POLITECNICO DI TORINO
Master's Degree in Electronic Engineering

Master's Degree Thesis

Implementation and validation of a hardware countermeasure against fault injection attacks

Supervisors:

Prof. Maurizio MARTINA
Prof. Guido MASERA
Eng. Mustapha EL MAJIHI

Candidate:

Davide MUSCIA

October 2024

If you do something and it turns out well enough,
you should go ahead and do something wonderful.
Steve Jobs

Se fai qualcosa e risulta abbastanza buona,
dovresti andare avanti a fare qualcosa di meraviglioso.
Steve Jobs

Abstract

Data integrity in a processor is crucial to ensure that computations are processed without errors, preserving the consistency and reliability of the results. This is especially important in applications such as automotive, aerospace, and other critical sectors. Modern processor architectures have been shown to be vulnerable to fault injection attacks, which involve injecting errors into the circuit using simultaneous laser beams that target single or multiple data bits to compromise their integrity and extract sensitive information. This method has proven effective over the past decade through the observation of faulty behavior. Designing countermeasures against fault injection attacks has become essential to ensure data integrity, particularly with the increasing use of open-source implementations such as RISC-V, where the attacker has full knowledge of the architecture. This work proposes a fault detection methodology called «permutation-based homomorphic tags». It involves providing a redundant hardware implementation that computes arithmetic and logic operations in a permuted domain associated with a specific key. The permuted execution ensures that no faults have been injected into the processed data by preventing attackers from consistently targeting the same bits, as the permutation key is randomly changed. The outcome of this study is the hardware implementation of a permuted Arithmetic Logic Unit (ALU) and a permuted multiplier , where two different techniques were explored: an iterative approach aimed at low area consumption, and the 2-way Karatsuba algorithm for reducing latency. The architecture was implemented targeting the 64-bit RISC-V CVA6 application processor. The design was validated on Xilinx Artix-7-100T and Kintex-7 FPGAs and it was estimated the cost of the countermeasure resulted in a 7.67x area overhead and a 2.4x increase in the critical path for the ALU, a 0.31x area overhead and a 1.44x increase in the critical path for the iterative multiplier, and a 32.3x area overhead and an 8.81x increase in the critical path for the Karatsuba multiplier.

Keywords: data integrity, fault injection attacks, countermeasure, permutation-based homomorphic tags, CVA6, RISC-V

Table of Contents

List of Tables VIII

List of Figures IX

Acronyms XII

1 Introduction 1

1.0.1 Thesis objectives and structure 4

2 Environment 5

2.1 SystemVerilog 5

2.2 CVA6 6

2.2.1 RISC-V 7

3 Homomorphic security tags and permutation 8

3.1 Definition of the countermeasure 8

3.2 Permutation 10

3.3 Depermutation 12

3.4 Effectiveness countermeasure 13

4 Permuted ALU 14

4.1 Elementary operations 16

4.1.1 Extraction single bit 16

4.1.2 Set single bit 17

4.2 Boolean operations 19

4.3 Addition operations 20

4.3.1 Subtracter 25

4.3.2 Comparator 26

4.4 Shift operations 28

4.4.1 Shift left by 1 28

4.4.2 Shift left by 2 30

4.4.3 Shift left by a power of 2 32
4.4.4 Barrel shifter 34
4.5 Count operations 36
4.5.1 Population count 37
4.5.2 Leading/trailing zeros count 38

5 Permuted multiplier 40
5.1 Iterative way 40
5.2 2-way Karatsuba algorithm 44
5.2.1 Pre-computation tree 49
5.2.2 Layer of multiplication 51
5.2.3 Reconstructive tree 52

6 Validation process and analysis 56
6.1 Hardware interface 57
6.1.1 Controller 58
6.2 Software interface 59
6.3 Results 61
6.3.1 Adder and shifter comparison 61
6.3.2 ALU comparison 62
6.3.3 Multiplier comparison 63

7 Conclusion 66

Bibliography 68

List of Tables

3.1 Fredkin gate truth table 9

4.1 ALU operations supported by the CVA6 core 15
4.2 Comparison cases as function of zero bit(z) and carry-out bit($carry$) 27

6.1 Area and critical path results for CLA, permuted CLA and permuted shifter 61
6.2 Area and critical path results for ALUs 62
6.3 Area and critical path results for CVA6 multiplier, permuted Iterative multiplier and 2-way Karatsuba approach in three different run strategies 63
6.4 Possible optimizations permuted multiplier components 65

7.1 Final integration choices for the CVA6 core 66

List of Figures

1.1 Block diagram representation of redundancy techniques: a) hardware redundancy, b) temporal redundancy, c) information redundancy . . 2
1.2 Dual modular redundancy and error detection example. The Permutation blocks(P) are stages of transformation of the operands for the new alternative domain while the Depermutation block(P^{-1}) is a stage of reconstruction of the native result. 3

2.1 Placement countermeasure in CVA6 pipeline 7

3.1 Fredkin gate block 9
3.2 Dichotomous tree for 32-bit permutation function 10
3.3 Permuted dichotomous tree example for 8-bit operand 11
3.4 Depermuted dichotomous tree example for 8-bit operand 12

4.1 Extraction block 16
4.2 Extraction dichotomous tree example for 8-bit permuted operand . 17
4.3 Set block 18
4.4 Set dichotomous tree example for 8-bit permuted operand 18
4.5 Permuted AND logic function example for 8-bit operand 19
4.6 Full-Adder block with sum(s) and carry-out(c_o) generation 20
4.7 Propagation problem for Ripple-Carry Adder connection typology . 21
4.8 Full-Adder block with generate(g) and propagate(p) generation . . . 21
4.9 Unrolling carry recurrence 22
4.10 Propagate/generate block 22
4.11 Dichotomous tree example for 8-bit CLA 23
4.12 Permuted propagate/generate block 24
4.13 Dichotomous tree example for 8-bit permuted CLA 24
4.14 Subtracter adaptation for native and permuted domain 25
4.15 Comparison examples for $>$, $<$, $\geq$, $=$ 26
4.16 White block 28
4.17 Permuted shift left by 1 example with White Block(WB) 29

4.18 Permuted shift left by 2 example with White Block(WB) 30
4.19 White key block 31
4.20 Black block 31
4.21 Permuted shift left by 2 example with White Block(WB), Black Block(BB) and White Key Block(WKB) 32
4.22 Black block key 33
4.23 Permuted shift left by a power of 2 adaptation for 16-bit K_D generation with White Block(WB) and Black Block Key(BBK). 34
4.24 Barrel shifter configuration with multiplexer selection 34
4.25 SIMD example for 16-bit permuted operand 35
4.26 Leading zeros example 36
4.27 Trailing zeros example 36
4.28 Population example 36
4.29 Population counter example for 8-bit permuted operand 37
4.30 Leading/trailing zeros block 38
4.31 Leading zeros counter example for 8-bit permuted operand 39

5.1 4-bit dot notation iterative approach 40
5.2 Permuted iterative multiplier architecture 42
5.3 128-bit permuted product management 43
5.4 Partial contribution deriving the 2-way Karatsuba algorithm 44
5.5 Trichotomous tree example for 16-bit permuted 2-way Karatsuba multiplier 47
5.6 Tree-like example structure for 16-bit parallelism reduction 48
5.7 Key-extraction example for 16-bit permuted operand 49
5.8 Carry decomposition formulas for m computation 50
5.9 Carry decomposition example for m computation with 8-bit operand 51
5.10 Key extension example from 4-bit operand to 8-bit 52
5.11 Key misalignment example with difference in the MSB position . . 53
5.12 Reorder example with difference in the MSB-1 position 54
5.13 Adaptation process from key_1 to key_2 example 55
5.14 Key misalignment example in m computation 55

6.1 Functional validation architecture 57
6.2 Functional Controller graph 58

Acronyms

ALU

Arithmetic Logic Unit

ASIC

Application Specific Integrated Circuit

AXI

Advanced eXtensible Interface

BB

Black Block

BBK

Black Block Key

CA2

2's Complement

CLA

Carry-Lookahead Adder

CPU

Central Processing Unit

CSA

Carry-Save Adders

CSR

Control and Status Register

CU

Control Unit

DUT

Design Under Test

FA

Full-Adder

FPU

Floating Point Unit

FSM

Finite State Machine

FVA

Functional Validation Architecture

ISA

Instruction Set Architecture

LSB

Least Significant Bit

MMU

Memory Management Unit

MSB

Most Significant Bit

OCB

Ones Count Block

PGB

Propagate Generate Block

PMP

Physical Memory Protection

PPGB
: Permuted Propagate Generate Block

RCA
: Ripple-Carry Adder

SIMD
: Single Instructions Multiple Data

UART
: Universal Asynchronous Receiver-Transmitter

WB
: White Block

WKB
: White Key Block

Chapter 1

Introduction

In recent years, the way we interact with our surroundings has changed dramatically, especially thanks to the rise of the Internet of Things.
These networked devices consist of sensors, which are managed by an increasing number of processors running programs and exchanging data. While this represents technological advancement, it also expands the attack surface. Among all applications, the automotive, aerospace, and defense industries, which require complete reliability and security, are the most sensitive in this regard.
The reason for this concern lies in the fact that modern processor architectures have been shown to be vulnerable to **fault injection attacks**, as noted in [1], [2], and [3]. Fault injection involves introducing errors that can cause timing malfunctions or logic errors in the circuit, enabling information exfiltration from the faulty behavior and computation on false data. This means corrupting the contents of circuit registers or control signals while the circuit is still operating, using multi beams laser attack, which is a kind of attack that raises problems difficult to manage among all the possibilities(electromagnetic injection, clock glitch, power glitches,...).
The objectives of these injections can vary, with the most critical being the maintenance of **data integrity** during program execution. Indeed, it is possible to bypass security controls, such as permissions and authenticity checks, to gain complete access to sensitive information from any running critical section.
The type of attack discussed here involves physical contact with the target system, which, for example, can cause voltage or current changes, such as those that occur in a bridging fault by introducing a short circuit.
Moreover, the rise of embedded systems in both the public and private sectors has significantly expanded the vulnerability landscape. With the growing use of open-source implementations like RISC-V, where attackers have full knowledge of the architecture, the level of sensitivity becomes even higher. The adoption of these implementations has proven necessary for many companies to reduce production

costs, by benefiting from community support without internal big expenses in the specific areas of validation and verification.
Consequently, designing countermeasures against fault injection attacks has become essential to ensure data integrity. Existing countermeasures consist of both correction and **detection methods**, but we propose focusing on the latter. While correction techniques show good effectiveness, they come at a much higher cost and are not as widely adopted.
Fault detection, in particular, consists of hardware or software techniques capable of recognizing any unauthorized alterations, with the objective of maintaining system functionality, data integrity, and security.
At the hardware level, systems can adopt three mechanisms as proposed by [4]:

- **Hardware redundancy** involves duplicating the circuit section to be protected in order to compare differences with the result obtained. It is the most straightforward and simplest to implement, but it has the highest cost in terms of area, as it doubles the resource usage. It can also be applied using a multiple modular redundancy approach, with costs proportional to the number of redundancies.
- **Temporal redundancy** involves repeating operations either in reverse or through duplication, so that the result of the computation can be compared with the previous or duplicated value. Among the three methods, it has the highest cost in execution time, as the same operation is performed twice.
- **Information redundancy** involves adding state information, such as parity bits or checksum. Of the three methods, it is the most efficient in terms of resource overhead, as it only extends the parallelism of the data being transmitted.

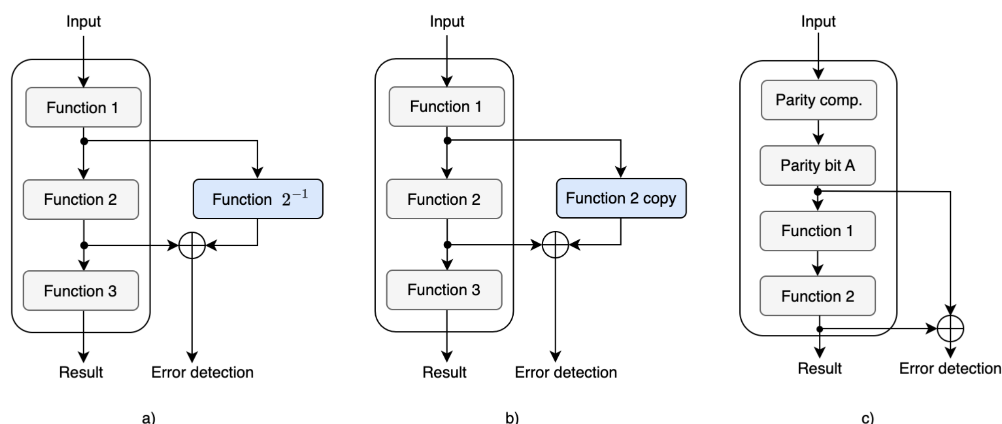


Figure 1.1: Block diagram representation of redundancy techniques: a) hardware redundancy, b) temporal redundancy, c) information redundancy

The technique discussed in this study is the dual modular hardware redundancy, which duplicates the path of interest, but this alone is not sufficient. The ability to target multiple bits simultaneously has made simple hardware replication ineffective and more vulnerable to injections. Therefore, an additional transformation of the data is necessary to enhance the system and make it more difficult for the attacker to select bits in the detection branch.
This work proposes a fault detection methodology called «**permutation-based homomorphic tags**», which provides a redundant hardware implementation that performs arithmetic and logic operations, in a permuted domain associated with a specific key that is randomly changed. The main goal is to preserve data integrity by verifying the correctness of the results.
Given that we aim to maintain data integrity in a processor, the choice of permutation is not trivial: it is required a parallel domain to the original one that operates at high frequencies and within the same clock cycle. The primary focus is to identify the best transformation for logical and arithmetic operations, and only then choose the replication area. Thus, the design, implementation, and adaptation of arithmetic-logic components, capable of handling the generated tags, are developed.
In our case, these architectural considerations were implemented targeting the 64-bit RISC-V CVA6 application core.

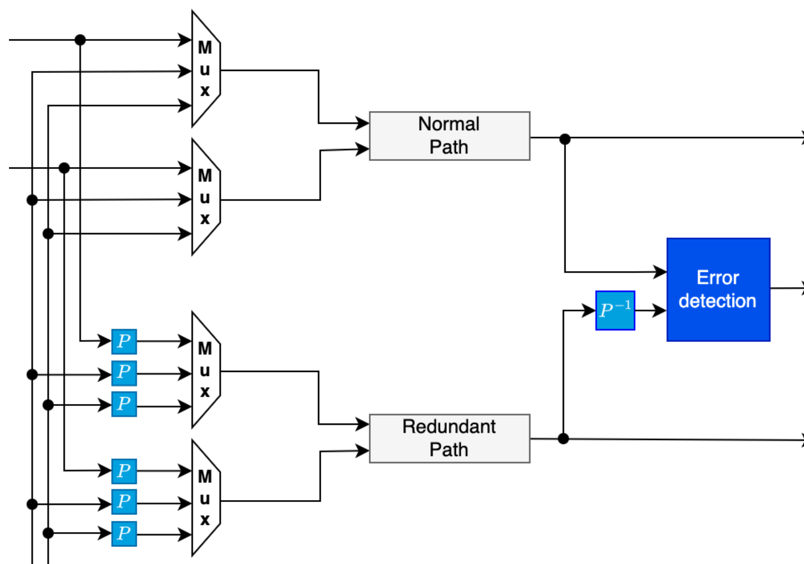


Figure 1.2: Dual modular redundancy and error detection example. The Permutation blocks(P) are stages of transformation of the operands for the new alternative domain while the Depermutation block(P^{-1}) is a stage of reconstruction of the native result.

1.0.1 Thesis objectives and structure

The objective of the internship is to develop a permuted Arithmetic Logic Unit(ALU) for a 64-bit core target, precisely the CVA6 application core of the openHW Group, that will be used by the team for Application Specific Integrated Circuit(ASIC) integration.
Here it is necessary to support all the extensions of the core according to the RISC-V Instruction Set Architecture(ISA), and then it is necessary to validate and verify all the modules developed in SystemVerilog.
After this introductory Chapter 1, the thesis is structured as follows:

- **Chapter 2**: presents a description of the countermeasure development environment. We discuss the language adopted in Section 2.1 and the CVA6 application processor, as the target core, along with its RISC-V ISA in Section 2.2.
- **Chapter 3**: introduces the permutation-based homomorphic tags countermeasure in Section 3.1, including the definition of the new representation domain and its properties.
- **Chapter 4**: presents the implementation of the arithmetic-logic components required for the realization of the redundant ALU. We start with two elementary operations, such as single-bit extraction and setting values, in Sections 4.1.1 and 4.1.2. We then address Boolean operations in Section 4.2, the addition operation and its adaptations in Section 4.3, the shift operation in Section 4.4, and count operations in Section 4.5.
- **Chapter 5**: presents the implementation of a permuted multiplier using an iterative approach in Section 5.1 and a 2-way Karatsuba approach in Section 5.2.
- **Chapter 6**: discusses the validation phase, providing a description of the validation architecture in Sections 6.1 and 6.2, followed by a discussion of the results obtained.
- **Chapter 7**: summarizes the conclusions of this thesis, including a brief overview of the decisions made regarding countermeasure integration.

Chapter 2

Environment

The topic of this thesis project is based on a previous research conducted by G. Leplus, O. Savry, L. Bossuet and M. Panigati, at CEA-Leti (Grenoble, France), which developed the theory of elementary components of countermeasures as noted in [5], [6] and [7].
Moreover the work is characterized by a dual aim: first to extend the existing adder and shifter architectures from an initial 32-bit parallelism to 64-bit parallelism; second to implement from scratch the «permuted ALU» consistent the one of the CVA6, aiming to handle all its extensions, as well as to explore a potential architecture for the multiplier, envisioned as a standalone component.
In this chapter it is introduced the hardware description language used, as proposed by the [8], [9], [10] articles, and the target core for the integration of the countermeasure as noted in [11] and [12].

2.1 SystemVerilog

SystemVerilog is the hardware description language used for countermeasure development.
It is a language that was standardized by the IEEE around 2005 with the primary goal of extending Verilog and combining VHDL to design, simulate, test and implement electronic systems.
Additionally to simplify the description of the hardware behavior from a syntactic point of view, it incorporates features of the C++ programming language, which makes easier the simulation and verification phases through a more object-oriented approach.

Its main features include:

1. **Syntax extensions and compatibility**: maintains compatibility with the previous language, so a project written in Verilog is still valid in SystemVerilog, despite the fact that the latter introduces new extensions that make writing code clearer and more fluent. For example, it introduces new data types such as *logic* in place of *wire* and *reg* without creating ambiguity and destitution between signals.

2. **Advanced data types and interconnections**: introduces from the software world the use of data types such as *enum* and *struct* that makes easier the reading of the code

3. **Classes and Assertions**: when writing Testbench, it allows an object-oriented description that can greatly simplify the code; assertions allow us to visualise that certain conditions during the test phase are fulfilled.

According to these features and the compatibility of our target core language, SystemVerilog is a versatile and ideal language for the development of our countermeasure where before one design language and another for verification were needed.

2.2 CVA6

CVA6 is a configurable 32- or 64-bit single-issue RISC-V application core developed by the OpenHW group for ASIC or FPGA implementations. Specifically, it is the name of a GitHub repository in SystemVerilog, where the CV prefix identifies it as a member of the CORE-V family, following th RISC-V specification, and the A6 indicates that it is an application class processor with a six-stage execution pipeline.
In terms of on-board processor functionality, the CVA6 is characterized by many hardware components: Control and Status Register(CSR), Advanced eXtensible Interface(AXI), L1 write-through or write-back configurable cache, optional Floating Point Unit(FPU), optional Memory Management Unit(MMU), optional Physical Memory Protection(PMP) and others.
However, our focus for implementing a countermeasure is limited to the ALU. The countermeasure in fact wants to act where the integrity of the data could be endangered, and therefore in correspondence with computation processes.
It is possible to conclude thus that the area of our competence it is mostly the Execution Stage of the pipeline, where the data manipulation takes place, and a small percentage of the general core pipeline where the permutation domain is propagated. Figure 2.1 shows this area zoomed in.

