

Automated Design Space Exploration and Datapath Synthesis for Finite Field Arithmetic with Applications to Lightweight Cryptography

by

Nuša Zidarič

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

© Nuša Zidarič 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Nele Mentens
Associate Professor, Dept. of Electrical Engineering, KU Leuven

Supervisor(s): Mark Aagaard
Associate Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo
Guang Gong
Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Internal Member: Catherine Gebotys
Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo
Anwarul Hasan
Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Internal-External Member: Alfred Menezes
Professor, Dept. of Combinatorics and Optimization,
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the 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.

Statement of Contributions

All direct or indirect sources used are acknowledged as references. This thesis consists of material which I co-authored: Subsection 3.2.8, Section 8.2, and Chapters 22 and 23: the nature of the joint work and my contributions are stated at the beginning of the aforementioned sections and in Chapter 21 for Chapters 22 and 23. The main reference for the joint work is [6].

Abstract

Today, emerging technologies are reaching astronomical proportions. For example, the Internet of Things has numerous applications and consists of countless different devices using different technologies with different capabilities. But the one invariant is their connectivity. Consequently, secure communications, and cryptographic hardware as a means of providing them, are faced with new challenges. Cryptographic algorithms intended for hardware implementations must be designed with a good trade-off between implementation efficiency and sufficient cryptographic strength. Finite fields are widely used in cryptography. Examples of algorithm design choices related to finite field arithmetic are the field size, which arithmetic operations to use, how to represent the field elements, etc. As there are many parameters to be considered and analyzed, an automation framework is needed.

This thesis proposes a framework for automated design, implementation and verification of finite field arithmetic hardware. The underlying motif throughout this work is “math meets hardware”. The automation framework is designed to bring the awareness of underlying mathematical structures to the hardware design flow. It is implemented in GAP, an open source computer algebra system that can work with finite fields and has symbolic computation capabilities. The framework is roughly divided into two phases, the architectural decisions and the automated design generation. The architectural decisions phase supports parameter search and produces a list of candidates. The automated design generation phase is invoked for each candidate, and the generated VHDL files are passed on to conventional synthesis tools. The candidates and their implementation results form the design space, and the framework allows rapid design space exploration in a systematic way. In this thesis, design space exploration is focused on finite field arithmetic.

Three distinctive features of the proposed framework are the structure of finite fields, tower field support, and on the fly submodule generation. Each finite field used in the design is represented as both a field and its corresponding vector space. It is easy for a designer to switch between fields and vector spaces, but strict distinction of the two is necessary for hierarchical designs. When an expression is defined over an extension field, the top-level module contains element signals and submodules for arithmetic operations on those signals. The submodules are generated with corresponding vector signals and the arithmetic operations are now performed on the coordinates. For tower fields, the submodules are generated for the subfield operations, and the design is generated in a top-down fashion. The binding of expressions to the appropriate finite fields or vector spaces and a set of customized methods allow the on the fly generation of expressions for implementation of arithmetic operations, and hence submodule generation.

In the light of NIST Lightweight Cryptography Project (LWC), this work focuses mainly on small finite fields. The thesis illustrates the impact of hardware implementation results during the design process of WAGE, a Round 2 candidate in the NIST LWC standardization competition. WAGE is a hardware oriented authenticated encryption scheme. The parameter selection for WAGE was aimed at balancing the security and hardware implementation area, using hardware implementation results for many design decisions, for example field size, representation of field elements, etc. In the proposed framework, the components of WAGE are used as an example to illustrate different automation flows and demonstrate the design space exploration on a real-world algorithm.

Acknowledgements

I would like to thank my supervisors, Professor Mark Aagaard and Professor Guang Gong. For the countless hours of discussions, for sharing their immense knowledge and experience, for their passion for teaching and research, and for their unending patience. For their understanding, support, and advice, for both academical and personal matters, I can never thank them enough! I am very grateful to have been a part of ComSec Lab, and for the opportunity to work on some very exciting projects, such as WAGE.

I would like to thank my committee members, Professor Nele Mentens, Professor Catherine Gebotys, Professor Anwarul Hasan, and Professor Alfred Menezes. For their time, good questions, and valuable comments. It has been a privilege!

Special thanks to Dr. Alexander Konovalov and the GAP Group for their quick help with GAP related issues.

Furthermore, I would like to thank Dr. Kalikinkar Mandal and other past and present members of the ComSec Lab. Many thanks for collaboration, seminars, explanations, discussions, and friendship. After all, a good conversation is the most important ingredient of tea.

I would like to thank Dr. Harrison Oakes for providing basic editing services in the form of addressing issues of grammar, spelling, and syntax for the Parts III-VI of this thesis, and for his friendship.

I would like to thank Professor Aleksandar Jurišić for introducing me to research and for following my progress.

Finally, I would like to thank David Logan for his support over the years. Time flies! Many thanks to the Prilešnik family for always being there for me. Last but not least, I want to thank my friends ... for just being.

Dedication

Niki

Staršem

Hvala!

Table of Contents

List of Figures	xv
List of Tables	xix
I Introduction	1
1 Introduction and motivation	3
1.1 Introduction	3
1.2 The automation framework and the design flow	5
2 Overview	10
2.1 Overview of the automation framework	10
2.2 WAGE	11
2.3 Contributions of the author	12
2.4 Roadmap	13
II Background and related work	16
3 Background	18
3.1 Finite fields	18
3.1.1 Finite field constructions	18
3.1.2 Finite field arithmetic	23
3.1.3 Multivariate polynomials	24
3.1.4 Computer Algebra Systems	25

3.1.5	GAP	26
3.2	Selected symmetric-key cryptographic primitives	27
3.2.1	Feedback shift registers	27
3.2.2	Stream ciphers	30
3.2.3	The eSTREAM project: Grain and Trivium	31
3.2.4	The WG stream cipher	33
3.2.5	Authenticated encryption with associated data	35
3.2.6	The CAESAR competition: Ascon and ACORN	36
3.2.7	The NIST Lightweight Cryptography project	37
3.2.8	The WAGE authenticated encryption scheme	38
3.3	Computer hardware	44
3.3.1	Implementation technologies: ASICs and FPGAs	44
3.3.2	Implementation efficiency and different metrics	45
3.3.3	Hardware design – datapath	47
3.3.4	Hardware design entry using VHDL	51
3.3.5	High-Level Synthesis	51
4	Related work	54
4.1	Hardware implementations of selected cryptographic schemes	54
4.1.1	Lighweight cryptography and its applications	54
4.1.2	The eSTREAM project: Grain and Trivium	56
4.1.3	WG hardware implementations	58
4.1.4	Use of tower fields in cryptography	61
4.2	Cryptographic hardware and complexities	64
4.2.1	General overview	64
4.2.2	Different architectures for finite field arithmetic	65
4.2.3	The XOR counts	66
4.3	Hardware design automation and synthesis tools	67
4.3.1	High-Level synthesis	68
4.3.2	Academic High-Level synthesis	68
4.3.3	High-Level synthesis and design automation related to cryptography	69
4.3.4	Compiling MATLAB onto FPGAs	70

III	The architectural decisions phase	72
5	The architectural decisions phase - finite field arithmetic perspective	74
5.1	Overview	74
5.2	Roadmap	76
5.3	The coarse architectural decisions	78
6	FSR package: feedback shift registers	80
6.1	Main functionality and structure of the FSR package	80
6.2	Examples for the FSR package	87
6.3	Summary of key insights	92
7	FFCSA package: finite field construction, search and algorithms	93
7.1	Main functionality of the FFCSA package	94
7.2	FFCSA profiling methods	99
7.3	Tower field bases	102
7.3.1	Generating tower field bases	102
7.3.2	FFCSA methods for generating TFBs	104
7.4	Algorithms: obtaining expressions for finite field arithmetic	106
7.4.1	Expressions obtained using matrix U	106
7.4.2	The matrix U methods in the FFCSA package	107
7.5	Summary of key insights	111
8	Case study: WG and WAGE	112
8.1	Case study: the WGcipher package	112
8.2	Case study: the WAGE package	116
IV	The automated design generation phase	120
9	The automated design generation phase - the hardware perspective	122

10 GAPtoVHDL package - common VHDL functionality	126
10.1 Common VHDL functionality	126
10.2 Binding of GAP variables and VHDL signals	130
10.3 Classification of expressions defined over finite fields	132
10.4 Summary and conclusion	135
11 FSRtoVHDL package - generating FSR based circuits	136
11.1 Classification of the FSR objects	136
11.2 Generating VHDL for individual FSR objects	138
11.2.1 VHDL packages	139
11.2.2 VHDL for the FSR (sub)modules	141
11.3 A cipher as a collection of basic FSR modules	148
11.3.1 The setup file and the top-level module ports	149
11.3.2 FSR-based system configuration	151
11.3.3 Extracting connections, multiplexers, and external step conditions	153
11.4 Summary and conclusion	157
12 CIRCUIT package: generating arbitrary datapaths	159
12.1 Overview	159
12.2 Roadmap	159
13 CIRCUIT package part 1: arbitrary finite fields	163
13.1 Motivation	163
13.1.1 The running example	163
13.1.2 Analysis	165
13.2 The signal domains and signals	167
13.2.1 The SignalDomain	167
13.2.2 The finite field as a vector space and the SIGNAL object	169
13.2.3 Methods for SignalDomain and SIGNAL objects	174
13.3 The signal package - keeping it all in one place	175
13.4 Summary of key insights	183

14	CIRCUIT package part 2: functional description of the algorithm	185
14.1	The functional description of the algorithm	185
14.2	The datapath based on classification of expressions	192
14.3	Summary of key insights	199
15	CIRCUIT package part 3: VHDL-ready design	202
15.1	The initial design	203
15.2	Top-down processing	207
15.3	Bottom-up processing	212
15.4	Connectors for the tower field elements and vectors	216
15.5	Putting it all together	220
15.6	Summary of key insights	222
16	CIRCUIT package part 4: generating VHDL for the datapath	225
16.1	VHDL packages	226
16.2	VHDL (sub)modules	227
16.3	Testbench generation	233
16.4	Switching the field structure for profiling	233
16.5	Summary and conclusion	236
V	Design space exploration	238
17	Design space exploration - overview	240
17.1	DSE from a mathematical perspective	240
17.2	Different profiles	242
18	DSE for arithmetic modules - basic building blocks	245
18.1	Normal basis multiplication for \mathbb{F}_{2^7}	245
18.2	Polynomial basis multiplication and inversion in \mathbb{F}_{2^7} using primitive defining polynomials	248
18.3	Polynomial basis multiplication and inversion for $\mathbb{F}_{2^{14}}$ using primitive defining polynomials	254

19 Design space exploration of an arbitrary datapath	256
20 The WG cipher design space exploration	257
20.1 Introduction	257
20.2 WGP and constant array implementations	258
20.3 WGT and the trace	259
20.4 The LFSR polynomial	260
VI WAGE	262
21 WAGE - overview	264
22 WAGE algorithm design - impact of profiling	265
22.1 Introduction	265
22.2 The size of the underlying finite field	266
22.3 The remaining nonlinear components	268
22.4 The LFSR feedback	269
22.5 The WAGE permutation hardware area estimate	270
22.6 Summary	271
23 Hardware design of the WAGE datapath	272
23.1 Short summary of the joint work	272
23.2 Hardware design of the WAGE datapath	273
23.3 The WAGE datapath and the FSRtoVHDL package	277
23.4 Summary	279
VII Conclusion	280
24 Conclusion and future work	281
Bibliography	285

VIII	Appendix	302
A	Additional mathematical background	304
	A.0.1 Polynomials	305
	A.0.2 Miscellaneous	306
B	WAGE in more detail	307
	B.1 WAGE loading and tag extraction	307
	B.2 WAGE permutation	309
C	The architectural decisions phase	311
	C.1 FSR package: feedback shift registers	311
	C.2 FFCSA package part 1: additional mathematical background	314
	C.2.1 Cyclotomic coset leaders	314
	C.2.2 Polynomial Φ function	314
	C.2.3 Number of irreducible and primitive polynomials	314
	C.2.4 Number of normal elements and bases	315
	C.2.5 Matrices for the matrix-vector multipliers	316
	C.2.6 Conditions for existence of optimal normal bases type I and II	317
	C.2.7 Dual bases	317
	C.2.8 Optimal normal bases	319
	C.2.9 Tower field bases - general case	320
	C.2.10 FFCSA package limitations	321
	C.3 FFCSA package part 2: additional examples	322
D	The automated design entry and implementation phase	325
	D.1 The FSRtoVHDL package	325
	D.2 The CIRCUIT package part 1	328
	D.3 The CIRCUIT package part 2	337
	D.4 The CIRCUIT package part 3	340
	D.5 The CIRCUIT package part 4	342

List of Figures

1.1	Design flow for automated DSE and datapath synthesis	6
1.2	Simplified design flow diagram and the perspectives (shaded grey)	7
2.1	Simplified design flow diagram and the automation framework (shaded grey)	10
2.2	The automation framework: packages	11
2.3	Roadmap through the design flow for automated DSE and datapath synthesis	15
3.1	Top level schematic of a n -stage FSR (Figure from [36])	29
3.2	Top level schematic of a n -stage FSR with a filter (Figure from [36])	30
3.3	Behavioral model of a stream cipher: (a) encryption and (b) decryption	31
3.4	The structure of Grain: run mode (Figure from [46])	32
3.5	The structure of Trivium (Figure from [47])	32
3.6	The structure of the WG keystream generator (Figure adapted from [50])	34
3.7	Schematic diagram of Ascon: mode of operation (Figure from [57])	37
3.8	The structure of ACORN (Figure from [38])	37
3.9	Rate (shaded) and capacity part of WAGE- \mathcal{AE} -128 (Figure from [6])	39
3.10	Schematic diagram of the WAGE- \mathcal{AE} -128 algorithm (Figure from [6])	43
3.11	Possible combinations of combinational and registered inputs and outputs	48
3.12	Fully parallel, group-level and serial computation in the absence of data dependencies	49
3.13	Serial, partially sequentail and fully sequential computation with data dependencies	50
5.1	Design flow: <i>the architectural decisions</i>	75
5.2	<i>The architectural decisions</i> : (a) the simplified design flow diagram, and (b) the packages for the first part of the GAP framework.	75

7.1	Finite field $\mathbb{F}_{((2^2)^2)^2}$ - tower construction	102
7.2	Decomposition of an element $A \in \mathbb{F}_{((2^2)^2)^2}$ w.r.t. $B_{\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}}$, $B_{\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}}$, $B_{\mathbb{F}_{2^2}/\mathbb{F}_2}$	103
8.1	Small region of WAGE permutation (from Figure 3.9 in Subsection 3.2.8)	118
9.1	Design flow: <i>the automated design generation</i>	122
9.2	<i>The automated design generation</i> : (a) the simplified design flow diagram, (b) the second part of the GAP framework (solid lines, not shaded).	123
11.1	Schematic for the LFSR $\ell(y) = y^4 + y^3 + y + \alpha$ with multiplexers, control signals and data ports	142
11.2	Original schematic of Grain (Figures from [46])	148
11.3	Unified schematic of Grain using FSR	149
12.1	Design flow: <i>the automated design generation</i>	160
13.1	Construction of finite fields $\mathbb{F}_{((2^2)^2)^2}$ and \mathbb{F}_{2^8} and VHDL signals	165
13.2	Circuit schematics (top-level modules) for expression (13.1): solid lines represent the ports and dashed lines the internal signals, boxes show the submodules needed: SQ - squarer, M - multiplier, $\times\gamma_1$ - multiplication with a constant γ_1	166
13.3	Two different tower field constructions	169
13.4	Circuit schematics annotated with SIGNAL and <i>sub-signal</i> showing the elements and the vectors for: (a) top-level module for expression (13.1) and (b) a submodule with 1 input port and 1 output port	174
13.5	SignalPkg for the tower construction of $\mathbb{F}_{((2^2)^2)^2}$ - left branch of diagram in Figure 13.3	182
14.1	Binding for AlgFunctionality object and their effects on the hardware module	191
14.2	The steps of writing a datapath	194
14.3	Last part of <i>the architectural decisions</i> and <i>the automated design generation</i> with key insights	201
15.1	The initial AlgDesign lists for SignalPkg corresponding to $\mathbb{F}_{((2^2)^2)^2}$: graphical representation of extracted (SMlist), generated (AFlist) and VHDL-ready (CIRClisT) submodules	204
15.2	The current stage in the design-flow diagram: (a) the initial design, and (b) the VHDL-ready datapath	205
15.3	Pass 1 of the ProcessSMAFloop for the AlgDesign with the expression in equation (13.1)	209
15.4	Pass 2 of the ProcessSMAFloop for the AlgDesign with the expression in equation (13.1)	210
15.5	Pass 3 of the ProcessSMAFloop for the AlgDesign with the expression in equation (13.1)	211

15.6	The ProcessAFCIRCLoop for the AlgDesign with the expression in equation (13.1)	214
15.7	The directed graphs for the additive constant γ_2	216
15.8	The ProcessDesign for the AlgDesign with the expression in equation (13.1)	221
15.9	The automated design generation with key insights	222
16.1	The (steps of writing a) datapath including the connectors and the registers	228
16.2	The automated design generation and the design space exploration: profiling loop invoking synthesis tools	234
16.3	Design flow: datapath synthesis (automation framework)	237
17.1	Design flow: detained design space exploration	241
18.1	Area results for the \mathbb{F}_{2^7} normal basis multiplication	247
18.2	Area results for the \mathbb{F}_{2^7} polynomial basis multiplication	252
18.3	Area results for the \mathbb{F}_{2^7} polynomial basis inversion	253
18.4	Area results for the $\mathbb{F}_{2^{14}}$ polynomial basis multiplication	254
18.5	Area results for the $\mathbb{F}_{2^{14}}$ polynomial basis inversion	255
19.1	Implementation results for module implementing the expression in equation (13.1): DSE for 720 different tower field constructions for $\mathbb{F}_{(2^2)^2}$	256
20.1	Schematics for the LFSR with original feedback and its <i>retimed</i> versions	261
23.1	Dataflow diagram for loading, showing 3 clock cycles and the path $D_0 \rightarrow S_8 \rightarrow S_7 \rightarrow S_6$ and $S_{11} \rightarrow S_{10}$ shaded	274
23.2	Dataflow diagram for replacing (1 clock cycle only), the paths $D_0 \rightarrow S_8$ and $D_1 \rightarrow \text{XOR} \rightarrow O_1$ shaded	274
23.3	Dataflow diagram for absorbing (1 clock cycle only), and the path $D_1 \rightarrow \text{XOR} \rightarrow O_1$ and $S_8 \rightarrow \text{XOR} \rightarrow S_8$ shaded	274
23.4	Dataflow diagram for permutation (1 clock cycle only), and the paths $S_5 \rightarrow \text{XOR} \rightarrow S_4$ and $S_9 \rightarrow S_8$ shaded	274
23.5	Small region of WAGE permutation: stages S_0, \dots, S_{10} with multiplexers, XOR and AND gates for the sponge mode	276
24.1	Design flow: datapath synthesis (automation framework) and detailed design space exploration	282

C.1	LFSR with feedback $y^4 + y^3 + y + \alpha$ over \mathbb{F}_{2^4} where generator where $\alpha = \omega^1 + \omega^2$ and ω is a root of $x^4 + x^3 + 1$ and constant(s) $\omega_0 = \alpha$	312
D.1	Two different tower field constructions	329

List of Tables

3.1	Specification parameters of WAGE	40
4.1	ASIC implementation results for Grain and Trivium found in literature	57
4.2	Post-PAR CMOS implementation results for WG-5 and WG-8	58
4.3	Post-PAR CMOS 65nm implementation results for the WG(16, 32) keystream generators	59
4.4	Pre-PAR CMOS 65nm implementation results for WG(16, 32) keystream generators	61
4.5	Pre-PAR CMOS 65nm implementation results for WGP modules with decimation 1: comparison of discrete component and constant array implementations from [120]	61
5.1	Part III examples	77
6.1	Structural similarities between LFSR, NLFSR, and FILFUN objects	82
6.2	Constructors for the FSR objects	83
6.3	Main functionality of the FSR package	84
6.4	Main functionality of the FSR package - continued	84
6.5	Miscellaneous and output formatting methods in the FSR package	85
6.6	Summary of key insights to the FSR package	92
7.1	Labels used for arguments in Tables 7.2 - 7.5	96
7.2	Main functionality of the FFCSA package	96
7.3	Main functionality of the FFCSA package - continued	97
7.4	Main functionality of the FFCSA package - continued	97
7.5	Main functionality of the FFCSA package - continued	98
7.6	Tower construction of $\mathbb{F}_{(2^2)^2}$	103
8.1	Main functionality of the WGcipher package	114

8.2	Main functionality of the WAGE package	119
9.1	Examples in Chapters 10 and 11	125
10.1	Main functionality of the GAPtoVHDL package	127
10.2	Main functionality of the GAPtoVHDL package - continued	128
10.3	Main functionality of the GAPtoVHDL package - continued	129
10.4	Classification of expressions in algebraic normal form given by the equation (10.1)	133
10.5	Summary of key insights to the GATtoVHDL package - listed chronologically	135
11.1	Classification of the FSR objects	137
11.2	Main functionality of the FSRtoVHDL package	139
11.3	Grain as a collection of FSRs	149
11.4	Connectors for the top-level datapath	151
11.5	Grain spreadsheet example	152
11.6	Main functionality of the FSRtoVHDL package - continued	153
11.7	Summary of key insights to the FSRtoVHDL package - listed chronologically	158
12.1	Examples in Chapters 13 and 14	161
12.2	Examples in Chapters 15 and 16	162
13.1	Main functionality of the CIRCUIT package - the SignalDomain attributes	168
13.2	Reference polynomials and their roots	168
13.3	GAP object SIGNAL with corresponding SignalDomain	172
13.4	The SIGNALs for three levels of a tower field	173
13.5	Main functionality of the CIRCUIT package - methods for the SignalDomain and the SIGNAL	175
13.6	Main functionality of the CIRCUIT package - methods for the SignalPkg	176
13.7	VHDL type definitions for different SIGNAL objects	178
13.8	VHDL type definitions for different SIGNAL objects - continued	178
13.9	Tower construction of $\mathbb{F}_{(2^2)^2}$ - left branch of diagram in Figure 13.3	179
13.10	Tower construction of $\mathbb{F}_{(2^2)^2}$ - roots and bases in GAP native representation	179
13.11	Summary of key insights to the CIRCUIT package - listed chronologically	184
14.1	Main functionality of the CIRCUIT package - the AlgFunctionality object	186

14.2	Short examples of partial and full binding	189
14.3	The CheckPortStrings test comparison	189
14.4	Main functionality of the CIRCUIT package - methods for the AlgFunctionality	195
14.5	The encoding for the inferred signals	196
14.6	The encoding for the submodules	198
14.7	Summary of key insights to the CIRCUIT package with new (partial) solutions from the AlgFunctionality - grouped conceptually	200
14.8	Summary of key insights to the CIRCUIT package with new (partial) solutions from the AlgFunctionality - continued	201
15.1	Main functionality of the CIRCUIT package - the AlgDesign object	206
15.2	Connectors for the tower field elements and vectors	218
15.3	Main functionality of the CIRCUIT package - methods for the AlgDesign	220
15.4	Summary of key insights to the CIRCUIT package with new solutions from AlgDesign	223
15.5	Summary of key insights to the CIRCUIT package with new solutions from AlgDesign - continued	224
16.1	Main functionality of the CIRCUIT package - writing the VHDL (sub)modules	227
18.1	\mathbb{F}_{2^7} normal elements, N-polynomials, and complexity of T C_T	247
18.2	Area results for the \mathbb{F}_{2^7} normal basis multiplication: Complexity C_T , and the <i>min. lib.</i> profile results before and after optimizations performed by the synthesis tools for the multiplication table T, the matrix U multiplier, and the rr_mo multiplier	248
18.3	\mathbb{F}_{2^7} primitive polynomials, their roots, and complexity of R, where R is the reduction matrix	250
18.4	Area results for the \mathbb{F}_{2^7} polynomial basis multiplication and inversion: Complexity C_R , and the <i>min. lib.</i> profile results before and after optimizations performed by the synthesis tools for matrix U multiplication and inversion in \mathbb{F}_{2^7}	251
20.1	Pre-PAR CMOS 65nm implementation results for the PB constant array WGP for \mathbb{F}_{2^8} with decimation exponent $d = 19$, using $f_{ref}(x) = x^8 + x^4 + x^3 + x^2 + 1$, where $f_{ref}(\omega) = 0$	258
22.1	Pre-PAR CMOS 65nm implementation results for the NB constant array WGP for \mathbb{F}_{2^7} with decimation exponent $d = 13$, using $f_{ref}(x) = x^7 + x + 1$, where $f_{ref}(\omega) = 0$	267
22.2	Pre-PAR CMOS 65nm implementation results for the PB constant array WGP for \mathbb{F}_{2^7} with decimation exponent $d = 13$, using $f_{ref}(x) = x^7 + x + 1$, where $f_{ref}(\omega) = 0$. The last column contains implementation results for the multiplication with ω_i	267
22.3	Pre-PAR CMOS 65nm implementation results for the NB multipliers and x^{33} modules for \mathbb{F}_{2^7} , using the reference defining polynomial $f_{ref}(x) = x^7 + x + 1$, where $f_{ref}(\omega) = 0$	269

22.4	WAGE permutation hardware area estimate [187]	271
23.1	Control table for WAGE	277
23.2	Specification parameters of WAGE SB	278
B.1	Loading into the shift register through data inputs D_4, D_3 and D_0	308
B.2	Examples of conversion of the field elements to HEX	309
B.3	Hex representation of WGP-16(X^d)	310
B.4	Hex representation of SB	310
B.5	Round constants of WAGE	310
C.1	LFSR with feedback $y^4 + y^3 + y + \alpha$ over \mathbb{F}_{2^4} with basis $B = [\beta_i] = [1, \alpha^7, \alpha^{14}, \alpha^6]$ where $\alpha = \omega^1 + \omega^2$ and ω is a root of $x^4 + x^3 + 1$.	311
C.2	LFSR with feedback $y^4 + y^3 + y + \alpha$ over \mathbb{F}_{2^4} where generator where $\alpha = \omega^1 + \omega^2$ and ω is a root of $x^4 + x^3 + 1$.	312
C.3	Element table for \mathbb{F}_{2^4} using basis $B = [\beta_i] = [1, \alpha^7, \alpha^{14}, \alpha^6]$ with generator α where $\alpha = \omega^1 + \omega^2$ and ω is a root of $x^4 + x^3 + 1$.	313
D.1	Reference polynomials and their roots	328
D.2	Tower construction of $\mathbb{F}_{((2^2)^2)^2}$ - left branch of diagram in Figure D.1 (see Example 13.3.1)	333
D.3	Tower construction of $\mathbb{F}_{((2^2)^2)^2}$ - right branch of diagram in Figure D.1	333
D.4	Tower construction of $\mathbb{F}_{((2^2)^2)^2}$ - roots and bases in GAP native representation for the left branch of diagram in Figure D.1 (see Example 13.3.1)	333
D.5	Tower construction of $\mathbb{F}_{((2^2)^2)^2}$ - roots and bases in GAP native representation for the right branch of diagram in Figure D.1	333

Part I

Introduction

Part I - Outline

1	Introduction and motivation	3
2	Overview	10

Chapter 1

Introduction and motivation

1.1 Introduction

Digital hardware is everywhere. Just over 15 years ago, “FPGAs on Mars” was the cover story of the Xilinx Xcell Journal [1]. Less than five years ago a NASA article “Tiny Microchips Enable Extreme Science” described a family of ASICs aboard the Juno spacecraft, currently in Jupiter’s orbit [2]. Back on Earth emerging technologies are reaching astronomical proportions. For example, the Internet of Things (IoT) has numerous applications and consists of countless different devices using different technologies with different capabilities. One invariant is the connectivity of IoT devices. Consequently, secure communications, and cryptographic hardware as a means of providing them, are faced with new challenges.

Cryptographic algorithms start with a specific application (e.g., wireless communications, RFID, etc.) and must consider hardware implementation requirements (e.g., constrained environment, high throughput, etc.), functional requirements (e.g., encryption, authentication, etc.) and security requirements (e.g., key size, resistance to known attacks, etc.). The algorithms intended for hardware implementations must be designed with a good trade-off between implementation efficiency and sufficient cryptographic strength. The implementations must follow good practices of hardware design while considering numerous design choices. Hardware design choices between sequential or pipelined datapath or fully exploited parallelism give great flexibility to custom hardware design, and allow the circuit to be tailored to the specific target application. For applications in constrained environments a small circuit area is prioritized.

Finite fields are widely used in cryptography. Algorithm design choices related to finite field arithmetic (FFA) include the field size, which arithmetic operations to use, how to represent the field elements, etc. Many of these decisions are made together. For example, using exponentiations to the powers of two and normal bases. Such exponentiations are implemented as simple cyclic shifts, i.e., rewiring which has negligible hardware area cost. This simple example illustrates the connection between an FFA parameter choice and efficient hardware implementation.

The aforementioned considerations lead the design decisions that must be made early in the design process. They will be called architectural decisions, and have a grave impact on the resulting hardware. As there are many parameters to be considered and set, an automation framework is needed.

This thesis describes a framework for the automated design, implementation and verification of finite field arithmetic hardware. The underlying motif throughout this work is “math meets hardware”. The automation framework is designed to bring an awareness of underlying mathematical structures to the hardware design flow. It is implemented in GAP [3], an open source computer algebra system that can work with finite fields and has symbolic computation capabilities. The framework is roughly divided into two phases, the architectural decisions and the automated design generation. The architectural decisions phase supports parameter search and produces a list of candidates. The automated design generation phase is invoked for each candidate, and the generated VHDL files are passed on to conventional synthesis tools. The candidates and their implementation results form the design space, and the framework allows rapid design space exploration (DSE) in a systematic way. In this thesis, design space exploration is focused on finite field arithmetic.

In the light of the completed eSTREAM competition [4] and the ongoing NIST Lightweight Cryptography Project (LWC) [5], this work focuses mainly on small finite fields. The framework is written in GAP, which uses a very efficient internal representation for small finite fields. Some features of the automation framework are the selection of field parameters for implementation, support for feedback shift register (FSR) based ciphers, arbitrary expressions over finite fields, tower field support, and on the fly submodule generation.

WAGE [6], a Round 2 candidate in the NIST LWC standardization competition, is a hardware oriented authenticated encryption scheme, built on top of the initialization phase of the Welch-Gong stream cipher [7]. The thesis illustrates the impact of hardware implementation results during the design process of WAGE. The parameter selection for WAGE was aimed at balancing the security and hardware implementation area, using hardware implementation results for many design decisions, for example field size, representation of field elements, etc. The components of WAGE are used as an example to illustrate different automation flows and demonstrate the design space exploration on a real-world algorithm.

The Welch-Gong (WG) stream cipher [7], which generates a keystream with proven randomness and cryptographic properties, was first proposed by Nawaz and Gong in 2005. The profile 2 (hardware applications with highly restricted resources) candidate WG-29 reached the phase 2 of the eSTREAM competition [4]. The WG stream cipher family is based on the Welch-Gong transformations on an m -sequence, produced by a linear feedback shift register (LFSR). The cipher instances are parametrized by the field size and the chosen LFSR. The automation framework in this thesis supports feedback shift registers and WG stream ciphers. The related design space exploration is focusing on the representation of field elements.

1.2 The automation framework and the design flow

This thesis presents a framework for automated design space exploration and datapath synthesis for finite field arithmetic. Figure 1.1 shows a detailed design flow with four phases:

- *the algorithm design*
- *the architectural decisions*
- *the automated design generation*
- *the design space exploration*

The algorithm design starts with a specific application (e.g., wireless communications, RFID, etc.) and must consider hardware implementation requirements (e.g., constrained environment, high throughput, etc.), functional requirements (e.g., encryption, authentication, etc.) and security requirements (e.g., key size, resistance to known attacks, etc.). Considering the application and functional requirements leads to a mathematical expression, which is then refined and modified based on hardware implementation requirements and security requirements until a suitable algorithm is designed. The focus of this thesis is design space exploration of finite field parameters and the synthesis of finite field expressions into hardware datapaths. Algorithm design choices related to finite fields are the field size, which arithmetic operations to use or avoid, how to represent the field elements, etc.

The architectural decisions phase starts with coarse architectural decisions (e.g., fully parallel datapath), and followed by the finite fields related design choices (e.g., representation of field elements). It supports the finite fields related parameter search and field constructions, and can generate expressions for implementation of arithmetic operations (i.e., basic building blocks for the datapath). The search algorithms produce a list of candidates, which differ in one or more parameter choices. From the perspective of a single candidate, the architectural decisions are finalized. *The automated design generation* is invoked for each candidate: the compilation algorithm parses the datapath expressions and the field structure to generate a fully functional synthesizable hardware module. *The automated design generation and the architectural decisions* form the main body of the design automation framework in this thesis.

The fully functional module, shown at the end of automated design generation in Figure 1.1, consists of generated VHDL files, testvectors and configuration files, which are passed on to conventional synthesis tools. The candidates and their implementation results form the design space, and the framework allows rapid *design space exploration* in a systematic way. In this thesis, design space exploration is focused on finite field arithmetic.

The algorithm design phase is usually completed before hardware implementations begin. This thesis includes a unique opportunity showing the impact of hardware implementation on algorithm design: the WAGE authenticated encryption scheme [6]. The small hardware area determined the following parameters of WAGE: field size, basis, and even which nonlinear components to use. The *architectural decisions – automated design generation – design space exploration loop* can reveal interesting findings that lead to modifications of existing algorithms. For example, new

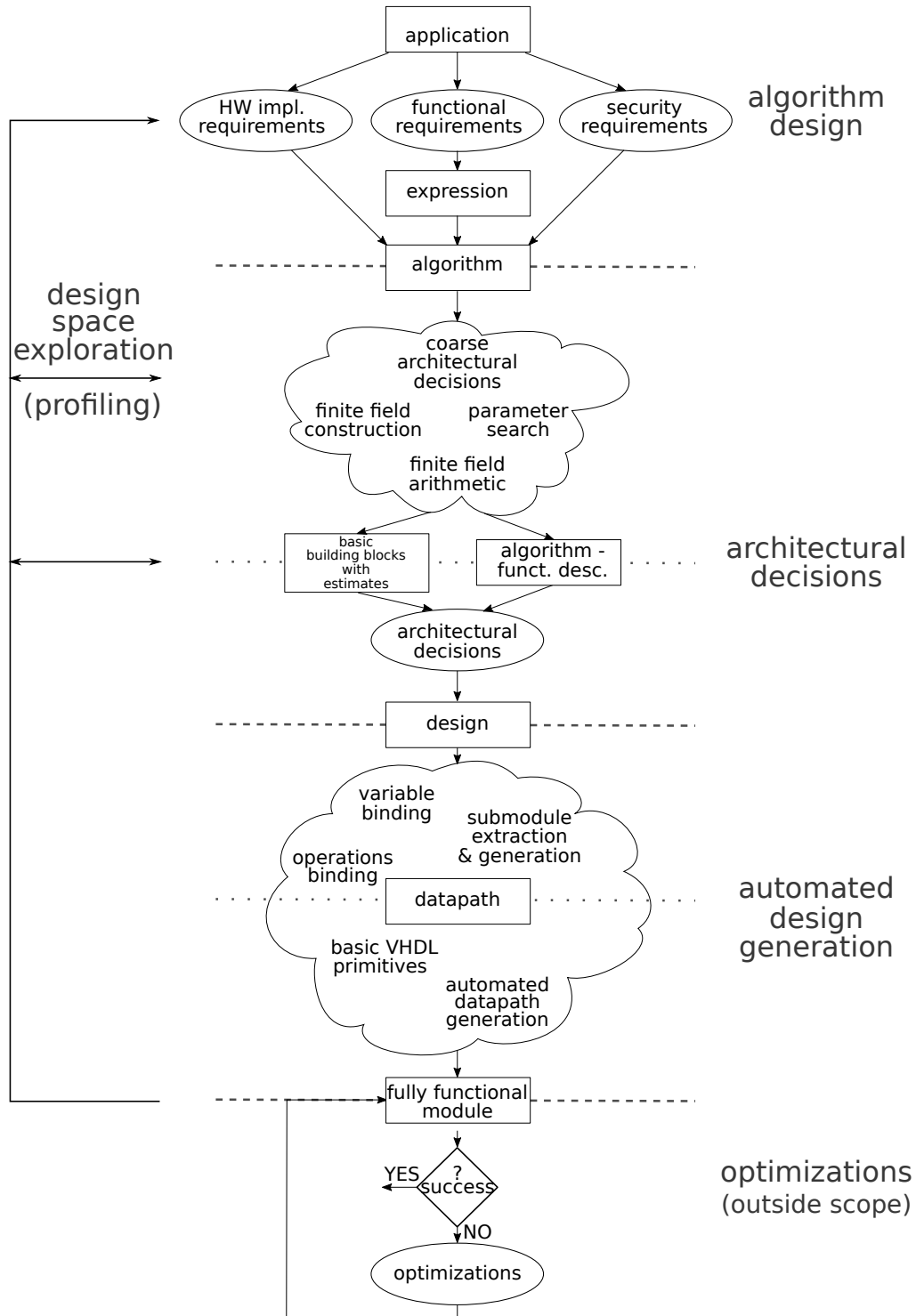


Figure 1.1: Design flow for automated DSE and datapath synthesis

LFSR polynomials were suggested during the implementation of the WG-16 stream cipher in [8]; it was possible to increase the number of the terms in the LFSR feedback at no cost to hardware.

The optimization phase, shown at the bottom of Figure 1.1, is outside the scope of this thesis, but deserves a short discussion. The aforementioned WG-16 stream cipher is defined over $\mathbb{F}_{2^{16}}$ and not considered to be lightweight. The hardware implementation of WG-16 benefits from pipelining and a tower field construction. Pipelining is a hardware optimization, and finding an adequate pipelining granularity can be a tedious process. Given a specific target, e.g., a clock period, the pipelining granularity can be modified in an automated *optimizations–design space exploration loop*, until the target is met. A related example of a hardware optimization is register retiming, which has the effect of fine-tuning a pipelined datapath. Hardware optimizations and design space exploration present an interesting future research direction.

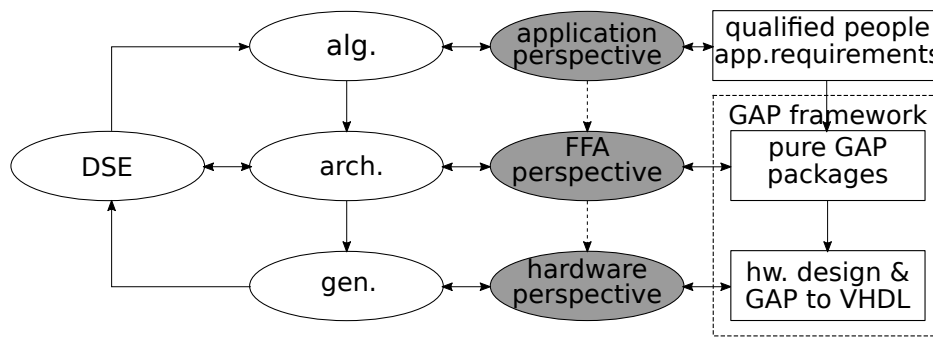


Figure 1.2: Simplified design flow diagram and the perspectives (shaded grey)

A simplified version of the design flow diagram, with abbreviations *alg.*, *arch.*, *gen.*, and *DSE* for *the algorithm design*, *the architectural decisions*, *the automated design generation*, and *the design space exploration*, respectively, is shown on the left side of Figure 1.2. On the right side of Figure 1.2 is the automation framework, designed to address the questions arising in a particular phase. In the middle of Figure 1.2 are the perspectives from which to tackle the problems identified with each phase:

- application perspective for *the algorithm design*
- finite field arithmetic perspective for *the architectural decisions* and *the design space exploration*
- hardware perspective for *the automated design generation*

The perspectives are natural and together they highlight the underlying motif throughout this thesis: “math meets hardware”. For a given phase, there is always a certain amount of overlap from other perspectives. The following detailed discussion about the challenges met in each phase will clarify these overlaps. The perspectives are shaded grey in Figure 1.2 as they are discussed in this section.

The architectural decisions phase – finite field arithmetic perspective

The tasks of architectural decisions are accomplished by parameter search, field constructions and algorithms for implementing a finite field operation. More specifically: by searching for (special) elements and (special) polynomials, generating matrices and bases, and finally, obtaining symbolic expressions. The entire automation framework is implemented in GAP, an open source computer algebra system that can work with finite fields and has symbolic computation capabilities. The following discussion summarizes the nature and the extent of hardware perspective involved in architectural decisions.

The algorithm itself affects the top-level architecture, where exploration of parallelism is very important and basic design decisions must be made, e.g., should the design be sequential to minimize the area, or is the throughput so important that a fully pipelined architecture is preferred. Additional considerations come into play for cryptographic hardware (e.g., lightweight cryptography, which is aiming for minimal design while avoiding both sequential and pipelined circuits). Sequential circuits exploit resource sharing to minimize the implementation area. Resource sharing comes at a cost (area increase): it infers multiplexers and registers for intermediate results, and can complicate the control circuit. While beneficial for large finite fields (e.g., for elliptic curve cryptography), this trade-off might fail for small finite fields (e.g., for lightweight cryptography). The algorithm designers follow well established practices, for example: LFSRs are very efficient in hardware, small Hamming weights will reduce the area and/or delay of the circuit, finite field inversions are expensive and should be avoided if possible, etc.

Mathematical parameters can be chosen to optimize the hardware. Well known examples for implementation of finite field applications are the use of trinomials and normal polynomials yielding optimal normal bases (ONB) as field defining polynomials. Trinomials are a good choice because they ensure a small delay for reduction and the ONBs ensure a smaller area complexity for the multipliers. However, neither of them exists for all finite fields, and a decision must be made whether this should affect the choice of the finite field or not. Another example is choosing the coefficients of the LFSR feedback polynomial that can be implemented efficiently in hardware. The established practice dictates choosing all the coefficients, except for the constant term, from the prime subfield, and then, given a chosen basis, the constant term is chosen to minimize either the delay or the area of the constant multiplier.

The automated design generation phase – the hardware perspective

The automated design generation phase faces many challenges related to finite field arithmetic. For example, a very simple mind-leap is to view a finite field as a vector space by representing the field w.r.t. to a selected basis. However, while GAP can do both, VHDL always needs a specific representation in terms of a basis, i.e., VHDL can only use a vector or an array of elements. This becomes especially challenging when trying to implement tower field arithmetic. Other examples are vectors of finite field elements without an interpretation basis, i.e., they do not belong to an extension field. Examples of this are the school-book two-step classic multiplication, which

first produces an intermediate result whose length is greater than the length of the basis, or the Grain stream cipher, which uses elements from two distinct shift registers as an input to a filtering function.

The next problem is the finite field arithmetic itself. The expressions contained by the algorithm involve operations, e.g., finite field multiplication or exponentiation. In VHDL, submodules, e.g., a multiplier, are used to implement all occurrences of a particular arithmetic operation. Given only the algorithm, the submodules must be extracted, perhaps implemented, and then bound to the operations via component instantiations. The phrase “perhaps implemented” is used because there are two options: a) the submodule is already implemented and ready to use (i.e., it is available as a part of a library), and b) the submodule will be generated on the fly. The latter requires the ability to generate the mathematical expressions needed for its implementation (e.g., using symbolic computation capabilities of GAP).

The design space exploration – finite field arithmetic perspective

Finite field arithmetic allows a vast design space, composed of the candidates found during the architectural decisions phase. *The design space exploration* relies on selected metric(s) to choose good design options from the set under exploration. The most commonly used metrics for hardware implementations are combinational delay, clock period or frequency, area, throughput, and other derived optimality metrics.

Values such as the delay and the area can be estimated theoretically by evaluating critical path delay and area of a module in terms of two-input logic gates. Alternatively, synthesis tools can be used to obtain more accurate, technology-dependent values. The first option will be called offline profiling and the second online profiling.

Chapter 2

Overview

2.1 Overview of the automation framework

The rightmost part of Figure 2.1, shaded grey, shows the automation framework, written in GAP. The packages that constitute the automation framework are shown Figure 2.2. The portion of the GAP framework which was designed for the needs of *the architectural decisions* consists of two basic packages, shown on the top: FSR Feedback Shift Registers, and FFCSA Finite Field Constructions, Search and Algorithms. On the left are two case-study packages, WG and WAGE, also a part of *the architectural decisions*. The most crucial part is the FFCSA package, which allows to choose optimal mathematical parameters for the implementation: the field defining polynomials, bases, transition matrices, and finally algorithms that produce expressions for arithmetic operations.

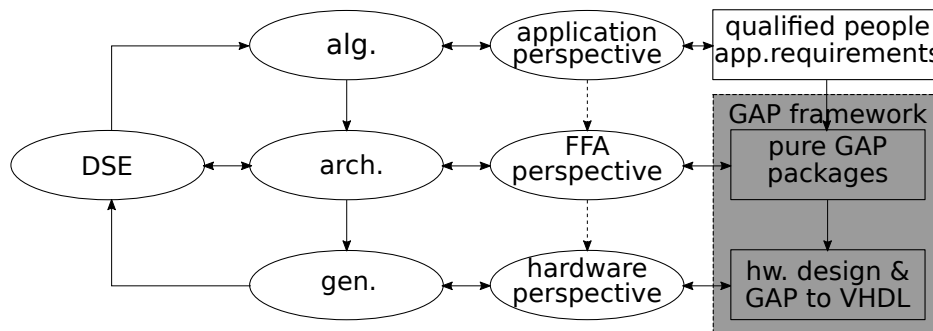


Figure 2.1: Simplified design flow diagram and the automation framework (shaded grey)

Below FFCSA is the first package used for *the automated design generation*, the GAPtoVHDL package (Figure 2.2). It provides common VHDL functionality for writing VHDL packages, concurrent VHDL statements, combinational assignments, component instantiations, registers, multiplexers, etc. The FSRtoVHDL package is, as its name suggests, specialized for the design generation of FSR objects, and heavily relies on packages FSR and GAPtoVHDL. Last is the package CIRCUIT, which is very complex, but capable of writing arbitrary datapaths defined over arbitrary finite fields. Three distinctive features of the CIRCUIT package are the structure of finite fields, tower field support, and on the fly submodule generation.

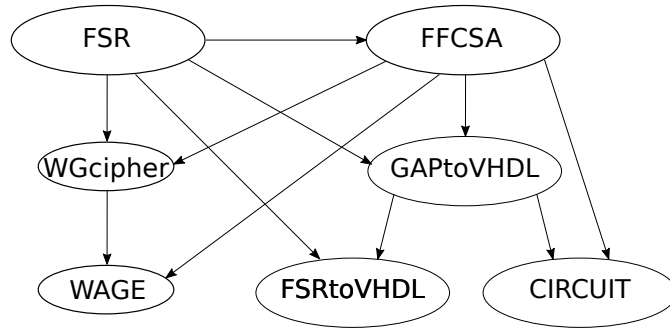


Figure 2.2: The automation framework: packages

The user provides a set of expressions to be implemented. The user may also provide the partial or complete field constructions to be used. The framework automatically performs design space exploration for the mathematical parameters not provided by the user.

2.2 WAGE

WAGE [6], a Round 2 candidate in the NIST LWC standardization competition [10], is a hardware oriented authenticated encryption scheme, built on top of the initialization phase of the WG stream cipher. The thesis illustrates the impact of hardware implementation results during the design process of WAGE.

Disclaimer 2.1: The WAGE authenticated encryption scheme

The WAGE authenticated encryption scheme is a joint work of the members of the ComSec Lab: Mark Aagaard, Riham AlTawy, Guang Gong, Kalikinkar Mandal, Raghvendra Rohit, and Nusa Zidaric (author of this thesis). The authorship above is alphabetically ordered. For the purpose of this thesis, it is important to specify my contribution to this work. In the WAGE sections of this thesis, there will always be a disclaimer, specifying my contributions.

2.3 Contributions of the author

This work presents a novel framework for datapath synthesis for finite field arithmetic, which is implemented in GAP from the ground-up. The first *architectural decisions–automated design generation flow* through the automation framework is focusing on synthesis for ciphers based on feedback shift registers (FSR). This part of the automation framework was motivated by the design space exploration for the WG stream cipher family. Major contributions to implemented FSR design flow are (i.) recognizing and exploiting structural similarities between the linear feedback shift registers (LFSRs), nonlinear feedback shift registers (NLFSRs), and filtering functions from a mathematical and a hardware perspective, and (ii.) introduction of a regular and external step for clocking the FSR objects as self-contained or with an external input. This allows a cipher to be modelled as a collection of basic blocks (LFSRs, NLFSRs and filtering functions). The packages developed for the FSR-based ciphers are the first to implement both LFSRs and NLFSRs over prime and extension fields in software (GAP package FSR in Figure 2.2). Furthermore, they provides (limited) synthesis of FSR based ciphers using VHDL (FSR–FSRtoVHDL tandem in Figure 2.2), which significantly reduces the human effort.

The second *architectural decisions–automated design generation flow* of the automation framework is the synthesis of arbitrary expressions over arbitrary finite fields. The first major challenge was the encoding of finite field structures in a way that (i.) allows switching between finite fields and vector spaces, and (ii.) is hierarchical to support tower fields. Another challenge was the on the fly submodule generation (for datapath operations), which was solved by (i.) the aforementioned encoding of finite field structures, and (ii.) binding of the expressions to the appropriate finite fields or vector spaces. Integrating the tower field support into the datapath synthesis is extremely useful as it increases the design space which can be covered by the proposed automation framework.

The framework offers a systematic approach to DSE from FFA perspective. The delay and the area can be estimated theoretically or using synthesis tools to obtain technology-dependent values. The logic synthesis implementation results for a simple expression defined over a small tower-field can be collected quickly: e.g., for $\mathbb{F}_{((2^2)^2)^2}$ with 720 constructions, the modules can be generated, simulated and synthesized in just over 3 hours. The automation framework significantly reduces human effort, saves time, and increases productivity. Furthermore, it enables researchers with little knowledge of hardware, to generate synthesizable VHDL code, using only a short GAP template.

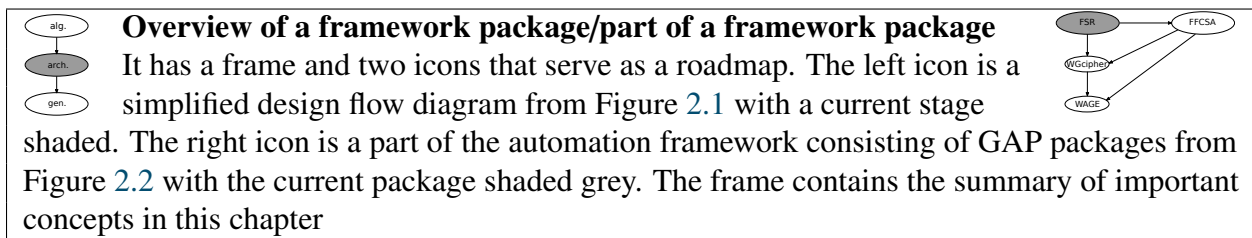
My contributions in WAGE design are the parameter search, (automated) hardware implementations during the algorithm design of WAGE, and contribution to the loading sequence and input ports, and tag extraction sequence and output ports. The final chapter contains the hardware design and manual implementation of the WAGE datapath.

