Tesi di Dottorato

Università degli Studi di Napoli "Federico II"

Dipartimento di Ingegneria Elettrica
e Tecnologie dell'Informazione

Dottorato di Ricerca in
Information Technology and Electrical Engineering

Cryptographic extensions for custom and GPU-like architectures

Domenico Argenziano

Il Coordinatore del Corso di Dottorato	Il Tutore
Ch.mo Prof. Daniele Riccio	Ch.mo Prof. Alessandro Cilardo

A. A. 2016–2017

*"To my parents,
to my sister and my family,
to my friends
and to Erminia, as well."*

Acknowledgments

First, I want to express my sincere gratitude to...

my tutor Alessandro Cilardo, for his great patience with me, my colleagues Mirko Gagliardi and Innocenzo Mungiello for their helpfulness, my temporary co-tutor Clemente Galdi for his invaluable contribution, Flemming Christensen for having allowed me to make an interesting abroad experience.

Contents

List of Figures

Introduction

The need for information security has increasily become a serious concern in the era of widespread digitalization, mobile networking and Cloud-based services. Even bigger concerns have arisen since Cloud computing and Big Data paradigms for storing, processing and manipulating digital data have achieved wide diffusion on a large scale and a large class of distributed applications over the past two decades. Traditional security techniques have focused mostly on the storage and transport of information, e.g. by encrypting files, authenticating servers and setting up secure channels. In Cloud computing scenario, even these techniques are starting to appear insufficient, since the peculiar aspect of Cloud computing is outsourcing data storage and computation to a third-party remote resource provider, which, in turn, will allocate programs and data over a large network of machines, with little to no control by data owners. Certainly, data can be stored and transmitted in encypted form, but if it has to be processed, then data should be in unencrypted form. In this case, the customer must trust the Cloud service provider not to leak, maliciously or unintentionally, confidential data while also preserving data integrity against illicit manipulations. The ideal solution would be a tecnique which would allow to carry out computation on encrypted data. This, which could seem like an impossible task, can be actually achieved using *homomorphic encryption*, which actually allows computation to take place on encrypted data on the server side. While existing encryption schemes, such as RSA, were already known to be at least partially homomorphic, it is only with the seminal work of Craig Gentry in 2009 that a *fully* homomorphic scheme was finally devised. An alternate technique is that of Yao's *Garbled Circuits* [2] and all derived techniques, also including hybrid approaches. Unfortunately, these schemes come with a prohibitively high computational cost which prevents them to be practically used in any real scenario. Nonetheless, noticeble progresses have been accomplished from the algorithmic point of view, reducing time complexity and, hopefully, further improvements may come in the future, which will make homomorphic encryption more practical. Even in that case, software and hardware optimization techniques are still needed in order to make its implementation feasible.

When we consider more standard cryptographic primitives, their computational cost, even though more reasonable, can still be quite high, especially in the case of public key cryptography. Such cost is often regarded as an undesired overhead, since some computing resources have to be taken away from *useful* tasks. Indeed, serious security protocols for Internet were adopted well after Internet itself widespread diffusion, when security threats had already became a major concern.

This thesis work will deal with the exploration of acceleration solutions for cryptographic operations, both those more advanced, such as homomorphic encryption, that more standard ones, such as AES for symmetric key cryptography and RSA and Elliptic Curves for public key cryptography. Given the peculiar nature of cryptographic operations, it is usually agreed that some kind of hardware acceleration is usually needed, since it can yields far better performances that software-only approaches based on commodity processors. Indeed, most security libraries (e.g. OpenSSL) tries to exploit special extensions, such as SIMD instructions, present on commercial CPUs, like Intel AVX.

Current trends in computing architectures offer a variety of solutions for acceleration, ranging from general-purpose multi/many-core processors to Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs).

FPGAs have a large potential for security-related processing and have proved to be an effective platform for cryptographic processing [3] as cryptoalgorithms have peculiar characteristics, like integer computation, bit-level manipulation, etc., that make standard platforms like CPUs and GPUs less competitive. On the other hand, hardware reconfigurability allows the designer to customize the system possibly based on specific parameters, e.g. a cryptographic key, making FPGAs an ideal platform for cryptographic acceleration [4, 5, 6] as well as for cryptanalytic purposes [7, 8, 9]. FPGA platforms have also been explored as a secure compute/storage environment [10, 11, 12] as well as for implementing special security-related features like Physically Unclonable

Functions [13].

The thesis presents the results achieved for the acceleration of operations essential to homomorphic cryptography, specifically, the integer multiplication of very long operands, based on the Schonhage-Strassen algorithm and implemented with an ad-hoc FPGA hardware. Then, we report the exploration of novelty approaches for cryptographic acceleration, based on vectorial dedicated architectures, such as General Purpose GPU, which are software programmable, with the corresponding implementation of symmetric and public key algorithms (namely, AES encryption and Montgomery multiplication) exploting dedicated units for improved performances.

The outline of the thesis is the following:

Chapter 1 presents the design of a custom accelerator for homomorphic encryption. More specifically, such accelerator is intended to perform high order number theoretic FFT, which is the main building block for very long operand multiplication. Such multiplication is the common bottleneck of homomorphic encryption schemes.

Chapter 2 describes an implementation of the Montgomery multiplication algorithm, widely used in common asymmetric schemes (such as RSA) based on the extension of a GPU-like core with special function units, exporting such new functionalities through an extended ISA which allowed to rewrote the original algorithm exploiting the hardware acceleration.

Chapter 3 describes extensions for a GPU-like core in order to accelerate AES-128 symmetric encryption. Two versions of the algorithm implementation are reported, the first which tries to exploit the Scrathpad memory already present in the core, the latter using completely ad hoc units, in order to achieve the best performances.

Chapter 4 reports the experimental results for the solutions presented in Chapter 2 and Chapter 3.

Chapter 1

Custom accelerator for homomorphic encryption

Since its widespread usage and popularity as a distributed platform for storing and processing heterogeneous data, Cloud computing environment [14, 15, 16] has posed serious problems from the point of view of security, since data confidentiality on a remote server cannot be reliably verified by the customer. This is not a concern when only the transmission or storage of data is taken into consideration, since it can be well protected using several cryptographic techniques. Specifically, symmetric key cryptosystems are usually used to ensure data confidentiality, while public key ones can satisfy integrity and authenticity requirements. This cannot be immediately applied in the case of executions of programs on confidential data, since such data cannot be encrypted. Therefore, in such case, the customer would have no choice but to trust the service provider not to violate data confidentiality nor to illicitly manipulate such data.

A promising answer to this concern may come from homomorphic encryption (HE). Introduced in its full form by the seminal work by Craig Gentry [17] just a few years ago, this sophisticated cryptographic technique allows arbitrary computation to take place on encrypted data, with little chance for the server performing the computation of accessing user data in plain form. Beside Gentry's scheme, based on the properties of ideal lattices, various alternative solutions have been proposed, the most relevant being the van Dijk, Gentry, Halevi and Vaikuntanathan's (DGHV) scheme over the integers, and the Brakerski and

Vaikuntanathan's scheme [18] based on the Learning with Errors (LWE) and Ring Learning with Errors (RLWE) problems [19]. While in their current instantiations homomorphic primitives handle information and operation at a very low level, i.e. at the bit level, in a perspective scenario HE can act as an enabling tool for a number of different applications, including multiparty computation, where several parties process a common, public function on their inputs while keeping their individual inputs private, medical applications, allowing patient records to remain confidential while being processed in the cloud; financial applications, electronic voting, etc. Despite its great potential, however, homomorphic encryption suffers from a high computational cost, both in terms of time and memory occupancy, which currently prevents its practical use.

In that respect, as implied by our introductory comment, the availability of dedicated FPGA-based acceleration in server settings might play a key role. Motivated by both the security challenges and emerging compute technologies of today's cloud scenarios, this chapter we present the architecture and FPGA-based implementation of a dedicated hardware accelerator addressing the prohibitive computing demand of homomorphic encryption. In particular, this chapter will describe a highly customized FPGA accelerator implementing ultralong-integer multiplication, the main performance bottleneck in most homomorphic encryption schemes. In the following, we'll present an implementation based on an Altera's Stratix V FPGA platform. The experimental results collected from the hardware synthesis show significant improvements in terms of execution time –under comparable hardware cost– against alternative solutions previously presented in the technical literature.

1.1 Applications

Before dwelling in the technical details of both homomorphic encryption and our proposed accelerator, we will briefly review some use cases which can greatly benefit from the use of such cryptographic system.

Delegation of computation This is the most immediate and obvious use case for homomorphic encryption. Nowadays, several possible delegation scenarios have emerged, ranging from the provisioning of virtual machines and platforms to remote data storage and applications, such as Google Apps. All these kinds of services share the risk of being physically located outside the control of the customer. Using homomorphic encryption, the customer sends to the service provider an encrypted version of his software and data. The code included in the customer package is then executed on the remote server in encrypted form and when its execution is completed, the encrypted results are sent back to the customer, who retrieves the plaintext result by decrypting with the secret key.

Secret Query Evaluation If a company needs to store a database using a Cloud service, it has to trust such service provider both for the privacy of database content itself and of the queries the customer is going to perform on that database. That is, the service provider, not only has access to the database as a plaintext, but it can also log any customer query, with detail about the specific search and its time. While it is possible, with conventional cryptography, to hide the query and its result from an external adversary by using authentication and secure channels. Nonetheless, in some scenario, the customer may want to hide the query and its result from the Cloud provider itself and, optionally, he may wish to hide the content of the database itself. An example of the first situation is a classic Google query, where the searched database is obviously not private but the user may desire to hide the content (and results) of his/her search. An example of the second case is when the database contain medical record, which are sensitive data and are therefore better stored encrypted.

Medical records private access This scenario represents the most strict version of the one in the previous paragraph. Patient records should be stored in encrypted form given the sensitive nature of such data. Homomorphic encryption can allow to perform queries on the database in encrypted form, with result decrypted by the user once it has been received. Users are usually medically doctors, that may wish to access records remotely, for diagnosis, monitoring and other tasks. Additionally, it should be possible for patients to control which persons he wants to share his medical history with.

Mobile Agents Using an homomorphic cryptosystem in order to protect mobile agents can be considered a special case of delegation. Similarlty to the delegation case, mobile code and related data are homomorphically but in this case their are not just sent to a single server, but are passed from one computing node to another. This scenario canbe useful for data collection, where the customer is interested in computing some result which depends on data coming from different sources. Each source (node) executes the encrypted code adding their own input data, computing a new intermediate result without revealing their input. Only the final recipient (the customer) will be able to decrypt the final result.

Multi-Party Computation Multi-Party (MP) computation protocols allow two parties or more parties to compute a public function over secret input data in a way that prevent any party to know the secret input share hold by any other party. This can be achieved with our system as sketched in the Mobile Code use case. In the MP scenario all participants act as both information provider and information recipient. An advantage of our system over the approach of the garbled circuits [2] is that the garbled circuits require much more communication between the interacting parties to generate and exchange the function representation.

E-Voting Electronic Voting is one of the earliest and most widely studied application field to have used homomorphic (mostly *partially* homomorphic, such as ElGamal) encryption to achieve vote privacy. In an election scenario, the homomorphic encryption may act as a powerful tool that enable an untrusted third party provider to compute the tally given the encrypted votes without knowing the content of such votes. One of the first work is that of Benaloh and Tuinstra [20], who used partial homomorphic encryption to implement *verifiable secret ballet elections* where anyone can verify the tally result. A voter employs such encryption scheme to encrypt his/her vote and as soon as the Polling Station has collected all votes, they are homomorphically tallied without disclosing any single vote and the secret key is used only at the end to decrypt and recover the result for each candidate. One main drawback is that a zero knowledge proof has to be supplied as a proof of validity. H. Jonker et al. [21] underline how fully homomorphic encryption schemes could be a significant improvement in e-voting systems, both supporting addition and multiplication and also preserving the structure of plaintexts.

Zero-knowledge proof Zero-knowledge proof is an important primitive of security protocols which enable one party to prove to another party that a certain statement is true without disclosing any additional information, that is, if the first party needs private information for such proof, it is not disclosed to anybody. It was first introduced by Goldwasser, Micali and Rackoff in 1985 [22]. Since then, it has received noticeble attention from the scientic world, given its potential for authentication system. In such a scenario, a user wants to prove his/her identity to a host by using some secret information (password) without communicating anything of such secret to the server. In this particular case, we have a zero-knowledge *proof of knowledge*.

Gentry himeself shows [17] in his dissertation that homomorphic encryption can be used in the construction of non-interactive zero knowledge (NIZK) proofs of small size.

1.2 Background

1.2.1 Mathematical background

In order to explain the working of a public key homomorphic cryptosystem, we'll firstly introduce a simple symmetric key scheme.

As encryption key, we choose an odd integer p, such that $p \in [2^{\eta-1}, 2^{\eta})$ Each plaintext m is made up of a single bit, $m \in \{0,1\}$, and is encrypted as $c = pq + 2r + m \Rightarrow c \equiv (2r + m) \bmod p$, where r (noise) and q are randomly chosen in suitable intervals, with $|2r| < p/2$. A ciphertext c is decripted as $m = (c \bmod p) \bmod 2$.

At this point, we can ask ourselves if this scheme is homomorphic with respect to some arithmetic operations. Let's consider then two ciphertexts, c_1, c_2, such that $c_1 = pq_1 + 2r_1 + m_1$,$c_2 = pq_2 + 2r_2 + m_2$. we have that:

- Addition: $c_1 + c_2 = p(q_1 + q_2) + 2(r_1 + r_2) + (m_1 + m_2)$
- Multiplication: $c_1c_2 = p(q_1q_2 + 2q_1r_2 + q_1m_2 + 2r_1q_2 + m_1q_2) + (4r_1r_2 + 2r_1m_2 + 2m_1r_2) + m_1m_2$

The previous scheme, which is a symmetric, can be easily turned in a public key one (still, as a toy scheme with no security guarantee): firstly, we compute a sequance of integers $x_i = q_ip + 2r_i$, with q_i, r_i chosen as previously stated. Such sequence will be the public key, while the integer p will be the secret key. Each x_i can be viewed as an *encryption of zero*, using the previous scheme. The new asymmetric encryption operation will be:

- $c = \sum x_i + m = \sum(q_i p + 2r_i) + m = p\underbrace{\sum q_i}_{q} + 2\underbrace{\sum r_i}_{r} + m$, where the summations are intended over a subset of the public key.

Decryption works as before.

As we have shown, we can carry out arithmetic operations on ciphertexts and getting them applied to the corresponding plaintexts. Decryption of a cipher text is correct as long as the noise is smaller than $p/2$. Anyway noise grows when we perform additions and multiplication; more specifically, addition will roughly double the noise while multiplication will square it, so the multiplication is usually the main concern. There are then only a limited number of operations we can perform before the noise becomes too big and the decryption will yield an incorrect result. Such an homomorphic scheme is called a *somewhat* homomorphic encryption scheme.

Before further analyzing the properties of FHE, we want the summarize the main existent groups of homomorphic schemes.

- Schemes preceding the Gentry's were essentially only partial or *somewhat* homomorphic, supporting limited kind of operations and, above all, only a limited number of such operations. A significant example is Paillier's cryptosystem [23], which supports both addition and multiplication by a constant. Table 1.1 presents a non exhaustive list of homomorphic schemes preceding Gentry's.

- Gentry's original work [17] presents an homomorphic schemes based on the properties of ideal lattices and, most importantly, he introduced the concept of *bootstrappability*, by which a somewhat homomorphic scheme obeying certain properties can be turned in a fully homomorphic one.

- FHE schemes subsequent to Gentry's work fall into three main categories:

  – Schemes derived directly from Gentry's ones. Main contributions to optimization have come from Smart and Vercauteren.

Cryptosystem	Kind of homomorphism	Assumption
RSA [24]	Multiplicative	Integer factorization
ElGamal [25]	Multiplicative	Computational/Decisional Diffie-Hellman
Pallier [23]	Additive and scalar multiplicative	Computationl/Decisional Composite Residuosity Class
Goldwasser-Micali [26]	Single-bit additive (XOR)	Quadratic Residuosity

Table 1.1: Partial homomorphic schemes precedent to Gentry's work

The most secure parameter setting yields a public key size of 2.3 GB.

- Dijk, Gentry, Halevi and Vaikuntanathan's scheme (DGHV) [18] over the integers. It is maybe the most understandable and will be the base for our following discussion. It has been subject to various optimizations and a secure parameter setting gives a public key size of about 802 MB.
- Brakerski and Vaikuntanathan's scheme [19] based on Learning with Errors and Ring Learning with Errors.

For simplicity's sake, in the following we'll focus on the FHE over the integer. Nonetheless, our accelerator can find application also in other FHE schemes, since it is not tied to a specific scheme but rather on the optimization of their common and most expensive arithmetic operations. We will also consider only single bit plaintexts and expressions containing only additions and multiplications mod 2. This is a common assumption in most FHE schemes, where operations over plaintexts reduce to logical XOR and AND. For this reason, functions over plaintexts are often referred to as *circuits*, since they can be expressed as a net of logical gates. An homomorphic encryption function maps a single bit plaintext to a very long integer in the ciphertext codomain. The logical XOR operation therefore will map to integer addition and, likewise, logical AND will map to integer multiplication. A boolean cirtuit will correspond to a polynomial function (additions and multiplications) in the ciphertext codomain.

Ageneral fully homomorphic encryption scheme comprises four primitives: *KeyGen, Encrypt, Decrypt* and *Evaluate*. It is therefore similar to a classic public key cryptosystem, with the addition of the *Evaluate* function. The *Evaluate* primitive takes as input a public key *pk*, a tuple of ciphertexts $\boldsymbol{c}$, a circuit C (i.e. a boolean expression) and outputs a ciphertext c_o, obtained applying che circuit on the input $\boldsymbol{c}$. Additionaly, *Evaluate* must take measures in order to contrast the noise growth, if we want to compute arbitrary circuits.

An encryption scheme ξ = (*KeyGen, Encrypt, Decrypt, Evaluate*) is *correct* for a class $\mathcal{S}$ of circuits if it is correct for all circuits $C \in S$. It is *fully* homomorphic if it is correct for any arbitrary circuit.

An additional consideration is that operations on ciphertexts not only increase the noise but also the ciphertext size itself. Therefore, another challenging property we desire is *compactness*, which informally means that the size of the ciphertext output of *Evaluate* does not depend on the size of the computed circuit. An encryption scheme ξ = (*KeyGen, Encrypt, Decrypt, Evaluate*) is *compact* if there exists a fixed polynomial bound $b(\lambda)$, such that, for any key pair (sk, pk), any circuit C and any finite ciphertext tuple $\boldsymbol{c}$ encrypted with pk, the size of the output of *Evaluate(pk, C,* $\mathbf{c}$*)* is no more than $b(\lambda)$.

