# AN EMBEDDED RISC-V CORE WITH FAST MODULAR MULTIPLICATION

by

Ömer Faruk Irmak

B.S., Computer Engineering, Istanbul Technical University, 2017

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2020

# ACKNOWLEDGEMENTS

I would like to first thank my thesis advisor Prof. Dr. Arda Yurdakul for her guidance and encouragement throughout development of my thesis. Her patience and continuous support were invaluable for me to continue this work.

I am also thankful that I had the chance to work with my friends at our RISC-V work group; Alp, Fatih and İbrahim. Together, they made my time through graduate school memorable.

Lastly, finishing this work would have not been possible without support from my family that helped me get through the stressful times and kept me going. I am forever in debt to them.

June 2020 Ömer Faruk IRMAK

# ABSTRACT

## AN EMBEDDED RISC-V CORE WITH FAST MODULAR MULTIPLICATION

While one of the biggest enabling factors of Internet of Things growth is cheap and capable hardware, maybe the biggest concern is privacy and security. Encryption and authentication need big power budgets, which battery-operated IoT end-nodes do not have. Hardware accelerators designed for specific cryptographic operations provide little to no flexibility for future updates. Custom instruction solutions are smaller in area and provide more flexibility for new methods to be implemented. One drawback of custom instructions is that the processor has to wait for the operation to finish. Eventually, the response time of the device to real-time events gets longer. In this work, we propose a processor with an extended custom instruction for modular multiplication, which blocks the processor, typically, two cycles for any size of modular multiplication. We adopted embedded and compressed extensions of RISC-V for our proof-of-concept CPU. Our design is benchmarked on recent cryptographic algorithms in the field of elliptic-curve cryptography. Our CPU with 128-bit modular multiplication operates at 136MHz on ASIC and 81MHz on FPGA. It achieves up to 13x speed up over software implementations while reducing overall power consumption by up to 95% with 41% average area overhead over our base architecture.

# ÖZET

## HIZLI MODÜLER ÇARPMA KABİLİYETLİ GÖMÜLÜ RISC-V İŞLEMCİ ÇEKİRDEĞİ

Nesnelerin İnterneti büyümesini sağlayan en büyük etkenlerden biri ucuz ve yetenekli donanım iken belki de en büyük endişe gizlilik ve güvenliktir. Gizlilik ve güvenlik sağlamak için gereken şifreleme ve kimlik doğrulama, pille çalışan Nesnelerin İnterneti uç düğümlerinin sahip olmadığı büyük güç bütçelerine ihtiyaç duyar. Literatürdeki mevcut donanım hızlandırıcıları belirli iş yükleri için tasarlanmışlardır, bu sebeple gelecekteki güncellemeler için çok az esneklik sağlar veya hiç esneklik sağlamazlar. Özel komut tabanlı çözümler, alan maliyeti olarak daha küçüktür ve uygulanacak yeni yöntem ve algoritmalar için daha fazla esneklik sağlarlar. Özel komutların bir dezavantajı, sistemin işlem bitene kadar beklemesi gerekmesidir. İşlem çok uzun sürdüğünde, cihazın gerçek zamanlı olaylara yanıt süresi uzar. Bu çalışmanın amacı modüler çarpma için özel komutla genişletilmiş bir işlemci önermektir. Bu yaklaşımda modüler çarpma, tipik bir durumda, işlemciyi iki saat çevrimi boyunca engelleyebilir. RV32EC komut setini temel aldığımız ve Verilog ile geliştirdiğimiz tasarımımız, Eliptik Eğri Kriptografisi (ECC) alanındaki güncel şifreleme algoritmaları üzerinde denenmiştir. 128 bit modüler çarpma içeren uygulamaya özel tümdevre (ASIC) tasarımında 136 MHz saat hızına ve alanda programlanabilir kapı dizileri (FPGA) üzerinde 81 MHz saat hızına ulaştık. Yazılımsal çözüme kıyasla çeşitli kriptografik eğrilerde on üç kata kadar hız artışı elde ederken, temel mimarimiz üzerinde ortalama % 41 alan artışı ile toplam güç tüketimini % 95'e kadar azaltmayı başardık.

# TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . . . 3
ABSTRACT . . . . . 4
ÖZET . . . . . i
LIST OF FIGURES . . . . . iii
LIST OF TABLES . . . . . v
LIST OF ACRONYMS/ABBREVIATIONS . . . . . vii
1. INTRODUCTION . . . . . 1
2. RELATED WORK . . . . . 4
2.1. Programming Oversights . . . . . 4
2.2. Network . . . . . 5
2.3. Encryption . . . . . 6
2.4. Security in IoT . . . . . 7
2.5. RISC-V . . . . . 9
3. ELLIPTIC CURVE CRYPTOGRAPHY (ECC) . . . . . 12
4. SPA-ATTACK RESISTANT MONTGOMERY MULTIPLICATION DATAPATH DESIGN . . . . . 15
5. MONTGOMERY MULTIPLICATION INSTRUCTION FOR RISC-V ISA . 19
5.1. Partial Execution Mode . . . . . 22
6. ANALYSIS . . . . . 24
6.1. Base Architecture . . . . . 24
6.2. Benchmarks . . . . . 26
6.3. Attack Analysis . . . . . 35
6.3.1. Passive Attacks . . . . . 36
6.3.2. Active Attacks . . . . . 40
7. CONCLUSION . . . . . 42
REFERENCES . . . . . 45

# LIST OF FIGURES

Figure 3.1. Hierarchy of ECC operations . . . . . 12

Figure 3.2. Sign/Verify runtime profile on a RV32EC architecture . . . . . 13

Figure 4.1. R2MM Pseudo-code, Multiplicand (A), Multiplier (B), Modulus (N), Result (S), $i^{th}$ bit of A ($A_i$) . . . . . 16

Figure 4.2. Basic R2MM Hardware . . . . . 17

Figure 4.3. MMUL Operation Pseudocode . . . . . 18

Figure 5.1. Candidate RISC-V instruction formats . . . . . 19

Figure 5.2. Memory layout for I-type MMUL . . . . . 20

Figure 5.3. Memory layout for R-type MMUL . . . . . 20

Figure 5.4. Memory layout for R4-type MMUL . . . . . 21

Figure 5.5. Integration of MMUL in datapath . . . . . 21

Figure 5.6. MMUL partial execution time diagram . . . . . 22

Figure 5.7. MMUL control register . . . . . 23

Figure 6.1. Base Architecture . . . . . 25

Figure 6.2. Modular multiplication function for FourQ implemented with MMUL custom instruction with atomic execution . . . . . 27

Figure 6.3. Modular multiplication function with Partial Execution enabled for FourQ . . . . . 28

Figure 6.4. Activity Graphs (a) Software, (b) Custom Instruction with Atomic Execution, (c) Custom Instruction with Partial Execution, (d) Partial Execution with Interrupts In Between . . . . . 37

Figure 6.5. Activity Graphs for Three MMUL Runs over GF($2^{127}$ -1) . . . . . 38

Figure 6.6. Crosscorrelation Graphs for Three MMUL Runs . . . . . 39

# LIST OF TABLES

Table 6.1. CPU Benchmark . . . . . 24

Table 6.2. Runtime of benchmarks in clock cycles, Base RV32EC Architecture (BA),
tom Instruction with Atomic Execution (CI-AE), Custom Instruction with Partial Execution (CI-PE) . . . . . 30

Table 6.3. Runtime of benchmarks in clock cycles, Base RV32EC Architecture (BA),
tom Instruction with Atomic Execution (CI-AE), Custom Instruction with Partial Execution (CI-PE) . . . . . 31

Table 6.4. Runtime of benchmarks in clock cycles, Base RV32EC Architecture (BA),
tom Instruction with Atomic Execution (CI-AE), Custom Instruction with Partial Execution (CI-PE) . . . . . 32

Table 6.5. Average Power Consumption (W) During A Modular Multiplication 33

Table 6.6. Average Dynamic Power Consumption Per Module (W) During A Modular Multiplication . . . . . 33

Table 6.7. Normalized Energy Consumption (Power x Clock Cycles) . . . . . 33

Table 6.8. Post P&R Frequency and Area (Xilinx XC7Z020-1 FPGA) . . . . 34

Table 6.9. Post Synthesis Frequency and Area (OSU018) . . . . . 34

Table 6.10. Overhead Comparison . . . . . 35

# LIST OF ACRONYMS/ABBREVIATIONS

| | |
|---|---|
| ISA | Instruction Set Architecture |
| RISC | Reduced Instruction Set Computing |
| IoT | Internet of Things |
| M2M | Machine to machine |
| ECC | Elliptic Curve Cryptography |
| ECDH | Elliptic-curve Diffie–Hellman |
| ECDSA | Elliptic Curve Digital Signature Algorithm |
| R2MM | Radix-2 Montgomery Multiplication |
| CSR | Control and Status Register |
| BA | Base RV32EC Architecture |
| CI-AE | Custom Instruction with Atomic Execution |
| CI-PE | Custom Instruction with Partial Execution |
| CISC | Complex Instruction Set Computing |
| FPGA | Field Programmable Gate Array |
| ASIC | Application Specific Integrated Circuit |
| CVE | Common Vulnerabilities and Exposures |
| DoS | Denial of Service |
| ARP | Address Resolution Protocol |
| IP | Internet Protocol |
| WSC | Warehouse Scale Computing |

# 1. INTRODUCTION

IoT market has been one of the driving forces of embedded hardware. It is projected to be a multi-billion dollar industry with billions of devices in operation [1]. Key enabler that is a must for this forecast to become a reality is cheap and capable hardware. There are multiple efforts, both in academia and industry, that are aiming to bring costs lower while making hardware more efficient in the tasks it is designed to perform. IoT end-node hardware should be secure and designed with power consumption in mind. Neither encryption nor authentication are light operations for a barebones hardware to handle efficiently. Therefore, there are efforts on both designing new lightweight algorithms [2] that suit better to less powerful processors and designing specialized hardware that tackles the heavy operations more efficiently [3].

IoT end-node workloads can be characterized as low frequency bursts of computationally expensive operations with sleep states in between [4] [5]. When designing a solution for energy efficient IoT end-nodes, low power modes should be utilized to the fullest extent. By definition, IoT devices are connected to a network and transfer the data they produce to another host. Secure communication involves complex encryption and authentication algorithms which considerably extend the on-state duration when implemented solely in software [6].

There exist implementations for faster cryptographic operations that would allow better utilization low power modes. They mainly adopt one of the two prominent approaches, which are designing custom accelerators [3] [7] [8] and extending the processor architecture with custom instructions [9] [10] [11]. Although high performance can be achieved with parallel and pipelined custom accelerators, the area overhead for heavily constrained IoT devices are unacceptable. Another drawback of custom accelerators is the lack of adaptability. They are designed to be highly effective for specific operations but they can only be used for a fixed set of operations. Security domain faces frequent developments. A single known vulnerability in a system is enough for someone to manipulate it however they like. For example, a device using vulnerable

NIST P-256 [12] to verify its signed firmware can be exploited to run unauthorized code. There are multiple known methods of exploiting RSA [13]. Lenstra et. al. [14] show that 0.2% of RSA public keys that can be found on the internet can be used to recover their RSA secret keys.

Custom instructions can be utilized for accelerating cryptographic operations, as well. Fundamental and complex operations in cryptography can be mapped to custom instructions and implemented in hardware with less resources compared to full custom accelerators. This makes using the same hardware for different algorithms possible as custom instructions can be used to implement any algorithm. If the current solution used turns out to be vulnerable, different solutions can be implemented via a software update without any significant performance penalty.

Unlike custom accelerators, custom instructions block the processor for the duration of its execution. For real-time applications one of the key performance metrics is the response time of the device. An industrial IoT node at the machine-to-machine (M2M) interface [15] or a hard real-time system such as a flight controller avionic may require to get precise readings from its environment as well as communicating over an encrypted channel [16] [17]. Instruction latency should be low enough to allow handling real time events as fast as possible.

Software ecosystem has been heavily influenced by open source and free software for a considerable amount of time, yet hardware arena has mostly sticked to proprietary technologies for so long. RISC-V [18], an open source and loyalty free ISA, backed by its foundation [19] is trying to mainstream the open source mentality in the computing hardware industry. Many open source implementations of RISC-V have been done both in academia [20] [21] [22] [23] and industry [24]. With support for custom extensions enabling novel implementations and being license free makes the RISC-V perfect for academic work like this one and for-profit corporations.

In this work we have designed a microprocessor core with its ISA extended with a custom instruction for Montgomery multiplication. Modular multiplication is highly

utilized in public key cryptography. Our proposed custom instruction implementation can be executed both atomically and partially in short iterations, therefore does not degrade system response time. We implemented Embedded and Compressed extensions of RISC-V (RV32EC) [19] as the base ISA of our proof-of-concept CPU, which is designed in Verilog. Design is benchmarked with operations on various cryptographic elliptic curves. Synthesis is done for both FPGA and ASIC targets to collect area and power consumption metric.

The remaining of this thesis is structured as follows. Section 2 provides a summary of previous work done in this field. Section 3 gives a brief introduction to anatomy of Elliptic Curve Cryptography (ECC) operations. Section 4 explains the hardware and software design choices made for the acceleration. Section 5 describes the microarchitecture designed as the base platform and gives benchmark results. Section 6 analyses the possible attack vectors against our design and Section 7 concludes the thesis.