2.4 Roadmap

- **Critical path through the thesis:** everything
- **Shortest viable path through this thesis:** each chapter in Parts III and IV contains an **Overview** box (see below). Chapters also contain **Summary** sections, which contain tables with summarized key insights (Key 2.1). The summary tables contain links to the actual Keys.

Visual aid for orientation

Figures 1.1 and 2.1 will be used as roadmaps within the chapters. When more details are needed, a magnified section of the design flow in Figure 1.1 will be used as well.



Key 2.1: This is a key insight

The main text includes key insights: summaries of very important concepts, crucial for design of the automation framework. They look like this.

■ **Implementation detail:** This is an implementation detail. It can be skipped. ■

The main text includes many implementation details using a smaller font. It has a bold title, and starts and end with ■, and can be skipped. Some of the examples have arrows \longleftrightarrow and \longleftarrow and can be skipped. Their purpose is to keep this document self contained. Some further examples were moved to appendix. Examples without the arrows are important for the explanation of concepts, and should not be skipped. Each introductory chapter (grey chapters on the roadmap in Figure 2.3) contains a table of examples, including the ones that were moved to an appendix section.

Thesis organization

As this work covers a broad topic, its is divided into many small chapters. Figure 2.3 shows the relationship of the individual chapters to the design flow for automated DSE and datapath synthesis. The chapters are organized into five parts as follows:

- **Part II - Background and related work**
This part contains the background (Chapter 3) and the related work (Chapter 4). Both cover finite field arithmetic, hardware implementations, and ciphers defined over finite fields.

- Part III - The architectural decisions phase
This part starts with an introductory chapter, which covers the overview and the roadmap for Part III, and an introductory discussion on coarse architectural decisions (Chapter 5). The remaining chapters follow the GAP packages in the automation framework: the FSR package, (Chapter 6), the FFCSA package (Chapter 7), and the Case study for WG and WAGE (Chapter 8).
- Part IV - The automated design generation phase
This part starts with an introductory chapter, which covers the overview and the roadmap (Chapter 9). Next are the GAP packages GAPtoVHDL (Chapter 10) and FSRtoVHDL (Chapter 11), followed by the CIRCUIT package. The CIRCUIT package is divided into five chapters based on the concepts they cover: the overview and roadmap (Chapter 13), the arbitrary finite fields (Chapter 13), the functional description of the algorithm (Chapter 14), transforming the functional description into a VHDL-ready design (Chapter 15), and generating VHDL for the datapath (Chapter 16).
- Part V - Design space exploration
This part starts with an overview of DSE from mathematical perspective, discussion on different profiles, and representation of implementation results (Chapter 17). The remaining DSE chapters are divided based on what they are profiling: DSE of basic building blocks (18), DSE of arbitrary circuits (Chapter 19), and DSE of WG (Chapter 20)
- Part VI - WAGE
This part start with the overview chapter (Chapter 21). Next chapter covers WAGE algorithm design (Chapter 22), which is in fact the highest DSE loop in Figure 1.1. Last chapter in Part VI shows the manual hardware design of the WAGE datapath (Chapter 23).

Each part has its own table of contents. Parts III-VI have their own introductory chapters, shaded grey on the roadmap in Figure 2.3. There are two *architectural decisions – automated design generation flows* through the automation framework:

- the first *architectural decisions – automated design generation flow* is focusing on synthesis for ciphers based on feedback shift registers. In Figure 2.3 this flow is shown in the middle-left column, consisting of roadmap chapters “ch.5 FSR” and “ch.10 FSRtoVHDL”
- the second *architectural decisions – automated design generation flow* is the synthesis of arbitrary expressions over arbitrary finite fields. In Figure 2.3 this flow is shown in the middle-right column, consisting of roadmap chapters “ch.6 FFCSA”, “ch.12 arb. fields” (arbitrary finite fields), “ch.13 Alg. Fun.” (functional description of the algorithm), “ch.14 Alg. Design”, and “ch.15 Circuit VHDL”. These are the chapters that belong to the GAP package CIRCUIT.

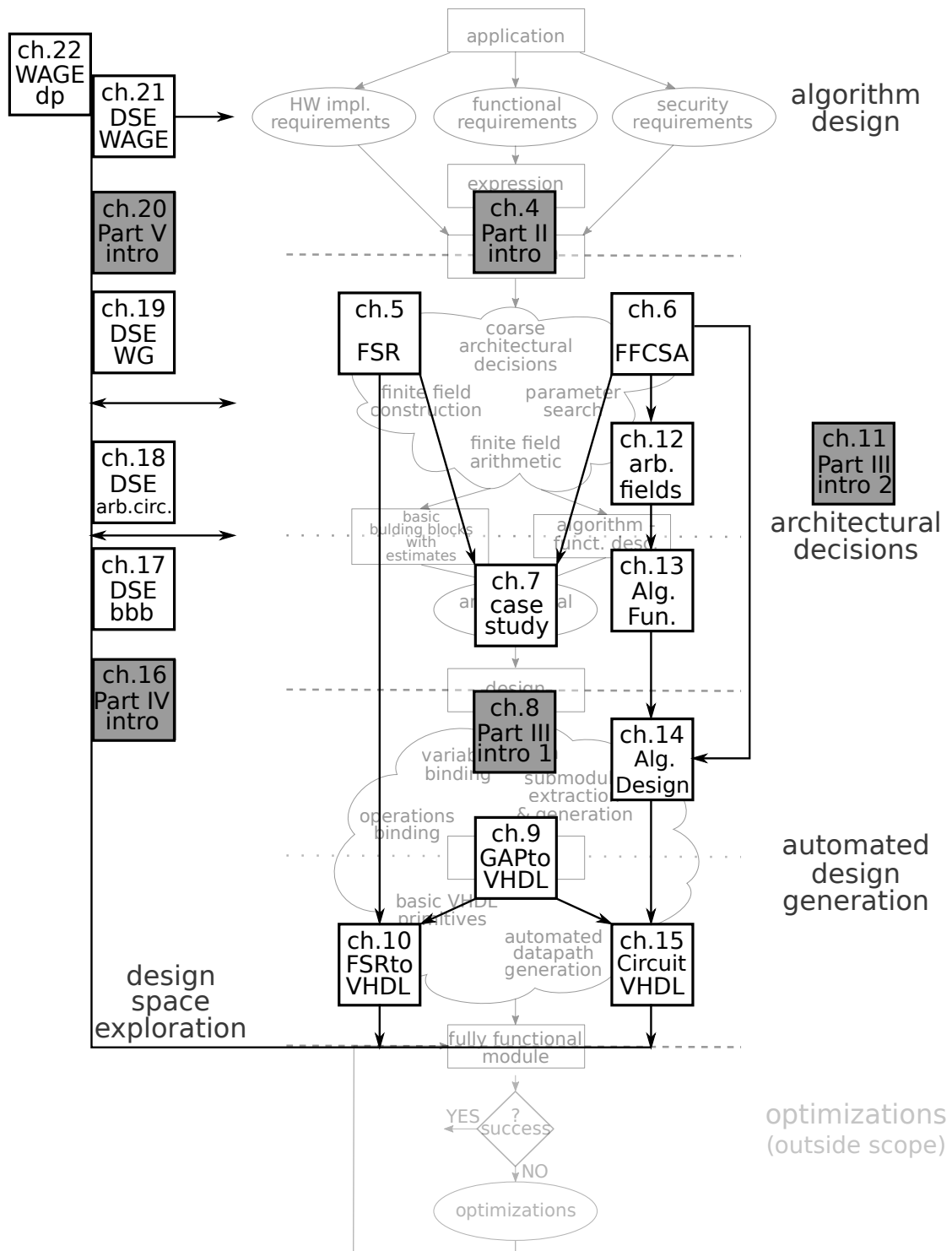


Figure 2.3: Roadmap through the design flow for automated DSE and datapath synthesis

Part II

Background and related work

Part II - Outline

3	Background	18
4	Related work	54

Chapter 3

Background

3.1	Finite fields	18
3.2	Selected symmetric-key cryptographic primitives	27
3.3	Computer hardware	44

3.1 Finite fields

This section covers basic definitions and properties of finite fields, extension fields and their defining polynomials, representations of field elements using different bases, and finally the notion of the trace function. For brevity, some definitions, e.g., definition of a group (Definition A.1), is presented in the Appendix A. Extensive literature on the subject exists, for example [11] or [12]. Further properties of finite fields will be presented in the main text, and some advanced concepts and properties will be presented in Appendices A and C.3 when needed.

3.1.1 Finite field constructions

Basic definitions

Definition 3.1 [13] *A nonempty set F , together with two binary operations addition “+” and multiplication “ \cdot ” is a field $\mathcal{F} = (F, +, \cdot)$, if*

- i. $(F, +)$ is a commutative group with (additive) identity 0*
- ii. $(F \setminus \{0\}, \cdot)$ is a commutative group with (multiplicative) identity 1*
- iii. multiplication is distributive over addition: $\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma \ \forall \alpha, \beta, \gamma \in F$*

For addition and multiplication, the following short notations are used for an arbitrary $\alpha \in \mathcal{F}$:

$$n\alpha = \underbrace{\alpha + \alpha + \cdots + \alpha}_n \quad \text{and} \quad \alpha^n = \underbrace{\alpha \cdot \alpha \cdots \alpha}_n, \quad (3.1)$$

where n is a positive integer. If the underlying set F has a finite number of elements, then \mathcal{F} is a *finite field*. The number of elements in a finite field is also called the *order of the field*, denoted $|\mathcal{F}|$.

The generalization of Fermat's little theorem (Lemma 2.1.16 in [11], Corollary 3.2 in [14]), states that every non-zero element α of a finite field of order q satisfies the following identity

$$\alpha^q = \alpha \quad (3.2)$$

Equation (3.2) also implies that $\alpha^{q-1} = 1$ and that multiplicative inverse can be computed as α^{q-2} . The smallest positive integer j , for which $\alpha^j = 1$, is called the *order of the element* $\alpha \in \mathcal{F}$ (Definition 2.1.40 in [15]). For \mathcal{F} of order q , the order of an arbitrary field element is always a divisor of $q - 1$. Elements of order $q - 1$ and are called a *primitive elements*. The multiplicative group¹ $(\mathcal{F} \setminus \{0\}, \cdot)$ is cyclic, with primitive elements as its generators. For cyclic group see Definition A.2 in Appendix A.

Let $\mathcal{F} = (F, +, \cdot)$ be a field. A subset $K \subseteq F$ together with the operations $+$ and \cdot forms a *subfield* $\mathcal{K} = (K, +, \cdot)$ of \mathcal{F} , if \mathcal{K} is itself a field with respect to the two operations. The inclusion symbol is used to denote \mathcal{K} is a subfield of \mathcal{F} , namely $\mathcal{K} \subset \mathcal{F}$. The field \mathcal{F} is also called an *extension field* of \mathcal{K} , denoted \mathcal{F}/\mathcal{K} . If $\mathcal{K} \neq \mathcal{F}$ and $\mathcal{K} \neq \{0\}$, then \mathcal{K} is a *proper subfield* of \mathcal{F} . A field that has no proper subfields is called a *prime field*. When \mathcal{F} is considered as a finite dimensional vector space over \mathcal{K} (see Definition A.5 in Appendix A), then the dimension of \mathcal{F} over \mathcal{K} is called *degree of extension*, denoted $[\mathcal{F} : \mathcal{K}]$ (Definition 1.3.4 in [16]). For a finite field \mathcal{F} with q elements, notation \mathbb{F}_q is used. If $q = p^m$, where p is a prime or a prime power and m is a positive integer, then \mathbb{F}_q is an extension field of F_p , and the degree of extension is m .

Theorem 3.1 [*Subfield criterion, Theorem 2.6 in [11]*] *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.*

Notion of subfields of a finite field adopts a “top-down” point of view. A “bottom-up” approach reveals the following: for a composite integer $m = n_1 \cdots n_k$, where $n_i, i = 1, \dots, k$ are positive integers (not necessarily primes), it is possible to 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 . Instead of constructing \mathbb{F}_{p^m} as a single extension of degree $[\mathbb{F}_{p^m} : \mathbb{F}_p] = m$ over the prime field, the field is constructed in k steps, building an extension $\mathcal{K}_i/\mathcal{K}_{i-1}$ of degree $[\mathcal{K}_i : \mathcal{K}_{i-1}] = n_i$ at step i , where n_i is a factor in decomposition of m , starting with $\mathcal{K}_0 = \mathbb{F}_p$. \mathcal{K}_{i-1} will be called the *base field* for extension $\mathcal{K}_i/\mathcal{K}_{i-1}$ (to differentiate it from the prime field). The base field \mathcal{K}_{i-1} is now embedded in \mathcal{K}_i as a subfield, $\mathcal{K}_{i-1} \subset \mathcal{K}_i$. The result of the procedure described above is a sequence of subfields:

$$\mathbb{F}_p = \mathcal{K}_0 \subset \mathcal{K}_1 \subset \cdots \subset \mathcal{K}_{k-1} \subset \mathcal{K}_k = \mathbb{F}_{(\dots((p^{n_1})^{n_2})\dots)^{n_k}} \cong \mathbb{F}_{p^m} \quad (3.3)$$

with their corresponding orders

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

¹notation \mathcal{F}^* is also used for $(\mathcal{F} \setminus \{0\}, *)$, where $*$ is the multiplication operation

Note at this point, 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 to the current extension, for example $\mathbb{F}_{(p^{n_1})^{n_2}} \cong \mathbb{F}_{p^{n_1 n_2}}$. Also, since $m = n_1 \cdots n_k$, the product of extension degrees equals the degree of extension of \mathbb{F}_{p^m} over \mathbb{F}_p (see Theorem 1.84 in [11], Satz 6 in [17])

$$[\mathcal{K}_k : \mathcal{K}_0] = [\mathcal{K}_k : \mathcal{K}_{k-1}] \cdots [\mathcal{K}_2 : \mathcal{K}_1] \cdot [\mathcal{K}_1 : \mathcal{K}_0] \quad (3.4)$$

The sequence of fields in expression (3.3) will be referred to as a *tower field* or a *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 composition of m ; we use the factors of m to build the tower of extensions.

Irreducible polynomials and field constructions

Let $\mathcal{K}[x]$ be a set of polynomials 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 m be the index of the last nonzero coefficient in f , i.e. $a_m \neq 0$ but $a_j = 0 \forall j > m$. Then a_m is called the *leading coefficient* of f and f is a polynomial of degree m . A polynomial with $a_m = 1$ 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 $-\infty$.

A polynomial $f \in \mathcal{K}[x]$ is *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 (Definition 1.57 in [11]). An element $\alpha \in \mathcal{K}$ is a *root* of a nonzero polynomial $f \in \mathcal{K}[x]$ if $f(\alpha) = 0$. A polynomial of degree m will have at most m distinct roots in \mathcal{K} (or any of its extensions). The lowest degree monic polynomial in $\mathcal{K}[x]$, having a root α is called the *minimal polynomial* of α (Definition 3.16 in [14]). The irreducible polynomial f can then be used to construct the finite field \mathcal{F}/\mathcal{K} , where $[\mathcal{F} : \mathcal{K}] = m$. The finite field $\mathcal{F} = (F, +, \cdot)$ contains polynomials of degree at most $m - 1$ with coefficients from \mathcal{K} :

$$F = \left\{ \sum_{i=0}^{m-1} a_i \alpha^i ; a_i \in \mathcal{K} \right\}$$

Details on the construction of \mathcal{F}/\mathcal{K} and on addition and multiplication of polynomials can be found in Section A.0.1 in Appendix A.

Noting that a given polynomial, which is irreducible over \mathcal{K} , has no roots in \mathcal{K} , an extension field \mathcal{F}/\mathcal{K} can be constructed by adjoining the root(s) of the irreducible polynomial to the base field \mathcal{K} . The field \mathcal{F} is the smallest extension field containing the root(s) of the polynomial. The polynomial whose root was used is called the *defining polynomial* of the extension field. As already mentioned, a polynomial of degree m can have at most m distinct roots. The finite fields

obtained by adjoining different distinct roots of an irreducible polynomial over \mathcal{K} are isomorphic. Furthermore, the larger the degree m , the larger is the number of irreducible polynomials of degree m over \mathcal{K} , and using different irreducible polynomials again yields isomorphic extensions. To construct a tower field as shown in expression (3.3), an irreducible polynomial of degree n_i is required to build the extension $\mathcal{K}_i/\mathcal{K}_{i-1}$. An irreducible polynomial having a primitive element as its root is called a *primitive polynomial*. Recall that primitive elements in \mathcal{F} have order $q - 1$, where $|\mathcal{F}| = q$.

Bases, conjugates and trace function

Let $[\mathcal{F} : \mathcal{K}] = m$, then any set $\{\alpha_0, \alpha_1, \dots, \alpha_{m-1}\}$ of m linearly independent elements $\alpha_i \in \mathcal{F}$ forms a *basis* of \mathcal{F} over \mathcal{K} . Notation $B_{\mathcal{F}/\mathcal{K}}$ will be used for a basis of \mathcal{F}/\mathcal{K} . A different ordering of the basis elements is considered as a distinct basis. For further details see [11, 12].

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 $\mathbb{F}_{q^m}/\mathbb{F}_q$). 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 (Corollary 2.11 in [11]). 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:

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

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*: $\alpha, \alpha^q, \alpha^{q^2}, \dots, \alpha^{q^{m-1}}$ (Theorem 2.14, Definition 2.17 in [11]). If the roots of the defining polynomial are linearly independent, then they generate a *normal basis* $\{\alpha, \alpha^q, \alpha^{q^2}, \dots, \alpha^{q^{m-1}}\}$ of \mathbb{F}_{q^m} over \mathbb{F}_q (Definition 2.32 in [11]). Each element $A \in \mathbb{F}_{q^m}$ can be represented in normal basis as:

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

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 [11], and for all i , $0 \leq i \leq m-1$, σ_i can be obtained as a composition of σ_1 , namely $\sigma_i = \sigma_1^i$ (Theorem 2.1.76, Remark 2.1.77 in [15]).

For every prime power q and every integer $m \geq 1$, \mathbb{F}_{q^m} has a normal basis over \mathbb{F}_q (Theorem 5.2.1 in [15]). 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* or N-polynomial. Two normal elements are said to be *different* if they are not conjugated. A normal basis can be found by finding a normal element using Theorem 3.2.

Theorem 3.2 [Theorem 5.2.11(1.) in [15]] For any $\alpha \in \mathbb{F}_{q^m}$, define

$$T_\alpha(x) = \sum_{i=0}^{m-1} \sigma^i(\alpha)x^i \in \mathbb{F}_{2^m}[x]$$

$\alpha \in \mathbb{F}_{q^m}$ is normal over \mathbb{F}_q if and only if $\gcd(T_\alpha(x), x^m - 1) = 1$ in $\mathbb{F}_{q^m}[x]$.

Different normal elements generate different normal bases. The particular normal basis has an impact on complexity of the arithmetic performed with field elements in normal basis representation. Normal bases can be evaluated using complexity C_T defined as the number of nonzero elements in multiplication table T (Definition 5.3.1 in [15]). Multiplication table T is an $(m \times m)$ matrix $T = [t_{ij}]$ over \mathbb{F}_q , defined for a particular normal element $\alpha \in \mathbb{F}_{2^m}$, in such a way that the coefficients t_{ij} satisfy:

$$\alpha \cdot \alpha^{q^i} = \sum_{j=0}^{m-1} t_{ij} \alpha^{q^j} \quad \text{for } 0 \leq i \leq m-1. \quad (3.5)$$

The complexity is bounded by $2m-1 \leq C_T \leq m^2 - m + 1$, and when $C_T = 2m-1$ the basis is said to be *optimal normal basis*, abbreviated ONB (Definition 5.2.5 in [15]).

Another interesting function involving conjugates of an element, is the *trace function* (Definition 2.22 in [11]). Let $\mathcal{F} = \mathbb{F}_{q^m}$ and $\mathcal{K} = \mathbb{F}_q$, then for element $\alpha \in \mathcal{F}$ the trace is defined as:

$$\text{Tr}_{\mathcal{K}}^{\mathcal{F}}(\alpha) = \sum_{i=0}^{m-1} \alpha^{q^i} = \alpha^{q^0} + \alpha^{q^1} + \dots + \alpha^{q^{m-1}} \quad (3.6)$$

The mapping $\text{Tr}_{\mathcal{K}}^{\mathcal{F}} : \mathcal{F} \rightarrow \mathcal{K}$ is called the trace of the element $\alpha \in \mathcal{F}$ with respect to the underlying subfield \mathcal{K} . If \mathcal{K} is a prime subfield, $\text{Tr}_{\mathcal{K}}^{\mathcal{F}}$ is called *absolute trace*. Note that the number of terms in the expression above equals the degree of extension $m = [\mathcal{F} : \mathcal{K}]$, i.e. it runs through all conjugates of α . The trace is independent of the chosen basis. Below are some useful properties of trace function (Theorem 2.23 in [11]).

Theorem 3.3 Let $\mathcal{F} = \mathbb{F}_{q^m}$ and $\mathcal{K} = \mathbb{F}_q$. Then the trace function $\text{Tr}_{\mathcal{K}}^{\mathcal{F}}$ satisfies the following properties:

1. $\text{Tr}_{\mathcal{K}}^{\mathcal{F}}(\alpha + \beta) = \text{Tr}_{\mathcal{K}}^{\mathcal{F}}(\alpha) + \text{Tr}_{\mathcal{K}}^{\mathcal{F}}(\beta)$ for all $\alpha, \beta \in \mathcal{F}$
2. $\text{Tr}_{\mathcal{K}}^{\mathcal{F}}(c\alpha) = c\text{Tr}_{\mathcal{K}}^{\mathcal{F}}(\alpha)$ for all $c \in \mathcal{K}, \alpha \in \mathcal{F}$

3. $\text{Tr}_{\mathcal{K}}^{\mathcal{F}}$ is a linear transformation from F onto K ,
where both F and K are viewed as vector spaces over K
4. $\text{Tr}_{\mathcal{K}}^{\mathcal{F}}(a) = ma$ for all $a \in K$
5. $\text{Tr}_{\mathcal{K}}^{\mathcal{F}}(\alpha^q) = \text{Tr}_{\mathcal{K}}^{\mathcal{F}}(\alpha)$ for all $\alpha \in F$

Theorem 3.4 [Theorem 2.26 in [11]] Let \mathcal{K} be a finite field, let \mathcal{F} be a finite extension of \mathcal{K} and \mathcal{E} a finite extension of \mathcal{F} . Then

$$\text{Tr}_{\mathcal{K}}^{\mathcal{E}}(\alpha) = \left(\text{Tr}_{\mathcal{F}}^{\mathcal{E}} \circ \text{Tr}_{\mathcal{K}}^{\mathcal{F}} \right) (\alpha) = \text{Tr}_{\mathcal{K}}^{\mathcal{F}} \left(\text{Tr}_{\mathcal{F}}^{\mathcal{E}}(\alpha) \right), \quad \text{for all } \alpha \in \mathcal{E}.$$

Last interesting notion is that of dual bases. For any given basis of \mathcal{F}/\mathcal{K} there exists a unique *dual basis* (Theorem 1.1 in [12]). Two bases $\{\alpha_1, \alpha_2, \dots, \alpha_{m-1}\}$ and $\{\beta_1, \beta_2, \dots, \beta_{m-1}\}$ of \mathcal{F}/\mathcal{K} are said to be *dual* (or complementary) if for $0 \leq i, j \leq m-1$

$$\text{Tr}_{\mathcal{K}}^{\mathcal{F}}(\alpha_i \beta_j) = \delta_{ij} \tag{3.7}$$

where $\delta_{ij} = 0$ if $i \neq j$ and $\delta_{ij} = 1$ if $i = j$ (page 117 in [18]).

3.1.2 Finite field arithmetic

This section gives a brief summary of finite field arithmetic, starting with operations of interest and listing some of the most known algorithms for finite field arithmetic. In-depth descriptions will follow in the remaining chapters when needed. The section is concluded with a brief discussion how representation of the field elements leads to more effective algorithms.

An important notion for finite field arithmetic is the *characteristic* (Definition 1.43 in [11]). For a finite field $\mathcal{F} = (F, +, *)$, the smallest positive integer p such that, using notation from equation (3.1), $p \cdot \alpha = 0$ for every $\alpha \in \mathcal{F}$ is called the *characteristic* of \mathcal{F} , denoted $\text{char}(\mathcal{F}) = p$. If such an integer does not exist, the characteristic is 0. If $\text{char}(\mathcal{F}) > 0$, then it is a prime number (Corollary 1.45 in [11]). The characteristic of a finite field is the order of its prime subfield. It is also the additive order of the multiplicative identity 1. Furthermore, for arbitrary elements $\alpha, \beta \in \mathcal{F}$, where $\text{char}(\mathcal{F}) = p$, the following holds for any $k \geq 1$ (Theorem 1.46 in [11]):

$$(\alpha + \beta)^{p^k} = \alpha^{p^k} + \beta^{p^k}. \tag{3.8}$$

This work only considers finite fields of characteristic $p = 2$, i.e., the binary fields, because of the ease of their implementation with current technologies. Working with characteristic 2 also implies that for any $\alpha \in \mathcal{F}$, $\alpha + \alpha = (1 + 1)\alpha = 0\alpha = 0$, i.e., the addition is performed modulo 2, which corresponds to bit-wise XOR. In turn, this means that α is its own additive inverse, and hence there is no subtraction as such.

The operations of interest are addition, multiplication, and exponentiation. Two special cases of exponentiation, with a fixed exponent, are squaring and inversion. For binary fields, the exponentiations to (other) powers of two could also be singled out. A special case of multiplication is multiplication with a constant. On a high level, this work does not contain detailed discussions specific algorithms to perform these operations, but there is extensive literature on this topic, for example [19, 20]. Only a few algorithms will be explained throughout the text when needed.

Except for addition, all other operations highly depend on the representation of the field elements. Regardless which basis is used, the products of basis elements will always occur, and these products must again be represented w.r.t. the basis used. This step is known as *reduction*, and occurs in any operation (except addition). Algorithms for finite field arithmetic sometimes take advantage of specific representation of the field elements, and sometimes use rewriting tricks, to obtain more efficient algorithms.

3.1.3 Multivariate polynomials

Let \mathcal{F} be a finite field with q elements, where q is a prime or a prime power. A multivariate function in t variables x_0, x_1, \dots, x_{t-1} is defined as follows:

$$\begin{aligned} f &: \mathcal{F}^t \rightarrow \mathcal{F} \\ f(x_0, x_1, \dots, x_{t-1}) &= \sum_{\forall (i_0, i_1, \dots, i_{t-1}) \in \mathbb{Z}_q^t} c_{i_0, i_1, \dots, i_{t-1}} x_0^{i_0} x_1^{i_1} \dots x_{t-1}^{i_{t-1}} \end{aligned} \quad (3.9)$$

with coefficients $c_{i_0, i_1, \dots, i_{t-1}} \in \mathcal{F}$ and where $i_j \in \mathbb{Z}_q$ for $0 \leq j < t$. The sum in equation (3.9) runs over all possible monomials $x_0^{i_0} x_1^{i_1} \dots x_{t-1}^{i_{t-1}}$, where $\forall x \in \mathcal{F} : x^q = x$ (equation (3.2)). The expression on the r.h.s. of equation (3.9) describes an univariate polynomial when $t = 1$, and a multivariate polynomial when $t > 1$. In this work, terminology polynomial and function will be used interchangeably, with functions used mostly to describe feedback and filtering functions.

The degree of a monomial is defined as the sum of all its exponents (eq. (3.10)). The monomial with the highest degree is called leading monomial. The degree of the polynomial as the maximum of the monomial degrees (eq. (3.11)). Hence, the degree of the polynomial is also the degree of its leading monomial. For readability, the notation $m_{i_0, i_1, \dots, i_{t-1}}$ is introduced for monomials:

$$\begin{aligned} m_{i_0, i_1, \dots, i_{t-1}} = m(x_0, x_1, \dots, x_{t-1}) &= x_0^{i_0} x_1^{i_1} \dots x_{t-1}^{i_{t-1}} \\ \deg(m(x_0, x_1, \dots, x_{t-1})) &= \sum_{j=0}^{t-1} i_j \end{aligned} \quad (3.10)$$

$$\deg(f(x_0, x_1, \dots, x_{t-1})) = \max_{\forall (i_0, i_1, \dots, i_{t-1}) \in \mathbb{Z}_q^t} \{\deg(m_{i_0, i_1, \dots, i_{t-1}})\} \quad (3.11)$$

Based on the degree of the polynomial function, given by equation (3.11), a multivariate polynomial is classified as constant for $\deg(f) = 0$, linear for $\deg(f) = 1$, and nonlinear function for $\deg(f) > 1$.

3.1.4 Computer Algebra Systems

This brief introduction to Computer Algebra Systems (CAS) was summarized from [21, 22, 23]. Computer algebra is used to manipulate symbolic operations following specific rules of algebra, and thus providing exact solutions (where possible), again in symbolic form. The notions computer algebra, symbolic computation, algebraic algorithms and algebraic manipulations etc. are pretty much used as synonyms. Symbols are representations of mathematical objects, such as numbers, polynomials, finite field, field elements, etc. A very important part of computer algebra is the development of algorithms. Most of the time, several algorithms with the same functionality are implemented, because their efficiency depends on their inputs. Also notable is the application domain because it contributes a great deal to the development of the CAS itself.

Modern CAS consist of a (small) kernel in C and a collection of software programs, mostly written in the system's own programming language. In terms of functionality, CAS are classified as special purpose (for example GAP) and general purpose (for example Maple or Mathematica), covering many application areas. In general, CAS can simplify or expand expressions, compute limits, integrals, differential equations, factor polynomials, manipulate matrices, etc.

The main advantage of computer algebra is the ability to automate long sequences of algebraic computations. The latter can be very resource intensive in terms of memory and computation time. When used for scientific purposes, the symbolic computation is often followed by numerical, which is again exact, unless the user specifies otherwise. Most CAS are interactive and use some form of notebooks, where inputs, "pretty" formatted outputs, documentation, 2D and 3D plots and more can be kept in one document.

A not so brief history of CAS is presented in [22]. Below is a brief summary of some widely used general-purpose CAS. A systematic and more detailed overview can be found in [21]. Almost all of them are based on C, also used for the critical part of the systems and have their own language, used for most of the packages and user programs.

Maple: Originally developed at the University of Waterloo in the early 80's, Maple is a commercial general purpose CAS [24]. It consists of an algebraic engine, external libraries (to save memory) and shared libraries with contributions from users. It provides a user interface with worksheets that are similar to notebooks.

Mathematica: was the first system with a user friendly environment (in the form of notebooks) combining symbolics, numerics and graphics [25]. It has nearly 5,000 built-in functions covering all areas of technical computing and is built to provide industrial-strength capabilities. A powerful feature is the Wolfram SystemModeler using Modelica language. It has a wide range of application areas, from Automotive to Life Sciences. For example, the package Digital, a subset of Electrical, offers digital electrical components, with types and models based on the VHDL standard.

MuPAD: Multiprocessing Algebra Data Tool, is a programming language that allows users to write algorithms and to define new data types in an object-oriented way [26]. It is optimized for operating on symbolic math expressions and provides a parallel problem solving environment.

MATLAB: MATLAB is a library of programs and a programming language [27]. It is not considered as a CAS, however, it provides symbolic computation using MuPAD², integrated to the Symbolic Math Toolbox. The toolbox provides functions in common mathematical areas such as calculus, linear algebra, algebraic and ordinary differential equations, equation simplification, and equation manipulation. MATLAB is widely used in engineering and science and referred to as The Language of Technical Computing [27]. MathWorks provides, among many others, two interesting features: **Simulink**, for modelling various applications (wireless communications, internet of things, computer vision, etc), and **Stateflow** modelling and simulation based on state machines and flow charts. Both can be used standalone or integrated with MATLAB. MathWorks offers an extensive hardware support, from ARM microprocessors, Altera and Xilinx FPGAs to PARROT Minidrones. Tools like MATLAB Coder, Simulink Coder, Embedded Coder generate C and C++ code and HDL Coder can generate synthesizable Verilog and VHDL code.

SageMath: SageMath is a free open-source mathematics software system³ [28]. It includes special purpose open-source packages and special purpose CAS packages, such as NumPy and SciPy, Maxima, GAP, R, Singular, etc. It is Python-based, uses web-based notebooks and interactive shell, has good visualization capabilities and provides interfaces to Mathematica and Magma.

A related package of interest is a SageMath package *Cryptography* [29], implementing *LFSR-Cryptosystem* over finite field \mathbb{F}_2 . The system is simple: it encrypts the message directly with the sequence generated by the LFSR using modulo 2 sum of sequence bit and message bit. It does not support extension fields. Another simple Mathematica package, called Symbolic Linear Feedback Shift Registers [30] can generate bit sequences from LFSRs. It also implements the Berlekamp-Massey algorithm.

3.1.5 GAP

GAP (Groups, Algorithms, Programming) is a specialized CAS, originally intended for group theory, but evolved to include vector spaces, algebras, matrices, polynomials, etc. [3]. The core system of GAP consists of a kernel, written in C, a library of functions written in the GAP language, a library of group theoretical data and shared packages, i.e., GAP packages that are self-contained extensions of the core system [31]. Basic objects encountered are rational numbers, finite field elements⁴, words, lists, and so on. Internally, the objects are either positional or component objects. Each object in GAP has a type, which stores its category (i.e. mathematical identity), representation, and attributes and properties (i.e. what is known about it) [32]. The type is stored as a long bitlist (a flag list), where each bit corresponds to a filter. The type of an object can change throughout its lifetime, as more becomes known about the object. Filters are used by method selection to allow different conditions on the arguments and thus allow overloading. In addition, to distinguish between good and optimal methods, filters are ranked, and the highest ranking method is chosen. Besides methods, GAP is also using functions. For example, writing to a file is always

²replaced Maple

³licensed under the GPL

⁴will be abbreviated ffe or FFE like in GAP documentation

implemented as a function. Attributes and properties store some values associated with the object once they have been computed. Properties are attributes whose value can be only true or false. For example, *Size* is an attribute and *IsFinite* is a (true) property of a finite field.

The GAP packages developed for this work will be described using objects, filters, attributes, properties, methods and functions. The functionality of packages is captured with a pair of declaration and installation files, and both together will be referred to as a *source file*. The GAP examples in this work follow the format used by GAPDoc package [33]. Some important GAP methods used will be referenced from the GAP Reference Manual [34]. Another advantage of GAP is its inclusion in SageMath, which allows all packages developed for this research to be loaded in SageMath as well.

3.2 Selected symmetric-key cryptographic primitives

This section covers selected symmetric-key cryptographic primitives. It begins with the feedback shift registers in Subsection 3.2.1: they are not a primitive, but are commonly used as their building blocks. The next three subsections are related to stream ciphers: introduction and general structure of stream ciphers in Subsection 3.2.2, followed by a short description of the eSTREAM project in Subsection 3.2.3, and introduction to WG stream cipher family in Subsection 3.2.4. The last four subsections are related to authenticated encryption. Introduction to authenticated encryption is provided in Subsection 3.2.5, followed by short descriptions of the CAESAR competition in Subsection 3.2.6 and of the ongoing NIST Lightweight Cryptography (LWC) project in Subsection 3.2.7. Subsection 3.2.8 is dedicated to the authenticated cipher WAGE, a round 2 candidate for the NIST LWC project. Block ciphers, although a big and important part of symmetric-key cryptography, are omitted from this discussion.

3.2.1 Feedback shift registers

This section begins with a brief overview of applications of feedback shift registers (FSRs), and then introduces some definitions and basic terminologies related to linear and nonlinear feedback shift registers (LFSR and NLFSR), and filtering generators [14, 36]. Further details can be found in numerous sources such as [11, 37]. Additional notions will be introduced when needed in Chapter 6, which describes the GAP package FSR.

Applications of feedback shift registers

Feedback shift registers play an important role in cipher design⁵. A well known early example of an LFSRs based stream cipher is A5/1, intended for securing GSM voice and data. A5/1 is

⁵stream ciphers will be introduced in section 3.2.2

built from three LFSRs with different periods and a stop-and-go majority function controlling their clocks. The output is computed as an XOR of outputs from all three LFSRs. A milestone in stream cipher design is the eSTREAM project [4], launched in 2004. All 3 hardware portfolio ciphers, Grain, MICKEY⁶ and Trivium (Subsection 4.1.2), as well as the software portfolio cipher Sosemanuk. The cipher ACORN [38], included in the CAESAR portfolio [39], is based on 6 LFSRs (Subsection 3.2.6). Last but not least, both of the stream ciphers used for encryption and integrity of communications in mobile networks, Snow3G and ZUC, use LFSRs over an extension field. Most of the remaining round-2 candidates in the NIST LWC project [5, 10] (Subsection 3.2.7) contain LFSRs for generation of round constants, e.g., ACE, ESTATE, KNOT, Saturnin, etc., or use a primitive, which uses LFSRs for generation of the round constants, e.g., (tweakable) block cipher GIFT (candidates GIFT-COFB, HYENA, etc.). Some candidates use FSRs for their internal state, for example Grain-128AEAD [40] or WAGE [6]. A detailed description of WAGE is presented in Section 3.2.8.

LFSRs are widely used in coding theory, for example in BCH and Reed-Muller codes [18]. Further examples are cyclic redundancy codes used in many communication and data storage devices for error-detection, and for pattern generation in built-in self testing for electronic circuits. The LFSRs have been used as counters in applications where the order of the sequence does not matter, for example Xilinx proposed⁷ the use of LFSR counters to address the memory [41]. Less noticeable is the use of LFSRs in algorithms for finite field arithmetic. For example, a serial circuit that requires multiplication by x , followed by reduction modulo the field defining polynomial, can be implemented as a LFSR with the field polynomial as feedback [42, 18].

What is a feedback shift register ?

An n -stage shift register over a finite field \mathcal{F} is an array of n registers (denoted $S_i, i = n-1, \dots, 0$), and each stage holds a value from the underlying finite field \mathcal{F} . The positive integer n is also referred to as the *length* of the FSR. This memory array is shifted with each step $S_i \rightarrow S_{i-1}$ for $i = n-1, \dots, 1$, and the vacant register S_{n-1} is updated with a new value obtained from the feedback function, hence the name *feedback shift register* (FSR). One of the stages is used to generate the output and each time the FSR is clocked, that stages produces a new element $s_i \in \mathcal{F}$. In this way, the FSR produces a sequence of elements:

$$\underline{s} = \{s_k\} = s_0, s_1, s_2, \dots \quad (3.12)$$

where $s_{k+n} = f(s_k, s_{k+1}, \dots, s_{k+n-1})$ for $k = 0, 1, \dots$, and f is a multivariate function in $t = n$ variables x_0, x_1, \dots, x_{n-1} as defined in equation (3.9) in Section 3.1.3. The variable x_i corresponds to the stage $S_i, i \in \{0, 1, \dots, n-1\}$. Based on the degree, given by equation (3.11), the distinction is made between *linear* (LFSR) and *nonlinear* (NLFSR) feedback shift registers. For the linear

⁶Galois-style feedback

⁷declared obsolete

case, the degree $\deg(f) = 1$, i.e., f is a linear function. However, this is not to be mistaken with the degree of the feedback polynomial of an LFSR, which is a function in one variable: this is an equivalent notation, defined in equation (3.13). Furthermore, when $q = 2$, $f(x_0, x_1, \dots, x_{n-1})$ is called a Boolean function in n variables.

In case of an LFSR, the feedback function (3.9) is given by $f(x_0, \dots, x_{n-1}) = \sum_{i=0}^{n-1} c_i x_i$ which can be represented with an univariate polynomial

$$h(y) = y^n + \sum_{j=0}^{n-1} c_j y^j \quad (3.13)$$

where y^j corresponds to the stage S_j for $j \in \{0, 1, \dots, n-1\}$, and y^n to the new value computed by the feedback. Coefficients c_j , $j \in \{0, 1, \dots, n-1\}$, of the polynomial in (3.13) belong to the underlying field \mathcal{F} .

A simple schematic of an n -stage FSR is shown in Figure 3.2, with the output sequence produced by stage S_0 .

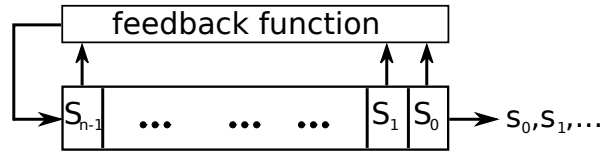


Figure 3.1: Top level schematic of a n -stage FSR (Figure from [36])

At any given moment, the contents of the FSR hold n values from the underlying finite field, and can be written as a vector of length n : $(s_0, s_1, \dots, s_{n-1}) \in \mathcal{F}^n$. This vector is called the *state* of the FSR and the state right after loading the *initial state*. The output sequence \underline{s} is completely determined by the feedback polynomial and the initial state. In case of $q = 2$, $\mathcal{F} = \mathbb{F}_2$, function f is a *Boolean function* and the FSR produces a binary sequence. In all other cases, the sequence is referred to as an q -ary sequence.

Filtering generators

A typical structure of a filtering generator is shown in Figure 3.2(b): it consists of a filter, i.e. a non-linear multivariate polynomial function, applied to an LFSR with n stages. Let $(s_k, s_{k+1}, \dots, s_{k+n-1}) \in \mathcal{F}^n$ be the k th state of the LFSR, $g(x_0, \dots, x_{t-1})$, a multivariate polynomial in t variables, where $t \leq n$, and (d_0, \dots, d_{t-1}) , a selection of t tap positions in the state, i.e., $0 \leq d_0 < d_1 < \dots < d_{t-1} < n$. The output sequence $\underline{a} = \{a_k\}$ is given by

$$a_k = g(s_{k+d_0}, \dots, s_{k+d_{t-1}}), \quad k = 0, 1, \dots$$

This is referred to as a filtering generator where g is called a filtering function, or simply filter, and $\underline{a} = \{a_k\}$, a filtering sequence.

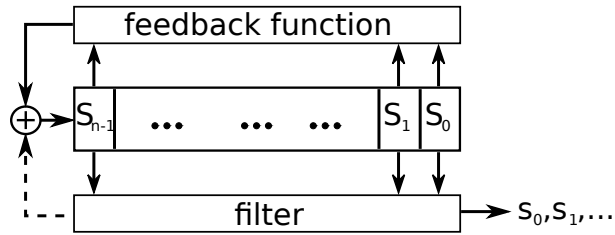


Figure 3.2: Top level schematic of a n -stage FSR with a filter (Figure from [36])

3.2.2 Stream ciphers

Stream ciphers are symmetric-key primitives, used to provide confidentiality. The story of stream ciphers began with Vernam’s shield, i.e. the one-time pad, in the early 20th century [43]. 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. However, 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. To address these problems, stream ciphers use a short pre-shared key, exchanged using a public-key cryptosystem, 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 3.3 shows the general behavioural model⁸ 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 (a pre-shared secret key and an initialization vector⁹ (IV)), to obtain the same keystream. The cipher operates in two phases: a key initialization phase (KI in Figure 3.3) and the running phase, when the PRSG algorithm outputs the keystream (PRSG in Figure 3.3), refer to [37] for details. The task of KI¹⁰ is to scramble the key and the 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 [37]. The KI itself is usually the PRSG algorithm running for a certain number of steps with output discarded and usually some value is added to the feedback of PRSG. The first keystream character is produced when cipher enters its running phase.

The word character is used 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

⁸ based on [37]

⁹ IV can be public

¹⁰ sometimes also called KIA for key initialization algorithm

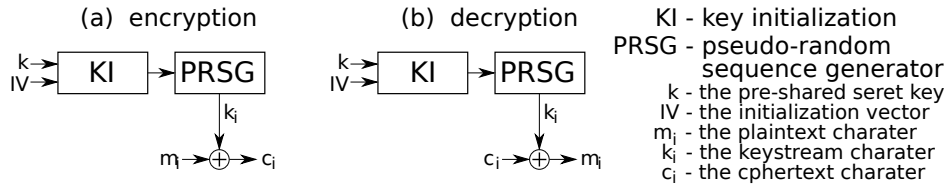


Figure 3.3: Behavioral model of a stream cipher: (a) encryption and (b) decryption

cycle, e.g. 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.

3.2.3 The eSTREAM project: Grain and Trivium

The eSTREAM project started in 2004 with the objective to promote research in stream cipher design [4]. 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 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. 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 the most relevant metric for comparison; some discussion on this topic can be found in [44]. In the Phase 3, the primary criteria besides security was the area complexity [4], another commonly used metric was the aforementioned throughput-to-area ratio. Grain and Trivium are presented in this section, MICKEY is omitted due to its larger area complexity.

Grain

There are two versions of Grain: the original 80-bit version (using an 80-bit secret key and 64-bit IV, called Grain, modified to Grain v1 and included in eSTREAM Portfolio) and Grain-128 (using a 128-bit key and IV of same length). The structure of Grain can be seen in Figure 3.4: it 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 FSRs and used as an input to a Boolean function. 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 [45, 46].

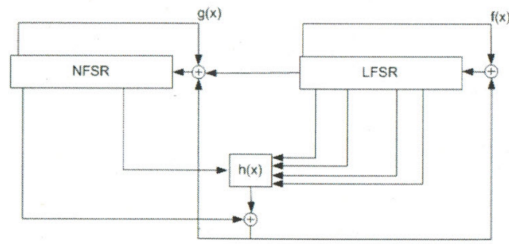


Figure 3.4: The structure of Grain: run mode (Figure from [46])

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 a total internal state of 288 bits. These three FSR's can be arranged into a circular shape, as can be seen in Figure 3.5. From the 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: (i.) to update the FSR's (using 4 bits from the previous FSR and one bit from the FSR being updated) and (ii.) to compute the keystream bit (by XORing 6 state bits, two from each FSR). For more details on Trivium, refer to [47].

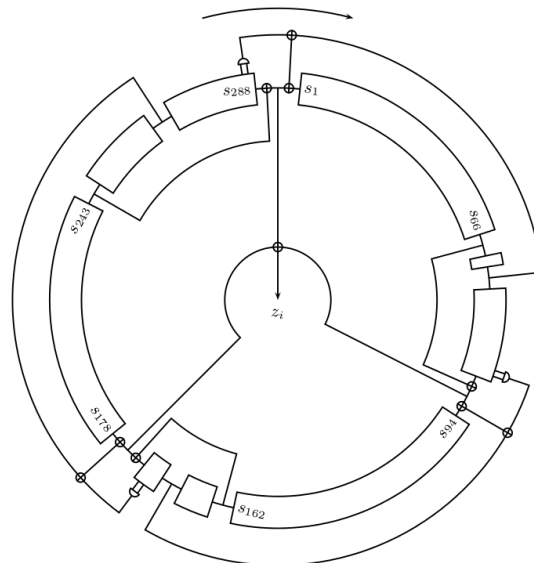


Figure 3.5: The structure of Trivium (Figure from [47])

3.2.4 The WG stream cipher

The WG stream cipher is a bit-oriented stream cipher, which generates a keystream with proven randomness and cryptographic properties. The WG stream cipher is a synchronous stream cipher based on the Welch-Gong (WG) transformations on an m -sequence. 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, [7]. For more information refer to [48, 49, 50]

LFSR over an extension field

FSRs were introduced in Subsection 3.2.1; this paragraph is focusing on an LFSR over an extension field \mathbb{F}_{2^m} . The shift register is an array of n registers, each of them holding an m -bit value (an element from \mathbb{F}_{2^m}); the registers are referred to as stages S_i , $i = n-1, \dots, 0$. The feedback function is a simple expression that involves only multiplications of field elements by constants and addition in \mathbb{F}_{2^m} :

$$\ell(x) = x^n + \sum_{i=1}^{n-1} c_i x^i + \gamma \quad (3.14)$$

over \mathbb{F}_{2^m} . Expression (3.14) is obtained from expression (3.13) by moving the constant term out of the sum, i.e., $\gamma = c_0 \in \mathbb{F}_{2^m}$. It is common practice to choose all other coefficients from the prime field \mathbb{F}_2 , in order to minimize the hardware. The polynomial $\ell(x)$ is a primitive polynomial, which ensures that the LFSR generates a maximal length sequence (m -sequence) with period $(2^m)^n - 1$. The polynomial function associated with the LFSR polynomial $\ell(x)$ from equation (3.14) is a function from $\mathbb{F}_{2^m}^n \rightarrow \mathbb{F}_{2^m}$ given by

$$f(x_0, x_1, \dots, x_{n-1}) = \gamma x_0 + \sum_{i=1}^{n-1} c_i x_i \quad (3.15)$$

where x_i corresponds to the stage S_i . With each step, the LFSR is updated as follows:

$$(S_0, S_1, \dots, S_{n-1}) \rightarrow (S_1, S_2, \dots, S_{n-1}, S_n),$$

where $S_n = f(S_0, S_1, \dots, S_{n-1})$ is computed as defined by equation (3.15).

Some terminologies used in this thesis:

- a step of the FSR or clocking of the FSR: the contents of the registers are shifted to the right and the register S_{n-1} is updated with the feedback value S_n ,
- taps or tap positions: the FSR stages entering the feedback. More precisely, the stages S_i corresponding to the indices i of $\ell(x)$ from equation (3.14) for which $c_i \neq 0$ for the LFSRs, and to the indices i of $f(x_0, \dots, x_{n-1})$ from equation (3.9) for which $c_{i_0, i_1, \dots, i_{n-1}} \neq 0$ for the NLFSRs,
- output tap: the stage used to output the sequence (usually S_{n-1} , to shorten the key initialization phase). Multiple output taps are possible (a vector output in each step),
- initial state: the contents of the FSR after loading.

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

$$\text{WGT-}m(X^d) = \text{Tr}(q(X^d + 1) + 1), \quad \text{for } X \in \mathbb{F}_{2^m} \quad \text{and} \quad \text{gcd}(d, 2^m - 1) = 1, \quad (3.16)$$

where $q(x) = x + x^{r_1} + x^{r_2} + x^{r_3} + x^{r_4}$ 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:

$$\begin{aligned} r_1 &= 2^k + 1 \\ r_2 &= 2^{2k} + 2^k + 1 \\ r_3 &= 2^{2k} - 2^k + 1 \\ r_4 &= 2^{2k} + 2^k - 1. \end{aligned} \quad (3.17)$$

The decimation exponent d is chosen to improve the cryptographic properties of the keystream and must have an efficient hardware implementation.