The simple toy scheme we built previously can obviously manage a limited number of operations; such a scheme is therefore a *somewhat* homomorphic encryption scheme. Gentry proved that we can build a full homomorphic scheme from a somewhat one that can evaluate a little more than is decryption function; such property is the so called *bootstrappability*. In the following we'll consider an encryption scheme where the decryption function complexity depends only on the security parameter λ. To such scheme we add two *augmented* decription circuits, both taking a secret key and *two* ciphertexts, which are firstly decrypted and then added mod 2 in the first circuit or multiplied in the second. The set of these two circuits is denoted by $D_\xi(\lambda)$.

Let's consider a ciphertext c as the encryption of m under the public key pk_1 with secret key sk_1. Let (pk_2, sk_2) be a second key pair. Let $\overline{sk}_1$ be the encryption of sk_1 under the second public key (given the previous assumption, each bit of sk_1 is encrypted separately) and let $\bar{c}_i = Encrypt(pk_2, c_i)$ the encryption under pk_2 of each bit of the ciphertext $\bar{c}$. Then $c'' = Evaluate(pk_2, D_\mathcal{E}, \langle\overline{sk}_1, \bar{c}_1, \ldots, \bar{c}_\gamma\rangle)$ yields the plaintext

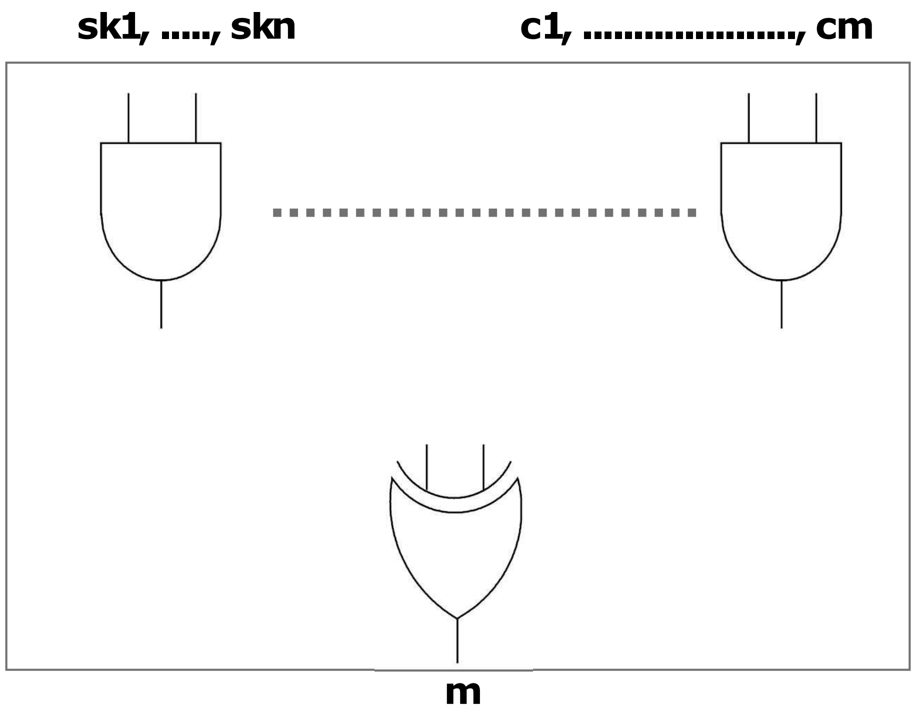
m encrypted under the new public key pk_2. This could seem a bit obscure but it is a simple consequence of the homomorphic nature of the scheme: we are considering the decryption function as a boolean circuit whose input bits are those of input bit are those of sk_1 and c. Instead of calculating directly this function, which whould yield the original plaintext m, we evaluate it homomorphically through *Evaluate*, that is, we consider the polynomial function corresponding to the decrypt boolean function (as stated before), and instead of the original bits of sk_1 and c, we consider their encryption under pk_2. The evaluation of such fuction, according to the homomorphic property, is an integer which is the encryption under pk_2 of the bit which we would get from evaluating the original boolean function. Such result would be the plaintext bit m, but it is actually encrypted pk_2. In practice, we removed the original encryption under pk_1 while keeping the encryption under pk_2 (as if we would remove a parcel internal envelope while it is still inside another external envelope). The procedure is shown in Figures 1.1 (a) and(b).

The overall result is to have reencrypted the original plaintext under a new key; the current noise is removed and the reencrytped ciphertext will have just the basic noise of a fresh encryption. This operation could therefore be considered a kind of *refresh.*

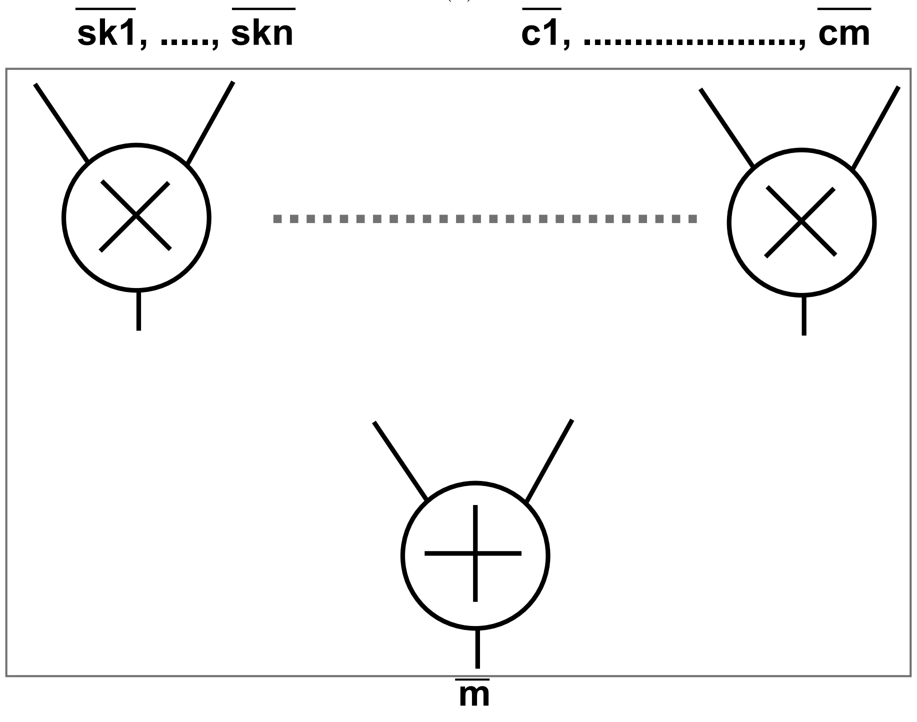
It is evident that since the decryption function is evalued as any other function, it contributes to increase the noise contained in the ciphertext. So the somewhat homomorphic encryption scheme, in order to be *bootstrappable*, must be able to support function at least as much complex as their decryption circuit. Actully, it should support at least one operation in addition to the decryption circuit, in order to have some elbow room for useful computation, before having to refresh the ciphertext. This additional operation must necessary be the multiplication (AND) since it produces the worst noise growth.

More formally, if ξ = (*KeyGen, Encrypt, Decrypt, Evaluate*) is an encryption scheme and $S_\xi(\lambda)$ is the set of all the circuits which are correct wrt ξ for each value of the security parameter λ, then we say that ξ is *bootstrappable* if $D_\xi(\lambda) \subseteq S_\xi(\lambda)$. With D_ξ we indicate the *augmented* decryption circuit, that is, a decryption plus an addition or a multiplication.

A somewhat homomorphic encryption scheme In this paragraph we start to introduce the FHE scheme over the integer of Dijk et al. [18]



(a)


$\overline{sk1}$, ..., $\overline{skn}$
$\overline{c1}$, ..., $\overline{cm}$
$\overline{m}$

(b)

Figure 1.1: Homomorphic evaluation of decryption function. In (a) is presented the original boolean function, while in (b) the corresponding arithmetic function evalued on the ciphertexts.

by first defining its underline somewhat homorphism. Firstly, we define the parameters of our scheme

- γ : bit-length of the integers x_i in the public key.
- η : bit-length of the secret key p.
- ρ : bit-length of the noise r_i.
- τ : cardinality of the public key (number of x_i elements).

Such parameters are subjected to the following constraints

- $\rho = \omega(log\lambda)$: protection against brute-force attacks on the noise.
- $\eta \geq \rho\Theta(\lambda log^2\lambda))$: in order to support deep enough circuits to evaluate the "squashed decryption circuit".
- $\gamma = \omega(\eta^2 log\lambda)$: protection against various lattice-based attacks.
- $\tau \geq \gamma + \omega(log\lambda)$: to use the leftover hash lemma in the reduction to approximate gcd.

An additional noise parameter is usually introduced: $\rho' = \rho + \omega(log\,\lambda)$. A possible choice for parameters is $\rho = \lambda$, $\rho' = 2\lambda$, $\eta = O(\lambda^2)$, $\gamma = O(\lambda^5)$ and $\tau = \gamma + \lambda$. An implementation setting is $\gamma \simeq 2 \cdot 10^7$, $\eta = 2652$, $\rho = 39$ with a public key size of about802 MB.

We introduce now the somewhat homomorphic encryption scheme after defining the set $\mathcal{D}_{\gamma,\rho}(p) = \{x = pq + r | q \in \mathbb{Z} \cap [0, 2^\gamma/p), r \in \mathbb{Z} \cap (-2^\rho, 2^\rho)\}$

- *KeyGen(λ)*: The secret key p is a η-bit odd integer chosen randomly from $(2\mathbb{Z} + 1) \cap [2^{\eta-1}, 2^\eta)$.
  For the public key, we choose randomly τ elements from the set $\mathcal{D}_{\gamma,\rho}(p)$ and relabel them so that x_0 is the largest. Repeat until x_0 is odd and x_0 is even mod p.
- *Encrypt(pk, m)*: We chose a random subset $S \subseteq \{1, 2, \ldots, \tau\}$ and a random integer $r \in (-2^{\rho'}, 2^{\rho'})$. We output $c = [m + 2r + 2\sum_{i \in S} x_i]_{x_0}$.
- *Evaluate(pk,C, $\boldsymbol{c}$)*: apply the circuit function of C to $\boldsymbol{c}$ as operations over the integers and return the result.

- *Decrypt(sk, c)*: $m = (c \bmod p) \bmod 2 = c - p \cdot \lfloor c/p \rceil \bmod 2 \xrightarrow{p\,is\,odd}$ $m = (c \bmod 2) \oplus (\lfloor c/p \rceil \bmod 2)$.

The encryption function is very similar to that of our toy scheme with the main variation that now we calculate also the remainder over x_0, which is easily explained. Let's consider

$$c = [m + 2r + 2\sum_{i \in S} x_i]_{x_0} \Rightarrow c = m + 2r + 2\sum_{i \in S} x_i + kx_0,$$
$$k \in \mathbb{Z} \Rightarrow c = m + 2r + 2\sum_{i \in S}(pq_i + r_i) + k(pq_0 + r_0) \Rightarrow$$
$$\Rightarrow c = \underbrace{m + 2(r + \sum r_i)}_{plaintext+noise} + \underbrace{p(\sum_{i \in S} 2q_i)}_{multiple\,of\,p} + k\underbrace{(pq_0 + r_0)}_{\equiv 2 \bmod p}$$

If k is small $\Rightarrow k \leq \tau$. That's why we chose x_0 to be even modulus p, so that it doesn't hide the parity of m.

Following Gentry's blueprint, we define a *permitted* circuit as one such that $\forall \alpha \geq 1$ and any set of integers $\leq 2^{\alpha(\rho'+2)}$ in absolute value, the output is at most $2^{\alpha(\eta-4)}$. $\mathcal{C}_{\mathcal{E}}$ denotes the set of permitted circuits. It can be proven that the previous scheme is correct for $\mathcal{C}_{\mathcal{E}}$. We recall that a fresh *ciphertext* has noise at most $2^{\rho'-2}$. if we apply *Evaluate* to it with a permitted circuit we have an output with noise at most $2^{\eta-4} < p/8$. A bound of $p/2$ would suffice, but this stricter bound will be useful in the following. Before continuing, we underline that the noise growth depends mainly by the multiplicative *depth* of the circuit, that is the *degree* of the multivariate polynomial homomorphically computed by the circuit.

At first glance it would seem possible to use the modular reduction by x_0 to keep the ciphertext size bounded during *Evaluate.* Unfortunately, it is not possible since after only one multiplication the ciphertext size gets much larger than x_0 so modular reduction will add a large multiple of x_0 introducing too much noise. Indeed, we have that for a *fresh* ciphertext $c < x_0$ and in the worst case $c \approx x_0$. For simplicity, we consider a squaring operation of c. Modular reduction is accomplished by adding an appropriate multiple of x_0, that is $c' = c^2 + kx_0\, k \in \mathbb{Z}$; k is roughly the quotient of c^2 by $x_0 \Rightarrow k \approx \frac{c^2}{x_0} \approx \frac{x_0^2}{x_0} = x_0$, which is much larger than the permitted noise. We observe that for bigger x_0 we have smaller k. Therefore, we add to the public key more elements like $x'_i = q'_i p + 2r'_i$ where the r'_i are choosen as before but the q'_i are larger. More precisely $q'_i \in \mathbb{Z} \cap [2^{\gamma+i-1}/p, 2^{\gamma+i}/p)$, $r'_i \in \mathbb{Z} \cap (-2^\rho, 2^\rho)$ with $i = 0, 1, \ldots, \gamma$ Whenever c get greater than 2^γ, we first reduce it modulo x'_γ, then modulo $x'_{\gamma-1}$ and so on down to x'_0(decreasing order). Without this variation, the output of *Evaluate* on two circuits that represent the same polynomial will be the same, but with this modification noise could be bigger in one case than in the other, yielding different outputs.

Scheme security The security of the scheme is based on the assumption of the hardness of the approximate integer greatest common divisors (approximate GCD). The (ρ, η, γ)-approximate GCD problem is defined as follow: given a polynomial number of samples from $\mathcal{D}_{\gamma,\rho}(p)$ for a random η-bit odd integer p, find p. Intuitively, the problem consists into finding p given a polynomial number of *near* multiples of p.

A fully homomorphic scheme According to Gentry's approach, a fully homomorphic scheme can be built starting from a *somewhat* homomorphic one which is also *bootstrappable*. In order to be bootstrappable, it should be able to homomorphically evaluate at least a circuit a bit more complex than the its own decryption cirtuit. It can be proved that the decryption circuit of the previous scheme doesn't belong to the set of permitted circuits. Gentry's approach is to transform the scheme to realize a *squashed* decryption circuit (that is, simpler and, above all, shallower). The is achieved by adding extra information to the public key about the secret key, making decryption phase more efficient. The ciphertext will contain more information in order to help decryption.

We describe then a modified scheme, with a *squashed* decryption function. We'll use three additional parameters: κ, θ, Θ. We add to the public key a set $\boldsymbol{y} = \{y_0, \ldots, y_\Theta\}$ of rational numbers in $[0, 2)$, so that there is a sparse subset S of size θ which satisfy $\sum_{i \in S} y_i \approx \frac{1}{p} \bmod 2$. As secret key, instead of using p we use the indicator vector of S.

The secret key is $\boldsymbol{s}$ and the public key $(pk, \boldsymbol{y})$.

The modified functions are as follow

- *KeyGen*:
  - Create sk and pk as before. Set $x_p = round(2^\kappa/p)$. Choose a random Θ-bit vector $\boldsymbol{s}$ with Hamming weight θ and let

Instance	λ	ρ	η	$\gamma \cdot 10^{-6}$	τ	Θ	Public key size
Toy	42	27	1026	0.15	158	144	77 KB
Small	52	41	1558	0.83	572	533	437 KB
Medium	62	56	2128	4.20	2110	1972	2.21 MB
Large	72	71	2698	19.35	7659	7897	10.3 MB

Table 1.2: Concrete parameters for the optimized FHE schemes

$S = \{i | s_i = 1\}$.

- Choose Θ random integers $u_i \in \mathbb{Z} \cap [0, 2^{\kappa+1})$ such that $\sum_{i \in S} u_i = x_p \bmod 2$. Set $y_i = u_i / 2^{\kappa}$.Each y_i is so a positive number in ¡2 with κ bits after the binary point.
- We get, as previously stated, that $\sum_{i \in S} y_i = (\frac{1}{p} - \Delta_p) \bmod 2$ with the error $\Delta_p < 2^{-\kappa}$.

- *Encrypt*: proceeds as before but at the end we do the following additional steps:
  - Compute $z_i = c \cdot y_i \bmod 2$, keeping only $n = \lceil log\theta \rceil + 3$ bits after the point.
  - Output both c and $\boldsymbol{z} = \langle z_1, \ldots, z_\Theta \rangle$.
- *Decrypt*: $m' = (c - round(\sum_i s_i z_i)) \bmod 2$.

It can be proved that this modified scheme is still correct for the set $C(\mathcal{P}_\mathcal{E})$ of circuits that compute the permitted polynomials, as previously defined. If $D_\mathcal{E}$ is the set of augmented decryption circuits, it can also be prove then $D_\mathcal{E} \subseteq C(\mathcal{P}_\mathcal{E})$; that is, $\mathcal{E}$ is bootstrappable. We will omit here such proofs for space reasons.

Optimizations There have been several efforts toward optimizing the scheme, in order to reduce time complexity and especially the public key size. Significant optimizations have been proposed by Coron et al. [27][28]. Their most important contributions are public key compression and modulus switching Table 1.2 reports the concrete parameters for the best of such optimized schemes.