# 2. RELATED WORK

Security of the data that our connected devices carry and generate is one of the biggest concerns of modern world. From our phones to fridges, everything we own somehow connects to the Internet. This allows a huge set of attack surfaces to be exposed to everyone on the Internet. Every device connected to the Internet, including IoT devices, suffer from similar vulnerabilities that can be at any point of the system. Programming errors can be turned into exploitable vulnerabilities with well crafted inputs to the system. Network traffic that the target generates or other vulnerable devices on the network can be exploited as well. Algorithmic or implementation weaknesses in key security infrastructure like encryption is another vector that attackers use. Security researchers are working hard to discover and propose countermeasures for possible vulnerabilities that can be exploited. Discovered vulnerabilities are first reported to relevant parties so that a fix can be deployed. After a certain period, researchers get to publish their findings and the vulnerability gets added to Common Vulnerabilities and Exposures (CVE) list [25].

## 2.1. Programming Oversights

Bugs introduced into software during development or oversights during the design phase are prominent sources of vulnerabilities [26]. Programming languages like C or C++ that put the sole responsibility of security, especially memory management, on developer and provide little to no guidance. A C programmer has to manage all the memory allocations which raises ownership of memory issues. This may result in errors like use-after-free (accessing a dynamically allocated memory that is already given back to the system) or a novice programmer can easily try to share local storage variables (ie. stack allocated) with functions outside its scope. But they are preferred for the low level access they provide to the system and their superior performance. New programming languages like Rust [27] are trying to provide similar levels of performance while not sacrificing from security. Software vulnerabilities may even be exploited for privilege escalation and arbitrary code execution. Privilege escalation can be summarized as an

attacker gaining higher level access to system resources that a regular user normally would not be able to access [28]. Arbitrary code execution exploits enable attacker to issue any command or execute their own code on the target system [29].

One of the most frequent programming error is buffer overflows [30] [31] [32] [33]. It happens when a program reads or writes to an index that is beyond the original buffer boundary. The attacks are crafted to cause overflow and access memory locations that user normally cannot. Changed memory location can be a part of the program state and cause program to behave differently. Overwritten sections can also be part of the executable code of the program. This can result in completely arbitrary code execution [34], which will naturally give an attacker almost infinite possibilities. Some CPU architectures support disabling execution [35] [36] on selected memory pages to prevent such attacks.

Format string vulnerabilities are also frequent in programs using standard C library functions. Functions like $printf/scanf$ take a format string and access program stack according to the format string. When user input is used as the format string, data in the stack can be read or changed easily. Similar to the buffer overflows, a wide range of possibilities from changing program state to running arbitrary code exist [37] [38].

## 2.2. Network

Devices that are connected to a Wide Area Network, like the Internet, is always under the threat of being target to an attack. Security of the network plays a great deal in security of the connected devices. First step of securing a network is keeping the attacker off the network. This requires an authentication mechanism to verify user identities, which may be an ECC based signature. ECC based key pairs can be used to sign a message and that message can later be used to verify the senders identity. If the network is public or the attacker happens to be an authorized user, access control policies [39] [40] to restrict the actions users can take in the network and infrastructure to monitor the user actions or network traffic should be in place to detect any anomaly in the network and take an appropriate action. Mostly, network based attacks aim to

either disrupt the service or extract valuable information from the traffic [41].

A widely practiced attack is called man-in-the-middle attacks. Attackers intercept the traffic in the network and analyze it to use for their advantage. If the traffic is not encrypted or the encryption can be cracked, attackers can extract valuable information like login credentials, credit card information or impersonate one end of the communication. Using an authentication algorithm like Elliptic Curve Digital Signature Algorithm (ECDSA) combined with a symmetric encryption can would render such an attack ineffective [34].

Denial of Service (DoS) attacks aim to cripple a computer system so that it cannot fulfill its primary tasks [42]. Attackers build botnets [43] with compromised devices that are infected by malware or already run exploitable software. Devices in the botnet generate traffic to the target system and aim to overwhelm it's network and processing capabilities. DoS attacks can be also amplified thorough what is called "UDP Amplification". Amplification is done by a UDP server when a small packet to the server triggers a big response packet. Botnet sends small query packets with a spoofed source address and server sends the amplified traffic to the target system.

Multiple network protocols are susceptible to spoofing. Address Resolution Protocol (ARP) messages can be spoofed to associate a MAC address to an Internet Protocol (IP) address which can be used to direct any traffic to any host. MAC address of a device, which is supposed to be globally unique and static, can also be changed with software. IP addresses, as seen in the UDP Amplification example, can also be spoofed to impersonate another system.

## 2.3. Encryption

Symmetric encryption algorithms like AES [44] are widely used for secure communication and storage of data. Data is both encrypted and decrypted using the same secret key. Assuming that the secret key is known to both parties and not communicated over an observable medium, they provide sufficient levels of security. Algorithms

like Diffie-Hellman Key Exchange [45] make it possible to generate a secret key shared between two ends of the communication, without ever transmitting the secret over the network. Attacks against symmetric encryption schemes stem from implementations leaking secret information via different side channels [46]. Variations in execution time, power consumption [47] or memory access pattern [48] that is dependent on the secret key can be analyzed to extract the secret key.

Public key encryption is a different approach, where the encryption and decryption process requires two keys. The public key is used to encrypt the data. Only the private key can be used for correctly decrypting the ciphertext. This allows the public key to be freely distributed. Asymmetric encryption keys are much longer compared to symmetric counterparts and the encryption/decryption process is slower. It is widely used for authentication. A message can be signed with the private key and the origin of the message can be verified with the public key. After the authentication, a shared symmetric secret key can be generated using algorithms like Diffie-Helmman or Elliptic Curve Diffie-Hellman [49] to encrypt the remaining traffic using symmetric encryption. RSA and ECC are two widely used public key encryption schemes. RSA has been shown to be vulnerable because of both algorithmic and implementation weaknesses [50] [51] [52]. Different ECC curves are also subject to side channel [53] and structural attacks [54].

## 2.4. Security in IoT

IoT networks consist of various layers and may span over large areas [55]. They should be secured in every point, from gateways to end-nodes, that can be used as an attacking point. Since end-nodes are most of the time easier to access for attackers, they should be the most hardened. Auer et. al. [56] propose "Universal Sensor Platform" as a secure hardware base to be used for sensor application end-nodes. One of the key components of their design are encryption and authentication, which they claim that can be carried out efficiently with the help of accelerators and ISA extensions that open source hardware brings.

For security, authentication is crucial. But authentication schemes are costly to use. Efforts like FourQ [57] and ARIS [58] and [59] try to lower the computational costs so that ECC based authentication is accessible to less capable devices as well. While lowering computational complexity of ECC, accelerating the process with specialized hardware is also possible.

ECC acceleration is a highly studied topic since its inception [60]. Main improvements in ECC acceleration are high throughput and low latency computation that comes with higher power consumption and bigger area trade off. Different methods like pipelining [61], precomputation [62], and parallelism [63] are used to achieve high performance. Although they provide high performance, they require additional hardware. Increasing the hardware complexity would in return increase the cost of development and manufacturing. Application areas such as IoT are sensitive to cost increases and work being done in the field should be in alignment with that.

ECC operations can be accelerated at different levels of abstraction and higher levels would be implemented in software. Scalar multiplication is the most studied operation [64] [65] as it is a computation demanding and frequent operation in ECC calculations. It is a necessity to have an efficient multiplication implementation. But the high complexity of the operation requires an equally complex hardware that is not suitable to be embedded in tightly constrained hardware. Point doubling and point squaring, which are simpler operations, can also be accelerated and combining these operations to carry out more complex operations like scalar multiplication is possible. Harb and Jarrah [8], took this approach for embedded applications yet the resulting hardware is magnitudes of larger than the platforms we are targeting. The measures they have taken against side channel attacks made the hardware even bigger. Ultra small platforms require smaller operations to be accelerated in hardware and rest to be implemented in software.

Previous work mostly focuses on customized, one time generated hardware for specific Galois Fields [7] [66] [67], which limits the adaptability of hardware to various or new curves that may come up in the lifetime of the systems that would use the

proposed solutions. Security of established standards day by day are found out to be compromised, being future proof is a key feature in this ever evolving domain of security.

Fully software implementations are also available [68] [69] [70]. While they can be flexible, they are very susceptible to side channel attacks. Efficient implementations require clever tricks like hand written assembly routines for better performance that limit the portability of the code and may leak important information via these side channels. When software implementations are hardened against these attacks, performance loss is introduced. Performance and energy efficiency is sub par compared to hardware implementations. Fully software implementations should be last resort for platforms that are both constrained by their hardware and power budget.

Heterogeneous and connected nature of some IoT networks enable work to be distributed between nodes to achieve overall better operation. Chang et. al. [71] offload the heavy cryptographic operations to more capable, GPU accelerated gateway nodes.

Similar to those done for ECC, acceleration efforts [72] [73] were shown for other algorithms, like RSA, that utilize modular arithmetic. A balance between speed and flexibility is targeted without much regard to hardware size. To balance the hardware/-software parts of their target algorithms, a careful approach is taken while designing both. Even with these efforts resulting software portability is low with no universal interface between hardware and software.

## 2.5. RISC-V

Software ecosystem has been heavily influenced by open source and free software for a considerable amount of time, yet hardware arena has mostly sticked to proprietary licenses for so long.

For decades there have been many different ISAs backed by industry or academia.

But not many have lived to see today, the ones still active are cumbered by licenses. This mainly have made such free and openness movement impossible with hardware. There are many different ways a free ISA could benefit the scene by creating a truly free open processor market;

(i) Free market will flourish competition, thus improving overall quality
(ii) Increase design reuse by enabling open-source designs which would ease the overall effort
(iii) Decrease the overall cost of SoC development, making cheap IoT nodes possible

Assuming the question whether if we need an open ISA or not is answered, next question is how it should be designed. Reduced Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC) are two dominant methodologies used to build an ISA around them. Yet, current implementations of CISC ISAs mostly translate high complexity instructions to RISC-like simple operations.

Relevant new CISC ISA development have been non-existent for a while. Even if CISC ecosystem was highly active, it may not be the best choice when it comes to designing a free ISA. CISC processors today are highly complex due to legacy code support, they need translate even the oldest instructions, that aren't relevant in any shape or form for new projects, to the microinstructions they actually support. That brings a lot of complexity to the design. Obviously, a new ISA won't have any legacy code to support, it only makes sens to ditch the middle translation layer for a new design.

Although commercial RISC ISAs becoming more like CISC these days, RISC domain has some active projects trying to stick to their roots. There are even some projects in the free domain, to name a few, relevant ones are OpenRISC, SPARC and RISC-V. SPARC family has both free and proprietary versions, OpenRISC and RISC-V are true free ISAs. What stands out RISC-V from other alternatives is that it is designed with modern concerns an ISA should address in mind.

RISC-V was designed with multiple use cases from low power to Warehouse Scale Computing (WSC) in mind [18];

(i) Base-plus-extension ISA scheme and compressed instructions target efficiency in constrained systems such as mobile devices and IoT nodes.
(ii) Quadruple Precision, Double Precision, Single Precision floating-point support for WSC
(iii) 32-bit, 64-bit and 128-bit address spaces for all ranges of devices from IoT to WSC systems

With the momentum RISC-V has, it's argued that it will be the Linux of hardware in the near future. Essentially it's an easy choice when it comes to picking up a free ISA to support and help build the ecosystem.

Both software and hardware ecosystem around RISC-V adopts free/open-source development mindset. Currently it is mostly driven by the RISC-V foundation [19] and its members. Foundation certifies implementations of RISC-V, maintains the ISA and organizes workshops all around the world frequently to get people interested in RISC-V collaborate easier.

RISC-V user [74] and privileged [75] ISA specifications are the ultimate texts to reference when doing any work related to RISC-V. User-level specification defines the base instruction and the extensions around that while privileged architecture specification defines the system level functionality that needs to be implemented by RISC-V systems in order to support complex operating systems and external devices.

# 3. ELLIPTIC CURVE CRYPTOGRAPHY (ECC)

ECC is a public-key cryptography approach that utilizes elliptic curves over finite fields. This approach is an alternative to the RSA algorithm, as it allows shorter length keys to achieve equivalent security level of much longer RSA keys [76].

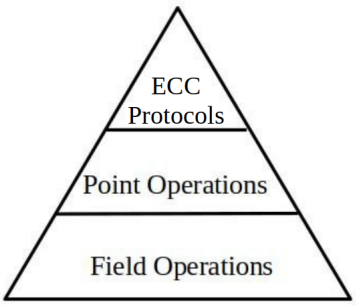



Figure 3.1. Hierarchy of ECC operations

As it can be seen in Figure 3.1, ECC operations can be broken down to different levels of abstraction. This hierarchy of operations can be split into two parts at any level to implement one part in software and the other part in hardware. Each step of this hierarchy takes increasingly more resources to implement in hardware. Decision of how to make the split should be made with performance and cost implications in mind.

Highest level in the hierarchy is the protocol level. Algorithms like Elliptic-curve Diffie–Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA) reside in this level. They define how key pairs should be generated, exchanged and used to sign/verify messages. They rely on mathematical complexity of point operations on an elliptic curve to provide the security. Some of the point operations on an elliptic curve like scalar multiplication (3.1), point doubling (3.2) and point squaring (3.3) can be formulated as;

$$P' = k * P \tag{3.1}$$

$$P' = 2 * P \tag{3.2}$$

$$P' = P^2 \tag{3.3}$$

where $P$ and $P'$ are points on the curve while $k$ is a scalar. Elliptic curves are defined over a Galois field and all point operations involve modular operations over these fields. They reside at the bottom layer of abstraction hierarchy.