Equation (3.16) can be split into two parts. The first part is the WG permutation $\text{WGP-}m(X^d)$, shown in equation (3.18). Second is the WG transformation $\text{WGT-}m(X^d)$, shown in equation (3.19); it is the trace function applied to the result of the WG permutation.

$$\text{WGP-}m(X^d) = q(X^d + 1) + 1 \quad (3.18)$$

$$\text{WGT-}m(X^d) = \text{Tr}(\text{WGP}(X^d)). \quad (3.19)$$

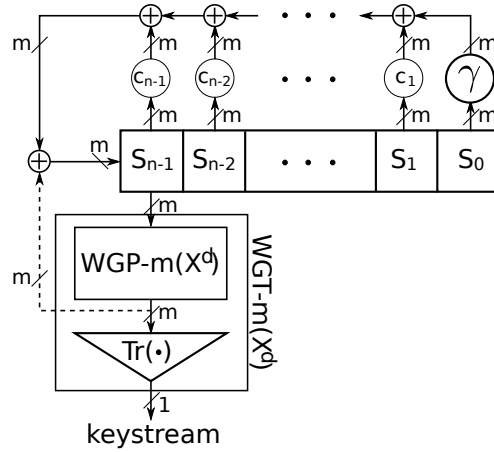


Figure 3.6: The structure of the WG keystream generator (Figure adapted from [50])

The WG keystream generator

The general structure of the WG keystream generator is shown in Figure 3.6: the LFSR of degree n and the WGT $-m(X^d)$, both defined over \mathbb{F}_{2^m} . The LFSR is loaded with the key and the initialization vector IV, then key initialization phase (KI) is run for $2n$ steps. During KI, the WG permutation value $WGP - m(X^d)$ is added to the LFSR feedback, as shown with the dashed arrow in Figure 3.6. During the running phase, the LFSR is update only by the feedback, and the keystream generator produces 1 bit of keystream per step of the LFSR. For encryption and decryption another XOR, not shown in Figure 3.6, is needed.

3.2.5 Authenticated encryption with associated data

Stream ciphers, introduced in Subsection 3.2.2, are used to provide confidentiality. According to Chen and Gong [37], confidentiality is a security feature to assure that information can only be received by eligible communication parties. Confidentiality is achieved through encryption mechanisms, for example using a stream cipher. Two other, important and inseparable security objectives are integrity and authenticity. Integrity is to prevent from altering the content of the message, and authenticity is to prevent from altering the perceived origin of the message. With symmetric-key based cryptography, integrity and authenticity can be achieved by generating a message authentication code, a data tag calculated on the message using a shared secret key.

There are three combinations of encryption and authentication, also called modes, distinguished by the order of the two: encrypt and authenticate, authenticate-then-encrypt, and encrypt-then-authenticate. In all three cases, both the ciphertext and the tag are transmitted. In the first case, encryption and authentication are independent and can be performed in parallel. In case of authenticate-then-encrypt, the tag is computed on the plaintext, and encryption applied to a concatenation of plaintext and the tag. In case of encrypt-then-authenticate, the tag is computed on the ciphertext.

These three modes are also referred to as generic composition approach, and can be extended to include associated data [51]. For example, using encrypt-then-authenticate approach, the tag is computed on a concatenation of the associated data and the ciphertext. Rogaway listed some considerations to this approach in [51], namely: the associated data must stay in plaintext, because it contains routing information, associated data must be authenticated, there must be a way to bind the associated data to the message, and the packet length shall not increase. Including associated data in authenticated encryption (AE), is called authenticated encryption with associated data (AEAD). Literature often refers to AEAD schemes simply as AE. The term authenticated encryption refers to algorithms that integrate authentication and encryption, as opposed to the generic composition approach. In [52], Wu and Preneel listed three approaches to design an integrated authenticated encryption algorithm: (i.) using a block cipher in a special mode, e.g., AES-GCM [53], and OCB [54], (ii.) using a stream cipher, then dividing the keystream into one part used for encryption and one part used for authentication, e.g., Grain-128a [55] and (iii.) designing dedicated AE algo-

rithms, which use the input data to update the state of the cipher, and extract the tag from the state after completed encryption, e.g., AEGIS [52], and WAGE [6].

3.2.6 The CAESAR competition: Ascon and ACORN

The CAESAR competition was launched in 2014 [39]. The acronym stands for “Competition for Authenticated Encryption: Security, Applicability, and Robustness”. The call for submissions asked for a family of authenticated ciphers that provides integrity and confidentiality for the plaintext, and integrity for associated data. (1) offer advantages over AES-GCM [53] and (2) are suitable for widespread adoption [56]. The final CAESAR portfolio is organized into three use cases, with the following finalists:

1. Lightweight applications (resource constrained environments): Ascon (first choice) [57], ACORN (second choice) [38]
2. High-performance applications: AEGIS-128 [52], OCB [54] (in alphabetical order, no preference)
3. Defense in depth: Deoxys-II (first choice) [58], COLM [59] (second choice)

Below are short descriptions of the two case 1 ciphers, namely for the lightweight applications.

Ascon

Ascon [57] is a sponge based cipher with internal state size of 320 bits. It uses a 128 bit key and nonce. The data block size is 64 bits and the tag length is 182 bits. The core permutation p begins with constant addition (using all together 12 different constants), followed by a substitution layer using 64 parallel 5-bit Sboxes, and a linear diffusion layer. During initialization and finalization, the number of rounds $a = 12$ is used, and during processing of associated data and of the plaintext/ciphertext, the number of rounds is $b = 6$. Figure 3.7 shows Ascon’s mode of operation, which is based on a duplex sponge mode.

ACORN

ACORN [38] is based on a stream cipher and has a 293 bit internal state, composed of six LFSRs of different lengths, all defined by trinomials, shown in Figure 3.8. It uses a 128 key and initialization vector (IV), and generates a 128bit tag. It uses two Boolean functions, both containing XOR and AND gates, used in the nonlinear feedback function f_i and in keystream bit function ks_i . The processing consists of four phases, the initialization, processing of associated data, encryption/decryption, and finalization.

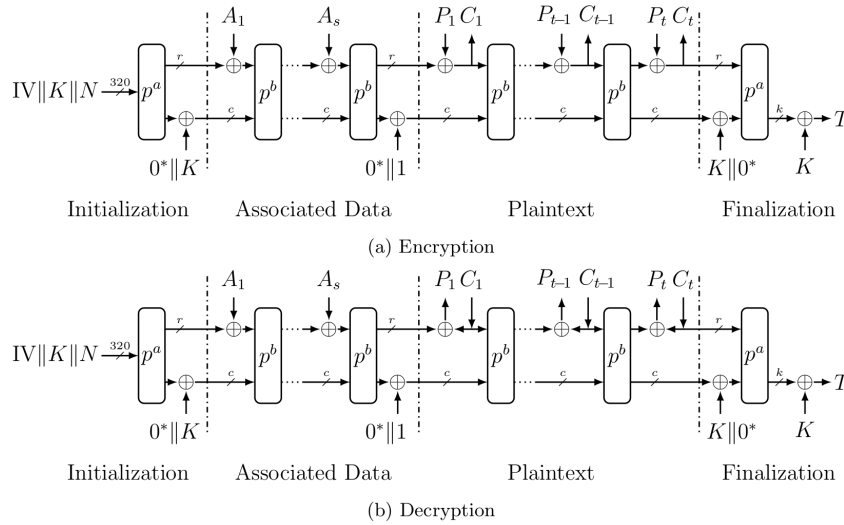


Figure 3.7: Schematic diagram of Ascon: mode of operation (Figure from [57])

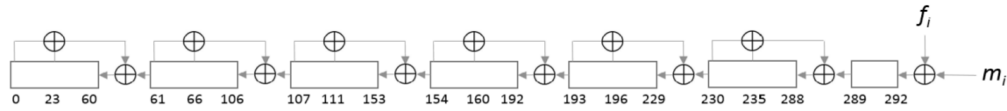


Figure 3.8: The structure of ACORN (Figure from [38])

3.2.7 The NIST Lightweight Cryptography project

In 2013, NIST (National Institute of Standards and Technology) initiated a lightweight cryptography project, and in 2018 a request for nominations of cryptographic algorithms for the Lightweight Cryptography Standardization Process [5, 60]. The algorithms shall implement AEAD functionality, and optional hash functionality. Round 1 of the LWC standardization project received 56 candidates, out of which, 32 candidates are remaining in the ongoing round 2 [10]. The authenticated encryption cipher WAGE is advanced to round 2 (subsection 3.2.8). Some round 2 candidates were mentioned in Subsection 3.2.1, demonstrating applications of LFSRs.

3.2.8 The WAGE authenticated encryption scheme

Disclaimer 3.1: The WAGE authenticated encryption scheme

The WAGE authenticated encryption scheme is a joint work of the members of the ComSec Lab, listed in alphabetical order: Mark Aagaard, Riham AlTawy, Guang Gong, Kalikinkar Mandal, Raghvendra Rohit, and Nusa Zidaric (author of this thesis). Section 3.2.8 is a background section, summarizing the specifications of the LWC candidate WAGE [6] (joint work), as submitted to the the NIST LWC project [5].

WAGE- \mathcal{AE} -128 is a hardware oriented authenticated encryption scheme, built on top of the initialization phase of the LFSR based, Welch-Gong WG stream cipher (see Section 3.2.4). WAGE is derived from WG and \mathcal{AE} . WAGE- \mathcal{AE} -128 is composed of the WAGE permutation operating in a unified duplex sponge mode [61]. The 259 bit WAGE permutation is iterative and has a round function built from an LFSR, a decimated Welch-Gong permutation WGP and small Sboxes SB. Details on cryptographic properties, such as differential uniformity and nonlinearity of the WGP and SB and final selection of the LFSR polynomial can be found in [6]. The parameter selection for WAGE was aimed at balancing the security and hardware implementation area, using hardware implementation results for many design decisions, e.g., field size, representation of field elements, LFSR polynomial, etc., as will be explained in depth in Chapter 22 about the impact of hardware implementations on the WAGE algorithm design. This section explains the WAGE permutation and its building blocks, followed by the short description of WAGE- \mathcal{AE} -128.

The block diagram of WAGE is shown in Figure 3.9. It depicts the LFSR and the nonlinear components of WAGE. Both the LFSR and the WGP are defined over \mathbb{F}_{27} , and the Sbox is a 7 bit permutation. The finite field \mathbb{F}_{27} is defined with the primitive polynomial $f(x) = x^7 + x^3 + x^2 + x + 1$, and the field elements are represented using the polynomial basis $\text{PB} = \{1, \omega, \dots, \omega^6\}$, where ω is the root of $f(x)$ (Table 3.1). The LFSR is defined by the feedback polynomial $\ell(x)$ (Table 3.1), which is primitive over \mathbb{F}_{27} . The 37 stages of the LFSR also constitute the internal state of WAGE, denoted $S^i = (S_{36}^i, S_{35}^i, \dots, S_1^i, S_0^i)$; the superscript i is used to mark the i -th iteration of the permutation.

For an element $x \in \mathbb{F}_{27}$, the decimated WG permutation with decimation $d = 13$ is defined in Table 3.1, i.e., WGP-16(x^{13}) from equation (3.18). The SB is defined bit-wise, with the input $x = (x_0, x_1, x_2, x_3, x_4, x_5, x_6)$, but the interpretation of the 7 bits is identical to the interpretation of the coefficients of the finite field element represented in its polynomial basis. The 7-bit SB uses a nonlinear transformation Q and a permutation P, which together yield one-round of the SB, namely $R = P \circ Q$. Transformations Q, P and R are given in table 3.1. The SB itself iterates the function R 5 times, followed by applying Q once, and then the 0th and 2nd components are complemented, as listed at the end of Table 3.1. Appendix B.2 shows the Sbox representation of WGP in Table B.3 and hexadecimal representation of SB in Table B.4.

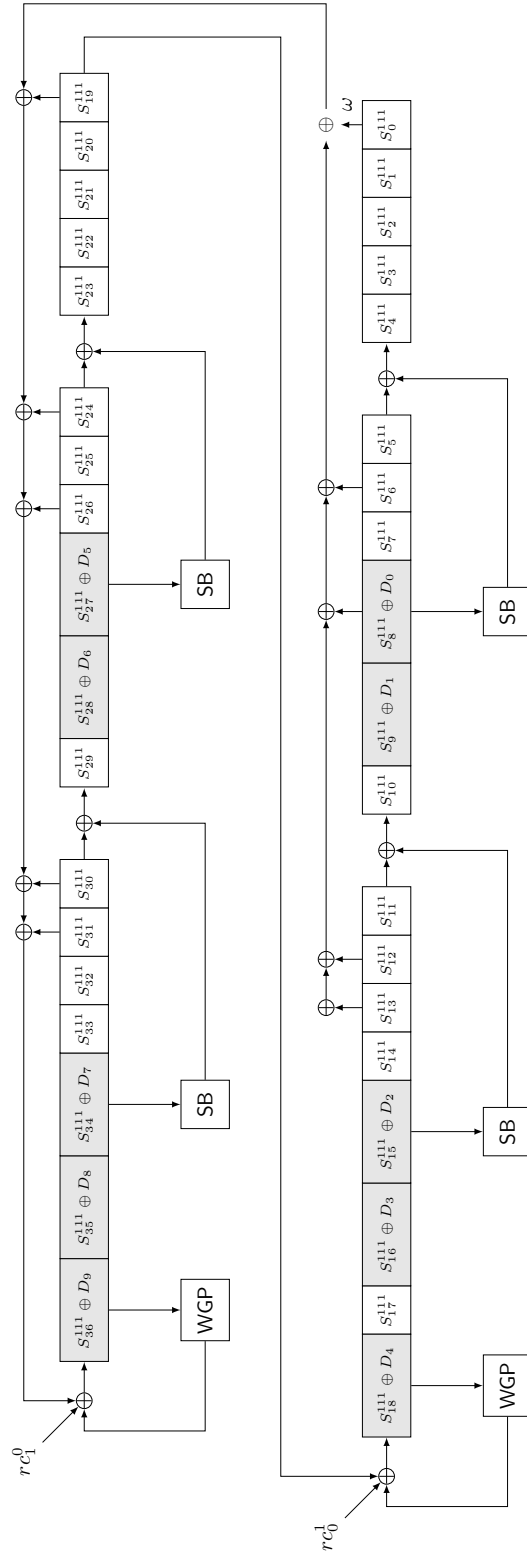


Figure 3.9: Rate (shaded) and capacity part of WAGE- \mathcal{AE} -128 (Figure from [6])

\mathbb{F}_{2^7}	$f(x) = x^7 + x^3 + x^2 + x + 1, f(\omega) = 0$	polynomial basis:	PB = $\{1, \omega, \dots, \omega^6\}$
$a \in \mathcal{F}$	$a = \sum_{i=0}^6 a_i \omega^i, a_i \in \mathbb{F}_2$	vector representation:	$[a]_{\text{PB}} = (a_0, a_1, a_2, a_3, a_4, a_5, a_6)$
LFSR	$\ell(y) = y^{37} + y^{31} + y^{30} + y^{26} + y^{24} + y^{19} + y^{13} + y^{12} + y^8 + y^6 + \omega, f(\omega) = 0$		
WGP7	$\text{WGP7}(x^d) = x^d + (x^d + 1)^{33} + (x^d + 1)^{39} + (x^d + 1)^{41} + (x^d + 1)^{104}, d = 13$		
SB Q	$Q(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \rightarrow (x_0 \oplus (x_2 \wedge x_3), x_1, x_2, \bar{x}_3 \oplus (x_5 \wedge x_6), x_4, \bar{x}_5 \oplus (x_2 \wedge x_4), x_6)$		
SB P	$P(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \rightarrow (x_6, x_3, x_0, x_4, x_2, x_5, x_1)$		
SB R	$R(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \rightarrow (x_6, \bar{x}_3 \oplus (x_5 \wedge x_6), x_0 \oplus (x_2 \wedge x_3), x_4, x_2, \bar{x}_5 \oplus (x_2 \wedge x_4), x_1)$		
SB	$(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \leftarrow R^5(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$		
	$(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \leftarrow Q(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$		
	$(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \leftarrow (\bar{x}_0, x_1, \bar{x}_2, x_3, x_4, x_5, x_6)$		

Table 3.1: Specification parameters of WAGE

The WAGE permutation is iterative: it repeats the round function $\text{WAGE-StateUpdate}(S^i)$ 111 times (see Algorithm 1). In each round, 6 stages of the LFSR are updated nonlinearly, while all the remaining stages are just shifted. A pair of 7-bit round constants (rc_0^i, rc_1^i) is XORed with the pair of stages (18, 36) to destroy similarity among state updates. Round constants are produced by an LFSR of length 7 with feedback polynomial $x^7 + x + 1$, implemented in a 2-way parallel configuration, see [6] for details.

Algorithm 1 WAGE permutation

```

1: Input :  $S^0 = (S_{36}^0, S_{35}^0, \dots, S_1^0, S_0^0)$ 
2: Output :  $S^{111} = (S_{36}^{111}, S_{35}^{111}, \dots, S_1^{111}, S_0^{111})$ 

3: for  $i = 0$  to 110 do:
4:    $S^{i+1} \leftarrow \text{WAGE-StateUpdate}(S^i, rc_0^i, rc_1^i)$ 
5: return  $S^{111}$ 

6: Function  $\text{WAGE-StateUpdate}(S^i)$ :
7:    $fb = S_{31}^i \oplus S_{30}^i \oplus S_{26}^i \oplus S_{24}^i \oplus S_{19}^i \oplus$ 
       $S_{13}^i \oplus S_{12}^i \oplus S_8^i \oplus S_6^i \oplus (\omega \otimes S_0^i)$ 
8:    $S_4^{i+1} \leftarrow S_5^i \oplus \text{SB}(S_8^i)$ 
9:    $S_{10}^{i+1} \leftarrow S_{11}^i \oplus \text{SB}(S_{15}^i)$ 
10:   $S_{18}^{i+1} \leftarrow S_{19}^i \oplus \text{WGP}(S_{18}^i) \oplus rc_0^i$ 
11:   $S_{23}^{i+1} \leftarrow S_{24}^i \oplus \text{SB}(S_{27}^i)$ 
12:   $S_{29}^{i+1} \leftarrow S_{30}^i \oplus \text{SB}(S_{34}^i)$ 
13:   $S_{36}^{i+1} \leftarrow fb \oplus \text{WGP}(S_{36}^i) \oplus rc_1^i$ 
14:   $S_j^{i+1} \leftarrow S_{j+1}^i$  where
       $j \in \{0, \dots, 36\} \setminus \{4, 10, 18, 23, 29, 36\}$ 
15: return  $S^{i+1}$ 

```

Algorithm 2 presents a high-level overview of WAGE- \mathcal{AE} -128. WAGE uses the unified sponge duplex mode for sLiSCP [61] to provide the AEAD functionality, as shown in Figure 3.10. The internal state S of WAGE is divided into two parts: the 64-bit rate part S_r and the capacity part S_c . The rate part corresponds to the shaded LFSR stages in Figure 3.9. The 0-th bit of stage S_{36} , i.e., $S_{36,0}$, and all bits of stages $S_{35}, S_{34}, S_{28}, S_{27}, S_{18}, S_{16}, S_{15}, S_9$ and S_8 constitute S_r , while all remaining bits in the state constitute S_c . Two domain separator bits ds_1 and ds_0 are XORed to the first two bits of S_c , namely $S_{0,1}^{111}$ and $S_{0,0}^{111}$ respectively. The domain separators are used to distinguish between four phases required for each encryption WAGE- \mathcal{E} and decryption WAGE- \mathcal{D} : the initialization, processing of associated data AD, encryption of a message M or decryption of the ciphertext C, and finalization phase.

Algorithm 2 WAGE- \mathcal{AE} -128 algorithm

<pre> 1: Authenticated encryption WAGE-$\mathcal{E}(K, N, AD, M)$: 2: $S \leftarrow \text{Initialization}(N, K)$ 3: if $AD \neq 0$ then: 4: $S \leftarrow \text{Processing-Associated-Data}(S, AD)$ 5: $(S, C) \leftarrow \text{Encryption}(S, M)$ 6: $T \leftarrow \text{Finalization}(S, K)$ 7: return (C, T) 8: Initialization(N, K): 9: $S \leftarrow \text{load-}\mathcal{AE}(N, K)$ 10: $S \leftarrow \text{WAGE}(S)$ 11: for $i = 0$ to 1 do: 12: $S \leftarrow (S_r \oplus K_i, S_c)$ 13: $S \leftarrow \text{WAGE}(S)$ 14: return S 15: Processing-Associated-Data(S, AD): 16: $(AD_0 \parallel \dots \parallel AD_{\ell_{AD}-1}) \leftarrow \text{pad}_r(AD)$ 17: for $i = 0$ to $\ell_{AD} - 1$ do: 18: $S \leftarrow (S_r \oplus AD_i, S_c \oplus 0^{c-7} \parallel 1 \parallel 0^6)$ 19: $S \leftarrow \text{WAGE}(S)$ 20: return S 21: Encryption(S, M): 22: $(M_0 \parallel \dots \parallel M_{\ell_M-1}) \leftarrow \text{pad}_r(M)$ 23: for $i = 0$ to $\ell_M - 1$ do: 24: $C_i \leftarrow M_i \oplus S_r$ 25: $S \leftarrow (C_i, S_c \oplus 0^{c-7} \parallel 0 \parallel 1 \parallel 0^5)$ 26: $S \leftarrow \text{WAGE}(S)$ 27: $C_{\ell_M-1} \leftarrow \text{trunc-msb}(C_{\ell_M-1}, M \bmod r)$ 28: $C \leftarrow (C_0, C_1, \dots, C_{\ell_M-1})$ 29: return (S, C) 30: $\text{pad}_r(X)$: 31: $X \leftarrow X \parallel 10^{r-1-(X \bmod r)}$ 32: return X 33: $\text{trunc-lsb}(X, n)$: 34: return $(x_{r-n}, x_{r-n+1}, \dots, x_{r-1})$ </pre>	<pre> 1: Verified decryption WAGE-$\mathcal{D}(K, N, AD, C, T)$: 2: $S \leftarrow \text{Initialization}(N, K)$ 3: if $AD \neq 0$ then: 4: $S \leftarrow \text{Processing-Associated-Data}(S, AD)$ 5: $(S, M) \leftarrow \text{Decryption}(S, C)$ 6: $T' \leftarrow \text{Finalization}(S, K)$ 7: if $T' \neq T$ then: 8: return \perp 9: else: 10: return M 11: Decryption(S, C): 12: $(C_0 \parallel \dots \parallel C_{\ell_C-1}) \leftarrow \text{pad}_r(C)$ 13: for $i = 0$ to $\ell_C - 2$ do: 14: $M_i \leftarrow C_i \oplus S_r$ 15: $S \leftarrow (C_i, S_c \oplus 0^{c-7} \parallel 0 \parallel 1 \parallel 0^5)$ 16: $S \leftarrow \text{WAGE}(S)$ 17: $M_{\ell_C-1} \leftarrow S_r \oplus C_{\ell_C-1}$ 18: $C_{\ell_C-1} \leftarrow \text{trunc-msb}(C_{\ell_C-1}, C \bmod r) \parallel$ $\text{trunc-lsb}(M_{\ell_C-1}, r - C \bmod r)$ 19: $M_{\ell_C-1} \leftarrow \text{trunc-msb}(M_{\ell_C-1}, C \bmod r)$ 20: $M \leftarrow (M_0, M_1, \dots, M_{\ell_C-1})$ 21: $S \leftarrow \text{WAGE}(C_{\ell_C-1}, S_c \oplus 0^{c-7} \parallel 0 \parallel 1 \parallel 0^5)$ 22: return (S, M) 23: Finalization(S, K): 24: for $i = 0$ to 1 do: 25: $S \leftarrow \text{WAGE}(S_r \oplus K_i, S_c)$ 26: $T \leftarrow \text{tagextract}(S)$ 27: return T 28: $\text{trunc-msb}(X, n)$: 29: if $n = 0$ then: 30: return ϕ 31: else: 32: return $(x_0, x_1, \dots, x_{n-1})$ </pre>
--	---

The rate part S_r of the state is used for both absorbing and squeezing. A 64-bit input block is padded with zeros to 70 bits, then absorbed through data inputs D_k , $k = 0, \dots, 9$ by XORing 7-bit words with the contents of the corresponding stages S_j^{111} , $j = 8, 9, 15, 16, 18, 27, 28, 34, 35, 36$. For each data input D_k there is a corresponding data output O_k , which outputs the aforementioned sum. For example, the stage S_{36} is updated by absorbing a padded message word from D_9 : $S_{36}^0 \leftarrow (S_{36,0}^{111}, S_{36,1}^{111}, \dots, S_{36,6}^{111}) \oplus (m_{63}, 0, \dots, 0)$. Depending on the phase, the data inputs D_k carry the associated data AD , message M (or ciphertext C), and the K bits. Since the rate part of internal state consists of 64 bits, AD , M or C , must be padded to a multiple of 64, using the padding rule (10*). The resulting padded bitstring is then divided into 64-bit blocks for absorbing. In Algorithm 2, ℓ_X , where $X \in \{M, AD, C\}$, denotes the length in blocks, i.e., number of 64-bit blocks of X after padding. Refer to [6] for details on padding.

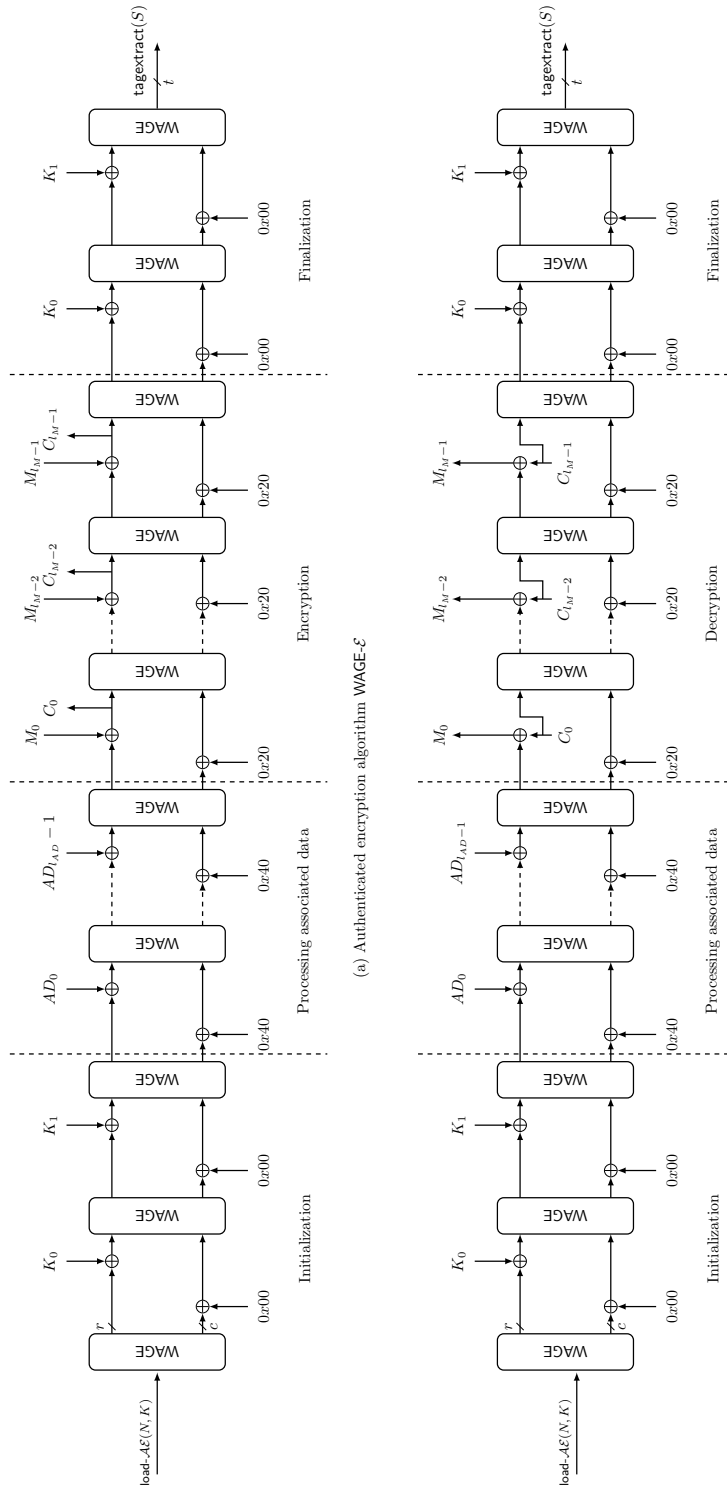


Figure 3.10: Schematic diagram of the WAGE- \mathcal{AE} -128 algorithm (Figure from [6])

3.3 Computer hardware

This section begins with implementation technologies in Subsection 3.3.1, followed by a section on implementation efficiency and different metrics in Subsection 3.3.2. Hardware design choices, focusing on datapath implementations, are presented in Subsection 3.3.3; the purpose of this subsection is establishing common terminology for the rest of the thesis.

3.3.1 Implementation technologies: ASICs and FPGAs

In this thesis, the term ASIC refers to (Standard-Cell-Based) Application Specific Integrated Circuit: the logic gates are pre-designed, pre-tested and pre-characterized ([62]), and finally stored in a library as standard cells. The design flow for ASICs starts with design entry using a hardware design language such as VHDL (Subsection 3.3.4) or Verilog. The synthesis tools (computer-aided design (CAD) tools) use the library cells to convert the HDL design into a chip layout. The implementation results for the case studies were obtained for the 65nm CMOS technology using Synopsis Design Compiler and Cadence SoC Encounter.

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. Basic logic unit is always a lookup-table (LUT), which allows implementation of an arbitrary Boolean function in n variables, where n depends on the specific FPGA device. The greatest advantage of SRAM-based FPGAs is their flexibility; modifying the designed and even implementing circuits is fast.

A brief comparison of ASICs and FPGAs

Compared to ASICs, FPGAs have a big advantage when time-to-market is critical due to a shorter development cycle. Another advantage of the FPGAs is reconfigurability, which it allows modifications and updates, and also makes FPGAs attractive for ASIC prototyping. Once fabricated, ASIC cannot be altered. Nevertheless, when comparing speed, area and power consumption, an equivalent ASIC circuit is always preferable. An ASIC solution is also extremely time consuming and expensive, however, these drawbacks disappear for large production volumes. As already mentioned, compared to ASIC, the area is always larger when the same design is implemented on an FPGA. Kuon and Rose [63] compared the performance of a 90-nm CMOS FPGA and 90-nm CMOS standard-cell ASIC using implementations of carefully designed benchmarks and found the following: the area complexity when implemented on an FPGA is in 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).

In general, a huge percentage of an FPGA device is used to provide the programmability. 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 programmable interconnects: a signal path in an equivalent ASIC circuit could be much shorter, hence lower delay.

An often overlooked feature of the FPGAs are the “free registers”. Usually, there is at least one register paired up with each LUT, and sometimes two registers. They can be used in different configurations, with or without reset/set and with or without chip-enable control signals. Furthermore, FPGAs, for example the Xilinx Spartan-6 devices have special shift register primitives SRL16 available on the so called SLICEM LUTs: they allow for a significant area reduction.

Another distinction comes from the nature of the LUTs: the look-up tables are nothing more than truth tables, and the most important parameter is the number of input and outputs of a LUT. These values are not set by the hardware designer, but are fixed for a given FPGA. For example, Xilinx FPGAs can have 4-input/1-output or 6-input/2-output LUTs, and can implement a Boolean function in maximum 4, respectively 6, variables. Often gate counts, i.e., number of NOT and two-input AND, OR, XOR¹¹ gates, are used to compare designs, as will be discussed in more detail in Section 4.2.3 in Chapter 4. For FPGAs, the gate counts are (almost) meaningless. For ASIC, the area can always be expressed in terms of gate counts, however, a lot depends on the specific ASIC library used.

3.3.2 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 [65]. 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, where a path is a sequence of interconnects and logical elements. 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.

The **throughput** measures the amount of data processed per time. Based on how the data and the time are measured, there are slight variations of throughput. They are presented in incremental fashion:

¹¹especially in binary finite field, where XOR gate is used for the modulo 2 addition

1. data measured in parcels, time measured in clock cycles

$$T_{\text{put}} = \frac{\#parcels}{\#clk.cycles} \left[\frac{parcel}{cycle} \right] \quad (3.20)$$

2. data measured in parcels, time measured in seconds

$$T_{\text{put}} = \frac{\#parcels}{\#clk.cycles} = \frac{\#parcels}{\#clk.cycles \cdot \frac{\#seconds}{clockcycle}} \left[\frac{parcel}{seconds} \right] \quad (3.21)$$

This form of throughput is obtained by including the clock cycle period, measured in seconds-per-clock cycle.

3. data measured in bits, time measured in seconds

$$T_{\text{put}} = \frac{\#parcels}{\#clk.cycles} = \frac{\#parcels \cdot \frac{\#bits}{parcel}}{\#clk.cycles \cdot \frac{\#seconds}{clockcycle}} \left[\frac{bits}{seconds} \right] \quad (3.22)$$

This form of throughput is obtained by including the bit-width of the parcel.

Throughout literature, the throughput is measured as $\frac{bit}{cycle}$, i.e. bits-per-cycle or as $\frac{word}{cycle}$, i.e. words-per-cycle; the latter corresponds with equation (3.20), using word as synonym for parcel or output. For example, during WG stream cipher (Subsection 3.2.4), the key initialization algorithm, the parcel is a m -bit word, yielding words-per-cycle. During the WG running phase, the parcel is 1 keystream bit, yielding bits-per-cycle. Some literature [44] defines the throughput as bits-per-cycle multiplied by the clock frequency, yielding bits-per-second.

The **area complexity** in FPGAs is given in terms of resources used by the design, for example the number of used slices (a collection of LUTs and registers, concrete configuration depends on the specific FPGA device), LUTs, storage elements, input-output blocks (IOBs), etc. Since both LUTs and registers are contained in slices, the number of slices will be used as the primary area metric for FPGAs. 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. GE is preferred metric to μm^2 , because it is believed to allow very rough comparisons across different fabrication technologies and gate libraries. However, as stressed by [66], the GE metric is technology specific, and direct comparison of area expressed in GEs across different technologies is not possible.

Power and energy are becoming more and more significant as metrics for various reasons: they affect battery life, can enforce a limit of the clock frequency, causes higher temperatures which in turn reduces the lifetime of the device, and increases dissipated heat of hand-held devices etc. In general, total power consumption depends on the number of logic cells in the circuit, connections between them, 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 ([44]). Power and energy are closely related. According to [67], energy is becoming more important for determining the lifetime of battery operated devices, suitable for lightweight applications. A frequently used metric is *energy-per-bit*, calculated by dividing the total power consumption by the throughput, both obtained at the same clock frequency [44].

Since it is difficult to compare two designs based on more than one metric (for example the clock period and the area), the derived metrics are used to measure design efficiency, 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, the optimality metrics are derived: the *throughput per time-area product* $\mathbf{o}_1 = \frac{\mathbf{T}}{tA} = \frac{\mathbf{T}f}{A}$ and *throughput per time-area-power product* $\mathbf{o}_2 = \frac{\mathbf{T}}{tAP} = \frac{\mathbf{T}f}{AP}$. The value \mathbf{T} in \mathbf{o}_1 and \mathbf{o}_2 is the throughput measured in parcels-per-cycle (equation (3.20)). Because power analysis is tedious it is often approximate it with area as $\frac{\mathbf{T}f}{A^2}$. This ratio is also preferred to the $\frac{\mathbf{T}}{AP}$, because of sensitivity of power analysis to differences between the cell libraries and to tool configurations [68]. 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 [69], or by increasing the frequency, which in turn causes increased area and power consumption. Metrics like $\frac{f}{A}$ and $\frac{f}{A^2}$ put a better perspective on the actual improvement of the design by some optimization attempt; they emphasize the tradeoffs between the throughput and area.

3.3.3 Hardware design – datapath

In general, hardware can be classified as datapath, storage and control, based on its primary task. Finite field applications certainly fall into the datapath class. The datapath circuits can be further distinguished based on exploitation of parallelism¹² and utilized degree of resource sharing. Both parallelism and resource sharing depend on the algorithm and data dependencies. This section gives a brief overview with the purpose of establishing common terminology [70, 71]. Two cases are covered, one without data dependencies and one with data dependencies. In both cases, a function F , performed four times, will be used.

The two cases are explained with the use of data-flow diagrams (DFDs), which can capture both the data-flow and behaviour. One instance of the function F is represented as a datapath component

¹²spatial parallelism, not temporal parallelism as in pipelining

F in a square. The subscripts are used to enumerate the instances of F in the circuit. The keywords replicate and reuse are not a part of the DFD notation and are used for clarification only. The horizontal dashed lines represent clock cycle boundaries. Each time a signal crosses the clock cycle boundary, registers are inferred. All registers are located at the outputs, i.e., all modules are considered to be *combinational-input/registered-outputs*, abbreviated CIRO. For example, a pipeline can be considered as a series of CIRO modules. All ports are always considered to be of the same type, mixing, e.g., have one registered and one combinational input, is not allowed. Possible combinations of combinational and registered inputs and outputs are shown in Figure 3.11, with the CIRO shown second. A register at the clock cycle boundary crossing is shown for one input of the RIRO example.

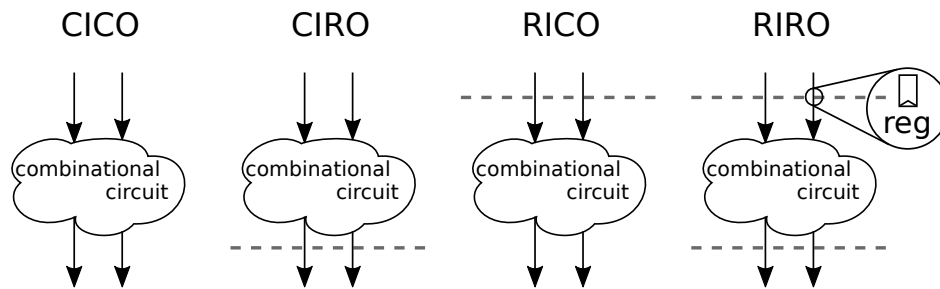


Figure 3.11: Possible combinations of combinational and registered inputs and outputs

Figure 3.12 shows datapaths with no data dependencies for the following implementation decisions: (a) fully parallel, (b) partially parallel, and (c) sequential datapath. The datapath DFDs are shown on top, and the actual hardware is shown below each DFD. The actual hardware contains only the hardware used, the multiplexers and the registers implied by the corresponding DFD, but the reuse is lost. Figure 3.12(a) shows an example of a fully parallel datapath. Four instances of F are used and all four output values computed concurrently. The simplest example of such a datapath is the bit-wise XOR on vectors of length 4. The partially parallel version of this datapath, computing two outputs at a time (per clock cycle), is shown in Figure 3.12(b). The outputs from the first clock cycle must be stored and preserved, hence chip enable control signals (CE) for the registers shown below the DFD. The last DFD is showing a sequential datapath, with only one instance of F, reused four times. Neglecting the area multiplexers and control signals, the datapath in Figure 3.12(a) has the highest area with four instances of F, and the datapath in Figure 3.12(c) the lowest area with only one F. Ignoring the multiplexers, the critical path and hence the clock period remains unchanged, but the number of clock cycles needed for the final result increases.

Figure 3.13 shows the datapaths with data dependencies that prevent parallelism. Figure 3.13(d) shows serial computation with no resource sharing. It uses four instances of F. Figure 3.13(e) has two instances of F, which are reused to obtain the final output over two clock cycles. This is a partially sequential datapath. The fully sequential datapath in Figure 3.13(f) has only one instance of F, which is reused four times, i.e., has a higher degree of resource sharing (reuse). The area in terms of datapath components F drops with higher reuse. The critical path and hence the clock period is reduced with higher reuse, but the number of clock cycles increases.

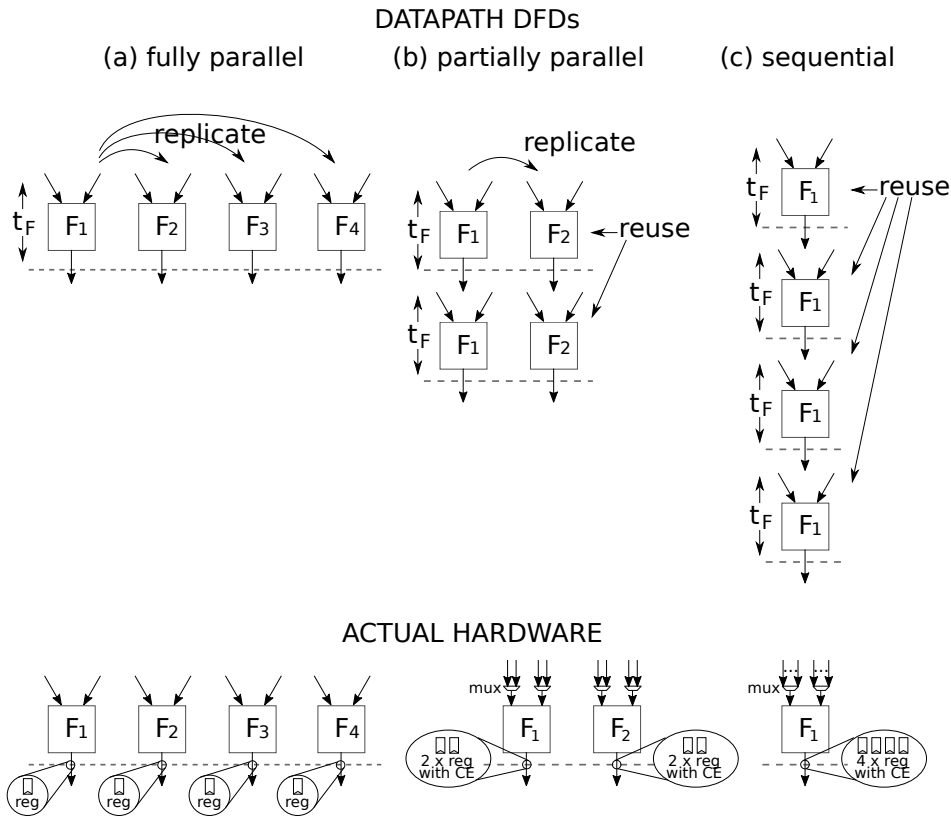


Figure 3.12: Fully parallel, group-level and serial computation in the absence of data dependencies

The DFDs also capture the tradeoffs between the different options. The examples are illustrating the *critical path* delay t_{cp} , that is, the delay through the longest viable combinational path through the circuit. If multiplexers are used, the delay becomes $t_{cp} = t_F + t_{mux}$, and the lower bound for the clock period is $t_{clk} = t_{cp} + t_{reg}$. The delays are simplified: the delays for the registers and the multiplexers are omitted, and it must be noted that especially the multiplexers can add a significant delay. Throughput given in terms of parcels processed (i.e. results computed) per clock period is lowered with every reuse. The datapaths without resource sharing, i.e., fully parallel in Figure 3.12(a) and serial in Figure 3.13(d), have throughput 1. Both datapaths with the word “partial” have throughput 1/2, and the two sequential datapaths (Figure 3.12(c), Figure 3.13(f)) have the lowest throughput of 1/4.

Only unpipelined datapath are considered. By changing (some) clock cycle boundaries into interstage borders, partially and fully pipelined datapaths can be obtained. Pipelining is beyond the scope of this work. This work focuses on unpipelined datapaths with no reuse, i.e., fully parallel or serial datapaths (Figures 3.12(a) and 3.13(d)).

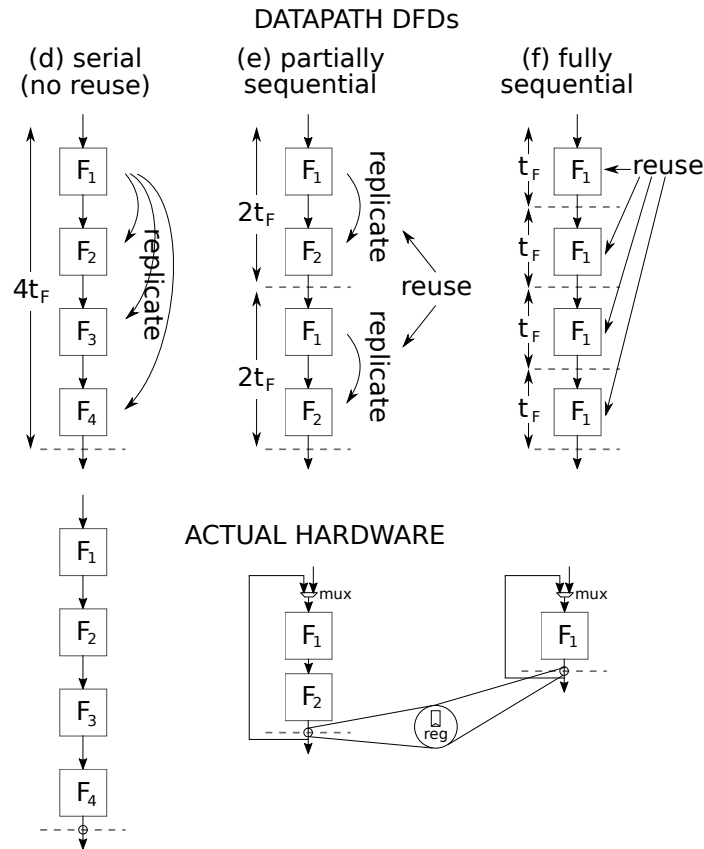


Figure 3.13: Serial, partially sequential and fully sequential computation with data dependencies

Constant array implementations

In a constant array implementation the output values (results of the implemented expression) are precomputed for all possible inputs and stored as constants. This is a lookup table design, where the input serves as an index to the array, selecting appropriate precomputed constant as output. Any datapath can be implemented as a constant array, with some reasonable limitations on its size. In ASIC, the constant arrays are not stored in hardware as actual memory arrays, but rather as a net of AND, OR, XOR and NOT gates, derived and optimized by the synthesis tools. As a consequence, the area needed for a constant array implementation depends on possible optimizations and differs from the theoretical estimate computed as $(\text{number of entries}) \times (\text{number of bits per entry})$. For an m -bit input, the array has 2^m possible entries. Number of bits per entry is the length of the precomputed constants in bits. Implementation using an actual memory module is also possible. Terminology constant array is preferred to lookup table to avoid the confusion with FPGA LUTs.

3.3.4 Hardware design entry using VHDL

The intent of this short section is to explain the terminology used in this thesis. It does not provide a language reference nor a summary there of, and will explain basic terminology in a very brief manner, omitting more advanced concepts.

VHDL stands for VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. VHDL is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware [72].

In VHDL a hardware circuit is called a *design entity* or a *module*. A module is defined by an *entity declaration* together with a corresponding *architecture body*; the two parts will be referred to simply as *entity* and *architecture*. The entity defines the interface between the module and the environment by specifying the input and output ports. The architecture defines the functionality of the module by specifying the relationships between the module inputs and outputs. It has two parts, declarative part and statement part, separated by the keyword `begin`. The optional declarative part contains declarations, e.g., of internal signals and constants. The statement part contains concurrent statements that describe the dataflow: assignments, processes, component instantiations etc. Assignments specify how (r.h.s. of the assignment) to drive a target signal (l.h.s. of the assignment). Examples of the r.h.s. of an assignment are another signal, or a Boolean expression. Conditional assignments have a `when-else` formatted r.h.s., and allow implementation of multiplexers. Processes can be either combinational or clocked, and the clocked processes are used to specify registers. Component instantiations allow hierarchical designs: the circuit is partitioned into (smaller and simpler) submodules (subcomponents). Each submodule has its own entity-architecture pair, but can be used (multiple times) as a component to build other modules. The component instantiation statement specifies the entity-architecture pair, which identifies the submodule, and a port-map, which specifies how this instance of the submodule is connected within the (parent) module. Extensive literature on VHDL exists, for example [73].

3.3.5 High-Level Synthesis

There are many good survey papers on high-level synthesis, e.g., McFarland *et al.* [74], Gajski *et al.* [75], Coussy *et al.* [76], followed by some recent surveys focusing on the current state of specific HLS tools [78, 79]. which give a good overview of the tasks HLS has to perform and some insight to its evolution. The task of HSL tools is to transform a behavioural description into register-transfer level (RTL) design [74, 76]. Due to McFarland [74], behavioural description specifies the way the system or its components interact with the environment, i.e., mapping from the inputs to the outputs. The behavioral description is entered using a high-level language (HLL), for example C, C++, or its extensions like SystemC or even MATLAB. The HLLs are unable to capture timing concepts like cycle-to-cycle behavior, however, the tools require designer-specified constraints (e.g. timing constraint) and optimization goal (e.g. timing-driven optimization).

The tasks of the HLS tools [74, 75, 76] include:

- compilation and modelling
- resource allocation
- scheduling of operations to clock cycles
- binding
- generation of the RTL architecture.

Compilation starts with operation decomposition, identification of data and control dependencies and transforms the behavioural description into dataflow graph (DFG) or control and dataflow graph (CDFG). A DFG can capture parallelism, but does not support loops. In a DFG, nodes represent the operations and edges their inputs and outputs. In a CDFG, nodes are the basic blocks, which contain data dependencies but do not include any branches, and edges, which can be conditional, capture the control flow between them. The only parallelism in a CDFG is within the basic blocks, but further analysis is needed to find parallelism between basic blocks. This is accomplished using techniques such as loop unrolling, loop pipelining, loop merging and loop tiling. To summarize, during this process the HSL tools extract parallelism, find the common subexpressions, perform loop unrolling, etc.

Allocation, scheduling and binding have access to a library of RTL components, i.e. available hardware resources. The components are annotated with characteristics such as area, delay, etc. Datapath (and control) allocation determines the type and number of components, generates the interconnects and performs hardware minimization. Scheduling cuts the DFG or CDFG into clock cycles, and schedules operations in such a way that the functionality is preserved. This process is aware of the available resources and an operation can also be scheduled to more than one clock cycle. Following is the binding: operations to components, variables to storage elements, transfers to interconnects. Optimizations such as register reuse for variables that have nonoverlapping lifetime is possible at this stage. However, different levels of binding are possible, where less binding delegates more tasks to logic and physical synthesis, which have more room for optimization as they have more accurate timing estimates and access to placement and routing. Different levels of binding are captured with code annotations in the RTL design generated by the HLS tool. Allocation, scheduling and binding can be done in different order, for example, if the optimization goal is to minimize the total area, including interconnect length, while meeting the timing constraints, HSL tools start with scheduling and perform allocation during scheduling. HLS tools use a lot of different approaches, from graph theory, game theory, genetic algorithms, integer linear programming, etc.

Another frequently mentioned theme is the distinction (and mix) of top-down and bottom-up approaches. McFarland [77] speaks about the evolution of HLS from experience of human designers, who rely on their knowledge of low-level characteristics of structures used in the implementation to guide high-level decisions. This information is included as a library and used to evaluate different RTL structures for the same behavioural description. Many authors also discuss partitioning, clustering and even place and route information used to improve RTL designs[74, 77, 81]. An early

attempt is the BUD program (bottom-up design), which performs global allocation and scheduling by evaluating different design decisions based on their effect on the design, i.e., a generate-and-test algorithm.