2.2.1 RISC-V

The CORE-V family is a member of a group of processors following the RISC-V specification.
It is a computer architecture started in 2021 by the Berkeley University, in order to propose an open and flexible ISA in respect Intel or Arm ISAs that, for example, are not open; «V» indicates the fifth generation of the project.
The ISA includes some faculties, through extensions, to provide more functionality: floating point(F), multiplication and division(M), atomic(A), compressed(C), or other manipulation. All this makes the ISA generic in nature, giving the possibility of customising its CPU according to the needs of the application.
In general following this approach can be observed three different features that makes this kind of implementation different:

- Open Source: RISC-V is an open-source architecture, which means that anyone can use, modify and implement the specification without paying royalties.
- Modularity: RISC-V is modularly designed, allowing developers to select specific functionalities
- Standardisation: RISC-V is supported by a foundation that promotes standardisation between implementations.

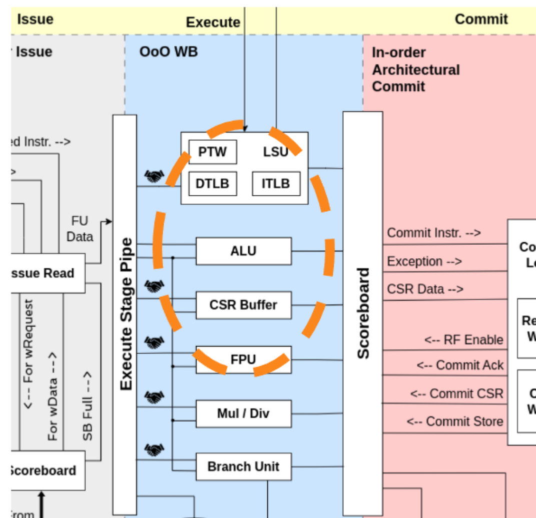


Figure 2.1: Placement countermeasure in CVA6 pipeline
Source: image taken from the CVA6 GitHub [12]

Chapter 3

Homomorphic security tags and permutation

The alternative domain plays a fundamental role in the countermeasure, as the main challenge is developing networks capable of delivering authentic results, where here the concept of authenticity refers to maintaining a direct connection to the original data. Even more crucial is ensuring excellent performance at the processor level while leaving for a potential verification stage.
This approach forms the basis for the countermeasure developed, through the definition of homomorphism.

3.1 Definition of the countermeasure

A homomorphism is

> "a map between two algebraic structures of the same type that preserves the operations of the structures"
>
> *Wikipedia, 2020, [13]*

In other words given an operation and two operands u and v we assert that ϕ is a homomorphism if the following equality holds:

$$\phi(u * v) = \phi(u) \cdot \phi(v) \tag{3.1}$$

This means that calculating the operation «$*$» on u and v and applying the function ϕ should generate the same result as applying ϕ on the two operands and only after performing the operation «$\cdot$».

Now if the intent is to create an alternative domain, while maintaining a connection through a verification stage, can be observed well how the transformation sought is convenient to be homomorphic. In fact, the complexity would come in looking, for each logical or arithmetic operation, the respective permuted version that would enforce the equality of the Formula 3.1.
For us, equality means that performing the operation «$*$» on u and v, and then converting to the new domain, must generate the same result as the operation «$\cdot$» on the authenticity tags. Consequently, considering that we are working with binary numbers, the choice of the function ϕ fell on the transposition function, i.e

> "the modification of the position of certain elements within an ordered object"
>
> *Treccani, 2020, [14]*

The function constituting the countermeasure, however, is not just about transposition. In fact, it is necessary a function over which it is possible to have decision power in reversing the ordering.
In this way can be constructed the «permutation» function, an association between a transposition and a key with the power to decide whether to make an inversion rather than an identity. Permute will consist of dividing an input data into blocks of bits and rotate them according to the associated control signal.
In searching for an elementary block capable of emulating the behavior just described, it is found a cell called «Fredkin gate», as in Figure 3.1, containing three inputs: two input data *X1* and *X2* and a selection signal *Sel*. According to its truth Table 3.1, swap is performed when the control signal is worth 1 and vice versa identity when the control signal is worth 0.

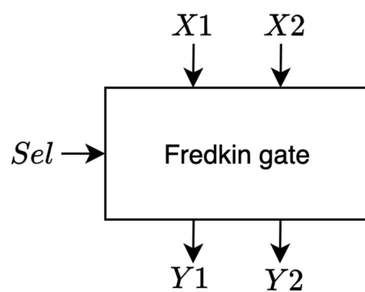


Figure 3.1: Fredkin gate block

Sel	$X1$	$X2$	$Y1$	$Y2$
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	0
1	1	0	0	1
1	1	1	1	1

Table 3.1: Fredkin gate truth table

Using this cell, as an elementary module, on a multilevel structure we are able to define the permutation.

3.2 Permutation

The permutation function is a dichotomous tree in which each node consists of a Fredkin gate. The idea of the tree structure is shown in Figure 3.2 in which can be observed the organization on multiple levels for a 32-bit data and how each block on each section is on the power of 2, consecutive and not overlapping.
The application of permutation begins dividing the input into two halves, with the Most Significant Bit(MSB) of the key serving as the control signal. Then, by progressing to the least significant bits, the same principle of division into two halves is applied to each sub-block, until the lowest level of the tree is reached, consisting of single-bit section.
It means that if it is taken a 64-bit data the first division would be in two 32-bit blocks, the second division would be in four 16-bit blocks and so on up to sixty-four 1-bit blocks. In the transition from one stage to the next, an output signal is generated at the initial level of parallelism, which becomes the input to the subsequent stage. This process occurs over a depth of $\log_2(n)$, where n represents the width of the input data.

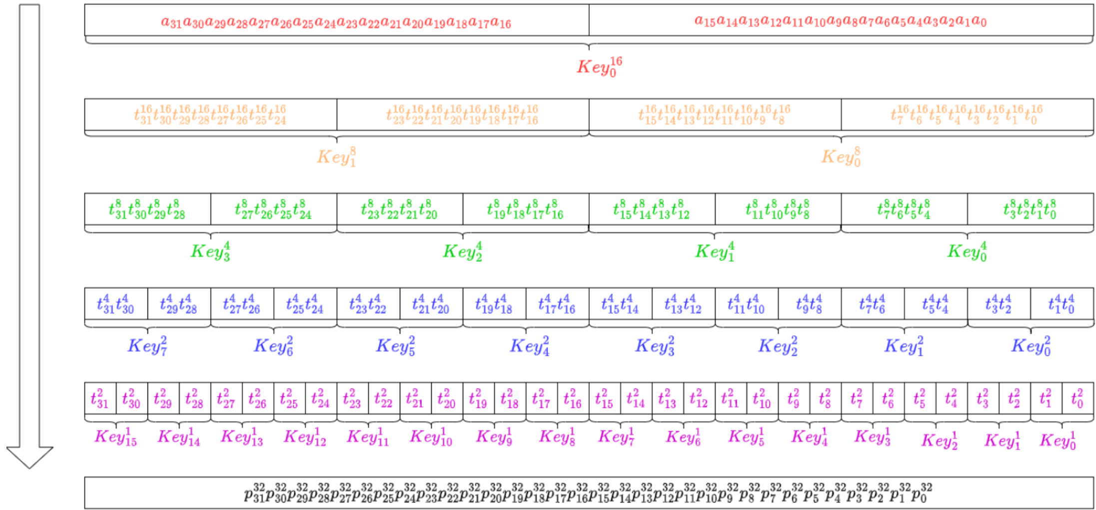


Figure 3.2: Dichotomous tree for 32-bit permutation function
Source: image taken from the [5] article

Constructing the permutation, it is used the notation Key_j^i to refer to a single bit of the key: i represents the size of the subsection on which to make the decision; j represents the position index of the key dedicated to that specific size. For example, the $Key^2_{[7:0]}$ notation represent the eight bits related to swap control for 2-bit groups.
To generalize now for a generic width of the data is fundamental to understand how

can be determined the size of the permutation key. Starting from the last Figure 3.2 and applying the function from the MSB toward the Least Significant Bit(LSB), can be seen how for 16-bit groups only one bit is needed, for 8-bit groups two bits are needed until sixteen bits for 1-bit groups. It means that key is characterized by 31 bits of width in correspondence to a 32-bit data.
In this way, the iterative application and the use of power-of-2 blocks allow to conclude that the dimension must be *n - 1* bits for data of width *n*. This property is illustrated in the example of Figure 3.3, which shows an 8-bit data and its corresponding 7-bit key with the notation $k = k_0^4 k_1^2 k_0^2 k_3^1 k_2^1 k_1^1 k_0^1$.

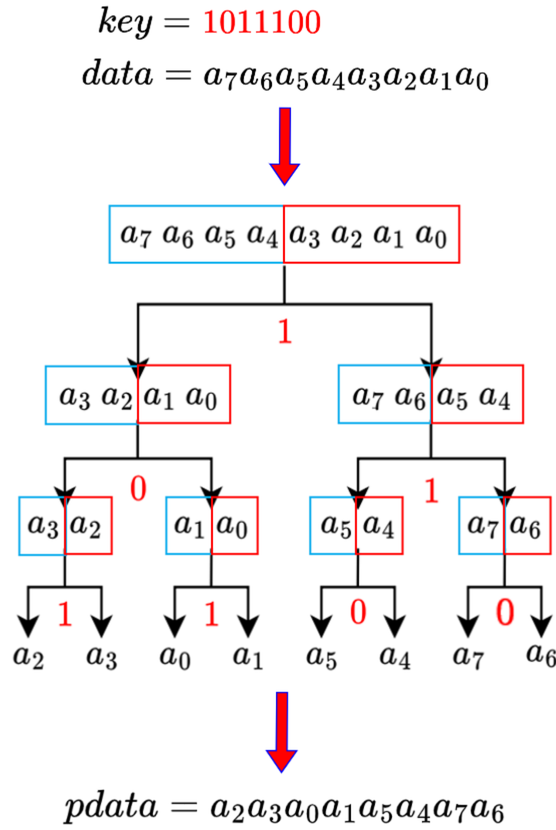


Figure 3.3: Permuted dichotomous tree example for 8-bit operand

Here according to the theory presented, an ascending key association and dichotomous tree with depth $\log_2(n = 8) = 3$ are observed, where the distinction into blue and red blocks facilitates the idea of division for each sub-section into two halves, on which to take a swap or identity decision.

3.3 Depermutation

Complementary to the permutation function it is needed the inverse operation to bring back the native domain, presumably in a symmetric way with respect to the previous one.
The new function in this case is called «depermutation» and creates the opposite association of the same key, from the LSB toward the MSB, generating an inverted alternation of swap and identity. The structure is still a dichotomous tree with depth $\log_2(n = 8) = 3$ just flipped, as Figure 3.4 reports.

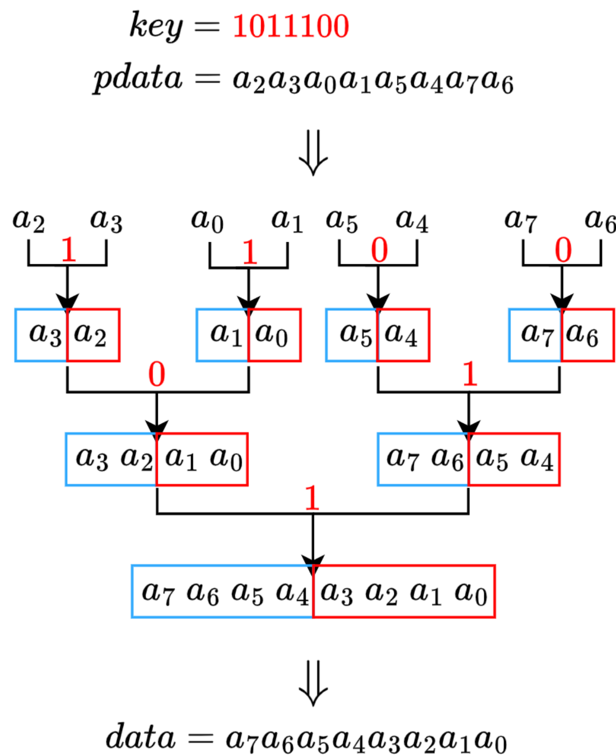


Figure 3.4: Depermuted dichotomous tree example for 8-bit operand

Here, making a comparison with the permutation Figure 3.3, can be easily observed the similarity with a symmetric application of the function where the distinction into blue and red blocks, once again, facilitates the idea of division for each sub-section into two halves, on which to take a swap or identity decision.

3.4 Effectiveness countermeasure

The proposed theory explains a technique capable of changing the representation domain, by being able to easily recreate native data when needed, where the homomorphic choice is the main aspect, because allows the insertion of a comparison stage to detect a possible attack.
However, it still did not answer the most important question:

Why would the proposed countermeasure work?

The fault injection technique uses simultaneous laser beams that target single or multiple data bits to compromise their integrity and extract sensitive information. Our aim is to create redundant hardware that computes arithmetic and logic operations in a permuted domain associated with a specific key, where the permuted execution ensures that no faults have been injected, into the processed data, by preventing attackers from consistently targeting the same bits, as the permutation key is randomly changed.
The last aspect of randomness corresponds to the crucial point because it does not allow predictions about the placement of the new weights and thus not predict the operations. In fact, as the permutation example in Figure 3.3 shows, the new representation will be directly dependent on each individual bit of the key: a change of a single bit could result in large variations in the representation.
It is still necessary that each network, developed in accordance with the technique, makes no reference to the native representation neither uses a static arrangement of weights. This means to have an internal propagation of bits that differs with each new key generated; in this way, the circuit will also possess their own intrinsic and completely random computation and will guarantee not predictable execution.
So far then, the new domain transformation technique has been presented; it must now concern with understanding how the different recurring operations can be handled.
If, for example, we were to refer to the sum operation, a random sorting of the new permuted data would not allow us to solve the calculation in the classical sense, but would require the search for dedicated hardware. Similarly, the same may apply to those simpler operations such as extraction and set that are normally done through indexing.

Chapter 4

Permuted ALU

The ALU is a fundamental component within a Central Processing Unit(CPU), responsible for performing arithmetic and logical operations. It represents the heart of the computational process, as it handles basic mathematical operations such as addition and subtraction and logical operations such as shift, AND, OR, NOT.
Each operation performed by the ALU is supervised by a Control Unit(CU), that decides which operation is to be performed according to the instructions, and which data to select from the many registers present.
The countermeasure discussed above is applied here, where the task is to adapt the elementary components to create a new redundant path, thereby forming a «permuted ALU». By recalling that CVA6 is the target core, can be observed that the ALU can be divided into four sections:

- Logic circuits: performs logic operations
- Addition: performs the sums and count operations
- Shift: performs shift operations
- Comparison: compare two operands

By adapting these four sections and their components, we are able to develop all 63 planned operations as listed in Table 4.1. To achieve this, two elementary operations of single-bit extraction and set are analysed first, and only after the Boolean operations, an adder, a shifter and finally some networks for count operations. The latter, despite containing the adder, are characterised by dedicated structures in order to handle permutation.

Operation	Description	Operation	Description
ANDL	AND	CPOP	Count set bits
ANDN	AND with inverted operand	CPOPW	Count set bits in word
ORL	OR	CLZ	Count leading zero bits
ORN	OR with inverted operand	CLZW	Count leading zero bits in word
XORL	Exclusive OR	CTZ	Count trailing zeros
XNOR	Exclusive NOR	CTZW	Count trailing zero bits in word
ADD	Add	MAX	Maximum
ADDW	Add word	MAXU	Unsigned maximum
ADDUW	Add unsigned word	MIN	Minimum
SUB	Subtract	MINU	Unsigned minimum
SUBW	Subtruct Word	SLTS	Set if Less Than, Signed
SLL	Shift Left Logical	SLTU	Set if Less Than, Unsigned
SRL	Shift Right Logical	BCLR	Single-Bit Clear
SRA	Shift Right Arithmetic	BCLRI	Single-Bit Clear (Immediate)
SLLW	Shift Left Logical Word	BEXT	Single-Bit Extract
SRLW	Shift Right Logical Word	BEXTI	Single-Bit Extract (Immediate)
SRAW	Shift Right Arithmetic Word	BINV	Single-Bit Invert
SH1ADD	Shift left by 1 and add	BINVI	Single-Bit Invert (Immediate)
SH2ADD	Shift left by 2 and add	BSET	Single-Bit Set
SH3ADD	Shift left by 3 and add	BSETI	Single-Bit Set (Immediate)
SH1ADDUW	Shift unsigned word left by 1 and add	SEXTB	Sign extend byte, 16-bit encoding
SH2ADDUW	Shift unsigned word left by 2 and add	SEXTH	Sign extend halfword, 16-bit encoding
SH3ADDUW	Shift unsigned word left by 3 and add	ZEXTH	Zero extend halfword, 16-bit encoding
ROL	Rotate Left (Register)	REV8	Byte-reverse register
ROR	Rotate Right (Register)	ORCB	Bitwise OR-Combine, byte granule
RORI	Rotate Right (Immediate)	SLLIUW	Shift-left unsigned word (Immediate)
ROLW	Rotate Left Word (Register)	CZERO_EQZ	Moves zero to a register *rd*, if the condition *rs2* is equal to zero
RORW	Rotate Right Word (Register)	CZERO_NEZ	Moves zero to a register *rd*, if the condition *rs2* is nonzero
RORIW	Rotate Right Word by Immediate	NE	Branch if not equal
EQ	Branch if equal	GES	Branch if greater than or equal signed
LTS	Branch if less than signed	GEU	Branch if greater than or equal unsigned
LTU	Branch if less than unsigned		

Table 4.1: ALU operations supported by the CVA6 core
Source: definitions from the [15] book and the [16] manual

4.1 Elementary operations

Elementary operations are understood to be extractions and sets of individual bits usually provided for specific operations. An extraction for example is fundamental in the case of arithmetic shift, for which it is essential to observe and replicate the sign bit; a set for example is fundamental for creating masks or activating status flags.

4.1.1 Extraction single bit

The extraction of a bit is normally achieved by a selection of the specific bit and so, taking the example of the sign extension, by simple indexing to the left most position.
In the case of a permuted operand, remembering that we are dealing with data whose weights are out of phase with the native representation, this is no longer possible. In general, for any given position, it is required to find a dedicated structure that brings us back to the bit of the desired weight by comparing the requested position with the corresponding bit of the key. A shape to fulfil this behavior corresponds to the elementary block shown in Figure 4.1.

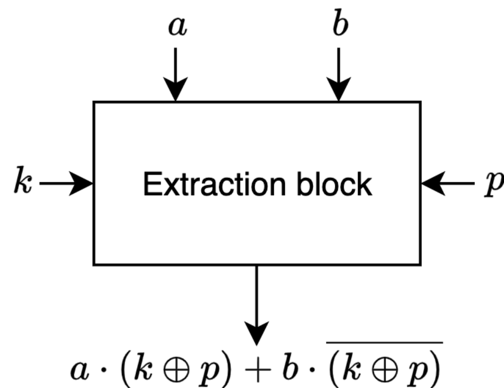


Figure 4.1: Extraction block
Source: image taken from the [6] article

The cell receives as input two data bits (a and b), a key bit (k) and a position bit (p). If k and p are different it propagates the input a, if k and p are equal it propagates the input b.
Obviously the block solves for only one decision bit; it is necessary to develop a generic network on n bits resulting from the connection of several levels as shown in Figure 4.2 for an 8-bit case and a key with the notation $k = k_0^4 k_1^2 k_0^2 k_3^1 k_2^1 k_1^1 k_0^1$.
As can be seen, the architecture has a similar shape to the one of the depermutation

in Figure 3.4, i.e. a descending dichotomous tree. This occurs when it is attempted to apply the key during the reconstruction phase, assigning it from the least significant index to the most significant one.
Also in this case the depth of the tree it is equal to $\log_2(n = 8)$=3 corresponding to the width of the position signal($p_2p_1p_0$). Each level it is thus linked to each bit of this signal and it allows to make a comparison with the corresponding key section.