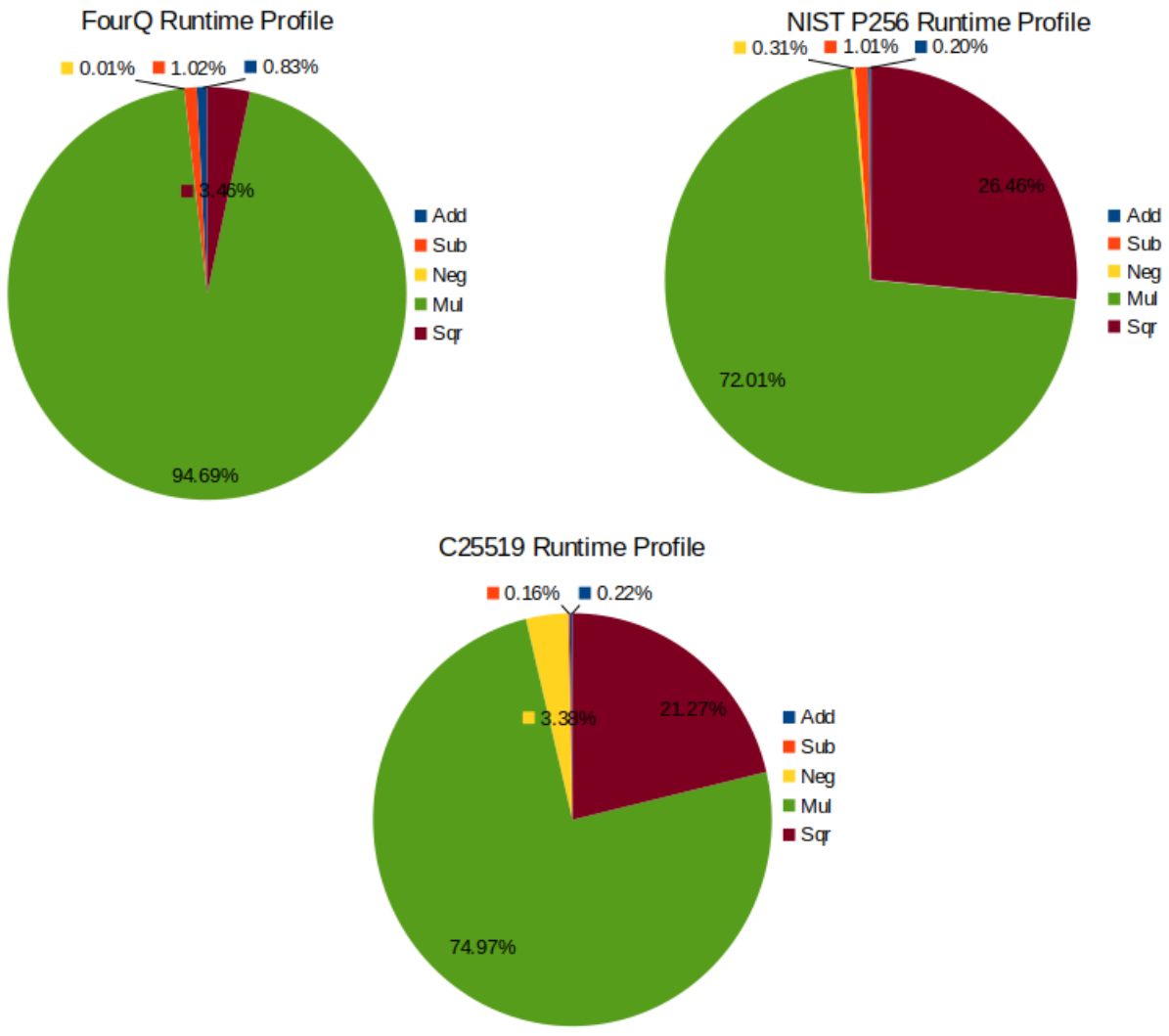



Figure 3.2. Sign/Verify runtime profile on a RV32EC architecture

Smallest building block of ECC operations are finite field operations like modular multiplication, modular addition, modular squaring. Accelerating these operations is a good starting point to achieve overall better performance and efficiency. We have profiled the time that is spent on the finite field arithmetic in sign/verify operations on different curves like FourQ [57], NIST P-256 [12] and Curve25519 [77] to determine which operations would be more beneficial to accelerate in RV32EC. Results in Figure 3.2 show that multiplication is the single most important operation to be accelerated in a fully software implementation.

Figure 3.2 shows that finite field multiplication is a good candidate for acceleration in constrained devices because while it is less demanding in terms of hardware resources compared to operations higher in the hierarchy, it is a real performance hog in ECC operations. It can also be used to carry out squaring operations as well.

# 4. SPA-ATTACK RESISTANT MONTGOMERY MULTIPLICATION DATAPATH DESIGN

Modular multiplication is the operation defined by the Equation (4.1):

$$P = (A * B) \ mod \ N \tag{4.1}$$

One of the key efficient algorithms in this area is Montgomery Multiplication [78]. Montgomery Multiplication replaces trial division operation with division-by-a-power-of-2 which can be efficiently executed on a general purpose processor since the numbers are represented in binary form. For operands with length of $n$ in bits, Montgomery multiplication calculates the Equation (4.2):

$$MMUL(A, B, N) = (A * B * R^{-1}) \ mod \ N \tag{4.2}$$

where;

$$R = 2^n \tag{4.3}$$

$$2^{n-1} < N < 2^n \tag{4.4}$$

$$gcd(R, N) = 1 \tag{4.5}$$

To get the result $P$, four Montgomery Multiplications should be performed assuming R' is precalculated.

$$R' = R^2 \ mod \ N \tag{4.6}$$

1. Bring operand $A$ to Montgomery domain

$$A' = MMUL(A, R', N) \tag{4.7}$$

2. Bring operand $B$ to Montgomery domain

$$B' = MMUL(B, R', N) \tag{4.8}$$

3. Perform multiplication in Montgomery domain

$$P' = MMUL(A', B', N) \tag{4.9}$$

4. Bring result back from Montgomery domain

$$P = MMUL(P', 1, N) \tag{4.10}$$

```
S = 0
for i=0 to n-1
   if S + A(i) * B is even
      then S = ( S + A(i) * B ) / 2
      else  S = ( S + A(i) * B + N ) / 2
   end if
end for
if S >= N then S = S – N
```

Figure 4.1. R2MM Pseudo-code, Multiplicand (A), Multiplier (B), Modulus (N), Result (S), $i^{th}$ bit of A ($A_i$)

We chose the Radix-2 Montgomery Multiplication (R2MM) algorithm [79] for the implementation. R2MM is suitable for simple hardware implementation because it is composed of simple operations that suit hardware better. R2MM can be implemented by a simple adder and shifter.

As it can be seen in Figure 4.1, there are multiple conditional operations in the algorithm. Conditional operations cause variations in run time of the algorithm which

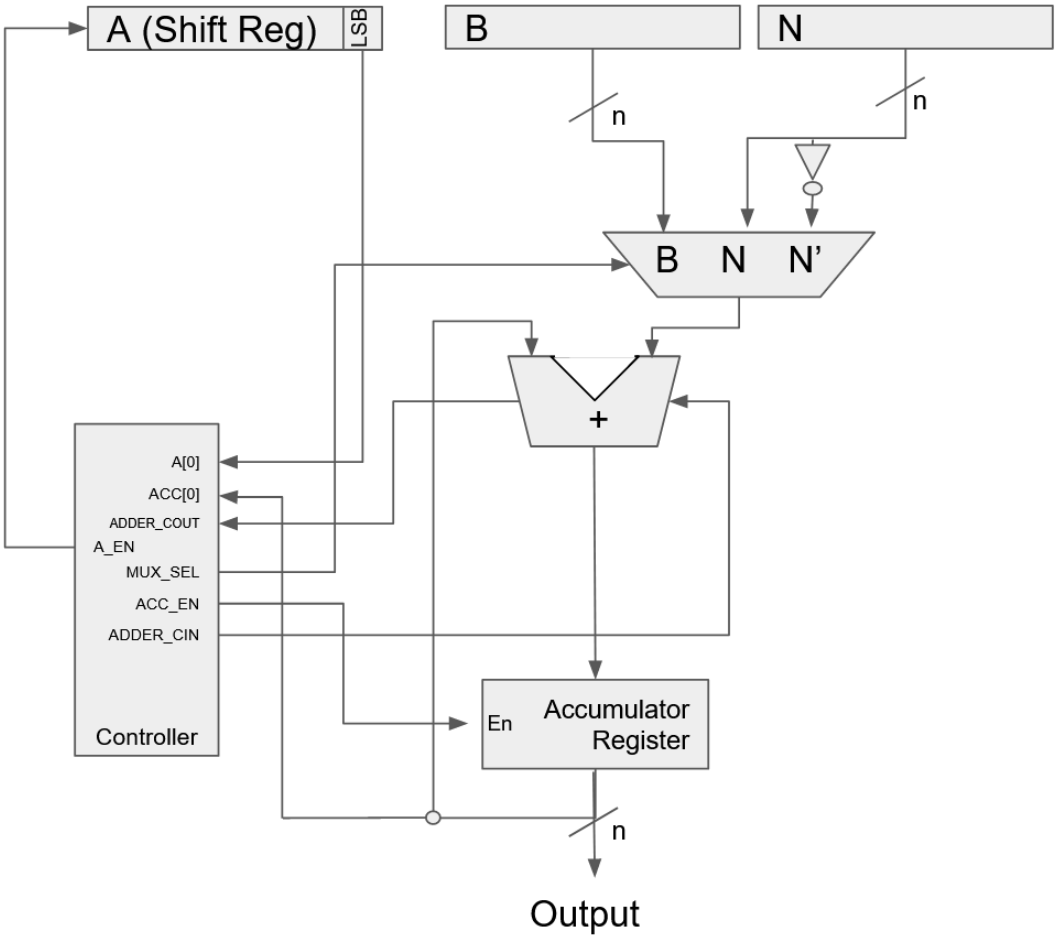



Figure 4.2. Basic R2MM Hardware

is a prime candidate to be exploited by SPA attacks. If this algorithm were to be implemented in software, additional measures should be taken [80]. But the hardware implementation has constant execution time as if-statements are only enable signals of memory elements.

Our hardware consists of an adder and an accumulator for storing the interim results, a controller for execution of the algorithm with conditional operations in Figure 4.1 implemented as enable signals of the accumulator register. First cycle of each loop iteration is used to calculate $Acc + A_{\mathrm{i}}B$. Since $\mathrm{A_i}$ is a single bit value, it serves as an enable signal of accumulator register, only if $\mathrm{A_i}$ is 1 the accumulator is updated with $Acc + B$ value. In the second cycle, modulus $N$ is added to value in the accumulator. Result of the addition is only saved to the accumulator register if existing value in accumulator ($Acc + A_{\mathrm{i}}B$) is odd. Operand A is stored in a shift register and shifted to

```
if CURRENT_STATE is LOOP_1
        MUX_SEL = B
        ADDER_CIN = 0
        A_EN = 0
        ACC_EN = A[0]
        NEXT_STATE = LOOP_2
else if CURRENT_STATE is LOOP_2
        LOOP_ITERATION += 1
        MUX_SEL = N
        ADDER_CIN = 0
        ACC_EN = ACC[0]
        A_EN = 1
        if LOOP_ITERATION equals n
                NEXT_STATE = LOOP_EXIT
        else
                NEXT_STATE = LOOP_1
else if CURRENT_STATE is LOOP_EXIT
        A_EN = 0
        MUX_SEL = B'
        ADDER_CIN = 1
        ACC_EN = ADDER_COUT
```

Figure 4.3. MMUL Operation Pseudocode

right by one bit in each iteration of the loop. Since operand $A$ is consumed one bit at a time, whole operand is not required to be in MMUL internal registers. Each word of operand $A$ is fetched from memory when previous word is completely shifted out of the internal register. For a RV32 machine, a new word of operand A is fetched after every 32 iterations of the loop. After the loop is finished, in one last cycle, modulus is subtracted from the accumulator to bring result back to $[0, N)$ range. If the result is still positive after the subtraction, accumulator is enabled and the final result is saved to the accumulator register to be written back to memory. Figure 4.3 summarizes flow of execution. Overall, one MMUL operation takes $2n + 1$ clock cycles for calculations, $3 * WORDS$ memory load operations for operand fetch and $WORDS$ memory write operations for writing back the result where $WORDS = ceil(n/32)$ for RV32 and $WORDS = ceil(n/64)$ for RV64. Operands up to $n$ bits can be processed with $n$ being limited by the MMUL hardware configuration.

# 5. MONTGOMERY MULTIPLICATION INSTRUCTION FOR RISC-V ISA

RISC-V has different instruction formats already defined, some of them can be seen in Figure 5.1. More than one could be used for our application with two main tradeoffs. Regardless of the instruction encoding, we decided MMUL instruction to work on memory addresses unlike any instruction in RISC-V specification. When it comes to multiprecision operations, defining a unified interface on memory addresses is more performant. The key point that has to be made clear is the layout of operands

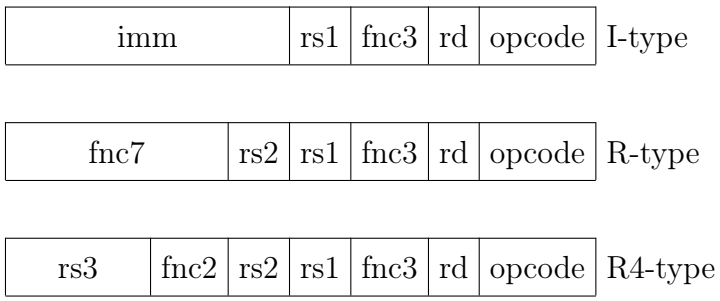



Figure 5.1. Candidate RISC-V instruction formats

in memory. Constraining how operands should be arranged may result in lower performance as it may require application code to rearrange operands in memory to get it in the way it is required. These constraints put on operands will result in a more flexible instruction with more fields to encode additional information that may be used to pack more functionality into the instruction. MMUL requires 3 operands and a space to store the result. This means 3 memory addresses for input and a single memory address for output. Length of the operands must be encoded in the instruction for flexibility. Instruction should be able to provide this information to the accelerating hardware.