The aforementioned brief discussion on evaluation of different RTL structures leads to the concept of design space exploration. One of the advantages of top-down approach is, thanks to [80], the flexibility in exploration of possible designs, e.g., how much of the possible parallelism is exploited. A brief discussion on the design space exploration, viewed through two metrics, namely area and delay, is also included in the survey paper by McFarland *et al.* [74]. It touches the problem of complexity arising from the extremely large number of possibilities and the difficulty evaluating designs in the early stages of the design flow. Numerous approaches to automated design space exploration (DSE) have been proposed for different levels of abstraction and always in conjunction with other (lower level) synthesis tools [82, 83, 84]. In its most basic forms, the design space is considered using area and delay metrics only, while others also include power [84]. As there are many parameters that affect the design in numerous ways, many iterations and design space reduction approaches are used.

Chapter 4

Related work

4.1	Hardware implementations of selected cryptographic schemes	54
4.2	Cryptographic hardware and complexities	64
4.3	Hardware design automation and synthesis tools	67

4.1 Hardware implementations of selected cryptographic schemes

With the purpose to outline lightweight cryptography and to show the applicability of FSR based systems, this section begins with two relevant topics; namely a brief historical discussion on lightweight cryptography and its applications (Subsection 4.1.1), and a collection of the hardware implementation results for Grain and Trivium, two FSR-based ciphers (Subsection 4.1.2). This subsection is followed by the hardware implementations of WG stream ciphers (Subsection 4.1.3). The WG-stream ciphers are also FSR-based, but operate over an extension field; for larger WG instances, tower field implementations are beneficial, hence presented in Subsection 4.1.4.

In this thesis, FSR based systems are presented in Chapter 6, the WG cipher is used as a case study throughout many chapters, and the WG-permutation based cipher WAGE is used as a case study in Part VI. As this thesis focuses on ASIC implementations, many FPGA results from the literature are omitted.

4.1.1 Lightweight cryptography and its applications

In the past 20 years, the trend started to move towards lightweight cryptography. In 2004, ECRYPT (European Network of Excellence for Cryptology) launched the eSTREAM competition (Subsection 3.2.3), which included stream ciphers for hardware applications with highly restricted resources (Profile 2) [4]. Stream ciphers were introduced in Subsection 3.2.2. In the same year, The State of the Art of Stream Ciphers, organized by ECRYPT, included a session dedicated to implementation issues [85, 86]. They suggest metrics for evaluating the implementation efficiency and mention the trade-offs between the security and the implementation cost. The cost was identified as the implementation area [85, 86], the power consumption [86], and the energy consumption

[85]. Both papers consider the different sizes of particular gates, e.g., a register is more expensive than an XOR gate. Some differences in hardware and software implementations were addressed in [85]. More on eSTREAM competition will follow in Subsection 4.1.2.

Eisenbarth *et al.* presented a detailed analysis and survey of both hardware and software implementations of lightweight ciphers [87]. They focus on trade-offs between three design goals, namely security, cost and performance. The survey covers symmetric ciphers (mainly block ciphers, but include a few stream ciphers), with detailed analysis of the hardware-oriented block cipher PRESENT as an example of a cipher designed to be lightweight [88, 89]. The survey also covers asymmetric ciphers, more specifically, elliptic curve cryptography processor used in hardware-software codesign; with the smallest area scalar multiplication reaching more than 10kGE. Hardware implementation of block cipher PRESENT in [87] required 1570 GE, and was further reduced to 1000 GE in [89]; the results are comparable to the 1294GE for Grain80 and 1857GE for Trivium, two of the eSTREAM Profile 2 finalists. Another lightweight block cipher is Hummingbird, with the area of 2225 GE [90, 91]. As a comparison, a serialized AES-128 encryption only core in [92] reported 2400GE, and [93] a 1947GE encryption and 2090GE decryption core. Batina *et al.* [94] performed a comprehensive area, power, and energy analysis of several lightweight block ciphers and compared them to AES. Their findings report, among other, some anomalies, e.g., the largest AES implementation, using a look-up table based S-box implementation, consumes the least dynamic power. Then in 2013, two families of lightweight block ciphers have been proposed, the hardware oriented SIMON and software oriented SPECK [95]. Both families are parametrized by the block and the key size, and their implementation area ranges from 523 to 958 GE. A new design Simeck, combining SIMON and SPECK, and parametrized in the same way, has an implementation area ranging from 505 to 924GE [96]. Another block cipher, designed to be lightweight, is GIFT [97]. It has two versions, both using 128-bit key; the smaller cipher has a 64-bit state and can be implemented using 1345GE, and the bigger cipher a 128-bit state, and needs 1997GE.

The CAESAR competition (Subsection 3.2.6) was launched in 2014 [39]. The final CAESAR portfolio is organized into three use cases, with use case 1 being the lightweight applications (resource constrained environments). In march 2018, Ascon and ACORN were chosen for the use case 1. The FPGA implementation area results on a Xilinx Spartan-6 FPGA device show 1640 LUTs for Ascon and 1396 LUTs for ACORN-32 (a high-speed version with 32 parallel output bits) [98]. The low-area ASIC implementation of Ascon in [99] presents 2.57GE. The trend for lightweight continues. Yu *et al.* [100] developed their own architecture, which on a 65 nm ASIC has an area of 1960 GE and is 13% smaller than architectures of Moradi [92], Hamalainen [101], and Mathew [93], using the same cell library.

A passage from the NIST LWC call for submissions (Subsection 3.2.7) presented the lightweight cryptography as follows [60]. Lightweight cryptography is a subfield of cryptography that aims to provide solutions tailored for resource-constrained devices. There has been a significant amount of work done by the academic community related to lightweight cryptography; this includes efficient implementations of conventional cryptography standards, and the design and analysis of new lightweight primitives and protocols [60]. In an earlier report [66], an example of a low-cost RFID tag is mentioned, that allows up to 2000GE for security.

Biryukov and Perrin conducted an extensive survey on lightweight symmetric cryptography [102]. The survey covers over 100 algorithms from academia, proprietary algorithms, and algorithms from government agencies. They suggest that lightweight cryptography should be divided into ultra-lightweight cryptography (highly specialized algorithms providing one function with high performance on one platform) and ubiquitous cryptography (more versatile algorithms, both in terms of functionality and implementation on a wide variety of platforms).

The brief (historical) discussion above provides some insight into lightweight cryptography, and establishes the target for the hardware implementation area. Unfortunately, the GE count is not a technology independent area metric for ASIC implementations, and the results stated above range from 22 to 350nm ASIC technologies. Furthermore, the implementations allowed different cells (e.g., scan flip-flops), use different effort levels for the synthesis tools (compile , compile ultra, clock gating), and last but not least, present the results of encryption cores without decryption, etc. Gong [103] defines the area requirements for lightweight cryptography as follows: a cryptographic primitive is said to be lightweight if it requires less than 2000GE, and for the cryptographic primitive together with a mode, e.g., authenticated encryption, the requirement is loosened up to 3000GE.

Block ciphers and elliptic curve cryptography are beyond the scope of this work. However, a discussion on tower-field constructions used in hardware implementations of the AES block cipher is included in Section 4.1.4.

Applications

The authors of [87] briefly introduced pervasive computing and mention RFID (radio frequency identification) in food-chains and health-monitoring applications. Similarly, Engels *et al.* mention RFID tags, smart cards and wireless sensor nodes, usage in access control, supply-chain management, home automation, and healthcare [90]. In [103] Internet of Things devices are covered, including sensors, actuators, RFID tags and microcontrollers equipped with radio frequency transceivers, with applications in industrial and building control, e-health, smart energy grid, home automation, self driving cars, etc. The main topic of [103] is lightweight cryptography for IoT, and a classification of IoT devices is based on how they connect to internet: (i.) simple transmitter/receiver pairs (e.g., RFID tags), (ii.) single input and single output devices (e.g., Bluetooth, ZigBee), and (iii.) multiple input and multiple output (e.g., WiFi, 4G-LTE, 5G) devices.

4.1.2 The eSTREAM project: Grain and Trivium

Both Grain and Trivium are FSR based and involve only simple binary operations (XOR, AND). They both have a small area and 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.

An ASIC implementation of eSTREAM candidates, reported by Gürkaynak *et al.* in [104], 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 possible 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 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 4.1. However, only Grain-128 was implemented with radix 32 in [44, 105].

Stream Cipher	Source	ASIC Technology	Area [μm^2] or [GE]	Frequency [MHz]	radix	Throughput T [Mbps]
Grain	[104]	250nm	119.821 μm^2	300	16	4475
	[107]	90nm	4911 μm^2	565	1	565
			10548 μm^2	495	16	7920
	[44]	130nm	1294 GE	724.6	1	724.6
			3239 GE	617.3	16	9876.5
1857 GE			925.9	1	926	
†		4617 GE	452.5	32	14480	
Trivium	[104]	250nm	144.128 μm^2	312	64	1856
	[107]	90nm	7428 μm^2	840	1	840
			13440 μm^2	800	64	51200
	[44]	130nm	2580 GE	327.9	1	327.9
1921 GE			348.4	64	22299.6	

Table 4.1: ASIC implementation results for Grain and Trivium found in literature

† marks implementations of Grain-128

In [107], Gaj *et al.* provided detailed FPGA and ASIC results for five eSTREAM candidates, including Grain and Trivium, focusing on parallelization 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. ASIC implementations of Phase 3 candidates, reported by Good and Benaissa in [44], consider many different performance metrics for comparison. The authors also 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 Sboxes, since they are significant consumer of area and power, among others.

Numerous FPGA implementations of Grain and Trivium can be found in [108, 106, 109, 110]. Some use special FPGA features, e.g., the Xilinx Virtex-II SRL16 primitive [106], while others consciously refrain from use of SRL primitives, e.g., [109]. The FPGA implementation results are omitted from Table 4.1.

4.1.3 WG hardware implementations

The first member of WG stream cipher family to be implemented in hardware was the eSTREAM candidate WG-29 [7]. For the $\mathbb{F}_{2^{29}}$, a type II optimal normal basis exists, which allows efficient field arithmetic. In [111], 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 [112]. The same paper also reports efficient polynomial basis implementations of WG-16.

An implementation of a lightweight WG stream cipher WG-5, targeting passive RFIDs, was reported in [68]. Selected metrics of their implementation results are shown in upper part of Table 4.2, omitting the power metric and optimality scores derived using the power results. 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. An interesting feature of this paper is their parameter selection, aiming to reduce the hardware cost. Based on ASIC implementation results for the chosen frequency of 100kHz, WG-5 outperforms the ciphers it was compared to, including Grain and Trivium.

WG Cipher	Source	ASIC Technology	dec	Area [GE]	Throughput [kbps]	Radix	$\frac{T}{A^2}$
WG-5 †	[68]	130nm	1	1229	100	1	66.2
			11	1235	100	1	65.6
			1	1350	200	2	110
			11	1360	200	2	108
WG Cipher	Source	ASIC Technology	Architecture	Area [GE]	Speed [MHz]	Radix	$\frac{T}{A}$
WG-8 ‡	[113]	65nm	CA	1786	500	1	0.28
				3942	610	11	1.70
			TF1	7523	229	1	0.03
				42762	122	11	0.03
			TF2	3162	260	1	0.08
				22668	205	11	0.10
			TF3	2981 GE	254	1	0.08
				19882	205	11	0.11

† WG-5 with 80-bit key and IV, and LFSR of length 32

‡ WG-8 with 80-bit key and IV, decimation exponent 19, and LFSR of length 20

Table 4.2: Post-PAR CMOS implementation results for WG-5 and WG-8

Another instance of lightweight WG stream ciphers, the WG-8, was reported in [113]. Selected metrics of their implementation results are shown in Table 4.2, omitting the power metric and optimality scores derived using the power results. It explores four different hardware architectures. The first implementation is a constant array based design, denoted “CA” in Table 4.2 (with one

array holding the WGP values and one array holding the WGT values). Then two tower constructions $\mathbb{F}_{(2^4)^2}$ were implemented, using different defining polynomials for the first extension, denoted “TF1” and “TF2” in Table 4.2. 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, denoted “TF3” in Table 4.2, 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 [8]. 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, the best results were achieved for the table look-up based design.

perf. goal	modules used	Speed [GHz]	Area [kGE]	\mathbf{o}_1	\mathbf{o}_2
WG(16, 32) implementation from [115]					
$\mathbb{F}_{((2^2)^2)^2}$ field construction					
WG(16, 32) - using $\ell_1(x)$					
f, \mathbf{o}_1	L1 d5 s7 c8	2.17	22.5	9.6	4.3
A, \mathbf{o}_2	L0 d1 s2 c2	0.88	10.9	8.1	7.4
WG(16, 32) - using $\ell_2(x)$					
f, \mathbf{o}_1	N1 d5 s7 c8	2.13	18.0	11.8	6.6
A, \mathbf{o}_2	N0 d1 s2 c2	0.93	11.5	8.1	7.0
$\mathbb{F}_{(2^4)^4}$ field construction					
WG(16, 32) - using $\ell_3(x)$					
f	C2 d9 s13 c12	2.44	26.3	9.27	3.5
A, $\mathbf{o}_1, \mathbf{o}_2$	C1 d7 s11 c10	1.79	17.0	10.5	6.2
WG(16, 32) - using $\ell_4(x)$					
f	T2 d9 s13 c12	2.38	27.0	8.8	3.3
A, $\mathbf{o}_1, \mathbf{o}_2$	T0 d7 s11 c10	2.08	20.8	10.0	4.8
WG(16, 32) implementation from [8]					
M_{16}/I_8 level		0.55	12.0	4.6	3.8

for $\ell_1, \ell_2, \ell_3, \ell_4$ see Section 20.4

Table 4.3: Post-PAR CMOS 65nm implementation results for the WG(16, 32) keystream generators

WG-16 was studied in [8, 9, 111, 112, 114] and is intended for use in confidentiality and integrity algorithms in mobile communications, such as 4G-LTE networks [114]. As such, WG-16 has stronger security requirements when compared to WG-5 and WG-8, and is using 128-bit key and

IV; is not a lightweight instance. New implementation of WG-16 [115] is using two different tower field constructions $\mathbb{F}_{((2^2)^2)^2}$ and $\mathbb{F}_{(2^4)^4}$ and three new LFSR polynomials ℓ_2, ℓ_3 and ℓ_4 in addition to the original polynomial ℓ_1 used in [8, 111, 114, 112]. The highest frequency WG(16, 32) keystream generator, obtained for the 65 nm ASIC library, reached the clock speed of 2.44GHz at 26.3kGE, and the smallest area keystream generator the clock speed of 0.88GHz at 10.9kGE, shown in Table 4.3. Modules were synthesized for different performance goals: highest frequency \mathbf{f} , smallest area \mathbf{A} , best optimality scores $\mathbf{o}_1 = \frac{\mathbf{T}\mathbf{f}}{\mathbf{A}}$ and $\mathbf{o}_2 = \frac{\mathbf{T}\mathbf{f}}{\mathbf{A}^2}$, where \mathbf{T} is the throughput in parcels-per-cycle. The highest frequency FPGA implementation on Xilinx Spartan 6 reached the clock speed of 256MHz using 631 slices for LFSR using ℓ_3 and $\mathbb{F}_{(2^4)^4}$ tower field construction.

Authors of [112] used three different implementation strategies: a standard, pipelined, and serialized design, coupled with two different multipliers. The results of [115] are compared to the [112] implementations using Karatsuba multiplier [116], which were both smaller and faster than their alternative, and are the only ones reported in Table 4.4. The 65nm ASIC results they present are obtained pre-PAR, and for a fair comparison, the pre-PAR results for modules using $\ell_1(x)$ from [115] are reported in Table 4.4. L1d5s7c8 shows a speedup of 1.82 at a moderate 27% increase in area when compared to pipelined architecture from [112], and module L0d1s2c2 a significant speedup of 5.75 at 22% area increase when compared to standard architecture from [112]. Their smallest (serial) design takes only 51% of the area used by the smaller implementation L0d1s2c2. However, their serialized architecture has a throughput $\frac{1}{6}$, and as was pointed out by [117], even a lightweight stream cipher should have a throughput at least $1 \frac{\text{bit}}{\text{clk.cycle}}$ and WG-16 is not a lightweight stream cipher. When compared to the ASIC pre-PAR results for the ZUC implementation reported in [118], L1d5s7c8 can reach the same frequency at a slightly higher area, but has significantly lower optimality scores due to the lower throughput.

An ongoing project, WG-lite Design Space Exploration [119], is exploring hardware implementations of WG ciphers defined over small finite fields \mathbb{F}_{2^m} , where $m = 5, 7, 8, 10, 11, 13, 14, 16$. From previous work on WG-5, WG-8, WG-16 and WG-29, it is known that for small instances, such as WG-5 and WG-8, constant array implementations yield a smaller area compared to implementation using discrete components, such as multipliers and exponentiations. One of the objectives of the WG-lite Design Space Exploration project is to find the threshold when the constant array implementations become larger than the discrete component implementations. The implementation results for the WG permutation and transformation modules using polynomial bases, implemented using (a) discrete components and (b) constant arrays are reported in [120]. A section from the implementation results from [120] is summarized in Table 4.5: it shows the implementation results for the non-decimated WG permutation modules using polynomial basis over all existing primitive polynomials for the given finite field \mathbb{F}_{2^m} . The results are presented as minimum and maximum area, followed by the mean and standard deviation. They show the tipping point between discrete component and constant array implementation for the finite field $\mathbb{F}_{2^{10}}$. Future work for the WG-lite Design Space Exploration includes normal basis implementations.

Basis used	design used	Speed [GHz]	Area [kGE]	\mathbf{o}_1	\mathbf{o}_2
WG(16, 32) implementation from [115]					
WG(16, 32) - using $\mathbb{F}_{((2^2)^2)^2}$ and $l_1(x)$					
TFB	L1 d5 s7 c8	2.50	13.5	1.9	1.4
TFB	L0 d1 s2 c2	1.11	9.81	1.1	1.2
WG(16, 32) implementation from [112]					
PB	standard	0.19	8.06	0.2	0.3
PB	pipelined	1.37	10.6	1.3	1.0
PB	serialized*	0.71	5.03	0.2	0.5
Other ciphers					
ZUC†[118]		2.50	12.5	64.0	5.12

$$* \text{ throughput } \mathbf{T} = \frac{1}{6} \frac{\text{bit}}{\text{clk.cycle}}$$

$$\dagger \text{ throughput } \mathbf{T} = 32 \frac{\text{bit}}{\text{clk.cycle}}, \text{ TSMC 65nm ASIC}$$

Table 4.4: Pre-PAR CMOS 65nm implementation results for WG(16, 32) keystream generators

	Discrete Components				Constant Array			
	min	max	mean	sd	min	max	mean	sd
WG-5	426	473	446	21	46	57	53	4
WG-7	1157	1390	1316	66	247	268	257	6
WG-8	1937	2078	2035	42	508	574	542	24
WG-10	2520	3018	2928	87	2173	2257	2211	18
WG-11	3134	3674	3567	81	5104	5470	5282	79
WG-13	5642	6366	6170	99	18163	19731	18985	289
WG-14	7136	8102	7885	116	35299	38164	37338	414
WG-16	11052	12302	12036	145	127171	137624	135513	1596

Table 4.5: Pre-PAR CMOS 65nm implementation results for WGP modules with decimation 1: comparison of discrete component and constant array implementations from [120]

4.1.4 Use of tower fields in cryptography

A paper [121] from 1974, published by Green and Taylor 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 is to represent the field elements as powers of the generator, and they also provide primitive polynomials of small degrees over the aforementioned base fields. The rest of [121] 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 is an inversion algorithm for elements of \mathbb{F}_{q^m} with $q = 2^n$, proposed in 1988 by Itoh and Tsuji [122]. 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 [123, 124, 125, 126].

There is a series of papers from the 90's published by Paar [127, 128, 129, 130], reporting gate counts for VLSI implementations of finite field arithmetic in composite fields using polynomial basis representation for all extensions; most of these results were also reported as a part of his PhD thesis [131]. 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 by Morii and Kasahara in [124]. 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, Paar and Soria-Rodriguez published a paper [132] describing hybrid components that use parallel circuits for arithmetic operations 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 and 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 $2\mu\text{m}$ ASIC implementation. Guajardo and Paar revisited Itoh-Tsuji inversion in large fields in 2002 paper [133]: 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 and m optimized using iterates of the Frobenius map.

Harper *et al.* [134] discuss larger fields. In their discussion of elliptic-curve cryptosystems, one of the chosen underlying finite fields $\mathbb{F}_{2^{104}}$ was implemented as composite field $\mathbb{F}_{(2^8)^{13}}$. Elements of $\mathbb{F}_{(2^8)^{13}}$ were represented as polynomials over \mathbb{F}_{2^8} and the 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 the normal basis, and concludes that the implementation with the 8-bit base field elements is very suitable for software implementations.

A 1999 technical report on composite field arithmetic [135] 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 coprime n and m . The authors report comparison of total time needed for squaring, multiplication and inversion implemented using polynomial basis to optimal normal basis type I (ONBI) or to optimal normal basis type II (ONBII) representation. Multiplication was not conducted directly in ONBI/ONBII. Instead, the elements were converted to a different basis representation for the multiplication, and the product back to the ONB: (a) to a shifted polynomial basis for ONBI (for details refer to [136]), and (b) a permutation of the ONBII for ONBII (for de-

tails refer to [137]). 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 [138] from 2003, Sunar *et al.* 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.

The tower construction $\mathbb{F}_{(2^8)^4}$ is used in Snow3G: the [139] 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, [140]. This tower construction was used by Rudra *et al.* [141] 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.* [142] 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 [141].

Mentens *et al.* [143] improve the original S-box in [142] 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 a 5% area reduction. All the aforementioned tower field constructions isomorphic to \mathbb{F}_{2^8} use polynomial bases at each level of the tower.

Canright [144] conducted an exhaustive search and tree structure analysis to find the best matrices while testing both polynomial and normal bases at each level of the tower $\mathbb{F}_{((2^2)^2)^2}$. He was focusing solely on the area reduction and not examining the delay. His work includes common subexpression eliminations and detailed analysis of available logic gates, e.g., including NAND and XNOR when beneficial. In 2010 a mixed basis tower field construction for $\mathbb{F}_{((2^2)^2)^2}$ was reported by [146]; their Itoh-Tsuji inverters accept an input in normal basis representation and output its inverse represented in the polynomial basis. The choice of the polynomial basis representation of the inverse was based on a slightly more efficient matrix for the affine transformation. The authors also emphasize the link between the Hamming weights of individual rows of transition matrices and between the critical path delays. Moradi *et al.* [92] added the threshold countermeasures of [145] to the AES implementation from [144]. A very interesting application of tower field constructions was presented in [147]. 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, such that $p(x)$ is primitive.

Exhaustive search for best implementations was also performed in works [93, 148, 149]. Mathew *et al.* [93] performed the exhaustive search over all $\mathbb{F}_{(2^4)^2}$ constructions, considering the Hamming weights and the possible subexpression eliminations of the transition matrices between the tower field construction using polynomial basis on each level and the F_{2^8} polynomial basis given by $x^8 + x^4 + x^3 + x + 1$ specified in the standard. Besides the transition matrices, the cost of arithmetic operations in $\mathbb{F}_{(2^4)^2}$ was considered as well. They found the lowest area hardware to use different $\mathbb{F}_{(2^4)^2}$ construction for encryption and decryption; the polynomials were different for both levels of $\mathbb{F}_{(2^4)^2}$. Yu *et al.* [100] benchmarked the AES architectures of Moradi, Hamalainen [101], and Mathew on four different ASIC cell libraries. They found that Moradi’s and Hamalainen’s architectures were measurable smaller than Mathews’. Reyhani-Masoleh *et al.* conducted exhaustive search for suitable polynomials for the $\mathbb{F}_{((2^2)^2)^2}$ construction, with normal basis for the first level and redundant normal bases for the remaining two levels of the tower field [148]. They use different tower field constructions for encryption and decryption, but implement a unified datapath, with emphasis on resource sharing between the two functionalities. Furthermore, they derived expressions using XOR, AND, OR, NAND and NOR operations and utilized OR-AND-Invert gates available in the ASIC library used.

4.2 Cryptographic hardware and complexities

This brief section begins with a general overview of cryptographic hardware and complexities (Subsection 4.2.1). As finite field arithmetic is a broad area, it is important to outline different arithmetic algorithm designs (Subsection 4.2.2). The last subsection is presenting some new directions in early predictions on the complexity of the resulting hardware (Subsection 4.2.3). This section presents the design space exploration aimed at finite fields.

A small set of well known arithmetic algorithms is introduced in Chapter 5, and then used for datapath synthesis in Parts IV; design space exploration from the finite fields perspective and analysis of produced datapaths is presented in Parts V and VI.

4.2.1 General overview

The background in Section 3.3.1 gave an overview of the implementation technologies and different metrics used for evaluating implementation efficiency. For a long time, the cryptographic community used (only) theoretical complexities to compare different algorithms. Subsection 4.2.3, will introduce some new directions. The following is a direct summary of Section 16.1 in [15]. First, three ways of evaluating algorithms for hardware are identified: (i.) the algorithmic complexity (number of arithmetic operations over the underlying field), (ii.) the amount of storage (temporary storage and storage for pre-computed values), and (iii.) the number of memory accesses. Then the space and time complexity are singled out as important metrics. The *space complexity* of an architecture is the number of its logic gates and the amount of storage. The *critical path* is the longest delay caused by the logic gates on the critical path, and the *time complexity* the amount

of time needed by the architecture to complete the required arithmetic operation upon receiving any portion of the input. The time complexity depends on the type of architecture: for a fully bit-parallel architecture the time complexity is the critical path delay, in all other cases, the time complexity is approximated as a product of the number of clock cycles and the critical path delay. Gate counts are used by many authors, for example in [150, 131, 151], to name just a few.

4.2.2 Different architectures for finite field arithmetic

This section is classifying the algorithms based on their hardware architecture as serial or parallel. Classifications can be found in most texts, e.g., [19, 20, 42, 150, 152] just to list a few. For example, the interleaved multiplication for \mathbb{F}_{2^m} , a finite-field variant of shift-and-add algorithm, is a bit-serial algorithm. One factor is received serially, at each step there is a multiplication by x followed by the reduction using the irreducible polynomial¹, and the product accumulated over m clock cycles. Because the product is accumulated, meaning that all m clock cycles are needed for the final value of every product bit, this multiplier is also called serial-input,parallel-output multiplier (SIPO) [42, 153]. A well known example of a bit-parallel multiplier is the Mastrovito multiplier [152]. The Mastrovito matrix combines one factor and the irreducible polynomial, while the other factor is used as the vector in this matrix-vector multiplier; all bits of the product are computed concurrently. This is an example of a parallel-input/parallel-output (PIPO) arithmetic circuit.

Some architectures are closely linked to the representation of field elements. Examples of inherently serial multipliers are the Massey-Omura multiplier in its original form, using normal basis representation of elements [154]. It exploits ease of squaring in normal basis, a simple cyclic shift. A single bit of the product is computed using a function derived from the normal basis used, then both factors are shifted², and the same function is used to obtain the next bit of the product. This is an example of a parallel-input/serial-output multiplier (PISO) [42, 153, 155]. A serial multiplier can be parallelized, as was shown by Reyhani-Masoleh and Hasan in [156].

Due to Reyhani-Masoleh and Hasan [157], the fully bit-serial and fully bit-parallel designs represent two ends of the architectural spectrum, with digit-serial architectures³ in between. When only one word is used to describe the circuit, e.g., serial, this usually refers to the generation of the output. The serial multipliers are realized with sequential circuits. A classification in [158] mentions sequential multipliers with serial output (SMSO) and sequential multipliers with parallel output (SMPO). In [132], Paar and Soria-Rodriguez propose a hybrid tower field multiplier with parallel architecture for the subfield and a serial implementation for the top-level. Similar descriptions extend to other arithmetic circuits, e.g., for the finite field inversions, see [159].

To summarize the aforementioned classifications: based on the nature of receiving inputs and generating outputs, an arithmetic circuit can be classified as a

¹ an LFSR can be used to implement this part of the algorithm

² to perform either squaring or square-root, depends on high-to-low or low-to-high generation of the product bits, respectively

³beyond the scope of this work

- serial-input/serial-output (SISO)
- serial-input/parallel-output (SIPO)
- parallel-input/serial-output (PISO)
- parallel-input/parallel-output (PIPO)

The architecture type is closely related to the complexities: bit-parallel circuits are estimated only in terms of their combinational complexity, usually counting only AND and XOR gates and the critical path delays. For sequential circuits, flip-flop counts are added to the space complexity, and number of clock cycles to the time complexity.

4.2.3 The XOR counts

In the past few years, a lot of attention was directed towards XOR counts of matrices used for the matrix-vector multiplications, in order to evaluate their efficiency [161]. Recent work is focusing on identifying the discrepancies between theoretical evaluations of algorithms and possible optimizations, such as reuse and subexpression eliminations.

As explained in [160], maximum distance separable (MDS) codes are used to construct linear layers for ciphers, and instead of implementing the MDS directly, a matrix A , such that A^k is MDS for a small k , is implemented; entries of this matrix are constant elements. The multiplication by a constant $\alpha \in \mathbb{F}_{2^m}$ is implemented in hardware as a matrix-vector multiplier (MV). Beierle *et al.* search for optimal implementation of multiplication by a given constant (and use exhaustive search for optimal bases) and use them for round-based implementations of the MDS matrices [160]. The papers distinguish between two metrics for efficiency of the matrix. First is the *direct XOR count*, d-XOR-count, which is defined either as the number of XOR gates needed to implement the MV multiplier [161, 162], or as the Hamming weight of the matrix used in the MV multiplier [163]. Second is the *sequential XOR count*, s-XOR-count, which takes into account that the results of the previous steps can be reused, which reduces the number of XOR gates needed [160, 161, 164]. The s-XOR-count can be obtained using modified Gauss-Jordan elimination [161]. Equation (4.1) below shows an example used by Kölsch [161] to illustrate the difference between direct and sequential XOR count. The matrix on the l.h.s. in equation (4.1) shows the direct XOR count 6, while the r.h.s. shows only 3 XORs if the results in the brackets are reused; the sequential XOR count of the matrix is thus 3.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_1 + a_2 \\ (a_1 + a_2) + a_3 \\ ((a_1 + a_2) + a_3) + a_4 \end{bmatrix} \quad (4.1)$$

The general directions in this research area are in searching for best XOR count MV matrices, characterizing them in terms of irreducible polynomials and bases used, and then using them to construct the MDS matrices. For the MDS matrices, different constructions, e.g., circular, Hadamard,

cyclic, etc. [162, 165], as well as different implementations, e.g., round based, serialized, etc. [162, 164, 165, 166], are considered. Most literature relies on local optimizations. Kranz *et al.* applied global optimizations to previously locally optimized MDS matrices [163].

Jean *et al.* examined both linear and non-linear layers [164]. The authors propose an optimization tool called LIGHTER, which implements graph-based search algorithms minimizing the cost sum along a path. The authors report improved implementations of many Sboxes, further optimized existing linear layers, and new lighter diffusion matrices. For the non-linear layers, LIGHTER outperforms ABC, a state-of-the-art academic synthesis and verification tool [167]. Some limitations of LIGHTER are: it does not always guarantee optimal implementation, especially for non-linear layers, and is in general limited to \mathbb{F}_{2^4} because of memory and computational limitations, however the authors report LIGHTER can optimize “some” functions defined over bigger fields.

4.3 Hardware design automation and synthesis tools

To identify some already existing solutions, this section presents an overview of recent advances in HLS (Subsection 4.3.1), followed by a section on some academic high-level synthesis tools (Subsection 4.3.2). The related work is mostly focusing on design space exploration from the hardware perspective. Subsection 4.3.3 describes a high-level synthesis framework called CRYKET, a domain-specific language Cryptol, and an open source environment ATHENa, a framework that interacts with commercial synthesis tools. These tools differ in both their target domain and their approach. Subsection 4.3.4 presents some challenges associated with digital signal processing.

Some of the presented case studies are using AES, which is defined over a finite field and often implemented using tower fields. However, no specific information about the bases used was given, and presumably the standard specification was used. In 4.3.4 the authors presented the problem of representation of data types for datapath synthesis. A similar problem, but with a different data type, was encountered and solved in this thesis. Unlike presented HLSs, the presented framework does not use a library of components, but selected arithmetic algorithms to generate the required components on the fly. Instead of a high-level language, a domain specific language or a graphical user interface, the starting point for the framework in this thesis is an open-source computer algebra system GAP.

The distinguishing characteristics of this thesis in the context of related work in synthesis and design automation for cryptography are the support for tower fields, design-space exploration of finite field parameters, early estimation of area and performance based on these parameters, and automated synthesis of arbitrary finite field expressions over arbitrary finite fields, without relying on a database of building blocks. In particular, this thesis is the first work to provide support for tower fields, not limited to a specific application, such as AES.

4.3.1 High-Level synthesis

The 2009 Introduction to High-Level Synthesis in [76], published 15 years after the first publication in the same journal [75], gives an overview of the tasks of HLS tools, briefly touches on synthesis flows and gives a case study of two industrial HLS tools, Catapult and Cynthesizer. They mention examples where the behavioural description is entered using for example C, C++ (Mentor Catapult), SystemC (Cadence C-to-Silicon) or MATLAB (Xilinx AccelDSP) and Simulink (Synopsys Simplify-DSP). In conclusion, the authors note that many features still need to be added and that as the scope of HLS tools is growing from block-level to subsystem to full-system design, new challenges and opportunities arise as the current HLS tools still need plenty of designers guidance.

The 2016 paper by Nane *et al.* [78] is a comprehensive survey, categorization and evaluation of FPGA HLS tools. They include an overview of optimizations on which an individual tool is focusing and an experimental evaluation of academic HLSs bambu [168], DWARV [169] and LegUp [170] in comparison with commercial tools with a conclusion that the quality difference between academic and commercial tools is not drastic and that, including commercial tools, no single tool produced the best result for all benchmarks used. However, the commercial tools support more features. In the conclusion, they also state the following: “However, software engineers need to take into account that optimizations that are necessary to realize high performance hardware (e.g., enabling loop pipelining and removing control flow) differ significantly from software-oriented ones (e.g., data reorganization for cache locality).”

Their work starts with a classification of HLS tools based on the design input language with two major categories, design specific languages (further classified as new languages invented for the tool flow and extensions of generic languages) and generic languages (further classified as procedural and object-oriented). They also separate HLS tools that were adapted for a specific application domain and thus use dedicated optimizations and solutions from the general tools with complete support for standard high-level languages, such as C. The target applications were classified as: all domains, imaging, streaming, stream/image, loop/pipeline, DSP, etc. Further, they categorize the tools as in use, abandoned and N/A, if not conclusive. For example, SPARK and AccelDSP, mentioned in the 2009 HLS introduction [76], were both classified as abandoned. They also identify the HLS tool objectives, e.g., speculation and code motion, exploiting spatial parallelism, loop optimizations etc. In the evaluation, different benchmarks were used and two sets of experiments performed: standard, using default settings, and performance-optimized, using compiler flags and code annotations. Performance metrics used to evaluate the generated circuits were number of cycles, maximum frequency after PAR, wall-clock time and area metrics.

4.3.2 Academic High-Level synthesis

GAUT: is an HLS tool that takes C/C++ input and generates corresponding RTL hardware modules (in VHDL) [171]. It is targeting digital signal processing (DSP) applications. GAUT generates hardware modules, composed of three units: a processing unit (PU), memory unit (MEMU), and a communication and interface unit (COMU). The PU includes the datapath and the FSM. The

synthesis flow includes compilation into a DFG, clustering, and three synthesis flows, one for each unit. It selects arithmetic operators from a component library, and performs allocation, scheduling and binding, followed by resizing and optimizations to meet the user provided constraints, such as throughput and clock period. Some features of GAUT are bitwidth-aware design flow (to optimize the hardware for area and power consumption [172]), joint scheduling algorithm (to optimize multimode architectures designed for time-wise mutually exclusive applications [173]).

LegUp: is an open-source HLS framework that takes C input and generates a hybrid architecture with an FPGA-based MIPS soft core and custom hardware accelerators (in Verilog) [170]. The LegUp uses hardware profiling to decide which functions to implement on a hardware accelerator and which on MIPS, and a C to assembly compiler LLVM to precharacterize each hardware operation, which allows early speed and area predictions.

bambu: is semi-automatic framework to assist during HLS, which allows to modify the order of HLS steps and allows to choose different algorithms to perform these steps [168]. It takes input in C and starts by constructing a call graph from the syntax tree. It is targeting memory-intensive applications, and contains several mechanisms to compute memory addresses and perform memory allocation. Then a HLS is used to produce the datapath, FSM and memory modules, and bambu assembles the modules into a netlist. It integrates commercial synthesis tools (e.g., Xilinx ISE, Synopsys Design Compiler) and supports both FPGA and ASIC. The testbenches are generated automatically from the initial C specifications.

4.3.3 High-Level synthesis and design automation related to cryptography

CRYKET: is a high-level synthesis framework, specific to symmetric key cryptography kernels. It integrates RunFein for block cipher support [175], and RunStream for stream cipher support [176]. Both use a graphical user interface (GUI), which allows a selection and parameter turning for a number of cryptographic kernels, e.g., block size, number of rounds, etc. Furthermore, the tool offers different hardware design choices, e.g., specifying the architecture in terms of parallelism of encryptions/decryptions, rounds, and bits. The hardware components are obtained from a comprehensive library of known components, and steps like binding and resource allocation must be performed. Finally, synthesizable Verilog is produced. The testbenches and a software implementation are generated as well. The HLS tool is very domain specific; RunFein offers a selection of Sboxes, permutations, finite field multipliers, etc., and RunStream offers a selection nodes, such as FSRs, MUXes, Boolean logic, etc. including finite field multipliers. These nodes can be parametrized by size, type, clocking. Both tools generate the datapath and control. A key challenge in the design of both tools was identifying a complete set of cryptographic kernels. The CRYKET implementations were extensively compared to manual RTL design and achieved very good results, e.g., rought a 5% area overhead for the RunStream implementations.

Cryptol: is a domain-specific language (DSL) for cryptography, resembling mathematical specification more closely than general purpose languages [177, 178]. It can generate a software implementation, a hardware implementation, and verification models. Its hardware design flow begins

with translation to in intermediate representation (IR), which is then compiled to low level signal processing intermediate representation (LLSPIR), which allows timing transformations to optimize the circuit. This stage provides profiling information such as longest path, latency, size, etc. and supports equivalence checking. LLSPIR is translated to VHDL. This is the entry point to several flows targeting FPGA devices, grouped together under the name “FPGA”. The designer can generate a formal model of a function which is in symbolic, LLSPIR or VHDL mode. Cryptol can model combinational and sequential circuits are supported, and with use of two programs, *par* and *seq*, it controls space-time tradeoffs. By default, the compiler will unroll and parallelize the sequence as much as possible. It also offers pipelining.

ATHENa: is an open source environment for fair, comprehensive, automated, and collaborative hardware benchmarking [179]. The acronym ATHENa stands for “Automated Tool for Hardware Evaluation”. Gaj *et al.* specified the goal of the project is to develop a methodology and an environment, that would allow for comprehensive, fair, reliable and practical software and hardware performance comparison among various: algorithms (candidates in cryptographic competitions, e.g., CAESAR), implementation methods (iterative, unrolled, pipelined, precomputation, table look-up, etc.), platforms (Xilinx vs. Altera FPGAs), and languages and tools (VHDL vs. Verilog, different versions of Xilinx Vivado) [179, 181]. Refer to [180] for more information on the ATHENa project, e.g., the database with FPGA (and ASIC) results candidates in cryptographic competitions, such as CAESAR, and new work, e.g., adaptations for the LWC competition.

In its initial form [179], ATHENa required a Perl interpreter, access to FPGA design environments (e.g., Xilinx ISE), a reference implementation in C, testvectors, and HDL code. Its main features include: (i.) running all steps of synthesis, implementation and timing analysis in batch mode, (ii.) support for devices and tools of two major FPGA vendors: Xilinx and Altera, (iii.) Generation of results for multiple FPGA families of a given vendor, (iv.) automated choice of a device within a given family of FPGAs assuming that the resource utilization does not exceed a certain limit, and (v.) automated optimization of results aimed at one of the three optimization criteria: speed, area, and ration speed to area. In 2017 Minerva, an automated hardware optimization tool, was proposed [182]. It finds the best target frequency and determines close to optimal settings for the tools (Xilinx Vivado Design Suite 2015.1) to achieve optimal performance (measured in throughput or in throughput-to-area ratio) for a large number of RTL designs. The potential use of HLS tools for benchmarking purposes was investigated in [183]. The findings report strong, but not ideal, correlation between RTL-based and HLS-based ranking of Round 3 CAESAR candidates in terms of the aforementioned performance metrics, suggest that HLS methodology could be beneficial in the early stages of the competitions, but do not advocate replacing RTL implementations with the HLS generated code.

4.3.4 Compiling MATLAB onto FPGAs

The use of MATLAB and HDL coder is widely spread in the DSP domain (Subsection 3.1.4). This paragraph describes MATCH, a predecessor of AccelFPGA [184]. In earlier work, the fact that MATLAB has no notion of type of its variables. The first step in the design flow is transformation

to an abstract syntax tree (AST), followed by a type-shape interface phase. The variables are analyzed and if no directive is provided, their type/shape is inferred: exact data type can be an integer, a floating point, a complex number etc., while the shape examples include dimensions of a matrix etc.. A good overview of the compiler is provided in [?], and it highlights some DSP specific steps, e.g., auto-quantization (floating point to fixed-point), and streaming (for data with a regular rate of flow).

Part III

The architectural decisions phase

Part III - Outline

5	The architectural decisions phase - finite field arithmetic perspective	74
6	FSR package: feedback shift registers	80
7	FFCSA package: finite field construction, search and algorithms	93
8	Case study: WG and WAGE	112

Chapter 5

The architectural decisions phase - finite field arithmetic perspective

5.1	Overview	74
5.2	Roadmap	76
5.3	The coarse architectural decisions	78

5.1 Overview

Figure 5.1 shows the current phase within the design flow from Figure 1.1, namely, *the architectural decisions* shown in a magnifying glass on the right. This part of the thesis discusses the pure GAP packages developed for *the architectural decisions* to support the finite field arithmetic perspective (Figure 5.2(a)). The simplified design flow diagram in the leftmost column of Figure 5.2(a) and Figure 5.2(b) will be used throughout this part of the thesis for orientation, with grey areas indicating the package under discussion. At the beginning of each chapter, a short summary is presented in a frame.

The mathematical aspect of the automation framework consists of two core packages: the feedback shift registers package FSR, and the finite field construction, search and algorithms package FFCSA. They are shown on top of Figure 5.2(b). The FSR package evolved to be standalone and is presented first. This chapter is concluded with two case studies: WG and WAGE. Chapter 20 describes the design space exploration of WG stream cipher in more detail, and Part VI describes WAGE in more detail, including the datapath synthesis. The arrows in Figure 5.2(b) indicate package dependencies.

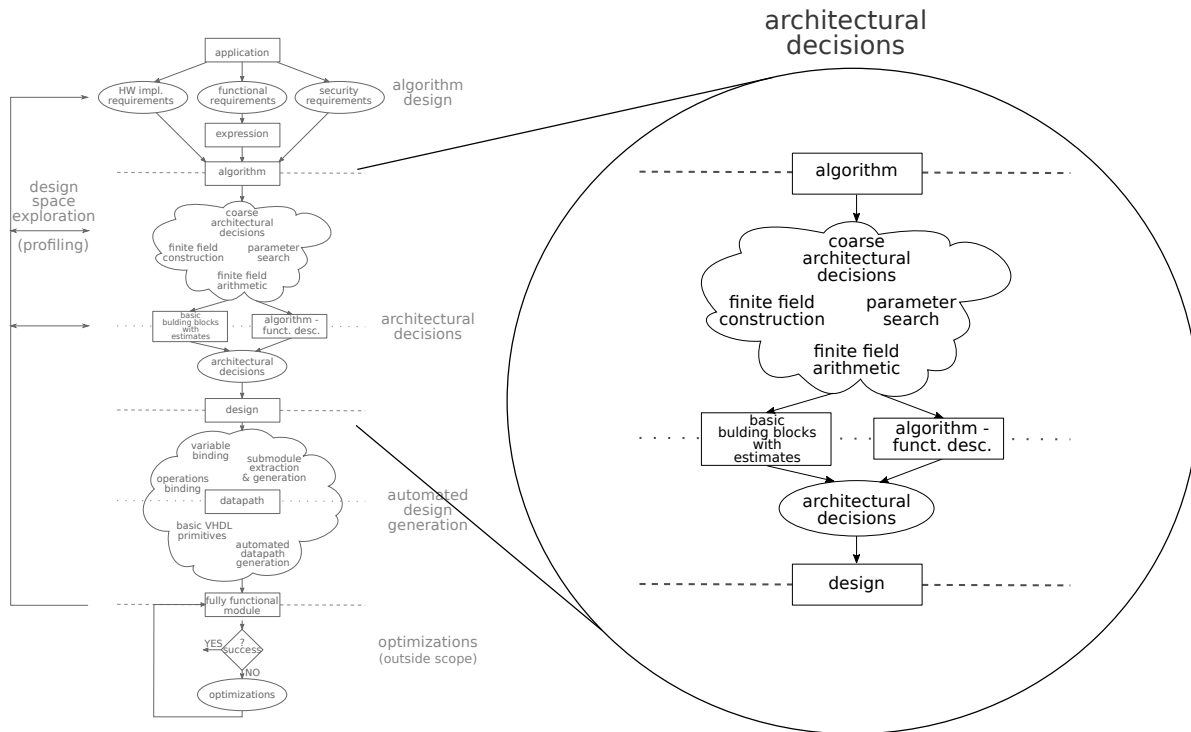
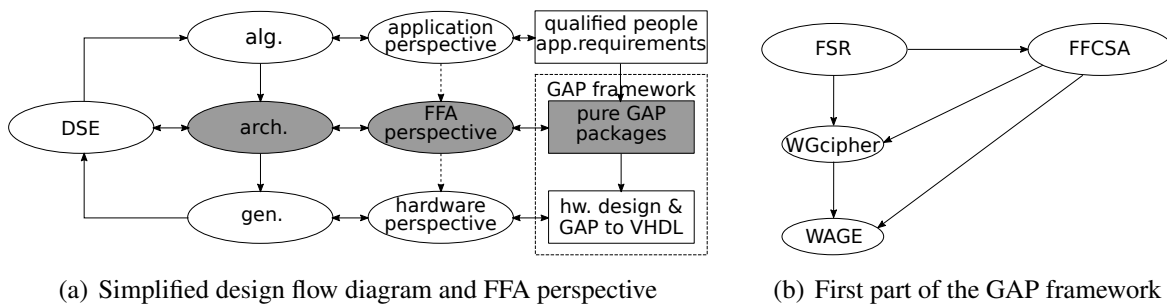


Figure 5.1: Design flow: *the architectural decisions*



(a) Simplified design flow diagram and FFA perspective

(b) First part of the GAP framework

Figure 5.2: *The architectural decisions*: (a) the simplified design flow diagram, and (b) the packages for the first part of the GAP framework.

5.2 Roadmap

The structure of Part III is as follows:

- The coarse architectural decisions - this chapter, Section 5.3
- FSR package: feedback shift registers - Chapter 6
 - Main functionality and structure of the FSR package (Section 6.1)
 - Examples for the FSR package (Section 6.2)
 - Summary of key insights (Section 6.3)
- FFCSA package: finite field construction, search and algorithms - Chapter 7
 - Main functionality of the FFCSA package (Section 7.1)
 - FFCSA profiling methods (Section 7.2)
 - Tower field bases (Section 7.3)
 - Algorithms: obtaining expressions for finite field arithmetic (Section 7.4)
 - Summary of key insights (Section 7.5)
- Case study: WG and WAGE - Chapter 8
 - Case study: the WGcipher package (Section 8.1)
 - Case study: the WAGE package (Section 8.2)

Table 5.1 lists all examples in Part III, examples moved to appendix, and related examples in Part IV. The table rows correspond to individual examples, grouped by the chapters. The columns are structured as follows: the first column gives the subsection in which the example can be found, the second column “Ex.” the example number, the third column the example title and short description if needed, the next two columns indicate whether the example includes GAP or VHDL code, and the last column specifies related examples (Related Ex.). The related examples are the continuation of the example in this row.

Section	Ex.	Title and keywords	GAP	VHDL	Related Ex.
FSR: feedback shift registers - Chapter 6					
Section 6.2	6.2.1	Miscellaneous methods for multivariate polynomials defined over a field	✓		
	6.2.2	A simple LFSR over an extension field: period, regular run and external run	✓		C.1.1 11.1.1, 11.2.1
	6.2.3	A simple FILFUN over the binary field: <i>LoadStepFSR</i> method and regular run	✓		11.1.1
	6.2.4	A symbolic NLFSR over a binary field: load with symbolic initial state	✓		11.1.1
Appendix C.1	C.1.1	A simple LFSR over an extension field - continued: TEX writing and drawing	✓		
FFCSA: finite field construction, search and algorithms - Chapter 7					
Section 7.1	7.1.1	A small complexity normal basis: interplay of different parts of the FFCSA package			7.2.1
	7.1.2	A small complexity transition matrix: interplay of different parts of the FFCSA package			7.2.1
	7.1.3	A finite field multiplier: interplay of different parts of the FFCSA package			
Section 7.2 profiling	7.2.1	FFCSA profiling examples for \mathbb{F}_{2^7}	✓		
Section 7.3.1 TF bases	7.3.1	$\mathbb{F}_{((2^2)^2)^2}$ tower field basis using a polynomial basis for each level: polynomial search and basis	✓		7.4.2
Section 7.4 Algorithms and expressions	7.4.1	Multiplication expressions for \mathbb{F}_{2^4}: Generalized alg., two-step classic alg.	✓		
	7.4.2	Multiplication expressions for $\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$ and \mathbb{F}_{2^8}: Generalized alg., two-step classic alg.	✓		
Case study: WG cipher and WAGE - Chapter 8					
Section 8.1 WG cipher	8.1.1	The WG7 keystream generator: loading, initialization, and a short keystream	✓		11.2.5

Table 5.1: Part III examples

5.3 The coarse architectural decisions

This short introductory discussion relies on the datapath classifications, presented in Subsection 3.3.3, and on the discussion about the architectures used for finite field arithmetic, presented in Subsection 4.2.2.

Hardware designs can be roughly divided into three classes based on the algorithm they implement: datapath, control, and memory. Finite field applications certainly fall into the datapath class. There is usually some control needed and sometimes storage is used. The mathematical expressions are usually implemented using *discrete components* and compositions of their outputs. The discrete components, i.e., basic building blocks, are modules implementing the arithmetic operations using logic gates. Their architecture depends on the algorithm and basis, e.g., the school-book two-step classic multiplication in polynomial basis.