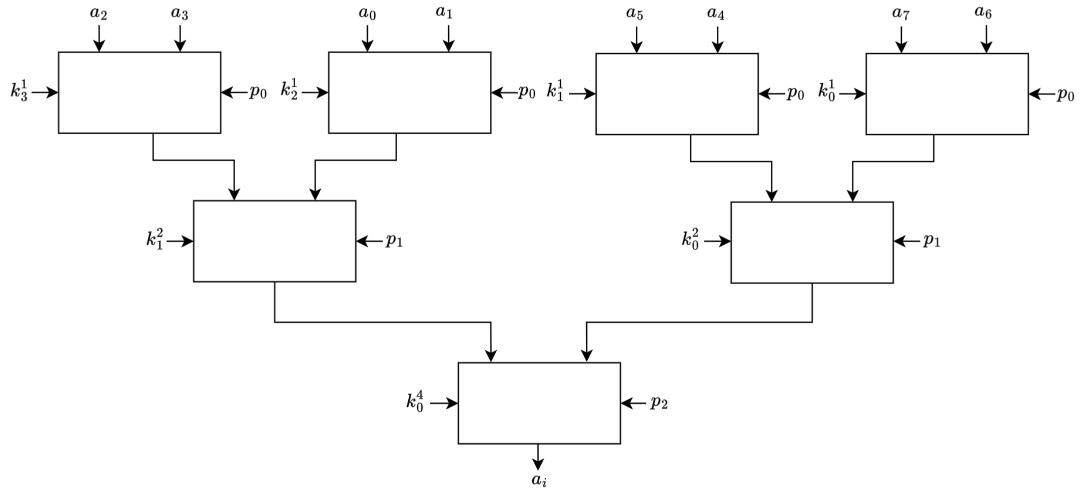


Figure 4.2: Extraction dichotomous tree example for 8-bit permuted operand
Source: image taken from the [6] article

4.1.2 Set single bit

A bit set is normally achieved by indexing the required specific bit and by the application of a logical function rather than a direct assignment. It means that the access is usually realised by indexing the desired weight followed by a specific operation.
In the case of a permuted operand, as with extraction, remembering that we are dealing with data whose weights are out of phase with the native representation, this is no longer possible. In general, for any given position, it is required to find a dedicated structure that brings us back to the bit of the desired weight, by comparing the requested position with the corresponding bit of the key.
A shape to satisfy this behavior corresponds to the elementary block shown in Figure 4.3.

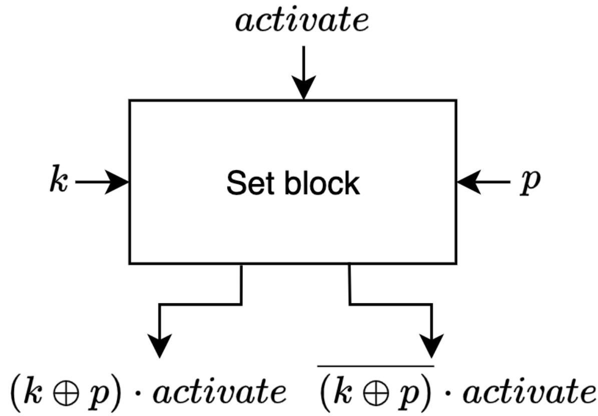


Figure 4.3: Set block
Source: image taken from the [6] article

The cell receives as input an activation bit (*activate*), a key bit (k) and a position bit (p). If k and p are different it propagates the input *activate* to the left, if k and p are the same it propagates the input *activate* to the right.
We thus observe that the *activate* input controls the activation of the entire network because if it were equal to 0 the network would never encounter a stimulus that results in an all-zero output.
Obviously, the block solves for only one decision bit; it is necessary to develop a generic network on n bits resulting from the linking of several levels as shown in Figure 4.4 for an 8-bit case and a key with the notation $k = k_0^4 k_1^2 k_0^2 k_3^1 k_2^1 k_1^1 k_0^1$.

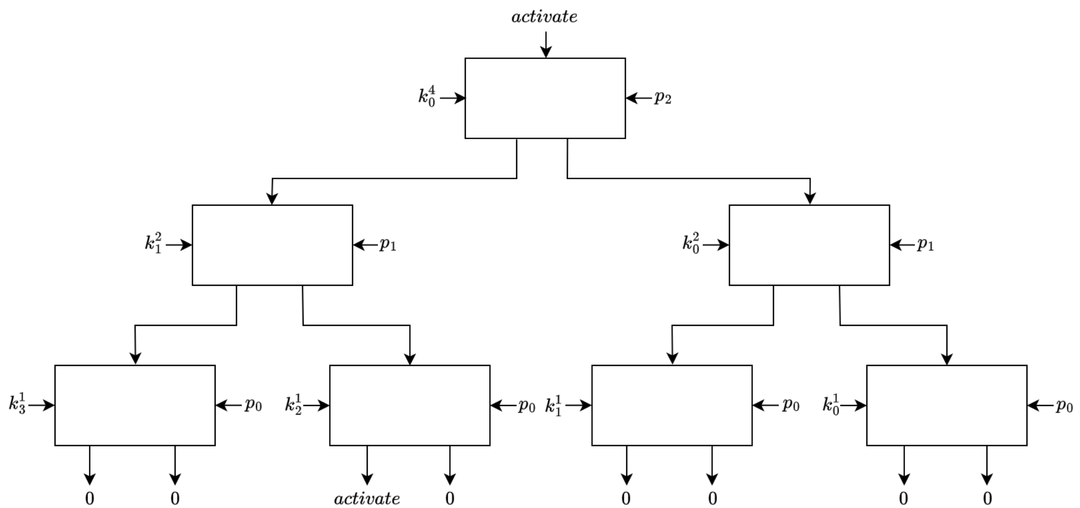


Figure 4.4: Set dichotomous tree example for 8-bit permuted operand
Source: image taken from the [6] article

As can be seen, the architecture has a similar shape to that of the permutation in Figure 3.3, i.e. an ascending dichotomous tree in a complementary way in respect the previous one of extraction.
This happens because, if the extraction involves starting from a data vector and arriving at a 1-bit output, the set performs the reverse operation by propagating a 1-bit input to the desired position accordingly to the permutation. There is in a such way an assignment of the key from the most significant index to the least significant one.
The output of this architecture it is a signal with the same parallelism as the data, where the idea is to render it as a mask. Indeed the bit with the logical value 1 it is aligned to the corresponding weight, allowing to apply some function in the new permuted domain.
Also in this case the depth of the tree it is equal to $\log_2(n = 8)$=3 corresponding to the width of the position signal($p_2p_1p_0$). Each level is thus linked to each bit of this signal and allows to make a comparison with the corresponding key section.

4.2 Boolean operations

The first class of operations to observe are the logical functions: ANDL, ANDN, ORL, ORN, XORL, XNOR.
In this case, since these are bit-wise operations, the concept of permutation becomes essential, inducing the need to introduce a simple permutation stage. In fact, the two operands involved, apart from the permutation, possess a weight alignment that allows the direct application of the logic function.
Figure 4.5 shows the example of the logical AND function, where can be seen how a dedicated network is not necessary.

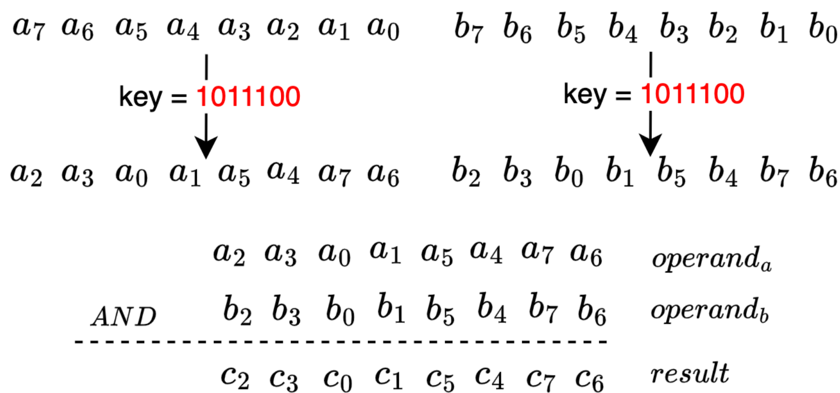


Figure 4.5: Permuted AND logic function example for 8-bit operand

4.3 Addition operations

The second elementary computation is the addition. Through an adder in fact we are able to manage various arithmetic operations: ADD, ADDW, ADDUW, SH1ADD, SH2ADD, SH3ADD, SH1ADDUW, SH2ADDUW, SH3ADDUW.
As for elementary and Boolean operations discussed in the previous sections, it is needed to evaluate whether to develop a basic architecture capable of support the new domain of the countermeasure. One initial idea could be the use of the classical adders: the Ripple-Carry Adder(RCA).
The architecture begins by analysing the binary addition generally described as a 2-input block that outputs the *sum* bit and the *carry* bit. However, generalizing over a multiple number of bits leads to connect the carry of the previous block as the input of the next one, resulting in a new 3-input cell called the «*Full-Adder*»(FA).
About this latter, given two data inputs a_i and b_i and a third input c_{in} of carry, can be derived the respective 3-input truth table and the output logical functions as shown in Figure 4.6.

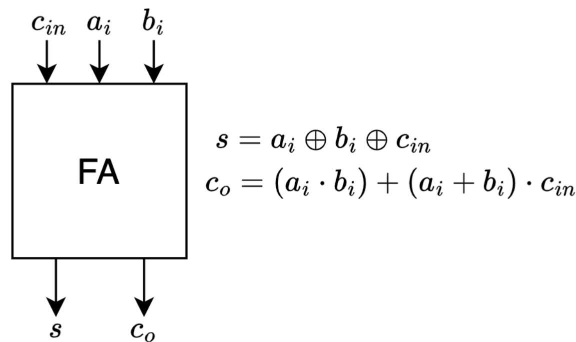


Figure 4.6: Full-Adder block with sum(s) and carry-out(c_o) generation

The architecture of an RCA consists of a chain of FAs and represents the simplest type that can be realized. However, the carry signal propagation between each block highlights the weakest point of this structure: the linear connection of the architecture, that first of all creates a significant critical path for high parallelism, complicates adaptation to the permuted domain.
The necessary propagation shown in Figure 4.7, due to the rearrangement introduced by the permutation function, is difficult to adapt from the first input carry as it is not known which block to attach it to.
Requiring too complex networks we are forced to shift our attention to different architectures. Trying to find a better solution among those in the mathematical literature, in the Carry-Lookahead Adder(CLA) configuration several affinities with the permutation are met.

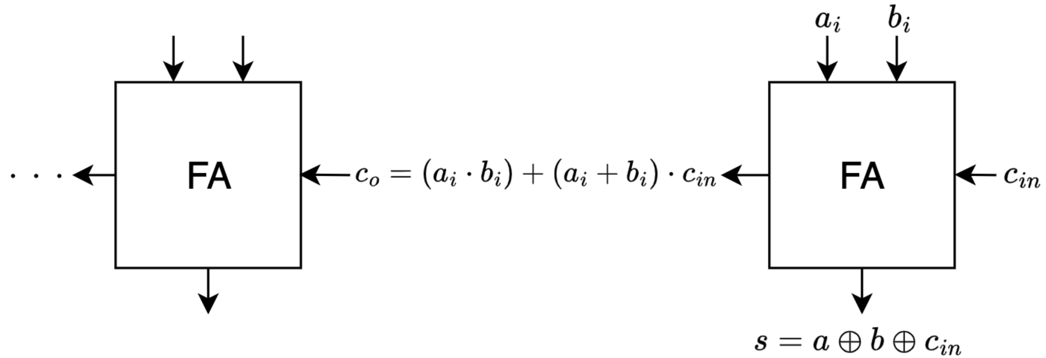


Figure 4.7: Propagation problem for Ripple-Carry Adder connection typology

The starting point for this design is the generation of the signals *propagate*(p) and *generate*(g) through which the carry-out can be re-expressed as their function. These are internal signals from inputs generated that given two bits of the same operand weight, such as a_i and b_i, correspond to their XOR and AND respectively. Accordingly their definitions are generated through an initial layer of FAs, that have been redefined without a carry-out, as shown in Figure 4.8.

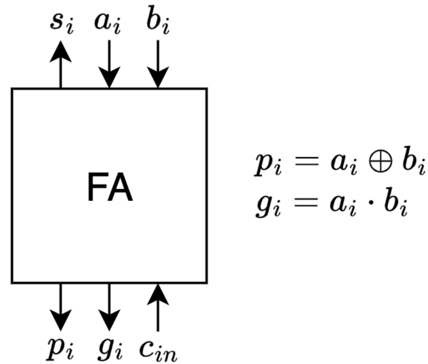


Figure 4.8: Full-Adder block with generate(g) and propagate(p) generation

To derive the architecture the crucial step lies in observing that the carry-out(index i) of the adder can be re-expressed as a function of the last *propagate* and *generate* at the index i-1:

$$\begin{matrix} g_i = a_i \cdot b_i \\ \\ p_i = a_i \oplus b_i \end{matrix} \quad \Rightarrow \quad c_i = a_{i-1} \cdot b_{i-1} + a_{i-1} \cdot c_{i-1} + b_{i-1} \cdot c_{i-1} = g_{i-1} + p_{i-1} \cdot c_{i-1}$$

Continuing, now the carry recurrence at the index i-1 can be unrolled founding directly a function of the input carry and of p and g terms derived from input operands as developed in Figure 4.9.
For simplicity of the propagation concept, we will now refer, in the derived tree structure, to left and right signals as shown in the following circuit solutions.

$$
\begin{aligned}
c_i &= g_{i-1} + c_{i-1}\, p_{i-1} \\
&= g_{i-1} + (g_{i-2} + c_{i-2}\, p_{i-2})\, p_{i-1} \\
&= g_{i-1} + g_{i-2}\, p_{i-1} + c_{i-2}\, p_{i-2}\, p_{i-1} \\
&= g_{i-1} + g_{i-2}\, p_{i-1} + g_{i-3}\, p_{i-2}\, p_{i-1} + c_{i-3}\, p_{i-3}\, p_{i-2}\, p_{i-1} \\
&= g_{i-1} + g_{i-2}\, p_{i-1} + g_{i-3}\, p_{i-2}\, p_{i-1} + g_{i-4}\, p_{i-3}\, p_{i-2}\, p_{i-1} + c_{i-4}\, p_{i-4}\, p_{i-3}\, p_{i-2}\, p_{i-1} \\
&= \ldots
\end{aligned}
$$

Figure 4.9: Unrolling carry recurrence
Source: image taken from the [17] article

The latter steps are the most important because if the substitution appears to be very complex, the different g and p signals could be combined with what is called «*propagate/generate block*»(PGB), reported in Figure 4.10.
It is a combination structure that receives a carry for a specific position, along with two pairs generated and propagated from the left and right. It manages their propagation through output signals, which are a combination of them to be used as interconnections for further PGBs, along with two output carries.
The output *propagate term* is set to 1 if both those of the input pair are worth 1; the *generate term* is set to 1 if the g from the left is worth 1 or if the p from the left and the g from the right are both worth 1.

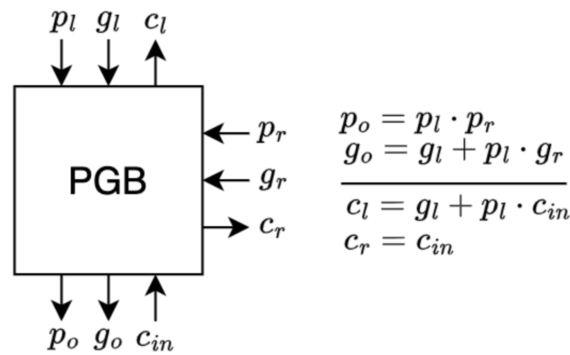


Figure 4.10: Propagate/generate block

With these observations, these last two blocks of Figures 4.8 and 4.10 give the possibility of depicting the final structure of the CLA, as the Figure 4.11 with an 8-bit case, where the several affinities mentioned come to the eye when observing a permutation-like configuration.
It is in fact the same dichotomous tree, with a depth equal to $\log_2(n = 8)$=3, starting from the bottom where the carry c_0 corresponds to c_{in}.

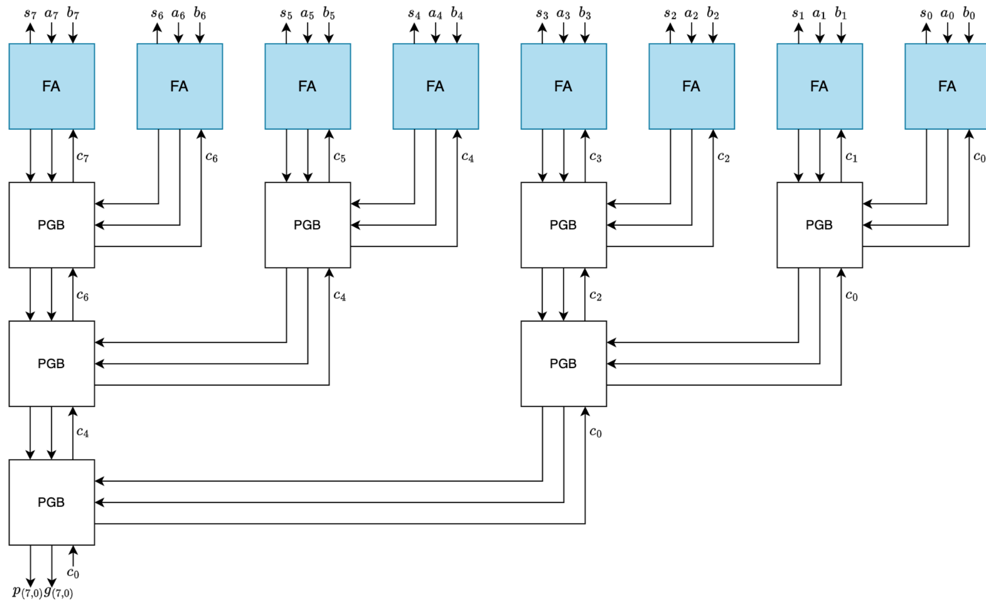


Figure 4.11: Dichotomous tree example for 8-bit CLA
Source: image derived from the [6] article

The adder works through the operands that enter at the top to calculate g and p signals, they flow to the bottom to calculate the intermediate carries from the input one and then data flow back up to calculate the sum.
This kind of propagation optimizes the design for the permutation by demonstrating that a carry does not need to be used simultaneously with its generation. In the final step of back-propagation using FAs, all carries arrive in the same moment, allowing the sum to be computed.
Starting from the input carry, the adaptation object becomes the propagation to the right weight of the internal carries by inserting the key as a decisional signal in the transmission directions.
In this way from the PBG can be derived the «*permuted propagate/generate block*»(PPGB), as shown in Figure 4.12, where the outputs correspond to a mix between the left and right ones depending on the key that equal to 0 would return to the native domain.