If application can guarantee that all operands will be in a certain offset from a base address in memory as shown in Figure 5.2, a single memory address stored in rs1 is enough for input operands. Thus I-type instruction format can be used. It leaves 15 bits of space in the instruction to be used for encoding other information. Length can

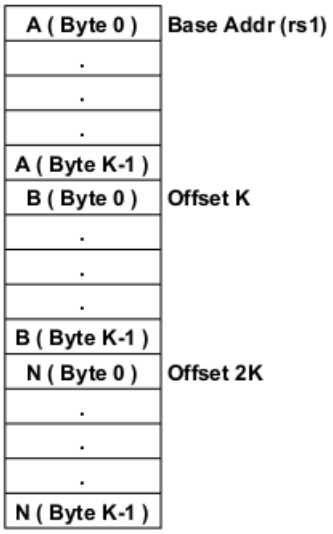



Figure 5.2. Memory layout for I-type MMUL

be encoded in bits in this format, giving us a maximum of 32768 bit operands.

Likewise, if multiplicand and multiplier are guaranteed to be always in fixed positions relative to each other but modulus may be in random addresses as shown in Figure 5.3; R-type format can be used. Two source registers, rs1 and rs2, would be used as base addresses. This leaves 10 bits of space which enables, if length is encoded in bits, maximum of 1024 bit operands.

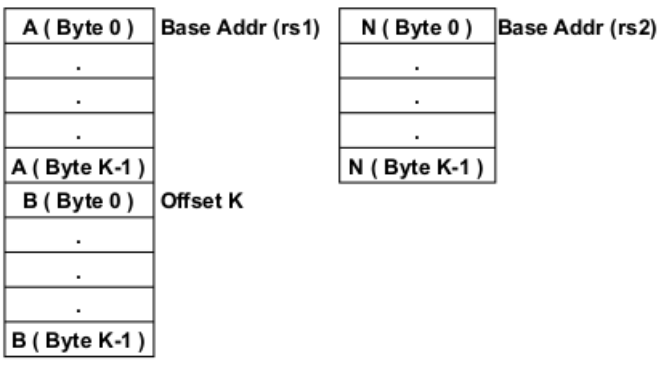



Figure 5.3. Memory layout for R-type MMUL

Lastly, for the best performance in all cases, if R4-type format is used; all operands can be in their independent addresses stored in rs1, rs2 and rs3 as shown in Figure 5.4. This format leaves only 5 bits which is not enough for length to be encoded in bits. Encoding operand length in total architecture words, ie. $WORDS$, is another

option which makes 1024 bit length operands for RV32 and 2048 bit operands for RV64 possible.

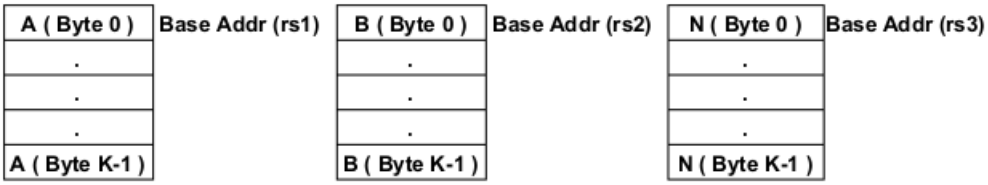



Figure 5.4. Memory layout for R4-type MMUL

In this work, we decided to use R4-type instruction format because it imposes no memory layout restrictions. Using GCC directive *.insn* [81] in this decision process sped up the development. Meanings of our fields are;

- rd - Register containing memory address for result
- rs1 - Register containing memory address for multiplicand
- rs2 - Register containing memory address for multiplier
- rs3 - Register containing memory address for modulus
- fnc3, fnc2 - Combined, they encode length of our operands in architecture words as follows

$$WORDS = (\text{fnc2} * 8) + \text{fnc3}$$

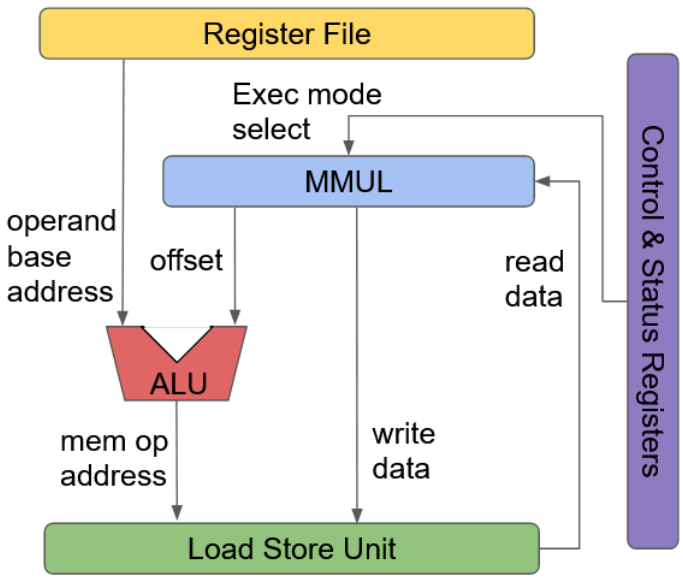



Figure 5.5. Integration of MMUL in datapath

As it can be seen in Figure 5.5, MMUL is coupled with the datapath of the processor. Addresses of operands are read directly from their respective registers of the Register File and fed to the ALU in the datapath. Memory address to be worked on is calculated in the ALU by adding the offset value supplied by the MMUL to the base address read from register file. LSU is triggered by MMUL module to load or store from the calculated address. Operands are loaded at the start of the execution and stored in MMUL module during the entire operation. All execution is controlled by MMUL itself.

## 5.1. Partial Execution Mode

As mentioned in Section 4, one MMUL operation takes $2n + 1$ clock cycles for calculations, $3 * WORDS$ memory load operations for operand fetch and $WORDS$ memory write operations for writing back the result where $WORDS = ceil(n/32)$ for RV32 and $WORDS = ceil(n/64)$ for RV64. As the instructions are atomic, for duration of this operation processor will be unresponsive to any event that may happen. For some applications this may be problematic because of the real time constraints they have. To remedy this we can move the loop in our algorithm from hardware to software, allowing our processor to service interrupts in between loop iterations.

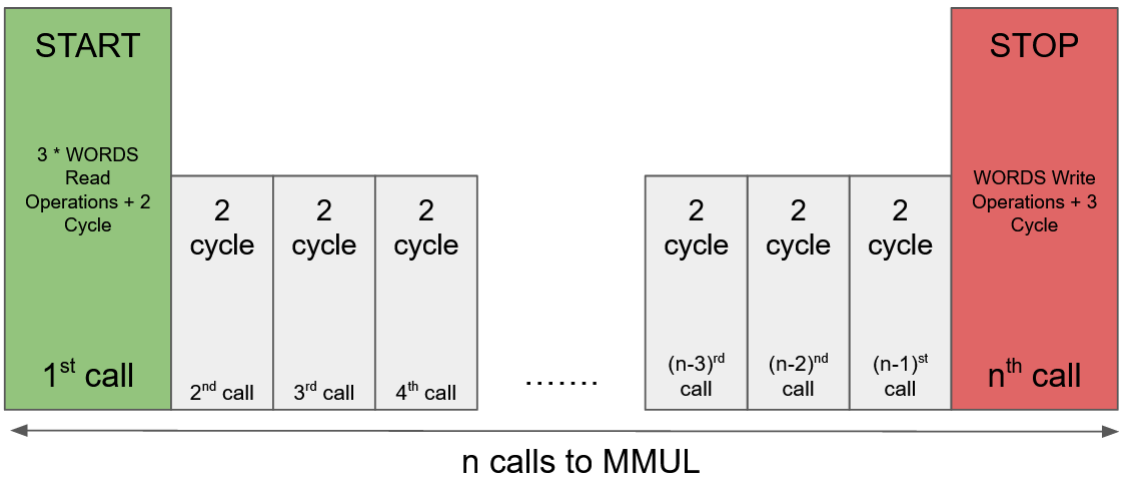



Figure 5.6. MMUL partial execution time diagram

To achieve such behaviour, which we call partial execution, our implementation has a special Control and Status Register (CSR) as shown in Figure 5.7. If partial

execution is enabled by a write with *csrwr* instruction to this register, which is directly connected to "Execution Mode Select" signal in Figure 5.5, current MMUL instruction is retired after every iteration of the loop in Figure 4.1. Application code has to execute another MMUL instruction for each bit of operands, ie. n calls to the MMUL for an n-bit x n-bit Montgomery multiplication operation, as shown in Figure 5.6. First call to the MMUL does the memory load operations, while last call writes back the result. In this case, maximum latency of MMUL instruction drops to either $3*WORDS$ memory load operations + 2 cycles or $WORDS$ memory write operations + 3 cycles depending on the memory operation latencies. Performance penalty of this, which will be later presented, is minimal when used with loop unrolling.

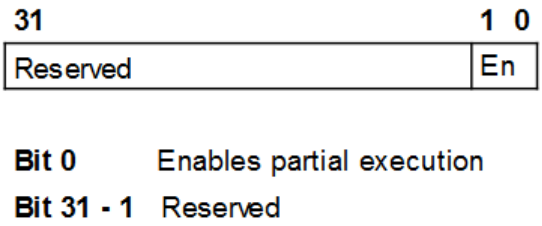



Figure 5.7. MMUL control register

# 6. ANALYSIS

## 6.1. Base Architecture

To set a baseline for our work, we designed an in-order two-stage RV32EC core. Core was designed to have minimal area footprint while maintaining comparable level of performance. RISC-V is adopted as the ISA as it allows custom instructions and have a rich set of encoding formats already defined.

Base user ISA of RISC-V is called RV32I and it includes basic arithmetic, control transfer and memory operation instructions. It provides bare minimum for modern languages to target as an architecture. A subset of RV32I is defined for embedded devices as well, RV32E. While the supported instructions stay the same, it allows 16 word register files (32 on RV32I) and makes some of the mandatory control and status registers optional to save hardware resources. RISC-V also has a compressed instruction set extension which defines 16-bit encodings of frequently used RV32I instructions to bring the code size down as well as increase performance by allowing multiple instructions to be fetched in a single memory operation.

Figure 6.1 shows the base architecture of our design. Fetch stage has two main components, realign buffer and decompressor. Realign buffer acts as a buffer for fetched instructions and is able to serve 16 or 32 bit instructions from half word aligned addresses. Decompressor unpacks 16-bit instructions to their 32-bit equivalents. Only 32-bit instructions are fed to the next stage.

Table 6.1. CPU Benchmark

| | **Our Work** | **microriscy [82]** | **Improvement %** |
|---|---|---|---|
| Coremark | 0.905 | 0.878 | 3% |
| Dhrystone | 1805 | 1644 | 10% |

Second stage handles the decoding and execution of the instruction stream. Apart from jumps, branches and memory operations; all instructions have single cycle execu-

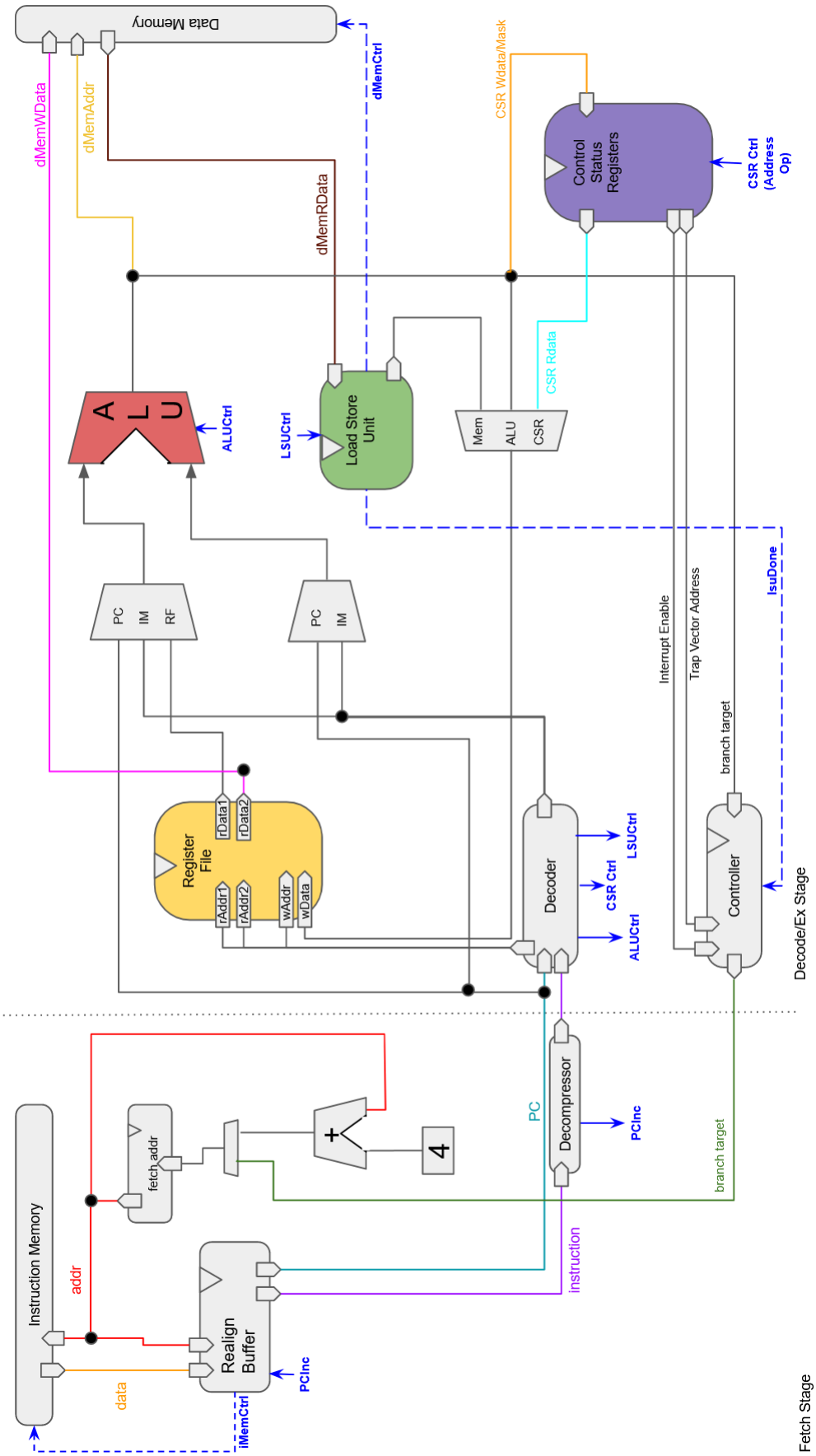



Figure 6.1. Base Architecture

tion time. Controller is capable of executing multi cycle instructions by delaying the retirement of current instruction. In the case of a jump or branch, fetch stage is cleared by the controller and execution continues from the target address. No prediction is done for branches.

