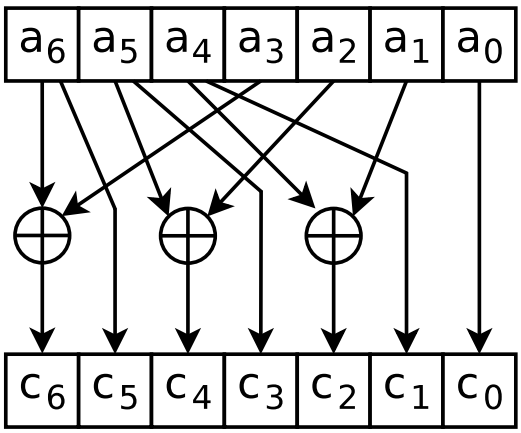


Figure 5.13: Binary-field squaring unit, s.t. $f(x) = x^7 + x + 1$ [16]

6. METHODOLOGY

We developed fully synthesizable Verilog models for the baseline processor ("Pete"), ISA-extended processor, and the Finite-Field Accelerator ("Monte"). Front-end synthesis was used to construct the arithmetic components, including the 17-bit by 17-bit multiplication blocks within Pete, and the 32-bit multiply-add unit within Monte. Post-synthesis power estimations for core logic on a 45 nm technology node with a 333MHz clock were performed using Synopsys Prime-Time, a timing and power analysis tool for CMOS logic [22, 44].

To estimate memory power, we used counters to keep track of the number reads and writes to and from the memories embedded within the testbench, and we used Cacti to extract estimates of energy per read/write and leakage power [38]. Unfortunately, no equivalent tool for estimating ROM power exists. As a conservative estimate, ROM dynamic power was assumed to be equivalent to a comparably sized RAM, while ROM static power was assumed to be zero.

Each cryptographic operation requires millions of clock cycles and is arduous to simulate post-synthesis. For power estimations using the techniques discussed above, we simulated a portion of the algorithm representative of the entire algorithm. To measure execution times, we emulated each microarchitecture using Verilator, a fast, two-state Verilog HDL simulator [45].

7. EVALUATION

7.1 Prime Fields

Figure 7.1 summarizes our results for prime fields by graphing the energy per operation (a signature followed by a verification) for each of the evaluated microarchitectures vs. key size. These results confirm each configuration's placement in the diagram in Figure 1.1 and show that special purpose hardware becomes much more attractive as greater security requirements are demanded. For $GF(p)$ ISA extensions, we observe between 1.32 to 1.45 factor improvement in energy efficiency over baseline, while for $GF(p)$ acceleration with Monte, we observe a 5.17 to 6.34 factor improvement. A point in between is the microarchitecture with ISA extensions and a 4KB instruction cache. For such a system, we see a 1.67 to 2.08 factor improvement in energy compared to baseline.

Furthermore, we observe that the energy consumed increases quite rapidly as the key size is increased. Examination of the data reveals the increase is substantially greater than quadratic for the pure software configuration, while for the microarchitecture with ISA extensions, the increase is closer to quadratic. The effect of key size is much more gradual for the energy consumed by the fully accelerated microarchitecture, coming in just slightly less than quadratic for 192-bit to 256-bit key sizes. After 256-bit, key size appears to have a greater effect on the accelerated architecture. This may be due to our choice of algorithm for inversion within Monte (Fermat's Little Theorem), which has a $O(n^3)$ computational complexity. We plan to further investigate this issue.

Figure 7.2 displays a side-by-side comparison of the energy, broken down into subcomponents, consumed per 192-bit and 256-bit operation vs. microarchitecture. The

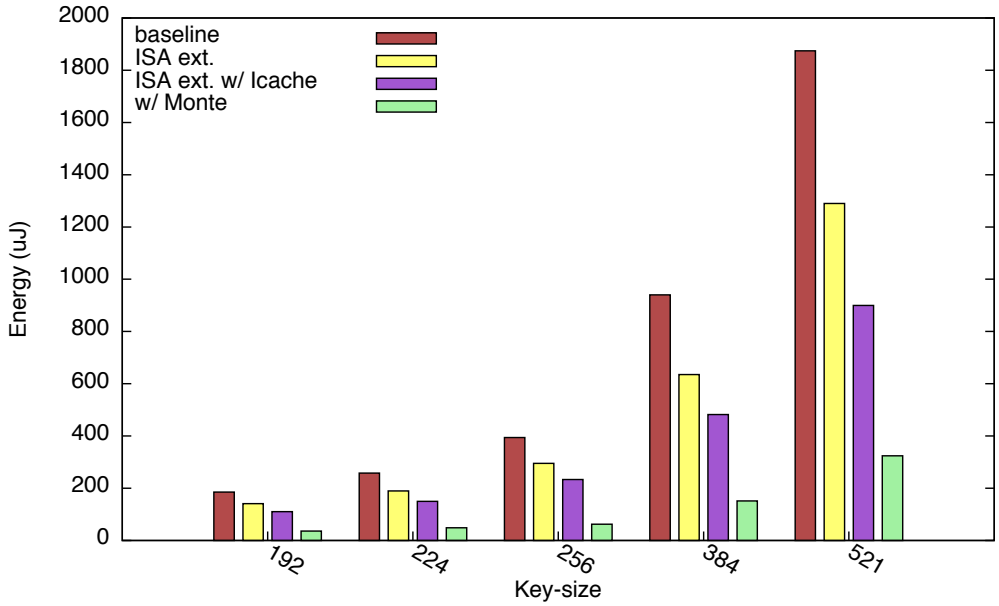


Figure 7.1: Energy per Sign + Verify vs. key size and microarchitecture for prime fields.

results show that a significant portion of the energy consumed by the baseline and ISA extended microarchitectures is spent in the ROM. In these microarchitectures, an instruction must be read nearly every cycle; therefore, the ROM is kept very active. We found that this is a common theme amongst low-power embedded processors [46]. We also note that the ROM energy is much less when Monte is in use. In such case, Monte's microcode ROM is producing most of the instructions so the ROM activity factor is dramatically lowered. For the hardware configuration with a 4KB instruction cache, we see that a significant portion of the ROM energy is traded for additional energy in the uncore portion of the system. Here we use the term *uncore* to refer to the instruction cache, program ROM controller, and the instruction and data buffers (discussed in Section 5.3) as well as miscellaneous multiplexing logic for the program ROM and the RAM. It should be noted that the instruction cache will improve the energy of the entire software system and not just the cryptographic

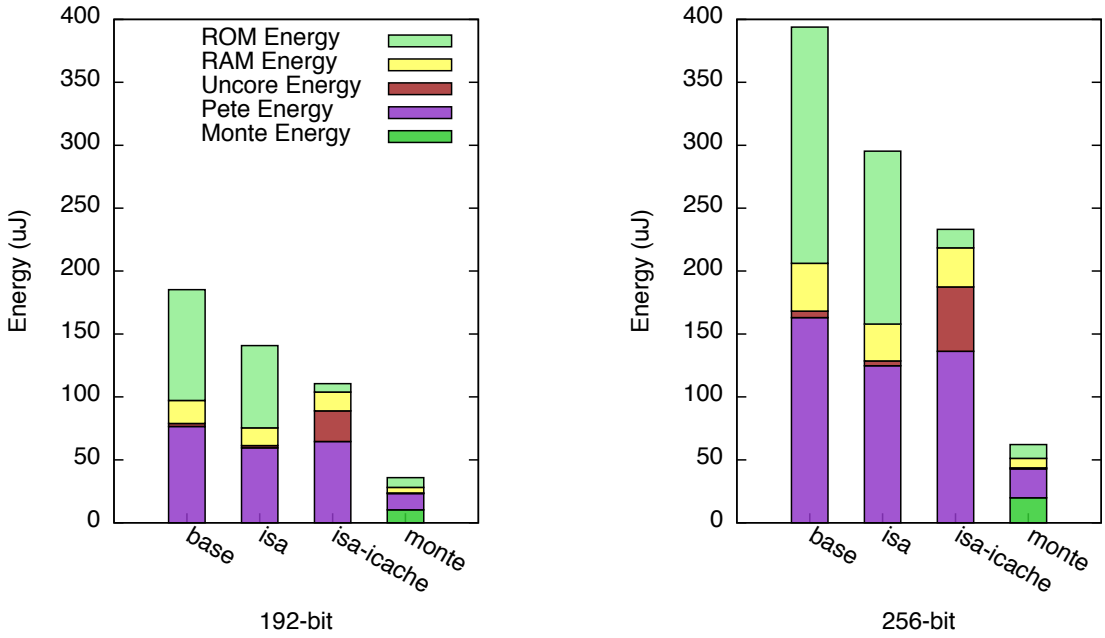


Figure 7.2: Breakdown of energy per Sign + Verify for 192 and 256-bit key sizes into various sub-components.

algorithms we evaluate here.

