

Hardware Implementations of the WG-16 Stream Cipher with Composite Field Arithmetic

by

Nusa Zidaric

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

© Nusa Zidaric 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The WG stream cipher family consists of stream ciphers based on the Welch-Gong (WG) transformations that are used as a nonlinear filter applied to the output of a linear feedback shift register (LFSR). The aim of this thesis is an exploration of the design space of the WG-16 stream cipher. Five different representations of the field elements were analyzed, namely the polynomial basis representation, the normal basis representation and three isomorphic tower field constructions of $\mathbb{F}_{2^{16}}$: $\mathbb{F}_{((2^2)^2)^2}$, $\mathbb{F}_{(2^4)^4}$ and $\mathbb{F}_{(2^8)^2}$. Each design option begins with an in-depth description of different field constructions and their impact on the top-level WG transformation circuit. Normal basis representation of elements for each level of the tower was chosen for field constructions $\mathbb{F}_{((2^2)^2)^2}$ and $\mathbb{F}_{(2^4)^4}$, and a mixed basis, with polynomial basis for the lower and normal basis for the higher level of the tower for $\mathbb{F}_{(2^8)^2}$. Representation of field elements affects the field arithmetic, which in turn affects the entire design. Targeting high throughput, pipelined architectures were developed, and pipelining was based on the particular field construction: each extension over the prime field offers a new pipelining possibility. Pipelining at a lower level of the tower field reduces the clock period. Most flexible pipelining options are possible for $\mathbb{F}_{((2^2)^2)^2}$, a highly regular construction, which permits an algebraic optimization of the WG transformation resulting in two multiplications being removed. High speed, achieved by adequate pipelining granularity, and smaller area due to removed multipliers deem the $\mathbb{F}_{((2^2)^2)^2}$ to be the most suitable field construction for the implementation of WG-16. The best WG-16 modules achieve a throughput of 222 Mbit/s with 476 slices used on the Xilinx Spartan-6 FPGA device xc6slx9 (using Xilinx Synthesis Tool (XST) for synthesis and ISE for implementation [47]) and a throughput of 529 Mbit/s with area cost of 12215 GEs for ASIC implementation, using the 65 nm CMOS technology (using Synopsys Design Compiler for synthesis [45] and Cadence SoC Encounter to complete the Place-and-Route phase).

Acknowledgements

I would like to thank my supervisors Guang Gong and Mark Aagaard for all the help and support and their endless patience in the past two years. I would also like to thank Xinxin Fan and Yin Tan for tireless explanations. Finally, a quick thanks to Aleksandar Jurisic, for sending me on this incredible journey.



Table of Contents

List of Tables	ix
List of Figures	xii
1 Introduction	1
2 Background, WG-16 stream cipher and related work	6
2.1 Implementation technologies: FPGAs and ASICs	6
2.1.1 Xilinx Spartan-6 FPGA	7
2.1.2 ASIC	8
2.1.3 Implementation efficiency and different metrics	8
2.1.4 FPGA vs. ASIC	10
2.2 Mathematical background	11
2.2.1 Definitions and terminology	11
2.2.2 Irreducible polynomials and field constructions	15
2.2.3 Bases, conjugates and trace function	17
2.3 Stream ciphers	19
2.3.1 General structure	19
2.3.2 A brief discussion on design principles	20
2.4 The WG stream cipher	25
2.4.1 Structure of WG-16	25

2.4.2	Security of the WG	32
2.5	Related work	33
2.5.1	WG hardware implementations	33
2.5.2	3GPP confidentiality and integrity algorithms: Snow3G and ZUC	35
2.5.3	The eSTREAM project: Grain and Trivium	40
2.5.4	Composite field arithmetic	45
3	WGP_T module and different field constructions	48
3.1	Finite field $\mathbb{F}_{2^{16}}$ - overview of field constructions	49
3.2	$\mathbb{F}_{2^{16}}$ with polynomial basis	52
3.2.1	Field construction	52
3.2.2	WGP_T module	52
3.3	$\mathbb{F}_{2^{16}}$ with normal basis	54
3.3.1	Field construction	54
3.3.2	WGP_T module	54
3.4	Tower construction $\mathbb{F}_{((2^2)^2)^2} \cong \mathbb{F}_{2^{16}}$	57
3.4.1	Field construction	57
3.4.2	Conversion matrices	64
3.4.3	Module WGP_T	69
3.5	Tower construction $\mathbb{F}_{(2^4)^4} \cong \mathbb{F}_{2^{16}}$	84
3.5.1	Field construction	84
3.5.2	Conversion matrices	88
3.5.3	Module WGP_T	90
3.6	Tower construction $\mathbb{F}_{(2^8)^2} \cong \mathbb{F}_{2^{16}}$	93
3.6.1	Field construction	94
3.6.2	Conversion matrices	95
3.6.3	Module WGP_T	96
3.7	Finite field $\mathbb{F}_{2^{16}}$ - summary of field constructions	98

4	Implementation	100
4.1	The WG-16 LFSR	103
4.1.1	Multiplication with ω^{2743}	104
4.1.2	Serial vs. parallel loading phase	105
4.2	$\mathbb{F}_{2^{16}}$ with polynomial basis - implementation	107
4.2.1	Analysis of Basic Building Blocks	107
4.2.2	Module WGP_T using polynomial basis	110
4.3	$\mathbb{F}_{2^{16}}$ with normal basis - implementation	113
4.3.1	Analysis of Basic Building Blocks	113
4.3.2	Module WGP_T using normal basis	116
4.4	Tower construction $\mathbb{F}_{((2^2)^2)^2} \cong \mathbb{F}_{2^{16}}$ - implementation	117
4.4.1	Analysis of Basic Building Blocks	117
4.4.2	Initial Design of Pipelined Architecture	130
4.4.3	Optimizations and final choice for module WGP_T	141
4.4.4	The FSM	147
4.4.5	The WG-16 module	155
4.5	Tower construction $\mathbb{F}_{(2^4)^4} \cong \mathbb{F}_{2^{16}}$ - implementation	157
4.5.1	Analysis of Basic Building Blocks	157
4.5.2	Module WGP_T - Design of Pipelined Architecture	166
4.6	Tower construction $\mathbb{F}_{(2^8)^2} \cong \mathbb{F}_{2^{16}}$ - implementation	168
4.6.1	Analysis of Basic Building Blocks	168
4.6.2	Module WGP_T - Design of Pipelined Architecture	174
4.7	Summary of implementations	176
4.7.1	The WGP_T_PB and the WGP_T_NB	178
4.7.2	The WGP_T_A16_2_BC8 and WGP_T_A8_2_BC8	178
4.7.3	The WGP_T_A4_2_BC4 and WGP_T_M4_I4_T2	179
4.7.4	The WGP_T_M8_I8_T3	179
4.7.5	Optimality analysis	180
4.7.6	The LFSR and the FSM	180
4.7.7	The WG-16	181

5 Conclusion and future work	183
Appendix	186
A Xilinx Spartan-6 FPGA	186
A.1 Basic structure	186
A.1.1 CLB - Configurable Logic Block	187
A.1.2 IOB - Input/Output Block	190
A.1.3 Interconnects	190
A.2 FPGA design flow	191
A.2.1 Levels of abstraction	191
A.2.2 Design flow	193
B More detailed discussions and additional material on field constructions and module WGP_T	195
B.1 Tower construction $\mathbb{F}_{2^{16}} \cong \mathbb{F}_{((2^2)^2)^2}$	195
B.1.1 Extension field $\mathbb{F}_{2^4} \cong \mathbb{F}_{(2^2)^2}$	195
B.1.2 Efficient conversion matrices between normal basis and tower field representation of $\mathbb{F}_{((2^2)^2)^2}$	197
B.2 Tower construction $\mathbb{F}_{2^{16}} \cong \mathbb{F}_{(2^4)^4}$	201
B.2.1 Different representations of the finite field with 16 elements and corresponding transition matrices	201
C Xilinx specific optimization for the serial LFSR	203
D Extended Euclidean Algorithm for inversion in polynomial basis	207
E Detailed gate count	215
Bibliography	218

List of Tables

2.1	Resources available on a Xilinx Spartan-6 FPGA xc6s1x9-csg324	7
2.2	Three phases of the WG-16 operation	30
2.3	Implementation results for Snow3G and ZUC found in literature	39
2.4	Implementation results for Grain and Trivium found in literature	44
3.1	Tower construction of $\mathbb{F}_{((2^2)^2)^2}$	57
3.2	Elements of \mathbb{F}_{2^2}	59
3.3	Addition in \mathbb{F}_{2^2}	59
3.4	Multiplication in \mathbb{F}_{2^2}	59
3.5	Addition in \mathbb{F}_{2^2}	60
3.6	Multiplication in \mathbb{F}_{2^2}	60
3.7	Values of $f(x) = x^2 + x + \alpha$ for elements of \mathbb{F}_{2^2}	61
3.8	Elements of $\mathbb{F}_{(2^2)^2}$	62
3.9	Candidates for irreducible polynomials $g(x) = x^2 + x + \lambda_i$ of degree 2 over $\mathbb{F}_{(2^2)^2}$	63
3.10	Elements $\mu^{2^i} \in \mathbb{F}_{(2^2)^2}$	71
3.11	Allocation table for the reused blocks	82
3.12	Tower construction of $\mathbb{F}_{(2^4)^4}$	84
3.13	Candidates for irreducible polynomials of degree 4 over \mathbb{F}_2	86
3.14	Elements of the finite field of order 16	87
3.15	Candidates for irreducible polynomials of degree 4 over \mathbb{F}_{2^4}	88
3.16	Tower construction of $\mathbb{F}_{(2^8)^2}$	94

4.1	The LFSR module - values of the control signals	104
4.2	The LFSR module - implementation results	104
4.3	The LFSR module compared with <code>parallelLFSR</code>	107
4.4	Basic building blocks for polynomial basis arithmetic - implementation results	110
4.5	Polynomial basis inversion I_{16} and <code>WGP_T</code> module <code>WGP_T_PB</code> - implementation results	112
4.6	Normal basis multiplier - generation of vectors v_i for $i = 1, \dots, 8$	115
4.7	Normal basis multiplier - implementation results	116
4.8	The <code>WGP_T</code> module <code>WGP_T_PB</code> - implementation results	117
4.9	Basis transition and exponentiation in $\mathbb{F}_{((2^2)^2)^2}$ - implementation results	128
4.10	Gate count: area and time complexities of building blocks in $\mathbb{F}_{((2^2)^2)^2}$	129
4.11	Basic building blocks for arithmetic in tower field $\mathbb{F}_{((2^2)^2)^2}$ - implementation results	129
4.12	Path $X \rightarrow X^d$ - implementation results	132
4.13	Path $X \rightarrow Y^{-1}$ - implementation results	134
4.14	Module <code>moduleA</code> - implementation results	136
4.15	Module <code>moduleB</code> - implementation results	139
4.16	Module <code>moduleC</code> in two versions - implementation results	140
4.17	The first <code>WGP_T</code> implementation	141
4.18	Optimized <code>moduleA</code> - implementation results	143
4.19	Module <code>moduleBC</code> pipelined at different levels of the tower $\mathbb{F}_{((2^2)^2)^2}$ - implementation results	146
4.20	Module <code>WGP_T</code> , pipelined at different levels of the tower $\mathbb{F}_{((2^2)^2)^2}$ - implementation results	147
4.21	Passing the same value three times	149
4.22	Six states for the WG-16 operation - values of the control signals	151
4.23	Module <code>FSM</code> with different parameters P , S and T - implementation results	152
4.24	Module <code>WG_A16_BC8</code> - behavior in initialization phase	154
4.25	Top module WG-16, pipelined at different levels of the tower $\mathbb{F}_{((2^2)^2)^2}$ - implementation results	155

4.26	Top module WG-16, pipelined at different levels of the tower $\mathbb{F}_{((2^2)^2)^2}$ - pipeline length and initialization phase	156
4.27	Comparison of M_4 blocks using different tower constructions	159
4.28	Basic building blocks for arithmetic in tower field $\mathbb{F}_{(2^4)^4}$ - implementation results	165
4.29	Module WGP_T_M4_I4_T2 using tower construction $\mathbb{F}_{(2^4)^4}$ pipelined at M_4/I_4 level - implementation results	166
4.30	Multipliers M_8 , $M_8 d$ and $M_8 b$ - implementation results	171
4.31	Basic building blocks for arithmetic in \mathbb{F}_{2^8} - implementation results	172
4.32	Basic building blocks for arithmetic in tower field $\mathbb{F}_{(2^8)^2}$ - implementation results	174
4.33	Module WGP_T_M8_I8_T3 using tower construction $\mathbb{F}_{(2^8)^2}$ pipelined at M_8/I_8 level - implementation results	175
4.34	Summary of WGP_T modules for all five field constructions	177
B.1	Elements of \mathbb{F}_{2^4} in polynomial basis $\{1, y, y^2, y^3\}$ and as powers of y	196
B.2	Different representations of \mathbb{F}_{2^4} over \mathbb{F}_2 viewed as vector spaces of dimension 4	201
C.1	Module LFSR - register count	204
D.1	EEA inversion in polynomial basis - implementation results	213
E.1	Area and time complexities of building blocks in Section 4.3 in terms of NAND gates	216
E.2	Area and time complexities of building blocks in Section 4.4.1 in terms of NAND gates	217
E.3	Area and time complexities of building blocks in Section 4.5.1 in terms of NAND gates	217

List of Figures

2.1	Behavioral model of a stream cipher: (a) encryption and (b) decryption	20
2.2	Structural model of a nonlinear filter generator	21
2.3	The WG-16 LFSR	26
2.4	Architecture of WG-16 stream cipher	30
2.5	Three phases of the WG-16 operation	31
2.6	Contents of the LFSR after the loading	32
2.7	The structure of Snow3G stream cipher	36
2.8	The structure of ZUC stream cipher	38
2.9	The structure of Grain stream cipher	41
2.10	The structure of Trivium stream cipher	42
3.1	Architecture of module <code>WGP_T</code>	49
3.2	Finite field $\mathbb{F}_{2^{16}}$ - possible tower constructions	51
3.3	Module <code>WGP_T</code> for field elements in polynomial basis representation	53
3.4	Module <code>WGP_T</code> for field elements in normal basis representation	55
3.5	Conversion between normal basis and tower field representation	65
3.6	A tree structure for the element $A = \sum_{j=0}^{15} \bar{a}_j t_j$ in the tower construction $\mathbb{F}_{((2^2)^2)^2}$	66
3.7	Transitivity of trace function in $\mathbb{F}_{((2^2)^2)^2}$	76
3.8	Data-dependency graph for <code>WGT-16</code> (X^d) computation	79
3.9	Dataflow diagram for <code>WGP-16</code> (X^d) computation	81

3.10	Module <code>WGP_T</code> with multiplier reuse using tower field $\mathbb{F}_{((2^2)^2)^2}$	83
3.11	Module <code>WGP_T</code> using tower field construction $\mathbb{F}_{(2^4)^4}$	90
3.12	Module <code>WGP_T</code> with multiplier reuse using tower field $\mathbb{F}_{((2^2)^2)^2}$	98
3.13	Module <code>WGP_T</code> for all other field constructions	99
4.1	Chapters 3 and 4 - roadmap	102
4.2	The <code>LFSR</code> module, connected to module <code>WGP_T</code>	104
4.3	The <code>parallelLFSR</code> module - parallel key/IV loading [15]	106
4.4	Inversion submodule I_{16} for inversion in polynomial basis	112
4.5	Module <code>WGP_T_PB</code> - pipelined architecture for module <code>WGP_T</code> in polynomial basis	112
4.6	Normal basis multiplier M_{16} - computation of coefficient $x_{j,i}$ with $k = (i + j)$ mod m in block M_{16} in $\mathbb{F}_{2^{16}}$	114
4.7	Squaring and inversion block S_2 in \mathbb{F}_{2^2}	118
4.8	Straightforward multiplication $(a_0, a_1)(b_0, b_1) = (c'_0, c'_1)$ from equation 4.6 .	119
4.9	More efficient multiplication $(a_0, a_1)(b_0, b_1) = (c_0, c_1)$ from equation 4.7 . .	119
4.10	Multiplication by α and α^2 in \mathbb{F}_{2^2}	119
4.11	Squaring and multiplication in $\mathbb{F}_{(2^2)^2}$	120
4.12	Inversion block I_4 in $\mathbb{F}_{(2^2)^2}$	122
4.13	Block M_λ	122
4.14	Block M_{λ^2}	122
4.15	Block M_β	122
4.16	Block $M_{\alpha\beta}$	122
4.17	Multiplication, squaring, and inversion in $\mathbb{F}_{((2^2)^2)^2}$	124
4.18	Block M_μ	124
4.19	Squaring and multiplication in $\mathbb{F}_{((2^2)^2)^2}$	125
4.20	Inversion block I_{16} in $\mathbb{F}_{((2^2)^2)^2}$	126
4.21	Multiplication, squaring, and inversion with: $M_\sigma = M_\alpha$ for $n = 4$ in $\mathbb{F}_{(2^2)^2}$, $M_\sigma = M_\lambda$ for $n = 8$ in $\mathbb{F}_{((2^2)^2)^2}$ and $M_\sigma = M_\mu$ for $n = 16$ in $\mathbb{F}_{((2^2)^2)^2}$. . .	127

4.22	Basis transition and exponentiation	127
4.23	Path $X \rightarrow X^d$ and different levels of pipelining	132
4.24	Module <code>path2_M8_I8</code> - path $X \rightarrow Y^{-1}$ pipelined at M_8/I_8 level	133
4.25	First decomposition of the <code>WGP_T</code> circuit into submodules <code>moduleA</code> , <code>moduleB</code> and <code>moduleC</code>	135
4.26	Modular view of submodules <code>moduleA</code> , <code>moduleB</code> and <code>moduleC</code> and connect- ing signals	135
4.27	Module <code>moduleA</code> - pipelined at M_{16}/I_8 level	135
4.28	Module <code>moduleB</code> - splitting into two pipeline stages (dashed line)	137
4.29	Module <code>moduleB</code>	138
4.30	XORing the 16 bits for the trace computation	139
4.31	Module <code>moduleC</code> with two different insterstage register placings	140
4.32	Module <code>moduleA</code> - pipelined at M_{16}/I_8 level	142
4.33	Module <code>moduleBC</code> - merging <code>moduleB</code> and <code>moduleC</code>	143
4.34	Module <code>moduleBC8</code> with two pipeline stages and with grey vertical line in- dicating the old pipleine stage border	145
4.35	The WG-16: modules <code>LFSR</code> , <code>WGP_T</code> and <code>FSM</code> connected	150
4.36	Six states for the WG-16 operation - the state transition diagram	152
4.37	Block M_4 in \mathbb{F}_{2^4} - computation of coefficient c_i	158
4.38	Inversion block I_4 in \mathbb{F}_{2^4}	159
4.39	Inversion block I_4 in \mathbb{F}_{2^4} - computation of coefficient i_i	161
4.40	Block M_{16} in $\mathbb{F}_{(2^4)^4}$ - component $conv4(s0, s1)$	162
4.41	Multiplication block M_{16} in $\mathbb{F}_{(2^4)^4}$	164
4.42	Inversion block I_{16} in $\mathbb{F}_{(2^4)^4}$	164
4.43	Module <code>WGP_T_M4_I4_T2</code> - pipelined architecture for module <code>WGP_T</code> using tower field construction $\mathbb{F}_{(2^4)^4}$	167
4.44	Module <code>WGP_T_M8_I8_T3</code> - pipelined architecture for module <code>WGP_T</code> using tower field construction $\mathbb{F}_{(2^8)^2}$	167
4.45	Multiplication block M_{16} in $\mathbb{F}_{(2^8)^2}$	173

4.46	Inversion block I_{16} in $\mathbb{F}_{(2^8)^2}$	173
A.1	Basic structure of an FPGA: CLBs - the large grey blocks, IOBs - smaller white blocks, vertical and horizontal interconnects	186
A.2	Arrangement of slices within the CLB [60]	186
A.3	Diagram of SLICEX [60]	188
A.4	Realization of 7 or 8-input Boolean functions using multiple slice LUTs . . .	189
A.5	Interconnect types [60]	191
A.6	Interconnects: (a) switchbox; (b) different connections between CLBs . . .	191
A.7	Levels of abstraction	192
A.8	Design flow	192
C.1	Module LFSR - Xilinx-ISE technology map view of SRL's	206
D.1	EEA inversion in polynomial basis - schematic for <code>new_r1</code>	212
E.1	Area and delay of NOT, AND, OR and XOR gates in terms of NAND gates . . .	216

Chapter 1

Introduction

Over the past decades, society has come to recognize the importance of communication security, and security solutions are now applied in many different areas. Cryptography offers a variety of primitives that are used to construct mechanisms to address different security objectives. Communication security is a wide area and we focus on the confidentiality aspect of information security. Confidentiality is achieved by means of encryption and decryption, and the cryptographic tool used to encrypt/decrypt a message is called a cipher. We further narrow down this area to symmetric-key ciphers, which can be divided into two groups: the block ciphers and the stream ciphers, and focus on the latter. As the name suggests, block ciphers encrypt the message block-by-block, whereby the term block refers to a fixed number of message bits. Stream ciphers on the other hand encrypt the message character by character, where character often refers to one bit, or maybe a 32-bit word.

Let us take a look at some applications that motivate the design of stream ciphers. We are surrounded by various devices and gadgets, such as cell phones, tablets and e-readers with wireless support, etc. or the less-noticeable resource-constrained devices, such as Radio-Frequency Identification (RFID) tags or sensor networks, and much communication is conducted over a wireless link. The wireless channel is more prone to transmission errors and no error-propagation is one of the strengths of stream ciphers: a single bit error affects the decryption of the entire block when a block cipher is used, but when a stream cipher is used, only the character in question is affected. In stream ciphers, the encryption itself is a simple modulo-2 addition of the message character and the keystream character, which is a simple and very fast operation. However the security of the stream cipher depends on the randomness properties of the keystream and the efficiency of the cipher

depends on the efficiency of the keystream generator. Hardware implementations are well suited for applications demanding high speed and high throughput solutions; examples of streams cipher falling into this category are Snow-3G and ZUC, which are used in the security architecture of the cellular 4G-LTE system [14, 16]. The second notable group of applications are the aforementioned hardware applications with restricted resources such as limited storage, gate count, or power consumption. The security concerns for resource constrained devices are addressed in the form of lightweight cryptography (both block and stream cipher, as well as some hybrid solutions exist). In 2004, a call for new stream ciphers appeared: the eSTREAM project [25], which targeted two specific groups, one of them being stream ciphers for hardware applications with highly restricted resources, denoted Profile 2. Examples of Profile 2 candidates that were included in the eSTREAM portfolio are the stream ciphers Grain and Trivium [23, 22].

A member of the Welch-Gong (WG) stream cipher family, WG-29 [3], entered the eSTREAM competition and proceeded to the Phase 2. Later on, lightweight variants WG-5 [9], WG-7 [10] and WG-8 [11] were proposed, to be used in RFID tags. The WG stream cipher family consists of stream ciphers based on the Welch-Gong (WG) transformation that is used as a nonlinear filter applied to the elements of an maximum-length sequence generated by an linear feedback shift register (LFSR). WG stream ciphers generate keystreams with mathematically proven randomness properties, such as long period, balance, ideal two-level autocorrelation etc. The WG-16 stream cipher, intended to be used in 4G-LTE networks, inherits these randomness properties, and is able to withstand the known attacks against the stream ciphers [8]. Cellular systems have high demands for speed and throughput and hardware solutions are more efficient in such environments. In this work, we present different hardware implementations of the WG-16 stream cipher.

The WG stream ciphers are composed of three components: the LFSR, the WG transformation and the finite state machine (FSM) controlling its operation. Both the LFSR and the WG transformation are defined over the finite field $\mathbb{F}_{2^{16}}$. The implementation of the LFSR is quite straightforward, but the WG transformation is more complex and is the critical component in the WG stream cipher, so we focus on the implementation of the WG transformation. We will call this component the `WGP_T` for the rest of this work. The `WGP_T` involves several exponentiations to powers of two, multiplications and an inversion of the $\mathbb{F}_{2^{16}}$ field elements. Although the finite field $\mathbb{F}_{2^{16}}$ is considered to be small, the aforementioned operations, especially inversion, are quite time and area consuming.

Generally speaking, there are several approaches to optimization of a hardware implemen-

tation. For example area reductions can be achieved by reusing resources, pipelining at different levels of granularity, etc. High throughput comes at the cost of area increase, for example exploiting the maximum level of parallelism, pipelining at a finer granularity and so on, and we must find the best trade-off between the area and the throughput. Algebraic optimizations also reduce the number of resources needed and are in some cases performed by the synthesis tools.

Another level of optimization begins by choosing the appropriate architecture for the design. For WG-16 we choose different field constructions and different representations of field elements, and explore the effects of our choices on the $\mathbb{F}_{2^{16}}$ arithmetic and the WGP_T. Polynomial bases and normal bases are quite common for finite field implementations. Both have their own advantages, for example exponentiation to powers of two is very simple when normal bases are used, but for $\mathbb{F}_{2^{16}}$, optimal normal bases, which would give the best architecture, do not exist. Motivated by research on the AES S-boxes, which require \mathbb{F}_{2^8} arithmetic: possible field constructions of \mathbb{F}_{2^8} were thoroughly explored, including the tower field $\mathbb{F}_{((2^2)^2)^2}$ with different bases for each level of the tower. We decide to explore the isomorphic tower field constructions of $\mathbb{F}_{2^{16}}$ and analyze five different representations of the field elements: the polynomial basis representation, the normal basis representation and three different tower field representations, namely $\mathbb{F}_{(((2^2)^2)^2)^2}$, $\mathbb{F}_{(2^4)^4}$ and $\mathbb{F}_{(2^8)^2}$. For each representation we first conduct an analysis of the basic building blocks, that is the field operations needed for the WGP_T. Inverters and multipliers can serve as good indicators of overall system area cost and delay. However, since we decide to base the pipelining granularity on the “granularity” of the tower field construction used, we must compare the blocks at the same level of the tower field. For example the normal basis implementation will include atomic multipliers working with 16-bit operands within a pipeline stage, while the $\mathbb{F}_{(2^4)^4}$ will include a multiplier working with 4-bit operands. We will denote the level of the pipelining with the width of the operands and letter M for multiplier and I for inverter; that would be M_{16} level and M_4 level of pipelining for the previous example. Furthermore, the aforementioned algebraic optimizations performed by synthesis tools result in discrepancies between the theoretical gate count and delay and the implementation results, thus providing another motivation for an actual implementation of the modules.

Let us briefly summarize the implementation results. The polynomial and normal basis WGP_T modules were implemented to provide a frame of reference for the tower field implementations. For field constructions $\mathbb{F}_{(((2^2)^2)^2)^2}$ and $\mathbb{F}_{(2^4)^4}$, we chose to use the normal basis representation of elements for each level of the tower. For the construction $\mathbb{F}_{(2^8)^2}$ a mixed basis was chosen: using polynomial basis representation and table look-up algorithms for

arithmetic operations at the lower level of the tower \mathbb{F}_{2^8} , and normal basis representation of elements at the top level of the tower. Due to the large number of look-up tables and a relatively large table size, this implementation produced the biggest **WGP_T** module with the longest clock period. The tower construction $\mathbb{F}_{(2^4)^4}$ permitted a single reasonable pipelining option for the **WGP_T** module, namely the pipelining at the M_4/I_4 level. In terms of speed, this module was a top candidate, but in terms of area cost it is very close to the polynomial basis **WGP_T** module. It also exhibited a highly regular structure that allowed many algebraic optimizations. The tower construction $\mathbb{F}_{((2^2)^2)^2}$ is also highly regular, giving very similar basic building blocks, that differ only in the width of the operands and gates, at each level of the tower. Different levels of pipelining are facilitated by $\mathbb{F}_{((2^2)^2)^2}$: the M_{16}/I_8 level, the M_8/I_8 level and the M_4/I_4 level. Pipelining at a lower level of the tower field reduces the clock period. For the tower construction $\mathbb{F}_{((2^2)^2)^2}$ an algebraic optimization that removes two multiplications was possible: consequently, the $\mathbb{F}_{((2^2)^2)^2}$ based **WGP_T** modules result in the best overall design in terms of performance and area among all the FPGA implementations.

This work is organized into three large parts: Chapter 2 covers the background, the definition of WG-16 and the related work; Chapter 3 gives an in-depth description of different field constructions and their impact on the **WGP_T** circuit; and Chapter 4 presents the implementations of the **WGP_T** circuits obtained in Chapter 3. The background material covered in Chapter 2 includes hardware implementation technologies (Section 2.1), mathematical background (Section 2.2) and stream ciphers (Section 2.3). The description of the WG-16 stream cipher is provided in Section 2.4. The related work covered includes the stream ciphers currently used in 4G-LTE networks (Section 2.5.2), the eSTREAM project finalists Grain and Trivium (Section 2.5.3), and the implementations using composite field arithmetic (Section 2.5.4). Chapter 3 is theoretical; it begins with an overview of field constructions, continues with a separate Section dedicated to each one of those constructions, and provides an overview at the end. The five different constructions narrow down to two different top-level designs for the **WGP_T** module, one for the $\mathbb{F}_{((2^2)^2)^2}$ and one for all other field constructions. Chapter 4 is the “implementation” Chapter and it closely follows the structure of Chapter 3. It explains the algorithms used for the arithmetic operations, i.e. the basic building blocks, which are then used in the **WGP_T** pipelines. In Chapter 5 we give the summary of results, provide conclusions and briefly discuss future work.

Readers primarily interested in the hardware aspects of this work may wish to focus on Chapter 4, and refer back as necessary to Chapter 3 to understand the five field construc-

tions and to Section 2.2 for the mathematical background. Readers primarily interested in the field constructions may wish to focus on Chapter 3 and Section 4.4, which describes the most optimal hardware implementation. Chapter 3 contains many examples to illustrate the theory of finite fields, that was summarized in Section 2.2. Throughout Chapters 3 and 4, supporting observations and examples appear as comments and may be skipped without loss. These remarks are distinguished by a smaller font size and are enclosed in “■”.

Chapter 2

Background, WG-16 stream cipher and related work

This Chapter begins with three preliminary sections: hardware implementations, mathematical background and stream ciphers. These three sections cover the background material needed for the contents of this thesis. Then the core section follows: the presentation of WG-16 stream cipher. The supplementary literature survey at the end of this chapter is needed to put the entire work into perspective: it covers the stream ciphers currently used in 4G/LTE networks, the eSTREAM project and the implementations using composite field arithmetic.

2.1 Implementation technologies: FPGAs and ASICs

FPGA (Field Programmable Gate Arrays) devices provide a high number of gates (in millions) and built-in high-level system functions, such as embedded processors, clock management systems, memory modules, DSP (digital signal processing) modules, serial transmitters, etc., integrated in a single device [48]. The greatest advantage of SRAM-based FPGAs is their flexibility; modifying the designed and even implemented circuit is fast and easy. Compared to ASIC FPGAs have a big advantage when time-to-market is critical due to a shorter development cycle. Nevertheless, when comparing speed, area and power consumption, an equivalent ASIC circuit is always preferable. But an ASIC solution is also extremely time consuming and expensive. Furthermore, once fabricated it cannot be altered. An extensive comparison of FPGAs and ASICs was performed in [44], and a

brief review of their findings is given at the end of this section. Nowadays, FPGAs can be found almost everywhere: in satellites, airplanes, modems, Mars Rover, face recognition systems, etc.

2.1.1 Xilinx Spartan-6 FPGA

For this thesis, a Xilinx Spartan-6 FPGA was chosen (`xc6s1x9-csg324`). Here we give a short description of Xilinx FPGA's. A more detailed description of FPGA features in terms of Spartan-6 family can be found in Appendix A. From a users point of view, the most important part of an FPGA are the Configurable Logic Blocks (CLBs), that are basic building blocks of the circuit. Each CLB is divided into two slices and each slice contains four Look-up Tables (LUTs), four primary and four secondary 1-bit storage elements and multiplexers to control the routing within the slice. The storage elements can be configured either as D-type flip-flops (DFFs) or latches; since using latches is considered a bad practice we shall only use DFFs, and will refer to storage elements as flip-flops or simply FFs from now on. LUTs are basically just memory arrays that hold the truth table of a Boolean function they implement. The CLBs are organized into a matrix, interwoven with configurable interconnects, and surrounded by special Input/Output Blocks (IOBs). The resources available on the chosen target device `xc6s1x9-csg324` are listed in Table 2.1.

# of Slices	1430
# of LUTs	5720
# slice registers	11440
# of user IOBs	200

Table 2.1: Resources available on a Xilinx Spartan-6 FPGA `xc6s1x9-csg324`

The design flow for FPGAs is described in detail in Appendix A. We used VHDL for design entry, Xilinx Synthesis Tool (XST) for synthesis and ISE for implementation [47]. The designs were verified using ModelSim [46] to run simulations for individual basic building blocks, for `WGP_T` modules and finally for the top-module itself.

2.1.2 ASIC

In this work, the term ASIC refers to Standard-Cell-Based ASIC: the logic components are pre-designed, pre-tested and pre-characterized [43], and finally stored in a library as standard cells. The design flow for ASICs starts with design entry, where the same code that was used for the FPGA implementation can be reused, but the rest of the process is different. Without going into details, let us just say that the CAD tools use the library cells to convert the VHDL design into a chip layout.

For this thesis, the CAD tools for ASIC were run only for the WG-16 designs that have shown the best performance on the FPGA. The results were obtained for the 65nm CMOS technology using Synopsis Design Compiler and Cadence SoC Encounter [45].

2.1.3 Implementation efficiency and different metrics

Performance of FPGA and ASIC implementations is described with three key metrics (dimensions): area, time and power. Other derived metrics are sometimes used, because they make predictions and comparisons between different design options easier.

The primary **time metrics** of a design are latency, clock period (and its reciprocal clock frequency) and total time. These terms apply in the same manner to both, FPGAs and ASICs. *Latency* is the time that elapses from the moment when the input data is available to the moment the results appear on the outputs, that is the delay between the input and the output [66]. In general, latency can refer to a particular module or the FPGA/ASIC itself, if we are talking about the top-module. If an algorithm can be realized with a purely combinational circuit (without storage elements), the time complexity equals to the delay of the signal along the critical path (a path is a sequence of interconnects and logical elements [48]). In sequential circuits, the time complexity is given by two parameters, the clock period, which depends on the critical path, and total time, which is the product of the clock period and the number of clock cycles needed. Because we are targeting for a pipelined design we will be primarily interested in clock period and throughput. The **throughput** measures the amount of data processed per unit of time, mostly given in bits per second [bps]. Another similar metric is **data rate** measured in bits-per-cycle, and throughput is computed by multiplying data rate with clock frequency.

The **area complexity** in FPGAs is given in terms of resources used by the design, for example the number of used slices, LUTs, storage elements, IOBs, etc. Data about resources available on our target device `xc6s1x9-csg324` are presented in Table 2.1. Area complexity for ASICs is measured by the amount of silicon used and can be given either

in μm^2 or in Gate Equivalents (GE). The latter is the area in μm^2 divided by the area of a two-input NAND gate.

Power is another metric in hardware performance evaluations, and its importance is becoming more and more significant for various reasons: it affects battery life, can force us to limit the clock frequency, causes higher temperatures which in turn reduces the lifetime of the device, increases dissipated heat of hand-held devices etc. In general, total power consumption depends on the number of logic cells in the circuit, on connections between them, on the underlying technology being used and finally on data that is being processed. In CMOS circuits, the total power consumption has two components: static power and dynamic power. Dynamic power is proportional to how often the signals change their value and on clock frequency. It is attributed to the evaluation of logic cell outputs and depends on two factors, the load capacitance of the cell that needs to be charged and the short circuit current occurring when the output of a cell is switched. The static power is caused by leakage currents and increases with decreasing size of transistors. It is roughly proportional to the area [31].

Note that the above is a very simplified description. In reality, power consumption depends on many factors in a complicated way: often changing one parameter that would make an improvement for example to dynamic component, would increase the static component of power consumption.

Since it is difficult to compare two designs based on more than one metric (for example the clock period and the area), we use the so called derived metrics, for example the *time-area product* or with the power consumption being more and more important, the *time-area-power product*. These two metrics are, just like the clock period and area, “the smaller the better”. However, it is more natural for us to look for the opposite, the “bigger number”, which is also one of the reasons why frequency is often preferred to clock period. Taking the reciprocal of these two products and keeping throughput in mind, we come up with another set of commonly used metrics, namely the *throughput per area ratio* $\frac{T}{A}$ and *throughput per product of area and power* $\frac{T}{AP}$. Because power analysis is tedious we often approximate it with area, thus obtaining the $\frac{T}{A^2}$. The $\frac{T}{A^2}$ ratio is also preferred to the $\frac{T}{AP}$, because of sensitivity of power analysis to differences between the cell libraries and to tool configurations [9]. There is yet another viewpoint to these metrics, namely the fact that high throughput comes at the cost of area increase, for example exploiting maximum level of parallelism or unrolling an iterative implementation into a pipeline [67], or by increasing the frequency, which in turn causes increased area and power consumption. Metrics like $\frac{T}{A}$ and $\frac{T}{A^2}$ put a better perspective on the actual improvement of the design

by some optimization attempt; they emphasize the trade-offs between the throughput and area.

In this thesis, we will report the area cost by listing the number of flip-flops, number of LUTs, and number of slices used by the design and will give the time parameters in terms of clock period for registered modules and block delay for the combinational modules implemented on the FPGA. The total numbers of resources available on the chosen target device `xc6s1x9-csg324` are listed in Table 2.1. We use the $\frac{T}{A^2}$ metric when the benefits of certain design options are not immediately clear. In such cases it is also beneficial to obtain the ASIC results as well. For the best FPGA design we also provide ASIC results obtained for the 65nm CMOS technology, in terms of gate equivalents for the area and clock period or block delay for the time dimension.

2.1.4 FPGA vs. ASIC

As already mentioned, compared to ASIC, the area is always larger when the same design is implemented on an FPGA. Comparing the performance of a 90-nm CMOS FPGA and 90-nm CMOS standard-cell ASIC using implementations of carefully designed benchmarks, [44] reports that the area complexity when implemented on an FPGA is on average approximately 35 times larger in comparison with the ASIC implementation, when comparing circuits that use logic only (that is only LUTs and interconnects), and that for other circuits the gap in area complexity can be reduced when using dedicated blocks in FPGAs (listing the use of multiply-accumulate logic in special DSP slices available on some FPGAs). The same author also directs attention towards two facts about FPGAs: first, they come in fixed discrete sizes and second, if only one resource within a cell is utilized, the cell is counted as used (also pointing out that the CAD tools give less effort to optimizations when the design is small relative to the device on which it is implemented). But, they also find that FPGAs are better equipped to handle larger designs, and that with same designs on an ASIC, area overhead occurs in order to maintain speed and signal integrity for longer connections.

In [44], ASIC implementations were compared to implementations on the fastest and slowest FPGA speed grades. For the fastest speed grade FPGAs, they found the FPGA implementations to be on the average 3.4 times slower for logic only designs. Again, the factor could be slightly reduced when using dedicated blocks on FPGAs, but only if there are enough dedicated blocks available. Another general observation made by the authors of [44] is that efficient usage of dedicated blocks in FPGAs reduces dynamic power consumption (due to smaller area and less interconnects).

In general, a huge percentage of an FPGA device is used to provide the programmability. Furthermore, since the general FPGA structure is fixed, there are always unused CLBs left over. Sometimes they even cannot be used because they end up isolated; there are just not enough routing resources available to reach them. In general, interconnect switching in FPGAs is slow, programmable routing takes up a lot of area and these interconnects have higher capacitance hence higher power consumption. Due to [64], some 40%-80% of overall design delay, 90% of area and up to 80% of total power dissipation are attributed to interconnects. Another problem due to the fixed interconnects: a signal path in an equivalent ASIC circuit could be much shorter, hence lower delay.

In the FPGA world, there are two layers to be taken into account: the high-level architecture of the design and the FPGA itself, fixed in structure. The latter problem is addressed through CAD tools provided by FPGA vendors, and the user has only little influence on how the resources are actually used.

2.2 Mathematical background

In the following section we will cover basic definitions and properties of finite fields, extension fields and their defining polynomials. We will introduce polynomial and normal bases and conclude the Section with the notion of trace function. Extensive literature on the subject exists, for example [69] or [70]. Further properties of finite fields will be presented in the remaining text when needed. Numerous examples illustrating the theory presented in this section will be encountered in Chapter 3.

2.2.1 Definitions and terminology

Unfortunately, we cannot begin this discussion without briefly introducing the notion of a group and a ring. We proceed with definition of a field, [73].

Definition 2.1 *A nonempty set G , together with a binary operation $\circ : \mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$, constitutes a group $\mathcal{G} = (G, \circ)$, if*

- i. for $\forall a, b \in \mathcal{G} : a \circ b \in \mathcal{G}$ (\mathcal{G} is closed under \circ),*
- ii. for $\forall a, b, c \in \mathcal{G} : (a \circ b) \circ c = a \circ (b \circ c)$ (associativity),*

iii. there \exists an element $e \in \mathcal{G} \ni: \forall g \in \mathcal{G} : e \circ g = g \circ e = g$ (identity),

iv. for $\forall g \in \mathcal{G}$ there $\exists f \in \mathcal{G} \ni: g \circ f = f \circ g = e$ (inverse).

\mathcal{G} is called commutative or Abelian group if its operation is commutative, i.e. for $\forall a, b \in \mathcal{G}: a \circ b = b \circ a$.

If the underlying set G is finite, then \mathcal{G} is a *finite* group, otherwise \mathcal{G} is infinite. The *order of group* \mathcal{G} is the number of elements in G , denoted $|G|$. The *order of element* $g \in \mathcal{G}$ is the smallest positive integer r such that $\underbrace{g \circ g \circ \dots \circ g}_r = e$; it is denoted $ord(g) = r$. The order of an element must divide the order of the group, i.e. for $\forall g \in \mathcal{G}: ord(g) \mid |G|$.

Definition 2.2 A (multiplicative) group \mathcal{G} is cyclic, if there exists an element $g \in \mathcal{G}$ such that for any $a \in \mathcal{G}$, there exists an integer i for which $a = g^i$.

The element g is called *generator* of the cyclic group and we write $G = \langle g \rangle = \{g^i; i \in \mathbb{Z}\}$. If \mathcal{G} is a finite group of order n , then $\langle g \rangle = \{e, g, g^2, \dots, g^{n-1}\}$ and $ord(g) = n$.

Definition 2.3 A ring $\mathcal{R} = (R, +, *)$ is a set R , together with two binary operations $+$ (addition) and $*$ (multiplication) on R , satisfying the following properties:

i. $(R, +)$ is a commutative group with additive identity denoted 0 ,

ii. operation $*$ is associative: $a * (b * c) = (a * b) * c$ for $\forall a, b, c \in R$

iii. multiplication is distributive over addition: $a * (b + c) = a * b + a * c$ and $(b + c) * a = b * a + c * a$ for $\forall a, b, c \in R$

Definition 2.4 A nonempty set F , together with two binary operations addition “ $+$ ” and multiplication “ $*$ ” is a field $\mathcal{F} = (F, +, *)$, if

i. $(F, +)$ is a commutative group with (additive) identity 0

ii. $(F \setminus \{0\}, *)$ is a commutative group with identity 1

iii. multiplication is distributive over addition: $a * (b + c) = a * b + a * c$ for $\forall a, b, c \in F$

If the underlying set F has a finite number of elements, then \mathcal{F} is a *finite field*. We will use the notation \mathbb{F}_q to denote a finite field with q elements. The *order* of a finite field is the number of elements in the field.

Very important fact, also referred to as generalization of Fermats little theorem ([76, 69]), states that every element α of a finite field of order q satisfies the identity:

$$\alpha^q = \alpha \tag{2.1}$$

Let $\mathcal{F} = (F, +, *)$ be a field. A subset $K \subseteq F$ together with operations $+$ and $*$ forms a *subfield* $\mathcal{K} = (K, +, *)$ of \mathcal{F} , if (K) is itself a field with respect to the two operations. We also refer to \mathcal{F} as an *extension field* of \mathcal{K} , denoted \mathcal{F}/\mathcal{K} . If $K \neq F$ and $K \neq \{0\}$, then \mathcal{K} is a *proper* subfield of \mathcal{F} . A field that has no proper subfields is called a *prime field*. The smallest subfield of any field is a prime subfield. We can also consider \mathcal{F} as a vector space over \mathcal{K} . The dimension of \mathcal{F} over \mathcal{K} is called *degree of extension*, denoted $[F : K]$, [69].

Definition 2.5 Let $\mathcal{F} = (F, +, *)$ be a field. The smallest positive integer p such that $\underbrace{1 + 1 + \dots + 1}_p = 0$ is called the characteristic of \mathcal{F} , denoted $\text{char}(\mathcal{F}) = p$. If such an integer does not exist, we say the field has characteristic 0, [69, 72, 76].

If $\text{char}(\mathcal{F}) > 0$, then it is a prime number. The characteristic of a finite field is the order of its prime subfield. It is also the additive order of the multiplicative identity 1. For arbitrary elements $\alpha, \beta \in \mathcal{F}$, where $\text{char}(\mathcal{F}) = p$, the following holds for any $k \geq 1$:

$$(\alpha + \beta)^{p^k} = \alpha^{p^k} + \beta^{p^k}. \tag{2.2}$$

We now summarize some facts about finite fields.

The theorem about the existence and uniqueness of finite fields states, that for every prime number p and each positive integer n , there exists a finite field with $q = p^n$ elements. Furthermore, this field is unique up to field isomorphism (see [69, 71]).

The finite field \mathbb{F}_q has order $q = p^m$, where $p = \text{char}(\mathbb{F}_q)$ and m is the degree of extension of \mathbb{F}_q over its prime subfield \mathbb{F}_p , i.e. $[\mathbb{F}_{p^m} : \mathbb{F}_p] = m$. Any set $\{a_0, a_1, \dots, a_{m-1}\}$ of m linearly independent elements $a_i \in \mathbb{F}_q$ forms a basis of the vector space \mathbb{F}_q over \mathbb{F}_p . For further details see [69, 70]. Let us now state another important result connected to order of finite field and degree of extension, namely the subfield criterion ([69]).

Theorem 2.1 [Subfield criterion] Let \mathbb{F}_q be a finite field with $q = p^m$ elements. Then every subfield of \mathbb{F}_q has order p^n , where n is a positive divisor of m . Conversely, if n is a positive divisor of m , then there is exactly one subfield of \mathbb{F}_q with p^n elements.

Speaking of subfields of a finite field we adopt a “top-down ” point of view. A “bottom-up” approach reveals the following: for a composite integer $m = n_1 \cdot \dots \cdot n_k$, where $n_i, i = 1 \dots k$ are positive integers (not necessarily primes), we can build \mathbb{F}_{p^m} as a tower of extensions $\mathbb{F}_{(\dots((p^{n_1})^{n_2})\dots)^{n_k}}$ over its prime subfield \mathbb{F}_p . Field constructions will be discussed in detail in the next section and concrete examples will follow in Chapter 3, however we have covered enough basics to explain the notion of a tower field. In general, instead of constructing \mathbb{F}_{p^m} as a single extension of degree m over the prime field \mathbb{F}_p , we proceed in k steps, building an extension of degree n_i at step i , where n_i is a factor in decomposition of m . Using notation $K_0 = \mathbb{F}_p, K_i = \mathbb{F}_{(\dots((p^{n_1})^{n_2})\dots)^{n_i}}$, we proceed as follows:

We construct $K_1 = \mathbb{F}_{p^{n_1}}$ as an extension of degree $[K_1 : K_0] = n_1$ over prime field \mathbb{F}_p , then field $K_2 = \mathbb{F}_{(p^{n_1})^{n_2}}$ as an extension of degree $[K_2 : K_1] = n_2$ over $K_1 = \mathbb{F}_{p^{n_1}}$, and continue the process until we reach the final extension $K_k = \mathbb{F}_{(\dots((p^{n_1})^{n_2})\dots)^{n_k}}$, which is an extension of degree $[K_k : K_{k-1}] = n_k$ over K_{k-1} .

So, in each step $i, i = 1, \dots, k$, we build an extension K_i over K_{i-1} , with degree of extension $[K_i : K_{i-1}] = n_i$. We will call K_{i-1} the *base field* for this extension (to differentiate it from the prime field). The base field K_{i-1} is now embedded in K_i as a subfield, which we will denote with the inclusion symbol, namely $K_{i-1} \subset K_i$. The result of the procedure described above is a sequence of subfields:

$$\mathbb{F}_p = K_0 \subset K_1 \subset \dots \subset K_{k-1} \subset K_k \cong \mathbb{F}_{p^m} \quad (2.3)$$

with their corresponding orders

$$p \leq p^{n_1} \leq p^{n_1 \cdot n_2} \leq \dots \leq p^{n_1 \cdot \dots \cdot n_k} = p^m, \text{ where } m = n_1 \cdot \dots \cdot n_k$$

Note at this point, that the field obtained with each extension is isomorphic to a field whose order is p to the power of partial product of extension degrees up till now, for example $\mathbb{F}_{(p^{n_1})^{n_2}} \cong \mathbb{F}_{p^{n_1 \cdot n_2}}$. Also, since $m = n_1 \cdot \dots \cdot n_k$, the product of extension degrees equals the degree of extension of \mathbb{F}_{p^m} over \mathbb{F}_p , namely $[K_k : K] = [K_k : K_{k-1}] \cdot \dots \cdot [K_2 : K_1] \cdot [K_1 : K_0]$.

We refer to the sequence of fields in expression (2.3) as *tower field* or *composite field*. The notion tower field indicates that the field was obtained as a tower of extensions, and the notion composite field is related to compositness of m ; we use the factors of m to build the tower of extensions.

2.2.2 Irreducible polynomials and field constructions

After stating the basic facts about finite fields, we now move toward field constructions.

Let $\mathcal{K}[x]$ be a set of polynomials $f(x)$ in the indeterminate x with coefficients from field \mathcal{K} :

$$f(x) = \sum_{i=0}^{\infty} a_i x^i, \quad a_i \in \mathcal{K}.$$

Let n be the index of the last nonzero coefficient in f , i.e. $a_n \neq 0$ but $a_j = 0$ for $\forall j > n$. Then a_n is called the *leading coefficient* of f and f is a polynomial of degree n . A polynomial with $a_n = 1$ is called *monic*. The coefficient a_0 is called the constant term, and polynomials that have $a_0 \neq 0$ but $a_j = 0$ for $\forall j > 0$ are called *constant polynomials* and have degree 0. The degree of the zero polynomial (i.e. $a_j = 0$ for $\forall j$) is defined to be $-\infty$.

Let $f, g \in \mathcal{K}[x]$ be two polynomials of degrees n and m respectively:

$$f(x) = \sum_{i=0}^{\infty} a_i x^i \quad \text{and} \quad g(x) = \sum_{i=0}^{\infty} b_i x^i.$$

Their sum is defined as

$$f(x) + g(x) = \sum_{i=0}^{\max\{n,m\}} (a_i + b_i) x^i$$

and their product as

$$f(x) \cdot g(x) = \sum_{k=0}^{m+n} c_k x^k \quad \text{where} \quad c_k = \sum_{i+j=k} a_i b_j$$

The set of polynomials $\mathcal{K}[x]$ over field \mathcal{K} , with addition and multiplication defined as above, is a commutative polynomial ring (see 2.3) with additive identity $f_0(x) = 0$ and multiplicative identity $f_1(x) = 1$, and has no divisors of zero, see [69].

Definition 2.6 A polynomial $f \in \mathcal{K}[x]$ is said to be irreducible over \mathcal{K} , if it has a positive degree and $f = g \cdot h$, for some $g, h \in \mathcal{K}[x]$, implies that either g or h is a constant polynomial.

An element $\alpha \in \mathcal{K}[x]$ is a *root* of a nonzero polynomial $f \in \mathcal{K}[x]$ if $f(\alpha) = 0$, i.e. when x takes on the value α the polynomial f evaluates to 0. Then we can write $f(x) = (x-\alpha) \cdot g(x)$ for some $g \in \mathcal{K}[x]$. A polynomial of degree n will have at most n distinct roots in \mathcal{K} (or any of its extensions), see [69]. The lowest degree monic polynomial in $\mathcal{K}[x]$, having a root α is called the *minimal polynomial* of α , [76].

We now have all the tools required to start a discussion about extension fields. Noting that a polynomial that is irreducible over \mathcal{K} has no roots in \mathcal{K} , we can construct an extension field of \mathcal{K} by adjoining its roots to the base field \mathcal{K} . We will refer to the polynomial whose root was used as the *defining polynomial* of the extension field.

Definition 2.7 *Let \mathcal{F} be an extension of \mathcal{K} . An element $\alpha \in \mathcal{F}$ is algebraic over \mathcal{K} , if there exists a nonzero polynomial $f \in \mathcal{K}[x]$ having α as root: $f(\alpha) = 0$.*

If every element $\alpha \in \mathcal{F}$ is algebraic over \mathcal{K} , then \mathcal{F} is an algebraic extension of \mathcal{K} . The minimal polynomial of an algebraic element $\alpha \in \mathcal{F}$ is irreducible in $\mathcal{K}[x]$. The degree of α is defined to be the degree of its minimal polynomial, [69].

Theorem 2.2 *Let \mathcal{K} be a subfield of \mathcal{F} and $\alpha \in \mathcal{F}$ an algebraic element of degree n over \mathcal{K} . The simple algebraic extension of \mathcal{K} obtained by adjoining α is*

$$\mathcal{K}(\alpha) = \mathcal{K}[\alpha] = \left\{ \sum_{i=0}^{n-1} a_i \alpha^i; a_i \in \mathcal{K} \right\}$$

$\mathcal{K}(\alpha)$ is an extension field of \mathcal{K} with α as its defining element, and can be considered as a n -dimensional vector space over \mathcal{K} (i.e. $[\mathcal{K}(\alpha):\mathcal{K}] = n$) with basis $\{1, \alpha, \alpha^2, \dots, \alpha^{n-1}\}$.

As already mentioned, a polynomial of degree n can have at most n distinct roots. The finite fields obtained by adjoining different distinct roots of an irreducible polynomial over \mathcal{K} are isomorphic. For the finite field \mathbb{F}_q over \mathbb{F}_p , $q = p^n$, the multiplicative group \mathbb{F}_q^* is cyclic. Its generator is an element of order $q - 1$ and is called a *primitive element*. An irreducible polynomial having a primitive element as its root is called a *primitive polynomial*.

2.2.3 Bases, conjugates and trace function

Let $q = p^n$. An element $\alpha \in \mathbb{F}_{q^m}$ generates the *polynomial basis* $\{1, \alpha, \alpha^2, \dots, \alpha^{m-1}\}$ of \mathbb{F}_{q^m} over \mathbb{F}_q if and only if α is a root of an irreducible polynomial $f \in \mathbb{F}_q[x]$ of degree m (i.e. f is the defining polynomial of \mathbb{F}_{q^m} over \mathbb{F}_q). For every finite field \mathbb{F}_q and any positive integer m , an irreducible polynomial in $\mathbb{F}_q[x]$ of degree m always exists. Furthermore, there exists at least one polynomial basis of \mathbb{F}_{q^m} over any of its subfields, see [69]. Elements of \mathbb{F}_{q^m} with defining polynomial $f \in \mathbb{F}_q[x]$ of degree m , can be viewed as polynomials in $\mathbb{F}_q[x]$, reduced modulo f . Each element $A \in \mathbb{F}_{q^m}$ can be represented in polynomial basis as follows:

$$A = \sum_{i=0}^{m-1} a_i \alpha^i; a_i \in \mathbb{F}_q$$

The polynomial f is irreducible over \mathbb{F}_q , but it has a root in \mathbb{F}_{q^m} , say $\alpha \in \mathbb{F}_{q^m}$. Furthermore, it has m distinct simple roots, given by the *conjugates*: $\alpha, \alpha^q, \alpha^{q^2}, \dots, \alpha^{q^{m-1}}$. If the roots of the defining polynomial are linearly independent, they generate the *normal basis* $\{\alpha, \alpha^q, \alpha^{q^2}, \dots, \alpha^{q^{m-1}}\}$ of \mathbb{F}_{q^m} over \mathbb{F}_q . Each element $A \in \mathbb{F}_{q^m}$ can be represented in normal basis as :

$$A = \sum_{i=0}^{m-1} a_i \alpha^{q^i}; a_i \in \mathbb{F}_q$$

For any finite field \mathbb{F}_{q^m} there exists a normal basis over its prime subfield and it consists of primitive elements of \mathbb{F}_{q^m} . Furthermore, there always exists at least one normal basis of \mathbb{F}_{q^m} over any of its subfields, [69]. The element $\alpha \in \mathbb{F}_{q^m}$, that generates a normal basis of \mathbb{F}_{q^m} over \mathbb{F}_q is called a *normal element* and the defining polynomial a *normal polynomial*, i.e. N-polynomial.

Normal bases will be discussed in more detail in Section 3.1

The conjugates of $\alpha \in \mathbb{F}_{q^m}$ with respect to \mathbb{F}_q can be obtained by applying the mappings

$$\sigma_i(\alpha) = \alpha^{q^i}, \quad 0 \leq i \leq m-1$$

to α . For all $i, 0 \leq i \leq m-1$, $\sigma_i : \mathbb{F}_{q^m} \mapsto \mathbb{F}_{q^m}$ is an automorphism. Note that the mappings $\sigma_0, \sigma_1, \dots, \sigma_{m-1}$ are distinct. The automorphism $\sigma_1(\alpha) = \alpha^q$ is called Frobenius automorphism of \mathbb{F}_{q^m} over \mathbb{F}_q , see [69], and for all $i, 0 \leq i \leq m-1$, σ_i can be obtained as a composition of σ_1 , namely $\sigma_i = \sigma_1^i$.

Recall the subfield criterion, which states that the subfields of \mathbb{F}_{q^m} are exactly the fields \mathbb{F}_{q^n} where $n|m$. The mapping $\sigma_n = \sigma_1^n$ fixes the elements of the subfield \mathbb{F}_{q^n} , i.e.: $\sigma_n(\alpha) = \alpha$ if and only if $\alpha \in \mathbb{F}_{q^n}$, see [71].

Another interesting function, defined on a finite field, that involves conjugates of an element, is the *trace function*:

$$\mathrm{Tr}_{\mathbb{F}_q}^{\mathbb{F}_{q^m}} = \sum_{i=0}^{m-1} \alpha^{q^i} = \alpha^{q^0} + \alpha^{q^1} + \cdots + \alpha^{q^{m-1}}$$

The mapping $\mathrm{Tr}_{\mathbb{F}_q}^{\mathbb{F}_{q^m}} : \mathbb{F}_{q^m} \rightarrow \mathbb{F}_q$, as defined above, is called the *trace* of the element $\alpha \in \mathbb{F}_{q^m}$ with respect to the underlying subfield \mathbb{F}_q . If \mathbb{F}_q is a prime subfield, $\mathrm{Tr}_{\mathbb{F}_q}^{\mathbb{F}_{q^m}}$ is called *absolute trace*, see [69]. Note that the number of terms in the expression above equals the degree of extension $m = [\mathbb{F}_{q^m} : \mathbb{F}_q]$, i.e. it runs through all conjugates of α . The trace is independent of the chosen basis. We will now give some useful properties of trace function.

Theorem 2.3 *Let $F = \mathbb{F}_{q^m}$ and $K = \mathbb{F}_q$. Then the trace function Tr_K^F satisfies the following properties:*

- i. $\mathrm{Tr}_K^F(\alpha + \beta) = \mathrm{Tr}_K^F(\alpha) + \mathrm{Tr}_K^F(\beta)$ for all $\alpha, \beta \in F$
- ii. $\mathrm{Tr}_K^F(c\alpha) = c\mathrm{Tr}_K^F(\alpha)$ for all $c \in K, \alpha \in F$
- iii. Tr_K^F is a linear transformation from F onto K ,
where both F and K are viewed as vector spaces over K
- iv. $\mathrm{Tr}_K^F(a) = ma$ for all $a \in K$
- v. $\mathrm{Tr}_K^F(\alpha^q) = \mathrm{Tr}_K^F(\alpha)$ for all $\alpha \in F$

Theorem 2.4 *[Transitivity of trace] Let K be a finite field, let F be a finite extension of K and E a finite extension of F . Then*

$$\mathrm{Tr}_K^E(\alpha) = (\mathrm{Tr}_F^E \circ \mathrm{Tr}_K^F)(\alpha) = \mathrm{Tr}_K^F(\mathrm{Tr}_F^E(\alpha)) \quad , \quad \text{for all } \alpha \in E$$

2.3 Stream ciphers

2.3.1 General structure

The story of stream ciphers began with Vernam's shield, i.e. the one-time pad, in the early 20th century [72]. It encrypts the plaintext one character at a time by XORing it with a keystream character; the ciphertext is decrypted in the same manner, by XORing a ciphertext character with the keystream character that was used for its encryption. Note that such encryption and decryption are very fast. Despite the fact that it is the only provable secure system ever used in practice (assuming a truly random keystream), it has one immediately obvious drawback: the one-time pad uses a keystream of the same length as the plaintext, and this keystream is shared between the sender and the receiver, which requires a secure transmission of the keystream itself. This is a general problem that is encountered in all symmetric-key cryptosystems and is solved by means provided by public-key cryptosystems and handshake protocols. But using these methods to exchange the key for one-time pad would be pointless, and redundant: why encrypt and send the key if we could do that with the message as well? But public-key systems are computationally way more demanding than symmetric key systems, and are hence used only for shorter messages, for example the pre-shared secret key.

To address these problems, today's stream ciphers use a short pre-shared key and a pseudo-random sequence generator (PRSG) to produce a sufficiently long keystream. The security of the stream cipher is now reduced to the security of the PRSG. The attacker's goal is to recover the secret key (seed) and the security of the PRSG is measured by the complexity of this task.

Figure 2.1 shows the general behavioral model of encryption and decryption using a stream cipher. The only difference between encryption and decryption is the "direction": encryption takes the plaintext as an input and outputs the ciphertext and decryption takes the ciphertext as an input and produces the plaintext as the output. The sender and the receiver are using the same PRSG with the same seed (pre-shared secret key and initial vector (IV)), to obtain the same keystream. The cipher operates in two phases: a key initialization phase (denoted KI in Figure 2.1) and the running phase, when the PRSG algorithm outputs the keystream (denoted PRSG in Figure 2.1), refer to [77] for details. The task of KI is to scramble the key and initialization value IV to produce the initial state for the PRSG. It is executed once per encryption session, it must be able to withstand known attacks and is designed to get the keystream as random as possible to make the task of recovering the secret key more difficult [77]. The KI itself is usually the PRSG algorithm

running for a certain number of steps either with output discarded or output added to the feedback of PRSG (for example dashed line in Figure 2.2). The first keystream character is produced when cipher enters its running phase.

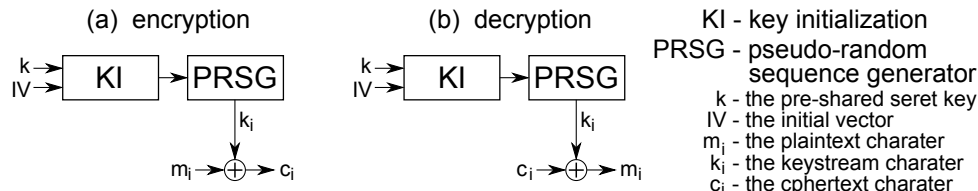


Figure 2.1: Behavioral model of a stream cipher: (a) encryption and (b) decryption

We use the word character when talking about the plaintext, keystream and ciphertext to avoid the distinction between word-oriented and bit-oriented stream ciphers. In a word-oriented stream cipher, the PRSG will produce a word of keystream per clock cycle, for example 8 or 32 bits, and the plaintext will be encrypted word by word, whereas a bit-oriented stream cipher produces one bit of keystream per clock cycle. Note that in the latter case, the plaintext can be encrypted bit by bit, or the keystream bits can be accumulated into words for word by word encryption. As already mentioned, the plaintext character is encrypted with a keystream character, and the obtained ciphertext must be decrypted using the same keystream character, which induces the requirement for synchronisation between the two parties involved; such ciphers are called synchronous stream ciphers.

2.3.2 A brief discussion on design principles

Here we present some design principles for stream ciphers and PRSGs (for details refer to [77]):

- efficiency in both hardware and software
- high throughput
- large period
- good randomness properties
- ability to resist known attacks

Efficiency in both hardware and software would be ideal, but in reality, some stream ciphers are more suited for software implementations and others for hardware implementations.

In general, word-oriented stream ciphers have a higher throughput, but it is more difficult to explore and to prove their randomness properties [3].

Randomness criteria for PRSG are statistical properties of the output sequence, which are meant to assure its indistinguishability from a truly random sequence: the balance property, run property, 2-level autocorrelation, low cross-correlation, ideal k -tuple distribution, etc. These randomness properties will be discussed in more detail in Section 2.3.2. A variety of test suites is available, for example NIST statistical test suite ([33]), to test these properties. But even if the keystream is able to pass these tests, it can still succumb to certain attacks, as will be discussed shortly.

In this work we focus on feedback shift register (FSR) based stream ciphers and their hardware implementations. Linear FSR's (LFSR) are easy to implement in hardware and have desired randomness properties. But stream ciphers based on LFSR's only cannot withstand known plaintext attacks, that is attempts to recover the key from the ciphertext and its corresponding plaintext; they succumb to techniques such as Berlekamp-Messey algorithm (BMA) or solving a system of linear equations to recover missing state bits ([77, 72]). An answer to these problems lies in introducing nonlinearity to the keystream, for example by using a nonlinear filtering function applied to the output(s) of the LFSR(s) (such designs are called "nonlinear filter generators"), using NLFSR's (nonlinear FSR's), irregular clocking of LFSR's, etc. General structure of a nonlinear filter generator is shown in Figure 2.2. The dashed line represents the output added to the LFSR feedback in the KI phase.

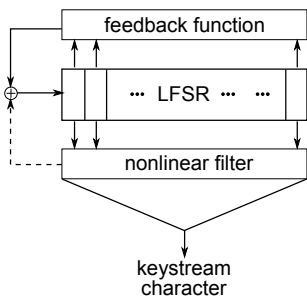


Figure 2.2: Structural model of a nonlinear filter generator

Nonetheless, stream ciphers still suffer from the same problem as one-time pad: they keystream should only be used once (hence the name one-time). Namely as soon as both, plaintext and ciphertext are known, the keystream can be recovered by simply XORing the

two, which makes decryption of any other ciphertext encrypted using the same key trivial [72]. To address this problem we need keystreams with sufficiently long period. In addition, initial vectors (IV's) are used to obtain different keystreams from the same key, and once the IV's run out the key should be changed (note that the IV is a publicly known parameter that is incremented with each session, and we must ensure it has a sufficient number of bits, based on the target application).

Randomness properties

Here we list and briefly describe the criteria that must be met by the binary pseudo-random sequence with period N . More detailed descriptions can be found in [76, 77].

1. **Long period:** we need a keystream of sufficient length and do not allow the sequence to repeat (ie, we take only one period as the keystream)
2. **Balance property:** a binary sequence has the balance property if in every period, the number of ones and zeros is nearly equal
3. **Run property:** A run of lengths k of zeros (or ones) is a subsequence of k consecutive zeros (or ones). The output sequence is said to have the run property, if in every period, half of runs have length one, one-fourth have length two, one-eighth have length three, etc., and there are equally many runs of zeros and of ones.
4. **k -tuple distribution:** for $k = \lceil \log N \rceil$, each binary k -tuple occurs nearly equally many times in one period.
5. **Two-level autocorrelation:** In depth explanation and formula for computation of the autocorrelation function for a periodic sequence can be found in [76, 77]. For this thesis it is enough to say that the autocorrelation counts agreements and disagreements between two sequences. The two-level autocorrelation property is satisfied, if the autocorrelation, computed between the sequence and its shifted version, takes on one of two possible values: (i.) value N , if the sequence is shifted by a multiple of N , or (ii.) t for all other shifts, where $t = -1$ for odd N and $t = 0$ for even N .
6. **Low-level cross correlation:** Detailed description of the cross correlation between a sequence and a shifted version of another sequence can be found in [76, 77]. For our purposes it is enough to say that we consider the cross correlation for sequences of same period N , and that we say the cross correlation is low, if its absolute value is limited by $c\sqrt{N}$ for some positive constant c .

7. **Large linear span:** Linear span of a binary sequence is defined to be the length of the shortest LFSR that can generate that sequence. Large linear span protects against the aforementioned BMA (note that the attack is also known as “linear span attack”) For details refer to [77].

Properties 2,3 and 5 are also known as Golomb’s randomness postulates, that can be extended for nonbinary sequences, that is sequences over \mathbb{F}_q , where q is some prime power, for details refer to [76].

Attacks

Cryptography and cryptanalysis are inseparable in many aspects: not only does cryptanalysis provide tools to assess security of a given cryptographic primitive, but also influences its design; the resistance against known attacks is an inevitable requirement. Just like other cryptographic primitives, stream ciphers too must obey the Kerckhoff’s principle demanding that all but the secret key must be publicly known. An attempt to recover this secret information is called “key recovery attack”. While it is the most powerful attack on a stream cipher, it is not the only possible goal of the attacker. A “message recovery attack” concentrates on decryption of a single message and a “distinguisher” aims at extracting some information about the encryption [78]. Another attack, that has a big impact in the world of stream ciphers, is the “state recovery attack”; not as severe as key recovery, but enables the attacker to generate the rest of the keystream and thus decrypt all future ciphertexts.

Whether aiming for key or state recovery, an exhaustive search can always be launched: the attacker tries all possible keys/states until the correct one is found. To avert exhaustive search the number of possible keys/states must be large enough to render the attack useless. For a k -bit key, the complexity of the exhaustive search is of the order of 2^{k-1} operations (expecting to find the correct key after searching about half of them)[72]. Attacker is interested in finding ways of accomplishing his task more efficient in comparison to exhaustive key search. A classification of attacks on stream ciphers can be found in [77]:

1. cryptanalysis (correlation attacks, algebraic attacks, linear cryptanalysis)
2. time-memory-data (TMD) trade-off attacks (exhaustive search with reduced complexity at the cost of using memory)
3. side-channel attacks (exploiting leakage: timing, power)

4. system and implementation attacks.

As was mentioned before, these attacks affect the design of the cipher itself. Recalling the nonlinear filter generator from Figure 2.2, let us describe two attempts of recovering the internal state of the LFSR from a known portion of the keystream, namely the correlation attack and the algebraic attack. The nonlinear filter used is a n -input Boolean function f , that must meet certain cryptographic properties, some of them arising from following attacks. The reader should refer to [77, 78] for details.

Correlation attack: In this scenario, the keystream is regarded as a noisy version of the LFSR sequence [3]. The actual nonlinear filtering function is approximated with some linear function, that can be used to derive a generator matrix for a linear code and the internal state recovered using maximum likelihood decoding. To protect against correlation attacks, we must use a Boolean function f , that is “correlation immune”, that is, the output of f (keystream) is uncorrelated to inputs of f (LFSR state).

Algebraic attack: These attacks are build upon the notion of describing the Boolean function f with a large system of multivariate polynomial equations over a finite field. The attack consists of two phases, first phase is finding the system of equations and the second phase solving it to obtain the internal state of the LFSR. To resist algebraic attacks the Boolean function used must have large “algebraic immunity”.