To set a baseline, Coremark and Dhrystone are run both on microriscy [82] and our core using the same memory modules. Results can be seen in Table 6.1. Even though our core is slightly faster than microriscy, they can be considered equal in terms of performance.

How MMUL module is integrated into the base architecture can be seen in Figure 5.5. Related modules are color coded so they can be easily identified.

## 6.2. Benchmarks

Resulting design is benchmarked with multiple ECC curves. Software implementations of FourQ (128-bit) [57], NIST P-256 (256-bit) [12], Curve25519 (256-bit) [77] and ARIS (an authentication scheme based on FourQ) [58] are run on our processor and microriscy [82] core from PULP for the base values. Later, modular multiplication and squaring implementations are replaced with a sequence of MMUL instructions and run on our modified core. No modifications are made to any other part of the code.

Example blocks of code from modified field multiplication functions for FourQ can be seen in Figure 6.2 and 6.3. They calculate $c = a * b \; mod \; n$. Since our encoding enables operands to be in unrelated memory addresses, arguments of the original function can be used without any rearrangement of data. Value $r$ is precalculated and stored. In function $fpmul1271$ line 12 and 13, operands $a$ and $b$ are brought to the Montgomery domain. Results are stored in intermediate variables $\_a$ and $\_b$. At line 14, multiplication in Montgomery domain is done. Lastly, result is brought back to [0, N) range in line 15. As line 15 suggests, any combination of addresses given to the instruction can be the same.

```
typedef uint32_t felm_t[4];
uint32_t r[4] = {4, 0, 0, 0};
uint32_t one[4] = {1, 0, 0, 0};
uint32_t n[4] = {0xffffffff, 0xffffffff,
                 0xffffffff, 0x7fffffff};

void MMUL(uint8_t WORDS,
          uint32_t* A, uint32_t* B,
          uint32_t* N, uint32_t* C ) {
uint8_t func2 = (WORDS >> 3) & 0x3;
uint8_t func3 = (WORDS) & 0x7;
asm volatile (
    ".insn r CUSTOM_0," #func3 "," #func2
    ", %[C], %[A], %[B], %[N]\n" :
    : [A] "r" (A), [B] "r" (B),
      [N] "r" (N), [C] "r" (C) );
}

void fpmul1271(felm_t a, felm_t b, felm_t c)
{ // With atomic execution of MMUL
  volatile uint32_t _a[4], _b[4];

  MMUL(4, a , r , n, _a);
  MMUL(4, b , r , n, _b);
  MMUL(4, _a, _b , n, c);
  MMUL(4, c , one, n, c);
}
```

Figure 6.2. Modular multiplication function for FourQ implemented with MMUL custom instruction with atomic execution

```
#define MMUL_CSR 0x800 // Address of MMUL CSR
void fpmul1271_pe(felm_t a, felm_t b, felm_t c)
{ // With partial execution
  volatile uint32_t _a[4], _b[4];
  write_csr(MMUL_CSR, 1);

  #pragma GCC unroll 128
  for (int i = 0; i < 128; i++) {
    MMUL(4, a , r , n, _a);
  }
  #pragma GCC unroll 128
  for (int i = 0; i < 128; i++) {
    MMUL(4, b , r , n, _b);
  }
  #pragma GCC unroll 128
  for (int i = 0; i < 128; i++) {
    MMUL(4, _a, _b , n, c);
  }
  #pragma GCC unroll 128
  for (int i = 0; i < 128; i++) {
    MMUL(4, c , one, n, c);
  }
}
```

Figure 6.3. Modular multiplication function with Partial Execution enabled for FourQ

Function $fpmul1271\_pe$ implements the same functionality as the $fpmul1271$, but utilizes partial execution feature of the MMUL instruction. Each assembly instruction in $fpmul1271$ is called in an unrolled loop allowing processor to handle asynchronous events in between loop iterations.

In Table 6.2, 6.3 and 6.4, runtime of ECC benchmarks in clock cycles can be seen. Full software benchmarks are run on microriscy [82] and our base architecture (BA) as the control group. Runs that modular multiplication operation is implemented with our custom instruction are labeled Custom Instruction with Atomic Execution (CI-AE) and Custom Instruction with Partial Execution (CI-PE). There is a significant speed up in all curve operations. This speed up contributes to lowering total energy consumption. This setup uses a memory module with single cycle read latency. Longer latency memories like EEPROM, FLASH are widely used as instruction memories. If a longer latency instruction memory was used in benchmarks, results would be even more in favour of our implementation. MMUL instruction takes multiple cycles thus allowing a new instruction to be fetched before it finishes, therefore CPU is less likely to be stalled waiting for new instructions.

Performance penalty of partial execution is negligible when paired with loop unrolling. Depending on the compiler output, if not interrupted, a Montgomery multiplication can be executed in the same amount of time as atomic execution.

For power consumption analysis, FPGA tools are used. Design is synthesized on a Xilinx XC7Z020-1 FPGA and activity data is gather with Xilinx's development environment, Vivado. Activity data is then used to increase dynamic power estimation accuracy.

Power consumption of different configurations can be seen in Table 6.5. While static power consumption shows only small changes within margin of error, dynamic power goes down significantly. This can be explained with the power consumption per design block during fully software and with custom instruction runs of the benchmark, which can be seen in Table 6.6. When executing solely standard instructions, as in BA

Table 6.2. Runtime of benchmarks in clock cycles, Base RV32EC Architecture (BA), Custom Instruction with Atomic Execution (CI-AE), Custom Instruction with Partial Execution (CI-PE)

| | Base RV32EC ISA | | Extended with MMUL custom instruction | | | |
|---|---|---|---|---|---|---|
| | microriscy [82] | BA | CI-AE | Speedup | CI-PE | Speedup |
| FourQ Benchmark | | | | | | |
| GF($p^2$) squaring | 19725 | 19365 | 2535 | 7.64 | 2553 | 7.59 |
| GF($p^2$) multiplication | 30066 | 29522 | 3861 | 7.65 | 3874 | 7.62 |
| GF($p^2$) inversion | 1387750 | 1365080 | 122131 | 11.18 | 148050 | 9.22 |
| SchnorrQ's key generation | 13655212 | 13386140 | 2022293 | 6.62 | 2052371 | 6.52 |
| SchnorrQ's signing | 14550101 | 14253297 | 2913228 | 4.89 | 2943336 | 4.84 |
| SchnorrQ's verification | 41340579 | 40592170 | 5717480 | 7.10 | 5813804 | 6.98 |
| Keypair generation (compressed) | 13604506 | 13347307 | 1996729 | 6.68 | 2026818 | 6.59 |
| Secret agreement (compressed) | 37232813 | 36622106 | 4998062 | 7.33 | 5092133 | 7.19 |
| Keypair generation (uncompressed) | 13586951 | 13346151 | 1996342 | 6.69 | 2026462 | 6.59 |
| Secret agreement (uncompressed) | 34384022 | 33717142 | 4710501 | 7.16 | 4753021 | 7.09 |

Table 6.3. Runtime of benchmarks in clock cycles, Base RV32EC Architecture (BA), Custom Instruction with Atomic Execution (CI-AE), Custom Instruction with Partial Execution (CI-PE)

| | Base RV32EC ISA | | Extended with MMUL custom instruction | | | |
|---|---|---|---|---|---|---|
| | microriscy [82] | BA | CI-AE | Speedup | CI-PE | Speedup |
| FourQ Benchmark | | | | | | |
| ECC Point doubling | 1691283 | 1659226 | 228432 | 7.26 | 229772 | 7.22 |
| ECC Point addition | 2405257 | 2359975 | 321217 | 7.35 | 322097 | 7.33 |
| Psi mapping | 6998077 | 6863310 | 956913 | 7.17 | 961162 | 7.14 |
| Phi mapping | 10423134 | 10216877 | 1469423 | 6.95 | 1475313 | 6.93 |
| Precomputation | 4407761 | 4321323 | 613983 | 7.04 | 616172 | 7.01 |
| Scalar multiplication (wo clearing cofactor) | 32452063 | 31835079 | 4466963 | 7.13 | 4508183 | 7.06 |
| Scalar multiplication (with clearing cofactor) | 34383841 | 33730968 | 4720296 | 7.15 | 4762786 | 7.08 |
| Fixed-base scalar mul | 13614439 | 13352275 | 1996163 | 6.69 | 2026367 | 6.59 |
| Double scalar mul | 38583566 | 37843606 | 5454204 | 6.94 | 5499222 | 6.88 |

Table 6.4. Runtime of benchmarks in clock cycles, Base RV32EC Architecture (BA), Custom Instruction with Atomic Execution (CI-AE), Custom Instruction with Partial Execution (CI-PE)

| | Base RV32EC ISA | | Extended with MMUL custom instruction | | | |
|---|---|---|---|---|---|---|
| | microriscy [82] | BA | CI-AE | Speedup | CI-PE | Speedup |
| ARIS Benchmark | | | | | | |
| Sign | 9615404 | 9293452 | 8128129 | 1.14 | 8155299 | 1.14 |
| Verify | 16048540 | 15652870 | 3069309 | 5.10 | 3127147 | 5.01 |
| NIST P-256 Benchmark | | | | | | |
| Key Generation | 138564751 | 136349759 | 10763769 | 12.67 | 10859727 | 12.56 |
| Sign | 139670301 | 137359076 | 11802994 | 11.64 | 11898438 | 11.54 |
| Verify | 158162663 | 155616115 | 12463289 | 12.49 | 12574720 | 12.38 |
| Curve25519 Benchmark | | | | | | |
| Key Generation | 155787773 | 153887987 | 11153517 | 13.80 | 11312294 | 13.60 |
| Sign | 155855237 | 153940299 | 11641586 | 13.22 | 11800242 | 13.05 |
| Verify | 311303424 | 307498179 | 22437824 | 13.70 | 22756803 | 13.51 |

Table 6.5. Average Power Consumption (W) During A Modular Multiplication

| | **Static** | **Dynamic** | **Total** |
|---|---|---|---|
| BA | 0.107 | 0.154 | 0.261 |
| CI-AE | 0.105 | 0.064 | 0.170 |
| CI-PE | 0.106 | 0.120 | 0.226 |

Table 6.6. Average Dynamic Power Consumption Per Module (W) During A Modular Multiplication

| | **BA** | **CI-AE** | **CI-PE** |
|---|---|---|---|
| Fetch Stage | 0.058 | 0.002 | 0.026 |
| Decoder | 0.014 | 0.001 | 0.006 |
| ALU | 0.031 | 0.001 | 0.008 |
| Register File | 0.012 | 0.002 | 0.003 |
| MMUL | 0 | 0.054 | 0.053 |

column of Table 6.6, every module of the CPU works synchronously. When MMUL instruction is in progress, rest of the CPU is idle. Biggest gain comes from the fetch stage because only four instruction fetches are needed per modular multiplication with our custom instruction and it is the biggest module in the design.

Table 6.7. Normalized Energy Consumption (Power x Clock Cycles)

| | | **BA** | **CI-AE** | **CI-PE** |
|---|---|---|---|---|
| FourQ | KeyGen | 1 | 0.10 | 0.13 |
| | Sign | 1 | 0.13 | 0.18 |
| | Verify | 1 | 0.09 | 0.12 |
| P256 | KeyGen | 1 | 0.05 | 0.07 |
| | Sign | 1 | 0.06 | 0.08 |
| | Verify | 1 | 0.05 | 0.07 |
| C25519 | KeyGen | 1 | 0.05 | 0.07 |
| | Sign | 1 | 0.05 | 0.07 |
| | Verify | 1 | 0.05 | 0.06 |

Both average power consumption and execution time go down in our implementation. Naturally, product of these two metrics follow this trend as well. Normalized

energy consumption values can be seen in Table 6.7. For FourQ, a 128-bit curve, roughly 90% of energy is saved while P-256/C25519, 256-bit curves, see savings up to 95%. As the prime that curves use gets bigger, performance increases and this results in higher energy savings. R2MM scales better to larger operands with it's O(n) time complexity [79] [83].

Frequency and area results can be seen in Table 6.8 and 6.9. Although MMUL itself is fairly small, it adds 33% area overhead to our base architecture and operating frequency goes down by 9% in FPGA synthesis. Using the TSMC OSU 0.18um technology, ASIC synthesis shows 49% area overhead and 8% decrease in operating frequency. Depending on the requirements of the application, a different implementation of Montgomery multiplication may be used for the required balance between performance gain and area overhead.