Another interesting observation that can be made in Figure 7.2 is that the energy consumed in the RAM decreases as the level of acceleration increases. This is partially due to reduced execution time decreasing the amount of energy spent on leakage power but primarily due to the fact that each acceleration technique aims to reduce access to memory. For example, product scanning used with the proposed instruction set extensions requires fewer stores than operand scanning. Similarly, Monte utilizes smaller, internal buffers and data forwarding to reduce accesses to shared memory. The instruction cache, however, does not significantly affect the RAM energy as seen by comparing the ISA extension microarchitectures with and without the instruction cache.

For further examination, Figure 7.3 shows the energy consumption broken down

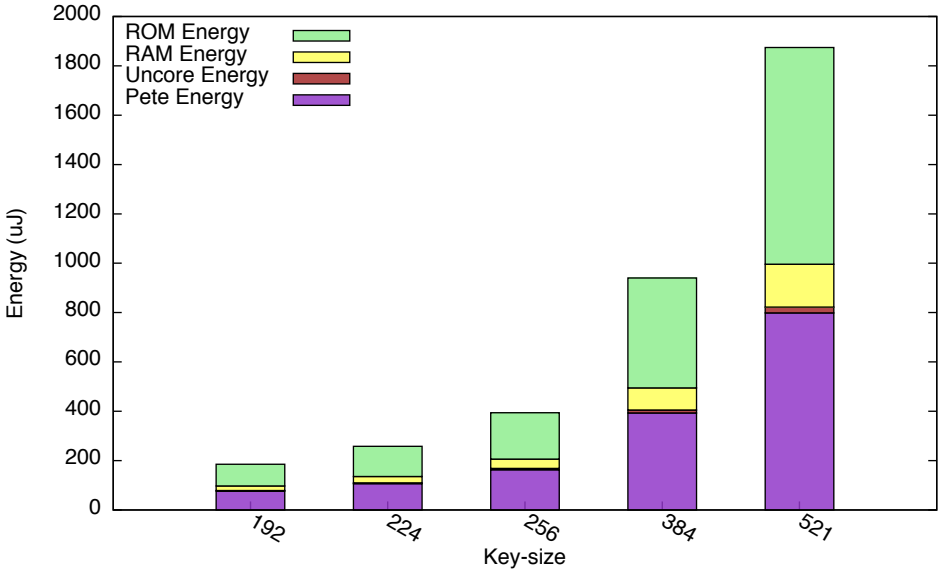
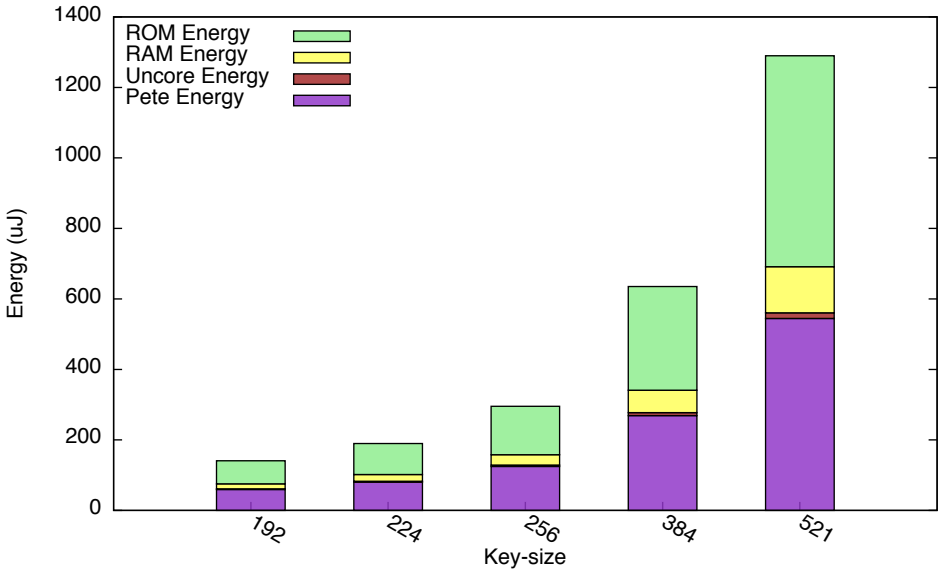


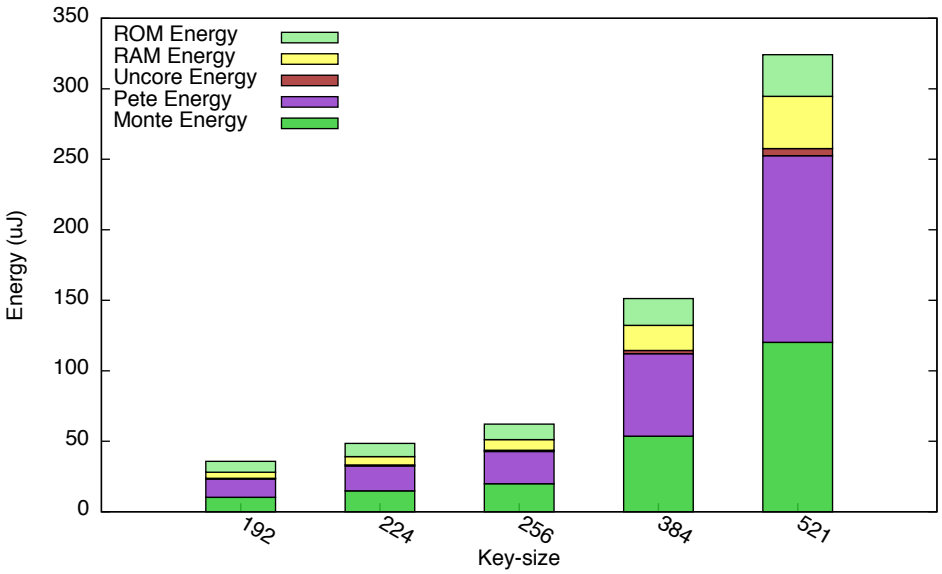
Figure 7.3: Energy per Sign + Verify vs. key size for our baseline with no hardware acceleration.

into sub-components for the baseline across the five prime fields recommended by NIST. Figure 7.4 shows the same for the ISA extended microarchitecture and the microarchitecture with Monte. For the baseline and ISA extended architectures, the processor core (i.e., Pete) energy changes mostly with execution time, as the power remains fairly constant while varying the key size. For instance, from 192-bit to 521-bit, Pete's power goes up by only 6% in the baseline configuration. The change in core power for the ISA extended microarchitecture was even less.

For the Monte configuration, Pete's power drops considerably (≈23%) compared to baseline and continues to decrease as key size is increased. This decrease is because Pete spends a significant amount of time stalled, while Monte performs the majority of the computation. Even while stalled, however, Pete is still the dominant energy consumer as shown in Figure 7.4b. After close examination of the power breakdown provided by Synopsys, it became clear that the dominant contributors to Pete's power is the clock network and registers, which still have a high activity factor while stalled.



(a) ISA extended



(b) W/ Monte

Figure 7.4: Energy per Sign + Verify vs. key size for the ISA extended microarchitecture and the architecture accelerated with Monte.

7.2 Binary Fields

While prime fields have a longer history of use, binary fields have been shown to offer a performance benefit and have a potential energy benefit as well [30]. Thus, we also evaluate the use of binary fields for ECDSA in our energy study. We start out with a software-only implementation on our existing microarchitecture, using Algorithm 6 with $w = 4$ and the fast reduction routines discussed in Section 4.2. Then we augment our microarchitecture with binary support by including carry-less multiply and multiply-accumulate instructions. Figure 7.5 compares our estimated energy consumption for the microarchitectures with and without binary support for the 5 NIST recommended binary fields.