For small fields, *constant array* implementations are possible. The values of the expression are precomputed for all possible inputs and stored as constants. The constant array is a lookup table design: the input serves as an index to the array, selecting the appropriate precomputed constant. There are slight variations in the hardware area of the array w.r.t. the basis used. The array can be built for the entire expression, a part of the expression, or for certain operations only. This can be considered as granularity, and the deciding factor is the constant array size and thus the implementation area. In the latter two cases, the final output must still be computed.

Naturally, hardware designers try to exploit as much parallelism as possible. However, constraints such as a desired small area can force the designer to explore other options, e.g. a sequential design with small combinational circuitry (this is a form of component reuse or resource sharing, see Subsection 3.3.3). There are other techniques to address other constraints, like pipelining to increase the throughput. Sometimes hybrid options must be considered. Furthermore, both the constant array and component based submodules can be used in any type of datapath, e.g., parallel, sequential, or even pipelined.

Algorithms that compute the output in a serial fashion are well suited for sequential designs, and in finite field arithmetic, certain choices of basis will yield such an algorithms, e.g., normal bases. One such example is the Massey-Omura multiplier, which holds the two factors in shift registers and produces one bit of product per clock cycle. This multiplier is an example of a parallel-input/serial-output module (PISO). On other occasions, where the throughput is more important than a small area, the designer will prefer algorithms that compute all bits of the output in parallel. The school-book example of a parallel-input/parallel-output (PIPO) multiplier is the two-step classic multiplier. An overview of different types of algorithms based on serial and parallel computation was presented in Subsection 4.2.2.

The decisions described in the previous paragraphs, are called “coarse architectural decisions” to differentiate them from the very specific decisions. An example of a specific or finalized decision is the choice of a particular irreducible polynomial from the list of candidates, i.e., a list of all polynomials of given degree that are irreducible over a given field. All design decisions must be finalized before the design entry. In Figure 5.1 “coarse architectural decisions” appear at the top

and the finalized “architectural decisions” in the oval shape at the bottom. Coarse architectural decisions are a subject of design space exploration from the hardware perspective. Focusing on constrained environments and lightweight cryptographic applications, the attention of this thesis is placed on unpipelined, fully parallel designs that do not take advantage of serial computation (Subsection 3.3.3). Such designs use only PIPO type basic building blocks (Subsection 4.2.2). However, the automation framework can produce expressions for serial basic building blocks, e.g., PISO type Massey-Omura multiplication. A PISO building block datapath is fully sequential (Subsection 3.3.3). Serial computation always requires additional control logic, which increases the hardware area beyond a reasonable tradeoff for lightweight cryptography.

Design space exploration from a mathematical perspective focuses on the hardware implementation results of modules obtained for the same functionality but with variation in one of the parameters. For example, using different algorithms for the underlying arithmetic operations, or using different field parameters for deriving the expressions for a given algorithm. This part is covered with parameter search, finite field constructions, and finite field arithmetic in Figure 5.1. It is realized with the package FFCSA, shown in Figure 5.2(b). Based on a metric of choice, e.g., hardware area, the architectural decisions are finalized.

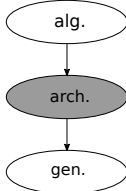
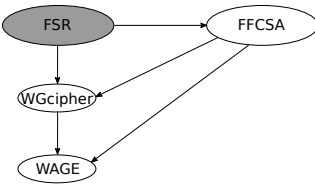
This discussion covered two categories: hardware design and finite field arithmetic, reflecting the “math meets hardware” motif from the perspective of coarse architectural decisions. Coarse architectural decisions are not automated, i.e., the initial choices are left to the user, but they can be easily revised after the results of the design space exploration.

Chapter 6

FSR package: feedback shift registers

6.1	Main functionality and structure of the FSR package	80
6.2	Examples for the FSR package	87
6.3	Summary of key insights	92

Feedback shift registers were introduced in Subsection 3.2.1. This section describes the implemented FSR package, which allows creation, initialization, and running of LFSRs, NLFSRs, and FILFUN (filtering function) objects. The FSR package also computes some of these objects' properties, e.g., the internal state size or period for LFSRs. As such, it is one of the core packages of the automation framework and enables the generation of test vectors for the (hardware) implementations.

	<p>Overview of the FSR package:</p> <p>The FSR package allows creation, initialization, and running of the LFSR, NLFSR, and FILFUN objects. The FSR package and its sister package, FSRtoVHDL, constitute an automation toolkit that significantly reduces the amount of human effort for both software implementations using GAP and hardware design generation in VHDL. A great advantage of the FSR package is its use of a regular and an external step. A stand-alone simple (N)LFSR object is self-contained: it is updated by the computed feedback value. This is called the regular step and run. The external step and run allow arbitrary filters (FILFUNs) to be added to the feedback of any FSR object. This provides the flexibility needed to represent a cipher as a collection of basic modules. The FSR package has the ability to load and run a symbolic initial state. The creation and running of any FSR requires only a few lines of GAP code and the outputs appear in the GAP prompt. The package also contains some miscellaneous methods, used throughout the GAP framework, and formatting functions for outputs to *.txt and *.tex files. To the author's knowledge, the FSR package is the first to implement both LFSRs and NLFSRs over prime and extension fields.</p>	
--	---	--

The FSR package is available at <https://nzidaric.github.io/fsr/>

6.1 Main functionality and structure of the FSR package

The FSR package and its sister package, FSRtoVHDL, constitute an independent part of the automation toolkit. The FSRtoVHDL package is used to generate the VHDL for an arbitrary LFSR,

NLFSR, or FILFUN (filtering function), or a cipher specified as a collection of FSR objects. FSR-toVHDL will be explained in detail in Chapter 11, however, the hardware implementation aspect was, in great part, guiding the development of the FSR package itself [36]:

1. Hardware-style thinking involves a modular approach: a cipher can be implemented as a collection of basic modules, which are identified as LFSR, NLFSR and FILFUN.
 - (a) For most (N)LFSRs used in practice, the feedback function can be modelled by a polynomial
 - (b) Arbitrarily complicated NLFSRs can be implemented by connecting an (N)LFSR with one or more FILFUNs.
 - (c) Complex FILFUNs can be implemented by connecting multiple simple FILFUNs
2. Recognizing and exploiting structural similarities between LFSRs, NLFSRs and FILFUNs, from both the mathematical and the hardware perspective, reduces the number of implemented objects, functions, and methods (Table 6.1).
3. Highly structural FSR package design is mandatory for automated VHDL generation: the FSR objects must store sufficient information for the implementation.

Key 6.1: A cipher as a collection of basic modules

A cipher can be represented as a collection of basic modules, LFSRs, NLFSRs, and FILFUNs, connected in various configurations.

Structural similarities from the mathematical point of view (item 2) are explained in Table 6.1. Item 3 is related to the transition from GAP objects to VHDL code, which was one of the biggest challenges in developing the toolkit and will be addressed in Chapter 11. It is partially addressed by the good design of FSR objects: all the information for VHDL implementation of a particular FSR is captured by the object as attributes. For example, the underlying finite field defines which VHDL data type to use for the signals in the hardware module.

Structural similarities between the three FSR objects are summarized in Table 6.1. LFSR and NLFSR GAP objects differ only in the degree of the multivariate polynomial used to define their feedback. A FILFUN is an FSR object without feedback, shifting, or storing, and its functionality is defined by a multivariate polynomial. The justification for such a design decision is twofold: (i.) filtering functions are similar to NLFSR feedback functions, and (ii.) FSRs with output filters are common, indicating that they will be used together. The differences between the NLFSRs and FILFUNs will be discussed shortly.

FSR object name	multivariate polynomial $f(x)$		FSR (feedback, memory)	output	
	linear	nonlinear		one or more state elm.	computed value of $f(x)$
LFSR	✓		✓	✓	
NLFSR		✓	✓	✓	
FILFUN	✓	✓			✓

Table 6.1: Structural similarities between LFSR, NLFSR, and FILFUN objects

Key 6.2: Structural similarities between FSR objects

The LFSR and NLFSR GAP objects differ only in the degree of the multivariate polynomial used to define their feedback. A FILFUN is an FSR object without feedback, shifting, or storing, and its functionality is defined by a multivariate polynomial. The justification for such a design decision is twofold: (i.) filtering functions are similar to (NLFSR) feedback functions, and (ii.) FSRs with output filters are common, hence they will be used together.

FSR objects can be created either as LFSRs, NLFSRs or FILFUNs (Table 6.2). They are created through a function call with various possibilities for input arguments, and will return an object with four components, and some case-specific attributes and properties¹. Only the values that can change during the FSR's lifetime are implemented as components:

- `init` - initial state of the FSR
- `state` - the current state of the FSR
- `numsteps` - number of steps since the object was created
- `basis` - the basis used for the representation of field elements

The components `init` and `state` are empty when the FSR is created. When the FSR is loaded, both components are updated with the initial state. The component `state` is updated with each FSR step, but `init` keeps the initial state until loaded anew. The integer `numsteps` keeps track of the number of steps performed for the FSR. The `numsteps` value is set to -1 when the FSR is created, to 0 when loaded, and then increments with each step. It is used for coding purposes to prevent an attempt of running an empty FSR and to stop the FSR once a certain threshold is reached, preventing it from looping indefinitely. The `basis` holds the current basis used for the representation of elements and can be changed when needed. The constructors for the FSR objects are listed in Table 6.2.

The behaviour of all three FSR objects is similar and captured with the following four methods, also listed in Table 6.3:

- `LoadFSR` - load the initial state into `init` and `state`, set `numsteps=0`

¹recall: GAP objects store information as attributes, properties, and components

name	mandatory arguments	optional arguments
LFSR	$\mathbb{F}_q, h(y)$ from eq. (3.13)	basis, (d_0, \dots, d_{t-1})
NLFSR	$\mathbb{F}_q, f(x_0, \dots, x_{n-1})$ from eq. (3.9), n	basis, (d_0, \dots, d_{t-1})
FILFUN	$\mathbb{F}_q, f(x_0, \dots, x_{t-1})$ from eq. (3.9)	basis
	$h(y) : \mathbb{F}_q \rightarrow \mathbb{F}_q$	(d_0, \dots, d_{t-1}) - output taps
	$f(x_0, \dots, x_{j-1}) : \mathbb{F}_q^j \rightarrow \mathbb{F}_q$, where $j = t, n$	n - length (number of stages)

Table 6.2: Constructors for the FSR objects

- **StepFSR** - shift stages S_{n-1}, \dots, S_1 to the right, compute the feedback value and use it to update S_{n-1} , increment `numsteps`, output the new sequence element
- **LoadStepFSR** - call **LoadFSR**, followed by a single **StepFSR** (intended for the FILFUNs, but works for (N)LFSR as well)
- **RunFSR** - perform a sequence of **StepFSR** calls

While LFSR and NLFSR differ only in feedback, the filters are a bit of an exception. However, the evaluation of NLFSR feedback and FILFUN output are computed in the same manner, and the same methods mentioned above are used for FILFUNs as well. Since a filter alone does not require any feedback, shifting or stages, i.e. hardware registers, the component state is used to hold the current values needed to evaluate the filtering function. The component state is not updated, but rather loaded anew with each step: the method **RunFSR** takes a list of “initial” states as input as shown in Example 6.2.3, then calls **LoadStepFSR** for each list entry.

The main functionality for an FSR object is summarized in Table 6.3. Table 6.3 differentiates between “a regular” and “an external” step and run. A stand-alone simple (N)LFSR object is self-contained: it is updated by the computed feedback value (regular step and run). Examples 6.2.2(c) and 6.2.3 show the regular run. The external **StepFSR** allows arbitrary filters to be added to the feedback of any (N)LFSR, e.g., to mask the output of the filtering function. The external step and run (Example 6.2.2(d)) allows the FSRs to be used directly as building blocks of many ciphers.

Key 6.3: Regular and external step and run

A stand-alone simple (N)LFSR object is self-contained: it is updated by the computed feedback value (regular step and run). The external step and run allow arbitrary filters (FILFUNs) to be added to the feedback of any FSR object. The external step and run allow for flexibility when representing a cipher as a collection of basic modules.

Further details, such as the attributes stored for each FSR object, are listed in Table 6.4. The columns correspond to the four main source files, and the rows are organized as filters and attributes (Subsection 3.1.5). The source *fsr* captures the common functionality for LFSRs, NLFSRs, and FILFUNs, including the methods listed in Table 6.3. The contents of the column *fsr* apply to

FSR - the FSR object *ext* - finite field element *extlist* - list of *ext*
ist - initial state *[]* - optional parameters *n* - integer

All FSR types: LFSR, NLFSR, FILFUN		
method name †	options	comments
LoadFSR (<i>FSR</i> , <i>ist</i>)	NA	load the initial state
StepFSR (<i>FSR</i> [, <i>ext</i>])	regular step	FSR self-contained: compute value x
	external step	adds an external elem. to the computed value: $x + ext$
	compute the feedback/function value x , use x or $x + ext$ to:	
	◦ update S_{n-1} after shifting stages S_{n-1}, \dots, S_1 for (N)LFSR	
	◦ output as the new element in case of FILFUN	
LoadStepFSR (<i>FSR</i> , <i>ist</i> [, <i>ext</i>])	regular step	FSR self-contained: compute value x
	external step	adds an external elem. to the computed value: $x + ext$
	combines methods LoadFSR, StepFSR to load the new values for variables before evaluating the function	
	this method is used by RunFSR for FILFUN objects	
RunFSR (<i>FSR</i> [, <i>n</i>]) (<i>FSR</i> , <i>ist</i> [, <i>n</i>])	regular run	with regular step
	external run	with external step
(<i>FSR</i> , <i>ist</i> [, <i>extlist</i>])	optional LoadFSR followed by sequence of StepFSR calls	

† - some (but not all) example arguments are shown in the brackets

Table 6.3: Main functionality of the FSR package

source	<i>fsr</i>	<i>lfsr</i>	<i>nlfsr</i>	<i>filfun</i>
filters	IsFSR	IsLFSR	IsNLFSR	IsFILFUN
attributes	FieldPoly UnderlyingField FeedbackVec OutputTap Length InternalStateSize Threshold	FeedbackPoly PeriodOfLFSR	MultivarPoly MonomialList IndetList	MultivarPoly MonomialList IndetList

Table 6.4: Main functionality of the FSR package - continued

all three FSR objects, while the other three columns show object-specific attributes and properties. The names used for filters, attributes, properties and methods are mostly self explanatory. The only exception is the `OutputTap` attribute, storing the stage(s) that serve as the source for the output sequence. If not set, the output stage is automatically set to S_0 . For hardware implementations, S_0 is not necessarily the best or only option. To save clock cycles during initialization phases of stream ciphers (Subsection 3.2.2), the desired output stage is S_{n-1} , where n is the length of the

(N)LFSR. To increase the throughput, multiple stages can be used to output a vector instead of a single element. Some of the attributes and properties from Table 6.4 are shown in Example 6.2.2.

■ **Implementation detail:** GAP allows hierarchy of filters, which are used in the GAP method selection mechanism. The three FSR objects have the following filters: `IsLFSR`, `IsNLFSR`, and `IsFILFUN`. Furthermore, they all belong to `IsFSR` (Table 6.4), which allows the methods `LoadFSR`, `StepFSR`, `LoadStepFSR`, and `RunFSR` (Table 6.3) to be implemented only once (in source file `fsr`). ■

■ **Implementation detail:** Some attributes listed in Table 6.4, e.g., `IndetList` for NLFSRs and FILFUNs, are stored merely to speed-up the computation. `IndetList` stores the indeterminates, i.e., variables, that were used to define the multivariate polynomial. The same information could easily be obtained from the `MultivarPoly` attribute itself. Similarly, the `FeedbackVec` attribute holds the coefficients of the feedback/filtering function. In case of LFSR, the new value is computed as a dot-product of `FeedbackVec` and the current state. For NLFSRs and FILFUNs, the value is computed by evaluating the `MultivarPoly` with current state values used for the corresponding indeterminates from the `IndetList`. The current state is kept by the component `state`. ■

F - finite field B - basis M - matrix
 f - polynomial ffe - finite field element vec - vector

<i>misc</i> †	<i>output</i>
<code>SplitCoeffsAndMonomials(F, f)</code>	<code>IntFFExt(B, ffe)</code>
<code>ReduceMonomialsOverField(F, f)</code>	<code>IntVecFFExt(B, vec)</code>
<code>DegreeOfPolynomialOverField(F, f)</code>	<code>IntMatFFExt(B, M)</code>
	<code>VecToString(B, vec) †</code>

† - these methods are used throughout other GAP packages in this work

Table 6.5: Miscellaneous and output formatting methods in the FSR package

The FSR package also includes miscellaneous methods for multivariate polynomials defined over a specific finite field (left column in Table 6.5). They are used by some FFCSA package methods (Chapter 7), and also play a very important role for automated design generation (Part IV). Below is a brief description of their functionality, also demonstrated in Example 6.2.1.

- `SplitCoeffsAndMonomials` takes a finite field \mathcal{F} with q elements and a multivariate polynomial f , and returns two lists, a list of coefficients and a list of monomials. Recall equation (3.9) from background Subsection 3.1.3, with modification to only non-zero coefficients $c_{i_0, i_1, \dots, i_{t-1}} \in \mathcal{F}$:

$$f(x_0, x_1, \dots, x_{t-1}) = \sum_{\substack{\forall (i_0, i_1, \dots, i_{t-1}) \in \mathbb{Z}_q^t \\ c_{i_0, i_1, \dots, i_{t-1}} \neq 0}} c_{i_0, i_1, \dots, i_{t-1}} x_0^{i_0} x_1^{i_1} \dots x_{t-1}^{i_{t-1}} \quad (6.1)$$

GAP uses algebraic normal form (ANF) [185] for expressions such as the r.h.s. in equation (6.1). Furthermore, the terms in equation (6.1) are ordered by the degree of monomials (recall equation (3.10)), and the variables within monomial by their number, not by their degree (see Section 66.17 in the GAP reference manual [34] for monomial orderings).

The ordering will be explained in more detail in Section 10.2. The two lists returned by `SplitCoeffsAndMonomials` are ordered the same way. The monomial corresponding to the constant term is 1, i.e., $x_0^0 x_1^0 \cdots x_{t-1}^0$, which assures that both lists are of the same length and that the input polynomial can be easily reconstructed by a dot-product of the two lists. The method also checks if all the coefficients really belong to the underlying finite field.

- `ReduceMonomialsOverField` takes a finite field \mathcal{F} with q elements and a multivariate polynomial f , as defined in equation (6.1), and returns a polynomial f_1 with the same coefficients but all exponents reduced modulo $q - 1$, i.e., $i_j \bmod (q - 1)$ for $0 \leq j < t$. This method is needed because in general, the coefficients and the exponents in equation (6.1) can be arbitrary. Restricting f to the finite field \mathcal{F} has two effects: (i.) all coefficients must belong to \mathcal{F} (i.e., $c_{i_0, i_1, \dots, i_{t-1}} \in \mathcal{F}$ as specified), and (ii.) due to the cyclic nature of finite fields, the exponents must be reduced accordingly.
- `DegreeOfPolynomialOverField` takes a finite field \mathcal{F} with q elements and a multivariate polynomial f , as defined in equation (6.1), calls `ReduceMonomialsOverField` to obtain f_1 , and returns the degree of the leading monomial in f_1 (recall equation (3.10)). By definition, the degree of the leading monomial is also the degree of the polynomial f_1 .

Key 6.4: Methods for multivariate polynomials

These methods take finite field \mathcal{F} with q elements and a multivariate polynomial f as an input. `SplitCoeffsAndMonomials` splits f into a list of coefficients and a list of monomials. `ReduceMonomialsOverField` returns a polynomial f_1 with the same coefficients as f , but all exponents reduced modulo $q - 1$. Method `DegreeOfPolynomialOverField` returns the degree of the polynomial f_1 , that was obtained with `ReduceMonomialsOverField`.

The FSR package contains output formatting functions (right column in Table 6.5), that produce both human friendly outputs and outputs for testbench generation. For example, the current state of the LFSR in Example 6.2.2, shown in the GAP code in Example 6.2.2(c, d), uses methods `IntFFExt (B, ffe)` and `IntVecFFExt (B, vec)`. The remaining FSR functions are TEX writing functions, that generate a *.tex* output. Examples showing TEX writing functions, such as `WriteTEXElementTableByGenerator`, `WriteTEXRunFSR` and `WriteTEXRunFSRByGenerator` can be found in Appendix C.1. The created (N)LFSR objects can be represented graphically using automatically generated *tikz* code. The LFSR from Example 6.2.2 is shown in Figure C.1.

6.2 Examples for the FSR package

Example 6.2.1 *Miscellaneous methods for multivariate polynomials defined over a field* \longleftrightarrow

The terms in an expression, such as the r.h.s. of equation (6.1), are ordered by the degree of monomials (recall equation (3.10)), and the variables within monomial by their number, not by their degree. This can be seen by comparing lines 2 and 3 in Example 6.2.1. Once the exponents in monomials are reduced, the order of terms changes, as can be seen by comparing f and $f1$. The actual degree of polynomial f , defined over \mathbb{F}_{2^4} , is 8, not 18, because $18 \equiv 3 \pmod{15}$, implying that the leading monomial is $x_0^2 x_1^5 x_4$, and not x_2^{18} .

Example 6.2.1

```

gap> K := GF(2);; x := X(K, "x");; F := FieldExtension(K, x^4 + x^3 + 1);;
gap> f := Z(2^4)^3*x_1^5*x_0^2*x_4 + x_0*x_2*x_3 + x_2^18 + Z(2^4);
x_2^18+Z(2^4)^3*x_0^2*x_1^5*x_4+x_0*x_2*x_3+Z(2^4)
gap> cmlist := SplitCoeffsAndMonomials(F, f);
[ [ Z(2)^0, Z(2^4)^3, Z(2)^0, Z(2^4) ], [ x_2^18, x_0^2*x_1^5*x_4, x_0*x_2*x_3, Z(2)^0
] ]
gap> cmlist[1] * cmlist[2];
x_2^18+Z(2^4)^3*x_0^2*x_1^5*x_4+x_0*x_2*x_3+Z(2^4)
gap> f1 := ReduceMonomialsOverField(F, f);
Z(2^4)^3*x_0^2*x_1^5*x_4+x_0*x_2*x_3+x_2^3+Z(2^4)
gap> DegreeOfPolynomialOverField(F, f1);
8
gap> SplitCoeffsAndMonomials(F, f1);
[ [ Z(2^4)^3, Z(2)^0, Z(2)^0, Z(2^4) ], [ x_0^2*x_1^5*x_4, x_0*x_2*x_3, x_2^3, Z(2)^0
] ]

```

\longleftarrow

Example 6.2.2 *A simple LFSR over an extension field* \longleftrightarrow

An LFSR can be uniquely described with a polynomial $\ell(y)$ of degree n over the field \mathbb{F}_{2^m} , also called the LFSR polynomial. It is well known that using a primitive LFSR polynomial guarantees a sequence of maximal length (m-sequence). In the following example an LFSR of degree 4 over \mathbb{F}_{2^4} is created, first using the polynomial $y^4 + y + \alpha$ and then $y^4 + y^3 + y + \alpha$, where $\alpha = \omega + \omega^2$ and ω is a root of the defining polynomial $f(x) = x^4 + x^3 + 1$ of \mathbb{F}_{2^4} . Please note that both α and ω are generators of \mathbb{F}_{2^4} . Table C.3 in Example C.1.1 in Appendix C.1 shows the elements in \mathbb{F}_{2^4} , represented in polynomial basis and as power of the generator α .

Example 6.2.2(a) shows the setup and creation of the LFSR test, and the attributes that were set at the time of the LFSR constructor call, followed by computation of the period and property check for maximum sequence LFSR.

Example 6.2.2(a)

```

gap> K := GF(2);; x := X(K, "x");; f := x^4 + x^3 + 1;;
gap> F := FieldExtension(K, f);; y := X(F, "y");; alpha := Z(2^4);;
gap> l := y^4 + y + alpha;; test := LFSR(K, f, l);
< empty LFSR given by FeedbackPoly = y^4+y+Z(2^4)>
gap> PrintAll(test);
empty LFSR over GF(2^4) given by FeedbackPoly = y^4+y+Z(2^4)
with basis = [ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6 ]
with feedback coeff = [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2^4) ]
with initial state = [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ]
with current state = [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ]
after initialization
with output from stage S_0
gap> KnownAttributesOfObject(test);
[ "LENGTH", "FieldPoly", "UnderlyingField", "FeedbackVec", "OutputTap", "FeedbackPoly"
]
gap> Length(test); FieldPoly(test); FeedbackVec(test); FeedbackPoly(test);
4
x^4+x^3+Z(2)^0
[ 0*Z(2), 0*Z(2), Z(2)^0, Z(2^4) ]
y^4+y+Z(2^4)
gap> PeriodOfLFSR(test); IsMaxSeqLFSR(test);
warning: the polynomial is reducible !!!
255
false

```

Example 6.2.2(a) used a reducible LFSR polynomial (see the warning displayed towards the end of example), namely $\ell(y) = y^4 + y + \alpha = (y^2 + y + \alpha^7)(y^2 + y + \alpha^9)$. As a result, this LFSR has a very short period, namely $(2^4)^2 - 1 = 255$. Whenever a reducible or an irreducible but not primitive LFSR polynomial is used, the sequence it produces belongs to a (short) cycle. Multiple cycles exist, and which cycle depends on the initial state. Any sequence obtained by the LFSR with reducible LFSR polynomial ℓ , can also be produced by a (combination of the) short LFSRs, obtained by the factors of ℓ . See example Example 6.2.2(b) for details. The first factor is primitive, the second factor is irreducible.

Example 6.2.2(b)

```

gap> factors := Factors(PolynomialRing(F), l);
[ y^2+y+Z(2^4)^7, y^2+y+Z(2^4)^9 ]
gap> test1 := LFSR(F, factors[1]);; test2 := LFSR(F, factors[2]);;
gap> PeriodOfLFSR(test1); PeriodOfLFSR(test2);
255
warning: the polynomial is irreducible !!!
85
gap> IsMaxSeqLFSR(test1);
true

```

Using a primitive LFSR polynomial of degree 4, maximum period is $(2^4)^4 - 1 = 35535$ and the LFSR will produce an m-sequence. Example 6.2.2(c) shows one such polynomial, namely $\ell(y) = y^4 + y^3 + y + \alpha$. The new LFSR will be used for the remainder of this example. The example is continued in Appendix C.1 in Example C.1.1, and the schematic of the LFSR is shown in Figure C.1.

Example 6.2.2(c)

```
gap> l := y^4 + y^3 + y + alpha;; test := LFSR(K, f, l);
< empty LFSR over GF(2^4) given by FeedbackPoly = y^4+y^3+y+Z(2^4) >
gap> PeriodOfLFSR(test); IsMaxSeqLFSR(test);
65535
true
```

Example 6.2.2(d) shows a regular run with loading of initial state `ist`, without extra inputs: the internal state and outputs are shown on each step. It is followed by a second run for 5 steps, without outputs on each step. The elements are represented w.r.t. canonical basis B for the run with outputs, and the sequences in GAP native representation. Finally the LFSR is re-loaded and the run repeated for 10 steps. Note that the sequence produced is has 11 elements: the output of stage S_0 before the first step is the first element of the resulting sequence.

Example 6.2.2(e) shows an external run with additional vector of field elements `elmvec`. At each step, new element from the vector (shown in column `elm` of the output) is added to the feedback value, as can be seen when comparing the elements of the internal state to the ones shown in Example 6.2.2(d). Sequences `seq` (Example 6.2.2(d)) and `seq1` (Example 6.2.2(e)) begin to differ as soon as the initial state is shifted out of the LFSR. Note that in Example 6.2.2(d,e) the `RunFSR` with argument `ist` will call `LoadFSR` first. The `RunFSR` calls without `ist` continue the run from the current state.



Example 6.2.3 *A simple FILFUN over the binary field* \longleftrightarrow

The following example shows a FILFUN over \mathbb{F}_2 . First, the `LoadStepFSR` call is shown, with values for each of the indeterminates in multivariate polynomial f , followed by a regular run with a sequence of inputs `inputsequence`. The `RunFSR` calls `LoadStepFSR` for each tuple in `inputsequence`.

Example 6.2.3

```
gap> K := GF(2);; x := X(K, "x");; f := x_0*x_1+x_2;;
gap> test := FILFUN(K, f);
< FILFUN of length 3 over GF(2),
  with the MultivarPoly = x_0*x_1+x_2>
gap> LoadStepFSR(test, [Z(2)^0, Z(2)^0, 0*Z(2)]);
Z(2)^0
gap> inputsequence := [[Z(2)^0, Z(2)^0, 0*Z(2)], [Z(2)^0, Z(2)^0, Z(2)^0], [0*Z(2),
  Z(2)^0, 0*Z(2)]];;
gap> outputsequence := RunFSR(test, inputsequence);
[ Z(2)^0, 0*Z(2), 0*Z(2) ]
```



Example 6.2.2(d)

```

gap> gap> ist := [0*Z(2), Z(2^4), Z(2^4)^5, Z(2)^0 ]; RunFSR(test, ist, 5, true);
using basis B := [ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6 ]
elm
[ 3, ... ...,0 ] with taps [ 0 ]
[ [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ], [ 1, 1, 0, 1 ], [ 1, 0, 0, 0 ] ] [ 1, 0, 0, 0 ]
[ [ 1, 0, 1, 1 ], [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ], [ 1, 1, 0, 1 ] ] [ 1, 1, 0, 1 ]
[ [ 1, 1, 0, 0 ], [ 1, 0, 1, 1 ], [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ] ] [ 0, 1, 1, 0 ]
[ [ 0, 1, 1, 1 ], [ 1, 1, 0, 0 ], [ 1, 0, 1, 1 ], [ 0, 0, 0, 0 ] ] [ 0, 0, 0, 0 ]
[ [ 1, 1, 0, 0 ], [ 0, 1, 1, 1 ], [ 1, 1, 0, 0 ], [ 1, 0, 1, 1 ] ] [ 1, 0, 1, 1 ]
[ [ 1, 0, 1, 0 ], [ 1, 1, 0, 0 ], [ 0, 1, 1, 1 ], [ 1, 1, 0, 0 ] ] [ 1, 1, 0, 0 ]
[ Z(2)^0, Z(2^2), Z(2^4), 0*Z(2), Z(2^4)^2, Z(2^4)^9 ]
gap> gap> RunFSR(test, 5, false);
[ Z(2^4)^11, Z(2^4)^9, Z(2^4)^3, Z(2)^0, Z(2^4)^2 ]
gap> seq := RunFSR(test, ist, 10, false);
[ Z(2)^0, Z(2^2), Z(2^4), 0*Z(2), Z(2^4)^2, Z(2^4)^9, Z(2^4)^11, Z(2^4)^9, Z(2^4)^3, Z(2)^0, Z(2^4)^2 ]

```

Example 6.2.2(e)

```

gap> elmvec := [Z(2)^0, Z(2)^0, 0*Z(2), Z(2^4)^7, Z(2^4)^6]; seq1 := RunFSR(test, ist, elmvec, true);
using basis B := [ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6 ]
elm
[ 3, ... ...,0 ] with taps [ 0 ]
[ 0, 0, 0, 0 ] [ [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ], [ 1, 0, 1 ], [ 1, 0, 0, 0 ] ] [ 1, 0, 0, 0 ]
[ 1, 0, 0, 0 ] [ [ 0, 0, 1, 1 ], [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ], [ 1, 1, 0, 1 ] ] [ 1, 1, 0, 1 ]
[ 1, 0, 0, 0 ] [ [ 1, 1, 0, 0 ], [ 0, 0, 1, 1 ], [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ] ] [ 0, 1, 1, 0 ]
[ 0, 0, 0, 0 ] [ [ 0, 1, 1, 1 ], [ 1, 1, 0, 0 ], [ 0, 0, 1, 1 ], [ 0, 0, 0, 0 ] ] [ 0, 0, 0, 0 ]
[ 0, 1, 0, 0 ] [ [ 0, 0, 0, 0 ], [ 0, 1, 1, 1 ], [ 1, 1, 0, 0 ], [ 1, 1, 0, 0 ] ] [ 0, 0, 1, 1 ]
[ 0, 0, 0, 1 ] [ [ 0, 0, 0, 1 ], [ 0, 0, 0, 0 ], [ 0, 1, 1, 1 ], [ 1, 1, 0, 0 ] ] [ 0, 0, 1, 1 ]
[ Z(2)^0, Z(2^2), Z(2^4), 0*Z(2), Z(2^4)^8, Z(2^4)^9 ]
gap> seq2 := RunFSR(test, 5);
[ Z(2^4)^11, 0*Z(2), Z(2^4)^6, Z(2^4)^8, Z(2^4)^9 ]
gap> Append(seq1, seq2); seq1; seq = seq1;
[ Z(2)^0, Z(2^2), Z(2^4), 0*Z(2), Z(2^4)^8, Z(2^4)^9, Z(2^4)^11, 0*Z(2), Z(2^4)^6, Z(2^4)^8, Z(2^4)^9 ]
false

```


Example 6.2.4 A symbolic NLFSR over a binary field \longleftrightarrow

The NLFSR constructor needs a multivariate polynomial and the length of the NLFSR n (Table 6.2). Each stage has a corresponding variable, but not all stages (variables) occur in the feedback, hence the need for n . Example 6.2.4 shows the NLFSR of length 6 with multivariate polynomial $x_0 + x_1x_3$ as feedback. It was loaded with symbols s_i , and run for 20 steps. After 16 steps, the sequence element with the maximal degree monomial is obtained: see `seq[17]` with degree 5.

Example 6.2.4

```
gap> K := GF(2);; f := x_0 + x_1*x_3;; n := 6;; test := NLFSR(F, f, n);;
gap> ist := [s_5, s_4, s_3, s_2, s_1, s_0];; seq := RunFSR(test, ist, 20);;
gap> for i in [1 ..10] do Print(" ", seq[i] , "\n\n"); od;
s_0
s_1
s_2
s_3
s_4
s_5
s_1*s_3+s_0
s_2*s_4+s_1
s_3*s_5+s_2
s_1*s_3*s_4+s_0*s_4+s_3
gap> Print(seq[16] , " -> ", DegreeOfPolynomialOverField(K, seq[16]), "\n\n");
s_0*s_1*s_4*s_5+s_1*s_2*s_3*s_4+s_1*s_3*s_4*s_5+s_2*s_3*s_4*s_5+s_0*s_1*s_4+s_0
*s_1*s_5+s_0*s_2*s_4+s_2*s_3*s_4+s_3 -> 4
gap> Print(seq[17] , " -> ", DegreeOfPolynomialOverField(K, seq[17]), "\n\n");
s_1*s_2*s_3*s_4*s_5+s_0*s_1*s_2*s_5+s_0*s_2*s_4*s_5+s_0*s_3*s_4*s_5+s_1*s_2*s_3
*s_5+s_1*s_3*s_4*s_5+s_2*s_3*s_4*s_5+s_0*s_1*s_2+s_1*s_2*s_3+s_1*s_2*s_5+s_1*s_3
*s_5+s_3*s_4*s_5+s_4 -> 5
```

If ran further, the obtained sequence will eventually start to repeat, i.e., there will be a cycle and its length is called symbolic period. For the NLFSR in this example, the symbolic period is 828. When loaded with actual initial states of finite field elements instead of symbols, cycles of lengths 3, 4, 6, 9, 12 and 23 were found. Initial state `ist=[1, 1, 1, 0, 0, 0]` yields a cycle of length 23. Note that 23 is the largest prime in factorization of 828, and that divisors of 828 are candidates for actual cycle lengths. Running the FSR objects loaded with symbolic states is very slow and requires a lot of memory. \longleftarrow

6.3 Summary of key insights

Table 6.6 summarizes the key insights highlighted in Section 6.1.

Key 6.1: A cipher as a collection of basic modules	✓✓	Section 6.1
A cipher can be represented as a collection of basic modules: LFSRs, NLFSRs, and FILFUNs, connected in various configurations.		
Key 6.2: Structural similarities between FSR objects	✓✓	Section 6.1
The LFSR and NLFSR GAP objects differ only in the degree of the multivariate polynomial used to define their feedback. A FILFUN is an FSR object without feedback, shifting, or storing, and its functionality is defined by a multivariate polynomial. The justification for such a design decision is twofold: (i.) filtering functions are similar to (NLFSR) feedback functions, and (ii.) FSRs with output filters are common, hence they will be used together.		
Key 6.3: Regular and external step and run	✓✓	Section 6.1
A stand-alone simple (N)LFSR object is self-contained: it is updated by the computed feedback value (regular step and run). The external step and run allow arbitrary filters (FILFUNs) to be added to the feedback of any FSR object. The external step and run allow for flexibility when representing a cipher as a collection of basic modules.		
Key 6.4: Methods for multivariate polynomials	✓✓	Section 6.1
These methods take finite field \mathcal{F} with q elements and a multivariate polynomial f as an input. <code>SplitCoeffsAndMonomials</code> splits f into a list of coefficients and a list of monomials. <code>ReduceMonomialsOverField</code> returns a polynomial f_1 with the same coefficients as f , but all exponents reduced modulo $q - 1$. <code>DegreeOfPolynomialOverField</code> returns the degree of the polynomial f_1 , that was obtained with <code>ReduceMonomialsOverField</code> .		

Notes: ✓✓ - solved

Table 6.6: Summary of key insights to the FSR package

Chapter 7

FFCSA package: finite field construction, search and algorithms

7.1	Main functionality of the FFCSA package	94
7.2	FFCSA profiling methods	99
7.3	Tower field bases	102
7.4	Algorithms: obtaining expressions for finite field arithmetic	106
7.5	Summary of key insights	111

```
graph TD; alg([alg.]) --> arch([arch.]); arch --> gen([gen.]);
```

Overview of the FFCSA package:

The FFCSA package is the main support, or rather, the prerequisite package for *the design space exploration*. FFCSA stands for finite field constructions, search and algorithms, and is divided into three major parts: search, bases, and algorithms. The first part is divided into search for field elements and for polynomials, that meet specified criteria. This criteria can be general (a normal element, an irreducible polynomial, etc.), or specialized (the smallest area element, etc.). The second part is dedicated to the generation of different bases: polynomial bases, normal bases, and their dual bases. When generating the tower field bases, different options are possible, using either the same type of basis on each level, or mixed bases, e.g., polynomial basis on one level and normal basis on the next. The third part consists of the algorithms, or rather, a collection of methods that generate the expressions required to implement an arithmetic operation according to a specified algorithm, parametrized for the current instance (defining polynomial, a basis, etc.). Many different algorithms are known in literature. The package supports basic functionality, e.g., the generalized algorithm for multiplication. The current emphasis is on parallel datapaths. Expressions for sequential implementations can be generated, and the modules can be implemented in conjunction with the FSR package, but the process is not yet fully integrated. Similarly, some algorithms rely on search results, but are implemented as stand-alone GAP scripts, e.g., a bit-parallel Reduced Redundancy Massey Omura multiplication. The expressions are generated for multiplications, arbitrary exponentiations, matrix-vector multipliers, and some special cases, e.g., the expression for the trace computation. Most FFCSA methods require only the method call and occasional initial parameter setup. The outputs appear in the GAP prompt. Future work includes method optimizations, extending the functionality, and extending the “reach” in terms of field size.

```
graph TD; FSR([FSR]) --> FFCSA([FFCSA]); WGcipher([WGcipher]) --> FFCSA; WAGE([WAGE]) --> FFCSA; FSR --> WGcipher; WGcipher --> WAGE;
```

■ **Implementation detail:** As the FSR package was intended to be standalone, the FFCSA, counter intuitively, reuses the helper functions from FSR, and thus the FSR package must be loaded first. ■

93

7.1 Main functionality of the FFCSA package

The following short examples (Examples 7.1.1-7.1.3) show the interplay of different parts of the FFCSA package and provide an insight into its structure. They illustrate the basic functionality of the FFCSA package: the constructions, the search and the algorithms. They are tasked with finding (special) elements, finding (special) polynomials, generating matrices and bases, computing Hamming weights, and finally, obtaining expressions for finite field arithmetic algorithms. The search is closely linked to *the design space exploration*: theoretical estimates, e.g., Hamming weights, can be used to make certain architectural decisions early in the design flow. More details on DSE are given in Section 7.2.

Example 7.1.1 *A small complexity normal basis*

A small complexity normal basis is believed to yield smaller hardware w.r.t. a normal basis with higher complexity [186]. A small complexity normal basis can be found with exhaustive search, which requires several steps: (i.) find the normal elements, (ii.) build their multiplication tables T , (iii.) compute the Hamming weights of tables T , and (iv.) find the minimum Hamming weight T . This example is covered in more detail in Section 7.2.

Example 7.1.2 *A small complexity transition matrix*

Another more complex example is a finite field for which no optimal normal bases exist, for example $\mathbb{F}_{2^{16}}$, but the algorithm requires a lot of exponentiations to powers of two. Normal bases are very convenient for the exponentiations. However, a tower field basis is more suitable for multiplications and inversions: the search criterion is to minimize the Hamming weights of the transition matrices between tower field and normal bases. A less obvious middle step is constructing the tower field itself, requiring (a search for) an irreducible polynomial for the next level at each level of the tower. This example is covered in more detail in Section 7.2.

Example 7.1.3 *A finite field multiplier*

Last but not least is the generation of expressions used for the hardware implementations, e.g., finite field multiplication. Specifically, to implement a \mathbb{F}_{2^4} multiplier, the following steps are performed: (i.) a specific defining polynomial of degree 4 is obtained by the search for defining polynomials, (ii.) the polynomial basis with the root of the defining polynomial is generated, (iii.) then matrix U [42] is generated for this basis using a vector of variables $[a_0, a_1, a_2, a_3]$, and (vi.) the expressions for the 4 components of the product are obtained by multiplying the matrix U (with indeterminates a_i) with the vector of variables $[b_0, b_1, b_2, b_3]$. Details are given in Section 7.4.

Tables 7.2 - 7.5 list the main FFCSA methods to demonstrate the capabilities of the package. The corresponding source files, listed on the top of each table, give insight into the structure of the package. Additional information about the methods is captured by listing their arguments, with the labels *ffe* for finite field elements, *f* for polynomials, etc. listed in Table 7.1. In some cases, method overloading was used but only the general case is listed in Tables 7.2 - 7.5. There are no

GAP attributes and properties, because this package does not construct any objects. Furthermore, there are many different methods that only differ in the search space, but have the exactly same arguments. This also implies that GAP's method selection cannot be used and that the methods must be differentiated by their names. The overlap between methods in the FFCSA package can be characterized as follows: (i.) same functionality - different implementation, (ii.) same functionality - reduced search space, and (iii.) similar functionality - different criterion. To reduce the size of tables, some methods are listed together with the following notation used as placeholders:

- ♣ - they have the same functionality and output but different implementation (e.g., `IsNormalFFE` and `IsNormalFFEB` in Table 7.3)
- ♠, •, ■ - they have similar functionality but work with a reduced search space:
 - ♠ - `IgnoreConjugates` methods: search space is reduced by ignoring the conjugates and only checking the elements obtained as g^c , where g is a generator of the field and c a coset leader (e.g., `NrNormalFFE` and `NrNormalFFEIgnoreConjugates` in Table 7.3)
 - - `OneGamma` or `FixedGamma` method: the search space for the polynomials is reduced by allowing only one coefficient (γ) to belong to the extension field, all other coefficients must belong to \mathbb{F}_2 . For `FixedGamma`, the value of γ is given too. (e.g., `FindPrimitivePolyOneGamma` and `FindPrimitivePolyFixedGamma` in Table 7.3)
 - - `FixedNumber` method: the search space is reduced by providing a fixed number of desired results; the search process is terminated when the threshold is reached (e.g., `Find□Poly■` and `Find□■` in Table 7.4)
- ◇, □, △, ▽ and ○ - they have the same functionality for a different search criterion:
 - - `Irreducible`, `Primitive` and `IrreducibleNotPrimitive` (e.g., `Find□PolyAll` in Table 7.4)
 - ◇ - `ONBI` and `ONBII`, or `L` and `R` (`FindONB◇Generator` in Table 7.4, `SLL` and `SRL Shift Left Logical` in Table 7.5)
 - △ - `To` and `Downto`: this distinction was adopted from VHDL signals, which are specified by their domain, range and direction (e.g., `(0 to 3)` or `(3 downto 0)`).
 - ▽ - basis type, e.g., `PB` for polynomial basis, `NB` for normal basis and `MB` for mixed basis in case of tower fields with different type of basis on each level.
 - - `Area` and `Delay` for profiling (examples in Section 7.2)

Notation ♣ used in tables always denotes the existence of both, the regular and the B (Basis) methods and ♠ the existence of both, the full and the `IgnoreConjugates` methods. A secondary reason to implement both is to check their correctness by comparing the outputs directly or, in the latter case, by expanding one output set to all conjugates before comparing. The theoretical values for expected number of particular objects were used as a final check for every search. The `Nr...` methods were implemented in the source file `misc`. There are several variants of the ◇ methods, and the specific cases have the options listed below the table, for example in Table 7.2, ◇ and in Table 7.4 for the `ONBI` and the `ONBII`, where `ONB` stands for optimal normal basis.

As was mentioned above, the notion of direction, e.g., `To` and `Downto`, was adopted from VHDL. An example of a similar phenomenon encountered in finite field arithmetic is the polynomial basis, which can be defined either as $\{1, \omega, \dots, \omega^{m-1}\}$, considered as “To” in this work, or as $\{\omega^{m-1}, \dots, \omega, 1\}$, considered as “Downto”. All normal bases are implicitly considered as “To” due to the increasing powers of the conjugates. All other bases are considered as “To”.

p	- characteristic	F	- finite field
q	- field size	ffe	- finite field element
m, n	- degree of polynomial or extension	f, N	- polynomials
M	- matrix	$B, B1, B2$	- bases
$edpl$	- extension field defining polynomial list	$vec, vec1, vec2$	- vectors
mbl	- mixed basis instruction list	i	- index w.r.t. M, B or vec
$Eblist$	- extension bases list	t	- threshold (integer)
edl	- list of extension degrees (integers)	e	- exponent
$[]$	- optional parameters	of	- output file name

Table 7.1: Labels used for arguments in Tables 7.2 - 7.5

<i>misc</i>	Section	<i>matrix</i>	Section	<i>weight</i>
<code>CCLeaders(p, m)</code> ^[1]	C.2.1	<code>MatrixMultByConst(B, ffe)</code> ^[2]	C.2.5	<code>WeightMatrixBoth(M)</code> ^[4,5]
<code>PolyPhi(F, f)</code> ^[1]	C.2.2	<code>TransitionMatrix($B1, B2$)</code> ^[2]	C.2.5	<code>WeightMatrix(M)</code> ^[4]
<code>Nr□Poly(F, m)</code>	C.2.3	<code>ReductionMatrixΔ(f)</code> ^[2]	C.2.5	<code>WeightMatrixMaxRow(M)</code> ^[4]
<code>NrNormalFFE\star(F)</code>	C.2.4	<code>MatrixU(B), MatrixUi(B, i)</code> ^[3]	7.4	<code>WeightPolynomial(f)</code> ^[4]
		<code>MatrixM(B), MatrixMi(B, i)</code> ^[3]	C.2.5	

Implementation notes:

- [1] - Cyclotomic Coset Leaders and Polynomial Φ function
- [2] - returns a matrix M , which is then used as a matrix-vector (MV) multiplier for computation $z = Ma$ where a is the input vector (vector on the right)
- [3] - used for multiplication of two arbitrary elements
- [4] - returns number of nonzero entries (Hamming weight)
- [5] - returns both `WeightMatrix` (area estimate) and `WeightMatrixMaxRow` (delay estimate)
- - three methods listed together: the `Irreducible`, `Primitive` and `IrreducibleNotPrimitive`
- Δ - direction “to” or “downto”, and IR for direction “to”, RI for direction “downto”, where I is the identity matrix and R the reduction matrix, see Section C.2 for details

Table 7.2: Main functionality of the FFCSA package

■ **Implementation detail: The Basis (B) methods**

Method `IsNormalFFE` was implemented based on Theorem 5.2.11(1.) in [15]. The Basis (B) method is actually a trick: In GAP, a set of linearly independent elements S is not considered a basis until its tied to the finite field F , which is achieved using `Basis(F, S)` (Section 61.5-2 in the GAP reference manual [34]): this method will return an object which had the property `IsBasis`, the finite field it belongs to and the list `BasisVectors` (Section 61.6-1 in the GAP reference manual [34]). If the set S is not linearly independent, `Basis(F, S)` will return “fail”. The `IsNormalFFEB(F, ffe)` method will construct the set S consisting of conjugates of ffe and call `Basis(F, S)`: if “fail” `IsNormalFFEB` returns “false”. All search methods use either `IsNormalFFE` or `IsNormalFFEB`, however the B (Basis) methods perform much faster than the regular ones, and were therefore used for most of the exhaustive searches performed. ■

<i>nb</i>	Section	<i>ffbases</i> ‡	Section
IsONBI(q, m)	C.2.6	GeneratePB Δ (F, ffe) ^[1]	
IsONBII($2, m$)	C.2.6	GenerateNB(F, ffe) ^[2]	
IsNormalFFE*(F, ffe)	†Thm.3.2	IsDualBasisPair($B1, B2$)	C.2.7
MultiplicationTableT(B)	†Eq.(3.5)	FindInvCyc(F, N)	C.2.7
ComplexityOfT(B)	3.1.1 & 18.1	GeneratorOfDBtoNB(F, ffe)	C.2.7
		GenerateDBtoNB(F, ffe)	C.2.7
		GenerateDBtoPB Δ (F, ffe)	C.2.7
		GenerateTFBfromEBlist($EBlist$)	7.3.2
		GenerateTFBfromEDPLwith ∇ ($edpl$ [mbl])	7.3.2

Implementation notes:

- † - Subection 3.1.1 in Section 3.1
- ‡ - all bases generated are linearly independent sets of m elements, where m is the degree of extension of F
- [1] - returns a basis of F , formed from $ffe = \alpha$ according to the direction given by Δ :
 $\{1, \alpha, \dots, \alpha^{m-1}\}$ for direction “to” or $\{\alpha^{m-1}, \dots, \alpha, 1\}$ for direction “downto”, where $m = [\mathbb{F}_{q^m} : \mathbb{F}_q]$
- [2] - returns a basis of F , formed from $ffe = \alpha$ as $\{\alpha, \alpha^q, \dots, \alpha^{q^{m-1}}\}$, where $m = [\mathbb{F}_{q^m} : \mathbb{F}_q]$
- ∇ - four methods listed together: the MB for mixed basis, and the PB, PBDownto, NB

Table 7.3: Main functionality of the FFCSA package - continued

<i>FindElm/FindElmSpecial</i> †	Section	<i>FindPoly/FindPolySpecial</i>	Section
FindNormalFFEs**(F) ^[1,2]	18.1	Find \square PolyAll($[of,] F, m$ [t]) ^[4]	
FindONB \diamond Generator*(F) ^[3]	C.2.8	FindEDPLAllfromEDL(edl)	7.3.2
ProfileNBGenerators(F)	7.2	FindPrimitivePoly•($[of,] F, m$ [ffe]) ^[5]	
FindSmallestTNBGenerator(F) ^[3]	7.2	FindPrimitivePolyExtraTapsFixedPoly•	20.4
ProfileNBtoBTransitionMat(B)	7.2	($[of,] F, f, t$) ^[6]	
ProfileGamma(F, B)	7.2		
FindSmallest \circ Gamma(F, B) ^[3]	7.2		

Implementation notes:

- [1] - for $F \leq \mathbb{F}_{2^{16}}$, see Section C.2.10 comment (1)
- [2] - returns a list of integers e , such that g^e is the normal element, where g is a generator of the field
- [3] - returns a finite field element
- [4] - when writing to output file of : write one polynomial per line
- [5] - ffe is needed for FixedGamma method
- [6] - f is a given primitive polynomial. t is number of extra tap positions
- \diamond - two methods listed together: the ONBI and the ONBII
- † - all methods are IgnoreConjugates, except the FindNormalFFEs**

Table 7.4: Main functionality of the FFCSA package - continued

algs/matrixalgs	
ChooseFieldElms $\Delta(F)$ ^[1,2]	FFA_mult_convolution(<i>vec1</i> , <i>vec2</i>)
S \diamond L(<i>vec</i> , <i>i</i>), R0 \diamond (<i>vec</i> , <i>i</i>)	FFA_mult_2stepClassic(<i>f</i> , <i>vec1</i> , <i>vec2</i> , <i>dir</i>)
MatrixExpression(<i>M</i> , <i>vec</i>)	FFA_mult_matrixU(<i>B</i> , <i>vec1</i> , <i>vec2</i>) ^[1]
TransitionMatrixExpression(<i>B1</i> , <i>B2</i> , <i>vec</i>)	FFA_sq_matrixU(<i>B</i> , <i>vec</i>) ^[1]
ReductionMatrixExpression $\Delta(f, vec)$	FFA_exp_matrixU(<i>B</i> , <i>vec</i> , <i>e</i>) ^[1]
MatrixMultByConstExpression(<i>B</i> , <i>ffe</i> , <i>vec</i>)	FFA_inv_matrixU(<i>B</i> , <i>vec</i>) ^[1]
MatrixUExpression(<i>B</i> , <i>vec</i>) ^[1]	FFA_trace(<i>B</i> , <i>vec</i>)

Implementation notes:

- \diamond - two methods listed together: L, R (e.g., SLL - Shift Left Logical)
- [1] - Section 7.4
- [2] - Section 10.2

Table 7.5: Main functionality of the FFCSA package - continued

■ **Implementation detail: The Find \square PolyAll methods**

The Find \square PolyAll(*F*, *m*) methods use the existing GAP functions IsPrimitivePolynomial(*F*, *poly*) (Section 66.4-12 in the GAP reference manual [34]) and IsIrreducibleRingElement(PolynomialRing(*F*), *poly*) (Sections 56.5-7 and 66.15-1 in the GAP reference manual [34]) to decide if a candidate polynomial *poly* meets the search criteria. The Nr \square Poly(*F*, *m*) are used to check the number of polynomials found; if there is a discrepancy, and error is triggered. ■

Key 7.1: Search for parameters and *the design space exploration*

The basic functionality of the FFCSA methods covers: the constructions, the search, and the algorithms. The methods are tasked with finding (special) elements, finding (special) polynomials, generating matrices and bases, computing Hamming weights, and finally, obtaining expressions for finite field arithmetic algorithms. The search is closely linked to *the design space exploration*: theoretical estimates, e.g., Hamming weights, can be used to make certain architectural decisions early in the design flow. The algorithms are mandatory for *the automated design generation*.