Table 6.8. Post P&R Frequency and Area (Xilinx XC7Z020-1 FPGA)

| | **Clock Freq. (MHz)** | **Slices** |
|---|---|---|
| Base Architecture (BA) | 89.03 | 487 |
| BA w/ 128bit MMUL | 81.15 | 649 |

Table 6.9. Post Synthesis Frequency and Area (OSU018)

| | **Clock Freq. (MHz)** | **Flip Flops** | **Gates** |
|---|---|---|---|
| Base Architecture (BA) | 148.46 | 872 | 8106 |
| BA w/ 128bit MMUL | 136.37 | 1305 | 12105 |

It is debated [84] that ECC is too complex to be used on IoT devices, yet even new lightweight algorithms introduce similar overheads when accelerated in hardware. In Table 6.10, a comparison of additional overhead of different custom instructions can be seen. Tehrani et. al. [85] accelerate Lightweight Block Ciphers on a RV32I platform. Their work is divided in to four categories. Category I is fully software implementation similar to our base architecture while other categories progressively add more complex acceleration capabilities with more instructions. Category II implements the parallel Sbox instruction, Category III implements permutation and the bit-level matrix multiplication instructions and Category IV implements the nibble-level parallel matrix multiplication instruction, no instruction latency is reported.

Table 6.10. Overhead Comparison

| | **Technology** | **Base RISC-V ISA** | **Area Overhead %** |
|---|---|---|---|
| Our work | FPGA (Xilinx) | RV32EC | 33% |
| Our work | ASIC (0.18um) | RV32EC | 49% |
| [85] Category II | ASIC (45nm) | RV32I | 51% |
| [85] Category III | ASIC (45nm) | RV32I | 55% |
| [85] Category IV | ASIC (45nm) | RV32I | 67% |

## 6.3. Attack Analysis

Security of field operation implementations of a system are crucial for overall security. There are existing attacks that target platforms with vulnerable field operations. ECDH encryption with Curve25519 implementation of Libgcrypt is shown to be vulnerable by Genkin et al. [86]. They exploit the field operations that are not implemented in constant time fashion. With Flush+Reload [87] attack, secret information can be extracted easily by examining the cache response times. Another paper by Alam et al. [88] showcases a vulnerability in a recent version of OpenSSL RSA implementation with field operations as its root cause. Field operations that have varying execution time or cache access patterns are exploited by observing the changes in electromagnetic emissions from the target.

Our Montgomery multiplication implementation is constant time and can be used to implement other computationally expensive field operations such as exponentiation and squaring. But for a secure ECC implementation, operations implemented in software should be constant time as well. Any side effects that these field operations implemented in software may have on the system state, like cache contents, should be carefully studied to ensure that no secret information is leaked. In this chapter we will analyze if incorporating our custom instruction to an already secure ECC implementation would compromise the security of the system. We will explain how our work will not expose new side channels to be exploited.

Fan et. al. [89], study the known side channel attacks and countermeasures

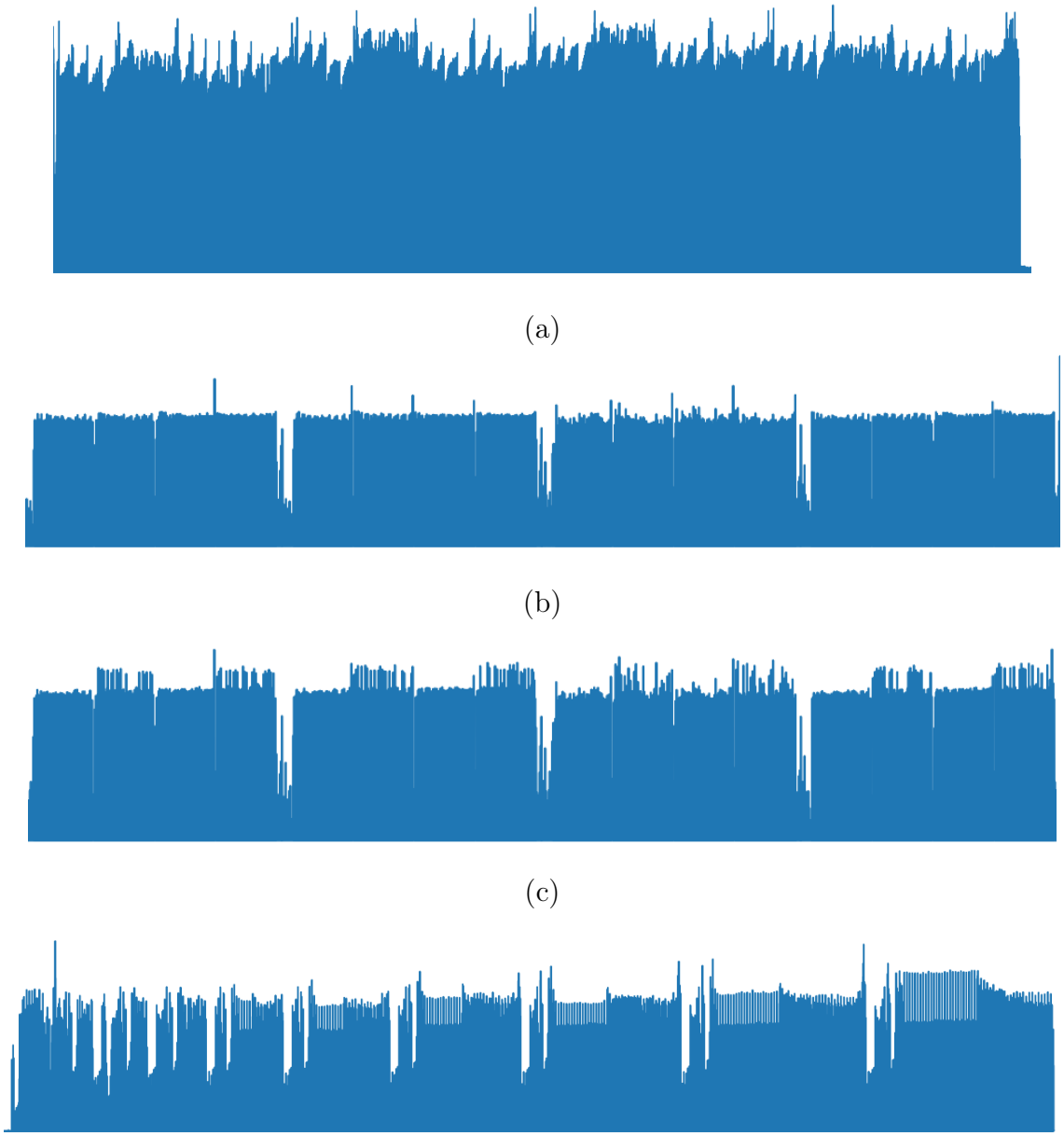
against them. They divide possible attacks mainly into two categories, namely, passive and active attacks. While passive attacks only observe the target to gather secret information, active attacks inject faults at either hardware or algorithmic steps to leak secret information.

### 6.3.1. Passive Attacks

Cache based attacks [90], exploit the cache contents after the target code runs. For example, if a lookup table is accessed according to the secret information, by testing the access times to various indexes of the lookup table secret information can be detected [91]. Our custom instruction fetches the operands fully into local registers and has a consistent cache access signature. An implementation that is secure against cache based attacks can adopt our custom instruction based modular multiplication with no security consequences.

Timing attacks analyze the execution time of a given code to infer the execution path that it follows [92]. When execution path depends on the secret information that is passed to this piece of code, it may be leaked. Our MMUL implementation is constant time and existing secure cryptography algorithm implementations can benefit from the performance increase without additional precautions.

Simple power analysis attacks require execution path to contain branches that are controlled by the secret information and lead to paths with different instantaneous power consumption [93]. To analyze robustness of our implementation against a power analysis attack, we used switching information from synthesized design to estimate instantaneous power consumption. In Figure 6.4 switching activity inside the core for a single modular multiplication can be seen. As they only represent activity inside the core, instantaneous power consumption of memory or peripheral are not included in the graphs. In (a) fully software implementation is profiled. Graph (b) shows the switching activity while executing a modular multiplication operation using the custom instruction in atomic execution mode. Graphs (c) and (d) show the profile of custom instruction with partial execution. There are interrupts being served in be-



(a)

(b)

(c)

(d)

Figure 6.4. Activity Graphs (a) Software, (b) Custom Instruction with Atomic Execution, (c) Custom Instruction with Partial Execution, (d) Partial Execution with Interrupts In Between

tween calls to the MMUL in Graph (d). Latter three graphs show four high activity regions separated by lower activity periods. Each of the high activity period corresponds to a Montgomery multiplication. Four Montgomery multiplications are needed to perform a modular multiplication as given in Equation 4.7 to Equation 4.10. The period where processor is doing the modular multiplication may be detected but it has

little correlation with the secret information that is used in the calculations.

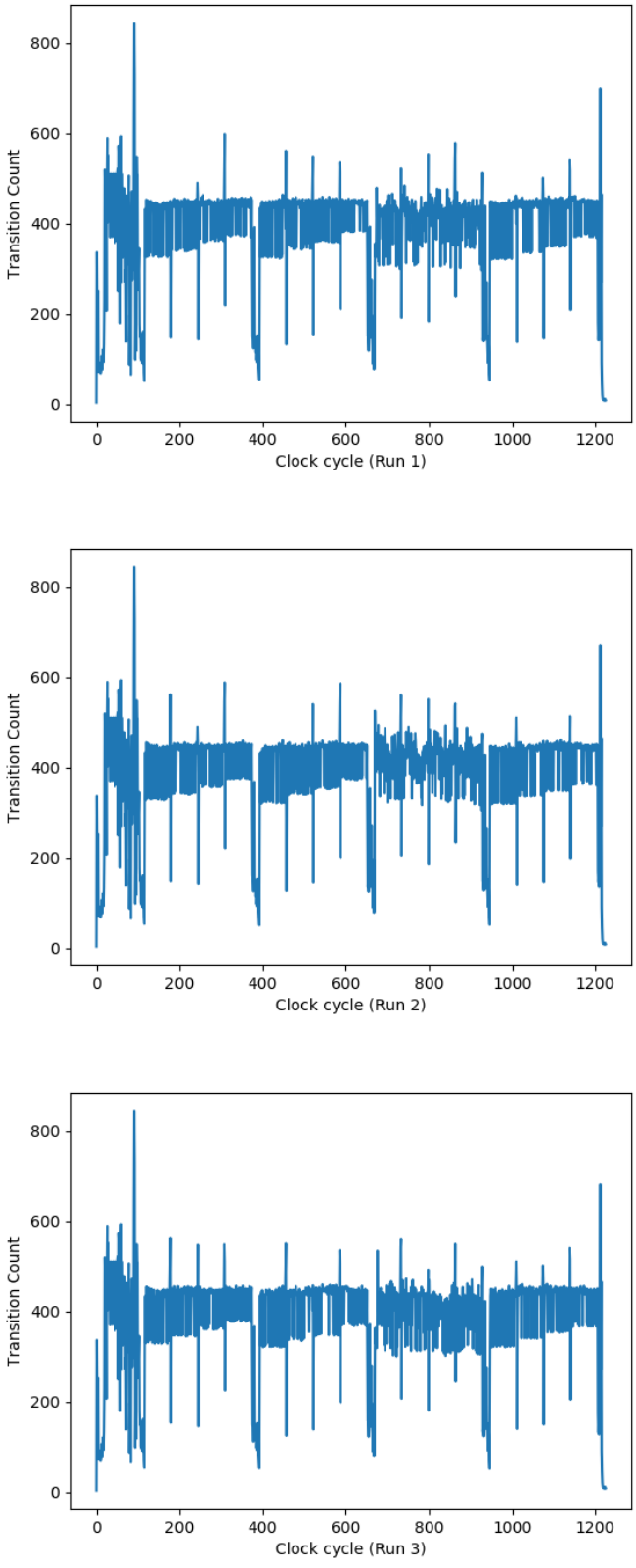



Figure 6.5. Activity Graphs for Three MMUL Runs over GF($2^{127}$ -1)