Resynchronisation attacks: These types of attacks are directed towards the initialization phase of the stream cipher using linear or differential cryptanalysis to recover partial or the entire secret key. Resynchronisation attacks point out the importance of the nonlinearity and the need for sufficiently many initialization steps to hide differential trails. A proper choice of the LFSR polynomial is also important [8, 12].

These are of course not all known attacks, but merely some examples of the threats. The lessons learned dictate some desired properties for the Boolean function f , that can be briefly summarized as follows: f should be balanced, have a high algebraic degree and algebraic immunity, must be highly nonlinear and correlation immune. We state these as facts, without going into details; interested reader should refer to [76, 77, 4, 7].

2.4 The WG stream cipher

The WG stream cipher is a bit-oriented stream cipher which generates a keystream with proven randomness and cryptographic properties. It was first proposed by Nawaz and Gong in 2005 and the profile 2 candidate WG-29 reached the phase 2 of the eSTREAM competition, [3]. The WG stream cipher is a synchronous stream cipher based on the Welch-Gong (WG) transformations, and the WG stream cipher family consists of WG stream ciphers and their decimated variants. We begin this section by first presenting individual components that are crucial to understanding the overall structure of the WG-16 stream cipher, which is explained in Section 2.4.1. We conclude this section with a short review of security of WG-16 in Section 2.4.2.

2.4.1 Structure of WG-16

The WG stream cipher is composed of an LFSR over an extension field, that outputs an m -sequence, which is then filtered with the WG transformation over the same extension field. Two key concepts were introduced in the sentence above: an LFSR and the Welch-Gong (WG) transformation; we shall first take a closer look at each of these components and then present the WG-16 stream cipher.

The LFSR

We have already briefly mentioned the LFSRs in Section 2.3 in the discussion about stream ciphers. An n -stage shift register over an extension field \mathbb{F}_{2^m} is an array of n registers, each of them holding an m -bit value (an element from \mathbb{F}_{2^m}). These registers are also referred to as stages, and are denoted $S_i, i = n-1, \dots, 0$. This memory array is shifted with each step: the contents of a register S_i are passed on to the neighboring register S_{i-1} and the vacant (first) register S_{n-1} is updated with a new value obtained from the feedback function. The feedback function is a simple expression that involves only multiplications of field elements by constants and addition in \mathbb{F}_{2^m} . The field elements entering the feedback function are just the contents of the LFSR. One of the n registers is chosen to be the output: in this way, the LFSR produces a sequence of \mathbb{F}_{2^m} field elements. For more details on LFSRs refer to [76, 77, 69].

The WG-16 has a 32 stage LFSR described by the polynomial

$$\ell(x) = x^{32} + x^{25} + x^{16} + x^7 + \omega^{2743} \tag{2.4}$$

over $\mathbb{F}_{2^{16}}$, where ω is the root of the defining polynomial $p(x) = x^{16} + x^5 + x^3 + x^2 + 1$ of the extension field $\mathbb{F}_{2^{16}}$. The polynomial $\ell(x)$ is a primitive polynomial, which ensures that the LFSR generates a maximal length sequence (m-sequence) with period $(2^{16})^{32} - 1$. The feedback function associated with the polynomial $\ell(x)$ from equation (2.4) is a function from $\mathbb{F}_{2^{16}}^{32} \rightarrow \mathbb{F}_{2^{16}}$ given by

$$f(x_0, x_1, \dots, x_{31}) = \omega^{2743}x_0 + x_7 + x_{16} + x_{25} \tag{2.5}$$

■ **Remark:** The element ω^{2743} was chosen because the multiplication matrix for $S_0 \cdot \omega^{2743}$ has the lowest Hamming weight, specifically 110. ■

With each step, the LFSR is updated as follows:

$$(S_0, S_1, \dots, S_{31}) \rightarrow (S_1, S_2, \dots, S_{31}, S_{32}),$$

where $S_{32} = f(S_0, S_1, \dots, S_{31}) = \mathbf{f}$ is computed as defined by equation (2.5).

The LFSR used in WG-16 is shown in Figure 2.3. When referring to “a step of the LFSR”, we mean that the contents of the registers are shifted to the right and the register S_{31} is updated with the feedback \mathbf{f} , also denoted S_{32} . Another commonly used term is “clocking of the LFSR”, i.e. we will be using clocked as a synonym for shifted or for performing one step. The LFSR stages S_{25}, S_{16}, S_7 and S_0 are referred to as “taps”, meaning that the LFSR described by polynomial $\ell(x)$ from equation (2.4) has “tap positions” 25, 16, 7 and 0.

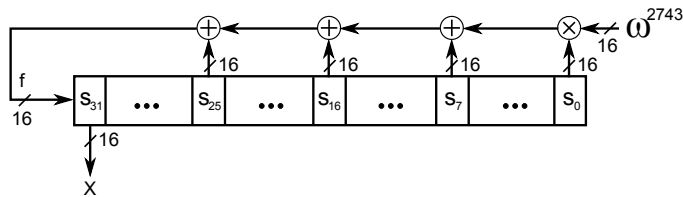


Figure 2.3: The WG-16 LFSR

The LFSR output is the stage S_{31} , marked X in Figure 2.3. Each time the LFSR is clocked, it produces a new output, so we get an output sequence $\mathbf{X} = X_1, X_2, \dots$. The “internal state” of the LFSR at step k are just the contents of all its registers: $(S_k, S_{k+1}, \dots, S_{k+31})$, where S_{k+i} denotes the element in LFSR stage S_i for $i = 0, 1, \dots, 31$ at step $k = 0, 1, \dots$. Note that the LFSR in Figure 2.3 has an internal state of 512 bits. For $k = 0, 1, \dots$, the $(k + 1)$ -st element of the output sequence is

$$X_{k+1} = f(S_k, S_{k+1}, \dots, S_{k+31}).$$

Note that the LFSR is first loaded with some non-zero initial state (corresponding to $k = 0$), after that, its behavior is completely determined by the recursive relationship $S_{k+1+31} = f(S_k, S_{k+1}, \dots, S_{k+31})$, $k = 0, 1, \dots$. Note that in the first step after the LFSR has been initialized, i.e. for $k = 0$, this gives the S_{32} , which was mentioned in the description above. The initialization of the LFSR will be discussed in detail together with the WG-16 cipher itself in Section 2.4.1.

The WG transformation

Let m be an integer that is not a multiple of 3, that is $m \bmod 3 \neq 0$. Then the WG transformation from \mathbb{F}_{2^m} to \mathbb{F}_2 is defined by

$$f(x) = \text{Tr}(q(x+1) + 1), \quad \text{for } x \in \mathbb{F}_{2^m}. \quad (2.6)$$

Equation (2.6) can be split into two parts: the WG permutation $\text{WGP}(x)$ (equation (2.7)), and the WG transformation $\text{WGT}(x)$ (equation (2.8)), which is the absolute trace applied to the result of the WG permutation.

$$\text{WGP}(x) = q(x+1) + 1 \quad (2.7)$$

$$\text{WGT}(x) = \text{Tr}(\text{WGP}(x)), \quad (2.8)$$

The polynomial $q(x) = x + x^{r_1} + x^{r_2} + x^{r_3} + x^{r_4}$ is a permutation polynomial from \mathbb{F}_{2^m} to \mathbb{F}_{2^m} . For a positive integer k , such that $3k \equiv 1 \pmod{m}$, the exponents are obtained as follows:

$$\begin{aligned} r_1 &= 2^k + 1 \\ r_2 &= 2^{2k} + 2^k + 1 \\ r_3 &= 2^{2k} - 2^k + 1 \\ r_4 &= 2^{2k} + 2^k - 1 \end{aligned}$$

For $m = 16$ parameter $k = 11$ was used, yielding the following coefficients:

$$\begin{array}{llll} r_1 &= 2^k + 1 & r_2 &= 2^{2k} + 2^k + 1 & r_3 &= 2^{2k} - 2^k + 1 & r_4 &= 2^{2k} + 2^k - 1 \\ &= 2^{11} + 1 & &= 2^{22} + 2^{11} + 1 & &= 2^{22} - 2^{11} + 1 & &= 2^{22} + 2^{11} - 1 \\ & & &= 2^6 + 2^{11} + 1 & &= 2^6 - 2^{11} + 1 & &= 2^6 + 2^{11} - 1 \end{array}$$

In computations above we used the finite field analogue of Fermat's little theorem 2.1 for a finite field element $x \in \mathbb{F}_{2^{16}}$:

$$x^{2^{22}} = \left(x^{2^{16}}\right)^{2^6} = x^{2^6}$$

Intuitively, a d decimation of an m -sequence is a transformation which, provided that $\gcd(d, 2^m - 1) = 1$, produces a new m -sequence by taking every d -th element of the original sequence until all elements of the original sequence are used up. Decimation can improve cryptographic properties of the produced keystream [7].

The WG-16 is a decimated stream cipher using decimation exponent $d = 1057$. We can now describe the WG-16 transformation of an element $X \in \mathbb{F}_{2^{16}}$ with the following four equations:

$$Y = X^d + 1 \tag{2.9}$$

$$q(Y) = Y + Y^{2^{11}+1} + Y^{2^6+2^{11}+1} + Y^{2^6-2^{11}+1} + Y^{2^6+2^{11}-1} \tag{2.10}$$

$$\text{WGP-16}(X^d) = q(Y) + 1 \tag{2.11}$$

$$\text{WGT-16}(X^d) = \text{Tr}(\text{WGP-16}(X^d)), \tag{2.12}$$

These four equations describe decimated WG-16 permutation followed by the trace computation and are grouped together into a component denoted **WGP_T**, that can be seen in Figure 2.4 shaded grey.

The WG-16 stream cipher

In the introductory text on stream ciphers (Section 2.3) we mentioned that modern stream ciphers operate in two phases, namely the key initialization phase (which will be referred to as initialization phase from now on) and the running phase, both of them using the same PRSG algorithm with minor differences. The PRSG algorithm used by members of WG stream cipher family is the WG transformation applied to the LFSR sequence. During the running phase, the LFSR is updated by the feedback $\mathbf{f} = f(S_k, S_{k+1}, \dots, S_{k+31})$ and each time the LFSR is clocked the WG-16 produces one bit of the keystream as $\text{WGT-16}(\text{WGP-16}(S_{k+31}^d))$. At each LFSR step $k = 0, 1, \dots, 63$ during the initialization phase, the LFSR is updated with the sum \mathbf{w} of the LFSR feedback \mathbf{f} and the element **WGP**, which is obtained by the decimated WG-16 permutation:

$$\mathbf{f} = f(S_k, S_{k+1}, \dots, S_{k+31})$$

$$\text{WGP} = \text{WGP-16}(S_{k+31}^d)$$

$$\mathbf{w} = \mathbf{f} + \text{WGP}$$

The element **WGP** adds nonlinearity to the linear feedback \mathbf{f} . The initialization phase takes 64 LFSR steps, which means that each parcel runs through the LFSR twice, or in other words, each LFSR stage S_i is updated 64 times. During the initialization phase the keystream bit is discarded.

The recurrence relations for updating the LFSR during the initialization and the running phase are summarized as follows:

$$S_{k+1+31} = \begin{cases} \omega^{2743} S_k + S_{k+7} + S_{k+16} + S_{k+25} + \text{WGP-16}(S_{k+31}^d), & 0 \leq k < 64 \\ \omega^{2743} S_k + S_{k+7} + S_{k+16} + S_{k+25}, & k \geq 64 \end{cases} .$$

We must not forget the loading phase, during which the initial LFSR state $(S_0, S_1, \dots, S_{31})$ is loaded into the LFSR. The values $S_0, S_1, S_2 \dots$ appear on the DIN (data-in) input serially: first the value S_0 is loaded into LFSR stage S_{31} , then the LFSR is shifted and element S_1 is loaded into stage S_{31} , etc To load all 32 field elements serially, the LFSR must be clocked 32 times. A detailed description of the initial state follows at the end of this section in 2.4.1. At this point we want to give the top-level structure of WG-16, and need to stress out that we are using the term WG-16 a bit loose: the schematic in Figure 2.4 is showing the architecture of WG stream cipher generator WG(16,32). The actual stream cipher then uses this component to generate the keystream and perform the encryption or decryption. For this thesis we only implement the keystream generator WG(16,32), and refer to it as WG-16.

■ **Remark:** In notation $\text{WG}(m, n)$: parameter m denotes the underlying finite field \mathbb{F}_2^m and parameter n the degree of the LFSR polynomial ℓ . ■

The two crucial components of WG-16, the LFSR and the **WGP_T**, were already introduced, and the three operating phases of the WG-16 were discussed above. The input to the LFSR stage S_{31} is different in each phase, and the number of LFSR steps also differs with phase. The three phases with corresponding S_{31} inputs and number of steps per phase are summarized in Table 2.2 below.

The three different inputs for S_{31} call for a 3/1 16-bit wide multiplexer at the S_{31} input. A third component, an FSM, is needed to control this multiplexer and the the keystream

WG-16 phase	input to S_{31}	# of steps per phase
loading	DIN	32
initialization	$w=f+WGP$	64
running	f	†

Table 2.2: Three phases of the WG-16 operation
 † upperbounded by the period of the LFSR sequence

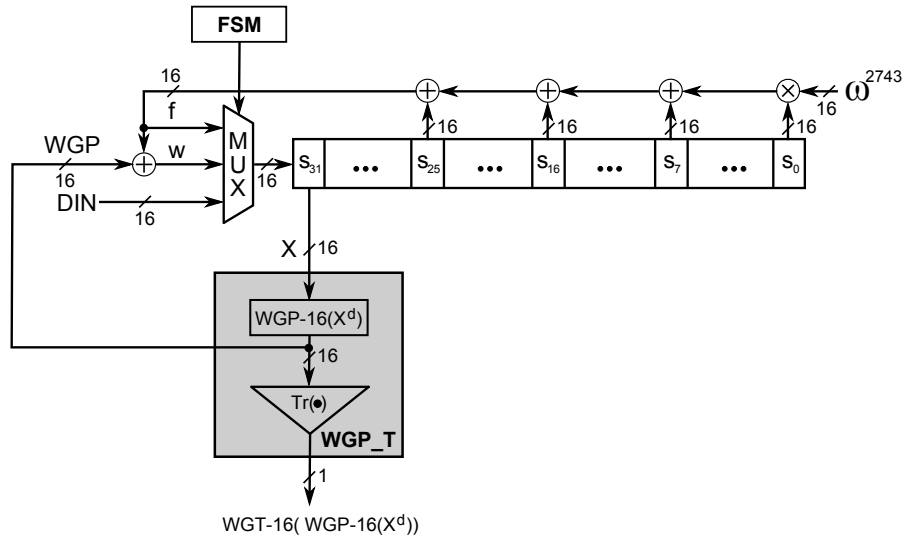


Figure 2.4: Architecture of WG-16 stream cipher

output. The top-level schematic of WG-16 with the LFSR, the WGP_T and the FSM is shown in Figure 2.4.

The FSM has three states, corresponding to the three phases: **load** for loading, **init** for initialization, and **run** for the running phase. The state transition diagram for the FSM is shown in Figure 2.5 on the right. The counter count keeps track of the number of LFSR steps required for each phase, see Table 2.2 for details.

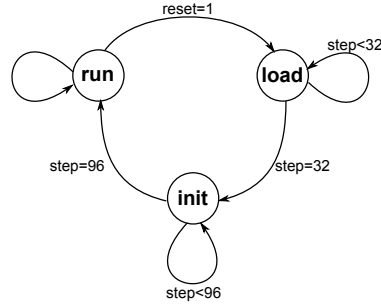


Figure 2.5: Three phases of the WG-16 operation

The initial state of the LFSR - the key and IV mixing

Recall the loading phase as it was described above: the initial state $(S_0, S_1, \dots, S_{31})$ is loaded into the LFSR serially in 32 consecutive clock cycles through the 16-bit data input DIN. Here we want to explain how the initial state is obtained; this process is sometimes called the initial key and IV mixing. The initial state of the LFSR is composed of a 128-bit key and 128-bit IV. Using notation $K = (k_0, k_1, \dots, k_{127})$, $IV = (iv_0, iv_1, \dots, iv_{127})$ and $(S_0, S_1, \dots, S_{31})$ for the initial state of the LFSR, where $S_i = (s_{i,0}, s_{i,1}, \dots, s_{i,15})$ for $i = 0, 1, \dots, 31$, the stage S_i is loaded as follows:

$$S_i = \begin{cases} (k_{8i}, k_{8i+1}, \dots, k_{8i+7}, iv_{8i}, iv_{8i+1}, \dots, iv_{8i+7}), & i = 0, 1, \dots, 15 \\ S_{i-16}, & i = 16, \dots, 31 \end{cases} \quad (2.13)$$

During the loading phase, the key, IV and the LFSR stages are viewed as collections of 8-bit data:

- $K = (K_0, K_1, \dots, K_{15})$ where $K_i = (k_{8i}, k_{8i+1}, \dots, k_{8i+7})$ for $i = 0, 1, \dots, 15$
- $IV = (IV_0, IV_1, \dots, IV_{15})$ where $IV_i = (iv_{8i}, iv_{8i+1}, \dots, iv_{8i+7})$ for $i = 0, 1, \dots, 15$
- $S_i = (S_{i,0}, S_{i,1})$ where $S_{i,j} = (s_{i,8j}, s_{i,8j+1}, \dots, s_{i,8j+7})$ for $j = 0, 1$ and $i = 0, 1, \dots, 31$

Using this notation, we can rewrite the loading rule from equation (2.13) as two separate rules for $S_{i,0}$ and $S_{i,1}$, $i = 0, \dots, 31$:

$$S_{i,0} = \begin{cases} K_i, & i = 0, 1, \dots, 15 \\ S_{i-16,0}, & i = 16, \dots, 31 \end{cases} \quad (2.14) \quad S_{i,1} = \begin{cases} IV_i, & i = 0, 1, \dots, 15 \\ S_{i-16,1}, & i = 16, \dots, 31 \end{cases} \quad (2.15)$$

The contents of the LFSR after the loading phase are shown in Figure 2.6. Stages S_i for $i = 0, \dots, 15$ are shaded grey and the unshaded stages S_i for $i = 16, \dots, 31$ are just an exact copy of the shaded stages. The LFSR in Figure 2.6 is split into half horizontally,

separating the two bytes $S_{i,0}$ and $S_{i,1}$. If we look at the “lower half” of the LFSR as an array of 8 bit registers $S_{i,0}$, $i = 0, \dots, 31$, we see that it contains exactly two copies of the 128-bit key K , which corresponds to the loading rule given in (2.14). Similarly, the “upper half” contains two copies of the 128-bit IV, corresponding to the loading rule (2.15).

S_{31}				S_{16}	S_{15}				S_1	S_0
IV_{15}	...			IV_0	IV_{15}	...			IV_1	IV_0
K_{15}	...			K_0	K_{15}	...			K_1	K_0
										$\left. \begin{matrix} \text{---} \\ \text{---} \end{matrix} \right\} S_{i,1}$ $\left. \begin{matrix} \text{---} \\ \text{---} \end{matrix} \right\} S_{i,0}$

Figure 2.6: Contents of the LFSR after the loading

2.4.2 Security of the WG

Cryptographic properties of WG generators were first discussed in [2] from two different perspectives:

- as WG transformation sequences, to show their randomness properties (recall discussion in Section 2.3.2), and
- as Boolean functions, for which the nonlinearity, algebraic degree, resilient property and linear span was established.

The keystream, produced by the WG(16,32) generator has the following randomness properties [8]:

- its period is $2^{512} - 1$
- it is balanced
- it has ideal 2-level autocorrelation
- it has ideal k -tuple distribution for $1 \leq k \leq 32$
- it has a large linear span that can be determined exactly as $2^{79.046}$

A 2014 paper [7] discussing the decimated WG transformation provides selection criteria for optimal parameters for the WG cipher family in order to achieve the maximum level of security. Optimal decimation value d should be chosen in such a way that in addition to $\gcd(d, 2^m - 1) = 1$, both, the algebraic degree and algebraic immunity for $\text{WGT-16}(X^d)$ are maximum. For WG-16, there are 31 decimation values that meet the above criteria. Additional (cryptographic) properties, such as low Hamming weight of d , large nonlinearity

of WGT-16(X^d) , etc, help us to select the optimal decimation value $d = 1057$.

Extensive cryptanalysis of the WG-16 and its resistance to known attacks can be found in [8]. The same paper also provides a proposal for the use of WG-16 in 4G-LTE network. Here we provide a short summary of their results in terms of time complexity of the attack:

- Exhaustive search: $\mathcal{O}(2^{511})$
- TMD trade-off: $\mathcal{O}(2^{256})$
- Algebraic attack: $\mathcal{O}(2^{155.764})$
- Correlation attack: $\mathcal{O}(2^{121.31})$

Note that the above are just the time complexities. The attacker also needs to collect some data, for example to launch the algebraic attack he needs about $2^{56.622}$ keystream bits. The 64 steps of initialization phase protect against differential attack and the cube attack [8].

2.5 Related work

This section is organized into four parts. In Section 2.5.1 we briefly summarize the existing WG hardware implementations. In Section 2.5.2 we present the stream ciphers that are currently being used 3GPP confidentiality and integrity algorithms Snow3G and ZUC. Then we talk about the eSTREAM project and briefly present the finalists Grain and Trivium. In Section 2.5.4 we review the use composite field arithmetic in cryptography and focus on the tower field constructions of the finite field $\mathbb{F}_{((2^2)^2)^2}$ used in AES hardware implementations.

2.5.1 WG hardware implementations

The first member of WG stream cipher family to be implemented in hardware was the eSTREAM candidate WG-29 [3]. For the $\mathbb{F}_{2^{29}}$, a type II optimal normal basis exists, which allows efficient field arithmetic. In [5], useful properties of the trace function were found, that allowed elimination of two multipliers. Switching to the polynomial basis representation of field elements, the same group later on improved their implementation results for WG-29 in [13]. The same paper also reports efficient polynomial basis implementations of WG-16.

An interesting variant of WG stream ciphers, the multi-output WG (MOWG) was proposed in [10]. In MOWG the trace function was replaced by a multi-output Boolean function. The MOWG was implemented over \mathbb{F}_{2^7} , $\mathbb{F}_{2^{11}}$ and $\mathbb{F}_{2^{29}}$, with output widths 3, 6 and 17 respectively, and different LFSR polynomials were chosen for both \mathbb{F}_{2^7} and $\mathbb{F}_{2^{11}}$. Also, for those two ciphers, table look-up based design was chosen and implemented using either random logic or ROM. Note that the tables were not used to implement the finite field arithmetic, but the WG itself, that is, one table was holding the WGP values for initialization phase and another table the MOWG values for the running phase. The WG(29,11,17) was implemented using superpipelined multiplier (multiplier pipelined into two stages) with multiplier reuse.

An implementation of a lightweight WG stream cipher WG-5, targeting passive RFIDs, was reported in [9]. The defining polynomial of \mathbb{F}_{2^5} , the characteristic polynomial for the LFSR and the decimation value were chosen not only based on resulting cryptographic properties but to produce the most optimal hardware. Based on ASIC implementation results for chosen frequencies of 100 and 200 kHz, WG-5 outperforms the ciphers it was compared to, including Grain and Trivium.

Another instance of lightweight WG stream ciphers, the recently implemented WG-8, was reported in [11]. It explores four different hardware architectures. The first implementation is a table look-up based design (with one table holding the WGP values and one table holding the WGT values). Then two tower constructions $\mathbb{F}_{(2^4)^2}$ were implemented, using different defining polynomials for the first extension. One of them used polynomial basis for \mathbb{F}_{2^4} and table look-up based field arithmetic, and the other one type I optimal normal basis, yielding efficient field arithmetic. The fourth design used the tower construction $\mathbb{F}_{((2^2)^2)^2}$, with normal basis representation of elements at each level of the tower, similar to the work in this thesis (the WG-8 work was conducted in parallel with WG-16). FPGA and ASIC implementation results were given for 1-bit and for 11-bit output versions for all four designs. Since the cipher is small enough, best results were achieved for the table look-up based design.

2.5.2 3GPP confidentiality and integrity algorithms: Snow3G and ZUC

The current 4G-LTE standards reuse the authentication and key agreement of UMTS and the confidentiality and integrity algorithms abbreviated UEA/UIA (USIM Encryption/Integrity Algorithm) for UMTS and were adopted in LTE and are known as EEA/EIA (EPS Encryption/Integrity Algorithm). In this section we give results of some other stream ciphers used nowadays. We begin by exploring Snow3G and ZUC, currently used as 3GPP confidentiality and integrity algorithms for protecting the radio interface in UMTS/LTE systems. Third instance, AES will be discussed separately at the end of this section, with focus on its tower field implementations.

Both Snow3G and ZUC are word-oriented stream ciphers that produce 32-bit keywords, using a 128-bit key and IV of the same length. They both have a 16-stage LFSR that is clocked 32 times during the initialization phase with nonlinear filter/function output F added to the LFSR feedback. Both of them employ bitwise XOR operation, modular addition and S-boxes.

Snow3G

The structure of Snow3G can be seen in Figure 2.7. The cipher is composed of three parts: a 16-stage LFSR over $\mathbb{F}_{2^{32}}$, a FSM with three registers and a filter applied to LFSR stage s_{15} . The finite field $\mathbb{F}_{2^{32}}$ is constructed as a tower field $\mathbb{F}_{(2^8)^4}$ by adjoining the root α of a degree 4 irreducible polynomial to the base field \mathbb{F}_{2^8} . The LFSR feedback function involves multiplications of LFSR elements with α and α^{-1} . Both the FSM and the filter use bitwise XOR, denoted \oplus in Figure 2.7, and integer addition modulo $(2^{32} - 1)$, denoted \boxplus . The FSM uses three registers R1, R2 and R3, holding 32 bits each, that are manipulated via two Rijandel based S-boxes S1 and S2; these S-boxes are the main source of nonlinearity in Snow3G. More precisely, the registers R2 and R3 are updated as follows: $R2 \leftarrow S1(R1)$, $R3 \leftarrow S2(R2)$ and R1 is updated with a value obtained from old values in registers R2, R3 and s_5 as $R1 \leftarrow R2 \boxplus (R3 \oplus s_5)$. The FSM takes the LFSR state s_5 as input and produces two outputs that are used in the filter where they are combined with LFSR state s_{15} to produce the value $F = (s_{15} \boxplus R1) \oplus R2$. The filter output F is either XORed with LFSR feedback during the initialization phase (marked with a dashed arrow in Figure 2.7) or XORed with LFSR state s_0 in the running phase to produce a keyword (marked with solid lines in Figure 2.7).

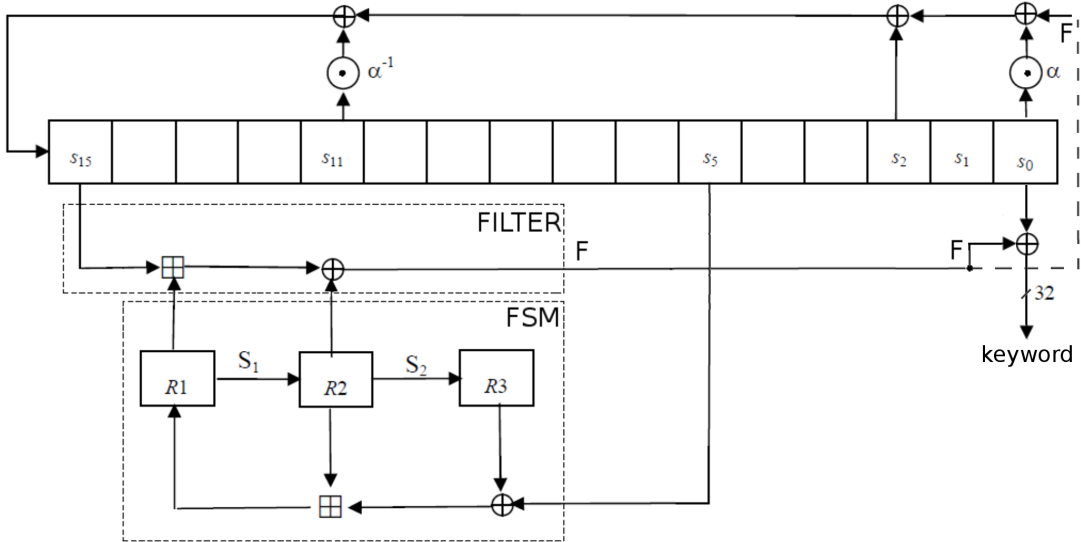


Figure 2.7: The structure of Snow3G stream cipher

The above description of Snow3G follows the 3GPP standard ([14]), which also gives the S-boxes and specifies efficient multiplication with α and α^{-1} using look-up Tables. As usual, all published papers give a lookup Table implementation of the S-boxes. In [15] an ASIC implementation of snow3G by P. Kistos, another phase is added to the cipher operation, the so called “initial operations”, whose task is to mix the key and IV as specified in standard ([14]). Usually, the key/IV mixing is done in software and then loaded into the LFSR serially, stage by stage. The ASIC design in question enables a fast loading phase at the cost of wider inputs (256 bits for the key and IV) and additional logic needed for the mixing itself. The same paper identifies the critical path for Snow3G to be the modular addition and proposes a carry lookahead adder to obtain the result in one clock cycle, thus significantly improving the throughput in comparison with a multi-cycle adder architecture. An FPGA implementation of Snow3G on an FPGA Spartan-3 device, also published by P. Kistos, is given in [20]. This paper provides implementations of several stream ciphers and their comparisons, and we shall revisit it at the end of this section. Two additional FPGA implementations of Snow3G were given in [21], both aiming at optimizing the look-up Tables used in feedback computation and for S-boxes. The first implementation (denoted “[21]-ver.I” in Table 2.3) uses Block RAM available on Xilinx Virtex-5 to reduce area complexity of their design. Using BRAM requires some changes to the algorithm itself, since the BRAM output is available in the next clock cycle. In their second version (denoted “[21]-ver.II” in Table 2.3), they implement Snow3G lookup-Tables

using Slice LUTs, which shows the expected area increase.

Implementation results for the Snow3G implementations mentioned above are listed in Table 2.3.

ZUC

ZUC is composed of three logical layers: a 16 stage LFSR with 31 bits per stage, bit reorganization layer BR and nonlinear function F. The architecture of the cipher can be seen in Figure 2.8. The LFSR feedback function involves left cyclic shifts and five additions modulo $(2^{31} - 1)$. BR layer extracts four 32-bit words from the LFSR cells that are then used as inputs to function F and in formation of keywords. The nonlinear function F uses two 32-bit registers whose values are manipulated using two linear transformations, a nonlinear S-box and left cyclic shifts. Other operations performed by ZUC algorithm are the bitwise XOR and addition modulo 2^{32} . For details refer to [16].

All implementations of ZUC identify the addition modulo $2^{31} - 1$ to be the critical path in ZUC. An FPGA implementation on a Virtex-5 device was given by P.Kistos in [19]. They implemented additions module $2^{31} - 1$ and modulo 2^{32} using available DSP blocks. This approach reduces the area complexity and increases the frequency. Also, the feedback function that requires 5 modular additions was implemented by first computing partial sums and then summing them up, which reduced the critical path of the feedback calculation. Further area reduction was achieved by implementing the S-box in ROM. The second paper by P. Kistos [20] gives a Spartan-3 implementation of ZUC, but no detailed description of optimization approaches is provided. A pipelined implementation of ZUC on a Xilinx Virtex-5 is reported in [17, 18]. The critical path for the LFSR feedback computation was pipelined into five stages, performing one modulo $(2^{31} - 1)$ addition per stage. The LFSR does not shift till the pipeline is full, after that it produces one keyword per cycle. Note that the tap positions were added to accommodate for the LFSR not shifting: adder inputs were equipped with multiplexers to choose between the “real” and “pipeline” taps. In comparison with other implementations, this causes an increase in area, but shows the highest throughput so far. Two other optimizations of the feedback computation were explored in [18]. First option (denoted “[18]-opt.I” in Table 2.3) computes the feedback over two consecutive clock cycles by computing the partial sums in the first cycle and summing them up in the second. Another area reducing optimization was made possible by the additional clock cycle: the ZUC S-box is composed of four smaller S-boxes, out of which two and two are alike. Instead of having 4 separate look-up tables, two lookup-tables,

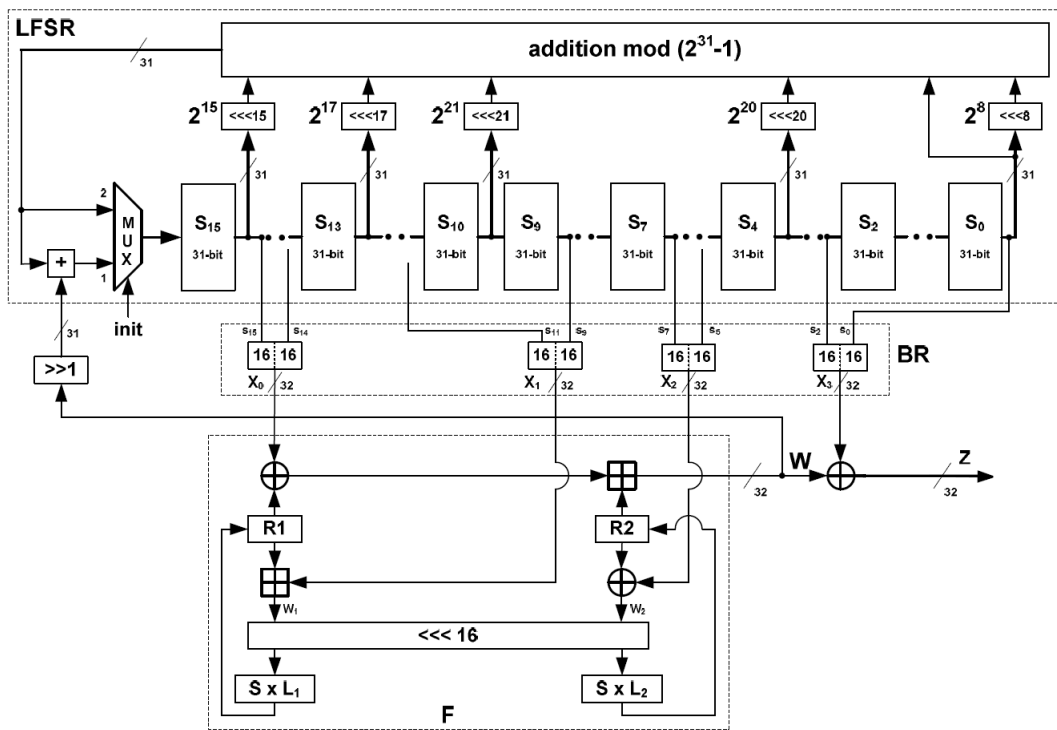


Figure 2.8: The structure of ZUC stream cipher

each used twice, suffice. As expected, this implementation has lowest throughput among the three implementations reported in [18]; the optimization did shorten the critical path and hence the clock period, but it takes two clock cycles for one keyword, which has a significant impact on the throughput. The last option (denoted "[18]-opt.II" in Table 2.3) uses Carry Save Adder tree structure to compute the feedback; it takes up more area, but computes the feedback in a single clock cycle.

As promised, we return to Kistos' paper [20]. This paper compares implementations of different stream ciphers on an Spartan3 FPGA. Snow3G and ZUC were the only two word-oriented stream ciphers explored, all others (Grain, Mickey, Trivium and E0) are bit oriented. In order to compare word-oriented and bit-oriented stream ciphers, the metric of throughput-to-area consumption ($\frac{T}{A} \left[\frac{Mbps}{\#slices} \right]$) was used. Even though at this point we are comparing only two ciphers that produce keywords of same width, this metric allows an assessment of the trade-off between the throughput and the hardware resources used. Note that we do not compute the $\frac{T}{A}$ for the ASIC implementation and for implementations using BRAM and ROM. The last column in Table 2.3 identifies implementation [21]-ver.II

Stream Cipher	Source	Device	Area [# of Slices]	Frequency [MHz]	Throughput T [Mbps]	$\frac{T}{A}$
Snow3G	[15]	ASIC (0.13 μ m)	25KGates	249	7.968	-
	[20]	XC3S700A	3559	104	3328	0.935
	[21]-ver.I	Virtex-5	188	322	10304	-
	[21]-ver.II	Virtex-5	356	376	12036	33.808
ZUC	[19]	Virtex-5	285	65	2080	7.29
	[20]	XC3S700A	1147	38	1216	1.060
	[17, 18]	Virtex-5	575	222.4	7111	12.367
	[18]-opt.I	Virtex-5	311	126	2016	6.482
	[18]-opt.II	Virtex-5	356	108	3456	9.708
	[21]	Virtex-5	395	172	5504	32

Table 2.3: Implementation results for Snow3G and ZUC found in literature

as the most efficient Snow3G implementation. But we only have one other implementation for comparison and there is one fact that cannot be overlooked: the clock speeds that can be achieved on Virtex devices are always greater in comparison with Spartan devices, due to a different technology used for Spartans and Virtex'. Furthermore, the Spartan3 has four 4-input LUTs per slice, and the question how can we compare the number of slices to Virtex-5, which uses four 6-input LUTs per slice. Unfortunately, no detailed description was provided for this implementation.

Nonetheless, all implementations of Snow3G result in a higher frequency in comparison with ZUC. Best ZUC implementation was the version with fully pipelined LFSR feedback computation reported in [17, 18], but does not really come close to Snow3G [21]-ver.II.

When comparing Snow3G and ZUC to WG from the hardware implementation point of view, a few differences are immediately obvious. Firstly, WG is a bit-oriented stream cipher and will not reach the high throughput of Snow3G and ZUC. Never the less, the $\frac{T}{A}$ ratio achieved by some bit oriented ciphers reported in [20] is higher then 0.935 achieved by proposed Snow3G implementation; this serves as an example that even a high throughput can not justify a huge area cost. Secondly, the main focus of optimizations in the implementations described above, was the modular addition, which is not present in WG computation. An interesting option is the pipelined LFSR feedback function, but the WG feedback is trivial in comparison with WG transformation, hence there is no need for a pipeline. Also worth mentioning at this point is the computation of the feedback over two

consecutive clock cycles in [18]-opt.I.

2.5.3 The eSTREAM project: Grain and Trivium

eSTREAM project started in 2004 with objective to promote research in stream cipher design [25]. Two specific goals were identified : stream ciphers for software applications with high throughput (Profile 1) and stream ciphers for hardware applications with highly restricted resources (Profile 2). The latter is of our particular interest. Evaluation criteria includes security, performance (in comparison with AES and other other eSTREAM candidates), justification and supporting analysis, simplicity, flexibility, etc. The proposed ciphers went through three phases of evaluation: the first round was flexible and allowed for changes to the ciphers to remove identified weaknesses before entering the second phase. The design of a secure stream cipher proved to be a difficult task.

Here we focus on eSTREAM Profile 2. In Phase 3, three ciphers were selected and included in the eSTREAM portfolio: Grain v1, MICKEY 2.0 and Trivium. For Profile 2, FPGA and ASIC implementation results were considered, but there were problems in identifying a the most relevant metric for comparison. An discussion on this topic can be found in [31]. In the Phase 3, the primary criteria besides security was the area complexity [25]. All three ciphers included in the eSTREAM portfolio have smaller area than AES, see [26]. Another metric that was commonly used to compare the performance of the candidates was the aforementioned throughput-to-area ratio CITE THEM. In this section, we present Grain and Trivium and omit MICKEY due to its larger area complexity.

Both Grain and Trivium are FSR based and involve only simple binary operations (XOR, AND). They both have a small area allow the possibility of increasing the throughput by simply implementing multiple filtering functions and jumping multiple FSR stages. In both cases, the key/IV must be loaded bit-by-bit.

Grain

There are two versions of Grain: the original 80-bit version (using an 80-bit secret key and 64-bit IV, called Grain v1, included in eSTREAM Portfolio) and Grain-128 (using a 128-bit key and IV of same length). We will give a short description of Grain v1 and then list the differences made in Grain-128. From now on when we talk of Grain we refer to

Grain v1.

The structure of Grain can be seen in Figure 2.9. Grain is composed of an 80-bit LFSR and an 80-bit NLFSR, giving a total internal state of 160 bits. The NLFSR is updated by a nonlinear feedback polynomial that is further XORed by a bit from LFSR. Five bits (four from the LFSR and 1 from NLFSR) are chosen from the FSR's and used as an input to a Boolean function (that was chosen to meet certain desired cryptographic properties). The keystream bit is obtained by XORing the output of this function with 7 state bits from the NLFSR. The initialization phase takes 160 cycles, during which this bit is XORed to update values for both FSR's.

The Grain-128 FSR's are 128 bits long, and have different feedbacks. The Boolean function is also changed and has a bigger number of inputs from both FSR's. At the end, an additional bit from the LFSR is XORed to form the keystream bit. Note that the number of tap positions is not only increased but also changed. The initialization phase now lasts 256 cycles.

For more details on Grain, refer to [23, 24].

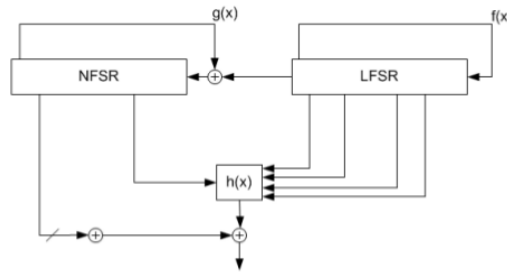


Figure 2.9: The structure of Grain stream cipher

Trivium

Trivium has a very simple design and can generate keystreams of length up to 2^{64} using an 80-bit secret key and IV of same length. It is composed of three FSR's of lengths 93, 84 and 111 bits respectively, which sums up to total internal state of 288 bits. These three FSR's can be arranged into a circular shape, as can be seen in Figure 2.10. From FSR point of view, the update functions of the three FSR's differ only in the tap positions. Each FSR has 5 tap positions used by the filtering function in two ways:

- to update the FSR's (using 4 bits from the previous FSR and one bit from the FSR being updated)

- to compute the keystream bit (by XORing 6 state bits, two from each FSR)

Initialization phase is equal to the running phase without keystream output and runs for $4 \cdot 288 = 1152$ clock cycles.

For more details on Trivium, refer to [22].

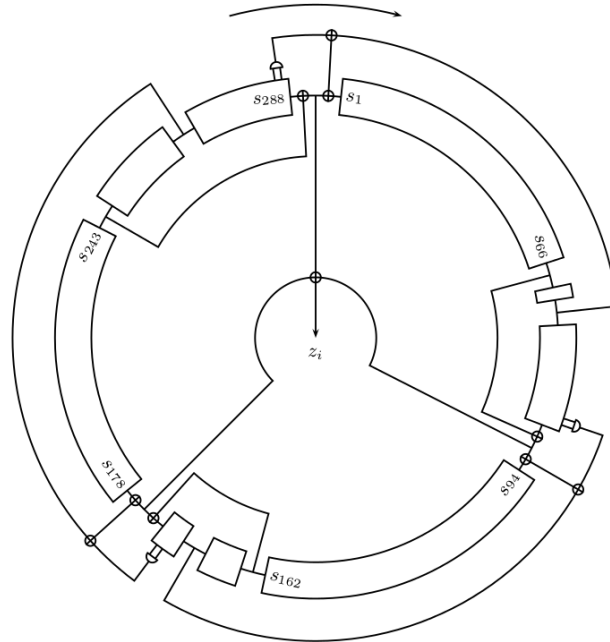


Figure 2.10: The structure of Trivium stream cipher

An ASIC implementation of eSTREAM candidates, reposted in [26], is using the term *radix* for the number of bits simultaneously generated by the algorithm. Without details, they mention Grain implementations with radices up to 16 and possibility of radix 32. For Trivium, they state that implementations with radix less than 64 would be wasteful and report a 54% increase in area, only 10% lower clock speed and a 40 times higher throughput-to-area ratio compared to Trivium with radix 1. Only the the results for radix 1 for both ciphers, and for the increased throughput versions radix 16 for Grain and radix 64 for Trivium are listed in Table 2.4. Only Grain-128 was implemented with radix 32 in [32, 31].

Another paper [27] identifies Grain and Trivium as the smallest and most efficient among the eSTREAM candidates and emphasizes their potential for higher radix. In [29], authors

report an FPGA implementation of Phase 2 candidates on a Xilinx Virtex-II, gaining remarkable area reductions from the use of SRL16 primitive (use of SRL16 is discussed in more detail in Section 4.1). They implemented Grain-128 and found the area increase caused by transition to 128-bit version unnoticeable. Authors of [28] provided detailed FPGA and ASIC results for five eSTREAM candidates, including Grain and Trivium, focusing on paralelization possibilities (aiming at increased throughput). They identified Grain as the cipher with minimum area complexity and Trivium as the cipher with maximum throughput-to-area ratio.

An FPGA implementation of Grain and Trivium, consciously refraining from use of SRL primitives, based on justification that it is device and design specific, was reported in [20]. Interesting implementation alternatives were explored by Rogawski [30]. He identified the feedback function for the NLFSR to be the critical path in Grain, and tried implementations using a lookup table for the feedback, but the straightforward combinational implementation remained superior in terms of area, power and throughput, so in Table 2.4 we omit results for the tabulated version. The same work also concentrates on the initialization phase. He identified the control unit as the critical component in Trivium. The 1552 clock cycle initialization was controlled by a combination of a 18-state and 64-state one-hot state machine for Trivium with radix 1 and a single 18-state one-hot state machine for Trivium with radix 64. Alternative implementation, called Trivium-64 enhanced, employed state-machine consisting of one 2-state and two 3-state one-hot state machines. Trivium-64 enhanced gave slightly better results.

ASIC implementations of Phase 3 candidates, are reported in [31], considering many different performance metrics. They provide general guidelines for low-resource hardware stream ciphers, recommending a nonlinear filter function that is not demanding in terms of area complexity, mentioning the importance of feedback tap selection for the shift registers (to ease the replication of filtering function(s)), avoiding S-boxes, since they are significant consumer of area and power, among others.

As a conclusion, let us state that Grain triumphs with smallest area while Trivium clearly exhibits the highest throughput-to-area ratio. From hardware perspective, WG-16 has a larger internal state (512 bits) and way more complicated filtering function: it involves several multiplications and exponentiations in \mathbb{F}_2^{16} , which can hardly compare to addition and multiplication in \mathbb{F}_2 in Grain and Trivium. It is demanding in terms of both, area and time complexity, and prevents the increase in throughput by simply replicating the WGT-16 .

Stream Cipher	Source	Device	Area [# of Slices]*	Frequency [MHz]	radix	Throughput T [Mbps]	$\frac{T}{A}$
Grain	[26]	ASIC ($0.25\mu m$)	119.821*	300	16	4475	37346*
	[27]	XC2S15	48	105	1	105	2.19
	[29]†	Virtex-II	48	181	1	181	3.77
	[28]	XC3S50	122	193	1	193	1.58
			356	155	16	2480	6.97
			4911*	565	1	565	0.115*
			10548*	495	16	7920	0.751*
	[30]	Cyclone	219	242	1	242	1.11
			508	512	16	3440	6.77
	[20]	XC3S700A	318	177	1	177	0.56
	[31]	ASIC ($0.13\mu m$)	1294GE	724.6	1	724.6	-
			3239GE	617.3	16	9876.5	-
			1857GE	925.9	1	926	-
4617GE			452.5	32	14480	-	
[32]	XC3S50	44	196	1	196	4.45	
		348	130	16	2080	5.98	
		50	196	1	196	3.92	
		534	133	32	4256	7.97	
Trivium	[26]	ASIC ($0.25\mu m$)	144.128*	312	64	1856	128833*
	[27]	XC2S15	40	102	1	102	2.55
	[29]†	Virtex-II	41	207	1	207	5.05
	[28]	XC3S50	188	201	1	201	1.07
			388	190	64	12160	31.34
			7428*	840	1	840	0.113*
			13440*	800	64	51200	3.81*
	[30]	Cyclone	393	295	1	295	0.75
			710	245	64	15680	22.08
	††		700	255	64	16320	23.31
	[20]	XC3S700A	149	326	1	326	2.19
	[31]	ASIC ($0.13\mu m$)	2580GE	327.9	1	327.9	-
			1921GE	348.4	64	22299.6	-
[32]	XC3S50	50	240	1	240	4.8	
		344	211	64	13504	39.26	

Table 2.4: Implementation results for Grain and Trivium found in literature

- † marks implementations of Grain-128,
- †† marks the implementation of Trivium-64 enhanced,
- * denotes that the metric uses μm^2 instead of # of Slices

2.5.4 Composite field arithmetic

In this section we want to provide a brief overview of finite field arithmetic based on isomorphic tower constructions. We cover finite fields of small order and continue with large finite fields, including both software and hardware implementation. At the end of this section, we present the tower field implementations of AES in more detail, since they are closely related to the work presented in this thesis.

Let us begin with a paper [80] from 1974, published by D.H. Green and I.S. Taylor, that presents five tables listing irreducible polynomials of small degrees over finite fields \mathbb{F}_q of small order, specifically $q = 4, 8, 9, 16$. Their preferred method of representing the field elements powers of the generator, and they also provide primitive polynomials of small degrees over the aforementioned base fields. The remainder of the paper is dedicated to applications of composite field arithmetic in error-correcting codes and FSR-based sequence generators. Also one of the oldest applications of isomorphic tower constructions that is of our interest, is an inversion algorithm for elements of \mathbb{F}_{q^m} with $q = 2^n$, proposed in 1988 by T. Itoh and S. Tsuji [81]. Using normal bases for both extensions, they compute the inverse in \mathbb{F}_{q^m} using subfield inversion (performed by cyclic shifts over \mathbb{F}_2 and multiplications in \mathbb{F}_q), cyclic shifts over \mathbb{F}_q and multiplications in \mathbb{F}_{q^m} . At that time, many authors described multiplication and inversion in \mathbb{F}_{2^m} taking advantage of the arithmetic in the subfield $\mathbb{F}_{2^{\frac{m}{2}}}$, for example [82, 83, 84, 85].

There is a series of papers from the 90's published by Christof Paar [87, 88, 89, 90], reporting gate counts for VLSI implementations of finite field arithmetic in composite fields using polynomial basis representation for all extensions; most of these results are a part of his PhD thesis [86]. He provided block diagrams for parallel multipliers, based on Karatsuba-Oftman algorithm, over $\mathbb{F}_{((2^n)^m)}$ and their optimizations for special cases $\mathbb{F}_{((2^n)^2)}$ and $\mathbb{F}_{((2^n)^4)}$. In his work he adapts the Itoh-Tsuji approach to inversion and relates it to the work in [83]. He also provided tables of m , n and the primitive polynomial used for the second extension, resulting in the most efficient implementation for the particular \mathbb{F}_{2^k} , for $k = nm \leq 32$ with k even. Then in 1997, C. Paar published a paper [91] describing hybrid components that use parallel circuits for arithmetic in the underlying base field \mathbb{F}_{2^n} as building blocks for serial circuits performing the arithmetic in the top-level $\mathbb{F}_{((2^n)^m)}$. Further optimization was possible by subfield decomposition $\mathbb{F}_{2^n} \cong \mathbb{F}_{(2^{\frac{n}{2}})^2}$. This work was targeting larger finite fields of order $n \cdot m > 140$ with coprime n, m , for use in elliptic curve cryptography. The paper provides experimental results for elliptic curve arithmetic over $\mathbb{F}_{2^{152}} \cong \mathbb{F}_{(2^8)^{19}}$ for a $2\mu m$ ASIC implementation. He revisited Itoh-Tsuji inversion in large

fields in 2002 paper [92]: the extension fields were constructed using either all-one polynomials (AOPs) or equally-spaced polynomials (ESPs), the field elements represented in polynomial basis, and exponentiation in $\mathbb{F}_{((2^n)^m)}$ for coprime n, m optimized using iterates of the Frobenius map.

One of the oldest papers dealing with larger fields with the title “Public-key Cryptosystems with Very Small Key Lengths” is from 1993 [93]. It talks about elliptic-curve cryptosystems and one of their chosen underlying finite fields was $\mathbb{F}_{2^{104}}$, implemented as composite field $\mathbb{F}_{(2^8)^{13}}$. Elements of $\mathbb{F}_{(2^8)^{13}}$ were represented as polynomials over F_{2^8} and table lookup algorithms were used for arithmetic in the base field. The paper reports significant speed-up when compared to an implementation of elliptic curve arithmetic over $\mathbb{F}_{2^{105}}$ using normal basis, and concludes that the implementation with the 8-bit base field elements is very suitable for a software implementation.

A 1999 technical report on composite field arithmetic [94] by Savas and Koc reports software implementations for certain fields of the form $\mathbb{F}_{((2^n)^m)}$, with $n = 13, 14, 15, 16$ and m chosen so that $n \cdot m < 512$ and n, m coprime. They report comparison of total time needed for squaring, multiplication and inversion implemented using polynomial basis to optimal normal basis 1 (ONB1) or to optimal normal basis 2 (ONB2) representation. In all cases table look-up algorithms in polynomial basis representation were used. Multiplication was not conducted directly in ONB1/ONB2, instead they used converted the elements to a different basis representation where the multiplication was performed and then the product converted back to the ONB, namely to (a) shifted polynomial basis for ONB1 (for details refer to [95]), and (b) a permutation of the ONB2 for ONB2 (for details refer to [96]). For both cases they used inversion based on Extended Euclidean Algorithm in polynomial basis representation of elements, also needing basis conversion. For multiplication and inversion the purely polynomial basis implementation outperforms both ONBs, and as expected, squaring in PB is slower (but absolutely negligible in comparison with multiplication or inversion). In a paper [97] from 2003, Sunar, Savas and Koc provide methods for efficient conversion between the binary field \mathbb{F}_{2^k} and the composite field $\mathbb{F}_{((2^n)^m)}$, where $k = nm$ for large k and n, m coprime.

Recall the tower construction $\mathbb{F}_{(2^8)^4}$ used in Snow3G: the [14] specification of the cipher assumes an implementation using table lookups for the first level of the tower.

Use of tower field constructions for AES hardware implementations

In the past decade, many implementations of AES benefited from a tower construction of \mathbb{F}_{2^8} . In 2001, Rijmen proposed to use the tower construction $\mathbb{F}_{(2^4)^2}$ for the AES S-box, [34]. This tower construction was used by A. Rudra et. al [?] for both, hardware and software implementation of AES. They employed tower field arithmetic for the ByteSub and MixColumn transformations. Their hardware results (without specifying the process used) show a circuit only half the size of other AES implementations at that time and it achieves four times higher throughput. Also in 2001, Satoh et. al [36] described a compact data path architecture for AES using the tower field construction $\mathbb{F}_{((2^2)^2)^2}$ to perform the inversion within the S-boxes, and thus achieving a 20% smaller S-box than the authors of [?]. Two years later, Satoh and Morioka [37] published another paper, that identifies the S-box as the critical component from the point of view of power consumption; using a multi-stage Positive Polarity Reed-Muller form they optimized the aforementioned composite field S-box, reducing the power consumption from $136\mu W$ to $29\mu W$ at $10MHz$ using $0.13\mu m$ 1.5V CMOS technology. Mentens et. al [38] improves the original S-box of Satoh et. al [36] by choosing the irreducible polynomials that minimize the Hamming weight of the basis conversion matrices, the matrix for constant multiplication used in the inverter and the matrix for the affine transformation used in the S-box, leading to an 5% area reduction. All the aforementioned tower field constructions isomorphic to \mathbb{F}_{2^8} use polynomial bases at each level of the tower. D. Canright [39] conducted an exhaustive search and tree structure analysis to find the best matrices while testing both, the polynomial and the normal bases at each level of the tower $\mathbb{F}_{((2^2)^2)^2}$. In his work, Canright was focusing on area reduction and not examining the delay. In 2010 a mixed basis tower field construction for $\mathbb{F}_{((2^2)^2)^2}$ was reported by [41]; their Itoh-Tsuji inverters accept an input in normal basis representation and output its inverse represented in the polynomial basis. Their choice for the polynomial basis representation of the inverse was based on a slightly more efficient matrix for the affine transformation. They also emphasize the link between the HAMming weights of individual rows of transition matrices and between the critical path delays. A very interesting application of tower field constructions was presented in [42]. The authors propose to use random tower construction as a countermeasure against side-channel attacks. They chose the $\mathbb{F}_{(2^4)^2}$ and fixed the defining polynomial for the lower level \mathbb{F}_{2^4} . For the second extension they use a polynomial of the form $p(x) = x^2 + x + \lambda$, whereby the element $\lambda \in \mathbb{F}_{2^4}$ is chosen randomly so that $p(x)$ is primitive.

Chapter 3

WGP_T module and different field constructions

The architecture of WG-16 was given in Figure 2.4 in Section 2.4.1. It consists of three main parts: the LFSR, the WGP_T component and the FSM. These three components will be implemented as three separate modules, and from now on, when we speak of WGP_T we refer to the WGP_T module. From description of WG transformation (2.4.1) it is obvious, that the WGP_T module is the most demanding component in WG-16 so it received the most attention. To achieve the best tradeoff between performance and area, different implementations of WGP_T, based on different field constructions, were explored. In this Chapter we present the different field constructions and give the design for the WGP_T modules for each field construction. The implementation of the particular circuit follows in chapter 4.

The LFSR module remains unchanged regardless of the WGP_T implementation and will only be discussed in Chapter 4. The general description of the FSM was already given in Section 2.4.1 and detailed structure of FSM is closely related to each particular WGP_T implementation. The actual FSM module was implemented only for the promising WGP_T modules and will be discussed in Chapter 4 when appropriate.

The top view of the WGP-T module is given in Figure 3.1 on the right. The WG-16 transformation as depicted in Figure 3.1 consists of two parts: the WG-16 permutation $WGP-16(X^d)$ and the trace computation $Tr(\cdot)$. The 16-bit output $WGP=WGP-16(X^d)$ is used as a nonlinear feedback to the LFSR during the initialization phase. In the running phase, the WGP-T module produces one keybit, denoted WGT, by applying the trace function to the WGP signal, that is $WGT= WGT-16(WGP-16(X^d))$.

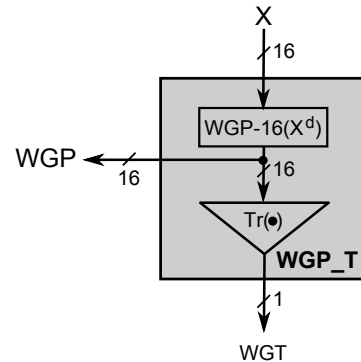


Figure 3.1: Architecture of module WGP-T

We begin this chapter with an overview of possible constructions of the finite field $\mathbb{F}_{2^{16}}$ and choose five constructions, that are discussed in five separate sections.

3.1 Finite field $\mathbb{F}_{2^{16}}$ - overview of field constructions

A simple approach is the construction of the finite field $\mathbb{F}_{2^{16}}$ as an algebraic extension of the prime field \mathbb{F}_2 , using the primitive polynomial $p(x) = x^{16} + x^5 + x^3 + x^2 + 1$. The root of $p(x)$ is a primitive element of $\mathbb{F}_{2^{16}}$, and will be denoted with ω , i.e. $p(\omega) = 0$. Hence, we write the polynomial basis of $\mathbb{F}_{2^{16}}$ over \mathbb{F}_2 as follows:

$$P = \{1, \omega, \omega^2, \dots, \omega^{15}\}$$

To find a normal basis we must first find a normal element; we use the following theorem, given in [74]:

Theorem 3.1 *Let θ be an element in \mathbb{F}_{q^m} . Then θ is a normal element of $\mathbb{F}_{q^m}/\mathbb{F}_q$ if and only if the polynomials $x^m - 1$ and $\sum_{i=0}^{m-1} \theta^{q^i} x^i$ in $\mathbb{F}_{q^m}[x]$ are relatively prime.*

Once we have found a normal element $\theta \in \mathbb{F}_{2^{16}}$, we can write down the normal basis:

$$N = \{\theta, \theta^2, \dots, \theta^{2^{15}}\}$$

Different normal elements generate different normal bases, and that has an impact on complexity of the arithmetic performed with field elements in normal basis representation.

To evaluate different normal bases we use the $(m \times m)$ matrix $T = [t_{ij}]$ over \mathbb{F}_2 , defined for a particular normal element $\theta \in \mathbb{F}_{2^{16}}$, in such a way that the coefficients t_{ij} satisfy:

$$\theta \cdot \theta^{q^i} = \sum_{j=0}^{m-1} t_{ij} \theta^{q^j} \quad \text{for } 0 \leq i \leq m-1. \quad (3.1)$$

The complexity C_N of normal basis generated by $\theta \in \mathbb{F}_{2^{16}}$ is defined as the number of nonzero elements in matrix T , for details see [70]. In general $C_N \geq 2m - 1$, and when $C_N = 2m - 1$ the normal basis is said to be *optimal* [75]. Matrix T is also called multiplication matrix, as will be explained in Section 4.3.1 in more detail; at this point we only mention that C_N corresponds to complexity of computation needed to obtain one coefficient of the product.

We perform an exhaustive search (with the GAP system [1]), using Theorem 3.1 and find 2048 normal elements in $\mathbb{F}_{2^{16}}$, and none of them generates an optimal normal basis; the minimum complexity that can be achieved is $C_N = 85$ [74]. The first normal element found is ω^{11} with $C_N = 123$. Complexity $C_N = 85$ was found for ω^{1117} , their T matrices are given in Appendix B.1.

Let us now turn our attention to tower field constructions. The subfield criterion 2.1 states that for a finite field \mathbb{F}_q with $q = p^m$ elements, there exists exactly one subfield of \mathbb{F}_q with p^n elements for each integer n that divides m . In case $m = 16$, possible values of n are 2, 4 and 8. The diagram in Figure 3.2 below shows possible constructions. The value on the connecting line represents the degree of extension $[F : K]$ for the extension F over K , depicted with this line. The degree of extension equals the degree of the irreducible polynomial used to construct the extension and is also the number of elements in the basis of F over K .

Following field constructions were explored in order to optimize the `WGP_T` module:

- construction of $\mathbb{F}_{2^{16}}$ with polynomial basis representation of elements (Section 3.2)
- construction of $\mathbb{F}_{2^{16}}$ with normal basis representation of elements (Section 3.3)
- tower construction $\mathbb{F}_{((2^2)^2)^2}$ (Section 3.4)
- tower construction $\mathbb{F}_{(2^4)^4}$ (Section 3.5)
- tower construction $\mathbb{F}_{(2^8)^2}$ (Section 3.6)

First construction listed above uses the polynomial basis with root ω of the defining polynomial $p(x)$ that was mentioned at the beginning of this section. For the second construction, the aforementioned ω^{1117} is selected to be the normal element generating the normal basis

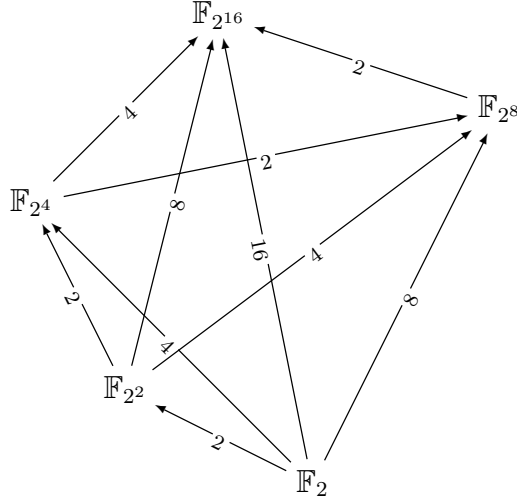


Figure 3.2: Finite field $\mathbb{F}_{2^{16}}$ - possible tower constructions

of $\mathbb{F}_{2^{16}}/\mathbb{F}_2$. The remaining three field constructions use towers of extension fields. If we recall the permutation polynomial $q(Y)$ (see equation (2.10) in Section 2.4.1) we see the common occurrence of exponentiation to powers of two: this operation can be performed very efficiently in normal basis representation, namely with a simple right cyclic shift. But that requires basis conversion between normal basis representation and tower field representation, and we will select the normal element that gives the most efficient conversion matrices. This will be discussed in more detail for each tower construction individually.

The rest of this chapter is dedicated to the five field constructions listed above, one construction per section. All the sections begin with a detailed description of the field construction and end with a schematic showing the top-level structure of the `WGP_T` module. The three tower constructions also need conversions between the tower field basis and normal basis, so another section describing the conversion is inserted before the `WGP_T` section. We will observe that all `WGP_T` modules, except the one using tower construction $\mathbb{F}_{((2^2)^2)^2}$, have basically the same top-level structure with only difference being the representation of element 1 using different bases. From implementation point of view, significant differences will occur, since different bases lead to different circuits for the basic field arithmetic. Algebraic optimization is possible for module `WGP_T` using tower construction $\mathbb{F}_{((2^2)^2)^2}$; this will be explained in more detail in Section 3.4.3.

3.2 $\mathbb{F}_{2^{16}}$ with polynomial basis

3.2.1 Field construction

Let us recall the polynomial basis construction of $\mathbb{F}_{2^{16}}$ over its prime subfield \mathbb{F}_2 from Section 3.1. The polynomial basis is the set $\{1, \omega, \omega^2, \dots, \omega^{15}\}$, where ω is a root of the irreducible polynomial $x^{16} + x^5 + x^3 + x^2 + 1$. Thus, a field element $A \in \mathbb{F}_{2^{16}}$ is represented as a linear combination of basis elements with coefficients $a_i, i = 0, \dots, 15$ from the prime subfield \mathbb{F}_2 :

$$A = \sum_{i=0}^{15} a_i \omega^i; a_i \in \mathbb{F}_2$$

3.2.2 WGP_T module

We begin by inspecting computation $\text{WGP-16}(X^d) = q(X^d \oplus_{16} 1) \oplus_{16} 1$, where X is an arbitrary nonzero element of $\mathbb{F}_{2^{16}}$ and $d = 1057$ the chosen decimation value. The decimation 1057 is represented as “10000100001” in binary and can be computed with 10 squarings and 2 multiplications. Let $Y = X^d \oplus_{16} 1 \in \mathbb{F}_{2^{16}}$. Inspecting the permutation polynomial

$$q(Y) = Y \oplus_{16} (Y^{2^{11}} \otimes_{16} Y) \oplus_{16} (Y^{2^{11}} \otimes_{16} Y^{2^6} \otimes_{16} Y) \oplus_{16} (Y^{2^6} \otimes_{16} Y^{-2^{11}} \otimes_{16} Y) \oplus_{16} (Y^{2^{11}} \otimes_{16} Y^{2^6} \otimes_{16} Y^{-1}) \quad (3.2)$$

we see that we require 11 squarers (6 to obtain Y^{2^6} and further 5 for $Y^{2^{11}}$), 7 multipliers and 2 inverters. Together with the 10 squarers and 2 multipliers required for the decimation X^d , we end up with a total of 21 squarers, 9 multipliers and 2 inverters. Since inversion is expensive, we replace two separate calculations of $(Y^{2^{11}})^{-1}$ and Y^{-1} by just one inversion Y^{-1} followed by the computation $(Y^{-1})^{2^{11}}$, i.e. we can omit one inverter at the cost of 11 squarers and now have 32 squarers in total. Using distributivity we rewrite equation (3.2) as

$$q(Y) = Y \oplus_{16} \left(Y \otimes_{16} (Y^{2^{11}} \oplus_{16} (Y^{-1})^{2^{11}} \otimes_{16} Y^{2^6}) \right) \oplus_{16} \left(Y^{2^6} \otimes_{16} Y^{2^{11}} \otimes_{16} (Y \oplus_{16} Y^{-1}) \right) \quad (3.3)$$

which reduces the number of multipliers from 7 to 4. The total $\text{WGP-16}(X^d)$ computation now requires 32 squarers, 6 multipliers and 1 inverter.

Using notation

- $Y = X^d \oplus_{16} 1$ with $d = 1057 = 2^{10} + 2^5 + 1$,
- $A = Y \otimes_{16} (Y^{2^{11}} \oplus_{16} (Y^{-1})^{2^{11}} \otimes_{16} Y^{2^6})$, and
- $B = Y^{2^6} \otimes_{16} Y^{2^{11}} \otimes_{16} (Y \oplus_{16} Y^{-1})$,

we can summarize WGT-16 with three simple equations:

- $q(Y) = Y \oplus_{16} A \oplus_{16} B$,
- $\text{WGP-16}(X^d) = q(Y) \oplus_{16} 1$ and finally
- $\text{WGT-16}(X^d) = \text{Tr}(\text{WGP-16}(X^d))$.

The architecture obtained following equation (3.3) can be seen in Figure 3.3.

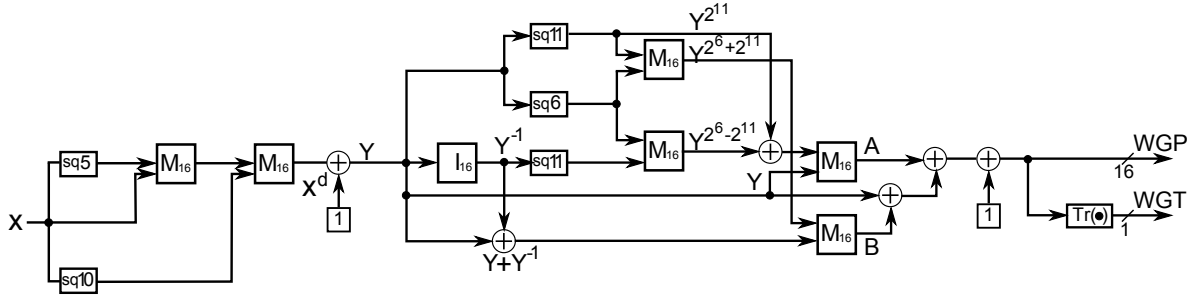


Figure 3.3: Module WGP_T for field elements in polynomial basis representation

For an arbitrary element $A \in \mathbb{F}_{2^{16}}$, using modular reduction by $\omega^{16} + \omega^5 + \omega^3 + \omega^2 + 1$, we obtain

$$\text{Tr}(A) = A \oplus_{16} A^2 \oplus_{16} A^{2^2} \oplus_{16} \dots \oplus_{16} A^{2^{15}} = a_{11} \oplus a_{13}, \text{ where } A = \sum_{i=0}^{15} a_i \omega^i.$$

3.3 $\mathbb{F}_{2^{16}}$ with normal basis

3.3.1 Field construction

The next field construction we are going to explore is the normal basis construction. Recall from Section 2.2.3 that a normal basis consist of conjugates of a normal element of $\mathbb{F}_{2^{16}}$ over \mathbb{F}_2 . In Section 3.1 we found 2048 normal elements in $\mathbb{F}_{2^{16}}$, none of which generate an optimal normal basis. Finding a normal basis of low complexity is essential for efficient multiplication of field elements ([70]). The normal element yielding the matrix T with lowest complexity $C_N = 85$ is the element ω^{1117} , where ω is the root of irreducible polynomial $p(x) = x^{16} + x^5 + x^3 + x^2 + 1$ (the matrix T for ω^{1117} can be seen at the end of Section B.1.2 in Appendix B.1.1). We use this element to generate the normal basis of $\mathbb{F}_{2^{16}}$ over \mathbb{F}_2 :

$$N = \{\theta, \theta^2, \dots, \theta^{2^{15}}\}, \text{ where } \theta = \omega^{1117} \in \mathbb{F}_{2^{16}} \text{ and } p(\omega) = 0.$$

An element $A \in \mathbb{F}_{2^{16}}$ is now represented as

$$A = \sum_{i=0}^{15} a_i \theta^{2^i}; a_i \in \mathbb{F}_2$$

3.3.2 WGP_T module

This section is organized as follows: we give the schematic for module `WGP_T` first and then explain the exponentiation to powers of two and the trace computation in detail.

