UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PH.D. THESIS

IN

INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

EXPLORING THE SHA-2 DESIGN SPACE

RAFFAELE MARTINO

TUTOR: PROF. ALESSANDRO CILARDO

COORDINATOR: PROF. DANIELE RICCIO

XXXII CICLO

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

Abstract

While SHA-2 is a ubiquitous cryptographic hashing primitive, its role in emerging application domains, such as blockchains or trusted IoT components, has made the acceleration of SHA-2 very challenging due to new stringent classes of requirements imposed by such domains, especially implementation cost and energy efficiency.

This Ph.D. thesis explores the SHA-2 design space from different viewpoints. Its first contribution is a reasoned classification of the many SHA-2 designs proposed in the literature according to their architectural choices, each of them having different implications on the application requirement. Based on this analysis, this thesis introduces a framework and a methodology for evaluating and comparing different implementation options, which is used to assess the impact of each architectural technique on the application requirements, as well as the effect of variations in the underlying target technology. The last contribution of this thesis explores a different approach, namely utilising a specific target technology with maximum efficiency, and the resulting SHA-2 accelerator shows the best area efficiency reported so far in the literature.

Preface

Some of the research described in this Ph.D. thesis has undergone peer review and has been published in scientific journals and conference proceedings. This is the list of the scientific contributions originated from the research work of this Ph.D. thesis.

Journal paper Raffaele Martino and Alessandro Cilardo, "A Flexible Framework for Exploring, Evaluating, and Comparing SHA-2 Designs". In *IEEE Access*, vol. 7, 2019, DOI: 10.1109/ACCESS.2020.2972265.

Journal paper Raffaele Martino and Alessandro Cilardo, "SHA-2 acceleration meeting the needs of emerging applications: A comparative survey", in *IEEE Access* (accepted for publication), DOI: 10.1109/ACCESS.2019.2920089.

Conference Proceedings paper Raffaele Martino and Alessandro Cilardo, "A Configurable Implementation of the SHA-256 Hash Function", in *Proceedings of the 14th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC 2019)*, in Advances on P2P, Parallel, Grid, Cloud and Internet Computing, in Lecture Notes in Networks and Systems, vol. 96, 2020, pp. 558-567, DOI: 10.1007/978-3-030-33509-0_52.10.1007/978-3-030-33509-0_52.

Contents

Contents vii

List of Figures xi

List of Tables xiii

List of Acronyms xv

List of Symbols xvii

Introduction xxi

1 SHA-2 and its Applications 1
- 1.1 Cryptographic Hash Algorithms . . . 1
- 1.2 The Secure Hash Algorithm . . . 2
  - 1.2.1 Algorithm Definition . . . 3
  - 1.2.2 SHA-2 variants . . . 7
- 1.3 Applications . . . 8
  - 1.3.1 Blockchains . . . 10
  - 1.3.2 Internet of Things . . . 12
  - 1.3.3 Trusted Computing . . . 13

2 Classification of SHA-2 Acceleration Approaches 17
- 2.1 Approaches to SHA-2 Acceleration . . . 17
  - 2.1.1 Programmable Processor Architectures . . . 19
  - 2.1.2 Accelerator Architectures . . . 20
  - 2.1.3 Optimisation Techniques . . . 24
- 2.2 SHA-2 Accelerator Architectures . . . 32
  - 2.2.1 Basic architectures . . . 32
  - 2.2.2 Shift register architectures . . . 33
  - 2.2.3 Architectures with precomputation . . . 35

2.2.4 Architectures with spatial reordering 37
2.2.5 Architectures with quasi-pipelining 38

3 Evaluation of SHA-2 Hardware Acceleration Approaches 39
3.1 The Need for a Common Evaluation Platform 39
3.1.1 Evaluation Methodology 40
3.2 Workbench Architecture 41
3.2.1 Compressor 42
3.2.2 Expander 43
3.2.3 Control Unit 44
3.2.4 Reconfigurable aspects controlled by source-level parameters 45
3.2.5 Reconfigurable aspects controlled by component declarations 48
3.2.6 Discussion 50
3.3 Experimental Results 50
3.3.1 Design comparison against a specific target 50
3.3.2 Architectural Exploration 52
3.3.3 Exploring a different target 57
3.4 Analysis of the Impact of Design Techniques on Application Metrics 57
3.4.1 Performance 59
3.4.2 Area occupation and area efficiency 61
3.4.3 Power and energy consumption 61
3.4.4 Implementation complexity 64
3.4.5 Impact on applications 65

4 Efficient Multi-Operand Addition on FPGAs 67
4.1 Compressor Trees based on Parallel Counters 67
4.2 Generalised Parallel Counters 69
4.2.1 Efficiency Parameters of a GPC 71
4.2.2 From the GPC to the Compressor Tree 72
4.3 GPCs for the 7-series Xilinx FPGAs 74
4.3.1 The Xilinx 7-series Look-Up Table 74
4.3.2 A GPC library for the 7-series FPGA 75
4.3.3 Optimising the mapping of the GPCs 77

5 Efficient Mapping of SHA-2 on FPGA 81
5.1 Overview of 7-series Xilinx FPGA architectural features . 81
5.1.1 7-series Xilinx FPGA organization overview 81
5.1.2 LUT capabilities 82
5.2 Efficient SHA-256 implementation on 7-series Xilinx FPGA 84

5.2.1 Compressor 84
5.2.2 Expander 88
5.2.3 Other Components 91
5.2.4 Data Path 92
5.3 Experimental Results 96
5.3.1 Comparison with the State of the Art 97

Conclusion **99**

Bibliography **101**

List of Figures

2.1 General architecture of a SHA-2 processor core 20
2.2 General architecture of a SHA-2 accelerator core. 21
2.3 Straightforward architecture of the Expander 22
2.4 Straightforward implementation of the transformation round 23
2.5 The architecture with spatial reordering proposed in [76] . . . 28
2.6 Application of Quasi-pipelining in SHA-2 31
2.7 Straightforward shift register architecture implementation . . 34

3.1 Methodology for comparing different hash circuit architectures 41
3.2 Top level entity of the proposed evaluation platform 42
3.3 Expander architecture, with stage chaining 43
3.4 Expander architecture, unrolled of a factor 4 44
3.5 FSM of the Control Unit with the Compressor and the Expander aligned 45
3.6 FSM of the Control Unit with the Expander moved ahead . . 46
3.7 Architecture of the `Naive` transformation round core 49

4.1 Examples of a single column counter and a GPC with 6 input bits, with the dot notation 70
(a) 6:3 counter 70
(b) (1,3,2;4) GPC 70
4.2 Architecture of the LUT of the 7-series Xilinx FPGA 75
4.3 Examples of GPC mapping 76
(a) (0,5,3) GPC 76
(b) (0,3,3;4) GPC 76
4.4 Optimisations of GPC mapping applied to the (0,3,1,2;5) GPC 77
(a) computing the two least significant output bit with the same LUT 77
(b) computing the least significant carry bit 77

5.1 Very simplified architecture of a slice 83

5.2 Architecture of a CLB 84
5.3 LUT-based shift register 85
5.4 Proposed Compressor Tree for the addition within the SHA-256 Compressor 88
5.5 Architecture of the Expander with redundant LUT-based shift registers 90
5.6 Proposed Compressor Tree for the addition within the Expander 91
5.7 Architecture of the proposed SHA-256 implementation 94
5.8 Finite State Machine for the optimised SHA-256 implementation 95

List of Tables

1.1 Characteristics of members of the SHA-2 family of algorithms 9

2.1 Synoptic overview of SHA-2 acceleration solutions 18

3.1 Detail of the architectures explored 51
3.2 SHA-256 implementation results on the Kintex FPGA 53
3.3 SHA-512 implementation results on the Kintex FPGA 54
3.4 Results with the addition of the final stage for SHA-256 . . . 55
3.5 Results with the addition of the final stage for SHA-512 . . . 56
3.6 SHA-256 implementation results on the Artix-7 FPGA 58
3.7 Impact of SHA-2 optimisation techniques on evaluation metrics 59
3.8 Requirements of surveyed applications relying on SHA-2 and recommended optimisations 65

4.1 Primitive GPCs for $n \leq 6$ 78
4.2 GPC library 79

5.1 Combined truth table for the Maj and Σ_0 functions 86
5.2 Combined truth table for the Ch and Σ_1 functions 87
5.3 Cost of the Compressor 88
5.4 Cost of the Expander 90
5.5 Cost of the constants ROM 91
5.7 Cost of the multiplexers 92
5.6 Truth table for a 4:2 multiplexer 93
5.8 Cost of the proposed SHA-256 architecture 94
5.9 Implementation results for the proposed SHA-256 design . . . 97
5.10 Comparison of the proposed SHA-256 implementation with other implementations in the literature on the same FPGA family 98

List of Acronyms

AES Advanced Encryption Standard

AH Authentication Header

ALU Arithmetic and Logic Unit

ASIC Application-Specific Integrated Circuit

BRAM Block RAM

CAD Computer-Aided Design

CLA Carry Look-ahead Adder

CLB Configurable Logic Block

CMOS Complementary Metal-Oxide-Semiconductor

CSA Carry Save Adder

DES Data Encryption Standard

DRBG Deterministic Random Bit Generation

DSA Digital Signature Algorithm

ESP Encapsulated Security Payload

FIFO First In, First Out

FIPS Federal Information Processing Standard

FPGA Field Programmable Gate Array

FSM Finite State Machine

GPC Generalised Parallel Counter

HDL Hardware Description Language

HMAC Hash-Based Message Authentication Code

I/O Input/Output

IDE Integrated Development Environment

ILP Integer Linear Programming

IoT Internet of Things

IPSec Internet Protocol Security

LUT Look-Up Table

MAC Message Authentication Code

MD5 Message Digest 5

NIST National Institute of Standards and Technology

PDB Padded Data Block

PRNG Pseudo-Random Number Generator

RAM Random Access Memory

RFID Radio Frequency Identification

ROM Read-Only Memory

RTL Register Transfer Level

SGX Software Guard Extensions

SHA Secure Hash Algorithm

SHS Secure Hash Standard

TEE Trusted Execution Environment

VHDL VHSIC (Very High Speed Integrated Circuits) Hardware Description Language

List of Symbols

Please note that arithmetic operations within SHA-2 are intended to be modular.

Secure Hash Algorithm

$A_t, B_t, \ldots, H_t$	State variables at the round t
$DM\,(M)$	Digest of message M
$DM_k\,(j)$	k-th hash accumulator for the computation of the j-th PDB
F	Hash rate of a SHA-2 implementation
K_t	Round constant for the round t
$L\,(x)$	Length of bit string x
M^*	Message to be hashed, after being padded
M	Message to be hashed
N_{clk}	Number of clock cycles required by a SHA implementation to output a new hash value
Q	Throughput of a SHA-2 implementation
R	Number of iterations, or *rounds*, performed by the hash function
S	Number of stages of pipelined implementations
UF	Unrolling factor of unrolled implementations
W_t	Expander output word at the round t
$Z_t\,(j)$	j-th portion of the state variable at the round t
Z	State variable for an hash computation

τ_{clk}	Minimum clock period of a SHA-2 implementation
f_{clk}	Maximum clock frequency of a SHA-2 implementation
l	Input block size of the hash function

Generalised Parallel Counters

J	Number of operands to be added
K_i	Number of input bits of rank i in a GPC
K	Number of output operands of a compressor tree
N	Bit width of the operands
O_k	Output operands of a compressor tree
X_j	Operand to be added
δ_r	Compression ratio of a GPC
δ	Compression difference of a GPC
η	Performance efficiency of a GPC
ε	Area efficiency of a GPC
b_i	Bit of rank i within a binary number
m	Fan-in of a GPC
n	Fan-out of a GPC

Operators

$+$	Arithmetic addition
$-$	Arithmetic subtraction
$/$	Arithmetic division
$<$	Strict minority
$\cdot$	Arithmetic multiplication
$\forall$	Universal quantifier
$\geq$	Loose majority
$\ggg_r$	Circular right shift

$\ggg$	Logical right shift
$\in$	Set membership
$[\bullet : \bullet]$	Bit range
$\lfloor \bullet \rfloor$	Floor operator
$\leq$	Loose minority
$\neg$	Logical NOT
$\oplus$	Logical XOR
$\prec$	Lexicographic minority
$\wedge$	Logical AND
$\|\|$	Bitwise concatenation

Technological Parameters

C	Capacitive load of a gate
F_{in}	Fan-in of LUTs of an FPGA
F_{out}	Fan-out of LUTs of an FPGA
P	Instantaneous power consumption of a circuit
U	Utilisation of an FPGA
V	Voltage supplied to a circuit
α	Switching activity of a circuit

Introduction

CRYPTOGRAPHIC hash functions underlie many aspects of our everyday life today. Their properties are the cornerstone of many security applications and protocols where tampering with either the content of messages, or the identity of the sender, or with both, is to be avoided. Popular hash algorithms used in the past include Message Digest 5 (MD5) [100] and the Secure Hash Algorithm (SHA) [89] family of algorithms, however the discover of security vulnerabilities in MD5 [122, 12] and SHA-1 [121, 109] has left SHA-2 as the most commonly used hash function nowadays.

Traditionally, their application domain has been network security, usually over the Internet. In this context, client devices are usually sufficiently powerful to perform the relatively limited number of hash computations required by the security protocols, while servers can possibly benefit from hardware acceleration of the hash operation [80]. The hardware accelerator is designed to deliver the maximum possible throughput, usually at the cost of increased area and power consumption.

Moreover, the peculiar properties of cryptographic hash functions have made them the essential ingredient for emerging innovative applications, like blockchains and distributed ledgers, involving a wide range of platforms from high-end servers down to resource-constrained Internet of Things (IoT) devices. The Bitcoin mining process [83], which heavily relies on the SHA-2 hash function, is extremely demanding in terms of energy efficiency, even making its profitability uncertain from the miner's standpoint. Additionally, the development of new domains such as IoT, with its low-cost, battery-powered devices needing to communicate securely [61], has also contributed to an increased demand for area-efficient and energy-efficient accelerators to be paired with the resource-constrained main processor.

Driven by the diverse sets of requirements posed by emerging applications, a number of different design techniques have been introduced, targeting the optimisation of area, energy or power consumption of the

resulting SHA-2 accelerator. Many of these techniques still deliver an increased throughput, but without excessive area or power penalties. Nevertheless, there are also optimisation techniques which are keen to sacrifice throughput in order to achieve significant area or power savings. One of the contributions of this thesis is to provide a classification of the design techniques which have been proposed for the design of the SHA-2 accelerator, and a systematic, evidence-supported analysis of the impact of each technique on the application requirements. These findings can be useful for the designer who is confronted with the task of designing a SHA-2 hardware accelerator under a given set of performance, area, energy and power requirements.

However, the choice of the best design alternative to meet a specific set of requirements is influenced also by the specific technological characteristics of the hardware which will be used to physically realise the accelerator, the impact of which is often difficult if not impossible to estimate on paper. One of the reasons for this is that many low-level characteristics of the target technology are not made known by the manufacturer, but are only available to the algorithms used by vendor-specific Computer-Aided Design (CAD) and Integrated Development Environment (IDE) tools used to translate the Hardware Description Language (HDL) description of the architecture into a physical circuit. Therefore, the fulfilment of strict requirements may force the designer to implement a number of alternatives in order to compare their actual performance.

In order to simplify this activity, this thesis proposes an evaluation framework which allows to obtain different architectures of the SHA-2 core simply by reconfiguring a number of parameters. This framework is particularly useful when the designer wants to evaluate a design originally proposed for a different application. In such a case, the design may take advantage of hypotheses specific to the original application, which are no longer valid in the context at hand. The proposed framework allows to evaluate each design without taking into account such assumptions.

A greater degree of control over the impact of the target technology over the implementation results can be obtained by working at a level lower than the Register Transfer Level (RTL). This makes it possible to take advantage of specific features of the target technology in order to achieve further gains in the optimisation objective. Therefore, one of the contributions of this thesis is the proposal of an architecture of the SHA-2 accelerator for the Xilinx 7-series Field Programmable Gate Array (FPGA) family, capable of achieving the best area efficiency reported in the literature. This architecture has been designed at the level of the

structural components of the 7-series FPGA.

The outline of this thesis is as follows.

Chapter 1 provides the context of this work by reviewing the SHA-2 algorithm and its applications. The first part of the chapter reviews the properties that make cryptographic algorithms like SHA-2 so relevant in many applications, and then reviews the algorithm itself. In the second part of the chapter, SHA-2 applications in various different field, requiring hardware implementations of the hash function, are surveyed. For each application, it is described how the properties of SHA-2 are exploited, and what specific requirements are placed on the underlying hardware implementing the hash function.

Chapter 2 reviews the body of technical literature on the hardware implementation of SHA-2. Architectures are classified according to their implementation approach and the optimisation techniques they employ. Then, each approach and optimisation technique is described in detail, with particular attention on the implications of employing the technique on application metrics. The second part of the chapter analyses each SHA-2 implementation proposal to show how different techniques have been combined to meet the designer's requirements.

Chapter 3 describes how to systematically evaluate different architectures for the SHA-2 hardware core with a newly introduced evaluation framework. Building on the material exposed in Chapter 2, an evaluation platform has been developed, that captures the commonalities between the different architectures while allowing a fair comparison between the alternatives. The chapter begins by introducing a methodology for the systematic comparison of different SHA-2 hardware designs under a given set of constraints. Afterwards, the architecture of the proposed evaluation platform is described. The last part of the chapter presents an analysis of the impact of each design technique on the application metrics and therefore on the application requirements, which builds on the experimental results obtained via the evaluation platform.

Chapter 4 introduces how to efficiently add multiple operands on a FPGA. The analysis described in the previous chapters clearly indicates that the bottleneck for the performance of a SHA-2 hardware implementation is the multi-operand addition, the efficient implementation of which strongly varies on different target technologies. The chapter presents a technique to perform multi-operand additions which has been proven to be effective when the target technology is an FPGA, and concludes by applying the technique to a specific FPGA family.

Chapter 5 proposes a new design for the hardware implementation

of SHA-2 which outperforms all the existing proposals in the literature in terms of area efficiency. This result is obtained by using a number of design strategies, including the multi-operand addition techique presented in Chapter 4. The chapter describes the architecture of the proposed SHA-2 design and compares it with the state of the art.

Chapter 1

SHA-2 and its Applications

CRYPTOGRAPHIC hash functions have been employed for decades as a fundamental building block of information security. Their properties ensure that message integrity as well as the sender's identity in a communication can be securely verified, provided that a secret has been shared between the communicating parties.

In more recent years, the particular properties of cryptographic hash functions have also paved the way for new application domains, like blockchains and distributed ledgers, with different sets of requirements.

This chapter deals with one of the most commonly used cryptographic hash functions, the SHA, and more specifically SHA-2. The SHA-2 algorithm is reviewed, along with its properties as a cryptographic hash function. After that, a brief overview of the landscape of applications that relies on SHA-2 is presented.

1.1 Cryptographic Hash Algorithms

A hash algorithm can be used for providing security services only if it ensures a set of properties, which are not necessarily guaranteed by general-purpose hash functions. Hash functions ensuring these security properties, like SHA-2, are called *cryptographic hash function*.

The *one-way* or *preimage resistance* property of cryptographic hash functions implies that it is computationally infeasible to compute the message M given its hash $DM(M)$[1]. The *second preimage resistance* property means that, given a hash value $DM(M)$, it is computationally infeasible to find a different message $M' \neq M$ that yields the same hash

[1] If $DM(M)$ is the hash value of M, M is called the *preimage* of $DM(M)$.

value. The *pseudo-randomness* property means that the hash value of a message must expose statistical randomness. Finally, the *collision resistance* property means that it is computationally infeasible to find a pair of messages M_1 and M_2 which produce the same hash value.

Each application has different requirements in terms of these properties for its underlying hash function [107]. In Section 1.3 a wide range of applications of SHA-2 are presented, not limited to security services, and their specific requirements in terms of cryptographic hash properties are discussed.

The security of a cryptographic hash algorithm is a measure of the computational complexity of breaking its properties, especially its collision resistance. It is measured in terms of the number of operations required to break the algorithm, expressed as a power of 2. More specifically, a hash function is said to have x *security bits* if 2^x operations are required to break it. The name stems from the fact that, for an unbroken hash algorithm, the only feasible attack relies on brute force, which implies a mean number of attempts exponential in the length of the hash value $L\left(DM\left(M\right)\right)$.

However, due to the birthday paradox, the number of attempts required on average to find two different messages hashing to the same value, i.e. to break the collision resistance property in its broadest sense, is $2^{L(DM(M))/2}$, making the number of security bits a half of the hash size.

The hash sizes of the members of the SHA-2 family have been chosen to match the security bits provided by symmetric encryption algorithms. For these cryptographic algorithms, the number of security bits coincides with the key length, hence SHA-2 hash sizes are twice the key length of a symmetric encryption algorithm. Specifically, a hash size of 224 implies the same number of security bits of the Triple Data Encryption Standard (DES) [88], while 256, 384, and 512 are twice the key sizes supported by the Advanced Encryption Standard (AES) [84].

1.2 The Secure Hash Algorithm

The Secure Hash Algorithm is a family of cryptographic hash functions defined by the National National Institute of Standards and Technology (NIST) and published as the Federal Information Processing Standard (FIPS) 180, Secure Hash Standard (SHS) [89], for being employed by U.S. government agencies. In the first version of the SHS, FIPS 180-0, published in 1993, only one hash function was described. This algorithm, now known as SHA-0, has been soon after replaced by SHA-1 in the

revised version of the standard, FIPS 180-1, published in 1995. While the only difference between SHA-0 and SHA-1 is a single bitwise rotation, SHA-0 turns out to be considerably weaker than SHA-1 [123].

SHA-2 was firstly introduced in 2001, when FIPS 180-2 defined three of its variants. These included the two main variants of SHA-2, which are SHA-256 and SHA-512, respectively the 32- and 64-bit versions of the same hash algorithm; and a truncated variant of SHA-512, SHA-384. Subsequent updates to the SHS added other truncated variants to the family. Namely, FIPS 180-3 in 2004 added SHA-224, which is a variant of SHA-256, whilst FIPS 180-4 added in 2012 two variants of SHA-512, SHA-512/224 and SHA-512/256, along with the general specification of a t-wide hash function called SHA-512/t.

In summary, SHA-2 is the set of the cryptographic hash algorithms defined in the SHS, excluding SHA-1. The following description will focus on SHA-256, while Section 1.2.2 will provide the detail of the other variants.

1.2.1 Algorithm Definition

SHA-2 is a *block-based* hash algorithm, meaning that it operates on blocks of fixed size l to produce a fixed-size hash value $DM(M)$ for a given input message M. For SHA-256, $l = 512\,\text{bit}$ and the hash size is 256 bit. Nevertheless, it is capable to process messages of practically any arbitrary length, up to a limit fixed by the padding. Padding is performed to ensure that the variable length $L(M)$ of the message M to be hashed is always a multiple of l.

Padding step

The message is padded with a single 1 bit, followed by as many 0 bits as needed to reach a length congruent to $448 \bmod 512$, so as to leave the last 64 bits to encode a representation of the original length $L(M)$. This means that the length of the message to be hashed must not exceed 2^{64} bit.

Padding is always performed, even when the length of the message does not strictly require it. Therefore, the number of message blocks required to hash a message can be written as $\lfloor L(M)/l \rfloor + 1$, which can also be written as a function of the padded message M^* in the form $L(M^*)/l$. Moreover, the actual maximum length of a message to be hashed is limited to $2^{64} - 65\,\text{bit}$.

To recall that padding has to be performed prior to any hash computation, each message block is usually referred to as the Padded Data Block (PDB).

Variables definition

The current hash value is used as an input for the hashing of the j-th PDB M_j, and is stored in eight $32-$bit accumulator variables DM_k $k \in [0, 7]$. Their initial values are the first 32 bits of the fractional parts of the square root of the first eight prime numbers, although they are also hard-coded in the standard [89] as follows:

$$\begin{aligned} DM_0\,(0) &= \texttt{0x6a09e667} \\ DM_1\,(0) &= \texttt{0xbb67ae85} \\ DM_2\,(0) &= \texttt{0x3c6ef372} \\ DM_3\,(0) &= \texttt{0xa54ff53a} \\ DM_4\,(0) &= \texttt{0x510e527f} \\ DM_5\,(0) &= \texttt{0x9b05688c} \\ DM_6\,(0) &= \texttt{0x1f83d9ab} \\ DM_7\,(0) &= \texttt{0x5be0cd19} \end{aligned} \tag{1.1}$$

For each PDB, the algorithm performs $R = 64$ iterations. Each iteration updates the value of the internal state Z, constituted by eight 32-bit working variables called A to H. To refer to the state variables as a whole, the following definition is used from now on:

$$\begin{aligned} Z_t\,(0) &= A_t \\ Z_t\,(1) &= B_t \\ Z_t\,(2) &= C_t \\ Z_t\,(3) &= D_t \\ Z_t\,(4) &= E_t \\ Z_t\,(5) &= F_t \\ Z_t\,(6) &= G_t \\ Z_t\,(7) &= H_t \end{aligned} \tag{1.2}$$

At the beginning of the processing of each PDB, the value of the

working variables matches the value of the accumulators:

$$\begin{aligned} A_0 &= DM_0(j) \\ B_0 &= DM_1(j) \\ C_0 &= DM_2(j) \\ D_0 &= DM_3(j) \\ E_0 &= DM_4(j) \\ F_0 &= DM_5(j) \\ G_0 &= DM_6(j) \\ H_0 &= DM_7(j) \end{aligned} \tag{1.3}$$

The message to be hashed determines the value of the variable W_t. Namely, this is a 32-bit variable whose value changes at every iteration as follows:

$$W_t = \begin{cases} M_j[32 \cdot t + 31 : 32 \cdot t] & 0 \leq t < 16 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & t \geq 16 \end{cases} \tag{1.4}$$

In other words, for the first 16 iterations, W_t is simply the t-th 32-bit word of M_j. Note also that all the additions involved in the SHA-2 family are performed modulo the variable size, which is 32 for SHA-256. The two functions $\sigma_0(x)$ and $\sigma_1(x)$ are defined as follows.

$$\begin{aligned} \sigma_0(x) &= x \ggg_r 7 \oplus x \ggg_r 18 \oplus x \gg 3 \\ \sigma_1(x) &= x \ggg_r 17 \oplus x \ggg_r 19 \oplus x \gg 10 \end{aligned} \tag{1.5}$$

Lastly, the algorithm employs R constants K_t, the value of which are hard-coded in the standard [89] as the first 32 bits of the fractional part of the cube root of the first 64 prime numbers.

Message block processing

The computation performed by each iteration is often referred to as the *step function*. Its main part is the computation of the following sub-functions:

$$T^1{}_t = H_t + \Sigma_1(E_t) + Ch(E_t, F_t, G_t) + K_t + W_t \qquad \forall t \in [0, R-1] \tag{1.6}$$

$$T^2{}_t = \Sigma_0(A_t) + Maj(A_t, B_t, C_t) \qquad \forall t \in [0, R-1] \tag{1.7}$$

where the *Choose*[2] and *Majority* functions are defined as

$$Ch\left(x,y,z\right)=\left(x\wedge z\right)\oplus\left(\neg x\wedge y\right) \tag{1.8}$$
$$Maj\left(x,y,z\right)=\left(x\wedge y\right)\oplus\left(x\wedge z\right)\oplus\left(y\wedge z\right) \tag{1.9}$$

whilst the $\Sigma_0\left(x\right)$ and $\Sigma_1\left(x\right)$ functions are defined as follows.

$$\begin{aligned}\Sigma_0\left(x\right)&=x\ggg_r 2\oplus x\ggg_r 13\oplus x\ggg_r 22\\ \Sigma_1\left(x\right)&=x\ggg_r 6\oplus x\ggg_r 11\oplus x\ggg_r 25\end{aligned} \tag{1.10}$$

The overall step function can then be written as

$$\begin{aligned}A_{t+1}&={T^1}_t+{T^2}_t && \forall t\in[0,R-1]\\ B_{t+1}&=A_t && \forall t\in[0,R-1]\\ C_{t+1}&=B_t && \forall t\in[0,R-1]\\ D_{t+1}&=C_t && \forall t\in[0,R-1]\\ E_{t+1}&=D_t+{T^1}_t && \forall t\in[0,R-1]\\ F_{t+1}&=E_t && \forall t\in[0,R-1]\\ G_{t+1}&=F_t && \forall t\in[0,R-1]\\ H_{t+1}&=G_t && \forall t\in[0,R-1]\end{aligned} \tag{1.11}$$

It is worth noting that only two working variables out of eight are actually updated at each iteration, whereas the other six take the value of the following variable. More specifically, not taking into account A_{t+1} and E_{t+1}, Eq. (1.11) can be rewritten as:

$$Z\left(k\right)_{t+1}=Z\left(k-1\right)_t \qquad \forall k\in[1,3]\cup[5,7] \tag{1.12}$$

Message block chaining

After all of the prescribed number of iterations R have been performed, the current hash value is updated as follows:

$$\begin{aligned}DM_0\left(j+1\right)&=DM_0\left(j\right)+A_R\\ DM_1\left(j+1\right)&=DM_1\left(j\right)+B_R\\ DM_2\left(j+1\right)&=DM_2\left(j\right)+C_R\\ DM_3\left(j+1\right)&=DM_3\left(j\right)+D_R\\ DM_4\left(j+1\right)&=DM_4\left(j\right)+E_R\\ DM_5\left(j+1\right)&=DM_5\left(j\right)+F_R\\ DM_6\left(j+1\right)&=DM_6\left(j\right)+G_R\\ DM_7\left(j+1\right)&=DM_7\left(j\right)+H_R\end{aligned} \tag{1.13}$$

[2]The value of x chooses whether the output of y or z is propagated to the output.