7.2 FFCSA profiling methods

The FFCSA package contains a set of profiling methods that are closely linked to the search for “special” elements and polynomials, i.e., *FindElmSpecial* and *FindPolySpecial* (Table 7.4). The *profile* is always a (set of) Hamming weight(s): a theoretical estimate of area, as was already noted for the *WeightMatrix* method, or delay, as was noted for the *WeightMatrixMaxRow* method (Table 7.2). Another method, linked to the Hamming weight of a matrix, is *ComplexityOfT* (Table 7.3). The following profiling functions are in place:

- *ProfileNBGenerators*(F) finds all normal elements of the finite field F and lists the $[c, e]$ as their profile, where e is the current exponent and c the *ComplexityOfT* of the normal basis generated by the normal element g^e , where g is the generator of the finite field F . Implementation details on *ComplexityOfT* will be shown in Section 18.1. *FindSmallestTNBGenerator*(F) uses the profile to find the element with the smallest *ComplexityOfT* value C_T . *FindSmallestTNBGenerator* is a solution to Example 7.1.1 in Section 7.1. A short example is shown in the GAP code in Example 7.2.1(a).
- *ProfileNBtoBTransitionMat*(B) finds all normal elements of the finite field F , generates their normal bases NB , and computes both transition matrices between the current normal basis NB and the basis B passed on as an argument, using *TransitionMatrix* (Table 7.2). The profile lists $[d1, A1, d2, A2, e]$, where e is the current exponent, $d1, A1$ the delay and area of the $NB \rightarrow B$ transition matrix, and $d2, A2$ the delay and area of the inverse matrix $B \rightarrow NB$. *FindSmallestAreaNBtoBProfile*(B) returns the profile with the smallest $A1+A2$ w.r.t. the given B . If two profiles have the same cumulative area, the delays are used as a secondary criterion, and if there is still a deuce, the profile with the smallest e is chosen. *ProfileNBtoBTransitionMat* is a solution to Example 7.1.2 in Section 7.1 for an arbitrary basis B . A short example is shown in the GAP code in Example 7.2.1(b).
- *ProfileGamma*(B) computes the profiles for elements g^e , where e is the current exponent and g is the generator of the finite field F . The profile is $[d, A, e1, e2, \dots]$, where d and A are the delay and area of the matrix for multiplication with the constants g^{e_i} , $i = 1, 2, \dots$. The exponents are grouped together when the delay and area of their corresponding matrices are the same. The matrices are obtained with *MatrixMultByConst* (Table 7.2). Unlike the profiling methods introduced above, the *ProfileGamma* returns a pre-processed profile, ordered by ascending delays. *FindSmallest \circ Gamma*, where \circ stands for *Area* or *Delay*, returns an element that fits the given criteria. For profiles with more than one exponent, the first exponent is selected, namely $e1$ giving g^{e1} . Short examples for normal and polynomial basis B are shown in the GAP code in Example 7.2.1(c, d).

■ **Implementation detail:** The exponents e for candidate elements g^e

The exponents are the elements returned by *CCLeaders*($2, m$) (Table 7.2), where m is the degree of extension of the finite field used. All methods using the coset leaders as exponents are *IgnoreConjugates* methods. ■

Example 7.2.1 *FFCSA profiling examples for* $\mathbb{F}_{2^7} \longleftrightarrow$

Example 7.2.1(a) shows the functionality of methods `ProfileNBGenerators` and `FindSmallestTNBGenerator`. Note that to find the normal basis with smallest complexity, only the call `FindSmallestTNBGenerator(F)` is needed, as shown by the last call Example 7.2.1(a). The setup in the first line of code, exponents that yield normal elements, and the profile are shown to clarify the example.

Example 7.2.1(a)

```
gap> F := GF(2^7);; g := Z(2^7);; B := Basis(F);;
gap> FindNormalFFEIgnoreConjugates(F);
[ 13, 19, 21, 27, 31, 43, 63 ]
gap> ProfileNBGenerators(F);
[ [ 19, 13 ], [ 27, 19 ], [ 21, 21 ], [ 27, 27 ], [ 25, 31 ], [ 21, 43 ], [ 21, 63 ] ]
gap> FindSmallestTNBGenerator(F);
Z(2^7)^13
gap> FindSmallestTNBGenerator(GF(2^11));
Z(2^11)^439
```

Example 7.2.1(b) shows methods `ProfileNBtoBTransitionMat` and `FindSmallestAreaNBtoBProfile` w.r.t. the polynomial basis of \mathbb{F}_{2^7} . The smallest cumulative area transition matrices are found for g^{43} .

Example 7.2.1(b)

```
gap> ProfileNBtoBTransitionMat(B);
[ [ 7, 29, 7, 31, 13 ], [ 7, 33, 6, 25, 19 ], [ 7, 33, 6, 27, 21 ],
  [ 7, 27, 6, 33, 27 ], [ 7, 31, 6, 29, 31 ], [ 7, 27, 5, 27, 43 ],
  [ 7, 25, 6, 33, 63 ] ]
gap> FindSmallestAreaNBtoBProfile(B);
[ 7, 27, 5, 27, 43 ]
```

Example 7.2.1(c) shows methods `ProfileGamma` and `FindSmallestAreaGamma` w.r.t. the normal basis of \mathbb{F}_{2^7} generated for g^{43} , the element selected in Example 7.2.1(b). The smallest area multiplication matrix is found for g^{43} . Note that this is exactly the normal element.

Example 7.2.1(d) shows methods `ProfileGamma` and `FindSmallestAreaGamma` w.r.t. the polynomial basis of \mathbb{F}_{2^7} generated for the root ω of defining polynomial $x^7 + x^3 + x^2 + x + 1$. The root ω is also a generator (see last line in the GAP in code Example 7.2.1(d)), and has the value $Z(2^7)^5$. The smallest area multiplication matrix is found for the root of defining polynomial ω .

Example 7.2.1(c)

```

gap> NB := GenerateNB(F, g^43);; profile := ProfileGamma(NB);;
gap> for i in profile do Display(i); od;
[ 4, 20, 1, 11 ]
[ 4, 22, 3 ]
[ 5, 24, 5 ]
[ 4, 19, 7, 47 ]
[ 5, 26, 9 ]
[ 5, 27, 13 ]
[ 6, 26, 15, 23 ]
[ 6, 31, 19 ]
[ 7, 29, 21 ]
[ 6, 23, 27 ]
[ 6, 32, 29 ]
[ 5, 29, 31 ]
[ 4, 21, 43 ]
[ 6, 28, 55 ]
[ 5, 25, 63 ]
gap> FindSmallestAreaGamma(NB);
Z(2^7)^7

```

Example 7.2.1(d)

```

gap> f := x^7 + x^3 + x^2 + x + 1;; F := FieldExtension(K, f);;
gap> w := RootOfDefiningPolynomial(F);
Z(2^7)^5
gap> PB := GeneratePB(F, w);; profile := ProfileGamma(PB);;
gap> for i in profile do Display(i); od;
[ 2, 10, 1 ]
[ 4, 16, 3 ]
[ 5, 25, 5 ]
[ 6, 32, 7 ]
[ 6, 31, 9, 29 ]
[ 6, 30, 11 ]
[ 5, 27, 13, 63 ]
[ 4, 25, 15 ]
[ 4, 22, 19, 21 ]
[ 5, 29, 23 ]
[ 6, 34, 27 ]
[ 5, 31, 31 ]
[ 6, 24, 43 ]
[ 4, 20, 47, 55 ]
gap> FindSmallestAreaGamma(PB);
Z(2^7)^5
gap> GeneratorOfField(F);
Z(2^7)^5

```



7.3 Tower field bases

7.3.1 Generating tower field bases

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), it is possible to 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 . Recall from Section 3.1 the equation (3.3), which for the example $\mathbb{F}_{((2^2)^2)^2}$ takes the following form

$$\mathbb{F}_2 \subset \mathbb{F}_{2^2} \subset \mathbb{F}_{(2^2)^2} \subset \mathbb{F}_{((2^2)^2)^2} \cong \mathbb{F}_{2^8}$$

For the generalized methods for generating tower field bases to arbitrary degree of extension and different bases, e.g., a mix of polynomial and normal bases, a distinction is made between *reference field* defining polynomials (RDP) and *extension field* defining polynomial (EDP). This distinction is explained with the example of $\mathbb{F}_{((2^2)^2)^2}$ shown in Figure 7.1:

- the *reference field* defining polynomials are labelled p_1, p_2, p_3 . They are the defining polynomials of the isomorphic reference finite fields. For example, p_3 is a degree 8 defining polynomial of the finite field \mathbb{F}_{2^8} , which is isomorphic to $\mathbb{F}_{((2^2)^2)^2}$.
- the *extension field* defining polynomials are labelled f_1, f_2, f_3 , and can be seen on the outer left side of the diagram. For example, the degree 2 polynomial f_3 is irreducible over \mathbb{F}_{2^2} , and used to construct the level $\mathbb{F}_{((2^2)^2)^2}$ as an extension of $\mathbb{F}_{(2^2)^2}$.

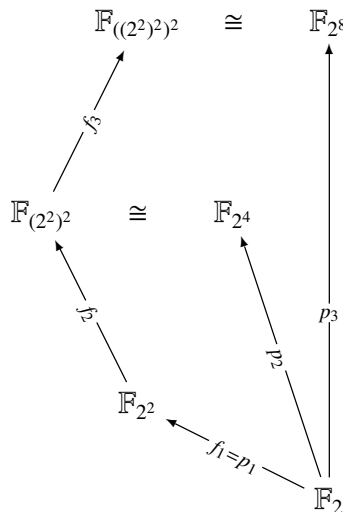


Figure 7.1: Finite field $\mathbb{F}_{((2^2)^2)^2}$ - tower construction

With each new extension $\mathbb{F}_{q^2}/\mathbb{F}_q$, a “per-level” basis (PLB) can be found. For this example, a polynomial basis with the root of the EDP f_i is used for each level. The “per-level” polynomial bases are shown in the third column in Table 7.6.

$$\mathbb{F}_2 \xrightarrow{f_1(x)} \mathbb{F}_{2^2} \xrightarrow{f_2(x)} \mathbb{F}_{(2^2)^2} \xrightarrow{f_3(x)} \mathbb{F}_{((2^2)^2)^2}$$

Finite field	EDP	“per-level” PB	Comments
\mathbb{F}_{q^2}	$f_i(x)$	$B_{\mathbb{F}_{q^2}/\mathbb{F}_q} = \{1, \rho\}$	$f_i(\rho) = 0$
$\mathbb{F}_{((2^2)^2)^2}$	$f_3(x)$	$B_{\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}} = \{1, \nu\}$	$f_3(\nu) = 0$
$\mathbb{F}_{(2^2)^2}$	$f_2(x)$	$B_{\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}} = \{1, \mu\}$	$f_2(\mu) = 0$
\mathbb{F}_{2^2}	$f_1(x)$	$B_{\mathbb{F}_{2^2}/\mathbb{F}_2} = \{1, \lambda\}$	$f_1(\lambda) = 0$

Table 7.6: Tower construction of $\mathbb{F}_{((2^2)^2)^2}$

An element $A \in \mathbb{F}_{((2^2)^2)^2}$ is represented w.r.t. the basis $B_{\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}}$ as $A = a_0 + a_1\nu$, where $a_0, a_1 \in \mathbb{F}_{(2^2)^2}$. Both coordinates a_0, a_1 are then represented w.r.t. the basis $B_{\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}}$, as $a_i = a_{i0} + a_{i1}\mu$, where $a_{i0}, a_{i1} \in \mathbb{F}_{2^2}$, and so on. The bases $B_{\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}}$, $B_{\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}}$ and $B_{\mathbb{F}_{2^2}/\mathbb{F}_2}$ are the “per-level” bases (see Table 7.6 for details). The PLBs lead to the construction of a tower field basis (TFB) of the isomorphic field $\mathbb{F}_{2^8}/\mathbb{F}_2$, obtained as a product of basis elements along the paths (arrows), as shown in Figure 7.2. The number of PLBs equals the number of levels in the tower field construction, but there is only one corresponding TFB. The obtained basis has 8 elements t_i , $0 \leq i \leq 7$:

$$TFB_{\mathbb{F}_{2^8}/\mathbb{F}_2} = \{t_0, t_1, \dots, t_7\} = \{1, \lambda, \mu, \mu\lambda, \nu, \nu\lambda, \nu\mu, \nu\mu\lambda\} \quad (7.1)$$

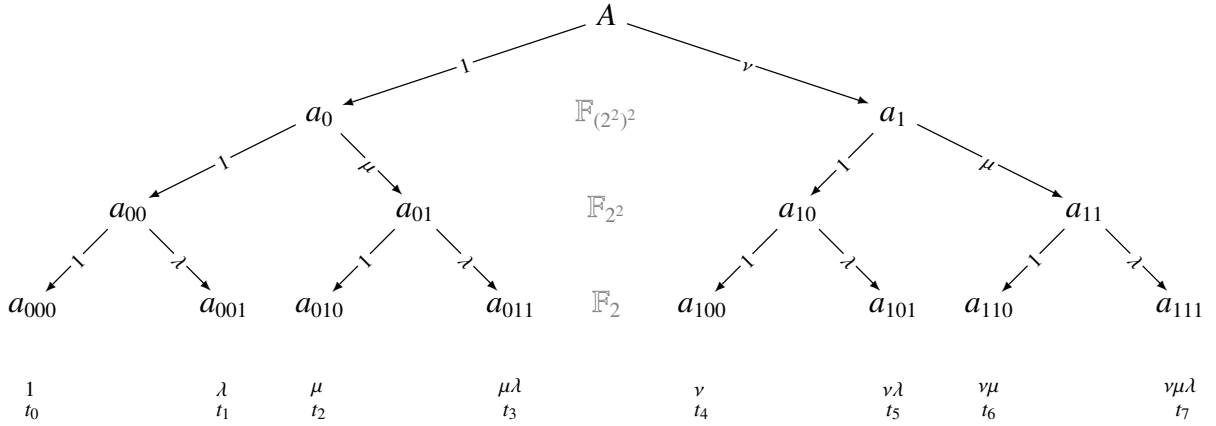


Figure 7.2: Decomposition of an element $A \in \mathbb{F}_{((2^2)^2)^2}$ w.r.t. $B_{\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}}$, $B_{\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}}$, $B_{\mathbb{F}_{2^2}/\mathbb{F}_2}$

The basis $TFB_{\mathbb{F}_{2^8}/\mathbb{F}_2}$ is different from the polynomial basis of $\mathbb{F}_{2^8}/\mathbb{F}_2$, obtained from the root ψ of the reference field polynomial, i.e., $p_3(\psi) = 0$:

$$PB_{\mathbb{F}_{2^8}/\mathbb{F}_2} = \{b_0, b_1, \dots, b_7\} = \{1, \psi, \psi^2, \psi^3, \psi^4, \psi^5, \psi^6, \psi^7\} \quad (7.2)$$

This can be seen from the two polynomials that are irreducible over the ground field \mathbb{F}_2 , namely the degree 2 polynomial $f_1(x)$ and the degree 8 polynomial $p_3(x)$. There is only one irreducible

polynomial of degree 2, $f_1(x) = x^2 + x + 1$, so even without fixing the polynomial $p_3(x)$ of degree 8, it can be shown that λ , a finite field element of order 2, can not generate the polynomial basis of $\mathbb{F}_{2^8}/\mathbb{F}_2$. Assume $\lambda = \psi$. From $f_1(\lambda) = 0$ it follows that $\lambda^2 = \lambda + 1$ and $\lambda^3 = \lambda^2 + \lambda = 1$. Then the elements in $\{1, \lambda, \lambda^2, \lambda^3, \dots, \lambda^7\} = \{1, \lambda, \lambda^2, 1, \dots, \lambda\}$ repeat, which means this set is linearly dependent and therefore not a basis. Thus, $\lambda \neq \psi$. Since $t_1 \neq b_1$, and bases are ordered sets, there is no need to check other basis elements; the two bases are different.

The tower field basis as given in equation (7.1), can be generalized to an arbitrary tower field construction, see Section C.2.9 in Appendix C.2 for more details.

7.3.2 FFCSA methods for generating TFBs

`GenerateTFBfromEBlist(PLBlist)` takes a list of “per-level” bases *PLBlist*. For the $\mathbb{F}_{(2^2)^2}$ used in this section, the input $[[1, \lambda], [1, \mu], [1, \nu]]$ returns the TFB in equation (7.1).

`GenerateTFBfromEDPLwithMB(edpl, mbl)` takes a list of extension defining polynomials *edpl*, and a “mixed basis list” *mbl*. The latter is a list of instructions that are used to generate the “per-level” bases. It is needed to select appropriate methods, e.g., ["NB", "to"] will call `GenerateNB` (Table 7.3) or `GenerateTFBfromEDPLwithNB`. Currently supported bases for the tower field construction are the polynomial basis (both directions), and the normal basis (direction “to”). First, the extension fields are built. Then, for each extension, the PLB is generated using the root of its EDP¹ and added to the *PLBlist*. Finally, the TFB is generated using `GenerateTFBfromEBlist`.

`FindEDPLAllfromEDL(edl)` is used to find the list of extension defining polynomials EDPs *edpl*. The argument *edl* is a list of extension degrees n_i , such that $m = n_1 \cdots n_k$. For the $\mathbb{F}_{(2^2)^2}$ used in this section, the input *edl* = [2, 2, 2].

The remaining methods all use the `GenerateTFBfromEDPLwithMB(edpl, mbl)` method with the parameter *mbl* fixed:

- `GenerateTFBfromEDPLwithPB` uses instructions ["PB", "to"] for each level
- `GenerateTFBfromEDPLwithPBdownto` uses instructions ["PB", "downto"] for each level
- `GenerateTFBfromEDPLwithNB` uses instructions ["NB", "to"] for each level

■ **Implementation detail: The mixed basis list instructions**

GAP has an object called `Dictionary` (Chapter 28 in the GAP reference manual [34]), and the *mbl* instructions are used to retrieve the strings “GeneratePB”, “GeneratePBdownto” and “GenerateNB” from the dictionary. Then, these strings are concatenated with brackets and the appropriate values for *F* and *ffe*. For example, the resulting string is “GenerateNB(F, ffe)”. Note that this is a string, but corresponds exactly to the format of the `GenerateNB` method (Table 7.3). Then, using the existing GAP function `EvalString` (Section 27.9-5 in the GAP reference manual [34]), GAP executes the `GenerateNB` method with the given arguments. Such approach was used in several parts of the automation framework. ■

¹ in case of normal bases, the EDP is expected to be a N-polynomial, and the method fails if it is not

Example 7.3.1 $\mathbb{F}_{((2^2)^2)^2}$ *tower field basis using a polynomial basis for each level* \longleftrightarrow

Example 7.3.1 shows the generation of the basis $TFB_{\mathbb{F}_{2^8}/\mathbb{F}_2}$, using the isomorphic tower field construction $\mathbb{F}_{((2^2)^2)^2}$, with the following EDPs:

$$f_1(x) = x^2 + x + 1, \quad f_2(x) = x^2 + \lambda x + 1, \quad f_3(x) = x^2 + \lambda x + \lambda^2 \mu$$

The initial setup for GAP code in Example 7.3.1(a) requires the list of extension defining polynomials *edpl*, which is selected from the output generated by the method `FindEDPLAllfromEDL`. The long outputs were manually shortened for this example. The remaining GAP code in Example 7.3.1(a) shows the tower field basis $TFB_{\mathbb{F}_{2^8}/\mathbb{F}_2}$ is generated using `GenerateTFBfromEDPLwithPB`, yielding TFB1.

Example 7.3.1(a)

```
gap> K := GF(2);; listall := FindEDPLAllfromEDL([2,2,2]);
[ [ x^2+x+Z(2)^0 ], [ x^2+Z(2^2)*x+Z(2)^0, x^2+Z(2^2)*x+Z(2^2), x^2+x+Z(2^2), x^2
+x+Z(2^2)^2, x^2+Z(2^2)^2*x+Z(2)^0, x^2+Z(2^2)^2*x+Z(2^2)^2 ], [ x^2+
Z(2^4)^3*x+Z(2)^0, x^2+Z(2^4)^3*x+Z(2^2), x^2+Z(2^4)^3*x+Z(2^4)^2,
... OMITTED FOR BREVITY ...
x^2+x+Z(2^4)^14, x^2+Z(2^2)*x+Z(2^4), x^2+Z(2^2)*x+Z(2^4)^2,
... OMITTED FOR BREVITY ...
x^2+Z(2^4)^13*x+Z(2^4)^14 ] ]
gap> edpl := [ listall[1][1], listall[2][1], listall[3][25] ];
[ x^2+x+Z(2)^0, x^2+Z(2^2)*x+Z(2)^0, x^2+Z(2^2)*x+Z(2^4) ]
gap> TFB1 := GenerateTFBfromEDPLwithPB(edpl);
Basis( GF(2^8), [ Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11, Z(2^8)^76, Z(2^8)^161,
Z(2^8)^178, Z(2^8)^8 ] )
```

Same basis TFB2 is obtained manually in Example 7.3.1(b) following the equation (7.1). The manual procedure is much longer than the three lines of GAP code in Example 7.3.1(a).

Example 7.3.1(b)

```
gap> f1 := edpl[1];; f2 := edpl[2];; f3 := edpl[3];;
gap> F1 := FieldExtension(K, f1);; lambda := RootOfDefiningPolynomial(F1);;
gap> F2 := FieldExtension(F1, f2);; mu := RootOfDefiningPolynomial(F2);;
gap> F3 := FieldExtension(F2, f3);; nu := RootOfDefiningPolynomial(F3);;
gap> TFBset := [ One(K), lambda, mu, mu*lambda, nu, nu*lambda, nu*mu, nu*mu*lambda ];;
gap> TFB2 := Basis(GF(2^8), TFBset); TFB1 = TFB2;
Basis( GF(2^8), [ Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11, Z(2^8)^76, Z(2^8)^161,
Z(2^8)^178, Z(2^8)^8 ] )
true
```

In Example 7.3.1(c) an RDP p_3 is chosen and its root ψ used to obtain the polynomial basis $PB_{\mathbb{F}_{2^8}/\mathbb{F}_2}$, as shown in equation (7.2), yielding the basis PB. The obtained PB (equation (7.2)) is indeed different from the TFB (equation (7.1)) TFB1.

```

gap> p3 := x^8+x^4+x^3+x^2+1;; F := FieldExtension(K, p3);;
gap> psi := RootOfDefiningPolynomial(F);
Z(2^8)
gap> PB := GeneratePB(F, psi);
Basis( GF(2^8), [ Z(2)^0, Z(2^8), Z(2^8)^2, Z(2^8)^3, Z(2^8)^4, Z(2^8)^5, Z(2^8)^6,
Z(2^8)^7 ] )
gap> TFB1 = PB; lambda = psi;
false
false

```

←

7.4 Algorithms: obtaining expressions for finite field arithmetic

7.4.1 Expressions obtained using matrix U

Expressions obtained with matrix U follow the *Generalized algorithm for multiplication* from Slide Set 3 in [42]. This method produces a matrix-vector multiplier where one of the factors is merged into the matrix U and then multiplied by the other factor. The generalized algorithm for multiplication was chosen because it works for an arbitrary basis. The two-step classic algorithm is limited to polynomial bases, and the Massey-Omura algorithm to normal bases. This section describes how to obtain the expressions for a multiplier. Then the multiplier is used to obtain various exponents, all the way up to the inversion.

Let $A, B, C \in \mathcal{F}$ and let $B_{\mathcal{F}/\mathcal{K}} = \{\alpha^{(0)}, \alpha^{(1)}, \dots, \alpha^{(m-1)}\}$ be an arbitrary basis of \mathcal{F}/\mathcal{K} , where $m = [\mathcal{F} : \mathcal{K}]$. The following are the representation of A w.r.t. basis $B_{\mathcal{F}/\mathcal{K}}$, its vector form and notation for the i -th coordinate of A , $0 \leq i \leq m - 1$:

$$A = \sum_{i=0}^{m-1} a_i \alpha^{(i)} \quad [A] = [a_0, a_1, \dots, a_{m-1}] \quad [A]_{(i)} = a_i$$

where $a_i \in \mathcal{K}$, $i = 0, 1, \dots, m - 1$. The matrix U is an $m \times m$ matrix with components $u_{i,j}$, $0 \leq i, j \leq m - 1$, obtained by multiplying element A with the j -th basis element $\alpha^{(j)}$ and then taking the i -th coordinate of the product $\alpha^{(j)} \cdot A$:

$$u_{i,j} = [\alpha^{(j)} \cdot A]_{(i)} \tag{7.3}$$

The columns of matrix U are exactly the vectors $[\alpha^{(j)} A]$. The product $C = A \cdot B$ can be written in matrix form as

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,m-1} \\ u_{1,0} & u_{1,1} & \dots & u_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ u_{m-1,0} & u_{m-1,1} & \dots & u_{m-1,m-1} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{bmatrix} \quad (7.4)$$

The expressions for the product C can then be obtained by multiplying the r.h.s of equation (7.4):

$$c_i = \sum_{j=0}^{m-1} u_{i,j} b_j \quad 0 \leq i \leq m-1 \quad (7.5)$$

The expressions obtained in this manner are used for the implementation of a multiplier circuit.

7.4.2 The matrix U methods in the FFCSA package

The FFCSA package implements several arithmetic operations (Table 7.5) using the matrix U obtained by equation (7.3). The multiplication, squaring, and exponentiations using matrix U are implemented as follows:

1. `ChooseFieldElms(F)` prepares GAP variables a_i and b_i for vectors $avec = [A]_{B_{\mathcal{F}/\mathcal{K}}}$ and $bvec = [B]_{B_{\mathcal{F}/\mathcal{K}}}$, to allow symbolic computation.
2. `MatrixU(B)` computes the matrix U with the elements based on $u'_{i,j} = [\alpha^{(i)}\alpha^{(j)}]$; that is the product of two basis elements represented w.r.t. basis $B = B_{\mathcal{F}/\mathcal{K}}$.
3. `MatrixUExpression($B, avec$)` computes the matrix U with elements obtained based on equation (7.3). It requires inputs $B = B_{\mathcal{F}/\mathcal{K}}$ and the vector of symbols $avec = [A]$.
4. `FFA_mult_matrixU($B, avec, bvec$)` first computes the matrix U with symbols a_i as $Uexpr = \text{MatrixUExpression}(B, avec)$, and then returns the product of the obtained matrix and the second vector: $Uexpr * bvec$.
5. `FFA_sq_matrixU($B, avec$)` first computes the matrix U expressions for $avec$, i.e., $Uexpr = \text{MatrixUExpression}(B, avec)$, and then uses the same vector again to obtain $Uexpr * avec$. Finally, the exponents of all m expressions are reduced modulo $|\mathcal{K}| - 1$.
6. `FFA_exp_matrixU($B, avec, e$)` computes the expressions for exponentiation A^e using a classic square and multiply method. Both squaring and multiplication are using matrix U , and the exponents are reduced modulo $|\mathcal{K}| - 1$ on each step.
7. `FFA_inv_matrixU($B, avec$)` computes the inverse expressions as exponentiation with e set to $|\mathcal{F}| - 2$ (Remark 2.1.17, page 14 in [15]): `FFA_exp_matrixU($B, avec, e$)`.

All reductions modulo $|\mathcal{K}| - 1$ use `ReduceMonomialsOverField` (Table 6.5).

Example 7.4.1 shows multiplication expressions obtained for \mathbb{F}_{2^4} using a polynomial basis. Example C.3.3 in Appendix C.3 shows the squaring expressions for the same field and basis. Example 7.4.2 shows the multiplication expressions for a tower field $\mathbb{F}_{(2^2)^2}/\mathbb{F}_{(2^2)^2}$.

Example 7.4.1 Multiplication expressions for \mathbb{F}_{2^4}

The following simple example shows how to compute the matrix U for a polynomial basis. The irreducible polynomial used in this example is $f(x) = x^4 + x + 1$ with root α , yielding $PB = \{1, \alpha, \alpha^2, \alpha^3\}$. Expressions used for the reduction are $\alpha^4 = \alpha + 1$, $\alpha^5 = \alpha^2 = \alpha$ and $\alpha^6 = \alpha^3 + \alpha^2$.

$$\begin{aligned} A &= a_0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3 \\ \alpha A &= a_0\alpha + a_1\alpha^2 + a_2\alpha^3 + a_3\alpha^4 \\ &= a_3 + (a_0 + a_3)\alpha + a_1\alpha^2 + a_2\alpha^3 \\ \alpha^2 A &= a_0\alpha^2 + a_1\alpha^3 + a_2\alpha^4 + a_3\alpha^5 \\ &= a_2 + (a_2 + a_3)\alpha + (a_0 + a_3)\alpha^2 + a_1\alpha^3 \\ \alpha^3 A &= a_0\alpha^3 + a_1\alpha^4 + a_2\alpha^5 + a_3\alpha^6 \\ &= a_1 + (a_1 + a_2)\alpha + (a_2 + a_3)\alpha^2 + (a_0 + a_3)\alpha^3 \end{aligned}$$

Obtained matrix U , with column vectors annotated on the top:

$$U = \begin{array}{c} \begin{array}{cccc} [A] & [\alpha A] & [\alpha^2 A] & [\alpha^3 A] \end{array} \\ \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 + a_3 & a_2 + a_3 & a_1 + a_2 \\ a_2 & a_1 & a_0 + a_3 & a_2 + a_3 \\ a_3 & a_2 & a_1 & a_0 + a_3 \end{bmatrix} \end{array}$$

GAP Example 7.4.1 shows the setup and matrix U , obtained by the method `MatrixUExpression`. At the end of the example, the output of `FFA_mult_matrixU(B, avec, bvec)` is shown: these are the expressions used for the hardware implementation. For example, to drive the multiplier output c_0 , the expression $a_0b_0 + a_1b_3 + a_2b_2 + a_3b_1$ must be implemented in hardware. The subexpressions seen in matrix U , e.g., $(a_0 + a_3)$, are not preserved in list `mult`, obtained by the `FFA_mult_matrixU`: they are transformed into the ANF form. The subexpression eliminations can in part be performed by the hardware synthesis tools.

Example 7.4.1(a)

```
gap> K := GF(2);; x := X(K, "x");; f := x^4+x+1;; F := FieldExtension(K, f);;
gap> PB := GeneratePB(F, RootOfDefiningPolynomial(F));; ChooseFieldElms(F);

variables
[ "a_0", "a_1", "a_2", "a_3" ]
[ "b_0", "b_1", "b_2", "b_3" ]
[ "d_0", "d_1", "d_2", "d_3", "d_4", "d_5", "d_6" ]
gap> U := MatrixUExpression(PB, avec);;
gap> for i in U do Display(i); od;
[ a_0, a_3, a_2, a_1 ]
[ a_1, a_0+a_3, a_2+a_3, a_1+a_2 ]
[ a_2, a_1, a_0+a_3, a_2+a_3 ]
[ a_3, a_2, a_1, a_0+a_3 ]
gap> mult := FFA_mult_matrixU(PB, avec, bvec);;
gap> for i in mult do Display(i); od;
a_0*b_0+a_1*b_3+a_2*b_2+a_3*b_1
a_0*b_1+a_1*b_0+a_1*b_3+a_2*b_2+a_2*b_3+a_3*b_1+a_3*b_2
a_0*b_2+a_1*b_1+a_2*b_0+a_2*b_3+a_3*b_2+a_3*b_3
a_0*b_3+a_1*b_2+a_2*b_1+a_3*b_0+a_3*b_3
```

Exact same expressions for the multiplication can be obtained with the school-book two-step classic method, shown in Example 7.4.1(b). `FFA_mult_2stepClassic(f, avec, bvec, "to")` calls method `FFA_mult_convolution(vec1, vec2)`, followed by and then the `ReductionMatrixExpressionDelta(f, dexpr)`. The latter two are shown in detail in Example C.3.2 in Appendix C.3.

Example 7.4.1(b)

```

gap> mult2sc := FFA_mult_2stepClassic(f, avec, bvec, "to");
gap> for i in mult2sc do Display(i); od;
a_0*b_0+a_1*b_3+a_2*b_2+a_3*b_1
a_0*b_1+a_1*b_0+a_1*b_3+a_2*b_2+a_2*b_3+a_3*b_1+a_3*b_2
a_0*b_2+a_1*b_1+a_2*b_0+a_2*b_3+a_3*b_2+a_3*b_3
a_0*b_3+a_1*b_2+a_2*b_1+a_3*b_0+a_3*b_3
gap> mult = mult2sc;
true

```



Example 7.4.2 *Multiplication expressions for $\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$ and \mathbb{F}_{2^8}* \longleftrightarrow

This example is a continuation of the Example 7.3.1. The input to the `FFA_mult_matrixU` method is the “per-level” polynomial basis $B3$, obtained for $\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$. It produces the expressions for the multiplication on the top level of the tower field $\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$. Note that `ChooseFieldElms` in the GAP code Example 7.4.2(a,b) return vectors of length 2, not 8. The multiplications in expressions for the product need a multiplier from the lower level $\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}$. Just as in Example 7.4.1 above, the generalized algorithm (matrix U) produces the same expressions (Example 7.4.2(a)) as the two-step classic multiplication (Example 7.4.2(b)).

The matrix U methods are independent of the type of basis used. Examples so far were showing only polynomial bases (at different levels of the tower field). Example 7.4.2(c) shows the multiplication expressions obtained by using the tower-field basis `TFB1` from Example 7.3.1. Same expressions would be obtained by first expressing the top level multiplier expressions (Example 7.4.2(a)), then replacing every multiplication with expressions obtained for the lower level multiplier, and of course being careful with the variables used, as shown in decomposition in Figure 7.2. Note that new variables for the lower levels would have to be created manually.

Example 7.4.2(a)

```

gap> B3 := GeneratePB(F3, nu); ChooseFieldElms(F3);
Basis( AsField( AsField( GF(2^2), GF(2^4) ), GF(2^8) ), [ Z(2)^0, Z(2^8)^76 ] )

variables
[ "a_0", "a_1" ]
[ "b_0", "b_1" ]
[ "d_0", "d_1", "d_2" ]
gap> multB3 := FFA_mult_matrixU(B3, avec, bvec);
gap> for i in multB3 do Display(i); od;
a_0*b_0+Z(2^4)*a_1*b_1
a_0*b_1+a_1*b_0+Z(2^2)*a_1*b_1
gap> lambda^2*mu; lambda;
Z(2^4)
Z(2^2)

```

Example 7.4.2(b)

```

gap> mult2scB3 := FFA_mult_2stepClassic(f3, avec, bvec, "to");
gap> for i in mult2scB3 do Display(i); od;
a_0*b_0+Z(2^4)*a_1*b_1
a_0*b_1+a_1*b_0+Z(2^2)*a_1*b_1
gap> multB3 = mult2scB3;
true
gap> IR := ReductionMatrixIR(f3);;
gap> for i in IR do Display((i)); od;
[ Z(2)^0, 0*Z(2), Z(2^4) ]
[ 0*Z(2), Z(2)^0, Z(2^2) ]

```

Example 7.4.2(c)

```

gap> ChooseFieldElms(GF(2^8));

variables
[ "a_0", "a_1", "a_2", "a_3", "a_4", "a_5", "a_6", "a_7" ]
[ "b_0", "b_1", "b_2", "b_3", "b_4", "b_5", "b_6", "b_7" ]
[ "d_0", "d_1", "d_2", "d_3", "d_4", "d_5", "d_6", "d_7", "d_8", "d_9", "d_10",
"d_11", "d_12", "d_13", "d_14" ]
gap> TFB1 := GenerateTFBfromEDPLwithPB(edpl);
Basis( GF(2^8), [ Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11, Z(2^8)^76, Z(2^8)^161,
Z(2^8)^178, Z(2^8)^8 ] )
gap> multTFB := FFA_mult_matrixU(TFB1, avec, bvec);;
gap> for i in multTFB do Print(i, "\n"); od;
a_0*b_0+a_1*b_1+a_2*b_2+a_3*b_3+a_4*b_6+a_4*b_7+a_5*b_6+a_6*b_4+a_6*b_5+a_6*b_6+a_7*
b_4+a_7*b_7
a_0*b_1+a_1*b_0+a_1*b_1+a_2*b_3+a_3*b_2+a_3*b_3+a_4*b_6+a_5*b_7+a_6*b_4+a_6*b_7+a_7*
b_5+a_7*b_6+a_7*b_7
a_0*b_2+a_1*b_3+a_2*b_0+a_2*b_3+a_3*b_1+a_3*b_2+a_3*b_3+a_4*b_4+a_4*b_5+a_4*b_6+a_5*
b_4+a_5*b_7+a_6*b_4+a_6*b_6+a_7*b_5+a_7*b_7
a_0*b_3+a_1*b_2+a_1*b_3+a_2*b_1+a_2*b_2+a_2*b_3+a_3*b_0+a_3*b_1+a_3*b_2+a_4*b_4+a_4*
b_7+a_5*b_5+a_5*b_6+a_5*b_7+a_6*b_5+a_6*b_7+a_7*b_4+a_7*b_5+a_7*b_6+a_7*b_7
a_0*b_4+a_1*b_5+a_2*b_6+a_3*b_7+a_4*b_0+a_4*b_5+a_5*b_1+a_5*b_4+a_5*b_5+a_6*b_2+a_6*
b_7+a_7*b_3+a_7*b_6+a_7*b_7
a_0*b_5+a_1*b_4+a_1*b_5+a_2*b_7+a_3*b_6+a_3*b_7+a_4*b_1+a_4*b_4+a_4*b_5+a_5*b_0+a_5*
b_1+a_5*b_4+a_6*b_3+a_6*b_6+a_6*b_7+a_7*b_2+a_7*b_3+a_7*b_6
a_0*b_6+a_1*b_7+a_2*b_4+a_2*b_7+a_3*b_5+a_3*b_6+a_3*b_7+a_4*b_2+a_4*b_7+a_5*b_3+a_5*
b_6+a_5*b_7+a_6*b_0+a_6*b_3+a_6*b_5+a_6*b_6+a_6*b_7+a_7*b_1+a_7*b_2+a_7*b_3+a_7*b_4+
a_7*b_5+a_7*b_6
a_0*b_7+a_1*b_6+a_1*b_7+a_2*b_5+a_2*b_6+a_2*b_7+a_3*b_4+a_3*b_5+a_3*b_6+a_4*b_3+a_4*
b_6+a_4*b_7+a_5*b_2+a_5*b_3+a_5*b_6+a_6*b_1+a_6*b_2+a_6*b_3+a_6*b_4+a_6*b_5+a_6*b_6+
a_7*b_0+a_7*b_1+a_7*b_2+a_7*b_4+a_7*b_7

```



7.5 Summary of key insights

This brief passage is a recap and extension of the Key 7.1 (the only key in Chapter 7). Since the FFCSA package falls into a very broad research area, the status of this package will remain \surd , i.e., partially solved.

The Finite Field Constructions, Search and Algorithms package (FFCSA) deals with field defining polynomials, bases, transition matrices, multiplication matrices, and Hamming weights (complexities). It includes various primitives for exhaustive search. Examples include search for normal elements, for optimal normal bases, and for primitive polynomials for the LFSRs with specified degree, number of taps, and coefficients. There are a variety of well-studied algorithms for implementing finite field arithmetic and it is important to make good decisions early on, as it saves a lot of time and effort. Hence, the FFCSA package is very important in the early design stages of any hardware implementation involving finite field arithmetic. As this is a very broad research area, the package provides only basic functionality but will gradually be extended with more sophisticated algorithms. In terms of subexpression elimination, this package is not intended to compete with or replace synthesis tools.

Chapter 8

Case study: WG and WAGE

8.1	Case study: the WGcipher package	112
8.2	Case study: the WAGE package	116

This chapter presents two customized GAP packages, WGcipher and WAGE. Both rely heavily on package FSR. Chapter 20 describes the design space exploration of WG in more detail, and Part VI describes WAGE in more detail, including the datapath synthesis.

8.1 Case study: the WGcipher package

The WG stream cipher family was introduced in Subsection 3.2.4. This chapter describes the GAP package named WGcipher, which plays a central role for the automated design of this family of stream ciphers. Using the FFCSA package, different representations of the finite field elements can be explored, and suitable LFSRs polynomials can be found quickly. The package is also used for testbenching purposes, i.e., to generate the testvectors for hardware implementations of WG.

<pre>graph TD; alg([alg.]) --> arch([arch.]); arch --> gen([gen.]);</pre>	<p>Overview of the WGcipher package:</p> <p>The basic functionality of this package is the computation of the WG permutation and transformation and its decimated versions, and then using them to filter the output of an LFSR. The WG and the LFSR together form the WG keystream generator WGksGen.</p> <p>The WGcipher is the first example of a cipher represented as a collection of FSR objects. It consists of an LFSR and a FILFUN, used together in two different configurations: the FILFUN output is used as the external input to the LFSR during the key initialization phase, but not during the running phase. This demonstrates the benefits of the regular and external step and run for the FSR objects. This package allows one to specify and run the WG cipher with a few lines of GAP code.</p>	<pre>graph TD; FSR([FSR]) --> FFCSA([FFCSA]); FSR --> WGcipher([WGcipher]); FFCSA --> WGcipher; FFCSA --> WAGE([WAGE]); WGcipher --> WAGE;</pre>
---	---	---

The WGcipher package allows the creation of a WG keystream generator WGksGen, which is built from two FSR objects, namely a FILFUN for the WG permutation and an LFSR. The FILFUN is called WGPfilter, and is encapsulated within another GAP object called WG. Both WG and LFSR are components of the GAP object WGksGen. The rationale for this structure is as follows: once the field size is fixed, so are all parameters of the WG, except for decimation. If the decimation is

changed, the expression for the WG permutation changes as well. The WG key stream generator consists of an LFSR followed by the WG transformation. Changing the LFSR while keeping the WG fixed yields a new instance. The WG cipher as a family is characterized by three parameters: the degree of extension m , the decimation d , and the LFSR polynomial of degree n [50]. The WG keystream generator itself needs another XOR gate to produce the ciphertext/plaintext (recall Figure 3.6 in background Subection 3.2.4).

The GAP object WG with a FILFUN for WG permutation

The WG GAP object is created with a function call `WG`, which needs at least the underlying field \mathbb{F}_{2^m} passed as an argument. From the degree of extension over the prime field, i.e., $m = [\mathbb{F}_{2^m} : \mathbb{F}_2]$, the value k and the exponents for the WG permutation are computed. The constructor will display an error message and return “fail” when an m that is a multiple of 3 is used. Values m , k , and the list of exponents are stored as attributes `WGm`, `WGk`, and `WGexponents`. WG has the components `decimation`, `basis` and `WGPfilter`. The `WGPfilter` component stores the `WGPfilter` object, a FILFUN with the same name, also defined over \mathbb{F}_{2^m} with the filtering function $\text{WGP-}m(X^d)$ as defined in equation (3.18). The decimation value d can be passed as an argument to the WG constructor or it can be changed later with the method `ChangeDecimation`. If the decimation is changed, the WGP filtering function is recomputed, a new FILFUN created and the value of component `WGPfilter` updated. $\text{WGT-}m(X^d)$ is implemented as $\text{WGP-}m(X^d)$ followed by the absolute trace. The methods for the WG object are listed in the first column in Table 8.1.

■ **Implementation detail:** The `WGexponents` and WG permutation $\text{WGP-}m(X^d)$

The exponents are computed following equation (3.17), with an added exponent $r_0 = 1$. Exponent r_0 allows to compute the permutation polynomial as $q(x) = x^{r_0} + x^{r_1} + x^{r_2} + x^{r_3} + x^{r_4}$, where $r_i \in \text{WGexponents}$:

$$\text{WGexponents} = [1, 2^k + 1, 2^{2k} + 2^k + 1, 2^{2k} - 2^k + 1, 2^{2k} + 2^k - 1]$$

The GAP code in Implementation detail 5.4.1 shows the computation of $\text{WGP-}m(X^d)$ from equation (3.18), with decimation d . The GAP function is called `WGPfun`: it returns the expression that is used to create the FILFUN for component `WGPfilter`. The method `StepWGP` (see Table 8.1) does not (re)produce the expression, but evaluates this expression for a given input finite field element.

Implementation detail 5.4.1

```
sz := Size(F);  x_0 := X(F, "x_0");
wgp := Z(2)^0;  y := x_0^d + Z(2)^0;
for j in [1..Length(exponents)] do
    wgp := wgp + y^(exponents[j] mod (sz - 1));
od;
wgp := ReduceMonomialsOverField(F, wgp);
```

■

F	- extension field	m	- degree of extension	n	- LFSR length
K	- prime field	f	- defining polynomial	l	- LFSR polynomial
B	- basis	d	- decimation exponent	p	- characteristic
of	- output file	tap	- LFSR output stage	ks	- key size
wg	- WG object	gen	- WGksGen object	$WGexponents$	- WG exponents
ffe	- finite field element	$ffelist$	- list of ffe	ist	- initial state
$[]$	- optional arguments	nr_steps	- number of steps		

WG	WGksGen
WG(F , [B , d])	WGksGen(wg , l , [tap , ks])
WG(K , f , [B , d])	WGksGen(wg , n , [tap , ks])
WG(p , m , [d])	
ChangeDecimation(wg , d)	LoadWG(gen , ist)
ChangeBasis(wg , B)	KiaWG(gen)
WGPfun(F , $WGexponents$, d)	RunWG(gen , nr_steps)
StepWG \diamond (wg , ffe)	LoadKiaRunWG(gen , ist , nr_steps)
WG \diamond (wg , ffe) †	WriteKiaWG(of , gen)
RunWG \diamond (wg , $ffelist$)	WriteRunWG(of , gen , nr_steps)

Implementation notes:

- \diamond - two methods listed together: the WGP and the WGT
- † - synonyms for StepWG \diamond calls

Table 8.1: Main functionality of the WGcipher package

The keystream generator: the LFSR and the WG

The key stream generator WGksGen encapsulates two GAP objects: the WG and the LFSR. The two mandatory arguments are the WG and either the LFSR polynomial ℓ or the degree of the LFSR polynomial n . Optional parameters include the LFSR output stage tap and the keysize ks in bits. The tap is to be used as the OutputTap attribute of the LFSR (Table 6.4 in Section 6), and is set to the highest stage $n - 1$ if not given. The keysize, with default 80 bits, sets a lower limit on the length n of the LFSR, i.e., on the degree of the LFSR polynomial ℓ , namely:

$$Degree(\ell) \geq \left\lceil \frac{2 * ks}{m} \right\rceil \quad (8.1)$$

where m is the degree of extension over the prime field, i.e., $m = [\mathbb{F}_{2^m} : \mathbb{F}_2]$.

Recall Key 6.1, which stated that a cipher can be represented as a collection of basic modules: LFSRs, NLFSRs and FILFUNs, connected in various configurations. The WGcipher is the first such example: the WGksGen is composed out of two FSR objects: a WG, which includes a FILFUN WGPfilter as a component, and an LFSR. During the running phase (RunWG), the LFSR output is the input to the FILFUN WGPfilter, and the output of WGPfilter is used as the input to the absolute trace to obtain one bit of the keystream. During the key initialization phase (KiaWG),

the configuration is slightly different: the output of `WGPfilter` is used as external input for the LFSR. This demonstrates the benefits of having both a regular and external step and run for the FSR objects (Key 6.3).