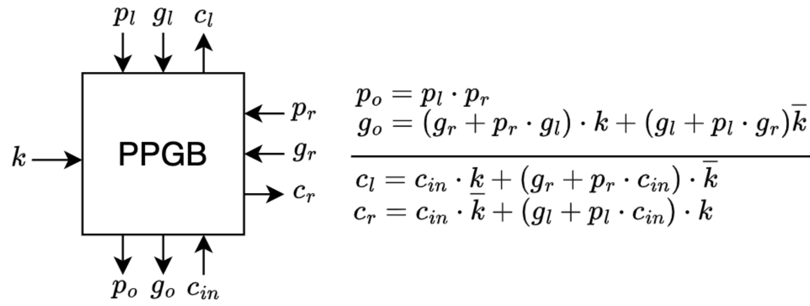


Figure 4.12: Permuted propagate/generate block

This last block is the only point required to find the final permuted architecture, analogous to the previous Figure 4.11, where each PPGB, with an input key bit, allows to correctly handle an operand with a random weight arrangement.
The final configuration of the adder is thus represented in Figure 4.13, once again for the 8-bit case and a key notation k = $k_0^4 k_1^2 k_0^2 k_3^1 k_2^1 k_1^1 k_0^1$, with a visible correspondence of the weights of the input bits with that of each internal carry resulting from the back-propagation.

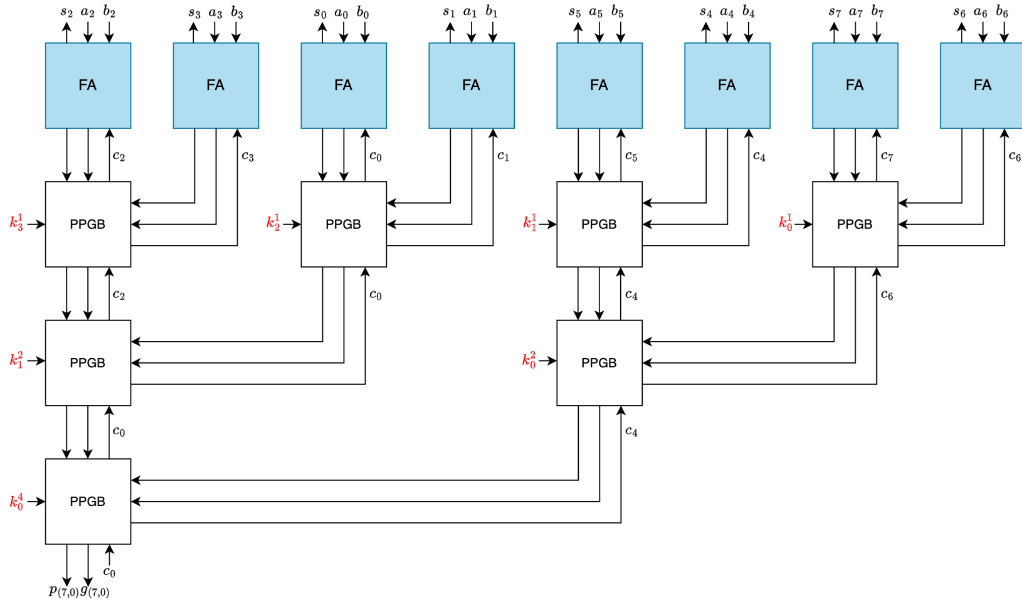


Figure 4.13: Dichotomous tree example for 8-bit permuted CLA
Source: image derived from the [6] article

4.3.1 Subtracter

Algebraic operations do not end with addition but also include subtraction operations such as SUB and SUBW.
While in a digital circuit addition is relatively simple to implement, due to the first configuration of the RCA, subtraction is more complicated, mainly due to the calculation of credits between the various bits.
This is why it is generally exploited the property whereby subtracting a number is equivalent to adding its complement with the entry of the 2's complement(CA2), a binary numerical representation for negative numbers. It is obtained by inverting all the bits of the operand and then adding 1, but the crucial aspect lies in the observation that if a number is added to its CA2, the result it is 0.
This is a fundamental property because it means that subtracting a number b from a number a is equivalent to add the CA2 of b to a providing a method to transform a subtracter into an adder through the same logical units, which can potentially be shared as a single component.
The steps to follow become two:

- reverse the bits of b
- add 1 to the negated result

Thus, using any summing machine as a basis, it is possible to integrate these two steps through a logical NOT($\circ$ its notation), before the connection of the operand b, and an input carry of 1.
The final extension required is the adaptation for permuted operands, as shown in Figure 4.14, where no further modifications are needed. In fact, the same changes applied in the native domain can be computed with the permuted adder, given that the logical operations do not require dedicated structures, and the +1 can be seamlessly integrated as input carry.

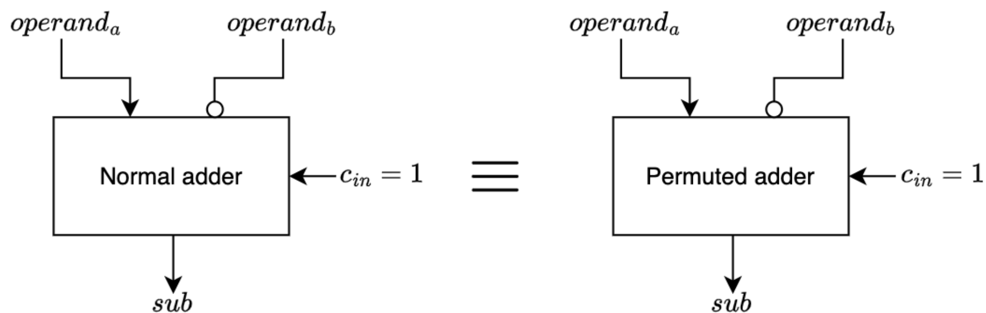


Figure 4.14: Subtracter adaptation for native and permuted domain
Source: image derived from the [6] article

4.3.2 Comparator

The architecture of an ALU often has instructions for comparing operands in order to implement set and branch instructions. This means, for example, taking decisions after checking whether the first operand a is greater than, less than or equal to the second operand b.
A unit of this type in our case would allow to cover several operations: MAX, MAXU, MIN, MINU, LTS, LTU, SLTS, SLTU, GES, GEU, EQ, EN.
In this way the goal becomes to build the unit where, fortunately, the previously defined logic can be reused. In particular, it is only needed the subtracter configuration with three additional output signals on which to take a decision:

- zero bit
- less unsigned bit
- less signed bit

Sometimes in the case of the CVA6 the result may not be the primary interest. For operations such as EQ, LTS... it is directly the single bit of *zero* or *less* that concerns us, which is why the CVA6 ALU has a dual output: the «result» of 64-bit and the «branch» of 1-bit.
An equality comparator, and therefore the unit for controlling the *zero* bit signal, can be easily computed using a XNOR function between the two operands that, as a logical function, does not have problems of adaptation. In this way, computing another AND with all the bits of the result the *zero* bit it is set to 0, indicating different inputs, otherwise, the *zero* bit it is set to 1, meaning the operands are equal and the condition is satisfied.
A relational comparator, and therefore the unit for controlling the *less* bit signal, must necessary be a subtracter. From here, having already dealt with its adaptation in the permuted domain, can be taken the previous configuration of the adder and insert some considerations from the obtained result.
Starting with the unsigned case, four examples are shown in Figure 4.15, allowing us to observe certain properties, particularly highlighting the values of the zero and carry bits.

$$
\begin{array}{lllll}
A = 5\,,\ B = 2 & \rightarrow & A > B & \rightarrow & c_{out} = 1\,,\ z = 0 \\
A = 1\,,\ B = 3 & \rightarrow & A < B & \rightarrow & c_{out} = 0\,,\ z = 0 \\
A = 6\,,\ B = 5 & \rightarrow & A \geq B & \rightarrow & c_{out} = 1\,,\ z = 0 \\
A = 6\,,\ B = 6 & \rightarrow & A = B & \rightarrow & c_{out} = 0\,,\ z = 1
\end{array}
$$

Figure 4.15: Comparison examples for $>$, $<$, $\geq$, $=$

The first example in the case of A = 5 and B = 2 shows how the difference of the two operands A-B = 3 returns carry bits equal to 1 and zero bits equal to 0. The second example in the case of A = 1 and B = 3 shows how the difference of the two operands A-B = -2 returns carry bit equal to 0 and zero bit equal to 0.
The third example similarly with A-B = 0 returns carry equal to 0 and zero bit equal to 1 and the fourth example with A-B = 1 returns 0 and 1 respectively.
Each of these examples illustrates that a decision can be made directly as a function of the carry and zero bits, accordingly with the request, and this is the main result because trying to generalise with any comparison request we are able to construct a unique logical decision function, to fix the *less* signal, deriving the Table 4.2.

Case	Logical function
$A > B$	$\text{carry} \cdot \overline{z}$
$A \geq B$	carry
$A < B$	$\overline{carry}$
$A \leq B$	$\overline{carry} + \text{z}$
$A = B$	z
$A \neq B$	$\overline{z}$

Table 4.2: Comparison cases as function of zero bit(z) and carry-out bit($carry$)

It was not the only case because in order to complete the unit of comparison there is the need to deal with the signed numbers that at the moment are uncovered. However luckily, in this case another additional dedicated structure is not necessary and a few expedients are sufficient. Precisely there are two possible scenarios:

- when the signs of the operands are the same, the previous Table 4.2 is still valid and the same subtracter configuration can be used.
- when the signs of the operands are different it is even easier to determine the comparison. In fact the subtracter is not longer necessary because can be compared directly the signs of the inputs. For example, if the operands have different signs, it could be possible to conclude that they are different or that the operand with sign equal to 0, i.e. a positive integer, is the greater.

To conclude for this last signed case, several shortcuts can be implemented using simple 2-bit logic gates, thereby conserving resources.
But as a final point, it is important to note that when interacting with permuted operands, if no further adjustments are needed for the unsigned case, as the permuted subtracter configuration is already complete, a final adaptation is required for the signed case: due to the random order of the bits, it is not possible to directly access the MSB and we are obliged to insert the dedicated network of extraction from Section 4.1.1.

4.4 Shift operations

The third fundamental computation is the shift. Through a shifter in fact we are able to manage various logic operations: SLL, SRL, SRA, SLLW, SRLW, SRAW, SH1ADD, SH2ADD, SH3ADD, SH1ADDUW, SH2ADDUW, SH3ADDUW, ROL, ROR, RORI, ROLW, RORW, RORIW, REV8, SLLIUW.
The manipulation refers to a bit-wise logical function that involves moving the bits of a number either to the right or to the left by a specific amount. The vacant positions created during this operation are filled with zeros, ones, or bits from the same operand, depending on the operation required.
Now to fulfil all these possible instructions and arrive at a permuted Barrel Shifter architecture, i.e. a shifter conformed to any amount and mode, it is necessary to go through four steps:

1. Shift left by 1
2. Shift left by 2
3. Shift left by a power of 2
4. Barrel Shifter

Of the various operations, we rely on the logical shift to the left as the most classic model and from which the others can be derived by means of some adaptation.

4.4.1 Shift left by 1

The shift left by 1 is the simplest case of all, being of one position only, that finds its starting point in a block called « *White block*»(WB) shown in Figure 4.16. The component has 3 data inputs a,b,c and a control input k, correspondent with the permutation key, that has the ability to recreate a rotation in its function.

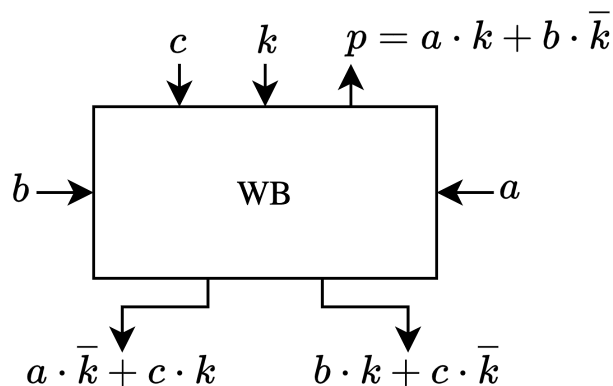


Figure 4.16: White block

The final structure, that can be derived, is a multi-level tree that provides the mean of rotation in a propagation of the input signals towards an upper level or a lower level. First of all a decision is made between the inputs a and b, on which to propagate upwards, via the signal *propagate*(p) and the permutation key(k): if k = 0 the input a is brought out on the left side, together with c on the right side and b in the propagate position; if k = 1 the input b is brought out on the right side, together with c on the left side and a in the propagate position.
The previously mentioned rotation concept became visible trying to construct the tree architecture where can be observed the third input c coming from an upper level, as shown in Figure 4.17 for a 4-bit case and the permuted data $a_0a_1a_3a_2$.

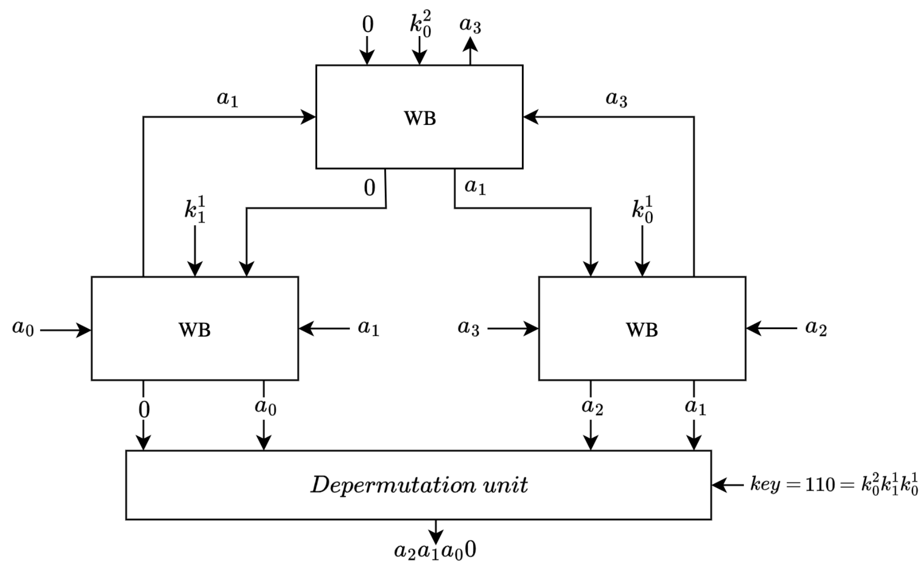


Figure 4.17: Permuted shift left by 1 example with White Block(WB)
Source: image derived from the [6] article

The final architecture is dichotomous as for permutation, with a key connection to each WB until the LSB, and a number of levels equal to $\log_2\left(\frac{n}{block}\right)$ where n represents the width and *block* the shift amount.
It is worth noting in the example the ejection of the bit corresponding to position a_3, in correspondence of the last *propagate* signal in the top position, and how in the same top block, since there are no other upper levels, the third input c is equal to 0, i.e. the point at which the vacant positions are fulfilled.
To conclude if it is observed in detail at the output level of the tree, can be observed such a structure is capable of placing the new zero-weighted bit in its permuted position, generating a signal equal to $a_2a_1a_00$ and thus the value we would expect with input $a_3a_2a_1a_0$.

4.4.2 Shift left by 2

The shift left by 2 is the second easiest case to handle, being of two positions only. This step is fundamental because trying to apply the same WB structure, it reveals that this block is no longer sufficient. In fact, the 4-bit example in Figure 4.18 shows how, with a structure that would now be only one level remembering the formula $\log_2(\frac{n}{block})$, it would arrive at an output equal to $a_0a_1 00$ when for an input of $a_3a_2a_1a_0$ it would be expected to get $a_1a_0 00$.

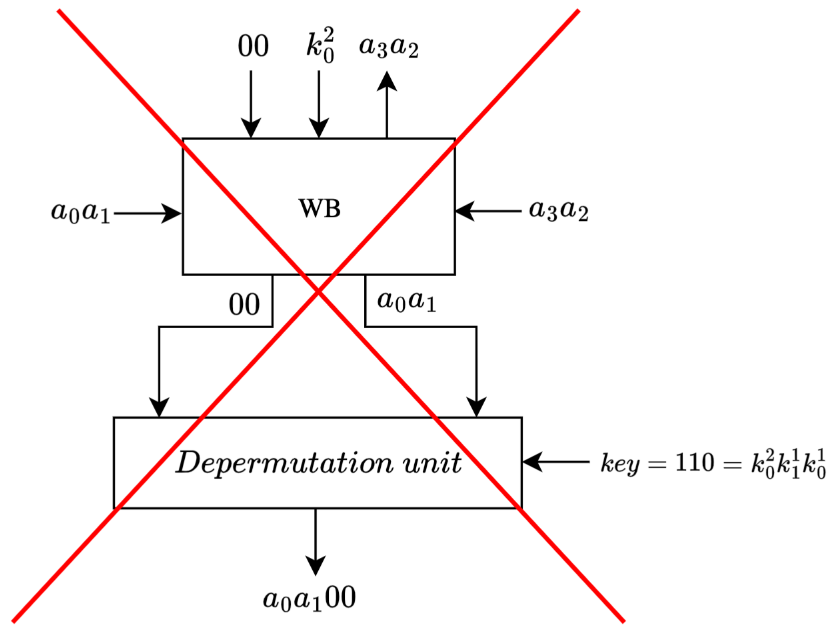


Figure 4.18: Permuted shift left by 2 example with White Block(WB)
Source: image derived from the [6]

Here the main problem arises from the fact that the structure has a depth of one such that the permutation key is partially associated. In this way is possible to conclude that for shifts with powers greater than one the WB is not able by itself to take into account the LSB of the key.
The example with the output $a_0a_1 00$ clearly demonstrates this error, where the most significant pair appears in reverse order from what is expected that could be resolved applying the respective LSB of the key with value 1.
The problem in other words can be described by observing that by shifting to higher powers we create a division of the data into sections larger than 1-bit and, in the event of a shift of these sections, they would respect a different segment of the key from the original one before the operation.

So, if our goal is to achieve generality for any given shift amount, we are forced to introduce something new. The two involved blocks are the «White Key Block»(WKB) and the «Black Block»(BB), as shown in Figures 4.19 and 4.20, respectively.

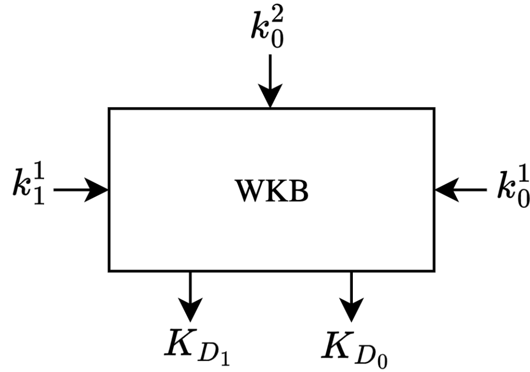


Figure 4.19: White key block
Source: image derived from the [6] article

The first is a block useful for generating a new key denoted K_D. The example with $n = 4$ and $block = 2$ provides a 3-input cell, matching with the bits of the key, where the one of the top, corresponding with the MSB, takes the decision if invert or not the least significant positions.
The vector K_D in fact does not possess a dimension identical to the native key but a reduced size with only the bits that could not be applied due to the insufficient depth of the tree. Precisely if the MSB is 1 the outputs are the two LSBs but in reverse order, if the MSB is 0 the order remains the same.