If $j < L(M^*)/l$, the algorithm continues with the processing of M_{j+1}; otherwise, there are no message blocks left and the message digest is $DM_0 \,||\, DM_1 \,||\, DM_2 \,||\, DM_3 \,||\, DM_4 \,||\, DM_5 \,||\, DM_6 \,||\, DM_7$.

It is worth noting, since it constitutes an hurdle for parallelisation in hardware implementations, that Eq. (1.13) implies that the processing of the PDB $j+1$ cannot start before the end of the processing of the PDB j, i.e. PDBs must be processed strictly *sequentially*.

1.2.2 SHA-2 variants

SHA-512 is the 64-bit variant of SHA-256. All the sizes are therefore doubled, namely the hash size is 512 bit, the block length l is 1024 bit, and the accumulator variables are 64-bit wise. Similarly to SHA-256, their initial values are the first 64 bits of the fractional part of the square root of the first eight prime numbers, and are hard-coded in the standard [89].

SHA-512 performs $R = 80$ iterations to hash a single PDB. This means that 80 values for the K_t constants are required. These values, which are hard-coded in the standard, have been determined analogously to SHA-256, as the first 64 bits of the fractional part of the cube root of the first 80 prime numbers.

The padding step reflects the doubling of the dimensions, since the message is padded with a single 1 followed by as many 0 bits as needed to reach a length congruent to 896 mod 1024, so as to leave the last 128 bits to represent the original length $L(M)$. This means that the limit on the length of the message to be hashed is increased up to $2^{128} - 129$ bit.

The only remaining differences between SHA-512 and SHA-256 involve the expressions of the $\sigma(x)$ and $\Sigma(x)$ functions, which become

$$\begin{aligned}
\sigma_0(x) &= x \ggg_r 1 \oplus x \ggg_r 8 \oplus x \gg 7 \\
\sigma_1(x) &= x \ggg_r 19 \oplus x \ggg_r 61 \oplus x \gg 6 \\
\Sigma_0(x) &= x \ggg_r 28 \oplus x \ggg_r 34 \oplus x \ggg_r 39 \\
\Sigma_1(x) &= x \ggg_r 14 \oplus x \ggg_r 18 \oplus x \ggg_r 41
\end{aligned} \tag{1.14}$$

The other four hash functions are actually variants of SHA-256 and SHA-512. The specifications of these variants are exactly the same as their base algorithms, but the output is truncated to a lower number of bytes and different initial values $DM_0(0)$ to $DM_7(0)$ are employed. Namely:

SHA-224 is the same function as SHA-256, except that the output is truncated to 224 bit and different initial values $DM_0(0)$ to $DM_7(0)$ are used;

SHA-384 is the same function as SHA-512, except that the output is truncated to 384 bit and different initial values $DM_0(0)$ to $DM_7(0)$ are used;

SHA-512/224 is the same function as SHA-512, except that the output is truncated to 224 bit and different initial values $DM_0(0)$ to $DM_7(0)$ are used;

SHA-512/256 is the same function as SHA-512, except that the output is truncated to 256 bit and different initial values $DM_0(0)$ to $DM_7(0)$ are used.

The interested reader is referred to the standard [89] for the selection of the initial values specific to each variant.

The different hash functions of the SHA-2 family are compared in Table 1.1. The first part of the table presents some functional characteristics of the hash algorithms. From this comparison, it is clear that the different variants of SHA-2 offer different combinations of security levels and block size. A symmetric encryption algorithm with equivalent security level is also listed. The second part of the table compares some internal parameters, which depend on the algorithm being derived from the SHA-256 or SHA-512 main variant.

1.3 Applications

The properties of cryptographic hash functions make them one of the most versatile cryptographic tools, used in a variety of security services [107].

Digital signature schemes based on asymmetric encryption, such as Digital Signature Algorithm (DSA) [85], employ hash functions to reduce their computational complexity while retaining the non-repudiation property. Due to the processing time of asymmetric encryption, it would be impractical to encrypt a large file to obtain its digital signature, so only the hash of the file is encrypted. Due to the collision resistance property, it is computationally infeasible to find a second file with the same hash, and hence the same signature, of the original file, therefore the non-repudiation of the signature is preserved.

Hash-Based Message Authentication Codes (HMACs) [10, 91] exploit the collision resistance property to provide message authentication,

Table 1.1: Characteristics of members of the SHA-2 family of algorithms

SHA-*	**256**	**512**	**224**	**384**	**512/224**	**512/256**
Hash size (bit)	256	512	224	384	224	256
Block size (bit)	512	1024	512	1024	1024	1024
Message size (bit)	$< 2^{64}$	$< 2^{128}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$	$< 2^{128}$
Security bits	128	256	112	192	112	128
Corresponding symmetric cipher	AES-128	AES-256	TDES	AES-192	TDES	AES-128
Word size (bit)	32	64	32	64	64	64
Number of rounds	64	80	64	80	80	80
Publication year	2001	2001	2004	2001	2012	2012

which ensures the integrity of the transmitted data. The message to be transmitted, along with a shared secret, is hashed to produce a Message Authentication Code (MAC) which can be used at the receiver side to verify that the message has not been altered. The inclusion of the shared secret, in fact, prevents an attacker from simply replacing the whole message-MAC pair, since the latter cannot be computed without knowing the shared secret. It has been proved that the security of HMAC is directly and formally related with the security of the hash function employed [10]. HMACs are employed by the Internet Protocol Security (IPSec) protocol [56], both in the Authentication Header (AH) [54] and in the Encapsulated Security Payload (ESP) [55] modes.

Pseudo-Random Number Generators (PRNGs) based on Deterministic Random Bit Generation (DRBG) [87] produce pseudo-random sequences by hashing a linearly increasing seed. Apart from being built directly from cryptographic hash functions, PRNGs can be built also from HMACs [87].

Most of the security protocols listed above are standardized by NIST [85, 91, 87], which requires the underlying hash function to be one of its approved hash functions. However, vulnerabilities have been found in SHA-1 [121], leading eventually to its breaking [109]. Given that alternative hash functions such as MD5 [100] were already known to be broken [122, 12], SHA-2 has been the only viable alternative for many

years. Although a new hash function, SHA-3, has been standardised by NIST in 2015 [90], it has not reached widespread diffusion yet. Considering the slow process that has taken place for SHA-2 to fully replace SHA-1 [93], along with the fact that there are no significant vulnerabilities to SHA-2, the new SHA-3 function is not expected to replace SHA-2 in the near future.

Moreover, SHA-3 is based on a completely different mathematical construction from SHA-2. Due to the different mathematical nature of the function, it it difficult to re-use the body of knowledge in SHA-2 and SHA-1 cryptanalysis for the new SHA-3 algorithm, hence more study is required to increase confidence in its cryptographic strength of the new SHA-3 function. On the other hand, SHA-2 has undergone intense cryptanalysis [43] for more than 15 years, and no weaknesses have been found yet, making SHA-2 attractive for innovative applications where long-term collision resistance is required [116].

1.3.1 Blockchains

The blockchain technology employs hash functions to ensure the integrity of a distributed *ledger*. At its very essence, a blockchain is an efficient implementation of a distributed database growing in time within a peer-to-peer network. In a blockchain, database transactions are grouped into blocks, which are concatenated by including the hash of the last block into the header of the next block. New blocks are added to the blockchain by running a distributed consensus algorithm, ensuring a consistent view of the database. Once the block sequence is agreed upon, the collision resistance property of the hash functions implies that it is infeasible to modify a transaction without being detected, since any change in the block would result in a different hash.

A critical role in ensuring the integrity of the distributed database is played by the distributed consensus algorithm. In fact, if an attacker is able to subvert the consensus protocol, they may force the network to agree on their own version of the blockchain, with a content of their own choice. Specifically, consensus protocols running on peer-to-peer networks must avoid *Sybil attacks* [34], i.e. attacks based on the capability of the attacker to present multiple identities, gaining an apparent majority which can drive the consensus. The Bitcoin cryptocurrency [83], which introduced the blockchain technology, bases its own distributed consensus protocol on SHA-256. It avoids Sybil attacks by forcing peers to perform a computationally-intensive task in order to participate in the consensus algorithm [116, 61].

The Bitcoin consensus protocol is also called *Nakamoto consensus* [13]. It mandates that a new block can be added to the network if its hash, computed by applying twice the SHA-256 hash function, is below a specific threshold, called *target*. To this end, the header contains a *nonce* which can be changed by peers in order to alter the hash value of the block. The Nakamoto consensus relies on the one-way property of SHA-256: were the SHA-256 function to be invertible, it would be possible to compute the value of the nonce leading to the required hash value by simply inverting the SHA-256 function [61]. Instead, a brute-force approach is required, with minor possible improvements stemming from details of the Nakamoto consensus protocol [24, 119].

The first node in the network capable of finding a new valid block announces it to the network, and gets the reward associated to the block, which consists of the sum of the fees of the transaction included in the new block, plus a pre-defined quantity of newly mined coins, hence the name "mining" given to the process. The increased interest in the Bitcoin cryptocurrency, especially during the speculative bubble of late 2017 and early 2018, during which the value of a Bitcoin almost topped 20 000 $, has made the mining process an extremely competitive process, paving the way for an entire industry of dedicated miner accelerators [112, 111] with extremely high demands on the underlying SHA-256 circuitry. Not only must such circuits be fast enough to compete profitably within the peer-to-peer network, but they must also be power efficient, in order not to have energy costs exceed mining revenues [92, 118].

But, even more importantly, the success of Bitcoin has raised interest in the potential of the underlying blockchain technology, and its applications beyond digital cryptocurrencies. In fact, investigating new blockchain applications is currently a flourishing research and development field [61]. The Bitcoin blockchain itself is rather limited in what it can be used for, mainly because of its transaction script language which is by design Turing-incomplete [116]. Furthermore, the Nakamoto consensus protocol has undergone some criticism for the large amount of energy it dissipates in a computation yielding no actual result [39]. Therefore, several alternative blockchains have been proposed, and most of them have been implemented. However, only a few of these alternatives are actually completely independent blockchains, or *altchains*, the best known of which being Ethereum [127]. On the contrary, most of the so-called *second-generation blockchains* actually rely on the Bitcoin blockchain itself in some way, mainly in order to avoid destructive attacks from the powerful network of Bitcoin miners [13]. This means that the Bitcoin blockchain, with its reliance on hardware implementations of the

SHA-256 algorithm, is today fundamental for the blockchain industry as a whole.

1.3.2 Internet of Things

The IoT field requires carefully designed hardware accelerators for SHA-2 in order to meet security requirements within its manifold constraints.

The IoT is based on infrastructures of low-end devices able to autonomously communicate across the Internet. It is an enabling paradigm for a number of innovative applications across different fields, each with its own security requirements [61]. Some cases, such as the biomedical sector [49], present strict security requirements due to the involvement of sensitive data, but almost any IoT application must face the threat of external attacks [61].

There are different ways for SHA-2 to contribute to the security of the IoT in the biomedical sector. One example is given in [46], where the pseudo-randomness of SHA-256 is exploited to produce a strong cryptographic key for the encryption of medical images which are to be sent over the cloud. It is worth mentioning that SHA-1 is still approved for use as PRNG [86]; however, authors of [46] opt for SHA-256 in order to obtain a longer, hence more secure, key. Another application is presented in [71], where the issue of black-hole routing attacks [126] is considered. The work [71] includes source node authentication with an HMAC based on the SHA-256 hash function in the routing algorithm, according to the ideas presented in [70], in order to prevent unauthorized nodes from sending malicious routing packets.

An example of IoT application where sensitive data are not involved, but security is relevant anyway, is provided by Radio Frequency Identification (RFID) systems, which are increasingly used in supply chain management to intelligently track parts along the supply chains and manage inventories [31]. A small transponder called *tag* is attached to an item, storing a unique serial number for the item which can be sent to an RFID reader to track the object, or to start more complex interactions between the reader and the tag.

The reading of RFID tags must be secured in order to preserve the privacy of both customers and companies [124]. Hence, a number of proposals for RFID access control have been put forward, mainly relying on hash functions and their one-way property. The scheme proposed in [124] avoids tracking by using a random number r in the tag response, which therefore consist of $(r||DM\,(ID||r))$. An improvement is proposed in [47], where transaction numbers are used to avoid replay attacks, and

the tag ID is updated at every successful transaction in order to prevent traceability. The tag ID is updated also in the scheme of [31], where MACs are used to authenticate both the tag and the reader.

In addition, since the IoT paradigm implies a peer-to-peer network, and a blockchain is essentially a database distributed over a peer-to-peer network, there is an increasing interest in applying the blockchain technology to the IoT field [61, 99, 20, 21]. IoT applications can benefit from the use of a blockchain providing integrity and non-repudiation of communications between the nodes [61, 99]. Furthermore, innovative IoT applications can be built taking advantage of *smart contracts*, which are applications run on a suitable blockchain capable of self-enforcing some business rules [20], or enabling the trading of IoT services directly by the nodes of the IoT network [137].

IoT-related security presents unique challenges due to the characteristics of the devices involved. In fact, a paramount issue in IoT applications is the energy consumption, since devices are usually battery-backed, and the battery life often determines the very lifetime of the device. Furthermore, passively-powered devices like RFID tags are limited in how much electrical *power* they can receive from the reader. This represents an interesting positioning in the design space, as the limiting factor is not posed by the mere energy budget but, inherently, by the instantaneous power that can be supplied. Such an issue is similar to the power cap incurred by high-end processors used in server settings, albeit scaled down to the deeply embedded realm. As a consequence, to meet their energy requirements, IoT devices are often limited in their computing capabilities [101]. On the other hand, this conflicts with the computational requirements of cryptographic primitives, like SHA-2, which are very demanding and place a significant overhead on IoT devices [69, 37, 124]. This trade-off reaches its extreme with blockchain-based applications [21], making it is extremely difficult for sensor nodes to participate in the consensus protocol [20].

Clearly, only carefully designed application-specific accelerators can provide enough flexibility to address the conflicting constraints described above in an effective way.

1.3.3 Trusted Computing

SHA-2 plays also a central role in ensuring mutual trust between computing platforms. A computing platform gains the *trust* of another computing platform when it is capable to prove that it is actually executing the software and/or hardware configuration it is supposed to execute.

This can be accomplished by presenting the other computing platform with an *attestation* of what is being executed, usually in the form of the hash of the source code in execution. Specifically, Intel's Software Guard Extensions (SGX) uses SHA-256 to provide an attestation for software with high security requirements [22]. These concepts, originally developed in the software context, can also be applied to the hardware, specifically reconfigurable hardware, of which the increasing diffusion in uncontrolled environment, like the cloud, comes with the same issue of trust.

In the last few years, there has been an emerging interest in including reconfigurable hardware in cloud systems, as major cloud providers have started having FPGAs in their facilities [19]. This trend stems from the recognition that reconfigurable hardware implementations of commonly used algorithms can achieve better performance and improved power efficiency compared to classical CPU-based implementations [52].

Among the various challenges posed by the employment of reconfigurable hardware in cloud settings, there is the need to secure the provided bitstream. In fact, when the reconfigurable hardware is not under their direct control, users need to be guaranteed that the hardware accelerator synthesized on the remote FPGA is the one they submitted, and has not being tampered with, for example by adding backdoors for leaking user's data. This is particularly relevant when sensitive data is involved [36]. Moreover, it is also necessary to avoid that a single malicious FPGA brings down the entire facility [19]. When the provided bitstream is secured, that the accelerator running on the FPGA can be trusted, and therefore can operate with sensitive data with the guarantee that such data is not leaked.

In [36] an architecture for trusted FPGA in the cloud is proposed, where a hardware SHA-2 accelerator is employed to authenticate the bitstream supplied to the FPGA. A similar system is employed in [48], which uses trusted accelerators on reconfigurable hardware for preventing software running on the CPU from accessing sensitive data by offloading them to the FPGA, although in the latter work the components of the secure system are synthesized on the programmable part of the FPGA along with the user's design, where they are included through a dedicated toolchain.

Interestingly, reconfigurable hardware is also used to implement Trusted Execution Environments (TEEs) similar to SGX, mainly aiming to avoid the need for the manufacturer, e.g. Intel, to directly verify the code running in the TEE [23]. An example of such approaches is proposed in [38], where the case is made for hardware SHA-2 circuits into the trusted pro-

cessor architecture in order to minimize the security overhead. Another proposal is outlined in [23], where a SHA-512 accelerator is employed as a PRNG.

Chapter 2

Classification of SHA-2 Hardware Acceleration Approaches

As discussed in Section 1.3, SHA-2 is employed in a wide range of different domains, each with its own specific requirements. This has lead to a high number of different hardware implementations of the algorithm, with the employment of different techniques tailored to different optimisation objectives. This chapter reviews the technical literature about the SHA-2 hardware implementations, and develops a classification for them.

2.1 Approaches to SHA-2 Acceleration

At the highest level, hardware implementations of SHA-2 can be classified according to two design approaches [41]:

Programmable processor architectures: these implementations follow the general-purpose processor paradigm, with one or more central buses, an Arithmetic and Logic Unit (ALU), an instruction memory, and an ad-hoc microcode; but all these components are designed to perform only the computation of the SHA-2 hash algorithm.

Accelerator architectures: also called *coprocessor* architectures, these implementations follow the approach of performing all the computation directly in hardware.

Table 2.1: Synoptic overview of SHA-2 acceleration solutions. For papers including more than one proposal, the best one is considered.

Approach	Class	Optimisations	Literature Proposals
Programmable processor (Section 2.1.1)	Base (Section 2.1.1)	Base	[32]
		Components Improvement (Section 2.1.3)	[41]
	Unrolled (Section 2.1.1)	Loop Unrolling (Section 2.1.3)	[33]
Accelerator (Section 2.1.2)	Base (Section 2.2.1)	Base	[106, 50, 105, 35]
		Components Improvements (Section 2.1.3)	[82]
		Loop Unrolling (Section 2.1.3)	[25, 77, 3, 135, 6]
		Loop Folding (Section 2.1.3)	[136, 40]
	Shift Register (Section 2.2.2)	Base	[75, 51, 57]
		Components Improvements (Section 2.1.3)	[110, 9]
		Loop Folding (Section 2.1.3)	[102, 60, 59, 125, 14, 1]
	Precomputation-Based (Section 2.2.3)	Variables Precomputation (Section 2.1.3)	[45, 44, 117, 18, 17, 5]
		Components Improvements (Section 2.1.3)	[4]
		Loop Unrolling (Section 2.1.3)	[67, 2]
	Reordering-Based (Section 2.2.4)	Basic Spatial Reordering (Section 2.1.3)	[76]
		Loop Unrolling (Section 2.1.3)	[8]
		Components Improvements (Section 2.1.3)	[80, 81, 78]
	Quasi-Pipelined (Section 2.2.5)	Quasi-Pipelining (Section 2.1.3)	[68, 28, 27, 113]
		Loop Unrolling (Section 2.1.3)	[74]

A further classification is based on the implementation strategy for the computational core of the accelerator. Table 2.1 presents a synoptic overview of the different approaches proposed in the literature to design SHA-2 hardware accelerators.

The remainder of this section will survey the programmable processor architectures and introduce the general components of the accelerator architecture, along with an analysis of the various optimisation techniques for the computational core; whereas the whole Section 2.2 will be focused on reviewing the different SHA-2 accelerator architectures proposed in the literature.

2.1.1 Programmable Processor Architectures

The data path of a SHA-2 implementation based on the programmable processor approach is shown in Fig. 2.1. It includes components commonly used in a processor design, such as an Input/Output (I/O) module, a Control Unit, a Program Counter, a Random Access Memory (RAM) used to hold the working variables and the message M to be hashed, and a Read-Only Memory (ROM) for the K_t constants and the initialisaton values $DM_k\,(0)$. These components are arranged around a central bus, along with components designed specifically for the SHA-2 computation, such as a computation unit dedicated to the expansion of M to produce the W_t words, and one or more computation units for implementing the compressor function. A detailed description of the data path of a processor architecture can be found in [32], where other elements commonly used in general-purpose processors, such as Memory Address Registers and Memory Data Registers, are also used.

In order to be executed by this kind of implementations, the SHA-2 algorithm must be described in terms of *micro-operations*, which are the operations directly executable by the architecture in a clock cycle. Since one iteration of the SHA-2 algorithm usually cannot be executed in a single clock cycle with a processor architecture, the iteration is described by a sequence of more than one micro-operation. Therefore, processor architectures require multiple clock cycles to hash a single message.

Optimisations of the processor architecture may involve the optimisation of some of its components, particularly the arithmetic units. In [41] a 4-input ALU is used to compute Eqs. (1.6) and (1.7). This optimisation is particularly well-suited for FPGAs, which very often provides 4-input Look-Up Tables (LUTs).

A different strategy relies on multiple instances of the same unit, so as to parallelise the execution of different micro-operations. In [33] the

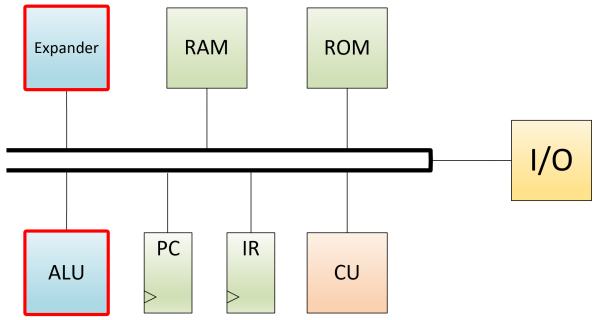


Figure 2.1: General architecture of a SHA-2 processor core. Customly designed components are highlighted.

number of cycles needed by a SHA-2 iteration is reduced by executing two micro-operations of the same iteration simultaneously, taking advantage of multiple arithmetic units.

2.1.2 Accelerator Architectures

Accelerator architectures are built around a combinatorial block, hereafter referred to as the *transformation round core*, which performs the SHA-2 step function as described in Section 1.2.1. The overall equation implemented by a transformation round core can be obtained by combining Eqs. (1.6), (1.7) and (1.11):

$$\begin{aligned}
A_{t+1} &= \Sigma_1(E_t) + Ch(E_t, F_t, G_t) + \Sigma_0(A_t) + Maj(A_t, B_t, C_t) \\
&\quad + H_t + K_t + W_t \\
B_{t+1} &= A_t \\
C_{t+1} &= B_t \\
D_{t+1} &= C_t \\
E_{t+1} &= \Sigma_1(E_t) + Ch(E_t, F_t, G_t) + D_t + H_t + K_t + W_t \\
F_{t+1} &= E_t \\
G_{t+1} &= F_t \\
H_{t+1} &= G_t
\end{aligned} \tag{2.1}$$

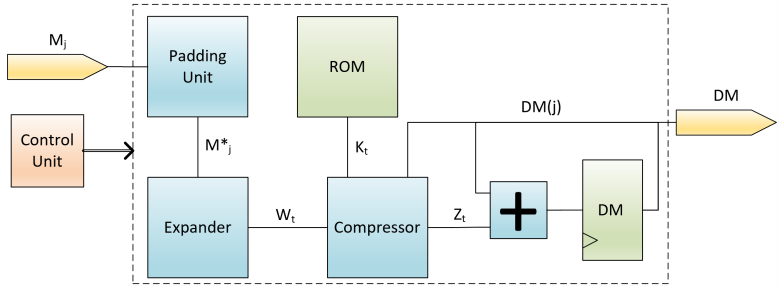


Figure 2.2: General architecture of a SHA-2 accelerator core.

General architecture

The architecture of a SHA-2 accelerator is directly derived from the structure of the algorithm, as described in Section 1.2, and it is shown at a high level in Fig. 2.2.

Prior to hashing, the incoming message needs to be padded. Therefore, a Padding Unit is responsible for producing PDBs. It is worth noting that the Padding Unit modifies only the last block of a message, leaving the others untouched. Since padding can be performed in software quite efficiently without affecting the overall security of the system, many implementations [18, 17, 51, 80, 81, 28, 27, 57, 14, 59, 125, 9] do not include the Padding Unit. These implementations take as input already formatted PDBs.

On the other hand, other implementations [106, 50, 105, 75, 35, 44, 135, 136, 110, 74] choose to include a Padding Unit directly in hardware. In this case, the original messages must be provided to the accelerator, while PDBs will be built internally. Especially for the area occupation metric, it must be taken into account whether a Padding Unit is included in the hardware design when comparing different implementations.

The PDB enters the Expander, which outputs the words W_t according to Eq. (1.4). The implementation of the Expander is usually quite straightforward, and is based on a 16-position shift register to store the required words with the necessary delay. The input of the shift register chain is the result of the computation of Eq. (1.4). By not taking the W_t value from the input of the shift register chain, but rather from one of the delay registers, it is possible to decouple the Expander from the critical path of the Compressor [17].

Most implementations adopt this architecture for the Expander, which is explicitly described in [105, 75, 44, 110, 51, 57, 1] and shown in Fig. 2.3.

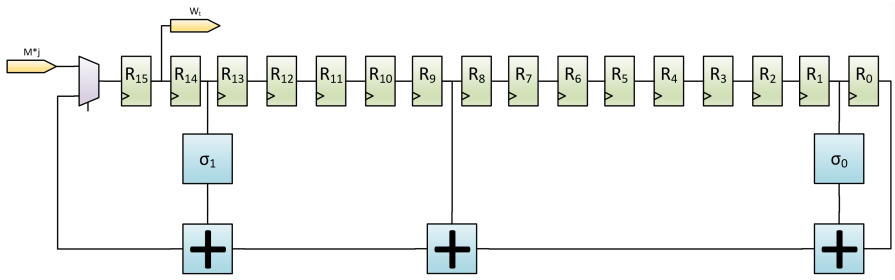


Figure 2.3: Straightforward architecture of the Expander

Possible improvements rely on special devices for implementing the shift register, such as Block RAMs (BRAMs) or First In, First Out (FIFO) memory queues [17], or the shift register mode of the Configurable Logic Blocks (CLBs) of Xilinx FPGAs [45]. A folded implementation of the Expander, employing only one adder, is proposed instead in [60, 59]. The unrolled implementation of the Expander is presented in [8, 74].

In [27], an architectural optimisation of the Expander is proposed, based on the *delay balancing* technique. This technique aims to shorten the critical path by placing more combinatorial logic on non-critical paths. The effect of this reordering is that non-critical paths, which do not determine the worst-case delay of the circuit, are stretched, while the critical path is shortened, resulting in a reduction of the circuit delay.

A ROM unit is employed to store and provide the K_t constants. Multi-mode architectures [105, 44] take advantage of the fact that the values for SHA-256 are the most significative halves of the values for SHA-512, therefore the ROM for the SHA-256 constants can be used also in the SHA-512 data path.

The current hash value needs to be retained until the end of the hashing of the current PDB, since it must be added to the value of the working state Z according to Eq. (1.13). In a straightforward implementation, 8 adders are required to perform this final addition simultaneously. However, in [18, 17] an efficient alternative technique is proposed to compute the new intermediate hash value without adding any latency, since it is calculated during the last stages.

The actual hash computation is performed within the Compressor, which is also where the critical path is located. The transformation round core is responsible for the computation of Eq. (2.1), and its straightforward implementation is shown in Fig. 2.4. However, Eq. (2.1) must be computed R times for each PDB. There are different techniques that

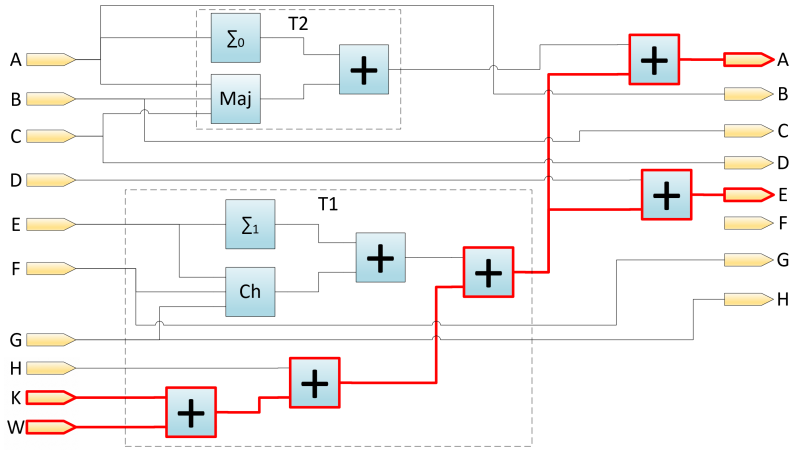


Figure 2.4: A straightforward implementation of the transformation round. The critical path is highlighted.

can be used to perform the whole computation from the transformation round core, which will be described next.

Loop rolling

The *loop rolling* technique consists of implementing an iterative algorithm by using the same component to perform iteratively the same computation. A feedback loop is employed to forward the output of the component to its input, so as to re-apply the function performed by the component.

Architectures employing this technique require $R + 1$ clock cycles to produce a hash value, since an additional clock cycle is employed for performing the last addition described by Eq. (1.13).

Pipelining

The *hardware pipelining* implementation technique consists of instantiating S times the circuitry required to perform a single iteration, with each instance performing a fraction of the total number of iterations. This means that SHA-2 architectures employing pipelining distribute the R rounds required by a single hash calculation onto S pipeline stages, each of which performing R/S iterations. However, within each pipeline stage, loop rolling is still employed to perform the R/S iterations.

Pipelined SHA-2 architectures clearly increase the steady-state throughput, since they are capable of outputting a new hash value every R/S cycle, or even less if loop unrolling is also employed: compared with the non-pipelined implementation employing the same transformation round core, the throughput improvement reaches a factor S.

However, pipelined implementations obviously also incur an area occupation increase by a factor S, and an increase in power consumption due to the presence of more register elements. SHA-2 pipelined architectures also have a non-negligible practical drawback: due to the fact that the computation of the intermediate hash value of the j-th PDB $DM(j)$ requires knowing the previous intermediate hash value $DM(j-1)$ according to Eq. (1.13), and taking into account the fact that the latency of the computation of a single PDB is unaffected by the pipelining technique, which works on multiple PDBs in parallel, *pipeline architectures cannot speed up the computation of a single message*. For this reason, some proposals [115] choose to instantiate multiple loop-rolled SHA-2 cores instead of a single pipelined implementation. SHA-2 architectures which choose to employ pipelining rely on the opportunity of processing different messages at once.