The `WGP_T` module using normal basis representation of elements is very similar to the `WGP_T` module using polynomial basis, which was described in Section 3.2. The only differences between the two modules arise from the different bases used. Using normal basis representation of field elements:

- the exponentiation to powers of two can be efficiently implemented by a simple right cyclic shift,
- the element 1 is represented as $1 = \theta + \theta^2 + \dots + \theta^{2^{15}}$, that is as $(1, 1, \dots, 1)$, so the XORing of a field element with the constant 1 can be implemented by a simple bitwise NOT operator, and
- the trace of a field element $A \in \mathbb{F}_{2^{16}}$ can be computed as $\text{Tr}(A) = \bigoplus_{i=0}^{15} a_i$.

Recall the **WGT-16** presentation that was used to obtain the **WGP-T** module using polynomial basis in Section 3.2:

- $Y = X^d \oplus_{16} 1$ with $d = 1057 = 2^{10} + 2^5 + 1$,
- $A = Y \otimes_{16} (Y^{2^{11}} \oplus_{16} (Y^{-1})^{2^{11}} \otimes_{16} Y^{2^6})$,
- $B = Y^{2^6} \otimes_{16} Y^{2^{11}} \otimes_{16} (Y \oplus_{16} Y^{-1})$,
- $q(Y) = Y \oplus_{16} A \oplus_{16} B$,
- $\text{WGP-16}(X^d) = q(Y) \oplus_{16} 1$ and finally
- $\text{WGT-16}(X^d) = \text{Tr}(\text{WGP-16}(X^d))$.

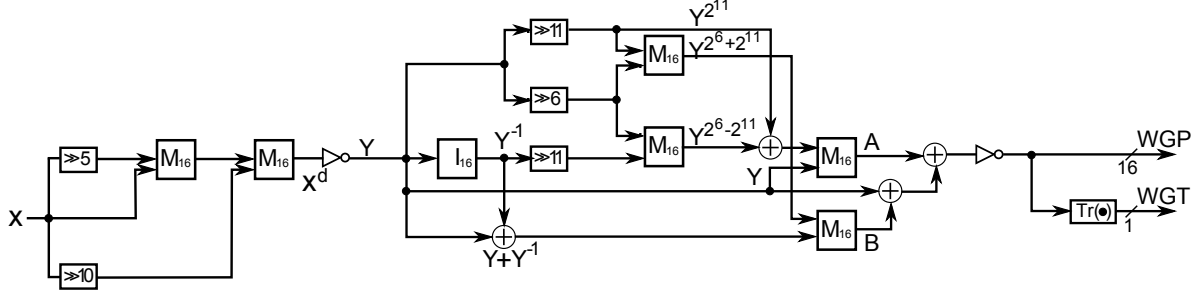


Figure 3.4: Module **WGP-T** for field elements in normal basis representation

Exponentiation to powers of two

For an element $A \in \mathbb{F}_{2^m}$, with coefficients $a_i \in \mathbb{F}_2$, A^{2^k} is computed as follows:

$$\begin{aligned}
 A^{2^k} &= \left(\sum_{i=0}^{m-1} a_i \alpha^{q^i} \right)^{2^k} = \sum_{i=0}^{m-1} a_i (\alpha^{q^i})^{2^k} \\
 &= a_0 \alpha^{2^k} + a_1 \alpha^{2 \cdot 2^k} + \dots + a_{m-2} \alpha^{2^{m-2} \cdot 2^k} + a_{m-1} \alpha^{2^{m-1} \cdot 2^k}
 \end{aligned}$$

For $k = 1, \dots, 15$, the $(m - k)$ -th term becomes the first term since

$$a_{m-k} \alpha^{2^{m-k} \cdot 2^k} = a_{m-k} \alpha^{2^m} = a_{m-k} \alpha$$

Similarly, the $(m - k + 1)$ -th term becomes the the second term and so on, hence a right cyclic shift by k positions. For $k = 1$, we get:

$$\begin{aligned} A^2 &= a_0\alpha^2 + a_1\alpha^{2^2} + \cdots + a_{m-2}\alpha^{2^{m-1}} + a_{m-1}\alpha^{2^m} \\ &= a_m\alpha + a_0\alpha^2 + a_1\alpha^{2^2} + \cdots + a_{m-2}\alpha^{2^{m-1}} \end{aligned}$$

In the Figure 3.4, the blocks denoted with $\gg k$ represent the exponentiation A^{2^k} .

Trace computation

For an arbitrary element $A \in \mathbb{F}_{2^{16}}$, represented using normal basis $N = \{\theta, \theta^2, \dots, \theta^{2^{15}}\}$, the trace is obtained as follows:

$$\begin{aligned} \text{Tr}(A) &= A \oplus_{16} A^2 \oplus_{16} A^{2^2} \oplus_{16} \dots \oplus_{16} A^{2^{15}} \\ &= (a_0\theta + a_1\theta^2 + \cdots + a_{15}\theta^{2^{15}}) \\ &+ (a_{15}\theta + a_0\theta^2 + \cdots + a_{14}\theta^{2^{15}}) \\ &+ \dots \\ &+ (a_1\theta + a_2\theta^2 + \cdots + a_0\theta^{2^{15}}) \\ &= \left(\sum_{i=0}^{15} a_i \right) \theta + \left(\sum_{i=0}^{15} a_i \right) \theta^2 + \cdots + \left(\sum_{i=0}^{15} a_i \right) \theta^{2^{15}} \\ &= \left(\sum_{i=0}^{15} a_i \right) (\theta + \theta^2 + \cdots + \theta^{2^{15}}) \\ &= \sum_{i=0}^{15} a_i \\ &= \bigoplus_{i=0}^{15} a_i \end{aligned}$$

From equation above we can see, that the trace function can be computed by XORing the coefficients of element A .

3.4 Tower construction $\mathbb{F}_{(((2^2)^2)^2)^2} \cong \mathbb{F}_{2^{16}}$

This section is divided into three subsections: the construction of the tower field, conversion between different basis representation of elements, and the module `WGP_T` itself.

3.4.1 Field construction

The tower construction $\mathbb{F}_{(((2^2)^2)^2)^2}$ uses extensions of degree two on each level of the tower, hence we need an irreducible polynomial of degree two on each step:

$$\mathbb{F}_2 \xrightarrow{e(x)} \mathbb{F}_{2^2} \xrightarrow{f(x)} \mathbb{F}_{(2^2)^2} \xrightarrow{g(x)} \mathbb{F}_{((2^2)^2)^2} \xrightarrow{h(x)} \mathbb{F}_{(((2^2)^2)^2)^2}.$$

The completed tower construction for $\mathbb{F}_{(((2^2)^2)^2)^2}$ is summarized in Table 3.1 below:

Finite Field \mathbb{F}_{2^n}	Normal Basis over $\mathbb{F}_{2(\frac{n}{2})}$	Normal element as power of ω	Defining polynomial
$\mathbb{F}_{2^{16}} \cong \mathbb{F}_{(((2^2)^2)^2)^2}$	$\{\delta, \delta^{256}\}$	$\delta = \omega^{45049}$	$h(x) = x^2 + x + \mu$, where $\mu = \beta + \lambda\gamma$
$\mathbb{F}_{2^8} \cong \mathbb{F}_{((2^2)^2)^2}$	$\{\gamma, \gamma^{16}\}$	$\gamma = \omega^{14392}$	$g(x) = x^2 + x + \lambda$, where $\lambda = \alpha^2\beta$
$\mathbb{F}_{2^4} \cong \mathbb{F}_{(2^2)^2}$	$\{\beta, \beta^4\}$	$\beta = \omega^{4369}$	$f(x) = x^2 + x + \alpha$
$\mathbb{F}_{2^2} \cong \mathbb{F}(2)^2$	$\{\alpha, \alpha^2\}$	$\alpha = \omega^{21845}$	$e(X) = x^2 + x + 1$

Table 3.1: Tower construction of $\mathbb{F}_{(((2^2)^2)^2)^2}$
 ω is a root of polynomial $x^{16} + x^5 + x^3 + x^2 + 1$, used to construct the isomorphic field F_{16}
 α a root of $e(x)$, β a root of $f(x)$, γ a root of $g(x)$ and δ a root of $h(x)$

A reader satisfied with information provided in Table 3.1 can proceed to Section 3.4.2.

The remainder of this section is organized as follows: first we give some additional theoretical results that can be applied to tower construction $\mathbb{F}_{(((2^2)^2)^2)^2}$. Lower levels of the tower have a small order and are easily manageable, hence described in more detail to illustrate the theory that was presented in Section 2.2. While moving up the tower we are forced to conduct an exhaustive search for irreducible polynomials.

Additional mathematical background

Noting that the degree of extensions equals the characteristic of the field, we list a theorem and its corollary that helps us in the search for irreducible polynomials, [69]:

Theorem 3.2 *Let $\sigma \in \mathbb{F}_q$ and $\text{char}(\mathbb{F}_q) = p$. Then the trinomial $x^p - x - \sigma$ is irreducible in $\mathbb{F}_q[x]$ if and only if it has no root in \mathbb{F}_q .*

Corollary 3.3 *With the notation of Theorem 3.2, the trinomial $x^p - x - \sigma$ is irreducible in $\mathbb{F}_q[x]$ if and only if $\text{Tr}_{\mathbb{F}_q}(\sigma) \neq 0$.*

We decide to use normal basis representation of elements at each level of the tower. This facilitates an efficient implementation of squaring using a simple right cyclic shift. To ensure that the irreducible polynomial used for the extension is also an N-polynomial we use the following result and present it as a fact:

Fact 3.4 [70, Corollary 4.20] *Let $f(x) = x^2 + a_1x + a_2$ be an irreducible quadratic polynomial over \mathbb{F}_q . Then $f(x)$ is a N-polynomial if and only if $a_1 \neq 0$.*

Extension field \mathbb{F}_{2^2}

For construction of the first extension \mathbb{F}_{2^2} we need a polynomial from $\mathbb{F}_2[x]$, that is irreducible over the prime field \mathbb{F}_2 , and has degree 2. Polynomials x^2 and $x^2 + x$ are obviously reducible and $x^2 + 1$ has root $1 \in \mathbb{F}_2$, so they cannot be used. We have to try a trinomial: the trinomial $e(x) = x^2 + x + 1$ is the only polynomial of degree 2 that is irreducible over \mathbb{F}_2 , hence the only polynomial that can be used to construct \mathbb{F}_{2^2} . Note that \mathbb{F}_{2^2} is a finite field with 4 elements and with \mathbb{F}_2 embedded as a subfield. So two elements of \mathbb{F}_{2^2} are already known, namely elements 0 and 1 from \mathbb{F}_2 . \mathbb{F}_{2^2} is constructed by adjoining the root of defining polynomial $e(x)$ to the elements $\{0, 1\}$ of the underlying base field \mathbb{F}_2 . We denote this root α , that is $e(\alpha) = 0$. The first element to join the set is α . Now all we have to do is to add all the elements that are needed for $\{0, 1, \alpha\}$ to become closed under addition and multiplication. From $e(\alpha) = \alpha^2 + \alpha + 1$ it follows that:

$$\alpha^2 = \alpha + 1, \tag{3.4}$$

so α^2 is the next element to join the set. The set $\{0, 1, \alpha, \alpha^2\}$ is now closed for addition, as can be seen from Table 3.3. We now check to see if this set is closed for multiplication by computing $\alpha^2 \cdot \alpha$, which can be obtained by simply multiplying equation (3.4) by α :

$$\alpha^3 = \alpha^2 + \alpha = 1. \tag{3.5}$$

Obtained product $\alpha^3 = 1$ is already contained in the set. We can also verify this using the finite field analog of Fermat's little theorem (equation (2.1)) with $q = 2^2$: $\alpha^{q-1} = \alpha^3 = 1$. Table 3.4 shows that the set $\{0, 1, \alpha, \alpha^2\}$ is indeed closed for multiplication. We have found all four elements of the finite field \mathbb{F}_{2^2} . From addition and multiplication tables given in Tables 3.3 and 3.4 respectively we can see that $(\mathbb{F}_{((2^2)^2)^2}, +)$ is a commutative group with additive identity 0 and $(\mathbb{F}_{((2^2)^2)^2} \setminus \{0\}, \cdot)$ is a commutative group with multiplicative identity 1. To satisfy all conditions that are listed in the definition 2.4 of a field in Section 2.2, the reader can check the distributivity of multiplication and addition.

By the fact 3.4, the polynomial $e(x) = x^2 + x + 1$ is also a N-polynomial, which makes its root α a normal element of $\mathbb{F}_{2^2}/\mathbb{F}_2$. An arbitrary element $A \in \mathbb{F}_{2^2}$ can be represented with normal basis $\{\alpha, \alpha^2\}$ as follows: $A = a_0\alpha + a_1\alpha^2$, $a_0, a_1 \in \mathbb{F}_2$. The elements of \mathbb{F}_{2^2} in their normal basis representation are given in Table 3.2, which was obtained using equation (3.4). The coordinates for element 1 (second row in the Table 3.2) were obtained from equation (3.5).

$A \in \mathbb{F}_{2^2}$	a_0	a_1
0	0	0
1	1	1
α	1	0
α^2	0	1

Table 3.2: Elements of \mathbb{F}_{2^2}

+	0	1	α	α^2
0	0	1	α	α^2
1	1	0	α^2	α
α	α	α^2	0	1
α^2	α^2	α	1	0

Table 3.3: Addition in \mathbb{F}_{2^2}

·	0	1	α	α^2
0	0	0	0	0
1	0	1	α	α^2
α	0	α	α^2	1
α^2	0	α^2	1	α

Table 3.4: Multiplication in \mathbb{F}_{2^2}

Extension field $\mathbb{F}_{(2^2)^2}$

The next level of the tower construction is $\mathbb{F}_{(2^2)^2}$, and we want to find the normal basis of $\mathbb{F}_{(2^2)^2}$ over \mathbb{F}_{2^2} . This is an extension of degree 2 so we expect the normal basis to contain two elements. We could just find a normal and over \mathbb{F}_{2^2} irreducible polynomial, that is a N-polynomial of degree two and use the fact that the conjugates of its root, that is the basis elements, sum up to 1 to complete the construction. This fact will be further explored in Section 3.4.3 later in this chapter.

■ **Remark:** Here we first take a little detour and explore the isomorphic field F_{2^4} to illustrate the subfield criterion 2.1 by presenting $\mathbb{F}_{(2^2)^2}$ as subfield of \mathbb{F}_{2^4} , and to show how the Frobenius mapping determines the subfield elements.

Extension of degree 4 over the prime field \mathbb{F}_2 - the finite field $\mathbb{F}_{2^4}/\mathbb{F}_2$

The finite field $\mathbb{F}_{(2^2)^2}$ is isomorphic to \mathbb{F}_{2^4} . The latter can be constructed using the irreducible polynomial x^4+x+1 and its root y . Element $A \in \mathbb{F}_{2^4}$ can be represented in the polynomial basis $\{1, y, y^2, y^3\}$ as a polynomial of degree less than four of the form $A = a_0 + a_1y + a_2y^2 + a_3y^3$ with coefficients $a_i \in \mathbb{F}_2, 0 \leq i \leq 3$. The 16 elements of \mathbb{F}_{2^4} , obtained using the relationship $y^4 + y + 1 = 0$, are given in Table B.1 in Appendix B.1.1.

\mathbb{F}_{2^2} as subfield of \mathbb{F}_{2^4} : The subfield criterion ensures that the field \mathbb{F}_{2^2} must be embedded in \mathbb{F}_{2^4} as a subfield. We use the Frobenius mapping σ_1^2 where $\sigma : \mathbb{F}_{2^4} \mapsto \mathbb{F}_{2^4}$ and $\sigma_1(x) = x^2$ to find this subfield:

$$\sigma_1^2(x) = x^4 = x \iff x \in \mathbb{F}_{2^2}$$

The results of σ_1^2 are listed in the fourth column of Table B.1 in Appendix B.1.1. We can see the four elements $0, 1, y^5$ and y^{10} that remain fixed under σ_1^2 ; they are the four elements of \mathbb{F}_{2^2} . We can check that the set $\{0, 1, y^5, y^{10}\}$ is closed under addition and multiplication by writing the addition and multiplication tables for these elements (Tables 3.5 and 3.6). From the tables it is trivial to check that $(\{0, 1, y^5, y^{10}\}, +, \cdot)$ has the properties of a field as listed in Definition 2.4, which means that $\{0, 1, y^5, y^{10}\}$ is a subfield of \mathbb{F}_{2^4} . We can also check that both y^5 and y^{10} are roots of polynomial $e(x) = x^2 + x + 1$.

$+$	0	1	y^5	y^{10}	\cdot	0	1	y^5	y^{10}	$e(y^5) = (y^5)^2 + y^5 + 1$
0	0	1	y^5	y^{10}	0	0	0	0	0	$= y^{10} + y^5 + 1$
1	1	0	y^{10}	y^5	1	0	1	y^5	y^{10}	$= 1 + y + y^2 + y + y^2 + 1$
y^5	y^5	y^{10}	0	1	y^5	0	y^5	y^{10}	1	$= 0$
y^{10}	y^{10}	y^5	1	0	y^{10}	0	y^{10}	1	y^5	$e(y^{10}) = (y^{10})^2 + y^{10} + 1$
										$= y^5 + y^{10} + 1$
										$= y + y^2 + 1 + y + y^2 + 1$
										$= 0$

Table 3.5: Addition in \mathbb{F}_{2^2} Table 3.6: Multiplication in \mathbb{F}_{2^2}

With mapping $\alpha \mapsto y^5$ we obtain a field isomorphism between $(\{0, 1, \alpha, \alpha^2\}, +, \cdot)$, from the first level of the tower construction, and the current subfield of interest $(\{0, 1, y^5, y^{10}\}, +, \cdot)$. Choosing $\alpha \mapsto y^{10}$ only results in a different isomorphism. Identifying the isomorphism between $(\{0, 1, \alpha, \alpha^2\}, +, \cdot)$ and $(\{0, 1, y^5, y^{10}\}, +, \cdot)$ is enough: it saves the time checking whether $(\{0, 1, y^5, y^{10}\}, +, \cdot)$ is a field. ■

Extension of degree 2 over the ground field \mathbb{F}_{2^2} - the finite field $\mathbb{F}_{(2^2)^2}$

To construct $\mathbb{F}_{(2^2)^2}$ as an extension of \mathbb{F}_{2^2} , we need a polynomial of degree 2 with coefficients from \mathbb{F}_{2^2} that is irreducible over \mathbb{F}_{2^2} . This polynomial will have the form $f(x) = f_2x^2 + f_1x + f_0 \in \mathbb{F}_{2^2}[x]$. The polynomial with $f_2 = f_1 = f_0 = 1$ is the polynomial $e(x)$ that was used to construct the base field \mathbb{F}_{2^2} itself and is obviously not irreducible over \mathbb{F}_{2^2} . The next logical choice is $f_2 = f_1 = 1$ and $f_0 = \alpha$, that is the trinomial $f(x) = x^2 + x + \alpha$. Following Theorem 3.2, we find that this polynomial does not have a root in \mathbb{F}_{2^2} (see Table 3.7) and is therefore irreducible over \mathbb{F}_{2^2} .

$A \in \mathbb{F}_{2^2}$	$f(A)$
0	α
1	α
α	α^2
α^2	α^2

Table 3.7: Values of $f(x) = x^2 + x + \alpha$ for elements of \mathbb{F}_{2^2}

We now know that we can construct $\mathbb{F}_{(2^2)^2}$ using $f(x) = x^2 + x + \alpha$, by adjoining its root β to elements in \mathbb{F}_{2^2} . Furthermore, due to 3.4, $f(x)$ is a N-polynomial and β is a normal element generating the normal basis $\{\beta, \beta^4\}$ of $\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}$.

■ **Remark:** Recall from Section 2.2.3 that an irreducible polynomial of degree 2 has two distinct simple roots, β and its conjugate β^4 . Closer inspection of polynomial $f(x)$ reveals the following: if we write $\alpha = y^5 = y + y^2$ we see that $f(y) = y^2 + y + y^5 = y^5 + y^5 = 0$, hence y is a root of $f(x)$ which implies $y = \beta$ or $y = \beta^4$. In either case, both β and β^4 are two of the four roots of irreducible polynomial $x^4 + x + 1$, which was used to construct $\mathbb{F}_{2^4}/\mathbb{F}_2$. Relationship $\beta^4 + \beta + 1 = 0$ can be used to check that conjugates of β are linearly independent and thus constitute a normal basis of $\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}$. ■

Using the normal basis $\{\beta, \beta^4\}$ we can represent an element $B \in \mathbb{F}_{(2^2)^2}$ as a polynomial $B = b_0\beta + b_1\beta^4$ with coefficients $b_0, b_1 \in \mathbb{F}_{2^2}$. Elements of $\mathbb{F}_{(2^2)^2}$ in their normal basis representation are given in the grey column in Table 3.8. They were obtained using relationships $\beta^2 + \beta + \alpha = 0$ and $\beta^4 + \beta + 1 = 0$. In the last column, we give the order of the element in the multiplicative group $\mathbb{F}_{(2^2)^2}^*$: we see that β is a primitive element of $\mathbb{F}_{(2^2)^2}$ and thus generates the entire $\mathbb{F}_{(2^2)^2}^*$.

The tower field basis for the tower of extensions $\mathbb{F}_2 \subset \mathbb{F}_{2^2} \subset \mathbb{F}_{(2^2)^2}$ is obtained by rewriting the coefficients of an element $B \in \mathbb{F}_{(2^2)^2}$ using normal bases $\{\beta, \beta^4\}$ of $\mathbb{F}_{(2^2)^2}$ over \mathbb{F}_{2^2} and $\{\alpha, \alpha^2\}$ of \mathbb{F}_{2^2} over \mathbb{F}_2 as follows:

$$\begin{aligned}
B &= b_0\beta + b_1\beta^4 \\
&= (b_{00}\alpha + b_{01}\alpha^2)\beta + (b_{10}\alpha + b_{11}\alpha^2)\beta^4 \\
&= b_{00}\alpha\beta + b_{01}\alpha^2\beta + b_{10}\alpha\beta^4 + b_{11}\alpha^2\beta^4 \\
&= b_{00}\beta^6 + b_{01}\beta^{11} + b_{10}\beta^9 + b_{11}\beta^{14}
\end{aligned}$$

In the last line of the expression above we used the relationships $\alpha = \beta^5$ and $\alpha^2 = \beta^{10}$. Note that elements $\beta^6, \beta^{11}, \beta^9$ and β^{14} are linearly independent which means that the set $\{\beta^6, \beta^{11}, \beta^9, \beta^{14}\}$ with four elements constitutes a basis of \mathbb{F}_{2^4} over \mathbb{F}_2 ; this is the tower

field basis we were looking for. The tower field representation of elements is given in the first four columns of Table 3.8: the coefficients $b_{00}, b_{01}, b_{10}, b_{11}$ correspond to basis elements $\beta^6, \beta^{11}, \beta^9, \beta^{14}$ respectively. For reference, the conversion matrices between the tower field basis and polynomial basis $\{1, \beta, \beta^2, \beta^3\}$ are given in Section B.1.1 in Appendix B.1.1.

tower field basis \mathbb{F}_{2^4} over \mathbb{F}_2				normal basis $\mathbb{F}_{(2^2)^2}$ over \mathbb{F}_{2^2}		B as power of β	order of B in $\mathbb{F}_{(2^2)^2}^*$
b_0^\dagger		b_1^\dagger		β	β^4		
β^6	β^{11}	β^9	β^{14}	b_0	b_1		
0	0	0	0	0	0	/	/
0	0	0	1	0	α^2	β^{14}	15
0	0	1	0	0	α	β^9	5
0	0	1	1	0	1	β^4	15
0	1	0	0	α^2	0	β^{11}	15
0	1	0	1	α^2	α^2	β^{10}	3
0	1	1	0	α^2	α	β^2	15
0	1	1	1	α^2	1	β^{13}	15
1	0	0	0	α	0	β^6	5
1	0	0	1	α	α^2	β^8	15
1	0	1	0	α	α	β^5	15
1	0	1	1	α	1	β^{12}	5
1	1	0	0	1	0	β	15
1	1	0	1	1	α^2	β^7	15
1	1	1	0	1	α	β^3	5
1	1	1	1	1	1	β^{15}	1

Table 3.8: Elements of $\mathbb{F}_{(2^2)^2}$ in tower field basis $\{\beta^6, \beta^{11}, \beta^9, \beta^{14}\}$ of $\mathbb{F}_{2^4}/\mathbb{F}_2$, in normal basis $\{\beta, \beta^4\}$ of $\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}$ - shaded grey and as powers of the generator β

■ **Remark:** The nonzero elements of subfield \mathbb{F}_{2^2} , embedded in $\mathbb{F}_{(2^2)^2}$, are represented as follows:

$$\begin{array}{lll}
 1 & = \beta + \beta^4 & \alpha & = \alpha \cdot 1 & \alpha^2 & = \alpha^2 \cdot 1 \\
 & = \beta^{15} & & = \alpha \cdot (\beta + \beta^4) & & = \alpha^2 \cdot (\beta + \beta^4) \\
 & & & = \alpha\beta + \alpha\beta^4 & & = \alpha^2\beta + \alpha^2\beta^4 \\
 & & & = \beta^5 & & = \beta^{10}
 \end{array}$$

■

Extension field $\mathbb{F}_{((2^2)^2)^2}$

For the extension $\mathbb{F}_{((2^2)^2)^2}$ over $\mathbb{F}_{(2^2)^2}$ we look for a polynomial of the form $g(x) = x^2 + x + \lambda \in \mathbb{F}_{(2^2)^2}[x]$. We want a simple expression for $\lambda \in \mathbb{F}_{(2^2)^2}$, so we try four different values composed of basis elements of $\mathbb{F}_{(2^2)^2}$ and of its subfield \mathbb{F}_{2^2} (refer to Table 3.8) and check if the polynomial is irreducible. The four candidates are listed in Table 3.9 below. Note that the four values λ_i , $i = 0, \dots, 3$, are exactly the elements of the tower field basis of \mathbb{F}_{2^4} over \mathbb{F}_2 .

Candidate λ_i	irreducible	primitive
$\lambda_1 = \alpha\beta = \beta^6$	✓	
$\lambda_2 = \alpha^2\beta = \beta^{11}$	✓	✓
$\lambda_3 = \alpha\beta^4 = \beta^9$	✓	
$\lambda_4 = \alpha^2\beta^4 = \beta^{14}$	✓	✓

Table 3.9: Candidates for irreducible polynomials $g(x) = x^2 + x + \lambda_i$ of degree 2 over $\mathbb{F}_{(2^2)^2}$

All four polynomials are irreducible, but only the two with a primitive constant term (λ_2 and λ_4) are also primitive. We choose $\lambda = \lambda_2 = \beta^{11} = \alpha^2\beta \in \mathbb{F}_{(2^2)^2}$.

■ **Remark:** Following corollary 3.3, let us compute the absolute trace of the constant term λ to show that $g(x)$ is indeed irreducible over $\mathbb{F}_{(2^2)^2}$:

$$\begin{aligned}
 \text{Tr}(\lambda) &= \lambda + \lambda^2 + \lambda^{2^2} + \lambda^{2^3} \\
 &= \beta^{11} + \beta^7 + \beta^{14} + \beta^{13} \\
 &= \alpha^2\beta + \beta + \alpha^2\beta^4 + \alpha^2\beta^4 + \alpha^2\beta + \beta^4 \\
 &= \beta + \beta^4 \\
 &= 1
 \end{aligned}$$

In the third line of the trace computation above, elements $\beta^{11}, \beta^7, \beta^{13}, \beta^{14} \in \mathbb{F}_{(2^2)^2}$ were represented in the normal basis $\{\beta, \beta^4\}$ (see the grey column of Table 3.8). ■

Elements of $\mathbb{F}_{((2^2)^2)^2}$ are represented with normal basis $\{\gamma, \gamma^{16}\}$, where γ is a root of $g(x)$, that is, an element $A \in \mathbb{F}_{((2^2)^2)^2}$ can be written as a linear combination $A = a_0\gamma + a_1\gamma^{16}$ with coefficients a_0, a_1 from the base field $\mathbb{F}_{(2^2)^2}$.

Extension field $\mathbb{F}_{(((2^2)^2)^2)^2}$

The last extension $\mathbb{F}_{(((2^2)^2)^2)^2}$ was obtained using the polynomial $h(x) = x^2 + x + \mu$, where $\mu = \beta + \lambda\gamma$. The constant term μ was chosen based on exhaustive search for the best

conversion matrices, described in Section 3.4.2. In Section 3.4.3 we show, that the absolute trace $\text{Tr}(\mu) = 1$, hence by corollary 3.3 the polynomial $h(x)$ is indeed irreducible over $\mathbb{F}_{((2^2)^2)^2}$.

The normal basis of $\mathbb{F}_{((2^2)^2)^2}$ over $\mathbb{F}_{((2^2)^2)^2}$ is the set $\{\delta, \delta^{256}\}$ with δ being the root of the polynomial $h(x)$, and the elements of $\mathbb{F}_{((2^2)^2)^2}$ can be represented as $A = a_0\delta + a_1\delta^{256}$, where $a_0, a_1 \in \mathbb{F}_{((2^2)^2)^2}$.

3.4.2 Conversion matrices

In Section 3.4 we have seen a bottom-up approach for the tower construction. In this section we descend back down the tower to find conversion matrices between the tower field basis representation of elements. We also describe the exhaustive search for the normal element of $\mathbb{F}_{2^{16}}$ that gives the most efficient conversion matrices between the tower field and normal basis representation.

If we recall the permutation polynomial $q(Y) = Y + Y^{2^{11}+1} + Y^{2^{11}+2^6+1} + Y^{2^6-2^{11}+1} + Y^{2^{11}+2^6-1}$ from Section 2.4.1, we see that exponentiation to powers of two is a very common operation, that can be performed either

- as a sequence of squarings (which was used in the field construction with polynomial basis in Section 3.2),
- or by transitioning to normal basis representation, performing the exponentiation in normal basis, and transitioning back to tower field representation of elements. Exponentiation to powers of two for elements in their normal basis representation was explained in Section 3.3: it is realized with a simple cyclic shift.

We decide for the second option: the cyclic shift is trivial, but we need basis conversion between normal basis representation and tower field representation of the elements.

For now, we will treat the finite field $\mathbb{F}_{2^{16}}$ over \mathbb{F}_2 as a vector space V of dimension 16 over \mathbb{F}_2 . We can represent the elements of $\mathbb{F}_{2^{16}}$ over \mathbb{F}_2 using

- polynomial basis P,
- normal basis N, and
- tower field basis T.

The conversion matrix is the matrix of identity map $id : V \mapsto V$ relative to the two bases [68]. We will denote the conversion matrix from basis A to basis B representation with M_B^A . The transition matrix in the opposite direction is simply the inverse matrix: $M_A^B = (M_B^A)^{-1}$.

Let us denote the vector space V in polynomial basis representation as V_P , in normal basis representation as V_N , and in tower field representation as V_T , and draw a diagram:

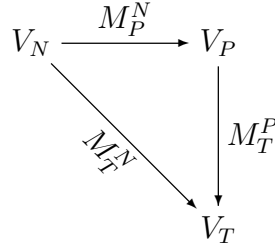


Figure 3.5: Conversion between normal basis and tower field representation

From the diagram in figure 3.5, we see that the transition from V_N to V_T is a composition of two identity maps $id_T^N = id_P^N \circ id_T^P$ and so we can obtain the matrix M_T^N of id_T^N by simple matrix multiplication:

$$M_T^N = M_T^P * M_P^N$$

We first compute the two transition matrices relative to polynomial basis $P = \{1, \omega, \dots, \omega^{15}\}$, that is uniquely defined with the root ω of the defining polynomial $x^{16} + x^5 + x^3 + x^2 + 1$ (see Section 3.1). To find the conversion matrices between the tower field representation and the polynomial basis representation let us now descend through the tower field to find the 16 basis elements of T . An element $A \in \mathbb{F}_{((2^2)^2)^2}$ can be written as: $A = d_0\delta + d_1\delta^{256}$, with coefficients $d_0, d_1 \in \mathbb{F}_{((2^2)^2)^2}$. Those can be written in basis $\{\gamma, \gamma^{16}\}$ of $\mathbb{F}_{((2^2)^2)^2}$ as $d_0 = c_{00}\gamma + c_{01}\gamma^{16}$ and $d_1 = c_{10}\gamma + c_{11}\gamma^{16}$. If we continue this procedure we obtain a tree structure that can be seen in Figure 3.6. The full expression for A is:

$$\begin{aligned}
A &= [((a_{0000}\alpha + a_{0001}\alpha^2)\beta + (a_{0010}\alpha + a_{0011}\alpha^2)\beta^4)\gamma \\
&\quad + ((a_{0100}\alpha + a_{0101}\alpha^2)\beta + (a_{0110}\alpha + a_{0111}\alpha^2)\beta^4)\gamma^{16}] \delta \\
&\quad + [((a_{1000}\alpha + a_{1001}\alpha^2)\beta + (a_{1010}\alpha + a_{1011}\alpha^2)\beta^4)\gamma \\
&\quad + ((a_{1100}\alpha + a_{1101}\alpha^2)\beta + (a_{1110}\alpha + a_{1111}\alpha^2)\beta^4)\gamma^{16}] \delta^{256}, \\
&\text{where } a_{0000}, a_{0001}, \dots, a_{1111} \in \mathbb{F}_2.
\end{aligned} \tag{3.6}$$

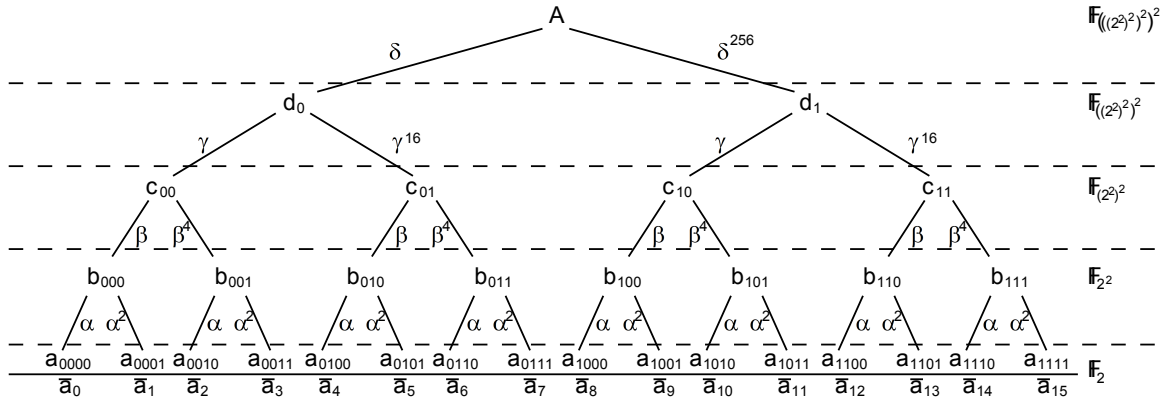


Figure 3.6: A tree structure for the element $A = \sum_{j=0}^{15} \bar{a}_j t_j$ in the tower construction $\mathbb{F}_{((2^2)^2)^2}$

Following the paths from the 16 leaves to the root of the tree in Figure 3.6, we can simply read the elements of basis $T = \{t_0, \dots, t_{15}\}$. If we think of the subscripts of coefficients $a_{0000}, a_{0001}, \dots, a_{1111} \in \mathbb{F}_2$ as 4-bit binary numbers, we can rewrite the leaf elements of the tree in Figure 3.6 as \bar{a}_j , $j = 0, \dots, 15$; corresponding elements \bar{a}_j can be seen at the bottom of the tree below the solid line. We can now write the field element $A \in \mathbb{F}_{((2^2)^2)^2}$ as a linear combination of basis T elements: $A = \sum_{j=0}^{15} \bar{a}_j t_j$. Considering the representation of the basis elements as a power the root ω of defining (primitive) polynomial of $\mathbb{F}_{2^{16}}$: $\alpha = \omega^{21845}$, $\beta = \omega^{4369}$, $\gamma = \omega^{14392}$ and $\delta = \omega^{45049}$, we obtain the basis elements t_j and their polynomial basis representations:

$$\begin{array}{llll}
t_0 & = & \alpha\beta\gamma\delta & = \omega^{20120} = \omega + \omega^6 + \omega^{10} + \omega^{12} + \omega^{13} \\
t_1 & = & \alpha^2\beta\gamma\delta & = \omega^{41965} = 1 + \omega + \omega^2 + \omega^3 + \omega^4 + \omega^5 + \omega^6 + \omega^9 + \omega^{10} + \omega^{11} + \omega^{12} + \omega^{15} \\
t_2 & = & \alpha\beta^4\gamma\delta & = \omega^{33227} = \omega^2 + \omega^3 + \omega^4 + \omega^7 + \omega^9 + \omega^{10} + \omega^{11} + \omega^{14} \\
t_3 & = & \alpha^2\beta^4\gamma\delta & = \omega^{55072} = 1 + \omega^2 + \omega^4 + \omega^7 + \omega^8 + \omega^{11} + \omega^{12} + \omega^{14} \\
t_4 & = & \alpha\beta\gamma^{16}\delta & = \omega^{39395} = 1 + \omega + \omega^2 + \omega^7 + \omega^8 + \omega^9 + \omega^{11} + \omega^{15} \\
t_5 & = & \alpha^2\beta\gamma^{16}\delta & = \omega^{61240} = \omega^2 + \omega^5 + \omega^6 + \omega^8 + \omega^{10} + \omega^{12} + \omega^{13} + \omega^{14} + \omega^{15} \\
t_6 & = & \alpha\beta^4\gamma^{16}\delta & = \omega^{52502} = \omega^2 + \omega^4 + \omega^6 + \omega^9 + \omega^{11} + \omega^{15} \\
t_7 & = & \alpha^2\beta^4\gamma^{16}\delta & = \omega^{8812} = \omega^3 + \omega^4 + \omega^5 + \omega^8 + \omega^9 + \omega^{10} + \omega^{11} + \omega^{14} \\
t_8 & = & \alpha\beta\gamma\delta^{256} & = \omega^{28590} = \omega + \omega^2 + \omega^4 + \omega^8 + \omega^{13} \\
t_9 & = & \alpha^2\beta\gamma\delta^{256} & = \omega^{50435} = \omega + \omega^3 + \omega^5 + \omega^6 + \omega^7 + \omega^8 + \omega^9 + \omega^{13} + \omega^{14} + \omega^{15} \\
t_{10} & = & \alpha\beta^4\gamma\delta^{256} & = \omega^{41697} = \omega^2 + \omega^3 + \omega^5 + \omega^9 + \omega^{11} + \omega^{12} \\
t_{11} & = & \alpha^2\beta^4\gamma\delta^{256} & = \omega^{63542} = \omega^3 + \omega^4 + \omega^6 + \omega^9 + \omega^{10} + \omega^{12} + \omega^{13} + \omega^{15} \\
t_{12} & = & \alpha\beta\gamma^{16}\delta^{256} & = \omega^{47865} = \omega^2 + \omega^3 + \omega^5 + \omega^6 + \omega^7 + \omega^{11} + \omega^{15} \\
t_{13} & = & \alpha^2\beta\gamma^{16}\delta^{256} & = \omega^{4175} = 1 + \omega^6 + \omega^7 + \omega^9 + \omega^{13} + \omega^{15} \\
t_{14} & = & \alpha\beta^4\gamma^{16}\delta^{256} & = \omega^{60972} = 1 + \omega^4 + \omega^{10} + \omega^{13} + \omega^{14} \\
t_{15} & = & \alpha^2\beta^4\gamma^{16}\delta^{256} & = \omega^{17282} = \omega + \omega^2 + \omega^3 + \omega^9 + \omega^{10} + \omega^{13}
\end{array}$$

Note that the elements t_0, \dots, t_{15} above can also be obtained from equation 3.6. We can now directly write the conversion matrix M_P^T from tower field representation to polynomial basis representation by writing the tower field elements t_j for $j = 0, \dots, 15$ in their polynomials representing as column vectors, starting with $t_0 = (0100001000101100)^T$, $t_1 = (1111111001111001)^T$, etc. Conversion matrix M_T^P is obtained as inverse of M_P^T .

$$\mathbf{M}_P^T = \begin{bmatrix}
0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0
\end{bmatrix}$$

$$\mathbf{M}_T^P = \begin{bmatrix}
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1
\end{bmatrix}$$

As already mentioned in Section 3.1 there are 2048 normal elements in $\mathbb{F}_{2^{16}}$. An exhaustive search revealed that the element $\theta = \omega^{1091}$ gives the optimal conversion matrices for the tower construction $\mathbb{F}_{((2^2)^2)^2}$. Using $\theta = \omega^{1091}$ we obtain the following normal basis $N = \{n_i\}$, where $n_i = \theta^{2^i}$ for $0 \leq i \leq 15$:

$$\begin{aligned}
n_0 &= \omega^{1091} &= 1 + \omega + \omega^5 + \omega^7 + \omega^{11} + \omega^{12} + \omega^{14} \\
n_1 &= \omega^{2182} &= 1 + \omega + \omega^2 + \omega^3 + \omega^4 + \omega^9 + \omega^{12} + \omega^{13} + \omega^{15} \\
n_2 &= \omega^{4364} &= \omega + \omega^2 + \omega^3 + \omega^4 + \omega^5 + \omega^6 + \omega^7 + \omega^8 + \omega^{11} + \omega^{12} + \omega^{14} + \omega^{15} \\
n_3 &= \omega^{8728} &= \omega^2 + \omega^3 + \omega^4 + \omega^5 + \omega^6 + \omega^9 + \omega^{13} + \omega^{14} + \omega^{15} \\
n_4 &= \omega^{174560} &= 1 + \omega^5 + \omega^7 + \omega^{12} + \omega^{13} \\
n_5 &= \omega^{34912} &= 1 + \omega^8 + \omega^{10} + \omega^{11} + \omega^{12} + \omega^{14} + \omega^{15} \\
n_6 &= \omega^{4289} &= 1 + \omega^3 + \omega^4 + \omega^5 + \omega^6 + \omega^7 + \omega^8 + \omega^{10} + \omega^{12} + \omega^{13} + \omega^{15} \\
n_7 &= \omega^{8578} &= 1 + \omega + \omega^5 + \omega^7 + \omega^8 + \omega^9 + \omega^{10} + \omega^{11} + \omega^{15} \\
n_8 &= \omega^{17156} &= 1 + \omega + \omega^4 + \omega^{10} + \omega^{11} \\
n_9 &= \omega^{34312} &= 1 + \omega^2 + \omega^4 + \omega^7 + \omega^{11} \\
n_{10} &= \omega^{3089} &= 1 + \omega^4 + \omega^6 + \omega^9 + \omega^{11} + \omega^{14} \\
n_{11} &= \omega^{6178} &= 1 + \omega + \omega^2 + \omega^3 + \omega^5 + \omega^7 + \omega^9 + \omega^{11} + \omega^{14} + \omega^{15} \\
n_{12} &= \omega^{12356} &= \omega^2 + \omega^5 + \omega^6 + \omega^7 + \omega^9 + \omega^{10} + \omega^{11} + \omega^{12} + \omega^{14} + \omega^{15} \\
n_{13} &= \omega^{24712} &= 1 + \omega^4 + \omega^5 + \omega^6 + \omega^8 + \omega^{13} + \omega^{14} + \omega^{15} \\
n_{14} &= \omega^{49424} &= 1 + \omega^3 + \omega^5 + \omega^6 + \omega^{12} + \omega^{13} \\
n_{15} &= \omega^{33313} &= 1 + \omega^6 + \omega^8 + \omega^{10} + \omega^{11} + \omega^{15}
\end{aligned}$$

From elements n_i in their polynomial basis representation we can write the conversion matrix M_P^N and obtain the inverse M_N^P :

$$\mathbf{M}_P^N = \begin{bmatrix}
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0
\end{bmatrix}$$

$$\mathbf{M}_N^P = \begin{bmatrix}
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0
\end{bmatrix}.$$

Finally we can compute $M_T^N = M_T^P * M_P^N$ and its inverse M_N^T :

$$\mathbf{M}_T^N = \begin{bmatrix}
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}$$

$$\mathbf{M}_N^T = \begin{bmatrix}
1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1
\end{bmatrix}.$$

Using the normal element $\theta = \omega^{1091}$ and the tower construction from previous Section we obtained optimal (i.e. minimum Hamming weight) conversion matrices M_T^N and M_N^T with Hamming weights 92 and 100 respectively.

The constant term $\mu = \beta + \lambda\gamma$ for the polynomial $h(x) = x^2 + x + \mu$ used to construct the extension $\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$ was also chosen based on exhaustive search involving different normal elements and different values of μ (taking primitive elements of $\mathbb{F}_{(2^2)^2}$ as candidates for μ) with minimal sum of the Hamming weights of M_T^N and M_N^T as search criterion. Other choices of μ in combination with different normal elements θ gave slightly worse results. For example when constructing the last level of the tower using $\mu = \gamma^{11}$ as the constant term in polynomial $h(x)$, the best conversion matrices are obtained for normal element ω^{713} ; their Hamming weights are 92 and 108, which is slightly worse than chosen $\theta = \omega^{1091}$ with $\mu = \beta + \lambda\gamma$.

3.4.3 Module WGP_T

The tower field construction $\mathbb{F}_{((2^2)^2)^2}$, as presented in previous Section (3.4.1), gives rise to some interesting properties of the trace function, that allow for optimization of the WGP_T module. This section is composed of three parts:

- Section 3.4.3 gives a retrospective on the tower construction using normal basis, ending with tower field basis representation of element 1,

- in Section 3.4.3 we explore the trace function and show how trace computation benefits from the regularity of the tower construction $\mathbb{F}_{(((2^2)^2)^2)^2}$, and finally
- in Section 3.4.3 we use the results from Section 3.4.3 to optimize the WGP_T module.

By re-examining the tower field $\mathbb{F}_{(((2^2)^2)^2)^2}$ we observe a highly regular structure with a similar construction for each extension; we can introduce the following notation, which applies to each level of the tower, and will be used throughout this section:

Let $F = \mathbb{F}_{q^2}$ be an extension of degree 2 over field $K = \mathbb{F}_q$, obtained using the defining polynomial $p(x) = x^2 + x + \sigma$ with $\sigma \in K$. The normal basis of F over K is the set $\{\rho, \rho^q\}$, where ρ is a root of $p(x)$. This yields the following representation for an arbitrary field element $A \in F$: $A = a_0\rho + a_1\rho^q$. Then, the trace of element $A \in F$ with respect to K is computed as $\text{Tr}_K^F(A) = A + A^q$.

Relationship between the trinomial $p(x)$ and the normal basis $\{\rho, \rho^q\}$

It is well known that the normal basis elements sum up to 1 (i.e. $\rho + \rho^q = 1$, a relationship that was used throughout Section 3.4.1), or in other words, that the trace of a basis element with respect to K equals 1, that is $\text{Tr}_K^F(\rho) = \text{Tr}_K^F(\rho^q) = 1$. In this section we give a lemma arising from the relationship between the trinomial $p(x)$ and the normal basis $\{\rho, \rho^q\}$, and then use this lemma to show that $\rho + \rho^q = 1$ really holds. Using this relationship for all levels of $\mathbb{F}_{(((2^2)^2)^2)^2}$, we show that the element 1, represented in the tower field basis, is a 16-bit vector of ones.

■ **Remark:** Detailed analysis of the trace function in tower construction $\mathbb{F}_{(((2^2)^2)^2)^2}$

We analyze the trace of the basis elements for each level of the tower construction $\mathbb{F}_{(((2^2)^2)^2)^2}$.

The first level of the tower is the extension $F = \mathbb{F}_{2^2}$ over the base field $K = \mathbb{F}_2$ with defining polynomial $e(x) = x^2 + x + 1$ and normal basis $\{\alpha, \alpha^2\}$. From $e(\alpha) = 0$ we obtain the relationship $\alpha^2 + \alpha + 1 = 0$, which yields $\text{Tr}_K^F(\alpha) = \alpha + \alpha^2 = 1$ and $\text{Tr}_K^F(\alpha^2) = \alpha^2 + \alpha^4 = 1$.

The next extension is $\mathbb{F}_{(2^2)^2}$ over \mathbb{F}_{2^2} with defining polynomial $f(x) = x^2 + x + \alpha$. Using previously introduced notation: $F = \mathbb{F}_{(2^2)^2}$, $K = \mathbb{F}_{2^2}$, $q = 4$, $\sigma = \alpha$ and a root of defining polynomial $\rho = \beta$ giving the normal basis $\{\beta, \beta^4\}$. Again, we obtain a relationship $\beta^2 + \beta + \alpha = 0$ which allows for $\beta^4 = (\beta^2)^2 = (\beta + \alpha)^2 = \beta^2 + \alpha^2 = \beta + \alpha + \alpha^2 = \beta + 1$. Using this last relationship, $\beta^4 = \beta + 1$, we obtain the traces of basis elements β and β^4 with respect to the subfield K , namely $\text{Tr}_K^F(\beta) = \beta + \beta^4 = 1$ and $\text{Tr}_K^F(\beta^4) = \beta^4 + \beta^{16} = \beta^4 + \beta = 1$.

Moving up another level of the tower field, we have $F = \mathbb{F}_{((2^2)^2)^2}$, $K = \mathbb{F}_{(2^2)^2}$, $q = 16$, $\sigma = \lambda$ and a root $\rho = \gamma$ of the defining polynomial $x^2 + x + \lambda$, giving the normal basis $\{\gamma, \gamma^{16}\}$. From relationship $\gamma^2 = \gamma + \lambda$ we obtain an expression for the basis element γ^{16} as a sequence of squares: $\gamma^2, \gamma^4 = (\gamma^2)^2, \gamma^8 = (\gamma^4)^2, \gamma^{16} =$

$(\gamma^8)^2 = \gamma + \lambda + \lambda^2 + \lambda^4 + \lambda^8$. Using $\lambda = \beta^{11}$ (see Section 3.4.1), we can compute $\lambda + \lambda^2 + \lambda^4 + \lambda^8 = 1$, thus obtaining expected relationship $\gamma^{16} = \gamma + 1$. Note that $\mathbb{F}_{((2^2)^2)^2}$ is a field with 256 elements and hence $\gamma^{256} = \gamma$. The traces of basis elements follow: $\text{Tr}_K^F(\gamma) = \gamma + \gamma^{16} = 1$ and $\text{Tr}_K^F(\gamma^{16}) = \gamma^{16} + \gamma^{256} = 1$.

The top level extension is $F = \mathbb{F}_{(((2^2)^2)^2)^2}$ over $K = \mathbb{F}_{((2^2)^2)^2}$ with $q = 256$, $\sigma = \mu$ and basis $\{\delta, \delta^{256}\}$ where δ is a root of polynomial $x^2 + x + \mu$, yielding the relationship $\delta^2 = \delta + \mu$. The expression for δ^{256} can be obtained by squaring δ :

$$\begin{aligned}
\delta^2 &= \delta + \mu \\
\delta^4 &= \delta + \mu + \mu^2 \\
\delta^8 &= \delta + \mu + \mu^2 + \mu^4 \\
\delta^{16} &= \delta + \mu + \mu^2 + \mu^4 + \mu^8 \\
\delta^{32} &= \delta + \mu + \mu^2 + \mu^4 + \mu^8 + \mu^{16} \\
\delta^{64} &= \delta + \mu + \mu^2 + \mu^4 + \mu^8 + \mu^{16} + \mu^{32} \\
\delta^{128} &= \delta + \mu + \mu^2 + \mu^4 + \mu^8 + \mu^{16} + \mu^{32} + \mu^{64} \\
\delta^{256} &= \delta + \mu + \mu^2 + \mu^4 + \mu^8 + \mu^{16} + \mu^{32} + \mu^{64} + \mu^{128}
\end{aligned}$$

i	μ^{2^i}	γ	γ^{16}
0	μ	β^6	β
1	μ^2	β^{10}	β^6
2	μ^4	1	β^3
3	μ^8	β^7	β^5
4	μ^{16}	β	β^6
5	μ^{32}	β^6	β^{10}
6	μ^{64}	β^3	1
7	μ^{128}	β^5	β^7
sum	$\sum_{i=0}^7 \mu^{2^i}$	1	1

Table 3.10: Elements $\mu^{2^i} \in \mathbb{F}_{((2^2)^2)^2}$ for $i = 0, \dots, 7$ and their sum

We obtain $\delta^{256} = \delta + \sum_{i=0}^7 \mu^{2^i}$. The elements $\mu^{2^i} \in \mathbb{F}_{((2^2)^2)^2}$ represented in normal basis $\{\gamma, \gamma^{16}\}$ and their sum are summarized in Table 3.10 on the right. We obtain $\sum_{i=0}^7 \mu^{2^i} = \gamma + \gamma^{16} = 1$, which results in $\delta^{256} = \delta + 1$, and furthermore $\text{Tr}_K^F(\delta) = \delta + \delta^{256} = 1$ and $\text{Tr}_K^F(\delta^{256}) = \delta^{256} + \delta^{65536} = \delta^{256} + \delta = 1$. ■

The purpose of this detailed analysis was to identify an interesting connection between the basis elements $\{\rho, \rho^q\}$, which holds on every level of construction $\mathbb{F}_{(((2^2)^2)^2)^2}$, which we present in the following Lemma:

Lemma 1 *Let $F = \mathbb{F}_{q^2}$ be an extension of degree 2 over field $K = \mathbb{F}_q$, obtained using the defining polynomial $p(x) = x^2 + x + \sigma$ with $\sigma \in K$, where $q = 2^m$ for some integer m . Let $\{\rho, \rho^q\}$ be the normal basis of F over K , where ρ is a root of $p(x)$. Then, the following relationship holds:*

$$\rho^q = \rho + \text{Tr}_K(\sigma),$$

where $\text{Tr}_K(\sigma)$ is the absolute trace of the constant term σ of defining polynomial $p(x)$ of F over K . Furthermore, $\text{Tr}_K(\sigma) = 1$.

Proof

$$\rho + \rho^q = \rho + (\rho^2 + \rho^2) + (\rho^2 + \rho^2) + \cdots + (\rho^{\frac{q}{2}} + \rho^{\frac{q}{2}}) + \rho^q \quad (3.7)$$

$$= (\rho + \rho^2) + (\rho^2 + \rho^4) + \cdots + (\rho^{\frac{q}{2}} + \rho^q) \quad (3.8)$$

$$= (\rho + \rho^2) + (\rho^2 + \rho^4) + \cdots + (\rho^{2^{m-1}} + \rho^{2^m}) \quad (3.9)$$

$$= (\rho + \rho^2) + (\rho + \rho^2)^2 + \cdots + (\rho + \rho^2)^{2^{m-1}} \quad (3.10)$$

$$= \sigma^{2^0} + \sigma^{2^1} + \cdots + \sigma^{2^{m-1}} \quad (3.11)$$

$$= \text{Tr}_K(\sigma) \quad (3.12)$$

For clarity, we assumed $m > 2$, but the argument above holds for any positive integer m . In the first line (3.8) we used the fact that in binary fields, $A + A = 0$ holds for an arbitrary field element $A \in \mathbb{F}$. In next two lines, we simply regrouped the terms, and then inserted $q = 2^m$ in line (3.10). In line (3.11) we use the fact that since ρ is a root of $p(x)$, we have $\rho^2 + \rho = \sigma$, and it is evident that this is the exact definition of $\text{Tr}_K(\sigma)$ for $K = \mathbb{F}_q$, where $q = 2^m$.

Finally, recall corollary 3.3, which uses $\text{Tr}_K(\sigma) \neq 0$ as a condition for the irreducibility of the polynomial $p(x)$ and that the absolute trace is a mapping onto the prime subfield, which in our case is the field $\mathbb{F}_2 = \{0, 1\}$, hence $\text{Tr}_K(\sigma) = 1$. \square

Lemma 1 basically shows that $\rho + \rho^q = 1$, and since $\text{Tr}_K^F(\rho) = \rho^{q^0} + \rho^{q^1} = \rho + \rho^q$ and $\text{Tr}_K^F(\rho^q) = (\rho^q)^{q^0} + (\rho^q)^{q^1} = \rho^q + \rho^{q^2} = \rho^q + \rho$, it follows that $\text{Tr}_K^F(\rho) = \text{Tr}_K^F(\rho^q) = 1$. The latter will be of importance when computing the trace function of a product of two field elements (see equation 3.14).

Tower field representation of element 1

Element 1 is an element of the prime field \mathbb{F}_2 , and is also an element of fields \mathbb{F}_{2^2} , $\mathbb{F}_{(2^2)^2}$, $\mathbb{F}_{((2^2)^2)^2}$ and $\mathbb{F}_{(((2^2)^2)^2)^2}$, hence can be represented as an element of each of these fields. Here we want to take a closer look at the tower field basis $T = \{t_0, t_1, \dots, t_{15}$ representation element $1 \in \mathbb{F}_{(((2^2)^2)^2)^2}$. Recall that the tower field basis T was derived in Section 3.4.2 with help of equation 3.6 and Figure 3.6. To distinguish between the element 1 in different (sub)fields \mathbb{F}_{2^2} , $\mathbb{F}_{(2^2)^2}$, $\mathbb{F}_{((2^2)^2)^2}$ and $\mathbb{F}_{(((2^2)^2)^2)^2}$, we introduce a notation using indices: $1_1 \in \mathbb{F}_2$, $1_2 \in \mathbb{F}_{2^2}/\mathbb{F}_2$, $1_4 \in \mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}$, $1_8 \in \mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$, and finally $1_{16} \in \mathbb{F}_{(((2^2)^2)^2)^2}/\mathbb{F}_{((2^2)^2)^2}$. Keeping in mind that we used a normal basis representation of elements with basis $\{\rho, \rho^q\}$ for each extension in the composite field $\mathbb{F}_{(((2^2)^2)^2)^2}$, we obtain the following representations of element 1:

- $1_2 = 1_1\alpha + 1_1\alpha^2 \in \mathbb{F}_{2^2}/\mathbb{F}_2$,
- $1_4 = 1_2\beta + 1_2\beta^4 \in \mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}$,

- $1_8 = 1_4\gamma + 1_4\gamma^{16} \in \mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$, and
- $1_{16} = 1_8\delta + a_8\delta^{256} \in \mathbb{F}_{(((2^2)^2)^2)^2}/\mathbb{F}_{((2^2)^2)^2}$.

Using relationships above, we obtain the following:

$$\begin{aligned}
1_{16} &= 1_8\delta + a_8\delta^{256} \\
&= (1_4\gamma + 1_4\gamma^{16})\delta + (1_4\gamma + 1_4\gamma^{16})\delta^{256} \\
&= ((1_2\beta + 1_2\beta^4)\gamma + (1_2\beta + 1_2\beta^4)\gamma^{16})\delta \\
&\quad + ((1_2\beta + 1_2\beta^4)\gamma + (1_2\beta + 1_2\beta^4)\gamma^{16})\delta^{256} \\
&= [((1_1\alpha + 1_1\alpha^2)\beta + (1_1\alpha + 1_1\alpha^2)\beta^4)\gamma \\
&\quad + ((1_1\alpha + 1_1\alpha^2)\beta + (1_1\alpha + 1_1\alpha^2)\beta^4)\gamma^{16}]\delta \\
&\quad + [((1_1\alpha + 1_1\alpha^2)\beta + (1_1\alpha + 1_1\alpha^2)\beta^4)\gamma \\
&\quad + ((1_1\alpha + 1_1\alpha^2)\beta + (1_1\alpha + 1_1\alpha^2)\beta^4)\gamma^{16}]\delta^{256} \\
&= 1_1\alpha\beta\gamma\delta + 1_1\alpha^2\beta\gamma\delta + \cdots + 1_1\alpha^2\beta^4\gamma^{16}\delta^{256} \\
&= 1_1t_0 + 1_1t_1 + \cdots + 1_1t_{15}
\end{aligned}$$

In the last two rows we skipped the detailed expansion; a reader can check the tower field basis elements t_j in Section 3.4.2. The element 1_{16} , represented in this basis, is a 16-bit vector of ones. Furthermore, the element 1 is a vector of ones at each level of this tower of extensions. Same holds for any composite field construction using normal basis at each level of the tower. Let us conclude this discussion with a practical consequence: in the tower construction $\mathbb{F}_{(((2^2)^2)^2)^2}$ element 1 is represented as $(1, 1, \dots, 1)$, hence adding the element 1 is achieved by a simple bitwise NOT operator.

Regularity of $\mathbb{F}_{(((2^2)^2)^2)^2}$ and transitivity of trace

Results presented in the current Section were also published in [6]. The regularity of tower construction $\mathbb{F}_{((2^2)^2)^2}$ was discussed in the previous Section 3.4.3: the trinomials used to construct all four extensions have the same form, namely $x^2 + x + \sigma$, of course with a different σ at each level. This symmetry has a nice consequence: multiplication of two field elements can be described with one equation for all levels of the tower field. Similar holds for squaring and inversion. Detailed description of these basic arithmetic operations will follow in Section 4.4.1. Schematic of the circuit performing multiplication as dictated by the equation (3.13) below can be seen in Figure 4.21(b) in Section 4.4.1.

The product of $A, B \in F$, in their normal basis representation $A = a_0\rho + a_1\rho^q$ and $B = b_0\rho + b_1\rho^q$, $a_i, b_i \in K$, $i = 0, 1$, can be computed as follows:

$$AB = ((a_0 + a_1)(b_0 + b_1)\sigma + a_0b_0)\rho + ((a_0 + a_1)(b_0 + b_1)\sigma + a_1b_1)\rho^q \quad (3.13)$$

Using this equation we obtain the following expression for the trace of the product with respect to subfield K :

$$\begin{aligned} \text{Tr}_K^F(AB) &= \text{Tr}_K^F(((a_0 + a_1)(b_0 + b_1)\sigma + a_0b_0)\rho + ((a_0 + a_1)(b_0 + b_1)\sigma + a_1b_1)\rho^q) \\ &= ((a_0 + a_1)(b_0 + b_1)\sigma + a_0b_0)\text{Tr}_K^F(\rho) + ((a_0 + a_1)(b_0 + b_1)\sigma + a_1b_1)\text{Tr}_K^F(\rho^q) \\ &= (a_0 + a_1)(b_0 + b_1)\sigma + a_0b_0 + (a_0 + a_1)(b_0 + b_1)\sigma + a_1b_1 \\ &= a_0b_0 + a_1b_1 \end{aligned} \quad (3.14)$$

In computation 3.14 above, trace properties *i.* and *ii.* from Theorem 2.3 were used to obtain the second line of equation. The third line follows by using $\text{Tr}_K^F(\rho) = \text{Tr}_K^F(\rho^q) = 1$, upon which the two terms containing σ cancel out, yielding the result.

Recall that $[F:K]=2$, which means that $F \cong \mathbb{F}_{2^n}$ and $K \cong \mathbb{F}_{2^{\frac{n}{2}}}$ for $n = 2, 4, 8, 16$. Using the symbol $\bigoplus_{\frac{n}{2}}$ for addition and $\bigotimes_{\frac{n}{2}}$ for multiplication of two elements in $\mathbb{F}_{\frac{n}{2}}$ we can write a generalized form for the trace expression in 3.14 as follows:

$$\text{Tr}_K^F(A \otimes_n B) = (a_0 \otimes_{\frac{n}{2}} b_0) \bigoplus_{\frac{n}{2}} (a_1 \otimes_{\frac{n}{2}} b_1) \quad (3.15)$$

As we are slowly closing in on a circuit for the WGP_T module, we switch from the implicate notation of a product to the use of $\bigotimes_{\frac{n}{2}}$ and replace $+$ with $\bigoplus_{\frac{n}{2}}$. At this high level of abstraction we regard both operators as 2-input/1-output gates. But there is an important difference: the multiplication $\bigotimes_{\frac{n}{2}}$, as given in equation 3.13 with $q = 2^{\frac{n}{2}}$, is quite complicated compared to the addition $\bigoplus_{\frac{n}{2}}$, which is in binary fields performed by a simple bitwise XOR gate. We now show that we can obtain the trace of the product without actually computing the product itself.

Proposition 1 *The absolute trace of the product AB of arbitrary field elements $A, B \in \mathbb{F}_{((2^2)^2)^2}$ can be computed as modulo-2 sum of the coordinates of the bitwise AND of elements $A = (a_0 \dots a_{15})$ and $B = (b_0 \dots b_{15})$, that is*

$$\text{Tr}(AB) = \bigoplus_{i=0}^{15} \left(a_i \odot_1 b_i \right).$$

Proof For simplicity we denote the levels of the tower construction with $K_0 = \mathbb{F}_2$, $K_1 = \mathbb{F}_{2^2}$, $K_2 = \mathbb{F}_{(2^2)^2}$, $K_3 = \mathbb{F}_{((2^2)^2)^2}$, and $K_4 = \mathbb{F}_{(((2^2)^2)^2)^2}$. Using an example, let us introduce a notation to simplify the computation below: the 16-bit vector $(a_0 a_1 \dots a_{15})$, denoted with $a_{0\dots 15}$, “splits” in half with each trace computation, where $a_{0\dots 7}$ represents the 8-bit vector $(a_0 \dots a_7)$ and $a_{8\dots 15}$ represents the 8-bit vector $(a_8 \dots a_{15})$. We now derive the expression for the absolute trace of the product $A \otimes B$ for $A, B \in K_4$:

$$\text{Tr}(AB) \tag{3.16}$$

$$= \text{Tr}_{K_0}^{K_4}(A \otimes_n B)$$

$$= \left(\text{Tr}_{K_3}^{K_4} \circ \text{Tr}_{K_2}^{K_3} \circ \text{Tr}_{K_1}^{K_2} \circ \text{Tr}_{K_0}^{K_1} \right) (A \otimes_n B) \tag{3.17}$$

$$= \text{Tr}_{K_0}^{K_1} \left(\text{Tr}_{K_1}^{K_2} \left(\text{Tr}_{K_2}^{K_3} \left(\text{Tr}_{K_3}^{K_4} (AB) \right) \right) \right) \tag{3.18}$$

$$= \text{Tr}_{K_0}^{K_1} \left(\text{Tr}_{K_1}^{K_2} \left(\text{Tr}_{K_2}^{K_3} \left(\left(a_{0\dots 7} \otimes_8 b_{0\dots 7} \right) \oplus_8 \left(a_{8\dots 15} \otimes_8 b_{8\dots 15} \right) \right) \right) \right) \tag{3.19}$$

$$= \text{Tr}_{K_0}^{K_1} \left(\text{Tr}_{K_1}^{K_2} \left(\text{Tr}_{K_2}^{K_3} \left(a_{0\dots 7} \otimes_8 b_{0\dots 7} \right) \oplus_4 \text{Tr}_{K_2}^{K_3} \left(a_{8\dots 15} \otimes_8 b_{8\dots 15} \right) \right) \right) \tag{3.20}$$

$$= \text{Tr}_{K_0}^{K_1} \left(\text{Tr}_{K_1}^{K_2} \left(\left(\left(a_{0\dots 3} \otimes_4 b_{0\dots 3} \right) \oplus_4 \left(a_{4\dots 7} \otimes_4 b_{4\dots 7} \right) \oplus_4 \left(\left(a_{8\dots 11} \otimes_4 b_{8\dots 11} \right) \oplus_4 \left(a_{12\dots 15} \otimes_4 b_{12\dots 15} \right) \right) \right) \right)$$

$$= \text{Tr}_{K_0}^{K_1} \left(\text{Tr}_{K_1}^{K_2} \left(\left(\left(a_{0\dots 3} \otimes_4 b_{0\dots 3} \right) \oplus_4 \left(a_{4\dots 7} \otimes_4 b_{4\dots 7} \right) \right) \oplus_2 \text{Tr}_{K_1}^{K_2} \left(\left(a_{8\dots 11} \otimes_4 b_{8\dots 11} \right) \oplus_4 \left(a_{12\dots 15} \otimes_4 b_{12\dots 15} \right) \right) \right)$$

$$= \text{Tr}_{K_0}^{K_1} \left(a_{0,1} \otimes_2 b_{0,1} \right) \oplus_1 \text{Tr}_{K_0}^{K_1} \left(a_{2,3} \otimes_2 b_{2,3} \right) \oplus_1 \text{Tr}_{K_0}^{K_1} \left(a_{4,5} \otimes_2 b_{4,5} \right) \oplus_1 \text{Tr}_{K_0}^{K_1} \left(a_{6,7} \otimes_2 b_{6,7} \right) \oplus_1$$

$$\text{Tr}_{K_0}^{K_1} \left(a_{8,9} \otimes_2 b_{8,9} \right) \oplus_1 \text{Tr}_{K_0}^{K_1} \left(a_{10,11} \otimes_2 b_{10,11} \right) \oplus_1 \text{Tr}_{K_0}^{K_1} \left(a_{12,13} \otimes_2 b_{12,13} \right) \oplus_1 \text{Tr}_{K_0}^{K_1} \left(a_{14,15} \otimes_2 b_{14,15} \right)$$

$$= \left(a_0 \otimes_1 b_0 \right) \oplus_1 \left(a_1 \otimes_1 b_1 \right) \oplus_1 \left(a_2 \otimes_1 b_2 \right) \oplus_1 \left(a_3 \otimes_1 b_3 \right) \oplus_1 \dots \oplus_1 \left(a_{15} \otimes_1 b_{15} \right)$$

$$= \bigoplus_{i=0}^{15} \left(a_i \otimes_1 b_i \right)$$

$$= \bigoplus_{i=0}^{15} \left(a_i \odot_1 b_i \right) \tag{3.21}$$

The first steps in above computation make use of the transitivity property of trace function (Theorem 2.4), for the compositions of trace functions (steps (3.18) and (3.19)) on the tower refer to the commutative diagram in Figure 3.7 on the right. Line 3.20 was obtained using equation 3.15, and line 3.21 using the trace property i . from Theorem 2.3. These two steps are then repeated two more times to obtain the final result in line 3.21, which is a simple and elegant expression involving one-bit addition and multiplication. The one-bit multiplication \otimes at the lowest level is a simple 1-bit AND gate, denoted with \odot (\otimes was simply replaced with \odot in the last step). \square

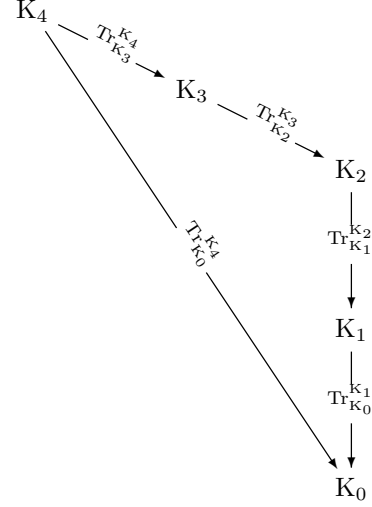


Figure 3.7: Transitivity of trace function in $\mathbb{F}_{((2^2)^2)^2}$

Corollary 1 For any elements $X = (x_0 \dots x_{15})$, $A = (a_0 \dots a_{15})$ and $B = (b_0 \dots b_{15})$ in $\mathbb{F}_{((2^2)^2)^2}$ we have $\text{Tr}(X^{2^w}) = \text{Tr}(X) = \bigoplus_{i=0}^{15} x_i$ and $\text{Tr}(AB) = \text{Tr}(A^{2^w} \odot_{16} B^{2^w})$ where w is an integer.