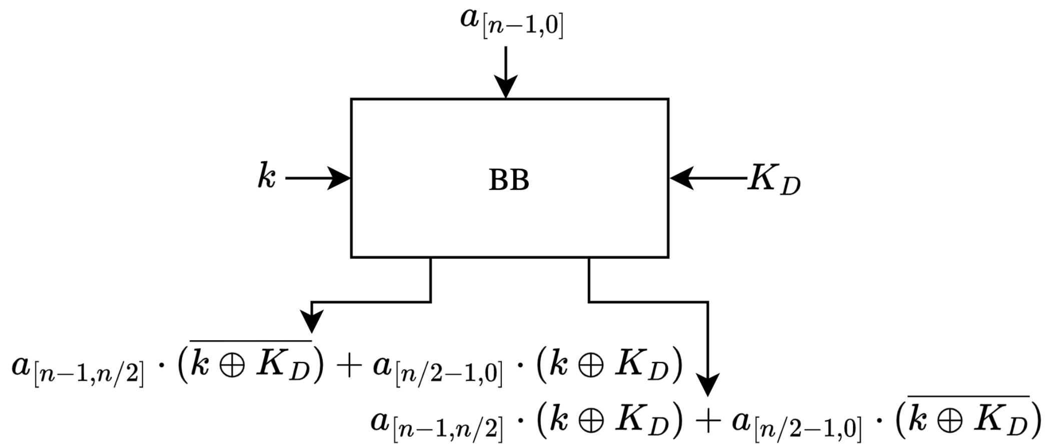


Figure 4.20: Black block
Source: image derived from the [6] article

The second block is capable of resolving inconsistencies by using the native key and the vector K_D, deciding whether to swap the halves of its input: if the signals k and K_D are equal the outputs are left and right sides in the same order, otherwise they are reversed.
Such as the shift by 1, in this way the mismatch between larger sections of a bit and respective segments of the permutation key is resolved and a new dichotomous tree structure can be derived. As shown in the 4-bit example in Figure 4.21 it is correctly obtained a_1a_000 from the permuted data $a_0a_1a_3a_2$.

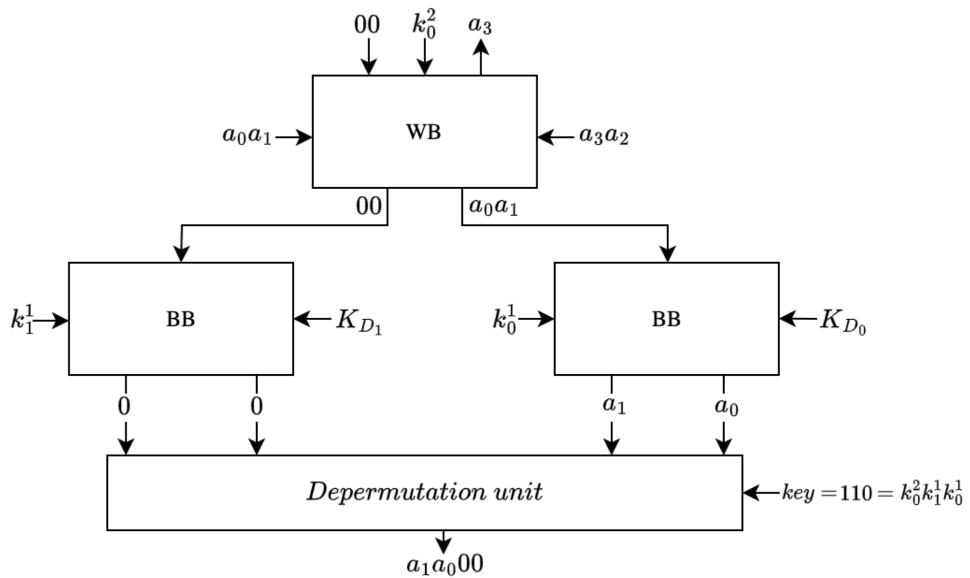


Figure 4.21: Permuted shift left by 2 example with White Block(WB), Black Block(BB) and White Key Block(WKB)

Source: image derived from the [6] article

Here the generation of the two components K_{D_0} and K_{D_1} is implicit: we would have to include the WKB to have a complete diagram.

4.4.3 Shift left by a power of 2

The shift left by a power of 2 is the third case which is already more complex than the previous ones.
In this case this step is fundamental because trying to apply the same structure previously mentioned there is a limitation: the vector K_D was generated with a simple block having to manage only a 3-bit key as the number of inputs of the block itself, not including generalities.
Therefore this step is crucial to extend the network in a new tree structure called

«White key tree». Here the purpose remains the same: it is requested to recognize any mismatches between the data sections and the key segments providing a new key K_D that bridges the inconsistencies.
The solution for a more generic shift amount is obtained by observing the previously developed approach for the data. Indeed the architecture is entirely analogous as it is sufficient to note that we are concerned with rotating the key's segments in the same direction as the data. The difference lies ensuring that no additional input bits are required and considering the possibility of inverting the LSBs at a later stage.
On this last aspect, the structure can be derived introducing a new block called «Black block key»(BBK) as shown in Figure 4.22.

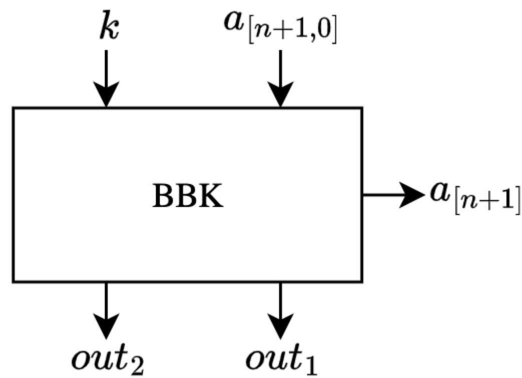


Figure 4.22: Black block key
Source: image derived from the [6] article

The cell receives an input bit of the key along with an its segments where the MSB of this latter is immediately taken to the output as part of the new vector K_D but is also used in a XOR function with k to decide whether to invert the LSBs.
In this context, it should be noted that the inversion does not mean to change the order of some bits but follows the permutation, swapping precisely two branches of the dichotomous tree.
The final architecture is thus provided through the example in Figure 4.23 for a 16-bit data and a shift amount of four where the three main aspects, already mentioned, can be observed:

- The architecture is analogous to that of the data.
- The top block of the structure has a link between the shifted-out value and the shifted-in value.

- The key could be divided into MSBs for the first levels of rotation and LSBs to manipulate internal swaps.

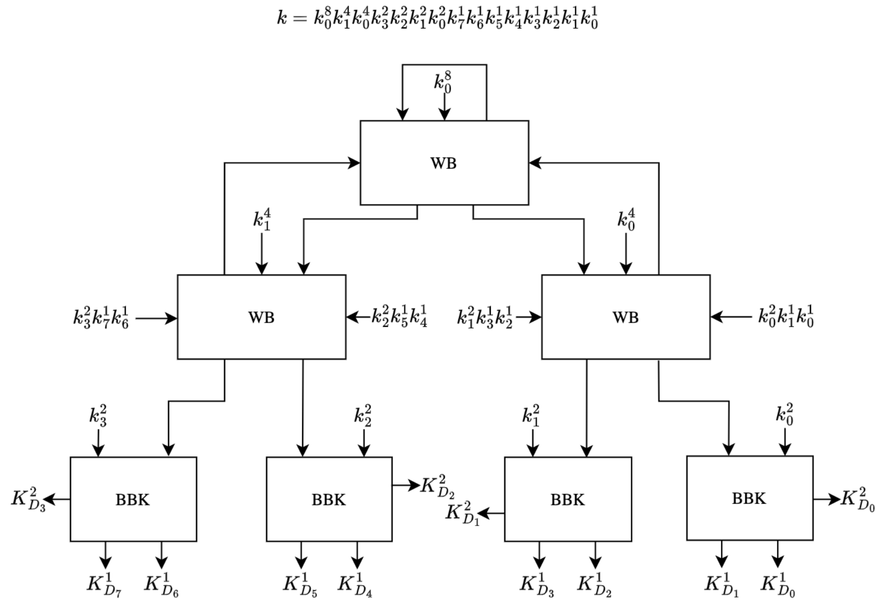


Figure 4.23: Permuted shift left by a power of 2 adaptation for 16-bit K_D generation with White Block(WB) and Black Block Key(BBK).

Source: image derived from the [6] article

4.4.4 Barrel shifter

The final implementation concerns the most complex case for handling any shift amount.
Unlike the previous steps, this is the only case in which directly a reuse of the last proposed architecture is applied: it is sufficient to organize in a multiplexer selection some blocks of shifts by a power of 2 as shown in Figure 4.24.

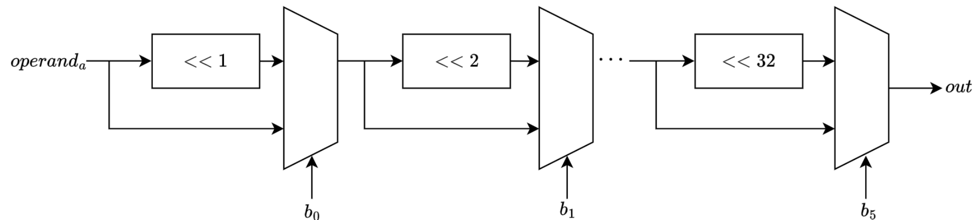


Figure 4.24: Barrel shifter configuration with multiplexer selection

Source: image derived from the [6] article

Knowing that a generic shift amount can be represented as the sum of the contributions from the powers of 2 of its bits, can be established a selection structure between a simple wire and the output of the shifter. Furthermore, working on 64 bits, this shift-amount, consisting of 6 bits of the second operand, corresponds to the mux selectors and tells whether or not to consider the shift of that weight.
As last remark the final implementation proposed until this moment knows how to manipulate permuted operands in the case of a logical shift to the left.
What to do for the other shift types? As initially stated starting with this last Barrel shifter configuration, small variations are sufficient:

- Logical shift to the right: can be obtained simply by observing that the negation of the key reverses the direction of rotation of the bits
- Arithmetical shift to the right: can be achieved by extracting the MSB and connecting it as an input to the upper block of the network
- Rotation to the left or to the right: reverse from the top block of the inputs without new insertion and without takes care about the permutation key

Additionally to these possible operations provided by the CVA6 ALU, our version of the shifter includes another feature. It is the Single Instruction Multiple Data(SIMD) option managed by means of the insertion of vectorization bit control that stops the requested operation with the specified parallelism.
Figure 4.25 shows the idea of this mode and consists of conceiving the received input data as consisting of multiple data bits over a reduced width, for which the requested operation is carried out separately.
In our case, three SIMD modes were implemented, working at the initial width of 64 bits: vectorization mode on 8-bit, on 16-bit and on 32-bit.

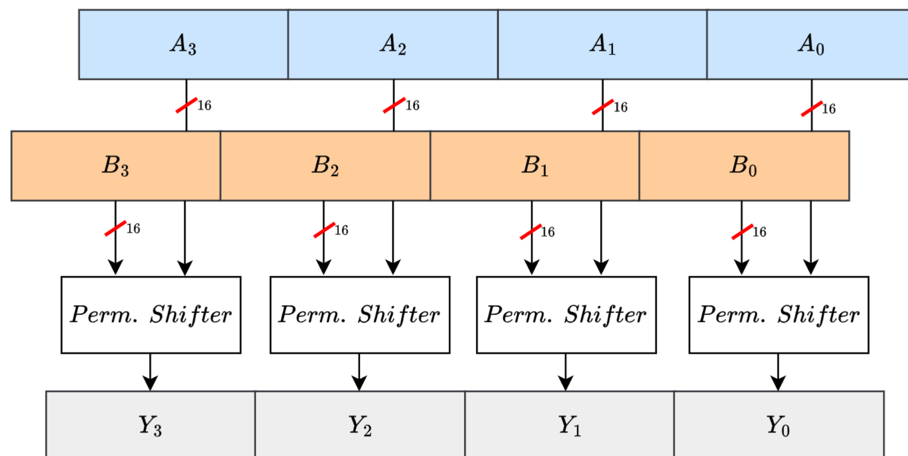


Figure 4.25: SIMD example for 16-bit permuted operand

4.5 Count operations

In order to conclude the permuted ALU, it is needed to go through counting operations. To begin with, it is important to define several groupings of bits:

- **Leading Zeros**: leading zeros are the consecutive zeros in the left most position before the first 1's and the specific operation corresponds in the count of these zeros at the beginning of the operand. In the example shown in Figure 4.26 the number of the left most zeros is 4.

$$0\,0\,0\,0\,1\,0\,1 \rightarrow \textit{leading zeros} = 4$$

Figure 4.26: Leading zeros example

- **Trailing Zeros**: trailing zeros are the consecutive zeros in the right most position before the first 1's and the specific operation corresponds in the count of these zeros at the end of the operand. In the example shown in Figure 4.27 the number of the right most zeros is 3.

$$1\,1\,0\,1\,0\,0\,0 \rightarrow \textit{trailing zeros} = 3$$

Figure 4.27: Trailing zeros example

- **Population**: population corresponds with the count of the 1's inside the operand. In the example shown in Figure 4.28, the number of 1s is 5.

$$1\,0\,1\,0\,1\,1\,1 \rightarrow \textit{population} = 5$$

Figure 4.28: Population example

The reason for these three definitions is to be found by looking at the ALU of CVA6 where CLZ, CLW, CTZ, CTZW, CPOP, CPOPW are required count operations and refer exactly to leading zeros, trailing zeros and population respectively.
For each of these cases, the adaptation to the permuted domain must be sought.

4.5.1 Population count

The first network to deal is the population counter, a unit that count the amount of 1s in the input operand, by solving operations such as CPOP and CPOPW.
For this type of operation there is a crucial point that needs to be underlined: the permutation is of no interest as it does not include changes to the binary representation.
The function in fact, as shown in the example of Figure 3.3, does nothing more than exchange randomly the sorting, according to the permutation key bits, leaving the number of 1s and 0s unchanged.
In this way the population counter shape can be derived easily from the CVA6 without changing the original structure as in Figure 4.29.

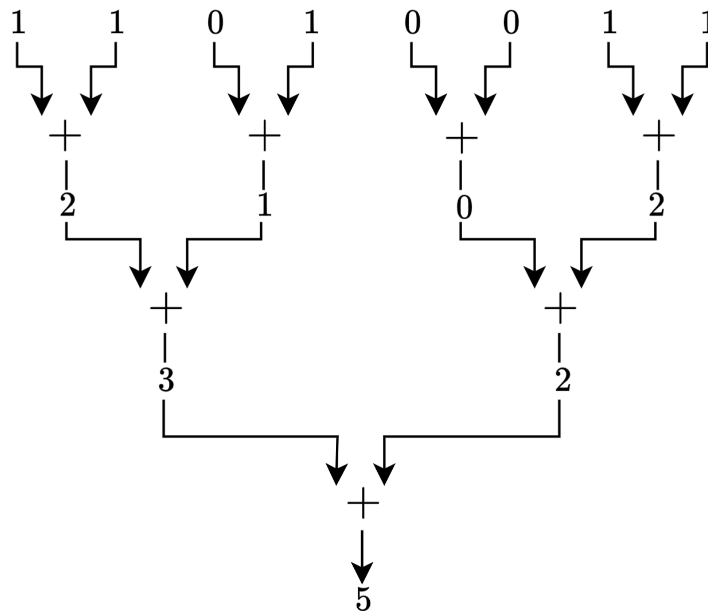
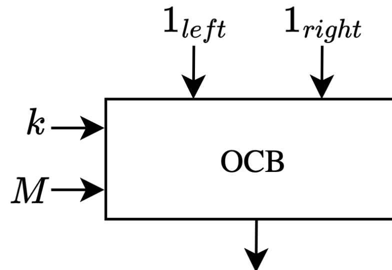


Figure 4.29: Population counter example for 8-bit permuted operand

Here precisely can be observed how it is a dichotomous tree structure, by summing pairs of bits and partial results up to the value of the permutation count.
To conclude, however, if the propagation of partial sums is unchanged by the type of transformation, the same does not happen when we have to care about obtaining a permuted count result. To solve this aspect, if in the case of the CVA6 the result was handled on 6 bits and then possibly extended internally to the ALU, in our case wanting to handle a permuted output, there is the need to use 64-bit permuted adders directly.

4.5.2 Leading/trailing zeros count

Leading and trailing zeros counters are the units that take care of counting the size of the homonym segments of the input operand, by solving operations such as CTZ, CLZ, CTZW, CLZW.
For these kind of sections can be observed the fundamental aspect opposite to that of the population: to observe the first 1 position, from the right or the left according to their request, it is necessary to somehow reconstruct the native operand to understand its real weight and stop the count.
The «Ones count block»(OCB) in Figure 4.30 is the elementary cell that follows this behavior, where can be observed how it propagates either their sum or just one side, receiving a key bit, 1s from the left and right and a maximum value for that level(M). Reference is made to the 1s because counting the zeros makes it more convenient to use the operand inverted bits as direct summing inputs.



$$1_{out} = (M \cdot (1_{left} + 1_{right})) + (\overline{M} \cdot (\bar{k} \cdot 1_{left} + k \cdot 1_{right})$$

Figure 4.30: Leading/trailing zeros block

The most important feature is the dependence from the key that possesses the control over propagation.
It means to have a description of a logic of each OCB that is more complex than a single adder as was the population counter. In terms of configuration, we are lucky that it is the same dichotomous tree, but with the distinction that each node is more composed. The need to introduce the key arises from the need to assess whether the count of zeros corresponds to the maximum possible for that level because otherwise, if it does not, it suggest the presence of a 1 that interrupts the sequence of zeros. Once identified, the network propagates the sum from a higher level using a simple wire, which can be implemented as a multiplexer.
As shown by the example in Figure 4.31, which represents an 8-bit counter in the specific case of leading zeros, each block implicitly subtends the adder and the multiplexer previously mentioned but it includes a key bit as input to indicate its subordination.

$$key = k_0^4 k_1^2 k_0^2 k_3^1 k_2^1 k_1^1 k_0^1 = 1011100$$

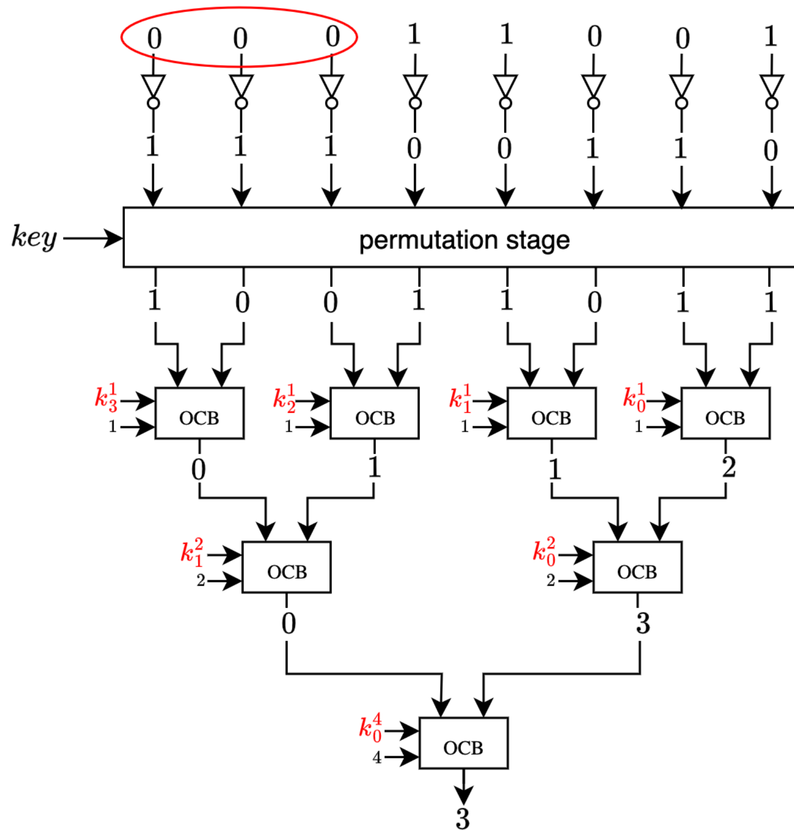


Figure 4.31: Leading zeros counter example for 8-bit permuted operand

From this example can be viewed how the counter network receives incoming operands where the number of leading zeros cannot be explicitly understood, by having the permutation. The key allows us to do the reconstruction with a configuration identical to that of the permutation, i.e. starting from its LSB and moving towards the MSB.
Finally, if the example shows a leading zeros counter, nothing changes moving the structure to the trailing zeros shape. Indeed its architecture can be represented in a similar way by changing the elementary block that, instead of favouring the passage of the left-hand side, promotes the propagation of the right-hand one.