In pipelined SHA-2 architectures, the output of each stage, which is fed as input to the following stage, is stored into the pipeline registers. This does not happen if the transformation round employs spatial reordering, since the pipeline register is moved to the middle of the stage, and stores intermediate values rather than the output of the stage. In these architectures, the input of the following stage is provided directly by the combinatorial part of the previous stage.

2.1.3 Optimisation Techniques

The implementation of the transformation round core can exploit a number of different techniques. Usually, architectures proposed in the literature combine in different ways more than one optimisation technique to take advantage of their combined effect. In fact, some techniques are not particularly impactful by themselves, but their employment turns out to be indispensable for the profitable application of other techniques. For example, loop unrolling and spatial reordering are often employed with the purpose of creating opportunities for the application of variables precomputation or component improvements.

This section discusses in isolation each of the techniques applied in the different hardware implementations of SHA-2. Section 2.2 will then illustrate how these techniques are combined in each full design proposal.

Components improvement

Performance can be improved by replacing the single components which perform the basic operations of the algorithm with more efficient implementations.

For the SHA-2 algorithms, this technique can be applied to the adders. Eq. (2.1) suggests that the two-input adders, which usually are implemented by Carry Look-ahead Adders (CLAs), can be replaced by three-input Carry Save Adder (CSA) with a latency only slightly higher than the latency of a single two-input adder.

However, this replacement is not always profitable, depending to other optimisations being in place, which may change the order of the performed operations. A good criterion to decide whether or not a Carry Save Adder should be used is stated in [80]: the CSA is appropriate if there is an already available input pending to be added to a sum that is being computed at the same time.

Variables precomputation

Some values can be computed well before they are needed, if the inputs on which they depend are already available. If the computation is on the critical path, this can directly reduce the critical path and hence improve throughput. Precomputation can be applied to SHA-2 at two distinct scales.

Looking at the SHA-2 algorithm as a whole, Eq. (1.4) shows that the message schedule does not depend on any intermediate result, and hence can be precomputed so as to make the proper W_t word available when needed. In fact, the first 16 values of W_t are available from the very beginning, and the time needed to compute a W_t value is usually less than the time required by Eq. (2.1). Moreover, due to the fact that the values of the constant K_t are known from the beginning of the computation, the precomputation of W_t allows for the precomputaion of the sums $W_t + K_t$, as done in [80].

Precomputation can also be exploited within the transformation round core, by computing in the current step some values that are not immediately used, but will be consumed by some of the following iterations. This type of precomputation is also favoured by Eq. (1.12), which means that the values needed for computation at the round t are actually available a few rounds earlier. More specifically, Eq. (1.12) implies that the values of the accumulator variables, apart those which are computed in the current round, are available at least one round earlier. By iterating

Eq. (1.12), it turns out that some accumulators are actually available two or even three rounds in advance.

Some examples of this kind of precomputation, also called *operation rescheduling* [18], are illustrated in Section 2.2.3; precomputation within the round also underpins the quasi-pipelining approach, discussed in Section 2.1.3.

Loop unrolling

The combinatorial block can perform more than one iteration of the algorithm in the same clock cycle. A transformation round core employing loop unrolling by a factor UF computes UF subsequent iterations in the same clock cycle, hence reducing the total number of iterations to R/UF.

To achieve loop unrolling by a factor UF, Eq. (2.1) must be rewritten to combine the results of UF subsequent iterations $(t, t-1, \ldots, t-(UF-1))$. Consider for instance an unrolling factor $UF = 2$, meaning that the iterations t and $t-1$ are to be combined. Taking into account Eqs. (1.6) to (1.7), Eq. (1.11) for the step t can be written as

$$\begin{aligned}
A_{t+1} &= {T^1}_t\left(E_t, F_t, G_t, H_t, K_t, W_t\right) + {T^2}_t\left(A_t, B_t, C_t\right) \\
B_{t+1} &= A_t \\
C_{t+1} &= B_t \\
D_{t+1} &= C_t \\
E_{t+1} &= D_t + {T^1}_t\left(E_t, F_t, G_t, H_t, K_t, W_t\right) \\
F_{t+1} &= E_t \\
G_{t+1} &= F_t \\
H_{t+1} &= G_t
\end{aligned} \tag{2.2}$$

and for the step $t-1$ the equation can be written as

$$\begin{aligned}
A_t &= {T^1}_{t-1}\left(E_{t-1}, F_{t-1}, G_{t-1}, H_{t-1}, K_{t-1}, W_{t-1}\right) \\
&\quad + {T^2}_{t-1}\left(A_{t-1}, B_{t-1}, C_{t-1}\right) \\
B_t &= A_{t-1} \\
C_t &= B_{t-1} \\
D_t &= C_{t-1} \\
E_t &= D_{t-1} + {T^1}_{t-1}\left(E_{t-1}, F_{t-1}, G_{t-1}, H_{t-1}, K_{t-1}, W_{t-1}\right) \\
F_t &= E_{t-1} \\
G_t &= F_{t-1} \\
H_t &= G_{t-1}
\end{aligned} \tag{2.3}$$

Combining Eq. (2.2) and Eq. (2.3) yields the equation expressing the value of the accumulators at iteration $t+1$ as functions of the value of the accumulators at the iteration $t-1$, which is the function performed by a transformation round core unrolled by a factor $UF = 2$[1]:

$$\begin{aligned}
{T^1}_{t-1} &= {T^1}_{t-1}\left(E_{t-1}, F_{t-1}, G_{t-1}, H_{t-1}, K_{t-1}, W_{t-1}\right) \\
A_{t+1} &= {T^1}_{t}\left(D_{t-1} + {T^1}_{t-1}, E_{t-1}, F_{t-1}, H_t, K_t, W_t\right) \\
&\quad + {T^2}_{t}\left({T^1}_{t-1} + {T^2}_{t-1}\left(A_{t-1}, B_{t-1}, C_{t-1}\right), A_{t-1}, B_{t-1}\right) \\
B_{t+1} &= {T^1}_{t-1} + {T^2}_{t-1}\left(A_{t-1}, B_{t-1}, C_{t-1}\right) \\
C_{t+1} &= A_{t-1} \\
D_{t+1} &= B_{t-1} \\
E_{t+1} &= C_{t-1} + {T^1}_{t}\left(D_{t-1} + {T^1}_{t-1}, E_{t-1}, F_{t-1}, H_t, K_t, W_t\right) \\
F_{t+1} &= D_{t-1} + {T^1}_{t-1} \\
G_{t+1} &= E_{t-1} \\
H_{t+1} &= F_{t-1}
\end{aligned} \tag{2.4}$$

As shown by Eq. (2.4), loop unrolling increases the critical path, due to the addition of another level of function T^1 followed by another sum in the computation of A_{t+1}. On the other hand, the number of iterations is reduced by a factor equal to the unrolling factor.

What is more, the unrolled loop may expose more opportunities for applying other optimisations that can reduce the critical path, further improving performance. For example, loop unrolling may expose the fact that some values are computed well before they are needed, and this circumstance enables the application of temporal precomputation [80].

Loop folding

Loop folding is the opposite transformation of loop unrolling, since it consists of splitting the execution of one iteration in multiple clock cycles. The advantage of doing so is the possibility of reusing the same functional block to perform different operations in the same iteration, hence reducing the total area occupation.

Usually, architectures employing loop folding incur an increase in latency, due to the steep rise in the number of clock cycles required to perform the whole computation [60]. Nevertheless, [136] proposes a

[1] In Eq. (2.4), the quantity ${T^1}_{t-1}$ is not a distinct value to compute, it has been defined only to make the other equations more readable.

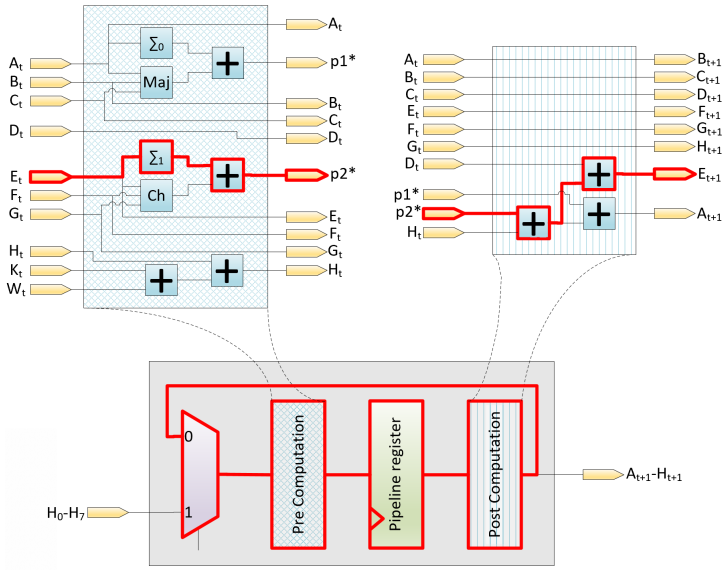


Figure 2.5: The architecture with spatial reordering proposed in [76]. The critical path is highlighted.

rescheduling of operations which avoids any increase in latency. This rescheduling takes into account the data dependencies in the SHA-2 algorithm, which prevent the simultaneous execution of all the additions.

Spatial reordering

In the architecture of the transformation round core, usually the pipeline registers are located at the beginning of the computation to hold the inputs or, more commonly, at the end of the computation to store the outputs. The *spatial reordering* technique, firstly introduced in [133] for SHA-1, consists of moving the pipeline register to the middle of the round, more specifically in the best position so as to obtain a reduction in the critical path by splitting the round itself into balanced, parallel halves.

This optimisation, essentially a form of variable precomputation, is also referred to as the *spatial precomputation* technique. It has the key advantage of avoiding any additional latency, since there are no additional registers.

Quasi-pipelining

Quasi-pipelining is a technique aimed to introduce pipelining within the transformation round core, taking advantage of Eq. (1.12) to perform data forwarding. The model has been formalised in [68], and can be potentially applied to any circuit which can be modeled as:

- a shift register chain of n positions $R_i, i \in [1, n]$;
- a number of combinatorial logic functions ϕ_i, including the identity, each of which taking as input one or more register values;
- a chain of combining operations, each of which being a commutative and associative binary operator to combine the results of the ϕ_i functions, feeding the shift register chain with the result.

Two additional non-shift registers are added and the end of the chain for the K constant and the W expanded word, with index $n+1$ and $n+2$ respectively. The latency of the combining operators, which for SHA-2 always coincide with the modular addition, is assumed greater than the latency of the ϕ_i functions, therefore the critical path runs from the R_{n+2} register through the combining operators chain, ending to the input of R_1.

In order to break this critical path, it may be first necessary to reorder the chain of combining operators, which is possible thanks to their commutativity and associativity properties. To this end, define the ϕ_i block as the couple of each ϕ_i operation and its associated combining operator[2]. Each ϕ_i block is associated with an index I_i constituted by the list of indices of the registers R_j which feed the ϕ_i function. The chain of combining operators can therefore be reordered by sorting the ϕ_i blocks according to the lexicographic order of their indices[3]. For SHA-2, the chain is already well-ordered.

The critical path can now be broken into so-called *quasi-pipeline sections* Q_j, again according to the index I_i of the ϕ_i blocks. Namely, a quasi-pipeline section includes all the ϕ_i blocks sharing the same first number in their index I_i. Quasi-pipelined sections are finally separated by registers.

For SHA-2, the application of the quasi-pipelining technique requires the circuit to be split in two halves due to the feedback in the middle

[2]The ϕ_{n+2} block does not include a combining operator.

[3]$I_i \prec I_j$ if and only if I_i is a prefix of I_j or, possibly after a common prefix, the first differing number of I_i is less than the corresponding number of I_j

of the chain required to compute E_t. The quasi-pipelining technique is then applied as graphically shown in Fig. 2.6 and described below:

$$\begin{array}{lll} \phi_1 = \Sigma_0(R_1) & \Rightarrow I_1 = \{1\} & \\ \phi_2 = Maj(R_1, R_2, R_3) & \Rightarrow I_2 = \{1,2,3\} & Q_1 = \{\phi_1, \phi_2\} \\ \phi_3 = \Sigma_1(R_5) & \Rightarrow I_3 = \{5\} & Q_2 = \{\phi_3, \phi_4\} \\ \phi_4 = Ch(R_5, R_6, R_7) & \Rightarrow I_4 = \{5,6,7\} \quad \Rightarrow & Q_3 = \{\phi_5\} \\ \phi_5 = R_8 & \Rightarrow I_5 = \{8\} & Q_4 = \{\phi_6\} \\ \phi_6 = R_9 & \Rightarrow I_6 = \{9\} & Q_5 = \{\phi_7\} \\ \phi_7 = R_{10} & \Rightarrow I_7 = \{10\} & \end{array} \tag{2.5}$$

Note that for the quasi-pipeline section Q_j, the common first number in the indices is greater than or equal to j, where the former possibility occurs when some numbers lacks as first one. For SHA-2, this happens starting from Q_2, due to the lack of 2 as leading number in the indices.

The quasi-pipeline sections can be optimised by employing the delay balancing technique, which implies in this case that paths shorter than the critical one can be stretched without incurring any performance penalty. For example, there is no need to separate Q_3 and Q_4, since both of them contain one modular addition, while Q_1 and Q_2 contain two modular additions. Similarly, there is no point in having Q_5 as a separate quasi-pipeline section, since it does not include any modular addition. The resulting quasi-pipeline sections are hence:

$$\begin{array}{l} Q_1 = \{\phi_1, \phi_2\} \\ Q_2 = \{\phi_3, \phi_4\} \\ Q_3 = \{\phi_5, \phi_6, \phi_7\} \end{array} \tag{2.6}$$

Taking into account that the quasi-pipeline sections are filled in descending order, at each iteration t the quasi-pipeline section Q_j computes its part of the round $t-q+j$, where q is the total number of quasi-pipeline sections. If $t < q - j$, quasi-pipeline section Q_j is not active, and its registers R_i are not clocked in order to fill the pipeline.

However, and differently from classical pipelining, all the quasi-pipeline sections operate on the same input registers R_i. This creates the need for data forwarding, which is allowed by the shift register configuration and the fact that the chain was lexicographically ordered. An array of *selecting functions* σ_i, i.e. multiplexers, is therefore added to perform data forwarding and completing the circuit.

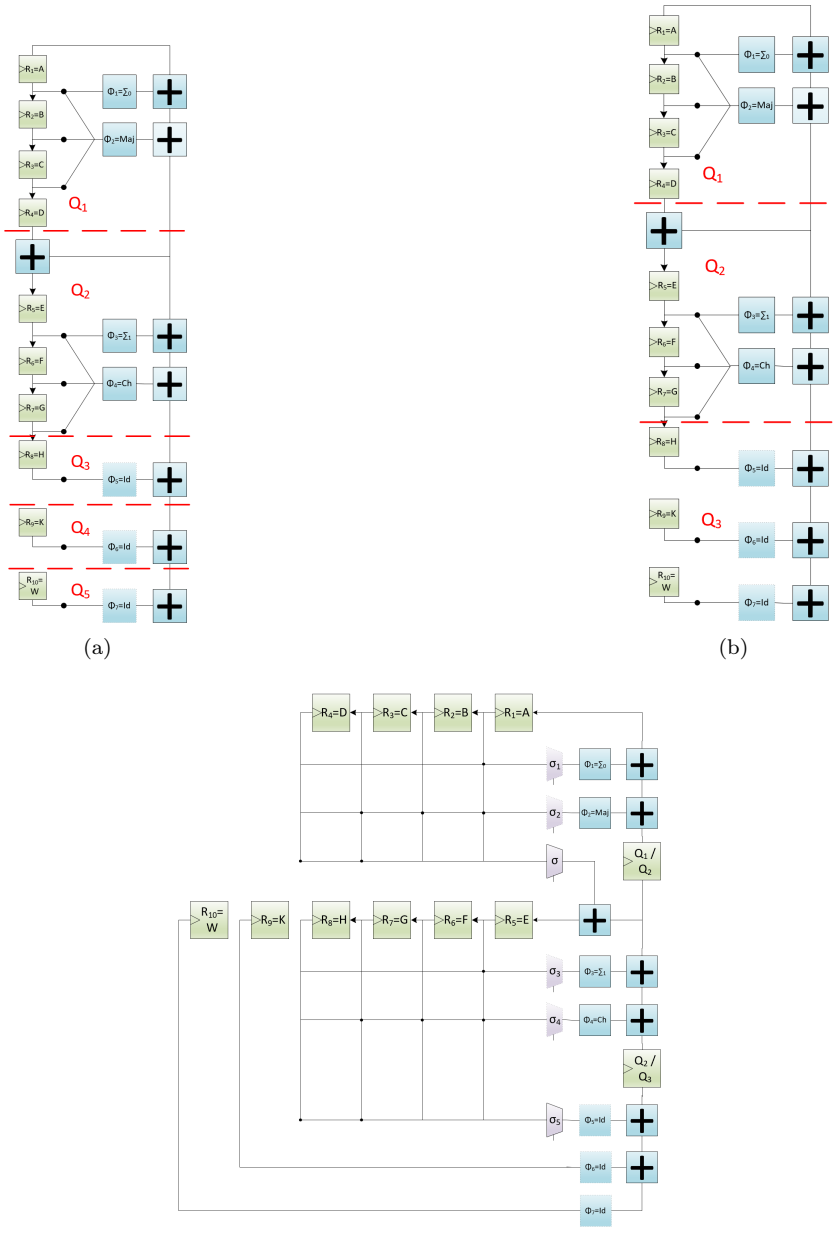


Figure 2.6: Application of Quasi-pipelining in SHA-2: (a) Quasi-pipelined model of SHA-2 with quasi-pipelined sections. (b) Final Quasi-pipelined Sections. (c) Complete Quasi-pipelined SHA-2 circuit. Light border denotes identity functions, which do not correspond to any actual circuits. Note that there are only two actual multiplexers in the final circuit.

2.2 SHA-2 Accelerator Architectures

The vast majority of hardware implementations of SHA-2 follow the coprocessor architecture approach. Despite the fact that it requires a higher design effort compared to the processor architecture approach, a coprocessor implementation can achieve better gains in terms of raw performance and area efficiency [51].

This section surveys the various SHA-2 architectural designs proposed in the technical literature, showing how each of them makes use of the techniques discussed in Section 2.1 to achieve different optimization objectives.

2.2.1 Basic architectures

The work in [106] proposes a hardware implementation of Eqs. (1.6) to (1.11) with dedicated logic, within a transformation round unit. Since the equations employ only bitwise logic operations, shift operations and modular additions, corresponding logic gates, bit reordering and modular adders are used to build up the transformation round unit. This unit is surrounded by a ROM, which provides the K_t constant values, and a number of support units. The Constants Unit supplies the transformation round core with the initialisation values $DM_k\,(0)$. The initial values $DM_k\,(0)$ are hard-coded in the LUTs of the FPGA, while for the subsequent PDBs the initialisation values $DM_k\,(j)$ to be provided by the Constants Unit are updated by the Modified Unit. This unit includes an array of adders which perform Eq. (1.13) to update the intermediate message digest. The W_t values are provided by a dedicated W_t - unit, which in turn is fed by the Padding Unit. A very similar architecture has been implemented in [50] on a Xilinx Virtex-5 FPGA, while [82] advocates the use of parallel adders to implement the additions leading to the computation of A_{t+1} and E_{t+1}.

[105] is a multi-mode variant of [106], where a Control Unit is in charge of reconfiguring the circuit to perform one of the different SHA-2 variants, according to the user's specifications. The most important aspect of [105] is the management of the different word widths of SHA-256 and SHA-512. This is tackled by clearing the least significant 32 bits of the data path when SHA-256 is selected. The multi-mode architecture of [35] takes advantage of the similarities between MD5, SHA-1, and SHA-256 to support all of them. On the other hand, it does not support SHA-512, which is supported by [105]. This exclusion is due to the fact that SHA-512 works on 64-bit words, whereas [35] is a 32-bit architecture.

A straightforward implementation of loop unrolling is presented in [25], where multiple instances of the transformation logic round are placed between registers. This allows for evaluating different values of the unrolling factor.

Loop unrolling by a factor 2 is also exploited in [77] with the aim of decreasing power consumption while increasing parallelism in the round function computation. This architecture is further optimized in [3] where multi-operand additions are compressed by using CSAs. The same optimisation techniques are exploited in [135], where a multi-mode architecture is proposed. This architecture is further optimised in [136] by observing that, due to data dependencies, the whole round function can be computed using only two CSAs without incurring any performance penalty, by properly scheduling the various additions.

In [5], a reordering within the transformation round is proposed. The following variable is defined:

$$\tau_t = H_t + K_t + W_t + D_t \tag{2.7}$$

and substituted into Eq. (2.1), leading to

$$\begin{aligned} A_{t+1} &= \Sigma_0(A_t) + Maj(A_t, B_t, C_t) + \Sigma_1(E_t) + Ch(E_t, F_t, G_t) + \tau_t - D_t \\ E_{t+1} &= \Sigma_1(E_t) + Ch(E_t, F_t, G_t) + \tau_t \end{aligned} \tag{2.8}$$

This architecture is further improved in [4] with the addition of CSAs, while in [6] the application of loop unrolling is explored, with the unrolling factor 4 yielding the best results.

In order to achieve low power consumption, [40] reduces the number of adders and simultaneously clocked registers by employing the loop folding technique. Only one adder is used to perform all the operations of the Compressor and the Expander, which therefore do not operate in parallel and can be disconnected from the clock network accordingly.

2.2.2 Shift register architectures

The architecture proposed in [75] exploits Eq. (1.12) in the implementation of Eq. (2.1). In fact, Eq. (1.12) implies that the accumulators can be chained into a shift registers fashion, where for E_{t+1} the incoming value of D_t is added with the output of the T_t^1 function, as shown in Fig. 2.7. The shift register chain is supplied $T_t^1 + T_t^2$ as input, which is the value of A_{t+1}. The coprocessor architecture proposed in [51] also follows the shift register approach, both for the SHA-256 and the full HMAC-SHA-256 implementations. The multi-mode variant of the shift register approach is presented in [57].

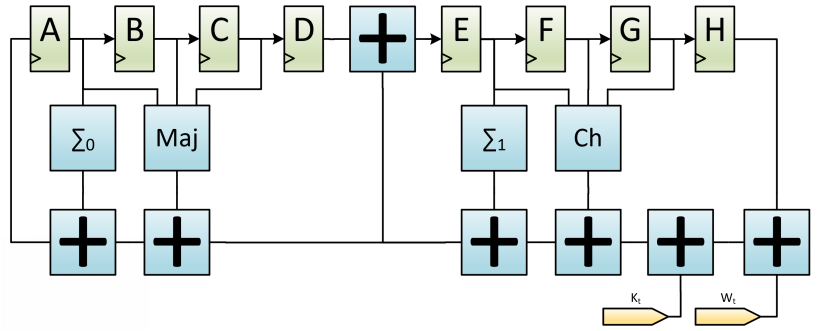


Figure 2.7: Straightforward shift register architecture implementation

The shift register architecture can be optimised to utilise a single adder to perform the final sum. This is illustrated in [102], where an adder from the data path is reused to this end.

The shift register approach is also adopted in [60], which is an architecture specifically tailored to low-power, area-constrained applications. The round function is implemented by reducing the number of operator blocks at the minimum, reusing the same operator block to perform multiple computations. The shift register architecture of the Compressor is therefore modified to work with a single adder, which subsequently adds different operands to compute the round function across several clock cycles, while the H_t register is used as the accumulator for the addition. Interestingly, a similar architecture is adopted also for the Expander, which requires four clock cycles to compute a word, compared with the seven clock cycles of the Compressor. The whole circuit requires 490 clock cycles to fully compute a hash, hence effectively trading throughput for area and power consumption. The multi-mode extension of this architecture is presented in [59].

Another proposal aimed at reducing area occupation, following the shift register architecture, is [1]. In this case, the area reduction is obtained by reducing the word size of the SHA-512 hash function, which is normally 64 bit, to a lower value, taking advantage of the fact that most of the operations involved in the hash algorithm can be computed in a bit-wise fashion. Implementations with the word size reduced to 32 bit, 16 bit and 8 bit are considered. Interestingly, the 32-bit variant achieves 72% of the throughput of the full-word-size implementation, meaning that it is more area efficient than the full-word-size counterpart. The architecture described in [14] also reduces the word width of the circuit

to 8 bit, exploiting also loop folding to achieve further area reduction.

A different way to improve the shift register architecture involves the use of parallel adders for the adder chain. In [110] the adder chain is implemented with 5-to-3 parallel adders, while [9] employs a single 7-3-2 parallel adder to compute A_{t+1}. The architecture proposed in [125] combines the use of parallel adders with the loop folding technique, in order to reduce area occupation limiting the throughput penalty, therefore maximizing the area efficiency.

2.2.3 Architectures with precomputation

The architecture proposed in [45] precomputes the sum K_t+W_t out of the main operational block, in order to shorten the critical path. Moreover, it employs a 5-to-3 parallel adder to compute A_{t+1}. The architecture is further optimised in [67], where loop unrolling by a factor 5 is used. Values required later in the unrolled chain of operations are precomputed as soon as possible.

Variable precomputation is employed in [18] to shorten the critical path at the iteration t. Taking into account that $H_t = G_{t-1}$ due to Eq. (2.1), and that the values K_t and W_t are known well before they are needed, the value

$$\begin{aligned} \delta_t &= H_t + K_t + W_t \\ &= G_{t-1} + K_t + W_t \end{aligned} \tag{2.9}$$

can be precomputed during round $t-1$, leading to the following computation during round t:

$$\begin{aligned} A_{t+1} &= \Sigma_0 (A_t) + Maj (A_t, B_t, C_t) + \Sigma_1 (E_t) + Ch (E_t, F_t, G_t) + \delta_t \\ E_{t+1} &= D_t + \Sigma_1 (E_t) + Ch (E_t, F_t, G_t) + \delta_t \end{aligned} \tag{2.10}$$

This architecture is further improved in [17] with two optimised variants for the Expander, relying upon BRAMs and FIFOs respectively.

The work in [5] pushes this approach further, proposing another precomputation based on the fact that $D_t = C_{t-1}$ again due to Eq. (2.1). The value τ_t defined in Eq. (2.7) can be precomputed during round $t-1$ as[4]

$$\begin{aligned} \tau_t &= \delta_t + D_k \\ &= \delta_t + C_{t-1} \end{aligned} \tag{2.11}$$

[4]To stress the fact that Eq. (2.11) is computed at round $t-1$ while Eq. (2.7) is computed at round t, [5] calls the latter δ'_t instead of τ_t

The computation of round t can therefore be reduced to

$$\begin{aligned} A_{t+1} &= \Sigma_0(A_t) + Maj(A_t, B_t, C_t) + \Sigma_1(E_t) + Ch(E_t, F_t, G_t) + \delta_t \\ E_{t+1} &= \Sigma_1(E_t) + Ch(E_t, F_t, G_t) + \tau_t \end{aligned} \tag{2.12}$$

In [5] three architectures are compared against the straightforward implementation of Eqs. (1.6) to (1.11) with a common platform, which employs a rolling loop to perform the whole SHA-256 computation. These architectures are the one resulting from Eqs. (2.7) and (2.8) originally proposed in [18], the one resulting from Eqs. (2.9) and (2.10) and the one resulting from Eqs. (2.11) and (2.12).

The architectures are further optimised in [4] by employing CSAs. Moreover, the common evaluation platform is improved by inserting two registers to break the critical path without requiring any additional clock cycle. These additional registers are located after the ROM memory storing the values or the K_t constant, and before the $DM(j)$ feedback loop multiplexer.

The multi-mode architecture proposed by [44] also exploits the precomputation of δ_t and τ_t, computing these two in parallel as

$$\begin{aligned} \delta_t &= H_t + K_t + W_t \\ &= G_{t-1} + K_t + W_t \\ \tau_t &= H_t + K_t + W_t + D_t \\ &= G_{t-1} + K_t + W_t + C_{t-1} \end{aligned} \tag{2.13}$$

While the first equation is the same as Eq. (2.9), the second equation is implemented by computing $G_{t-1}+C_{t-1}$ in parallel with the sum K_t+W_t, and the result of the latter is added, in parallel, to both the former and G_{t-1}.

The architecture presented in [117] exploits variable precomputation even further, in order to introduce a form of pipelining within the transformation round. In the first stage, Eq. (2.9) is computed and stored into a register, while in the second stage computes Eq. (1.6). A similarly aggressive precomputation is performed in [2] in the context of a two-unrolled architecture.

2.2.4 Architectures with spatial reordering

[76] introduces the use of spatial reordering within the design of the transformation round core for SHA-2. The computation of

$$\begin{aligned} P1_t^* &= \Sigma_0\left(A_t\right) + Maj\left(A_t, B_t, C_t\right) \\ P2_t^* &= \Sigma_1\left(E_t\right) + Ch\left(E_t, F_t, G_t\right) \\ H_t^* &= H_t + K_t + W_t \end{aligned} \tag{2.14}$$

is performed before the pipeline registers, while the computation of

$$\begin{aligned} A_{t+1} &= P1_t^* + P2_t^* + H_t^* \\ E_{t+1} &= D_t + P1_t^* \end{aligned} \tag{2.15}$$

is performed after the register, along with Eq. (1.12). The reordering implies that the critical path includes the two adders for the computation of A_{t+1} and the following adder for the computation of $P1_t^*$, along with the Maj function.