Proof First part of Corollary 1 follows directly from Proposition 1 with $A = X$ and $B = 1 = (1 \dots 1)$, giving $\text{Tr}(X) = \bigoplus_{i=0}^{15} (x_i \odot_1 1) = \bigoplus_{i=0}^{15} x_i$.

Noting that the absolute trace of an element from \mathbb{F}_{2^m} is but a sum of all of its m distinct conjugates, the absolute trace of the element raised to a power of 2 will also be the sum of the same conjugates, since $X^{2^m} = X$ and hence the summands begin to cycle at some point.

Alternatively we can write $\text{Tr}(X^{2^w}) = \sum_{i=0}^{m-1} (X^{2^w})^{2^i} = \sum_{i=0}^{m-1} (X^{2^i})^{2^w}$, and since we are working with binary fields $\sum_{i=0}^{m-1} (X^{2^i})^{2^w} = \left(\sum_{i=0}^{m-1} X^{2^i} \right)^{2^w} = (\text{Tr}(X))^{2^w}$, and $(\text{Tr}(X))^{2^w} = \text{Tr}(X)$ because the absolute trace maps onto the prime field \mathbb{F}_2 .

Say we can write X as a bitwise AND of elements A and B , i.e. $X = A \odot_{16} B$. From the first part of this corollary we obtain $\text{Tr}(X) = \bigoplus_{i=0}^{15} x_i = \bigoplus_{i=0}^{15} (a_i \odot_1 b_i)$ and from proposition 1

$\text{Tr}(AB) = \bigoplus_{i=0}^{15} (a_i \odot_1 b_i)$, hence $\text{Tr}(AB) = \text{Tr}(A \odot_{16} B)$. And since $\text{Tr}(X) = \text{Tr}(X^{2^w})$, the result follows: $\text{Tr}(AB) = \text{Tr}((AB)^{2^w}) = \text{Tr}(A^{2^w} B^{2^w}) = \text{Tr}(A^{2^w} \odot_{16} B^{2^w})$. \square

Corollary 2 *For any elements $A = (a_0 \dots a_{15})$, $B = (b_0 \dots b_{15})$ and $C = (c_0 \dots c_{15})$ in $\mathbb{F}_{(((2^2)^2)^2)^2}$ we have $\text{Tr}(A \odot_{16} C) \oplus_1 \text{Tr}(B \odot_{16} C) = \text{Tr}((A \oplus_{16} B) \odot_{16} C)$.*

For proof refer to [5].

Algebraic optimization arising from $\mathbb{F}_{(((2^2)^2)^2)^2}$

Finally, we are ready to take a look at $\text{WGT-16}(X^d)$ and begin with a short description of the decimation. Then we direct our attention to the $\text{WGT-16}(X^d)$, simplify its computation using the results from the previous Section 3.4.3, “translate” the obtained equation for $\text{WGT-16}(X^d)$ into hardware and show its data-dependency graph. Finally we examine the $\text{WGP-16}(X^d)$ that is needed during the initialization. We conclude the Section with WGP-T module constructed as an integrated hardware that can compute both $\text{WGT-16}(X^d)$ and $\text{WGP-16}(X^d)$.

The decimation

As already mentioned in Section 3.2, $d = 1057$ can be written as “10000100001” in binary, that is $2^{10} + 2^5 + 1$, resulting in $X^d = X^{2^{10}} \otimes_{16} X^{2^5} \otimes_{16} X$. For a field element in its normal basis representation, value X^{2^k} can be computed by a with a right cyclic shift for k positions. As was already slightly indicated in Section 3.4.2 with conversion matrices and in the proof of proposition 1, multiplication is carried out in the tower field, hence we must convert the three factors $X^{2^{10}}$, X^{2^5} and X into their tower field representation before multiplying. Transformation $\text{WGT-16}(X^d)$ is computed as the absolute trace $\text{Tr}(q(X^d \oplus_{16}) \oplus_{16} 1)$.

The **WGT-16**(X^d) computation in running phase

Let $Y = X^d \oplus_{16} 1$. Recalling that $\text{Tr}(1) = 0$ and using trace property *i.* from Theorem 2.3 we obtain the following expression for $\text{WGT-16}(X^d)$:

$$\begin{aligned}
& \text{WGT-16}(X^d) \\
&= \text{Tr}(q(Y) \oplus_{16} 1) \\
&= \text{Tr}(q(Y)) \oplus_1 \text{Tr}(1) \\
&= \text{Tr}(q(Y)) \\
&= \text{Tr}\left(Y \oplus_{16} Y^{2^{11}+1} \oplus_{16} Y^{2^6-2^{11}+1} \oplus_{16} Y^{2^{11}+2^6+1} \oplus_{16} Y^{2^{11}+2^6-1}\right) \\
&= \text{Tr}\left(Y \oplus_{16} Y^{2^{11}+1}\right) \oplus_1 \text{Tr}\left(Y^{2^6-2^{11}+1}\right) \oplus_1 \text{Tr}\left(Y^{2^{11}+2^6+1} \oplus_{16} Y^{2^{11}+2^6-1}\right) \quad (3.22)
\end{aligned}$$

Let us now take a look at the second trace function in 3.22 and rewrite:

$$\begin{aligned}
Y^{2^6-2^{11}+1} &= Y \otimes_{16} Y^{2^6-2^{11}} \\
&= Y \otimes_{16} Y^{2^{11}(2^{11}-1)}
\end{aligned}$$

The relationship $X^{2^{22}} = X^{2^6}$ (due to 2.1) was used above. Following Corollary 1 we get the following expressions for the second and the last trace function in 3.22:

$$\begin{aligned}
& \text{Tr}\left(Y \otimes_{16} Y^{2^{11}(2^{11}-1)}\right) && \text{Tr}\left(Y^{2^{11}+2^6+1} \oplus_{16} Y^{2^{11}+2^6-1}\right) \\
= \text{Tr}\left(\left(Y \otimes_{16} Y^{2^{11}(2^{11}-1)}\right)^{2^6}\right) && = \text{Tr}\left(Y^{2^6} \odot_{16} \left(Y^{2^{11}+1} \oplus_{16} Y^{2^{11}-1}\right)\right) \quad (3.24) \\
= \text{Tr}\left(Y^{2^6} \odot_{16} Y^{2^6 \cdot 2^{11}(2^{11}-1)}\right) && \\
= \text{Tr}\left(Y^{2^6} \odot_{16} Y^{2^{22}-2^{17}}\right) \quad (3.23)
\end{aligned}$$

We then merge the terms 3.23 and 3.24 using corollary 2. Putting it all together we obtain

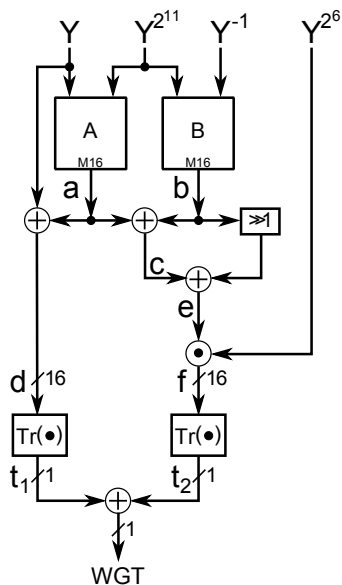
$$\text{WGT-16}(X^d) = \text{Tr}\left(Y \oplus_{16} Y^{2^{11}+1}\right) \oplus_1 \text{Tr}\left(Y^{2^6} \odot_{16} \left(Y^{2^{22}-2^{17}} \oplus_{16} Y^{2^{11}+1} \oplus_{16} Y^{2^{11}-1}\right)\right) \quad (3.25)$$

The elements occurring in (3.25) can be computed as follows:

1. $Y^{2^{11}+1} = Y^{2^{11}} \otimes_{16} Y$ and $Y^{2^{11}-1} = Y^{2^{11}} \otimes_{16} Y^{-1}$, which requires a cyclic shift for 11 positions, denoted $\gg 11$, to compute $Y^{2^{11}}$, an inverter for Y^{-1} and two multipliers;
2. $Y^{2(2^{11}-1)} = \left(Y^{2^{11}-1}\right)^2$ requires an additional shift for one position;
3. Y^{2^6} requires a cyclic shift for 6 positions, that is $\gg 6$;
4. two computations of the form $\bigoplus_{i=0}^{15} x_i$, one for each trace function;
5. remaining operations require 16-bit XOR and AND gates, and of course the final 1-bit XOR for adding up the two traces;

Let us briefly revisit the terms in 3.23 and 3.24. The original 16-bit multiplication $Y \otimes_{16} Y^{2^{11}(2^{11}-1)}$ in 3.23 was replaced with a far more efficient 16-bit AND due to properties that arise from $\mathbb{F}_{((2^2)^2)^2}$ tower construction. Similarly, we use a 16-bit AND instead of a multiplier in 3.24. Furthermore, Corollary 2 enabled us to merge the two terms, thus eliminating one of the 16-bit AND gates.

The data-dependency graph for the $\text{WGT-16}(X^d)$ computation is shown in Figure 3.8.



For clarity, we introduce the following notation to be used in Figure 3.8 on the left:

$$\begin{aligned}
 a &= Y^{2^{11}+1} & e &= b^2 \oplus_{16} c \\
 b &= Y^{2^{11}-1} & f &= Y^{2^6} \odot_{16} e \\
 c &= a \oplus_{16} b & t_1 &= \text{Tr}(d) \\
 d &= Y \oplus_{16} a & t_2 &= \text{Tr}(f)
 \end{aligned}$$

The $\text{WGT-16}(X^d)$ from equation 3.25 is now computed as $\text{WGT-16}(X^d) = t_1 \oplus_1 t_2$

Figure 3.8: Data-dependency graph for $\text{WGT-16}(X^d)$ computation

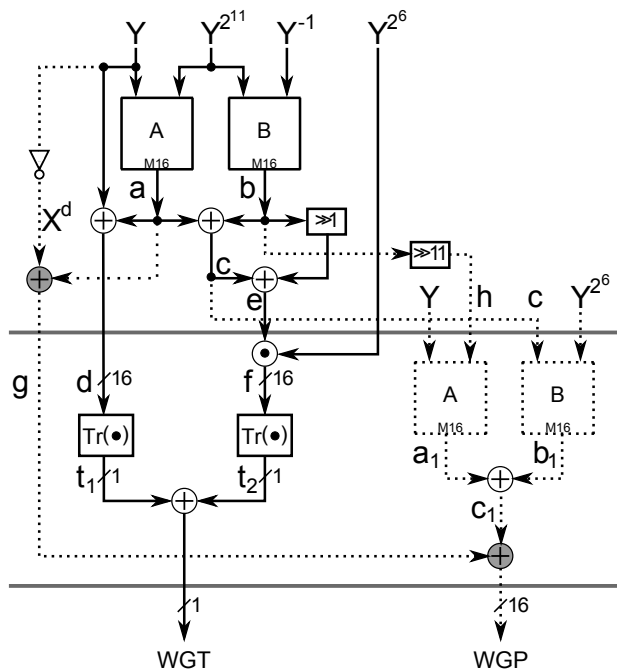
The $\text{WGT-16}(X^d)$ computation shown in Figure 3.8 does not begin with input X , but assumes the values $Y = X^d \oplus_{16} 1$, $Y^{2^{11}}$, Y^{2^6} and Y^{-1} have already been obtained. Exponentiations Y^{2^k} are just cyclic shifts, as was explained in Section 3.3.2, but require transition to normal bass and back to tower basis representation. Inversion is more complex and will be explained in detail in Section 4.4.1. In Figure 3.8, we can see the two multipliers, denoted A and B , performing multiplications $Y^{2^{11}} \otimes_{16} Y$ and $Y^{2^{11}} \otimes_{16} Y^{-1}$. The shift $\gg 1$ denotes the exponentiation b^{2^1} , i.e. the squaring that produces the value $Y^{2(2^{11}-1)}$. Shifting is performed in normal basis representation and the basis conversion is omitted from Figure 3.8 for simplicity. The final two blocks marked $\text{Tr}(\bullet)$ perform the trace operations $\bigoplus_{i=0}^{15} d_i$ and $\bigoplus_{i=0}^{15} f_i$. This concludes the computation conducted during the running phase.

The $\text{WGP-16}(X^d)$ computation - the initialization phase

Let us now look at the $\text{WGP-16}(X^d) = q(Y) \oplus_{16} 1$ needed during initialization phase:

$$\begin{aligned}
& \text{WGP-16}(X^d) \\
&= q(Y) \oplus_{16} 1 \\
&= 1 \oplus_{16} Y \oplus_{16} Y^{2^{11}+1} \oplus_{16} \left(Y \otimes_{16} Y^{2^{11}(2^{11}-1)} \right) \oplus_{16} \left(Y^{2^6} \otimes_{16} \left(Y^{2^{11}+1} \oplus_{16} Y^{2^{11}-1} \right) \right) \\
&= \left(X^d \oplus_{16} Y^{2^{11}+1} \right) \oplus_{16} \left(Y \otimes_{16} Y^{2^{11}(2^{11}-1)} \right) \oplus_{16} \left(Y^{2^6} \otimes_{16} \left(Y^{2^{11}+1} \oplus_{16} Y^{2^{11}-1} \right) \right) \quad (3.26)
\end{aligned}$$

Examining $\text{WGT-16}(X^d)$ (expression 3.25) and $\text{WGP-16}(X^d)$ (expression 3.26), we identify the common terms occurring in both expressions, namely: $a = Y^{2^{11}+1}$, $b = Y^{2^{11}-1}$ and $c = a \oplus_{16} b$. Instead of drawing a completely new diagram for $\text{WGP-16}(X^d)$ computation, we simply expand the $\text{WGT-16}(X^d)$ data-dependency graph in Figure 3.8; the new diagram is shown in Figure 3.9: the solid lines indicate the $\text{WGT-16}(X^d)$ computation and the dotted lines indicate the new elements that were added for the $\text{WGP-16}(X^d)$ computation.



Additional notation

$$\begin{aligned}
 g &= X^d \oplus_{16} a \\
 h &= b^{2^{11}} \\
 a_1 &= Y^{2^6} \otimes_{16} c \\
 b_1 &= Y \otimes_{16} h \\
 c_1 &= a_1 \oplus_{16} b_1
 \end{aligned}$$

The $\text{WGT-16}(X^d)$ from equation 3.26 is now computed as $\text{WGP-16}(X^d) = g_1 = g \oplus_{16} c_1$

Figure 3.9: Dataflow diagram for $\text{WGP-16}(X^d)$ computation

The two 16-bit multiplications, that were eliminated due to properties exhibited by the trace in tower construction $\mathbb{F}_{((2^2)^2)^2}$, have to be performed: there are now four multipliers in Figure 3.9. Computation of $\text{WGP-16}(X^d)$ is needed only in the initialization phase, and has to be computed exactly 64 times. In the running phase we compute $\text{WGT-16}(X^d)$ without computing $\text{WGP-16}(X^d)$ first. The 64 iterations of the initialization phase are negligible compared to the running phase, so we find that a trade-off can be made: instead of having two additional multipliers we can reuse the two existing ones that are also used to compute the $\text{WGT-16}(X^d)$ and serially compute $\text{WGP-16}(X^d)$ over two consecutive clock cycles. We add clock boundary lines (grey horizontal lines in Figure 3.9); for the same reason, the two dotted multipliers were marked A and B , just as the two solid-lined multipliers, to indicate that they are the same piece of hardware. We also notice that since now the computation is stretched over two clock cycles, we can use the same XOR gate to compute $g = X^d \oplus_{16} a$ in the first clock cycle and $\text{WGP-16}(X^d) = g \oplus_{16} c_1$ in the next clock cycle. To indicate the reuse, the two XOR gates are shaded grey in the Figure 3.9. Allocation table for the reused components (multipliers A and B and the XOR gate) can be seen in Table 3.11 below. It shows that we need six additional multiplexers MUX_i ($i = 1, 2, \dots, 6$), one for each input port (all the reused components have two input ports,

denoted `i1` and `i2` in Table 3.11). The six multiplexers are listed in the last line of Table 3.11, and the values in the first column (column clock cycle) can be used as the control signal `sel` to control the multiplexer outputs.

clock cycle / <code>sel</code>	Multiplier		Multiplier		XOR	
	M_{16}		M_{16}		gate	
	A		B			
	i1	i2	i1	i2	i1	i2
0	$Y^{2^{11}}$	Y	Y^{-1}	$Y^{2^{11}}$	X^d	a
1	c	Y^{2^6}	h	Y	g	c_1
	MUX ₁	MUX ₂	MUX ₃	MUX ₄	MUX ₅	MUX ₆

Table 3.11: Allocation table for the reused blocks

During the first clock cycle, the control signal `sel` is at low logic level and thus MUX_{*i*} ($i = 1, 2, \dots, 6$) generate the signals $Y^{2^{11}}$, Y , Y^{-1} , $Y^{2^{11}}$, X^d and $a = Y^{2^{11}+1}$ at their outputs, respectively. At this time the intermediate products $a = Y^{2^{11}} \otimes_{16} Y$ and $b = Y^{-1} \otimes_{16} Y^{2^{11}}$ and the value $g = X^d \oplus_{16} a$ are computed. In the next clock cycle, the control signal `sel` is pulled up, which enables six multiplexers to feed correct operands to multipliers and XOR gate and the two multipliers are reused to obtain values $a_1 = Y^{2^6} \otimes_{16} c = Y^{2^6} \otimes_{16} (Y^{2^{11}+1} \oplus_{16} Y^{2^{11}-1})$ and $b_1 = Y \otimes_{16} h = Y \otimes_{16} Y^{2^{11}(2^{11}-1)}$. The inputs Y and Y^{2^6} have the same value as in the first clock cycle, that is they are held constant. Three additional registers will be needed to hold the intermediate values, that is, the values that must be returned to the inputs in the second clock cycle:

- Reg₁ for value $c = Y^{2^{11}+1} \oplus_{16} Y^{2^{11}-1}$,
- Reg₂ for value $h = Y^{2^{11}(2^{11}-1)}$, and
- Reg₃ for value $g = X^d \oplus_{16} Y^{2^{11}+1}$.

The WGP_T module

The integrated hardware architecture, that can compute both the $\text{WGP-16}(X^d)$ in the initialization phase and $\text{WGT-16}(X^d)$ in the running phase, can be seen in Figure 3.10. On the left we see the field element X entering the module in normal basis representation: the initial exponentiations are carried out in the normal basis as right cyclic shifts denoted $\gg 5$ and $\gg 10$, immediately followed by the conversion to tower field basis representation. From this moment on, all operations except for the exponentiations are carried out in the tower field. The respective basis conversions are denoted with blocks M_{NT} and M_{TN} . There are four multipliers M_{16} , performing 16-bit multiplications \otimes . The two multipliers on the left are needed for the initial decimation. The remaining two multipliers in the middle are (re)used to compute the keystream $\text{WGT-16}(X^d)$ and the initialization feedback $\text{WGP-16}(X^d)$: they are marked A and B to keep consistency with the dataflow diagram for the $\text{WGP-16}(X^d)$ computation in Figure 3.9 and the part of the circuit used for $\text{WGP-16}(X^d)$ only is shown with dotted lines and blocks. Figure 3.10 is fitted with same notation as the dataflow diagram in Figure 3.9. The six multiplexers from Table 3.11 and the three registers Reg_1 , Reg_2 and Reg_3 can be seen in the Figure 3.10. Same notation was used:

$$\begin{aligned}
 a &= Y^{2^{11}+1} & c &= a \oplus_{16} b & e &= b^2 \oplus_{16} c & g &= X^d \oplus_{16} a & a_1 &= Y^{2^6} \otimes_{16} c \\
 b &= Y^{2^{11}-1} & d &= Y \oplus_{16} a & f &= Y^{2^6} \odot_{16} e & h &= b^{2^{11}} & b_1 &= Y \otimes_{16} h \\
 t_1 &= \text{Tr}(d) & t_2 &= \text{Tr}(f) & & & & & c_1 &= a_1 \oplus_{16} b_1
 \end{aligned}$$

The keystream (denoted with WGT in Figure 3.10) is computed as $\text{WGT-16}(X^d) = t_1 \oplus_1 t_2$, and the initialization feedback WGP as $\text{WGP-16}(X^d) = g_1 = g \oplus_{16} c_1$.

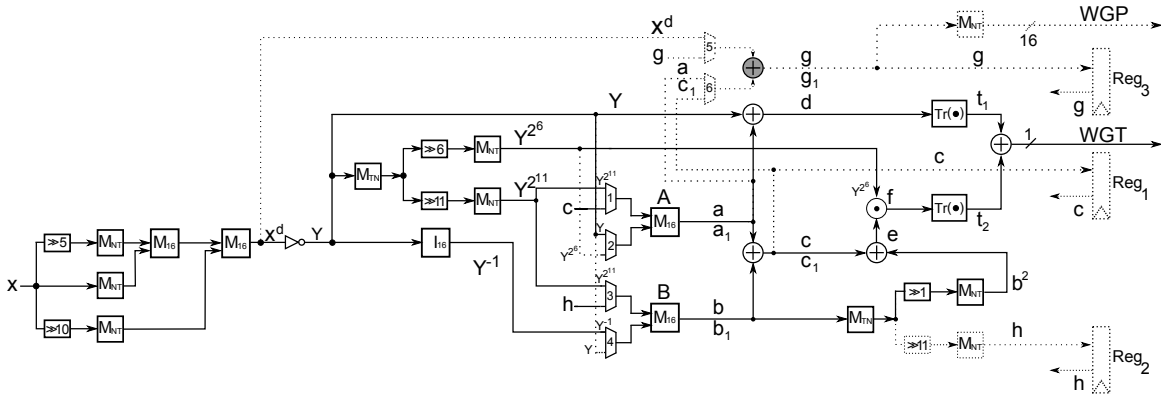


Figure 3.10: Module WGP_T with multiplier reuse using tower field $\mathbb{F}_{((2^2)^2)^2}$

3.5 Tower construction $\mathbb{F}_{(2^4)^4} \cong \mathbb{F}_{2^{16}}$

3.5.1 Field construction

Tower construction $\mathbb{F}_{(2^4)^4}$ has two levels above the prime field \mathbb{F}_2 . To construct the top level $\mathbb{F}_{(2^4)^4} \cong \mathbb{F}_{2^{16}}$ we will need two irreducible polynomials of degree four; polynomial $e(x) \in \mathbb{F}_2[x]$ and polynomial $f(x) \in \mathbb{F}_{2^4}[x]$:

$$\mathbb{F}_2 \xrightarrow{e(x)} \mathbb{F}_{2^4} \xrightarrow{f(x)} \mathbb{F}_{(2^4)^4}.$$

The construction $\mathbb{F}_{(2^4)^4}$ is summarized in the following Table 3.12:

Finite Field \mathbb{F}_{2^n}	Normal Basis over $\mathbb{F}_{2(\frac{n}{4})}$	Normal element as power of ω	Defining polynomial
$\mathbb{F}_{2^{16}} \cong \mathbb{F}_{(2^4)^4}$	$\{\beta, \beta^{16}, \beta^{256}, \beta^{4096}\}$	$\beta = \omega^{2206}$	$f(x) = x^4 + x^3 + x^2 + \lambda \dagger$
\mathbb{F}_{2^4}	$\{\alpha, \alpha^2, \alpha^4, \alpha^3\}$	$\alpha = \omega^{13107}$	$e(x) = x^4 + x^3 + x^2 + x + 1$

Table 3.12: Tower construction of $\mathbb{F}_{(2^4)^4}$

ω is a root of polynomial $x^{16} + x^5 + x^3 + x^2 + 1$, used to construct the isomorphic field $\mathbb{F}_{2^{16}}$

$$\dagger \lambda = \alpha + \alpha^3$$

A reader satisfied with information provided in Table 3.12 can proceed to Section 3.5.2.

The rest of the Section provides a detailed analysis of the tower construction and discussion about the choice of defining polynomials, some based on theoretical background and some on exhaustive search.

Additional mathematical background

Theoretical results that could help prove the irreducibility of polynomials, do not exist for this construction, so we are forced to conduct an exhaustive search for best suitable polynomials. The first level of the construction is still a small field, so we can go into more details. For the second level of the tower, we have many more options and need to find a way to narrow them down.

We decide to use normal basis representation of elements at both levels of the tower, therefore we need N-polynomials. To ensure that the irreducible polynomial used for the extension is also a N-polynomial, i.e. a normal polynomial, we use the following fact:

Fact 3.5 [70, Corollary 4.19] *Let $m = p^e$ and $f(x) = x^m + a_1x^{m-1} + \dots + a_m$ be an irreducible polynomial over \mathbb{F}_q . Then $f(x)$ is a N-polynomial if and only if $a_1 \neq 0$.*

Note that 3.4 used in Section 3.4.1, is actually a special case of the fact 3.5.

Extension field \mathbb{F}_{2^4}

■ Search for an irreducible polynomial

We begin the tower construction by finding an irreducible polynomial of degree 4 with coefficients from \mathbb{F}_2 . Such a polynomial will have the form $e(x) = x^4 + e_3x^3 + e_2x^2 + e_1x + 1$ of course with leading coefficient 1 and with a nonzero constant term; otherwise, $e(x)$ would have the factor x and would be reducible. Also, a polynomial over \mathbb{F}_2 needs an odd number of nonzero terms, otherwise it would be divisible by $x + 1$. Considering these facts, the choice narrows down to a trinomial or the *all one polynomial*, abbreviated AOP, of degree 4, that is polynomial $x^4 + x^3 + x^2 + x + 1$. The four candidates are listed in Table 3.13. The polynomial $e_2(x) = x^4 + x^2 + 1$ can be written as $(x^2 + x + 1)^2$, but other three polynomials $e_1(x)$, $e_3(x)$ and $e_4(x)$ are irreducible over \mathbb{F}_2 . We can easily check irreducibility of binary polynomials of such a small degree with exhaustive search, even by hand if need be, but let us provide some helpful mathematical background. For details refer to [69].

Definition 3.1 *Let $p(x)$ be a polynomial of degree m with coefficients in \mathbb{F}_q , such that $p(0) \neq 0$. The smallest positive integer s , for which $p(x)|(x^s - 1)$, is called the order of polynomial $p(x)$, denoted $\text{ord}(p)$.*

If the above defined polynomial $p(x)$ is irreducible over \mathbb{F}_q , then its order divides $q^m - 1$, [69]. Furthermore, $p(x)$ is primitive over \mathbb{F}_q if and only if it is monic, does not have element 0 as root (that is $p(0) \neq 0$), and has the maximum order, that is $\text{ord}(p) = q^m - 1$.

For \mathbb{F}_{2^4} we have $q = 2$ and $m = 4$, which gives maximum order 15. Only possible orders of the the elements of \mathbb{F}_{2^4} , and therefore only possible orders of their minimal polynomials, are 15,5,3 and 1. So if any of the remaining candidates $e_1(x)$, $e_3(x)$ and $e_4(x)$ are primitive, they will have order $2^4 - 1 = 15$. If the polynomial is irreducible, but not primitive its root and hence the polynomial itself, will have order 5 (order 3 is not possible since $m > 3$). Orders of all four candidates are given in Table 3.13: trinomials $e_1(x)$ and $e_3(x)$ are primitive, AOP is (only) irreducible and, as expected $\text{ord}(e_2(x)) = 6 \nmid 15$ since $e_2(x)$ is reducible.

We found three candidates for construction of \mathbb{F}_{2^4} , but the roots of $e_1(x)$ are not linearly independent (let $e_1(\alpha) = 0$, then $\alpha^{2^3} = \alpha^8 = \alpha + \alpha^2 + \alpha^{2^2}$), and therefore do not form a normal basis and the polynomial $e_1(x)$ is not a N-polynomial. The fifth column in Table 3.13 is labeled “normal”: due to fact 3.5, the polynomials $e_3(x)$ and $e_4(x)$ are N-polynomials. Only two polynomials left are $e_3(x) = x^4 + x^3 + 1$ and the AOP $e_4(x)$. We check the weight C_N of their T matrices and find $C_N = 7 = 2m - 1$ for the AOP and $C_N = 9$ for $e_3(x)$. We choose AOP $e_4(x)$ as defining polynomial of \mathbb{F}_{2^4} since it produces an optimal normal basis. ■

Polynomial	ord($e_i(x)$)	irreducible	primitive	normal	C_N
$e_1(x) = x^4 + x + 1$	15	✓	✓		
$e_2(x) = x^4 + x^2 + 1$	6				
$e_3(x) = x^4 + x^3 + 1$	15	✓	✓	✓	9
$e_4(x) = x^4 + x^3 + x^2 + x + 1$	5	✓		✓	7

Table 3.13: Candidates for irreducible polynomials of degree 4 over \mathbb{F}_2

We construct the finite field \mathbb{F}_{2^4} by adjoining the root α of the all one polynomial (AOP) $e_4(x) = x^4 + x^3 + x^2 + x + 1$ (shaded grey in Table 3.13) to the elements of prime field \mathbb{F}_2 . From the AOP we obtain the relationships $\alpha^4 = \alpha^3 + \alpha^2 + \alpha + 1$ and $\alpha^{2^3} = \alpha^8 = \alpha^3$ used for the representation with normal basis $\{\alpha, \alpha^2, \alpha^4, \alpha^3\}$. Note that normal basis elements have order 5, the same as AOP. In Table 3.14, the elements $A \in \mathbb{F}_{2^4}$ (obtained by AOP with root α) are represented using:

- i. polynomial basis $\{1, \alpha, \alpha^2, \alpha^3\}$: $A = p_0 + p_1\alpha + p_2\alpha^2 + p_3\alpha^3$ - first column of Table 3.14 and
- ii. normal basis $\{\alpha, \alpha^2, \alpha^4, \alpha^3\}$: $A = n_0\alpha + n_1\alpha^2 + n_2\alpha^4 + n_3\alpha^3$ - fifth (shaded grey) column of Table 3.14;

Table 3.14 is divided into two parts: the five columns on the left belong to the current tower construction \mathbb{F}_{2^4} over \mathbb{F}_2 using AOP of degree 4. The normal basis representation shown in column four of Table 3.14 is the sixth representation of \mathbb{F}_{2^4} we have seen so far; three other bases for the finite field with 16 elements were found in Section 3.4.1. An interested reader might see a discussion on different representations and conversion between them in Appendix B.2.1.

Since the root of the AOP α is not a primitive element (because $\text{ord}(\alpha) = 5$, see Table 3.13) and does not generate $\mathbb{F}_{2^4}^*$, we take the primitive element $\alpha + \alpha^3 = \lambda$ for representation of field elements as powers of λ .

■ **Remark:** *Conversion matrices*

Transition matrices between polynomial (i.) and normal (ii.) basis representation, relevant for the current construction \mathbb{F}_{2^4} , are given below. ■

$$M_P^N = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad M_N^P = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

\mathbb{F}_{2^4} - current field construction				$\mathbb{F}_{(2^2)^2}$ - field construction from Section 3.4.1				
polynomial basis \mathbb{F}_{2^4} over \mathbb{F}_2		A as polynomial	order of A	A as power of λ	normal basis \mathbb{F}_{2^4} over \mathbb{F}_2		A as power of β	
1	α				α^2	α^3		α
p_0	p_1	p_2	p_3		n_0	n_1	n_2	n_3
0	0	0	0	/	0	0	0	0
0	0	0	1	α^3	0	0	0	1
0	0	1	0	α^2	0	1	0	0
0	0	1	1	$\alpha^2 + \alpha^3$	0	1	0	1
0	1	0	0	α	1	0	0	0
0	1	0	1	$\alpha + \alpha^3$	1	0	0	1
0	1	1	0	$\alpha + \alpha^2$	1	1	0	0
0	1	1	1	$\alpha + \alpha^2 + \alpha^3$	1	1	0	1
1	0	0	0	1	1	1	1	1
1	0	0	1	$1 + \alpha^3$	1	1	1	0
1	0	1	0	$1 + \alpha^2$	1	0	1	1
1	0	1	1	$1 + \alpha^2 + \alpha^3$	1	0	1	0
1	1	0	0	$1 + \alpha$	0	1	1	1
1	1	0	1	$1 + \alpha + \alpha^3$	0	1	1	0
1	1	1	0	$1 + \alpha + \alpha^2$	0	0	1	1
1	1	1	1	$1 + \alpha + \alpha^2 + \alpha^3$	0	0	1	0

tower field basis \mathbb{F}_{2^4} over \mathbb{F}_2		normal basis $\mathbb{F}_{(2^2)^2}$ over \mathbb{F}_{2^2}		A as power of β	
β^6	β^{11}	β^9	β^{14}		β
t_0	t_1	t_2	t_3	b_0	b_1
0	0	0	0	0	0
0	0	1	0	0	α
1	0	0	0	α	0
1	0	1	0	α	α
1	1	1	0	1	α
1	1	0	0	1	0
0	1	1	0	α^2	α
0	1	0	0	α^2	0
1	1	1	1	1	1
1	1	0	1	1	α^2
0	1	1	1	α^2	1
0	1	0	1	α^2	α^2
0	0	0	1	0	α^2
0	0	1	1	0	1
1	0	0	1	1	0
1	0	0	1	α	α^2
1	0	1	1	α	1

Table 3.14: Elements of the finite field of order 16
 $\mathbb{F}_{2^4}/\mathbb{F}_2$ - on the left: with normal basis $\{\alpha, \alpha^2, \alpha^4, \alpha^3\}$ representation shaded grey
 $\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}$ - on the right: tower field representation and normal basis representation from Table 3.8 in Section 3.4.1

Extension field $\mathbb{F}_{(2^4)^4}$

The extension $\mathbb{F}_{(2^4)^4}$ over \mathbb{F}_{2^4} is obtained by adjoining the root β of the irreducible polynomial $f(x) = x^4 + x^3 + x^2 + \lambda$, yielding the normal basis $\{\beta, \beta^{16}, \beta^{256}, \beta^{4096}\}$. The choice of the defining polynomial is discussed below.

■ Search for an irreducible polynomial

Now we need to find a suitable polynomial $f(x)$, irreducible over \mathbb{F}_{2^4} , for the second level of the tower construction. We want a simple expression for $f(x)$, so we try polynomials of the form $f(x) = x^4 + f_3x^3 + f_2x^2 + f_1x + f_0$ with coefficients f_i from \mathbb{F}_{2^4} ; the term simple referring to values f_i , $i = 3, 2, 1$, say 0 or 1. From experience with the previous tower construction $\mathbb{F}_{((2^2)^2)^2}$, we choose the constant term f_0 to be either the element 1, or a primitive element of the subfield, or an element from the normal basis of the subfield. Note that the AOP is not irreducible over \mathbb{F}_{2^4} . According to this empirical “rules”, the most interesting candidates among 16320 irreducible polynomials over \mathbb{F}_{2^4} are listed in Table 3.15 below. Nonzero field elements represented as powers of λ are given in the fourth column of Table 3.14.

Polynomial	irreducible
$f_1(x) = x^4 + x^3 + x^2 + x + \delta$ where $\delta \in \{1, \lambda, \alpha, \alpha^2, \alpha^4, \alpha^3\}$	
$f_2(x) = x^4 + x^3 + x^2 + \lambda^{(2^i)}$ for $i = 0, 1, 2, 3$	✓
$f_3(x) = x^4 + x^3 + x + \lambda^{(2^i)}$ for $i = 0, 1, 2, 3$	✓

Table 3.15: Candidates for irreducible polynomials of degree 4 over \mathbb{F}_{2^4}

Note that f_2 and f_3 actually stand for a “family” of polynomials that differ only in the constant term: it can be either λ or one of its conjugates.

Among the 8 candidates from the f_2 and f_3 family we chose the polynomial in the same manner as when searching for polynomials for the previous tower $\mathbb{F}_{((2^2)^2)^2}$: by conducting an exhaustive search for the normal element of $\mathbb{F}_{2^{16}}$ with defining polynomial $x^{16} + x^5 + x^3 + x^2 + 1$ that gives the lowest Hamming weight conversion matrices between the normal basis of $\mathbb{F}_{2^{16}}$ and the tower field representation in question. All eight polynomials give transition matrices whose Hamming weights sum up to 176, with the normal element ω^{5053} for polynomials f_2 and ω^{7759} for polynomials f_3 . Transition matrices are of course different for each particular normal element and conjugate of λ . ■

3.5.2 Conversion matrices

Efficient conversion matrices between the normal basis and tower field representation of $\mathbb{F}_{(2^4)^4}$ elements are needed for exponentiation to powers of two. Conversion matrices between the described tower field basis of $\mathbb{F}_{(2^4)^4}$ and the normal basis of are obtained as follows:

$$A = (a_{00}\alpha + a_{01}\alpha^2 + a_{02}\alpha^4 + a_{03}\alpha^3)\beta + (a_{10}\alpha + a_{11}\alpha^2 + a_{12}\alpha^4 + a_{13}\alpha^3)\beta^{16} + (a_{20}\alpha + a_{21}\alpha^2 + a_{22}\alpha^4 + a_{23}\alpha^3)\beta^{256} + (a_{30}\alpha + a_{31}\alpha^2 + a_{32}\alpha^4 + a_{33}\alpha^3)\beta^{4096},$$

where $a_{ij} \in \mathbb{F}_2$ for $i, j = 0, 1, 2, 3$.

The basis elements represented as a power the root of the defining polynomial of $\mathbb{F}_{2^{16}}$ are $\alpha = \omega^{13107}$ and $\beta = \omega^{2206}$. The tower basis of $\mathbb{F}_{(2^4)^4}$ consists of elements

$$\begin{array}{llll} t_0 = \alpha\beta & t_4 = \alpha\beta^{16} & t_8 = \alpha\beta^{256} & t_{12} = \alpha\beta^{4096} \\ t_1 = \alpha^2\beta & t_5 = \alpha^2\beta^{16} & t_9 = \alpha^2\beta^{256} & t_{13} = \alpha^2\beta^{4096} \\ t_2 = \alpha^4\beta & t_6 = \alpha^4\beta^{16} & t_{10} = \alpha^4\beta^{256} & t_{14} = \alpha^4\beta^{4096} \\ t_3 = \alpha^3\beta & t_7 = \alpha^3\beta^{16} & t_{11} = \alpha^3\beta^{256} & t_{15} = \alpha^3\beta^{4096} \end{array}$$

In the exhaustive search over all normal elements of $\mathbb{F}_{2^{16}}$ with defining polynomial $x^{16} + x^5 + x^3 + x^2 + 1$, we find the best conversion matrices for the normal element ω^{5053} . The conversion matrices M_N^T and M_T^N , with their respective Hamming weights 80 and 96, are given below (since the procedure is the same as was used in finding conversion matrices for $\mathbb{F}_{((2^2)^2)^2}$ in Section 3.4.2, we omit some intermediate steps).

$$M_N^T = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$M_T^N = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

3.5.3 Module WGP_T

The previous tower construction $\mathbb{F}_{((2^2)^2)^2}$ gave rise to some useful properties of the trace function presented in Section 3.4.3, that significantly simplified the module WGP_T . However, the tower construction $\mathbb{F}_{(2^4)^4}$ in this section does not allow such optimizations. Hence, we are left with no other option but to first compute the value WGP-16(X^d) and then compute its trace.

We follow the equation 3.3 which was used for module WGP_T in polynomial basis representation of field elements in Section 3.2:

- $Y = X^d \oplus_{16} 1$ with $d = 1057 = 2^{10} + 2^5 + 1$,
- $A = Y \otimes_{16} (Y^{2^{11}} \oplus_{16} (Y^{-1})^{2^{11}} \otimes_{16} Y^{2^6})$,
- $B = Y^{2^6} \otimes_{16} Y^{2^{11}} \otimes_{16} (Y \oplus_{16} Y^{-1})$,
- $q(Y) = Y \oplus_{16} A \oplus_{16} B$,
- WGP-16(X^d) = $q(Y) \oplus_{16} 1$ and finally
- WGT-16(X^d) = $\text{Tr}(\text{WGP-16}(X^d))$.

The new module WGP_T is similar to the module used with $\mathbb{F}_{2^{16}}$ in polynomial basis with a few differences:

- exponentiation to powers of 2 is implemented with a right cyclic shift in normal basis representation,
- element 1 is represented by a vector of ones (1,1,...,1) (derived similarly as was done in Section 3.4.3), hence adding 1 to an element is done by inverting its bits, that is
- different computation of the trace function, which will be discussed shortly

Obtained circuit is shown in Figure (3.11).

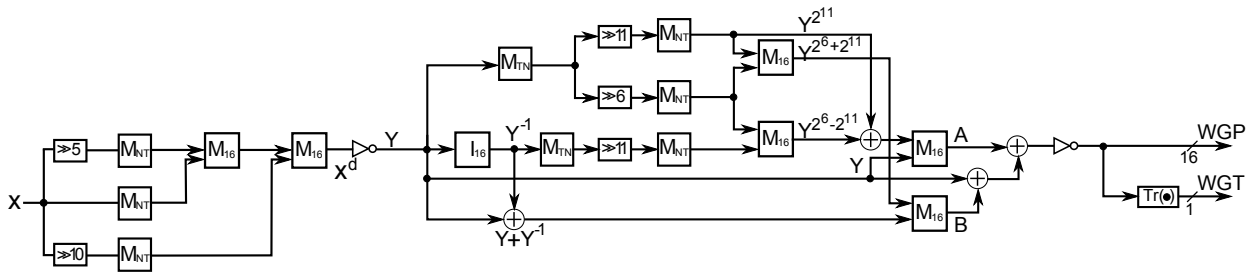


Figure 3.11: Module WGP_T using tower field construction $\mathbb{F}_{(2^4)^4}$

Trace computation

As was mentioned above, we first compute $Z = \text{WGP-16}(X^d)$ and then obtain its trace $\text{Tr}(Z)$. We have three options for computing the trace:

1. computing $\text{Tr}_K^{K_1}(\text{Tr}_{K_1}^{K_2}(Z))$ in tower field $\mathbb{F}_{(2^4)^4}$ where $K = \mathbb{F}_2$, $K_1 = \mathbb{F}_{2^4}$ and $K_2 = \mathbb{F}_{(2^4)^4}$
2. converting to normal basis representation and computing the absolute trace $\text{Tr}_K^F(Z_N) = \bigoplus_{i=0}^{15} z_{Ni}$ of $Z_N = M_N^T \cdot Z$ directly, $F = \mathbb{F}_{2^{16}}$ and $K = \mathbb{F}_2$
3. converting to polynomial basis representation and computing the absolute trace $\text{Tr}_K^F(Z_P) = z_{P11} \oplus z_{P13}$ of $Z_P = M_P^T \cdot Z$, where $F = \mathbb{F}_{2^{16}}$ and $K = \mathbb{F}_2$

Based on the short discussion below we decide for the option 1, i.e. computing the trace directly in the tower field representation. At the end of this section we will derive the expression $\text{Tr}_K^{K_1}(\text{Tr}_{K_1}^{K_2}(Z)) = \bigoplus_{k=0}^{15} z_k$.

The second option seems to be the least promising: the trace is computed as modulo-2 addition of the coefficients, but we need a conversion to normal basis first. Comparing to the first option, which also comes down to a simple modulo-2 sum of coefficients, the transition to normal bases needed for option 2 is an overhead.

The third option seems to be very simple, but we are still discouraged by the basis conversion. And that is not even the biggest downside: it is impossible to beat the total of 15 XOR gates that are needed for computing the trace as described in option 1. In fact, we expect about 15 XOR gates to obtain only 2 out of 16 coefficients of Z_P .

■ **Remark:**

Matrix \mathbf{M}_P^T on the right is the transition matrix from tower field basis to polynomial basis. Let Z be the element represented in the tower field basis and Z_P the polynomial basis representation of element Z , which is obtained as $Z_P = \mathbf{M}_P^T \cdot Z$. Instead of computing Z_P , we decide to only compute the two components that we need, namely the z_{P11} and z_{P13} ; their corresponding rows in matrix \mathbf{M}_P^T are shaded grey. We can see the two rows are exact complement of each other and that the $z_{P11} \oplus z_{P13}$ is just a modulo-2 sum of the coefficients of element Z . Since the trace function is basis independent, this is not a surprising result.

$$\mathbf{M}_P^T = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

$$z_{P11} \oplus z_{P13} = \bigoplus_{i=0}^{15} z_i$$

■

Trace computation in the tower field

Using notation $K = \mathbb{F}_2$, $K_1 = \mathbb{F}_{2^4}$ and $K_2 = \mathbb{F}_{(2^4)^4}$, let us compute the trace $\text{Tr}_K^{K_1}(\text{Tr}_{K_1}^{K_2}(Z))$. We begin with computation of $\text{Tr}_{K_1}^{K_2}(Z)$, that is with the trace of element Z with respect to the subfield \mathbb{F}_{2^4} , hence we write the element Z in terms of basis $\{\beta, \beta^{16}, \beta^{16^2}, \beta^{16^3}\}$ of $\mathbb{F}_{(2^4)^4}/FE$, that is $Z = z_0\beta + z_1\beta^{16} + z_2\beta^{16^2} + z_3\beta^{16^3} \in \mathbb{F}_{(2^4)^4}$, with $z_i \in \mathbb{F}_{2^4}$ for $i = 0, \dots, 3$:

$$\begin{aligned}
\text{Tr}_{K_1}^{K_2}(Z) &= Z + Z^{16} + Z^{16^2} + Z^{16^3} \\
&= z_0\beta + z_1\beta^{16} + z_2\beta^{16^2} + z_3\beta^{16^3} \\
&\quad + z_1\beta + z_2\beta^{16} + z_3\beta^{16^2} + z_0\beta^{16^3} \\
&\quad + z_2\beta + z_3\beta^{16} + z_0\beta^{16^2} + z_1\beta^{16^3} \\
&\quad + z_3\beta + z_0\beta^{16} + z_1\beta^{16^2} + z_2\beta^{16^3} \\
&= (z_0 + z_1 + z_2 + z_3)(\beta + \beta^{16} + \beta^{16^2} + \beta^{16^3}) \\
&= z_0 + z_1 + z_2 + z_3
\end{aligned}$$

In the second line of expression above we use the fact that exponentiation to powers of 16 in the tower field $\mathbb{F}_{(2^4)^4}$ equals to right cyclic shifts, as will be explained in the remark ‘‘By the power of Q’’ below. The last line of the expression was obtained using the fact that the basis elements sum up to 1.

■ **Remark:** *By the power of Q:* With F_{q^4} , where $q = 2^4 = 16$, we can write:

$$Z^q = (z_0\beta + z_1\beta^q + z_2\beta^{q^2} + z_3\beta^{q^3})^q = z_0\beta^q + z_1\beta^{q^2} + z_2\beta^{q^3} + z_3\beta,$$

since by the analogue of Fermat’s little theorem 2.1 we have $\beta^{q^4} = \beta$ and $z_i^q = z_i$ for $i = 0, \dots, 3$. ■

If we now expand elements $z_i \in \mathbb{F}_{2^4}$ into the form $z_i = z_{i,0}\alpha + z_{i,1}\alpha^2 + z_{i,2}\alpha^{2^2} + z_{i,3}\alpha^3$ with coefficients $z_{i,j} \in \mathbb{F}_2$ for $i, j = 0, \dots, 3$, we can rewrite the expression above as follows:

$$\begin{aligned}
&z_0 + z_1 + z_2 + z_3 \\
&= (z_{0,0} + z_{1,0} + z_{2,0} + z_{3,0})\alpha \\
&\quad + (z_{0,1} + z_{1,1} + z_{2,1} + z_{3,1})\alpha^2 \\
&\quad + (z_{0,2} + z_{1,2} + z_{2,2} + z_{3,2})\alpha^{2^2} \\
&\quad + (z_{0,3} + z_{1,3} + z_{2,3} + z_{3,3})\alpha^{2^3} \\
&= w_0\alpha + w_1\alpha^2 + w_2\alpha^{2^2} + w_3\alpha^{2^3} = W
\end{aligned}$$

From the last line of this second expression, where we used notation $w_j = z_{0,j} + z_{1,j} + z_{2,j} + z_{3,j}$, it is easier to see that $\text{Tr}_{K_1}^{K_2}(Z)$ is indeed an element of $K_1 = \mathbb{F}_{2^4}$. We now continue to

compute the second trace, using relationships $\alpha^5 = 1$ and $\alpha + \alpha^2 + \alpha^{2^2} + \alpha^{2^3} = 1$:

$$\begin{aligned}
\mathrm{Tr}_K^{K_1}(W) &= W + W^2 + W^{2^2} + W^{2^3} \\
&= w_0\alpha + w_1\alpha^2 + w_2\alpha^{2^2} + w_3\alpha^{2^3} \\
&+ w_1\alpha + w_2\alpha^2 + w_3\alpha^{2^2} + w_0\alpha^{2^3} \\
&+ w_2\alpha + w_3\alpha^2 + w_0\alpha^{2^2} + w_1\alpha^{2^3} \\
&+ w_3\alpha + w_0\alpha^2 + w_1\alpha^{2^2} + w_2\alpha^{2^3} \\
&= (w_0 + w_1 + w_2 + w_3)(\alpha + \alpha^2 + \alpha^{2^2} + \alpha^{2^3}) \\
&= w_0 + w_1 + w_2 + w_3
\end{aligned}$$

If we now put it all together, replacing w_j with $z_{0,j} + z_{1,j} + z_{2,j} + z_{3,j}$ for $j = 0, \dots, 3$, we obtain the following expression for the absolute trace of element $Z \in \mathbb{F}_{(2^4)^4}$:

$$\begin{aligned}
\mathrm{Tr}(Z) &= \mathrm{Tr}_K^{K_1}(\mathrm{Tr}_{K_1}^{K_2}(Z)) \\
&= (z_{0,0} + z_{1,0} + z_{2,0} + z_{3,0}) \\
&+ (z_{0,1} + z_{1,1} + z_{2,1} + z_{3,1}) \\
&+ (z_{0,2} + z_{1,2} + z_{2,2} + z_{3,2}) \\
&+ (z_{0,3} + z_{1,3} + z_{2,3} + z_{3,3}) \\
&= \bigoplus_{j=0}^3 \left(\bigoplus_{i=0}^3 z_{i,j} \right) \tag{3.27}
\end{aligned}$$

$$= \bigoplus_{k=0}^{15} z_k \tag{3.28}$$

The last line of computation 3.28 was obtained as follows: the indices i, j in line 3.28 can be interpreted as base-4 notation, yielding $k = 4i + j$, which means that only the order of summation is different $z_0 + z_4 + z_8 + \dots$, and since the addition in \mathbb{F}_2 is commutative, we can just change the order of the summation.

3.6 Tower construction $\mathbb{F}_{(2^8)^2} \cong \mathbb{F}_{2^{16}}$

Our last WGP_T implementation based on composite fields is the tower construction $\mathbb{F}_{(2^8)^2}$. The purpose of this implementation was to explore possible advantages of algorithms that are based on table look-ups; these methods have shown speedups in software applications (see 2.5.4).

3.6.1 Field construction

The tower construction $\mathbb{F}_{(2^8)^2} \cong \mathbb{F}_{2^{16}}$ has two levels above the prime field \mathbb{F}_2 . We construct the composite field using two irreducible polynomials: polynomial $e(x) \in \mathbb{F}_2[x]$ and polynomial $f(x) \in \mathbb{F}_{2^8}[x]$:

$$\mathbb{F}_2 \xrightarrow{e(x)} \mathbb{F}_{2^8} \xrightarrow{f(x)} \mathbb{F}_{(2^8)^2}.$$

The construction $\mathbb{F}_{(2^8)^2}$ is summarized in the following Table 3.16:

Finite Field \mathbb{F}_{2^n}	Basis of \mathbb{F}_{2^n} over $\mathbb{F}_{2^{(\frac{n}{4})}}$	The root as power of ω	Defining polynomial
$\mathbb{F}_{2^{16}} \cong \mathbb{F}_{(2^8)^2}$	$\{\beta, \beta^{2^{56}}\}$	$\beta = \omega^{2^{0921}}$	$f(x) = x^2 + x + \lambda \dagger$
\mathbb{F}_{2^8}	$\{1, \alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7\}$	$\alpha = \omega^{2^{57}}$	$e(x) = x^8 + x^4 + x^3 + x^2 + 1$

Table 3.16: Tower construction of $\mathbb{F}_{(2^8)^2}$

ω is a root of polynomial $x^{16} + x^5 + x^3 + x^2 + 1$, used to construct the isomorphic field $F_{2^{16}}$

$$\dagger \lambda = \alpha^{11}$$

A reader satisfied with information provided in Table 3.16 can proceed to Section 3.6.2.

Extension field \mathbb{F}_{2^8}

Following the same routine as with previous tower constructions, we first need to select an irreducible polynomial of degree 8 over the prime field. In order to use table look-up methods for \mathbb{F}_{2^8} arithmetic, we need a primitive polynomial, that is a polynomial whose root generates the multiplicative group $\mathbb{F}_{2^8}^*$. There are no irreducible trinomials of degree 8, so we have to try a pentanomial. The exhaustive search reveals 17 irreducible pentanomials, 12 out of which are also primitive. We choose the first primitive polynomial found, that is the polynomial $e(x) = x^8 + x^4 + x^3 + x^2 + 1$ and use polynomial basis $\{1, \alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7\}$, where α is the root of $e(x)$, to represent the elements of \mathbb{F}_{2^8} . We will discuss the table look-up based methods in detail in Section 4.6.1; nothing more needs to be said at this point, so we proceed to the next extension.

Extension field $\mathbb{F}_{(2^8)^2}$

In order to construct an extension of degree 2 over \mathbb{F}_{2^8} , we need a polynomial of the form $f(x) = x^2 + x + \lambda$ that is irreducible over \mathbb{F}_{2^8} . Exhaustive search for the constant term $\lambda \in \mathbb{F}_{2^8}$ yielding an irreducible $f(x)$, reveals five candidates: $\alpha^5, \alpha^9, \alpha^{10}, \alpha^{11}$ and α^{15} . Among the five irreducible polynomials only one is also primitive, namely the polynomial $f(x) = x^2 + x + \alpha^{11}$.

■ **Remark:** We can check the irreducibility of $f(x) = x^2 + x + \lambda$ in accordance with Corollary 3.3 by computing the absolute trace of $\lambda = \alpha^{11}$:

$$\begin{aligned}
 \text{Tr}(\lambda) &= \lambda + \lambda^2 + \lambda^{2^2} + \lambda^{2^3} + \lambda^{2^4} + \lambda^{2^5} + \lambda^{2^6} + \lambda^{2^7} \\
 &= \alpha^7 + \alpha^6 + \alpha^5 + \alpha^3 + \alpha^7 + \alpha^6 + \alpha^5 + \alpha^3 + \alpha \\
 &+ \alpha^7 + \alpha^6 + \alpha^5 + \alpha^3 + \alpha^2 + \alpha + \alpha^7 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha \\
 &+ \alpha^7 + \alpha^6 + \alpha^5 + \alpha + 1 + \alpha^7 + \alpha^5 + \alpha^3 + \alpha^2 + \alpha + 1 + \alpha^5 + \alpha^4 + \alpha \\
 &+ \alpha^6 + \alpha^5 + \alpha^3 + \alpha^2 + 1 = 1
 \end{aligned}$$

■

The normal basis of $\mathbb{F}_{(2^8)^2}/\mathbb{F}_{2^8}$ is $\{\beta, \beta^{2^8}\}$, where β is the root of $f(x) = x^2 + x + \alpha^{11}$. The field element $A \in \mathbb{F}_{(2^8)^2}$ can be represented as $A = a_0\beta + a_1\beta^{2^8}$, with coefficients $a_0, a_1 \in \mathbb{F}_{2^8}$.

3.6.2 Conversion matrices

As before, we need to find the matrices for efficient conversion between the obtained tower field basis of $\mathbb{F}_{(2^8)^2}$ and a normal basis of $\mathbb{F}_{2^{16}}$ with defining polynomial $x^{16} + x^5 + x^3 + x^2 + 1$. The exhaustive search reveals the normal element producing the normal basis of $\mathbb{F}_{2^{16}}$, for which the conversion matrices have the minimum Hamming weight 208, to be ω^{1789} .

The element $A \in \mathbb{F}_{(2^8)^2}$ can be expanded as follows:

$$\begin{aligned}
 A &= (a_{00} + a_{01}\alpha + a_{02}\alpha^2 + a_{03}\alpha^3 + a_{04}\alpha^4 + a_{05}\alpha^5 + a_{06}\alpha^6 + a_{07}\alpha^7) \beta + \\
 &\quad (a_{10} + a_{11}\alpha + a_{12}\alpha^2 + a_{13}\alpha^3 + a_{14}\alpha^4 + a_{15}\alpha^5 + a_{16}\alpha^6 + a_{17}\alpha^7) \beta^{2^8} \\
 &\quad \text{where } a_{ij} \in \mathbb{F}_2 \text{ for } i = 0, 1 \text{ and } j = 0, 1, 2, 3, 4, 5, 6, 7.
 \end{aligned}$$

The basis elements represented as a power the root of defining polynomial of $\mathbb{F}_{2^{16}}$ are $\alpha = \omega^{257}$ and $\beta = \omega^{20921}$, giving the 16 elements of the tower field basis:

$$\begin{array}{llll}
t_0 = \beta & t_4 = \alpha^4\beta & t_8 = \beta^{256} & t_{12} = \alpha^4\beta^{256} \\
t_1 = \alpha\beta & t_5 = \alpha^5\beta & t_9 = \alpha\beta^{256} & t_{13} = \alpha^5\beta^{256} \\
t_2 = \alpha^2\beta & t_6 = \alpha^6\beta & t_{10} = \alpha^2\beta^{256} & t_{14} = \alpha^6\beta^{256} \\
t_3 = \alpha^3\beta & t_7 = \alpha^7\beta & t_{11} = \alpha^3\beta^{256} & t_{15} = \alpha^7\beta^{256}
\end{array}$$

Below are the obtained conversion matrices M_N^T and M_T^N with their Hamming weights 106 and 102.

$$M_T^N = \begin{bmatrix}
1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{bmatrix}$$

$$M_N^T = \begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}$$

3.6.3 Module WGP_T

This tower construction yields a circuit for WGP_T that is almost identical to the circuit in Figure 3.11, which was obtained for the tower construction $\mathbb{F}_{(2^4)^4}$. There are two differences:

- instead of NOT operator that was used for addition of element 1 in the previous two tower constructions, we now perform modulo-2 addition with element \mathbf{x}^{8080} , and
- the expression for the trace computation is different from expression obtained in 3.28 for tower construction $\mathbb{F}_{(2^4)^4}$

If we consider replacing the two NOT operators in Figure 3.11, as was described above, and think of M_{16} , I_{16} and $\text{Tr}(\bullet)$ modules as black boxes, there is no need to provide a separate schematic for the module WGP_T for the current tower construction $\mathbb{F}_{(2^8)^2}$.

■ **Remark:** The value x“8080” above is the element 1 represented in tower field basis of $\mathbb{F}_{(2^8)^2}/\mathbb{F}_2$. Namely, the element 1 can be rewritten as $1 = \beta + \beta^{256} = 1_8 \cdot \beta + 1_8 \cdot \beta^{256}$ with the two coefficients $1_8 \in \mathbb{F}_{2^8}$. Element $1_8 \in \mathbb{F}_{2^8}$ is written in polynomial basis as “10000000”, which equals x“80” in hexadecimal representation. ■

Trace computation in the tower field

Let us now take a look at the trace computation, using similar notation as before: $K = \mathbb{F}_2$, $K_1 = \mathbb{F}_{2^8}$ and $K_2 = \mathbb{F}_{(2^8)^2}$. Using associativity of the trace function we compute the trace of the element $Z = z_0\beta + z_1\beta^{256} \in \mathbb{F}_{(2^8)^2}$, with $z_i \in \mathbb{F}_{2^8}$ for $i = 0, 1$ as follows:

$$\begin{aligned} \text{Tr}_{K_1}^{K_2}(Z) &= Z + Z^{2^8} \\ &= z_0\beta + z_1\beta^{256} + (z_0\beta + z_1\beta^{256})^{2^8} \\ &= (z_0 + z_1)(\beta + \beta^{256}) \\ &= z_0 + z_1 \end{aligned}$$

In the third line of expression above we use the fact that $\beta^{2^{16}} = \beta$, and in the last line the relationship $\beta + \beta^{256} = 1$. Since the z_0, z_1 belong to the subfield \mathbb{F}_{2^8} , we can rewrite them in terms of the polynomial basis $\{1, \alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7\}$ as follows:

$$z_i = \sum_{j=0}^7 z_{i,j} \alpha^j,$$

where $i = 0, 1$ and $z_{i,j} \in \mathbb{F}_2$. Using notation $w_j = z_{0,j} + z_{1,j}$, for $j = 0, \dots, 7$, the sum $z_0 + z_1$ becomes $W = \sum_{j=0}^7 w_j \alpha^j \in \mathbb{F}_{2^8}$. Omitting the details, the bottom level trace function then yields:

$$\begin{aligned} \text{Tr}_K^{K_1}(W) &= W + W^2 + W^{2^2} + W^{2^3} + W^{2^4} + W^{2^5} + W^{2^6} + W^{2^7} \\ &= w_5 \\ &= z_{0,5} + z_{1,5} \end{aligned} \tag{3.29}$$

If we now rewrite 3.29 using $k = 8i + j$, that is understanding the indices of $z_{i,j}$ as a base-8 notation, we obtain the following expression for the trace function of the element in its tower field basis representation $Z = \sum_{k=0}^{15} z_k \cdot t_k \in \mathbb{F}_{(2^8)^2}$:

$$\text{Tr}(Z) = \text{Tr}_K^{K_1}(\text{Tr}_{K_1}^{K_2}(Z)) = z_5 + z_{13} \tag{3.30}$$

3.7 Finite field $\mathbb{F}_{2^{16}}$ - summary of field constructions

We can summarize this chapter as follows: due to the properties of the trace function arising from the highly regular tower construction $\mathbb{F}_{((2^2)^2)^2}$, optimizations were possible in the computation of $\text{WGT-16}(X^d)$. Resulting WGP_T module in Figure 3.12 has only 4 multipliers and two of them are reused for the $\text{WGP-16}(X^d)$ computation during the initialization phase. All other constructions lead to the same top-level architecture with 6 multipliers that can be seen in Figure 3.13.

Architecture 1 - multiplier reuse in $\mathbb{F}_{((2^2)^2)^2}$

Here we just repeat the circuit from Section 3.4.3, that was obtained by multiplier reuse permitted by the use of construction $\mathbb{F}_{((2^2)^2)^2}$. Notation used:

$$\begin{aligned}
 a &= Y^{2^{11}+1} & c &= a \oplus_{16} b & e &= b^2 \oplus_{16} c & g &= X^d \oplus_{16} a & a_1 &= Y^{2^6} \otimes_{16} c \\
 b &= Y^{2^{11}-1} & d &= Y \oplus_{16} a & f &= Y^{2^6} \odot_{16} e & h &= b^{2^{11}} & b_1 &= Y \otimes_{16} h \\
 t_1 &= \text{Tr}(d) & t_2 &= \text{Tr}(f) & & & & & c_1 &= a_1 \oplus_{16} b_1
 \end{aligned}$$

The keystream WGT is computed as $\text{WGT-16}(X^d) = t_1 \oplus_1 t_2$, and the initialization feedback WGP as $\text{WGP-16}(X^d) = g_1 = g \oplus_{16} c_1$. A detailed description of the circuit can be found at the end of Section 3.4.3.

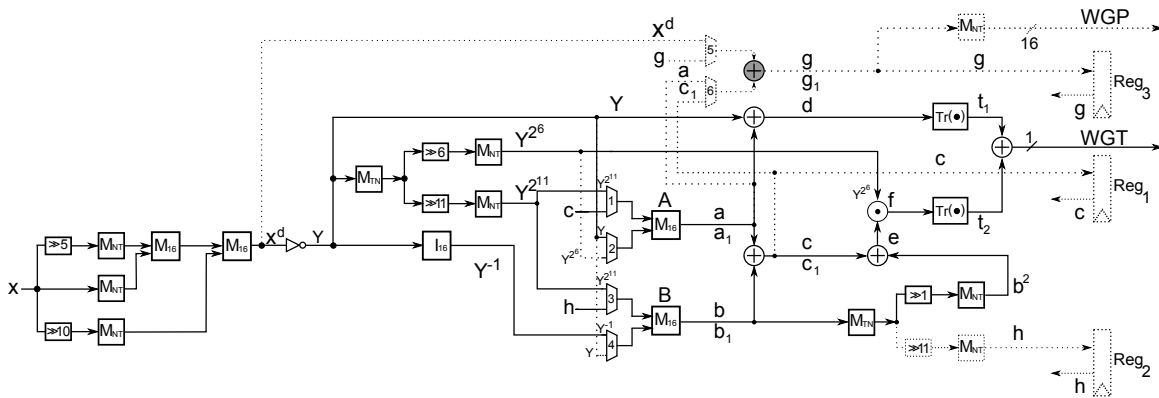


Figure 3.12: Module WGP_T with multiplier reuse using tower field $\mathbb{F}_{((2^2)^2)^2}$

Architecture 2 - all other field constructions

The remaining four WGP_T modules differ at the top-level only in the exponentiations to the powers of 2, representation of element 1 and different trace computation. At this point we consider the multiplier and the inverter as black boxes, but these submodules in fact differ significantly for each field construction. Marking the exponentiation blocks with 2^k , we show the unified top-level schematic for the WGP_T module in Figure 3.13. Notation used:

- $Y = X^d \oplus_{16} 1$ with $d = 1057 = 2^{10} + 2^5 + 1$,
- $A = Y \otimes_{16} (Y^{2^{11}} \oplus_{16} (Y^{-1})^{2^{11}} \otimes_{16} Y^{2^6})$,
- $B = Y^{2^6} \otimes_{16} Y^{2^{11}} \otimes_{16} (Y \oplus_{16} Y^{-1})$,
- $q(Y) = Y \oplus_{16} A \oplus_{16} B$,
- $\text{WGP-16}(X^d) = q(Y) \oplus_{16} 1$ and finally
- $\text{WGT-16}(X^d) = \text{Tr}(\text{WGP-16}(X^d))$.

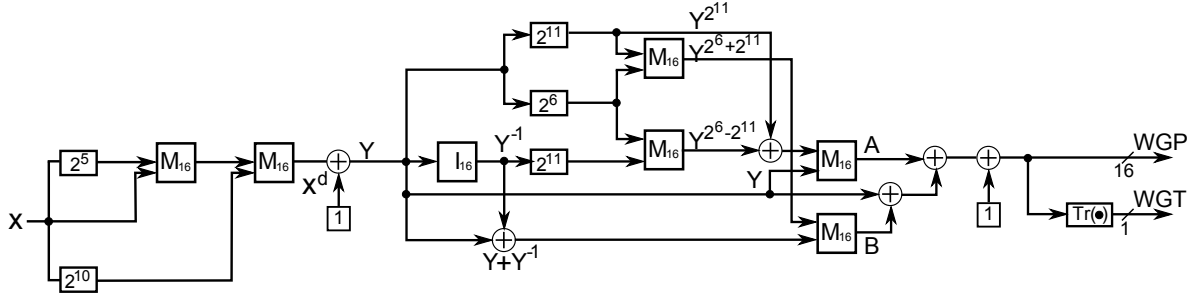


Figure 3.13: Module WGP_T for all other field constructions

For details and differences refer to the following Sections:

- construction of $\mathbb{F}_{2^{16}}$ with polynomial basis representation of elements (Section 3.2)
- construction of $\mathbb{F}_{2^{16}}$ with normal basis representation of elements (Section 3.3)
- tower construction $\mathbb{F}_{(2^4)^4}$ (Section 3.5)
- tower construction $\mathbb{F}_{(2^8)^2}$ (Section 3.6)

Chapter 4

Implementation

Chapter 4 closely follows the structure of Chapter 3, as can be seen in Figure 4.1: with the exception of Section 4.1 all the Sections in Chapters 3 and 4 are paralleled. In Chapter 3 we represented each field construction and developed the top-view circuit of the `WGP_T` module. The exponentiation to the powers of two, representation of the element 1 and trace computation were explained, but the multiplication and inversion blocks were regarded as black boxes because they are closely linked with the field construction itself, i.e. they depend on the basis chosen for representation of the field elements. While the Chapter 3 could be called “theoretical”, the Chapter 4 is dedicated to the actual implementation on a FPGA. Sections 4.2 to 4.6, paralleled with their respective field constructions in Sections 3.2 to 3.6, give detailed descriptions of the five `WGP_T` modules and their implementations. Each of these Sections begins with the description and implementation of the basic building blocks (the submodules performing the finite field arithmetic in the particular basis) and continues with implementation of the `WGP_T` module. In order to achieve a higher throughput, we decide to pipeline the `WGP_T` modules; that is, we divide the `WGP_T` circuit into smaller parts (*pipeline stages*) and separate them by registers. The benefit of a pipelined architecture is twofold: (a) the critical path between the registers is shorter, resulting in shorter clock period, and (b) a different chunk of data (also called *parcel*) can be processed in each stage at the same time and finally passed on to the next pipeline stage, which means that there is a certain level of overlapping for each new computation. The decision about the number of pipeline stages for the `WGP_T` module depends on the module itself (meaning that the top-view circuits developed in the Chapter 3 naturally dictate the insertion of interstage registers at certain positions) and on the particular field construction, which leads to differences in the basic building blocks for the `WGP_T` module. Decisions about the pipelining granularity also affect the third component in WG-16, the FSM. It

is not surprising that the FSM will be different for each particular `WGP_T` implementation. We avoid actually implementing the corresponding FSM's and wait for the `WGP_T` implementation results: the FSM (and finally the entire `WG-16` itself) will be implemented only for the best `WGP_T` modules. Section 4.7 of Chapter 4 gives an overview and a detailed analysis of the implementation results.

Remark: We report the FPGA implementation results for particular modules in terms of resources used by the module: number of flip-flops, denoted `#FFs`, number of LUTs, denoted `#LUTs` and number of slices `#Slices`. The “route-thru” LUTs and memory LUTs were taken into account the same as the LUTs used for logic. For the time complexity we give the clock period for registered modules and block delay for the combinational modules, both denoted as `t`, given in nanoseconds. The total resources available on the chosen Xilinx Spartan-6 FPGA `xc6s1x9-csg324` were listed in Table 2.1 in Section 2.1. We report the post place-and-route results obtained using Xilinx-ISE v14.5 [47]. For the best design we also provide ASIC results obtained for the 65nm CMOS technology, using Synopsys Design Compiler for synthesis [45] and Cadence SoC Encounter to complete the Place-and-Route phase, in terms of gate equivalents `GE` for the area and clock period or block delay `t` for the time complexity. We used VHDL (Very-High-Speed Integrated Circuit HDL, where HDL stands for Hardware Description Language) for design entry.