1.2.2 Previous works

Before describing the approach we followed, in this section we'll firstly reviews the existing research works concerning the implementation of the HE primitives. We are interested here in the so-called *Fully Homomorphic Encryption* (FHE) [17], allowing arbitrary and unlimited sequences of operations on encrypted data. A first implementation of a variant of the original Gentry's scheme [17] was proposed by Gentry and Halevi [29]. Their implementation, despite various optimizations and small-size security parameters, takes more than one second for encrypting a single bit on an Intel Xeon server. Recent software implementations include [30, 31]. An open-source implementation, hcrypt [32] is available on-line. Another library [33] contains an optimized implementation that reaches a significant speed-up over the previous implementation. Several research works concerning FHE computing platforms have looked for alternative architectures, particularly GPUs and FPGAs. GPUs offer high throughput and efficiency for data intensive computing, such as vector and linear algebra problems. FHE schemes can benefit from this architecture, since they are highly parallelizable with respect to data. FPGA technology offers, on the other hand, the flexibility of implementing a custom and targeted architecture at a low cost, as opposed to Application Specific Integrated Circuits (ASICs). Moreover, several FPGAs include built-in optimized blocks for multiply-and-accumulate operations, which can be effectively exploited when implementing large multiplication. A recent GPU implementation is presented in [34], based on NVIDIA GTX 690. It reaches spead-ups of 174, 7, and 13 for encryption, decryption and reencryption with respect to Gentry and Halevi's software implementation. Other recent implementations on GPUs are [35, 36]. Among the FPGA implementations, one of the most recent is presented in [1], which compares FPGAs and GPUs, namely Altera Stratix V and NVIDIA Tesla C2050. Their implementation efforts are fundamentally focused on the FFT multiplier building block. The authors conclude that the FPGA version is at least twice as fast as the GPU one, with lower power consumption. [37] and [38] proposes a full custom ASIC implementation of large-operand multiplication. The design includes also a 768 Kbit cache to minimize I/O transactions. A single multiplication is performed in 7.7 ms at 666 MHz. The authors also built the first custom hardware implementation [39] of the cryptographic primitives of Gentry-Halevi FHE scheme. The design includes optimizations previously introduced in [34] to reduce the number of FFT computations. Speed-up of 1.24, 99.44, and 10.32 for decryption, encryption and recryption are achieved compared to the software implementation of Gentry and Halevi.

As mentioned, most existing implementations focus around optimizing the most time consuming operations used by the various encryption schemes: multiplication and modular reduction. Such operations are performed on very large operands (millions of bits) and can benefit from better asymptotic algorithms. Most current implementations rely on the Schonhage-Strassen algorithm (SSA) for the multiplication of large integers. The SSA algorithm exploits the properties of the Discrete Fourier Transform over integers, and it pays off for operands of at least 100,000 bits.

Last, [40] proposed an FFT-based large integer multiplier, along with a Barrett reduction module. Their design was implemented on a Xilinx Virtex-7 FPGA and included the encryption primitive of Coron *et al.* FHE scheme [41, 42]. The results show a remarkable speed-up compared to existing software implementations.

Our research follows this work line, trying to realize FPGA based acceleration support for the execution of the most time consuming operations required by FHE schemes. In the following sections we will present our approach adopted and the results obtained so far.

1.3 Approach followed

Our research work is aimed to support probabilistic, fully homomorphic algorithms which comprise various approaches, such as the integer-based approach first introduced by M. van Dijk *et al.*, and schemes based on Lattice problems and Learning with Errors. We aim to exploit the potential of a highly-customized FPGA design for accelerating the most time consuming operations used by the encryption primitives: long-integer multiplication and modular reduction. Since the latter can be reduced to a combination of one or more multiplications, the priority is on accelerating multiplication on ultra-large operands (in the order of millions of bits). The FHE primitives can be implemented on the top of our accelerator, i.e. in software, this way decoupling the proposed design from the specific FHE scheme. We explored the efficient implementation of asymptotically faster (but inherently more complex) multiplication/reduction algorithms in place of usual schemes used for moderately large operands (thousands of bits).

The main algorithms available to perform integer multiplication are the following (in decreasing order of complexity):

- School-book (plain) ($O(n^2)$)
- Karatsuba ($O(n^{log3}) \approx O(n^{1.585})$)
- Toom-Cook ($O(n^{\frac{log5}{log3}}) \approx O(n^{1.465})$)
- Schönhage-Strassen ($O(n \cdot log\, n \cdot log\, log\, n)$)
- Fürer ($O(n \cdot log(n) \cdot 2^{O(log^{\star} n)})$)

Algorithms with lower computational complexity have higher hidden constants, since they are based on a more complex approach. Karatsuba and Toom-Cook algorithms are best suited for middle sized operand (in the order of a thousand bits) such as those required by the RSA scheme.

While Fürer's algorithm is almost prohibitively complex, Schönhage-Strassen algorithm (SSA), which exploits the properties of the Number-Theoretic Discrete Fourier Transform, can offer an asymptotically better complexity for operands of at least 100,000 bits, where the cost of hidden constant is better amortized.

Algorithm 1. Schönhage-Strassen multiplication

Input: operands x and y, prime modulus p, unit radix ω, DFT point number k, word width b.

Output: $z = x \cdot y$.

- Decompose operand x into a sequance x_t of b bit words, with $0 \leq t < k$. The first half of such sequence (x_0 to $x_{k/2-1}$) is made up with actual bits from operand x, while the second half of the sequence ($x_{k/2}$ to x_{k-1}) is filled with zeros. The same oepration is performed on operand y. Such sequences are considered as polynomial coefficients and the relationship with the original operands is given by Equation (1.1)

$$x = \sum_{i=0}^{k} (x_i \bmod p) \cdot 2^{i \cdot b}, \; y = \sum_{i=0}^{k} (y_i \bmod p) \cdot 2^{i \cdot b} \qquad (1.1)$$

- Compute a k-point integer DFT over the finite field $\mathbb{Z}/p\mathbb{Z}$ of sequences x_t and y_t, getting the sequences X_t and Y_t, with $0 \leq t < k$. The operations are summarized in Equation (1.2)

$$X_t = \sum_{i=0}^{k-1} x_i \cdot \omega^{i \cdot t} \pmod{p},\ Y_t = \sum_{i=0}^{k-1} y_i \cdot \omega^{i \cdot t} \pmod{p} \tag{1.2}$$

- Compute a point-wise multiplication over the finite field $\mathbb{Z}/p\mathbb{Z}$, getting a k-point sequence Z_t, with $0 \leq t < k$, as per Equation (1.3)

$$Z_t = X_t \times Y_t \pmod{p} \tag{1.3}$$

- Compute the k-point Inverse DFT of sequence Z_t over the finite field $\mathbb{Z}/p\mathbb{Z}$, getting the sequence z_t of length k, as per Equation (1.4)

$$z_t = k^{-1} \sum_{i=0}^{k-1} Z_i \omega^{-i \cdot t} \pmod{p} \tag{1.4}$$

- Compute the final result by performing the sum in Equation (1.5), propagating the long carry chain

$$z = \sum_{i=0}^{k} z_i \cdot 2^{i \cdot b} \tag{1.5}$$

The computational complexity of SSA is $O(n \cdot \log n \cdot \log\log n)$. The most time consuming operation in SSA is the computation of the DFT (and Inverse DFT) over the integer. The DFT is a well known transform, but it is more commonly found in signal processing, where it is defined on the complex field $\mathbb{C}$ and where the Fourier domain has a cllear and intuitively interpretation in terms of frequencies which compose a given signal. Anyway it is possible to define a *Generalized* DFT, which applies to a more general algebraic structure and is more abstract in concept (the Fourier domain has no interpretation in terms of frequencies), but nonetheless the most relevant DFT properties (especially the theorem of convolution) still hold. The complex exponentials are replaced by more general roots of unity, which anyways have to obey some properties, namely being *primitive* and *principal* roots. The Generalized DFT can then be applied to the modular integer arithmetic, subject to some constraints. We begin firstly by proving the following theorem.

Theorem 1.1 *Let R be a commutative unitary ring, N a natural number with $N \geq 2$ and $N \cdot 1_R$ a unit in the ring and suppose ω is a N^{th} principal root of unity. Then the homomorphism*

$$\begin{aligned}\psi : R[x] &\longrightarrow R^N \\ f(x) &\longmapsto (f(\omega^0), \dots, f(\omega^{N-1}))\end{aligned}$$

is suriettive with kernel $(x^N - 1)$.

The induced isomorphism $\varphi : R[X]/(x^N - 1) \to R^N$ is called the Discrete Fourier Transform *with respect to ω. It is represented by the matrix $DFT(N, \omega) = (\omega^{pq})_{1 \leq p,q \leq N} \in GL(N, R)$ and the inverse transformation by $IDFT(N, \omega) = N^{-1} \cdot (\omega^{-pq})_{1 \leq p,q \leq N} = N^{-1} DFT(N, \omega^{-1})$.*

[1]

Proof. $\ker \psi = \{f(x) \in R[x] : \psi(f(x)) = \underline{0} \Rightarrow (f(\omega^0), f(\omega^1), \dots, f(\omega^{N-1})) = (0, 0, \dots, 0)$

Therefore $\ker \psi = \cap_{i=0}^{N-1} (x - \omega^i)$.We must show that $\ker \psi = (x^N - 1)$.

By induction on j we get then that for $j = 0$, $f(\omega^j) = 0 \Rightarrow f(1) = 0 \Rightarrow f(x) = (x - 1) \cdot g(x)$.

Now let's suppose that $\forall i : 0 \leq i \leq j, N > j > 0$ we have that $f(\omega^i) = \prod_{i=0}^{j} (x - \omega^i) \cdot g(x)$.

Now let's consider the next component: $f(\omega^{j+1}) = 0 \Rightarrow \prod_{i=0}^{j} (\omega^{j+1} - \omega^i) \cdot g(\omega^{j+1}) = 0$

We can write also $\prod_{i=0}^{j} \omega^i (\omega^{j+1-i} - 1) \cdot g(\omega^{j+1}) = 0$. By the principality of ω, the factors $(\omega^{j+1-i} - 1)$ are not zero divisors $\Rightarrow g(\omega^{j+1}) = 0 \Rightarrow g(x) = (x - \omega^{j+1}) \cdot u(x)$ [2]

Then $\ker \psi = (x^N - 1)$. The induced homomorphism $\varphi : R[X]/(x^N - 1) \to im(\psi)$ is indeed an isomorphism with associated matrix $DFT(N, \omega)$.

We still need to show that φ is surjective, which can be done showing that $DFT(N, \omega)$ is invertible, that is $DFT(N, \omega) \cdot IDFT(N, \omega) = I(N)$.

$$a_{p,r} = \sum_{q=0}^{N-1} \omega^{pq} \cdot N^{-1} \omega^{-qr} = N^{-1} \cdot \sum_{q=0}^{N-1} \omega^{pq} \cdot \omega^{-qr} =$$

[1] *Unit* means an invertible element. Not to be confused with *unity.* If N is invertible then it cannot be divisible by the characteristic. So we are implicitly impling that ω is also primitive.

[2] $\cap_{i=0}^{N-1} (x - \omega^i)$ Intersection of the ideals generated by $(x - \omega^i)$,, i.e. the set of the polynomials $(x - \omega^i) \cdot g(x)$

$$N^{-1} \cdot \sum_{q=0}^{N-1} \omega^{q(p-r)} = \begin{cases} N^{-1} \sum_{q=0}^{N-1} 1 = 1 & for\, p = r \\ 0 & for\, p \neq r \end{cases}$$

The last descends from the principality of ω.

In order to apply the DFT to a modular integer ring $\mathbb{Z}_M$, some properties must be satisfied: firstly, the number of points N must be an invertible element in $\mathbb{Z}_M$, which is satisfied if $\gcd(N, M) = 1$. Then we must find a primitive root of unity, that is, a root of unity with order N. The multiplicative group generated by such root is cyclic and, according to Lagrange's theorem, its order must be a divisor of the order of any of its super-groups. The maximal multiplicative group in $\mathbb{Z}_M$ is given by all its invertible elements. It is denoted $\mathbb{Z}_M^{\star}$ and its order if $\varphi(M)$, where φ is the *totient* function. Therefore we must have that N divides $\varphi(M)$.

Computing directly the DFT would give a time complexity of $O(N^2)$; a more efficient method if using the *Fast Fourier Transform* algorithm of Cooley-Tukey [43] with a complexity of $O(N \cdot logN)$. It can be showed that their *divide-et-impera* method (which was actually invented by C.F. Gauss) can be applied also in the case of the DFT over the integer. Instead of using the classic Radix-2 based implementation, we choose to apply the generalized FFT approach in order to decompose the overall FFT into higher order sub-FFTs.

Starting from the basic DFT formula $F[k] = \sum_{n=0}^{N-1} f[n]\omega_N^{kn}$, we decompose N as $N = N_1 \cdot N_2$, so the input and output vectors can be split into N_1 sub-sequences of length N_2. Let $n = N_2 n_1 + n_2$ and $k = N_1 k_2 + k_1$ with $n_1, k_1 \in \{0, 1, \ldots, N_1 - 1\}$ and $n_2, k_2 \in \{0, 1, \ldots, N_2 - 1\}$. Then the DFT can be written as:

$$\begin{aligned} F[N_1 k_2 + k_1] &= \sum_{n=0}^{N-1} f[n]\omega_N^{kn} = \\ &= \sum_{n_2=0}^{N_2-1} [(\sum_{n_1=0}^{N_1-1} f[N_2 n_1 + n_2] \cdot \omega_{N_1}^{n_1 k_1}) \cdot \omega_N^{n_2 k_1}] \cdot \omega_{N_2}^{n_2 k_2} \end{aligned} \tag{1.6}$$

Despite the better time complexity, one of the main problem of the FFT (regardless of the chosen radix) is that its memory access pattern is

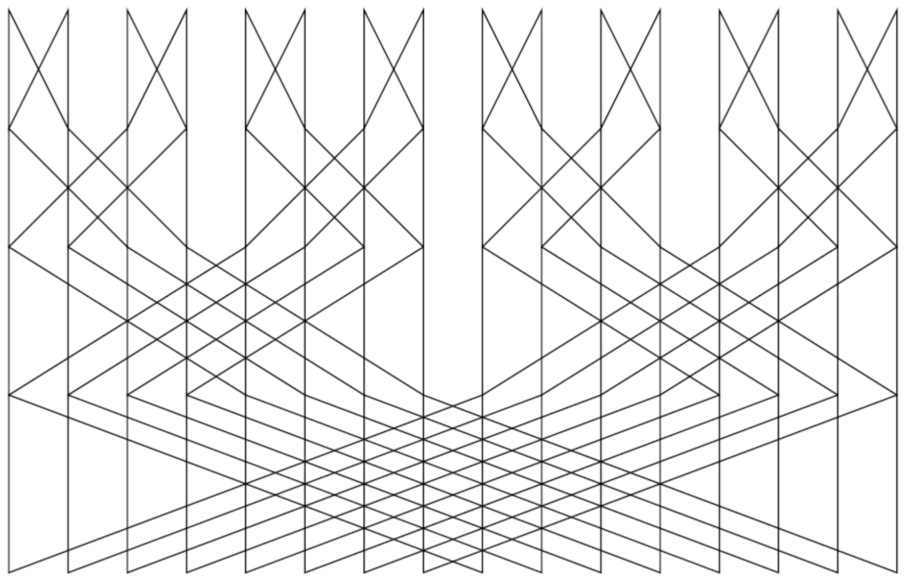


Figure 1.2: Butterfly scheme of the Radix-2 FFT.

highly non sequential, therefore badly affecting access time for cache and DDR memory. As can be seen from Figure 1.2 for the case of Radix-2 FFT, memory accesses are more spread out as we proceed from onr stage to the next one. This is the so called *butterly* access pattern. In custom hardware implementation, it is usual to use a scratchpad memory, with a dedicated address space and outside of any cache hierarchy. We'll follow a similar approach and we'll add a dedicated logic to manage the particular FFT access pattern.

We choose to perform the computation in the finite field $\mathbb{Z}/p\mathbb{Z}$, with prime p. By selecting a proper prime p, the modular multiplication in the finite field can be computed rapidly as simple shifts. In our implementation, we choose the Solinas prime number $p = 2^{64} - 2^{32} + 1$, since its particular properties greatly simplify arithmetic computing, especially multiplication. Table 1.3 shows some possible choices for the FFT modulus from the scientific literature, along with the corresponding bit width requirement, a feasible choice for the FFT number of points k and the relative root of unity ω.

In the following, we suppose to deal with operands of 786432 bits, which corresponds to the small security parameter setting for DGHV

	Modulus	Bit width	k	ω
Special form [44]	$2^{32}+1$	33	64	2
	$2^{64}+1$	65	128	2
Solinas form [45]	$2^{64}-2^{32}+1$	64	128	7
General form [46]	$2^{32}-2^{20}+1$	32	64	17

Table 1.3: Possible choices of modulus and corresponding parameters

and is used in many research papers. Operands are decomposed into 32K coefficients of 24 bits.

We need to apply FFT on 64K points, in order to accommodate the multiplication result. By applying Equation 1.6 recursively on the 64K-point DFT, it can be computed with three stages using radix-64 and radix-16 sub-transforms:

$$\sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1}\left[\sum_{n_3=0}^{N_3-1}\left(a[n]\omega_{N_3}^{n_3k_3}\right)\omega_{N_2N_3}^{n_2k_3}\omega_{N_2}^{n_2k_2}\right]\omega_N^{n_1k_2'}\omega_{N_1}^{n_1k_1} =$$
$$= \sum_{n_1=0}^{15}\sum_{n_2=0}^{63}\left[\sum_{n_3=0}^{63}\left(a[n]\omega_{N_3}^{n_3k_3}\right)\omega_{N_2N_3}^{n_2k_3}\omega_{N_2}^{n_2k_2}\right]\omega_N^{n_1k_2'}\omega_{N_1}^{n_1k_1} \qquad (1.7)$$

where

$$n = N_1N_2n_3 + N_1n_2 + n_1,$$
$$k_2' = k_3 + N_3k_2$$

Each stage can be efficiently parallelized, according to the available computing resources.

1.4 Architecture of the FPGA-based accelerator

The essential challenge addressed by this work is the support for ultralarge-operand FFT computation. One design objective we set for the proposed accelerator was the inherent support for scalability to ultralarge operands which, unlike many cryptographic primitives in different contests, may require a flexible and composable design solution applicable either to on- or off-chip scenarios, possibly in multi-FPGA settings available on the server side. Consequently, for the implementation of the 64K-point FFT building block, we devised a flexible distributed approach, relying on several nodes connected in a hypercube topology, which matches exactly the logical topology of the distributed FFT algorithm. The solution was initially prototyped on a multi-board platform based on low-end devices (Altera Cyclone V) then extended to a hybrid on-/off-chip solution relying on a larger device, i.e. an Altera Stratix V FPGA. The distributed approach, distinguishing our proposal from previous related works like [1], matches very well the FFT computation and ensures several advantages compared to shared memory approaches, such as better scalability and reduced use of global routing resources, which may be a major performance bottleneck especially on FPGAs. Using a hypercube topology, the number of communication stages for FFT computation is the hypercube dimension d. Figure 1.5 shows the hypercube network in the case of four computing nodes; Figure 1.6 shows instead the case for eight computing nodes.

In each stage, a node communicates only with one of its d neighbors, one for each stage.