The same authors propose in [80] a methodology for the optimisation of the block responsible for the computation of the round function, built around the spatial reordering technique. This methodology takes also advantage of loop unrolling, component improvements and variables precomputation, and leads to the following computation ahead of the pipeline registers:

$$\begin{aligned} p1_{t+1} &= \Sigma_0\left(A_{t-1}\right) + Maj\left(A_{t-1}, B_{t-1}, C_{t-1}\right) \\ p2_{t+1} &= \left(D_{t-1} + H_{t-1} + \left(K+W\right)_{t-1}\right) + \Sigma_1\left(E_{t-1}\right) + Ch\left(E_{t-1}, F_{t-1}, G_{t-1}\right) \\ p3_{t+1} &= D_{t-1} + \left(K+W\right)_{t-1} + \Sigma_1\left(E_{t-1}\right) + Ch\left(E_{t-1}, F_{t-1}, G_{t-1}\right) \\ p4_{t+1} &= \left(K+W\right)_t + G_{t-1} \\ p5_{t+1} &= p4_{t-1} + C_{t-1} \\ p6_{t+1} &= E_{t-1} + Ch\left(p2_{t+1}, E_{t-1}, F_{t-1}\right) \end{aligned} \tag{2.16}$$

In the above equation, $(K+W)$ denotes that the sum is pre-computed out of the operational block, due to data prefetching. Note also that $p2_{t+1}$ is not computed from $p3_{t+1}$, due to the fact that the sum $D_{t-1} + H_{t-1} + (K+W)_{t-1}$ is performed by a CSA. The results of Eq. (2.16) are stored in the pipeline registers along with A_{t-1}, B_{t-1}, E_{t-1} and F_{t-1},

allowing for the following computation after the pipeline registers:

$$\begin{aligned}
A_{t+1} &= \Sigma_0\left(B_{t+1}\right) + Maj\left(B_{t+1}, A_{t-1}, B_{t-1}\right) + \left(p4_{t+1} + p6_{t+1} + \Sigma_1\left(p2_{t+1}\right)\right) \\
B_{t+1} &= p3_{t+1} + p1_{t+1} \\
C_{t+1} &= A_{t-1} \\
D_{t+1} &= B_{t-1} \\
E_{t+1} &= p6_{t+1} + \Sigma_1\left(p2_{t+1}\right) + p5_{t+1} \\
F_{t+1} &= p2_{t+1} \\
G_{t+1} &= E_{t-1} \\
H_{t+1} &= F_{t-1}
\end{aligned} \tag{2.17}$$

It is worth noting that, although [80] proposes a methodology made up of a sequence of techniques to be applied in order to obtain the optimised hash core, the application of many of the techniques is not straightforward, but requires a careful analysis of the circuit by the designer. The methodology is further improved in [81], most notably by adding recursion, obtaining an even improved hash core. This latter version is evaluated against different FPGA platforms in [8], where the corresponding architecture of the Expander is also presented. Finally, [78] presents the multi-mode hash accelerator based on the same techniques.

2.2.5 Architectures with quasi-pipelining

The first quasi-pipelined architecture for SHA-2 is introduced in [28], together with an improved variant employing delay-balancing. A slightly different version is presented in [27], where the Expander is also optimised with the delay balancing technique. An independent theoretical analysis carried out in [66] confirms that this design is optimal, with respect to the throughput, at the architectural level.

Unrolling by factors 2 and 4 of the quasi-pipelining architecture is presented in [74], obtaining an improvement in throughput only for SHA-512 with unrolling factor 2.

Without applying the fully-structured quasi-pipelined model described in Section 2.1.3, the work in [113] exploits the same basic ideas of splitting the adder chain with registers, and exploiting Eq. (1.12) for data forwarding, in this case of the E_t accumulator. This work also employs a form of precomputation by summing the value K_t with the word W_t directly in the Expander, so as to remove this sum from the critical path.

Chapter 3

Evaluation of SHA-2 Hardware Acceleration Approaches

FROM the discussion of Section 2.1 it is clear that accelerator architectures follow the same approach to the hardware implementation, i.e. focusing on the optimisation of the transformation round core, then building the whole hashing circuit around the optimised round core. This raises the question of determining the best implementation, a question which does not have a single, generally valid answer. In fact, the best implementation can only be determined once the optimisation objective have been set, since different implementations optimises different aspects of the design.

This chapter presents an evaluation methodology for the systematic evaluation of different SHA-2 accelerator designs, based on the usage of a common evaluation platform which reduces the implementation effort required by that comparison. Finally, the impact of each optimisation technique on application metrics is discussed, from both theoretical analysis and the experimental results; and this impact is linked to the application requirements listed in Section 1.3.

3.1 The Need for a Common Evaluation Platform

As explained in Section 2.2, each SHA-2 accelerator design exploits one or, more commonly, more than one of the optimisation techniques described in Section 2.1.3.

Despite the common approaches and the functional compatibility between the various designs, it is still challenging to compare architectures presented in different articles, for two main reasons.

First, each work develops its own control and supporting circuitry, which influences the reported performance of the proposal. Second, the experimental results are also influenced by the specific target technology and synthesis toolchain, whose impact is deeply intertwined with purely architectural aspects.

To face these issues, a common evaluation platform called *SHA-2 workbench* has been developed as a flexible, easy-to-use exploration framework for the evaluation and comparison of different alternatives for the hardware implementation of SHA-2. Such platform provides a control and supporting circuit flexible enough to accommodate a wide range of different designs, compared to similarly aimed comparison platforms which can be found in previous works, such as [5, 4], where only a limited number of design techniques are supported.

3.1.1 Evaluation Methodology

A particular implementation of the transformation round core which is captured in an HDL can be easily plugged into the framework and synthesized for a given target. The common evaluation platform facilitates the task of comparing different designs factoring out the impact of the target hardware technology and related software toolchain, which typically introduce a great deal of variability and unpredictability in the design performance. The workbench also ensures that the obtained results solely depend on the optimisation techniques implemented by the design proposal of the round core, effectively supporting an extensive architectural exploration for SHA-2 implementations.

An architecture which fulfills a given set of constraints can be found according to the process outlined in Fig. 3.1. Once the designer has chosen a transformation round core, a full hash circuit architecture can be obtained by simply tuning the parameters of the architecture. The resulting circuit can be implemented against a target FPGA. If all the design constraints are met, the exploration stops. Otherwise, another iteration of the exploration is performed, where the same transformation round core can be evaluated with different architectural parameters.

The designer may also choose to insert a newly developed transformation round core in the evaluation loop. In fact, the proposed SHA-2 workbench also facilitates the development of new transformation round cores, since the designer can focus solely on the implementation of their own optimisations, then properly configuring the framework to obtain a complete hash circuit.

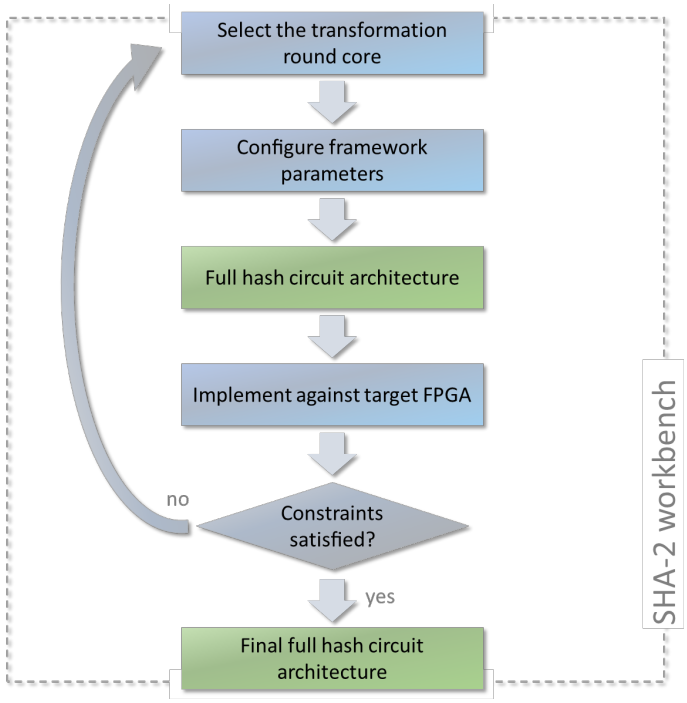


Figure 3.1: Methodology for comparing different hash circuit architectures

3.2 Workbench Architecture

The architecture of the proposed evaluation platform implements a SHA-2 hash core which takes as input a full PDB and produces as output the corresponding hash value. The framework can be configured to produce a SHA-256 or SHA-512 hash core. Message padding and generation of PDBs must be performed externally to the SHA-2 core. Figure 3.2 shows the data path of the proposed workbench.

Once the chosen implementation of the transformation round core is plugged into the workbench, the overall SHA-2 core is built employing the loop rolling or the pipelining techniques described in Section 2.1.2, according to a configuration parameter. When the former is chosen, the combinatorial circuitry is instantiated only once, whereas with the latter configuration, the workbench replicates the same combinatorial block multiple times.

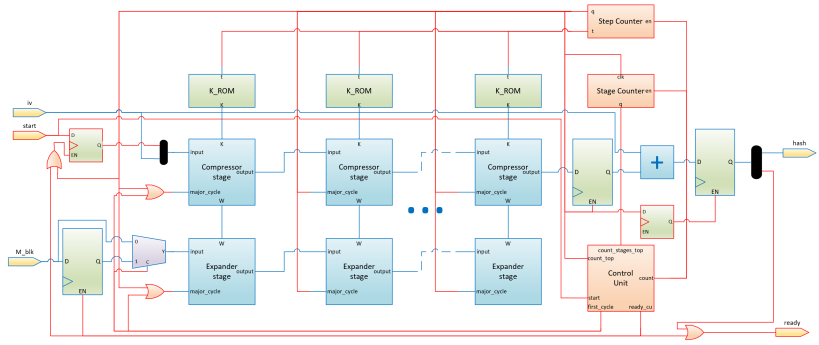


Figure 3.2: Top level entity of the proposed evaluation platform. Operative part is highlighted in blue, Control part in red.

The architecture is composed by two parallel paths, which become two parallel pipelines when pipelining is enabled, one for the Compressor and one for the Expander. The number of pipeline stages can be configured, and can be also set to 1 to disable pipelining at all. For each stage of the Compressor pipeline, there is also an associated ROM containing the values of the K_t constant relevant for the stage.

3.2.1 Compressor

The round registers, clocked by the external base clock, are employed also as pipeline registers. The two functions are controlled by a multiplexer, placed before the register within the round, and driven by a *major cycle* signal. The major cycle signal is produced by the round counter, which also outputs the address input for the ROMs.

The compressor pipeline registers is expected to contain at least the 8 working variables and a validity flag, which is set during the first stage and is carried until output, to signal that the value of the output hash register is meaningful. If required by the transformation round core, the compressor pipeline can also be configured to have the compressor pipeline registers containing additional working variables. When this is the case, an initialisation unit - not shown in Fig. 3.2 - is instantiated before the Compressor pipeline to compute the initial values for these variables. This is described in greater detail in Section 3.2.4.

Each stage of the Compressor pipeline is an instance of the transformation round core selected by the designer. The compressor pipeline may be configured to work with a transformation round core which employs

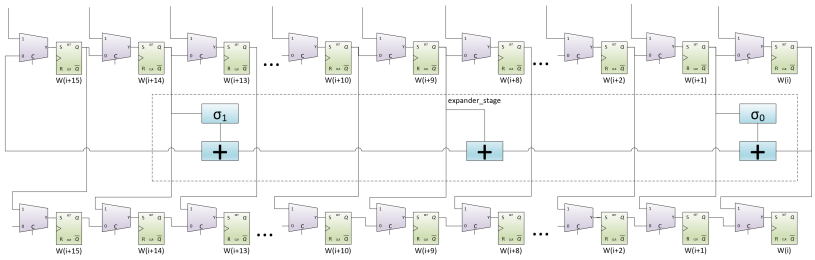


Figure 3.3: Expander architecture, with stage chaining

loop unrolling and system-level data prefetching. If the latter optimisation is to be used, initial values for the K_t and W_t parameters, not used by the first stage, are forwarded to the initialisation unit.

The compressor ends with the chaining sum, which may be configured to be placed into a separate stage. Both the optional final stage, and the initialisation unit, work as separate stage even if pipelining of the transformation round core is disabled.

3.2.2 Expander

Within the Expander, the round registers work as 16-position word-wide shift registers during the stage, turned into parallel registers when the major cycle signal is asserted, again by means of a multiplexer array. Since the last shift of the stage works with the major cycle signal asserted, it is not written to the shift register. Instead, it must be captured by properly rearranging the connection with the register of the following stage, as shown in Fig. 3.3.

To perform unrolling, the shift register chain of each Expander stage is split into a number of chains being equal to the unrolling factor, as shown in Fig. 3.4, since that number of expanded words W_t must be generated at each clock cycle. Words are distributed among splitted chains cyclically with respect to their positions within the original chain.

According to Eq. (1.4), the 16 initial words which the input message is splitted into are in big-endian order, causing a reverse sorting of the input message, which must be taken in little-endian order. This creates the need to reversing the input of the Expander, and this reversal must in turn be taken into account when splitting the expander into stages.

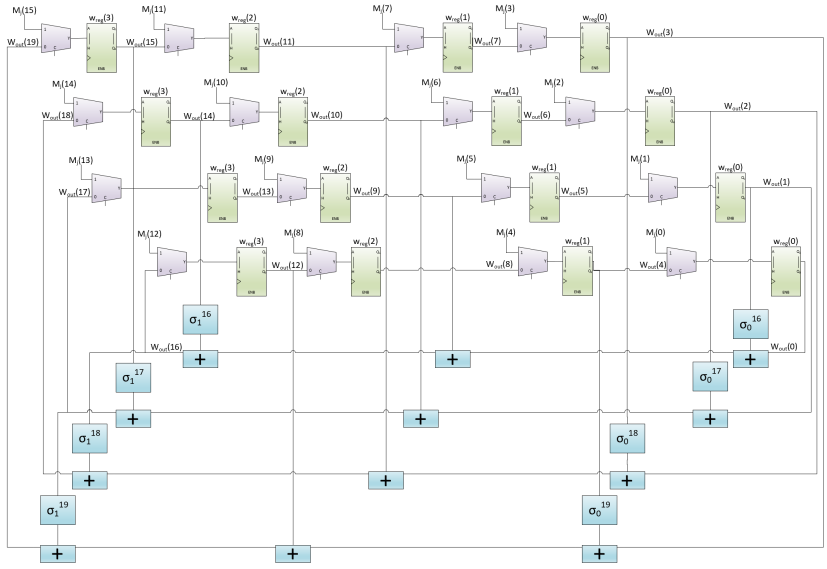


Figure 3.4: Expander architecture, unrolled of a factor 4

3.2.3 Control Unit

The Control Unit is responsible for properly driving the internal signals of the circuitry. It provides for the correct loading of the two pipelines at the very beginning of the operations of the circuit, when the major cycle signal is not active yet. Moreover, it properly enables and clears the round counter.

Basically, the Control Unit should only keep the round counter enabled during the computation and reset it at the end. Hence, in principle, it should be made only of two states, `idle` and `compute`.

An additional `last_stages` state is added to flush the pipeline when no new PDBs are provided to the core for hashing. Since the number of pipeline stages is configurable, it is not viable to employ a Finite State Machine (FSM) state per pipeline stage. Instead, a stage counter is employed, fed by the major clock cycle and enabled during the `last_stages` state. As for the round counter, the counting value of the stage counter can be computed from the values of the `generic` parameters of the design. When the round counter signals that the pipeline is fully flushed, the FSM can go into the `idle` state.

The Control Unit is responsible also for properly driving the timing

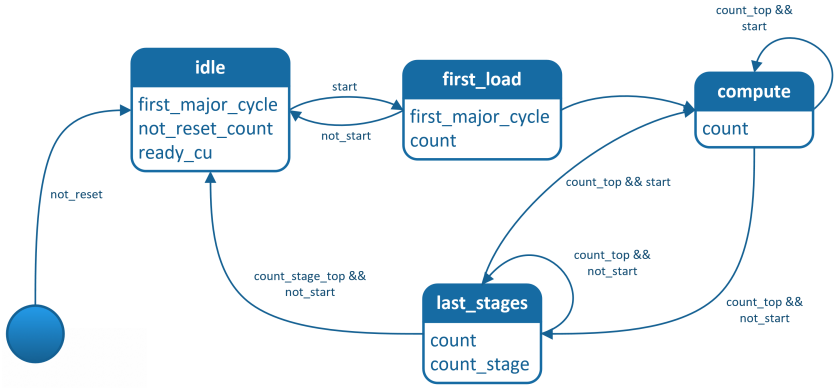


Figure 3.5: FSM of the Control Unit with the Compressor and the Expander aligned

of the circuitry. Due to the presence of the multiplexers, it is possible to load the pipeline only when the major cycle signal is asserted, but since upon reset the major cycle signal is cleared, the very first major cycle would be lost. To avoid this, the Control Unit introduces two control signals, which enable the Compressor and the Expander respectively to receive incoming data also during the very first major cycle. This is done during an additional stage, `first_load`, as shown in Fig. 3.5.

In this version of the FSM, the control signals for initialising the Compressor and the Expander pipeline are actually the same signal. However, depending on the chosen architecture of the transformation round, the data path may incur a further timing issue, related to the constant ROM. Such cases are handled by the Control Unit via an additional stage, `second_load`, which delays only the Compressor pipeline by one clock cycle, preventing it from accepting inputs. To work correctly, the Expander must instead start working, hence the need of differentiating the two control signals. The modified FSM is shown in Fig. 3.6, whilst Section 3.2.4 discusses the conditions on the round architecture under which the modified FSM must be instantiated.

3.2.4 Reconfigurable aspects controlled by source-level parameters

The following characteristics of the architecture can be reconfigured by setting the corresponding `generic` parameter in the HDL code, which has been written in the VHSIC Hardware Description Language (VHDL):

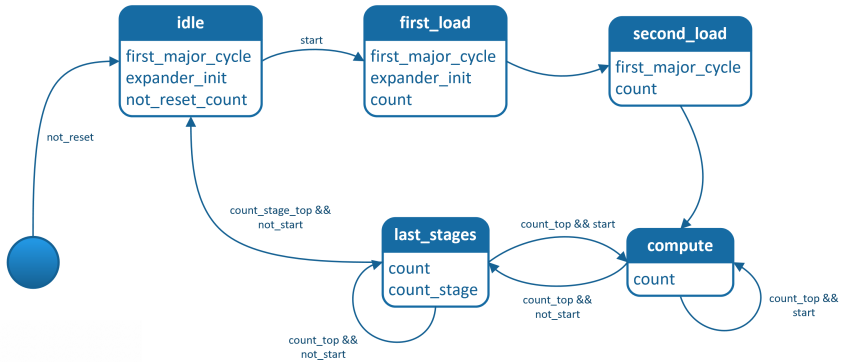


Figure 3.6: FSM of the Control Unit with the Expander moved ahead

Hash size The selection of the hash size implies the selection of the hash function to perform. The `WIDTH` parameter can hence be set to 256 to select SHA-256, or 512 to select SHA-512. The word width of the circuit is set appropriately based on the value of the `WIDTH` parameter.

Number of pipeline stages The number of pipeline stages can be set directly as the value of the `PIPELINE_STAGES` parameter. Set this parameter to 1 to disable pipelining. This value is also the number of PDBs which can be processed at the same time by the architecture. The number of pipeline stages must divide the number R of rounds[1].

Unrolling factor The unrolling factor of the design must be set as the value of the `UNROLLING_FACTOR` parameter. Set this parameter to 1 to disable unrolling. Note that this value must be set consistently with the unrolling factor of the selected internal transformation round, otherwise the architecture will not work. Moreover, the unrolling factor must divide the number R of rounds[2].

Working variables The parameter `PIPELINE_WIDTH`, which is expressed as number of words, must be at least 8, since the Compressor pipeline register must hold the 8 SHA-2 accumulators. However, an optimisation may use additional working variables, as done in [76, 80, 81]. The additional working variables can be stored in the pipeline register, if nec-

[1]This constraint is checked by an `assert` in the code.
[2]The latter constraint is checked by an `assert`.

essary, by specifying a number of words greater than 8 in the `PIPELINE_WIDTH` parameter.

When there are additional working variables shared between rounds, hence requiring to be stored in the pipeline register, these variables need to be initialised for the first round, possibly from the initial values of the accumulators and/or the initial values of K_t and W_t. To do so, when the value of `PIPELINE_WIDTH` is greater than 8, an instance of the VHDL `entity` called `Initialisation_block` is instantiated, which is expected to provide the initial values for the additional working variables. Hence, an implementation of the transformation round which requires additional working variables is expected to provide an `architecture` for the `Initialisation_block` VHDL `entity`.

The initialisation block effectivelty acts as an additional pipeline stage, not included in the `PIPELINE_STAGES` figure, and lasting only one clock cycle. Due to the pipelined architecture, the additional clock cycle affects only the latency of the circuit, and not its throughput, even if `PIPELINE_STAGES` is set to 1.

Prefetch shift The value of the parameter `PREFETCH_STEPS` must be set to 0 unless data prefetching is in place. When the value of this parameter is greater than 0, the values of K_t and W_t fed to the transformation round at each round are anticipated to `PREFETCH_STEPS` rounds. Put another way, at the round t the transformation core is fed with the values of K_t and W_t corresponding to the round $t +$ `PREFETCH_STEPS`. The value of `PREFETCH_STEPS` must be set irrespectively to the value of `UNROLLING_FACTOR`, since the actual value is internally adjusted so as to take into account the unrolling. The first `PREFETCH_STEPS` values of K_t and W_t are also made available to the initialisation block for precomputing the initial values of the additional working variables, if present.

Timing The boolean parameter `FIX_TIME` allows for reconfiguring the timing of the part of the design which provides K_t and W_t constants, as required by the standard [89].

Due to the fact that the K_t ROM takes as input the value of the stage counter, there is a one-cycle delay between the counter increment and the corresponding value of the constant. If the pipeline register is placed before any use of the K_t constants, as it happens in the `Naive` transformation round architecture, the pipeline register itself compensates for the delay and hence no further action is required, so `FIX_TIME` must be set to `false`. Otherwise, if the value of K_t is used before the pipeline

registers, as in transformation round architectures which make use of the precomputation technique, it is required to introduce a one-cycle delay to align the value of K_t with the other operands.

This is done by the Control Unit as described in Section 3.2.3, and this additional clock cycle impacts only the first computation, hence not affecting steady-state throughput. To instantiate the appropriate FSM for architectures which employ precomputation, `FIX_TIME` must be set to `true`. When this is the case, the major cycle of the Compressor is delayed by one clock cycle, by means of a flip-flop, in order to keep the other stages aligned with the first stage, which is the only one directly fixed by the Control Unit.

Final sum If the `FINAL_SUM` boolean parameter is set to `true`, an additional register is placed just before the final sum, actually resulting in an additional stage, which is not included in the `PIPELINE_STAGES` figure. The output register of this stage is also the output register of the whole circuit. This stage requires only one clock cycle, but if its output register had been enabled by the major cycle signal, its latency would have been the same of the other pipeline stages, which is $^{R}/_{(S \cdot UF)}$. This is avoided by using as enable signal for the output register a one-cycle-delayed version of the major cycle signal. Due to the pipelined architecture, throughput is not affected by the presence of the final stage, even if `PIPELINE_STAGES` is set to 1.

If the architecture employs spatial reordering and at least one adder is placed before the pipeline register, however, it is not profitable to add the final stage, hence `FINAL_SUM` can be set to `false`. When this is the case, the output of the last stage is directly fed into the adders.

It is worth noting that, when an adder is placed before the registers, it is most likely the one which performs the $H_t + W_t$ or the $H_t + K_t$ sums. As a consequence, an architecture which sets `FINAL_SUM` to `false` usually also needs to set `FIX_TIME` to `true`. However, it was chosen to keep these two parameters distinct in order to provide greater flexibility for new optimised implementations of the transformation round.

3.2.5 Reconfigurable aspects controlled by component declarations

The architecture of the `Compressor_pipeline_stage` component can be specified by a VHDL `configuration` declaration in order to configure the transformation round. Alternatives are implemented as different architectures of the `Transf_round` entity.

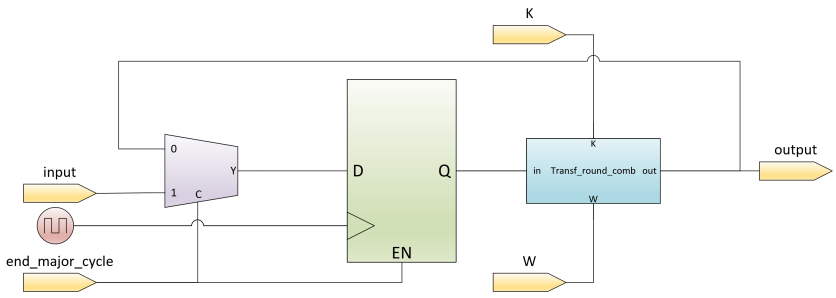


Figure 3.7: Architecture of the `Naive` transformation round core

The following transformation round architectures are provided:

`Naive` A straightforward implementation of the transformation round, with the architecture illustrated in Fig. 3.7. The pipeline register is placed before all the combinatorial parts so as to compensate for the delay of the constants ROM, as discussed in Section 3.2.4.

`Precomputed_UF1` An implementation of the round function with precomputation, presented in [4].

`Reordering_UF1` An architecture with precomputation and spatial reordering, presented in [76].

`Reordering_UF2` An architecture with precomputation, spatial reordering, unrolling and CSAs, presented in [81]

When using the `Naive` transformation round core, the combinatorial part can be further customised by specifying an architecture for the `Transf_round_comb` component, shown in Fig. 3.7, which must implement the round function. The following architectures for the combinatorial component are provided:

`Naive` A straightforward implementation of the round function.

`Unrolled` An implementation of the round function unrolled by a factor 4.

An optimisation of the transformation round which is entirely combinatorial should be implememented as an architecture of the `Transf_round_core` entity for usage within the `Naive` architecture of the `Transf_round` entity. This way the timing issue described in Section 3.2.4 is

avoided. On the other hand, if an optimisation involves the pipeline register, this must be implemented as an architecture for the `Transf_round` VHDL entity.

3.2.6 Discussion

The main aim of the proposed SHA-2 workbench is to provide a fully configurable design solution to be used to explore and evaluate existing and possibly new architectures.

Most of the previous literature proposals for SHA-2 can in fact be seen as a particular instance of the configurable solution presented here. Only architectures that completely redefine the data path of the SHA-2 core, such as [68], are not suitable to be described by the framework presented in this chapter.

Furthermore, works that take advantage of the particular application of the SHA-2 function to enhance the performance of the overall system, such as [81], also fall partially out of the scope of the framework. Such architectures can be integrated into our framework for the part regarding the SHA-2 design, but they cannot exploit the benefits of the particular application. This is an intended goal of the research effort, which aims to make it easier and fairer to compare SHA-2 implementations on their own.

3.3 Experimental Results

The proposed architecture has been synthesized, placed, and routed with the Xilinx Vivado IDE 2017.4 for an extensive range of configurations. Since the proposed design is meant to be used as a part of a larger system, the VHDL description has been synthesized in *Out of Context* mode.

3.3.1 Design comparison against a specific target

Different design alternatives have been analyzed assuming the same target technology, in order to show how the proposed framework can help in comparing fairly different designs on the same target technology, allowing the designer to identify the best one for a given platform.

The target platform considered in this section is the Xilinx Kintex UltraScale+ XCKU5P, which is a 16 nm FPGA featuring more than 200k LUTs [130, 131].

Table 3.1: Detail of the architectures explored. The number of stages S is the value of the `PIPELINE_STAGES` `generic` parameter, while the unrolling factor UF is the value of the `UNROLLING_FACTOR` parameter.

Nº	Core type	S	UF	PIPELINE_WORDS	PREFETCH_STEPS	FIX_TIME	FINAL_SUM_AS_STAGE
1	`Naive`	1	1	8	0	`false`	`true`
2	`Naive`	1	4	8	0	`false`	`true`
3	`Naive`	4	1	8	0	`false`	`true`
4	`Naive`	4	4	8	0	`false`	`true`
5	`Precomputed_UF1`	1	1	8	0	`true`	`true`
6	`Precomputed_UF1`	4	1	8	0	`true`	`true`
7	`Reordered_UF1`	1	1	8	0	`true`	`false`
8	`Reordered_UF2`	1	2	14	4	`true`	`false`
9	`Reordered_UF1`	4	1	8	0	`true`	`false`
10	`Reordered_UF2`	4	2	14	4	`true`	`false`

The exploration started by comparing architectures with and without pipelining, and with and without unrolling. Table 3.1 summarizes the architectures that have been explored, along with the corresponding parameter configurations. The unrolled variant of the `Naive` transformation round is obtained by replacing the `Naive` combinatorial part with the `Unrolled` implementation. The `Reordered_UF2` core is the only which employs additional working variables and data prefetching. It is worth mentioning that `Reordered_UF1` also makes use of additional variables, but these are temporary, stored in the reordered pipeline register but not shared between rounds.

Table 3.2 lists the implementation results for all the architectures described in Table 3.1 on the Kintex UltraScale+ XCKU5P FPGA for the SHA-256 hash algorithm. Table 3.3 lists the results of the same implementation for the SHA-512 hash algorithm. Performance is expressed in terms of the hash rate, which is equivalent to the throughput in $\mathrm{Mbit\,s^{-1}}$ except for the constant value of the hash size. On the other hand, efficiency values are computed with respect to the throughput in $\mathrm{Mbit\,s^{-1}}$.

It is worth recalling that, when pipelining is enabled, the same combinatorial circuitry is replicated throughout the stages, increasing the

hash rate S times without degrading, in principle, the critical path delay of the design.