The implementation results show that pipelining at a lower level of the tower field reduces the clock period, while increasing the area. As was explained in Section 3.4.3, the tower construction $\mathbb{F}_{((2^2)^2)^2}$ leads to algebraic optimizations, which remove two multipliers. This field construction also allows the biggest freedom in choosing appropriate pipelining granularity. It is thus not surprising that $\mathbb{F}_{((2^2)^2)^2}$ turns out to be the best option for implementation of `WGP_T` .

Chapter 3		Chapter 4	
3	WGP_T module and different field constructions	4	implementation
3.1	Finite field $\mathbb{F}_{2^{16}}$ overview	4.1	the WG-16 LFSR
3.2	$\mathbb{F}_{2^{16}}$ with polynomial basis	4.2	$\mathbb{F}_{2^{16}}$ with polynomial basis
3.2.1	field construction	4.2.1	basic building blocks
3.2.2	WGP_T module	4.2.2	WGP_T module
3.3	$\mathbb{F}_{2^{16}}$ with normal basis	4.3	$\mathbb{F}_{2^{16}}$ with normal basis
3.3.1	field construction	4.3.1	basic building blocks
3.3.2	WGP_T module	4.3.2	WGP_T module
3.4	tower $\mathbb{F}_{((2^2)^2)^2}$	4.4	tower $\mathbb{F}_{((2^2)^2)^2}$
3.4.1	field construction	4.4.1	basic building blocks
3.4.2	conversion matrices	4.4.2,3	WGP_T module
3.4.3	WGT_P module	4.4.4	the FSM
3.5	tower $\mathbb{F}_{(2^4)^4}$	4.4.5	the WG-16
3.5.1	field construction	4.5	tower $\mathbb{F}_{(2^4)^4}$
3.5.2	conversion matrices	4.5.1	basic building blocks
3.5.3	WGP_T module	4.5.2	WGP_T module
3.6	tower $\mathbb{F}_{(2^8)^2}$	4.6	tower $\mathbb{F}_{(2^8)^2}$
3.6.1	field construction	4.6.1	basic building blocks
3.6.2	conversion matrices	4.6.2	WGP_T module
3.6.3	WGP_T module	4.7	summary and conclusions
3.7	finite field $\mathbb{F}_{2^{16}}$ summary		

Figure 4.1: Chapters 3 and 4 - roadmap

4.1 The WG-16 LFSR

As already mentioned in Section 2.4.1, the $WG_d(16, 32)$ LFSR uses the feedback polynomial $\ell(x) = x^{32} + x^{25} + x^{16} + x^7 + \omega^{2743}$. The LFSR has 32 stages, denoted S_k , i.e. it is composed of 32 serially connected 16-bit registers. The operation of this memory array is controlled with a 1-bit `lfsr_en` control signal: when `lfsr_en` is set, the registers shift to the right and the register S_{31} takes a new value; otherwise, the registers hold their values. The enable signal `lfsr_en` is needed because the LFSR steps do not correspond to clock cycles. Namely, during:

- loading phase: a new value for the LFSR is received every clock cycle, through the 16-bit data input `DIN`;
- initialization phase: a new value for the LFSR is available when the $WGP(S_{31}^d)$ is computed - the number of cycles required for the $WGP(S_{31}^d)$ computation depends on the particular implementation of the `WGP_T` module. The result $WGP(S_{31}^d)$ is then XOR-ed with the LFSR feedback value `f`;
- running phase: the LFSR is updated every clock cycle with the LFSR feedback `f`.

The LFSR feedback due to $\ell(x)$: $\mathbf{f} = (\omega^{2743} \odot_{16} S_0) \oplus_{16} S_7 \oplus_{16} S_{16} \oplus_{16} S_{25}$.

To feed the LFSR with the appropriate value, two multiplexers (and two corresponding control signals) are needed, as can be seen in Figure 4.2. Multiplexer 1, controlled with signal `load`, chooses between the data input `DIN` and multiplexer 2 output. Multiplexer 2 is controlled with signal `init`; it will pass the value $\mathbf{w} = WGP(S_{31}^d) \oplus_{16} \mathbf{f}$ when `init` is set and the feedback value `f` otherwise. Note that all multiplexer inputs and outputs are 16-bits wide. The input/output tables for the two multiplexers can be seen in Figure 4.2. The circuit for module `LFSR` is shaded in Figure 4.2. Table 4.2 shows the implementation results for module `LFSR`.

The values of the control signals for the LFSR module are given in the Table 4.1. These values will be set by the FSM control circuit. During the initialization phase, the signal `lfsr_en` will be driven by the output signal `doneWGP` that is set when the new $WGP(S_{31}^d)$ is available. During the running phase, we want to be able to stop the cipher - which means we need to stop the LFSR as well - for that purpose, a chip enable `ce` signal will drive the `lfsr_en` signal. The `load` and `init` signals choose which value the LFSR is updated with (as was explained in the beginning of this section); the update value `a` (i.e. the value of the signal `a` in Figure 4.2) is listed in the last column of the Table 4.1.

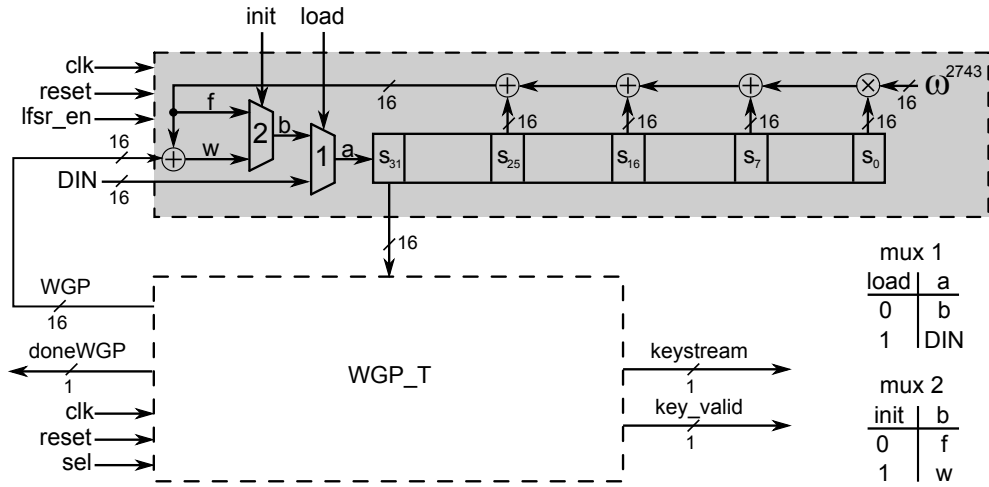


Figure 4.2: The LFSR module, connected to module WGP_T

	lfsr_en	load	init	a
loading	1	1	0	DIN
initialization	doneWGP	0	1	w
running	ce	0	0	f

Table 4.1: The LFSR module - values of the control signals and the input a of the LFSR, depending on loading, initialization and running phase

Module	FPGA Results			
	#FFs	#LUTs	#Slices	t [ns]
LFSR	152	145	47	3.191

Table 4.2: The LFSR module - implementation results

4.1.1 Multiplication with ω^{2743}

The LFSR stays the same for all five WGP_T modules with only one difference: the multiplication with the constant ω^{2743} , which depends on the choice of basis for the representation of field elements. It is implemented in a separate submodule based on the multiplication matrix, and when we change the WGP_T module we must change the multiplication matrix

accordingly. The matrix given below was derived for tower construction $\mathbb{F}_{((2^2)^2)^2}$, using the normal element $\theta = \omega^{1091}$. Note that choice of element $s = \omega^{2743}$ gives the optimal multiplication matrix with Hamming weight 110 when $\theta = \omega^{1091}$ is used. The matrix was obtained by taking the normal basis representation of elements $s \cdot \theta^{2^i}$, $0 \leq i \leq 15$:

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

4.1.2 Serial vs. parallel loading phase

We begin this section by discussing a parallel loading phase that allows to fill the LFSR in only one clock cycle. We compare the parallel implementation to the serial loading phase used and explain advantages of the latter design option when implemented on a Xilinx FPGA device.

Snow3G and ZUC implementations reported by [15] and [19] use a parallel loading of key/IV, implemented via OR gates between the LFSR stages, as can be seen in Figure 4.3:

- in the loading phase, the contents of LFSR registers S_i are set to zero, while the corresponding key/IV values appear on the second input to the OR gate, hence, at the end of the clock cycle, register S_{i+1} is updated to the corresponding key/IV value;
- during the running phase, the second input to the OR gate is set to zero, so that the OR gates just propagate the values through the LFSR states.

The Snow3G LFSR consists of 16 stages, 32 bit each, hence the OR gate inputs are 32-bit wide. Top level architecture has two 128-bit inputs for the key and IV respectively. A submodule called *Initial Operations* then mixes the key and the IV to form 16 32-bit values that are then routed to OR gates between the LFSR registers. No further details on *Initial Operations* were given in the original paper [15], but we can assume the parallel key/IV loading is done in a single clock cycle.

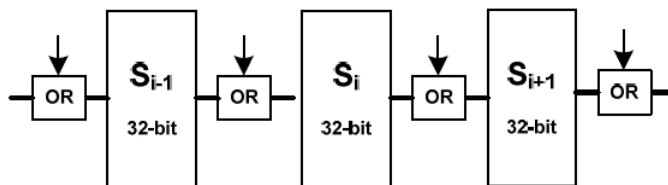


Figure 4.3: The parallelLFSR module - parallel key/IV loading [15]

A serial loading phase (as is used in this WG-16 implementation) for Snow3G would take 16 clock cycles, but only needs a 32-bit input for the “premixed” LFSR initial values. The *Initial Operations* circuit would be omitted completely. Since the initialization phase for Snow3G takes 32 clock cycles, discards the first output after initialization, then enters running phase where it produces a new keystream word every clock cycle, the additional resources needed for *Initial Operations*, OR gates to accommodate parallel loading and wider input signals seems to be a reasonable area-time trade-off compared to a serial loading phase.

As was explained in Section 2.4.1, the initial mixing of the key and IV are not a part of WG-16 circuit; the initial LFSR values are precomputed and the loaded into LFSR registers serially in 32 cycles through a 16-bit input port DIN, as is shown in Figure 4.3. Another consideration, which renders parallel loading of initial key/IV values unnecessary, is the long initialization phase. The latter requires 64 steps, and each step requires a computation of new WGP value. We are aiming for a pipelined architecture of the WGP_T module, having P pipeline stages. Even for a low $P = 5$ (which is extremely optimistic), the initialization phase would take 320 clock cycles. A 32 cycle loading phase diminishes in comparison with long initialization phase.

To change our implementation according to [15] we insert the OR gates between the LFSR registers and change the input ports to the LFSR submodule. Also, the multiplexer MUX1 from Figure 4.2 is omitted and the OR gate at the input to S_{31} connected directly to the MUX2 output b. Now we need two 128-bit inputs for key and IV. A problem arises: the selected FPGA Spartan6 xc6slx9-3csg324 has only 200 IOBs. In order to compare the two LFSR modules, we run implementation on xc6slx45-csg484, that has a sufficient number of available IOBs. The results of this modified module called parallelLFSR are listed in Table 4.3 (note that the results from original LFSR module are included for easier comparison).

We immediately see that the LFSR with parallel key/IV loading is twice the size of the

Module	FPGA Results			
	#FFs	#LUTs	#Slices	t [ns]
LFSR	152	145	47	3.191
parallelLFSR	512	297	86	4.777

Table 4.3: The LFSR module compared with parallelLFSR

original module. It also has a longer clock period.

Another interesting observation is the big difference in number of flip-flops used. In the original serial LFSR design, Xilinx-ISE tools had more freedom for optimization. The tools were able to implement several Sections of LFSR, more specifically, the Sections between the feedback tap positions, using shift register primitives SRL16 available on SLICEM LUTs. A detailed analysis and register count is provided in Appendix C. Note that we do not have a corresponding ASIC library cell for the SRLs - the ASIC version of WG-16 will thus implement a full 512 registers LFSR.

If we now return to the LFSR with parallel load, we find exactly 512 registers. The inferred OR gates at the register inputs prevent the use of SRL16 [57]. Also, the 32 additional 32-bit OR gates require some additional logic. Since the 512 FFs are no mystery, the parallel module was not inspected thoroughly.

4.2 $\mathbb{F}_{2^{16}}$ with polynomial basis - implementation

In this section we present the implementation of module WGP_T with polynomial basis, which was described in Section 3.2. The first part of this section presents the implementation of the basic building blocks for each level of the tower and the second part the implementation of module WGP_T itself.

4.2.1 Analysis of Basic Building Blocks

Reduction in polynomial basis: Multiplication and squaring of polynomials of degree less than 16 results in a polynomial $d(x)$ of degree less than 31 and modular reduction is

required. The reduction is carried out based on relationship $x^{16} = x^5 + x^3 + x^2 + 1$ and its multiplications by x as follows:

$$\begin{aligned}
x^{16} &= x^5 + x^3 + x^2 + 1 \\
x^{17} &= x^6 + x^4 + x^3 + x \\
x^{18} &= x^7 + x^5 + x^4 + x^2 \\
x^{19} &= x^8 + x^6 + x^5 + x^3 \\
&\dots \\
x^{27} &= x^{14} + x^{13} + x^{11} + x^5 + x^3 + x^2 + 1 \\
x^{28} &= x^{15} + x^{14} + x^{12} + x^6 + x^4 + x^3 + x \\
x^{29} &= x^{15} + x^{13} + x^7 + x^4 + x^3 + 1 \\
x^{30} &= x^{14} + x^8 + x^4 + x^3 + x^2 + x + 1
\end{aligned} \tag{4.1}$$

Note that there are no irreducible trinomials of degree 16 over \mathbb{F}_2 , so the reduction is not optimal; for example the computation of the coefficient c_3 of term x^3 of the reduced result requires 7 XOR gates:

$c_3 = d_3 \oplus d_{16} \oplus d_{17} \oplus d_{19} \oplus d_{27} \oplus d_{28} \oplus d_{29} \oplus d_{30}$, where d_i are the coefficients of the double-length polynomial $d(x)$.

The results of submodule `reduction` are given in Table 4.4.

Squaring: An element $A(x) \in \mathbb{F}_{2^{16}}$ has coefficients $a_i \in F_2$ and $(a_i + a_j)^2 = a_i^2 + a_j^2$ holds for some $i, j = 0, 1, \dots, 15, i \neq j$, therefore the square $A^2(x)$ can be written as follows:

$$A(x) \cdot A(x) \equiv \left(\sum_{i=0}^{m-1} a_i x^i \right) \cdot \left(\sum_{i=0}^{m-1} a_i x^i \right) \equiv \sum_{i=0}^{m-1} a_i x^{2i} \pmod{x^{16}x^5 + x^3 + x^2 + 1} \tag{4.2}$$

The equation (4.2) basically says that the bits of vector A are spread and middle (odd) bits set to 0: $a_0 a_1 \dots a_{14} a_{15} \Rightarrow a_0 0 a_1 0 \dots a_{14} 0 a_{15}$. This step is then followed by reduction. Results of submodule `sq16`, that includes reduction, are listed in Table 4.4.

Exponentiation to powers of 2: Exponentiation to powers of 2 in polynomial basis is simple but by far not as trivial as in normal basis, which is basically just a simple shift (see Section 3.3.2). When polynomial basis representation of the field elements is used, $A^{2^k}(x)$ can be implemented as a sequence of k squarers. As a reference, the implementation results of a submodule that connects 5 squarers, `sq16.5` are given Table 4.4 as well. Additional

pen and paper analysis, that might simplify the consecutive expansions followed by reductions to a simple expression for each individual bit of the result was not performed; we relied on the synthesis tools for optimization.

Multiplication: Multiplication of two polynomials $A(x), B(x) \in \mathbb{F}_{2^{16}}$ can be implemented as a simple convolution. The product $D(x) = A(x)B(x)$ is a polynomial of degree less than 31, and needs to be reduced. Coefficients of $D(x)$ are computed as follows:

$$d_k = \begin{cases} \sum_{i=0}^k a_i b_{k-i}, & k = 0, \dots, 15; \\ \sum_{i=k}^{30} a_{k-i+15} b_{i-15}, & k = 16, \dots, 30; \end{cases} \quad (4.3)$$

Different multiplication algorithms could give better implementation results, however the above 2-step classic multiplication simple and appropriate enough for a field as small as $\mathbb{F}_{2^{16}}$. The results of the multiplier `mul16`, that includes reduction, are listed in Table 4.4.

■ **Remark: Outline of an alternative design:** Since we aim for a pipelined design, we could use for, for example, Montgomery representation of field elements, performing the transition to Montgomery representation at the beginning of the pipeline, and transition back to polynomial basis just before the trace computation. All the operations in between can easily be conducted for Montgomery operands. This would allow to spread one operation into several pipeline stages, for example 4 stages, computing 4 bits of the product per stage. ■

Inversion: The critical element in circuit from Figure 3.3 from Section 3.2 is the inversion. Two different approaches to inversion were investigated: the Extended Euclidean Algorithm (EEA) and a square and multiply method. A detailed description of EEA implementation can be found in Appendix D.

The square and multiply method starts with the finite field analogue of Fermat's little theorem $Y^{-1} = Y^{2^m-2}$. For a nonzero element $Y \in \mathbb{F}_{2^{16}}$ we can write:

$$\begin{aligned} Y^{-1} &= Y^{2^{16}-2} = (Y^{2^{15}-1})^2 \\ Y^{2^{15}-1} &= Y \cdot Y^{-1} \cdot Y^{2^{15}-1} = Y \cdot Y^{2^{15}-2} = Y \cdot (Y^{2^{14}-1})^2 \\ Y^{2^{14}-1} &= Y^{(2^7-1)(2^7+1)} = Y^{(2^7-1) \cdot 2^7} \cdot Y^{2^7-1} \end{aligned}$$

Continuing the procedure we obtain:

$$\begin{aligned}
 U &= Y^{2^7-1} = Y \cdot (Y^{2^6-1})^2 \\
 V &= Y^{2^6-1} = Y^{(2^3-1)(2^3+1)} = Y^{(2^3-1) \cdot 2^3} \cdot Y^{2^3-1} \\
 W &= Y^{2^3-1} = Y \cdot (Y^{2^2-1})^2 = Y \cdot (Y^3)^2 \\
 Y^3 &= Y \cdot Y^2
 \end{aligned}$$

Putting this back together, we can compute the inverse as

$$Y^{-1} = \left(Y \cdot \left(U^{2^7} \cdot U \right)^2 \right)^2 \quad \text{where} \quad U = Y \cdot \left(W^{2^3} \cdot W \right)^2 \quad \text{and} \quad W = Y \cdot (Y^3)^2. \quad (4.4)$$

The circuit obtained in such a way can be implemented in a six stage pipeline, and it was implemented in the submodule I_{16} . Its schematic is shown in Figure 4.4: the grey vertical dashed lines represent the pipeline stage borders and a 16-bit interstage register is implemented for each signal crossing the border. Since it is a pipelined module the implementation results for the inverter are listed in Table 4.5 instead of in Table 4.4; the latter shows the results for other combinational arithmetic modules.

Basic Building Block	FPGA Results		
	# of LUTs	# of Slices	t [ns]
reduction	17	10	8.122
sq16	8	6	7.301
sq16_5	19	9	8.070
mul16	119	46	11.812

Table 4.4: Basic building blocks for polynomial basis arithmetic - implementation results

4.2.2 Module WGP_T using polynomial basis

Following the circuit in Figure 3.3 derived in Section 3.2 and pipelined structure of inverter I_{16} we develop a 12 stage pipeline for implementation of the WGT module; it can be seen in Figure 4.5. The grey shaded area is the inversion submodule I_{16} . For better performance, the blocks SQ6 and SQ5 in Figure 3.3 have been broken up and distributed among the pipeline stages of the inverter. One multiplier was also embedded in the last stage of the inverter pipeline. Implementation results for `wgtPB` module, together with the results

for the inverter I_{16} , are given in Table 4.5. We can see that inversion takes up roughly a third of the entire area complexity, which is not surprising, since inversion is considered one of the most expensive operations in general.

Module	FPGA Results			
	#FFs	#LUTs	#Slices	t [ns]
I ₁₆	248	1054	369	7.271
WGP_T_PB	616	1827	603	7.438

Table 4.5: Polynomial basis inversion I₁₆ and WGP_T module WGP_T_PB - implementation results

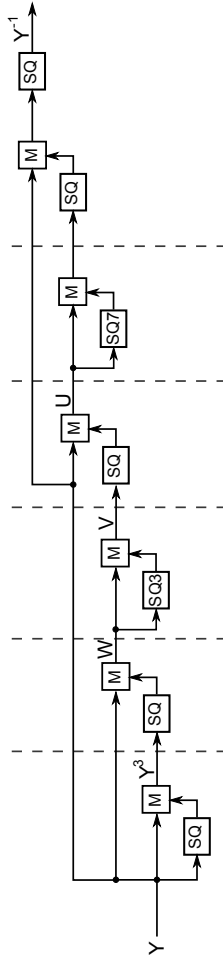


Figure 4.4: Inversion submodule I₁₆ for inversion in polynomial basis

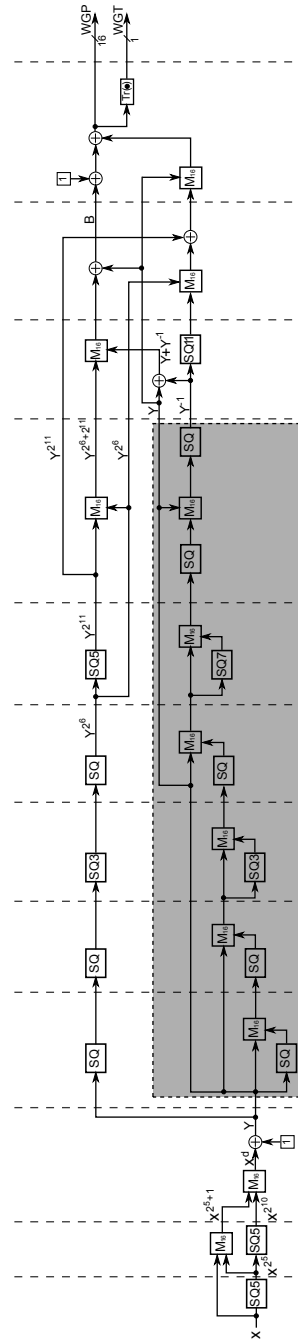


Figure 4.5: Module WGP_T_PB - pipelined architecture for module WGP_T in polynomial basis

4.3 $\mathbb{F}_{2^{16}}$ with normal basis - implementation

The normal basis representation of $\mathbb{F}_{2^{16}}$ elements was first introduced in Section 3.1, where we mentioned the exhaustive search for the the normal element, and then revisited in Section 3.3, where the top-level schematic of the `WGP_T` module was presented.

4.3.1 Analysis of Basic Building Blocks

In Section 3.3.2 we discussed the exponentiation of field elements to powers of two and have explained that it can be implemented as a simple cyclic shift. The inversion of field elements can be implemented using the square and multiply method, which was discussed in the previous Section 4.2.1 and summarized in equation (4.4). The inversion submodule using the normal basis representation of field elements looks similar to `inv16` in Figure 4.4, with the squaring blocks replaced by right cyclic shifts for the appropriate number of bits. The inversion submodule employs the multiplication block. Hence, multiplication is the fundamental building block of the normal basis implementation. We used the Massey-Omura parallel multiplier, that was designed in [99]. In this section we provide a brief description of the multiplier; interested reader should refer to [99, 65].

Recall from Section 3.1 the representation of the element $A \in \mathbb{F}_{2^m}$ in normal basis $N = \{\theta, \theta^2, \dots, \theta^{2^{m-1}}\}$:

$$A = \sum_{i=0}^{m-1} a_i \theta^{2^i} = a_0 \theta + a_1 \theta^2 + \dots + a_{m-1} \theta^{2^{m-1}} \text{ where } a_i \in \mathbb{F}_2$$

The above sum can also be written as a product of two vectors, one of them being the vector $\mathbf{a} = (a_0, a_1, \dots, a_{m-1})$ of coefficients of the element A and the other a (transposed, marked with T) vector of basis elements $\boldsymbol{\theta} = (\theta, \theta^2, \dots, \theta^{2^{m-1}})$, as follows:

$$A = \mathbf{a}\boldsymbol{\theta}^T = \boldsymbol{\theta}\mathbf{a}^T$$

The product C of two elements $A, B \in \mathbb{F}_{2^m}$ can now be written as:

$$\begin{aligned} C &= AB \\ &= (\mathbf{a}\boldsymbol{\theta}^T)(\boldsymbol{\theta}\mathbf{b}^T) \\ &= \mathbf{a}(\boldsymbol{\theta}^T\boldsymbol{\theta})\mathbf{b}^T \\ &= \mathbf{a}\mathbf{M}\mathbf{b}^T \end{aligned}$$

The $(m \times m)$ matrix \mathbf{M} is also called multiplication matrix and the element of this matrix in row i and column j , where $i, j = 0, \dots, m-1$, is the product of two basis elements $\theta^{2^i} \theta^{2^j}$, hence $\mathbf{M} = \left[\theta^{2^i + 2^j} \right]_{i,j=0}^{m-1}$. Note that this matrix should not be mistaken with multiplication matrix T defined with equation (3.1) in Section 3.1. The i -th row of matrix T is the normal basis representation of the element $\theta \cdot \theta^{2^i}$, for $i = 0, \dots, m-1$, which makes the matrix T the multiplication matrix of the normal element θ . Matrix \mathbf{M} is a symmetric matrix with normal basis elements on its diagonal, and can be decomposed as $\mathbf{M} = \mathbf{D} + \mathbf{U} + \mathbf{U}^T$. Matrix \mathbf{U} is a triangular matrix whose nonzero components can be expressed in terms of powers of δ_i , where $\delta_i = \theta^{1+2^i}$ for $i = 1, \dots, v$, $v = \lceil \frac{m-1}{2} \rceil$, for more details refer to [99]. Using the δ_i and $x_{j,i} = (a_j b_{(i+j) \bmod m} + a_{(i+j) \bmod m} b_j)$, where $i = 1, \dots, v$, $j = 0, \dots, m-1$ and $v = \lceil \frac{m-1}{2} \rceil$ and for m even, the product C is finally given as

$$C = \sum_{j=0}^{m-1} a_j b_j \theta^{2^{(j+1)} \bmod m} + \sum_{i=1}^{v-1} \sum_{j=0}^{m-1} x_{j,i} \delta_i^{2^j} + \sum_{i=1}^{v-1} x_{j,v} \delta_v^{2^j} \quad (4.5)$$

We now transform the equation 4.5 into a circuit, beginning with the middle sum. Let us first take a closer look at the term $x_{j,i} = (a_j b_{(i+j) \bmod m} + a_{(i+j) \bmod m} b_j)$. Setting $k = (i+j) \bmod m$, we define $\text{conv}(j, k) = a_j b_k + a_k b_j$.

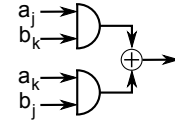


Figure 4.6: Normal basis multiplier M_{16} - computation of coefficient $x_{j,i}$ with $k = (i+j) \bmod m$ in block M_{16} in $\mathbb{F}_{2^{16}}$

For $m = 16$, we get $v = 8$, which means that we need the values of δ_i for $i = 1, \dots, 7$ to compute the middle sum in equation 4.5. Each δ_i generates the vector v_i , which we denote partial product, by first obtaining the vector $x = (x_{0,i}, x_{1,i}, \dots, x_{15,i})$, then generating shifted versions of x_i as dictated by δ_i and XORing them with x_i . The procedure is summarized in Table 4.6 below. The second column of the table shows the normal basis representation of the element δ_i , and completely dictates the generation of the corresponding $v_i = \bigoplus_{\ell \in L} (x_i \ll \ell)$ (notation will be explained shortly): the index set L is listed in the fourth column. The third column shows how the j th bit of vector x_i is obtained by listing the values (u, k) for $x_i(j) = \text{conv}((j+u) \bmod 16, (j+k) \bmod 16)$, where $k = (i+u) \bmod m$ and u is the value above a certain bit in the representation of δ_i . The distance between the 1's in δ_i equals the number of bits the x_i is shifted to the left: we shall use notation $(x_i \ll \ell)$ for a left cyclic shift for ℓ positions where the starting point is considered to be the bit with the mark u . The last column of Table 4.6 shows the Hamming weight of δ_i , which equals the number of 16-bit XOR gates needed to obtain the vector v_i .

i	δ_i in normal basis	(u, k)	L	$\text{HW}(\delta_i)$
1	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0	(2, 3)	0, 1, 2, 3	4
2	0 0 0 0 1 0 0 1 1 0 0 1 0 0 0 0	(5, 7)	0, 3, 4, 7	4
3	1 1 0 0 0 1 1 1 0 1 0 0 0 1 1 0	(0, 3)	0, 2, 3, 7, 9, 10, 11, 15	8
4	0 1 1 0 0 0 0 1 0 0 1 1 0 1 1 0	(2, 6)	0, 1, 3, 4, 7, 12, 13	7
5	1 0 0 0 0 0 0 1 0 1 1 1 1 0 0 0	(0, 5)	0, 4, 5, 6, 7, 9	6
6	1 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1	(0, 6)	0, 1, 3, 5, 7	5
7	0 1 0 0 1 1 0 0 0 0 1 0 1 0 1 1	(1, 8)	0, 1, 3, 5, 10, 11, 14	7
8	0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0	(2, 10)	-	-

Table 4.6: Normal basis multiplier - generation of vectors v_i for $i = 1, \dots, 8$

■ **Remark:** note at this point that the 9 vectors δ_i for $i = 0, \dots, 8$ exactly correspond to the first 9 rows of the multiplication matrix T for the normal basis obtained by the element $\theta = \omega^{1117}$, which is listed in B.1.2. ■

For example to generate vector v_2 we first obtain $x_2(j) = \text{conv}((j + 5) \bmod 16, (j + 7) \bmod 16)$ where $j = 0 \dots, 15$. Then we generate the shifted versions of x_2 and XOR them together:

$$v_2 = x_2 \oplus (x_2 \ll 3) \oplus (x_2 \ll 4) \oplus (x_2 \ll 7).$$

Generation of v_2 requires two 16-bit AND gates and four 16-bit XOR gates; note that the number of 16-bit XOR gates corresponds to the hamming weight of δ_2 .

The last sum in equation (4.5) is simple: the value δ_8 is included in Table 4.6 and $v_8(j) = \text{conv}((j + 2) \bmod 16, (j + 10) \bmod 16)$ where $j \in \{0, \dots, 15\} \setminus \{6, 14\}$ and $v_8(j) = 0$ for $j = 6, 14$. The first sum in equation (4.5) is also very simple: $v_0 = (\mathbf{a} \odot \mathbf{b}) \gg 1$, that is v_0 is obtained by a 16-bit AND of the two factors and a right cyclic shift for one bit.

The vector \mathbf{c} of the product $C = AB$ is now obtained by adding these vectors: $\mathbf{c} = \bigoplus_{i=0}^8 v_i$. The FPGA results of the implemented multiplier M_{16} are listed in table 4.7 below.

Basic Building Block	FPGA Results		
	# of LUTs	# of Slices	t [ns]
M ₁₆	285	102	13.923

Table 4.7: Normal basis multiplier - implementation results

■ **Remark: Gate count:** Generation of vectors v_i for $i = 1, \dots, 7$ takes the area of 41 16-bit XOR gates and 14 16-bit AND gates. There are additional 9 16-bit XOR gates needed to sum up the vectors v_i , which gives a total of 800 XOR gates and 224 AND gates. In addition, the deepest XOR tree is needed for computation of v_3 , which gives a delay of 3 XOR gates and one AND gate, and additional delay of 4 XOR gates is inferred by the final $\bigoplus_{i=0}^8 v_i$, resulting in $T_A + 7T_X$ delay through the multiplier block. Detailed gate count in terms of NAND gates can be seen in Table E.1 in Appendix E. The authors of [99] predict a circuit with m^2 AND gates and $\frac{m}{2}(C_N + m - 2)$ XOR gates with a delay of $T_A + \lceil \log_2(C_N + 1) \rceil T_x$. For $m = 16$ and $C_N = 85$ this equals to 256 AND gates, 792 XOR gates and a delay of $T_A + 7T_x$, which is very close to our multiplier.

Note that this is an optimized multiplier, with reduced redundancy; the straightforward parallel multiplication requires mC_N AND gates and $m(C_N - 1)$ XOR gates, which would be 1360 AND gates and 1344 XOR gates: the optimization resulting from the use of reduced redundancy multiplier [99] is significant. ■

4.3.2 Module WGP_T using normal basis

The differences between the module WGP_T using polynomial basis from the previous Section 4.2 and WGP_T using normal basis are:

- right cyclic shifts replace squaring
- different multiplication module
- different representation of element 1 (adding 1 is implemented with a NOT)
- different trace computation for $A \in \mathbb{F}_{2^{16}}$: $\text{Tr}(A) = \bigoplus_{i=0}^{15} a_i$

Keeping these differences in mind, we construct an 11 stage pipeline, which looks exactly like the pipeline in Figure 4.5 showing the WGP_T module with polynomial basis, with one more difference: we omit the first pipeline border between the initial shifting and the first multiplier. The implementation results are listed in Table 4.8.

Although the normal basis implementation has its advantages, for example the trivial exponentiations to the powers of two, the difference in area between the polynomial basis multiplier and the normal basis multiplier is significant, and is of course reflected in the area of the two WGP_T modules: even though they reach practically the same clock period, the area of the WGP_T module in the normal basis implementation is doubled.

Module	FPGA Results			
	# of FFs	# of LUTs	# of Slices	t [ns]
WGP_T_NB	606	3835	1168	7.576

Table 4.8: The WGP_T module WGP_T_PB - implementation results

4.4 Tower construction $\mathbb{F}_{((2^2)^2)^2} \cong \mathbb{F}_{2^{16}}$ - implementation

We now present the implementation of the first tower field based WGP_T module. In Section 4.4.1 we analyze the basic building blocks for each level of the tower field $\mathbb{F}_{((2^2)^2)^2}$. The tower construction $\mathbb{F}_{((2^2)^2)^2}$ offers many pipelining possibilities that are explored in Section 4.4.2. The initial pipeline developed in Section 4.4.2 is then subjected to optimizations (Section 4.4.3), and the best modules are chosen to be connected to the WG-16 module; for the chosen pipelines the FSM had to be implemented as well. The FSM is presented in Section 4.4.4, and the final results for the WG-16 modules are presented in Section 4.4.5.

4.4.1 Analysis of Basic Building Blocks

For an implementation using tower field construction $\mathbb{F}_{((2^2)^2)^2}$ we need to implement several arithmetic operations on each level of the tower. The field construction was presented in detail in Section 3.4. We will need finite field squaring, multiplication and inversion. Exponentiations to powers of 2 are performed in the top level of tower construction in normal basis representation, which requires efficient transition matrices, that were listed in Section 3.4.2. In this section we discuss the basic building blocks and their implementation as we ascend through the tower field.

Arithmetic operations in \mathbb{F}_{2^2} .

Squaring: For a non-zero element $A \in \mathbb{F}_{2^2}$, the square of A is calculated as follows:

$$\begin{aligned} A^2 &= (a_0\alpha + a_1\alpha^2)^2 \\ &= a_0\alpha^2 + a_1\alpha^4 \\ &= a_1\alpha + a_0\alpha^2 \\ &= s_0\alpha + s_1\alpha^2 = S. \end{aligned}$$

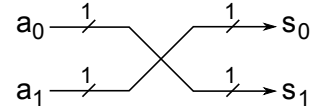


Figure 4.7: Squaring and inversion block S_2 in \mathbb{F}_{2^2}

The coordinates of the square are $s_0 = a_1$ and $s_1 = a_0$, which is implemented by simply rewiring the inputs a_0, a_1 (see Figure 4.7). Note that the inverse of $A \in \mathbb{F}_{2^2}$ is equivalent to the square, since $A^{-1} = A^{2^2-2} = A^2$.

Multiplication: Multiplication of elements $A = a_0\alpha + a_1\alpha^2$ and $B = b_0\alpha + b_1\alpha^2$, where $a_0, a_1, b_0, b_1 \in \mathbb{F}_2$, is computed as follows:

$$\begin{aligned} AB &= (a_0\alpha + a_1\alpha^2)(b_0\alpha + b_1\alpha^2) \\ &= (a_0b_1 + a_1b_0 + a_1b_1)\alpha + (a_0b_1 + a_1b_0 + a_0b_0)\alpha^2 \\ &= c'_0\alpha + c'_1\alpha^2 = C. \end{aligned} \tag{4.6}$$

We can rewrite the coefficients to get more efficient multiplication (similar to Karatsuba Algorithm, for details refer to [79]):

$$\begin{aligned} a_0b_1 + a_1b_0 + a_1b_1 &= (a_0 + a_1)(b_0 + b_1) + a_0b_0 = c_0 \\ a_0b_1 + a_1b_0 + a_0b_0 &= (a_0 + a_1)(b_0 + b_1) + a_1b_1 = c_1 \end{aligned} \tag{4.7}$$

Computing coefficients as shown in equations 4.7 allows a more efficient implementation, as can also be seen from Figures 4.8 and 4.9. In 4.8 we see the straightforward multiplication as given by equation 4.6, which requires 4 AND gates and 3 XOR gates. Figure 4.9 from equation 4.7 contains 3 AND gates and 4 XOR gates. On this level there really is no difference. But when we go up the tower construction, we will replace the AND gates by multiplication modules (for example, in module M_4 for multiplication in $\mathbb{F}_{(2^2)^2}$, the AND gates will be replaced by modules M_2), then, the extra AND gate, i.e. extra multiplication module, will have a significant impact on the performance. The multiplication modules will be revisited in more detail, including the gate count, at the end of this section. The second, more efficient design (Figure 4.9), is implemented in module M_2 .

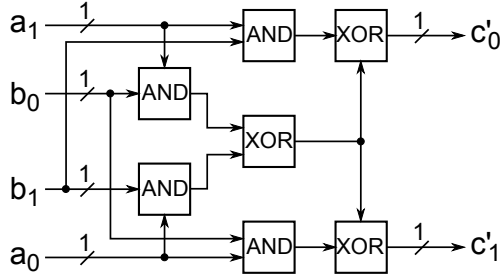


Figure 4.8: Straightforward multiplication $(a_0, a_1)(b_0, b_1) = (c'_0, c'_1)$ from equation 4.6

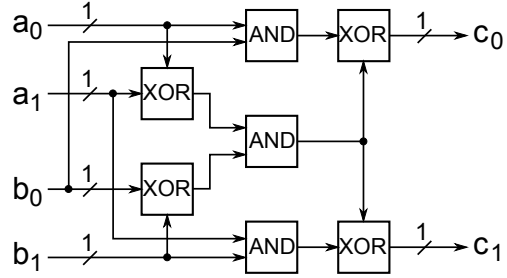


Figure 4.9: More efficient multiplication $(a_0, a_1)(b_0, b_1) = (c_0, c_1)$ from equation 4.7

Auxiliary computations in \mathbb{F}_{2^2} :

The multiplications of $A \in \mathbb{F}_{2^2}$ with constants α and α^2 are carried out as follows (see Figure 4.10):

$$\begin{aligned} \alpha A &= a_0\alpha^2 + a_1(\alpha + \alpha^2) = a_1\alpha + (a_0 + a_1)\alpha^2, \\ \alpha^2 A &= a_0(\alpha + \alpha^2) + a_1\alpha = (a_0 + a_1)\alpha + a_0\alpha^2. \end{aligned} \quad (4.8)$$

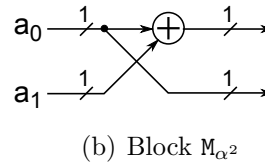
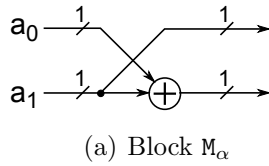


Figure 4.10: Multiplication by α and α^2 in \mathbb{F}_{2^2}

Arithmetic operations in $\mathbb{F}_{(2^2)^2}$.

Squaring: An element $A \in \mathbb{F}_{(2^2)^2}$ is represented with normal basis $\{\beta, \beta^4\}$ as $A = a_0\beta + a_1\beta^4$, where $a_0, a_1 \in \mathbb{F}_{2^2}$. For a non-zero element $A \in \mathbb{F}_{(2^2)^2}$, the square of A is calculated as follows (see Figure 4.11(a)):

$$\begin{aligned} A^2 &= (a_0\beta + a_1\beta^4)^2 = a_0^2\beta^2 + a_1^2\beta^8 \\ &= a_0^2[(\alpha + 1)\beta + \alpha\beta^4] + a_1^2[\alpha\beta + (\alpha + 1)\beta^4] \\ &= [(a_0^2 + a_1^2)\alpha + a_0^2]\beta + [(a_0^2 + a_1^2)\alpha + a_1^2]\beta^4 \\ &= s_0\beta + s_1\beta^4 = S. \end{aligned}$$

Multiplication: Let $A = a_0\beta + a_1\beta^4$ and $B = b_0\beta + b_1\beta^4$, where $a_0, a_1, b_0, b_1 \in \mathbb{F}_{2^2}$. A multiplication $C = AB$ in $\mathbb{F}_{(2^2)^2}$ is computed as follows (see Figure 4.11(b)):

$$\begin{aligned}
 AB &= (a_0\beta + a_1\beta^4)(b_0\beta + b_1\beta^4) \\
 &= a_0b_0\beta^2 + (a_0b_1 + a_1b_0)\beta^5 + a_1b_1\beta^8 \\
 &= [(a_0 + a_1)(b_0 + b_1)\alpha + a_0b_0]\beta + [(a_0 + a_1)(b_0 + b_1)\alpha + a_1b_1]\beta^4 \\
 &= c_0\beta + c_1\beta^4 = C.
 \end{aligned}$$

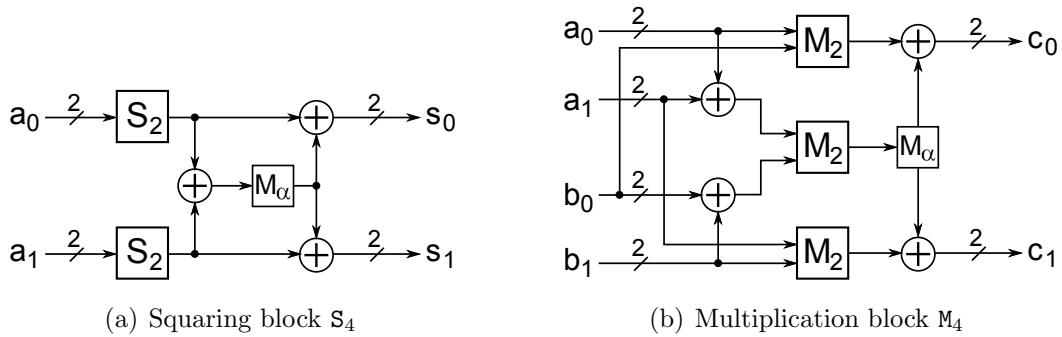


Figure 4.11: Squaring and multiplication in $\mathbb{F}_{(2^2)^2}$

In Figure 4.11(a) we can see two squarers S_2 from the lower level of the tower construction being used to obtain the terms a_0^2 and a_1^2 . Module M_4 in Figure 4.11(b) contains three parallel multipliers M_2 for the computation of partial products $(a_0 + a_1)(b_0 + b_1)$, a_0b_0 and a_1b_1 . Both S_4 and M_4 also use the submodule M_α .

Inversion: For the computation of inverse in composite fields we used the Itoh-Tsuji algorithm (for details refer to the original paper [81], or to [86]).

The inverse of a non-zero element $A \in \mathbb{F}_{(2^n)^m}$ is computed as

$$A^{-1} = (A^r)^{-1} \cdot A^{r-1}, \quad r = \frac{2^{n \cdot m} - 1}{2^n - 1} \tag{4.9}$$

Note that for any $A \in \mathbb{F}_{q^m}$ and for $r = \frac{q^m - 1}{q - 1}$, the element A^r belongs to the subfield \mathbb{F}_q .

For $A = a_0\beta + a_1\beta^4 \in \mathbb{F}_{(2^2)^2}$, we have $r = 5$ and want to compute $A^{-1} = (A^5)^{-1}A^4$.

Let us first look at the second factor A^4 because it is easily obtained and easily implemented (by simply rewiring the inputs); the computation can be seen on the right. Recall that $q = 2^2$ and so A^{2^2} is the Frobenius mapping of A with respect to \mathbb{F}_{2^2} , which is the 4th power operation, a simple cyclic shift.

$$\begin{aligned} A^{5-1} &= A^4 \\ &= (a_0\beta + a_1\beta^4)^4 \\ &= a_0\beta^4 + a_1\beta^{16} \\ &= a_1\beta + a_0\beta^4. \end{aligned}$$

The first factor $D = A^5$ can be computed using A^4 as follows:

$$\begin{aligned} D = A^5 &= AA^4 = (a_0\beta + a_1\beta^4)(a_1\beta + a_0\beta^4) & (4.10) \\ &= a_0a_1\beta^2 + a_0^2\beta^5 + a_1^2\beta^5 + a_0a_1\beta^8 \\ &= a_0a_1(\beta^2 + \beta^8) + (a_0 + a_1)^2\alpha(\beta + \beta^4) \\ &= a_0a_1 + (a_0 + a_1)^2\alpha \end{aligned}$$

In the computation above we used the relationships $\beta + \beta^4 = 1$, $\beta^2 + \beta^8 = 1$ and $\beta^5 = \alpha\beta + \alpha\beta^4$ (refer to Table 3.8 in Section 3.4.1).

To check that A^5 is indeed an element of the subfield the equation in 4.10 is rewritten on the right: since $a_0, a_1, \alpha, \alpha^2 \in \mathbb{F}_{2^2}$, obviously $A^5 \in \mathbb{F}_{2^2}$.

$$\begin{aligned} A^5 &= (a_0 + a_1)^2\alpha + a_0a_1(\alpha + \alpha^2) \\ &= ((a_0 + a_1)^2 + a_0a_1)\alpha + a_0a_1\alpha^2 \end{aligned}$$

But this means that the inverse $D^{-1} = (A^5)^{-1}$ can be computed using the inverter module S_2 from the lower level of the tower construction. The inverse $I = A^{-1}$ is now obtained using (4.9) as follows:

$$\begin{aligned} A^{-1} &= D^{-1} \cdot A^4 \\ &= D^{-1}(a_1\beta + a_0\beta^4) \\ &= a_1D^{-1}\beta + a_0D^{-1}\beta^4 \\ &= i_0\beta + i_1\beta^4 = I. \end{aligned}$$

The inversion module I_4 can be seen in Figure 4.12. Blocks M_2 , S_2 and M_α are used to obtain the value A^5 , which is then lead through inverter I_2 (which in fact is the block S_2) to obtain $D^{-1} = (A^5)^{-1}$. Note that a_0, a_1 and D in the computation above are elements of the subfield \mathbb{F}_{2^2} ; thus we can obtain their products i_0 and i_1 with two parallel multiplication blocks M_2 .

Auxiliary computations in $\mathbb{F}_{(2^2)^2}$: At a higher level of the tower construction multiplications of $A \in \mathbb{F}_{(2^2)^2}$ with constants $\lambda, \lambda^2, \beta$ and $\alpha\beta$ will be applied. The equations (on the left) and their corresponding circuits (on the right) can be seen below:

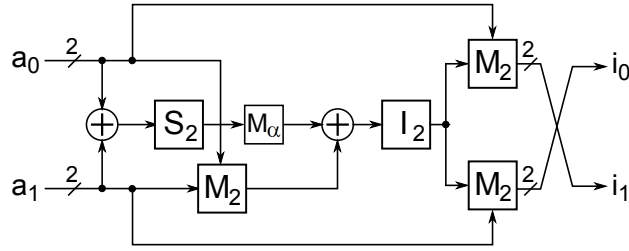


Figure 4.12: Inversion block I_4 in $\mathbb{F}_{(2^2)^2}$

$$\begin{aligned}
 \lambda A &= \alpha^2 \beta A \\
 &= a_0 \alpha^2 [(\alpha + 1)\beta + \alpha \beta^4] + a_1 \alpha^2 (\alpha \beta + \alpha \beta^4) \\
 &= (a_0 \alpha + a_1) \beta + (a_0 + a_1) \beta^4
 \end{aligned}$$

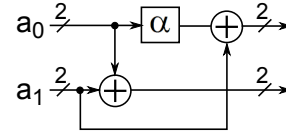


Figure 4.13: Block M_λ

$$\begin{aligned}
 \lambda^2 A &= \alpha \beta^2 A \\
 &= a_0 \alpha \beta^3 + a_1 \alpha \beta^6 = a_0 \alpha (\beta + \alpha \beta^4) + a_1 \alpha^2 \beta \\
 &= (a_0 \alpha + a_1 \alpha^2) \beta + (a_0 \alpha^2) \beta^4
 \end{aligned}$$

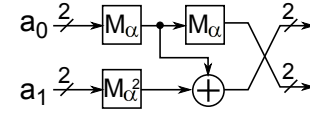


Figure 4.14: Block M_{λ^2}

$$\begin{aligned}
 \beta A &= a_0 [(\alpha + 1)\beta + \alpha \beta^4] + a_1 (\alpha \beta + \alpha \beta^4) \\
 &= [a_0 + (a_0 + a_1)\alpha] \beta + [(a_0 + a_1)\alpha] \beta^4
 \end{aligned}$$

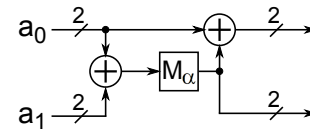


Figure 4.15: Block M_β

$$\begin{aligned}
 \alpha \beta A &= a_0 (\beta + \alpha^2 \beta^4) + a_1 (\alpha^2 \beta + \alpha^2 \beta^4) \\
 &= (a_0 + a_1 \alpha^2) \beta + (a_0 \alpha^2 + a_1 \alpha^2) \beta^4
 \end{aligned}$$

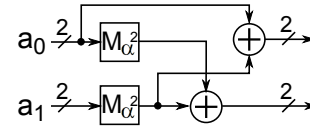


Figure 4.16: Block $M_{\alpha\beta}$

Note that on this level of the tower all arithmetic blocks (modules) use submodules M_α and M_α^2 (Figure 4.10)

Arithmetic operations in $\mathbb{F}_{((2^2)^2)^2}$.

Squaring: For a non-zero element $A \in \mathbb{F}_{((2^2)^2)^2}$, the square of A is calculated as follows:

$$\begin{aligned}
 A^2 &= (a_0\gamma + a_1\gamma^{16})^2 = a_0^2\gamma^2 + a_1^2\gamma^{32} \\
 &= a_0^2[(\lambda + 1)\gamma + \lambda\gamma^{16}] + a_1^2[\lambda\gamma + (\lambda + 1)\gamma^{16}] \\
 &= [(a_0^2 + a_1^2)\lambda + a_0^2]\gamma + [(a_0^2 + a_1^2)\lambda + a_1^2]\gamma^{16} \\
 &= s_0\gamma + s_1\gamma^{16} = S.
 \end{aligned}$$

The squarer S_8 can be seen in Figure 4.17(a). Again, we see two squarers S_4 from the lower level of the tower, but now the module M_α is replaced by M_λ . The highly regular structure of this tower construction is becoming apparent. We will shortly see that the multiplication block M_8 also strongly resembles the multiplication blocks from lower levels of the tower.

Multiplication: Let $A = a_0\gamma + a_1\gamma^{16}$ and $B = b_0\gamma + b_1\gamma^{16}$, where $a_0, a_1, b_0, b_1 \in \mathbb{F}_{(2^2)^2}$. A multiplication $C = AB$ in $\mathbb{F}_{((2^2)^2)^2}$ is carried out as follows (see Figure 4.17(b)):

$$\begin{aligned}
 AB &= (a_0\gamma + a_1\gamma^{16})(b_0\gamma + b_1\gamma^{16}) \\
 &= a_0b_0\gamma^2 + (a_0b_1 + a_1b_0)\gamma^{17} + a_1b_1\gamma^{32} \\
 &= [(a_0 + a_1)(b_0 + b_1)\lambda + a_0b_0]\gamma + \\
 &\quad [(a_0 + a_1)(b_0 + b_1)\lambda + a_1b_1]\gamma^{16} \\
 &= c_0\gamma + c_1\gamma^{16} = C.
 \end{aligned}$$

Inversion: Using Itoh-Tsuji algorithm (4.9), the inverse of a non-zero element $A = a_0\gamma + a_1\gamma^{16} \in \mathbb{F}_{((2^2)^2)^2}$ is computed as $A^{-1} = D^{-1}A^{16}$, where $D = A^{17}$.

The Frobenius mapping of A with respect to \mathbb{F}_{2^4} , which is the 16th power operation, is computed on the right. Again, it is a simple rewiring of the inputs.

$$\begin{aligned}
 A^{16} &= (a_0\gamma + a_1\gamma^{16})^{16} \\
 &= a_0\gamma^{16} + a_1\gamma^{256} \\
 &= a_1\gamma + a_0\gamma^{16}
 \end{aligned}$$

On the right, we see the equation for $D = A^{17}$. We used the relationships $\gamma^{32} + \gamma^2 = 1$ and $\gamma^{17} = \lambda$.

$$\begin{aligned}
 D &= A^{17} \\
 &= AA^{16} \\
 &= (a_0\gamma + a_1\gamma^{16})(a_1\gamma + a_0\gamma^{16}) \\
 &= a_0a_1\gamma^2 + a_0^2\gamma^{17} + a_1^2\gamma^{17} + a_0a_1\gamma^{32} \\
 &= a_0a_1(\gamma^2 + \gamma^{32}) + (a_0 + a_1)^2\lambda \\
 &= a_0a_1 + (a_0 + a_1)^2\lambda
 \end{aligned}$$

The inverse I of A is calculated as :

$$\begin{aligned}
 A^{-1} &= D^{-1} \cdot A^{16} \\
 &= D^{-1}(a_1\gamma + a_0\gamma^{16}) \\
 &= a_1D^{-1}\gamma + a_0D^{-1}\gamma^{16} \\
 &= i_0\gamma + i_1\gamma^{16} = I,
 \end{aligned}$$

where $D^{-1} = (A^{17})^{-1}$ can be computed with subfield $\mathbb{F}_{(2^2)^2}$ inversion block I_4 . The corresponding circuit can be seen in Figure 4.17(c).

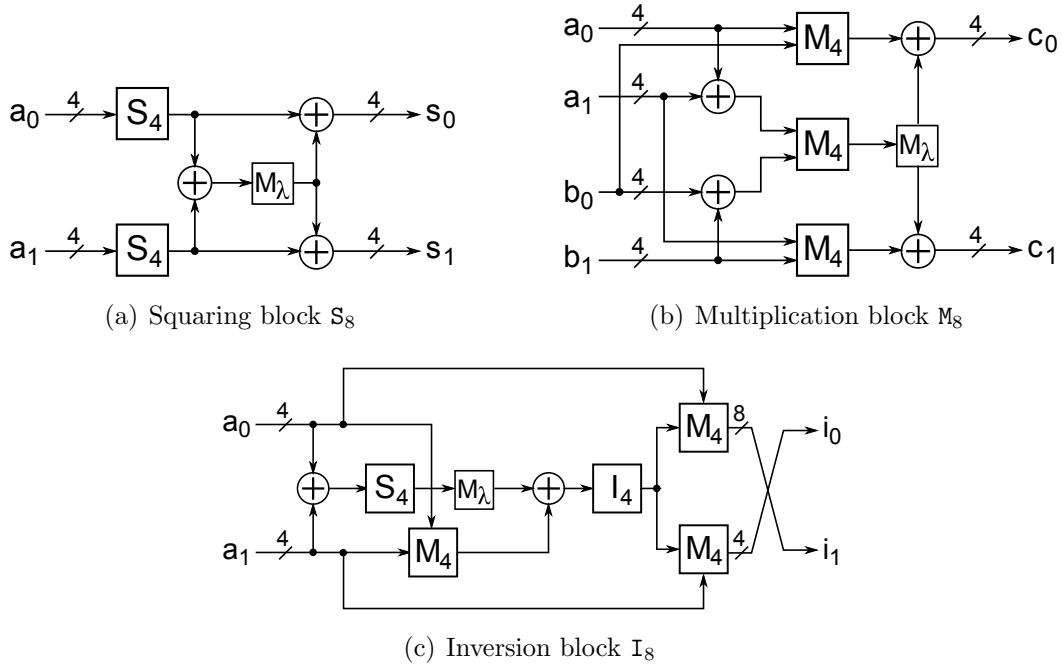


Figure 4.17: Multiplication, squaring, and inversion in $\mathbb{F}_{((2^2)^2)^2}$

Auxiliary computations in $\mathbb{F}_{((2^2)^2)^2}$: The multiplication of $A \in \mathbb{F}_{((2^2)^2)^2}$ with constant $\mu = \beta + \lambda\gamma \in \mathbb{F}_{((2^2)^2)^2}$ is carried out as follows:

$$\begin{aligned}
 \mu A &= (\beta + \lambda\gamma)(a_0\gamma + a_1\gamma^{16}) \\
 &= a_0\beta\gamma + a_1\beta\gamma^{16} + a_0\lambda\gamma^2 + a_1\lambda\gamma^{17} \\
 &= [a_0(\alpha\beta) + (a_0 + a_1)\lambda^2]\gamma + \\
 &\quad [a_1\beta + (a_0 + a_1)\lambda^2]\gamma^{16}
 \end{aligned}$$

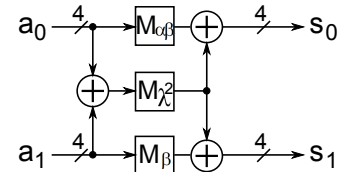


Figure 4.18: Block M_μ

Arithmetic operations in $\mathbb{F}_{((2^2)^2)^2}$

Squaring: For a non-zero element $A \in \mathbb{F}_{((2^2)^2)^2}$, the square of A is calculated as follows (see Figure 4.19(a)):

$$\begin{aligned}
 A^2 &= (a_0\delta + a_1\delta^{256})^2 = a_0^2\delta^2 + a_1^2\delta^{512} \\
 &= a_0^2[(\mu + 1)\delta + \mu\delta^{256}] + a_1^2[\mu\delta + (\mu + 1)\delta^{256}] \\
 &= [(a_0^2 + a_1^2)\mu + a_0^2]\delta + [(a_0^2 + a_1^2)\mu + a_1^2]\delta^{256} \\
 &= s_0\delta + s_1\delta^{256} = S.
 \end{aligned}$$

Multiplication: Let $A = a_0\delta + a_1\delta^{256}$ and $B = b_0\delta + b_1\delta^{256}$, where $a_0, a_1, b_0, b_1 \in \mathbb{F}_{((2^2)^2)^2}$. A multiplication $C = AB$ in $\mathbb{F}_{((2^2)^2)^2}$ is computed as follows (see Figure 4.19(b)):

$$\begin{aligned}
 AB &= (a_0\delta + a_1\delta^{256})(b_0\delta + b_1\delta^{256}) \\
 &= a_0b_0\delta^2 + (a_0b_1 + a_1b_0)\delta^{257} + a_1b_1\delta^{512} \\
 &= [(a_0 + a_1)(b_0 + b_1)\mu + a_0b_0]\delta + \\
 &\quad [(a_0 + a_1)(b_0 + b_1)\mu + a_1b_1]\delta^{256} \\
 &= c_0\delta + c_1\delta^{256} = C.
 \end{aligned}$$

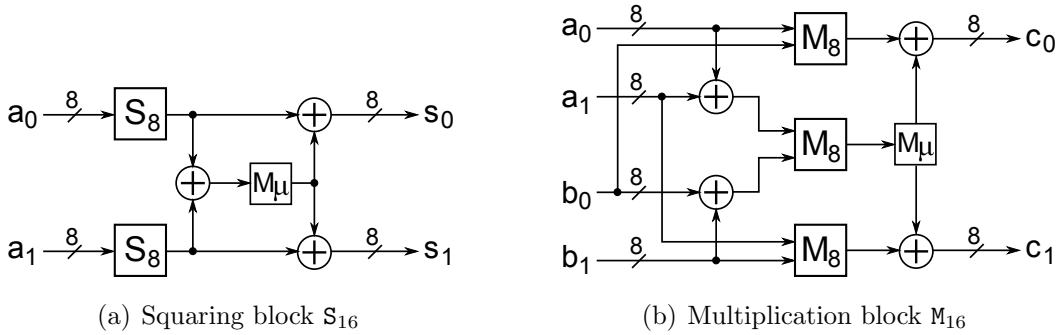


Figure 4.19: Squaring and multiplication in $\mathbb{F}_{((2^2)^2)^2}$

Inversion: Similar to the lower levels of the tower, we begin by writing the Frobenius mapping of element $A = a_0\delta + a_1\delta^{256}$ with respect to \mathbb{F}_{2^8} . The expression can be seen on the right, and once again, it dictates a simple rewiring of the inputs.

$$\begin{aligned}
 A^{256} &= (a_0\delta + a_1\delta^{256})^{256} \\
 &= a_0\delta^{256} + a_1\delta^{65536} \\
 &= a_1\delta + a_0\delta^{256}
 \end{aligned}$$

Letting A be a non-zero element in $\mathbb{F}_{((2^2)^2)^2}$, the inverse I of A can be calculated using

(4.9) as follows (see Figure 4.20):

$$\begin{aligned}
 A^{-1} &= D^{-1} \cdot A^{256} \\
 &= D^{-1}(a_1\delta + a_0\delta^{256}) \\
 &= (a_1D^{-1}\delta + a_0D^{-1}\delta^{256}) \\
 &= i_0\delta + i_1\delta^{256} = I,
 \end{aligned}$$

where $D^{-1} = (A^{257})^{-1}$ can be computed with subfield $\mathbb{F}_{((2^2)^2)^2}$ inversion block I_8 , and the value D by:

$$\begin{aligned}
 D = A^{257} &= AA^{256} = (a_0\delta + a_1\delta^{256})(a_1\delta + a_0\delta^{256}) \\
 &= a_0a_1\delta^2 + a_0^2\delta^{257} + a_1^2\delta^{257} + a_0a_1\delta^{512} \\
 &= a_0a_1(\delta^2 + \delta^{512}) + (a_0 + a_1)^2\mu \\
 &= a_0a_1 + (a_0 + a_1)^2\mu
 \end{aligned}$$

In the expression above the relationships $\delta^2 + \delta^{512} = 1$ and $\delta^{257} = \mu$ were used.

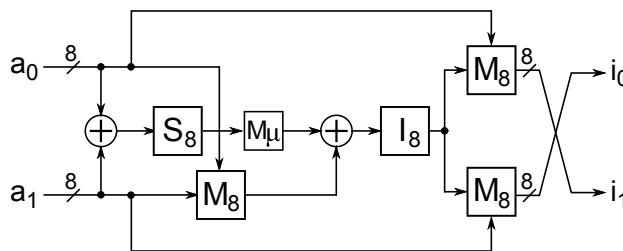


Figure 4.20: Inversion block I_{16} in $\mathbb{F}_{((2^2)^2)^2}$

Summary of Basic Building Blocks in $\mathbb{F}_{((2^2)^2)^2}$

In this section we wish to point out the regularity of this tower construction by summarizing the basic building blocks for $\mathbb{F}_{((2^2)^2)^2}$ arithmetic: using the common notation introduced in Section 3.4.3, we can depict the arithmetic blocks for each level of the tower with a common schematic, as seen in Figure 4.4.1. The inverter I_2 is of course an exception.

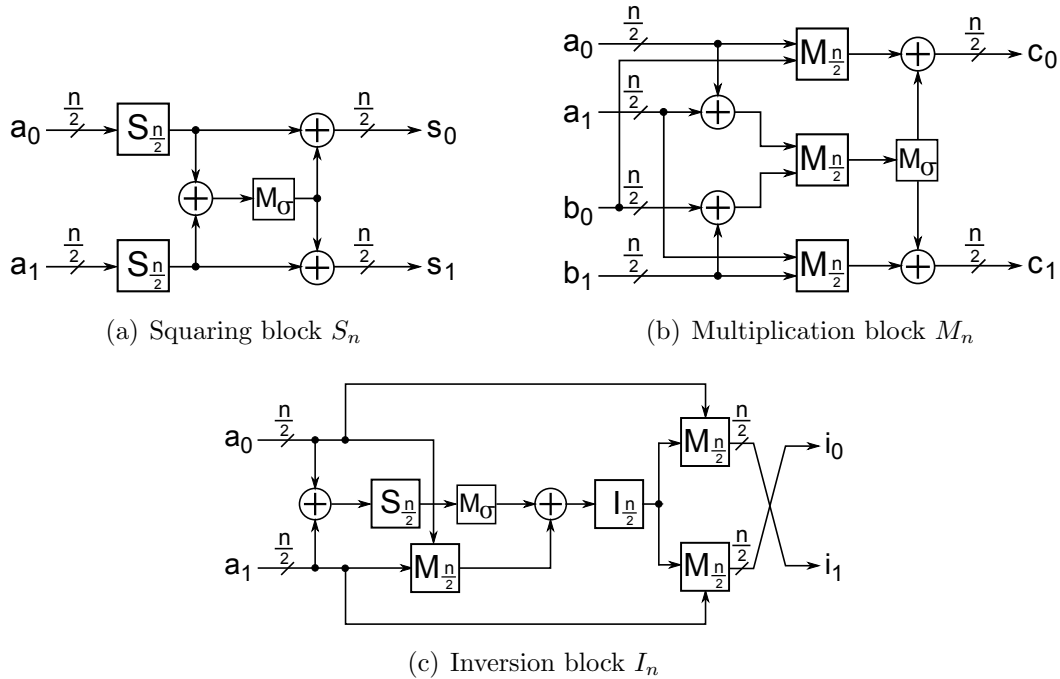


Figure 4.21: Multiplication, squaring, and inversion with: $M_\sigma = M_\alpha$ for $n = 4$ in $\mathbb{F}_{(2^2)^2}$, $M_\sigma = M_\lambda$ for $n = 8$ in $\mathbb{F}_{((2^2)^2)^2}$ and $M_\sigma = M_\mu$ for $n = 16$ in $\mathbb{F}_{(((2^2)^2)^2)^2}$

Exponentiation to powers of 2

The most efficient approach is computing the value $A^{2^k} \in \mathbb{F}_{2^m}$ in the normal basis representation, where the actual exponentiation can be done with a simple right cyclic shift by k positions, as was explained in Section 3.3.2.

A shift can be implemented by simply rewiring the outputs and inputs. But since all other computations are performed on elements in tower field representation, a transition from tower field representation to normal basis representation is required. Efficient conversion matrices were given in Section 3.4.2. Figure 4.22 shows the entire datapath:

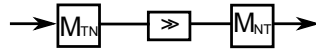


Figure 4.22: Basis transition and exponentiation

Submodules M_{NT} and M_{TN} perform the multiplication of a vector (a finite field element)

with the transition matrix.

Implementation results for the conversion modules M_{NT} and M_{TN} are given in Table 4.9. Results for the entire datapath $S_1: M_{TN} \rightarrow \gg 1 \rightarrow M_{NT}$, as is depicted in Figure 4.22, are given in the last two columns of Table 4.9: S_1 as purely combinational circuit, and since we are aiming at a pipelined design, S_{1p} connected between two registers.

	M_{NT}	M_{TN}	S_1	S_{1p}
FFs	/	/	/	32
LUTs	17	18	25	29
Slices	7	7	11	12
T	7.788	8.195	8.653	2.825

Table 4.9: Basis transition and exponentiation in $\mathbb{F}_{((2^2)^2)^2}$ - implementation results

Implementation results for basic building blocks

■ *Remark: Gate count*

We summarize area and time complexities of the building blocks in tower construction with normal bases in Table 4.10, where N_X (resp. T_X) and N_A (resp. T_A) denote the number (resp. the delay) of XOR gates and AND gates, respectively.

The tower construction described in this section allows a hardware architecture with a highly regular structure, having almost identical basic building blocks for each layer, as can be observed in Figure 4.4.1. This high level of regularity allows accurate prediction of area complexities for basic building blocks on higher level of the tower field, based on results obtained in the base field. If we refer to Table 4.10 and compare area complexities for multipliers M_2 and M_4 , we can observe that M_4 will contain three M_2 blocks (so 12 XOR gates and 2 AND gates), a M_α block (one XOR gate) and four 2-bit XOR gates, adding up to a total of 21 XOR gates and 9 AND gates in multiplier M_4 . ■

The basic building blocks for performing the tower field arithmetic have been implemented as combinational circuits on the target FPGA and ASIC platforms; the implementation results are summarized in Table 4.11.

The FPGA device (i.e., Spartan-6 XC6SLLX9) used in our implementation features 6-input and 2-output LUTs, which can implement any 6-input Boolean functions. For example, recall that the two output bits c_0 and c_1 of M_2 , namely

$$\begin{aligned} c_0 &= (a_0 + a_1)(b_0 + b_1) + a_0b_0 \\ c_1 &= (a_0 + a_1)(b_0 + b_1) + a_1b_1, \end{aligned}$$

Tower Field	Building Block	N_X	N_A	Critical Path Delay
\mathbb{F}_{2^2}	Squaring (S_2)	0	0	0
	Multiplication (M_2)	4	3	$2T_X + T_A$
$\mathbb{F}_{(2^2)^2}$	Squaring (S_4)	7	0	$3T_X$
	Multiplication (M_4)	21	9	$5T_X + T_A$
	Inversion (I_4)	17	9	$5T_X + 2T_A$
$\mathbb{F}_{((2^2)^2)^2}$	Squaring (S_8)	31	0	$7T_X$
	Multiplication (M_8)	84	27	$9T_X + T_A$
	Inversion (I_8)	100	36	$17T_X + 3T_A$
$\mathbb{F}_{(((2^2)^2)^2)^2}$	Squaring (S_{16})	114	0	$13T_X$
	Multiplication (M_{16})	312	81	$15T_X + T_A$
	Inversion (I_{16})	427	117	$39T_X + 4T_A$
	Conversion M_{NT}	76	0	$3T_X$
	Conversion M_{TN}	84	0	$4T_X$

Table 4.10: Gate count: area and time complexities of building blocks in $\mathbb{F}_{(((2^2)^2)^2)^2}$

Basic Building Block	FPGA Results			ASIC Results	
	# of LUTs	# of Slices	t [ns]	Area [GE]	t [ns]
S_2 / I_2	0	0	5.512	0.0	0.00
M_2	1	1	6.669	22.9	0.17
S_4	2	2	6.984	25.0	0.28
M_4	11	5	8.517	10.3	0.57
I_4	2	2	6.984	75.4	0.56
S_8	6	3	7.118	116	0.73
M_8	40	14	10.613	401	1.13
I_8	41	15	12.915	400	2.13
S_{16}	24	10	8.322	442	1.49
M_{16}	148	52	13.925	1440	2.09
I_{16}	147	60	22.826	1684	5.02
M_{NT}	17	7	7.800	219	0.33
M_{TN}	18	8	7.963	210	0.36

Table 4.11: Basic building blocks for arithmetic in tower field $\mathbb{F}_{(((2^2)^2)^2)^2}$ - implementation results

are 4-input Boolean functions, computed on the same values of inputs a_0, a_1, b_0 and b_1 . Hence, the M_2 multiplication can be realized on one LUT, using both outputs. In M_4 block, we expect to find four LUTs connected to the four output bits (the product) and the three LUTs for three M_2 blocks, which gives the minimum of 7 LUTs. The remaining LUTs are inferred to implement the XOR gates at the inputs. Similarly, going to the M_8 level, we expect 8 LUTs on the outputs, together with the 33 LUTs for the three M_4 multipliers. Note that the lower level blocks M_4 's are not integrated into M_8 directly. Instead they are broken down and their signals are rerouted without altering the functionality. Moreover, parts of M_4 blocks are combined with the last XORs, merged with computations from M_λ block, and realized in the LUTs connected directly to the M_8 outputs. Note that time and area complexities strongly depend on placement and routing of on the actual FPGA device and that it is difficult to predict which optimization will be performed automatically by `Xilinx-ISE`. Nevertheless, an approximate area complexity estimation still can be made and we can expect to find at least 16 output LUTs and 120 LUTs for M_8 multiplications in M_{16} block.