The `WGksGen` object `gen` has two components¹: the LFSR `lfsr := gen!.LFSR` and the `wg := gen!.WG`. The `lfsr` generates an m -sequence of \mathbb{F}_{2^m} elements through the stage selected by the tap, for WG cipher usually the highest stage S_{n-1} , which will be referred to simply as the LFSR output `lfsr_seq`. The LFSR output is connected to the `FILFUN` within the `wg`, i.e., it is the primary input for the `WGPfilter`. The `FILFUN` is always used for regular step and run, i.e., there is no external input. `LoadWG` calls the method `LoadFSR`, which simply loads the initial state `ist` into the `lfsr`. This behaviour is no different from the `LoadFSR` behaviour (Table 6.3 in Section 6). Next is the key initialization algorithm (phase) of the WG cipher `KiaWG`. During the initialization phase, the WG permutation is used as a nonlinear filter added to the LFSR feedback: the `wgp` value is obtained with the `StepWGP` on the `FILFUN`, and is then used as the external input for the external step of the LFSR. Specifically, `KiaWG` performs `lfsr_seq = StepFSR (lfsr, wgp)`, followed by `wgp = StepWGP (wg, lfsr_seq)` called $2n$ times, where n is the length of the LFSR. The first `wgp` value is computed on the `lfsr_seq` element after loading. The method `RunWG` is similar to `KiaWG`, with the following differences: (i.) the `lfsr` uses regular step, i.e., `StepFSR (lfsr)`, (ii.) the `wg` uses `StepWGT`, and (iii.) the number of steps is specified as `nr_steps`, and corresponds to the desired number of keystream bits.

■ **Implementation detail:** Because the `lfsr` stores the state after each step, there is no need to pass the finite field element `fsr_seq` from `LoadWG` to `KiaWG` and then to `RunWG`. The `KiaWG` and `RunWG` simply start with the current `lfsr_seq` element before performing the first `StepWGP` and `StepWGT` respectively. ■

Example 8.1.1 The WG7 key stream generator \longleftrightarrow

The initial GAP setup is shown in Example 8.1.1(a). A short example of WG7 key stream generator `wg7`, loaded with an artificial initial state, is shown in Example 8.1.1(b). The `wg` permutation instance output displays the degree of extension $m = 7$ and the decimation $d = 63$. The `wg7` keystream generator displays $m = 7$, $n = 23$, output-tap from the LFSR stage S_{22} , and $d = 63$. The `wg7` is loaded and the key-initialization is run for $2 * n = 46$ steps, after which the detailed state of the keystream generator is printed. Finally, 50 bits of keystream are generated.

Example 8.1.1(a)

```
gap> m := 7;; d := 63;;
gap> K := GF(2);; x := X(K,"x");; y := X(GF(2^m),"y");;
gap> f := x^7+x^6+x^5+x^3+x^2+x+Z(2)^0;;
gap> F := FieldExtension(K, f);; w := RootOfDefiningPolynomial(F);
Z(2^7)^19
gap> VecToString(B, ist);
[ "0000000", "0000001", "0000001", "0000001", "0000001", "0000001", "0000001",
"0000001", "0000001", "0000001", "0000001", "1110001", "1111110", "1111110",
"1111110", "1111110", "1111110", "1111110", "1111110", "1111110", "1111110",
"1111110", "1111110" ]
```

¹`objectname!.componentname` is a way of accessing components in GAP

```

gap> wg := WG(K, f, d);
< WG( 7, 63 ) >
gap> ll := y^23+y^12+y^10+y^9+y^8+y^7+y^6+y^3+y^2+y + w;;
gap> wg7 := WGksGen(wg, ll);
< WG( 7, 23, [ 22 ], 63 ) >
gap> LoadWG(wg7, ist);; KiaWG(wg7);; Print(wg7);
WG( 7, 23, [ 22 ], 63 )
with field polynomial = x^7+x^6+x^5+x^3+x^2+x+Z(2)^0
WG exponents = 1, 2^5 + 1, 2^3 + 2^5 + 1, 2^3 - 2^5 + 1, 2^3 + 2^5 - 1
and LFSR over GF(2^7) given by FeedbackPoly = y^23+y^12+y^10+y^9+y^8+y^7+y^6+y^3+y^2
+y+Z(2^7)^19
with basis =[ Z(2)^0, Z(2^7)^19, Z(2^7)^38, Z(2^7)^57, Z(2^7)^76, Z(2^7)^95, Z(2^7)^
114 ]
with current state =[ Z(2^7)^2, Z(2^7)^122, Z(2^7)^95, Z(2^7)^28, Z(2^7)^71, Z(2^7)^
75, Z(2^7)^64, Z(2^7)^109, Z(2^7)^70, Z(2^7)^41, Z(2^7)^46, Z(2^7)^56, Z(2^7)^71,
Z(2^7)^64, Z(2^7)^106, Z(2^7)^62, Z(2^7)^48, Z(2^7)^66, Z(2^7)^61, Z(2^7)^15, Z(2^7)
^93, 0*Z(2), Z(2^7)^123 ]
after 46 steps
with LFSR output from state S_([ 22 ])
gap> sequence := RunWG(wg7, 50);; VecToString(sequence);
"10100110001010100111111100010100001110011100111111"

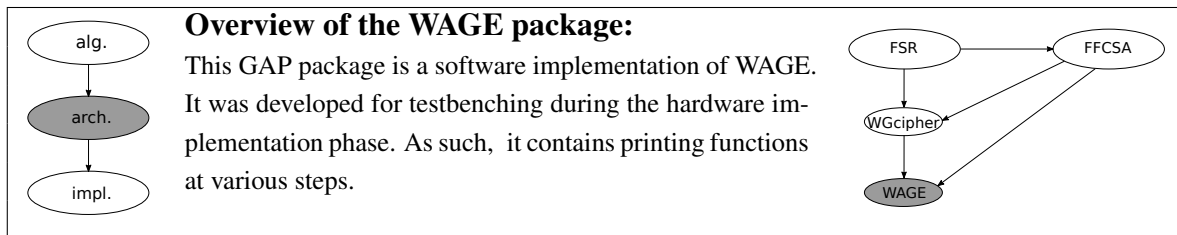
```



8.2 Case study: the WAGE package

Disclaimer 8.1: The WAGE authenticated encryption scheme

Section 8.2 summarizes the GAP package WAGE, used to generate testvectors for the testbenching of WAGE hardware implementations. It is an independent implementation (by the author of the thesis), and was crosschecked with the reference software implementation [6].



WAGE was introduced in Subsection 3.2.8. This section describes the GAP package WAGE, and is structured as follows: first, the modelling of the internal state of WAGE is explained, followed by the LFSR, WGP, the Sbox SB, and generation of round constants. Then the main functionality

of the package is explained and listed in Table 8.2. The implementation details related to the FSR package are highlighted to demonstrate the power of the external step for the FSR objects.

The WAGE LFSR by itself can not be modelled as an FSR object. The XOR gates require it to be broken down into smaller regions. This is explained on a small region of the internal state; the stages $S_8 \rightarrow S_0$, shown in Figure 8.1, reveal the following

- during absorbing:
 - stages $S_7 \rightarrow S_1$ keep their values, i.e., they are not shifted
 - the new input is added to stage S_8
 - the domain separator is added to the stage S_0
- during WAGE permutation:
 - stages $S_8 \rightarrow S_5$ and $S_3 \rightarrow S_0$ are shifted
 - stage S_4 is updated with the sum of the previous S_5 content and the SB output (line 8 in Algorithm 1 in Section 3.2.8)

Regardless of the phase (absorbing or permutation), the internal state of WAGE is fragmented, and cannot be modelled as a single LFSR object with feedback polynomial $\ell(y) = y^{37} + y^{31} + y^{30} + y^{26} + y^{24} + y^{19} + y^{13} + y^{12} + y^8 + y^6 + \omega$, $f(\omega) = 0$ (Table 3.1). Instead, the feedback is modelled as a FILFUN object and the WAGE state is modelled separately as a collection of short LFSRs without a feedback and GAP variables $s\#$, where $\#$ is replaced by the stage number. Using $F = \mathbb{F}_{27}$, the region in Figure 8.1 has the stages $S_8 \rightarrow S_5$ modelled as variable $s8$ followed by $s7_5 := \text{LFSR}(F, y^3)$, $S_4 \rightarrow S_1$ as $s4_1 := \text{LFSR}(F, y^4)$, and the stage S_0 as variable $s0$. The entire WAGE state is modelled as:

```
[s36, s35, s34, s33_30, s29, s28, s27, s26_24, s23_19, s18, s17, s16, s15, s14_11, s10, s9, s8, s7_5, s4_1, s0]
```

and function `WAGE_State()` is used to retrieve the current state as a single array of 37 elements, i.e., the values of the variables and the current state of the individual LFSRs in correct order. The LFSR feedback is modelled as a FILFUN:

```
feedback := FILFUN(F, x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 * w)
```

The WG permutation becomes `dwgpfil := FILFUN(F, wgp)`, where `wgp` is computed as shown in the Implementation detail 5.4.1 GAP code in Section 8.1 for decimation exponent $d = 13$. Finally, there is the small LFSR over $K = \mathbb{F}_2$ for the round constants: `lfsrc := LFSR(K, x^7+x+1, [6, 5, 4, 3, 2, 1, 0])`.

The remainder of the setup takes care of the indices, because GAP enumerates the arrays starting from 1, i.e., the highest stage S_{36} and input D_9 are at index 1, stored as `is36` and `k9`.

The Sbox SB was implemented as functions: `WAGE_Rsb` and `WAGE_Qsb`, which directly follow the specifications in Table 3.1 and are used in function `WAGE_sb` to implement SB. The LFSR

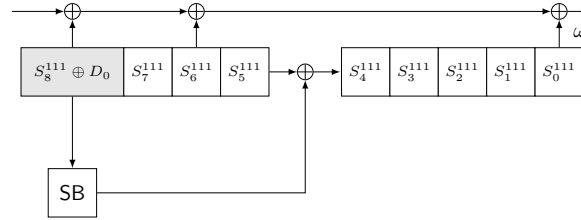


Figure 8.1: Small region of WAGE permutation (from Figure 3.9 in Subsection 3.2.8)

feedback value is computed by `WAGE_Feedback()` function, which retrieves the current state using `WAGE_state()` call to obtain the state values at `is31`, `is30`, `is26`, `is24`, `is19`, `is13`, `is12`, `is8`, `is6`, `is0`, and then evaluates the FILFUN feedback using `LoadStepFSR` (Table 6.3 in Section 6). The loading of the internal state is achieved by two functions: `WAGE_LoadZero()`, followed by the `WAGE_Load(Dk_listlist)`. While all the individual variables and LFSRs can be loaded directly, the aim was to test the hardware implementations, hence the two loading functions simulate the loading by shifting. Function `WAGE_Run`, implementing WAGE permutation, performs 111 iterations of `WAGE_Step(rc1, rc0)`. All functions rely on `StepFSR` with external input; because the individual LFSRs are modelled as shift registers without feedback, the external inputs are stored unmodified.

■ **Implementation detail:** The `WAGE_LoadZero()` call is mandatory because of the FSR package structure: the FSR objects must be loaded before the `StepFSR` method can be used. The `WAGE_Load(Dk_listlist)` function first retrieves the current state `state := WAGE_State()`. The small region of the state, shown in Figure 8.1, is loaded as shown in Implementation detail 7.2.1. The input `Dk_listlist` contains 9 sub-lists, with 10 elements each (for 10 D_k). The WAGE state is namely loaded over 9 clock cycles, and the value t keeps track of the current loading cycle. ■

Implementation detail 7.2.1

```

s0 := state[is1];      # shift s1 -> s0
StepFSR(s4_1, state[is5]); # shift s5 -> s4_1
StepFSR(s7_5, state[is8]); # shift s8 -> s7_5
s8 := Dk_listlist[t][k0]; # LOAD D0 => s8

```

■ **Implementation detail:** The `WAGE_Step` first retrieves the current state `state := WAGE_State()`. The small region of the state, shown in Figure 8.1, is updated as shown in Implementation detail 7.2.2. The external input for `s4_1` is the sum of the previous stage element `state[is5]` and the SB output `in_s4`. The new value for `s8` is the previous stage element `state[is9]`, i.e., the input D_0 is not used. The feedback value is obtained using `WAGE_Feedback` function, and the two WGP values using the external `LoadStepFSR` on the `dwgppfil`. Stage `s36` is updated as shown in Implementation detail 7.2.3. ■

Implementation detail 7.2.2

```

in_s4 := WAGE_sb(Coefficients(B, state[is8]))*BasisVectors(B);
s0 := state[is1]; # shift s1 -> s0
StepFSR(s4_1, state[is5] + in_s4); # shift s5+SB -> s4_1 SB ON
StepFSR(s7_5, state[is8]); # shift s8 -> s7_5
s8 := state[is9]; # shift s9 -> s8 INPUT OFF

```

vec - 7 bit vector
rc1, rc0 - round constants for one round of WAGE permutation
Dk_listlist - initial state for loading (through D_k)
Dk_list, Ok_list - current input and output data (for D_k and O_k)
ds - domain separator
keyhex, noncehex - key and nonce in HEX
adhex, ptxthex, ctxthex - associated data, plaintext and ciphertext in HEX

<code>WAGE_State()</code>	retrieve current internal state
<code>WAGE_sb(<i>vec</i>)</code>	get the SB output for value <i>vec</i>
<code>WAGE_Feedback()</code>	compute current feedback value
<code>WAGE_Load(<i>Dk_listlist</i>)</code>	load WAGE
<code>WAGE_Step(<i>rc1, rc0</i>)</code>	one round of WAGE permutation
<code>WAGE_Run()</code>	WAGE permutation
<code>WAGE_Absorb(<i>Dk_list, ds</i>)</code>	return <i>Ok_list</i> with the sums, absorb
<code>WAGE_Replace(<i>Dk_list, ds</i>)</code>	return <i>Ok_list</i> with the sums, replace
<code>WAGE_TAGextract()</code>	return the tag
<code>WAGE_Enc(<i>keyhex, noncehex, adhex, ptxthex</i>)</code>	encryption
<code>WAGE_Dec(<i>keyhex, noncehex, adhex, ctxthex</i>)</code>	decryption

Table 8.2: Main functionality of the WAGE package

Implementation detail 7.2.3

```

fb := WAGE_Feedback();
in_s36 := LoadStepFSR(dwgpfil, state[is36]) ;
s36 := fb + in_s36 + rc1;
  
```

`WAGE_Absorb` and `WAGE_Replace` compute the sums of the *Dk_list* elements and the corresponding rate elements, and return them as a list *Ok_list*. Both functions also update the corresponding rate stages: with the sums for absorbing, and with data inputs D_k unmodified for replacing. This is the reason for decoupling `s8` and `s7_5`: `s8` must be updated, but `s7_5` does not shift. The FSR package design does not allow to update only a single stage: the shift always happens, hence it is not possible to use `s8_5 := LFSR(F, y^4)`.

The encryption and decryption are performed by `WAGE_Enc` and `WAGE_Dec` respectively. Both use the functions listed in Table 8.2 in proper order. The format of each part of encryption or decryption is always the same: action followed by `WAGE_Run`, whereby the action is either `WAGE_Load`, `WAGE_Absorb` or `WAGE_Replace`, with an appropriate domain separator for initialization, processing of associated data, encryption/decryption or finalization.

The package contains many miscellaneous functions for partitioning the key and nonce for loading, partitioning of data, padding, etc. and functions that write various testvectors.

Part IV

The automated design generation phase

Part IV - Outline

9	The automated design generation phase - the hardware perspective	122
10	GAPtoVHDL package - common VHDL functionality	126
11	FSRtoVHDL package - generating FSR based circuits	136
12	CIRCUIT package: generating arbitrary datapaths	159
13	CIRCUIT package part 1: arbitrary finite fields	163
14	CIRCUIT package part 2: functional description of the algorithm	185
15	CIRCUIT package part 3: VHDL-ready design	202
16	CIRCUIT package part 4: generating VHDL for the datapath	225

Chapter 9

The automated design generation phase - the hardware perspective

The automated design generation, the current phase within the design flow from Figure 1.1, is shown on the right side of Figure 9.1 in a magnifying glass. The simplified design flow diagram in the leftmost column of Figure 9.2(a) and Figure 9.2(b) will be used throughout Part IV as a roadmap with grey areas indicating the package under discussion. Note that Figure 9.2(b) still shows shaded packages FSR and FFCSA, with the objective to stress their importance for the automated design generation. However, the icons will use only the lower part of Figure 9.2(b), showing packages GAPtoVHDL, FSRtoVHDL, and CIRCUIT.

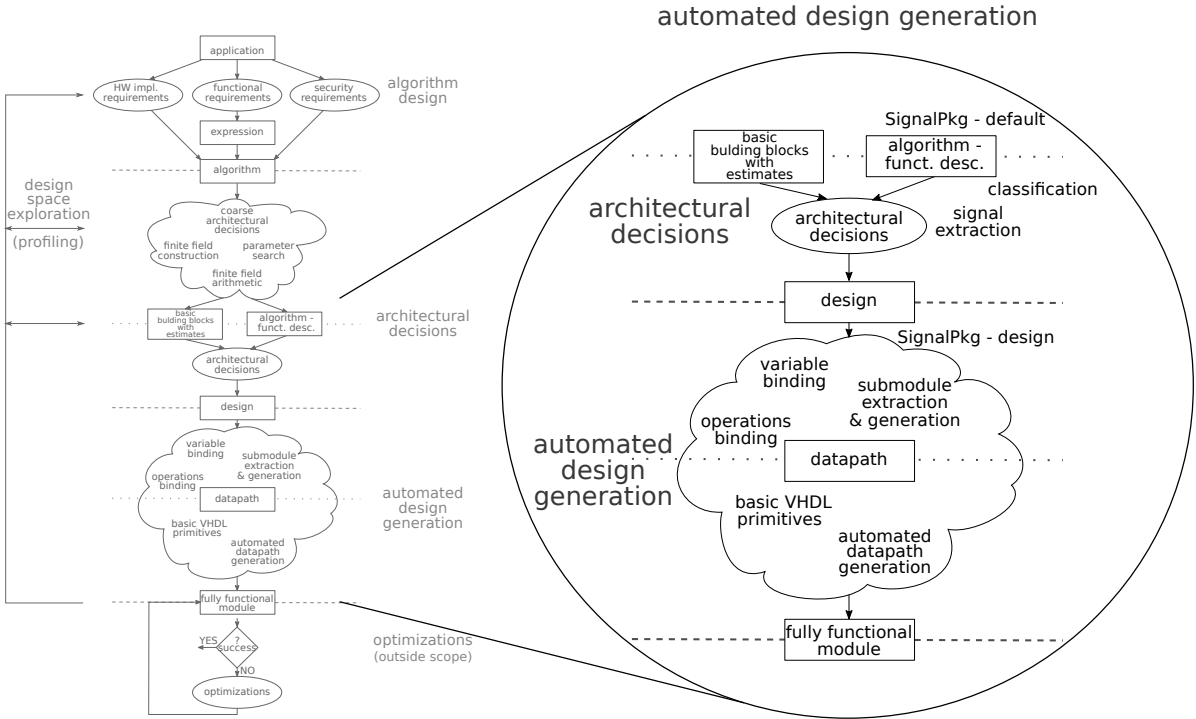


Figure 9.1: Design flow: the automated design generation

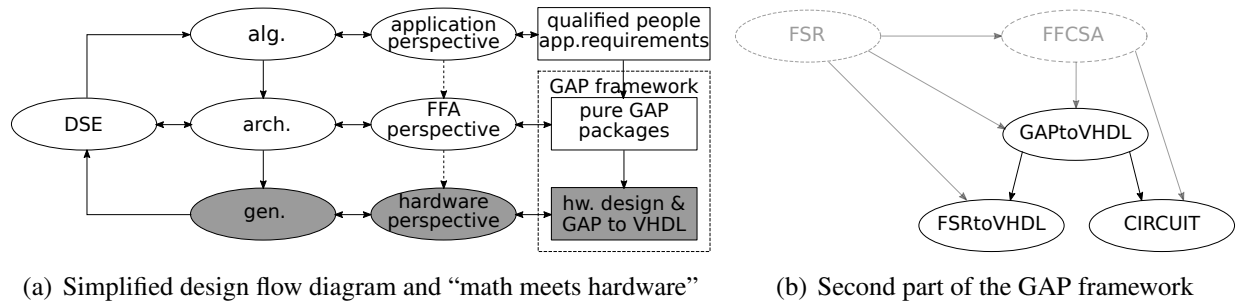


Figure 9.2: *The automated design generation*: (a) the simplified design flow diagram, (b) the second part of the GAP framework (solid lines, not shaded).

While the FSR package was designed for hardware (FSRtoVHDL), this is not the case for the FFCSA package. Adding the information needed for VHDL implementations to the FFCSA package itself would unnecessarily increase its complexity. Another reason to keep the FFCSA package simple is its functionality: not all parts of the package are intended for VHDL implementation. In fact, only the algorithms for the finite field arithmetic are intended for VHDL implementations. To allow VHDL implementations, the CIRCUIT package was developed. It contains an encapsulation for the FFCSA derived expressions that holds all the information needed for translation to VHDL code.

GAPtoVHDL, FSRtoVHDL, and CIRCUIT all use the FSR package. While FSRtoVHDL needs the FSR objects, GAPtoVHDL and CIRCUIT mostly use the FSR miscellaneous functions (Table 6.5). The GAPtoVHDL package contains common VHDL functionality, e.g., functions for writing VHDL packages, combinational statements with AND and XOR gates, component instantiations, etc. Only the CIRCUIT package has access to the FFCSA functionality.

There are two *architectural decisions* – *automated design generation flows* through the framework:

- the first flow focuses on synthesis of ciphers based on feedback shift registers [36]. It consists of GAP packages FSR (Chapter 6) and FSRtoVHDL (Chapter 11), the latter of which relies on package GAPtoVHDL (Chapter 10).
- the second flow enables the synthesis of arbitrary expressions over arbitrary finite fields, realized by the GAP package CIRCUIT (Chapters 13-16). The CIRCUIT package also relies on GAPtoVHDL (Chapter 10).

The magnifying glass shows some intersection of *the automated design generation* and *the architectural decisions*. There is no intersection for the first *architectural decisions* – *automated design generation flow* (FSR and FSRtoVHDL). However, there will be an intersection for the second *architectural decisions* – *automated design generation flow* involving the CIRCUIT package. Chapters 12-16 will explain why the transition is not as strict as shown in the design flow diagram.

Roadmap

Part IV covers three GAP packages: GAPtoVHDL, FSRtoVHDL and CIRCUIT, shown in Figure 9.2(b). Below is the roadmap for packages GAPtoVHDL and FSRtoVHDL. The CIRCUIT package is large and will be explained in many segments; it has its own roadmap in Chapter 12. Table 9.1 lists the examples shown in Chapters 10 and 11, examples moved to the appendix, and related examples. The table rows correspond to individual examples, grouped by the chapters. The columns are structured as follows: the first column gives the subsection in which the example can be found, the second column “Ex.” the example number, the third column the example title and short description if needed, the next two columns indicate whether the example includes GAP or VHDL code, and the last column specifies related examples (Related Ex.). The related examples are the continuation of the example in this row.

The structure of the GAPtoVHDL and FSRtoVHDL half of Part IV is as follows:

- GAPtoVHDL package - common VHDL functionality - Chapter 10
 - Common VHDL functionality (Section 10.1)
 - Binding of GAP variables and VHDL signals (Section 10.2)
 - Classification of expressions defined over finite fields (Section 10.3)
 - Summary and conclusion (Section 10.4)
- FSRtoVHDL package - generating FSR based circuits - Chapter 11
 - Classification of the FSR objects (Section 11.1)
 - Generating VHDL for individual FSR objects (Section 11.2)
 - A cipher as a collection of basic FSR modules (Section 11.3)
 - Summary and conclusion (Section 11.4)

Section	Ex.	Title and keywords	GAP	VHDL	Related Ex.
GAPtoVHDL package - common VHDL functionality - Chapter 10					
Section 10.1 Common VHDL functionality	10.1.1	VHDL type definition and constant declaration example		✓	
	10.1.2	Use of GATE_line and XOR_line methods	✓		
Section 10.2 Binding	10.2.1	Polynomials defined over \mathbb{F}_2: bin_monomial and bin_polynomial: with and without binding	✓		
Section 10.3 Classification	10.3.1	Classification of arbitrary expressions defined over \mathbb{F}_2 and \mathbb{F}_{2^s}	✓		11.1.1
FSRtoVHDL package - generating FSR based circuits - Chapter 11					
Section 11.1 Classification	11.1.1	Classification of FSRs defined over \mathbb{F}_2 and \mathbb{F}_{2^s}	✓		
Section 11.2 The packages	11.2.1	A simple LFSR–VHDL packages field_pkg and fsr_pkg: Ex. 6.2.2 continued		✓	11.2.2
Section 11.2 The entity	11.2.2	A simple LFSR – schematic and entity: Ex. 11.2.1 continued		✓	11.2.3
Section 11.2 The architecture	11.2.3	A simple LFSR – the datapath: Ex. 11.2.2 continued		✓	
	11.2.4	Grain NLFSR – the datapath		✓	11.3.1
	11.2.5	The WG7 constant array implementation Ex. 8.1.1 continued	✓	✓	
Section 11.3 Cipher as a collection of FSRs	11.3.1	Grain modelled as a collection of FSRs: modelling, schematic, and external step conditions			11.3.2 11.3.3 11.3.4
	11.3.2	Grain setup file: user input, Ex 11.3.1 continued	✓		11.3.5
	11.3.3	Grain – the filled-out spreadsheet template: user input, Ex. 11.3.1 continued			
	11.3.4	Grain spreadsheet template - continued: Manager, multiplexers, and select connector	✓		
Section 11.3 The datapath	11.3.5	Grain – top-level datapath: VHDL for the top-level module, Ex 11.3.1 cont.		✓	D.1.1
Appendix D.1	D.1.1	Grain – top-level datapath: full example package dp_pkg.vhd, the entity, the architecture		✓	

Table 9.1: Examples in Chapters 10 and 11

Chapter 10

GAPtoVHDL package - common VHDL functionality

10.1 Common VHDL functionality	126
10.2 Binding of GAP variables and VHDL signals	130
10.3 Classification of expressions defined over finite fields	132
10.4 Summary and conclusion	135

	<p>Overview of the GAPtoVHDL package:</p> <p>The GAPtoVHDL package can write constant array implementations and testvectors to verify their functional correctness. Furthermore, it generates packages with the VHDL type definitions and declarations of (some) constants. The data types are named <code>ffe_Fid</code>, with <i>field identifier</i> <code>Fid</code>, a positive integer that enumerates the finite fields in the design. By providing common VHDL functionality, the GAPtoVHDL package is the backbone of <i>automated generation</i>. It can write assignments, component instantiations and registers. The package also allows binding of GAP variables and VHDL signal names, which provides (more) flexibility for concurrent assignments. Conceptually, the most important is classification of the ANF expressions, which is the base for datapath synthesis.</p>	
--	---	--

10.1 Common VHDL functionality

The functionality of the GAPtoVHDL package is captured in several source files, e.g., *clerk* and *packages*, shown in Table 10.1, and *modules* and *testbenches* for constant array implementations, shown in Table 10.2. Conceptually, the most important part of the GAPtoVHDL package is classification of mathematical expressions, which is the base for the automated design entry of datapaths, explained in Section 10.3.

The methods in *clerk* are tasked with creating VHDL files, writing use clauses to include appropriate libraries and packages, and retrieving the strings used for custom VHDL data types. The data types are written into a `field_pkg.vhd` file by the `Write_field_pkg` function (Table 10.1). The ports and signals are always declared as a `ffe_Fid` type, where `ffe` stands for finite field

element, and Fid is a natural number that serves as *field identifier*; the finite fields in the design are simply enumerated. The VHDL Example 6.1.1 shows a fraction of a generated VHDL package containing a type definition for elements of the finite field \mathbb{F}_{2^2} , written by the function `Write_field_ext_pkg`. Note that the field identifier Fid is chosen independently of the degree of extension m . The keyword `to`, already mentioned in connection to bases in the FFCSA package (Section 7), is needed to define the range of the new type, and the basis itself to define the constants. The VHDL package writing functions are listed in Table 10.1.

Key 10.1: VHDL type definition and constant declaration

GAPtoVHDL writes a `field_pkg.vhd` package including the VHDL type definitions and (some) constant declarations. The data types are named `ffe_Fid`, with *field identifier* Fid , a positive integer that enumerates the finite fields in the design independently of their degree of extension.

Example 10.1.1 *VHDL type definition and constant declaration example*

VHDL Example 6.1.1 shows a type definition for the elements of \mathbb{F}_{2^2} with Fid 2, written by `Write_field_ext_pkg`. The constants `ffe_2_one` and `ffe_2_zero` are written w.r.t. the polynomial basis of \mathbb{F}_{2^2} .

— VHDL Example 10.1.1 —

```

constant ffe_2_dim : natural := 2;
subtype ffe_2 is std_logic_vector(0 to ffe_2_dim - 1);

--- element 1 in chosen basis:
constant ffe_2_one: ffe_2 := "10";
--- element 0 in chosen basis:
constant ffe_2_zero: ffe_2 := "00";

```

<i>clerk</i>	<i>packages</i>
Get_ffe_strings	Write_field_poly
Open_VHDL_file	Write_field_bin_pkg
Close_VHDL_file	Write_field_ext_pkg
Write_VHDL_comments	Write_vect_space_pkg
Write_VHDL_use	Write_function_pkg
	Write_field_pkg

Table 10.1: Main functionality of the GAPtoVHDL package

<i>modules</i>	<i>testbenches</i>
Write_arch_const_array_Permutation	Write_tb_linput_Permutation
Write_arch_const_array_Functional	Write_tb_linput_Functional

Table 10.2: Main functionality of the GAPtoVHDL package - continued

For writing the constant arrays modules, only two functions are needed (Table 10.2):

- `Write_arch_const_array_Permutation` for all expressions mapping $\mathbb{F}_{2^m} \rightarrow \mathbb{F}_{2^m}$
- `Write_arch_const_array_Functional` for all expressions mapping $\mathbb{F}_{2^m} \rightarrow \mathbb{F}_2$

The only difference between the two functions is in the VHDL formatting: VHDL uses the delimiters " for `std_logic_vector` and ' for `std_logic` data types. `Write_arch_const_array_Permutation` is used for inversion, exponentiation, or any other composed expression mapping $\mathbb{F}_{2^m} \rightarrow \mathbb{F}_{2^m}$, such as the WG permutation (Subsection 3.2.4). `Write_arch_const_array_Functional` is used for absolute trace computations and any other functions mapping $\mathbb{F}_{2^m} \rightarrow \mathbb{F}_2$, such as the WG transformation (Subsection 3.2.4). In both cases, the expression implemented by the constant array is a function in one variable. The module produced has a single input and a single output port. It can be extended to more than one input, and to prevent a drastic increase in the array size and hence its hardware area, a bit-width upper limit, which triggers an error when exceeded, is set. The basis chosen for the representation of elements plays an important role for the constant array implementations: the arrays, i.e., lookup tables, are accessed with input bits used as a memory address. The testbench functions are implemented in a similar way. The test-vectors created are written into two files, one with inputs (stimulus) and one with expected outputs (specifications).

■ **Implementation detail:** Using only two functions to generate such a wide variety of modules is only possible because GAP has an `EvalString` (Section 27.9-5 in the GAP reference manual [34]). The expression to be evaluated is simply passed to `Write_arch_const_array_*` function as a string, representing a valid GAP expression. Only other restriction is the use of a global variable `gffe` for the element. To write the entire array, `gffe` takes the values of all field elements. ■

The source file `VHDLcommon` contains concurrent VHDL statements, combinational assignments, component instantiations, registers, multiplexers, etc. The methods in the upper half of Table 10.3 produce *lines*, which are appropriately formatted strings that appear as the r.h.s. of VHDL assignments. Actual assignments are written with the function `Write_line`.

To reduce the number of different functions, the assignments take a *target* and *line*, where the *target* is a string for the target signal and the *line* the appropriately formatted r.h.s. string. The *line* is obtained using one or a combination of several methods, e.g., methods that insert logic gates between signals or methods that write multiplexers.

■ **Implementation detail:** Most of its functionality is implemented as GAP methods: this reduces the number of writing functions and gives the advantage of GAP method selection, which allows method overloading. As per GAP convention, functions are to be used for writing to files, hence the *Write* functionality is implemented as GAP functions. ■

<i>source</i>	- name of source signal (string)	<i>srclist</i>	- list of <i>sources</i>
<i>line</i>	- r.h.s. of an VHDL assignment (string)	<i>linelist</i>	- list of <i>lines</i>
<i>target</i>	- name of target signal (string)	<i>trgtlist</i>	- list of <i>targets</i>
<i>gate</i>	- gate (string), allowed gates: and, or, xor	<i>idxlist</i>	- list of integers (indices)
<i>sellist</i>	- names of select signals (list of strings)	<i>selvals</i>	- list of select values
<i>delimiter</i>	- VHDL delimiters " and '	<i>f</i>	- monomial/polynomial
<i>bindlist</i>	- a list of bindings	<i>top</i>	- top switch (integer 0 or 1)
<i>clabel</i>	- unique component label (string)	<i>rlabel</i>	- unique register label (string)
<i>entlist</i>	- list: entity name, architecture name	<i>portmap</i>	- list of VHDL signals
<i>cntllist</i>	- names of register control signals (strings)	<i>cntlvals</i>	- list of control values
<i>resetlist</i>	- names of reset source signals or constants (strings)		
[]	- optional parameters	<i>of</i>	- output file name

VHDLcommon	
method/function	possible arguments
GATE_line	(<i>source</i> , <i>idxlist</i> , <i>gate</i>), (<i>srclist</i> , <i>idxlist</i> , <i>gate</i>), (<i>linelist</i> , <i>gate</i>)
XOR_line, AND_line	(<i>source</i> , <i>idxlist</i>), (<i>srclist</i> , <i>idxlist</i>)
MUX_line	(<i>srclist</i> , <i>sellist</i> , <i>selvals</i> , <i>delimiter</i>) ^[1,2]
bin_monomial	(<i>f</i> , [<i>bindlist</i> ,] <i>top</i>) ^[3,4,5]
bin_polynomial	
Write_line	(<i>of</i> , <i>target</i> , <i>line</i>) ^[6]
Write_component_inst	(<i>of</i> , <i>clabel</i> , <i>entlist</i> , <i>portmap</i>) ^[7]
Write_registers	(<i>of</i> , <i>rlabel</i> , <i>cntllist</i> , <i>cntlvals</i> , <i>trgtlist</i> , <i>srclist</i> , <i>resetlist</i>) ^[8,9]

Implementation notes:

- [1] - requirements: $\text{Length}(\textit{sellist}) = \text{Length}(\textit{selvals}) = n$, $\text{Length}(\textit{srclist}) = n + 1$
VHDL format: *srclist*_{*i*} when *sellist*_{*i*} = *selvals*_{*i*} ... else *srclist*_{*n+1*}, *i* = 1, ..., *n*
- [2] - VHDL uses delimiters " for std_logic_vector and ' for std_logic
- [3] - polynomial *f* requirement: *f* must have a zero constant term
- [4] - *bindlist* requirements: for each GAP variable, there must be a pair of strings in the form [GAP_variable, VHDL_signal], see Example 10.2.1
- [5] - *top* requirement: add the "i_" prefix if VHDL signals refer to the input ports
- [6] VHDL format: *target* <= *line* ;
- [7] - *portmap* requirement: use positional mapping
- [8] - requirements: $\text{Length}(\textit{cntllist}) = \text{Length}(\textit{cntlvals}) = 3$
cntllist format: [*clk_name*, *reset_name*, *chip_enable_name*]
*cntllist*₁ always interpreted as clock signal, *cntllist*₂ as reset signal, *cntllist*₃ as chip enable
only clock signal is mandatory, e.g., ["clk", 0, 0] - no reset, no chip enable
*cntlvals*₁ format: if *cntlvals*₁ = 1 VHDL format is rising_edge(*clk*), else falling_edge(*clk*)
*cntlvals*_{*i*} format for *i* = 2, 3 VHDL format: if *cntllist*_{*i*} = *cntlvals*_{*i*} then
- [9] - requirements: $\text{Length}(\textit{trgtlist}) = \text{Length}(\textit{srclist}) = \text{Length}(\textit{resetlist})$
VHDL format: *trgtlist*_{*i*} <= *srclist*_{*i*} and *trgtlist*_{*i*} <= *resetlist*_{*i*}

Table 10.3: Main functionality of the GAPtoVHDL package - continued

Key 10.2: Common VHDL primitives

GAPtoVHDL package is the backbone of *the automated design generation*. It provides common VHDL functionality: it can write assignments, component instantiations, and registers. To reduce the number of different functions, the assignments take a *target* and a *line*, where the *target* is a string for the target signal and the *line* the appropriately formatted r.h.s. string. The *line* is obtained using one or a combination of several methods, e.g., methods that insert logic gates between signals or methods that write multiplexers.

Example 10.1.2 Use of GATE_line and XOR_line methods \longleftrightarrow

Example 10.1.2 shows the use of method GATE_line using all three sets of arguments from Table 10.3. Last GATE_line call demonstrates the flexibility of this method. Further examples are shown in Example 10.2.1.

Example 10.1.2

```
gap> line1 := GATE_line(["i_a", "i_a", "i_b"], [0,1,3], "and");
" i_a(0) and i_a(1) and i_b(3) "
gap> line2 := GATE_line("i_a", [0,1,3], "xor");
" i_a(0) xor i_a(1) xor i_a(3) "
gap> line3 := XOR_line(["a", "a", "b"], [0,1,3]);
" a(0) xor a(1) xor b(3) "
gap> line4 := GATE_line([line2, line3], "and");
" ( i_a(0) xor i_a(1) xor i_a(3) ) and ( a(0) xor a(1) xor b(3) ) "
```



10.2 Binding of GAP variables and VHDL signals

The FSR package automatically creates the GAP variables x_i to be used for entering multivariate polynomials for NLFSR feedbacks or for FILFUNs. Variable creation is hidden from the user, but at the time of the FSR package loading the following message is displayed: “You can now use 200 variables $x_0 \dots x_{199}$ ”. The variables s_i , intended for symbolic computation (Example 6.2.4), are created at the same time. The FFCSA method ChooseFieldElms, listed in Table 7.5, also creates the GAP variables a_i , b_i , and d_i .

Each GAP variable used so far was created using the GAP operation Indeterminate, or rather its synonym X (Section 66.1 in the GAP reference manual [34]). When a GAP variable is created using the *Indeterminate* call, it is assigned a nonzero integer. This integer is used for lexicographic ordering of variables from smallest to largest. For details on monomial ordering, see Section 66.17 in the GAP reference manual [34]. The FSR package variables x_i and s_i start at 1000 and 2000, respectively. The FFCSA variables a_i , b_i , and d_i at 3000, 3500, 4000, respectively.

These integers can be easily adjusted if needed. Consequently, if all those variables were used in the same monomial, the x_i would be ordered before a_i .

To create GAP variables with the same name as VHDL signals is not only tedious but also unfriendly for the user and prone to errors. With each new call of `ChooseFieldElms`, the previously created variables are removed and new variables are created. This is important because they are used for FFCSA algorithms and vectors *avec* and *bvec* must have proper length that corresponds to the length of the basis used for representation of elements, recall Examples 7.4.1 and 7.4.2. A variable binding is a pair of strings in the form `[GAP_variable, VHDL_signal]`. Use of variable binding is shown in Example 10.2.1. This is the first example of binding in the automation framework.

Example 10.2.1 *Polynomials defined over \mathbb{F}_2 : bin_monomial and bin_polynomial* \longleftrightarrow

Methods `bin_monomial` and `bin_polynomial` (Table 10.3) that transform a given polynomial, defined over \mathbb{F}_2 , into a line of VHDL code containing AND and XOR gates. The VHDL signal name used for the `bin_polynomial` call is *feedback*, hence three bindings are needed: a binding rule must be present for *each* GAP variable x_0 , x_1 and x_2 . In this case, *feedback* is an internal signal, hence the `top` switch is set to 0. The `bin_monomial` call for *line2* is used with `top` switch set to 1, which appends *i_* to every signal name, assuming that the signals are also the VHDL modules input ports. Because no binding is given, the VHDL signal names correspond to the GAP variable names. Last two calls in Example 10.2.1 show additional use of `GATE_line` method and the usage of `MUX_line` method (Table 10.3).

Example 10.2.1

```
gap> f1 := x_0*x_1*x_2 + x_0*x_1;;
gap> bindrule := [{"x", "feedback"}, {"x", "feedback"}, {"x", "feedback"}];;
gap> line1 := bin_polynomial(f1, bindrule, 0);
" ( feedback(0) and feedback(1) and feedback(2) ) xor ( feedback(0) and
feedback(1) ) "
gap> f2 := a_0*b_1;;
gap> line2 := bin_monomial(f2, 1);
" i_a(0) and i_b(1) "
gap> line3 := GATE_line([line1, line2], "xor");
" ( ( feedback(0) and feedback(1) and feedback(2) ) xor ( feedback(0) and
feedback(1) ) ) xor ( i_a(0) and i_b(1) ) "
gap> line4 := MUX_line([line1, line2], ["i_sel"], ["0"], "\'");
" ( ( feedback(0) and feedback(1) and feedback(2) ) xor ( feedback(0) and
feedback(1) ) ) when i_sel = '0' else ( i_a(0) and i_b(1) )"

```



Key 10.3: Binding of GAP variables and VHDL signal names

A variable binding is a pair of strings in the form

`[GAP_variable, VHDL_signal]`

10.3 Classification of expressions defined over finite fields

An expression is a set of instructions on how to compute a result, and hence also a set of instructions on how to generate the datapath. An expression consists of variables and arithmetic operations, while a datapath consists of signals and gates or components implementing corresponding operations. The classification of expressions is needed for systematic construction of datapaths.

All three packages (GAPtoVHDL, FSRtoVHDL, CIRCUIT) use the FSR package, especially the FSR miscellaneous functions listed in Table 6.5: `SplitCoeffsAndMonomials` and `DegreeOfPolynomialOverField` are the core methods for automated design entry. All the expressions that can currently be converted to VHDL are given in the algebraic normal form (ANF). Recall equation (3.9), with non-zero coefficients $c_{i_0, i_1, \dots, i_{t-1}} \in \mathcal{F} \cong \mathbb{F}_q$, $q \geq 2$:

$$f : \mathcal{F}^t \rightarrow \mathcal{F}$$

$$f(x_0, x_1, \dots, x_{t-1}) = \sum_{\substack{\forall (i_0, i_1, \dots, i_{t-1}) \in \mathbb{Z}_q^t \\ c_{i_0, i_1, \dots, i_{t-1}} \neq 0}} c_{i_0, i_1, \dots, i_{t-1}} x_0^{i_0} x_1^{i_1} \dots x_{t-1}^{i_{t-1}} \quad (10.1)$$

where $i_j \in \mathbb{Z}_q$ for $0 \leq j < t$. The expression (10.1) is defined over \mathcal{F}^t with the variables x_j , $0 \leq j < t$, defined over \mathcal{F} . As was mentioned in Section 6, the terms in equation (10.1) are ordered by the degree of monomials (recall equation (3.10)), and the variables within a monomial by their number, not by their exponent. Furthermore, for every monomial, if an exponent $i_j = 0$, the variable x_j is not present in this monomial. The generalization of Fermats little theorem, $\forall x_j : x_j^q = x_j$ (equation (3.2)) implies that the maximum value of an exponent for any variable in (10.1) is $q - 2$. This is also the exponent for the finite field inversion, i.e., $x_j^{q-2} = x_j^{-1}$, however, inversion is treated as exponentiation to a fixed exponent. The ANF expressions in equation (10.1) are expected to have the exponents i_j reduced modulo $q - 1$, i.e., $\forall i_j : 0 \leq i_j \leq q - 2$; if in doubt, the FSR method `ReduceMonomialsOverField`¹ can be used.

The FSR package miscellaneous methods `SplitCoeffsAndMonomials` and `DegreeOfPolynomialOverField` (Table 6.5), and GAP method `LeadingMonomial` (Section 66.6-4 in the GAP reference manual [34]) are imperative for the classification of the ANF expressions. Method `LeadingMonomial` provides an easy access to the exponents i_j .

The classification, shown in Table 10.4, is based on the following conditions (columns 1-5):

- the finite field \mathbb{F}_q , $q \geq 2$, over which the polynomial f is defined
- the presence of constants from the extension field, i.e., coefficients $c_{i_0, i_1, \dots, i_{t-1}} \in \mathbb{F}_q \setminus \mathbb{F}_2$ for $q > 2$, *excluding* the constant term
- the presence of a non-zero constant term, denoted ct , i.e., $ct \neq 0$
- the degree of polynomial f , i.e., $\deg(f)$

¹ several FFCSA methods for manipulating expressions symbolically use `ReduceMonomialsOverField`

- the maximum exponent i_j in f , where the maximum is taken over *all* terms in f , denoted $\max\{i_j\}$

On the right-most side of Table 10.4 is a column with arithmetic operations that (can) occur in each class: addition \oplus (bit-wise XOR), multiplication with a constant $\times\gamma$, multiplication M, and exponentiation E.

conditions					classification		finite field arithmetic			
\mathbb{F}_q	$c_{i_0, i_1, \dots, i_{t-1}} \in \mathbb{F}_q \setminus \mathbb{F}_2$	$ct \neq 0$	$\deg(f)$	$\max\{i_j\}$	class	examples	\oplus	$\times\gamma$	M	E
–	–	✓	0	0	0	any γ , element 0	×	×	×	×
$q = 2$	×	–	1	1	1	$x_0 + x_1, x_0 + x_1 + 1, \dots$	✓	×	×	×
$q > 2$	✓	–	1	1	2	$x_0 + x_1 + \gamma, x_0 + \gamma_1 x_1 + \gamma_2, \dots$	✓	✓	×	×
$q = 2$	×	–	> 1	1	3	$x_0 x_1 x_2, x_0 x_1 + x_2 + 1, \dots$	✓	×	✓ _A	×
$q > 2$	×	–	> 1	1	4	$x_0 x_1 x_2, x_0 x_1 + x_2 + 1, \dots$	✓	×	✓	×
	✓	–			5	$\gamma_1 x_0 x_1 x_2, x_0 x_1 + \gamma_1 x_2 + \gamma_2, \dots$	✓	✓	✓	×
	×	–		6	$x_0 + x_0^3, x_0 x_1^{2m-2} + x_0^2 x_1 x_2, \dots$	✓	×	✓	✓	
	✓	–		> 1	7	$\gamma_1 x_0, x_1 x_2 + x_0^2 + \gamma_2, \dots$	✓	✓	✓	✓
– ... dont care					$\gamma, \gamma_2 \in \mathbb{F}_q, \gamma_1 \in \mathbb{F}_q \setminus \mathbb{F}_2, q > 2$		✓ _A ... AND			

Table 10.4: Classification of expressions in algebraic normal form given by the equation (10.1)

Classification is implemented by the method `Which_class(expression, F, ca, bitwidth)`. F is the finite field \mathcal{F} , over which *expression* is defined. The last two input parameters are related to the constant array implementations: if $ca=1$ and the total number of input bits is smaller than the given *bitwidth*, classify as [8, using const_array]. The total number of input bits is computed as (number of variables) \times (degree of extension over the prime field [$\mathcal{F} : \mathbb{F}_2$]).

■ **Implementation detail:** Method `SplitCoeffsAndMonomials` returns two lists of same length, a list of coefficients *clist* and a list of monomials *mlist*. `LeadingMonomial` returns a list *lm*, which is formatted as [`var_` j_1 , i_{j_1} , `var_` j_2 , i_{j_2} , ...], where $j_1 < j_2$; each variable with a positive exponent has two entries, the variable number `var_` j , followed by its exponent i_j . These three lists are used to extract information about the arithmetic operations:

- (\oplus) *addition* will occur when `Length(clist) = Length(mlist) > 1`, i.e., #terms > 1 .
- ($\times\gamma$) *multiplication with a constant* is detected by finding all coefficients in *clist*, that are different from 1. Special care is needed for the last element in *clist*, which can be either a multiplicative or an additive constant:
 - if last monomial in *mlist* is 1, the last element in *clist* is a constant term, i.e., additive constant ($+\gamma$)
 - otherwise, the last element in *clist* is a coefficient of a monomial, i.e., multiplicative constant ($\times\gamma$)
- (M) *multiplication* is needed when `Length(lm) > 2`, i.e., the list returned by `LeadingMonomial` has more than two entries, indicating that the #variables within monomial > 1 (must be checked for every monomial).
- (E) *exponentiation* is needed for every $i_j > 1$ entry in *lm* list returned by `LeadingMonomial` (must be checked for every variable within every monomial).

As a final note: for classification, there is no need to know how many exponentiations are in the expression, nor where they are, it is enough to find the first $i_j > 1$. ■

Key 10.4: Classification of expressions defined over finite fields

The classification of expressions is a basis for the inference of internal signals, extraction and generation of submodules, and for the automated generation of a datapath. The classification is based on the finite field over which the expression is defined.

Example 10.3.1 *Classification of arbitrary expressions* \longleftrightarrow

The Example 10.3.1 shows different classification examples over the prime field \mathbb{F}_2 and over extension field \mathbb{F}_{2^8} , including the class 8 for the constant array implementations. The example expressions are taken from Table 10.4.

Example 10.3.1

```

gap> K := GF(2);; F := GF(2^8);;
gap> random := Z(2^8)^3;;
gap> Which_class(random, F, 0, 0);
[ 0, "constant in ext field" ]
gap> random := a_0 + b_0;;
gap> Which_class(random, K, 0, 0); Which_class(random, F, 0, 0);
[ 1, "linear over binary field" ]
[ 1, "linear over binary ext field, all coeffs from bin field" ]
gap> random := a_0 + a_1 + Z(2^8)^3;; Which_class(random, F, 0, 0);
[ 2, "linear with coeffs from binary ext field" ]
gap> random := x_0*x_1 + x_2 + 1;;
gap> Which_class(random, K, 0, 0); Which_class(random, F, 0, 0);
[ 3, "non-linear over binary field" ]
[ 4, "non-linear over binary ext field, all coeffs from bin field" ]
gap> random := Z(2^8)^3*x_0*x_1;; Which_class(random, F, 0, 0);
[ 5, "non-linear with coeffs from binary ext field" ]
gap> random := a_0*a_1*a_2+a_0^2;; Which_class(random, F, 0, 0);
[ 6, "non-linear over binary ext field, all coeffs from bin field" ]
gap> random := Z(2^8)*a_0*a_1*a_2+a_0^2+Z(2^8)^3;; Which_class(random, F, 0, 16);
[ 7, "non-linear with coeffs from binary ext field" ]
gap> random := Z(2^8)*a_0*a_1*a_2+a_0^2+Z(2^8)^3;; Which_class(random, F, 1, 16);
NOTE: cant do const_array because bitwidth > 16 fail
gap> random := Z(2^8)*a_0*a_1*a_2+a_0^2+Z(2^8)^3;; Which_class(random, F, 1, 32);
[ 8, "non-linear using const_array" ]

```

\longleftarrow

10.4 Summary and conclusion

Table 10.5 summarized the key insights highlighted Chapter 10. GAPtoVHDL is the backbone of *the automated design generation* (Key 10.1). Keys 10.1 and 10.4 introduce two notions, that will be encountered throughout Part IV: the *field identifier* `Fid`, and the classification of expressions. The field identifier serves as abstraction that simplifies the datapath generation. Classification is very important to synthesis and will be used for the inference of internal signals, extraction and generation of submodules, and for the automated generation of a datapath.