The number of computation stages l instead depends on the FFT decomposition, as previously shown. We must have $l > d$ in order to

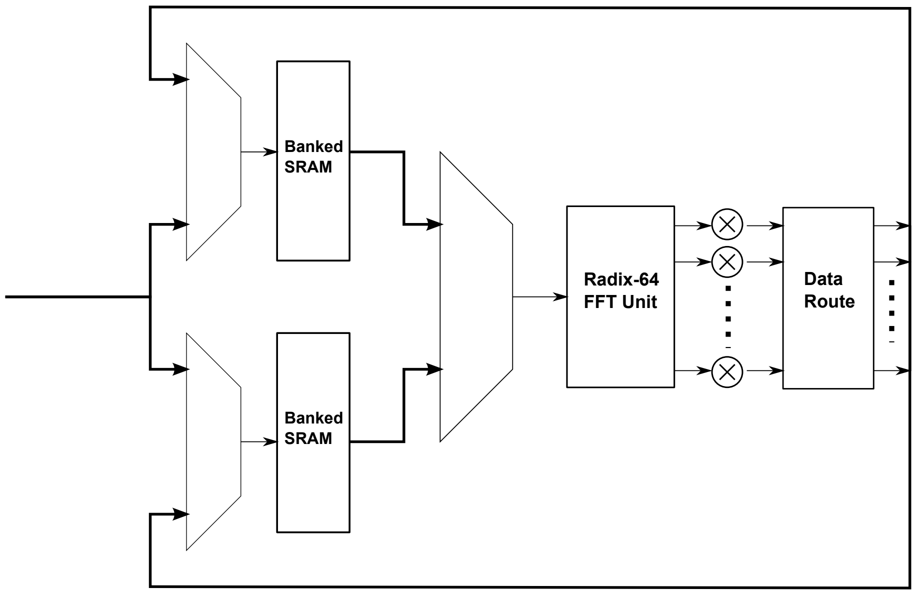


Figure 1.3: Architecture of a 64K FFT processing element.

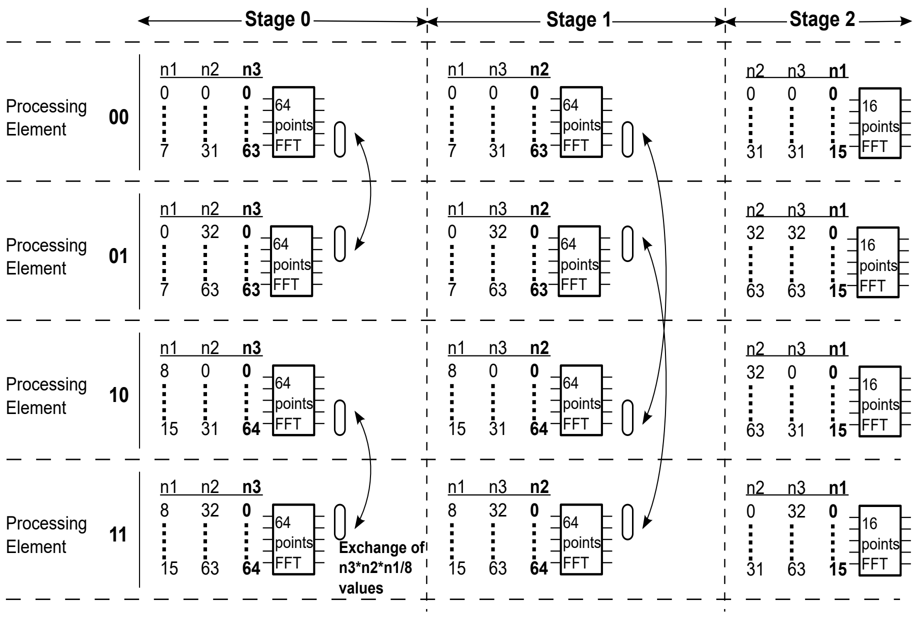


Figure 1.4: Data distribution.

correctly interleave computation and communication. If $l > d+1$, communication takes places only after the first d computation stages while the subsequent stages are computation only. The overall architecture of a single node, which we will call *Processing Element*, is shown in Figure 1.3.

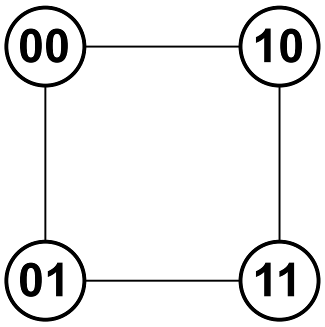


Figure 1.5: Hypercube topology for a system comprising four Processing Elements.

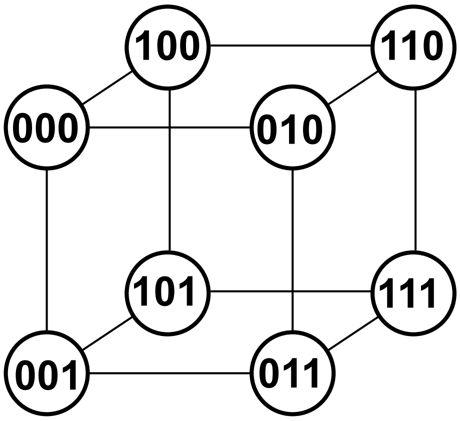


Figure 1.6: Hypercube topology for a system comprising eight Processing Elements.

The core computing element is the Radix-64/16 FFT unit, which computes the basic sub-transforms. Since in our distributed scheme communication will indeed overlap with computing, double buffering is used: while a buffer is feeding current input values, the other one is filled with new values coming partly by the same node and partly from one of its neighbors. At the end of a computation stage, the roles of the buffers are swapped. Buffers are based on a banked architecture which uses the SRAM primitive blocks of the underlying FPGA architecture. Additionally, we also need a group of modular multipliers for twiddle factor multiplications, required between two consecutive FFT computation stages. The data route component is responsible for the proper ordering of FFT output points before writing in the memory buffers.

1.4.1 Data distribution and exchange pattern

Below we consider the computing of a 64K-point FFT with four processing elements. In the initial data distribution phase, the 64K elements vector is partitioned among the four processing elements, considering also the proper decomposition reordering. Then, computing and data exchange stages take place, in a interleaved (but actually overlapped) way. During a computing stage each node can execute in a completely independent way.

We recall also that, according to the previous formula, we can decompose the 64K FFT as per Equation 1.7. Figure 1.4 summarizes the sequence of computing and communication stages: bold style is used to highlight the index (among n_1, n_2, and n_3) involved in the current sub-FFT computing and subsequent data exchange.

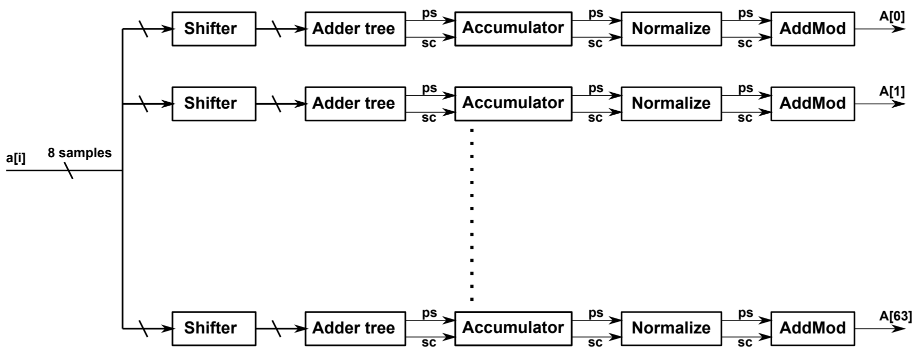


Figure 1.7: Architecture of the baseline Radix-64 unit [1].

1.4.2 FFT-64 unit

The Radix-64 unit (or FFT-64) is the basic building block which is capable of computing the sub-transforms that make up the overall FFT. It can be easily extended for computing also Radix-16 FFT, though this will not be shown here. In the chosen finite field, the 64^{th} root of unity is 8, so multiplications in the FFT formula become just shifts, as follows:

$$A[k] = \sum_{i=0}^{63} a[i]\omega_{64}^{i \cdot k} = \sum_{i=0}^{63} a[i]8^{i \cdot k} \pmod{p} \qquad (1.8)$$

Since $8^{64} \pmod{p} = 2^{192} \pmod{p} = 1$, no intermediate value can exceed 192 bits.

The unit proposed here builds on a baseline scheme [1] shown in Figure 1.7. Input samples are read 8-by-8 and are fed to 64 separated computing chains, one for each *frequency* component. Each chain comprises a shifter bank, where the eight samples are multiplied by their respective twiddle factor. Shifted values are summed by an adder tree to produce a partial sum. To avoid the latency of long carry chains, a carry save solution is adopted. The output is then made up of two vectors, which are not merged until the very last block (*AddMod*). The accumulator will sum up the partial sums in eight consecutive clock cycles. After the eight clock cycles, the transform is complete and the value in the accumulator is modular reduced. The *Normalize* block computes a first coarse reduction by applying Equation 1.9, which applies to 128-bit numbers and exploits the properties of the chosen modulus:

$$a \cdot 2^{96} + b \cdot 2^{64} + c \cdot 2^{32} + d = 2^{32}(b+c) - a - b + d \tag{1.9}$$

The result will require at most one extra addition or subtraction with the module p. This last operation is performed in the *AddMod* component.

The baseline scheme is highly redundant since much work can be shared among the components. Our architecture exploits such redundancy through several structural solutions, which, as will be shown in the implementation details, result in a significant reduction of area usage and higher parallelism.

In order to exploit common work between the FFT components, we apply Equation 1.6 to the 64-point FFT:

$$\begin{aligned}\sum_{i=0}^{63} a_i \omega^{i\cdot k} &= \sum_{j=0}^{7} \left[\left(\sum_{i=0}^{7} a_{i\cdot 8+j} \omega_8^{i\cdot k_1} \right) \cdot \omega_{64}^{j\cdot k_1} \right] \cdot \omega_8^{j\cdot k_2} = \\ &= \sum_{j=0}^{7} \left(\sum_{i=0}^{7} a_{i\cdot 8+j} \omega_8^{i\cdot k_1} \cdot \omega_{64}^{j\cdot k_1} \right) \cdot \omega_8^{j\cdot k_2} \end{aligned} \tag{1.10}$$

The expression between parenthesis is computed by the first stage in Figure 1.8, where eight samples from the memory are shifted and summed; This is done only for eight frequency components (denoted by k_1). Then, such partial sums are multiplied by the twiddle factor $\omega_8^{j\cdot k_2}$ while the external sum is performed by each accumulator. The multiplication by $\omega_8^{j\cdot k_2}$ leads to eight possible shifts, but they can be reduced to four if we consider that one half of the twiddle factors are the opposite of the other half (the partial sum will be subtracted in the accumulator instead of being summed). The four factors needed are then 2^0, 2^{24}, 2^{48}, and 2^{72} (respectively, no shift, shift by 24, 48, and 72 bits). Accumulators can be thought of as being partitioned into eight blocks of eight accumulators (block number corresponds to k_2 in Equation 1.10). Each block contains a multiplexer selecting which of the four shifts is needed, according to the block number and the current computing step

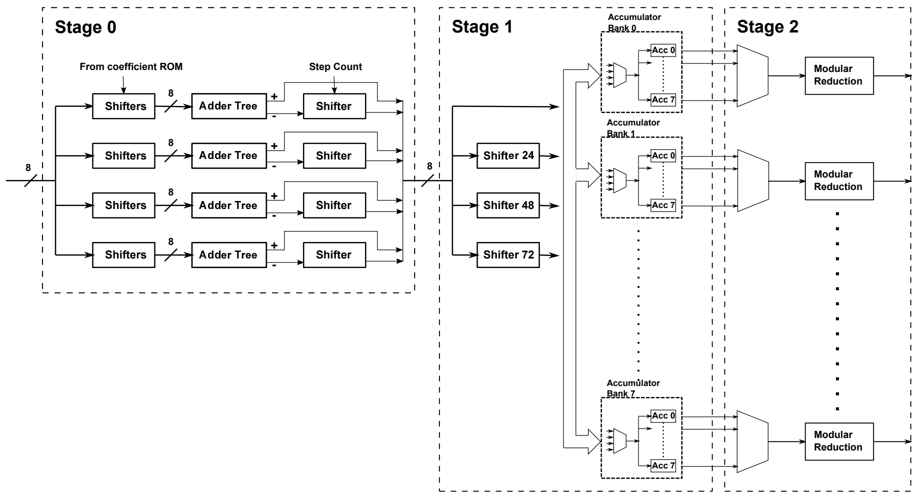


Figure 1.8: Architecture of FFT64 unit.

(respectively, index k_2 and j). Each block receives also a *subtract* signal (not shown in figure).

The first stage is itself optimized by computing only four of the eight components using the relation:

$$\sum_{i=0}^{7} a_{i\cdot 8+j}\omega_8^{i\cdot k_1}\cdot\omega_{64}^{j\cdot k_1} = \sum_{i=0}^{7} a_{i\cdot 8+j}\omega_8^{i\cdot(k_1'+4)}\cdot\omega_{64}^{j\cdot(k_1'+4)} =$$

$$= \sum_{i=0}^{7} a_{i\cdot 8+j}\omega_8^{i\cdot k_1'}\cdot\omega_2^{i}\cdot\omega_{64}^{j\cdot k_1'}\cdot\omega_{16}^{j} \text{ for } k=4,5,6,7$$

We can see that components 4 to 7 can be computed similarly to the first four, except for the multiplication by a factor ω_{16}^{j} and the fact the in the summation odd terms are taken with negative sign. This is done by modifying the adder tree so that it outputs also the difference between the sums of even and odd terms (such modification adds very little complexity to the adder tree).

After eight computing steps, the accumulators contain the FFT output which needs to be modular reduced. While the baseline scheme uses 64 modular reduction components, one for each accumulator, we observe that the maximum average throughput, even in a fully pipelined solution, is eight components per clock cycle. Consequently, we use only eight modular reductors, one for each accumulator block, preceded by a multiplexer which switches to a different component at each clock cycle. So we yield exactly eight frequency components for each clock cycle. Our solution has a twofold advantage: First, it reduces the area occupancy of the FFT64 unit and the memory parallelism required (eight words vs. 64). Second, it realizes part of the work of the Data Route component, since at every clock cycle we produce eight values which are appropriately spaced out for memory writing.

We also identified a couple of minor optimizations. We merged carry-save vectors immediately after the adder tree, reducing area usage. The carry propagation latency penalty can be mitigated by adding a pipeline stage. Furthermore, before Stage 1, we reduce the bit-width of each value by applying Equation 1.9. This further reduces the area, particularly routing resource usage.

Last, we recall that the FFT-64 unit can be adapted, with minor modifications, to compute also Radix-8, Radix-16, and Radix-32 FFTs. This gives us greater flexibility in choosing an FFT order other than 64K.

1.4.3 Internal Bank Memory

Our internal memory needs to support the specific FFT memory access pattern yet guarantee an appropriate degree of parallelism. A simple *linear* bank memory ensures parallel read accesses (with consecutive words in a row mapped to different banks) but it would cause write accesses to collide on the same bank. These considerations led us to adopt a two-dimensional bank scheme as in Figure 1.9. Each square is a basic memory block, which is dual port SRAM, with a depth of 256 words and word-width of 64 bits implemented as native FPGA memory blocks (namely, two Altera M20K hard core blocks).

A 4x4 array of basic memory blocks gives a size of 256Kb which can hold a vector of 4096 points. The scheme in the figure considers only one of the dual port of a basic block, for visual clarity. Read access is column-wise, write access row-wise. Access parallelism is eight words per clock cycle, either during reading or writing.

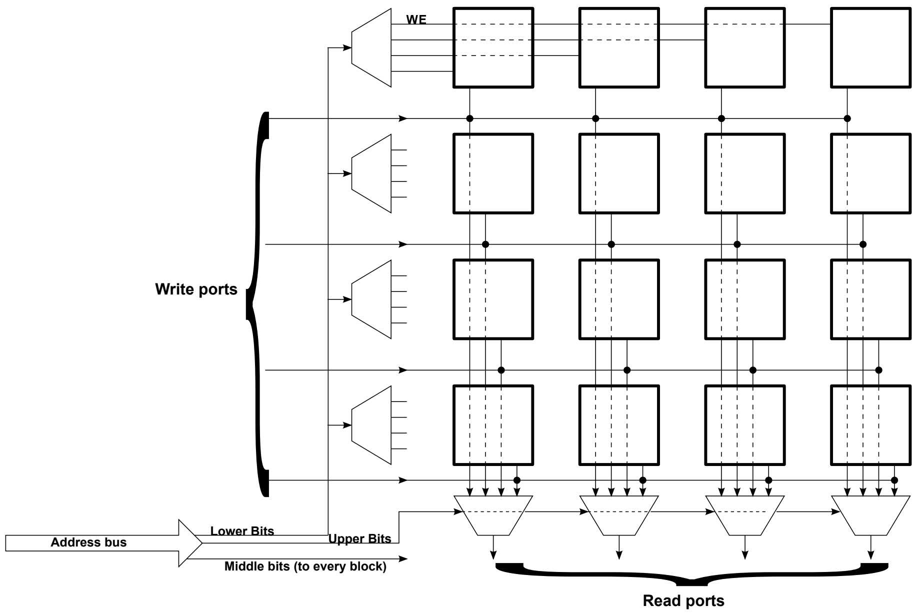


Figure 1.9: Architecture of banked memory buffer.

1.4.4 Modular multiplier

The output points of inner FFTs have to be multiplied by appropriate twiddle factors before they can be used by the external FFT. We chose to use DSP blocks for greater efficiency in terms of area and speed. To realize 64x64 multiplication we can split our operands in 32-bit components and use a basic 32x32-bit DSP multiplier, which requires only two DSP blocks. Using school-book multiplication, four 32x32-bit multipliers are needed; partial products are then summed and modular reduced using Equation 1.9.

1.4.5 Data route

The job of this component is to properly order the output points coming from the modular multipliers, ensuring their correct writing in memory as well as computing the correct addresses according to the current computation step. As mentioned earlier, the complexity of this component is greatly reduced since part of its job is performed by the FFT-64 unit. In fact, it is just a memory address generator.

1.5 Implementation and performance

To implement the proposed accelerator, we targeted a Stratix V 5SGSMD8N3F45I4 FPGA, as in [1], using VHDL as the design entry language. Most of the implementation effort was put on the Radix-64 unit, the banked memory, and modular multipliers. All of them have been adeguately tested and work as expected. By careful pipelining some subsystems, the components could be synthesized with an operating frequency of 200 MHz.