Chapter 5

Permuted multiplier

The multiplication in the mathematical literature can be approached using different hardware implementation: the Carry-Save Adders(CSA), the parallel approach with Wallace or Dadda tree, high-root fast multipliers and decomposition algorithms. In our case, remembering to target the permuted domain, two methodologies are adapted:

- Multiplication recurrence or iterative way
- 2-way Karatsuba algorithm

5.1 Iterative way

The iterative approach is the simplest process that takes inspiration from decimal multiplication. It calculates several partial products and only after put them in addition to reconstruct the result, following the old school approach in dot notation as shown in Figure 5.1.

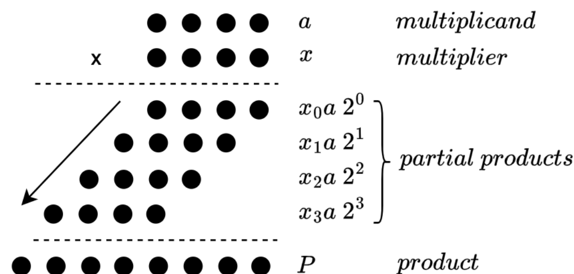


Figure 5.1: 4-bit dot notation iterative approach
Source: image taken from the [17] article

Here can be observed how the multiplication starts by multiplying each bit of the second operand by the first, from right to left. The first result is written right aligned thanks the least significant weight of multiplication in respect all the others case, where a shift by one is required to account the positional value of those bits. Continuing iteratively for each available bit, the product can be obtained and written as the sum of the partial products. The crucial point lies in observing some mathematical recollections that allow to describe a partial product at iteration j+1 as function of the previous one at iteration j as in the Formula 5.1.

$$
\begin{aligned}
P &= x_0 a \cdot 2^0 + x_1 a \cdot 2^1 + x_2 a \cdot 2^2 + x_3 a \cdot 2^3 \\
&= ((((x_0 a \cdot 2^4)2^{-1} + x_1 a \cdot 2^4)2^{-1} + x_2 a \cdot 2^4)2^{-1} + x_3 a \cdot 2^4)2^{-1} \\
p^{(j+1)} &= (p^{(j)} + x_j a\ 2^k)2^{-1} \qquad (5.1)
\end{aligned}
$$

This formula corresponds to the starting point for the architecture of the iterative multiplier which, remembering that only represents a single partial product, allows us to develop the pseudo-code of Algorithm 1.
Here a and x are the input operands, X, Y_0 and Y_1 are the registers through which to store the intermediate results and c_o represents the carry coming from the sum operation.
Precisely the two registers Y_1 and Y_0 compose the 128-bit result as 64-bit sections, most significant and less significant respectively, due to the need to handle the permutation domain on which a 63-bit key is supported and does not allow classical management.

Algorithm 1 | Iterative multiplication pseudo-code

```
    procedure MUL(a,x)                                   ▷ a • x
2:      X= a, Y0 = x, Y1 = 0, cnt = 0                    ▷ Initialization

4:      while cnt < n do
            if Y0[0] == 1 then
6:              c0||Y1 = X + Y1                          ▷ Addition X and last Y1
                c0||Y1||Y0 »= 1                          ▷ Right shift for next step
8:          else
                c0||Y1||Y0 »= 1                          ▷ Right shift for next step
10:         end if
            return Y1||Y0                                ▷ Result after n cycles
12:     end while

14: end procedure
```

Moving the focus on a possible hardware implementation for a multiplication between 64-bit operands, the Formula 5.1 and the Algorithm 1 suggest the need for only one shifter by 1, instead the complete Barrel shifter configuration, and one adder thanks the interpretation of the term 2^k as the direct selection of the MSBs of the result. This is why the pseudo-code, where sum computation is present, uses X and Y_1 as operands.
More in details, to be coherent with the subdivision of the result in Algorithm 1 and as shown in Figure 5.2, there is the need to split the shifter into two 64-bit units, that allows to handle the permutation domain, and there is a loop configuration.

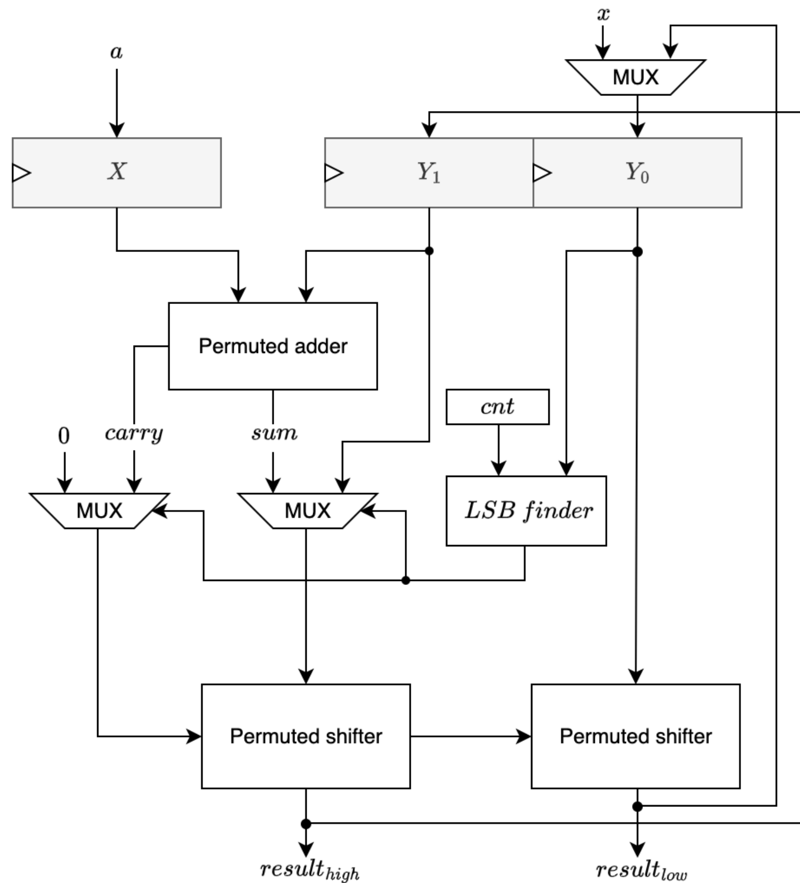


Figure 5.2: Permuted iterative multiplier architecture
Source: image derived from the [6] article

About this last aspect in fact, the Formula 5.1, which represents a single partial product, in order to compute the final result requires multiple iterations equal to the width of the operands.

In particular, the algorithm involves an initialization phase in which the operands a and x are stored in the respective X and Y_0 registers, while the Y_1 register together with the counter are reset. This reset state is necessary because the counter keeps track of how many iterations are still required to read the registers correctly.
At each cycle it is taken the LSB of the Y_0 register, in which the multiplier is contained, and if the bit is a 1 is computed the addition of the multiplicand before entering into the most significant shifter. Otherwise the addition is not performed and the value contained in Y_1 undergoes to a direct shift to the right. In the other side simultaneously the register Y_0, which initially contains the multiplier x, is shifted at each iteration by entering the shifted-out bit of the most significant shifter.
In others words when shifting, it is inserted the carry-out of the adder into the MSB of the most significant result portion and similarly, when extracting the LSB of Y_1, it is connected to the MSB of the least significant result portion, effectively shifting out the current LSB of Y_0, which corresponds to the decisional bit just compared.
In this way to conclude with the previously mentioned method, at the end of the 64 cycles, the registers Y_1 and Y_0 no longer store the multiplier x or a reset null value but rather the resulting product.
As last remark on the permuted management of the result, the use of two shifters comes to our help as it is requested to handle a 128-bit data with a 63-bit key. Consequently, as shown by Figure 5.3, there are two permuted separated sections according to the same key which are analysed individually when reconstructing the result.

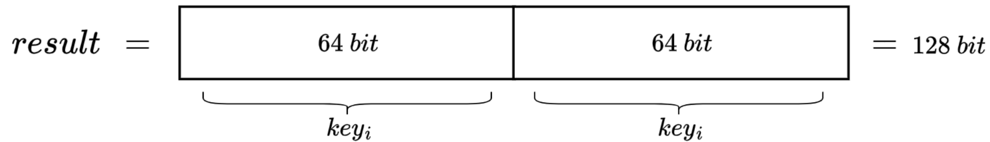


Figure 5.3: 128-bit permuted product management

5.2 2-way Karatsuba algorithm

This section is a new approach taken in an attempt to improve the previous implementation through the application of the Karatsuba algorithm.

> "The Karatsuba algorithm is a fast multiplication algorithm discovered by Anatoly Karatsuba in 1960 and published in 1962. It is a divide and conquer algorithm that reduces the multiplication of two n-digit numbers to three multiplication of n/2-digit numbers and to at most $n^{1.58}$ single-digit multiplications. It means that it is asymptotically faster than the traditional algorithm with n^2 single-digit products"
>
> *Wikipedia, 2024, [18]*

The 2-way typology corresponds to what is shown in Figure 5.4 based on the idea of being able to write both the operands in two separated and adjacent sections of MSBs and LSBs, respectively called left and right side.

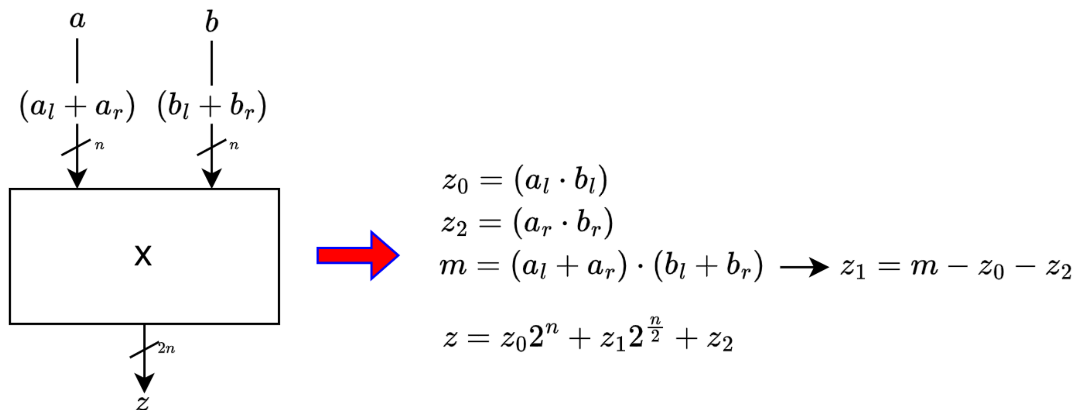


Figure 5.4: Partial contribution deriving the 2-way Karatsuba algorithm

The algorithm again involves calculating three partial products, but this time in a parallel manner in order to reconstruct the result later:

- z_0 corresponds to the product of the most significant sections
- z_2 corresponds to the product of the least significant sections
- z_1 corresponds to a combination of most significant and least significant sections.

The last term z_1 involves a first computation of an intermediate value m, given by the product of the sums of the most significant and least significant segments of the same operand. Only after the other two terms z_0 and z_2 are needed to find the final z_1 component.
The reconstruction, as shown in the last formula, is characterised by a shift for each partial product according to the weight of their operands and, as in the iterative method, by their sum:

- $z_0 \ll$ n, it is a multiplication between most significant bits
- $z_1 \ll \frac{n}{2}$, it is a mixed multiplication
- z_2 deriving from the least significant sections can be directly used as a partial product.

The crucial point of our architecture it is that, if n is higher than 2 and a power of 2, the three partial products can be computed recursively calling another time the Karatsuba algorithm at each new multiplication. This last step can be done by creating a tree structure on several levels until a minimum width is reached.
Trying to describe this approach according to a pseudo-code, it results in the Algorithm 2 in which the recursiveness has to be revised in the recall of the same procedure *Karatsuba_mul*.

Algorithm 2 | 2-way recursive Karatsuba algorithm

```
    procedure Karatsuba_mul(a,b)                          ▷ a • x
 2:
        a_l = a·2^(n−k)                                   ▷ left a
 4:     a_r = a[k-1:0]                                    ▷ right a
        b_l = b·2^(n−k)                                   ▷ left b
 6:     b_r = b[k-1:0]                                    ▷ right b

 8:     z_0 = Karatsuba_mul(a_l,b_l)                      ▷ Recursive proc call
        z_2 = Karatsuba_mul(a_r,b_r)                      ▷ Recursive proc call
10:     z_1 = Karatsuba_mul((a_l + a_r),(b_l + b_r)) - z_0 - z_2   ▷ Recursive proc call

12:     return z_0 · 2^n + z_1 · 2^(n/2) + z_2
    end procedure
```

The idea lies in repeatedly calling the algorithm with continuous division into three new contributions of each term z_0, z_1, z_2 as these are new multiplications.

From an implementation perspective, it is now necessary to translate the pseudo-code and, more importantly, address the challenge of handling the permutation of the operands.
The proposed solution is to divide the architecture into three main networks:

- **pre-computation tree**: is the initial structure in which recursiveness finds its place by dividing each new multiplication into the three contributions z_0, z_1, z_2 taking into account the domain of the countermeasure.

- **layer of multiplication**: is the structure that performs true multiplication in the classical sense having reached the minimum level of pre-computation with 2-bit operands. Assuming (a + b)(c + d) with a,b,c,d single bits the result is computed as p = ac + ad + bc + bd.

- **reconstruction tree**: is the final structure in which a partial reconstruction is calculated at each level according to the formula $z = z_0 \cdot 2^n + z_1 \cdot 2^{\frac{n}{2}} + z_2$.

Each of the three components is a trichotomous tree, as suggested by the definition of the algorithm, in which three partial products are computed and where the trichotomy consists of dividing and reconstructing each block into three more at the next level.
To give an idea, an example with 16-bit operands is shown in Figure 5.5 where, as can be seen, the network consists of 8 different blocks:

1. Left Right block ⇒ LR □
2. Left Right carry block ⇒ LRC ■
3. Karatsuba basic block ⇒ K ■
4. Karatsuba carry block ⇒ KC ■
5. Reconstructive basic block ⇒ z_0, z_2 ■ ■
6. Reconstructive carry block ⇒ m ■
7. Reconstructive last carry block ⇒ *last m* ■
8. Reconstructive top block ⇒ *top* ■

The colour system used serves to underline that each block has a different function, in order to respect the algorithm and manage the permutation, in which cannot be derived a simple logical function considering the complex usage of adders, shifters, extractors and others permuted networks.

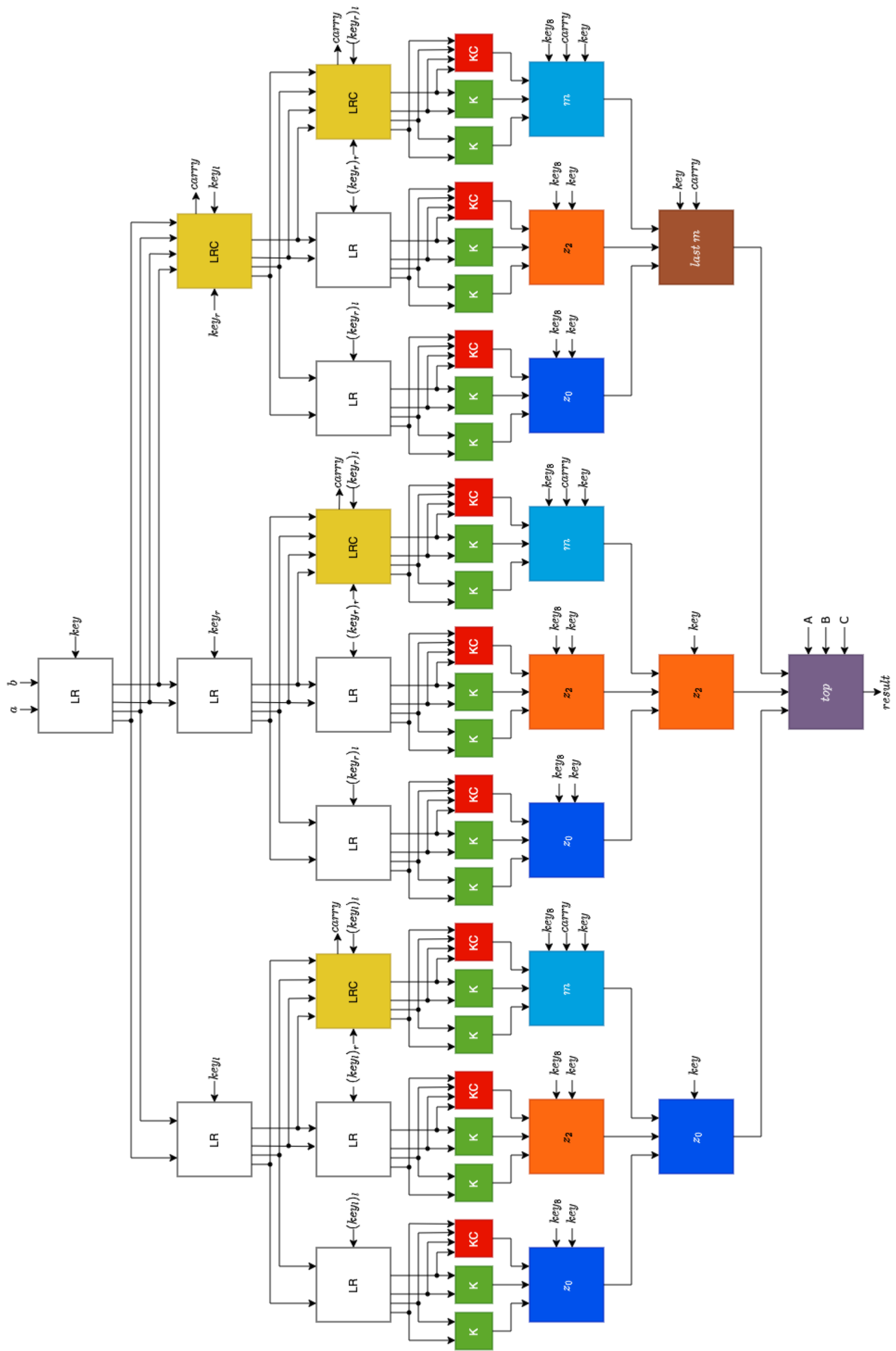


Figure 5.5: Trichotomous tree example for 16-bit permuted 2-way Karatsuba multiplier

Here can be observed a parallel configuration of the architecture requiring a reduction of the network's internal signals in an attempt to compress the size of the multiplier, as shown in Figure 5.6:

- Pre-computation phase halves the parallelism between levels, from 16 bits down to a minimum of 2 bits.
- Multiplication phase computes the first partial products over a width of 8 bits. This happens because of the term m, which on a minimum size of 6 bits, brought to use the next power of 2.
- Reconstructive phase returns to the maximum width of 16 bits like the parallelism of the input operands.