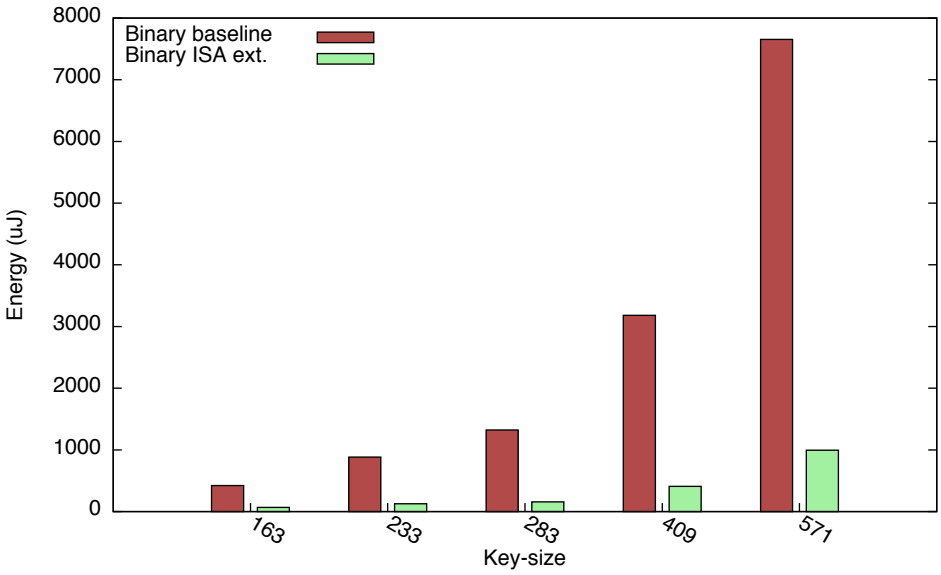


Figure 7.5: Energy per Sign + Verify vs. key size for binary fields.

As shown in Figure 7.5, the software without binary support is less energy efficient than the ISA extended version by a factor of 6.40 to 8.46. When a carry-less multiplier

is not part of the microarchitecture, the computational complexity for binary-based ECC is much higher because the multiplication must be emulated with shifts and adds. As a result, we do not recommend the use of binary fields without hardware support. For further evaluation, Figure 7.6 depicts the energy breakdown of a sign and verify operation for binary field ISA extensions across the NIST fields.

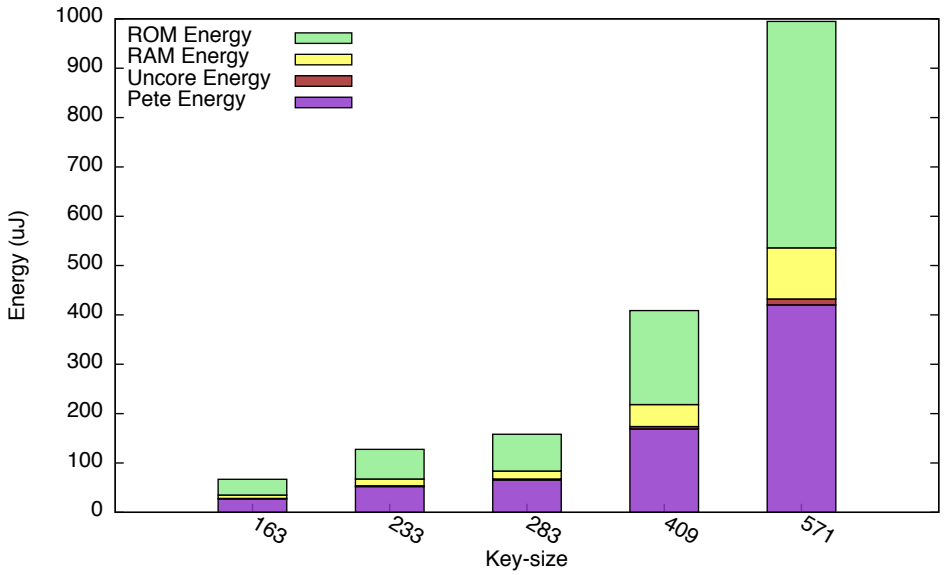


Figure 7.6: Energy per Sign + Verify vs. key size for binary ISA extensions.

7.3 Prime Fields vs. Binary Fields

In the ISA extended microarchitecture, binary fields have the advantage of carry-less add and a less computationally complex squaring algorithm (i.e., $O(n)$ as opposed to $O(n^2)$). Additionally, the same algorithm used for prime field multiplication (Algorithm 3) can be used for binary field multiplication. The result is a 1.30 to 2.11 factor improvement over prime ISA extensions comparing fields of equivalent security.

Figure 7.7 graphs the energy per operation (signature + verification) for prime and binary fields of equivalent security. One interesting thing to note is the reduced computational complexity with binary fields compared to prime fields. At the smallest key size, the binary field is smaller than the prime field, and the binary ISA configuration consumes 52.2% less energy. At 256/283-bit, the field sizes cross over such that the binary field is larger. However, in this case, binary is still 46.5% less energy. At the largest key size, the binary field is considerably larger than its prime counterpart, and consequently, this configuration yields the lowest improvement (22.8% less energy).

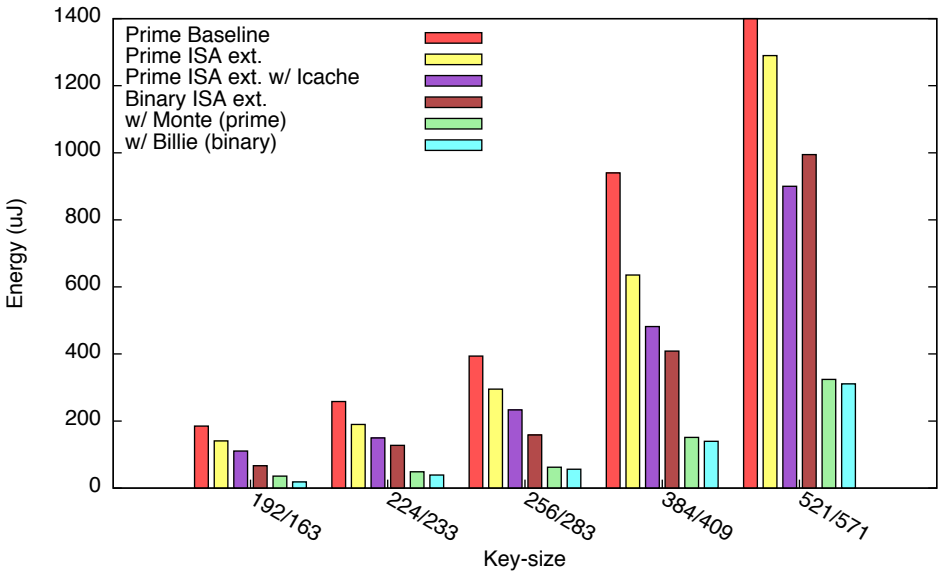


Figure 7.7: Energy per Sign + Verify vs. key size, comparing prime and binary fields of equivalent security.

Figure 7.7 also compares our ECDSA energy estimates of Billie with the other systems. For full $GF(2^m)$ acceleration with Billie, we observe a 1.92 factor improvement over Monte for 163-bit. However, as we move out to larger field sizes, the energy

cost for Billie converges with that of Monte. Thus, our binary-field accelerator is not scaling well past 163-bit.

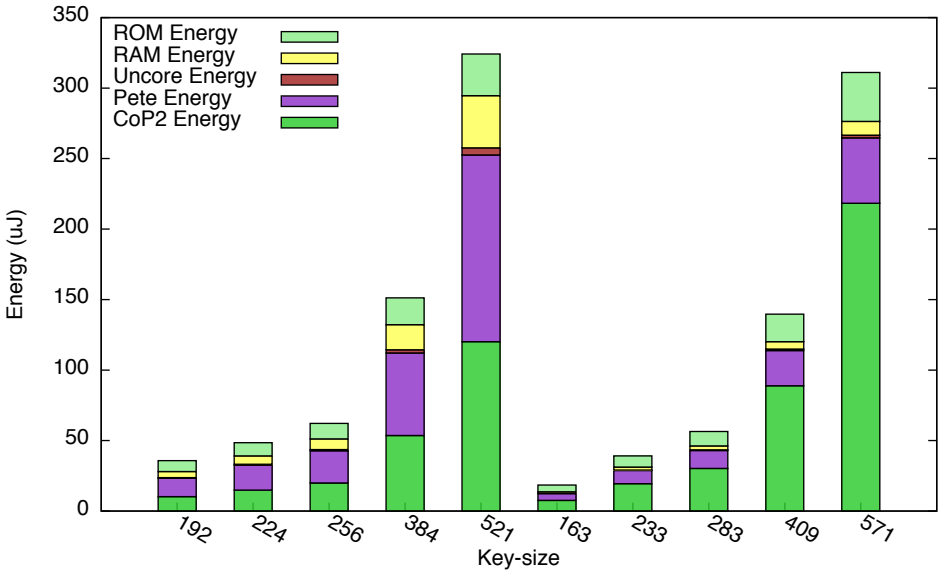


Figure 7.8: Energy per Sign + Verify vs. key size for Monte (left) and Billie (right)

For a side-by-side comparison, Figure 7.8 shows the energy consumption broken into subcomponents for both Monte (left) and Billie (right). It is interesting to note that when Monte is used, Pete is still consuming most of the energy, in spite of being idle for the majority of the computation. There are two reasons for this: First, we are not using clock or power gating techniques because Pete is still fetching instructions for Monte, so Pete's clock network is still active. Second, Monte has less logic overhead than Pete, and the size is fixed, regardless of the field being used. Billie on the other hand, is the primary consumer of energy when used. Unfortunately, we were unable to effectively model Billie's register file with Cacti due to its non-standard access width. Thus, we had to synthesize the register file with flip-flops,

which makes for an inefficient implementation. Furthermore, Billie is designed to scale in hardware with the field size and is consequently much larger than Pete. For example, the 163-bit implementation requires 45% more area than Pete, while the 571-bit implementation requires five times the area of Pete. However, when ECDSA is accelerated with Billie, on average only 38% of the execution time is spent on scalar point multiplication. The rest of the time, Billie is idle, wasting energy, while Pete performs the additional protocol arithmetic modulo the group order. In future work, we would like to investigate the use of clock and power gating techniques to eliminate this inefficiency.