Below we present a performance estimate, first addressing the FFT and then the SSA multiplication. The FFT-64 unit is able to output an FFT every eight clock cycles, while a FFT-16 will take two clock cycles. A single 64K points FFT will take:

$$T_{FFT} = 2 \cdot (T_C \cdot 8 \cdot 1024)/P + (T_C \cdot 2) \cdot 4096/P$$

where T_C is the clock period, i.e. 5 ns, while P is the number of Processing Elements (here, four). The first term refers to the first two stages, with 1024 FFTs on 64 points. The second term refers to the last stage where 4096 FFTs on 16 points are computed.

By replacing the clock period and using four Processing Elements, we get

$$T_{FFT} = 20480ns + 10240ns \approx 30.7\mu s$$

A full SSA multiplication requires three FFTs (two direct FFTs for the inputs and one inverse FFT for the output). Besides we must perform a component-wise multiplication on two vector of 64K components and the final carry recovery addition on the inverse FFT components. The remaining resources can accommodate at least 32 additional modular multipliers for component-wise multiplication, yielding:

$$T_{DOTPROD} = T_C \cdot 65536/32 \approx 10.2\mu s$$

The final carry recovery can be efficiently computed with an *ad hoc* adder structure, which is th object of another research work. Its maximum delay is approximately $20\mu s$. So the overall time for a complete SSA multiplication is $\approx 122\mu s$.

Table 1.4 summarizes the resource usage and performance comparison between the proposed solution and [1]. Notice that [1] only presents quantitative results for the FFT operation. For our comparison, we conservatively assumed a zero difference for the remaining dot-product and carry recovery operations. Overall, the combination of the optimizations presented above resulted in a 34.4% saving in terms of hardware resources as well as a 332% gain in terms of execution speed. Further performance gain could be achieved by allocating more computing nodes, by using the saved area, but we aim to exploit such resources for implementing the above mentioned carry recovery block. Finally, table 1.5 reports a performance comparison between our solution, [1], [asic] (which is a 90nm ASIC solution), [gpu12] and [gpu13] which are based on NVIDIA C2050 GPU.

	Proposed here	[1]
ALMs	104000 (40%)	231000 (88%)
Registers	116000 (11%)	336377 (31%)
DSP blocks	256 (13%)	720 (37%)
M20K SRAM	8 Mbit (20%)	–

Table 1.4: Comparison of resource usage.

	Proposed here	[1]	[asic]	[gpu12]	[gpu13]
FFT(μs)	30.7	125	–	250	–
Multiplication(μs)	122	405	206	765	583

Table 1.5: Comparison of excution time.

Chapter 2

GPU extension for Montgomery Multiplication

Modular multiplication is the fundamental operation of most public-key cryptosystems, both those based on exponentiation (RSA, Diffie-Hellman, El Gamal) as well as Elliptic Curve schemes based on prime Finite Fields $\mathbb{F}_p$. Modular reduction, part of modular multiplication, is the most demanding in term of execution time and, if performed in software using classical integer division, would make public-key cryptography impractical, given the large operands involved (e.g., 224 bits for ECC schemes, 2048 bits for RSA).

2.1 Modular Multiplication

2.1.1 Montgomery modular reduction and multiplication

In his breakthrough paper [47], Peter Montgomery introduced a novel and efficient approach to perform modular reduction without any trial division. It consists in computing $P \cdot R^{-1} \bmod M$ which can be much faster than computing $P \bmod M$. Factor R is called *Montgomery radix* and must satisfy $R > M$ and $\gcd(R, M) = 1$. The Montgomery reduction algorithm is reported in Algorithm 1. Montgomery reduction can be extended to include a multiplication by computing $A \cdot B \cdot R^{-1} \bmod M$, where A and B are the multiplicands. Given the condition $P \leq (RM - 1)$, Montgomery reduction ensures that the result is less than $2M$, therefore an additional subtraction may be needed to obtain the reduced result. R is commonly chosen to be a power of two to speedup computation, since reduction modulo M is replaced by reduction and division by R. Since Montgomery reduction/multiplication

Algorithm 1 Montgomery reduction.

```
Require:
    modulus $M$
    auxiliary modulus $R$, such that $\gcd(M, R) = 1$ and $M < R$
    positive integer $P$ such that $P < M^2$
    pre-computed constant $M' \equiv -M^{-1} \bmod R$ such that $M' \in [0, R)$
Ensure: $Z = P \cdot R^{-1} \bmod M$, such that $Z \in [0, M)$.
1: $Q \leftarrow (P \bmod R) \cdot M' \bmod R$
2: $T \leftarrow (M \cdot Q + P)/R$
3: if $T \geq M$ then
4:     $Z \leftarrow T - M$
5: else
6:     $Z \leftarrow T$
7: end if
```

doesn't compute directly a modular reduction/multiplication, in order to obtain the latter the input operands to be in a proper representation, the so called *Montgomery representation*. Given operands A and B, their Montgomery representation is computed as $A' \equiv A \cdot R \bmod M$ and $B' \equiv B \cdot R \bmod M$. By taking the canonical representatives of such residue classes (that is, the remainders modulo M) we also obtain that $A', B' \in [0, M)$. The sum of two numbers in Montgomery representation is still in Montgomery representation, since

$$A \cdot R \bmod M + B \cdot R \bmod M \equiv (A + B) \cdot R \bmod M \tag{2.1}$$

This is not true for multiplication since

$$(A \cdot R \bmod M) \cdot (B \cdot R \bmod M) \equiv A \cdot B \cdot R^2 \bmod M \tag{2.2}$$

Anyway, if use Montgomery multiplication, we get

$$\begin{aligned} A' \cdot B' \cdot R^{-1} \bmod M = (A{\cdot}R \bmod M) \cdot (B{\cdot}R \bmod M) \cdot R^{-1} \bmod M = \\ = A{\cdot}B{\cdot}R^2{\cdot}R^{-1} \bmod M = A{\cdot}B{\cdot}R \bmod M \end{aligned} \tag{2.3}$$

which is now in the correct representation.

Given a function *MONTMUL* to compute the Montgomery multiplication, it can be used also to compute the Montgomery representation of an operand the following way

$$\text{MONTMUL}(A, R^2) = A{\cdot}R^2{\cdot}R^{-1} \bmod M = A{\cdot}R \bmod M \tag{2.4}$$

Likewise, it can be used to convert back from Montgomery to ordinary representation

$$\text{MONTMUL}(A', 1) = A{\cdot}R{\cdot}R^{-1} \bmod M = A \bmod M \tag{2.5}$$

In software implementations, large operands are usually decomposed into a sequence of w-bit words. The above *bulk* version of Montgomery reduction/multiplication can be adapted to work word-wise, as shown in Algorithm 2. In this case, the radix R is chosen to be 2^w and reduction is applied progressively starting from the least significant word of P.

2.1.2 Barrett modular multiplication

For completeness sake, we preset another important algorithm for modular reduction. Paul D. Barrett introduced in 1986 a novel technique in order to speed-up the implementation of the RSA encryption algorithm on an *off-the-shelf* digital signal processor (in the original article, results for Texas Instrument's TMS 32010 were reported). Algorithm 3 reports the algorithm scheme. The general idea is that in order to obtain the remainder modulo p, we must subtract from input x a proper multiple of the modulo itself, namely $\sigma \cdot p$. Barrett's algorithm gives a way to compute the approximate quotient σ.

2.1.3 Camparison with Montgomery reduction

The Barrett and Montgomery reduction algorithms are both useful to implement a faster modular reductions. They share some similarities

- thier input operand range is $[0, M^2)$
- they are based on the use of precomputed constants ($M' \equiv -M^{-1} \bmod R$ for Montgomery and $\lfloor \frac{2^{2k}}{M} \rfloor$ for Barrett)
- they can be combined with multiplication, since both require $0 \leq a, b < n$, so we have $0 \leq ab < n^2$

Algorithm 2 Montgomery reduction (word-wise version).

```
Require:
    odd modulus $M = (m_{s-1}, \ldots, m_1, m_0)$
    auxiliary modulus $R$, power of 2 such that $R > M$
    operands $P = (p_{2s-1}, \ldots, p_1, p_0)$ such that $P < M^2$
    pre-computed constant $m' \equiv -m_0^{-1} \bmod 2^w$, such that $m' \in [0, 2^w)$
    where $s = \lceil n/w \rceil$, $n$ is the bit length of $M$, $w$ is the word length.
Ensure: $Z = P \cdot R^{-1} \bmod M$.
 1: $t \leftarrow 0$
 2: for $i \leftarrow 0 \,..\, s-1$ do
 3:     $q \leftarrow p_i \cdot m' \bmod 2^w$
 4:     $h \leftarrow 0$
 5:     for $j \leftarrow 0 \,..\, s-1$ do
 6:         $(h, l) \leftarrow m_j \cdot q + p_{i+j} + h$
 7:         $p_{i+j} \leftarrow l$
 8:     end for
 9:     $(h, l) \leftarrow p_{i+s} + h + t$
10:     $p_{i+s} \leftarrow l$
11:     $t \leftarrow h$
12: end for
13: for $i \leftarrow 0 \,..\, s-1$ do
14:     $z_i \leftarrow p_{i+s}$
15: end for
16: $z_s \leftarrow t$
17: if $Z \geq M$ then
18:     $Z \leftarrow Z - M$
19: end if
```

Algorithm 3 Barrett modular reduction [48]

```
Require: $x$ ($2m$ bits), modulus $p$ ($m$ bits), with $x < p^2$, precomputed
    constant $p' = \lfloor \frac{2^{2k}}{p} \rfloor$, with $k \in \mathbb{N} : p < 2^k$ (e.g., $k = \lceil log_2 p \rceil$).
Ensure: $y \equiv x \bmod p$, $0 \leq y < p$.
1: $\sigma \leftarrow \lfloor \frac{x \cdot p'}{2^{2k}} \rfloor$
2: $t \leftarrow x - \sigma \cdot p$
3: if $t \geq p$ then
4:     $y \leftarrow t - p$
5: else
6:     $y \leftarrow t$
7: end if
```

- both require an optional final subtraction in order to get the correct result
- they perform 2 internal multiplications per reduction.

Among the differences there is the fact that Montgomery reduction requires numbers to be converted into and out of *Montgomery representation* (which, as we have seen, can be done using the very same Montgomery multiplication); Barrett reduction, instead, operates on regular numbers directly. On the other side, for a modulus of k bits, Montgomery internally performs two k-by-k bit multiplications yielding a $2k$ bit result, while Barrett internally performs a $2k$-by-k bit multiplication yielding $3k$ bits, plus a k-by-k bit multiplication yielding $2k$ bits. Therefore, Barrett algorithm is computationally a bit more expensive. For these reasons, Montgomery multiplication is more suitable for modular exponentiation, with several multiplication using the same modulus.

Finally, Montgomery is based on modular arithmetic and exact division, whereas Barrett is based on approximating the real reciprocal of the modulus with bounded precision.

2.2 Previous works

In the years, several efforts have been made in order to efficiently implement Montgomery multiplication on different platforms. The authors in [49] analyze the problems posed by resource constrained environments and propose an implementation for an 8-bit AVR microcontroller. Other researchers proposed solutions based on completely custom multipliers implemented on FPGA technology, such as the work by Yang et al. [50] for Xilinx Virtex-6 FPGAs. Massolino et al. [51] target instead low-power FPGA devices, specifically Microsemi IGLOO-2. In the recent years, efforts have been focused on exploiting data level parallelism available with modern CPU SIMD extensions or in general-purpose GPUs. Seo et al. [52] propose a highly optmized algorithm targeted at the ARM NEON pipeline. Zheng et al. [53] instead exploit the high performance floating-point capabilities of NVIDIA GeForce GTX TITAN GPUs.

2.3 From the algorithm to the GPU-like datapath

This section gives a thorough description of the approach taken to customize the GPU datapath. Figure 2.6 shows the baseline core architecture of the GPU-like processor provided by the MANGO H2020 project. Only the functional units of interest for the algorithm implementation are shown. In order to extend the core to support Montgomery multiplication, we mostly focused on the existing integer MAC pipeline and the surrounding hardware used to store and transfer data. The first key choice to make, of course, concerns the specific algorithm used to implement Montgomery multiplication, so as to maximize the match with the hardware mechanisms exposed in the GPU-like datapath.

2.3.1 The *Nu+* GP-GPU datapath

Before proceeding to describe the cryptographic extension for our GPU-like, we'll first outline the main characteristics of the *Nu+* core. The overall architecture is shown in Figure 2.1. The core is based on a RISC architecture with in-order instruction execution. Control logic is kept as much simple as possible. The architecture masks memory and operation latencies by heavily relying on hardware multithreading. These two points represent the essence of the GP-GPU approach, where it is desirable to dedicate most of the hardware resource to funtional units rather than control logic. This way, a higher degree of parallelism, in terms of lanes and threads, is available for accelerating highly data-parallel kernels. We precise here that by thread we mean actually a

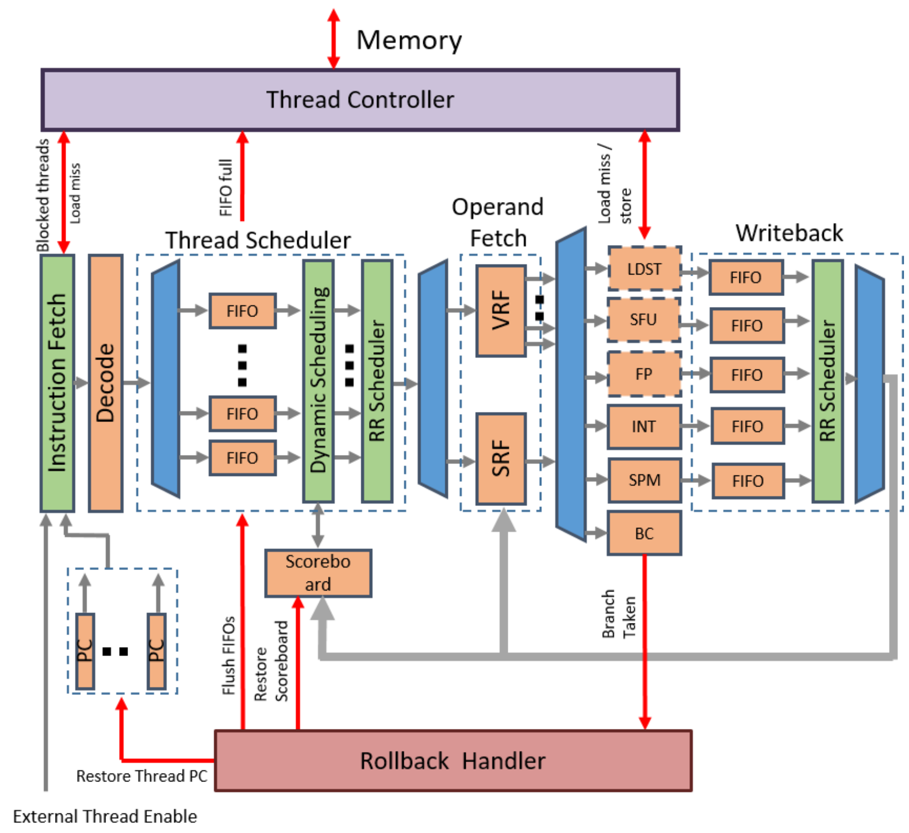


Figure 2.1: Overall view of *Nu+* datapath

vector or SIMD thread, which would corresponds to a *thread warp* in the NVIDIA CUDA technology. In *Nu+*, each hardware thread has its own PC, register file, and control registers. The number of threads is user configurable.

All threads share the same functional units; each execution pipeline comprise several hardware lanes, with each scalar unit (e.g. and adder or multiplier) replicated per each lane. Each thread can then execute SIMD instructions, computing, for exemple, N additions in parallel and simultaneously, subject to data independence in the operands. Beside a memory hierarchy, comprising multi-level caching and coherence protocols, the core supports also a high-throughput non-coherent *Scratchpad* memory (SPM), corresponding to the shared memory in the NVIDIA terminology. *Nu+* is intended to be an highly customizable core, to be adapted to different class of applications. The following are some of the configurable parameters

- Number of cores
- Number of threads
- Number of lanes
- Number of registers per thread
- Cache set-size
- Cache associativity degree
- Cache line size
- SPM bank count

The core supports predicated execution, that is, each SIMD instruction can have an execution mask which indicate for each lane if the corresponding result must be stored in the VRF. Such mask is contained in a scalar register of the SRF; it is usually set by using comparison instructions, but manually writing is howerver possible. Execution mask allows to simulate the execution of multiple scalar threads in parallel mapping each of them to a lane (in this case, a scalar thread corresponds to a CUDA thread using NVIDIA terminology. Divergence points, such as *if-then-else* statements, may make lockstep execution impossible, since some threads could follow the *then* brach, while others the *else*. By using predicated execution, the instructions of both branches are always executed, but their execution is conditioned by to complimentary masks, which correspond to the sets of threads for which the *if* condition holds respectively *true* and *false*. Such behaviour is exemplified in Figure 2.2. Masks can obviously used for other purposes, but the example above is the typical of a high level language compiler (such as C) when used to parallelize scalar threads.

The *Nu+* core and corresponding many-core architecture is currently emulated using an FPGA-based platform. Reconfigurable hardware can be considered both as a possible platform for actual deployment, which useful whenever reconfigurability for different applications is desirable, but, more generally, as a prototyping tool to support design and test of

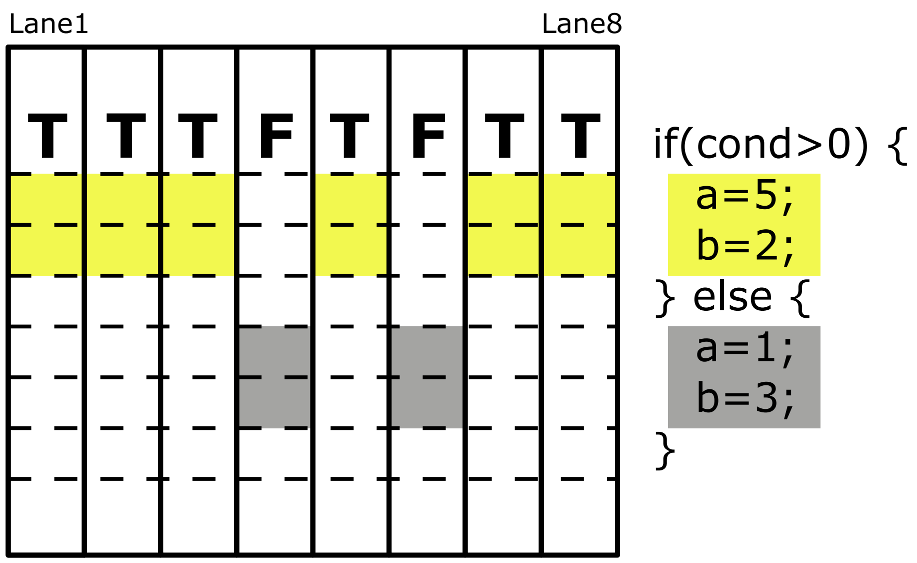


Figure 2.2: Predicated execution using masks.

extensions and optimizations for various classes of applications. In the latter case, the architecture could also be implemented as ASIC, if the trade-off cost-performance is acceptable.