4.4.2 Initial Design of Pipelined Architecture

In this section we are further exploring our first tower construction by pipelining the `WGP_T` module at different levels of the tower. The implementation results of the basic building blocks in Table 4.11 provide some insight about how the complexities change with each level of the tower. To determine appropriate pipeline stages for the circuit in Figure 3.10 we begin with a critical path analysis for different paths through the circuit pipelined at different levels (Section 4.4.2). We then continue with the first decomposition of the `WGP_T` module into submodules (Section 4.4.2) and then analyze these submodules individually (in Sections 4.4.2, 4.4.2 and 4.4.2). In Section 4.4.2 we present the first `WGP_T` module using the tower construction $\mathbb{F}_{((2^2)^2)^2}$, obtained by connecting the aforementioned submodules.

Critical path analysis - pipelining granularity

The first, and most natural option is using basic building blocks from the top level of the tower $\mathbb{F}_{((2^2)^2)^2}$, i.e. the blocks S_{16} , M_{16} and I_{16} ; this is the level of pipelining that was done for the previous modules `WGP_T` described in Sections 4.2 and 4.3. Another option is going lower to the level $\mathbb{F}_{(2^2)^2}$ and pipelining at a finer granularity with building blocks S_8 , M_8 and I_8 or at lower S_4 , M_4 and I_4 . To analyze the behavior of multipliers the datapath $X \rightarrow X^d$ (the decimation on the left corner of Figure 3.10) was chosen and implemented in four versions:

- no pipeline (module `path1`)
- pipelined at M_{16} level - 3 pipeline stages (Figure 4.23(a) - module `path1_M16`)
- pipelined at M_8 level - 5 pipeline stages (module `path1_M8`)
- pipelined at M_4 level - 7 pipeline stages (module `path1_M4`)

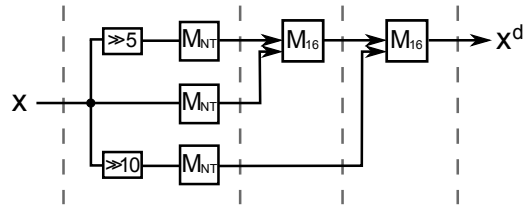
The module without a pipeline was implemented as a reference for the pipelined versions. Note that the initial exponentiations X^{2^5} and $X^{2^{10}}$, together with basis conversion, are carried out in the first stage of the pipeline in all pipelined modules. In the second design option (i.e., pipelining at M_{16} level), the two multiplications XX^{2^5} and $X^{2^{10}}X^{2^5+1}$ are computed atomically, with two M_{16} modules placed in two consecutive pipeline stages (Figure 4.23(a)). The third design option, referred to as pipelining at M_8 level, is simply implementing multipliers M_{16} with inter-stage registers inserted between three parallel M_8 modules and the M_μ module. With the pipelining at an even lower level, we pipeline the M_8 multipliers by inserting another stage border between the parallel M_4 modules and the M_λ modules. The positions of the new stage borders splitting the M_{16} and M_8 can be seen in Figure 4.23(b); the accurate schematics of M_{16} and M_8 were shown in Figures 4.19(b) and 4.17(b) in Section 4.4.1. The five stages of module `path1_M8` with the two M_{16} multipliers pipelined at M_8 level can be seen in the first half of the circuit in Figure 4.24.

Figure 4.23(c) shows the multiplier M_{16} pipelined at the M_4 level into three pipeline stages. In the first stage we can see the 9 M_4 modules in parallel, followed by three M_λ modules in the second stage; they belong to the three parallel M_8 multipliers (shaded grey) that are now segregated. The M_{16} multiplication is concluded in the third stage, that contained the module M_μ .

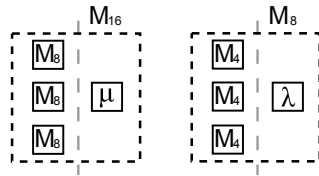
The implementation results of the four modules are given in Table 4.12. We can clearly see how pipelining at a lower granularity reduces the clock period. The difference in number of LUTs when comparing module `path1_M16` to other three modules seems surprising, but a closer examination reveals that the synthesis tools are responsible for this difference: all other three modules have a high number of LUTs where both LUT outputs were used.

Examining implementation results of the basic building blocks in Table 4.11, we can see that even though multiplier M_{16} and inverter I_{16} are very similar in terms of area, there is a very long block delay for the inverter I_{16} ; it is obvious that using the inversion module I_{16} inside a pipeline stage is not the best option. We identify the critical path $X \rightarrow Y^{-1}$ (decimation followed by inversion) and explore the following pipelining options:

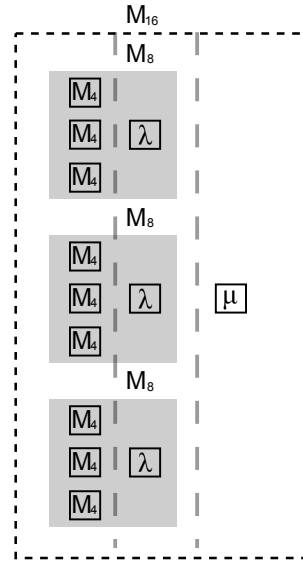
- pipelined at M_{16}/I_{16} level - 4 pipeline stages (module `path2_M16_I16`)
- pipelined at M_{16}/I_8 level - 7 pipeline stages (module `path2_M16_I8`)



(a) Module path1_M16- path $X \rightarrow X^d$ - pipelined at M_{16} level



(b) Module M_{16} - pipelined at M_8 level and Module M_8 - pipelined at M_4 level



(c) Module M_{16} - pipelined at M_4 level

Figure 4.23: Path $X \rightarrow X^d$ and different levels of pipelining

Module	FPGA Results			
	#FFs	#LUTs	#Slices	t [ns]
path1	-	398	144	23.796
path1_M16	112	510	167	6.399
path1_M8	76	430	145	5.072
path1_M4	280	398	140	4.195

Table 4.12: Path $X \rightarrow X^d$ - implementation results

- pipelined at M_8 / I_8 level - 9 pipeline stages (module `path2_M8_I8`, see Figure 4.24(c))
- pipelined at M_4 / I_4 level - 16 pipeline stages (module `path2_M4_I4`)

The inverter I_{16} pipelined at M_8 / I_8 level has been implemented in four pipeline stages:

1. the initial multiplication module M_8 and squaring module S_8 in parallel
2. the M_μ module
3. the inversion I_8 module
4. the last two multiplication modules M_8 in parallel

These pipeline stages can be seen in (Figure 4.24) - the part of the circuit that belongs to the inverter I_{16} is shaded grey. The five pipeline stages on the left of the inverter are the module `path1_M8`. To pipeline I_{16} at level M_4 / I_4 , we further split the stages above and obtain an inverter that stretches over 9 pipeline stages:

1. the initial multiplication module M_8 and squaring module S_8 in parallel:
 - 1.1. the 3 M_4 and 2 S_4 in parallel
 - 1.2. the two M_λ modules
2. the M_μ module
3. the inversion I_8 module:
 - 3.1. the M_4 and the S_4 in parallel
 - 3.2. the two M_λ module
 - 3.3. the I_4 module
 - 3.4. the two M_4 modules
4. the last two multiplication modules M_8 in parallel:
 - 4.1. the 6 M_4 modules in parallel
 - 4.2. the two M_λ modules

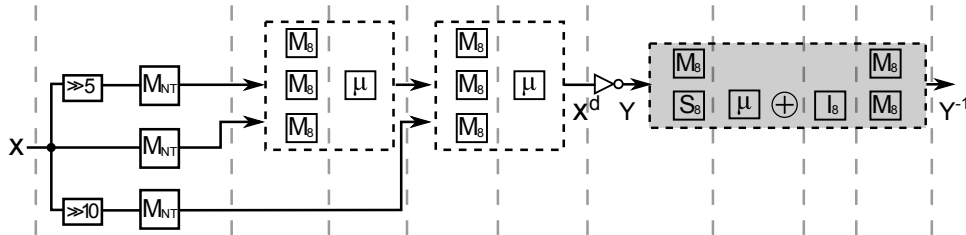


Figure 4.24: Module `path2_M8_I8` - path $X \rightarrow Y^{-1}$ pipelined at M_8 / I_8 level

The implementation results of module `path2_M16_I16` confirm the prediction that we should avoid block I_{16} inside a pipeline stage. Area complexity, apart from the number of registers, of the first three modules is very similar. There is a jump in number of slices used for the module `path2_M4_I4`, which is a result of a high number of registers needed, but the clock period is clearly in favor of this module. However, due to the big area consumption, the combined metric $\frac{T}{A^2}$ listed in the last column of Table 4.13 favors the module `path2_M8_I8`.

Module	FPGA Results				$\frac{T}{A^2}$
	#FFs	#LUTs	#Slices	t [ns]	
path2_M16_I16	128	575	167	14.214	2.5
path2_M16_I8	208	602	189	7.328	3.8
path2_M8_I8	272	544	168	5.930	5.9
path2_M4_I4	546	575	226	4.328	4.5

Table 4.13: Path $X \rightarrow Y^{-1}$ - implementation results

Module WGP_T - First decomposition into submodules

For creating a pipelined design, the integrated hardware architecture WGP_T in Figure 3.10 from Section 3.4.1 has been decomposed into three submodules `moduleA`, `moduleB` and `moduleC`, as shown in Figure 4.25. On the left, marked with a dashed line and shaded dark grey, we see the `moduleA` containing the common computational components that are shared by the initialization and running phase, and outputting the values $Y^{2^6}, Y^{2^{11}}, Y^{-1}$ and X^d . The two multipliers that are reused during the initialization phase, together with registers and multiplexers that aid this computation, are implemented in `moduleB` (in the middle of Figure 4.25, marked with a solid line, shaded light grey). This module outputs the values $Y \oplus_{16} Y^{2^{11}+1}, Y^{2^6}, Y^{2^{11}+1} \oplus_{16} Y^{2^{11}-1}$ and $Y^{2(2^{11}-1)}$ needed for the trace computation in running phase, and the value WGP for feedback to the LSFR in the initialization phase. The module `moduleC` on the right in 4.25, marked with a dotted line and shaded dark gray, conducts the final trace computations.

Figure 4.26 shows the decomposition described above from a higher level, with the three submodules as black boxes, and clearly visible signals that pass from one submodule to another.

Module `moduleA`

The first submodule `moduleA` is basically the path $X \rightarrow Y^{-1}$, i.e. `path2` from the previous Section, together with exponentiations Y^{2^6} and $Y^{2^{11}}$. Module `moduleA_M16_I8`, that can be seen in Figure 4.27 below, is actually module `path2_M16_I8` with two exponentiation circuits added to pipeline stages 4 and 6. Similarly, the two exponentiation circuits were added to pipeline stages 6 and 8 of module `path2_M8_I8` to obtain `moduleA_M8_I8`, and to pipeline stages 15 and 16 of module `path2_M4_I4` to get `moduleA_M4_I4`.

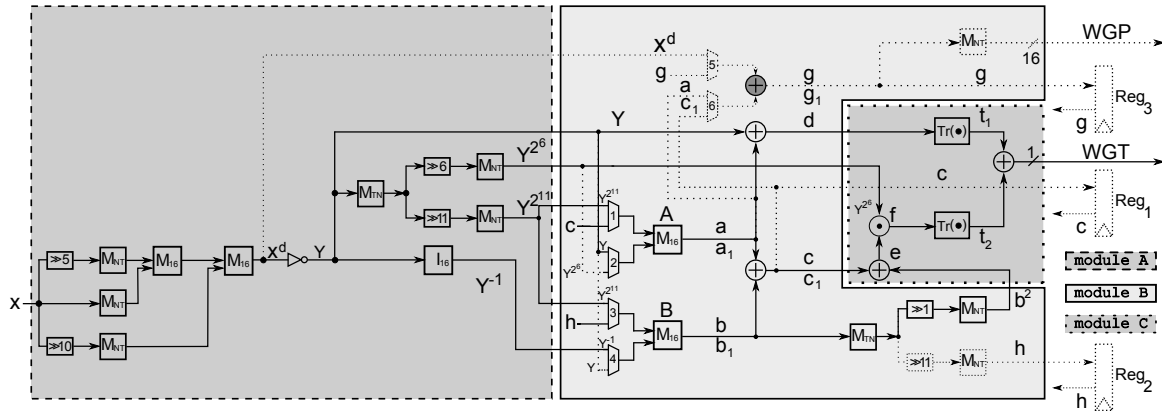


Figure 4.25: First decomposition of the WGP_T circuit into submodules moduleA, moduleB and moduleC

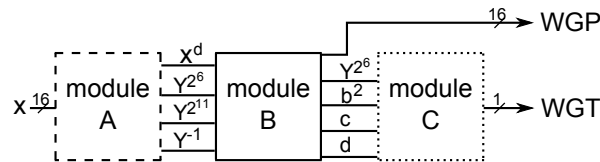


Figure 4.26: Modular view of submodules moduleA, moduleB and moduleC and connecting signals

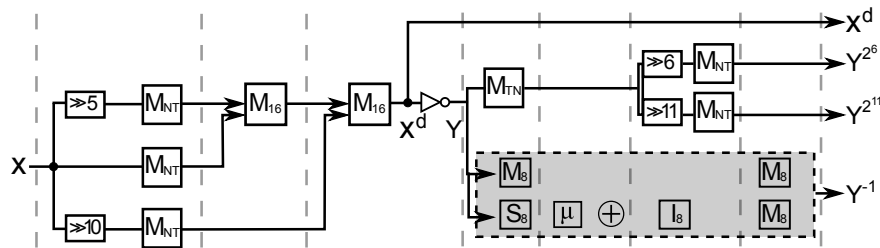


Figure 4.27: Module moduleA - pipelined at M_{16} / I_8 level

It is desirable to have the ability to pause the keystream; a chip enable signal, acting as a stop and go control, is needed for that. We add a control signal **ce** to the pipeline of the WGP_T module, to control the operation of the registers: if **ce** is set the module operates normally and updates the registers at the end of each clock cycle, if **ce** is cleared, the registers keep their old value - the pipeline is stopped. A select signal **sel** will be needed to control the operation of multiplexers in module moduleB; for the operation of **sel** please see Table 3.11 in Section 3.4.3. The two control signals **ce** and **sel** will propagate through the pipeline together with the input from the LFSR; we need to add two 1-bit registers at

each stage border.

Module	FPGA Results				$\frac{T}{A^2}$
	#FFs	#LUTs	#Slices	t [ns]	
moduleA_M16_I8	382	659	235	7.154	2.5
moduleA_M8_I8	468	599	225	6.000	3.3
moduleA_M4_I4	626	633	277	4.172	3.1

Table 4.14: Module `moduleA` - implementation results

All three modules listed in Table 4.14 show an expected increase in area but approximately the same clock period; in the case of `moduleA_M16_I8` and `moduleA_M4_I4` the period is even a little bit shorter. The difference between the three modules is in terms of $\frac{T}{A^2}$ now smaller, but still in favor of the `moduleA_M16_I8`.

Module `moduleB`

During initialization phase, `moduleB` computes the value $WGP-16(X^d)$, and during running phase prepares the values needed for the final trace computation in `moduleC`, described in the next Section 4.4.2. For the $WGP-16(X^d)$ computation, the two multipliers belonging to `moduleB` (we denoted them multiplier A and B) can be reused as explained in Section 3.4.3. The module `moduleB` was pipelined into two stages, B1 performing the multiplications atomically (using modules M_{16}), and B2 performing conversions between the bases and exponentiation. The pipeline stage border is shown in Figure 4.28 with a dashed vertical line. The inputs (X^d , Y^{2^6} , $Y^{2^{11}}$ and Y^{-1}) and outputs ($d = Y \oplus_{16} Y^{2^{11}+1}$, Y^{2^6} , $c = Y^{2^{11}+1} \oplus_{16} Y^{2^{11}-1}$ and $b^2 = Y^{2(2^{11}-1)}$) are omitted from the figure and they are all 16 bit wide. Circuit implementing `moduleB`, including all the registers is depicted in Figure 4.29.

Let us first discuss $WGP-16(X^d)$ computation. For the operation of `sel` please see Table 3.11 in Section 3.4.3: for example, when `sel=0`, the multiplexer MUX_1 passes the value $Y^{2^{11}}$ to the multiplier A. In pipeline stage B1, with control signal `sel=0`, the multiplier A produces the intermediate product $a = Y^{2^{11}}Y$ and multiplier B the product $b = Y^{-1}Y^{2^{11}}$ (recall that $Y = X^d \oplus_{16} 1$). The latter value b is used in two ways: (a) it is passed unchanged to interstage register `regB` and (b) it is XORed with the product from multiplier A to produce the value $c = a \oplus_{16} b$, which is passed on to interstage register `regAB`. The

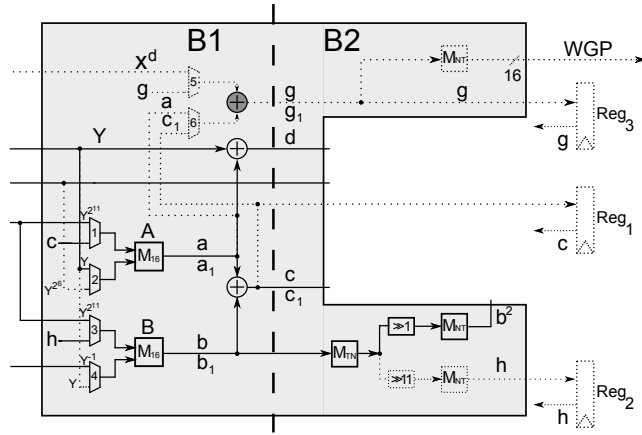


Figure 4.28: Module `moduleB` - splitting into two pipeline stages (dashed line)

product a from multiplier A is also used to compute $g = X^d \oplus_{16} a$; this value is passed to interstage register `regG`.

The value $h = b^{2^{11}}$ is obtained in the second stage B2 from the `regB` value b by transitioning to normal basis representation and raising the obtained normal basis element to the power 2^{11} , i.e. a right cyclic shift for 11 bits, followed by conversion back to tower field representation. This result is stored in register `reg2`. To keep events synchronized we pass the two values from interstage registers `regAB` and `regW` through stage B2 unchanged and store them in registers `reg1` and `reg2` respectively. This concludes the first round of $\text{WGP-16}(X^d)$ computation. In the second round of $\text{WGP-16}(X^d)$ computation we need to compute $a_1 = Y^{2^6} \otimes_{16} c$ and $b_1 = Y \otimes_{16} h$ (by reusing multipliers A and B) and XOR these two values with the previously computed g . Factors c , h and the value g are available on signals `back1`, `back2`, and `back3` respectively. The missing values Y^{2^6} and Y are basically just the `moduleB` input Y^{2^6} and the inverted input X^d . Instead of passing them through B1 and B2 and then returning them to the multipliers in B1 we choose the following approach: we simply send each input X through `moduleA` three times, with different values for the control signal `sel`, a detailed description of the FSM will follow in Section 4.4.4.

During the running phase we do not need to reuse the multipliers and the control signal `sel` remains low the entire time. Module `moduleC` needs the inputs ($d = Y \oplus_{16} Y^{2^{11}+1}$, Y^{2^6} , $c = Y^{2^{11}+1} \oplus_{16} Y^{2^{11}-1}$ and $b^2 = Y^{2(2^{11}-1)}$). Value Y^{2^6} is just passed through the module `moduleB`

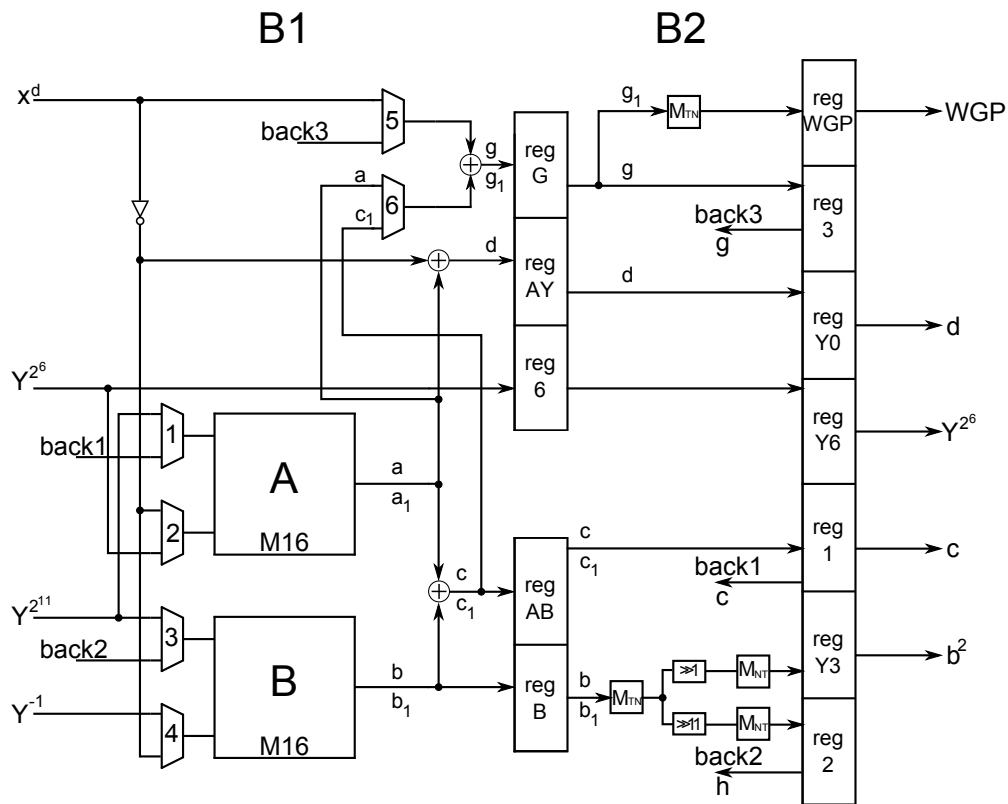


Figure 4.29: Module `moduleB`

and the value c was already discussed (it is kept in register `reg1`). Value d is obtained by XORing the output of multiplier A with the inverted input X^d . It is passed to register `regAY` and then to `regY0`. Value b^2 is obtained in stage B2 by squaring the product from multiplier B, which is kept in register `regB`; the square is passed on to register `regY3`.

Implementation results for module `moduleB`, pipelined at the M_{16} level, are listed in Table 4.15 below. Comparing `moduleB` with `moduleA_M16_I8`, also pipelined at the M_{16} level, we immediately notice the increase of clock period. This higher period is a consequence of the multiplexers at the M_{16} inputs and the additional logic (XOR gates and even multiplexers) at the M_{16} outputs. Note also that the stages B1 and B2 are not exactly balanced: stage B1 requires much more logic and routing elements. To optimize `moduleB` we will try to: (a) merge modules `moduleB` and `moduleC`, (b) pipeline `moduleB` at a lower level and (c) take the part of the stage B1 circuit and move it over the pipeline stage border into stage B2.

These measures will be discussed in Section 4.4.3.

Module	FPGA Results			
	#FFs	#LUTs	#Slices	t [ns]
moduleB	196	665	220	7.600

Table 4.15: Module `moduleB` - implementation results

Module `moduleC`

As we can see in Figure 4.25, `moduleC` finalizes the computation of trace value $\text{WGT-16}(X^d)$. In Section 3.4.3 we summarized the trace computation in equation (3.25) as follows:

$$\begin{aligned}
 e &= b^2 \oplus_{16} c & f &= Y^{2^6} \odot_{16} e \\
 t_1 &= \text{Tr}(d) & t_2 &= \text{Tr}(f) \\
 \text{WGT-16}(X^d) &= t_1 \oplus_1 t_2
 \end{aligned}$$

From Corollary 1 in Section 3.4.3 we see that the actual computation of the two trace values can be carried out by XORing the bits of their arguments.

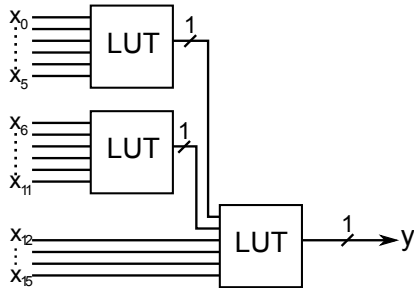


Figure 4.30: XORing the 16 bits for the trace computation

Since Spartan-6 LUTs have 6 available inputs, the value y can be obtained with 3 LUTs XORing their inputs as shown in Figure 4.30 on the left. Two versions of this module, `moduleC1` and `moduleC2`, shown in Figures 4.31(a) and 4.31(b) respectively, with different placing of the interstage registers, were explored, to reduce latency as much as possible. Their results are listed in Table 4.16. Both modules are insignificant in comparison with `moduleB`.

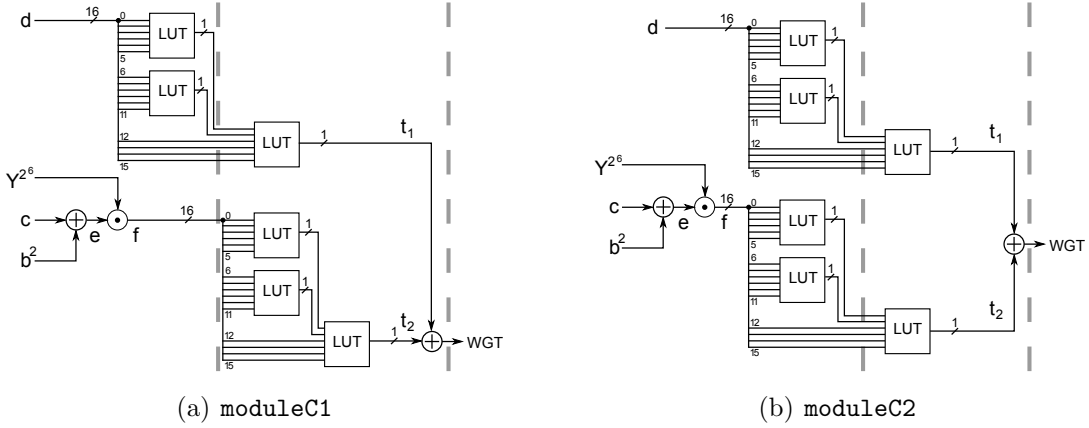


Figure 4.31: Module `moduleC` with two different interstage register placings

Module	FPGA Results			
	#FFs	#LUTs	#Slices	t [ns]
<code>moduleC1</code>	34	23	13	1.782
<code>moduleC2</code>	17	17	8	2.290

Table 4.16: Module `moduleC` in two versions - implementation results

Module `WGP_T`

This is the first implementation of `WGP_T` submodule and directly follows the image 4.26; it is a simple concatenation of modules A, B and C. Two versions were implemented, using two differently pipelined modules `moduleA`, namely the pipelining at the level M_{16}/I_8 and at the level M_8/I_8 . Since module `moduleB` was pipelined at the M_{16} level, there is no point in using the submodule `moduleA_M4_I4` at this time, but we will revisit this option when we optimize module `moduleB`. The two implemented `WGP_T` modules `WGP_T_ABC16` and `WGP_T_ABC8` differ in the number of pipeline stages:

- `WGP_T_ABC16`: `moduleA_M16_I8` \Rightarrow `moduleB` \Rightarrow `moduleC2` having a $7+2+2=11$ stage pipeline.
- `WGP_T_ABC8`: `moduleA_M8_I8` \Rightarrow `moduleB` \Rightarrow `moduleC2`, having a $9+2+2=13$ stage pipeline;

The implementation results given in Table 4.17 below indicate that the two modules are quite equivalent: surprisingly, the `WGP_T_ABC16` even has a slightly shorter clock period. Compared with results obtained for module `moduleA`, see Table 4.14, we see that due to the stage B1 the advantages of pipelining at M_8 level have been lost completely.

Module	FPGA Results			
	#FFs	#LUTs	#Slices	t [ns]
<code>WGP_T_ABC16</code>	605	1322	488	8.663
<code>WGP_T_ABC8</code>	671	1243	444	8.191

Table 4.17: The fist `WGP_T` implementation
`WGP_T_ABC16` - submodule `moduleA` pipelined at M_{16} level
`WGP_T_ABC8` - submodule A pipelined at M_8 level

4.4.3 Optimizations and final choice for module `WGP_T`

In this section we discuss some modifications of the original `moduleA`, `moduleB` and `moduleC` described in previous Section. For the first part of the circuit, `moduleA`, we identify and remove redundant registers. Three possible optimization approaches were mentioned at the end of Section 4.4.2, namely (a) merge modules `moduleB` and `moduleC`, (b) pipeline `moduleB` at a lower level and (c) take the part of the stage B1 circuit and move it over the pipeline stage border into stage B2. They were realized as follows: in Section 4.4.3 we describe the option (a). Then we try to pipeline the obtained merged module at the M_8 level and it turns out that we have to rearrange the two pipeline stages to do so; the resulting module `moduleBC8` combining the three approaches (a), (b) and (c) is described in Section 4.4.3. Both optimized modules are still not ready to be used with the module `moduleA_M4_I4`, hence we explore another possibility: we pipeline the merged module `moduleBC` at the M_4 level; the obtained module `moduleBC4` is described in Section 4.4.3. Finally in Section 4.4.3 we give the implementation results for the `WGP_T` modules using these optimizations.

Module `moduleA` - reducing the number of registers

In Figure 4.32 below we show the second half (after the decimation computation) of `moduleA_M16_I8` from Figure 4.27, equipped with register names below the stage borders:

the names of the 16-bit registers are shown in black and the names of the 8-bit registers in grey. The arrows between them indicate how the values propagate between the registers: a solid arrow between two registers means that the value was simply passed through the pipeline stage unchanged, while the dashed line indicates that in this stage the value was somehow changed. For clarity we show the `moduleA_M16_I8` pipeline stage numbers at the top of the Figure 4.32. Taking a closer look at the registers at the border between stage **3** and stage **4** we see two 16-bit and two 8-bit registers that are basically all holding the same value - the decimated input, its inverse, and the two halves of the inverse:

reg30	reg35	reg31	reg32
X^d	$Y = \text{not}(X^d)$	Y_{HI}	Y_{LO}

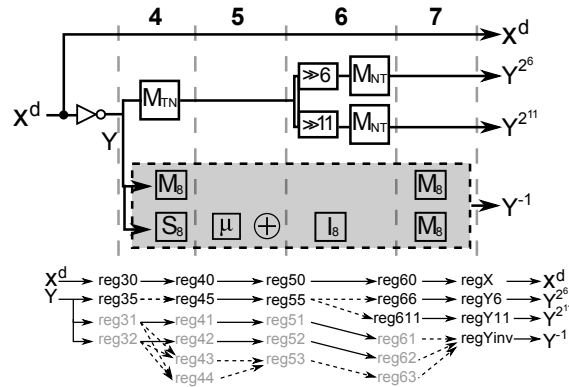


Figure 4.32: Module `moduleA` - pipelined at M_{16}/I_8 level

In stage 4 of the pipeline the inverse computation begins using the values in 8-bit registers `reg31` and `reg32`. The same two values are also needed for the final two multiplications inside inversion, so they will be propagated unchanged through the next three pipeline stages. We can completely remove these half-registers by letting 16-bit register `reg30` hold the value $Y = \text{not}(X^d)$ and in stage **4** routing its contents to (a) transition submodule M_{TN} and (b) to M_8 and S_8 submodules. For these two submodules we need to split the signal into two halves, $Y_{HI} = Y_{8...15}$ and $Y_{LO} = Y_{0...7}$. In the end we invert the value of `reg30` and output (X^d). This way, we eliminated 32 FFs (`reg35`, `reg31` and `reg32`) at the end of decimation computation, but now also the registers `reg41`, `reg42`, `reg51`, `reg52`, `reg61` and `reg62` become obsolete, so we save 80 FFs altogether. Implementation results for the `moduleA_M16_I8_2` with changes described above are given in Table 4.18, together with the results of the original `moduleA` for comparison. We can see the area reduction as well as a

shorter clock period. The numbers differ from the estimated 80 FFs, because Xilinx-ISE automatically removed registers `reg31`, `reg32` and connected values from `reg35` instead, thus saving two 8-bit registers in the original implementation `moduleA_M16_I8`, not the really the best effort. We provide the ASIC results as well: due to more flexible routing, we can only observe an area reduction.

Module	FPGA Results				ASIC Results	
	# of FFs	# of LUTs	# of Slices	t [ns]	Area [GE]	t [ns]
<code>moduleA_M16_I8</code>	382	659	235	7.154	5956	1.59
<code>moduleA_M16_I8_2</code>	318	659	227	6.738	5417	1.59
<code>moduleA_M8_I8</code>	468	599	225	6.000	-	-
<code>moduleA_M8_I8_2</code>	404	604	211	6.261	-	-

Table 4.18: Optimized `moduleA` - implementation results

Module `moduleBC` - merging `moduleB` and `moduleC`

The two stage pipeline for `moduleB` can be seen in Figure 4.29 in Section 4.4.2. The two pipeline stages B1 and B2 are unbalanced, so we integrate the module `moduleC` into stage B2 while leaving the stage B1 unchanged. Resulting circuit is shown in Figure 4.33.

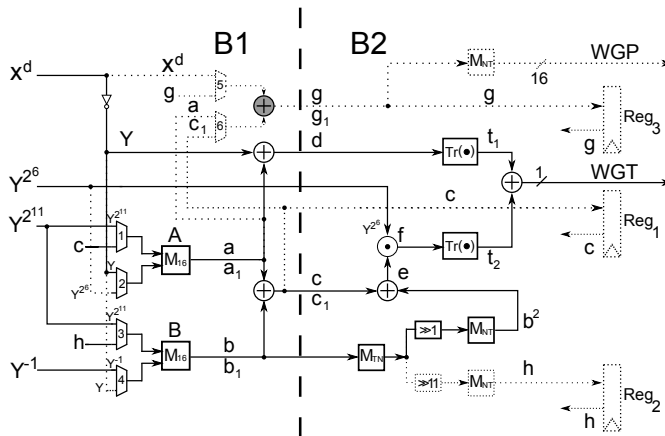


Figure 4.33: Module `moduleBC` - merging `moduleB` and `moduleC`

This implementation has two advantages: it saves 2 pipeline stages and the values $WGP-16(X^d)$ and $WGT-16(X^d)$ are both available at the end of B2. Implementation results for `moduleBC`

are given in Table 4.19, together with implementation results of `moduleBC8` and `moduleBC4`, and (repeated) `moduleB` results. Comparing the new module `moduleBC` to `moduleB` we find the two modules practically equivalent, with decreased number of FFs but a slight increase in the number of slices. We did save 2 pipeline stages and are now able to output $WGP-16(X^d)$ and $WGT-16(X^d)$ at the end of the pipeline.

Module `moduleBC8` - pipelining at the M_8 level

As mentioned earlier, the effects of pipelining at the M_8 level are nullified by the use of atomic M_{16} multipliers in the `moduleBC`. Hence, we decide for a finer granularity and instead of using two parallel M_{16} multipliers in the first stage, we break them up and move the pipeline stage between M_8 and M_μ modules, as can be seen in Figure 4.34. The input values for the six M_8 multipliers are still selected by the same four multiplexers and the control signal `sel`. Figure 4.34 also shows the additional logic that splits the selected values into two 8-bit parts (denoted LO and HI) and XORs the values for multipliers A2 and B2 (refer to Figure 4.21(b) in Section 4.4.1). There are six 8-bit registers to hold the results of six 8-bit multipliers M_8 (the multipliers and registers are marked A1,A2,A3 and B1,B2,B3 for reference to M_{16} multipliers A and B used in `moduleBC`). The 16-bit registers are colored grey in Figure 4.34 for better visibility.

Instead of simply inserting another pipeline stage between the M_8 and M_μ blocks and having a three stage pipeline, we merge the rest of the multiplication with the “old” `moduleBC` stage B2 - in Figure 4.34 we can see the “old” pipeline stage shown with a dashed grey vertical line. In stage B2 we see two M_μ blocks, marked μ_A and μ_B . The results of these two modules are XORed with other partial 8-bit products from stage B1 and the final 16-bit products a and b (or a_1 and b_1 in the second round) are recomposed by concatenating corresponding LO and HI halves. The two products are then used in the exact same way as in `moduleBC`. Since the computation of $WGP-16(X^d)$ always needs one of the products (inputs to multiplexer 6), we moved the multiplexers 5 and 6 over into the second pipeline stage B2. The rest of the circuit in stage B2 remains unchanged.

In fact, all we did was to move the `moduleBC` stage border between the M_8 multipliers and the M_μ blocks. Implementation results for this module are listed in Table 4.19. We can see improvement in area and time complexity compared to both `moduleB` and `moduleBC`. Note that implementation results of `moduleA` pipelined at M_8 /IB level indicate that we can expect better performance from `moduleB` as well. To achieve it, we could proceed as follows: first we insert a pipeline stage border right at the inputs of the six M_8 multipliers (to separate the overhead created by the multiplexers and the routing circuit splitting

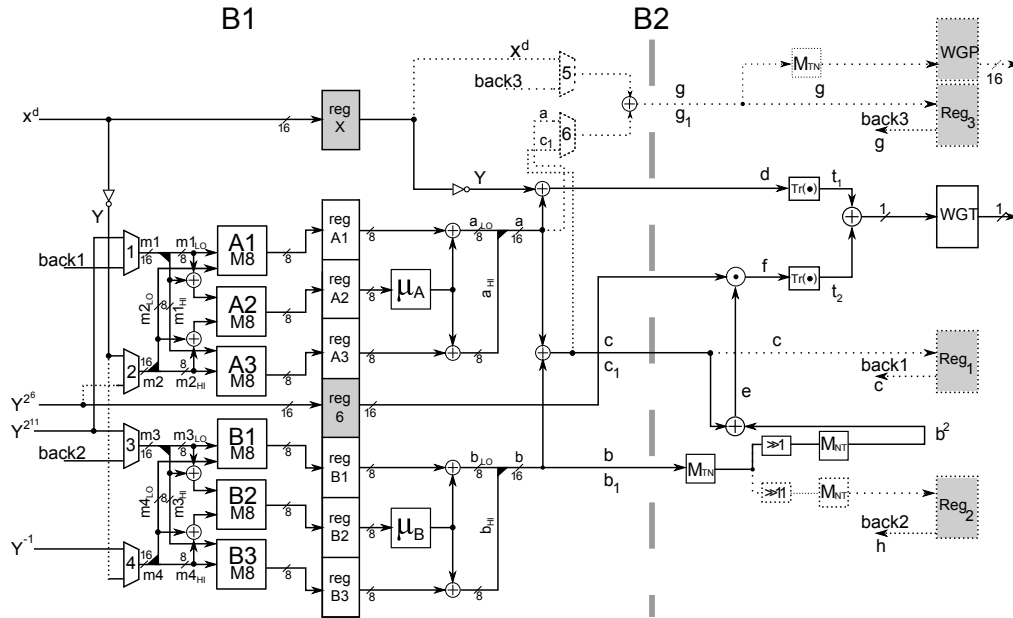


Figure 4.34: Module `moduleBC8` with two pipeline stages and with grey vertical line indicating the old pipeline stage border

the operands for the multipliers), then insert another stage border with registers storing the 16-bit products after the M_μ blocks but pushing the multiplexers 5 and 6 over this stage border into the last (that is fourth) pipeline stage. We decide to keep a two-stage `moduleBC8` module and implement another version of `moduleBC`, pipelined at the M_4 level.

Module `moduleBC4` - pipelining at the M_4 level

The `moduleBC4` module was pipelined at the M_4 level. It results in a six stage pipeline with following pipeline stages:

1. the initial multiplexers and and the routing circuit splitting the operands for the multipliers
2. the 18 parallel M_4 modules
3. the 6 parallel M_λ modules
4. the 2 parallel M_μ modules
5. the multiplexers 5 and 6 and the shift modules
6. final trace computation (`moduleC`)

The three stages for the M_{16} multiplications, that is stages **2**, **3** and **4**, contain two parallel M_{16} 's pipelined as shown in Figure 4.23(c). At the end of stage **5**, the values c , h and g

that are being reused for the $WGP-16(X^d)$ computation are ready. The value $WGP-16(X^d)$ itself is also ready at the end of stage **5** in the second round of computation. Stage **6** is actually the `moduleC`, but was implemented without the stage border: we have seen in Section 4.4.2 that `moduleC` is not a critical module (see implementation results in Table 4.16) Implementation results for the new module are listed in Table 4.19: it is the most promising `moduleBC` implementation so far.

Module	FPGA Results				$\frac{T}{A^2}$
	#FFs	#LUTs	#Slices	t [ns]	
<code>moduleB</code>	196	665	220	7.600	2.7
<code>moduleBC</code>	147	688	225	7.624	2.6
<code>moduleBC8</code>	154	575	208	7.000	3.3
<code>moduleBC4</code>	503	472	165	4.428	8.3

Table 4.19: Module `moduleBC` pipelined at different levels of the tower $\mathbb{F}_{((2^2)^2)^2}$ - implementation results

We can immediately see the big difference between the number of slices used by `moduleBC4` in comparison to all other versions of `moduleBC`. The main reason for such a big difference is the peculiarity of Xilinx-ISE; 40% of LUTs in module `moduleBC4` use both outputs, while this percentage drops to 13% for module `moduleBC8`, and below 10% for module `moduleBC`. As expected, the module `moduleBC8` showed better results than `moduleBC`. So far the $M_4 I_4$ pipelining resulted in the shortest period but the $\frac{T}{A^2}$ metric was in favor of the M_8 / I_8 pipelining with smaller area. For the first time, the $\frac{T}{A^2}$ metric is in favor of M_4 pipelining.

Module `WGP_T` with different vesions of module `A` and `moduleBC`

Now we inspect the `WGP_T` module with different versions of `moduleA` and `moduleBC`. `WGP_T` modules are named so that they reflect which submodules were used; `moduleA16_2_BC8` for example indicates that submodules `moduleA_M16_I8_2` and `moduleBC8` were used. Based on the implementation results for different variants of `moduleBC` (see Table 4.19) we choose `moduleBC8` for implementation with both `moduleA_M16_I8_2` and `moduleA_M8_I8_2`. For pipelining at the M_4 level we have only one possibility. Following modules were implemented (we include `moduleABC16` and `moduleABC8` from Section 4.4.2 as a reference point):

- `WGP_T_ABC16`: `moduleA_M16_I8` \Rightarrow `moduleB` \Rightarrow `moduleC2`
- `WGP_T_A16_2_BC8`: `moduleA_M16_I8_2` \Rightarrow `moduleBC8`

- WGP_T_ABC8: moduleA_M8_I8 \Rightarrow moduleB \Rightarrow moduleC2
- WGP_T_A8_2_BC8: moduleA_M8_I8_2 \Rightarrow moduleBC8
- WGP_T_A4_2_BC4: moduleA_M4_I4 \Rightarrow moduleBC4

Module	FPGA Results				$\frac{T}{A^2}$
	#FFs	#LUTs	#Slices	t [ns]	
WGP_T_ABC16	605	1322	488	8.663	4.9
WGP_T_A16_2_BC8	467	1262	474	6.601	6.7
WGP_T_ABC8	671	1243	444	8.191	6.2
WGP_T_A8_2_BC8	535	1195	436	6.519	8.1
WGP_T_A4_2_BC4	1129	1128	401	4.939	12.6

Table 4.20: Module WGP_T , pipelined at different levels of the tower $\mathbb{F}_{((2^2)^2)^2}$ - implementation results

Both optimized WGP_T modules WGP_T_A16_2_BC8 and WGP_T_A8_2_BC8 show significant reduction in clock period in comparison with the initial WGP_T_ABC16 and WGP_T_ABC8 respectively. The area reduction is not so significant: we saved 14 slices using the WGP_T_A16_2_BC8 and 8 slices with WGP_T_A8_2_BC8. The impact of module moduleBC4 is clearly visible: the WGP_T_A4_2_BC4 is not only the fastest but also the smallest WGP_T module.

4.4.4 The FSM

Finally, we are ready to discuss the FSM running the two submodules (LFSR and WGP_T) in detail. We have already mentioned a few things about the FSM:

- i. the cipher operates in three phases: the loading phase, initialization phase and running phase (see Figure 2.5 in Section 2.4.1);
- ii. during initialization phase the LFSR is updated 64 times (see Table 2.2 in Section 2.4.1);
- iii. LFSR is controlled by three signals: `lfsr_en`, `load` and `init` (see Table 4.1 in Section 4.1 or Table 4.22 below);

- iv. a control signal `sel` is needed to control the multiplexers in `moduleB` (see Table 3.11 in Section 3.4.3);
- v. we only allow to stop the keystream during the running phase - the control signal `ce` takes in top level input value `ce_i` only during the running phase, otherwise it is set to 1 (Section 4.1);

The loading phase

The diagram in Figure 2.5 from Section 2.4.1 shows three phases of WG-16, the loading phase, the initialization phase and the running phase. For the rest of this section, we will use the term “phase” when referring to the diagram in Figure 2.5 and the term “state” when referring to the actual implementation of the FSM. The first change to the simplified FSM in Figure 2.5 is adding the `idle` state and a `start` input, that causes the state transition to loading phase. The loading phase is straightforward: in $M = 32$ consecutive clock cycles 32 key/IV values are loaded into the LFSR serially. We need a single counter `l_count` to leave the loading phase (state `load` in Figure 4.36) after M clock cycles. Recall from Section 4.1 that the `lfsr_en` signal stays active during the entire loading phase, causing the LFSR to shift in every clock cycle, i.e. the LFSR steps coincide with the clock cycles. The values of the remaining two control signals `load` and `init` are set to values 1 and 0 respectively, to allow the input from the data input port `DIN` to be loaded into the LFSR, see Table 4.22.

The initialization phase

The most complex part of the FSM is related to the initialization phase. The integrated hardware architecture `WGP_T`, which implements both $WGP-16(X^d)$ and $WGT-16(X^d)$ computation, is basically a $P + S + T$ stage pipeline, where P denotes the number of pipeline stages in `moduleA`, S denotes the number of stages in `moduleB` and T the number of stages in `moduleC`. During the initialization the LFSR shifts only once with each computed $WGP = WGP-16(X^d)$ value, and a new `WGP` is available after $P + 2S$ clock cycles; the plus $2S$ is caused by reusing the components. Let us first describe how to simply reuse the multipliers on the example `moduleBC` (see Figure 4.33); the case when $S = 2$ (this is the case with modules `moduleB`, `moduleBC`, `moduleBC8`, the only exception is `moduleBC4`).

■ Case study - module `moduleBC`

As was just mentioned, a new `WGP` is available every $P + 4$ clock cycles. Instead of storing the values for the second round of computation if `moduleBC`, we just resend the same input value X 3 times with different control signals `sel`. Let X_i represent the value X sent in cycle i , and let the pair (X, sel) indicate the value and its corresponding select signal. In three consecutive clock cycles, we send the values $(X_1, 0)$, $(X_2, 0)$ and $(X_3, 1)$ through the pipeline, see Table 4.21. Note that X_1, X_2 and X_3 are the same finite

field element, but enter the pipeline in different clock cycles. The bubble \bigcirc in Table 4.21 belongs to the previous WGP computation and the “dont care” element $-$ belongs to the same WGP computation.

	B1	B2
$P + 1$	$(X_1, 0)$	$(\bigcirc, 0)$
$P + 2$	$(X_2, 0)$	$(X_1, 0)$
$P + 3$	$(X_3, 1)$	$(X_2, 0)$
$P + 4$	$(-, 0)$	$(X_3, 1)$

Table 4.21: Passing the same value three times

After P clock cycles, module B gets the following input values, computed from X_1 : $X^d, Y^{2^6}, Y^{2^{11}}, Y^{-1}$. We purposely omit the subscript 1 to emphasize that inputs X_1, X_2 and X_3 produce the same moduleA outputs, i.e. the same values will appear on the moduleBC inputs in three consecutive clock cycles. Actually we do not really care what happens in clock cycle $P + 2$ with parcel $(X_2, 0)$; the value could easily be another bubble propagating through the pipeline. Sending the value $(X_2, 0)$ in the second clock cycle simplifies the FSM. In clock cycle $P + 3$ (that is at the beginning of the second round of WGP-16(X^d) computation in stage B1) the value **sel=1** enables the values on signals **back1**, **back2**, and **back3** to be used in stage B1. These three signals route the values from registers **reg1**, **reg2** and **reg3** respectively. In cycle $P + 3$ computed products are XORed to the value c_1 . Multiplexers 5 and 6 now pass through the values g and c_1 , which are XORed to obtain the value g_1 . In cycle $P + 4$ this value is converted to its normal basis representation and passed to register **regWGP**; the calculation of WGP-16(X^d) is hereby finished. ■

To reuse the components in moduleB we need to pass the same input X through the pipeline S times with the control signal **sel=0** (that is input $(X_i, 0)$ for $i = 0, \dots, S - 1$) and then one more time with control signal **sel=1** (that is input $(X_S, 1)$). We do not really care about the computations in WGP_T for the remaining $P + (S - 1)$ clock cycles: we could choose to send bubbles, but this would require a 16-bit wide 2-to-1 multiplexer on the input to moduleA (the multiplexer would pass value **s31** in states **initI**, **runI** and **runII** and the bubble otherwise). We removed registers at the beginning of WGP_T and consider **s31** as a part of the pipeline: inserting a multiplexer at this point would add logic to the first pipeline stage and increase the clock period. Instead, we just keep sending the same value X , but will refer to this “don’t care” X as “bubble”, marked \bigcirc , for the remaining discussion. The WGP_T data input is hard-wired to LFSR state **s31**, which is holding the value X . As just mentioned, we consider the LFSR stage **s31** as a part of the pipeline, which adds another “bubble” cycle before the new input value for WGP_T module is available, resulting in $P + S$ bubbles. Based on valid inputs vs. bubbles, we split the initialization phase into two states:

- **initI**: new inputs $(X_i, 0)$ for $i = 0, \dots, S - 1$ and $(X_S, 1)$, where X is the value in LFSR state **s31**
- **init II**: filling $P + S$ bubbles into WGP_T pipeline: $(\bigcirc, 0)$

We need $2M = 64$ initialization steps to complete the initialization phase. The first valid $\text{WGT} = \text{WGT-16}(X^d)$ will be available after $P + S + T$ clock cycles. We start the running phase, but break it into two states:

- **runI**: new inputs from LFSR state s_{31} , but no output
- **runII**: normal running phase producing valid $\text{WGT-16}(X^d)$ outputs

By setting the number of clock cycles spent in **runI** to $P + S + T - 1$, we produce the first valid output keystream in the first clock cycle of **runII**. From this point on, the **WG** module produces a new keystream bit every clock cycle, unless the keystream is intentionally paused by ce_i input. By setting $\text{ce_i}=0$, the LFSR stops and the **WGP_T** pipeline outputs another $P + S + T$ keystream bits before setting the output valid signal o_v to 0. In the running phase the signal sel is set to 0 for $\text{WGT-16}(X^d)$ computation. Control signals load and init are both set to 0, so now the LFSR operates without the **WGP** input, updating the state s_{31} only from feedback f , as is indicated in the column a in Table 4.22. Top view of the module **WG** with all three components (the LFSR, the **WGP_T** module and the FSM) and corresponding control signals can be seen in figure 4.35. The inputs of **WG** are the 1-bit control signals clk , reset , ce_i and start , and the 16-bit data input DIN used for the key/IV loading. The two 1-bit output signals are the **keystream** and the **key_valid** bit. The operation of multiplexers 1 and 2 was already discussed in Section 4.1 and is also clear from the Table 4.22. The multiplexer 3 disconnects the **WGT** bit from the **keystream** output when $\text{o_v}=0$.

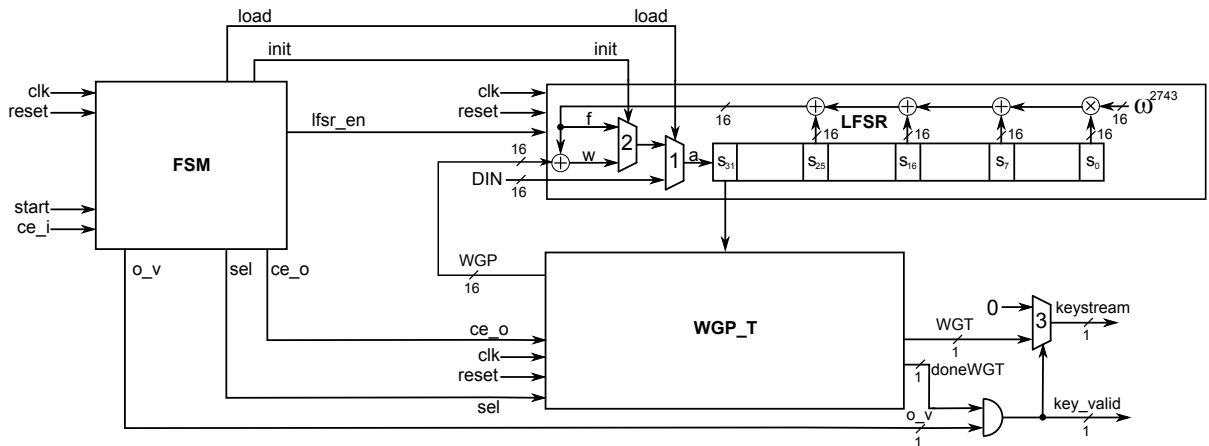


Figure 4.35: The WG-16: modules LFSR, WGP_T and FSM connected

		target submodule						
		LFSR				WGP_T		
phase	state	lfsr_en	load	init	a†	ce_o	sel	o_v
idle	idle	0	0	0	f	0	0	0
loading	load	1	1	0	DIN	0	0	0
initialization	initI	0	0	1	w	1	0 0 1	0
	initII	d	0	1	w	1	0	0
running	runI	ce_i	0	0	f	ce_i	0	0
	runII	ce_i	0	0	f	ce_i	0	1

Table 4.22: Six states for the WG-16 operation - values of the control signals

DIN	-	top-level data input	f	-	LFSR feedback
ce_i	-	top-level chip enable input	w	-	f + WGP
d	-	the doneWGP signal	†	-	the LFSR input

The state **initI** is controlled with a counter **s_count**: when **s_count=S** the input ($X_S, 1$) enters **WGP_T** pipeline and FSM moves to state **initII** where **s_count** is reset. In state **initII** we send $P + S$ bubbles into **WGP_T** by setting the control signal **sel** to 0 and letting the same input X roam through the pipeline. A counter **p_count** increments every clock cycle. We treat the **s31** as part of the pipeline (either last or first). Signal **lfsr_en** is tied to signal **d** which is only set when the new **WG-16(X^d)** value is ready (denoted **doneWGP** in the legend for Table 4.22 and in Figure 4.2 and Table 4.1 in Section 4.1) and that is when $\mathbf{p_count} = P + (S - 1)$. At the same time, a decision is made whether to return to state **initI** and start the next initialization step or jump to first step of running phase (**runI**). A counter **count** is used to keep track of number of initialization steps: if $\mathbf{count} < 2M - 1$ continue initialization, if $\mathbf{count} = 2M - 1$ the 64 steps of initialization are completed and we start the running phase. Note that counter **p_count** is reset in first initialization stage, that is the state **initI**. At the transition from initialization phase to running phase, we leave the counter **p_count** running and need to clear another $P + S + T$ bubbles, therefore we set the condition for transition to the state **runII** to $2(P + (S - 1)) + T + 1$.

Implementation results for the FSM module for different parameters of P , S and T are listed in Table 4.23. We can see that apart from a different number of registers (due to different length of counters) the three modules are almost identical in terms of complexity. The FSM is indeed very simple and is not a critical component in the WG-16.

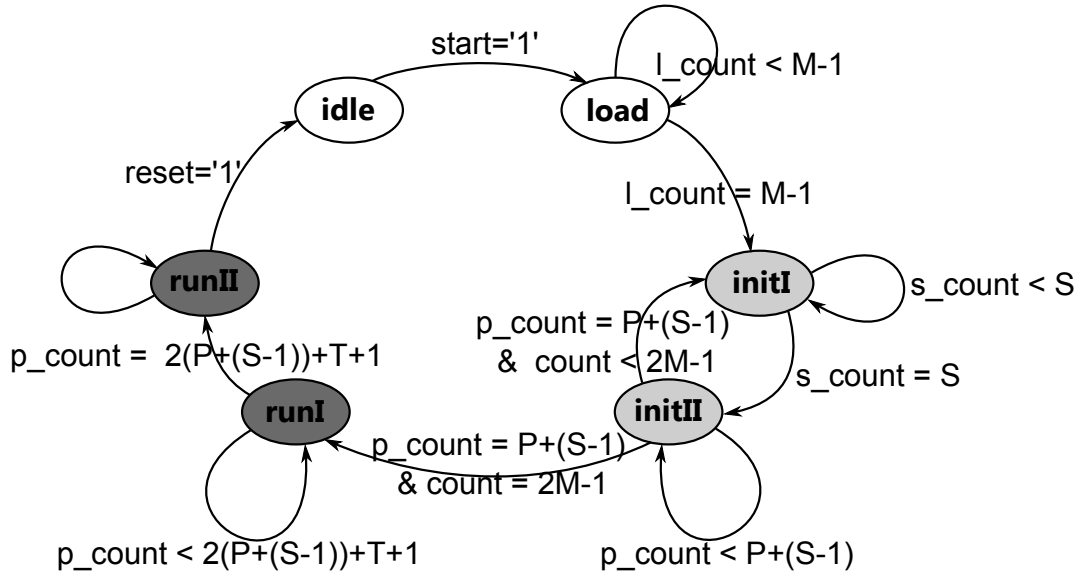


Figure 4.36: Six states for the WG-16 operation - the state transition diagram

Parameter			FPGA Results			
P	S	T	#FFs	#LUTs	#Slices	t [ns]
7	2	0	28	42	13	2.460
9	2	0	28	40	13	2.730
16	5	1	31	43	13	2.534

Table 4.23: Module FSM with different parameters P , S and T - implementation results

■ Case study - module WG_A16_BC8 initialization phase

In Table 4.24 we show the computation of the first $WGP-16(X^d)$ value in the initialization phase of the WG_A16_BC8 with FSM parameters $P = 7, S = 2$ and $T = 0$. The columns are the consecutive clock cycles (denoted *clk*), the clock cycle 0 denoting the last cycle of the loading phase. The first row is the LFSR state *s31*: that is the value that enters the pipeline, i.e. the parcel moving through the pipeline. The next three lines give the corresponding control signals for the LFSR: (*load*, *init*, *lfsr_en*)=(1,0,1) for **load** and (*load*, *init*, *lfsr_en*)=(0,1,0) for **initI**. When the control signal changes (is the value in the row changes), the change is marked explicitly. When the signal holds its value, we mark it with \circ if the unchanged value is 0 and with \bullet if the unchanged value is 1. For values of the signals that are more than 1-bit wide we use \dots to indicate the value had not changed. The \bigcirc is again used for bubbles. In row “input” we can see the current input to the pipeline. The rows below, separated with a double horizontal line, are the contents of the interstage registers: the row *A1* shows the parcel in the registers between the first and the second pipeline stage in **moduleA**, that is the stable value that is currently manipulated in the second stage of the pipeline. A single vertical line marks the end of **moduleA** and the beginning of **moduleBC8**. Another double horizontal line marks the end of the pipeline registers. Below this line we can see the remaining FSM signals:

- the **state** signal
- the **s_count** counter (counting from 0 to S)
- the control signal **sel**
- the **p_count** signal (counting to $P + (S - 1)$)
- the **count** signal
- the signal **d** for the **lfsr_en** update

Note that **sel** is an output of the FSM, included at this spot for clarity because it depends on the value of the **s_count** counter, while the remaining signals at the bottom of table 4.24 are internal to the FSM. The signal **s_count** is implemented as one-hot $(S + 1)$ -bit counter, but for visibility we listed the position of the bit 1 within the counter. The value **sel** is hardwired to the bit S in **s_count**.

As mentioned before, one initialization step takes $P + 2S$ clock cycles, that is 12 clock cycles for this example. The first **WGP** value is computed from the input α : in state **initI** (clock cycles 1,2 and 3) the values $(\alpha_1, 0)$, $(\alpha_2, 0)$ and $(\alpha_3, 1)$ are sent through the pipeline. The transition to **initII** phase occurs when **s_count** = 2. In clock cycle 4 the **s_count** is reset and the counter **p_count** is incremented. The first **WGP** value is available on the **WGP_T** output in clock cycle 12, where also the signal **d** is set to 1 (this signal is tied to the **lfsr_en** signal) and the value **p_count** = 8 triggers transition to state **initI**, because the current value of **count** is 0. At the same time **count** is incremented and its new value is visible in clock cycle 13. These 12 clock cycles, corresponding to one initialization step are separated with double vertical lines on the left and on the right. The new input to the **WGP_T** β is available in the 13th clock cycle - this is the beginning of the next initialization step. ■

clk	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17...
s31	○	α	α	α	α	α	α	β	β	β	β	β...
load	1	0	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
init	0	1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●...
lfsr_en	1	0	○	○	○	○	○	○	○	○	○	0	1	0	○	○	○	○
input		α ₁	α ₂	α ₃	○	○	○	○	○	○	○	○	○	β ₁	β ₂	β ₃	○	○
A1		○	α ₁	α ₂	α ₃	○	○	○	○	○	○	○	○	○	β ₁	β ₂	β ₃	○
A2			○	α ₁	α ₂	α ₃	○	○	○	○	○	○	○	○	○	β ₁	β ₂	β ₃ ...
A3				○	α ₁	α ₂	α ₃	○	○	○	○	○	○	○	○	○	β ₁	β ₂ ...
A4					○	α ₁	α ₂	α ₃	○	○	○	○	○	○	○	○	○	β ₁ ...
A5						○	α ₁	α ₂	α ₃	○	○	○	○	○	○	○	○	○
A6							○	α ₁	α ₂	α ₃	○	○	○	○	○	○	○	○
A7								○	α ₁	α ₂	α ₃	○	○	○	○	○	○	○
B1									○	α ₁	α ₂	α ₃	○	○	○	○	○	○
B2										○	α ₁	α ₂	α ₃	○	○	○	○	○
state														initI	initII
s_count	0	0	1	2	0	0	1	2	0
sel	0	0	0	1	0	○	○	○	○	○	○	○	○	0	0	1	0	○
p_count	0	0	0	0	0	1	2	3	4	5	6	7	8	9	0	0	0	1...
count	0	0	1
d	0	○	○	○	○	○	○	○	○	○	○	0	1	0	○	○	○	○

Table 4.24: Module WG-A16_BC8 - behavior in initialization phase using parameters: $P = 7, S = 2$ and $T = 0$

4.4.5 The WG-16 module

The following top-level WG-16 modules were implemented:

- WG_A16_BC8 with WGP_T_A16_2_BC8 (moduleA_M16_I8_2 \Rightarrow moduleBC8)
- WG_A8_BC8 with WGP_T_A8_2_BC8 (moduleA_M8_I8_2 \Rightarrow moduleBC8)
- WG_A4_BC4 with WGP_T_A4_2_BC4 (moduleA_M4_I4 \Rightarrow moduleBC4)

Implementation results for the three WG-16 modules are listed in Table 4.25.

Module	FPGA Results					ASIC Results		
	# of FFs	# of LUTs	# of Slices	t [ns]	$\frac{T}{A^2}$	Area [GE]	t [ns]	$\frac{T}{A^2}$
WG_A16_BC8	647	1450	441	7.495	6.8	12215	1.89	3.5
WG_A8_BC8	715	1389	461	7.129	6.6	12695	1.82	3.4
WG_A4_BC4	1388	1364	476	4.504	9.8	14628	1.46	3.2

Table 4.25: Top module WG-16, pipelined at different levels of the tower $\mathbb{F}_{((2^2)^2)^2}$ - implementation results

As expected, the WG_A4_BC4 using the WGP_T pipelined at the M_4/I_4 level results in the FPGA implementation with the shortest clock period, and although it is the biggest WG-16 module it shows the best $\frac{T}{A^2}$. We are a bit surprised at the results for the other two modules, WG_A16_BC8 and WG_A8_BC8: based on the FPGA implementation results for the corresponding two WGP_T modules (the optimized version in Table 4.20), we would expect the WG-16 using the M_8/I_8 level pipelining to outperform the M_{16}/I_{16} level pipelining. The clock period of WG_A8_BC8 is shorter, but due to area, the $\frac{T}{A^2}$ metric favors the WG_A16_BC8 module. We might be surprised that the number of slices used increases by the lower level of pipelining. Recalling the results for the WGP_T modules (Table 4.20), where we observed the decreasing numbers of slices, and since the changes in the FSM for each WGP_T and negligible and the LFSR stays the same, this increase appears inconsistent. Number of slices for area cost is sometimes not the best metric: the number of LUTs is decreasing with finer pipelining in both Table 4.20 and Table 4.25, and considering the 145 LUTs for the LFSR (Table 4.2) and approximately 40 LUTs for the FSM (Table 4.23), the number of LUTs for the WG-16 modules in Table 4.25 are as expected. The differences in the number of slices used are most likely due to place and route. If considering the clock period, ASIC results draw a similar picture: the shortest clock period was achieved by

WG_A4_BC4, and the other two modules are very close together. But the area gap between the FPGA modules is insignificant in comparison with the differences between the ASIC implementations: we find an area increase of approximately 2000 GEs when changing from the M_8/I_8 level to the M_4/I_4 level of pipelining. This area increase is also captured by the $\frac{T}{A^2}$ metric in the last column of Table 4.25, that was obtained using the ASIC results. Based in $\frac{T}{A^2}$, the most efficient ASIC implementation is the module WG_A16_BC8 using the pipeline WGP_T_A16_2_BC8 composed of moduleA pipelined at the M_{16}/I_8 level and moduleBC pipelined at the M_8 level. That is, the multipliers used in the decimation were not pipelined, and the multipliers that had to be reused were pipelined at the M_8 level. The big difference in area is caused by the increased number of registers. In the FPGA, each LUT output can be paired with a FF for free, and the long pipeline penalty is not visible (yet). The M_4/I_4 level pipelining is the most optimal choice for the FPGA, but at the same time the worst choice for the ASIC implementation.

An important consideration for the choice of the WG-16 module is also the number of pipeline stages (i.e. the depth of the pipeline). The initialization phase namely takes $64 \cdot (P + 2S)$ clock cycles. The depth of the pipeline, data-rate during initialization, measured in WGP per cycle and the total number of clock cycles needed for the initialization are given in table 4.26. Based on implementation results listed in Table 4.25 and on a shorter initialization phase of WG_A16_BC8, the WG_A16_BC8 is a better choice than WG_A8_BC8. However, if the initialization phase is negligible compared to the length of the keystream needed, once initialized the WG_A4_BC4 gives the highest frequency.

Module	depth of pipeline as $P + S + T$	data-rate (init) [WGP/cycle]	# cycles for init
WG_A16_BC8	7+2+0	$\frac{1}{11}$	704
WG_A8_BC8	9+2+0	$\frac{1}{13}$	832
WG_A4_BC4	16+5+1	$\frac{1}{26}$	1664

Table 4.26: Top module WG-16, pipelined at different levels of the tower $\mathbb{F}_{((2^2)^2)^2}$ - pipeline length and initialization phase

4.5 Tower construction $\mathbb{F}_{(2^4)^4} \cong \mathbb{F}_{2^{16}}$ - implementation

In this section we present implementation of module `WGP_T` using tower construction $\mathbb{F}_{(2^4)^4}$, that was described in Section 3.5.1. First part of this section presents implementation of basic building blocks for each level of the tower and the second part the implementation of module `WGP_T` itself.

4.5.1 Analysis of Basic Building Blocks

Arithmetic operations in \mathbb{F}_{2^4} .

Basic arithmetic operations needed for `WGP_T` are multiplication and inversion. We omit squaring, because it is only needed in inversion block I_4 , and since that module uses exponentiations to powers of 2 as well, we decide to perform squaring by transitioning to normal basis representation of $\mathbb{F}_{2^{16}}$ and shifting. At the end of this section we present some auxiliary submodules, that will be needed on the higher level of the tower.

For details about the finite field \mathbb{F}_{2^4} , that was constructed as an extension of degree 4 using the AOP $e(x) = x^4 + x^3 + x^2 + x + 1$, refer to Section 3.5.1.

Multiplication in \mathbb{F}_{2^4} : Let $A = a_0\alpha + a_1\alpha^2 + a_2\alpha^4 + a_3\alpha^3$ and $B = b_0\alpha + b_1\alpha^2 + b_2\alpha^4 + b_3\alpha^3$, where $a_i, b_i \in \mathbb{F}_2$, for $i = 0, 1, 2, 3$. The product $C = AB$ in $\mathbb{F}_{(2^2)^2}$ is computed as follows (relationships $\alpha^6 = \alpha$, $\alpha^7 = \alpha^2$ and $\alpha^8 = \alpha^3$, derived from $e(\alpha) = 0$, were used below):

$$\begin{aligned}
 AB &= (a_0\alpha + a_1\alpha^2 + a_2\alpha^4 + a_3\alpha^3)(b_0\alpha + b_1\alpha^2 + b_2\alpha^4 + b_3\alpha^3) \\
 &= (a_1b_2 + a_2b_1 + a_3b_3)\alpha \\
 &+ (a_2b_3 + a_3b_2 + a_0b_0)\alpha^2 \\
 &+ (a_3b_0 + a_0b_3 + a_1b_1)\alpha^4 \\
 &+ (a_0b_1 + a_1b_0 + a_2b_2)\alpha^3 \\
 &+ (a_0b_2 + a_2b_0 + a_1b_3 + a_3b_1)\alpha^5
 \end{aligned}$$

In the expression above the last component α^5 was left unreduced to aid the implementation: due to the AOP used in field construction, we find $\alpha^5 = 1 = \alpha + \alpha^2 + \alpha^4 + \alpha^3$, which means that the value $F = a_0b_2 + a_2b_0 + a_1b_3 + a_3b_1$ will be added to every other

component of the product. Let us define $\text{conv}(j, k) = a_j b_k + a_k b_j$ and $\mathbf{s}(n) = a_n b_n$ and rewrite coefficients of the product $C = AB = c_0\alpha + c_1\alpha^2 + c_2\alpha^4 + c_3\alpha^3$:

$$\begin{aligned} c_0 &= \text{conv}(1, 2) + \mathbf{s}(3) + F \\ c_1 &= \text{conv}(2, 3) + \mathbf{s}(0) + F \\ c_2 &= \text{conv}(3, 0) + \mathbf{s}(1) + F \\ c_3 &= \text{conv}(0, 1) + \mathbf{s}(2) + F \end{aligned}$$

So for $i = 0, 1, 2, 3$, the coefficient c_i is computed as

$$c_i = \text{conv}((i + 1) \bmod 4, (i + 2) \bmod 4) + \mathbf{s}((i + 3) \bmod 4) + F,$$

where F can be obtained using the same function conv

$$F = \text{conv}(0, 2) + \text{conv}(1, 3).$$

We obtained a Massey-Omura like multiplier (for details see [98]), that uses the same function with different (shifted) inputs for all components of the product. The circuit for computation of a single coefficient c_i is shown in Figure 4.37. The grey block represents the $\text{conv}(j, k)$ computation and $\mathbf{s}(n)$ is the AND gate below the conv block. The following index notation was used in Figure 4.37: $j = (i + 1) \bmod 4$, $k = (i + 2) \bmod 4$ and $n = (i + 3) \bmod 4$.