The results show that the `Reordered` transformation round architecture slightly outperforms the `Naive` implementation, while the `Precomputed` transformation round implementation underperforms the straightforward implementation, both in the basic and pipelined variants, reported in Table 3.2a and Table 3.2b respectively. On the other hand, the 2-unrolled variant of the `Reordered` transformation round architecture, despite the fact that it includes more optimisations, underperforms the unrolled `Naive` transformation round, both in the basic and pipelined variants, reported in Table 3.2c and Table 3.2d respectively.

Overall, the best architecture depends on the evaluation metric. The 4-stage pipeline based on a non-optimised transformation round core unrolled by a factor of 4, numbered 4 in the tables, turns out to be the architecture with the highest hash rate and the best power efficiency, but the 4-stage pipeline based on the highly optimised `Reordered_UF2` transformation round, numbered 9 in the tables, shows the best area efficiency.

3.3.2 Architectural Exploration

As an example of effective architecture exploration enabled by the SHA-2 workbench, the analysis of the critical paths reported by the hardware synthesis tool shows that the final adder was on the critical path even when the circuit is synthesized with the `Reordered` cores, which theoretically should not be the case [81]. This observation obviously raises the question of whether separating that adder with a register could benefit the critical path and hence the hash rate.

With the proposed framework, obtaining the modified architecture is as fast as switching a boolean parameter. So, designs from 7 to 10 in Table 3.1 have been reimplemented with the modified architecture. Architectures labelled with a quote (') in the table correspond to those in Table 3.1 having the same number, with the addition of the final stage.

Results are reported in Table 3.4 for SHA-256 and Table 3.5 for SHA-512. All but one architectures benefit from the introduction of the additional stage. It must be stressed that this gain could not have been exposed by a theoretical analysis of the critical path, making it necessary to go down to the implementation of each alternative.

Table 3.2: SHA-256 implementation results on the Kintex FPGA. Area efficiency is expressed in $\mathrm{Mbit\,s^{-1}/LUT}$, power efficiency is expressed in $\mathrm{Mbit\,s^{-1}/mW}$

(a) Base architectures

Nº	Critical	Hash rate	Area		Power Consumption (W)			Efficiency	
	Delay	(Mhash/s)	LUT	FF	Static	Dynamic	Total	Area	Power
1	2.398	6.516	1578	1875	0.451	0.122	0.573	1.057	2.911
5	3.004	5.201	1619	1866	0.451	0.115	0.566	0.822	2.353
7	2.297	6.802	1485	1640	0.451	0.131	0.582	1.173	2.992

(b) Pipelined architectures

Nº	Critical	Hash rate	Area		Power Consumption (W)			Efficiency	
	Delay	(Mhash/s)	LUT	FF	Static	Dynamic	Total	Area	Power
3	2.745	22.769	5314	4302	0.453	0.412	0.865	1.097	6.738
6	3.050	20.492	5385	4314	0.454	0.430	0.883	0.974	5.941
9	2.646	23.621	4986	4312	0.454	0.434	0.887	1.213	6.817

(c) Unrolled architectures

Nº	Critical	Hash rate	Area		Power Consumption (W)			Efficiency	
	Delay	(Mhash/s)	LUT	FF	Static	Dynamic	Total	Area	Power
2	7.750	8.065	2793	1960	0.451	0.099	0.550	0.739	3.754
8	5.146	6.073	2907	2194	0.452	0.164	0.616	0.535	2.524

(d) Unrolled and pipelined architecures

Nº	Critical	Hash rate	Area		Power Consumption (W)			Efficiency	
	Delay	(Mhash/s)	LUT	FF	Static	Dynamic	Total	Area	Power
4	9.012	27.741	9316	4188	0.453	0.347	0.800	1.097	8.877
10	5.607	22.294	8670	5184	0.455	0.581	1.036	0.658	5.509

Table 3.3: SHA-512 implementation results on the Kintex FPGA. Area efficiency is expressed in $\mathrm{Mbit\,s^{-1}/LUT}$, power efficiency is expressed in $\mathrm{Mbit\,s^{-1}/mW}$

(a) Base architectures

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Power
1	2.812	4.445	3130	3689	0.452	0.222	0.674	0.727	3.377
5	5.145	2.430	3097	3690	0.451	0.130	0.581	0.402	2.141
7	2.738	4.565	2784	3240	0.452	0.233	0.685	0.840	3.412

(b) Pipelined architectures

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Power
3	3.121	16.021	10424	8540	0.456	0.772	1.228	0.787	6.680
6	5.562	8.990	10079	8539	0.454	0.522	0.977	0.457	4.712
9	3.096	16.150	9825	8269	0.456	0.791	1.247	0.842	6.631

(c) Unrolled architectures

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Power
2	10.012	4.994	5585	3892	0.452	0.167	0.619	0.458	4.131
8	9.392	2.662	5828	4343	0.452	0.205	0.657	0.234	2.074

(d) Unrolled and pipelined architecures

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Power
4	12.003	16.663	17970	8304	0.455	0.586	1.041	0.475	8.195
10	10.287	9.721	18274	10469	0.457	0.807	1.263	0.271	4.027

Table 3.4: Results with the addition of the final stage for SHA-256. Area efficiency is expressed in $\text{Mbit s}^{-1}/\text{LUT}$, power efficiency is expressed in $\text{Mbit s}^{-1}/\text{mW}$

(a) `Reordered_UF1`, without pipelining

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Power
7	2.297	6.802	1485	1640	0.451	0.131	0.582	1.173	2.992
7'	2.250	6.944	1487	1898	0.451	0.134	0.585	1.196	3.039

(b) `Reordered_UF2`, without pipelining

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Power
8	5.146	6.873	2907	2194	0.452	0.164	0.616	0.535	2.524
8'	4.935	6.332	2914	2451	0.452	0.164	0.616	0.556	2.632

(c) `Reordered_UF1`, with 4-stage pipelining

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Power
9	2.646	23.621	4986	4312	0.454	0.434	0.887	1.213	6.817
9'	2.502	24.980	5088	4570	0.454	0.427	0.881	1.257	7.259

(d) `Reordered_UF2`, with 4-stage pipelining

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Power
10	5.607	22.294	8760	5184	0.455	0.581	1.036	0.658	5.509
10'	5.607	22.294	8649	5455	0.455	0.576	1.030	0.660	5.541

Table 3.5: Results with the addition of the final stage for SHA-512. Area efficiency is expressed in $\mathrm{Mbit\,s^{-1}/LUT}$, power efficiency is expressed in $\mathrm{Mbit\,s^{-1}/mW}$

(a) `Reordered_UF1`, without pipelining

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Efficiency Power
7	2.738	4.565	2784	3240	0.452	0.233	0.685	0.840	3.412
7'	2.689	4.649	2793	3754	0.453	0.223	0.676	0.852	3.521

(b) `Reordered_UF2`, without pipelining

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Efficiency Power
8	9.392	2.662	5828	4343	0.452	0.205	0.657	0.234	2.074
8'	8.793	2.843	5839	4856	0.452	0.217	0.669	0.249	2.176

(c) `Reordered_UF1`, with 4-stage pipelining

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Efficiency Power
9	3.096	16.150	9825	8269	0.456	0.791	1.247	0.842	6.631
9'	3.122	16.015	9821	8786	0.456	0.740	1.196	0.835	6.856

(d) `Reordered_UF2`, with 4-stage pipelining

Nº	Critical Delay	Hash rate (Mhash/s)	Area LUT	Area FF	Power Consumption (W) Static	Dynamic	Total	Efficiency Area	Efficiency Power
10	10.287	9.721	18274	10469	0.457	0.807	1.263	0.272	3.940
10'	10.346	9.666	18264	10983	0.456	0.773	1.229	0.271	4.027

3.3.3 Exploring a different target

To investigate the influence of the target platform on the measured results, the evaluation was repeated targeting a different technology, namely the Xilinx Artix-7 XC7A200T device, a 28 nm FPGA [129], smaller and slower compared to the Kintex device considered in the previous section. Table 3.6 shows the implementation results.

Some of the considerations made for the Kintex target can be repeated for the Artix-7. The `Precomputed` transformation round still underperforms the straightforward implementation, and the `Reordered_UF2` transformation round still underperforms the unrolled `Naive` transformation round.

However, there are also significant differences. The `Reordered` transformation round no longer outperforms the straightforward implementation, both in the basic and in the pipelined variants, reported in Table 3.6a and Table 3.6b respectively; in the latter case, the `Reordered` transformation round even underperforms the `Naive` variant. More surprisingly, the addition of the final stage turns out to be counterproductive, leading to an increase in the critical path delay. This suggests that the gain observed for the Kintex target, which was not expected by the theoretical analysis of the critical path, needs to be evaluated on a platform-specific basis.

3.4 Analysis of the Impact of Design Techniques on Application Metrics

Table 3.7 compares the various techniques described in Section 2.1 in terms of their effects on performance, area occupation and energy efficiency, and the implementation complexity of each of them. The table also lists the most representative works employing each technique. As described in Section 2.2, each design usually exploits more than one technique in order to meet the stated objectives.

Most design techniques are primarily aimed to performance, at the expense of increased area occupation and energy consumption. These approaches are best evaluated in terms of *area efficiency* or *area-delay product*, and *power efficiency* or *power-delay product*, in order to assess whether the price in terms of area occupation and energy consumption actually pays off.

Without loss of generality, throughout this section only single-PDB messages will be considered, as is customary when talking about performance metrics.

Table 3.6: SHA-256 implementation results on the Artix-7 FPGA. Area efficiency is expressed in $\mathrm{Mbit\,s^{-1}/LUT}$, power efficiency is expressed in $\mathrm{Mbit\,s^{-1}/mW}$

(a) Base architectures

Nº	Critical	Hash rate	Area		Power Consumption (W)			Efficiency	
	Delay	(Mhash/s)	LUT	FF	Static	Dynamic	Total	Area	Power
1	6.273	2.491	1593	1865	0.122	0.063	0.185	0.400	3.447
5	8.214	1.902	1565	1866	0.122	0.054	0.176	0.311	2.767
7	6.274	2.490	1412	1642	0.122	0.070	0.192	0.452	3.321
7'	6.500	2.404	1422	1900	0.122	0.062	0.184	0.433	3.344

(b) Pipelined architectures

Nº	Critical	Hash rate	Area		Power Consumption (W)			Efficiency	
	Delay	(Mhash/s)	LUT	FF	Static	Dynamic	Total	Area	Power
3	7.002	8.926	4923	4302	0.122	0.223	0.346	0.464	6.604
6	9.290	6.728	4971	4301	0.122	0.188	0.311	0.346	5.541
9	7.103	8.799	4795	4299	0.122	0.210	0.332	0.470	6.785
9'	7.449	8.390	4754	4557	0.122	0.202	0.329	0.452	6.629

(c) Unrolled architectures

Nº	Critical	Hash rate	Area		Power Consumption (W)			Efficiency	
	Delay	(Mhash/s)	LUT	FF	Static	Dynamic	Total	Area	Power
2	20.262	3.085	2767	1960	0.122	0.048	0.170	0.285	4.645
8	13.709	2.280	2895	2195	0.122	0.086	0.208	0.202	2.808
8'	13.988	2.234	2897	2452	0.122	0.080	0.202	0.197	2.831

(d) Unrolled and pipelined architecures

Nº	Critical	Hash rate	Area		Power Consumption (W)			Efficiency	
	Delay	(Mhash/s)	LUT	FF	Static	Dynamic	Total	Area	Power
4	21.563	11.594	9072	4188	0.122	0.183	0.305	0.327	9.731
10	15.494	8.068	8566	5185	0.122	0.123	0.407	0.241	5.074
10'	15.494	8.068	8468	5455	0.122	0.122	0.397	0.244	5.202

Table 3.7: Impact of SHA-2 optimisation techniques on evaluation metrics. Stronger impacts are highlighted

Architectural Technique	Impact on Throughput	Impact on Area	Impact on Power	Complexity	Employed by
Pipelining	**positive**	**negative**	**negative**	**low**	[3, 76, 80, 81, 8]
Variables Precomputation	**positive**	negative	negative	**low**	[45, 67, 17, 18, 5] [4, 44, 2]
Loop Unrolling	positive	**negative**	**positive**	**low**	[25, 2, 80, 81, 8] [67, 135, 136]
Loop Folding	**negative**	**positive**	negative	**high**	[60, 136, 110, 40] [14, 125]
Spatial Reordering	**positive**	negative	negative	**high**	[76, 80, 81, 8]
Quasi - Pipelining	**positive**	negative	negative	**low**	[68, 28, 27, 113]

3.4.1 Performance

Performance refers to how fast data is processed. The most used metric to assess performance of hash circuits is the *throughput*, defined as the the number of bits delivered per unit of time. If the hash circuits is capable of outputting a new hash value every N_{clk} clock cycles, and the clock period is τ_{clk}, the throughput can be written as

$$Q = \frac{L(DM(M))}{\tau_{clk} \cdot N_{clk}} = \frac{L(DM(M)) \cdot f_{clk}}{N_{clk}} \tag{3.1}$$

The number of bits of the output depends on the selected hash function and does not offer any degree of freedom for improving throughput, unless the designer is in the position of choosing which hash function to employ. For this reason, when comparing designs for the SHA-2 function with different hash sizes, it is preferable to refer to the *hash rate*, defined as

$$F = \frac{1}{\tau_{clk} \cdot N_{clk}} = \frac{f_{clk}}{N_{clk}} \tag{3.2}$$

The other two terms of Eq. (3.1) are instead fully dependent on architectural design decisions. The clock period is lower bounded by the critical path delay, hence it is directly influenced by architectural techniques which impact the critical path. Variable precomputation, spatial

reordering and quasi-pipelining all reduce the critical path, therefore these techniques are beneficial for the throughput metric.

The number of clock cycles between two consecutive outputs is the latency required to compute the hash of a single PDB. For architectures capable to perform one iteration per clock cycle, N_{clk} is $R+1$ unless the architecture can handle the final sum concurrently with the execution of the last iteration, such as [18, 17], or has the adder in parallel with the critical path, such as [80, 81]; in such cases N_{clk} is R. Architectures with loop folding, on the other hand, require multiple clock cycles per iteration, making this technique negative for throughput.

Pipelining does not modify the number of clock cycles required to produce a single hash message. Instead, it reduces the average value of N_{clk} by processing more PDBs simultaneously. The throughput is consequently increased by a factor equal to the number of pipeline stages, only slightly reduced by a small increase in the critical path delay, as confirmed by the results reported in Section 3.3.

Loop unrolling has a twofold effect on performance. On one hand, it directly reduces the number of cycles required to hash a message by computing multiple iterations in a single clock cycle. On the other hand, it substantially increases τ_{clk} due to the increased number of combinatorial levels required to compute the different iterations. The overall effect of loop unrolling hence depends on whether the reduction of the number of clock cycles compensates for the increase in the critical path. This does not happen in [67], while it happens in [80]. This discussion does not take into account the fact that loop unrolling can enable further transformations which can reduce the critical path, as mentioned in Section 2.1.3.

Moreover, while the reduction of N_{clk} is platform-independent, the increase in τ_{clk} does depend on the underlying technology. This implies that the overall effect of loop unrolling on throughput is technology-dependent. For the experimental data reported in Section 3.3 it is clear that loop unrolling by a factor 4 is profitable for the straightforward implementation of SHA-256 for both the targets analysed. In fact, the increase factor of the critical path delay due to the unrolling is between 3× for the Artix-7 and 3.3× for the Kintex, i.e., less than the unrolling factor 4. This is no longer the case for the `Reordered` implementations: the increase factor here is between 2.1× and 2.3×, which is larger than the unrolling factor 2. This suggests that a more aggressive unrolling would be profitable also for the `Reordered` transformation round core. It must be also stressed, however, that the `Reordered_UF2` implementation involves a number of architectural differences compared to the

`Reordered_UF1` implementation, so it is not a simple unrolling of the same architecture. For SHA-512, the effect of unrolling is vastly reduced, since the increase factor in the critical path becomes $\sim 3.8\times$ for the straightforward implementation and $\sim 3.4\times$ for the `Reordered` implementation.

3.4.2 Area occupation and area efficiency

Area occupation refers to the size of the circuit when implemented in Application-Specific Integrated Circuit (ASIC) technology. On reconfigurable technologies such as FPGA, area occupation refers to the utilisation of device resources [30]. It is a relevant factor for the final cost of the design, since a larger design requires a larger device for the physical implementation [65].

Usually, area occupation is traded off for performance. Techniques like pipelining and loop unrolling, which are aimed to increase throughput, have a severely adverse impact on area occupation, since the circuit must be replicated as many times as the number of pipeline stages or the unrolling factor. However, unlike the case of multiple instances of the same circuit, only the data path needs to be replicated. Conversely, loop folding is aimed to reduce area occupation, at the expense of degraded throughput.

The experimental results presented in Section 3.3 show that the area occupation increase factor due to unrolling is $\sim 1.8\times$ for the straightforward implementation and $\sim 2\times$ for the *Reordered* implementation. Even in the former case, this area increase factor is higher than the increase factor for throughput, In fact, the increase factor for throughput due to unrolling corresponds to UF/DIF, where DIF is the increase factor in the critical path delay discussed in Section 3.4.1. This figure is $\sim 1.2\times$ for the straightforward implementation, which is lower than the increase factor in terms of area occupation, resulting in a negative impact of loop unrolling on area efficiency.

Other techniques, such as variable precomputation, spatial reordering and quasi-pipelining have a limited impact on the area occupation of SHA-2 accelerators since the additional resources utilised by these architectures are just a handful of registers.

3.4.3 Power and energy consumption

As with any digital system, energy represents the main operational cost for SHA-2 hardware accelerators, while power consumption refers to the amount of energy consumed by the circuit per unit of time.

Interestingly, operational costs might not be the only factor of interest, as instantaneous power consumption may sometimes have strong implications on the physical design of the accelerator. In fact, the energy absorbed per unit of time by a circuit is dissipated as heat, which must be driven away from the hardware to avoid damaging the circuit by overheating [114]. The cost of cooling clearly increases with the amount of power to be driven away [16], while thermal requirements can even place an upper bound on the amount of functionality that can be integrated in a chip [15], no matter what the cost constraints are; this latter issue is referred to as *power cap*. If N_M is the number of clock cycles required to hash a single message M, and P is the instantaneous power consumption of the accelerator, the energy consumption of hashing a message can be written as

Other applications, including passively-powered circuits, are limited in the amount of power they can be supplied. The constraint on the SHA-2 functionality placed by power supplying limitations is even tougher than the one posed by power dissipation, since there are no measures that can be used to mitigate the problem, unlike the case of cooling for dissipation. In such occurrences, the input power constraint can only be met by means of architectural optimisations [124].

$$J_M = P \cdot \tau_{clk} \cdot N_M = \frac{P \cdot N_M}{f_{clk}} \tag{3.3}$$

since $\tau_{clk} \cdot N_M$ is the time required to hash the message M. For architectures processing one message at a time, $N_M = N_{clk}$, while for architectures processing more than one message simultaneously, the identity $N_M = N_{clk}$ holds on average over multiple messages. Therefore, the average energy consumption per message hash can be written as

$$J_M = P \cdot \tau_{clk} \cdot N_{clk} = \frac{P}{F} \tag{3.4}$$

with F being the hash rate, defined by Equation (3.2).

The equation above suggests that the energy consumption can be reduced by either increasing the hash rate, or decreasing the power consumption. The hash rate has been analyzed in Section 3.4.1, so the remainder of this section will focus on power consumption.

The power consumption of a circuit is usually divided into a static and a dynamic component. The static component depends mainly on technological aspects, such as the power supply V, or the threshold voltage of the transistors, amongst others [7]. On the other hand, the dynamic component can be influenced by architectural decisions [42] and

therefore by the particular techniques employed in designing the SHA-2 circuit.

Energy is absorbed from the supply by a Complementary Metal-Oxide-Semiconductor (CMOS) gate during a transition from 0 to V, and is dissipated as heat during the subsequent transition from V to 0. Therefore, only the former transition leads to energy consumption [16]. For this reason, the *switching activity* α is often defined as the probability of a transition from 0 to V within a clock cycle, in order to avoid a 1/2 factor throughout the power consumption formulae.

At the gate level, the dynamic power consumption can be written as [16]:

$$P_d = \alpha \cdot f_{clk} \cdot C \cdot V^2 \tag{3.5}$$

where C is the capacitive load of the gate. For FPGAs, the power consumption can be rewritten to take into account the utilisation U of each resource in the whole device after programming [103]:

$$P_d = f_{clk} \cdot V^2 \cdot \sum_i \alpha_i \cdot C_i \cdot U_i \tag{3.6}$$

These formulae show the impact of the clock frequency on power consumption. Techniques that optimise throughput of the SHA-2 accelerator through decreasing the critical path end up increasing the power consumption, due to the corresponding increase of f_{clk}. This is the case of variable precomputation, spatial reordering and quasi-pipelining.

As mentioned above, in massively parallel systems, increasing f_{clk} puts more pressure on cooling system, whose dissipation capabilities may limit the maximum clock frequency, or poses an inherent limitation for passively-powered devices. However, this power increase does not lead to an increase of the energy consumed per hash, since f_{clk} also appears on the denominator of Eq. (3.3).

It is worth noting that if the optimisation of throughput is not due to a decrease in τ_{clk} but to a decrease in N_{clk}, this argument does not apply. In such cases, there is no power consumption penalty, and there is a reduction in the energy consumption per hash due to the throughput increase, as shown by Eq. (3.4). Techniques that increase throughput by decreasing N_{clk} include pipelining and loop unrolling.

Another factor impacting power consumption is area occupation [11], since more low-level devices need to be powered, and physical data and clock nets are longer [29]. This is shown also by Eq. (3.6). Therefore, designs which optimise resources utilisation, such as loop folding, also

improve power efficiency. On the other hand, techniques leading to increased area occupation, such as pipelining and loop unrolling, also face increased power consumption.

A more direct effect of architectural choices on power consumption is linked with the number of register operations [79, 3]. Since each register is read and written once per clock cycle, reducing the number of clock cycles needed to compute a hash also reduces the number of operations performed by registers, hence the dynamic power dissipation due to register operations. From this point of view, loop unrolling turns out to be beneficial in terms of power savings due to register operations.

The overall effect to be expected on power consumption due to each design technique is summarized in Table 3.7. Pipelining has an adverse effect on power consumption, due to its increase in area and register operations. Loop unrolling has opposite implications on power consumption, increasing the area occupation on one hand, but reducing frequency and register operations on the other. Similarly, loop folding has a twofold impact on power consumption. It leads to power savings due to the area reduction, but this also comes at the cost of increased register operations. The predominant effect ultimately depends on the underlying technology employed for the SHA-2 accelerator.

3.4.4 Implementation complexity

This comparison criterium refers to the effort required for the designer to apply the architectural technique to a circuit, which is especially relevant in cases where a customized system is to be built.

Pipelining and loop unrolling are structured techniques, that can also be applied automatically by modern CAD tools. Variable precomputation implies for the designer to break the critical path, which for SHA-2 is clearly located in the computation of A_{t+1}. Computations that do not depend on A_t and E_t can be moved ahead to the previous round. Quasi-pipelining is a more sophisticated technique, but it can be applied without too much effort as it is clearly documented in [68].

On the other hand, loop folding and spatial reordering require a significant intervention by the designer. For the former, the designer must identify which components can be shared, establish the schedule of operations, and then implement it. For the latter, it is up to the designer to balance the paths between the two halves resulting from reordering.

Table 3.8: Requirements of surveyed applications relying on SHA-2 and recommended optimisations

Application	Requirements				Recommended
	Throughput	Area	Energy	Power	Optimisations
Web server	moderate	minor	minor	minor	Pipelining Loop Unrolling Precomputation Spatial Reordering Quasi-Pipelining
Bitcoin mining	critical	major	critical	major	Loop Unrolling Precomputation Spatial Reordering Quasi-Pipelining
IoT (cryptography)	moderate	critical	critical	minor	Precomputation Spatial Reordering Quasi-Pipelining
RFID	minor	critical	minor	critical	Loop Folding
IoT (mining)	critical	critical	critical	major	Precomputation Spatial Reordering Quasi-Pipelining
Trusted FPGA	minor	moderate	minor	minor	Loop Folding

3.4.5 Impact on applications

Different applications place different constraints on the underlying SHA-2 circuitry. Table 3.8 lists the constraints incurred by the applications discussed in Section 1.3 on the SHA-2 hardware accelerator, in order to suggest the optimisation techniques best suitable for each application.

High-performance Web servers, providing security services relying on SHA-2, are mainly concerned with throughput, since they need to scale in the number of concurrent users they can serve. For this application, switching from a software implementation to an application-specific SHA-2 processor may bring substantial savings in terms of energy, i.e. operational costs. The need of processing different messages simultaneously, and the relatively large budget for circuit area and cost, indicate that pipelining, among other throughput-focused techniques, can

be greatly beneficial for this class of applications.

Similarly, applications like trusted FPGA computing are loosely constrained, since SHA-2 is performed rather infrequently and does not pose strict throughput or energy constraints, while the main concern is typically the area occupation, and the security scheme reduces the area available for the user logic. This can be addressed by using the loop folding technique.

Bitcoin mining rigs have far more stringent constraints, especially on throughput and energy consumption, and both of them must be fulfilled in order to design a profitable miner. Area requirements must also be kept under control, both for allowing parallel instances of the miner to be instantiated within the same device, and to contain the cost of the rig. The same goes for power, which can become a limiting factor in the design of a massively parallel architecture. This set of requirements can be addressed by using the loop unrolling technique combined with throughput-oriented techniques such as variable precomputation or spatial reordering. In this way, the loop unrolling expands the applicability of the other techniques while delivering energy savings.

IoT applications typically feature heavily constrained low-cost devices. Area occupation of the SHA-2 circuit must therefore be contained for these applications. Most importantly, energy consumption must be kept as low as possible, in order not to waste the limited energy budget of these devices. Of course, porting mining within an IoT environment would add the requirement for high throughput and is normally performed by the most powerful elements in the network. In this context, techniques that deliver increased throughput of the accelerator with limited impact on area and throughput, such as variables precomputation, spatial reordering, and quasi-pipelining can be utilised.

For passively-powered devices such RFID tags, a limited amount of energy per unit of time can be delivered to the device, while the overall energy consumption, or even duration, of the operation is less of a concern. Therefore, the strict constraint is on power rather than energy or throughput. Moreover, the severe cost limitations placed on the practical usability of RFID tags translate to strict constraints on area occupation, which suggests implementations based on the loop folding technique.

Chapter 4

Efficient Multi-Operand Addition on FPGAs

Implementing SHA-2 on an FPGA target technology is a particularly challenging task, due to the nature of operations involved in the SHA-2 algorithm.

The critical path in any SHA-2 implementation is located within the Compressor, and is mainly determined by the time needed to perform the addition. Multi-operand additions can be implemented with compressor trees based on parallel counters, which have been proven to be particularly efficient on ASIC technology. However, due to the specific architecture of FPGAs, parallel counters efficient on ASICs cannot be synthesized efficiently on FPGAs

This chapter illustrates how to efficiently implement a multi-operand addition on an FPGA, providing both the theoretical background and a practical case study for a concrete FPGA, which will be used in Chapter 5 as one of the building blocks of an efficient implementation of a SHA-2 accelerator on that target technology.

4.1 Compressor Trees based on Parallel Counters

Let $X_{J-1}, X_{J-2}, \ldots, X_1, X_0$ be a set of J N-bit operands to be added. One approach to perform the addition would be to employ a tree of binary N-bit adders, called an *adder tree*. Such a solution comes with the obvious disadvantage of a delay proportional to $\log_2 J$ times the delay of the binary adder. A more efficient alternative strategy has been proposed, originally in the context of partial product reduction in parallel multipliers, based on the usage of parallel counters [120, 26].

A *compressor tree* is a circuit that takes as input J N-bit operands X_j and computes two values, S and C, such that [96]:

$$S + C = \sum_{j=0}^{J-1} X_j \tag{4.1}$$

The addition of Eq. (4.1) can be computed with a standard binary adder. In essence, the compressor tree is the generalisation of the Carry Save Adder to a number of operands greater than 3.

Actually, a compressor tree may be designed to produce more than two outputs. In general, a compressor tree outputs K operands O_k such that [72]:

$$\sum_{k=0}^{K-1} O_k = \sum_{j=0}^{J-1} X_j \tag{4.2}$$

This can be advantageous if a fast adder with more than two operands is available in the target technology. Current FPGAs offer support for fast ternary addition [104], which can be exploited by rewriting Eq. (4.2) as [96]:

$$O_1 + O_2 + O_3 = \sum_{j=0}^{J-1} X_j \tag{4.3}$$

A compressor tree is advantageous over an adder tree if its delay scales with the number of operands less than logarithmically. To obtain this, rather than being summed, bits of the operands are *counted* by specialised counter circuits.

A *single-column parallel counter*, also called an *m:n counter*, is a circuit that takes m input bits, counts the number of bits which value of is 1, and outputs the result as an unsigned n-bit integer [94]. Therefore, for a given number of input bits m, the number of output bits n is

$$n = \lceil \log_2 (m + 1) \rceil \tag{4.4}$$

A 1:1 counter makes no sense, since it simply replicates in output its input bit, a 2:2 counter is an *half adder*, and a 3:2 counter is a *full adder* [94].

Parallel counters are usually represented graphically with the so-called *dot notation*, which is shown in Fig. 4.1a with a 6:3 counter used as an example. Input bits are represented by dots, surrounded by an oval representing the counter. The oval is connected to the dots representing

output bits, which are connected together to indicate the outputs of the same counter. This notation is also useful when designing compressor trees built from parallel counters. The compressor tree shown in Fig. 5.4 of Chapter 5 can be regarded as an example of such a compressor tree.

4.2 Generalised Parallel Counters

Single column parallel counters count m bits from the same column of the operands and outputs the result on n bits. This can be generalised by counting m bits from an arbitrary number of columns, outputting the results again on n bits.