In the following, we briefly sketch out the most significant *Nu+* core pipeline units.

Instruction Fetch stage Instruction Fetch stage schedules the next thread PC and load the corresponding instruction. The current thread is selected from a pool of eligible threads by the Thread Controller. Available threads are scheduled in a Round Robin fashion. At boot time, the Thread Controller can initialize each thread PC through a dedicated interface.

This stage contains an instruction cache, which is set associative and has two stages. Once an eligible thread is selected, the unit computes its next PC, and determines if the next instruction cache line is already in the instruction cache memory. If not, an instruction memory transaction is issued to the Network Interface and the thread is blocked until the instruction line is not retrieved from main memory. Finally, this module handles the PC restoring in case of rollback. When a rollback occurs and the rollback signals are set by Rollback Handler stage, the Instruction

Fetch module restore the PC of the thread that issued the rollback with a previous value.

Instruction Scheduler stage After decoding, instructions are stored in FIFOs, one for each thread. The Dynamic Scheduler checks data hazard and decides which thread can be scheduled and issued to the Operand Fetch. At this stage only data hazards and some structural hazards (such as those for the FPU) are checked, other structural hazards are checked on-the-fly in the Writeback stage. The Dynamic Scheduler relies on a light scoreboarding system; based on the scoreboard status and on the instructions which are currently at the top of the non empty FIFOs, the Dynamic Scheduler decides which threads are able to be scheduled, and, among them, a thread is selected using a round-robin arbiter. Then, the the chosen decoded instruction is issued to the Operand Fetch and the related thread scoreboard is updated accordingly. Each scoreboard keeps track of the destination registers with currently pending (not completed) instructions. When an instruction completes, the corresponding destination register is released by the Writeback stage by clearing the corresponding bit in the scoreboard. If a rollback occurs, the Dynamic Scheduler flushes the FIFO of the thread which issued the rollback, and restores its scoreboard to the previous state.

Operand Fetch stage Operand Fetch prepares operands for the Execution pipelines (FPU, ALU, etc.). This stage contains two register files: a scalar register file (SRF) and a vector register file (VRF). A SRF register size is word size (typically, 32 bits), a VRF register size is equal to the scalar size multiplied by the number of lanes. Each thread has its own pair of SRF and VRF. There are two substages of which the first is the actual register file read access for the two input operands, while the latter computes the actual values for operands, for example, for a memory access instruction it computes the effective memory address adding the base address from a register with an immediate offset.

Some registers of the SRF are reserved for special register which contain thread-level status, such as the Program Counter, the Stack Pointer and the Mask register. Actually the PC is not stored in a real register, but contained in the interface registers between *Nu+* pipeline stage. The Operand Fetch possesses also a write interface receiving signals from the Writeback stage. In case of vectorial operation, a signal vector containing the lane mask indicates which lane is affected by the current operation.

When a masked instruction is issued, the mask register itself is read during Operand Fetch stage. When one of the operand is immediate and the instruction is vectorial, its value is replied on each vector element.

Writeback stage The Writeback stage manage the writing of outcoming results from the execution pipelines into the register files. The execution pipelines have different lengths and latencies (memory latencies are even not known at compile time), so instructions issued in different cycles could arrive at the Writeback in the same clock cycle. The Writeback module avoids structural hazards on-the-fly, using a set of queues, one for each execution pipeline: in each queue, the corresponding result is stored, an arbiter selects one result in a round-robin fashion and issue the destination register writing to the Operand Fetch stage. Each queue stores all the information needed for a writeback operation, such as destination register and write mask.

Rollback handler Rollback Handler restores PCs and scoreboards of the thread that issued a rollback. A typical case of rollback is a taken jump or a trap; in this case, the Brach module in the Execution pipeline issues a rollback request to this stage, and passes to the Rollback Handler the thread ID that issued the rollback, the old scoreboard and the PC to restore. Furthermore, the Rollback Handler flushes all issued and the queued requests of this thread still in the pipeline.

Thread controller Thread Controller handles the pool of executable thread. This module blocks threads that cannot proceed due cache misses or hazards. Also, Thread Controller handles threads wake-up when the blocking conditions are removed.

Furthermore, the Thread Controller interfaces the Instruction Cache with the main memory, since the architecture supports only one level of caching for instructions. In other words, when an instruction cache miss occurs, the data is retrieved directly from the main memory.

2.3.2 Algorithm *GPU-ization*

Different choices are available to implement Montgomery multiplication. Firstly, one can perform multiplication and Montgomery reduction separately or in an interleaved fashion. The former allows the adoption of more advanced multiplication algorithms, such as the Karatsuba decomposition, while the latter allows an optimized overlapping of the two operations, yielding a computation time lower than the sum of the two separate computations. When performing a school-book multiplication, based on two nested loops, one can essentially take two main approaches:

- *Operand Scanning*: operand words are read sequentially, in increasing order of weight, from the least to the most significant one. The operand scanned by the inner loop is obviously scanned multiple times.
- *Product Scanning*: one word of the final result is considered at a time and is stored only once. The approach requires less temporary memory and fits well in resource constrained embedded systems. On the other hand, because of the algorithm asymmetry, it is less susceptible to optimizations and parallelization.

The *Operand Scanning* approach can be easily integrated with Montgomery reduction, allowing two main options [49]:

- *Coarsely Integrated Operand Scanning* (CIOS): the outer loop contains two inner loops, the first one computing the current partial product (as in the normal multiplication algorithm) $\bar{p} = \bar{a} \cdot b_i$. Then the second inner loop applies the Montgomery reduction to the partial product.
- *Finely Integrated Operand Scanning* (FIOS): Montgomery reduction is integrated in the multiplication inner loop, i.e. each partial product component is reduced as soon as it is computed. The approach is taken by Algorithm 4 below.

For the customized GPU-like processor, we focused our efforts on the FIOS algorithm which, in general, offers more opportunities of optimization and fits our main goal since it concentrates all the computation in the inner loop. We recall from the FIOS algorithm that in the inner loop, the very first iteration differs from the subsequent ones only in that it computes the quotient q, while all other operations are the same. Figure 2.3 shows a schematic view of the first iteration of the inner loop, while Figure 2.4 show the subsequent iteration. Figure 2.5,

Algorithm 4 FIOS Montgomery multiplication.

```
1: $P \leftarrow 0$
2: **for** $i \leftarrow 0$ .. $M - 1$ **do**
3:     $(t'_h, t_l) \leftarrow a_0 \cdot b_i + p[0]$
4:     $q \leftarrow t_l \cdot m'_0 \bmod 2^w$
5:     $(t''_h, t_l) \leftarrow m[0] \cdot q + t_l$
6:     **for** $j \leftarrow 1$ .. $M - 1$ **do**
7:         $(t'_h, t_l) \leftarrow a[j] \cdot b[i] + t'_h + t''_h$
8:         $(t''_h, t_l) \leftarrow m[j] \cdot q + p[j] + t_l$
9:         $p[j - 1] \leftarrow t_l$
10:    **end for**
11:    $(t'_h, t_l) \leftarrow p[M] + t'_h + t''_h$
12:    $p[s - 1] \leftarrow t_l$
13:    $p[s] \leftarrow t'_h$
14: **end for**
15: **if** $p \geq m$ **then**
16:     $p \leftarrow p - m$
17: **end if**
```

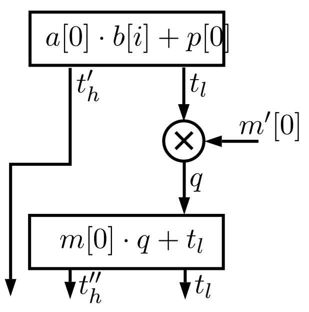


Figure 2.3: Inner loop: first iteration

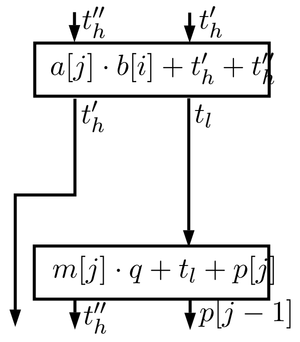


Figure 2.4: Inner loop: other iterations

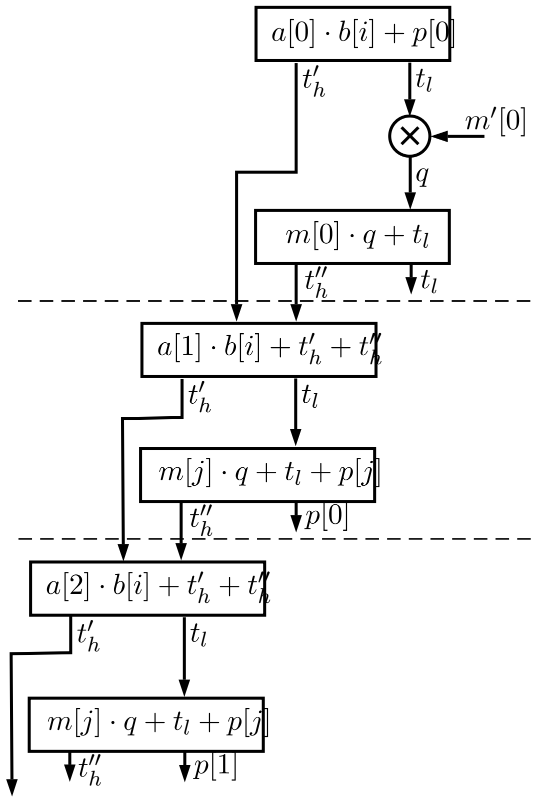


Figure 2.5: Inner loop iterations

instead, highlights the close data dependencies between iterations of the

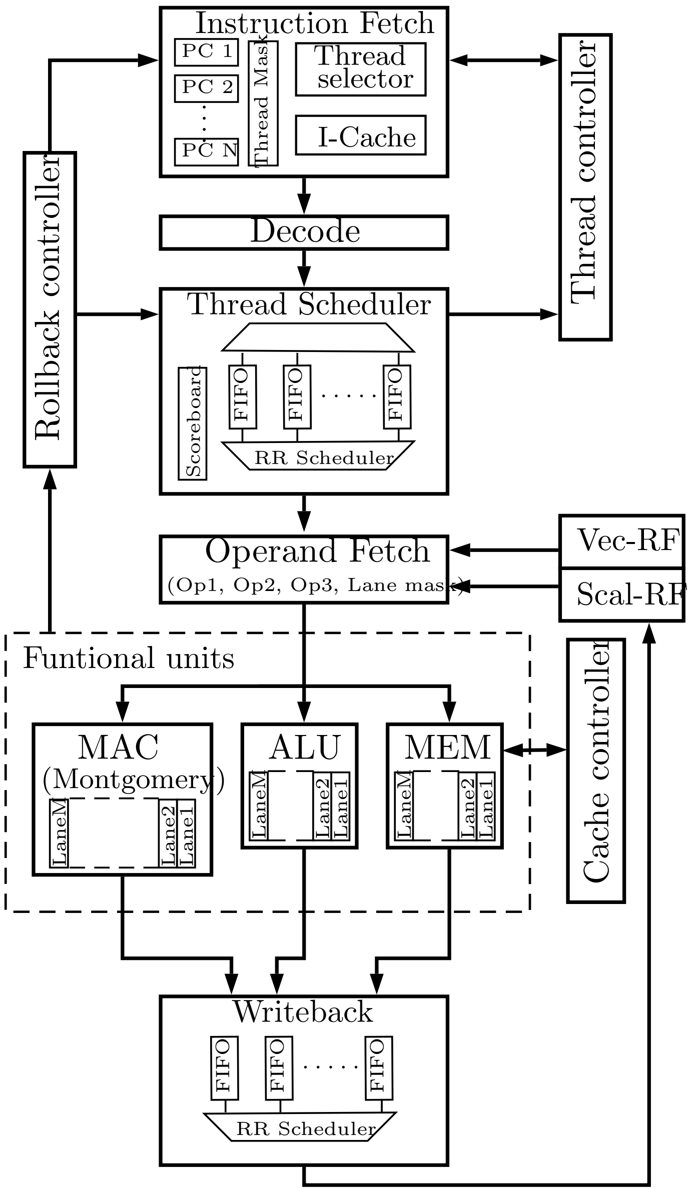


Figure 2.6: GPU-like pipeline. Only the functional units of interest are shown.

inner loop.

Besides the selection of the algorithm, therefore, we needed to address a number of additional design choices to meet our objective of providing acceleration support for Montgomery multiplication in the GPU datapath. These included:

- Data dependencies and access patterns;
- Data movement and bandwidth;
- Width and number of lanes;
- Predicated execution;
- Integration of the Montgomery unit;
- Definition of an extended Instruction Set.

These considerations led us to the extended SIMD MAC unit shown in Figure 2.7.

2.3.3 Data dependencies and access patterns

In order to parallelize the internal loop of the algorithm, we needed to remove the data dependencies between subsequent iterations. Such dependencies consist in vectors H' and H''. We decided to postpone their sum to the beginning of the next iteration. This does not affect the correct operation of the algorithm. As shown in Figure 2.7, vectors H' and H'' are stored and internally propagated to be summed in the next Montgomery instruction. In the actual pipeline implementation, H' and H'' are generated in a way enabling the next instruction to be issued immediately rather than wait for the previous one to complete.

2.3.4 Data movement and bandwidth

Our extended MAC unit requires that several input operands produce several results, which easily exceed the data bandwidth available in our GPU core. To balance the data movement across GPU-like datapath, we chose to partition input/output operands into two groups, one stored in the regular SIMD register file, the other half stored in registers internal to the MAC unit. Such partitioning is shown in Table 2.1.

Table 2.1: Operand type and location.

Operand	Type	Storage
a	Vector	External
b	Scalar	External
m	Vector	Internal
p	Vector	External
m'	Scalar	Internal
H'	Vector	Internal
H''	Vector	Internal

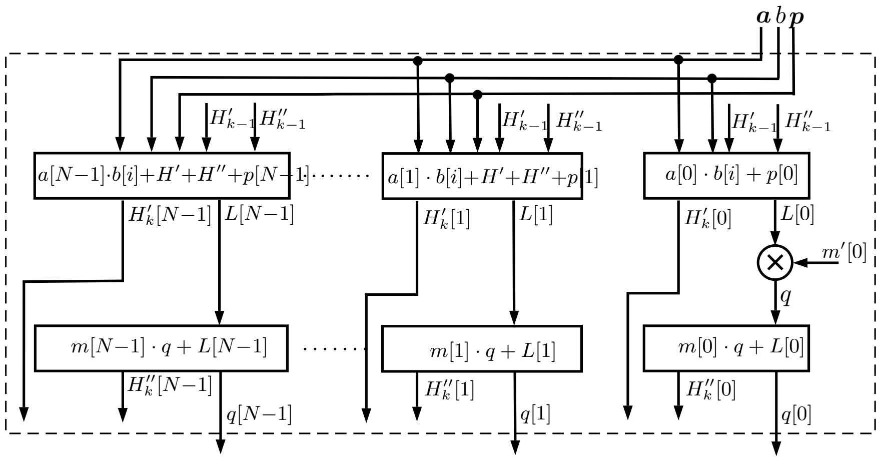


Figure 2.7: Montgomery unit architecture.

We modified our GPU pipeline, in particular the *Operand Fetch* and *Writeback* stages, in order to be able to read three operands and write two results. Few modifications were needed since we already had two separated vector and scalar register files, which makes it feasible to read from two vector ports and one scalar port at a time (the second scalar port is reserved for reading the execution mask). For the same reason, we can write a vector and a scalar at the same time. Since our *register-register* instruction format only supports two source registers and one destination, the additional source and destination operand is an implicit scalar register.

2.3.5 Width and number of lanes

Although GPUs are basically SIMD processors, they usually lack the ability to configure the number of lanes and word width (e.g. 4×32 bits or 2×16 bits) as in CPU SIMD extensions. In GPUs, there is usually a fixed number of lanes with a fixed word length (32 or 64 bits). In Montgomery multiplication software implementations, a major concern is the choice of the component bit width and how it correlates to the processor word width. This choice is important to accomodate for high order components generated by multiplications and carry/borrow values generated by additions/subtractions. We chose to keep a fixed word width and manage high-order components generated by MAC instructions by storing them in internal registers. For additions/subtractions, we introduced *ad hoc* instructions for extended arithmetic. (More details are given below.)

2.3.6 Predicated execution

Having a fixed number of lanes makes it possible to have predicated execution at a lane level, by using lane masks. This is very common in modern GPUs, since it allows emulating the execution of scalar threads by mapping each of them to a lane and creating a SIMD thread (see NVIDIA CUDA threads). In graphics applications, each scalar thread is normally a pixel or vertex shader. Lane masks allow, for example, emulating divergence among threads in *if-else* constructs. In our GPU-like core, the lane mask is stored in an implicit fixed scalar register and each vector instruction specifies if it is masked or not. An example of this masked execution is showed in Figure 2.2.

We use masking to access specific lane subsets, especially during *Carry Recovery* and *Final Subtraction*.

2.3.7 Integration of the Montgomery unit

The MAC unit extended for Montgomery multiplication is made up of three macro-stages: the first one computes the current partial product, summing also vectors H' and H'' from the previous iteration. This stage is made up of an array of extended Multiply-and-Add units, each capable of computing the function $res = a \cdot b + c + d + e$. The second stage involves only computing q using a half-precision multiplier on the first lane, while the other lanes have just delay registers to keep results aligned. The last stage is an array of Multiply-and-Add units, which apply the reduction formula $res = q \cdot m + l$ in every lane, where l is the lower order component from the first stage. Figure 2.7 summarizes the above description, highlighting the unit external interface with the GPU-like pipeline, namely the arrows crossing the dashed box. The unit has a latency of 13 clock cycles and is perfectly pipelined.

2.3.8 Definition of an extended Instruction Set

All the above features must be exposed to the programmers through the processor ISA to let them write the complete Montgomery multiplication code based on the hardware acceleration facility. We therefore extended our GPU ISA with the instructions presented in Table 2.2. In addition to the ones listed in the table, we also make wide use of the SHUFFLE instruction, which was already available in our GPU-like core, as in most GPUs.