Figure 7.9 depicts the energy consumption broken into subcomponents of all of the hardware accelerated architectures for both 192/163- and 256/283-bit field sizes. As illustrated, Billie reduces the RAM energy even further by keeping the entire scalar point multiplication within her register file. The majority of the RAM accesses when Billie is used are from Pete while performing arithmetic modulo the group order.

7.4 Power Consumption

Figure 7.10 provides a plot of the overall system power, broken down into static and dynamic components, for each of the evaluated microarchitectures. Here we averaged the baseline runs across all binary and prime fields because we saw little variation in power due to different software configurations. We did the same for the ISA extended architectures. In fact, we see almost no difference in overall system power consumption between baseline and the ISA extended architecture ($< 1\%$).

The hardware configuration with the Instruction Cache (IC) on average consumes 14.5% less power than the configuration without the cache. Despite the extra hardware cost of the cache and the subsequent static power increase, the reduction in ROM instruction reads leads to a significant power reduction overall. The configu-

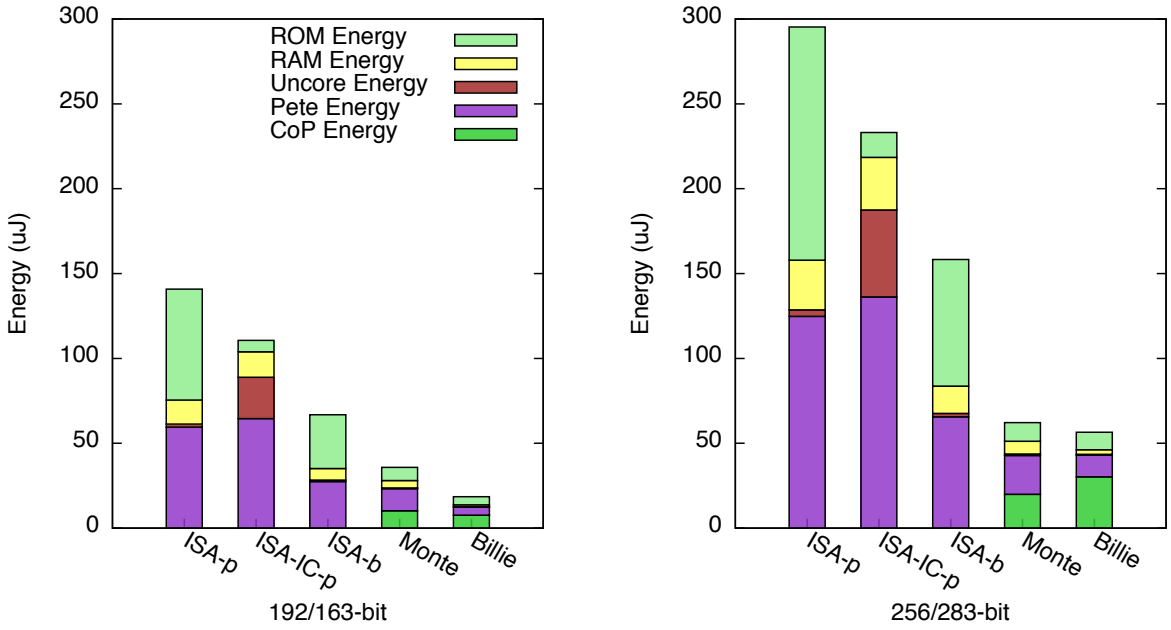


Figure 7.9: Breakdown of energy per Sign + Verify for 192/163- and 256/283-bit key sizes into various sub-components.

ration with Monte reduces the power draw even further (18.6% less power compared to baseline) by reducing the activity on Pete and the ROM. The systems with Billie, however, consume the most power overall. As previously mentioned, Billie is significantly larger than Pete, and the amount of resources Billie consumes grows approximately linear with field size. Also, we synthesized Billie's large register file, using flip-flops instead of RAM, which contributes to the approximately linear increase power.

Although the static power in Figure 7.10 appears to be a minor portion of the overall power (8.5%), much of the dynamic power includes the clock network. Thus, our system could still benefit substantially from power and clock gating techniques. This is especially true for the Billie accelerated systems in which Billie is idle but still consuming power for 62% of the ECDSA operation.

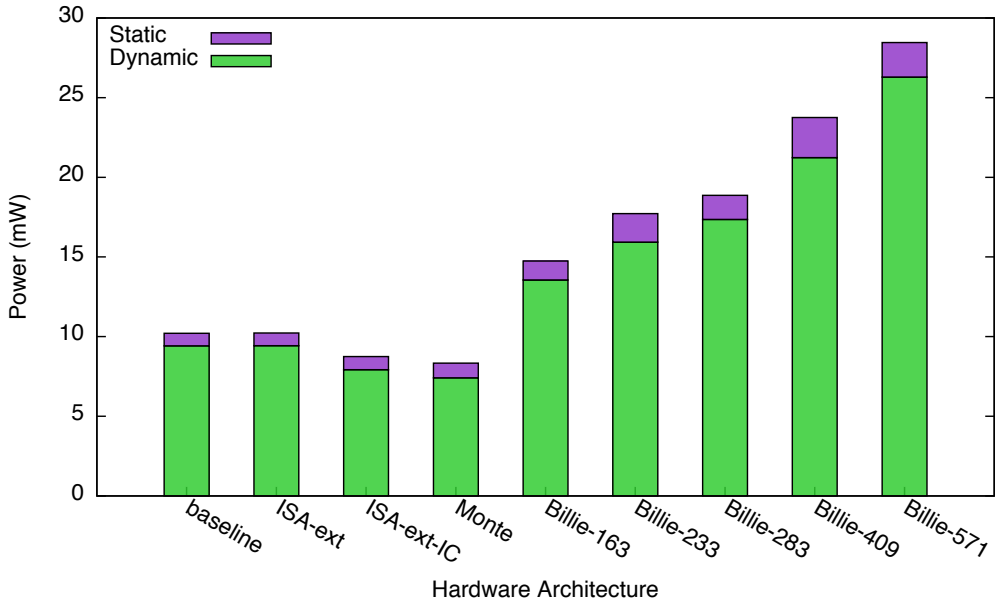


Figure 7.10: Static and dynamic power of evaluated microarchitectures.

7.5 Evaluation with Instruction Cache

Due to instruction fetch's significant contribution to the overall energy per operation, we felt further investigation was warranted. Our initial evaluation was to model our three systems with an ideal 4KB direct-mapped instruction cache with a 16-byte line size.[1] Although this cache model is unrealistic, it reveals the best case energy benefit for each system. Figure 7.11 shows the energy improvement of each system with an instruction cache across three key sizes.

Observe that the energy benefit of the instruction cache is much higher for the microarchitectures without Monte. This is because instruction fetch contributes much less to the overall energy consumption when using an accelerator. In such a case, the processor is mostly idle, while the accelerator performs the majority of the computation. Although Monte fetches internal microinstructions to carry out the

[1] In this model, we used Cacti to estimate the energy of the entire cache.

multi-precision computation, the microcode table is only 64 entries and thus has an insignificant effect on energy. Further proof of this assertion is illustrated by the fact that the already small energy benefit decreases as the key size increases, i.e., as more of the computation is shifted to the accelerator. A final observation is that the benefit of the instruction cache does not appear to be dependent on the key size for the microarchitectures without Monte; however, this is simply a result of not modeling cache misses.