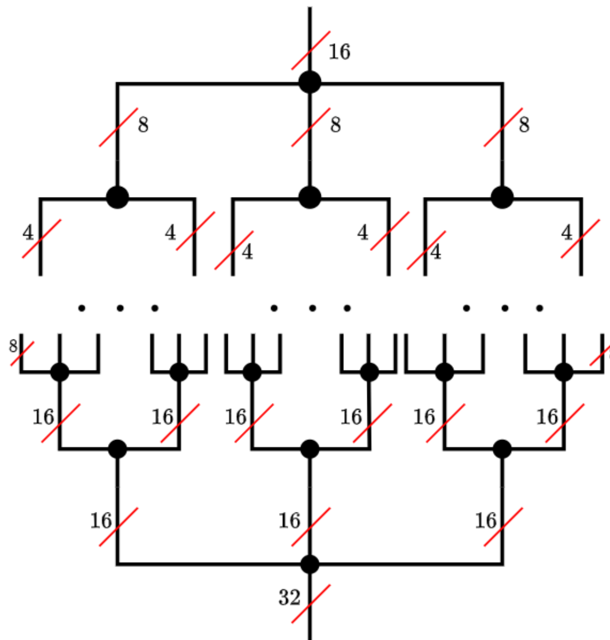


Figure 5.6: Tree-like example structure for 16-bit parallelism reduction

The 32-bit result in the example is presented in its normal parallelism, the double of the operands, but once again is recomposed as two separated and adjacent sections. In the specific case of 128-bit result, considering our aim for 64-bit multiplication, the management follows the Figure 5.3 with a single 63-bit key.
However, during the development phase of the entire network, several issues related to permutation management needed to be addressed. To provide a more in-depth analysis, each of the three macro-areas it is proposed in greater detail.

5.2.1 Pre-computation tree

The pre-computation tree is the network through which Karatsuba's algorithm finds a recursive application.
The idea of the tree is to divide the native 64-bit multiplication into the three partial products z_0, m(z_1 is rebuilt later), z_2 that are further multiplications with 32-bit operands. In this way, finding the multiplicand and the multiplier as a new powers of 2, there is the possibility of decomposing each partial term into three new further contributions.
This principle of recurrence leads to the construction of the pre-computation tree with a continuous recall of the algorithm up to the bottom level where the operands reach the minimum size of two bits. However, problems arise here, and two of them are identified: the subkeys extraction and the carry management in m computation.

1) Subkeys extraction

Decreasing internal parallelism leads to maintaining as permuted each section of data extracted.
Not being able to play with the indexes, by means of a simple extraction, the permuted domain forced to use a dedicated network capable to select the specific bits that the section would be subjected during the permutation phase.
The Figure 5.7 provides an example for a 16-bit data showing the extraction of the red or blue segments. This allows for an easy derivation of the key association for each possible data block, according to the permutation function, figuring out how to proceed.

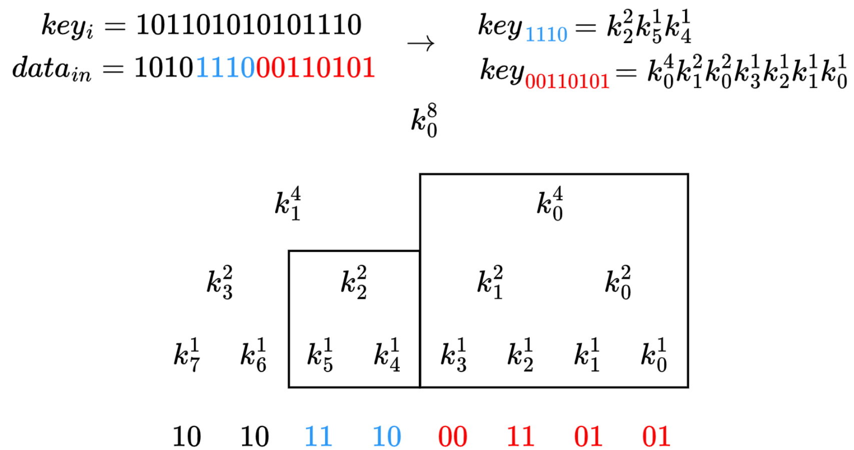


Figure 5.7: Key-extraction example for 16-bit permuted operand

2) Carry management in m computation

Recursion is not an immediate concept; it involves an adaptation to return powers of 2.
The contribution z_1 does not simply involve multiplication by passing through the internal value m, where the recurrence introduced by the first addition generates a carry-out bit. In this way it brings the internal signal width to *n+1* bits, i.e. an odd representation.
The problem of this approach comes here, for example after the first application of the algorithm, trying to apply the same procedure on the new 33-bit multiplication, where it is obvious the difficulty to handle the new recursion.
Luckily the adaptation finds a solution by decomposing the multiplication into two contributions: one belonging to the carry bit and one associated with the others n bits which can be re-written as a new powers of 2 multiplication.
This procedure exploits mathematical properties that extract the contribution of one bit deriving several contributions that, in the case of the most significant bit, are three terms as shown in Figure 5.8:

- A corresponds to the contribution if both carries are equal to 1, making the AND and shifting the result by two times the parallelism of the operands.
- B corresponds to the contribution if the carry of the first operand is 1. In this case the bits of the second operand, except its carry, are shifted by n bits.
- C corresponds to the contribution if the carry of the second operand is 1. Opposite to the previous case, the bits of the first operand, except its carry, are shifted by n bits.

$$m = (a_l + a_r) \cdot (b_l + b_r) \rightarrow \textit{size operand} = n \rightarrow (c_{a_{[n]}}\ sum_{a_{[n-1:0]}}) \cdot (c_{b_{[n]}}\ sum_{b_{[n-1:0]}})$$

$$\Rightarrow \text{carry contrib.} = \text{A} + \text{B} + \text{C} \begin{cases} \text{A} = (c_{a_{[n]}}\ \&\ c_{b[n]}) << 2n \\ \text{B} = (c_{a[n]})?\ \ sum_{b_{[n-1:0]}} << n : 0 \\ \text{C} = (c_{b_{[n]}})?\ \ sum_{a_{[n-1:0]}} << n : 0 \end{cases}$$

$$\boxed{m = \text{carry contrib.} + \ (sum_{a_{[n-1:0]}}) \cdot (sum_{b_{[n-1:0]}})}$$

Figure 5.8: Carry decomposition formulas for m computation

The crucial point of this property is to restore the perfect conditions to apply a new recursion step using powers of two operands. At this level in fact the carry is

not necessary and instead ends up directly as input signals of the reconstruction tree.
In Figure 5.5 this process is carried out within the yellow blocks which differ from the white ones for z_0 and z_2, where just key extraction is sufficient. Finally to get a better idea of how the extraction of the three contributions works, an example is given in Figure 5.9.

$$a = 10101100 \qquad sum_a = (1010 + 1100) = 10110$$
$$\rightarrow \qquad \Rightarrow \begin{cases} A = 100000000 \\ B = 10010000 \\ C = 01100000 \end{cases}$$
$$b = 10101111 \qquad sum_b = (1010 + 1111) = 11001$$

Figure 5.9: Carry decomposition example for m computation with 8-bit operand

The example shows the case where both the carries are equal to 1 and thus each term(A, B, C) are different from zero. In the contributions sum_a and sum_b, the decisional carry is represented by the MSB.

5.2.2 Layer of multiplication

The multiplication layer is the level where the partial products are truly computed. The previous structure was in fact concerned with a recursive application of the algorithm without really computing anything, up to 2-bit last partial products z_0 and z_2 and the 3-bit term m.
Regarding the first two, i.e. $\mathbf{z_0}$ and $\mathbf{z_2}$, the choice is to divide for the umpteenth time the operands into single bits in order to calculate the partial products according to the Karatsuba algorithm. The new process is related with the internal m, and so z_1, where the choices are to write the term in its extended Formula 5.2 made by m_1 and m_0.

$$z_0 = (a_1 \cdot b_1)$$
$$z_2 = (a_0 \cdot b_0)$$
$$m = (a_1 + a_0) \cdot (b_1 + b_0) \;\rightarrow\; z_1 = m - z_0 - z_2 = a_1 \cdot b_0 + a_0 \cdot b_1 = m_1 + m_0$$

$$z = z_0 2^n + (m_1 + m_0) 2^{\frac{n}{2}} + z_2 \tag{5.2}$$

Regarding the third contribution m, and so $\mathbf{z_1}$, following the same procedure of z_0 and z_2 but with an internal value m in a 3-bit parallelism, one more adaptation is required: decomposition of the Figure 5.8 and the direct addition of the carry contribution.

In this way can be determined the third term z_1 as the Formula 5.3, ready to be used.

$$m = carry + (a_1 a_0) \cdot (b_1 b_0) \Rightarrow z_1 = carry + (a_1 a_0) \cdot (b_1 b_0) - z_0 - z_2$$

$$z_1 = carry + (z_0' 2^n + (m_1' + m_0') 2^{\frac{n}{2}} + z_2') - z_0 - z_2 \qquad (5.3)$$

As these last Formulas indicate, in any case, it is possible to conclude that the first actual computation always follows the algorithm but through a step of 4 partial products instead of 3.

5.2.3 Reconstructive tree

The reconstruction tree is the network through which Karatsuba's algorithm is applied in order to get back the result. In fact, after the multiplication layer there are 243 partial products on 8 bits to be handled.
The idea of the tree is to apply the last formula of Figure 5.4 to compute the terms z_0, z_2 and m of the next layer and to arrive at the last three contributions, that allow to express the output result.
Here, however, in order to have a correct propagation, two difficulties need to be managed: the key extension and the change of the permutation key.

1) Key extension

Increasing internal parallelism leads to maintain as permuted each section of the extended data where a classical extension is not valid.
Normally indeed, to extend a data, some zeros are placed in front of the number up to the new parallelism but additionally, in a permuted path, also the key needs an extension with an insertion of zeros that must follow the permutation.
A different approach is required because the old data, in its new extended version, corresponds with the LSBs section which needs to be controlled by the same key bits as before the extension. Consequently, traditional zero padding cannot be applied as shown in Figure 5.10.

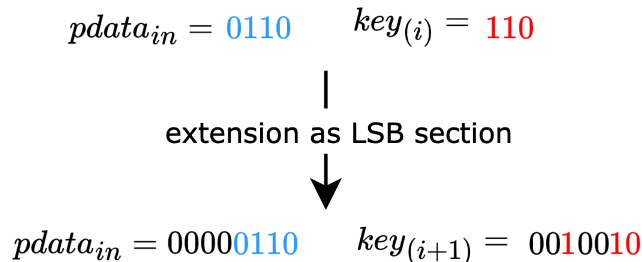


Figure 5.10: Key extension example from 4-bit operand to 8-bit

2) Change key of permutation

The extension of the key is not the only issue to solve.
Trying for example to go from 4-bit to 8-bit data, over the normal key extension, with an increment from 3-bit to 7-bit, a shift to a new target domain is required. The derived carry that arrives from the decomposition of a previous level it is a clear example because its representation is certainly different from a simply extended key. In fact the carries that go directly into the reconstruction tree have their own domain that is certainly different.
In this way it became something to solve also if the the final aim is to return to the native 63-bit key where it is preferable to progressively use larger sections, not simple extensions with zeros, as they lack a clear connection to the permutation domain.
To address this issue, two dedicated components, named «reorder key» and «change key», are proposed. The preliminary step in developing at these two modules is to figure out how to switch from one permutation key to another, without passing through the native representation of the data.
Starting from the first of the two blocks, it answers a specific question:

What differentiates two keys?

This question is a crucial point because represents the source of the problem. Here, in order to reply, can be observed the applications of any two permutations in which the keys already differ only by their MSB as in Figure 5.11.
Remembering then that a 1 means swap and a 0 means identity, if the first two sections correspond to the left and right of the data itself we can see how well a mismatch between the applications is created from the outset. Continuing the permutation towards the LSBs of the key means having groups of swapped bits and with each new different pair a further inversion of the splits.

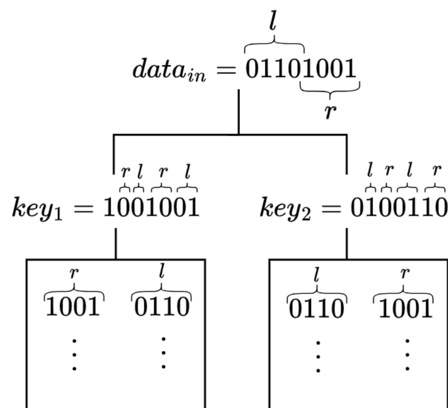
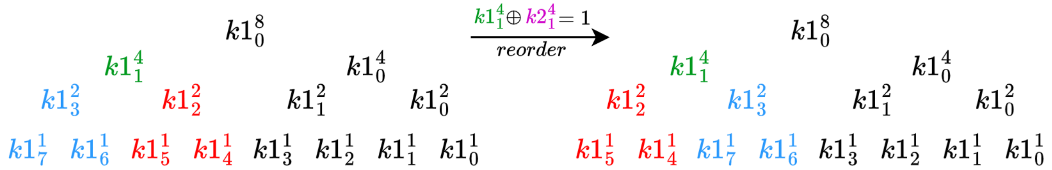


Figure 5.11: Key misalignment example with difference in the MSB position

The reorder network called «reorder key» is exactly the kind of circuit that overcomes this problem. It deals precisely with resolving the misalignment by comparing from the MSB to the LSB the bits of the source key against the target one, creating inversions for each difference encountered.
This circuit is necessary because an inversion can touch several bits at the same time: if there is a first difference in the MSB, it means having to invert the next two bits dealing with $n/4$ bit groups, the next four bits dealing with $n/8$ bit groups, and so on. It corresponds in practice to an inversion according to the permutation function as shown by the example in Figure 5.12 with the difference about the MSB-1.

$$k1 = k1^8_0 k1^4_1 k1^4_0 k1^2_3 k1^2_2 k1^2_1 k1^2_0 k1^1_7 k1^1_6 k1^1_5 k1^1_4 k1^1_3 k1^1_2 k1^1_1 k1^1_0 = 010\ldots$$

$$k2 = k2^8_0 k2^4_1 k2^4_0 k2^2_3 k2^2_2 k2^2_1 k2^2_0 k2^1_7 k2^1_6 k2^1_5 k2^1_4 k2^1_3 k2^1_2 k2^1_1 k2^1_0 = 000\ldots$$



$$rk1 = k1^8_0 k1^4_1 k1^4_0 k1^2_2 k1^2_3 k1^2_1 k1^2_0 k1^1_5 k1^1_4 k1^1_7 k1^1_6 k1^1_3 k1^1_2 k1^1_1 k1^1_0$$

Figure 5.12: Reorder example with difference in the MSB-1 position

The solution, however, does not end there because up to this point it was concerned with understanding the disparity between two keys. Now the next step needed is to make the transition from one key to another via, what is called «change key».
Luckily for this case a property of permutations comes to our aid: when two keys differ only by bits at the same level of the tree, transitioning from one key to the other can be achieved by applying a new permutation stage with the key being the XOR of the two.
Following the aim to apply the rule between two generic keys, it is first needed the last «reorder key» network to solve the mismatch of two domains that are not directly comparable.
In this way, the newly reordered and target keys have weights on the same plane, allowing a XOR function to generate the new value and subsequently apply the permutation.
With this approach the final structure can be concluded that consist of a «reorder key» and a «change key» in cascade as shown in Figure 5.13.

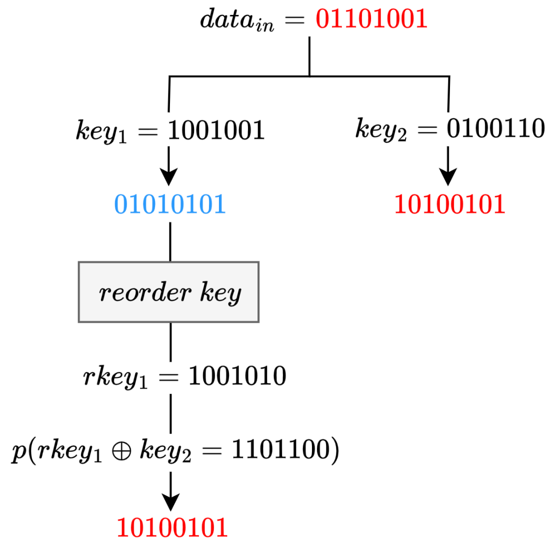


Figure 5.13: Adaptation process from key_1 to key_2 example

The problem observed so far is not only present in the reconstruction phase but already in the pre-computation. In fact, in the internal value m it is needed to take care about the left and right addition of the operands a and b, each with its own permutation key as shown in Figure 5.14.
The permuted adder of Section 4.3 was characterized by a single permutation key, the same between the two operands, leading us back to the problem just presented. In this case then, it is necessary to choose an adaptation to one of the two keys and reintroduce the cascade of «reorder key» and «change key».

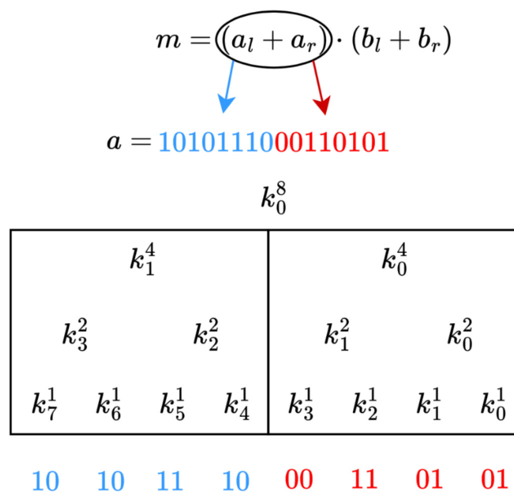


Figure 5.14: Key misalignment example in m computation

Chapter 6

Validation process and analysis

The functional validation means to check the correctness of the results of the permuted hardware architectures on FPGAs with a software model used as a golden reference.
This step is crucial before a designed structure can truly be considered functional. Indeed the simulation phase alone cannot fully evaluate the physical and performance characteristics of a hardware design by providing a superficial analysis; it is nice from a behavioural point of view but very bad from a physical point of view. Additionally parameters such as delay, noise and interferences are very complex to be predicted under a single simulation aspect where, on the contrary, ideal models with simplified and approximated propagation times are assumed. In this way the simulation does not account real-time execution, so that a very long on-board projection might only take a few seconds. Finally, it may have simulated a simple internal module, neglecting the integration of the entire system and, consequently, skipping factors that could compromise the functionality.
On-board validation allows us to verify all these aspects by providing a characterisation of the design on several points:

- Energy consumption
- Maximum operating frequency
- Heat management
- Latency and throughput
- Interaction with peripheral devices

6.1 Hardware interface

The architectures proposed in Chapters 4 and 5 have been described in SystemVerilog and have been implemented using the Xilinx Vivado hardware design tool.
The validation was developed targeting the Xilinx Artix-7 100T and the Kintex-7 FPGAs through Vivado Default run strategy for synthesis and implementation. Precisely the Kintex-7 was chosen to be used for those architectures that were too large for which the Artix-7 100T was not sufficient.
Each design was tested with an operating frequency in accordance with the critical path estimated after an initial synthesis phase. For example, the maximum frequency used was 50MHz in the cases of the adder and iterative multiplier, while the minimum was 6.125MHz in the case of the Karatsuba arrangement.
Each of the proposed modules were developed according to a fully combinatorial architecture, which required a specific support structure to enable their testing. In order to ensure communication with these modules, was implemented a Functional Validation Architecture(FVA), such as the one illustrated in Figure 6.1 and as noted in [19].

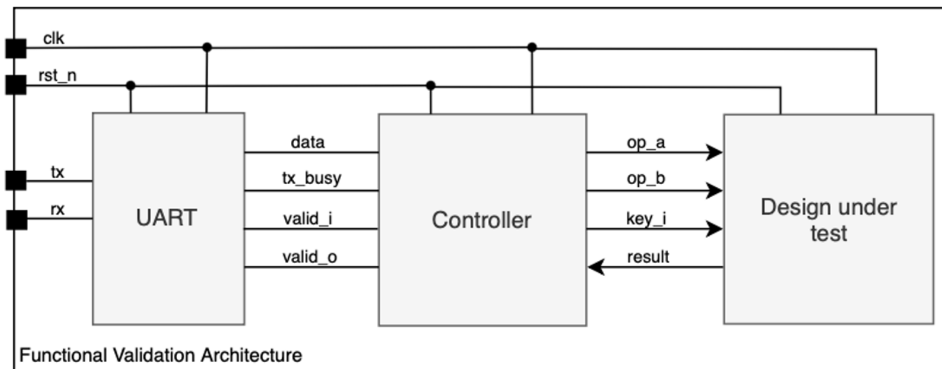


Figure 6.1: Functional validation architecture
Source: image taken from the [19] article

Here can be observed in the first line the Design Under Test(DUT) including the permuted adder, permuted shifter, permuted ALU and permuted multiplier while, in the second entry, the contour architecture consists of two additional components:

1. Universal Asynchronous Receiver-Transmitter(UART) which serves the communication interface with the host computer, facilitating the data transfer

2. Finite State Machine(FSM) or controller responsible for managing both data flow and protocol control signals, ensuring effective coordination with the DUT.