2.4 Details of the Montgomery functional unit

2.4.1 Data arrangement

For the correct operation of the algorithm, operand components should be arranged in a proper fashion. Assume we have 256-bit operands and that our SIMD units have four lanes of 32 bits. Then, each operand will have eight components and will occupy two vector registers. Their contents will be denoted by $(3, 2, 1, 0)$ and $(7, 6, 5, 4)$, where each index represents the weight of each component with respect to the Montgomery radix 2^{32}. The first Montgomery operation will involve as operands the vector register $A(3, 2, 1, 0)$ and scalar register $B(0)$, the partial product $P(3, 2, 1, 0)$, as well as $M(3, 2, 1, 0)$. The outputs are lower order $L(3, 2, 1, 0)$ and higher order $H(4, 3, 2, 1)$ components of the updated partial product. The first ones are stored externally in register $P(3, 2, 1, 0)$ while the second ones are stored internally in the unit. The last ones should then be accumulated in the next Montgomery operation, but this operation would involve the second halves of operands A, P, and M, namely components $(7, 6, 5, 4)$, which have incompatible weights w.r.t internal register components. It is therefore convenient to arrange A, P, and M operands in an interleaved fashion, as shown in Figure 2.8. In order to achieve such data arrangement, we make use of the INTLVL and

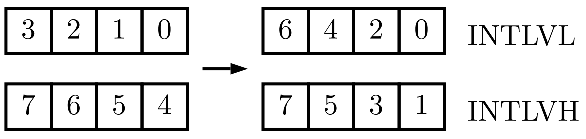


Figure 2.8: Data interleaving using INTLVL and INTLVH instructions (case of operands occupying two registers).

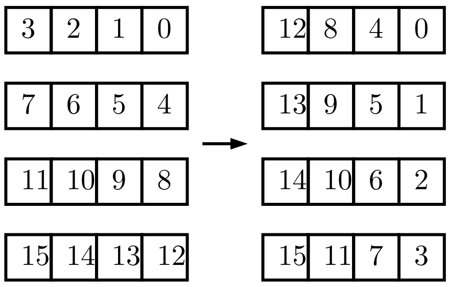


Figure 2.9: Data interleaving using INTLVL and INTLVH instructions (case of operands occupying four registers).

INTLVH instructions, already presented in Table 2.2. The effect of these instructions is shown in Figure 2.8, in the case of each operand occupying two registers. Figure 2.9 instead shows data arrangement for operands occupying four registers. At the end of the algorithm, the result will be in interleaved form, so it will be de-interleaved using DEINTLVL and DEINTLVH instructions.

2.4.2 Carry-less multiplication

This stage is carried out by scanning each component of operand b and computing the corresponding partial product plus Montgomery reduction. Each of such iterations is made up of a sequence of one MONT1 instruction and zero or more MONT2 instructions, according to how many registers each operand occupies. MONT1 and MONT2 instructions require four input operands (A, B, P, and M) but, as mentioned in the previous section, we can supply only three of them. Therefore, before each of these instructions, an internal register is loaded with the current portion of modulus M by using instruction LOADM. Figure 2.10 shows one iteration together with the data arrangement, as previously discussed. Component 0 is always equal to zero, as expected by Mont-

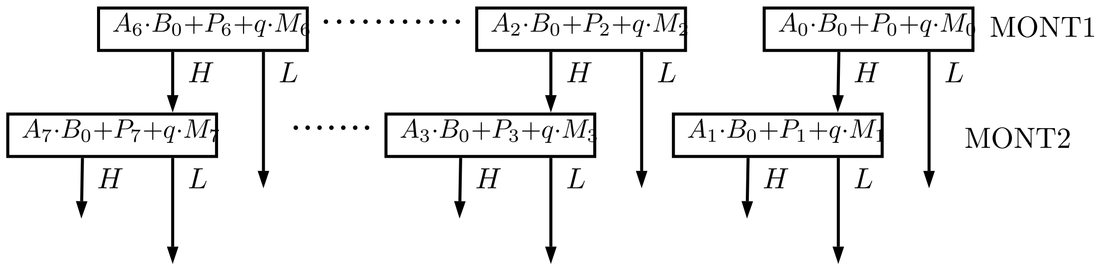

$A_6{\cdot}B_0{+}P_6{+}q{\cdot}M_6$
$A_2{\cdot}B_0{+}P_2{+}q{\cdot}M_2$
$A_0{\cdot}B_0{+}P_0{+}q{\cdot}M_0$
MONT1
$A_7{\cdot}B_0{+}P_7{+}q{\cdot}M_7$
$A_3{\cdot}B_0{+}P_3{+}q{\cdot}M_3$
$A_1{\cdot}B_0{+}P_1{+}q{\cdot}M_1$
MONT2
H
L

Figure 2.10: First iteration of carry less multiplication.

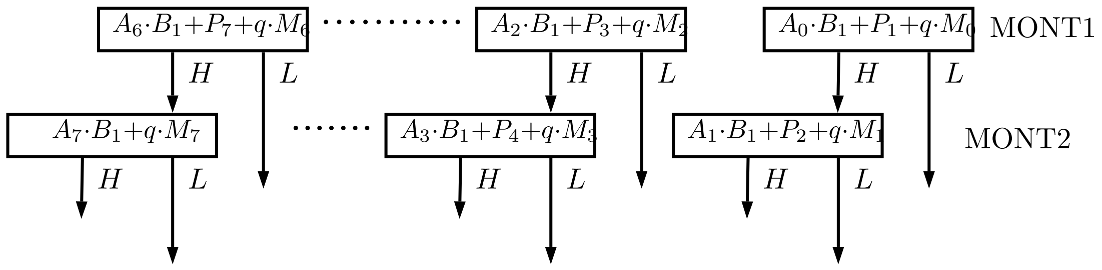

$A_6{\cdot}B_1{+}P_7{+}q{\cdot}M_6$
$A_2{\cdot}B_1{+}P_3{+}q{\cdot}M_2$
$A_0{\cdot}B_1{+}P_1{+}q{\cdot}M_0$
MONT1
$A_7{\cdot}B_1{+}q{\cdot}M_7$
$A_3{\cdot}B_1{+}P_4{+}q{\cdot}M_3$
$A_1{\cdot}B_1{+}P_2{+}q{\cdot}M_1$
MONT2
H
L

Figure 2.11: Second iteration of carry less multiplication.

gomery algorithm. Output vectors will be used in the next iteration as operand p, but in a circular shift order, e.g., $P(6,4,2,0)$ is used as $P(0,6,4,2)$. As an example, Figure 2.11 shows the second iteration. High order components, stored in registers H' and H'', are propagated internally between subsequent Montgomery instructions, but carry is not propagated through all components. Using a Carry Save-like approach, such propagation is postponed to the Carry Recovery stage. The last instruction in each iteration generates higher order components unsuitable to be summed in the first instruction of the next iteration (in the example of Figure 2.10, they have weights $(8,6,4,2)$). Internal registers H' and H'' have then to be stored in the regular SIMD register file and retrieved when executing an iteration starting with proper component weights. This is achieved by using instructions LOADH, STOH1 and STOH2, listed in Table 2.2.

2.4.3 Carry recovery and final subtraction

In order to generate the correct result carries have to be propagated through all the components of the product result. This is achieved by using instructions for extended arithmetic, namely ADDC (see Table 2.2), which support the sum of input carries, stored in an implicit scalar register, and generation of output carries, which can themselves be used as input in a subsequent ADDC instruction. The final subtraction is performed in a similar way, using SUBB instruction, which supports input/output borrow bits. The final subtraction is always executed, which ensures that the overall execution time is independent of the input data, and its result is stored separately from Carry Recovery output. Predicated MOVE instructions are used to store the final correct result in constant time, according to the sign of the subtraction result.

Opcode	Pipeline	Description
MONT1	MAC	Compute partial product and apply Montgomery reduction. Compute q. To be used only at the beginning of each iteration.
MONT2	MAC	Compute partial product and apply Montgomery reduction. Do not compute q. To be used after MONT1 to process other operand components.
STOH1	MAC	Read MAC internal register H'.
STOH2	MAC	Read MAC internal register H''.
LOADH	MAC	Load MAC internal register H' and H''.
LOADM	MAC	Load MAC internal register M (modulus) and MP ($m' = -m^{-1} \bmod 2^{32}$).
INTLVL	ALU	Interleave lanes of the operands. Return the lanes in even positions.
INTLVH	ALU	Interleave lanes of the operands. Return the lanes in odd positions.
DEINTLVL	ALU	De-interleave lanes of the operands. Return low order lanes.
DEINTLVH	ALU	De-interleave lanes of the operands. Return high order lanes.
ADDC	ALU	Sum the first two operands and the carry contained in the third operand. Return the sum as first result and the generated carry as the second result.
SUBB	ALU	Subtract from the first operand the second operand and the borrow contained in the third operand. Return the difference as the first result and the generated borrow as the second result.

Table 2.2: Extended ISA to support Montgomery multiplication.

Chapter 3

AES extensions for GPU

In 1997 the *National Institute for Standards and Technology* publishes a request for proposal to select a new symmetric cipher, in order to supersede the old DES. Of the five finalists, NIST chose in 2000 the Rijndael cipher, which is actually a family of ciphers, among which three where standardized [54], all with a block size of 128 bits and with key size of 128, 192, 256 bits.

3.1 Algorithm

Algorithm 5 AES encryption algorithm [54]

```
 1: $state \leftarrow state \oplus k_0$
 2: for $i \leftarrow 1 .. N_r - 1$ do
 3:     $state \leftarrow SubBytes(state)$
 4:     $state \leftarrow ShiftRows(state)$
 5:     $state \leftarrow MixColumns(state)$
 6:     $state \leftarrow state \oplus k_i$
 7: end for
 8: $state \leftarrow SubBytes(state)$
 9: $state \leftarrow ShiftRows(state)$
10: $state \leftarrow state \oplus k_{N_r}$
```

The direct cipher algorithm is shown in 5. N_r is the number of rounds, that, for AES128, AES192 and AES256 is respectively, 10, 12 and 14. k_0, ..., k_{N_r} are the round keys.

Summarizing, the operations needed in each round are

- Byte sustitution
- Row shift
- Column mixing
- Key Xor-ing

3.2 Architecture

In order to implement AES support in the GPU core, we explored two possible approaches: a first implementation, that we call *Resource-friendly*, tried to exploit hardware facilities already present in the core in order to minimize the requirement of additional hardware resources required. A second implementation, using completely dedicated functional unit, was design to achieve better performances at the cost of more hardware resources usage.

3.2.1 Resource-friendly implementation

Data arrangement: according to the AES algorithm, the AES 16 byte block is arranged in the 4x4 matrix *column-wise*. In most implementation each 32 bit word (or a lane in a vector machine) corresponds to a column. In our implementation, we chose that each lane correspond to a *row*.

$$S_{4,4} = \begin{pmatrix} s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \\ s_4 & s_8 & s_{12} & s_{16} \end{pmatrix} \tag{3.1}$$

Byte substitution

As defined by the AES NIST standard [54], the byte substitution operation is performed by taking the multiplicative inverse in the finite field GF(2^8) of each byte in the state matrix. The element 0 (zero), which does not have a multiplicative inverse, maps to itself.

lane 3				lane 2			
s_{16}	s_{12}	s_8	s_4	s_{15}	s_{11}	s_7	s_3

lane 1				lane 0			
s_{14}	s_{10}	s_6	s_2	s_{13}	s_9	s_5	s_1

Table 3.1: Data arrangement in vector register

Then compute the affine transformation: $b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$, where b_i is the i^{th} bit of the byte, while c_i is the i^{th} bit of the constant 01100011.

Even though the byte substitution is a known and deterministic function and could be computed directly, it is more usual to compute it using a look-up table. Such approach has the advantage to be adaptable to different ciphers, where the substitution function could not be known analytically (e.g. DES) or to complex to compute. We chose such approach and adapted it in order to exploit the ScratchPad Memory (SPM) already present in our GPU pipeline. We recall that the look-up memory approach has the disadvantage to require an high degree of parallelism in memory access. This could be achieved by using multiple ports, but a few can be physically achieved. The alternative is to duplicate the look-up table, with an increase in resource usage

Our SPM is made up of several banks, where we suppose the the number of bank access is at least equal to the number of available lanes. This way, if a vector memory access is well aligned it will generate no bank conflict and there will be no additional stall cycles.

In the following, we'll suppose that the number of banks is at least as much as the number of lanes.

Since each lane contains a row of the state matrix, that is four bytes, four look-up instruction will be required to perform a complete byte substitution. Each instruction will take into consideration one byte at a time, using it to generate an address to access the SPM (Operand Fetch II stage). The address generated is such that each lane access its own bank. The output byte is written in the homologous position in the destination register, using the Write Enable facility already present in the baseline version of our GPU.

The four instructions will obviously write different byte positions in the same destination register. This condition generates unnecessary

Write-after-Write hazards and, if source and destination registers are the same, also Read-after-Write hazards. Since SPM access time are long, latency stalls can be filled using multithread, but since we are considering also a scenario where few threads are used and/or latency, we wanted to introduce an optimization to latency, but which also didn't require complex control logic, which would be against the GPU approach. The solution was to partly delegate the hazard control to the software/programmer side. The programmer will specify in a bit of the Look-up instruction if he desires the destination register to be annotated in the scoreboard table. In the negative case, is up to him to manage data dependencies.

For the ByteSubstitution application, only the last of the four lookup instruction will annotate the destination register, so that false dependencies between lookups will be removed, but a subsequent instruction using the result (ColumnMix) will be stalled if necessary.

The correcteness is guaranteed by the fact the execution is in-order, different thread don't have any common register so interference is not possible and each lookup will read/write a different byte in the source/destination registers.

In our approach, there can never be any bank access conflict, so each lookup operation will always take the minimun number of clock cycles and, more importantly, it will take always *constant* time. Such property is desirable in order to twart *memory timing* attacks.

Row shift

Figure 3.1 show the rotate stage of an AES round. Since we chose to arrange the state matrix in a row wise fashion, we can perform such operation using the classic circular shift instruction. Optimization.

Mix column

Dedicated special function unit. It includes a "routing" stage which mixes data from different lanes. Mix column acts column-wise and each column is considered as a polynomial over GF(2^8). Each column is muliplied modulo $x^4 + 1$ by the constant polynomial $a(x) = 3x^3 + x^2 + x+2$. Given the peculiar nature of such operation, we chose to implement it with a dedicated unit. It whould be possible to implement it using a software approach with standard instructions, but it would require too

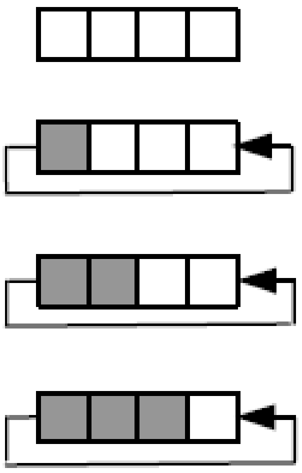


Figure 3.1: Row rotate.

many instruction. With our choice of data arrangement, Column Mixing requires to compute with data coming from different lanes, implying an intense use of *shuffle* operations. Since we are already implementing a dedicated functional unit, we chose to include into such unit the crossbar logic in order manage the cross-lane data movements with a reduced number of clock cycles.

AddRoundKey

Adding the round key requires just to Xor the round key with the current intermediate ciphertext. Even though the XOR instruction is already present, we chose to include such operation in the MixColumns stage, in order to reduce the number of operations required per round.

3.2.2 Optimizations

We implemented some simple optimizations in order to get better performances: firstly, we included the *Row Shift*, which corresponds to a byte-level rotate operation, into second stage of *Operand Fetch* unit. Such byte rotate function is used in the *Mix Column* instruction, before actual mixing is performed. As we have alread said, we included the key xor-ing inside the SFU implementing *Mix Column*. This didn't cause

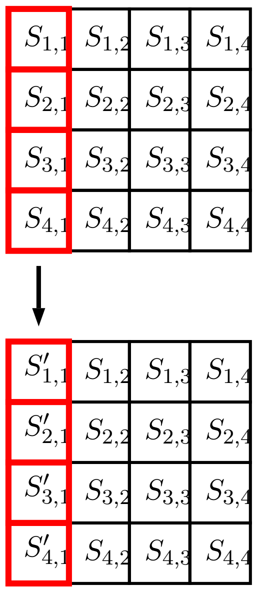

$S_{1,1}$ $S_{1,2}$ $S_{1,3}$ $S_{1,4}$
$S_{2,1}$ $S_{2,2}$ $S_{2,3}$ $S_{2,4}$
$S_{3,1}$ $S_{3,2}$ $S_{3,3}$ $S_{3,4}$
$S_{4,1}$ $S_{4,2}$ $S_{4,3}$ $S_{4,4}$
$S'_{1,1}$ $S_{1,2}$ $S_{1,3}$ $S_{1,4}$
$S'_{2,1}$ $S_{2,2}$ $S_{2,3}$ $S_{2,4}$
$S'_{3,1}$ $S_{3,2}$ $S_{3,3}$ $S_{3,4}$
$S'_{4,1}$ $S_{4,2}$ $S_{4,3}$ $S_{4,4}$

Figure 3.2: Column mixing stage.

a significant increase in latency, butjust a bit more resources are used since the unit now uses two operands instead of just one.

3.2.3 Dedicated HW implementation

We design also a Special Function Unit dedicated entirely to AES encryption, with a single instruction implementing all the round steps described above. This was done in order to achieve the best performance. The more important difference with the previous implementation, is the inclusion in the SFU of dedicated memory blocks to use as lookup table for *Byte Substitution*. The cost for the performance speed-up is an increase in resource usage, which, as we'll see in the chapter dedicated to results, is not that significant.