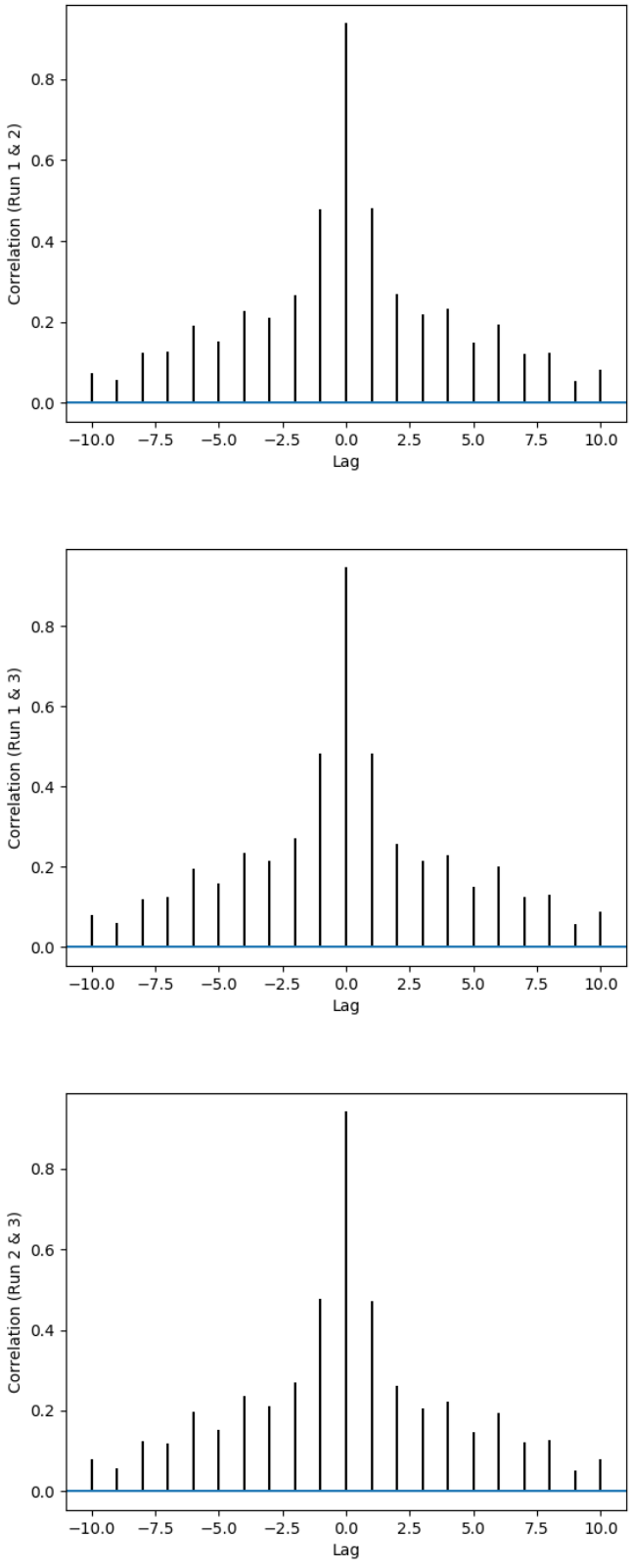



Figure 6.6. Crosscorrelation Graphs for Three MMUL Runs

Figure 6.5 shows the switching activity of our core executing three MMUL instructions with three arbitrary 128-bit input sets over GF($2^{127}$ -1). Figure 6.6 shows the correlation between graphs in Figure 6.5. Correlation is calculated as the cross-correlation between the number of signals that change state each clock cycle in each run. Crosscorrelation is calculated after DC components of switching activities are eliminated. As it can be seen, there is a high correlation between the switching activities between different input sets, thus inputs to the calculation has minimal effect on instantaneous power consumption. Input sets for the Figure 6.5 are;

- 0x4264c4c035d6ec7afbd55e860197ff68 , 0xe36d976eef32928cd1a32bef94160a9b
- 0x06333b57432d1a222af67d6cd46702a5 , 0xa22646c4413685fb8f50025b882f11a7
- 0x134488cce1c4cd94eb6b1433cfb85da5 , 0x17e4f5134a1f17e97112f460595ad253

Attacks like Differential Power Analysis, Refined Power Analysis and Comparative SCA studied by Fan et. al. [89] exploit this small correlation between inputs and power traces of an operation. Protection against these attacks require countermeasures at the higher levels of ECC hierarchy. The most straight forward countermeasure is randomization of operands of higher level operations like scalar multiplication. An implementation that has taken needed precautions against such attacks will not be negatively affected from using our custom instruction for field multiplication.

### 6.3.2. Active Attacks

Fan et. al. [89] divide active attacks into three main categories. One of them is safe-error based analysis attacks. Safe-error attacks try to exploit the fact that some faults will not be changing the computation result. The second category is weak-curve based analysis attacks. Weak-curve attacks switch the point in scalar multiplication with a point in a weak curve. Finally the last category is differential fault analysis attacks that use the changes between faulty and valid results to extract the secret scalar. All of these attacks try to exploit weaknesses of higher levels in ECC hierarchy and therefore they are not in the scope of our work.

Attacks that tamper with the physical hardware to leak secrets are also studied in ECC domain. Fouque et. al [94] show that a high power laser can be used to manipulate inner storage elements of a processing hardware. Our implementation does not take additional precautions to protect itself against such attacks. Depending on the underlying physical implementation, such attacks can be used against our design as well. Additional corrective hardware with error correcting codes can be utilized to detect and correct any fault injected by an outside element.

# 7. CONCLUSION

In this thesis acceleration of ECC for constrained platforms is studied. Literature shows that ECC is a good target to be accelerated since it involves computationally expensive operations. Various levels that ECC can be accelerated at is also explored. Accelerating at protocol or point operation level are proved to be costly in terms of hardware resources. Efforts on accelerating at those levels are highly specialized. Specialization results in solutions with low adaptability which limits their use in different scenarios. Field operations are the simplest operations that build the basis of the ECC. For resource constrained systems, they are the ideal candidate to be accelerated with their smaller are footprint and high adaptability to different applications.

To select the field operations to be accelerated, we profiled the runtime of various ECC implementations and found out that vast majority of execution time, up to %95 for some curves, gets spent on modular multiplication. Modular multiplication can be implemented with different algorithms. Montgomery multiplication is one of the efficient algorithms that implements modular multiplication. Hardware implementation friendly Radix-2 Montgomery Multiplication was our algorithm of choice for our modular multiplication implementation. A simple enough operation such as modular multiplication can easily be mapped to a custom instruction. RISC-V specification allow different platforms to have their custom instructions. But unlike a separate memory-mapped accelerator, custom instructions block the processor for the duration of the operation. To remedy this drawback, we introduced a partial execution mode that can be controlled with a Control and Status Register (CSR). Partial execution mode allows breaking down the Montgomery multiplication operation into smaller iterations so that system can service interrupts or do a context switch in between.

For our custom instruction, we analyzed different predefined RISC-V instruction encodings. Among the available encodings, main trade off is between performance and maximum operand size. The more fields are reserved for operand addresses, the less space is left to encode operand length. We chose R4-type encoding for the best

performance and designed a Montgomery Multiplication that would be integrated into the datapath of our processor.

For the base of our work, a two-stage in-order RV32EC core is designed and implemented in Verilog. All the benchmarks are run on that base platform and on another similar core from literature. These results are used as our base case. Various ECC libraries using different curves are then benchmarked with our custom instruction in both atomic and partial execution modes. Up to 13x speed up is achieved compared to fully software implementation. Results show that curves that use bigger primes benefit more from our custom instruction.

Dynamic power consumption is measured through activity analysis. Fully software implementations of modular multiplication activates large portion of the processor, therefore results in high dynamic power consumption. While running our custom instruction, switching activity drops significantly and results in lower dynamic power consumption. Since our instruction is a multi-cycle instruction, fetch stage is the biggest contributor in this drop in the dynamic power consumption. With speed up and the drop in average power consumption, up to 95% energy is saved.

The design, with a 128-bit MMUL module, is then realized on both FPGA and ASIC. Our FPGA platform of choice was Xilinx XC7Z020-1. Synthesis and P&R on FPGA resulted in 649 slices being used and maximum clock frequency of 81MHz. An ASIC synthesis is also done with TSMC OSU018, which is an open source 0.18µm technology, as the target technology. Resulting ASIC design uses 12015 gates and 1305 flip flops, and has a maximum clock frequency of 136 MHz. Compared to base the architecture, average area overhead is 41% and decrease in maximum clock frequency is 9%. It is debated that ECC is too complex to be used on IoT devices, yet even new lightweight algorithms introduce similar overheads when accelerated in hardware.

Our work can be extended with evaluation of different modular multiplication algorithms that can be used to implement our custom instruction. Their performance and area overhead can be evaluated. We showed trade offs between different instruction

encodings, future work can evaluate their performance and may extend the system with different modular operation instructions that are computationally expensive like reduction.

The last conditional subtraction can be eliminated if the loop in Figure 4.1 is extended by two iterations. This is simply done by adding two bits to the higher bits of operand and increasing the operand length. Eliminating subtraction simplifies the hardware implementation as the additional multiplexer port and negation of operand $N$ is no longer needed which decreases the hardware size by 14% in our implementation. Change in the operand length ($n$) affects both software and hardware. With the existing state machine, partial execution would require 2 additional calls to MMUL. This can be remedied with merging the last three cycles of the loop into a single call to MMUL to maintain the same operation. The precalculated value R in Listing 6.2 Line 2 should also be recalculated with updated $n$. Overall, it would require software developers to be more aware of the underlying hardware implementation. That is why we decided to pay the additional hardware cost to provide a more consistent interface to software.

# REFERENCES


1. Columbus, L., "2017 Roundup Of Internet Of Things Forecasts", https://www.forbes.com/sites/louiscolumbus/2017/12/10/2017-roundup-of-internet-of-things-forecasts, 2017, accessed on 1 Sep 2020.

2. Division, C. S., I. T. Laboratory, N. I. of Standards, Technology and D. of Commerce, "Lightweight Cryptography", https://csrc.nist.gov/projects/lightweight-cryptography, accessed on 1 Sep 2020.

3. Blaner, B., B. Abali, B. M. Bass, S. Chari, R. Kalla, S. Kunkel, K. Lauricella, R. Leavens, J. J. Reilly and P. A. Sandon, "IBM POWER7+ processor on-chip accelerators for cryptography and active memory expansion", *IBM Journal of Research and Development*, Vol. 57, No. 6, pp. 3:1–3:16, 2013.

4. Taneja, M., "A framework for power saving in IoT networks", *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 369–375, 2014.

5. Hendrickson, A., "Sleep, Sense, Connect: Low-Power IoT Design", https://www.electronicdesign.com/technologies/embedded-revolution/article/21805040/sleep-sense-connect-lowpower-iot-design, 2017, accessed on 1 Sep 2020.

6. Guo, G., Q. Qian and R. Zhang, "Different Implementations of AES Cryptographic Algorithm", *IEEE 17th International Conference on High Performance Computing and Communications, IEEE 7th International Symposium on Cyberspace Safety and Security, and IEEE 12th International Conference on Embedded Software and Systems*, pp. 1848–1853, 2015.

7. Turan, F. and I. Verbauwhede, "Compact and Flexible FPGA Implementation of Ed25519 and X25519", *ACM Trans. Embed. Comput. Syst.*, Vol. 18, No. 3, pp.

1–21, 2019.

8. Harb, S. and M. Jarrah, "FPGA Implementation of the ECC Over GF(2m) for Small Embedded Applications", *ACM Trans. Embed. Comput. Syst.*, Vol. 18, No. 2, pp. 17:1–17:19, 2019.

9. Nan, L., X. Zeng, Q. Ding, W. Li, Y. Du and L. Chen, "Research of Special Instructions for Finite Field X Multiplications of Cryptographic Algorithms", *IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pp. 1608–1613, 2018.

10. Bartolini, S., I. Branovic, R. Giorgi and E. Martinelli, "A performance evaluation of ARM ISA extension for elliptic curve cryptography over binary finite fields", *16th Symposium on Computer Architecture and High Performance Computing*, pp. 238–245, 2004.

11. Gueron, S., "White Box AES Using Intel's New AES Instructions", *10th International Conference on Information Technology: New Generations*, pp. 417–421, 2013.

12. Hess, P., "SEC 2: Recommended Elliptic Curve Domain Parameters", https://www.secg.org/sec2-v2.pdf, 2010, accessed on 1 Sep 2020.

13. Overmars, A., *Modern Cryptography - Current Challenges and Solutions*, chap. Survey of RSA Vulnerabilities, IntechOpen, 2019.

14. Lenstra, A., J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung and C. Wachter, *Ron was wrong, Whit is right*, Tech. rep., IACR, 2012.

15. Miorandi, D., S. Sicari, F. D. Pellegrini and I. Chlamtac, "Internet of things: Vision, applications and research challenges", *Ad Hoc Networks*, Vol. 10, No. 7, pp. 1497 – 1516, 2012.

16. Bernsmed, K., C. Froystad, P. H. Meland and T. A. Myrvoll, "Security requirements for SATCOM datalink systems for future air traffic management", *IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, pp. 1–10, 2017.

17. Akram, R. N., K. Markantonakis, K. Mayes, P. Bonnefoi, D. Sauveron and S. Chaumette, "Security and performance comparison of different secure channel protocols for Avionics Wireless Networks", *IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pp. 1–8, 2016.

18. Asanović, K. and D. A. Patterson, *Instruction Sets Should Be Free: The Case For RISC-V*, Tech. Rep. UCB/EECS-2014-146, EECS Department, University of California, Berkeley, 2014.

19. "Official RISCV Foundation Website", https://riscv.org/, accessed on 1 Sep 2020.

20. Asanović, K., R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo and A. Waterman, *The Rocket Chip Generator*, Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.

21. Celio, C., D. A. Patterson and K. Asanović, *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*, Tech. Rep. UCB/EECS-2015-167, EECS Department, University of California, Berkeley, 2015.

22. Gautschi, M., P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak and L. Benini, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 25, No. 10, pp. 2700–2713, 2017.

23. Duran, C., D. L. Rueda, G. Castillo, A. Agudelo, C. Rojas, L. Chaparro, H. Hur-

tado, J. Romero, W. Ramirez, H. Gomez, J. Ardila, L. Rueda, H. Hernandez, J. Amaya and E. Roa, "A 32-bit RISC-V AXI4-lite bus-based microcontroller with 10-bit SAR ADC", *IEEE 7th Latin American Symposium on Circuits Systems (LASCAS)*, pp. 315–318, 2016.

24. "RISC-V SweRV Core Available to Open Source - Western Digital", https://blog.westerndigital.com/risc-v-swerv-core-open-source/, 2019, accessed on 1 Sep 2020.

25. "Common Vulnerabilities and Exposures (CVE)", `https://cve.mitre.org/`, accessed on 1 Sep 2020.

26. Tweneboah-Koduah, S., K. E. Skouby and R. Tadayoni, "Cyber Security Threats to IoT Applications and Service Domains", *Wireless Personal Communications*, Vol. 95, No. 1, p. 169–185, 2017.