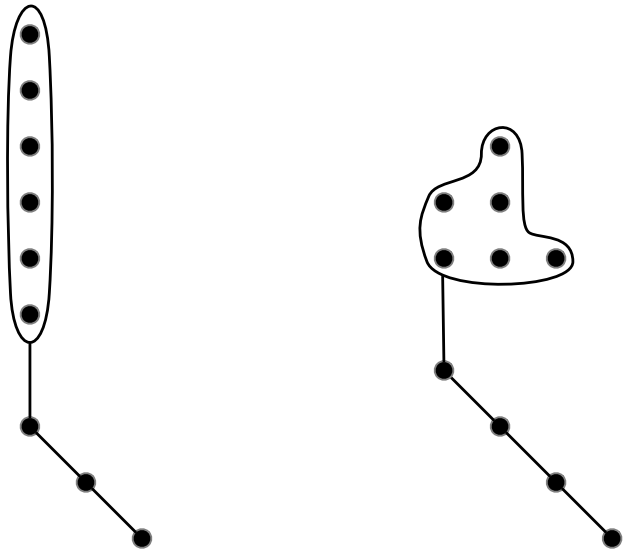
To be more specific, let $B = (b_{I-1}, b_{I-2}, \ldots, b_0)$ an I-bit unsigned binary integer. The *rank* of the bit b_i is its subscript i, which indicates its position in the integer, and therefore its weight. In fact, the bit b_i of rank i contributes $b_i \cdot 2^i$ to the overall value represented by B. The value represented by B can be converted in a base-10 value with the following formula:

$$B = \sum_{i=0}^{I-1} b_i \cdot 2^i \tag{4.5}$$

An m:n counter, by definition, counts m bits of the same rank. In the context of multi-operand addition, a *column* is a set of bits having the same rank, typically from different operands [96]. If the common rank of the input bits is i, the output bits of the counter have rank i, $i+1$, …, $i+n-1$ respectively [94].

A *Generalised Parallel Counter (GPC)* is a counter capable of summing bits of different ranks [94]. Formally, a GPC is defined as a tuple $(K_{n-2}, K_{n-1}, \ldots, K_0; n)$ where n is the number of output bits of the GPC, and K_i is the number of bits of rank i summed by the GPC [108]. An example of GPC is shown in Fig. 4.1b; similarly to single column counters, the dot notation is used to represent GPCs and the compressor trees designed with them.

A GPC can add bits from up to $n-1$ different ranks, from 0 to $n-2$. In fact, adding more than 1 bits of rank $n-1$ would require output bits of rank n at least, while the range of output bit ranks goes from 0 to $n-1$. The same applies if there is only one bit of rank $n-1$, but there is carry from the lower ranks. Hence, it would only be possible to have a single input bit of rank $n-1$ for a GPC with n output bits, with no carry from the lower ranks. But this would imply that this single input bit is not actually added to anything, instead it is simply



(a) 6:3 counter (b) (1,3,2;4) GPC

Figure 4.1: Examples of a single column counter and a GPC with 6 input bits, with the dot notation. Note that a GPC may require more output bits than the single column counter with the same number of input bits.

propagated to the output. In general, it is useless to have an input bit being directly propagated to the output, therefore GPCs presenting such bits are deemed *unreasonable* [94]. For the same reason, a reasonable GPC must also meet the following condition [95]:

$$K_0 > 1 \tag{4.6}$$

In fact, $K_0 = 0$ would imply that the output bit of rank 0 is always 0, while $K_0 = 1$ would imply that the input bit of rank 0 simply propagates to the output.

For a GPC with m input bits and n output bits, the following quite straightforward relations also hold:

$$\sum_{i=0}^{n-2} K_i = m \tag{4.7}$$

$$\sum_{i=0}^{n-2} K_i \cdot 2^i \leq 2^n - 1 \tag{4.8}$$

The first relation defines the number of input bits, whereas the last relation ensures that the output can be represented with n bits. On the other hand, the number of output bits of a GPCs with a given set of input bits $K_0, K_1, \ldots, K_{I-1}$ can be computed by inverting Eq. (4.8):

$$n = \left\lceil \log_2 \left(\sum_{i=0}^{I-1} K_i \cdot 2^i + 1 \right) \right\rceil \tag{4.9}$$

4.2.1 Efficiency Parameters of a GPC

GPCs can be evaluated according to different metrics, tailored towards different design objectives.

The *compression difference* of a GPC is the difference between the number of input bits and the number of output bits [98]:

$$\delta = m - n \tag{4.10}$$

Sometimes this metric is evaluated by the ratio rather than the difference [94, 72]:

$$\delta_r = \frac{m}{n} \tag{4.11}$$

Reasonable GPCs should have $\delta > 1$ [72].

The *area efficiency* of a GPC, also called *area degree* in [98], is the ratio between the compression difference δ and the number of logic resources λ needed [63]:

$$\varepsilon = \frac{\delta}{\lambda} \tag{4.12}$$

The *performance efficiency* of a GPC, also called *performance degree* in [98], is the ratio between the compression difference δ and the critical delay τ of the GPC:

$$\eta = \frac{\delta}{\tau} \tag{4.13}$$

4.2.2 From the GPC to the Compressor Tree

Once GPCs have been defined, they must be combined to construct a full compressor tree. This task is usually tackled in a two-step fashion. First, a library of GPCs is built, and then an algorithm to generate compressor trees from the GPCs in the library is applied. This approach effectively decouples the problem of designing GPCs from the problem of constructing the whole compressor tree.

Various different algorithms have been proposed in the literature to obtain the compressor tree from the GPC library. These falls into two categories: heuristics, and formulations based on Integer Linear Programming (ILP).

Design of the GPC Library

The design of effective GPCs is influenced by a number of architectural features of the target FPGA, therefore GPCs in the library are architecture-specific [97], and typically designed by hand [98]. Since the FPGA CAD tools are usually unable to infer the best mapping of each GPC onto the components of the underlying target architecture, the library must also be implemented manually [98].

Architectural characteristics of the target FPGA typically suggest constraints on the maximum value of m and n for the GPCs in the library. A *primitive* GPC is a GPC which satisfies the I/O constraints of the library. However, not all primitive GPCs need to be implemented. In fact, the functionality of a primitive GPCs can be implemented by a GPC with more inputs, and possibly more outputs, by setting some input bits to 0 so as to match the shape of the input of the GPC to be implemented. A *covering* GPC is a GPC which the functionality of cannot be implemented by any other primitive GPC in the library; in other words its functionality cannot be implemented by another GPC fulfilling the I/O constraints [95].

Implemented GPCs in the library are associated with some of the efficiency parameters introduced in Section 4.2.1, which will be used by the compressor tree synthesis strategy to select the most appropriate GPC to employ every time more than one of them can be used. In heuristic-based strategies, typically this means that GPCs are sorted according to one efficiency parameter, and the choice of this efficiency parameter is made according to which specific design objective is to meet [97].

Heuristic-based Approaches

The heuristic proposed in [94] tries to remove the highest possible number of bits at each iteration, therefore favouring GPCs with the highest compression ratio. At every iteration, the heuristic considers the highest remaining column, and selects the GPC with the highest compression ratio which can cover uncovered bits in the selected column and in the surrounding ones. This heuristic is improved in [95] to take into account the difference in delays between the output bits of the GPCs. Output bits from the previous layer with the highest delays are connected with the inputs in the next layer which drive the shortest paths. A different generalisation of the heuristic is proposed in [98], where the selection order of GPCs is no longer based only on the compression ratio, but can follow different metrics, in order to meet different design objectives.

An alternative approach is proposed in [72], which starts by establishing the heights of all levels of the compressor tree according to a rule similar to Dadda's [26]. The height h_l of level l is computed with respect to the m:n single column counter with the maximum compression ratio in the library, according to the following:

$$h_l = \begin{cases} K & l = 0 \\ \left\lfloor h_{l-1} \cdot \frac{m}{n} \right\rfloor & l > 0 \end{cases} \tag{4.14}$$

where K is the number of outputs of the compressor tree, as per Eq. (4.2). Only columns with height exceeding h_l at level l are reduced by means of GPCs. Authors of [72] found that this heuristic produces compressor trees with fewer GPCs than those generated by heuristics without intermediate limits, similar to what happens with Dadda trees compared with Wallace trees [26].

ILP-based Approaches

[96] proposes an ILP model for mapping compressor trees onto GPCs with a maximum of 6 input bits, and 3 output bits, aimed at minimising the overall number of layers.

The ILP model formulated in [73] for 6-input GPCs, minimises the number of GPCs in the compressor tree after having minimised the number of its level. The same target is employed in the ILP model proposed in [64], which supports arbitrarily defined GPCs and also the pipelining of the compressor tree, by means of a 1:1 counter representing a flip-flop. This model is extended in [62] in order to accomodate row-based compressors. In [134], the ILP model is extended to take into account, and favour, the possibility of resource sharing between GPCs.

4.3 GPCs for the 7-series Xilinx FPGAs

GPCs can be efficiently implemented in FPGAs because they map quite well onto the underlying component of this target technology.

More specifically, they take advantage of the fact that LUTs of an FPGA can implement any logical function of their inputs. For GPCs, this means that LUTs can count any number of input bits irrespective of their ranks. A GPC with input bits m less than the number of input bits F_{in} of the LUT can be implemented in exactly n/F_{out} LUTs, where n is the number of the output bits of the GPC and F_{out} is the number of output bits of the LUT [96].

Clearly, architectural parameters such as the fan-in and the fan-out of the LUTs bear influence on the design of the GPCs. In some works [94, 96, 72, 73], the number of input bits m of the GPCs is upper bounded to the fan-in F_{in} of the LUTs. Other works [63, 58] take advantage of the carry chain component of the FPGA to combine more than one LUT in order to implement bigger GPCs.

4.3.1 The Xilinx 7-series Look-Up Table

7-series Xilinx FPGAs are equipped with 6-input LUTs, able to implement any arbitrary 6-input, 1-output logic function. This operating mode uses the O6 output pin for the output of the logic function.

Actually, this LUT has two outputs. In fact, as shown in Fig. 4.2, the 6-input LUT can be broken down into two 5-input LUTs with shared inputs. The two LUTs can be programmed to implement two entirely independent logic functions as long as they share their inputs; when this is the case the most significant of the 6 input bits must be driven high, otherwise the O6 and O5 pins would output the same bit. Alternatively, a 6-input and a 5-input logic function with shared inputs can also be implemented, but in this case the 5-input function must share also the output values with the 6-input function when one of the input of the latter, mapped as its most significant input, is 0. Finally, by setting in each function as *don't care* the input of the other function, a single LUT can implement two completely independent logic functions, with no shared inputs, provided that one function has at most 3 inputs and the other has at most 2 inputs.

In summary, ignoring the uncommon scenario of two functions sharing the same output values, a single LUT of the 7-series Xilinx FPGA family can implement:

- an arbitrarily defined 6-input, 1-output logic function; or

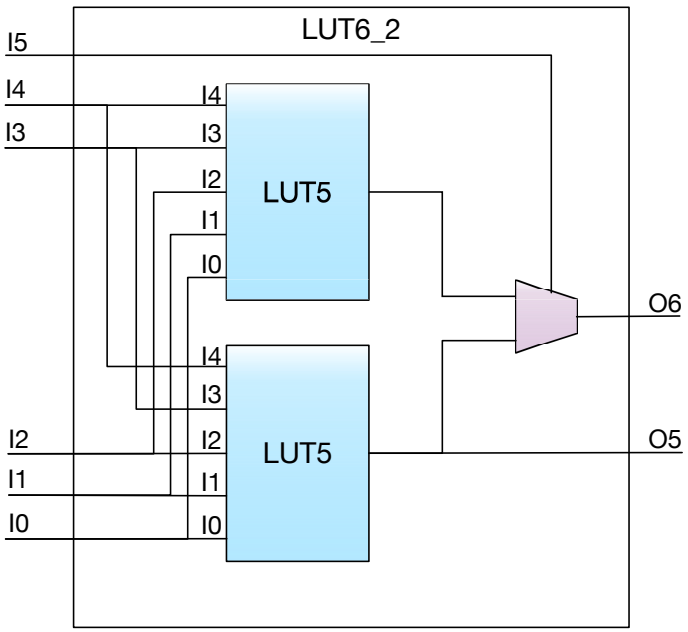


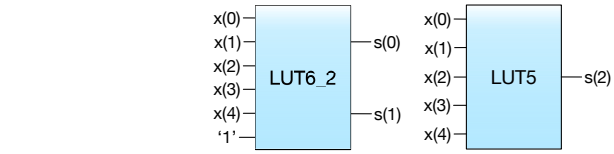
Figure 4.2: Architecture of the Look-Up Table (LUT) of the 7-series Xilinx FPGA. Adapted from [132].

- two arbitrarily defined 5-input, 1-output logic functions with shared inputs; or
- two arbitrary, independent logic functions of 3 and 2 inputs.

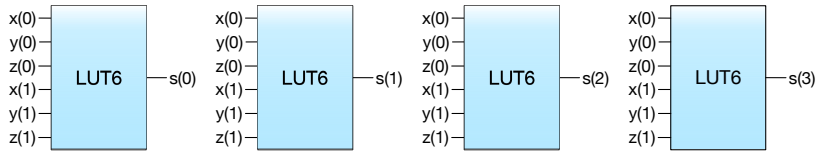
4.3.2 A GPC library for the 7-series FPGA

In order to efficiently map the additions involved in the SHA-2 algorithm on the 7-series FPGA, a GPC library has been defined. Taking into account that these additions do not involve a dramatically high number of operands, the number of inputs m has been constrained to the number of inputs of the LUTs, i.e. $F_{in} = 6$. This in turn constraints also the number of outputs n, according to Eq. (4.9).

In order to take into account both area and performance requirements, a definition of coverage stricter than the one given in Section 4.2.2



(a) (0,5,3) GPC



(b) (0,3,3;4) GPC

Figure 4.3: Examples of GPC mapping with (a) less than 6 inputs (b) 6 inputs

has been adopted. Namely, GPC^x covers GPC^y if and only if **all** the following conditions are met:

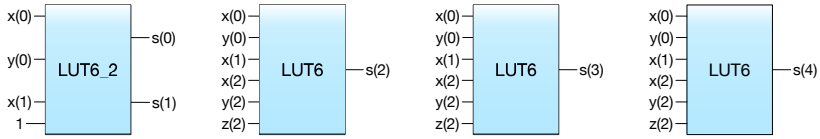
$$K_i^x \geq K_i^y \qquad \forall i \in (0, 1, \ldots, n^y - 2) \tag{4.15a}$$
$$\lambda^x \leq \lambda^y \tag{4.15b}$$
$$\tau^x \leq \tau^y \tag{4.15c}$$
$$\varepsilon^x \geq \varepsilon^y \tag{4.15d}$$

On the contrary, the covering condition given in Section 4.2.2 implies the satisfaction of Eq. (4.15a) only. The stricter definition of coverage provided here allows for the selection of different GPCs to meet different design objectives.

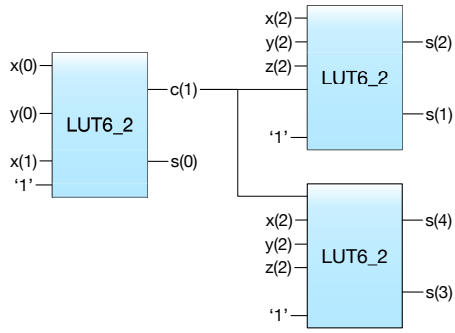
Table 4.1 lists all the primitive GPCs. Since $m \leq F_{in}$, each GPC can be implemented with a single level of LUTs. Since all LUTs of a GPC takes the same m inputs, the dual 5-input LUT option can be exploited to reduce the number of required LUTs, which can be written as a function of the number of output bits:

$$\lambda = \begin{cases} \left\lceil \frac{n}{2} \right\rceil & m \leq 5 \\ n & m = 6 \end{cases} \tag{4.16}$$

For $m = 6$ the dual-LUT option cannot be used, hence the number of required LUTs increases sharply. This mapping is exemplified in



(a) computing the two least significant output bit with the same LUT



(b) computing the least significant carry bit

Figure 4.4: Optimisations of GPC mapping applied to the (0,3,1,2;5) GPC

Fig. 4.3b. As a consequence, 6-input GPCs cannot cover 5-input GPCs since they fail to satisfy Eq. (4.15b).

4.3.3 Optimising the mapping of the GPCs

The straightforward mapping described in Section 4.3.2 is not always the optimal one. This can be seen by observing that an absolute lower bound on the number LUTs required to implement the GPC with n output bits depends on the fanout of the LUTs and can be written as

$$\lambda_{inf} = \left\lceil \frac{n}{F_{out}} \right\rceil \tag{4.17}$$

Taking into account the dual LUT option, each LUT can output 2 bits at most, therefore for this FPGA family

$$\lambda_{inf} = \left\lceil \frac{n}{2} \right\rceil \tag{4.18}$$

When $m \leq 5$, the straightforward mapping is optimal, while for 6-input GPCs, more efficient mapping can often be obtained by properly reorder-

Table 4.1: Primitive GPCs for $n \leq 6$

GPC	δ	λ	ε	τ	Covered by
2-input GPCs					
(2;2)	1	1	1	τ_{LUT}	(3;2)
3-input GPCs					
(3;2)	1	1	1	τ_{LUT}	Covering
4-input GPCs					
(0,4;3)	1	2	0.5	τ_{LUT}	(0,5;3)
(2,2;3)	1	2	0.5	τ_{LUT}	(2,3;3)
(1,3;3)	1	2	0.5	τ_{LUT}	(1,4;3)
5-input GPCs					
(0,5;3)	2	2	1	τ_{LUT}	Covering
(1,4;3)	2	2	1	τ_{LUT}	Covering
(1,3;3)	2	2	1	τ_{LUT}	Covering
(0,3,2;4)	1	2	0.5	τ_{LUT}	Covering
(1,1,3;4)	1	2	0.5	τ_{LUT}	Covering
(1,2,2;4)	1	2	0.5	τ_{LUT}	Covering
(2,1,2;4)	1	2	0.5	τ_{LUT}	Covering
6-input GPCs					
(0,6;3)	3	3	1	τ_{LUT}	Covering
(1,5;3)	3	3	1	τ_{LUT}	Covering
(0,2,4;4)	2	4	0.5	τ_{LUT}	Covering
(0,3,3;4)	2	4	0.5	τ_{LUT}	Covering
(0,4,2;4)	2	4	0.5	τ_{LUT}	Covering
(1,1,4;4)	2	4	0.5	τ_{LUT}	Covering
(1,2,3;4)	2	4	0.5	τ_{LUT}	Covering
(1,3,2;4)	2	4	0.5	τ_{LUT}	Covering
(2,1,3;4)	2	4	0.5	τ_{LUT}	Covering
(2,2,2;4)	2	4	0.5	τ_{LUT}	Covering
(0,3,1,2;5)	1	5	0.2	τ_{LUT}	Covering

Table 4.2: GPC library

GPC	δ	λ	ε	τ
(3;2)	1	1	1	τ_{LUT}
(0,5;3)	2	2	1	τ_{LUT}
(1,4;3)	2	2	1	τ_{LUT}
(1,3;3)	2	2	1	τ_{LUT}
(0,3,2;4)	1	2	0.5	τ_{LUT}
(1,1,3;4)	1	2	0.5	τ_{LUT}
(1,2,2;4)	1	2	0.5	τ_{LUT}
(2,1,2;4)	1	2	0.5	τ_{LUT}
(0,6;3)	3	3	1	τ_{LUT}
(1,5;3)	3	3	1	τ_{LUT}
		2	1.5	$2\tau_{LUT}$
(0,2,4;4)	2	4	0.5	τ_{LUT}
		3	0.66	$2\tau_{LUT}$
(0,3,3;4)	2	4	0.5	τ_{LUT}
		3	0.66	$2\tau_{LUT}$
(0,4,2;4)	2	4	0.5	τ_{LUT}
		3	0.66	$2\tau_{LUT}$
(1,1,4;4)	2	3	0.66	τ_{LUT}
(1,2,3;4)	2	3	0.66	τ_{LUT}
(1,3,2;4)	2	3	0.66	τ_{LUT}
(2,1,3;4)	2	3	0.66	τ_{LUT}
(2,2,2;4)	2	3	0.66	τ_{LUT}
(0,3,1,2;5)	1	4	0.25	τ_{LUT}
		3	0.33	$2\tau_{LUT}$

ing their inputs so as to exploit the dual 5-input LUT option also for these GPCs.

To be more specific, consider those GPCs which have 5 input or less in the lowest two ranks, i.e. satisfying the condition

$$K_0 + K_1 \leq 5 \tag{4.19}$$

Correspondingly, let s_0 and s_1 be the lowest-rank output bits. These output bits does not depend on the input bits of higher ranks. Thanks to the condition above, these two bits can be generated by a single LUT configured with the dual LUT option, which takes as inputs the $K_0 + K_1$ lowest-rank input bits. This implementation reduces the cost of the GPC without adversely affecting any other metrics; in other words, this implementation is covering according to Eq. (4.15). Therefore, this implementation of each suitable GPC simply replaces the straightforward one in the library.

The same argument does not hold for output bits of higher ranks: the output bit s_i of rank i depends on all the input bits of rank equal or lower than i, due to the carry. However, if an increase in the delay of the GPC can be accepted, a different mapping can be employed, where the carry from the lowest rank, c_1, is computed by the same LUT that computes s_0. This allows for a reduction in the number of inputs required to compute the other outputs, which can therefore be computed by half the LUTs, configured with the dual LUT option. This mapping can be applied under a looser condition than the one for the mapping described above, which is

$$K_0 \leq 5 \tag{4.20}$$

However, mappings with carry do not lead to covering GPCs due to the increase in the delay, which implies that Eq. (4.15c) is not satisfied. For this reason, Table 4.2 lists this mapping as an **alternative** for each suitable GPC, which cannot be used on the critical path of a design, but can lead to additional area savings if the associated cost in terms of delay can be paid. Moreover, for all GPCs with $n = 4$ satisfying Eq. (4.19), the two mappings require the same number of LUTs $\lambda = 3$, which means also the same efficiency $\varepsilon = 0.66$ due to the fact that the compression difference is in any case $\delta = 6 - 4 = 2$; but the mapping with carry has an increased delay of $\tau = 2\tau_{LUT}$ and is consequently covered by the mapping without delay.

(0,3,1,2;5) GPC is the only one satisfying Eq. (4.19) for which both the proposed optimisations can be applied, and the mapping with carry leads to a number of LUTs less than the one without carry. In this case, the mapping with carry allows to achieve the lower bound on the number of LUTs $\lambda_{inf} = 3$. On the other hand, when $n = 4$ the lower bound $\lambda_{inf} = 2$ is not reached due to the necessity of computing the additional output c_1.

Chapter 5

Efficient Mapping of SHA-2 on FPGA

Most of SHA-2 implementations discussed in Chapter 2 are characterised by RTL optimisations. Since these optimisations does not take into account any specific feature of the target technology, the resulting designs are platform-independent, meaning that they can be efficiently implemented on any target platform leading to some improvement. However, when particularly strict constraints are to be satisfied, platform-dependent designs are needed to obtain maximum performance from the target technology. This chapter shows how to efficiently implement a SHA-256 accelerator on the 7-series Xilinx FPGA family.

5.1 Overview of 7-series Xilinx FPGA architectural features

This section provides a brief summary of the architectural features of the 7-series Xilinx FPGA family that have been used to implement the SHA-256 accelerator. It is not intended to provide a comprehensive description of the 7-series architecture, which the interested reader can find in the 7-series user guide [128] and the 7-series library guide [132]

5.1.1 7-series Xilinx FPGA organization overview

The basic logic element in the 7-series FPGA is called a *slice*. A slice contains:

- 4 function generators, implemented as LUTs of the type described in Section 4.3.1;
- 8 flip-flops, which can be configured to register one of the two outputs of a corresponding function generator, or a signal from the overall switch matrix;
- multiplexers to connect outputs of the function generator within the slice, in order to implement any arbitrary logic function of up to 8 inputs;
- a fast lookahead carry logic component, to perform arithmetic operations.

Slices are arranged into *Configurable Logic Blocks (CLBs)*. Each CLB contains two slices, which are independently connected to the overall switch matrix, and are not connected between them. In particular, each of the two slices of a CLB is placed along a different column, where slices aligned on the same column from different CLBs have their carry chains connected, in order to form longer vertical carry chains.

5.1.2 LUT capabilities

Apart from the function generator capability described in Section 4.3.1, 7-series LUTs offer a number of additional functionalities. The ones relevant for the SHA-256 accelerator implementation are described here.

Shift register

Each function generator can be configured as a 32-bit shift register, implemented without using any of the flip-flops available in the slice. This configuration is supported via dedicated pins, included those for the clock and clock enable signals.

It is worth noting that such a shift register is initialised at programming time, and cannot be set or reset when the FPGA is operational, since this shift register has no parallel input.

A LUT-based shift register has no parallel output either, therefore only the serial output of the shift register can be read. However, an additional bit within the shift register can be read through the O6 LUT output, and this bit can be dynamically chosen via the LUT inputs, as shown in Fig. 5.3. When the LUT is configured as shift register, LUT input signals are called address bits.

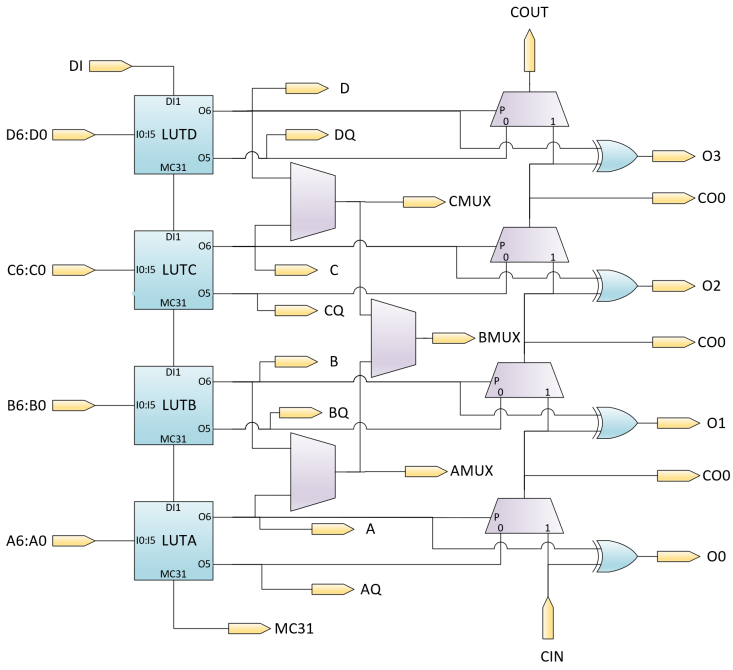


Figure 5.1: Very simplified architecture of a slice. LUTs, function multiplexers, the carry logic and the shift register connections are shown. Signal marked as outputs can be chosen as actual slice outputs, but not all of them simultanously. The 8 flip-flops are not shown since they can buffer any of the outputs. For more detailed information, the interest reader is referred to [128].

This feature can be used to build shift register shorter than 32 bits, by selecting the depth via the address bits. This implies that, for shorter shift registers, no intermediate bit can be read. In particular, two shift registers up to 16-bit can be implemented in the same LUT if they share the address bits. Shift registers longer than 32 bits, up to 128 bits, can be built within a slice via the function multiplexers.

ROM

Function generators can also be configured as 64×1-bit ROM. This means that ROMs up to 256×1-bit can be implemented with a single slice, without using any flip-flop. The ROM value is provided at

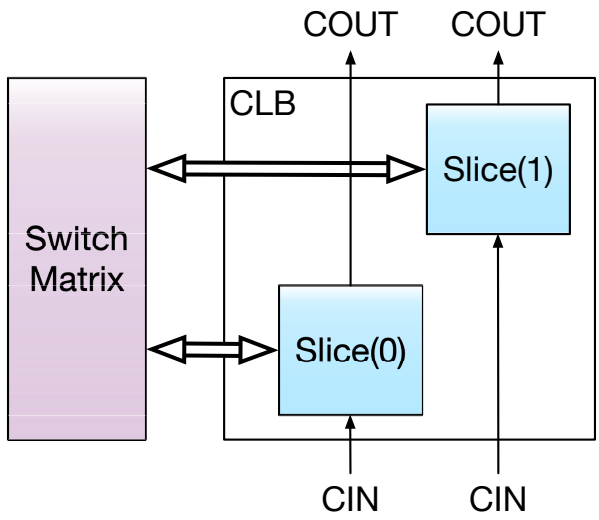


Figure 5.2: Architecture of a CLB. Adapted from [128]

programming time.

5.2 Efficient SHA-256 implementation on 7-series Xilinx FPGA

5.2.1 Compressor

From Eq. (2.1) it is clear that the Compressor is mainly constituted by additions, plus a set of logic functions. Therefore, the efficient implementation of the Compressor requires an efficient implementation of the multi-operand additions.