6.1.1 Controller

The controller or FSM is the module that coordinates the communication between host computer and DUT. This takes the data transmitted by the UART as input, to dictate the timing of the output and send it for processing, and receives the result as input to take care of its transmission, respecting the «busy» condition of the communication channels.
In order to correctly manage a combinatorial module with input and output register wrappers, the FSM is composed of 4 states(IDLE, WAIT_TX, WAIT_COMP, OUTPUT_SEND) as shown in the graph in Figure 6.2. Here, the specific count value for reading the result is set to 10, after a first estimation of the process duration, to ensure that the computation is correctly brings to the output by the circuit.
Finally it is important to note that for the iterative multiplier, employing 64 clock cycles, there is the need to handle an additional validation signal to ensure that the product is read at the correct time, not reported in the example.

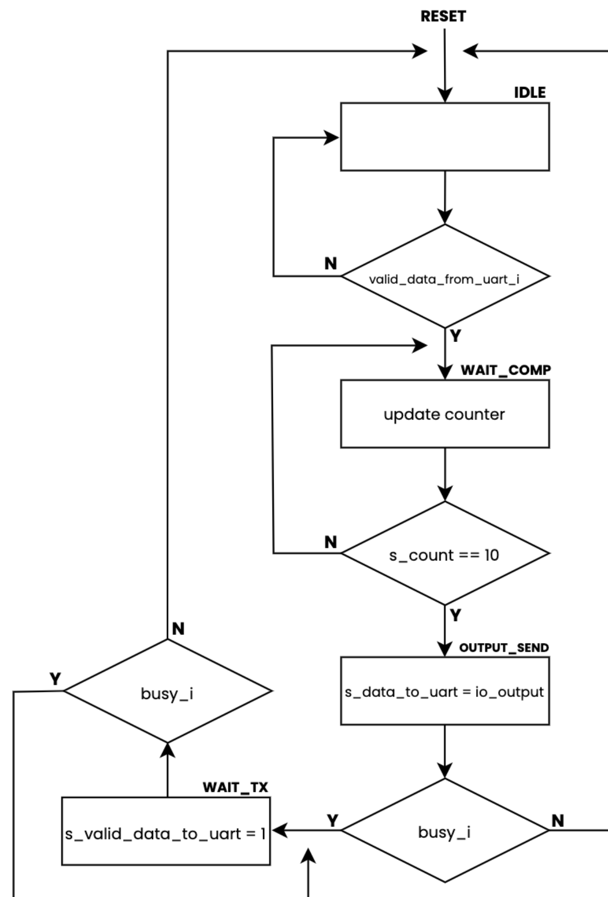


Figure 6.2: Functional Controller graph

6.2 Software interface

As a software golden model, a Python program was developed. Here the main purpose is to randomly generate the input signals for each DUT and, by using a software implementations of the permutation and depermutation functions, to perform the respective operations of the design in the native domain by comparing the two output results.
The entire process makes it possible to verify the correctness of an instruction and identify any discrepancies between the two versions of the path. In the event of differences between the software and hardware outputs, an integer counter reports the error and displays it immediately on the screen.
The program integrates perfectly itself with the FVA, thanks to the usage of the Serial library, which manages communication between the software and hardware fixing a BaudRate at the optimal speed of 115200bps. Given the frequencies involved of a few MHz, in fact, there is no problem in respecting the Formula 6.1 binding the two parameters to guarantee reliable and correct interaction.

$$f_{ck} \geq \frac{Baud\ rate}{2} \tag{6.1}$$

Regarding to data again, a function is implemented to construct a single byte-sized string related with the DUT, specified by a macro-variable. Similarly the read-out via UART of the result works where a single string it is used before depermutation and comparison. In general can be concluded that, using the FVA, communication is efficient and simple, being able to launch millions of test points.
Below there are three code used in the test of the permuted ALU:

- Listing 6.1: shows the random generation of the inputs in which there is a specific set of the most significant byte since it is the ALU operation.
- Listing 6.2: is the permutation function analogous to the hardware function via transposition call.
- Listing 6.3: is the function communicating with the UART which takes care of sending and receiving new data and results maintaining the error count.

The ALU case is the only test that does not involve direct software computation for each operation required, having decided to use the CVA6 ALU as the reference model. In all other cases, in fact, the desired action is performed through the use of symbols such as +, » and *, i.e. the equivalent software implementations.

```python
def random_value(size=N_BYTE_RX, verbose=False):
    message_in = bytearray()
    message_in.append(int(format(random.randint(0, 56), '06b') + '01',2))
    for i in range(size-1): message_in.append(random.randint(0, 255))
    if (verbose): print("generated random message = ", message_in.hex())
    return message_in
```

Listing 6.1: Python random ALU inputs generation

```python
def permutation(width, transposition_value, data_i, key_i):
    levels = math.ceil(math.log2(width))
    permute_data = [None] * (levels + 1)
    permute_data[levels] = data_i

    key_i_rev = list(f"{int(key_i, 16):0{64}b}")[1:64]

    key_i = []
    for j in range(len(key_i_rev)):
        key_i.append(key_i_rev[62 - j])

    for i in range(levels - 1, -1, -1):
        if (list(f"{transposition_value:0{6}b}")[i] == '1'):
            key_segment_start = (width - 2) - (pow(2, (levels - i - 1)) -
 1) + 1
            key_segment_end = (width - 2) - (pow(2, (levels - i)) - 2)
            key_segment = key_i[key_segment_end:key_segment_start]
            permute_data[i] = transposition(width, pow(2, i),
permute_data[i + 1], key_segment)
        else:
            permute_data[i] = permute_data[i + 1]
    return permute_data[0]
```

Listing 6.2: Python permutation function

```python
def test_with_random_inputs(verbose=False):
    hardware_error = 0
    software_error = 0
    for i in range(ITERATION_NUMBER):
        message_sent = random_value()
        key_i = message_sent.hex().zfill(49)[2:18]
        USB_serial.write(message_sent)
        message_read = USB_serial.read(N_BYTE_TX)
        if (message_read.hex().zfill(33)[0:2] != "00"):
            hardware_error += 1
        presult = depermutation(64, 63, message_read.hex().zfill(33)
[2:18], key_i).zfill(16)
        if( message_read.hex().zfill(33)[18:34] != presult):
            software_error += 1
    print("Error from hardware verification: ", hardware_error)
    print("Error from software verification: ", software_error)
```

Listing 6.3: Python results comparison from UART module

6.3 Results

For the acquisition of the results, the process is divided into several steps, following a chronological sequence to first obtain the desired ALU.
Initially, attention is paid to fundamental components such as adder and shifter, on the second line there is the ALU as a whole, and to conclude there is the observation and the verification of the multiplier as a standalone component.
For compatibility with the FVA, and thus with the hardware and software interface levels proposed in the previous two sections, wrappers were defined for each DUT containing input and output registers, which also enabled the estimation of the maximum operating frequency.

6.3.1 Adder and shifter comparison

The adder and shifer components, in view of their complexity and importance for almost all operations, are the first on which a validation phase was carried out. In fact, they are the elementary cells in the comparator and the counters, and finally in both the methods of the proposed multipliers.
Their verification was carried out on Artix-7 100T through which, after observing the correct operation of both, post-implementation data was collected on the area used and the critical path, as shown in Table 6.1.

Component	Area(Slices LUT)	T_C(ns)
CLA	152	6.2
Permuted CLA	242	7.6
Permuted Shifter	2493	24.2

Table 6.1: Area and critical path results for CLA, permuted CLA and permuted shifter

Here the main feature from the collection is the result of a normal CLA through which an initial estimation of the permutation cost could be made. Precisely the great similarity with the permuted version was exploited by defining the PGB of the Figure 4.10 and implementing the same network.
Starting from this point can be highlighted how there is an increase in area, as already anticipated in general with the use of hardware redundancy in Chapter 1, and how it is possible to estimate in detail the cost by introducing the key management.
The results extracted tell us that the area has an increment factor of about 1.59x while the critical path has an increment factor of about 1.22x. As we might have

expected, this performance indicates that a countermeasure, while securing the application, makes all the parameters involved worse.

6.3.2 ALU comparison

The next step is the construction of the entire permuted ALU according to that of CVA6.
Following this approach, the software development was slightly different from a program point of view, simulating in general the behaviour of all operations. In fact, having the ALU of the CVA6 available, it is decided to directly perform a hardware check by loading both paths on board, as in a simulation of the redundancy after integration of the countermeasure.
The software program in this case is realised by means of a simple serial one-byte read-out of the resulting value from the hardware comparison, and in order to ensure that this stage also did not return erroneous values, it is still chosen to send the outputs of both ALUs to assess their consistency.
The table 6.2 shows the obtained area and critical path results.

Component	**Area(Slices LUT)**	**T_C(ns)**
CVA6 ALU	2718	11.5
*Permuted ALU**	82859	58.4
Partial Permuted ALU	20865	27.6

*validation process with Kintex-7

Table 6.2: Area and critical path results for ALUs

These performances clearly indicate that no single permuted version is reported. In fact, the final version decided to use is the «partial permuted ALU» which, as implied, does not have all 63 permuted operations.
The reason lies in the count operations of Section 4.5 where the tree structures have each node consisting of the logic of a permuted adder, in the case of population counter, and some additional gates, in the case of leading/trailing zeros counter.
Following this approach a 64-bit adder for 63 total nodes takes up a minimum of 15k Slices LUT just for these structures that, making an initial comparison with the ALU of CVA6, means a huge overhead since each counter should already be 5 times the entire unit. The solution adopted in this way is to bypass these networks by leaving the counters unchanged in their native shape by adding stages of depermutation of the operands and permutation of their result.
By doing so as shown in Table 6.2, the final result has an area of about 21k Slices LUT considered acceptable, leading to increment factors of about 7.67x in the surface case and about 2.4x in the critical path case.

6.3.3 Multiplier comparison

The last analysis concerns the multiplier proposed as a standalone component, which in the case of the CVA6 is an Array multiplier with a single latency clock cycle.
The verification is carried out on Artix-7 100T, in the case of the Iterative architecture, and on Kintex-7, in the case of the Karatsuba architecture, through which, after observing the correct operation of both, the post-implementation data relative to the area used and the critical path was collected in the Table 6.3.

Component	Area (Slices LUT)	T_C (ns)	Number of cycles	Run strategy
CVA6 multiplier	2858	9.0	1	*Vivado Default*
Permuted iterative multiplier	876	13.0	64	*Vivado Default*
*Permuted Karatsuba multiplier**	100135	143.3	1	*Vivado Default*
*Permuted Karatsuba multiplier**	126531	79.3	1	*High Performance*
*Permuted Karatsuba multiplier**	92230	92.9	1	*High Area Optimization*

*valiation process with Kintex-7

Table 6.3: Area and critical path results for CVA6 multiplier, permuted Iterative multiplier and 2-way Karatsuba approach in three different run strategies

Iterative way

The first iterative route has several contrasting aspects, seeming better in some of them.

- Area: the results show a smaller occupied area due to the use of a single adder and of two shifters as proposed in Figure 5.2, which however would appear to be inconsistent with Table 6.1 showing the single shifter. The reason for this advantage arises from the fact that the standalone architecture is a Barrel shifter, i.e. a union of several levels through a multiplexer selection, and just the simplest degree of a single shift position is required, bringing in a very compressed solution.

- Critical path and latency: the results show minimal overhead regarding the critical path, suggesting that it is a highly efficient architecture. However, the number of cycles and thus the latency required to complete an operation, cannot be overlooked. For the required case of a 64-bit multiplier, the iterative version must pass through the same circuit 64 times to compute the same number of partial products, compared to just a single cycle in the Array configuration.

The performance thus obtained indicates an increase factor of 0.31x in the case of the surface area, an increase factor of 1.44x in the case of the critical path and a factor of 64x in the case of the number of cycles.

2-way Karatsuba

The second way according to the Karatsuba algorithm does not seem to be encouraging at first glance.

- Area: the results show a much larger occupied surface area. The motivation goes back to what was said in Section 5.2, namely how each tree is composed of permuted adders, shifters and extractors that raise the parameters of the final architecture. Although several different strategies have been tried, none have led to big improvements.
- Critical path and latency: the results once again show a large deterioration of the critical path to about 10 times the reference one without, however, introducing any deterioration from the latency point of view.

The performance thus obtained indicates an increase factor of 32.3x in the case of surface area, an increase factor of 8.81x in the case of critical path and a factor of 1x in the case of number of cycles.

Optimizations

Before concluding that both the implementations have excessive overhead, it is still possible to explore their optimization, as shown in Table 6.4.
The Karatsuba architecture in particular despite has each parameter much higher than the initial model, it is a parallel architecture using only one clock cycle for a multiplication.
This indicates that there is a potential for improvement by applying the pipeline technique, which could enhance performance where, ideally with 10 stages of registers, it might be possible to raise the maximum operating frequency to 100 MHz.
This approach could also provide the additional advantages of pipelining, such as an increase in throughput with one data output per clock cycle, although the number of cycles for a single operation would correspond to the number of registers used. The only drawback remains the area, as pipelining increases the area occupied through the additional registers rather than reducing it.
Shifting the attention to the iterative architecture there is no way of realising optimizations because of its loop structure that obliges to compute the multiplication with a number of cycles equal to the parallelism whatever improvements we manage to apply.

Component	Parameter to optimize	Y/n
Iterative	*Throughtput*	✗
Karatsuba	*Throughput*	✓
Iterative	*Critical path*	✗
Karatsuba	*Critical path*	✓

Table 6.4: Possible optimizations permuted multiplier components

Chapter 7

Conclusion

This thesis work presents a hardware architecture for protection against fault injection attacks, focusing on the ALU and the multiplier of the CVA6 core. In designing a redundant detection system, with the logic already duplicated, the main objective was to minimize the cost in terms of area and to maximize performances. The implemented architectures appear to be potentially compatible with any core, having provided the right balance between critical path and area, where future developments could focus on timing optimization or on the introduction of new algorithms that better match the permutation function used.
Taking into account the target features, the surface occupied by the CVA6 core(≅50k LUT) and the target frequency(≅50MHz), we have chosen to integrate only the permuted ALU leaving the set of multiplication operations uncovered as shown in Table 7.1.
A further research direction in this way, as previously mentioned, involves the design of new implementations for multiplication, with the goal of finding a compromise between latency and area, unlike the two approaches studied so far which are at the edge of performances. In addition, a specific architecture for the division operation, currently not covered, could be integrated.
Therefore to preserve data integrity, considerable effort will be required to develop countermeasures with negligible overhead to ensure total security of systems against injections.

Component	Integrated Y/n	Implementated Y/n
partial permuted ALU	YES	YES
permuted iterative multiplier	NO	YES
permuted Karatsuba multiplie	NO	YES

Table 7.1: Final integration choices for the CVA6 core

Bibliography

[1] José Manuel Martín-Valencia, Hipólito Guzmán-Miranda, and Miguel Ángel Aguirre Echánove. «FPGA-based mimicking of cryptographic device hacking through fault injection attacks». In: *2015 IEEE International Conference on Industrial Technology (ICIT)*. 2015, pp. 1576–1580. DOI: 10.1109/ICIT.2015.7125321 (cit. on p. 1).

[2] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. «Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures». In: *Proceedings of the IEEE* 100.11 (2012), pp. 3056–3076. DOI: 10.1109/JPROC.2012.2188769 (cit. on p. 1).

[3] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. «Fault injection techniques and tools». In: *Computer* 30.4 (1997), pp. 75–82. DOI: 10.1109/2.585157 (cit. on p. 1).

[4] Francisco Eugenio Potestad-Ordóñez, Erica Tena-Sánchez, Antonio José Acosta-Jiménez, Carlos Jesús Jiménez-Fernández, and Ricardo Chaves. «"Hardware Countermeasures Benchmarking against Fault Attacks"». In: *Applied Sciences* 12.5 (2022). DOI: 10.3390/app12052443 (cit. on p. 2).

[5] Olivier Savry Gaëtan Leplus and Lilian Bossuet. «AKHACIA : Arborescent Keyed Homomorphic tAgs for Confidentiality, Integrity and Authenticity of data in CPU pipeline». In: *CEA-LETI, University Grenoble Alpes* (submitted for publication) (cit. on pp. 5, 10).

[6] Olivier Savry Matteo Panigati Massimo Poncino. «Implementing homomorphic security tags in CPU pipeline». In: *Politecnico di Torino, CEA-LETI* (2023). secreted thesis. URL: https://webthesis.biblio.polito.it/29381/ (cit. on pp. 5, 16–18, 23–25, 29–34, 42).

[7] Gaëtan Leplus. «Processeur résistant et résilient aux attaques de fautes et aux attaques par canaux auxiliaires». In: (2023). 2023STET0059. URL: http://www.theses.fr/2023STET0059/document (cit. on p. 5).

[8] Verification Guide. *SystemVerilog tutorial for beginners*. Online. 2024. URL: https://verificationguide.com (cit. on p. 5).

[9] Nima Honarmand. *A Brief Introduction to SystemVerilog*. Slides Online, Stony Brook University. 2015. URL: https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp15/cse502/slides/03-systemverilog.pdf (cit. on p. 5).

[10] Sistemi SystemVerilog. *Il linguaggio SystemVerilog nel progetto di FPGA*. Online. 2009. URL: https://elettronica-plus.it/wp-content/uploads/sites/2/2009/06/20090401010_11.pdf (cit. on p. 5).

[11] David A. Patterson and John L. Hennessy. *Computer organization and design*. USA: Morgan Kauffmann, 2018 (cit. on p. 5).

[12] OpenHW Group. *CVA6: An open-source RISC-V CPU Core*. GitHub Online. 2024. URL: https://github.com/openhwgroup/cva6 (cit. on pp. 5, 7).

[13] Wikipedia contributors. *Omomorfismo*. Online. 2020. URL: https://it.wikipedia.org/wiki/Omomorfismo (cit. on p. 8).

[14] Treccani. *Trasposizione*. Online. 2020. URL: https://www.treccani.it/vocabolario/trasposizione/ (cit. on p. 9).

[15] David A. Patterson and Andrew Waterman. *The RISC-V Reader*. CA,USA: Strawberry Canyon LLC San Francisco, 2017 (cit. on p. 15).

[16] OpenHW Group. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. GitHub Online. 2024. URL: https://github.com/riscv/riscv-isa-manual (cit. on p. 15).

[17] Guido masera. «Part 2_A: Arithmetic circuits». In: *Politecnico di Torino, Slides Integrated Systems Architecture*. 2022 (cit. on pp. 22, 40).

[18] Wikipedia contributors. *Karatsuba algorithm*. Online. 2024. URL: https://en.wikipedia.org/wiki/Karatsuba_algorithm (cit. on p. 44).

[19] Davide Zoni, Andrea Galimberti, and William Fornaciari. «Flexible and Scalable FPGA-Oriented Design of Multipliers for Large Binary Polynomials». In: *IEEE Access* 8 (2020), pp. 75809–75821. DOI: 10.1109/ACCESS.2020.2989423 (cit. on p. 57).