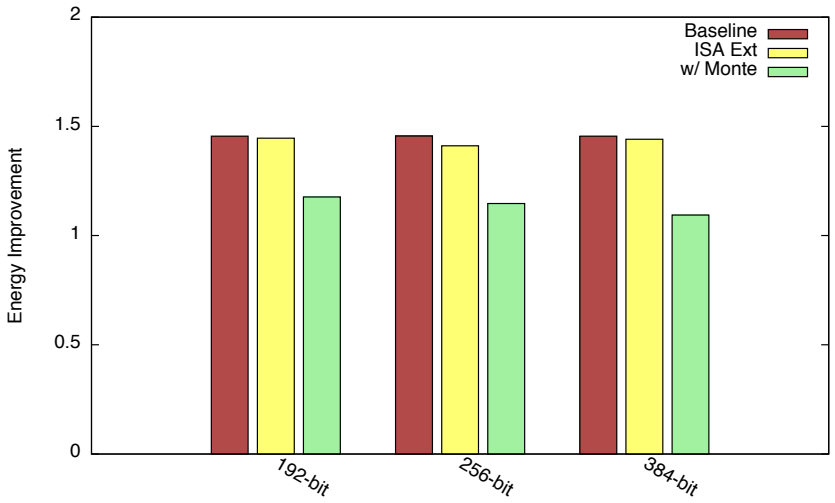


Figure 7.11: Energy improvement with ideal instruction cache vs. key size.

The next step was to design an instruction cache in Verilog and run a full-system energy estimation, using Cacti only to estimate the energy of the memories. In this case, we fixed our workload to a P192 Sign/Verify operation and varied the instruction cache size from 1KB to 8KB. Additionally, we simulated our cache with and without a prefetcher. Figure 7.12 shows our results broken down into subcomponents. One surprising result is that there is not a lot of variation between the cache configurations. Due to a higher number of cache misses, the configurations with the

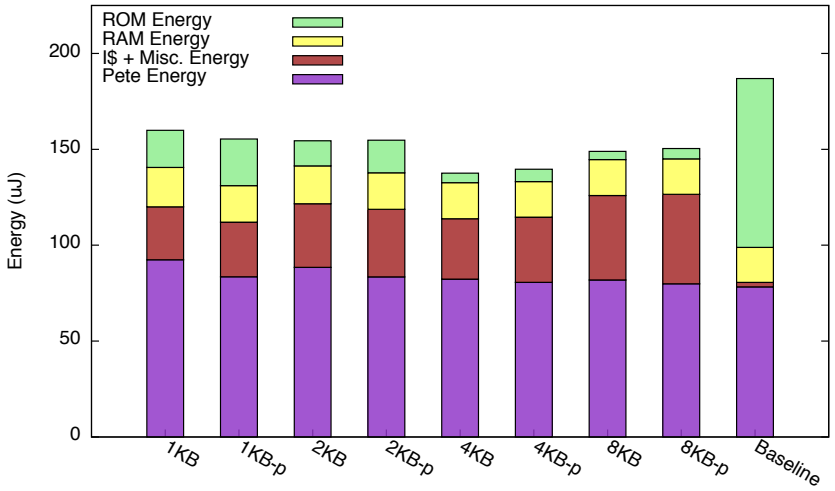


Figure 7.12: Energy per 192-bit Sign + Verify with real instruction cache for various cache configurations. The "-p" indicates the presence of a prefetcher.

smaller caches lose the most amount of energy to the program ROM. However, the energy consumption of the instruction cache goes up as the cache size is increased, counteracting the benefit of reduced ROM reads.

For Pete, the general trend is that the energy consumption goes down as cache size is increased or prefetching is utilized. This is due to the fact that the power of the processor core does not vary significantly across different cache configurations. Thus, the configuration with the smallest run time will yield the highest core efficiency. Although Pete's power does not vary significantly, it is interesting to note that the power always goes up as the run time decreased. For instance, from a 1KB cache to a 4KB cache, Pete's power increases by 3.23%. This is cause by higher average activity factors in the logic as Pete spends less time waiting for a miss to be serviced.

The energy consumption of the RAM remains relatively constant throughout the different configurations, only varying by 11.1%. Because the RAM is fairly small (16KB), the static power has less of an effect on the overall energy consumption.

Even for the 1KB, non-prefetcher configuration, the static power contribution to the RAM energy is only 41.7%. The rest of the RAM energy is based on the number of reads and writes, which is not affected by the instruction cache configuration. Although like the processor logic, the general trend of the RAM energy is to decrease as the run time decreases.

The energy improvement due to prefecting decreases as the cache size increases, and past 4KB, the prefetcher actually has a negative impact. This phenomenon can be explained by the decreasing number of cache misses as the cache size is increased. In the case of a 1KB cache, the prefetcher improves performance by 11.5%, but with an 8KB cache, it only improves the performance by a mere 2.0%. With larger caches, there is less opportunity for prefetching to improve performance. As seen in Figure 7.12, the negative impact of prefetching comes from the increased number of reads from program ROM as well as the increased power consumption in the instruction cache. It might be possible to further improve the prefetching algorithm for use with smaller cache configurations. Also, the benefits of prefetching would be much greater in a system with high-latency access to main memory [39]. In our system, however, the miss penalty is only three clock cycles.

It is interesting to note the drop in energy consumption from 2KB to 4KB is much greater than from 1KB to 2KB. This is primarily due to the fact that the number of cache misses drops more significantly from 2KB to 4KB. For example, from 1KB to 2KB, the number of cache misses decreases by only 33.7% as opposed to 65.2% from 2KB to 4KB. Similarly, from 4KB to 8KB, the decrease is a mere 18.3%. This tells us that our working set is somewhere around 4KB. Putting the aforementioned conclusions together, we see that the 4KB instruction cache without a prefetcher has the lowest energy consumption. Compared to our baseline microarchitecture, this equates to a 35.8% improvement in energy.

For further evaluation, we modeled the 4KB instruction cache configuration across the five NIST prime fields with ISA extensions enabled (Figure 7.13). For prime-field support, this hardware configuration represents our most energy efficient configuration without the assistance of a separate coprocessor. As we would expect, all of the components except the ROM access scales.

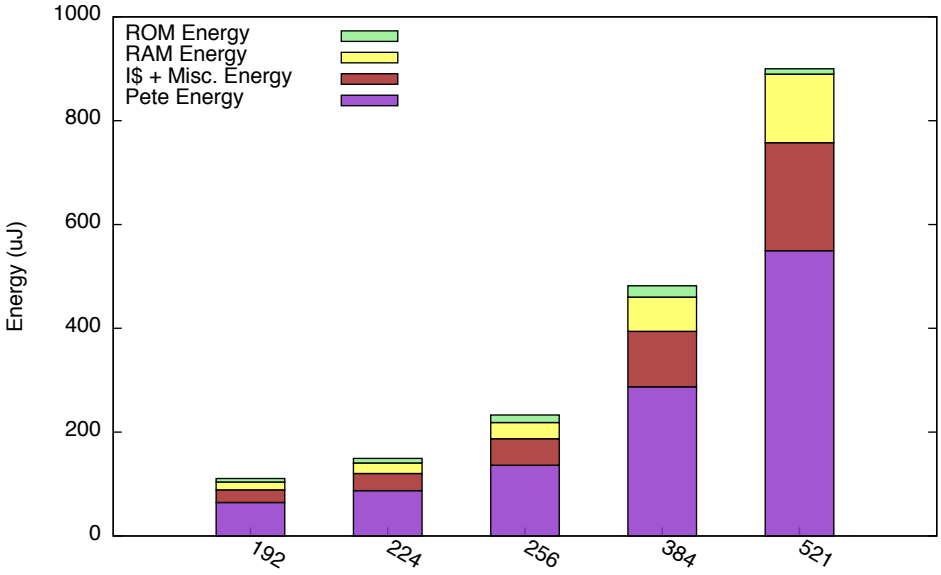


Figure 7.13: Energy per Sign + Verify vs. key size for prime ISA extended microarchitecture with 4KB instruction cache.

7.6 Performance Evaluation

For reference, we have included Table 7.1, which shows the latency (clock cycles) per cryptographic operation for various prime-field configurations. The combined Signature and Verification latency closely models an SSL handshake on the client side.