The first architectural choice concerns the logic functions. Due to their bitwise nature, in principle these functions could be fused into the first stage of the multi-operand adder consuming their results, and the inputs of the logic function becomes input to the modified adder. However, the resulting increase in the number of input operands would result in an area increase largely outweighting the area saving due to the fusion. This is clear if each logic function is regarded, at the bit level, as a GPC. Each logic function takes 3 bits as inputs and produces a single bits as output, so its "compression difference" is $\delta = 3 - 1 = 2$. Each logic function can be implemented in a single LUT, so $\lambda = 1$ and its

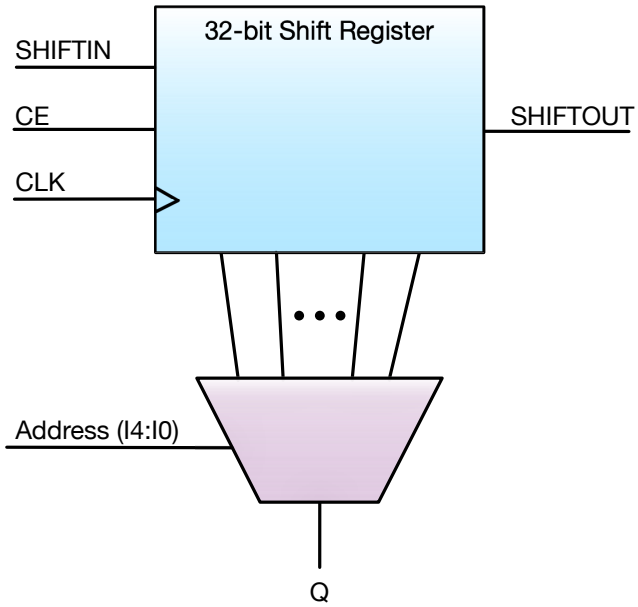


Figure 5.3: LUT-based shift register. Adapted from [128]

"area efficiency" is $\varepsilon = {}^{2}/_{1} = 2$. Looking at the GPC library reported in Table 4.2, it is clear that such a value of area efficiency is higher than those of any of the available GPC. Also when GPCs with more than 6 inputs are considered, recent works in the literature [58, 62, 134] show that 2 is currently an upper bound for the area efficiency of GPCs.

The argument against the fusion of the logic functions into the compressor trees is definitely won by observing that the area efficiency of such implementation can be greater than 2, since two logic functions can be implemented in the same LUT, obviously taking advantage of the dual LUT option, with a proper rearrangement of bits. Namely, a single LUT can compute the i-th bit of Maj and the $(i+2)$ mod 32-th bit of Σ_0, since these two functions end up sharing the input but A_i, and the whole pair of function can be regarded as a 5-input, 2-output function; the same applies for the i-th bit of Ch and the $(i+6)$ mod 32-th bit of Σ_1. The corresponding truth tables are reported in Table 5.1 and Table 5.2 respectively.

For the additions, one option is to design two separate adders for the computaton of A_t and E_t, the first being a 7-input adder and the second

Table 5.1: Combined truth table for the Maj and Σ_0 functions. Operations on indexes are intended modulo 32.

$C(i)$	$B(i)$	$A(i)$	$A(i-20)$	$A(i-11)$	$Maj(i)$	$\Sigma_0(i+2)$
1	1	1	1	1	1	1
1	1	1	1	0	1	0
1	1	1	0	1	1	0
1	1	1	0	0	1	1
1	1	0	1	1	1	0
1	1	0	1	0	1	1
1	1	0	0	1	1	1
1	1	0	0	0	1	0
1	0	1	1	1	1	1
1	0	1	1	0	1	0
1	0	1	0	1	1	0
1	0	1	0	0	1	1
1	0	0	1	1	0	0
1	0	0	1	0	0	1
1	0	0	0	1	0	1
1	0	0	0	0	0	0
0	1	1	1	1	1	1
0	1	1	1	0	1	0
0	1	1	0	1	1	0
0	1	1	0	0	1	1
0	1	0	1	1	0	0
0	1	0	1	0	0	1
0	1	0	0	1	0	1
0	1	0	0	0	0	0
0	0	1	1	1	0	1
0	0	1	1	0	0	0
0	0	1	0	1	0	0
0	0	1	0	0	0	1
0	0	0	1	1	0	0
0	0	0	1	0	0	1
0	0	0	0	1	0	1
0	0	0	0	0	0	0

Table 5.2: Combined truth table for the Ch and Σ_1 functions. Operations on indexes are intended modulo 32.

$G(i)$	$F(i)$	$E(i)$	$E(i-19)$	$E(i-5)$	$Ch(i)$	$\Sigma_1(i+6)$
1	1	1	1	1	1	1
1	1	1	1	0	1	0
1	1	1	0	1	1	0
1	1	1	0	0	1	1
1	1	0	1	1	1	0
1	1	0	1	0	1	1
1	1	0	0	1	1	1
1	1	0	0	0	1	0
1	0	1	1	1	0	1
1	0	1	1	0	0	0
1	0	1	0	1	0	0
1	0	1	0	0	0	1
1	0	0	1	1	1	0
1	0	0	1	0	1	1
1	0	0	0	1	1	1
1	0	0	0	0	1	0
0	1	1	1	1	1	1
0	1	1	1	0	1	0
0	1	1	0	1	1	0
0	1	1	0	0	1	1
0	1	0	1	1	0	0
0	1	0	1	0	0	1
0	1	0	0	1	0	1
0	1	0	0	0	0	0
0	0	1	1	1	0	1
0	0	1	1	0	0	0
0	0	1	0	1	0	0
0	0	1	0	0	0	1
0	0	0	1	1	0	0
0	0	0	1	0	0	1
0	0	0	0	1	0	1
0	0	0	0	0	0	0

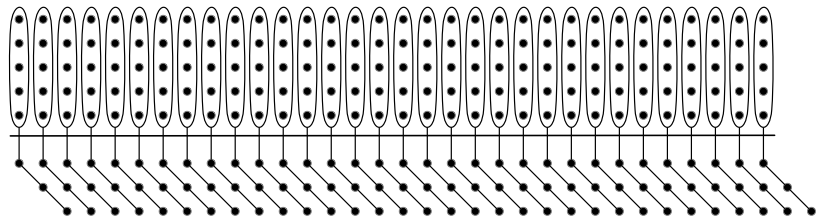


Figure 5.4: Proposed Compressor Tree for the addition within the SHA-256 Compressor

being a 6-input adder. Although that solution may produce a slightly faster compressor, area considerations suggests that it is more efficient to calculate the ${T^1}_t$ sum first, according to Eq. (1.6), and then compute

$$\begin{aligned} A_t &= {T^1}_t + \Sigma_0 (A_t) + Maj (A_t, B_t, C_t) \\ E_t &= {T^1}_t + D_t \end{aligned} \tag{5.1}$$

taking into account the availability of the ternary adder [104].

The 5-input adder can be efficiently implemented with a compressor tree of only one level, constituted of (0,5;3) GPCs, as shown in Fig. 5.4, and is completed by a ternary adder. As shown in Table 4.2, each (0,5;3) GPCs requires 2 LUTs, and a GPC per bit is required. The overall cost of the 5-input adder, reported in Table 5.3, is obtained by recalling that an I-bit ternary adder, as well as a I-bit binary adder, requires I LUTs to be implemented.

Table 5.3: Cost of the Compressor

Component	**Quantity**	**LUT**	**FF**
Logic function	2	32	0
5-input adder	1	92	0
Binary/ternary adder	2	32	0
Total		**224**	**0**

5.2.2 Expander

The implementation of the Expander discussed in Section 3.2.2 is particularly expensive, since it requires a flip-flop and two multiplexers per

bit of the input message, which for SHA-256 are 512. This cost can be reduced by removing the parallel behaviour of the shift register. Doing so allows for the saving of the two multiplexers per input bit, but places a restriction on the input pattern of the circuit, since the input message must now be presented serially to the circuit, in order to be fed into the shift register. More precisely, the 512-bit input message must be splitted into 16 32-bit words, which must be presented in little-endian fashion in the first 16 cycles.

Having removed the parallel behaviour, the shift register of the Expander can be implemented by LUTs, taking advantage of the shift register capability discussed in Section 5.1.2. In this way, the number of flip-flops needed by the Expander is greatly reduced, balancing the number of flip-flops required to store the state variables and the final result. Only 32 flip-flops are now required by the Expander, to implement a single register which is used both to store the current word of the input message, and to decouple the combinatorial path of the Expander from the one of the Compressor.

Implementing the shift register with LUTs poses another challenge: according to Eq. (1.4), three intermediate values are required to compute the value of W_{t+16}. As mentioned in Section 5.1.2, from shift registers shorter than 32 bit no intermediate value can be read. The solution is to replicate the shift register chain four times, each one with different length, in order to produce all the required values for the computation of Eq. (1.4). Since the chains have different lengths and therefore different address bits, they cannot share LUTs.

The proposed architecture for the Expander is illustrated in Fig. 5.5. The two logic functions σ_0 and σ_1 are 3-input, 1-output logic functions at the bit level, without sharing any of their inputs: therefore, each of them needs 1 LUT per bit, or 32 LUTs in total. The implementation of the multiplexer is described in Section 5.2.3.

The last component to implement is the adder, which has to sum 4 operands. The best solution for the compressor tree of this adder has been found to be the one employing (0,3,3;4) GPCs. The resulting compressor tree, shown in Fig. 5.6, has only one level and is completed by a ternary adder.

A further optimisation has become possible thanks to the fact that the resulting delay of the Expander obtained so far was lower than the delay of the Compressor. This has allowed for the employment of the area-optimised version of the (0,3,3;4) GPC, further reducing the cost of the adder without altering the overall critical path.

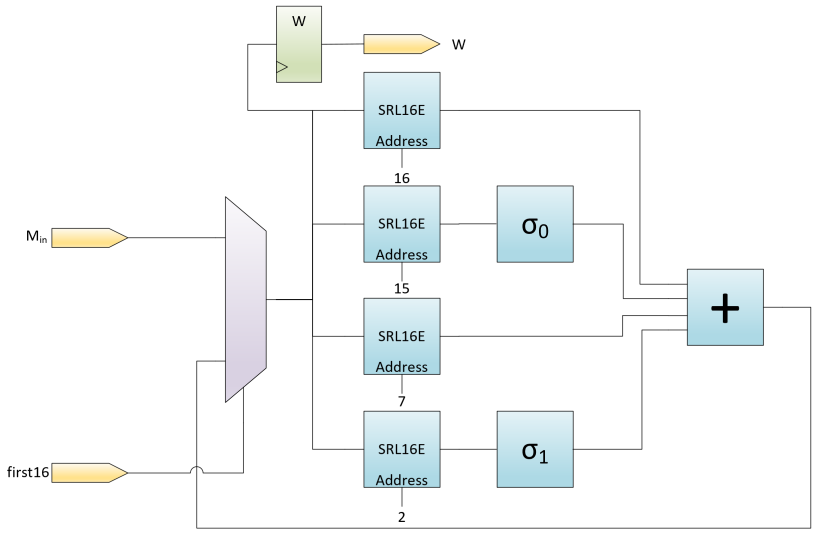


Figure 5.5: Architecture of the Expander with redundant LUT-based shift registers. `SRL16E` is the name used in the Vivado 7-series library [132] to instantiate a LUT-based shift register.

Table 5.4: Cost of the Expander

Component	**Quantity**	**LUT**	**FF**
Shift register	4	32	0
Logic function	2	32	0
Multiplexer	1	16	0
4-input adder	1	80	0
Buffer register	1	0	32
Total		**288**	**32**

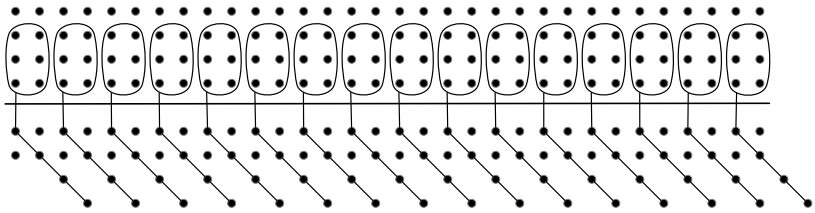


Figure 5.6: Proposed Compressor Tree for the addition within the Expander. Note that the third and fourth rows of the output do not overlap at any rank, therefore these two row constitute a single output operand.

5.2.3 Other Components

ROM

The constants ROM is implemented with LUTs, taking advantage of the fact that, for SHA-256, the depth of the constants ROM matches exactly the ROM depth that can be implemented in a single LUT, as described in Section 5.1.2. This means that 32 LUTs are enough to implement the whole component.

Table 5.5: Cost of the constants ROM

Component	Quantity	LUT	FF
64×32-bit ROM	1	32	0
Total		**32**	**0**

Multiplexers

Binary multiplexers are implemented by LUTs, and their cost is optimised with an observation that allows to take advantage of the 6-input, 2-output LUTs of the 7-series family.

A 2:1 multiplexer is a 3-input, 1-output logical function. As described in Section 4.3.1, 7-series LUTs cannot implement two arbitrarily defined 3-input functions, hence a straightforward implementation of a $2n$:n multiplexer requires n LUTs. Nevertheless, the n pairs of bits to be multiplexed share the same selection signal. Therefore, the multiplexing of two pairs of bits can be regarded as a 5-input, 2-output logical

function, which can be implemented in a single LUT configured with the dual LUT option.

In the SHA-256 data path, 8 64:32 multiplexers are required to multiplex the state variables $Z(k)$.

Table 5.7: Cost of the multiplexers

Component	Quantity	LUT	FF
64:32 multiplexer	8	32	0
Total		**256**	**0**

5.2.4 Data Path

The overall data path of the proposed SHA-256 architecture is shown in Fig. 5.7. The data path includes registers for storing the accumulator variables and the final hash value, and binary adders to compute Eq. (1.13). In a straighforward implementation, 8 adders would be required to perform this calculation in a single clock cycle. However, due to Eq. (1.12), the final value of accumulator variables A_t to D_t and E_t to H_t appears in the A and E register respectively, up to the fourth clock cycle from the last. Therefore, the architecture originally proposed in [18] can be used to compute Eq. (1.13) with only two adders in the last four clock cycles. Two additional $4:1 \times 32$, or $128:32$, multiplexers are required to select the proper value of the primary input IV to use at each clock cycle. Since each $4:1$ multiplexer is a 6-input, 1-output logical function, each of these multiplexers require 32 LUTs, the same cost of a binary adder. In total, the original cost of 8 binary adders can be halved to the cost of two adders and two $128:32$ multiplexers.

The data path is completed by three LUTs, required to compute the particular values 0, 14 and 60 of the stage counter. The first is used to drive the rolling multiplexer, which only for the first round must fed the Compressor with the externally provided initialisation values. The value 14 is used to signal when the initial loading phase of the Expander is over, according to Eq. (1.4), while the value 60 is employed to signal to the Control Unit the start of the final sum. Values 14 and 60, and not 15 and 61, are detected since a delay of a clock cycle introduced by the FSM needs to be taken into account. Finally, a flip-flop is used to produce a completion output signal, by delaying the overflow output of the counter of one clock cycle to take into account the final sum.

Table 5.6: Truth table for a 4:2 multiplexer

sel	y_1	x_1	y_0	x_0	o_1	o_1
1	1	1	1	1	1	1
1	1	1	1	0	1	1
1	1	1	0	1	1	0
1	1	1	0	0	1	0
1	1	0	1	1	1	1
1	1	0	1	0	1	1
1	1	0	0	1	1	0
1	1	0	0	0	1	0
1	0	1	1	1	0	1
1	0	1	1	0	0	1
1	0	1	0	1	0	0
1	0	1	0	0	0	0
1	0	0	1	1	0	1
1	0	0	1	0	0	1
1	0	0	0	1	0	0
1	0	0	0	0	0	0
0	1	1	1	1	1	1
0	1	1	1	0	1	0
0	1	1	0	1	1	1
0	1	1	0	0	1	0
0	1	0	1	1	0	1
0	1	0	1	0	0	0
0	1	0	0	1	0	1
0	1	0	0	0	0	0
0	0	1	1	1	1	1
0	0	1	1	0	1	0
0	0	1	0	1	1	1
0	0	1	0	0	1	0
0	0	0	1	1	0	1
0	0	0	1	0	0	0
0	0	0	0	1	0	1
0	0	0	0	0	0	0

Table 5.8: Cost of the proposed SHA-256 architecture

Component	LUT	FF
Compressor	224	0
Expander	288	32
Constants ROM	32	0
State register	0	256
Hash register	0	256
Final addition	128	0
Rolling multiplexer	128	0
Top level entity	3	1
Total	**804**	**545**

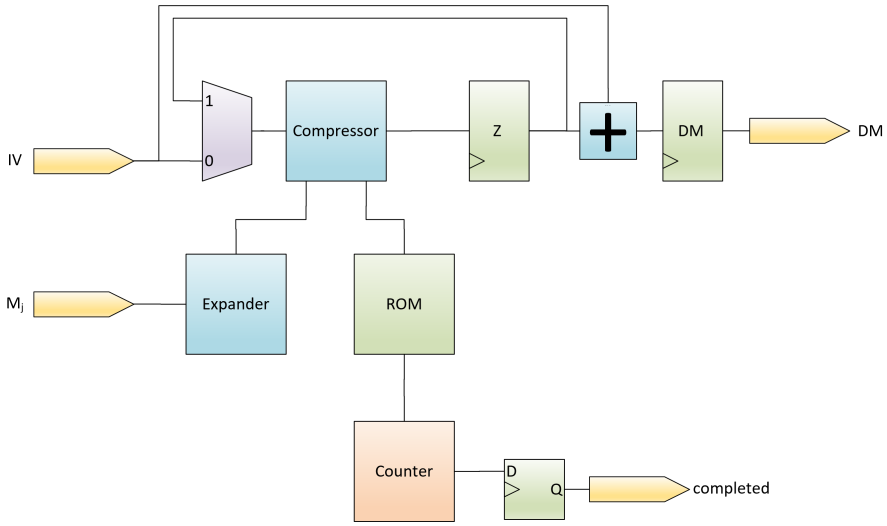


Figure 5.7: Architecture of the proposed SHA-256 implementation

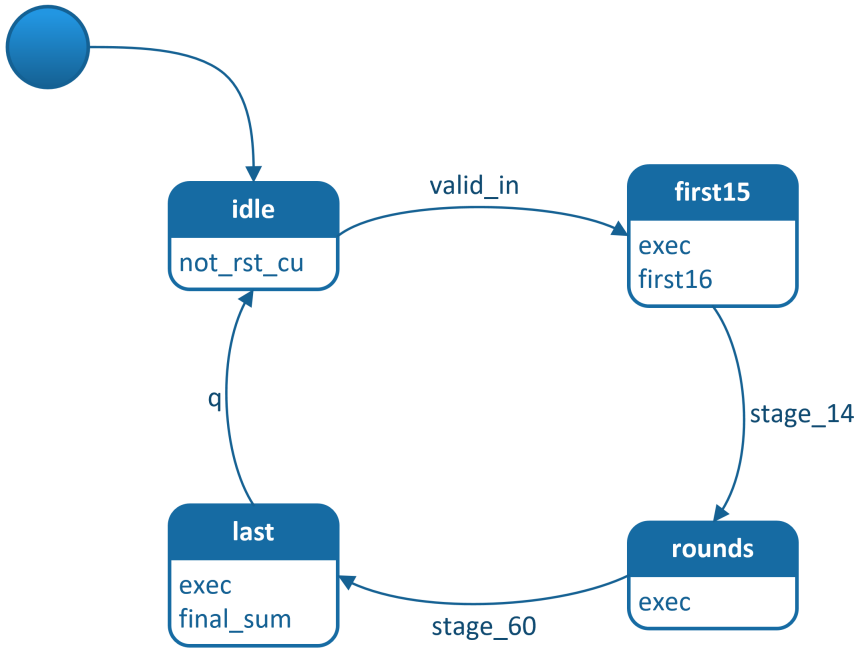


Figure 5.8: Finite State Machine for the optimised SHA-256 implementation

Control Part

The Control Part is composed of a counter for the stages, the value of which is needed as the address of the ROM, and the simple FSM shown in Fig. 5.8. The Control Unit has three tasks to undertake. First, it must enable the round computation until all the stages have been computed; this is done via the `exec` signal which enables both the counter and the stage registers. Second, to avoid off-by-one errors due to the final stage implementing Eq. (1.13) a reset of the counter at the end of each computation is necessary. The last task is to enable the hash registers only during the final four rounds, avoiding unnecessary writes to save energy.

Both the counter and the FSM have been implemented at behavioural level, leaving the task of their actual synthesis to the Vivado tool. Therefore, their cost has not been accurately computed, but it can be safely assumed that it must be of the order of the tens of LUTs and flip-flops.

5.3 Experimental Results

The proposed SHA-256 implementation has been described in VHDL and synthesized, placed and routed with the Xilinx Vivado IDE 2019.2 for the Xilinx Kintex UltraScale+ XCKU5P FPGA [130, 131]. Since the proposed design is meant to be used as a part of a larger system, the VHDL description has been synthesized in *Out of Context* mode.

Table 5.9 shows the post-implementation data reported by the tool, and compares them with their corresponding expected values, reported in Table 5.8. Moreover, Table 5.9 reports also the number of slices required by the design. Unfortunately, this piece of data is not provided by the Vivado tool, but it can be estimated from the knowledge of the architecture of the FPGA introduced in Section 5.1.1. To be more specific, the minimum number of slices required by a design can be estimated as:

$$Slices = \min\left(\frac{LUT}{4}, \frac{FF}{8}\right) \tag{5.2}$$

where LUT and FF indicate respectively the number of LUTs and flip-flops required by the design as reported by the Vivado tool. Actually, the number of slices computed through Eq. (5.2) is the theoretical *minimum* number of slices required by the design. In fact, placing and routing constraints, or usability constraints of logic elements within the same slice, may result in an higher number of slices truly occupied when the design is programmed onto the FPGA.

It can be seen that the obtained results closely resemble the predicted values, especially for the number of slices, which strictly matches the predicted one[1]. A closer exploration of data provided by the Vivado tool allows to determine the source of the difference in the number of flip-flops: as it was expected, the 36 additional flip-flops are the ones required to implement the Control Part, which were not taken into account into the theoretical estimation, as described in Section 5.2.4. A similar number of LUTs is also required to implement the Control Part, but this is largely absorbed by savings that the tool is capable of obtaining from other parts of the design.

[1] The predicted number of slices is computed through Eq. (5.2) using the predicted values for the number of LUTs and flip-flops provided in Table 5.8.

Table 5.9: Implementation results for the proposed SHA-256 design. The number of slices is computed according to Eq. (5.2). In brackets the difference versus the theoretically estimated values reported in Table 5.8.

LUTs	Flip-flops	Slices	Critical delay
807 (+3)	581 (+36)	202 (+1)	3.5 ns

5.3.1 Comparison with the State of the Art

To assess the validity of the proposed approach, Table 5.10 compares the results obtained by the proposed SHA-256 design with other implementations of the same SHA-2 family member. Table 5.10 reports data provided by the authors in their proposals, so only designs originally implemented on Xilinx Virtex FPGAs have been considered. This is exactly the issue tackled by the evaluation framework presented in Chapter 3, which allows for the fair comparison of different designs originally implemented on different targets by greatly simplifying the implementation of the Compressor. However, the SHA-256 design proposed here involves optimisations of the whole data path, and in particular a completely different implementation of the Expander, therefore falling out of the scope of the framework in its current form, as mentioned in Section 3.2.6.

The proposed SHA-256 design requires less slices, in many cases by far, when compared with architectures with similar or superior throughput. In particular, proposals heavily oriented on the maximisation of throughput such as [81, 78], are willing to pay a significant area price for achieving this objective. This trade-off is reasonable, since these proposals achieve a decent area efficiency, however the area efficiency of the proposed SHA-256 design is clearly superior.

Proposals oriented to area-constrained scenarios can achieve even less area occupation than the proposed SHA-256 design, such as [125, 41], but they do so by using techniques which have a hugely negative impact on performance, in particular the loop folding technique, described in Section 2.1.3. The reduction in throughput leads to a reduction in the area efficiency metric, which is considerably lower than the one of the proposed SHA-256 design.

By exploiting low-level features of the target FPGA, the proposed SHA-256 design achieves the best area efficiency reported in the literature, with a reasonably high throghput. It is also worth noting that the target FPGA has been one of the Kintex-5 family, which is significantly

Table 5.10: Comparison of the proposed SHA-256 implementation with other implementations in the literature on the same FPGA family

Proposal	**FPGA**	**Slices**	**Frequency** (MHz)	**Cycles per hash**	**Throughput** ($\mathrm{Mbit\,s^{-1}}$)	**Area Efficiency**
This one	Kintex-5	233	285.71	65	2250.55	9.66
[125]	Virtex-6	197	354	129	1405.02	7.13
[53]	Virtex-5	387	202.54	65	1595.39	4.12
[50]	Virtex-5	2796	179.08	64	1432.64	0.51
[82]	Virtex-5	N/D	N/D	N/D	1539.60	1.13
[78]	Virtex-7	1402	204	32	3264	2.33
[41]	Virtex-5	139	64.45	280	117.85	0.85
[4]	Virtex-2	1149	114.55	65	902.30	0.79
[81]	Virtex-6	1831	172	8	11008	6.01

less performant in itself than the Virtex family employed by the other proposals, further confirming the validity of the approach.

Conclusion

Choosing the best accelerator architecture for the hardware implementation of SHA-2 under the strict set of requirements imposed by one of the emerging applications is neither an easy task nor a straightforward one. Each design technique has often manifold implications on the application metrics of performance, area occupation, and power or energy consumption. What is more, the application of a sequence of techniques can yield combined effects which are more than the sum of the effects of each technique. The discussion of Chapter 2 highlighted that loop unrolling is particularly useful when performance is the main design objective, not only for the benefits brought on its own, but mainly for its capability of enabling other performance-improving optimisations. On the other hand, designs for resource-constrained environment will avoid loop unrolling due to its severely adverse impact on area occupation; in such cases the preferred design technique turned out to be loop folding.

Application with mixed requirements will find their specific place in the design space thanks to the methodology presented in Chapter 3. The SHA-2 workbench proposed there has been proven capable of providing insights into the interplay between design techniques and the underlying target technology which cannot be exposed by theoretical analysis, while allowing for a fair and time-effective comparison between different alternatives. As shown in the end of the chapter, the framework can successfully be employed to determine which technique is more suitable to meet the specific requirements of any given application.

The approach of Chapter 3 is the best one to follow when portability of the design between different target technologies is in the set of requirements. On the other hand, when the target technology is fixed, a deep knowledge of its features allows the designer to reach points in the design space which are unaccessible to the methodology of Chapter 3. This has been proven in Chapter 5, where the design for a SHA-2 accelerator tailored for the Xilinx 7-series FPGA family has been proven to have the best area efficiency reported so far in the literature.

It is worth stressing that, apart for the already mentioned lack of portability, the technique proposed in Chapters 4 and 5 comes with the drawback of requiring a great effort to the designer, who has to manually translate logical functions into their actual components, as done in Chapter 4 for the multi-operand adder. For some of the applications presented in Chapter 1 requirements are so strict that such an effort is necessary, but for other applications requirements can be matched with dramatically less effort, which implies less time-to-market, by following the methodology of Chapter 3. In conclusion, the more narrow, "vertical" approach of Chapters 4 and 5 and the more extensive, "horizontal" one of Chapters 2 and 3 must be seen as different alternatives to be exploited in different contexts.

Bibliography

[1] Imtiaz Ahmad and A. Shoba Das. "Hardware implementation analysis of SHA-256 and SHA-512 algorithms on FPGAs". In: *Computer and Electrical Engineering* 31 (6 2005), pp. 345–360. DOI: 10.1016/j.compeleceng.2005.07.001.

[2] F. Aisopos et al. "A Novel High - Throughput Implementation of a Partially Unrolled SHA-512". In: *MELECON 2006 - IEEE Mediterranean Electrotechnical Conference*. 2006, pp. 61–65. DOI: 10.1109/melcon.2006.1653036.

[3] Konstantinos Aisopos et al. "High throughput implementation of the new Secure Hash Algorithm through partial unrolling". In: *SiPS 2005 - IEEE Workshop on Signal Processing Systems, Design and Implementation*. 2005, pp. 99–103. DOI: 10.1109/SIPS.2005.1579846.

[4] I. Algredo-Badillo et al. "FPGA-based implementation alternatives for the inner loop of the Secure Hash Algorithm SHA-256". In: *Microprocessors and Microsystems* 37 (6-7 2012), pp. 750–757. DOI: 10.1016/j.micpro.2012.06.007.

[5] Ignacio Algredo-Badillo et al. "Novel hardware architecture for implementing the inner loop of the SHA-2 algorithms". In: *DSD 2011 - 14th Euromicro Conference on Digital System Design*. 2011. DOI: 10.1109/DSD.2011.75.

[6] Ignacio Algredo-Badillo et al. "Throughput and Efficiency Analysis of Unrolled Hardware Architectures for the SHA-512 Hash Algorithm". In: *ISVLSI 2012 - 10th IEEE Computer Society Annual Symposium on VLSI*. 2012, pp. 63–68. DOI: 10.1109/ISVLSI.2012.63.

[7] Amara Amara, Frédéric Amiel, and Thomas Ea. "FPGA vs. ASIC for low power applications". In: *Microelectronics Journal* 37 (8 2006), pp. 669–677. DOI: 10.1016/j.mejo.2005.11.003.

[8] George S. Athanasiou et al. “Optimising the SHA-512 cryptographic hash function on FPGAs”. In: *IET Computers & Digital Techniques* 8 (2 2014). DOI: 10.1049/iet-cdt.2013.0010.

[9] Ling Bai and Shuguo Li. “VLSI Implementation of High-Speed SHA-256”. In: *ASICON 2009 - 8th IEEE International Conference on ASIC*. 2009. DOI: 10.1109/ASICON.2009.5351591.

[10] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. “Keying Hash Functions for Message Authentication”. In: *CRYPTO 1996 - 16th Annual International Cryptology Conference*. Berlin, Heidelberg: Springer, 1996. DOI: 10.1007/3-540-68697-5_1.

[11] Luca Benini and Giovanni De Micheli. “Static Assignment for Low Power Dissipation”. In: *IEEE Journal of Solid-State Circuits* 30 (3 1995), pp. 158–268. DOI: 10.1109/4.364440.

[12] John Black, Martin Cochran, and Trevor Highland. “A Study of the MD5 Attacks: Insights and Improvements”. In: *FSE 2006 - 12th International Conference on Fast Software Encryption*. Berlin, Heidelberg: Springer, 2006. DOI: 10.1007/11799313_17.

[13] Joseph Bonneau et al. “SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies”. In: *36th IEEE Symposium on Security and Privacy*. IEEE, 2015. DOI: 10.1109/SP.2015.14.