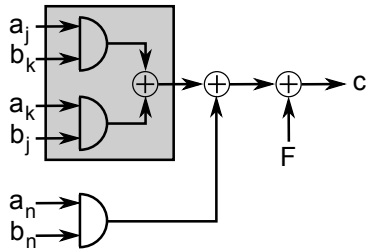


Figure 4.37: Block M_4 in \mathbb{F}_{2^4} - computation of coefficient c_i

Let us look at the gate count to compare this M_4 with M_4 from tower construction $\mathbb{F}_{((2^2)^2)^2}$ in terms of AND and XOR gates and in terms of NAND gates (columns 5 and 6 of Table 4.27), because mere comparison of AND and XOR gates becomes difficult otherwise. For details see Appendix E. Inspecting the gate count, we find that this M_4 has a bigger area, but a

M_4	Gate Count					FPGA Results		
	Area		Critical Path	Area	Critical Path	# of	# of	Delay
Tower Field	N_A	N_X	Delay - T_A and T_X	N	Delay - T	LUTs	Slices	[ns]
$\mathbb{F}_{(2^4)^4}$	16	15	$T_A + 3T_X$	92	11	10	4	7.781
$\mathbb{F}_{(((2^2)^2)^2)^2}$	21	9	$T_A + 5T_X$	66	14	11	5	8.517

Table 4.27: Comparison of M_4 blocks using different tower constructions

$\mathbb{F}_{(2^4)^4}$ vs. $\mathbb{F}_{(((2^2)^2)^2)^2}$ (Section 4.4.1) in terms of gate count and actual FPGA implementation results:
 N_A, T_A area and delay of one AND gate, N_X, T_X area and delay of one XOR gate,
 N, T area and delay of one NAND gate

shorter delay. However, FPGA implementation results in the last three columns of Table 4.27 show that current M_4 has an advantage in terms of area and delay, indicating successful optimizations done by Xilinx ISE. The difference between the modules is minimal.

Inversion in \mathbb{F}_{2^4} : Using the generalization of Fermat's little Theorem we can compute the inverse of an element $A \in \mathbb{F}_{2^4}$ as

$$A^{-1} = A^{2^4-2} = A^{14} = A^2 \cdot A^4 \cdot A^8 = A^2 \cdot A^{2^2} \cdot A^{2^3}.$$

The three factors above can be obtained by a right cyclic shift for 1, 2 and 3 positions respectively. Two blocks M_4 are used to obtain the inverse. The circuit of block I_4 can be seen in Figure 4.38

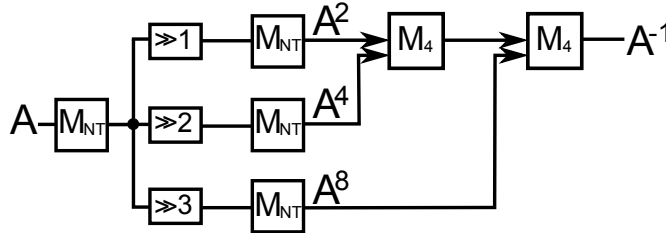


Figure 4.38: Inversion block I_4 in \mathbb{F}_{2^4}

The shift operations are considered for free, if we disregard the transition matrices, since they are a simple rewiring of the inputs. But a single multiplier occupies the area of 92 NAND gates and has a delay of 11 NAND gates. We end up with area of 184 and delay of 22 NAND gates for inversion block I_4 . This area complexity looks terrifying in comparison with the multiplier and even worse when compared with area of 50 NAND gates and delay

of 22 NAND gates for module I_4 from tower construction $\mathbb{F}_{(((2^2)^2)^2)^2}$.

The big area complexity motivates some additional optimization that was achieved by computing the two products; since \mathbb{F}_{2^4} is a small field, we can afford such computational effort. We start by computing the product $A^2 \cdot A^4$:

$$\begin{aligned}
A^2 \cdot A^4 &= (a_3\alpha + a_0\alpha^2 + a_1\alpha^4 + a_2\alpha^3)(a_2\alpha + a_3\alpha^2 + a_0\alpha^4 + a_1\alpha^3) \\
&= (a_1(a_2 + a_3) + a_0)\alpha + (a_2(a_0 + a_3) + a_1)\alpha^2 \\
&+ (a_3(a_0 + a_1) + a_2)\alpha^4 + (a_0(a_1 + a_2) + a_3)\alpha^3 \\
&+ (a_0a_3 + a_0a_1 + a_1a_2 + a_2a_3)\alpha^5
\end{aligned}$$

Once again, the component α^5 was left separated to ease the rest of the computation. The second step is multiplying the obtained product by $A^8 = a_3\alpha + a_0\alpha^2 + a_1\alpha^4 + a_2\alpha^3$ and using the relationship $\alpha^5 = \alpha^4 + \alpha^3 + \alpha^2 + \alpha$. Note that coefficients $a_0, a_1, a_2, a_3 \in \mathbb{F}_2$, thus the addition $a_i + 1$ inverts the bit a_i and we can freely use notation $\bar{a}_i = a_i + 1$. The following relationships for the coefficients of the inverse $A^{-1} = i_0\alpha + i_1\alpha^2 + i_2\alpha^4 + i_3\alpha^3$ were obtained:

$$\begin{aligned}
i_0 &= (a_3 + a_0)a_1\bar{a}_2 + a_2\overline{\bar{a}_3a_0} \\
i_1 &= (a_0 + a_1)a_2\bar{a}_3 + a_3\overline{\bar{a}_0a_1} \\
i_2 &= (a_1 + a_2)a_3\bar{a}_0 + a_0\overline{\bar{a}_1a_2} \\
i_3 &= (a_2 + a_3)a_0\bar{a}_1 + a_1\overline{\bar{a}_2a_3}
\end{aligned}$$

Again, we are able to find a pattern: $i_i = (a_{i3} + a_{i0})a_{i1}\bar{a}_{i2} + a_{i2}\overline{\bar{a}_{i3}a_{i0}}$, where $i0 = i$, $i1 = (i + 1) \bmod 4$, $i2 = (i + 2) \bmod 4$ and $i3 = (i + 3) \bmod 4$. In terms of FPGA, the equations show that each output bit of the inversion block is a Boolean function of 4 input bits, hence 1 LUT for 2 output bits using both LUT outputs O_5 and O_6 . The part of the inversion block I_4 circuit for obtaining one coefficient is shown in Figure 4.39. Now we can estimate the area and the delay of the new inversion module (note that \bar{a}_i indicates a NOT gate, and to avoid further complications in gate count, we will just give the gate count in terms of NAND gates: we obtain the area/delay of 19/9 NAND gates for one coefficient of the inverse i_i , which gives a total area of 76 and delay of 9 NAND gates. Not only is the optimized inversion block smaller than the non-optimized version, it also has a significantly shorter propagation delay. We also notice better results compared with M_4 from this tower construction, and a significantly shorter delay, but an increase when compared to the inversion module I_4 from tower construction $\mathbb{F}_{(((2^2)^2)^2)^2}$ (please refer to Appendix E Tables

E.2 and E.3 for details). The FPGA implementation results for the original I_4 module (denoted I_4 - bad) and the optimized module (denoted I_4 - good) can be seen in Table 4.28. We notice the modules are practically identical: a closer inspection of LUT contents reveals Xilinx ISE was able to perform the same optimizations as our pen-and-paper method.

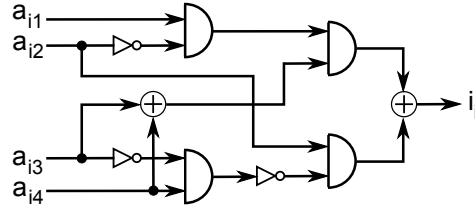


Figure 4.39: Inversion block I_4 in \mathbb{F}_{2^4} - computation of coefficient i_i

Auxiliary computations in \mathbb{F}_{2^4} : Other basic building blocks that will be needed for arithmetic in upper level of the tower are multiplications of $A = (a_0, a_1, a_2, a_3) \in \mathbb{F}_{2^4}$ with different constants. Recall from Section 3.5.1 that α is the normal element and $\lambda = \alpha + \alpha^3$ the generator of \mathbb{F}_{2^4} . These auxiliary submodules needed are the following:

- $AL^2 = a_3\alpha + (a_0 + a_2 + a_3)\alpha^2 + (a_1 + a_2)\alpha^4 + (a_0 + a_1 + a_2 + a_3)\alpha^3$
- $AL\alpha^2 = (a_0 + a_1 + a_3)\alpha + a_2\alpha^2 + (a_0 + a_1 + a_2)\alpha^4 + (a_1 + a_3)\alpha^3$
- $AL\alpha^3 = (a_0 + a_1 + a_2)\alpha + (a_2 + a_3)\alpha^2 + (a_0 + a_2 + a_3)\alpha^4 + (a_0 + a_1)\alpha^3$
- $AL^2\alpha^4 = (a_0 + a_2)\alpha + (a_0 + a_1 + a_2)\alpha^2 + a_3\alpha^4 + (a_1 + a_2 + a_3)\alpha^3$

All of four multiplications above can be implemented with a circuit that has a delay of 2 and area of 4 XOR gates (i.e. delay 6 and area of 16 NAND gates). An exception is the module for multiplication with λ^2 , which has the area of 5 XOR gates. Separate modules for the above computations prove to be more efficient than using a cascade of two modules. Let us demonstrate this with an example: using a module for multiplication by $\lambda\alpha$ is more efficient than multiplication by λ followed by multiplication by α , as can be seen from the equations for all three multiplications given below:

- $A\lambda = (a_1 + a_2 + a_3)\alpha + (a_0 + a_1)\alpha^2 + (a_0 + a_1 + a_2 + a_3)\alpha^4 + a_2\alpha^3$
- $A\alpha = a_2\alpha + (a_0 + a_2)\alpha^2 + (a_2 + a_3)\alpha^4 + (a_1 + a_2)\alpha^3$
- $A\lambda\alpha = (a_0 + a_1 + a_2 + a_3)\alpha + a_0\alpha^2 + (a_0 + a_1 + a_3)\alpha^4 + (a_2 + a_3)\alpha^3$

Multiplication with λ and multiplication with α require a delay of 2 and area of 4 XOR gates each, giving a delay of 4 and area of 8 XOR gates in total. At the same time, the multiplication with $\lambda\alpha$ has a delay of 2 and area of 4 XOR gates. For sure, synthesis tools would perform this optimization as well.

Arithmetic operations in $\mathbb{F}_{(2^4)^4}$

We now proceed to the arithmetic in the top level of the tower, and discuss multiplication and inversion in $\mathbb{F}_{(2^4)^4}$. A discussion of operations such as squaring or exponentiation to powers of 16 is not necessary, since this is the top level and those operations can be carried out by conversion to normal basis representation and shifting.

Multiplication in $\mathbb{F}_{(2^4)^4}$: The circuit for the product $C = AB$, where $A, B \in \mathbb{F}_{(2^4)^4}$, $A = a_0\beta + a_1\beta^{16} + a_2\beta^{256} + a_3\beta^{4096}$, $B = b_0\beta + b_1\beta^{16} + b_2\beta^{256} + b_3\beta^{4096}$, with $a_i, b_i \in \mathbb{F}_{2^4}$, $i = 0, 1, 2, 3$ was derived as follows:

$$\begin{aligned} & (a_0\beta + a_1\beta^{16} + a_2\beta^{256} + a_3\beta^{4096})(b_0\beta + b_1\beta^{16} + b_2\beta^{256} + b_3\beta^{4096}) \\ = & a_0b_0\beta^2 + (a_0b_1 + a_1b_0)\beta^{17} + a_1b_1\beta^{32} \\ & + (a_0b_2 + a_2b_0)\beta^{257} + (a_1b_2 + a_2b_1)\beta^{272} + a_2b_2\beta^{512} \\ & + (a_0b_3 + a_3b_0)\beta^{4097} + (a_1b_3 + a_3b_1)\beta^{4112} + (a_3b_2 + a_2b_3)\beta^{4352} + a_3b_3\beta^{8192}, \end{aligned}$$

using relationships below:

$$\begin{array}{lll} \beta^2 & = & \alpha^2\beta + \lambda\alpha^3\beta^{16} + \lambda^2\alpha^2\beta^{256} + \alpha^4\beta^{4096} & \beta^{32} & = & (\beta^2)^{16} & \beta^{512} & = & (\beta^2)^{16^2} \\ \beta^{17} & = & \lambda^2\beta + \lambda\alpha^3\beta^{256} + \lambda\alpha^2\beta^{4096} & \beta^{272} & = & (\beta^{17})^{16} & \beta^{4352} & = & (\beta^{17})^{16^2} \\ \beta^{257} & = & \lambda^2\alpha^4\beta + \lambda^2\alpha^4\beta^{256} & \beta^{4112} & = & (\beta^{257})^{16} & \beta^{8192} & = & (\beta^2)^{16^3} \\ \beta^{4097} & = & (\beta^{17})^{16^3} & & & & & & \end{array}$$

Let us now introduce some additional notation:

$$\begin{array}{ll} k_0 & = (a_0 + a_1)(b_0 + b_1) \\ k_1 & = (a_1 + a_2)(b_1 + b_2) \\ s_0 & = a_0b_0 & k_2 & = (a_2 + a_3)(b_2 + b_3) \\ s_1 & = a_1b_1 & k_3 & = (a_0 + a_2)(b_0 + b_2) \\ s_2 & = a_2b_2 & k_4 & = (a_0 + a_3)(b_0 + b_3) \\ s_3 & = a_3b_3 & k_5 & = (a_1 + a_3)(b_1 + b_3) \end{array}$$

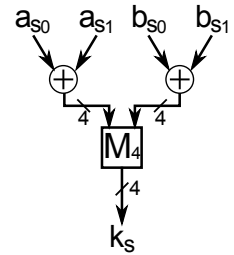


Figure 4.40: Block M_{16} in $\mathbb{F}_{(2^4)^4}$ - component $conv4(s_0, s_1)$

The four components of $C = c_0\beta + c_1\beta^{16} + c_2\beta^{256} + c_3\beta^{4096}$ simplify to the following:

$$\begin{aligned} c_0 &= k_0\lambda^2 + k_1\lambda\alpha^2 + k_2\lambda\alpha^3 + k_3\lambda^2\alpha^4 + s_0 \\ c_1 &= k_1\lambda^2 + k_2\lambda\alpha^2 + k_4\lambda\alpha^3 + k_5\lambda^2\alpha^4 + s_1 \\ c_2 &= k_2\lambda^2 + k_4\lambda\alpha^2 + k_0\lambda\alpha^3 + k_3\lambda^2\alpha^4 + s_2 \\ c_3 &= k_4\lambda^2 + k_0\lambda\alpha^2 + k_1\lambda\alpha^3 + k_5\lambda^2\alpha^4 + s_3 \end{aligned}$$

Inserting $k_s = (a_{s0} + a_{s1})(b_{s0} + b_{s1})$ for $s = 1, \dots, 5$ and $s_0, s_1 = 0, \dots, 3$ as defined above, and using notation $(a_{s0} + a_{s1})(b_{s0} + b_{s1}) = \text{conv4}(s_0, s_1)$ we find the following pattern:

$$c_i = \text{conv4}(i, i_1) \cdot \lambda^2 + \text{conv4}(i_1, i_2) \cdot \lambda\alpha^2 + \text{conv4}(i_2, i_3) \cdot \lambda\alpha^3 + \text{conv4}(i, i_2) \cdot \lambda^2\alpha^4 + s_i$$

where $i_0 = i$, $i_1 = (i + 1) \bmod 4$, $i_2 = (i + 2) \bmod 4$ and $i_3 = (i + 3) \bmod 4$.

Elements s_i and $k_s = \text{conv4}(s_0, s_1)$ are somewhat similar to the elements $s(i)$ and $\text{conv}(i, j)$ from the lower level, but use the subfield multiplication block M_4 . Also notice the reverse order modulo-2 addition and multiplication in modules conv4 . Schematic for conv4 can be seen in Figure 4.40.

Schematic of the multiplication block M_{16} can be seen in Figure 4.41. Gate count reveals an area of 1480 and the delay of 32 NAND gates. In comparison with M_{16} multiplication block that was obtained using tower construction $\mathbb{F}_{((2^2)^2)^2}$, the current M_{16} block is a bit bigger but has a smaller delay. However, the FPGA results in Table 4.28 are not so encouraging: the area gap between the two M_{16} blocks is over 30 slices in favor of M_{16} from $\mathbb{F}_{((2^2)^2)^2}$ implementation, but the current M_{16} module is still faster.

Inversion in $\mathbb{F}_{(2^4)^4}$: For inversion we can again use the Itoh-Tsuji algorithm, that was explained in detail in Section 4.4.1. The inverse of element $A \in \mathbb{F}_{(2^4)^4}$ is computed as $A^{-1} = (A^r)^{-1}A^{r-1}$, where $r = \frac{2^{4 \cdot 4} - 1}{2^4 - 1} = 4369$. Decomposition $A^{r-1} = A^{16^3 + 16^2 + 16}$ leads to the circuit that can be seen in Figure 4.42. The shift blocks on the Figure are shaded grey to emphasize the exponentiation to powers of 16 (instead of powers of 2 we have encountered so far). There is another grey block in Figure 4.42, namely the inversion block I_4 . This block is emphasized since it operates on elements of the base field \mathbb{F}_{2^4} ; it takes a 4-bit input and produces a 4-bit output. The question arises: which 4 bits are to be connected to the inputs of I_4 ? The input to I_4 is element A^r , which is an element of the subfield \mathbb{F}_{2^4} (4.4.1), say $A^r = b \in \mathbb{F}_{2^4}$. We can write:

$$\begin{aligned} b &= b \cdot 1 \\ &= b(\beta + \beta^{16} + \beta^{16^2} + \beta^{16^3}) \\ &= b\beta + b\beta^{16} + b\beta^{16^2} + b\beta^{16^3} \in \mathbb{F}_{(2^4)^4} \end{aligned}$$

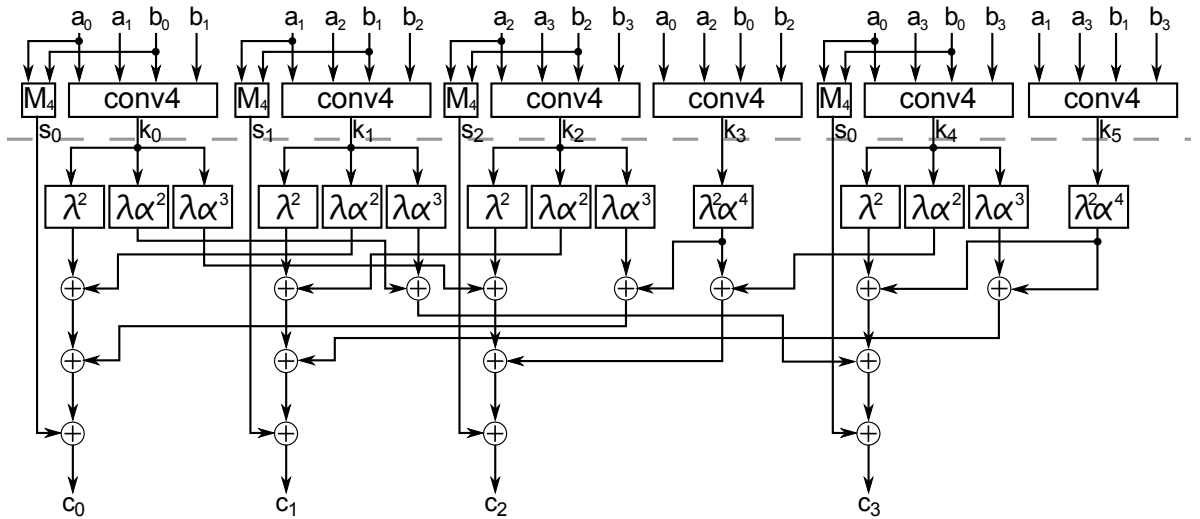


Figure 4.41: Multiplication block M_{16} in $\mathbb{F}_{(2^4)^4}$

This was the last missing piece: before computing $(A^r)^{-1}$ we must represent A^r as a 4 bit element. Based on the discussion above we simply take first 4 bits of the 16-bit A^r and connect them to I_4 input. The I_4 output is then expanded to its 16-bit form by copying the 4 output bits into remaining 12 bits, thus representing the inverse $(A^r)^{-1}$ in basis $\{\beta, \beta^{16}, \beta^{16^2}, \beta^{16^3}\}$. We can now use the M_{16} multiplier to obtain the inverse $A^{-1} = I \in \mathbb{F}_{(2^4)^4}$ we were trying to find.

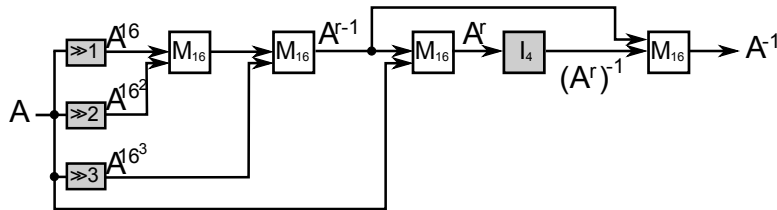


Figure 4.42: Inversion block I_{16} in $\mathbb{F}_{(2^4)^4}$

When designing the inverter I_4 for inversion in the subfield \mathbb{F}_{2^4} , we used an algebraic optimization that was able to significantly reduce the delay and the gate count. But it turns out that synthesis tools were able to perform the optimization as well, which resulted in two inverters having the same time and area complexity. Hence we rely on the tools to be able to perform atleast some optimization for I_{16} . Problem might arise when using the

inversion in pipelined `WGP_T` : inserting a pipeline stage between the first two multipliers might prevent the tools from finding a good solution. Implementation results for module I_{16} are a bit disappointing: the module is very greedy in terms of both, the area and the delay. The tools were not able to perform many optimization, most likely because the value A^{r-1} is used again in the final multiplication (see Figure 4.42). Module I_{16} from current tower $\mathbb{F}_{(2^4)^4}$ is four times bigger and more than 3 ns slower compared to module I_{16} from tower $\mathbb{F}_{((2^2)^2)^2}$.

Area and time complexities of basic building blocks

The FPGA implementation results of basic building blocks for tower field construction $\mathbb{F}_{(2^4)^4}$ are collected in Table 4.28. Results for submodules that perform multiplications with constants \mathbb{F}_{2^4} level of the tower were omitted.

Basic Building Block	FPGA Results		
	# of LUTs	# of Slices	t [ns]
M_4	10	4	7.781
I_4 - bad	2	2	6.883
I_4 - good	2	2	6.861
M_{16}	240	84	13.340
I_{16}	611	251	26.109
M_{NT}	18	8	7.982
M_{TN}	15	9	7.961

Table 4.28: Basic building blocks for arithmetic in tower field $\mathbb{F}_{(2^4)^4}$ - implementation results

The trace computation

The last missing puzzle in need of our attention, before we go on to the module `WGP_T` , is the implementation of the trace function. At the end of Section in equation 3.28, the following expression for the absolute trace of element $Z \in \mathbb{F}_{(2^4)^4}$ was obtained:

$$\text{Tr}(Z) = \bigoplus_{k=0}^{15} z_k$$

Hence, the absolute trace of an element from $\mathbb{F}_{(2^4)^4}$ is nothing else but just the modulo-2 sum, that is **XOR**, of its 16 coordinates. Therefore, we can just reuse the trace computation that was developed for tower construction $\mathbb{F}_{((2^2)^2)^2}$ in Section 4.4.2.

4.5.2 Module WGP_T - Design of Pipelined Architecture

The entire idea of using tower field constructions is based on the possibility of pipelining the modules at a finer granularity. Considering poor performance of submodules M_{16} and I_{16} , the only reasonable pipelining option for tower construction $\mathbb{F}_{(2^4)^4}$ is pipelining at M_4/I_4 level. Module M_{16} interstage registers were inserted at the position indicated by the grey dashed horizontal line in Figure 4.41 showing the circuit for M_{16} . Note that the modules above the line consist solely of one layer of **XOR** gates and one layer of M_4 submodules. The layer below the pipeline border consists of multiplications with constants followed by three layers of **XOR** gates. The decision where to insert the pipeline registers was based on the gate count for module M_{16} : delay of 14 **NANAD** gates above and 15 **NAND** gates below the border is the most balanced pipelining option.

Top level schematic of module **WGP_T** for tower construction $\mathbb{F}_{(2^4)^4}$ is shown in Figure 3.11. The implemented 20-stage pipeline can be seen in Figure 4.43. Implementation results for module **WGP_T** using tower construction $\mathbb{F}_{(2^4)^4}$ pipelined at M_4/I_4 level are given in Table 4.29 below:

Module	FPGA Results			
	#FFs	#LUTs	#Slices	t [ns]
WGP_T_M4_I4_T2	1050	1722	589	4.918

Table 4.29: Module **WGP_T_M4_I4_T2** using tower construction $\mathbb{F}_{(2^4)^4}$ pipelined at M_4/I_4 level - implementation results

Although we have a bigger area the clock period is comparable with the one achieved by the M_4/I_4 level **WGP_T** module from tower construction $\mathbb{F}_{((2^2)^2)^2}$.

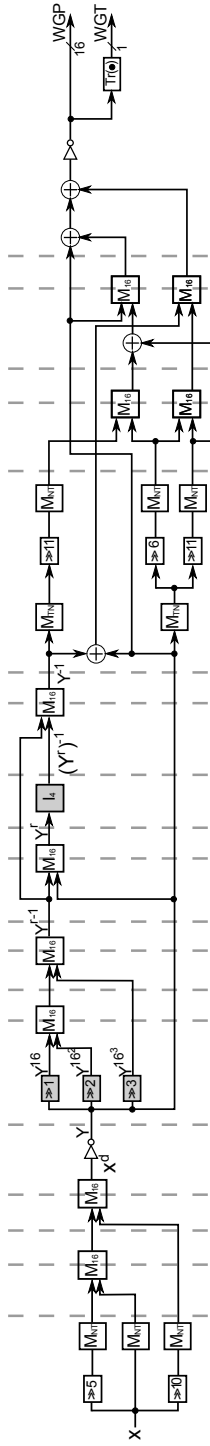


Figure 4.43: Module WGP_T_M4_I4_T2 - pipelined architecture for module WGP_T using tower field construction $\mathbb{F}_{(2^4)^4}$

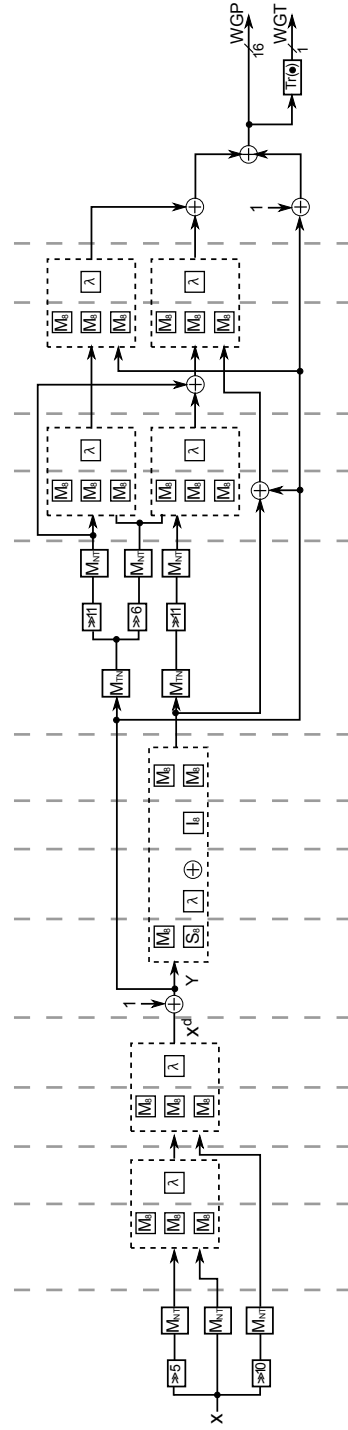


Figure 4.44: Module WGP_T_M8_I8_T3 - pipelined architecture for module WGP_T using tower field construction $\mathbb{F}_{(2^8)^2}$

4.6 Tower construction $\mathbb{F}_{(2^8)^2} \cong \mathbb{F}_{2^{16}}$ - implementation

4.6.1 Analysis of Basic Building Blocks

As already mentioned in Section 3.6.1, this tower construction is aiming at the implementation of table look-up based algorithms. The tower field basis for $\mathbb{F}_{(2^8)^2}$ is a mixed basis, using polynomial basis for the lower level of the tower \mathbb{F}_{2^8} and normal basis for the top level. The polynomial basis for \mathbb{F}_{2^8} was chosen, because it is a common practice to implement the table look-up methods using polynomial basis, but there is no particular reason for doing so.

Arithmetic in \mathbb{F}_{2^8}

One of the earliest applications of table look-ups for \mathbb{F}_{2^8} arithmetic used in cryptography dates back to 1992, see [93]. The field \mathbb{F}_{2^8} was constructed by adjoining the root α of a primitive polynomial to the prime field \mathbb{F}_2 , and thus all the elements of $\mathbb{F}_{2^8}^*$ can be represented as powers of the generator α : let $\bar{a}_i = (a_0, \dots, a_7)$ be the vector representation of the element $A = \alpha^i = \sum_{j=0}^7 a_j \alpha^j$ for $i = 0, \dots, 254$. Two look-up tables are precomputed and stored:

- the "log" table `ltable` storing the exponents of $\mathbb{F}_{2^8}^*$ elements: vector \bar{a}_i serves as `ltable` index, by which we access the exponent i , and
- the "antilog" table `atable`, storing the elements of $\mathbb{F}_{2^8}^*$: now the exponent i serves as the index by which we access the value \bar{a}_i .

The two tables are given below. Let us take a look at a couple of short examples to demonstrate how the tables are accessed:

- element $1 \in \mathbb{F}_{2^8}$ can be represented as $1 = \alpha^0$, so it is the element `aTable[0]`, and since the binary representation of 1 is "10000000", the exponent `x"0"` is stored at `ltable[128]`;
- element $\alpha + \alpha^3 + \alpha^4 + \alpha^5 = \alpha^9 \in \mathbb{F}_{2^8}$ is the element `atable[9]`, and since its binary representation "01011100" equals decimal 92, the value `x"09"` is stored in `ltable[92]`.

Note that the polynomial basis representation is least-significant bit (LSB) first, and is later interpreted as most-significant bit (MSB) first when used to access the tables. There

is a simple reason for that, namely, we want the tables to work as a regular memory arrays (which makes the implementation easier), and most systems are MSB first.

`atable`:

```
{0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01, 0xb8, 0x5c, 0x2e, 0x17, 0xb3, 0xe1, 0xc8, 0x64
0x32, 0x19, 0xb4, 0x5a, 0x2d, 0xae, 0x57, 0x93, 0xf1, 0xc0, 0x60, 0x30, 0x18, 0xc0, 0x06, 0x03
0xb9, 0xe4, 0x72, 0x39, 0xa4, 0x52, 0x29, 0xac, 0x56, 0x2b, 0xad, 0xee, 0x77, 0x83, 0xf9, 0xc4
0x62, 0x31, 0xa0, 0x50, 0x28, 0x14, 0x0a, 0x05, 0xba, 0x5d, 0x96, 0x4b, 0x9d, 0xf6, 0x7b, 0x85
0xfa, 0x7d, 0x86, 0x43, 0x99, 0xf4, 0x7a, 0x3d, 0xa6, 0x53, 0x91, 0xf0, 0x78, 0x3c, 0x1e, 0x0f
0xbf, 0xe7, 0xcb, 0xdd, 0xd6, 0x6b, 0x8d, 0xfe, 0x7f, 0x87, 0xfb, 0xc5, 0xda, 0x6d, 0x8e, 0x47
0x9b, 0xf5, 0xc2, 0x61, 0x88, 0x44, 0x22, 0x11, 0xb0, 0x58, 0x2c, 0x16, 0x0b, 0xbd, 0xe6, 0x73
0x81, 0xf8, 0x7c, 0x3e, 0x1f, 0xb7, 0xe3, 0xc9, 0xdc, 0x6e, 0x37, 0xa3, 0xe9, 0xcc, 0x66, 0x33
0xa1, 0xe8, 0x74, 0x3a, 0x1d, 0xb6, 0x5b, 0x95, 0xf2, 0x79, 0x84, 0x42, 0x21, 0xa8, 0x54, 0x2a
0x15, 0xb2, 0x59, 0x94, 0x4a, 0x25, 0xaa, 0x55, 0x92, 0x49, 0x9c, 0x4e, 0x27, 0xab, 0xed, 0xce
0x67, 0x8b, 0xfd, 0xc6, 0x63, 0x89, 0xfc, 0x7e, 0x3f, 0xa7, 0xeb, 0xcd, 0xde, 0xf1, 0x8f, 0xff
0xc7, 0xdb, 0xd5, 0x69, 0x8c, 0x46, 0x23, 0xa9, 0xec, 0x76, 0x3b, 0xae, 0xea, 0x75, 0x82
0x41, 0x98, 0x4c, 0x26, 0x13, 0xb1, 0xe0, 0x70, 0x38, 0x1c, 0x0e, 0x07, 0xbb, 0xe5, 0xca, 0x65
0x8a, 0x45, 0x9a, 0x4d, 0x9e, 0x4f, 0x9f, 0xf7, 0xc3, 0xd9, 0xd4, 0x6a, 0x35, 0xa2, 0x51, 0x90
0x48, 0x24, 0x12, 0x09, 0xbc, 0x5e, 0x2f, 0xaf, 0xef, 0xcf, 0xdf, 0xd7, 0xd3, 0xd1, 0xd0, 0x68
0x34, 0x1a, 0x0d, 0xbe, 0x5f, 0x97, 0xf3, 0xc1, 0xd8, 0x6c, 0x36, 0x1b, 0xb5, 0xe2, 0x71, 0x00}
```

`ltable`:

```
{0xff, 0x07, 0x06, 0x1f, 0x05, 0x37, 0x1e, 0xcb, 0x04, 0xe3, 0x36, 0x6c, 0x1d, 0xf2, 0xca, 0x4f
0x03, 0x67, 0xe2, 0xc4, 0x35, 0x90, 0x6b, 0xb, 0x1c, 0x11, 0xf1, 0xfb, 0xc9, 0x84, 0x4e, 0x74
0x02, 0x8c, 0x66, 0xb7, 0xe1, 0x95, 0xc3, 0x9c, 0x34, 0x26, 0x8f, 0x29, 0x6a, 0x14, 0x0a, 0xe6
0x1b, 0x31, 0x10, 0x7f, 0xf0, 0xdc, 0xfa, 0x7a, 0xc8, 0x23, 0x83, 0xbb, 0x4d, 0x47, 0x73, 0xa8
0x01, 0xc0, 0x8b, 0x43, 0x65, 0xd1, 0xb6, 0x5f, 0xe0, 0x99, 0x94, 0x3b, 0xc2, 0xd3, 0x9b, 0xd5
0x33, 0xde, 0x25, 0x49, 0x8e, 0x97, 0x28, 0x16, 0x69, 0x92, 0x13, 0x86, 0x09, 0x39, 0xe5, 0xf4
0x1a, 0x63, 0x30, 0xa4, 0x0f, 0xcf, 0x7e, 0xa0, 0xef, 0xb4, 0xdb, 0x55, 0xf9, 0x5d, 0x79, 0xad
0xc7, 0xfe, 0x22, 0x6f, 0x82, 0xbe, 0xba, 0x2c, 0x4c, 0x89, 0x46, 0x3e, 0x72, 0x41, 0xa7, 0x58
0x00, 0x70, 0xbf, 0x2d, 0x8a, 0x3f, 0x42, 0x59, 0x64, 0xa5, 0xd0, 0xa1, 0xb5, 0x56, 0x5e, 0xae
0xdf, 0x4a, 0x98, 0x17, 0x93, 0x87, 0x3a, 0xf5, 0xc1, 0x44, 0xd2, 0x60, 0x9a, 0x3c, 0xd4, 0xd6
0x32, 0x80, 0xdd, 0x7b, 0x24, 0xbc, 0x48, 0xa9, 0x8d, 0xb8, 0x96, 0x9d, 0x27, 0x2a, 0x15, 0xe7
0x68, 0xc5, 0x91, 0x0c, 0x12, 0xfc, 0x85, 0x75, 0x08, 0x20, 0x38, 0xcc, 0xe4, 0x6d, 0xf3, 0x50
0x19, 0xf7, 0x62, 0xd8, 0x2f, 0x5b, 0xa3, 0xb0, 0x0e, 0x77, 0xce, 0x52, 0x7d, 0xab, 0x9f, 0xe9
0xee, 0xed, 0xb3, 0xec, 0xda, 0xb2, 0x54, 0xeb, 0xf8, 0xd9, 0x5c, 0xb1, 0x78, 0x53, 0x9c, 0xea
0xc6, 0xd, 0xfd, 0x76, 0x21, 0xcd, 0x6e, 0x51, 0x81, 0x7c, 0xbd, 0xaa, 0xb9, 0x9e, 0x2b, 0xe8
0x4b, 0x18, 0x88, 0xf6, 0x45, 0x61, 0x3d, 0xd7, 0x71, 0x2e, 0x40, 0x5a, 0xa6, 0xa2, 0x57, 0xaf}
```

Multiplication in \mathbb{F}_{2^8} : Two \mathbb{F}_{2^8} elements $A = \alpha^i$ and $B = \alpha^j$, represented as powers of the generator α , can be multiplied as follows:

$$A \cdot B = \alpha^i \cdot \alpha^j = \alpha^{(i+j)} \pmod{2^8-1}$$

This is the basic idea behind the table look-up multiplication: first, we need to access the `ltable` twice to obtain both indices (by reading the `ltable` contents at addresses \bar{a}_i and \bar{a}_j), then we add them up modulo $2^8 - 1$, and use the obtained reduced sum for accessing `atable`:

$$A \cdot B = \text{atable}[(\text{ltable}[\bar{a}_i] + \text{ltable}[\bar{a}_j]) \pmod{255}]$$

Inversion in \mathbb{F}_{2^8} : To obtain the inverse of the element $A = \alpha^i \in \mathbb{F}_{2^8}$, we proceed by rewriting (note the use of relationship $\alpha^{2^8} = 1$):

$$A^{-1} = \alpha^{-i} = 1 \cdot \alpha^{-i} = \alpha^{255} \cdot \alpha^{-i} = \alpha^{255-i}$$

Reduction of the exponent modulo $2^8 - 1$ can be omitted, because $0 \leq i \leq 254$. This yields the following look-up method for inversion:

$$A^{-1} = \mathbf{atable}[255 - \mathbf{ltable}[\bar{a}_i]]$$

Squaring in \mathbb{F}_{2^8} : Examining the relationship

$$A^2 = (\alpha^i)^2 = \alpha^{(2i) \bmod 2^8-1}$$

the squaring can be implemented as follows:

$$A^2 = \mathbf{atable}[(2 \cdot \mathbf{ltable}[\bar{a}_i]) \bmod 255]$$

Multiplication by λ in \mathbb{F}_{2^8} : Recall from Section 3.6.1 that the second level of the tower was constructed using a polynomial with the constant term λ ; we can easily imagine that multiplication of a field element $A = \alpha^i \in \mathbb{F}_{2^8}$ with $\lambda = \alpha^{11} \in \mathbb{F}_{2^8}$ will be required at the top level of the tower. Relationship

$$\lambda \cdot A = \alpha^{11} \cdot \alpha^i = \alpha^{(11+i) \bmod 2^8-1}$$

dictates the following implementation:

$$\lambda \cdot A = \mathbf{atable}[(11 + \mathbf{ltable}[\bar{a}_i]) \bmod 255]$$

The primitive root α can only represent the elements of the multiplicative group $\mathbb{F}_{2^8}^*$. To keep the table access simple, we add the element at the beginning of **ltable**: this way, **ltable** is accessible by \bar{a}_i directly (otherwise we would have to decrement the index \bar{a}_i before accessing memory). Similarly, we add the remaining element 0 at the end of **atable**. Note that these two elements are never actually accessed.

Each of the two tables stores 256 8-bit values, requiring a total memory of 512 bytes.

Xilinx FPGA implementation options

On a Xilinx FPGA, there are three options for the implementation of the look-up tables:

- *logic only,*
- *distributed RAM, and*
- *block RAM (BRAM).*

Module	FPGA Results		
	#LUTs	#Slices	t [ns]
M_8 <i>logic only</i>	124	37	15.227
M_{8d} <i>distributed RAM</i>	132	39	17.000
M_{8b} <i>block RAM</i>	31	12	9.806

Table 4.30: Multipliers M_8 , M_{8d} and M_{8b} - implementation results

We explore these variations on the multiplier block M_8 , since it is the most demanding one in terms of table look-ups: it needs to access the `ltable` twice and `atable` once, reading 3 8-bit words of memory all together. The implementation results of the multipliers M_8 - *logic only*, M_{8d} - *distributed RAM* and M_{8b} - *block RAM* are given in Table 4.30.

■ **Remark:** *Detailed description of the three implementation options*

Multiplier M_8 - implementation with logic only: Let \bar{t} denote the 8-bit value stored at `ltable` $[\bar{a}_i]$. In this case, the tables are stored in SLICEL LUTs, for details refer to Section A.3 in Appendix. To read a single bit \bar{t}_i of memory, all four slice LUTs are being used: 6 bits of \bar{a}_i are used as LUT inputs, then two and two LUTs are connected together via the F7AMUX and F7BMUX respectively, with one of the unused bits of \bar{a}_i as the control signal for the multiplexers, and finally, the two outputs of F7AMUX and F7BMUX are connected to the F8MUX that is controlled by the last remaining bit of \bar{a}_i .

All four slice LUTs are used for one bit of \bar{t} , for the entire \bar{t} we need 32 LUTs, and for all three 8-bit memory words 96 LUTs. The remaining LUTs in M_8 module are used for modular addition.

Multiplier M_{8d} - implementation with distributed RAM: SLICEM LUTs can be used as normal function generators, as SRLs (see Sections A and C in Appendix for details) or as distributed RAM. For this implementation, we use a dual-port RAM with one synchronous write port (even though it is written only when initialized) and two asynchronous read ports for `ltable` and a single-port RAM with synchronous write and asynchronous read for `atable`, and set the attribute `ram_style` "distributed" for both RAMs.

Multiplier M_{8b} - implementation with block RAM: Yet another memory is available on the Spartan-6: special block RAM that is organized into special columns at the borders of the FPGA. Both, read and write operation are synchronous. Again, we implement a dual-port RAM for the `ltable` and a single-port RAM for the `atable`, both with the attribute `ram_style` set to "block". ■

Comparing the three multipliers, we find the block RAM module M_8b to give best results. But since it is our intention to compare this `WGP.T` module with the ones from previous tower constructions, we do not want to force the use of BRAM. We decide to build the remaining modules using just combinational logic and let the synthesis tools infer distributed or block RAM when possible. Implementation results of the modules for \mathbb{F}_{2^8} arithmetic are summarized in Table 4.31 below. Note the bigger area of the multiplier M_8 , which is the result of three table look-ups that are required for the multiplication, while all other operations need only two look-ups and hence only two tables.

Basic Building Block	FPGA Results		
	# of LUTs	# of Slices	t [ns]
M_8	124	37	15.227
I_8	72	21	10.728
S_8	76	21	12.891
M_λ	79	26	12.546

Table 4.31: Basic building blocks for arithmetic in \mathbb{F}_{2^8} - implementation results

Arithmetic in $\mathbb{F}_{(2^8)^2}$

Multiplication in $\mathbb{F}_{(2^8)^2}$: Let $A = a_0\beta + a_1\beta^{256}$ and $B = b_0\beta + b_1\beta^{256}$ be two elements in $\mathbb{F}_{(2^8)^2}$, with coefficients $a_0, a_1, b_0, b_1 \in \mathbb{F}_{2^8}$. Using the relationships $\beta^2 + \beta + \lambda = 0$ and $\beta + \beta^{256}$, we can compute the product AB as follows:

$$\begin{aligned}
AB &= (a_0\beta + a_1\beta^{256})(b_0\beta + b_1\beta^{256}) \\
&= a_0b_0\beta^2 + (a_0b_1 + a_1b_0)\beta^{257} + a_1b_1\beta^{512} \\
&= a_0b_0((1 + \lambda)\beta + \lambda\beta^{256}) + (a_0b_1 + a_1b_0)(\lambda\beta + \lambda\beta^{256}) + a_1b_1(\lambda\beta + (1 + \lambda)\beta^{256}) \\
&= ((a_0 + a_1)(b_0 + b_1)\lambda + a_0b_0)\beta + ((a_0 + a_1)(b_0 + b_1)\lambda + a_1b_1)\beta^{256}
\end{aligned}$$

The multiplication block M_{16} is shown in Figure 4.45. Please note the similarity of this block to the M_{16} from tower construction $\mathbb{F}_{(((2^2)^2)^2)^2}$ that can be seen in Figure 4.19(b): the two circuits are virtually the same apart from the multiplication with the constant, that is λ in case of $\mathbb{F}_{(2^8)^2}$ and μ in case of $\mathbb{F}_{(((2^2)^2)^2)^2}$. The submodules M_8 are of course different as well.

Inversion in $\mathbb{F}_{(2^8)^2}$: Just as multiplication, inversion is also very similar to inversion in $\mathbb{F}_{(((2^2)^2)^2)^2}$, (4.20). We begin the computation of inverse of $A = a_0\beta + a_1\beta^{256}$ by computing

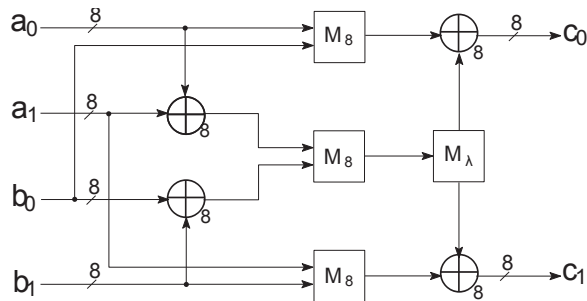


Figure 4.45: Multiplication block M_{16} in $\mathbb{F}_{(2^8)^2}$

the Frobenius mapping of A with respect to \mathbb{F}_{2^8} :

$$A^{2^8} = (a_0\beta + a_1\beta^{256})^{256} = a_0\beta^{256} + a_1\beta^{65536} = a_1\beta + a_0\beta^{256}.$$

The inverse of A is computed using the Itoh-Tsuji algorithm, described in Section 4.9, as follows:

$$A^{-1} = D^{-1} \cdot A^{256} = D^{-1}(a_1\beta + a_0\beta^{256}) = (a_1D^{-1}\beta + a_0D^{-1}\beta^{256}) = i_0\beta + i_1\beta^{256},$$

where D^{-1} for $D = (A^{257})$ can be computed with subfield $\mathbb{F}_{((2^2)^2)^2}$ inversion block I_8 . Using the Frobenius mapping of A the expression for D simplifies as follows:

$$\begin{aligned} D &= A^{257} = A \cdot A^{2^8} \\ &= (a_0\beta + a_1\beta^{256})(a_1\beta + a_0\beta^{256}) \\ &= a_0a_1 + (a_0 + a_1)^2\lambda \end{aligned}$$

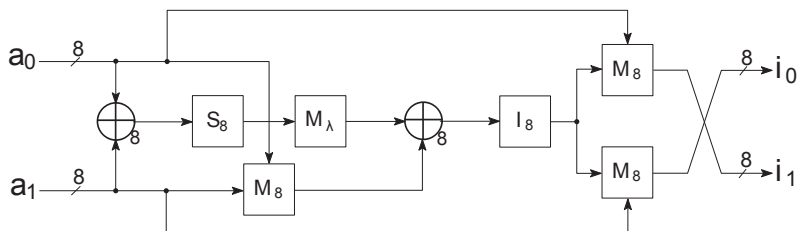


Figure 4.46: Inversion block I_{16} in $\mathbb{F}_{(2^8)^2}$

Implementation results of basic building blocks are given in Table 4.32. Based on comparison of *logic only*, *distributed RAM* and *block RAM* variants of module M_8 , all basic building blocks below use the *logic only* implementation of table look-ups.

Basic Building Block	FPGA Results		
	# of LUTs	# of Slices	t [ns]
M ₈	124	14	15.227
I ₈	72	21	10.728
M ₁₆	463	141	26.437
I ₁₆	610	185	40.144

Table 4.32: Basic building blocks for arithmetic in tower field $\mathbb{F}_{(2^8)^2}$ - implementation results

We can observe a poor performance of basic building blocks for the tower construction $\mathbb{F}_{(2^8)^2}$ in comparison to the basic building blocks for tower constructions $\mathbb{F}_{(((2^2)^2)^2)^2}$ and $\mathbb{F}_{(2^4)^4}$. The straight-forward table look-up based methods seem to be inconvenient for hardware implementation, the benefits in software, described in literature (see Section 2.5.4), seem to be lost.

4.6.2 Module WGP_T - Design of Pipelined Architecture

In Section 3.6.3, we described the circuit for WGP_T, that is almost identical to the top level module for $\mathbb{F}_{(2^4)^4}$ implementation shown in Figure 3.11, with NOT operator replaced by XOR with element x “8080”, and a different expression for the trace function. However, different basic building blocks in the two towerings affect the decisions about the number of pipeline stages. Again, it is only logical to pipeline at a finer granularity, that is at the M₈/I₈ level. The pipelined architecture, resulting in 15 pipeline stages, can be seen in Figure 4.44.

Note that since the basic building blocks at the top level of the tower are very similar for $\mathbb{F}_{(((2^2)^2)^2)^2}$ and for $\mathbb{F}_{(2^8)^2}$, the actual pipelining of WGP_T for $\mathbb{F}_{(2^8)^2}$ looks similar to pipelining for the tower construction $\mathbb{F}_{(((2^2)^2)^2)^2}$ as well. The implementation results for the $\mathbb{F}_{(2^8)^2}$ WGP_T are listed in Table 4.33

For reasons discussed in previous Section, we do not subject the WGP_T module to any constraints regarding the implementation of table look-up: we use the modules described as *logic only*, and let the Xilinx-ISE optimize the module by inferring distributed RAM and block RAM when possible. There are 21 multiplication blocks M₈, 7 blocks for multiplication with constant M_λ, one squarer and one inverter in WGP_T, and since three look-ups

Module	FPGA Results			
	#FFs	#LUTs	#Slices	t [ns]
WGP_T_M8_I8_T3	693	3013	932	11.936

Table 4.33: Module WGP_T_M8_I8_T3 using tower construction $\mathbb{F}_{(2^8)^2}$ pipelined at M_8/I_8 level - implementation results

are needed per multiplier and two look-ups per any other block, there are 81 268x8-bit look-up tables in WGP_T . The synthesis tools chose to implement 25 of total 30 tables **atable** in *block RAM* and the remaining 56 tables *distributed RAM*. The choice seems logical, since the **ltable** outputs (i.e. the exponents) are always manipulated with, while the **atable** outputs serve as module outputs as well. The remaining 5 tables **atable** were implemented as *distributed RAM* instead of *block RAM*, because there was no more *block RAM* available.

4.7 Summary of implementations

We explored five different field constructions that lead to seven `WGP_T` pipelines; their FPGA implementation results are collected in Table 4.34. Table 4.34 is divided into two parts. The upper part of the table contains the FPGA results for the `WGP_T` modules and the computed $\frac{T}{A^2}$, followed by some additional information about the pipelines, such as the total number of multipliers in `WGP_T`, the level at which the multipliers and the inverters were pipelined, and the depth of the resulting pipeline. In the bottom part of Table 4.34 we show the implementation results for the two most important building blocks, the inverter and the multiplier. The results are given for building blocks that correspond to the level of pipelining, for example if the pipelining was done at the M_4 level, the “biggest” multiplier that was used atomically within a pipeline stage is the multiplier M_4 , whose results are shown in the table. In the following discussion we grouped the modules based on their performance, i.e. the clock period.

In the first two rows we can see the module `WGP_T_PB` using polynomial basis representation of elements (Section 4.2) and the module `WGP_T_NB` using normal basis representation of elements (Section 4.3). Both finite fields were constructed as extensions of degree 16 over the prime field so we chose to pipeline the modules at the M_{16} level. The tower construction $\mathbb{F}_{((2^2)^2)^2}$ offers many pipelining possibilities, and we present the results for three different versions, pipelined at different levels. All three of them have a short clock period and the smallest area on the FPGA. The first two modules `WGP_T_A16_2_BC8` and `WGP_T_A8_2_BC8` have a similar structure and comparable performance, and will be discussed together. The third $\mathbb{F}_{((2^2)^2)^2}$ design `WGP_T_A4_2_BC4`, pipelined at the M_4/I_4 level of the tower, will be discussed together with the module `WGP_T_M4_I4_T2` that was pipelined at the same granularity and has practically the same clock period. Finally we discuss the module `WGP_T_M8_I8_T3`, pipelined at the M_8/I_8 level, that uses table look-ups for the arithmetic operations at the first level of the tower $\mathbb{F}_{(2^8)^2}$.

	WGP_T module				$\frac{T}{A^2}$	# of mult.	pipeline	
	#FFs	#LUTs	#Slices	t [ns]			level	depth †
$\mathbb{F}_{2^{16}}$	WGP_T_PB	1827	603	7.438	3.7	6	M_{16}	12
$\mathbb{F}_{2^{16}}$	WGP_T_NB	3835	1168	7.576	0.9	6	M_{16}	11
$\mathbb{F}_{(((2^2)^2)^2)^2}$	WGP_T_A16_2_BC8	1262	474	6.601	6.7	4	M_{16}/I_8	9/11 †
	WGP_T_A8_2_BC8	1195	436	6.519	8.1		M_8/I_8	11/13 †
	WGP_T_A4_2_BC4	1128	401	4.939	12.6		M_4/I_4	22/26 †
$\mathbb{F}_{(2^4)^4}$	WGP_T_M4_I4_T2	1722	589	4.918	5.8	6	M_4/I_4	20
$\mathbb{F}_{(2^8)^2}$	WGP_T_M8_I8_T3	3013	932	11.936	0.9	6	M_8/I_8	15

	WGP_T Inverter				WGP_T Multiplier			
	level	#LUTs	#Slices	t [ns]	level	#LUTs	#Slices	t [ns]
$\mathbb{F}_{2^{16}}$	WGP_T_PB	1054	369	7.271	M_{16}	119	46	11.812
$\mathbb{F}_{2^{16}}$	WGP_T_NB	1875	607	8.309	M_{16}	285	102	13.923
$\mathbb{F}_{(((2^2)^2)^2)^2}$	WGP_T_A16_2_BC8	41	15	12.915	M_{16}^*	148	52	13.925
	WGP_T_A8_2_BC8	41	15	12.915	M_8	40	14	10.613
	WGP_T_A4_2_BC4	2	2	6.984	M_4	11	5	8.517
$\mathbb{F}_{(2^4)^4}$	WGP_T_M4_I4_T2	2	2	6.861	M_4	10	4	7.781
$\mathbb{F}_{(2^8)^2}$	WGP_T_M8_I8_T3	72	21	10.728	M_8	124	14	15.227

Table 4.34: Summary of WGP_T modules for all five field constructions

- top: the FPGA results for the WGP_T modules
- bottom: the FPGA results for the inverters and multipliers used by the corresponding WGP_T modules
- † the values are given for the running/initialization phase
 - * the number of interstage registers in the pipelined square and multiply inversion
 - ★ the multipliers in the decimation part were pipelined at M_{16} and the two reused multipliers at M_8 level

4.7.1 The WGP_T_PB and the WGP_T_NB

Both designs use the square and multiply inversion, based on equation (4.4), and were implemented with the same placement of interstage registers, with one exception: the first and the second stage were merged in the normal basis variant. This decision was based on the simplicity of the exponentiation of normal basis elements to the powers of two, which is cyclic shift of the corresponding vector. The inversion pipeline stages in module WGP_T_NB basically contain only M_{16} multipliers with shifted inputs. The exponentiation in polynomial basis is not as trivial as a cyclic shift, but still quite simple: it can be performed by a series of squarings. Due to a bit more complicated exponentiations, we can say the module WGP_T_PB was coarsely pipelined, containing an entire M_{16} multiplier accompanied by a squarer or a series of squarers inside a single pipeline stage. We could choose a pipeline where only M_{16} multipliers are contained within a stage, but that would add 7 stages to the pipeline. We wanted to keep the WGP_T_PB and WGP_T_NB pipelines as similar as possible to get a better intuition about how the underlying arithmetic affects the overall design. The two WGP_T modules achieve an almost identical clock period. However, due to an approximately two times bigger normal basis multiplier, the size of the WGP_T_NB is also roughly twice the size of WGP_T_PB.

4.7.2 The WGP_T_A16_2_BC8 and WGP_T_A8_2_BC8

Due to the algebraic optimization (Section 3.4.3) that was able to remove two multipliers, we consider the circuit WGP_T in Figure 3.10 to be the most promising one, hence the most effort was spent for its optimization. A thorough discussion comparing the three modules, pipelined at different granularities, can be found in Section 4.4.5. The pipelining for WGP_T was chosen to avoid the use of I_{16} inverter and was initially targeting for a pipeline of approximately the same depth as the pipelines for the polynomial and the normal basis modules. The multipliers were pipelined in the following way: in module WGP_T_A16_2_BC8, the two decimation M_{16} s were kept complete, and remaining two multipliers pipelined at the M_8 level; the second module WGP_T_A8_2_BC8 is similar, with the decimation multipliers pipelined at the M_8 level as well. These two modules achieve practically the same clock period, and surprisingly, the number of slices needed for the longer pipeline drops. However, when the corresponding WG-16 modules were implemented, the longer pipeline also resulted in the larger area, and both FPGA and ASIC results (see Table 4.25) are clearly in favor of module WGP_T_A16_2_BC8, that also has a shorter pipeline and hence a shorter initialization phase.

4.7.3 The WGP_T_A4_2_BC4 and WGP_T_M4_I4_T2

Considering the behavior of the two $\mathbb{F}_{((2^2)^2)^2}$ pipelines discussed above (Section 4.7.2), the comparison of the two WGP_T modules pipelined at the M_4/I_4 level leads to a very important conclusion: pipelining at a lower level of the tower field will reduce the clock period (and of course increase the area and the depth of the pipeline), and the true benefit of the $\mathbb{F}_{((2^2)^2)^2}$ tower construction lies in the trace property, that removes two multipliers and thus causes an adequate area reduction.

The third $\mathbb{F}_{((2^2)^2)^2}$ module WGP_T_A4_2_BC4 was pipelined at the M_4/I_4 level and resulted in a very long pipeline; but also a very short clock period and small number of used slices. The WGP_T_M4_I4_T2 module was obtained using the $\mathbb{F}_{(2^4)^4}$ tower construction. It is also pipelined at the M_4/I_4 level, it has a shorter pipeline and slightly shorter clock period, but needs 188 slices more, and so exhibits a smaller $\frac{T}{A^2}$. Taking a closer look at the two I_4 inverters and the two M_4 multipliers, we find the following: the building blocks obtained with $\mathbb{F}_{(2^4)^4}$ tower construction have a shorter period and a smaller (M_4) or comparable (I_4) area. The $\mathbb{F}_{(2^4)^4}$ module needs 6 multipliers, while the $\mathbb{F}_{((2^2)^2)^2}$ module WGP_T_A4_2_BC4 only needs 4 and the inversion I_{16} is smaller for the tower construction $\mathbb{F}_{((2^2)^2)^2}$.

We can now declare the $\mathbb{F}_{((2^2)^2)^2}$ tower construction to be the most appropriate for the implementation of WG-16.

4.7.4 The WGP_T_M8_I8_T3

Our final pipeline is the module WGP_T_M8_I8_T3 that was obtained for the tower construction $\mathbb{F}_{(2^8)^2}$. This module uses table look-up algorithms for the lower level arithmetic. Despite different optimization attempts by the synthesis tools, we find a large area and a slow clock period. We cannot say that the ground field \mathbb{F}_{2^8} is too large for a table look-up design, but the WGP_T needs too many tables, namely three tables for multiplication and two tables for all other operations, and since we are using a pipelined design, each operation needs its own set of look-up tables. The block RAM is used up and the use of distributed RAM increases the clock period. Note that these kinds of optimizations are FPGA specific.

4.7.5 Optimality analysis

We have already established that the three $\mathbb{F}_{((2^2)^2)^2}$ modules result in the smallest area and have short clock periods. The low area cost of these modules is a direct consequence of the reduced number of multipliers. The metric $\frac{T}{A^2}$ was used to assess the optimality of the `WGP_T` modules, and it clearly favors the M_4/I_4 level $\mathbb{F}_{((2^2)^2)^2}$ module `WGP_T_A4_2_BC4`. From performance point of view, the $\mathbb{F}_{(2^4)^4}$ module `WGP_T_M4_I4_T2` is a competitive design. However, the $\frac{T}{A^2}$ metric ranks all three $\mathbb{F}_{((2^2)^2)^2}$ modules above the $\mathbb{F}_{(2^4)^4}$ design; a clear testimony to the importance of the area reduction facilitated by the $\mathbb{F}_{((2^2)^2)^2}$ tower construction.

Based on the optimality, the polynomial basis design `WGP_T_PB` is next in line. The polynomial basis M_{16} multiplier we used is a simple multiplication followed by the reduction, but based on implementation results, this is the smallest M_{16} with the shortest delay among the five top-level multipliers. Further exploration could lead to a smaller and faster multiplier, and potentially a better overall `WGP_T` design. A polynomial basis approach using an optimized polynomial basis multiplier was used in [13]. They also rearranged the exponents in the permutation polynomial $q(Y)$ to avoid a direct computation of inverse, thus eliminating three multipliers that are needed in our design. Their ASIC implementation results show a fast pipelined module with a smaller area.

The performance of our normal basis implementation `WGP_T_NB` might be comparable with our polynomial basis design `WGP_T_PB`, but it needs twice the area. The $\frac{T}{A^2}$ metric places it at the bottom of our optimality scale, together with the `WGP_T_M8_I8_T3` module, which is not only area-demanding but also very slow due to table look-ups.

4.7.6 The LFSR and the FSM

Implementation of the LFSR is straightforward and the FPGA implementation results are listed in Table 4.2 in Section 4.1. With 47 slices and a clock period just above 3 ns, this module is not critical in any way. For the selected target FPGA, the Xilinx-ISE tools perform an optimization: instead of implementing a series of registers between two tap positions, special LUTs, capable of implementing a shift register, were used. The optimized version uses less than 30% of the original number of FFs needed. This optimization is not possible for an ASIC implementation, where we expect to find all 512 FFs.

The FSM was implemented only for the $\mathbb{F}_{((2^2)^2)^2}$ modules, and the same FSM (with different parameters) was used. The FSM is simple, small and fast.

4.7.7 The WG-16

Based on the FPGA results in Table 4.34, we conclude that the $\mathbb{F}_{((2^2)^2)^2}$ tower construction is the most beneficial choice for WGP_T. Three WG-16 modules, using the three pipelines given by the $\mathbb{F}_{((2^2)^2)^2}$ tower construction, were implemented (Section 4.4.5). The FPGA and ASIC implementation results are listed in Table 4.25 and are followed by an in-depth discussion and comparison of the three modules. Please note the difference between the FPGA and the ASIC results for the WG-16 that were implemented using the three $\mathbb{F}_{((2^2)^2)^2}$ pipelines, revealing the impact of the large number of registers on the area used by the ASIC implementation (see Table 4.25). Consequently, the same $\frac{T}{A^2}$ metric using the results of the ASIC implementation points to a different choice, namely the module WGP_T_A16_2_BC8. Comparing the FPGA results of the three WG-16 modules, the WGP_T module WGP_T_A4_2_BC4 remains the best overall design.

Taking a closer look at the results of the three WG-16 modules (both the FPGA and the ASIC implementation results), we can observe the effects of the level of pipelining: moving to a finer granularity, which corresponds to descending to a lower level of the $\mathbb{F}_{((2^2)^2)^2}$ tower field, reflects in a decreased clock period and increased area cost.

Comparison with other stream ciphers

In Chapter 2 we presented some other stream ciphers in two separate sections. In Section 2.5.2 we covered two stream ciphers used in 3GPP confidentiality and integrity algorithms, Snow3G and ZUC, and in Section 2.5.3 two ciphers, that were included in the eSTREAM portfolio, namely Grain and Trivium. We do not attempt to compare WG-16 to the eSTREAM candidates, because WGT-16(X^d) 16 is not intended for constrained environments. There are other members of WG family, that were designed for such applications (and hence comparable with Grain and Trivium), for example WG-5 [9] or WG-8 [11].

WG-16 based confidentiality and integrity algorithms were proposed in [8], but since both Snow3G and ZUC are word-oriented stream ciphers, producing a 32-bit keyword per cycle, their comparison with the bit-oriented WG-16 is difficult. Besides the high throughput, which is a direct consequence of a 32-bit keyword produced each clock cycle, that can be

observed in Table 2.3, another thing immediately draws our attention: the device chosen for the implementation of Snow3G and ZUC was, in most cases, a Virtex-5 FPGA. Both, Virtex-5 and Spartan-6 have 6-input/2-output LUTs, so we can compare the designs in terms of area (ignoring the $\mathbb{F}_{(2^8)^2}$ design since Spartan-6 can not compare to Virtex-5 in term of memory resources). Also, Virtex devices are in general always faster than the same generation Spartan, and the Virtex-5 is still faster than Spartan-6. However, WG stream ciphers also have provable randomness and cryptographic properties.

Chapter 5

Conclusion and future work

In Chapter 3 we presented five isomorphic field constructions for $\mathbb{F}_{2^{16}}$. The first construction we explored uses the defining polynomial of $\mathbb{F}_{2^{16}}$, which is given with the specification of WG-16, and polynomial basis representation of the field elements. Next we used the normal element, yielding the multiplication matrix with the smallest Hamming weight, for the normal basis representation of $\mathbb{F}_{2^{16}}$ elements. For the three composite fields, we had to find an appropriate irreducible polynomial for each extension. For the lower levels of towers of extensions, we relied on pen-and-paper methods based on the theoretical background from Section 2.2. As the order of the extension fields grew, we used the computer algebra system GAP, that was also used to conduct the exhaustive search for best conversion matrices for each tower field.

We encounter three different ways of raising the field element to powers of two, namely the series of squarings when polynomial basis was used, a simple right cyclic shift when normal basis was used, and transition to normal basis representation followed by a shift and a transition back to the tower field representation. The five field constructions also give different representations of the element 1. The trace function may be independent of the basis, but when taking the basis into account, simplified expressions, resulting in simple hardware, were found. For tower construction $\mathbb{F}_{((2^2)^2)^2}$ some interesting properties exist that allowed us to remove two multipliers. At the end of Chapter 3 we summarized the five top-level circuits obtained with different field constructions into two groups: the $\mathbb{F}_{((2^2)^2)^2}$ WGP_T module with 4 multipliers (Figure 3.12) and the WGP_T module with 6 multipliers for all other field constructions (Figure 3.13).

In Chapter 3 we treated the basic building blocks, that is the submodules implementing the basic finite field arithmetic, as black boxes and focused on the top-level architecture for **WGP_T**. In Chapter 4, we discuss the algorithms for the field arithmetic for each field construction individually; the algorithms are closely dependent on the basis that is used to represent the field elements. Based on the field construction and on FPGA results of the basic building blocks we made decisions about the pipelining: how many stages, where to insert the stage borders etc. The FPGA implementation results for the basic building blocks and finally for the four **WGP_T** modules that share the top-level architecture show the following: the differences in the implementation results between the four **WGP_T** modules, sharing the same top-level architecture, do not lie in the method of exponentiation to powers of two, representation of element 1 or in trace computation, but in the basic building blocks and in the level of pipelining; needless to say, the differences are enormous, and the structural similarity of these four modules is lost with the actual field arithmetic.

The tower construction $\mathbb{F}_{((2^2)^2)^2}$ is also highly regular, giving very similar basic building blocks that differ only in the width of the operands and gates, at each level of the tower. For the tower construction $\mathbb{F}_{((2^2)^2)^2}$, which offers many pipelining possibilities, three different **WGP_T** modules, pipelined at different levels were implemented, and pipelining at a lower level of the tower field reduces the clock period. Since the $\mathbb{F}_{((2^2)^2)^2}$ based **WGP_T** modules only need four multipliers, they also have the smallest area cost. A **WG-16** module was implemented for each of the three $\mathbb{F}_{((2^2)^2)^2}$ **WGP_T** pipelines. Two **WGP_T** modules were chosen, the module pipelined at the M_4/I_4 level for the FPGA implementations and the module pipelined at the M_{16}/I_8 level for ASIC implementations. The low level of pipelining means more registers and less space for optimizations within the stage. With FPGAs, registers are so to say free, already there. But in ASIC, pipelining at a finer granularity comes at the cost of area increase, and the M_4/I_4 option is no longer the best overall design, although it stays the fastest design and could be preferred when area and power consumption are not critical.

The presented work is an exploration of the design space for **WG-16**. Note that **WG-16** is very complex, but it benefits from the tower construction $\mathbb{F}_{((2^2)^2)^2}$ in three ways: by the existence of efficient basic building blocks for arithmetic in $\mathbb{F}_{((2^2)^2)^2}$, by having several options for the level of pipelining and by the algebraic optimization, which reduces the number of multipliers needed.

Another extremely important task is a survey of existing optimized hardware multipliers and inverters for finite fields using polynomial and normal basis representation of field

elements. Surely we did not exhaust all possibilities for $\mathbb{F}_{2^{16}}$. For example an interesting choice would be the tower construction $\mathbb{F}_{(2^8)^2}$ with the optimal dual basis representation for the \mathbb{F}_{2^8} elements. For different finite fields \mathbb{F}_{2^m} , with a composite $m \geq 14$ other bases, could prove advantageous.

Appendix A

Xilinx Spartan-6 FPGA

A.1 Basic structure

For this thesis, a Xilinx Spartan-6 FPGA was chosen (xc6s1x9-csg324). Therefore, we will describe some general FPGA features in terms of Spartan-6 family.

FPGAs are composed of a large number of Configurable Logic Blocks (CLBs), that are the basic building blocks of the circuit. CLBs are organized into a matrix, surrounded by special Input/Output Blocks (IOBs), interwoven with configurable interconnects that convey signals between CLBs and between CLBs and IOBs. [53, 58]. The basic structure of an FPGA device is depicted in Figure A.1.