Table 7.2 provides the latencies per crytographic operation (clock cycles) for the

Table 7.1: Latency per operation (100K clock cycles) for prime-field microarchitectures

μarchitecture	Key Size	Sign	Verify	Signature + Verification
Baseline	192-bit	26.9	34.27	61.2
Baseline	224-bit	37.2	47.9	85.1
Baseline	256-bit	57.2	72.8	130.0
Baseline	384-bit	133.6	174.9	308.5
Baseline	521-bit	297.2	304.8	602.0
ISA Ext	192-bit	20.5	25.6	46.1
ISA Ext	224-bit	27.5	34.6	62.1
ISA Ext	256-bit	42.7	53.7	96.4
ISA Ext	384-bit	90.9	114.6	205.5
ISA Ext	521-bit	184.0	230.5	414.5
W/ Monte	192-bit	6.0	7.5	13.4
W/ Monte	224-bit	8.3	10.3	18.6
W/ Monte	256-bit	10.9	13.4	24.2
W/ Monte	384-bit	28.2	34.9	63.0
W/ Monte	521-bit	64.5	78.2	142.7

binary-field systems.

To demonstrate the computational efficiency of Billie, Figure 7.14 shows the execution time of a scalar point multiplication versus the digit size of the multiplier. In all other results but Figure 7.14 we use a 3-bit digit size for the $GF(2^m)$ multiplication unit. The 3-bit digit size was chosen because it was shown to be energy-optimal in prior work [41]. For comparison, we graph prior work by Guo *et al.* that attempts to eliminate control bottlenecks by integrating an 8-bit microprocessor into their $GF(2^m)$ accelerator [42]. We plot points of prior work that were specifically noted to be energy optimal and for which we have an equivalent implementation. Note that we graph results for the sliding-window algorithm as well as the Montgomery

Table 7.2: Latency per operation (100K clock cycles) for binary-field microarchitectures

μarchitecture	Key Size	Sign	Verify	Signature + Verification
Baseline	163-bit	58.8	80.3	139.1
Baseline	233-bit	122.3	166.3	288.6
Baseline	283-bit	182.0	248.7	430.7
Baseline	409-bit	414.4	611.0	1025.5
Baseline	571-bit	1034.9	1420.2	2455.0
ISA Ext	163-bit	9.7	12.5	22.1
ISA Ext	233-bit	18.3	23.5	41.7
ISA Ext	283-bit	24.4	27.4	51.8
ISA Ext	409-bit	55.0	76.6	131.7
ISA Ext	571-bit	136.2	180.0	316.2
W/ Billie	163-bit	1.9	2.3	4.2
W/ Billie	233-bit	3.4	4.0	7.4
W/ Billie	283-bit	4.6	5.4	10.0
W/ Billie	409-bit	9.0	10.6	19.6
W/ Billie	571-bit	16.7	19.7	36.4

scalar point multiplication. In all cases, our Montgomery algorithm implementation outperforms prior work due to the efficient coprocessor interface we employ. Additionally, our sliding-window algorithm implementation outperforms both Montgomery implementations by a significant margin. We felt the comparison to prior work was fair because the functional units in our work are similarly designed [41].

The increased performance of the sliding-window algorithm is responsible for some of the energy efficiency gain in our work. Increased performance leads directly to a shorter run time and a shorter run time leads to a lower amount of energy lost due to static power. The register file in Billie also allows flexibility in algorithm design. Individually, both the sliding-window algorithm used for single-point multiplication

(signature) and the twin-point multiplication (verification) fit in the storage space of Billie, precomputed points included.

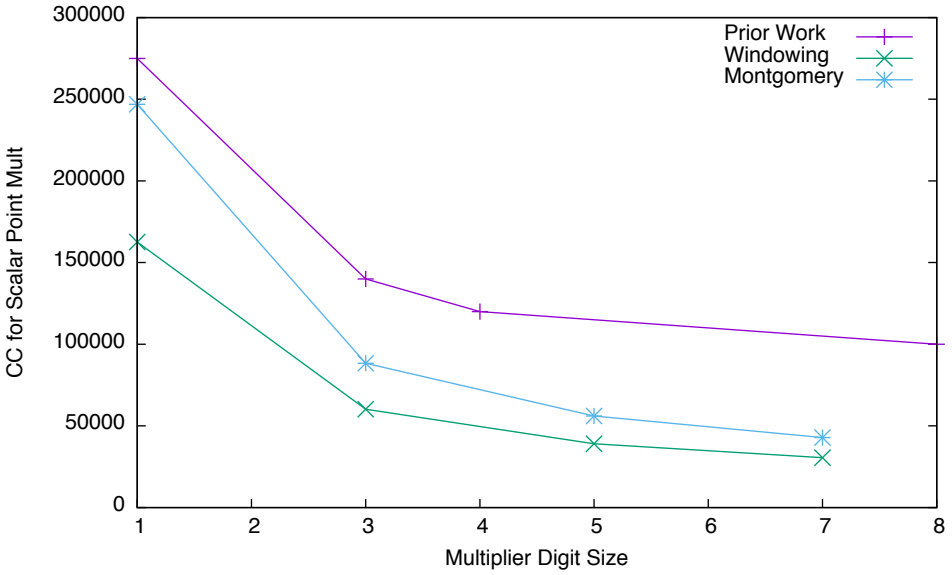


Figure 7.14: Performance for 163-bit scalar point multiply comparing Billie to prior work [42].

7.7 Double Buffer Evaluation

To quantify the energy savings of Monte's instruction reordering scheme, we estimated energy consumption for 384-bit ECDSA with double buffering removed. The results demonstrate that overlapping data movement with computation amounts to a 13.5% improvement in energy consumption. The energy savings come from less idle time for Pete and Monte in addition to a reduction in the number of reads to shared memory. For the 192-bit key size, we measured a 9.4% reduction in energy due to double buffering. Therefore, Monte with double buffering scales better with larger key sizes. This is explained by the increasing time that data movement costs

as the key size grows.

7.8 Baseline Validation

To validate the energy efficiency of our baseline microarchitecture, we measured Pete against a similarly configured Microblaze processor (i.e. 5-stage pipeline, no cache, no MMU, full 32-bit by 32-bit multiplier, binary divider) on the Xilinx Virtex-5 platform [47]. The synthesis results reveal that Pete requires 34.3% more LUT-flip-flop pairs (i.e. more FPGA fabric); however, Pete requires 75.0% fewer Digital Signal Processing (DSP) blocks compared to Microblaze. We attribute the difference in resource consumption to our Karatsuba multiplier. Muti-cycle multiplication performs more addition and requires control logic, all of which utilize LUTs, while parallel multiplication maps well to the DSP hardware blocks on the Virtex-5. This trade-off is a win in the ASIC realm, which is the target technology for this study. In terms of performance, Pete outperforms Microblaze by 17.7% for a 384-bit ECDSA signature and verification operation. Note that this is in spite of a longer latency multiplication unit, which demonstrates an advantage of a separated multiplication unit over ISAs without it.

To validate the efficiency of our multiplier, we synthesized Pete for a 45 nm technology library with various multiplier configurations and measured the power of Pete with each configuration using the methods further explained in the Methodologies section. Compared to Pete with a traditional operand-scanning, multi-cycle multiplier with the same latency, our measurements showed a 4.69% decrease in dynamic power and a 3.47% increase in static power. Because dynamic power dominates, Karatsuba's technique yielded an average power savings of 3.52%. Compared to Pete with a parallel pipelined multiplier as found in many of the modern 32-bit RISC cores, our Karatusba, multi-cycle multiplier demonstrated a 10.6% and 28.4%

improvement in dynamic and static power, respectively. This equates to a 13.4% power savings overall. Further investigation is necessary to determine how much energy savings this yields.

7.9 FFAU Evaluation

This section of the paper provides the results of our FFAU study prior to the development of the full ECC hardware/software system. The purpose of this study was to characterize the core computation logic for our accelerator and to determine the most optimal datapath width in terms of energy and power. To demonstrate the energy efficiency of the FFAU design, we measured the average power and execution time of Montgomery multiplication for key sizes of 192-bit, 256-bit, and 384-bit. These results assume a 100 MHz clock and 0.9V supply voltage for logic and 0.7V for memory. Table 7.3 provides a breakdown of the average static and dynamic power consumed by the 8-bit, 16-bit, 32-bit, and 64-bit variants of our design for each of the Montgomery multiplications. In all cases, dynamic power is the dominant component. This is primarily due to the small memories, low supply voltage, and high utilization of the arithmetic logic. The leakage power provides us some insight into how much power will be consumed if power gating is not utilized while the FFAU is idle.

Table 7.4 provides the total average power along with execution time and energy per Montgomery multiplication with respect to the datapath width. When comparing integer key sizes from smallest to largest, we note that average power only increases slightly, whereas the computation time increases quadratically. The increase in average power is mainly due to a linear increase in memory, which accounts for an increase in leakage power. The significant increase in computation time is due to the $O(n^2)$ nature of the multiplication operation. When comparing datapath

Table 7.3: Area utilization, static power, and dynamic power vs. datapath width.