[14] Xiaolin Cao, Liang Lu, and Maire O’Neill. “A Compact SHA-256 Architecture for RFID Tags”. In: *ISSC 2011 - 22nd IET Irish Signals and Systems Conference*. 2011, pp. 6–11.

[15] Amantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. “Low-Power CMOS Digital Design”. In: *IEEE Journal of Solid-State Circuits* 27 (4 1992), pp. 473–484. DOI: 10.1109/4.126534.

[16] Anantha P. Chandrakasan and Robert W. Brodersen. “Minimizing Power Consumption in Digital CMOS Circuits”. In: *Proceedings of the IEEE* 53 (4 1995), 498–523. DOI: 10.1109/5.371964.

[17] Ricardo Chaves et al. “Cost-Efficient SHA Hardware Accelerators”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6 (8 2008), pp. 999–1008. DOI: 10.1109/TVLSI.2008.2000450.

[18] Ricardo Chaves et al. “Improving SHA-2 Hardware Implementations”. In: *CHES 2006 - 8th Workshop on Cryptographic Hardware and Embedded Systems*. 2006. DOI: 10.1007/11894063_24.

[19] Fei Chen et al. “Enabling FPGAs in the cloud”. In: *CF 2014 - 11th ACM Conference on Computing Frontiers*. 2014. DOI: 10.1145/2597917.2597929.

[20] Konstantinos Christidis and Michael Devetsikiotis. “Blockchains and Smart Contracts for the Internet of Things”. In: *IEEE Access* 4 (2016). DOI: 10.1109/ACCESS.2016.2566339.

[21] Marco Conoscenti, Antonio Vetro, and Juan Carlos De Martin. “Blockchain for the Internet of Things: a Systematic Literature Review”. In: *AICCSA 2016 - 13th IEEE/ACS International Conference on Computer Systems and Applications*. IEEE, 2017. DOI: 10.1109/AICCSA.2016.7945805.

[22] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Research rep. 086. IACR Cryptology ePrint Archive, 2016. URL: https://eprint.iacr.org/2016/086.pdf.

[23] Aimee Coughlin et al. “Breaking the Trust Dependence on Third Party Processes for Reconfigurable Secure Hardware”. In: *FPGA 2019 - 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 2019. DOI: 10.1145/3289602.3293895.

[24] Nicolas T. Courtois, Marek Grajek, and Rahul Naik. “Optimizing SHA256 in Bitcoin Mining”. In: *CSS 2014 - 3rd International Conference on Cryptography and Security Systems*. Berlin, Heidelberg: Springer, 2014. DOI: 10.1007/978-3-662-44893-9_12.

[25] F. Crowe et al. “Single-Chip FPGA Implementation of a Cryptographic Co-Processor”. In: *FPT 2004 - 3rd IEEE International Conference on Field-Programmable Technology*. 2005, pp. 279–285. DOI: 10.1109/fpt.2004.1393279.

[26] Luigi Dadda. “Some schemes for parallel multipliers”. In: *Alta Frequenza* 34.5 (1965), pp. 349–356.

[27] Luigi Dadda, Marco Macchetti, and Jeff Owen. “An ASIC Design for a High Speed Implementation of the Hash Function SHA-256 (384, 512)”. In: *GLSVLSI 2004 - 14th ACM Great Lakes Symposium on VLSI*. 2004, pp. 421–425. DOI: http://doi.acm.org/10.1145/988952.989053.

[28] Luigi Dadda, Marco Macchetti, and Jeff Owen. “The Design of a High Speed ASIC Unit for the Hash Function SHA-256 (384, 512)”. In: *DATE 2004 - Design, Automation and Test in Europe Conference & Exhibition*. Vol. 3. 2004, pp. 70–75. DOI: 10.1109/DATE.2004.1269207.

[29] Lamping Deng, Kanwaldeep Sobti, and Chaitali Chakrabarti. "Accurate models for estimating area and power of FPGA implementations". In: *ICASSP 2008 - IEEE International Conference on Acoustics, Speech and Signal Processing*. 2008, pp. 1417–1420. DOI: 10.1109/ICASSP.2008.4517885.

[30] Lanping Deng et al. "Accurate Area, Time and Power Models for FPGA-Based Implementations". In: *Journal of Signal Processing Systems* 63 (1 2011), pp. 39–50. DOI: 10.1007/s11265-009-0387-7.

[31] Tassos Dimitriou. "A Lightweight RFID Protocol to protect against Traceability and Cloning attacks". In: *SECURECOMM 2005 - 1st International Conference on Security and Privacy for Emerging Areas in Communications Networks*. 2005, pp. 59–66. DOI: 10.1109/SECURECOMM.2005.4.

[32] James Docherty and Albert Koelmans. "A Flexible Hardware Implementation of SHA-1 and SHA-2 Hash Functions". In: *ISCAS 2011 - 44th IEEE International Symposium on Circuits and Systems*. 2011, pp. 1932–1935. DOI: 10.1109/ISCAS.2011.5937967.

[33] Sandra Dominikus. "A Hardware Implementation of MD4-Family Hash Algorithms". In: *ICECS 2002 - 9th IEEE International Conference on Electronics, Circuits, and Systems*. Vol. 3. 2002, pp. 1143–1146. DOI: 10.1109/ICECS.2002.1046454.

[34] John R. Douceur. "The Sybil Attack". In: *IPTPS 2012 - 1st International Workshop on Peer-to-Peer Systems*. Berlin, Heidelberg: Springer-Verlag, 2012. DOI: 10.1007/3-540-45748-8_24.

[35] Sylvain Ducloyer et al. "Hardware implementation of a multimode hash architecture for MD5, SHA-1 and SHA-2". In: *DASIP 2007 - Conference on Design & Architectures for Signal & Image Processing*. 2007.

[36] Ken Eguro and Ramarathnam Venkatesan. "FPGAs for trusted cloud computing". In: *FPL 2012 - 22nd International Conference on Field Programmable Logic and Applications*. IEEE, 2012. DOI: 10.1109/FPL.2012.6339242.

[37] Mohammed El-Hajj et al. "Analysis of Cryptographic Algorithms on IoT Hardware platforms". In: *CSNet 2018 - 2nd Cyber Security in Networking Conference*. 2018. DOI: 10.1109/CSNET.2018.8602942.

[38] Dmitry Evtyushkin et al. “Flexible Hardware-Managed Isolated Execution: Architecture, Software Support and Applications”. In: *IEEE Transactions on Dependable and Secure Computing* 15 (3 2018). DOI: 10.1109/TDSC.2016.2596287.

[39] Peter Fairley. “Blockchain world - Feeding the blockchain beast if bitcoin ever does go mainstream, the electricity needed to sustain it will be enormous”. In: *IEEE Spectrum* 54.10 (Oct. 2017). DOI: 10.1109/MSPEC.2017.8048837.

[40] Martin Feldhofer and Christian Rechberger. “A Case Against Currently Used Hash Functions in RFID Protocols”. In: *OTM 2006 - On the Move to Meaningful Internet Systems Workshop*. 2006, pp. 372–381. DOI: 10.1007/11915034_61.

[41] Rommel García et al. “A compact FPGA-based processor for the Secure Hash Algorithm SHA-256”. In: *Computers and Electrical Engineering* 40 (1 2014). DOI: 10.1016/j.compeleceng.2013.11.014.

[42] Feng Ge, Pranjal Jain, and Ken Choi. “Ultra-Low power and High Speed Design and implementation of AES and SHA1 Hardware cores in 65 Nanometer CMOS Technology”. In: *EIT 2009 - IEEE International Conference on Electro/Information Technology*. 2009, pp. 405–410. DOI: 10.1109/EIT.2009.5189651.

[43] Henri Gilbert and Helena Handschuh. “Security Analysis of SHA-256 and Sisters”. In: *SAC 2003 - International Workshop in Selected Areas in Cryptography*. 2004, pp. 175–193. DOI: https://doi.org/10.1007/978-3-540-24654-1_13.

[44] Ryan Glabb et al. “Multi-mode operator for SHA-2 hash functions”. In: *Journal of Systems Architecture* 53 (2-3 2007), pp. 127–138. DOI: 10.1016/j.sysarc.2006.09.006.

[45] Tim Grembowski et al. “Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512”. In: *ISC 2002 - International Conference on Information Security*. 2002, pp. 75–89. DOI: 10.1007/3-540-45811-5_6.

[46] Rafik Hamza et al. “Hash Based Encryption for Keyframes of Diagnostic Hysteroscopy”. In: *IEEE Access* 5 (2017). DOI: 10.1109/ACCESS.2017.2762405.

[47] Dirk Henrici and Paul Müller. "Hash-based enhancement of location privacy for radio-frequency identification devices using varying identifiers". In: *PERCOM 2004 - 2nd IEEE Annual Conference on Pervasive Computing and Communications Workshops*. 2004, pp. 149–153. DOI: 10.1109/PERCOMW.2004.1276922.

[48] Boeui Hong et al. "FASTEN: An FPGA-Based Secure System for Big Data Processing". In: *IEEE Design & Test* 35 (1 2018). DOI: 10.1109/MDAT.2017.2741464.

[49] S. M. Riazul Islam et al. "The Internet of Things for Health Care: A Comprehensive Survey". In: *IEEE Access* 3 (2015). DOI: 10.1109/ACCESS.2015.2437951.

[50] Chanbok Jeong and Youngmin Kim. "Implementation of Efficient SHA-256 Hash Algorithm for Secure Vehicle Communication using FPGA". In: *ISOCC 2014 - International SoC Design Conference*. 2014, pp. 224–225. DOI: 10.1109/ISOCC.2014.7087617.

[51] Marcio Juliato and Catherine Gebotys. "Tailoring a reconfigurable platform to SHA-256 and HMAC through custom instructions and peripherals". In: *ReConFig 2009 - International Conference on Reconfigurable Computing and FPGAs*. 2009, pp. 195–200. DOI: 10.1109/ReConFig.2009.40.

[52] Christoforos Kachris and Dimitrios Soudris. "A Survey on Reconfigurable Accelerators for Cloud Computing". In: *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*. 2016. DOI: 10.1109/FPL.2016.7577381.

[53] Fatma Kahri et al. "Efficient FPGA Hardware Implementation of Secure Hash Function SHA-256/Blake-256". In: *SSD 2015 - 12th IEEE International Multi-Conference on Systems, Signals & Devices*. 2015, p. 5. DOI: 10.1109/SSD.2015.7348105.

[54] S. Kent. *IP Authentication Header*. RFC 4302. RFC Editor, Dec. 2005. URL: https://tools.ietf.org/html/rfc4302 (visited on 04/29/2019).

[55] S. Kent. *IP Encapsulating Security Payload (ESP)*. RFC 4303. RFC Editor, Dec. 2005. URL: https://www.rfc-editor.org/rfc/rfc4303.txt (visited on 04/29/2019).

[56] S. Kent and S. Seo. *Security Architecture for the Internet Protocol*. RFC 4301. RFC Editor, Dec. 2005. URL: https://tools.ietf.org/html/rfc4301 (visited on 04/29/2019).

[57] M. Khalil, M. Nazrin, and Y. W. Hau. "Implementation of SHA-2 Hash Function for a Digital Signature System-on-Chip in FPGA". In: *ICED 2008 - 12th International Conference on Electronic Design*. 2008. DOI: 10.1109/ICED.2008.4786681.

[58] Burhan Khurshid and Roohie Naaz Mir. "High Efficiency Generalized Parallel Counters for Xilinx FPGAs". In: *HiPC 2015 - 22nd IEEE International Conference on High Performance Computing*. 2015, pp. 40–46. DOI: 10.1109/HiPC.2015.41.

[59] Mooseop Kim, Deok Gyu Lee, and Jaecheol Ryou. "Compact and unified hardware architecture for SHA-1 and SHA-256 of trusted mobile computing". In: *Personal and Ubiquitous Computing* 17 (5 2013), pp. 921–932. DOI: https://doi.org/10.1007/s00779-012-0543-0.

[60] Mooseop Kim, Jaecheo Ryou, and Sungik Jun. "Efficient hardware architecture of SHA-256 algorithm for trusted mobile computing". In: *INSCRYPT 2008 - International Conference on Information Security and Cryptology*. 2009, pp. 240–252. DOI: 10.1007/978-3-642-01440-6_19.

[61] Djamel Eddine Kouicem, Abdelmadjid Bouabdallah, and Hicham Lakhlef. "Internet of things security: A top-down survey". In: *Computer Networks* 141 (2018). DOI: 10.1016/j.comnet.2018.03.012.

[62] Martin Kumm and Johannes Kappauf. "Advanced Compressor Tree Synthesis for FPGAs". In: *IEEE Transactions on Computers* (2018). DOI: 10.1109/TC.2018.2795611.

[63] Martin Kumm and Peter Zipf. "Efficient High Speed Compression Trees on Xilinx FPGAs". In: *Methoden Und Beschreibungssprachen Zur Modellierung Und Verifikation Von Schaltungen Und Systemen* (1 2014).

[64] Martin Kumm and Peter Zipf. "Pipelined Compressor Tree Optimization using integer Linear Programming". In: *FPL 2014 - 24th International Conference on Field Programmable Logic and Applications*. 2014, p. 8. DOI: 10.1109/FPL.2014.6927468.

[65] Ian Kuon and Jonathan Rose. "Measuring the Gap Between FPGAs and ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26 (2 2007), pp. 203–215. DOI: 10.1109/TCAD.2006.884574.

[66] Yong Ki Lee, Herwin Chan, and Ingrid Verbauwhede. "Iteration Bound Analysis and Throughput Optimum Architecture of SHA-256 (384,512) for Hardware Implementations". In: *WISA 2007 - International Workshop on Information Security Appication*. 2007. DOI: https://doi.org/10.1007/978-3-540-77535-5_8.

[67] Roar Lien, Tim Grembowski, and Kris Gaj. "A 1 Gbit/s Partially Unrolled Architecture of Hash Functions SHA-1 and SHA-512". In: *CT-RSA 2004 - Cryptographers' Track at the RSA Conference*. 2004, pp. 324–338. DOI: 10.1007/978-3-540-24660-2_25.

[68] Marco Macchetti and Luigi Dadda. "Quasi-Pipelined Hash Circuits". In: *ARITH 2005 - 17th IEEE Symposium on Computer Arithmetic*. 2005, pp. 222–229. DOI: 10.1109/ARITH.2005.36.

[69] Lukas Malina et al. "On perspective of security and privacy-preserving solutions in the internet of things". In: *Computer Networks* 102 (2016). DOI: 10.1016/j.comnet.2016.03.011.

[70] Avijit Mathur, Thomas Newe, and Muzaffar Rao. "Defence against black hole and selective forwarding attacks for medical WSNs in the IoT". In: *Sensors* 16 (1 2016). DOI: 10.3390/s16010118.

[71] Avijit Mathur et al. "A secure end-to-end IoT solution". In: *Sensors and Actuators A: Physical* 263 (2017). DOI: 10.1016/j.sna.2017.06.019.

[72] Taeko Matsunaga, Shinji Kimura, and Yusuke Matsunaga. "Multi-Operand Adder Synthesis on FPGAs Using Generalized Parallel Counters". In: *ASP-DAC 2009 - 14th Asia and South Pacific Design Automation Conference*. 2009, pp. 337–342. DOI: 10.1109/ASPDAC.2010.5419871.

[73] Taeko Matsunaga, Shinji Kimura, and Yusuke Matsunaga. "Power and Delay Aware Synthesis of Multi-Operand Adders Targeting LUT-based FPGAs". In: *ISLPED 2011 - International Symposium on Low Power Electronics and Design*. 2011, pp. 217–222. DOI: 10.1109/ISLPED.2011.5993639.

[74] Robert P. McEvoy et al. "Optimisation of the SHA-2 Family of Hash Functions on FPGAs". In: *ISVLSI 2006 - 4th IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*. 2006, pp. 317–322. DOI: 10.1109/ISVLSI.2006.70.

[75] M. McLoone and J. V. McCanny. "Efficient Single-Chip Implementation of SHA-384 and SHA-512". In: *FPT 2002 - 1st IEEE International Conference on Field-Programmable Technology*. IEEE, 2003. DOI: 10.1109/FPT.2002.1188699.

[76] H. Michail et al. "Novel high throughput implementation of SHA-256 hash function through pre-computation technique". In: *ICECS 2005 - 12th IEEE International Conference on Electronics, Circuits, and Systems*. 2005. DOI: 10.1109/ICECS.2005.4633433.

[77] H. E. Michail et al. "High-Speed and Low-Power Implementation of Hash Message Authentication Code through Partially Unrolled Techniques". In: *MIV 2005 - 5th International Conference on Multimedia, Internet & Video Technology*. 2005, pp. 130–135.

[78] H. E. Michail et al. "On the development of high-throughput and area-efficient multi-mode cryptographic hash designs in FPGAs". In: *Integration, the VLSI Journal* 47 (4 2014), pp. 387–407. DOI: 10.1016/j.vlsi.2014.02.004.

[79] Harris Michail et al. "A Low-Power and High-Throughput Implementation of the SHA-1 Hash Function". In: *ISCAS 2005 - 38th IEEE International Symposium on Circuits and Systems*. 2005, pp. 4086–4089. DOI: 10.1109/ISCAS.2005.1465529.

[80] Harris E. Michail et al. "A Top-Down Design Methodology for Ultrahigh-Performance Hashing Cores". In: *IEEE Transactions on Dependable and Secure Computing* 6 (4 2009), pp. 255–268. DOI: 10.1109/TDSC.2008.15.

[81] Harris E. Michail et al. "On the Exploitation of a High-Throughput SHA-256 FPGA Design for HMAC". In: *ACM Transactions on Reconfigurable Technology and Systems* 5 (1 2012), 2:1–2:28. DOI: 10.1145/2133352.2133354.

[82] Anane Mohamed and Anane Nadjia. "SHA-2 Hardware Core for Virtex-5 FPGA". In: *SSD 2015 - 12th IEEE International Multi-Conference on Systems, Signals & Devices*. 2015. DOI: 10.1109/SSD.2015.7348110.

[83] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 04/29/2019).

[84] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*. FIPS 197. U.S. Department of Commerce, Nov. 26, 2011. DOI: https://doi.org/10.6028/NIST.FIPS.197.

[85] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. FIPS 186-4. U.S. Department of Commerce, July 2013. DOI: 10.6028/NIST.FIPS.186-4.

[86] National Institute of Standards and Technology. *NIST Policy on Hash Functions*. July 5, 2015. URL: https://csrc.nist.gov/projects/hash-functions/nist-policy-on-hash-functions (visited on 12/11/2017).

[87] National Institute of Standards and Technology. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. SP 800-90A Rev. 1. U.S. Department of Commerce, June 2015. DOI: 10.6028/NIST.SP.800-90Ar1.

[88] National Institute of Standards and Technology. *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*. SP 800-67 Rev.2. U.S. Department of Commerce, Nov. 2017. DOI: 10.6028/NIST.SP.800-67r2.

[89] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. FIPS 180-4. U.S. Department of Commerce, Aug. 2015. DOI: 10.6028/NIST.FIPS.180-4.

[90] National Institute of Standards and Technology. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. FIPS 202. U.S. Department of Commerce, July 5, 2015. DOI: 10.6028/NIST.FIPS.202.

[91] National Institute of Standards and Technology. *The Keyed-Hash Message Authentication Code (HMAC)*. FIPS 198-1. U.S. Department of Commerce, July 2008. DOI: 10.6028/NIST.FIPS.198-1.

[92] Karl J. O'Dwyert and David Malone. "Bitcoin Mining and its Energy Footprint". In: *ISSC 2014 / CIICT 2014 - Joint 25th IET Irish Signals & Systems Conference and China-Ireland International Conference on Information and Communications Technologies*. 2014, pp. 280–285. DOI: 10.1049/cp.2014.0699.

[93] Chris Palmer and Ryan Sleevi. *Gradually sunsetting SHA-1*. Ed. by Google Security Blog. Google. Oct. 3, 2014. URL: https://security.googleblog.com/2014/09/gradually-sunsetting-sha-1.html (visited on 04/27/2019).

[94] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. "Efficient Synthesis of Compressor Trees on FPGAs". In: *ASP-DAC 2008 - 13th Asia and South Pacific Design Automation Conference*. 2008, pp. 138–143. DOI: 10.1109/ASPDAC.2008.4483927.

[95] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. "Exploiting Fast Carry-Chains of FPGAs for Designing Compressor Trees". In: *FPL 2009 - 19th International Conference on Field Programmable Logic and Applications*. 2009, pp. 242–249. DOI: 10.1109/FPL.2009.5272301.

[96] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. "Improving Synthesis of Compressor Trees on FPGAs via Integer Linear Programming". In: *DATE 2008 - Design, Automation and Test in Europe Conference & Exhibition*. 2008, pp. 1256–1261. DOI: 10.1109/DATE.2008.4484851.

[97] Hadi Parandeh-Afshar et al. "Compressor Tree Synthesis on Commercial High-Performance FPGAs". In: *ACM Transactions on Reconfigurable Technology and Systems* 4 (4 2011), p. 19. DOI: 10.1145/2068716.2068725.

[98] Hadi Parandeh-Afshar et al. "Improved Synthesis of Compressor Trees on FPGAs by a Hybrid and Systematic Design Approach". In: *IWLS 2010 - 19th International Workshop on Logic and Synthesis*. 2010, pp. 193–200.

[99] Ana Reyna et al. "On blockchain and its integration with IoT. Challenges and opportunities". In: *Future Generation Computer Systems* 88 (2018). DOI: 10.1016/j.future.2018.05.046.

[100] Ron Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321. RFC Editor, 1992. URL: https://www.ietf.org/rfc/rfc1321.txt (visited on 04/29/2019).

[101] Farzad Samie, Lars Bauer, and Jörg Henkel. "IoT Technologies for Embedded Computing: A Survey". In: *CODES 2016 - 11th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 2016. DOI: 10.1145/2968456.2974004.

[102] Akashi Satoh and Tadanobu Inoue. "ASIC-Hardware-Focused Comparison for Hash Functions MD5, RIPEMD-160, and SHS". In: *ITCC 2005 - International Conference on Information Technology: Coding and Computing*. 2005. DOI: 10.1109/ITCC.2005.92.

[103] Li Shang, Alireza S. Kaviani, and Kusuma Bathala. "Dynamic power consumption in Virtex™-II FPGA family". In: *FPGA 2002 - 10th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 2002, pp. 157–164. DOI: 10.1145/503048.503072.

[104] James M. Simkins and Brian D. Philofsky. "Structures and methods for implementing ternary adders/subtractors in programmable logic devices". Pat. US 7,274,211 B1. Xilinx, Inc. Sept. 25, 2007.

[105] N. Sklavos and O. Koufopavlou. "Implementation of the SHA-2 hash family standard using FPGAs". In: *Journal of Supercomputing* 31 (3 2005). DOI: 10.1007/s11227-005-0086-5.

[106] N. Sklavos and O. Koufopavlou. "On the Hardware Implementations of the SHA-2 (256, 384, 512) Hash Functions". In: *ISCAS 2003 - 36th IEEE International Symposium on Circuits and Systems*. 2003. DOI: 10.1109/ISCAS.2003.1206214.

[107] William Stallings. *Cryptography and Network Security: Principles and Practice*. 7th ed. Pearson Education, Inc., 2017.

[108] William J. Stenzel, William J. Kubitz, and Gilles H. Garcia. "A Compact High-Speed Parallel Multiplication Scheme". In: *IEEE Transactions on Computers* (10 Oct. 1977), pp. 948–957. DOI: 10.1109/TC.1977.1674730.

[109] Marc Stevens et al. *Announcing the first SHA1 collision*. Ed. by Google Security Blog. Feb. 23, 2017. URL: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html (visited on 04/27/2019).

[110] Wanzhong Sun et al. "Design and Optimized Implementation of the SHA-2(256, 384, 512) Hash Algorithms". In: *ASICON 2007 - 7th International Conference on ASIC*. 2007, pp. 858–861. DOI: 10.1109/ICASIC.2007.4415766.

[111] Michael Bedford Taylor. "Bitcoin and The Age of Bespoke Silicon". In: *CASES 2013 - International Conference on Compilers Architecture and Synthesis for Embedded Systems*. IEEE, 2013. DOI: 10.1109/CASES.2013.6662520.

[112] Michael Bedford Taylor. "The Evolution of Bitcoin Hardware". In: *IEEE Computer* 50 (9 2017). DOI: 10.1109/MC.2017.3571056.

[113] Kurt K. Ting et al. "An FPGA Based SHA-256 Processor". In: *FPL 2002 - 12nd International Conference on Field Programmable Logic and Applications*. 2002. DOI: 10.1007/3-540-46117-5_60.

[114] Vivek Tiwari et al. "Reducing Power in High-performance Microprocessors". In: *DAC 1998 - 35th Design Automation Conference*. 1998, pp. 732–737. DOI: 10.1145/277044.277227.

[115] Mihai Togan, Adrian Floarea, and Gigi Budariu. "Design and implementation of cryptographic modules on FPGA". In: *European Conference for the Applied Mathematics and Informatics*. 2010, pp. 149–154.

[116] Florian Tschorsch and Björn Scheuermann. "Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies". In: *IEEE Communications Surveys & Tutorials* 18 (3 2015). Ed. by IEEE. DOI: 10.1109/COMST.2016.2535718.

[117] Hoang Anh Tuan, Katsuhiro Yamazaki, and Shigeru Oyanagi. "Three-Stage Pipeline Implementation for SHA2 Using Data Forwarding". In: *FPL 2008 - 18th International Conference on Field Programmable Logic and Applications*. 2008, pp. 29–34. DOI: 10.1109/FPL.2008.4629903.

[118] Sveinn Valfells and Jón Helgi Egilsson. "Minting Money With Megawatts". In: *Proceedings of the IEEE* 104 (9 2016). DOI: 10.1109/JPROC.2016.2594558.

[119] Matthew Vilim, Henry Duwe, and Rakesh Kumar. "Approximate Bitcoin Mining". In: *DAC 2016 - 53rd ACM/EDAC/IEEE Design Automation Conference*. 2016. DOI: 10.1145/2897937.2897988.

[120] C. S. Wallace. "A Suggestion for a Fast Multiplier". In: *IEEE Transacrions on Electronic Computers* EC-13 (1 1964), pp. 14–17. DOI: 10.1109/PGEC.1964.263830.

[121] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. "Finding Collisions in the Full SHA-1". In: *CRYPTO 2005 - 25th Annual International Cryptology Conference*. Berlin, Heidelberg: Springer, 2005. DOI: 10.1007/11535218_2.

[122] Xiaoyun Wang and Hongbo Yu. "How to Break MD5 and Other Hash Functions". In: *EUROCRYPT 2005 - 21st Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Berlin, Heidelberg: Springer, 2005. DOI: 10.1007/11426639_2.

[123] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. "Efficient Collision Search Attacks on SHA-0". In: *CRYPTO 2005 - 25th Annual International Cryptology Conference*. 2005. DOI: 10.1007/11535218_1.

[124] Stephen A. Weis et al. "Security and Privacy of Low-Cost Radio Frequency Identification Systems". In: *1st International Conference on Security in Pervasive Computing*. 2003, pp. 201–212. DOI: https://doi.org/10.1007/978-3-540-39881-3_18.

[125] Ming Ming Wong, Vikramkumar Pudi, and Anupam Chattopadhyay. "Lightweight and High Performance SHA-256 using Architectural Folding and 4-2 Adder Compressor". In: *IVLSI-SoC 2018 - IEEE/IFIP International Conference on VLSI and System-on-Chip*. 2019. DOI: 10.1109/VLSI-SoC.2018.8644825.

[126] Anthony D. Wood and John A. Stankovic. "Denial of Service in Sensor Networks". In: *Computer* 35 (10 2002). DOI: 10.1109/MC.2002.1039518.

[127] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. 2014. URL: https://ethereum.github.io/yellowpaper/paper.pdf (visited on 05/02/2019).

[128] Xilinx Inc. *7 Series FPGAs Configurable Logic Block*. User Guide 474. Version 1.8. Sept. 27, 2016.

[129] Xilinx, Inc., ed. *7 Series FPGAs Data Sheet: Overview*. Feb. 27, 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (visited on 02/21/2019).

[130] Xilinx Inc., ed. *Kintex UltraScale+ FPGA Product Brief*. 2016. URL: https://www.xilinx.com/support/documentation/product-briefs/kintex-ultrascale-plus-product-brief.pdf (visited on 02/21/2019).

[131] Xilinx Inc., ed. *UltraScale Architecture and Product Data Sheet: Overview*. Feb. 2, 2019. URL: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf (visited on 02/21/2019).

[132] Xilinx Inc. *Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide*. User Guide 953. Version 2019.2., Oct. 30, 2019.

[133] Ioannis I. Yiakoumis et al. "Maximizing the hash function of authentication codes". In: *IEEE Potentials* 25 (2 2006). DOI: 10.1109/MP.2006.1649004.

[134] Yuelai Yuan et al. "Area Optimized Synthesis of Compressor Trees on Xilinx FPGAs Using Generalized Parallel Counters". In: *IEEE Access* 7 (2019), pp. 134815–134827. DOI: 10.1109/access.2019.2941985.

[135] M. Zeghid et al. "A Reconfigurable Implementation of the New Secure Hash Algorithm". In: *ARES 2007 - 2nd International Conference on Availability, Reliability and Security*. 2007, pp. 281–285. DOI: 10.1109/ARES.2007.17.

[136] Medien Zeghid et al. “Architectural design features of a programmable high throughput reconfigurable SHA-2 Processor”. In: *JIAS - Journal of Information Assurance and Security* 3 (2 2008), pp. 147–158.

[137] Yu Zhang and Jiangtao Wen. “The IoT electric business model: Using blockchain technology for the internet of things”. In: *Peer-to-Peer Networking and Applications* 10 (4 2017). DOI: 10.1007/s12083-016-0456-1.