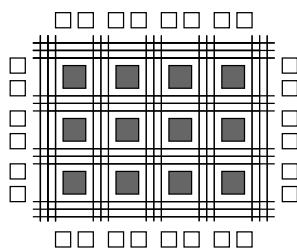


Figure A.1: Basic structure of an FPGA:
CLBs - the large grey blocks,
IOBs - smaller white blocks,
vertical and horizontal interconnects

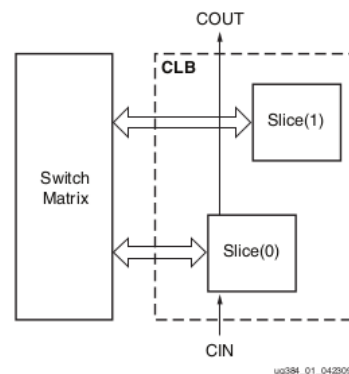


Figure A.2: Arrangement of slices within the CLB [60]

A.1.1 CLB - Configurable Logic Block

Spartan-6 CLB contains two (similar) slices (Figure A.2). A slice contains 4 LUTs (Look-up Tables), also called function generators, storage elements and multiplexers to control memory inputs and slice outputs. Figure A.3 shows the diagram of the basic slice called SLICEX, which is contained in every CLB. The second slice in the CLB is either SLICEM or SLICEL. They have the basic structure of SLICEX, but contain additional logic. Both SLICEM and SLICEL contain wide-function multiplexers and dedicated carry logic to perform fast arithmetic addition and subtraction. The two slices inside the CLB are not directly connected, but the carry structure connects SLICEM/SLICEL from neighboring CLBs vertically upwards, as can be seen in Figure A.2. SLICEM LUTs are modified (with additional data inputs and write enable) in a way that allows the LUTs to be used as 64-bit distributed RAM or variable-length shift registers.

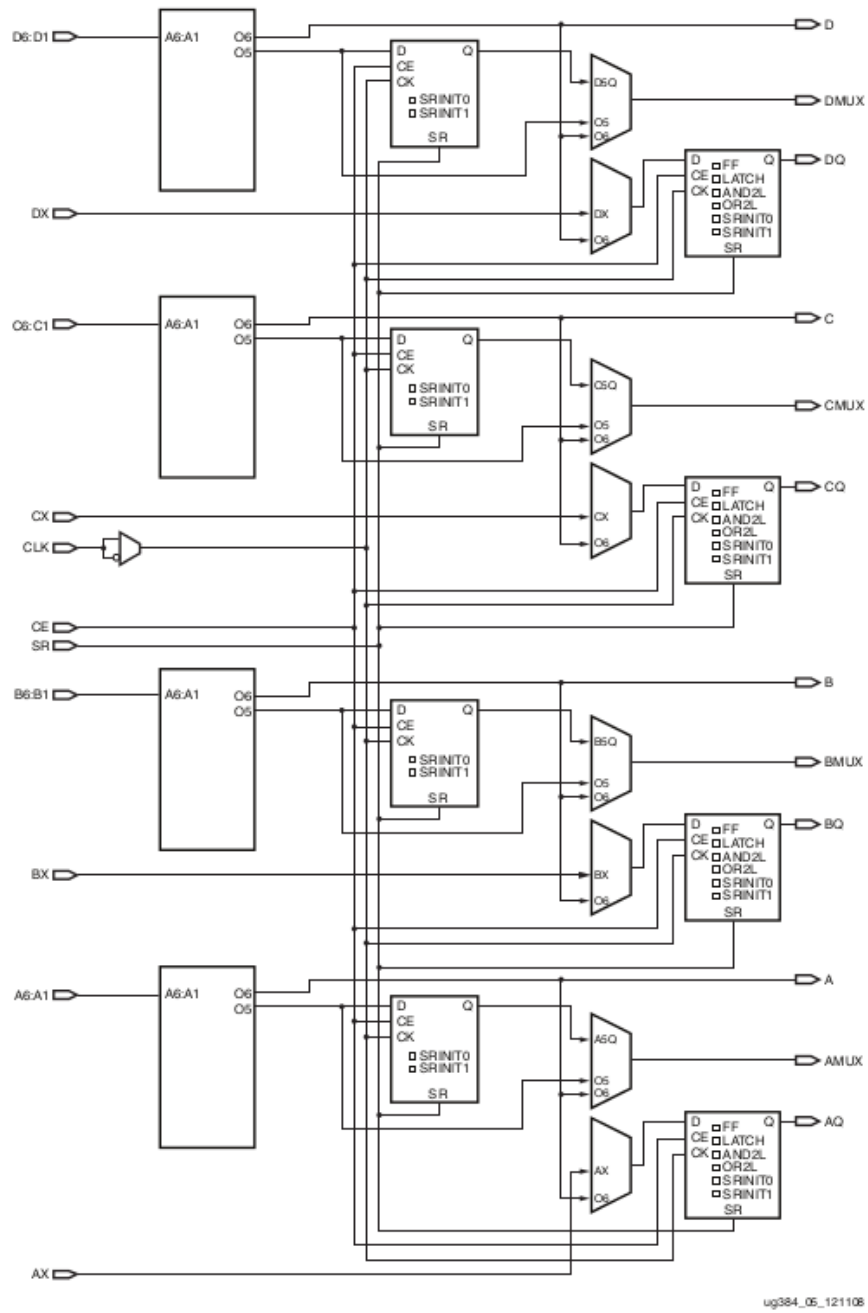


Figure A.3: Diagram of SLICEX [60]

LUT

The n -input/1-output LUT is a 2^n -word memory array that holds the truth Table of the desired n -input Boolean function. The LUT inputs are address signals that choose the appropriate word whose content is transferred to the LUT output. Spartan-6 FPGAs have 6-input/2-output LUTs (the two outputs are denoted O_6 and O_5). A single LUT can implement one 6-input Boolean function, whose output is available on O_6 or two 5-input Boolean functions with outputs O_6 and O_5 (the two functions must operate on the same input values). The LUT outputs can be:

- connected directly to the slice output (both O_6 and O_5)
- used in additional logic (both O_6 and O_5)
- connected as input to a memory element (O_6 only)
- connected as input of one of the three wide-function multiplexers for realization of 7 or 8-input Boolean functions (O_6 in SLICEM/SLICEL only)

Figure A.4 shows how the two wide-function multiplexers F7AMUX and F7BMUX facilitate the realization of a 7-input Boolean function by choosing an output from one of the two LUTs connected to them. For an 8-input Boolean function an additional multiplexer F8MUX, that connects all four LUTs is available. Boolean functions in more than 8 variables can be implemented using several slices [60].

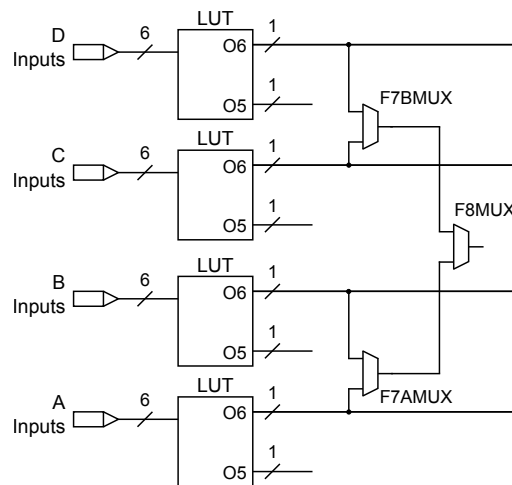


Figure A.4: Realization of 7 or 8-input Boolean functions using multiple slice LUTs

Storage elements

Two storage elements belong to each LUT in the slice, one for each LUT output [60]. The storage element driven by the LUT output O_6 can be configured either as a D-type flipflop (DFF) or a latch (Note: this storage element actually gets its input from a multiplexer that chooses between O_6 and the direct slice input). The second storage element takes O_5 LUT output; it operates as a DFF and can only be used when the first storage element is configured as DFF.

Throughout this work, whenever we speak of flipflops or registers in an FPGA design, we refer to D-type flipflops.

A.1.2 IOB - Input/Output Block

As seen in Figure A.1, the matrix of CLBs is surrounded by IOBs. They provide configuration of pins as inputs or outputs and include storage elements to accommodate signal delay and serial-to-parallel/parallel-to-serial converters[62]. It is also possible to control output strength and slew rate, they provide on-chip termination and can be configured to a variety of I/O standards. The chosen `xc6s1x9-csg324` FPGA device provides 200 IO pins that can be configured by the user.

A.1.3 Interconnects

Interconnect is a programmable network of vertical and horizontal routing channels, composed of the connection segments of different lengths and pass transistors to enable inputs and outputs of of CLBs and IOBs [58]. Using segments of different lengths reduces the latency of a particular data-path and this facilitates optimal connectivity. Global signals and clock signals use longlines, that do not have any switches that would slow down the signal. So called fast interconnects are used to route outputs back to inputs, single interconnects are used to connect neighboring blocks and “diagonal” connections are achieved with double and quad interconnects, as can be seen in Figure A.5. The vertical and horizontal segments are connected through so called switchboxes composed of SRAM cells (Figure A.6(a)); if the SRAM cell holds the value '1', the switch between two segments is closed and the connection established [61]. Figure A.6 (b) shows the interconnects and switchboxes, the thick lines mark different established connections [50].

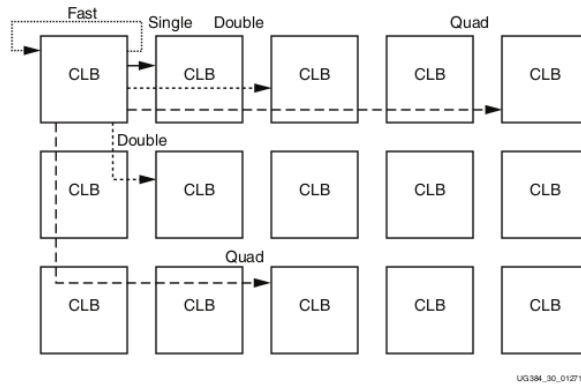


Figure A.5: Interconnect types [60]

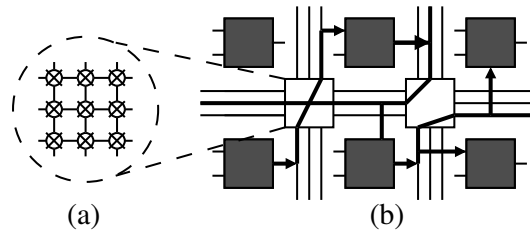


Figure A.6: Interconnects: (a) switchbox; (b) different connections between CLBs

A.2 FPGA design flow

A.2.1 Levels of abstraction

In order to explain the FPGA design flow we first need to introduce level of abstraction. There are different ways to make the distinction between the levels, depending on how much detail we want to include (for example, transistor level could be divided into two levels, the upper switch level and the actual transistor level). The hierarchy depicted in Figure A.7 is sufficient for this thesis.

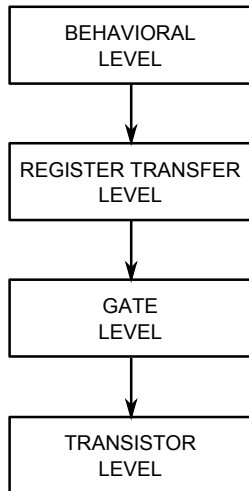


Figure A.7: Levels of abstraction

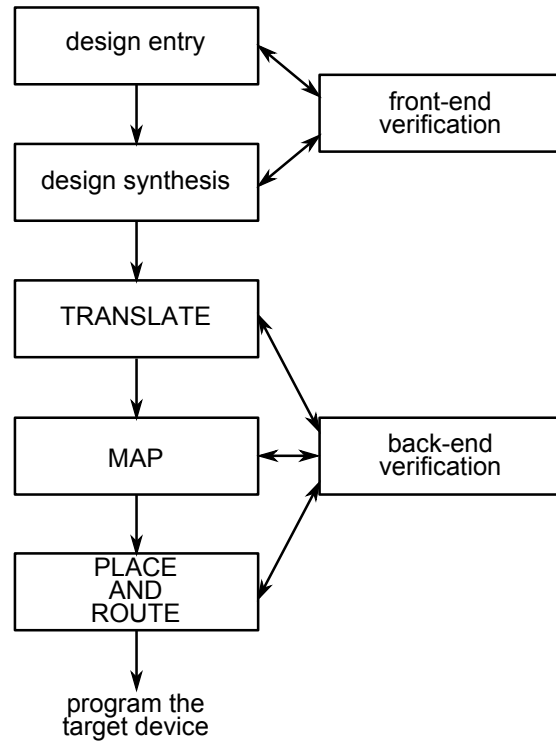


Figure A.8: Design flow

We will present the four levels of abstraction from a top-down approach, since this is more natural from the developers point of view. A good explanation of what an abstraction actually is was given in [66]: "An abstraction is a simplified model of the system, showing only the selected features and ignoring associated details. The purpose of an abstraction is to reduce the amount of data to a manageable level, so that only the critical information is presented."

As the name **behavioral level** indicates, this level describes the behavior of the system in terms of data-path and algorithm and is not concerned with details like registers, logic cells and connection between them. This leads directly to the next lower level of abstraction, the **register transfer level** (RTL). The line between behavioral level and RTL is thin and sometimes they are not distinguished at all. The basic building blocks of RTL are modules. A module is basically a black box with inputs and outputs. Signals pass through functional units, storage units and routing units, and are usually grouped together into more complex data types. So basically, RTL captures the architecture of the circuit in terms of registers and combinational signals and uses a common clock signal for the storage elements, i.e.

the events are synchronized to the rising/falling edge of the clock signal[66]. Beyond that, RTL is technology independent. Going to the next lower level, the **gate level**, we enter the binary world, i.e. all the signals are treated as logic '1' or logic '0'. Functionality of the circuit (input-output relationship) is described using Boolean functions. Building blocks are simple gates (**XOR**, **OR**, **NOT**, ...) and we are looking at a network of gates and registers from a certain library - we enter technology specific domain. Also, the technology specific propagation delay of a gate is known. Area complexity at this level is given with a technology independent measure called gate count, the basic gate usually being the two-input **NAND** gate. Gates are nothing else but circuits consisting of transistors. Including this detail, we proceed to the next level, **Transistor level** (sometimes called switch level), which is the lowest level. It is a network of capacitors and resistors, and a detailed layout of components and their interconnections is available. Here we enter the domain of continuous time, voltages and currents, and the behavior is described by differential equations [66].

A.2.2 Design flow

Programming an FPGA device begins with specifying the digital circuit (this step is usually called **design entry**, refer to Figure A.8). Here we have two options: describing its functionality using some HDL (Hardware Description language) or building the schematics of the circuit using a graphical interface and available circuit elements. In the thesis we are using VHDL (Very-High-Speed Integrated Circuit HDL), which results in a RTL description of the circuit. The next step is **design synthesis**. Synthesis generates a logic circuit from the VHDL description [66]. VHDL code is checked for possible syntax errors; if none are found, the compiler translates the VHDL to corresponding components, such as adders, LUTs, registers, finite state machines, etc. , and connects all signals. Note that all of the above is on a structural level and does not yet contain details about the target device. For a example, at this stage, a 3-input Boolean function will be mapped to a 3-input LUT, even though the target device only contains 6-input LUTs. Netlist file is generated during the process [54]. Synthesis tools, including Xilinx-ISE, will also perform optimizations, such as finding the canonical disjunctive normal form for a boolean function, at this step.

At this stage of the design, **front-end verification** should be carried out. This includes formal verification (i.e. formal proofs of the correctness of the design using mathematical methods), functional simulation and static timing analysis. Functional simulation intends to verify the behavioral and structural design. Behavioral simulation, just like the behavioral level of abstraction, will be the fastest one, but will not provide much information (for

example, no timing information can be obtained). Structural simulation needs more details and can be performed after the synthesis. It allows to verify the functional correctness and includes static timing (i.e. events are synchronized to rising/falling edge of the clock signal). Simulations at this stage are based on estimated parameters of the circuit[51]. More detailed simulations can be done in later stages of development cycle.

The first step in design implementation is **translate**. It combines all the netlist files and timing constraints into a single netlist that holds the entire design. It also identifies appropriate system library components. The output of this process is a gate level logical description of the entire design and preserves its original hierarchy [49].

Next step, called **map**, maps the logical design to the available resources on the target device (IOBs, CLBs, etc.). While grouping the gates into physical components different optimizations are performed. Unused signals and duplicated logic are removed. Another example of a task performed by MAP is decomposing an 8-input Boolean function and implementing using all the slice LUTs, their outputs connected via multiplexers, as was described in SectionA.1.1; MAP must make sure that the outcome is functionally equivalent to the original function. Optimizations must also consider different timing and area constraints. The output of this process is a mapped netlist file containing the physical representation of the design [49].

The actual “physical design” is the **place and route** (PAR) process. The placement process selects a location for each (logic) component from the mapped netlist file. To find the best placement of the components, the process is executed in several phases trying different locations for each component. Besides timing constraints, the placement also depends on the routing. Routing process establishes connections between the cells and can also change the placement if needed.

During implementation a back-end verification can be performed, with more accurate information about the design and the target device after each implementation step. For example, a detailed timing simulation with accurate estimates of block and routing delays can be performed after PAR.

If the design meets the specifications and performance goals, the programming file (bit stream) is generated and loaded onto the target device.

Appendix B

More detailed discussions and additional material on field constructions and module WGP_T

B.1 Tower construction $\mathbb{F}_{2^{16}} \cong \mathbb{F}_{((2^2)^2)^2}$

B.1.1 Extension field $\mathbb{F}_{2^4} \cong \mathbb{F}_{(2^2)^2}$

Extension of degree 4

Elements of \mathbb{F}_{2^4} , constructed as extension of degree 4 over the prime field using the irreducible polynomial $x^4 + x + 1$ and its root y , in their polynomial basis representation $\{1, y, y^2, y^3\}$ are given in Table B.1 below:

\mathbb{F}_{2^4} with defining polynomial $x^4 + x + 1$ in polynomial basis						
1	y	y^2	y^3			
a_0	a_1	a_2	a_3	polynomial	power of y	σ_1^2
0	0	0	0	0	/	0
0	0	0	1	y^3	y^3	y^{12}
0	0	1	0	y^2	y^2	y^8
0	0	1	1	$y^2 + y^3$	y^6	y^9
0	1	0	0	y	y	y^4
0	1	0	1	$y + y^3$	y^9	y^6
0	1	1	0	$y + y^2$	y^5	y^5
0	1	1	1	$y + y^2 + y^3$	y^{11}	y^{14}
1	0	0	0	1	y^{15}	1
1	0	0	1	$1 + y^3$	y^{14}	y^{11}
1	0	1	0	$1 + y^2$	y^8	y^2
1	0	1	1	$1 + y^2 + y^3$	y^{13}	y^7
1	1	0	0	$1 + y$	y^4	y
1	1	0	1	$1 + y + y^3$	y^7	y^{13}
1	1	1	0	$1 + y + y^2$	y^{10}	y^{10}
1	1	1	1	$1 + y + y^2 + y^3$	y^{12}	y^3

Table B.1: Elements of \mathbb{F}_{2^4} in polynomial basis $\{1, y, y^2, y^3\}$ and as powers of y

Conversion matrices between polynomial basis and tower field representation

The matrix M_P^T for transition from tower field representation to polynomial basis representation is obtained by simply rewriting the four tower field basis elements in polynomial basis $\{1, \beta, \beta^2, \beta^3\}$ as $t_i = \sum_{j=0}^3 t_{ij} \beta^j$, using Table B.1 with $y = \beta$, which yields vectors $t_i = (t_{i0}, t_{i1}, t_{i2}, t_{i3})$. These vectors are columns of transition matrix, that is t_i is the i -th column of M_P^T . Matrix M_T^P is obtained as inverse of M_P^T .

$$\begin{array}{ll} t_0 = \beta^6 = \beta^2 + \beta^3 & \mapsto (0, 0, 1, 1) \\ t_1 = \beta^{11} = \beta + \beta^2 + \beta^3 & \mapsto (0, 1, 1, 1) \\ t_2 = \beta^9 = \beta + \beta^3 & \mapsto (0, 1, 0, 1) \\ t_3 = \beta^{14} = 1 + \beta^3 & \mapsto (1, 0, 0, 1) \end{array} \quad M_P^T = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \quad M_T^P = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

An element represented in polynomial basis can be converted into tower field representation by multiplication $M_{PT} \cdot v_P = v_T$, where v_P denotes the vector in polynomial basis and v_T the vector of the element in tower field representation. For example $\beta^{12} = 1 + \beta + \beta^2 + \beta^3$ gives $v_P = (1, 1, 1, 1)$, so we have :

$$M_T^P \cdot v_P = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = v_T$$

Let us check that v_T indeed represents β^{12} :

$$\beta^6 + \beta^9 + \beta^{14} = \alpha\beta + \alpha\beta^4 + \alpha^2\beta^4 = \alpha\beta + \beta^4 = \beta^{12}$$

B.1.2 Efficient conversion matrices between normal basis and tower field representation of $\mathbb{F}_{((2^2)^2)^2}$

Here we give the matrix T and conversion matrices for three different normal elements, the first normal element found ω^{11} , the normal element giving lowest Hamming weight conversion matrices ω^{1091} , and the normal element ω^{1117} giving the matrix T with best C_N , but very high Hamming weight for the conversion matrices. The value $\mu = \beta + \lambda\gamma$ was used.

Normal element $\theta = \omega^{11}$:

Matrix T has $C_N = 123$, Hamming weight of both conversion matrices is 124.

B.2 Tower construction $\mathbb{F}_{2^{16}} \cong \mathbb{F}_{(2^4)^4}$

B.2.1 Different representations of the finite field with 16 elements and corresponding transition matrices

In the Section 3.4.1 we presented the finite field \mathbb{F}_{2^4} in following three ways:

- vi. \mathbb{F}_{2^4} as an extension field of degree 4 over \mathbb{F}_2 , using the defining polynomial $x^4 + x + 1$ with root y and polynomial basis $\{1, y, y^2, y^3\}$ - see Table B.1 (note that this is polynomial $e_1(x)$ from Table 3.13);
- vii. $\mathbb{F}_{(2^2)^2}$ as an extension of degree 2 over \mathbb{F}_{2^2} , using the defining polynomial $x^2 + x + \alpha$ with root β and normal basis $\{\beta, \beta^4\}$ - the elements of this field were originally represented in Table 3.8 and are now listed in column 8 in Table 3.14 in appropriate order as $A = b_0\beta + b_1\beta^4$;
- viii. \mathbb{F}_{2^4} over \mathbb{F}_2 with tower field basis resulting from aforementioned construction $\mathbb{F}_{(2^2)^2}$ with basis $\{\beta^6, \beta^{11}, \beta^9, \beta^{14}\}$. Field elements represented in this basis were originally represented in first column of Table 3.8 and are now listed in column 7 in Table 3.14 in appropriate order as $A = t_0\beta^6 + t_1\beta^{11} + t_2\beta^9 + t_3\beta^{14}$;

All five representations of \mathbb{F}_{2^4} are isomorphic to one another, they describe elements of one and only finite field of order 16. At this point we will take a closer look at the constructions that consider \mathbb{F}_{2^4} as an extension of degree 4 over \mathbb{F}_2 and will treat them as four dimensional vector spaces; for clarity we list the vector spaces and their bases in Table B.2.

Vector space	basis	†
V_{e_1}	$P_1 = \{1, y, y^2, y^3\}$ with y as root of $e_1(x)$	iii.
V_T	$T = \{\beta^6, \beta^{11}, \beta^9, \beta^{14}\}$ with $\beta = y$ as root of $e_1(x)$	v.
V_{e_4}	$P_4 = \{1, \alpha, \alpha^2, \alpha^3\}$ with α as root of AOP $e_4(x)$	i.
V_N	$N = \{\alpha, \alpha^2, \alpha^4, \alpha^3\}$ with α as root of AOP $e_4(x)$	ii.

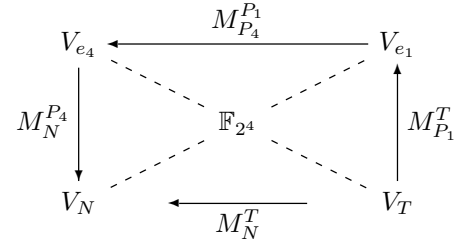
Table B.2: Different representations of \mathbb{F}_{2^4} over \mathbb{F}_2 viewed as vector spaces of dimension 4
 † reference to the field construction as listed in this section

To construct a transition matrix from V_N to V_T we start by finding a transition matrix between V_{e_1} and V_{e_4} , that is between the two polynomial basis representations. To achieve this, we take a generator of the finite field defined by AOP $e_4(x)$ and map it to generator

$y \in P_1$ of field defined by $e_1(x)$. Note that $\alpha \in P_2$ has order 5 and cannot generate the 15 elements of the multiplicative group, so we take an element of order 15, for example $\lambda = \alpha + \alpha^3$; this element generates the $\mathbb{F}_{2^4}^*$ obtained with AOP. Transition mapping is obtained as follows:

$$\begin{aligned} y &\mapsto \lambda = \alpha + \alpha^3 \\ y^2 &\mapsto \lambda^2 = \alpha + \alpha^2 \\ y^3 &\mapsto \lambda^3 = \alpha \\ y^{15} &\mapsto \lambda^{15} = 1 \end{aligned}$$

$$M_{P_4}^{P_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$



Commutative diagram showing transition from tower field basis to normal basis representation

Then we compute matrix M_N^T as dictated by the commutative diagram above, that is $M_N^T = M_N^{P_4} \cdot M_{P_4}^{P_1} \cdot M_{P_1}^T$ and obtain its inverse M_T^N . Matrix $M_N^{P_4}$ was derived earlier in this section and matrix $M_{P_1}^T$ in Section 3.4.1.

$$M_N^T = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \quad M_T^N = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Elements of column 7 in Table 3.14 were obtained from elements in column 5 using the transition matrix M_T^N .

Appendix C

Xilinx specific optimization for the serial LFSR

In this section we explain in detail the Xilinx-ISE optimization for the LFSR using serial loading phase (see Section 4.1.2). Below is a small Section from Synthesis report for module LFSR:

Code	
<hr/>	
1	Final Register Report
3	Macro Statistics
4	# Registers : 16
5	Flip-Flops : 16
6	# Shift Registers : 64
7	6-bit shift register : 16
8	7-bit shift register : 16
9	9-bit shift register : 32

We see that the optimization tools will use shift registers. As mentioned in Section A, SLICEM can be used to implement a shift register. There are two primitives: a 16-bit shift register SRL16 and a 32-bit shift register SRL32. The latter cannot be used because of the tap positions determined by the LFSR polynomial $\ell(x)$. This situation is mirrored in the synthesis report above : a 6-bit shift register ends with S_{25} , a 9-bit shift register ends with S_{16} , a 9-bit shift register ends with S_7 , and finally a 7-bit shift register ends with S_0 .

A closer analysis in `FPGA Editor` reveals the following. For LFSR stage S_{31} , 8 slices SLICEX, each with two registered outputs (primary registers configured as `fdre` - a D-type flip flop with synchronous reset and enable signal) were used, resulting in total of 16

FFs for 16-bit state S_{31} . These registered outputs were then routed to two SLICEM's as follows: a single SLICEM contains 4 LUTs conFigured as the SRL16 primitive, and two outputs connected to their corresponding registers, which gives us 8 SRL16's per SLICEM and a total count of 8 output slice registers, conFigured as `fde`. The two SLICEM's together account for 16 FFs. These 16 registered output signals are then routed to four slices SLICEX, each of them using the four primary output registers conFigured as `fdre`; these hold the content of S_{25} and are then routed for the feedback computation and to the next two SRL16's (i.e. two SLICEM's) for obtaining the state S_{16} . This SLICEM/SLICEX sequence continues down to state S_0 . The slices and registers used for the entire LFSR are listed in the Table C.1 below, to show exactly why the LFSR needs only 152 registers instead of the expected $32 \cdot 16\text{-bit} = 512$ registers.

	# of used SLICEX	# of FFs per SLICEX	# of used SLICEM	# of FFs per SLICEM	DFFs subtotal
S_{31}	8	2			16
S_{26}			2	8	16
S_{25}	4	4			16
S_{17}			2	8	16
S_{16}	4	4			16
S_8			2	8	16
S_7	4	4			16
S_1			2	8	16
S_0	3	4			12
S_0	1	4			4
$reset_{5...8}$		4			4
$reset_{1...4}$	1	4			4
# of FFs for the entire LFSR					152

Table C.1: Module LFSR - register count

To account for the remaining 8 FFs (for signals $reset_1, reset_2, \dots, reset_8$ listed in Table C.1), we need to explain a bit more about SLICEM (see A). Compared to the regular 6-input/2-output LUTs that can be found in SLICEL and SLICEX, the SLICEM LUTs have additional inputs and outputs and can be conFigured as distributed RAM element or a Shift Register LUT (SRL). For the later, there are two configurations, namely SRL32 (a 32-bit shift register with one input DI1 and output O5) or SRL16 implementing a 16-bit shift register. It is possible to implement two SRL16 primitives inside one LUT, using two inputs DI1, DI2 and both outputs O5 and O6. When conFigured as SRL, the 6 "regular" LUT inputs act as address signals to enable asynchronous read needed to implement shorter shift registers, [56].

Each of the 8 SLICEM used for the LFSR has two SRL16's per LUT, with two LUT outputs connected to two corresponding FF's implementing synchronous read. Recall the register report given above: 5-bit shift register implements LFSR stages s_{30}, \dots, s_{26} with s_{30}, \dots, s_{27} in the actual LUT and the 5-th stage s_{26} being the corresponding FF, ie the SRL16 is set to realize a $4 + 1$ shift register. Bit 6 of the 6-bit shift register found by the synthesis tool is the corresponding state s_{25} bit, located in a separate SLICEX. Figure C.1 shows the two bits of the first two LFSR stages, $s_{31}<0>$, $s_{31}<1>$ and $s_{25}<0>$, $s_{25}<1>$. Elements in the Figure are grouped together into two grey blocks, left for the elements belonging to SLICEM, followed by the elements of SLICEX on the right. The two tall elements in SLICEM are the two SRL16 primitives (`Mshreg_s25_0` and `Mshreg_s25_1`), that are implemented in a single SLICEM LUT. They shift the two bits through the LFSR stages s_{30}, \dots, s_{27} with the two output FFs `s25_01` and `s25_11` as the 5th register in the sequence, holding the bits 0 and 1 of s_{26} . A very important detail is that when the LUTs are conFIGured as SRL's, the SLICEM registers cannot be set or reset, [56]. WG implementation uses a synchronous reset, which has to be implemented for all LFSR stages as well. Since the bits currently contained in SLICEM cannot be reset directly, the reset is realized by routing the output `s25_01`, which holds the bit $s_{26}<0>$, to a SLICEX LUT `s25_011` implementing an AND operation with a signal that simulates the reset, the `reset_5`. The result of this operation is then stored in `s25_0` FF, which holds bit 0 of LFSR stage s_{25} . This flip flop is the actual tap position.

There are 8 serially connected flip flops of type `fdre` (dark grey blocks in Figure C.1 denoted `reset_IBUF_shift1`, \dots , `reset_IBUF_shift1`), first one of them connected to logical '1'. When the module is in its normal operation, these registers propagate the value '1'. The output of the 5th register (marked `reset_5`, denoted with a solid line in Figure C.1) is connected to the SLICEX LUT as one of the inputs to the AND gate. When `reset_5` = 1, the LUT `s25_011` just passes through the value from the SLICEM shift register. When the `reset` signal is set (dashed line in Figure C.1), the SLICEX registers `s31_0`, `s31_1`, `s25_0`, `s25_1` and all the "reset" registers are cleared. In this way, the 8 "reset" registers produce 8 values '0', ie a "reset chain". For the next 5 clock cycles, the value of `reset_5` will be set to '0' and will clear the bits coming from the SLICEM SRL, namely the stages s_{30}, \dots, s_{26} , via the AND gate.

The next 4 SLICEM's in the design are used to implement stages s_{24}, \dots, s_{17} and s_{15}, \dots, s_8 , both as 8-bit SRLs, using the signal `reset_8` from the "reset chain". The last two SLICEM's implement stages s_6, \dots, s_1 as 6-bit SRL, using the reset signal `reset_6`. Thus, all 152 FFs have been accounted for.

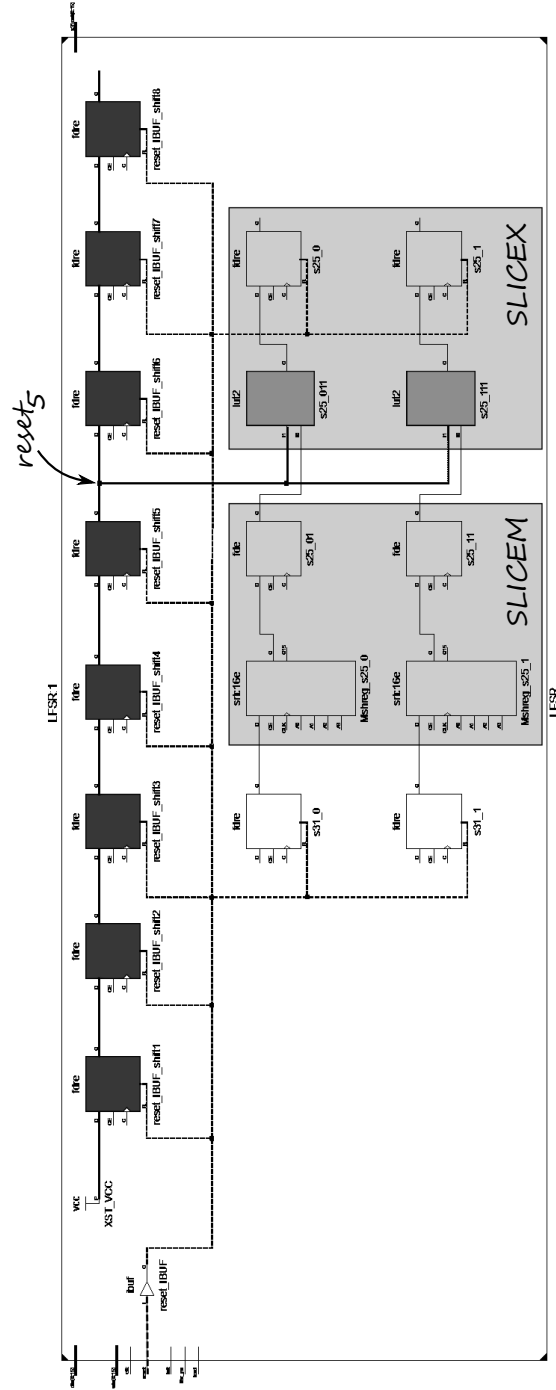


Figure C.1: Module LFSR - Xilinx-ISE technology map view of SRL's

Appendix D

Extended Euclidean Algorithm for inversion in polynomial basis

The division algorithm for polynomials states that for a nonzero polynomial $a(x)$ and a polynomial $f(x)$, $\deg(f) \geq \deg(a)$, there exists a nonzero polynomial $q_0(x)$ and a polynomial $r_0(x)$, such that $f(x) = q_0(x)a(x) + r_0(x)$, where $0 \leq \deg(r_0) < \deg(a)$. If $r_0(x) \neq 0$, we repeat the procedure: we now need to find $q_1(x)$ and $r_1(x)$ such that $a(x) = q_1(x)r_0(x) + r_1(x)$, $0 \leq \deg(r_1) < \deg(r_0)$. Continuing this procedure we obtain a strictly decreasing sequence

$\deg(r_i) : \deg(r_0) > \deg(r_1) > \dots \geq 0$, eventually obtaining zero polynomial $r_n(x) = 0$ after finite number of steps $n \in \mathbb{N}$. For simplicity, we will write a instead of $a(x)$.

$$\begin{aligned}
 r_{-2} &= f &= q_0 a + r_0 && 0 \leq \deg(r_0) < \deg(a) \\
 r_{-1} &= a &= q_1 r_0 + r_1 && 0 \leq \deg(r_1) < \deg(r_0) \\
 && r_0 &= q_2 r_1 + r_2 && 0 \leq \deg(r_2) < \deg(r_1) \\
 && & \vdots && \\
 && r_{n-3} &= q_{n-1} r_{n-2} + r_{n-1} && 0 \leq \deg(r_{n-1}) < \deg(r_{n-2}) \\
 && r_{n-2} &= q_n r_{n-1} + r_n && 0 = \deg(r_n) < \deg(r_{n-1})
 \end{aligned} \tag{D.1}$$

Equations (D.1) can be rewritten in the form $r_i = r_{i-2} - q_i r_{i-1}$ for $i = 0, 1, \dots, n$.

The value r_{n-1} is the greatest common divisor of a and f :

$$\gcd(a, f) = \gcd(r_{-2}, r_{-1}) = \gcd(r_{-1}, r_0) = \dots = \gcd(r_{n-1}, r_n) = r_{n-1}$$

Before last equality holds because $r_n = 0$ in $\gcd(r_{n-1}, 0) = r_{n-1}$.

The EEA does not only find the $\gcd(a, f)$, but also gives the solution of the equation

$$ax + fy = \gcd(a, f) \tag{D.2}$$

It starts from two extremes:

$$\begin{aligned} a \cdot 0 + f \cdot 1 &= f \\ a \cdot 1 + f \cdot 0 &= a \end{aligned} \tag{D.3}$$

and finds two sequences $\{p_i\}$ and $\{q_i\}$, for which the following holds: $ap_i + fq_i = r_i$. Equations D.3 give the initial values $r_{-2} = b$, $p_{-2} = 0$, $p_{-1} = 1$, $r_{-1} = a$, $q_{-1} = 1$ and $q_{-2} = 0$. Then following the example (D.1), in each step we look for s_i and r_i , such that $r_{i-2} = s_i r_{i-1} + r_i$, where $0 \leq \deg(r_i) < \deg(r_{i-1})$. This allows the computation of

$$\begin{aligned} p_i &= p_{i-2} - s_i p_{i-1} \\ q_i &= q_{i-2} - s_i q_{i-1} \end{aligned} \tag{D.4}$$

As already mentioned, the procedure terminates with $r_n = 0$, obtaining the solution to the equation (D.2): $x = p_{n-1}$, $y = q_{n-1}$, $r_{n-1} = \gcd(a, f)$

Our goal is to find the inverse $p(x)$ of polynomial $a(x) \in \mathbb{F}_{2^m}$, i.e. we are solving the congruence $a(x)p(x) \equiv 1 \pmod{f(x)}$, with $f(x)$ being the defining polynomial of \mathbb{F}_{2^m} . We can rewrite it as $a(x)p(x) + f(x)q(x) = 1$ and find the inverse $p(x)$ using EEA.

We are working in a binary field, so - in equations (D.4) becomes + and since we are only interested in the inverse $p(x)$, we skip the calculation of the sequence $\{q_i\}$ and only return the inverse (EEA finds the gcd and both, x and y from equation (D.2)).

Since $f(x)$ is defining polynomial of the field, all the field elements are reduced modulo $f(x)$ and hence coprime to $f(x)$; we know that the greatest common divisor (the value r_{n-1}) will be 1. We skip the last line in equations (D.1) and terminate when register r_1 becomes $r_{n-1} = 1$, a constant polynomial of degree 0; hence choosing $\deg(r_1) \neq 0$ for the loop condition. At that point the desired inverse is also obtained and held in register p_1 . The procedure described above is held in the following algorithm:

Algoritem 1 EEA for polynomials 1

VHOD: polynomial a
IZHOD: polynomial p (inverse a^{-1})
1: $r_2 \leftarrow f, p_2 \leftarrow 0$
2: $r_1 \leftarrow a, p_1 \leftarrow 1$
3: **while** $\deg(r_1) \neq 0$ **do**
4: $s \leftarrow r_2 \text{ div } r_1$
5: $r \leftarrow r_2 + sr_1$
6: $p \leftarrow p_2 + sp_1$
7: $r_2 \leftarrow r_1, r_1 \leftarrow r$
8: $p_2 \leftarrow p_1, p_1 \leftarrow p$
9: **end while**
10: **return** p_1

Command **div** in step 4 of alg. 1 returns the corresponding q_i from equations (D.1). Of course it is not the actual division, but instead, the polynomial r_1 is multiplied by x until $\deg(r_1) = \deg(r_2)$, ie:

$$d = \deg(r_2) - \deg(r_1)$$

$$s \leftarrow x^d$$

$$r \leftarrow r_2 + s \cdot r_1$$

We now get rid of redundant registers **s** and **r** by setting

$$d = \deg(r_2) - \deg(r_1)$$

$$r_2 \leftarrow r_2 + x^d \cdot r_1$$

$$p_2 \leftarrow p_2 + x^d \cdot p_1$$

If $d < 0$ we swap registers: $r_1 \leftrightarrow r_2$ and $p_1 \leftrightarrow p_2$.

Algoritem 2 EEA for polynomials 2

VHOD: polynomial a
IZHOD: polynomial p (inverse a^{-1})
1: $r_2 \leftarrow f, p_2 \leftarrow 0$
2: $r_1 \leftarrow a, p_1 \leftarrow 1$
3: **while** $\deg(r_1) \neq 0$ **do**
4: $d \leftarrow \deg(r_2) - \deg(r_1)$
5: **if** $d < 0$ **then**
6: $r_2 \leftrightarrow r_1, p_2 \leftrightarrow p_1, d \leftarrow -d$
7: **end if**
8: $r_2 \leftarrow r_2 \oplus x^d r_1$
9: $p_2 \leftarrow p_2 \oplus x^d p_1$
10: **end while**
11: **return** p_1

Each iteration in algorithm 2 reduces either degree of r_2 or degree of r_1 for d : worse case being $d = 1$ in each iteration, which results in maximum possible number of iterations $(2m - 1)$. Multiplication by x^d can be implemented by simple shifting. Nonetheless, two problems remain: how to keep track of degree of a polynomial and the while loop (the latter is VHDL specific - you cannot implement a loop without knowing the exact number of iterations). Both problems can be solved at once: instead of multiplying once by x^d we multiply by x d times.

Algorithm 3 keeps track of d by increasing it when r_1 is begin multiplied and decreasing it when r_2 is begin multiplied by x . When both polynomials have the same degree (both leading coefficients are 1), they are added together (XOR) - this is done in steps 10 and 11. Since this will decrease the degree of r_2 , we know it will have to be multiplied by x (step 13), so d will be set to 1 (step 17) and the registers swapped (steps 15 and 16).

Note: in step 19 we divide p_1 instead of multiplying p_2 and decrease d .

Algoritem 3 EEA for polynomials 3

VHOD: polynomial a IZHOD: polynomial p (inverse a^{-1})

```
1:  $r_2 \leftarrow f, p_2 \leftarrow 0$ 
2:  $r_1 \leftarrow a, p_1 \leftarrow 1$ 
3: for  $i = 0$  to  $2m - 1$  do
4:   if  $r_1(m) = 0$  then
5:      $r_1 \leftarrow xr_1$ 
6:      $p_1 \leftarrow xp_1$ 
7:      $d \leftarrow d + 1$ 
8:   else
9:     if  $r_2(m) = 1$  then
10:       $r_2 \leftarrow r_2 \oplus r_1$ 
11:       $p_2 \leftarrow p_2 \oplus p_1$ 
12:    end if
13:     $r_2 \leftarrow xr_2$ 
14:    if  $d = 0$  then
15:       $\{r_2 \leftrightarrow r_1\}$ 
16:       $\{p_1 \leftarrow xp_2, p_2 \leftarrow p_1\}$ 
17:       $d \leftarrow 1$ 
18:    else
19:       $p_1 \leftarrow p_1/x$ 
20:       $d \leftarrow d - 1$ 
21:    end if
22:  end if
23:   $i \leftarrow i + 1$ 
24: end for
25: return  $p_1$ 
```

Let us examine what happens in lines 4 to 22 of the above algorithm. Using `new_r1`, `new_r2`, `new_p1`, `new_p2` and `new_d` for registers that hold the updated values for each iteration, and setting MSB bits $\text{rm1}=r_1(m)$, $\text{rm2}=r_2(m)$ and $\text{dbit}=0$ for $d = 0$ and $\text{dbit}=1$ for $d > 0$, we get the following:

$$\text{new_r1} = \begin{cases} xr_1 & , \text{rm1} = 0; \\ r_1 & , \text{rm1} = 1, \text{dbit} = 1; \\ xr_2 & , \text{rm1} = 1, \text{dbit} = 0, \text{rm2} = 0; \\ x(r_2 + r_1) & , \text{rm1} = 1, \text{dbit} = 0, \text{rm2} = 1; \end{cases}$$

$$\text{new_p1} = \begin{cases} xp_1 & , \text{rm1} = 0; \\ p_1/x & , \text{rm1} = 1, \text{dbit} = 1; \\ xp_2 & , \text{rm1} = 1, \text{dbit} = 0, \text{rm2} = 0; \\ x(p_2 + p_1) & , \text{rm1} = 1, \text{dbit} = 0, \text{rm2} = 1; \end{cases}$$

$$\text{new_r2} = \begin{cases} r_2 & , \text{rm1} = 0; \\ r_1 & , \text{rm1} = 1, \overline{\text{dbit}} = 1; \\ xr_2 & , \text{rm1} = 1, \overline{\text{dbit}} = 0, \text{rm2} = 0; \\ x(r_2 + r_1) & , \text{rm1} = 1, \overline{\text{dbit}} = 0, \text{rm2} = 1; \end{cases}$$

$$\text{new_p2} = \begin{cases} p_2 & , \text{rm1} = 0; \\ p_1 & , \text{rm1} = 1, \overline{\text{dbit}} = 1; \\ p_2 & , \text{rm1} = 1, \overline{\text{dbit}} = 0, \text{rm2} = 0; \\ p_2 + p_1 & , \text{rm1} = 1, \overline{\text{dbit}} = 0, \text{rm2} = 1; \end{cases}$$

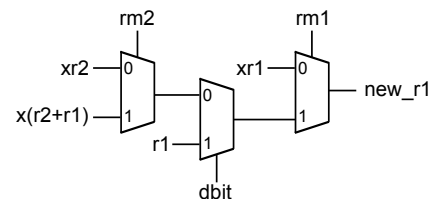


Figure D.1: EEA inversion in polynomial basis - schematic for `new_r1`

Instead of having an integer d we can take a register of the same length as r_1 and r_2 and have all values but the one set to zero. We start with $10\dots 0$ and shift it right for one bit for $d + 1$ and for $d - 1$ shift left. (note: when we enter the iteration for the first time, r_2 will hold the irreducible polynomial, hence $\text{rm2}=1$, and r_1 will hold polynomial $a(x)$, we need to add a condition that will set the shift reg to $10\dots 0$ in the first iteration).

$$\text{new_d} = \begin{cases} 1 & , [\text{rm1} = 0, \text{dbit} = 0] \text{ or } [\text{rm1} = 1, \text{dbit} = 0]; \\ d + 1 & , \text{rm1} = 0, \text{dbit} = 1; \\ d - 1 & , \text{rm1} = 1, \text{dbit} = 1; \end{cases}$$

The above expressions for `new_r1`, `new_r2`, `new_p1`, `new_p2` and `new_d` are implemented in a submodule called `eea_step`, which computes one iteration of the algorithm 3. This submodule must be connected either in a FSM running the $2m$ iterations or into a pipeline of the same length. As already mentioned, we can terminate one step earlier, but must not forget the final shift of the `new_p1` register, ie the inverse is $p = p_1/x$.

Inversion 2

Implementation results for inversion modules

We are constructing a pipelined WGT module, so we choose a to implement inversion in a pipeline. There are several options regarding the level of pipelining when using EEA:

1. a single `eea_step` in a pipeline stage, resulting in 31 stage pipeline (module `invP1`)
2. two steps in a pipeline stage (module `eea_2_step`), resulting in a 16 stage pipeline (module `invP2`)
3. four steps in a pipeline stage (module `eea_4_step`), resulting in a 8 stage pipeline (module `invP4`)
4. eight steps in a pipeline stage (module `eea_8_step`), resulting in a 4 stage pipeline (module `invP8`)

Note that for options 2 to 4, the last pipeline stage contains one step less than other stages. The second inversion module `inv16` was implemented directly from schematic in Figure ??.

Implementation results of all inversion modules are given in Table D.1 below:

Basic Building Block	FPGA Results			
	# of FFs	# of LUTs	# of Slices	Block Delay [ns]
<code>eea_step</code>	/	86	36	15.338
<code>invP1</code>	2635	2847	919	6.149
<code>eea_2_step</code>	/	181	76	18.990
<code>invP2</code>	1361	3561	1193	7.843
<code>eea_4_step</code>	/	362	150	21.387
<code>invP4</code>	680	3849	1217	9.039
<code>eea_8_step</code>	/	742	343	32.919
<code>invP8</code>	344	3827	1193	15.866
<code>inv16</code>	248	1054	369	7.271

Table D.1: EEA inversion in polynomial basis - implementation results

The `invP1` module alone is already twice the size of the entire WGT-16 implementation using tower construction 1, therefore, we do not continue the implementation using EEA. The square and multiply method (module `inv16`) gives way better results; it takes up less than 35% of the area of `invP1`, but has approximately 15% longer clock period. However,

module `inv16` has a 6 stage pipeline, which results in approximately 43ns delay through the module, whereas the 31 stage pipeline of `invP1` gives about 190 ns delay through the inverter. In comparison, that means more than 75% speedup using module `inv16`. Implementation results of inverter `inv16` encourage the implementation of WGT module using polynomial basis.

Appendix E

Detailed gate count

In this section we provide a detailed gate count to allow us to compare basic building blocks from different tower constructions. Number of AND and XOR gates proves to be insufficient for the task due to big differences in architecture of individual modules. It also gives a better understanding of relative sizes of squarer and inverter blocks compared to the multiplier blocks at the same level of the tower within a particular tower construction. We provide area complexity and critical path delay through the gate in terms of area and delay of 1 NAND gate. The numbers used here are given in Figure E.1. Note that the actual gate equivalence numbers from ASIC designers might differ from these.

For an FPGA design, gate equivalents are irrelevant, because LUTs are used to implement boolean functions; in this case, we only need to know the number of inputs the function has.


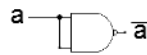
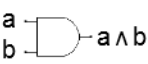
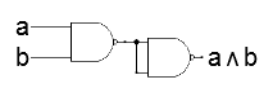
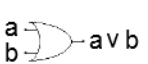
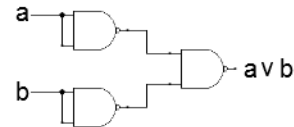
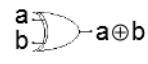
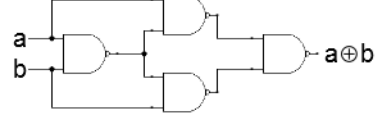
GATE	EQUIVALENT CIRCUIT WITH NAND GATES	N = area of 1 NAND gate T = delay through 1 NAND gate	
		AREA	DELAY
 \bar{a}		1N	1T
 $a \wedge b$		2N	2T
 $a \vee b$		3N	2T
 $a \oplus b$		4N	3T

Figure E.1: Area and delay of NOT, AND, OR and XOR gates in terms of NAND gates

$\mathbb{F}_{2^{16}}$ NB	Building Block	Area N	Critical Path Delay - T
	Multiplication (M_{16})	3648	23

Table E.1: Area and time complexities of building blocks in Section 4.3 in terms of NAND gates

Tower Field	Building Block	Area N	Critical Path Delay - T
\mathbb{F}_{2^2}	Squaring (\mathbf{S}_2)	0	0
	Multiplication (\mathbf{M}_2)	10	8
	\mathbf{M}_α	4	3
	\mathbf{M}_{α^2}	4	3
$\mathbb{F}_{(2^2)^2}$	Squaring (\mathbf{S}_4)	28	9
	Multiplication (\mathbf{M}_4)	66	14
	Inversion (\mathbf{I}_4)	50	22
	\mathbf{M}_λ	20	6
	\mathbf{M}_{λ^2}	28	6
	\mathbf{M}_β	20	9
	$\mathbf{M}_{\alpha\beta}$	24	6
$\mathbb{F}_{((2^2)^2)^2}$	Squaring (\mathbf{S}_8)	124	21
	Multiplication (\mathbf{M}_8)	302	23
	Inversion (\mathbf{I}_8)	348	68
	\mathbf{M}_μ	110	15
$\mathbb{F}_{(((2^2)^2)^2)^2}$	Squaring (\mathbf{S}_{16})	454	42
	Multiplication (\mathbf{M}_{16})	1264	41
	Inversion (\mathbf{I}_{16})	1452	153

Table E.2: Area and time complexities of building blocks in Section 4.4.1 in terms of NAND gates

Tower Field	Building Block	Area N	Critical Path Delay - T
	Multiplication (\mathbf{M}_4)	92	11
	Inversion (\mathbf{I}_4) - bad	184	22
	Inversion (\mathbf{I}_4) - GOOD	76	9
$\mathbb{F}_{(2^4)^4}$	Multiplication (\mathbf{M}_{16})	1480	32
	Inversion (\mathbf{I}_{16}) †	/	/

Table E.3: Area and time complexities of building blocks in Section 4.5.1 in terms of NAND gates

Bibliography

- [1] GAP - Groups, Algorithms, Programming - a System for Computational Discrete Algebra, available at <http://www.gap-system.org/>
- [2] G. Gong, A.M. Youssef, "Cryptographic Properties of the Welch-Gong Transformation Sequence Generators", *IEEE Trans. on Information Theory*, 48(11):2837-2846, November 2002
- [3] Y. Nawaz and G. Gong, "The WG Stream Cipher", *eSTREAM PHASE 2 Archive*, available at http://www.ecrypt.eu.org/stream/p2ciphers/wg/wg_p2.pdf, 2005.
- [4] Y. Nawaz, "Design of Stream Ciphers and Cryptographic Properties of Nonlinear Functions", PhD thesis, Univ. of Waterloo, 2007
- [5] H. El-Razouk, A. Reyhani-Masoleh and G. Gong, "New Implementations of the WG Stream Cipher", Technical Reports, CACR 2012-31, available at <http://cacr.uwaterloo.ca/techreports/2012/cacr2012-31.pdf>
- [6] X. Fan, N. Zidaric, M. Aagaard, and G. Gong, "Efficient Hardware Implementation of the Stream Cipher WG-16 with Composite Field Arithmetic", *TrustED@CCS 2013*: 21-34 available at <http://cacr.uwaterloo.ca/techreports/2013/cacr2013-23.pdf>
- [7] K. Mandal, G. Gong, X. Fan, M. Aagaard, "Optimal parameters for the WG stream cipher family", *Cryptography and Communications* (2014)6:117-135, available at <http://cacr.uwaterloo.ca/techreports/2013/cacr2013-15.pdf>
- [8] X. Fan and G. Gong, "Specification of the Stream Cipher WG-16 Based Confidentiality and Integrity Algorithms", Technical Reports, CACR 2013-06, available at <http://cacr.uwaterloo.ca/techreports/2013/cacr2013-06.pdf>

- [9] M.D. Aagaard, G. Gong, R.K. Mota, “Hardware Implementations of the WG-5 Cipher for Passive RFID Tags”, HOST 2013: 29-34
- [10] C.H. Lam, M. Aagaard, and G. Gong, “Hardware Implementations of Multi-output Welch-Gong Ciphers”, March 2009, available at <http://cacr.uwaterloo.ca/techreports/2011/cacr2011-01.pdf>
- [11] G. Yang, X. Fan, M. Aagaard and G. Gong, “Design Space Exploration of the Lightweight Stream Cipher WG-8 for FPGAs and ASICs”, The 8th Workshop on Embedded Systems Security (WESS’13), ACM Press, Article No. 8, September 29, 2013,
- [12] H. Wu, “Cryptanalysis and Design of Stream Ciphers,, PhD thesis, Katholieke Universiteit Leuven, Belgium, July 2008
- [13] H. El-Razouk, A. Reyhani-Masoleh, and G. Gong. “New Hardware Implementations of WG(29;11) and WG-16 Stream Ciphers Using Polynomial Basis,” Accepted to IEEE Trans. on Computers, available at <http://cacr.uwaterloo.ca/techreports/2014/cacr2014-02.pdf>
- [14] ETSI/SAGE Specification version 1.1: “Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G Specification”, Sept. 2006, available at <http://www.gsma.com/technicalprojects/wp-content/uploads/2012/04/snow3gspec.pdf>
- [15] P. Kitsos, G. Selimis, and O. Koufopavlou, “High Performance ASIC Implementation of the SNOW 3G Stream Cipher”, available at http://dsmc.eap.gr/en/members/pkitsos/papers/Kitsos_c35.pdf
- [16] ETSI/SAGE Specification version 1.6: “Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification”, June 2011, available at <http://www.gsma.com/technicalprojects/wp-content/uploads/2012/04/eea3eia3zucv16.pdf>
- [17] Z. Liu, L. Zhang, J. Jing, and W. Pan, “Efficient Pipelined Stream Cipher ZUC Algorithm in FPGA”, The first International workshop on ZUC algorithm
- [18] L.Wang, J. Jing, Z. Liu, L. Zhang, and W. Pan, “Evaluating Optimized Implementations of Stream Cipher ZUC Algorithm on FPGA”, ICICS 2011, LCNS 7043

- [19] P. Kitsos, N. Sklavos, and A.N. Skodras, “An FPGA Implementation of the ZUC Stream Cipher”, DSD 2011
- [20] P. Kitsos, N. Sklavos, G. Provelengios, and A.N. Skodras, “FPGA-based performance analysis of stream ciphers ZUC, Snow3g, Grain V1, Mickey V2 Trivium and E0”, *Microprocessors & Microsystems*, Volume 37, Issue 2, March, 2013, pp. 235-245
- [21] L. Zhang, L. Xia, Z. Liu, J. Jing and Y. Ma, “Evaluating the Optimized Implementations of SNOW3G and ZUC on FPGA”, *TrustCom 2012*: 436-442
- [22] C. De Canniere and B. Preneel, “Trivium”, *New Stream Cipher Designs - the eSTREAM finalists*, Springer-Verlag, Berlin Heidelberg, 2008
- [23] M. Hell, T. Johansson, and W. Meier, “Grain - A Stream Cipher for Constrained Environments”, available at <http://www.ecrypt.eu.org/stream/ciphers/grain/grain.pdf>
- [24] M. Hell, T. Johansson, A. Maximov, and W. Meier, “The Grain Family of Stream Ciphers”, *New Stream Cipher Designs - the eSTREAM finalists*, Springer-Verlag, Berlin Heidelberg, 2008
- [25] M. Robshaw, “The eSTREAM Project”, *New Stream Cipher Designs - The eSTREAM Finalists*, Springer-Verlag, Berlin Heidelberg, 2008
- [26] F.K. G  rkaynak, P. Luethi, N. Bernold, R. Blattmann, V. Goode, M. Marghitola, H. Kaeslin, N. Felber, and W. Fichtner, “Hardware Evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt”, available at <http://www.ecrypt.eu.org/stream/papersdir/2006/015.pdf>
- [27] T. Good, W. Chelton, and M. Benaissa, “Review of stream cipher candidates from a low resource hardware perspective”, available at <http://www.ecrypt.eu.org/stream/papersdir/2006/016.pdf>
- [28] K. Gaj, G. Southern, and R. Bachimanchi, “Comparison of hardware performance of selected Phase II eSTREAM candidates”, available at <http://www.ecrypt.eu.org/stream/papersdir/2007/026.pdf>
- [29] P. Bulens, K. Kalach, F. Standaert, and J. Quisquater, “FPGA Implementations of eSTREAM Phase-2 Focus Candidates with Hardware Profile”, available at <http://www.ecrypt.eu.org/stream/papersdir/2007/024.pdf>

- [30] M. Rogawski ,“Hardware evaluation of eSTREAM Candidates: Grain, Lex, Mickey128, Salsa20 and Trivium”, available at <http://www.ecrypt.eu.org/stream/papersdir/2007/025.pdf>
- [31] T. Good and M. Benaissa, “ASIC Hardware Performance”, New Stream Cipher Designs - The eSTREAM Finalists, Springer-Verlag, Berlin Heidelberg, 2008
- [32] D. Hwang, M. Chaney, S. Karanam, N. Ton, and K. Gaj, “Comparison of FPGA-Targeted Hardware Implementations of eSTREAM Stream Cipher Candidates” , available at http://ece.gmu.edu/~kgaj/publications/conferences/GMU_SASC_2008.pdf
- [33] “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications”, Special Publication 800-22, available at <http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf>
- [34] V. Rijmen, “Efficient Implemenetation of the Rijndael S-box”, available at <http://www.networkdls.com/Articles/sbox.pdf>
- [35] A. Rudra, P.K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi, “Efficient Rijndael Encryption Implementation with Composite Field Arithmetic”, CHES 2001, LNCS 2162:171-184
- [36] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, “A Compact Rijndael Hardware Architecture with S-Box Optimization”, Advances In Cryptology - ASIACRYPT 2001, LNCS 2248, 2001, pp 239-254
- [37] S. Morioka and A. Satoh, “An Optimized S-Box Circuit Architecture for Low Power AES Design”, CHES 2002, LNCS 2523:172-186, 2003
- [38] N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede, “A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-Box”, CT-RSA 2005, LNCS 3376, 2005, pp 323-333
- [39] D. Canright, “A Very Compact S-Box for AES”, CHES 2005 , LNCS 3659, 2005, pp 441-455
- [40] S. Nikova, V. Rijmen, and M. Schlaeffler, “Using Normal Bases for Compact Hardware Implementations of the AES S-Box”, Security and Cryptography for Networks, LNCS 5229, 2008, pp 236-245

- [41] Y. Nogami, K. Nekado, T. Toyota, N. Hongo, and Y. Morikawa, "Mixed Bases for Efficient Inversion in $\mathbb{F}_{((2^2)^2)^2}$ and Conversion Matrices of SubBytes of AES", CHES 2010, LNCS 6225, 2010, pp 234-247
- [42] A. Bonnecaze, P. Liardet, and A. Venelli, "AES side-channel countermeasure using random tower field constructions", *Designs, Codes and Cryptography*, December 2013, Volume 69, Issue 3, pp 331-349
- [43] Michael John Sebastian Smith, "Application-specific integrated circuits", Addison-Wesley 1998, available at <http://iroi.seu.edu.cn/books/asics/ASICs.htm>
- [44] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs", *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, Volume 26, Issue 2, February 2007, pp.203 - 215
- [45] Synopsys Design Compiler, <http://www.synopsys.com/Tools/Implementation/RTLSynthesis/>
- [46] ModelSim, <http://www.mentor.com/products/fv/modelsim/>
- [47] Xilinx, <http://www.xilinx.com/>
- [48] (2011) Xilinx - Glossary (UG659) available at http://people.wallawalla.edu/~larry.aamodt/engr433/xilinx_10/xilinx_10_gls.pdf
- [49] (2008) Xilinx - Development System Reference Guide (10.1) available at <http://www.xilinx.com/itp/xilinx10/books/docs/dev/dev.pdf>
- [50] T. Huffmire, C. Irvine, Thudy D. Nguyen, T. Levin, R. Kastner, and T. Sherwood, *Handbook of FPGA Design Security*, Springer, 2010
- [51] Stefan Mangard, Elisabeth Oswald, and Thomas Popp, *Power analysis attacks - Revealing the secrets of smart cards*, Springer, 2007
- [52] Francisco Rodriguez-Henriquez, N.A. Saqib, A. Diaz Perez, and Cetin Kaya Koc, *Cryptographic Algorithms on Reconfigurable Hardware*, Springer, 2006
- [53] (2008) Xilinx - Programmable Logic Design - Quick Start Guide (UG500(v1.0)) available at http://www.xilinx.com/support/documentation/boards_and_kits/ug500.pdf

- [54] S. Brown and Z. Vranesic, *Fundamentals of digital logic with VHDL design*, 3rd ed. , McGraw-Hill, 2009
- [55] Xilinx - FPGA Design Flow Overview
available at http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_fpga_design_flow_overview.htm
- [56] Xilinx -Spartan-6 FPGA Configurable Logic Block, UG384
available at http://www.xilinx.com/support/documentation/user_guides/ug384.pdf
- [57] Xilinx - XST User Guide, UG627
available at http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/xst.pdf
- [58] S. Brown and J.N Rose, “Architecture of FPGAs and CPLDs: A Tutorial”, IEEE Design and Test of Computers, Vol. 13, No. 2, pp. 42-57, 1996. available at <http://www.eecg.toronto.edu/~jayar/pubs/brown/survey.pdf>
- [59] (2011) Xilinx - Spartan-6 Family Overview (DS160(v2.0))
available at http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf
- [60] (2010) Xilinx - Spartan-6 FPGA Configurable Logic Block - User Guide (UG384(v1.1))
available at http://www.xilinx.com/support/documentation/user_guides/ug384.pdf
- [61] J. Rose, A.El Gamal, and A. Sangiovanni-Vincentelli, “Architecture of Field-Programmable Gate Arrays”, Proc. of the IEEE, Vol.81, No. 7, July 1993 available at <http://www.eecg.toronto.edu/~jayar/pubs/rose/PIEEE93a.pdf>
- [62] (2013) Xilinx - Spartan-6 FPGA SelectIO Resources - User Guide (UG381(v1.5))
available at http://www.xilinx.com/support/documentation/user_guides/ug381.pdf
- [63] A. Telikepalli, “Power vs. Performance: The 90 nm Inflection Point”, WP223 (v1.1) May 12, 2005
- [64] A. Rahman, S. Das, A. Chandrakasan, and R. Reif, “Wiring Requirement and Three-Dimensional Integration Technology for Field Programmable Gate Arrays”, IEEE Trans. Very Large Scale Integration (VLSI) Systems, Vol.11,No.1,February 2003

- [65] J. Deschamps, J.L. Imana and G. D. Sutter , *Hardware Implementation of Finite-Field Arithmetic*, McGraw-Hill,2009
- [66] Pong P. Chu, *RTL HARDWARE DESIGN USING VHDL - coding for Efficiency, Portability, and Scalability*, Wiley & Sons, 2006
- [67] Steve Kilts,*Advanced FPGA design - Architecture, Implementation, and Optimization*, Wiley & Sons, 2007
- [68] S. MacLane and G. Birkhoff,*Algebra*, The Macmillan Company, 1967
- [69] R. Lidl and H. Niederreiter, *Finite fields*, Encyclopedia of Mathematics and its Applications, Vol.20, Cambridge University Press, 1997
- [70] A. Menezes, I. Blake, S. Gao, R. Mullin, S. Vanstone, and T. Yaghoobian, *Applications of Finite Fields* , Kluwer Academic Publishers, 1993
- [71] G.L.Mullen and D.Panario, *Handbook of Finite Fields*, CRC Press, 2013
- [72] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*, CRC Press , 1996
- [73] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag, 2004
- [74] A. Masuda, L. Moura, D. Panario, and D. Thomson, “Low Complexity Normal Elements over Finite Fields of Characteristic Two”, IEEE Trans. Computers, 57, pp. 990-1001, 2008
- [75] R.C.Mullin, I.M. Onyszchuk, S.A. Vanstone, R.M. Wilson, “ Optimal Normal Bases in $GF(p^n)$ ”, Discrete Applied Mathematics 22 (1988/89) 149-161, North-Holland
- [76] S.W. Golomb and G. Gong,*Signal Design for Good Correlation: For Wireless Communication, Cryptography, and Radar*, Cambridge University Press, 2005
- [77] L.Chen and G. Gong, *Communication System Security* , CRC Press, 2012
- [78] A. Joux, “Algorithmic cryptanalysis”, CRC Press, 2009
- [79] J. von zur Gathen and J Shokrollahi, “Efficient FPGA-based Karatsuba multipliers for polynomials over F_2 ”, SAC 2005, LNCS Vol. 3897, 2006, pp. 359-369

- [80] D. H. Green and I. S. Taylor, "Irreducible polynomials over composite Galois fields and their applications in coding techniques", PROC. IEE, Vol.121, No.9, September 1974
- [81] T. Itoh and S. Tsuji, "A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases", Information and Computation 78,171-177 (1988)
- [82] I.S. Hsu, T.K. Truong, I.S. Reed, and N. Glover, "A VLSI architecture for performing finite field arithmetic with reduced table lookup", Linear Algebra and its Applications, 98:249-262, 1988
- [83] M. Morii and M. Kasahara, "Efficient construction of gate circuit for computing multiplicative inverses over $GF(2^m)$ ", Transactions of the IEICE, E72(1):37-42, January 1989
- [84] V.B. Afanasyev, "Complexity of VLSI implementation of finite field arithmetic", II. International Workshop on Algebraic and Combinatorial Coding Theory, pages 6-7, Leningrad, USSR, September 1990
- [85] V.B. Afanasyev, "On the complexity of finite field arithmetic", 5th Joint Soviet-Swedish International Workshop on Information Theory, pages 9-12, Moscow, USSR, January 1991
- [86] C. Paar, "Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields," PhD thesis, (English translation), Inst. for Experimental Mathematics, Univ. of Essen, Essen, Germany, June 1994
- [87] C. Paar, "Fast finite field arithmetic for VLSI design", 3rd Benelux-Japan Workshop on Coding and Information Theory, page 7, Institute for Experimental Mathematics, University of Essen, Germany, August 1993
- [88] C. Paar, "A parallel Galois field multiplier with low complexity based on composite fields", 6th Joint Swedish-Russian Workshop on Information Theory, pages 320-324, Molle, Sweden, August 1993
- [89] C. Paar, "Low complexity parallel multipliers for Galois fields $GF((2^n)^4)$ based on special types of primitive polynomials", 1994 IEEE International Symposium on Information Theory, Trondheim, Norway, June 1994
- [90] C. Paar, "A new architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields", IEEE Trans. Computers, Vol. 45, No. 7, July 1996

- [91] C. Paar, “Fast Arithmetic Architectures for Public-Key Algorithms over Galois Fields $GF((2^n)^m)$ ”, EUROCRYPT '97, LNCS 1233, 1997, pp. 363-378
- [92] J. Guajardo and C. Paar, “Itoh-Tsuji Inversion in Standard Basis and Its Application in Cryptography and Codes”, Designs, Codes and Cryptography, 25(2):207-216, February 2002
- [93] G. Harper, A. Menezes, and S. Vanstone, “Public-Key Cryptosystems with Very Small Key Lengths”, Advances in Cryptology - EUROCRYPT '92, Workshop on the Theory and Application of Cryptographic Techniques, Balatonfűzfő, Hungary, May 24-28, 1992, Proceedings
- [94] E. Savas and C. K. Koc, “Efficient methods for Composite Field Arithmetic”, Technical Report, December 1999, available at <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.82.532>
- [95] C.K. Koc and B. Sunar, “Low-Complexity Bit-Parallel Canonical and Normal Basis Multipliers for a Class of Finite Fields”, IEEE Trans. Computers, 47(3):353-356, March 1998
- [96] B. Sunar and C.K. Koc, “An efficient optimal normal basis type II multiplier”, IEEE Trans. Computers, 50(1):83-87-356, January 2001
- [97] B. Sunar, E. Savas, and C.K. Koc, “Constructing Composite Field Representations for Efficient Conversion”, IEEE Trans. Computers, 52(11):1391-1398, November 2003
- [98] J.L. Massey and J.K. Omura, *Computational Method and Apparatus for Finite Field Arithmetic*, US Patent No. 4587627, 1986
- [99] A. Reyhani-Masoleh and M.A. Hasan, “A New Construction of Massey-Omura Parallel Multiplier over $GF(2^m)$ ”, IEEE Trans. Computers, 51(5):511-520, May 2002