27. "Rust Programming Language", `https://www.rust-lang.org/`, accessed on 1 Sep 2020.

28. "CVE-2020-3144", https://nvd.nist.gov/vuln/detail/CVE-2020-3144, 2020, accessed on 1 Sep 2020.

29. "CVE-2020-10987", https://nvd.nist.gov/vuln/detail/CVE-2020-10987, 2020, accessed on 1 Sep 2020.

30. "CVE-2019-1010298", https://www.cvedetails.com/cve/CVE-2019-1010298/, 2019, accessed on 1 Sep 2020.

31. "CVE-1999-0018", https://www.cvedetails.com/cve/CVE-1999-0018/, 1999, accessed on 1 Sep 2020.

32. "CVE-1999-0002", https://www.cvedetails.com/cve/CVE-1999-0002/, 1999, accessed on 1 Sep 2020.

33. "CVE-2019-1010060", https://www.cvedetails.com/cve/CVE-2019-1010060/, 2019, accessed on 1 Sep 2020.

34. Liu, J. and W. Sun, "Smart Attacks against Intelligent Wearables in People-Centric Internet of Things", *IEEE Communications Magazine*, Vol. 54, No. 12, pp. 44–49, 2016.

35. "Intel® 64 and IA-32 Architectures Software Developer's Manual Vol. 3A", `https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html`, accessed on 1 Sep 2020.

36. "Arm® Architecture Reference Manual", `https://developer.arm.com/documentation/ddi0487/latest`, accessed on 1 Sep 2020.

37. "CVE-2013-6809", https://www.cvedetails.com/cve/CVE-2013-6809/, 2013, accessed on 1 Sep 2020.

38. "CVE-2004-1373", https://www.cvedetails.com/cve/CVE-2004-1373/, 2004, accessed on 1 Sep 2020.

39. Gusmeroli, S., S. Piccione and D. Rotondi, "IoT Access Control Issues: A Capability Based Approach", *Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 787–792, 2012.

40. Ouaddah, A., A. A. Elkalam and A. A. Ouahman, "Towards a Novel Privacy-Preserving Access Control Model Based on Blockchain Technology in IoT", Á. Rocha, M. Serrhini and C. Felgueiras (Editors), *Europe and MENA Cooperation Advances in Information and Communication Technologies*, pp. 523–533, Springer International Publishing, Cham, 2017.

41. Alladi, T., V. Chamola, B. Sikdar and K.-K. R. Choo, "Consumer IoT: Security vulnerability case studies and solutions", *IEEE Consumer Electronics Magazine*,

Vol. 9, No. 2, pp. 17–25, 2020.

42. "Understanding Denial-of-Service Attacks", https://us-cert.cisa.gov/ncas/tips/ST04-015, 2019, accessed on 1 Sep 2020.

43. Kolias, C., G. Kambourakis, A. Stavrou and J. Voas, "DDoS in the IoT: Mirai and Other Botnets", *Computer*, Vol. 50, No. 7, p. 80–84, 2017.

44. "Announcing the ADVANCED ENCRYPTION STANDARD (AES)", https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf, 2001, accessed on 1 Sep 2020.

45. Diffie, W. and M. Hellman, "New directions in cryptography", *IEEE Transactions on Information Theory*, Vol. 22, No. 6, pp. 644–654, 1976.

46. Pammu, A. A., K. Chong, W. Ho and B. Gwee, "Interceptive side channel attack on AES-128 wireless communications for IoT applications", *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 650–653, 2016.

47. Mangard, S., "A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion", P. J. Lee and C. H. Lim (Editors), *Information Security and Cryptology — ICISC 2002*, pp. 343–358, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

48. Acıiçmez, O., W. Schindler and Ç. K. Koç, "Cache Based Remote Timing Attack on the AES", M. Abe (Editor), *Topics in Cryptology – CT-RSA 2007*, pp. 271–286, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

49. Barker, E., L. Chen, A. Roginsky, A. Vassilev and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf, 2018, accessed on 1 Sep 2020.

50. Brumley, D. and D. Boneh, "Remote Timing Attacks Are Practical", *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, p. 1, USENIX Association, USA, 2003.

51. Acıiçmez, O., Çetin Kaya Koç and J.-P. Seifert, "On the power of simple branch prediction analysis", *ACM Symposium on Information, Computer and Communications Security (ASIACCS'07)*, pp. 312–320, ACM Press, 2007.

52. Pellegrini, A., V. Bertacco and T. Austin, "Fault-based attack of RSA authentication", *Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 855–860, 2010.

53. Biehl, I., B. Meyer and V. Müller, "Differential Fault Attacks on Elliptic Curve Cryptosystems", M. Bellare (Editor), *Advances in Cryptology — CRYPTO 2000*, pp. 131–146, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

54. "Choosing safe curves for elliptic-curve cryptography", `http://safecurves.cr.yp.to/`, accessed on 1 Sep 2020.

55. Nundloll, V., B. Porter, G. S. Blair, J. Cosby, B. Emmett, B. Winterbourn, G. Dean, P. Beattie, R. Shaw, D. Jones, D. Chadwick, M. Brown, W. Shelley and I. Ullah, "The Design and Deployment of an End-to-end IoT Infrastructure for the Natural Environment", *CoRR*, Vol. abs/1901.06270, 2019.

56. Auer, L., C. Skubich and M. Hiller, "A Security Architecture for RISC-V based IoT Devices", *Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1154–1159, 2019.

57. Costello, C. and P. Longa, "FourQ: four-dimensional decompositions on a Q-curve over the Mersenne prime", T. Iwata and J. H. Cheon (Editors), *Advances in Cryptology Conference - ASIACRYPT 2015*, p. 214–235, Springer Berlin Heidelberg, 2015.

58. Behnia, R., M. O. Ozmen and A. A. Yavuz, "ARIS: Authentication for Real-Time IoT Systems", *IEEE International Conference on Communications (ICC)*, pp. 1–6, 2019.

59. Sridharan, R. and T. Jeyaprakash, "Security for Data in IOT Using a New APS Elliptic Curve Light Weight Cryptography Algorithm", *Artificial Intelligence Techniques for Advanced Computing Applications Lecture Notes in Networks and Systems*, p. 137–146, 2020.

60. Koblitz, N., "Elliptic Curve Cryptosystems", *Mathematics of Computation*, Vol. 48, No. 177, pp. 203–209, 1987.

61. Chelton, W. N. and M. Benaissa, "Fast Elliptic Curve Cryptography on FPGA", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 16, No. 2, pp. 198–205, 2008.

62. Sining Liu, F. Bowen, B. King and Wei Wang, "Elliptic curves cryptosystem implementation based on a look-up table sharing scheme", *IEEE International Symposium on Circuits and Systems*, p. 4, 2006.

63. Choi, H. M., C. P. Hong and C. H. Kim, "High Performance Elliptic Curve Cryptographic Processor Over GF($2^{163}$)", *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*, pp. 290–295, 2008.

64. Khan, Z. and M. Benaissa, "Throughput/Area-efficient ECC Processor Using Montgomery Point Multiplication on FPGA", *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 62, No. 11, pp. 1078–1082, 2015.

65. Kodali, R. K. and H. S. Budwal, "High performance scalar multiplication for ECC", *International Conference on Computer Communication and Informatics*, pp. 1–4, 2013.

66. Furbass, F. and J. Wolkerstorfer, "ECC Processor with Low Die Size for RFID

Applications", *IEEE International Symposium on Circuits and Systems*, pp. 1835–1838, 2007.

67. Alrimeih, H. and D. Rakhmatov, "Fast and Flexible Hardware Support for ECC Over Multiple Standard Prime Fields", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 22, No. 12, pp. 2661–2674, 2014.

68. Zhou, L., C. Su, Z. Hu, S. Lee and H. Seo, "Lightweight Implementations of NIST P-256 and SM2 ECC on 8-bit Resource-Constraint Embedded Device", *ACM Trans. Embed. Comput. Syst.*, Vol. 18, No. 3, pp. 23:1–23:13, 2019.

69. Liu, Z. and J. Großschädl, "New Speed Records for Montgomery Modular Multiplication on 8-Bit AVR Microcontrollers", D. Pointcheval and D. Vergnaud (Editors), *Progress in Cryptology – AFRICACRYPT 2014*, pp. 215–234, Springer International Publishing, Cham, 2014.

70. Microsoft, "FourQlib", `https://github.com/microsoft/FourQlib`, 2017, accessed on 1 Sep 2020.

71. Chang, C., W. Lee, Y. Liu, B. Goi and R. C. . Phan, "Signature Gateway: Offloading Signature Generation to IoT Gateway Accelerated by GPU", *IEEE Internet of Things Journal*, Vol. 6, No. 3, pp. 4448–4461, 2019.

72. Zheng, X., C. Xu, X. Hu, Y. Zhang and X. Xiong, "The Software/Hardware Codesign and Implementation of SM2/3/4 Encryption/Decryption and Digital Signature System", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2019.

73. Sharif, M. U., R. Shahid, K. Gaj and M. Rogawski, "Hardware-software codesign of RSA for optimal performance vs. flexibility trade-off", *26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, 2016.

74. Waterman, A., Y. Lee, D. A. Patterson and K. Asanović, *The RISC-V Instruction*

*Set Manual, Volume I: User-Level ISA, Version 2.1*, Tech. Rep. UCB/EECS-2016-118, EECS Department, University of California, Berkeley, 2016.

75. Waterman, A., Y. Lee, R. Avizienis, D. A. Patterson and K. Asanović, *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9*, Tech. Rep. UCB/EECS-2016-129, EECS Department, University of California, Berkeley, 2016.

76. Gayoso Martínez, V., L. Hernandez Encinas and C. Sánchez Ávila, "A Survey of the Elliptic Curve Integrated Encryption Scheme", *Journal of Computer Science and Engineering*, Vol. 2, pp. 7–13, 2010.

77. Bernstein, D. J., "Curve25519: New Diffie-Hellman Speed Records", M. Yung, Y. Dodis, A. Kiayias and T. Malkin (Editors), *Public Key Cryptography - PKC 2006*, pp. 207–228, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

78. Montgomery, P. L., "Modular Multiplication without Trial Division", *Mathematics of Computation*, Vol. 44, No. 170, pp. 519–521, 1985.

79. Tenca, A. F. and C. K. Koc, "A scalable architecture for modular multiplication based on Montgomery's algorithm", *IEEE Transactions on Computers*, Vol. 52, No. 9, pp. 1215–1221, 2003.

80. Mamiya, H., A. Miyaji and H. Morimoto, "Efficient Countermeasures against RPA, DPA, and SPA", M. Joye and J.-J. Quisquater (Editors), *Cryptographic Hardware and Embedded Systems - CHES 2004*, pp. 343–356, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

81. "GCC insn directive", https://embarc.org/man-pages/as/RISC_002dV_002dFormats.html, accessed on 1 Sep 2020.

82. Davide Schiavone, P., F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand and L. Benini, "Slow and steady wins the race? A comparison of ultra-low-power

RISC-V cores for Internet-of-Things applications", *27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pp. 1–8, 2017.

83. Karatsuba, A. and Y. P. Ofman, "Multiplication of Many-Digital Numbers by Automatic Computers", *Dokl. Akad. Nauk SSSR*, Vol. 145:2, pp. 293–294, 1963.

84. Samaila, M. G., J. B. F. Sequeiros, T. Simões, M. M. Freire and P. R. M. Inácio, "IoT-HarPSecA: A Framework and Roadmap for Secure Design and Development of Devices and Applications in the IoT Space", *IEEE Access*, Vol. 8, pp. 16462–16494, 2020.

85. Tehrani, E., T. Graba, A. S. Merabet, S. Guilley and J. Danger, "Classification of Lightweight Block Ciphers for Specific Processor Accelerated Implementations", *26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 747–750, 2019.

86. Genkin, D., L. Valenta and Y. Yarom, "May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519", *ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, p. 845–858, Association for Computing Machinery, 2017.

87. Yarom, Y. and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack", *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, p. 719–732, USENIX Association, USA, 2014.

88. Alam, M., H. A. Khan, M. Dey, N. Sinha, R. Callan, A. Zajic and M. Prvulovic, "One&Done: A Single-Decryption EM-Based Attack on OpenSSL's Constant-Time Blinded RSA", *27th USENIX Security Symposium (USENIX Security 18)*, pp. 585–602, USENIX Association, Baltimore, MD, 2018.

89. Fan, J., X. Guo, E. De Mulder, P. Schaumont, B. Preneel and I. Verbauwhede, "State-of-the-art of secure ECC implementations: a survey on known side-channel

attacks and countermeasures", *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 76–87, 2010.

90. Page, D., "Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel", *IACR Cryptology ePrint Archive*, Vol. 2002, p. 169, 2002.

91. Mantel, H., A. Weber and B. Köpf, "A Systematic Study of Cache Side Channels Across AES Implementations", E. Bodden, M. Payer and E. Athanasopoulos (Editors), *Engineering Secure Software and Systems*, pp. 213–230, Springer International Publishing, Cham, 2017.

92. Coppens, B., I. Verbauwhede, K. D. Bosschere and B. D. Sutter, "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors", *30th IEEE Symposium on Security and Privacy*, pp. 45–60, 2009.

93. Popp, T., S. Mangard and E. Oswald, "Power Analysis Attacks and Countermeasures", *IEEE Design Test of Computers*, Vol. 24, No. 6, pp. 535–543, 2007.

94. Fouque, P.-A., N. Guillermin, D. Leresteux, M. Tibouchi and J.-C. Zapalowicz, "Attacking RSA–CRT Signatures with Faults on Montgomery Multiplication", E. Prouff and P. Schaumont (Editors), *Cryptographic Hardware and Embedded Systems – CHES 2012*, pp. 447–462, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.