Datapath Width	Area(cell units)	Static Power	Dynamic Power
Key Size: 192-bit			
8-bit	2,091	32.3 μW	166.2 μW
16-bit	4,244	59.3 μW	311.9 μW
32-bit	11,329	159.1 μW	659.9 μW
64-bit	36,582	530.6 μW	1,472.7 μW
Key Size: 256-bit			
8-bit	2,091	34.0 μW	186.2 μW
16-bit	4,244	61.6 μW	310.2 μW
32-bit	11,327	161.4 μW	684.4 μW
64-bit	36,582	532.9 μW	1,613.4 μW
Key Size: 384-bit			
8-bit	2,168	35.4 μW	197.1 μW
16-bit	4,322	65.0 μW	321.6 μW
32-bit	11,405	164.3 μW	888.5 μW
64-bit	36,664	535.7 μW	1,686.5 μW

bit widths of the FFAU, the average power increases less than quadratically as the datapath width doubles. The net result is that the energy per CIOS operation tends to decrease as the datapath width increases.

To demonstrate this, Figure 7.15 charts the amount of energy consumed per 192-bit, 256-bit, and 384-bit operation for each of the variants of the FFAU. Due to the fact that the CIOS algorithm is not perfectly quadratic, the decrease in energy consumed per operation does not continue. As can be seen for the 192-bit key case with a 32-bit datapath, at some point increasing the datapath width starts to increase the energy consumed, leading to an optimal datapath width in terms of energy for a given key size. We believe this trend continues for larger key sizes; however, the optimal datapath width is greater than or equal to 64-bits.

The results show that an algorithm with a $O(n^2)$ time complexity favors a larger datapath when considering energy efficiency, while the energy efficiency of a $O(n)$

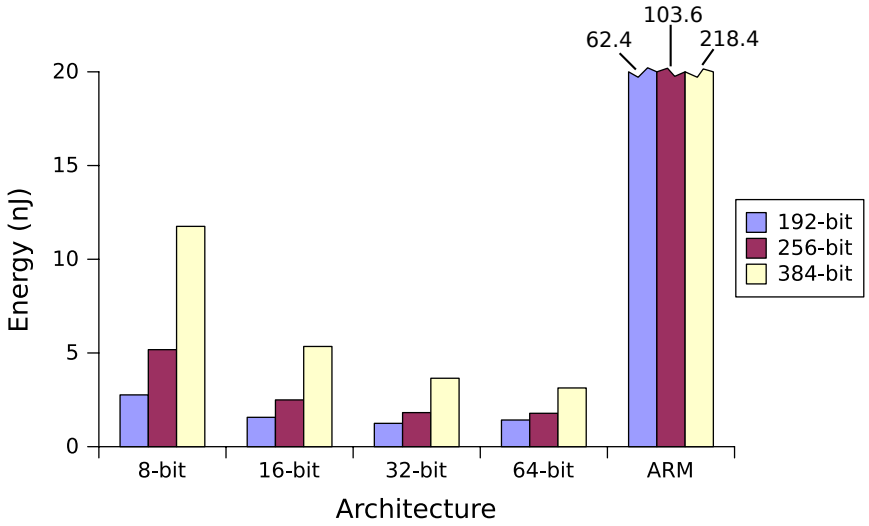


Figure 7.15: Energy per Montgomery multiplication vs. datapath width

algorithm will not be significantly affected by datapath size. Moreover, for an algorithm exhibiting a $O(1)$ behavior, a decrease in datapath size will yield an increase in energy efficiency.

In order to provide some insight into the relative energy efficiency of the FFAU, Figure 7.15 also includes the energy per operation estimations for the ARM Cortex-M3 operating at 100 MHz with a 0.9V supply voltage. Table 7.5 lists the energy estimations for the ARM processor since they extend beyond the scale of the graph in Figure 7.15. In terms of performance, the FFAU on average yields a 10x improvement over the ARM.

Table 7.4: Average power, execution time, and energy per Montgomery multiplication vs. datapath width

Width	Average Power	Ex. Time	Energy
Key Size: 192-bit			
8-bit	198.5 μW	13,920 ns	2.763 nJ
16-bit	371.2 μW	4,220 ns	1.566 nJ
32-bit	819.0 μW	1,520 ns	1.245 nJ
64-bit	2,004.3 μW	710 ns	1.423 nJ
Key Size: 256-bit			
8-bit	220.2 μW	23,510 ns	5.176 nJ
16-bit	371.8 μW	6,710 ns	2.495 nJ
32-bit	845.7 μW	2,150 ns	1.818 nJ
64-bit	2,146.3 μW	830 ns	1.782 nJ
Key Size: 384-bit			
8-bit	232.5 μW	50,550 ns	11.755 nJ
16-bit	386.6 μW	13,830 ns	5.347 nJ
32-bit	888.5 μW	4,110 ns	3.652 nJ
64-bit	2,222.3 μW	1,410 ns	3.133 nJ

Table 7.5: Average power and energy per modular multiplication vs. key size for the ARM Cortex-M3

Key Size	Ex. Time	Average Power	Energy
192-bit	13,870 ns	4,500 μW	62.4 nJ
256-bit	23,010 ns	4,500 μW	103.6 nJ
384-bit	48,530 ns	4,500 μW	218.4 nJ

8. CONCLUSIONS AND FUTURE WORK

In conclusion, we have provided a thorough analysis of the design space for ultra-low energy asymmetric cryptography across a broad range of security levels, including up to 571-bit key sizes. We began by evaluating the energy per asymmetric cryptographic operation (ECDSA signature + verification) on an efficient baseline architecture centered around a pipelined RISC processor. We then included simple, yet beneficial prime-field instruction set extensions to our microarchitecture and evaluated the improvement in terms of energy per operation compared to baseline. Next, we introduced a reconfigurable, prime-field accelerator to our microarchitecture and measured the energy per operation against the baseline and the ISA extended architectures. To reduce the energy impact of instruction fetch from program ROM, we integrated an instruction cache into our ISA extended architecture and evaluated the energy benefit. For a comparison of prime and binary fields, we extended our microarchitecture to include support for binary fields. To do so, we added two carry-less arithmetic instructions to our extended ISA and compared the results to the prime-field implementations. Finally, we augmented our system with a non-configurable, binary-field accelerator and evaluated the energy consumption.

Our analysis showed that the energy benefit of hardware acceleration increases substantially as the required level of security increases. We also demonstrated that, depending on the energy cost of instruction reads, the accelerated microarchitecture can reduce power as well as execution time, which exaggerates the advantages of hardware acceleration when considering an energy-delay product. We will now take a moment to discuss on-going research and future work.

Our evaluation has revealed some interesting avenues for future work. First,

we discovered that over half of Billie's energy is being consumed in the synthesized register file. Thus, we would like to evaluate the energy consumption of Billie with a register file implemented in more efficient memory (SRAM) technology, rather than flip-flops. To do so, we plan on modeling the register file with HSPICE, using a 45nm transistor model [48]. Second, we found that when accelerating $GF(2^m)$, the protocol arithmetic modulo the group order (inversion specifically) becomes the limiting factor, because it does not map to the accelerator. In computer design terminology, Amdahl's law strikes again [21]. Therefore, we plan on investigating various methods for accelerating the modular inversion. Finally, we found that our binary-field accelerator does not scale well in terms of energy efficiency. This is primarily due to the increase in power consumption as the field size increases. As a result, we plan on modeling our system such that we can turn off Billie when she is not in use. Furthermore, we would like to experiment with divide and conquer algorithms in software that would facilitate larger field size computation on a smaller variant of Billie.

Our existing methodologies make use of Cacti to estimate the energy consumed in our memories, including our ROM. A significant portion of our overall energy is being consumed in the ROM, and Cacti was never intended to model ROM. We understand that our use of Cacti for ROM energy estimation introduces inaccuracies in our evaluation. Although we do not feel the inaccuracies would be impactful enough to change the conclusions drawn from this study, more detail modeling of ROM memory would increase our overall confidence in our work and hopefully provide future insight into energy efficient embedded systems. As discussed in Chapter 6, the energy consumption of our logic was evaluated at a 45nm technology node. We would like to perform circuit level modeling of our ROM with Synopsys HSPICE [48]. To do so, we will use the Predictive Technology Model (PTM) provide by Arizona

State University (ASU) for our 45 nm model card [49].