3.2.4 Inverse cipher

The AES FIPS standard [54] presents a possible implementation of the inverse cipher, where the round and algorithm structure is the same as in the direct cipher, which is an advantage in software but especially in hardware or hardware supported implementations. The Inverse Cipher

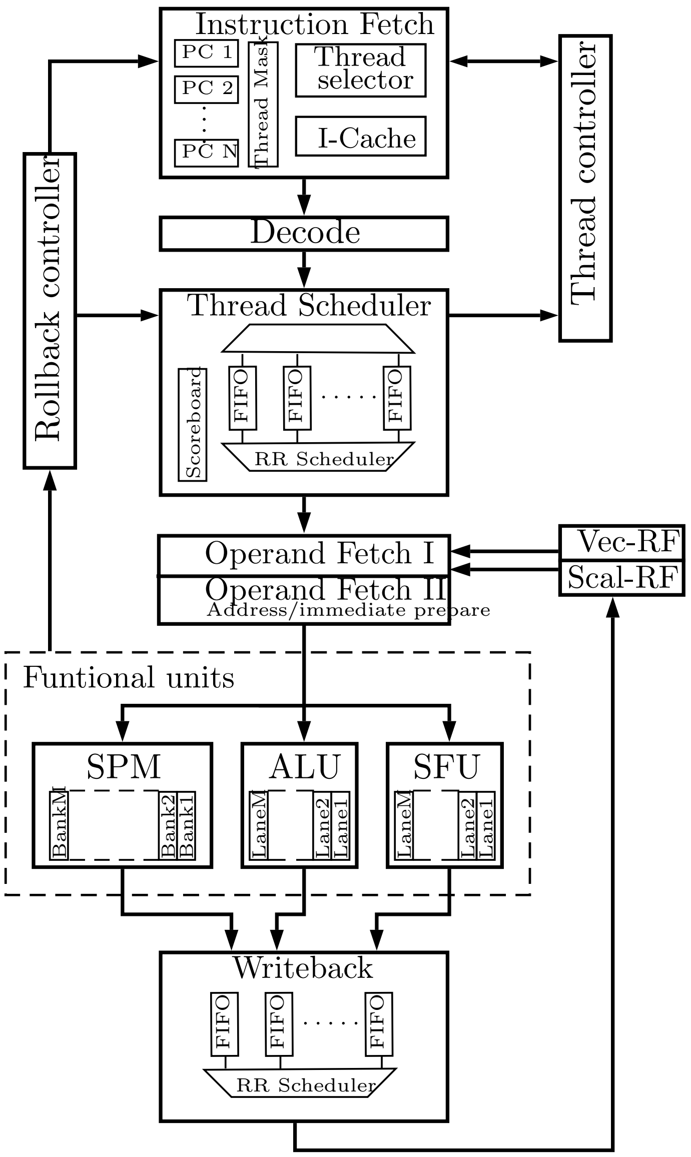


Figure 3.3: GPU pipeline for resource-friendly AES implementation.

Algorithm 6 AES inverse cipher algorithm

```
1: $state \leftarrow state \oplus k_{N_r}$
2: for $i \leftarrow N_r - 1$ downto 1 do
3:     $state \leftarrow InvSubBytes(state)$
4:     $state \leftarrow InvShiftRows(state)$
5:     $state \leftarrow InvMixColumns(state)$
6:     $state \leftarrow state \oplus k_i$
7: end for
8: $state \leftarrow InvSubBytes(state)$
9: $state \leftarrow InvShiftRows(state)$
10: $state \leftarrow state \oplus k_0$
```

algorithm, presented in Algorithm 6, stems from the observation that SubBytes and ShiftRows commute and obviously so are the corresponding inverse operations. Besides, the MixColumn transformation is linear so we have that

$$MixColumnds(state \oplus k_i) = MixColumns(state) \oplus MixColumns(k_i) \tag{3.2}$$

The same is valid for the inverse transformation. That means that in the inverse cipher we can swap (commute) InvMixColumns and AddRoundKey if we modify the Key Scheduling to include the computing of InvMixColumns. The resulting algorithm is shown in 6. The similarity of direct and inverse cipher is helpful also in our implementation, since the only thing to do is to invert each operation. For SubBytes, it suffices to preload the SPM with the inverse look-up table. ShiftRows is also easily inverted. MixColumn unit needs to implement the inverse transformation represented by the multiplication modulo $x^4 + 1$ by the constant polynomial $a^{-1}(x) = 11x^3 + 13x^2 + 9x + 14$, while the routing scheme among the lanes is the same.

Chapter 4

Experimental results

In this chapter we'll summerize the result of the implementations respectively for the Montgomery multiplication and AES-128 encryption.

4.1 Montgomery multiplication

Our GPU design presented in the chapter about Montgomery multiplication was implemented on a Xilinx Zynq-7030 SoC device (7z030fbg484-1). The design was built on top of the GPU-like core provided by the MANGO H2020 project, which is highly customizable in terms of parameters such as the number of lanes or hardware threads. We used a basic setting of four lanes, four threads, and a word width of 32 bits in order to simplify the comparison with other works. A higher number of lanes is likely to provide more benefits in that it would enable a higher degree of parallelism and improved amortization of the control logic overhead over the functional unit area. The Montgomery MAC pipeline was synthesized with a maximum frequency of 200 MHz, while the whole core has a maximum frequency of 110 MHz. The critical paths involve the cache controller in the processor, confirming that the customized unit does not compromise the maximum frequency.

The design was described in System Verilog and functionally verified through extensive simulations relying on the GMP library [55] to generate test vectors automatically. For the timing evaluation, we assume that operands are already available in the register file while program instructions are available in the instruction cache, similar to the works used here for comparisons, which do not take data access latencies into account for the timing results. We wrote the software Montgomery routine for the 256×256 and 512×512 multiplication, using low-level assembly code[1]. The performance results are shown in Table 4.1. We compared our solution against a recent work in the technical literature demonstrating the use of SIMD instructions for Montgomery multiplication, namely Seo et al. [52], presenting results for NEON extensions in ARM Cortex-A9 and Cortex-A15 (both are multi-issue CPUs, but A15 has two SIMD pipelines), as well as a recent work by Massolino et al. [51] which instead relies on a pure hardware solution. The values reported for our implementation represent the average cycle count per single Montgomery multiplication when all four threads are active. Between parenthesis, we also indicate the cycle count for the case of a single active thread.

	256-bit MontMul	512-bit MontMul
Our GPU-like core	212 (634)	466 (1322)
[52] (A9)	658	2254
[52] (A15)	308	1485
[56] (A15)	n/a	1408
[51]	328	1093

Table 4.1: Performance results (# cycles/multiplication).

As shown above, in the 256-bit case, our solution has only a slightly higher clock count in the single thread case, mainly due to the long latency of the MAC pipeline. On the other hand, in the multi-thread case, such long latencies are hidden by interleaving the execution of multiple threads. Indeed, the results are quite good even with just four threads. For the 512-bit multiplication, even the single thread case becomes competitive, since more independent Montgomery instructions are needed per iteration, which can themselves hide the MAC latencies. As a further comparison targeting larger operands, we considered F. Zheng et al. [53]. They implement 1024-bit multiplication exploiting the floating point pipeline on an NVIDIA GTX TITAN device. By normalizing their result with respect to the clock frequency and the number of cores and

[1]The *** H2020 project also provides a full LLVM-based compiler toolchain. The toolchain can be easily extended for new hardware instructions which can be invoked from high-level C/C++ programs as *intrinsics* functions.

lanes, we infer a cycle count of 5090 per multiplication, which is significantly higher than the projected clock count of our customized GPU-like core.

To provide some indication about the area overheads of our customized extension, Table 4.2 shows the resource utilization in the baseline and extended version along with the corresponding percentage increase. The baseline version contains the following functional units: integer ALU, Multiply-And-Accumulate unit, scratchpad memory, L1 instruction cache, L1 data cache controller and coherence controller. There are two register files, a vector one and a scalar one, each containing 64 registers. The customized version extended the ALU and MAC units with the support for the instructions listed in Table 2.2. In order

	Baseline	Extended	Δ
Slice LUTs	24305 (30.9%)	27014 (34.4%)	+11.1%
Slice Registers	30832 (19.6%)	35847 (22.8%)	+16.3%
Block RAM Tile	76.5 (28.9%)	84.5 (31.9%)	+10.4%
DSPs	16 (4%)	27 (6.7%)	+68.8%

Table 4.2: Resource utilization for Montgomery extensions.

to assess performance vs. area trade-offs, we also wrote a version of the 256-bit Montgomery multiplication for the baseline core and measured the average clock count in the multi-threaded case, which turned out to be 558. That means that the resource overheads listed in Table 4.2 are rewarded by a performance speed-up of 263%.

4.2 AES-128 encyption

For the verification of implementation of the AES-128 algorithm, we followed the same approach described by the previous section. The prototyping device is the same (Xilinx Zynq-7030) with the same parameter setting for the GPU-like core. The results for the synthetized frequency are the same as in the previous section. The software implementation of the algorithm, using our ISA extensions, was hand-written in assembly, both for the resource-friendly and dedicated unit version. Test vectors, consisting of pairs of plaintext and ciphertext, were generated using OpenSSL library. The design was thoroughly tested using simulations, for both versions.

Table 4.3 reports the performance of the AES128 encryption implementation. The results are expressed as number of clock cycles per plaintext byte. Each row refers to one of the two versions of AES extensions we realized. Comparison is made against Intel AES-Ni extensions, which currenty are among the most performing AES implementation. Table 4.4 reports the area usage results, with reference to a base line version of *Nu+* core. Each *delta* column expresses the resource percentage increase between each extended version and the baseline version. The first extended *Nu+* version refers to the *Resource Friendly* one, the second to the *Dedicated HW* version.

cycles/byte	Our	Intel AES NI
Resource friendly	4.87	1.3
Dedicated HW	0.84	

Table 4.3: Performance results for AES extensions

	Baseline	Extended1	$\Delta 1$	Extended2	$\Delta 2$
Slice LUTs	21869	24920	+13.9%	24642	+12.7%
F7 Muxes	1410	1555	+10.3%	1413	+0.2%
F8 Muxes	9	33	+267%	16	+77.8%
Slice Registers	26795	28207	+5.3%	28490	+6.3%
Block RAM Tile	38.5	38.5	+0.0%	38.5	+0.0%
DSPs	0	0	+0.0%	0	+0.0%

Table 4.4: Resource utilization for AES extensions.

Conclusions

This thesis work presented an FPGA implementation of a dedicated hardware accelerator for very long operand multiplication as a basic building block of a homomorphic encryption processor. It described the high-level architectural concept and implementation as well as the experimental data collected from hardware synthesis.

Similarly it presented also a GPU-like processor with custom extensions, targeting Montgomery multiplication and AES encryption as an example of a performance-critical operation for a large class of cryptographic workloads.

Indeed, the GPU-like paradigm can be regarded as a general approach to build application-specific hardware multi-threaded/vector processors. Following the domain specific customization, such GPU-like accelerators can be implemented as Application-Specific Instruction Processors, and they could even be profitable for FPGA implementation, at least in those cases where the customized hardware becomes the dominat part, providing in both scenarios the key advantage of full programmability and tight integration with higher-level software code.

Overall, the results point out the great potential posed by FPGA technologies in the acceleration of compute-intensive cryptographic algorithms.

Bibliography

[1] W. Wang and X. Huang. Fpga implementation of a large-number multiplier for fully homomorphic encryption. *ISCAS*, pages 2589–2592, 2013.

[2] Andrew Chi-Chih Yao. How to generate and exchange secrets. *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986.

[3] Çetin K. Koç. *Cryptographic Engineering*. Springer Science & Business Medi, 2008.

[4] D. Suzuki. How to maximize the potential of FPGA resources for modular exponentiation. In *Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, volume 4727 of *LNCS*, pages 272–288. Springer, 2007.

[5] G. de Meulenaer, F. Gosset, M. M. de Dormale, and J.-J. Quisquater. Integer factorization based on elliptic curve method: Towards better exploitation of reconfigurable hardware. In *Proceedings of the 15th Annual Symposium on Field-Programmable Custom Computing Machines (FCCM) 2007*, pages 197–206. IEEE, 2007.

[6] S. Ghosh, D. Mukhopadhyay, and D. Roychowdhury. High speed flexible pairing cryptoprocessor on FPGA platform. In *Proceedings of the 4th international conference on Pairing-based cryptography*, pages 450–466. Springer-Verlag, 2010.

[7] T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, 2008.

[8] Rivyera S3-5000, January 2011.

[9] A. Cilardo and N. Mazzocca. Exploiting vulnerabilities in cryptographic hash functions based on reconfigurable hardware. *IEEE Transactions on Information Forensics and Security*, 8(5):810–820, May 2013.

[10] A. Cilardo, M. Barbareschi, and A. Mazzeo. Secure distribution infrastructure for hardware digital contents. *Computers Digital Techniques, IET*, 8(6):300–310, 2014.

[11] A. Cilardo, A. Mazzeo, L. Romano, and G.P. Saggese. An FPGA-based key-store for improving the dependability of security services. In *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, pages 389–396, Feb 2005.

[12] Mario Barbareschi, Ermanno Battista, Antonino Mazzeo, and Sridhar Venkatesan. Advancing WSN physical security adopting TPM-based architectures. In *Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on*, pages 394–399. IEEE, 2014.

[13] Mario Barbareschi, Pierpaolo Bagnasco, and Antonino Mazzeo. Supply voltage variation impact on Anderson PUF quality. In *Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 2015 10th International Conference on*, pages 1–6. IEEE, 2015.

[14] Yousef K. Sinjilawi, Mohammad Q. AL-Nabhan, and Emad A. Abu-Shanab. Addressing security and privacy issues in Cloud Computing. *Journal of Emerging Technologies in Web Intelligence*, 5(2):192–199, 2014.

[15] F. Moscato, F. Amato, A. Amato, and R. Aversa. Model-driven engineering of cloud components in metamorp(h)osy. *International Journal of Grid and Utility Computing*, 5(2):107–122, 2014.

[16] F. Amato, A. Mazzeo, V. Moscato, and A. Picariello. Exploiting cloud technologies and context information for recommending touristic paths. *Studies in Computational Intelligence*, 511:281–287, 2014.

[17] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[18] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. *EUROCRYPT*, pages 24–43, 2010.

[19] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. *CRYPTO*, pages 505–524, 2011.

[20] Josh Benaloh and Dwight Tuinstra. Receipt-free secret-ballot elections (extended abstract). In *STOC '94 Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 544–553. ACM, 1994.

[21] Hugo Jonker, Sjouke Mauw, and Jun Pang. Privacy and verifiability in voting systems: Methods, developments and trends. *Computer Science Review*, 10(Supplement C):1 – 30, 2013.

[22] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *STOC '85 Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 291–304. ACM, 1985.

[23] Pascal Paillier. *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*, pages 223–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[24] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, 4:169–180, 1978.

[25] Taher El Gamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Blakley G.R., Chaum D. (eds) Advances in Cryptology. CRYPTO 1984. Lecture Notes in Computer Science, vol 196. Springer, Berlin, Heidelberg*, 1984.

[26] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28:270–299, 1984.

[27] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. *Fully Homomorphic Encryption over the Integers with Shorter Public Keys*, pages 487–504. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[28] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. *Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers*, pages 446–464. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[29] C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. *Proc. Advances in Cryptology-EUROCRYPT 2011*, pages 129–148, 2011.

[30] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. *IACR Cryptology ePrint Archive*, 2012:99, 2012.

[31] J.S. Coron, T. Lepoint, and M. Tibouchi. Batch fully homomorphic encryption over the integers. *IACR Cryptology ePrint Archive*, 2013:36, 2013.

[32] H. Perl, M. Brenner, and M. Smith. hcrypt, 2011.

[33] S. Halevi and V. Shoup. Helib, homomorphic encryption library, 2012.

[34] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar. Exploring the feasibility of fully homomorphic encryption. 99:1, 2013.

[35] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and B. Sunar. Accelerating fully homomorphic encryption using GPU. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5, Sept 2012.

[36] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and B. Sunar. Exploring the feasibility of fully homomorphic encryption. *IEEE Transactions on Computers*, 64(3):698–706, 2015.

[37] Y. Doröz, E. öztürk, and B. Sunar. Evaluating the hardware performance of a million-bit multiplier. *Digital System Design (DSD),16th Euromicro Conference on*, 2013.

[38] Wei Wang, Xinming Huang, N. Emmart, and C. Weems. VLSI design of a large-number multiplier for fully homomorphic encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(9):1879–1887, 2014.

[39] Y. Doröz, E. öztürk, and B. Sunar. Accelerating fully homomorphic encryption in hardware. *draft, Under Review*, 2013.

[40] X. Cao, C. Moore, M. O'Neill, N. Hanley, and E. O'Sullivan. High speed fully homomorphic encryption over the integers. *Workshop on Applied Homomorphic Cryptography, to appear*, 2014.

[41] J.S. Coron, A. Mandal, D. Naccache, and M. Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. *CRYPTO*, pages 487–504, 2011.

[42] J.S. Coron, D. Naccache, and M. Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. *EUROCRYPT*, pages 446–464, 2012.

[43] James W. Cooley and John W. Tukey. *An Algorithm for the Machine Calculation of Complex Fourier Series*, volume 19 of *Mathematics of Computation*, pages 297–301. American Mathematical Society, 1965.

[44] Kassem Kalach and J.P. David. Hardware implementation of large number multiplication by fft with modular arithmetic. In *3rd International IEEE Northeast Workshop on Circuits and Systems Conference, NEWCAS 2005*, volume 2005, pages 267 – 270, 07 2005.

[45] Jerome A. Solinas. *Generalized Mersenne Prime*, pages 509–510. Springer US, Boston, MA, 2011.

[46] Niall Emmart and Charles C. Weems. High precision integer multiplication with a gpu. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1781–1787, 2011.

[47] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.

[48] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 311–323, 1986.

[49] Zhe Liu and Johann Großschädl. New Speed Records for Montgomery Modular Multiplication on 8-Bit AVR Microcontrollers. *IACR Cryptology ePrint Archive*, 2013:882, 2013.

[50] Yatao Yang, Chao Wu, Zichen Li, and Junming Yang. Efficient FPGA implementation of modular multiplication based on Montgomery algorithm. *Microprocessors and Microsystems - Embedded Hardware Design*, 47:209–215, 2016.

[51] Pedro Maat C. Massolino, Lejla Batina, Ricardo Chaves, and Nele Mentens. Low Power Montgomery Modular Multiplication on Reconfigurable Systems. *IACR Cryptology ePrint Archive*, 2016:280, 2016.

[52] Hwajeong Seo, Zhe Liu, Johann Großschädl, Jongseok Choi, and Howon Kim. Montgomery Modular Multiplication on ARM-NEON Revisited. *IACR Cryptology ePrint Archive*, 2014:760, 2014.

[53] Fangyu Zheng, Wuqiong Pan, Jingqiang Lin, Jiwu Jing, and Yuan Zhao. Exploiting the Floating-Point Computing Power of GPUs for RSA. In *Information Security - 17th International Conference, ISC 2014*, pages 198–215, Oct. 2014.

[54] Advanced Encryption Standard. http://csrc.nist.gov/publications /fips/fips197/fips-197.pdf.

[55] GNU Multiple Precision Arithmetic Library. https://gmplib.org/. Accessed: 2017-09-15.

[56] Hwajeong Seo, Zhe Liu, Johann Großschädl, and Howon Kim. Efficient arithmetic on ARM-NEON and its application for high-speed RSA implementation. *Security and Communication Networks*, 9:5401–5411, 2015.