Currently, our system assumes that non-volatile memory is made up of ROM in the purist sense. In other words, the ROM is not reprogrammable after fabrication. For some target devices, such as IMDs, this is an unrealistic assumption. Thus, we would like to model our system assuming a flash Electronic Erasable Programmable ROM (EEPROM) memory technology in place of the ROM. This component of analysis requires an accurate simulation model card of flash technology for a 45nm technology node. This model card provides the HSPICE simulator with the electrical characteristics of the flash cells and is essential for accurate modeling of flash memory.

REFERENCES

[1] G. E. Moore, *Cramming more components onto integrated circuits.* New York, NY: McGraw-Hill, 1965.

[2] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography.* Boca Raton, FL: CRC press, 2010.

[3] D. Halperin, T. Heydt-Benjamin, B. Ransford, S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. Maisel, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," in *IEEE Symposium on Security and Privacy, 2008*, pp. 129–142, 2008.

[4] D. Halperin, T. Kohno, T. S. Heydt-Benjamin, K. Fu, and W. H. Maisel, "Security and privacy for implantable medical devices," *Pervasive Computing, IEEE*, vol. 7, no. 1, pp. 30–39, 2008.

[5] G. Gaubatz, J.-P. Kaps, E. Ozturk, and B. Sunar, "State of the art in ultra-low power public key cryptography for wireless sensor networks," in *Pervasive Computing and Communications Workshops, 2005. PerCom 2005 Workshops. Third IEEE International Conference on*, pp. 146–150, IEEE, 2005.

[6] L. Batina, J. Guajardo, T. Kerins, N. Mentens, P. Tuyls, and I. Verbauwhede, "Public-key cryptography for rfid-tags," in *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops' 07. Fifth Annual IEEE International Conference on*, pp. 217–222, IEEE, 2007.

[7] D. Naccache and D. M'Raihi, "Cryptographic smart cards," *Micro, IEEE*, vol. 16, no. 3, pp. 14–16, 1996.

[8] A. S. Wander, N. Gura, H. Eberle, V. Gupta, and S. C. Shantz, "Energy analysis of public-key cryptography for wireless sensor networks," in *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on*, pp. 324–328, IEEE, 2005.

[9] K. Pabbuleti, D. Mane, and P. Schaumont, "Energy budget analysis for signature protocols on a self-powered wireless sensor node," in *Radio Frequency Identification: Security and Privacy Issues*, pp. 123–136, Springer, 2014.

[10] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.

[11] P. FIPS, "186-4. digital signature standard (dss)," *National Institute of Standards and Technology (NIST)*, 2013.

[12] W. Diffie and M. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, vol. 22, no. 6, pp. 644–654, 1976.

[13] N. R. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha, "A study of the energy consumption characteristics of cryptographic algorithms and security protocols," *Mobile Computing, IEEE Transactions on*, vol. 5, no. 2, pp. 128–143, 2006.

[14] W. Trappe and L. C. Washington, *Introduction to Cryptography with Coding Theory (2nd Edition)*. Upper Saddle River, NJ: Prentice-Hall, Inc., 2005.

[15] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[16] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. New York, NY: Springer, 2004.

[17] J. López and R. Dahab, "Fast multiplication on elliptic curves over gf (2m) without precomputation," in *Cryptographic Hardware and Embedded Systems*, pp. 316–327, Springer, 1999.

[18] V. Gupta, D. Stebila, S. Fung, S. C. Shantz, N. Gura, and H. Eberle, "Speeding up secure web transactions using elliptic curve cryptography," in *11th Annual Network and Distributed System Security Symposium (NDSS 2004)*, vol. 364, 2004.

[19] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of elliptic and hyperelliptic curve cryptography*. Boca Raton, FL: CRC press, 2010.

[20] D. Sweetman, *See mips run linux*. San Francisco, CA: Elsevier, Inc., 2 ed., 2007.

[21] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. San Francisco, CA: Elsevier, Inc., 5 ed., 2012.

[22] G. Yip, "Expanding the synopsys primetime® solution with power analysis." `http://www.synopsys.com/Tools/Implementation/SignOff/CapsuleModule/ptpx-wp.pdf`, 2006.

[23] A. D. Targhetta and P. V. Gratz, "An Energy Efficient Datapath for Asymmetric Cryptography," in *The 3rd Workshop on Energy Efficient Design (WEED)*, 2011.

[24] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen, "A dynamic voltage scaled microprocessor system," *Solid-State Circuits, IEEE Journal of*, vol. 35, no. 11, pp. 1571–1580, 2000.

[25] M. Keller, A. Byrne, and W. P. Marnane, "Elliptic curve cryptography on fpga for low-power applications," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 1, p. 2, 2009.

[26] J. Goodman and A. P. Chandrakasan, "An energy-efficient reconfigurable public-key cryptography processor," *Solid-State Circuits, IEEE Journal of*, vol. 36, no. 11, pp. 1808–1820, 2001.

[27] E. Wenger and M. Hutter, "Exploring the design space of prime field vs. binary field ecc-hardware implementations," in *Information Security Technology for Applications*, pp. 256–271, Springer, 2012.

[28] NSA, "Mathematical routines for the nist prime elliptic curves." `www.nsa.gov/ia/_files/nist-routines.pdf`, 2010.

[29] H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates," in *Advances in CryptologyASIACRYPT98*, pp. 51–65, Springer, 1998.

[30] J. Großschädl and E. Savaş, "Instruction set extensions for fast arithmetic in finite fields gf(p) and gf(2^m)," in *Cryptographic Hardware and Embedded Systems-CHES 2004*, pp. 133–147, Springer, 2004.

[31] M. Brown, D. Hankerson, J. López, and A. Menezes, *Software implementation of the NIST elliptic curves over prime fields*. New York, NY: Springer, 2001.

[32] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[33] C. Kaya Koc, T. Acar, and B. S. Kaliski Jr, "Analyzing and comparing montgomery multiplication algorithms," *Micro, IEEE*, vol. 16, no. 3, pp. 26–33, 1996.

[34] D. Hankerson, J. L. Hernandez, and A. Menezes, "Software implementation of elliptic curve cryptography over binary fields," in *Cryptographic Hardware and Embedded SystemsCHES 2000*, pp. 1–24, Springer, 2000.

[35] B. Parhami, *Computer arithmetic: algorithms and hardware designs*. New York, NY: Oxford University Press, Inc., 2009.

[36] J. Großschädl, A. Szekely, and S. Tillich, "Algorithm exploration for long integer modular arithmetic on a sparc v8 processor with cryptography extensions," in *Proceedings of the 2nd International Conference on Embedded Software and Systems (ICESS 2005)*, pp. 187–194, 2005.

[37] A. D. Targhetta, D. E. Owen Jr., and P. V. Gratz, "The Design Space of Ultra-low Energy Asymmetric Cryptography," in *The IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[38] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 2009.

[39] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pp. 364–373, IEEE, 1990.

[40] M. Koschuch, J. Lechner, A. Weitzer, J. Großschädl, A. Szekely, S. Tillich, and J. Wolkerstorfer, "Hardware/software co-design of elliptic curve cryptography on an 8051 microcontroller," in *Cryptographic Hardware and Embedded Systems-CHES 2006*, pp. 430–444, Springer, 2006.

[41] S. Kumar, T. Wollinger, and C. Paar, "Optimum digit serial gf(2^m) multipliers for curve-based cryptography," *Computers, IEEE Transactions on*, vol. 55,

no. 10, pp. 1306–1311, 2006.

[42] X. Guo and P. Schaumont, "Optimizing the hw/sw boundary of an ecc soc design using control hierarchy and distributed storage," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 454–459, European Design and Automation Association, 2009.

[43] S. Anderson, J. Earle, R. Goldschmidt, and D. Powers, "The ibm system/360 model 91: floating-point execution unit," *IBM Journal*, vol. 11, no. 1, pp. 34–53, 1967.

[44] S. Haider, Virginia Tech, "Power estimation with osu standard cell library and synopsys tools (primetime-px)." `https://computing.ece.vt.edu/wiki/Synopsys_Tutorial:_Power_Estimation`, 2008.

[45] W. Snyder, "Verilator 3.681." `http://www.veripool.org/wiki/verilator`, 2008.

[46] J.-M. Puiatti, C. Piguet, E. Sanchez, and J. Llosa, "Low-power vliw processors: A high-level evaluation," in *in proc. of PATMOS 98, October*, pp. 399–408, 1998.

[47] Xilinx, "Microblaze processor reference guide ug081 (v9.0)." `http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf`, 2008.

[48] Synopsys, "HSPICE user guide: Simulation and analysis." `www.synopsys.com`, 2008.

[49] Arizona State University, "Predictive technology model." `ptm.asu.edu`, 2011.