Key 10.1: VHDL type definition and constant declaration	✓✓	Section
GAPtoVHDL writes a <code>field_pkg.vhd</code> package including the VHDL type definitions and (some) constant declarations. The data types are named <code>ffe_Fid</code> , with <i>field identifier</i> <code>Fid</code> , a positive integer that enumerates the finite fields in the design independently of their degree of extension.		10.1
Key 10.2: Common VHDL primitives	✓✓	Section
GAPtoVHDL package is the backbone of <i>the automated design generation</i> . It provides common VHDL functionality: it can write assignments, component instantiations, and registers. To reduce the number of different functions, the assignments take a <i>target</i> and a <i>line</i> , where the <i>target</i> is a string for the target signal and <i>line</i> the appropriately formatted r.h.s. string. The <i>line</i> is obtained using one or a combination of several methods, e.g., methods that insert logic gates between signals or methods that write multiplexers.		10.1
Key 10.3: Binding of GAP variables and VHDL signal names	✓✓	Section
A variable binding is a pair of strings in the form <code>[GAP_variable, VHDL_signal]</code>		10.2
Key 10.4: Classification of expressions defined over finite fields	✓✓	Section
The classification of expressions is a basis for the inference of internal signals, extraction and generation of submodules, and for the automated generation of a datapath. The classification is done based on the finite field over which the expression is defined.		10.3

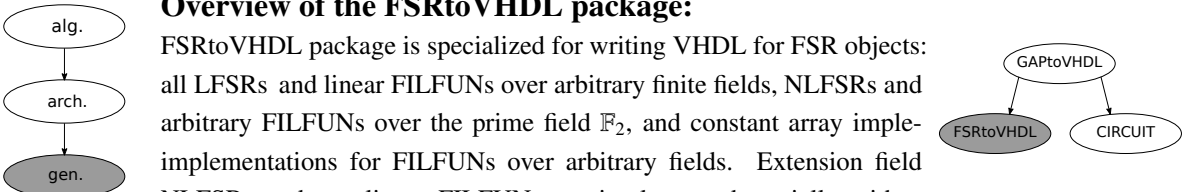
Notes: ✓✓ - solved

Table 10.5: Summary of key insights to the GATtoVHDL package - listed chronologically

Chapter 11

FSRtoVHDL package - generating FSR based circuits

11.1 Classification of the FSR objects	136
11.2 Generating VHDL for individual FSR objects	138
11.3 A cipher as a collection of basic FSR modules	148
11.4 Summary and conclusion	157



Overview of the FSRtoVHDL package:

FSRtoVHDL package is specialized for writing VHDL for FSR objects: all LFSRs and linear FILFUNs over arbitrary finite fields, NLFSRs and arbitrary FILFUNs over the prime field \mathbb{F}_2 , and constant array implementations for FILFUNs over arbitrary fields. Extension field NLFSRs and non-linear FILFUNs are implemented partially with a black-box component instantiation. The black-box submodule can be implemented using the CIRCUIT package. The FSRtoVHDL package also provides mechanisms for implementing ciphers and other FSR-based systems, by modelling the system as a collection of FSRs. This chapter shows how hardware design principles influenced the design of the FSR package and highlights the similarities and differences between LFSRs, NLFSRs, and FILFUNs.

11.1 Classification of the FSR objects

In Section 10.3, the classification of expressions in algebraic normal form, given by the equation (10.1), was explained and shown in Table 10.4. This classification is revisited for FSR objects and presented in Table 11.1, with the following modifications:

- a new column for the type of FSR object: LFSRs are only possible for class 1 and 2 datapaths (upper half of Table 11.1) and NLFSRs are always classified above 3 (lower half of Table 11.1). FILFUNs can be classified into any of classes 1-7; when $deg(f) = 1$, the FILFUN is linear and classified as 1 or 2 (upper half of Table 11.1).
- class 0 is skipped: a constant function can be used for LFSR feedback, e.g., $ct = 1$ yields a shift register. Class 0 is not possible for the NLFSR objects and is not used for FILFUNs; if a constant is ever needed, it can be modelled as the external step and run input.

- because class 0 is skipped, the constant term column $ct \neq 0$ is removed as well
- the LFSRs are represented with a univariate polynomial $h(x)$ from equation (3.13). Their classification w.r.t. Table 10.3 corresponds to their multivariate representation f (see equation (10.1)). As a consequence, the constant term is treated together with other coefficients, as it will require matrix-vector multipliers for all $c_{i_0, i_1, \dots, i_{t-1}} \in \mathbb{F}_q \setminus \mathbb{F}_2$ for $q > 2$
- classes 4-7, which can occur for NLFSRs and FILFUNs, are merged because they require submodules for multiplication and exponentiation. The VHDL can be generated by implementing the feedback/filtering function as a “black-box”; this decision will be elaborated in the discrete components architecture implementations discussion in Section 11.2.2.
- class 8 for constant array implementations is added to Table 11.1; it is only used for the FILFUN objects, as will be explained in the constant array architecture implementations discussion in Section 11.2.2.

conditions				class w.r.t. Table 10.4	implementation status	finite field arithmetic			
FSR type	\mathbb{F}_q	$\deg(f)$	$c_{i_0, i_1, \dots, i_{t-1}} \in \mathbb{F}_q \setminus \mathbb{F}_2$			\oplus	\times	γ	M
LFSR or FILFUN	$q = 2$	1	\times	1	fully	\checkmark	\times	\times	\times
	$q > 2$		\checkmark	2	fully	\checkmark	\checkmark	\times	\times
NLFSR or FILFUN	$q = 2$	> 1	\times	3	fully	\checkmark	\times	\checkmark_A	\times
	$q > 2$		–	4-7	partially, black-box	\checkmark	\checkmark	\checkmark	\checkmark
FILFUN	–		–	–	8	fully	constant array		

Table 11.1: Classification of the FSR objects

■ **Implementation detail:** As noted above, the LFSRs are represented with an univariate polynomial $h(x)$ from equation (3.13), but their classification corresponds to their multivariate representation f (see equation (10.1)). Instead of transforming h into f , polynomial h is classified directly, and classes 2, 5 and 7 w.r.t. Table 10.3 or $ct \in \mathbb{F}_q \setminus \mathbb{F}_2$ collapse into class 2 in Table 11.1. The constant term ct is obtained with method `ConstTermOfFSR`. For all other cases, the LFSR *expression* is re-classified as class 1. ■

■ **Implementation detail:** The classification is performed by the method `Which_class(FSR, ca, bitwidth)`, which parses the values for *expression* and F , then calls the `Which_class(expression, F, ca, bitwidth)` method from the `GAPtoVHDL` package (Section 10.3). The *expression* is obtained as `FeedbackPoly` in case of an LFSR and as `MultivarPoly` in case of an NLFSR or FILFUN (see Table 6.4 in Section 6). The *expression* is defined (and classified) over the finite field F obtained using `UnderlyingField(FSR)`. ■

Example 11.1.1 Classification of FSRs \longleftrightarrow

The Example 11.1.1 shows different FSR classification examples. The output of classification has the format `[kind, class, comment]`, whereby the integer `kind` is used to distinguish the (N)LFSRs from FILFUNs:

- 1 for (N)LFSRs: they have internal state (registers)
- 0 for FILFUNs: no internal state

Further differences between the two kinds of FSRs will be discussed in Section 11.2, when the VHDL generation is discussed. The kind is set to 0 for FILFUNs and to 1 otherwise. It is used for methods to which do not take the entire FSR object as one of its arguments.

The first LFSR shows an example of class 1 feedback over an extension field, which only requires XOR gates for its implementation. Remaining two class 2 LFSRs have already been discussed in Example 6.2.2 in Section 6. Note the first two LFSRs in Example 11.1.1 have reducible feedback polynomials. Next, three different FILFUN classifications are shown, all with kind=0. The second FILFUN was used for Example 6.2.3 in Section 6. Last example shows the NLFSR from Example 6.2.4 in Section 6.

Example 11.1.1

```

gap> K := GF(2);; x := X(K, "x");; f := x^4 + x^3 + 1;;
gap> F := FieldExtension(K, f);; y := X(F, "y");; gen := Z(2^4);;
gap> l := y^4 + y + 1;; test := LFSR(F, l);;
gap> Which_class(test, 0, 0);
[ 1, 1, "lfsr over binary (ext) field" ]
gap> l := y^4 + y + gen;; test := LFSR(F, l);;
gap> Which_class(test, 0, 0);
[ 1, 2, "lfsr with coeffs from binary ext field" ]
gap> l := y^4 + y^3 + y + gen;; test := LFSR(F, l);;
gap> Which_class(test, 0, 0);
[ 1, 2, "lfsr with coeffs from binary ext field" ]
gap> l := x_0 + x_1 + x_2;; test := FILFUN(K, l);;
gap> Which_class(test, 0, 0);
[ 0, 1, "filfun linear over binary field" ]
gap> l := x_0*x_1 + x_2;; test := FILFUN(K, l);;
gap> Which_class(test, 0, 0);
[ 0, 3, "filfun non-linear over binary field" ]
gap> l := x_0*x_1 + x_2;; test := FILFUN(F, l);;
gap> Which_class(test, 0, 0);
[ 0, 4, "filfun non-linear over binary ext field, all coeffs from bin field" ]
gap> l := x_0 + x_1*x_3;; n := 6;; test := NLFSR(K, l, n);;
gap> Which_class(test, 0, 0);
[ 1, 3, "nlfsr non-linear over binary field" ]

```



11.2 Generating VHDL for individual FSR objects

This section is organized as follows: first, the VHDL packages are explained (Subsection 11.2.1), followed by (sub)modules (Subsection 11.2.2). The latter is further divided into three parts: the entity, the discrete components architecture, and the constant array architecture.

All GAP functions involved in design entry are listed in Table 11.2. The FSR objects in the design are enumerated with the *FSR identifier* FSRid, and the finite fields in the design with the *field*

identifier *Fid*. The last two lines in Table 11.2 show the `Write_fsr` function calls, which automatically perform all the steps needed to generate VHDL (sub)modules. The simple case requires the following arguments: (*FSR*, *FSRid*, *Fid*, *strlist*), where *strlist* contains two strings, the target folder name, and the comment for the VHDL files. The second option has two additional module instructions: the *fsr_extcond* condition for external step (see `StepFSR` with external input in Table 6.3 and Key 6.3 in Section 6), and the *fsr_ca* condition for the constant array implementations (Table 10.2 in Section 10.1).

FSR - the FSR object *FSRid* - FSR identifier (integer)
fsr_extcond - external step condition *Fid* - field identifier (integer)
fsr_ca - constant array condition *strlist* - [*folder*, *comment*]

<i>packages</i>	<i>modules</i>	
<code>Get_fsr_strings</code>	<code>Write_fsr_entity</code>	<code>Write_fsr_arch_outputs</code>
<code>Write_fsr_instance_pkg</code>	<code>Write_fsr_arch_declarations</code>	<code>Write_fsr_arch_muxes</code>
<code>Write_fsr_pkg</code>	<code>Write_fsr_arch_MVconstants</code>	<code>Write_fsr_arch_stateregz</code>
<code>Write_dp_pkg</code>	<code>Write_fsr_arch_bin_monomials</code>	<code>Write_fsr_arch</code>
	<code>Write_fsr_arch_feedback</code>	<code>Write_fsr_arch_black_box_feedback</code>
	<code>Write_fsr(FSR, FSRid, Fid, strlist)</code>	
	<code>Write_fsr(FSR, FSRid, Fid, fsr_extcond, fsr_ca, strlist)</code>	

Table 11.2: Main functionality of the FSRtoVHDL package

11.2.1 VHDL packages

The FSRtoVHDL package writes two packages: one with signals for the finite field(s) and another with signals for the FSR(s). It uses functions from the GAPtoVHDL package to write the finite field information. If more FSRs are present in the design, first all the FSRs are collected and all of their `UnderlyingFields` into duplicate free lists. The field package is written first, using the GAPtoVHDL writing functions listed in Table 10.1 in Section 10.1; the different fields enumerated with the field identifier *Fid*, and if there is only one field in the package, the *Fid* can be omitted. Similarly, the FSR objects in the design are enumerated with the FSR identifier *FSRid*, which again is just an integer. The method `Get_fsr_strings` will use *FSRid* for the constants and (sub)type names, e.g., constant `fsr_FSRid_len` for the length of the FSR. To write the `fsr_pkg.vhd`, each FSR identifier *FSRid* is paired with the field identifier *Fid* that belongs to the `UnderlyingField` of the particular FSR. Then `Get_ffe_strings` is used to obtain the needed values, e.g., the string for the data type used for the field elements `ffe_Fid`.

Key 6.2 suggest exploiting structural similarities between the three FSR types. However, it is important to distinguish the (N)LFSRs from FILFUNs. Recall from Table 6.1:

- (N)LFSRs: internal state (array), shifting, one input, possibly multiple output taps
- FILFUNs: no internal state, array input (instead of state), a single output (tap)

For example, the data type used for the internal state of (N)LFSRs is defined as

```
type fsr_FSRid_state is array (fsr_FSRid_len - 1 downto 0) of ffe_Fid;
```

where FSRid and Fid are replaced by appropriate integers. Similarly, an external input (for external step and run), which is possible for all three kinds of FSRs, is defined in the package as

```
subtype fsr_FSRid_ext is ffe_Fid;
```

The VHDL data types defined in the `fsr_pkg.vhd` are then used for the input and output ports and internal signals of the FSR modules. All the information needed to write the package is obtained from the FSR attributes listed in Table 6.4 in Section 6.

Key 11.1: The structure of the FSR objects

FSR objects are structured very consistently: each FSR has two data inputs, the loading input `i_fsr` and the external step input `i_ext`, and one output called `o_fsr`. Their VHDL data types are defined in `fsr_pkg.vhd` and easily accessible via the FSRid.

Example 11.2.1 A simple LFSR – VHDL packages `field_pkg` and `fsr_pkg` ↔

This example is a continuation of Example 6.2.2 with an LFSR of length 4 defined over \mathbb{F}_2^4 with generator α . The LFSR is defined by $\ell(y) = y^4 + y^3 + y + \alpha$ (see Example 6.2.2(b)). The VHDL Example 11.2.1(a) shows the VHDL type definition for elements with Fid=1 in `field_pkg.vhd` (constants are omitted for brevity). The VHDL Example 11.2.1(b) shows all constants and type definitions for the LFSR with FSRid=1 in `fsr_pkg.vhd`.

VHDL Example 11.2.1(a)

```
constant ffe_1_dim : natural := 4;
subtype ffe_1 is std_logic_vector(0 to ffe_1_dim - 1);
```

VHDL Example 11.2.1(b)

```
constant fsr_1_len : natural := 4;
constant ot_1 : natural := 1;
subtype fsr_1_input is gf_elem_1;
type fsr_1_state is array (fsr_1_len - 1 downto 0) of ffe_1;
constant fsr_1_input_zero: fsr_1_input := ff_1_zero;
subtype fsr_1_ext is ffe_1;
constant fsr_1_ext_zero: fsr_1_ext := ffe_1_zero;
subtype fsr_1_output is ffe_1;
```



11.2.2 VHDL for the FSR (sub)modules

The entity

FSR modules are written with the functions listed in Table 11.2. The following discussion refers to `ffe_Fid` as the “field element” of the `UnderlyingField` of the FSR with a given `FSRid`. For FSR related VHDL data types recall `fsr_pkg` from the VHDL Example 11.2.1(b). Example 11.2.1 is continued in Example 11.2.2, which shows the schematic of the LFSR used in the example and ports in VHDL Example 11.2.2. The entity uses function `Write_fsr_entity` from Table 11.2, which follows the following principles:

- the structured `fsr_pkg` removes a lot of clutter from the ports¹:
 - the primary input port `i_fsr` is always of the type `fsr_FSRid_input`, which is a single field element in the case of (N)LFSRs and an array field elements of length `fsr_FSRid_len` in the case of FILFUNs
 - all FSRs have a single output port `o_fsr` of type `fsr_FSRid_output`. In the case of (N)LFSRs the output port can be either a single field element or an array of field elements of length² `ot_FSRid`. In the case of FILFUNs the output port is always a single field element.
- the (N)LFSRs have an internal state:
 - they will need a clock signal `clk`
 - the loading takes advantage of the shifting: through the `i_fsr` port, the highest stage is loaded, which implies a multiplexer and input control port `load` to choose between `i_fsr` and the feedback
 - all the registers have a chip enable called `fsr_en`. To keep the hardware area small, there is no reset signal, and the chip enable can be removed to further reduce the area.
- all FSRs can perform an external step and run (Key 6.3), which requires a secondary data input `i_ext` of type `fsr_FSRid_ext`, which is always a single field element. There are three options, passed on as the argument `fsr_extcond` for external step condition:
 - external step and run is never used (no `i_ext` input port): $fsr_extcond = -1$
 - external step and run is always used: $fsr_extcond = 0$
 - external step and run is used sometimes, implying a multiplexer and input control port `ext` to choose between `i_ext` and the feedback: $fsr_extcond = 1$

Example 11.2.2 A simple LFSR – schematic and entity \longleftrightarrow

This example is a continuation of Example 6.2.2 and 11.2.1, with an LFSR of length 4 defined over \mathbb{F}_{2^4} with generator α . The LFSR is defined using the primitive polynomial $\ell(y) = y^4 + y^3 + y + \alpha$ (see Example 6.2.2(b)). The type

¹ and internal signals later on

² length of the `OutputTap` attribute

definitions with $Fid=1$ and $FSRid=1$ were shown in Example 11.2.1. Figure 11.1 is an extension of Figure C.1 (shaded grey) from Example C.1.1 in Appendix C.1. Figure 11.1 shows all elements needed for the hardware module: both multiplexers, control signals (dashed arrows) and data ports (solid arrows). Both the loading and the external step multiplexer are shown in the schematic, with their control inputs $load$ and ext respectively. The control signals for the registers are shown only for stage S_3 , but apply to all four stages. The output is annotated as o_fsr with the stage number 0, a convention used because more output taps are possible.

VHDL Example 11.2.2

```

port(
  i_fsr:  in fsr_1_input;  -- data input
  i_ext:  in fsr_1_ext;    -- external input
  clk:    in std_logic;    -- clock input
  fsr_en: in std_logic;    -- fsr enable signal
  load:   in std_logic;    -- load control signal
  ext:    in std_logic;    -- init control signal
  o_fsr:  out fsr_1_output -- data output
);

```

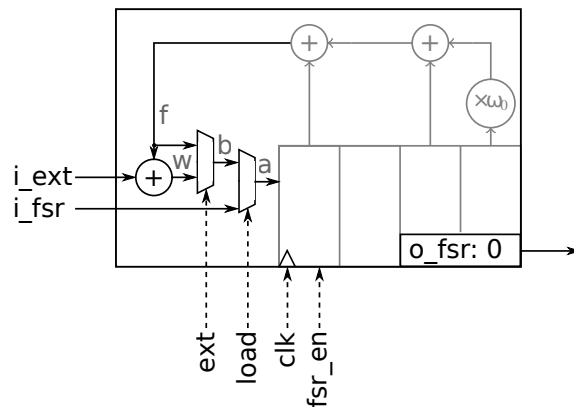


Figure 11.1: Schematic for the LFSR $\ell(y) = y^4 + y^3 + y + \alpha$ with multiplexers, control signals and data ports



The architecture: signal declarations

The first part in architecture are the declarations of internal signals. The first paragraph below refers to discrete component architectures and the second to constant array architectures.

For the (N)LFSRs, the internal state (state array) is declared as type `fsr_FSRid_state`, followed by signals for the multiplexers and feedback (signals shown in Figure 11.1, shaded grey), of type `ffe_Fid`. Then, for classes 1-3 (see Table 11.1), the term and monomial signals are declared based on the number of extension field coefficients $\gamma_{i_0, i_1, \dots, i_{t-1}} \in \mathbb{F}_q/\mathbb{F}_2$ and the number of monomials in the `FeedbackPoly` or `MultivarPoly` respectively. For example, for n monomials, a new type

`mon_array` is defined as an array of n field elements `ffe_Fid`, and if only one monomial is needed, it is declared directly as `ffe_Fid`. See Examples 11.2.3 and 11.2.4.

The number of signal declarations for constant array architectures is much shorter. The feedback and multiplexers signals of type `ffe_Fid` are still present. Then, the type definition for the constant array is needed, followed by the actual constant declaration. See Example 11.2.5.

The architecture: discrete components

The architecture is written by the functions listed in Table 11.2, which are called according to the classification of the FSR objects (Table 11.1):

[class 2] - for each extension field constant $\gamma_{i_0, i_1, \dots, i_{t-1}} \in \mathbb{F}_{2^m} / \mathbb{F}_2$, call the function `Write_fsr_arch_MVconstants`, which performs the following steps:

- generate the matrix for the matrix-vector (MV) multiplier using the FFCSA package³ method `MatrixMultByConst` (Table 7.2)
- write the MV submodule using `ffe_Fid` type for its input and output port. Note that `ffe_Fid` represents field elements of \mathbb{F}_{2^m} , which is constructed as a single extension.
- write the component instantiation for the MV submodule using `Write_component_inst` (Table 10.3). The port map input is the appropriate stage of the state array in the case of (N)LFSRs, or the appropriate coordinate of the `i_fsr` in the case of FILFUNs. The port map output is (the appropriate coordinate⁴ of) the `term` signal.

[class 3] - call the function `Write_fsr_arch_bin_monomials`, which:

- for each monomial calls `bin_monomial` (Table 10.3). The binding is either to the appropriate stage of the state array in the case of (N)LFSRs, or the appropriate coordinate of the `i_fsr` in the case of FILFUNs. The obtained product is assigned to (the appropriate coordinate⁵ of) the `monomial` signal using function `Write_line` (Table 10.3).

[class 1-3] - call `Write_fsr_arch_feedback` to write the final sum using `XOR_line` and `Write_line` (Table 10.3). The source signals (summands) for `XOR_line` depend on the class:

[class 1] - the sources will be only stages of the state array in the case of (N)LFSRs or the appropriate coordinate of the `i_fsr` in the case of FILFUNs

[class 2] - the sources will contain `term` outputs of the MV submodules in addition to sources which appear in class 1 FSRs

[class 3] - the sources will contain `monomial` signals

[class 4-7] - call `Write_fsr_arch_black_box_feedback`, which simply instantiates a black-box component called `black_box_FSRid`. The port map input is the state array in the case of NLFSRs, and `i_fsr` in the case of FILFUNs. The port map output is always

³ the FSRtoVHDL package contains its own copy of this method, loaded only in case FFCSA is not available

⁴ in the case of multiple MVs, the `term` signal is declared as an array of field elements

⁵ in the case of multiple monomials, the `monomial` signal is declared as an array of field elements

the signal `f`. The actual submodule can be implemented either using the `CIRCUIT` package (Chapters 13-16) or manually.

The rest of the FSR module uses the remaining functions listed in Table 11.2 independently of classification, but depending on the type of FSR:

- (N)LFSRs have `load` and `ext` multiplexers driving the input to the highest stage of the state array (see the LFSR schematic in Figure 11.1 in Example 11.2.2). The FILFUNs have no internal state or feedback and do not require a load multiplexer. The `ext` multiplexer is driving the output `o_fsr` for FILFUNs. All multiplexers are written by the `Write_fsr_arch_muxes` function. To summarize:
 - the (N)LFSRs always have the load multiplexer. Figure 11.1 in Example 11.2.2 shows an LFSR with both multiplexers.
 - all FSR objects can have the multiplexer switching between external and regular step and run, controlled by the input `ext` (recall the `fsr_extcond` from the entity discussion).
- (N)LFSRs have one or more stages of the state array driving (the coordinates of) the output port `o_fsr`, written by the function `Write_fsr_arch_outputs`
- (N)LFSRs have registered state array, written by the function `Write_fsr_arch_stateregz`

Two examples are shown below, the datapath for a simple 4-stage LFSR in Example 11.2.3 and the datapath for the Grain NLFSR in Example 11.2.4.

The architecture: constant arrays

The constant array implementations were discussed in the Subsection 3.3.3, and functions for their hardware implementations in Table 10.2 in Section 10. The bit-width upper limit on the constant array input(s) is currently set to 16 bits, to prevent the constant array to grow too large. The experience with the synthesis results for the WG constant array implementations [113, 120, 119] show that 16 is a generous number. In FSRtoVHDL, the constant arrays are possible for the FILFUN objects; (N)LFSR objects would usually exceed the bit-width threshold.

For constant array implementations, the `Write_fsr` function call is used with the parameter `fsr_ca` set. If a class 8 expression is encountered, the `Write_fsr_arch` will choose between the functions `Write_arch_const_array_permutation` and `Write_arch_const_array_functional` (Table 10.2) calls based on the degree of extension of the `UnderlyingField` over its prime subfield $m = [\mathbb{F}_{2^m} : \mathbb{F}_2]$, i.e., it will choose the permutation when $m > 1$. Example 11.2.5 shows a constant array implementation of the WG7 permutation, modelled as a FILFUN object.

Example 11.2.3 A simple LFSR – the datapath \longleftrightarrow

This example is a continuation of Examples 6.2.2, 20.3.1 and 11.2.2: the LFSR with primitive polynomial $\ell(y) = y^4 + y^3 + y + \alpha$. The MV submodule notation is $\alpha = \omega^{13}$, where ω is the root of the field defining polynomial $f(x)$. The datapath in VHDL Example 11.2.3 implements the schematic in Figure 11.1.

VHDL Example 11.2.3

```
signal sa : fsr_1_state;
signal a,b,w: ffe_1;
signal fsimple, fcomplex, f: ffe_1;
signal term: ffe_1;
begin
-- MVmultipliers for ext. field constants
-- MVmultiplier for constant  $w^{13}$ 
ms1 : entity work.MVmult_ffe_1(w_1_13) port map ( sa(0), term );

-- feedback
fsimple <= sa(3) xor sa(1) ;
f <= fsimple xor term;
w <= i_ext xor f;

-- output
o_fsr <= sa(0);

--muxes at fsr input
---- feedback vs (i_ext xor feedback )
b <= ( w ) when ext = '1' else ( f );
---- prev. mux output vs i_fsr input
a <= ( i_fsr ) when load = '1' else ( b );

--stage registers and shifting
stages: for i in fsr_1_len - 1 downto 1 generate
  stage_shift: process(clk) begin
    if rising_edge(clk) then
      if fsr_en='1' then
        sa(i-1) <= sa(i);
        ... OMITTED FOR BREVITY ....
      end generate stages;

stage_new: process(clk) begin
  if rising_edge(clk) then
    if fsr_en='1' then
      sa(fsr_1_len - 1) <= a;
    end if;
  end if;
end process;
```

\longleftarrow

Example 11.2.4 Grain NLFSR – the datapath \longleftrightarrow

The VHDL Example 11.2.4 shows a fraction of the Grain NLFSR datapath, with the FSRid=2. As the only finite field in the design is \mathbb{F}_2 , no Fid is used, and the data type for the field elements is simple ffe. Because of the length of the multivariate polynomial $h(x)$, parts of the generated code are omitted for brevity. The example shows the type definition for mon_array. To simplify the Write_fsr_arch_feedback function, the state array stages entering the sum are rewired as monomial signals, e.g., stages 60 and 63. After the two coordinates of the output o_fsr have been assigned, the rest of the code (for registers) is up to the FRSid identical to the previous Example 11.2.3 and hence omitted.

VHDL Example 11.2.4

```
signal sa : fsr_2_state;
signal a,b,w: ffe;
signal fsimple, fcomplex, f: ffe;
type mon_array is array (0 to 21) of ffe;
signal monomial : mon_array;
begin
-- monomial0: x_21*x_28*x_33*x_37*x_45*x_52
  monomial(0) <= sa(21) and sa(28) and sa(33) and sa(37) and sa(45) and sa(52) ;

-- monomial1: x_9*x_15*x_21*x_28*x_33
  monomial(1) <= sa(9) and sa(15) and sa(21) and sa(28) and sa(33) ;

          ... OMITTED FOR BREVITY ....

-- monomial20: x_60
  monomial(20) <= sa(60) ;

-- monomial21: x_63
  monomial(21) <= sa(63) ;

-- feedback
  f <= monomial(0) xor ... OMITTED FOR BREVITY .... xor monomial(21) ;
  w <= i_ext xor f;

-- output
  o_fsr(0) <= sa(0);
  o_fsr(1) <= sa(63);

          ... OMITTED FOR BREVITY ....
```

\longleftarrow

Example 11.2.5 *The WGP constant array implementation* \longleftrightarrow

This example is a continuation of Example 8.1.1 in Section 8.1. Example 11.2.5 it shows the GAP code for the constant array implementation of the WG permutation $WGP-m(X^d)$ with $m = 7$ and $d = 63$. The FSR is the FILFUN `dwgpfil`, obtained as the `WGPfilter` component of the WG object (see Section 8.1). The package writing function calls are omitted from this example. The FSR module is written using `Write_fsr(FSR, FSRid, Fid, fsr_extcond, fsr_ca, strlist)`. Assuming that the LFSR, not shown in this example, will have $FSRid = 1$, the `dwgpfil` has the $FSRid = 2$, and that \mathbb{F}_2 will have $Fid = 1$, the field \mathbb{F}_{2^7} is given $Fid = 2$. Hence, the first two integers after the `dwgpfil` are both 2. Next is the $fsr_extcond = -1$, i.e., only regular step and run is used, followed by the $fsr_ca = 1$, the constant array instruction. The architecture with the type definition used and the declaration for the constant array is shown in VHDL Example 11.2.5.

Example 11.2.5

```
gap> wg := WG(K, f, d);;
gap> dwgpfil := wg!.WGPfilter;;
gap> folder := "phdGapExamples/FSRtoVHDL/ex3wgp";;
gap> comment := "test for Write_fsr for example3";;
gap> strlist := [folder, comment];;
gap> Write_fsr(dwgpfil, 2, 2, -1, 1, strlist);;
OutputTextFile(/home/.../phdGapExamples/FSRtoVHDL/ex3wgp/fsr_2.vhd) done!
class: 8      filfun non-linear using const_array
mv = x_0^126+x_0^125+x_0^124+x_0^110+x_0^109+x_0^108+x_0^95+x_0^91+x_0^79+x_0^75+
x_0^62+x_0^61+x_0^60+x_0^59+x_0^47+x_0^46+x_0^45+x_0^44+x_0^43
OutputTextFile(/home/.../phdGapExamples/FSRtoVHDL/ex3wgp/fsr_2-const_array.vhd) done!
```

VHDL Example 11.2.5

```
signal f, w: ffe_2;
type fsr_2_table_ty is array( 0 to 2**ffe_2_dim - 1) of ffe_2;

constant fsr_2_table: fsr_2_table_ty :=
(fffe_2'("0000000"),
 ffe_2'("0101101"),
    ... OMITTED FOR BREVITY ...
 ffe_2'("1110000")
);
begin

  f <= fsr_2_table( to_integer( unsigned( std_logic_vector( i_fsr ) ) ) );

--muxes at fsr output
  o_fsr <= f;      --the thing just computed
```

\longleftarrow

11.3 A cipher as a collection of basic FSR modules

As was mentioned in Chapter 6, a cipher can be implemented as a collection of basic modules, which were identified as LFSR, NLFSR and FILFUN (see Key 6.1). The generation of VHDL for individual FSR objects was explained in Section 11.2. This section focuses on the implementation of an entire cipher, composed of various FSRs.

The workflow in this section will be closely linked to the Grain stream cipher example. Grain was chosen because its structure as a collection of FSRs is more complex than the WG cipher, and thus able to explain the FSRtoVHDL top-level datapath generation in more detail. Example 11.3.1 shows how to model Grain as a collection of FSRs. Modelling a cipher as a collection of FSRs is the task of the designer and is sometimes quite challenging.

Example 11.3.1 Grain modelled as a collection of FSRs

Grain [46] was introduced in Subsection 3.2.3. The structure of Grain is shown Figure 11.2: it includes an 80-bit LFSR (with $f(x)$), an 80-bit NLFSR (with $g(x)$) and a filtering function $h(x)$, which takes the input bits from both the LFSR and the NLFSR. The result of this function is masked by a bit from the NLFSR to produce the keystream bit.

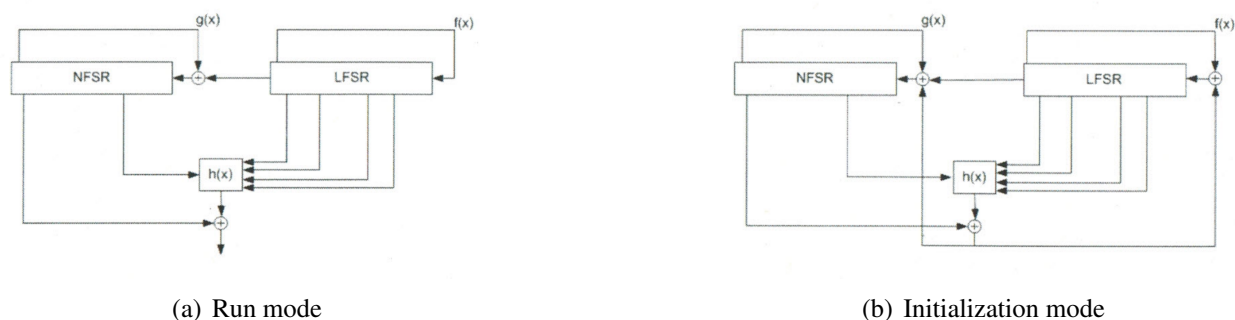


Figure 11.2: Original schematic of Grain (Figures from [46])

The two modes of operation from Figure 11.2 are combined into a single schematic in Figure 11.3 and summarized in Table 11.3. During the run mode, result of $h(x)$ is masked by a bit from the NLFSR to produce the keystream bit, see Figure 11.2(a,b). This XOR gate will be called *masking*. The filter $h(x)$ is implemented as a FILFUN object called `filfun1`, and the *masking* is used as its external input: the *masking* XOR gate is absorbed by `filfun1`, the external step is always used ($fsr_extcond = 0$) and the output of `filfun1` is the *masked h(x)* result. During the initialization (Figure 11.2(b)), the *masked h(x)* output is fed back to both the LFSR and the NLFSR. In case of the LFSR, there are no difficulties: the LFSR will use the *masked h(x)* for external step during the initialization ($fsr_extcond = 0$). The NLFSR behaviour is more complex. Both schematics in Figure 11.2(a,b) show the updating of the NLFSR with a sum of the feedback $g(x)$ and the LFSR output stage S_0 at all times. Hence, the LFSR output stage S_0 will be used as external input for the external step of the NLFSR ($fsr_extcond = 0$). Figure 11.2(b) shows a 3-input XOR gate during the initialization phase. This 3-input XOR is split into two 2-input XORs: first one is the always used external step discussed above, and is absorbed by the NLFSR object. The extra XOR gate will have to be modelled as another

FILFUN object `filfun2` with multivariate polynomial x_0 with *masked* $h(x)$ as primary `filfun2` input and the LFSR output stage S_0 as the external `filfun2`. Then, a multiplexer will be added to use the LFSR output stage S_0 as the NLFSR external input during running mode, and the sum of LFSR output stage S_0 and *masked* $h(x)$ (the `filfun2` output) during the initialization phase.

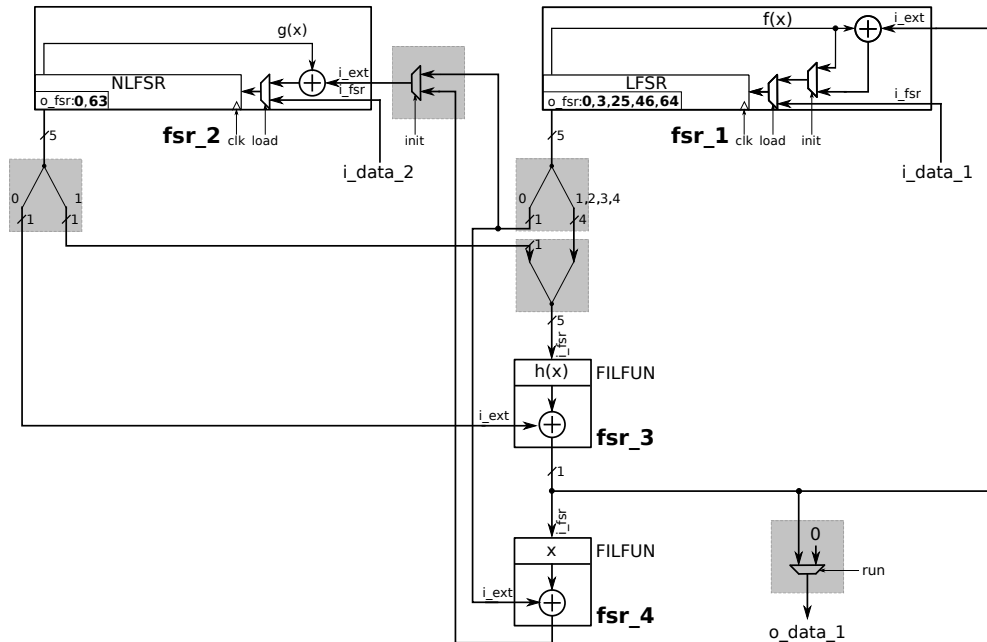


Figure 11.3: Unified schematic of Grain using FSR

FSR object		ext. step condition	object name in	
constructor	polynomial	$fsr_extcond$	GAP	VHDL [†]
LFSR	$f(x)$	1 - initialization	<code>lfsr</code>	<code>fsr_1</code>
NLFSR	$g(x)$	0 - always	<code>nlfsr</code>	<code>fsr_2</code>
FILFUN	$h(x)$	0 - always	<code>filfun1</code>	<code>fsr_3</code>
FILFUN	x	0 - always	<code>filfun2</code>	<code>fsr_4</code>

[†] - also in the spreadsheet template *.tsv

Table 11.3: Grain as a collection of FSRs

11.3.1 The setup file and the top-level module ports

The cipher will be implemented as a top-level module, i.e., datapath, which will contain all other FSRs as submodules. The user creates a simple *setup* file for the cipher to capture the following information in lists:

- all the FSRs in the cipher
- a list with architecture instructions
- lists with top-level data inputs and outputs
- a list with possible FSM states

The architecture instruction is set to integer 1 for constant array and 0 for discrete components implementation. The instructions list has the same length as the list of FSRs, and the FSRs and instructions are matched by their list position. FSRtoVHDL has a simple GAP object called INOUT, which is used to specify the top-level inputs and outputs. INOUT takes two arguments, the underlying finite field F and length m , and represents a signal of m elements from F . The INOUT objects are translated to VHDL data types. Using the `Write_dp_pkg` function (Table 11.2) a third package, `dp_pkg.vhd`, is implemented. For the array types the direction is always set to “to”. Last part of the *setup* is the list of possible FSM states: the FSM implementation is not a part of this toolkit, but the states are needed to extract internal connections between the FSRs (the shaded parts in Figure 11.3 will be extracted using the “state” keywords) and for correctness checks, as will be explained shortly.

Example 11.3.2 *Grain setup file* \longleftrightarrow

The setup file (*.g extension) for the cipher is shown in Example 11.3.2. The FSR objects use the names listed in the fourth column in Table 11.3. The FSRs are followed by the INOUTs for the top-level inputs and outputs. Then the data is collected in lists: `fsrlist` and `arch_instruction` list for the FSR objects, the `top_data_inputs` with the input for the `lfsr` and input for the `nlfsr` loading, the `top_data_output` for the keystream bit, and the list of FSM states. The state keywords correspond to the states that will drive the multiplexer control inputs. The last two rows take care of management, e.g., the target folder for the generated VHDL files and the module name `mname` for the top-level entity.

Example 11.3.2

```
K := GF(2);; x := X(K, "x");;
f := Z(2)^0 + x^13 + x^23 + x^38 + x^51 + x^62 + x^80;;
g := x_62 + x_60 + ... OMITTED FOR BREVITY ... + x_52*x_45*x_37*x_33*x_28*x_21;;
h := x_1+x_4+... OMITTED FOR BREVITY ...+x_1*x_2*x_4+x_2*x_3*x_4;;
lfsr := LFSR(K, f, [0,3,25,46,64]);; nlfsr := NLFSR(K, g, 80, [0,63]);;
filfun1 := FILFUN(K, h);; filfun2 := FILFUN(K, x);;
input1 := INOUT(UnderlyingField(lfsr), 1);;
input2 := INOUT(UnderlyingField(nlfsr), 1);;
output := INOUT(K, 1);;
fsrlist := [lfsr, nlfsr, filfun1, filfun2];;
arch_instruction := [0,0,0,0];;
top_data_inputs := [input1, input2];;
top_data_outputs := [output];;
fsm_states := ["load", "init", "run"];;
mfolder := "phdGapExamples/FSRtoVHDL/testGRAIN";;
mname := "Grain"; strlist := [mfolder , "test for Grain"];;
```

\longleftarrow

11.3.2 FSR-based system configuration

An interactive *Manager* function is invoked with the setup file as its argument. The *Manager* parses the setup file and writes an empty *spreadsheet template file* *.tsv. At this point, the *Manager* halts and waits for the user instruction. The user then fills and stores the spreadsheet, and types “yes” into the GAP prompt. No further user interaction is required. The *Manager* function reads back the spreadsheet, which now contains the configuration for the entire FSR-based system, and begins with synthesis for the top-level datapath. All submodules are written in the process.

The spreadsheet lists all possible sources as columns and all possible targets as rows. The structure of the FSR objects is used to write the empty spreadsheet template: each FSR can have only two data inputs, the load data input *i_fsr* and the input for the external step *i_ext*, and a single output *o_fsr*. Furthermore, since the FSR objects are simply enumerated by the *FSRid* (see the last column in Table 11.3) all rows and columns can easily be extracted from the FSR list and the list with top-level data inputs and outputs.

Key 11.2: The top-level datapath and configuration template

The top-level datapath is specified with a spreadsheet template that lists all possible sources as columns and all possible targets as rows. The possible sources are the top-level data inputs and the *o_fsr* outputs of all FSRs, and the possible targets are the top-level outputs and both the *i_fsr* and the *i_ext* inputs of all FSRs in the cipher, see Key 11.1. The naming convention follows *fsr_FSRid_i_fsr*, *fsr_FSRid_i_ext*, and *fsr_FSRid_o_fsr*.

The designer has to fill out the spreadsheet using *connector* rules, the user-specified FSM state keywords, and keyword *always*. The state keywords are used to extract the multiplexers, and the keyword *always* is interpreted as a “hard-wired” signal. All state keywords are crosschecked with the information obtained in the setup file, and any deviation triggers an error.

The *connectors* are listed in the first column in Table 11.4: *merge*, *split*, *select*, *expand* and *permutation*, also depicted graphically. The table contains very simple rules, followed by examples showing the format of each connector.

<i>connector</i>	rules					example
	source(s) range	target(s) length	exclusive positions	same permutation	(sub)field	
⊠ <i>merge</i>	✓	✓	✓		✓	[2,1,-1],[-1,-1,0]
⊠ <i>split</i>	✓	✓			✓	[0,1], [2]
▷ <i>select</i>	✓	✓			✓	[1,3]
◁ <i>expand</i>	✓	✓			✓	[0,1,1,1]
□ <i>permutation</i>	✓	✓		✓	✓	[1,3,0,2]

Table 11.4: Connectors for the top-level datapath

The format of the connectors is very intuitive: connectors are lists of indices of the source signal(s). As such, one of the rules is that all the values (indices) fall into the range of the source signal(s). Only the *merge* connector can have more than one source. The next rule is about the length of the target signal(s): there must be a driver for each coordinate of the target, hence the length of the list(s) must match the length of the target signal(s). Only the *split* connector can have more than one target. The *merge* connector example shows two lists of length 3, assuming the target signal has length 3. In the first list, the last index has a value of -1, indicating this index will come from another source. Indeed, the second list has the values -1 twice, and than 0 for the last entry. The positions of the -1 values must be exclusive among all the lists, and eventually, a driver must be found for each position. The *select* and *expand* connectors are quite selfexplanatory, and the only constraint on the *permutation* is that it must be an actual permutation of indices where every index in the range appears exactly once. The last rule checked is the that the indexed values are of the same type and can be connected with a wire. The FSRtoVHDL package works for \mathbb{F}_2 or \mathbb{F}_{2^m} , so this is a simple check. Basis transitions are not yet supported and must be inserted manually.

The Manager will recognize the connectors in the filled-out spreadsheet (configuration) and check the rules listed in Table 11.4. For example, if two index lists are found in the same row of the spreadsheet, they are interpreted as a *merge* connector. Similarly, an index list that is shorter than the length of the source signal is interpreted as the *select* signal. Note that *split* can always be regarded as multiple *select* connectors, which is how the Manager reads the spreadsheet.

	i_data_1	i_data_2	fsr_1_o_fsr	fsr_2_o_fsr	fsr_3_o_fsr	fsr_4_o_fsr
o_data_1	×	×	×	×	×	run
fsr_1_i_fsr	load	×	×	×	×	×
fsr_1_i_ext	×	×	×	×	init	×
fsr_2_i_fsr	×	load	×	×	×	×
fsr_2_i_ext	×	×	[0] run	×	×	init
fsr_3_i_fsr	×	×	[1, 2, 3, 4, -1]	[-1, -1, -1, -1, 1]	×	×
fsr_3_i_ext	×	×	×	[0]	×	×
fsr_4_i_fsr	×	×	×	×	always	×
fsr_4_i_ext	×	×	[0]	×	×	×

Table 11.5: Grain spreadsheet example

■ **Implementation detail:** The top-level data ports are specified as INOUT objects. These objects have an underlying finite field F , which is added to the list of all fields in the design, and the duplicate free list of finite fields is used to write the `field_pkg.vhd`. Hence, the fields used for the top-level ports have a field identifier `Fid`, and their VHDL data types are defined using this `Fid`. ■

Example 11.3.3 *Grain – the filled-out spreadsheet template* \longleftrightarrow

Table 11.5 shows the filled out spreadsheet template (Grain configuration), built from information provided in the template from Example 11.3.2. The first two columns are the top-level data input ports, `i_data_1` and `i_data_2`, for

the two INOUT objects in `top_data_inputs` list in Example 11.3.2. The other columns are the FSR output ports, obtained from the `fsrlist` in Example 11.3.2. The first row is the top-level data output port, `o_data_1`, for the INOUT in the `top_data_outputs` list in Example 11.3.2. All other rows are the two possible inputs to the four FSR objects: the primary data input `i_fsr` and the external step input `i_ext`. The names of the FSR objects are as listed in last column of Table 11.3.1. all `fsr_FSRid_*` signals will appear as internal signals in the generated VHDL for the top-level datapath. The easiest way to follow Table 11.5 is to compare it with the schematic in Figure 11.3, which has annotations for the `i_fsr` and `i_ext` inputs to each FSR.

For example, `i_data_1` is connected to the `i_fsr` input of `fsr_1` during load, and the `fsr_4_o_fsr` is connected to top-level output `o_data_1` during run. The `fsr_2_i_ext` (the `nlfsr`) is connected to `fsr_1_o_fsr[0]` during run, as indicated by the `select` rule [0], and `fsr_4_o_fsr` during init. The `fsr_3_i_fsr` is connected to *merged* signals from `fsr_1_o_fsr[1,2,3,4]`, which correspond to the `lfsr` stages S_3, S_{25}, S_{46} and S_{64} , and `fsr_2_o_fsr[1]`, which corresponds to the `nlfsr` stage S_{63} , see the output taps lists in the constructor calls for the FSRs in Example 11.3.2. The keyword `always` means the `fsr_3_o_fsr` will be hardwired to the `fsr_4_i_fsr`.

←

11.3.3 Extracting connections, multiplexers, and external step conditions

<i>manager</i>	<i>toplvl</i>
Write_template	Get_fsr_portmap
Parse_template	Get_dp_signals
Separate_tables	Get_dp_assignments
Extract_extcondition	Write_dp_entity
Extract_muxcondition	Write_dp_arch
Manager	

Table 11.6: Main functionality of the FSRtoVHDL package - continued

The `Manager` will separate the configuration into three tables that will be used for the extraction of the connections, external step conditions, and for multiplexers (inference):

- **conditions:** connections, extraction of FSR internal multiplexers use. This table is used for the method `Extract_extcondition`
- **connectors:** connections
- **multiplexer conditions:** remaining connections and multiplexers. This table is used for the method `Extract_muxcondition`

Connections are possible between submodules and to the top-level data inputs and outputs.

The external step condition is set based on the keywords in the conditions table. If in the filled-out spreadsheet no keyword is given but there is a connector rule present, this cell in the conditions table is changed to `always`, and the corresponding FSR gets the `fsr_extcond = 0` (the external step is always used). Only an empty `i_ext` row will get the `fsr_extcond = -1` (the external step

is never used). If a keyword different from `always` is found, and only one cell in the row is filled, the `fsr_extcond = 1` is used. The latter case occurred for FSRs `fsr_1`, `fsr_3` and `fsr_4` in the Grain example (Table 11.5). If exactly two cells in the `i_fsr` row are filled, an multiplexer is inferred and the corresponding `fsr_extcond = 0`: the FSR will always use an external step, but a multiplexer outside the FSR submodule is driving the `i_ext` port. This is the case for the `fsr_2` in in the Grain example (Table 11.5).

The second table is used for rule checking and to specify the connections that depend on the *connectors*. The third table, multiplexer conditions, is a reduced version of the conditions table: all of the “inside FSR already available multiplexers” are removed. As such, it captures the *inferred* datapath multiplexers. This table is used to specify additional multiplexer control top-level inputs, the internal multiplexer signals and for connecting the multiplexer outputs to appropriate targets. An example of the multiplexer table is shown in Example 11.3.4.

Example 11.3.4 *Grain spreadsheet template - continued* ←→

This example shows the Manager invoked for the setup `*.g` file from Example 11.3.2, and the separation of tables. The first two lines of the conditions table are shown, then the output is omitted for brevity, followed by the multiplexer conditions table, with last four empty rows again omitted. The latter has only two rows with cells different from “x”. These two rows capture the two multiplexers shaded grey in Figure 11.3: the output multiplexer in the first row, and the multiplexer for the `i_ext` input to `fsr_2` (the `nlfsr`) in the fifth row. The first multiplexer has one source connected to `fsr_4_o_fsr`, and the other source hardwired to constant 0, because the `o_data_1` row has only one keyword. The second multiplexer has two keywords, hence its sources are `fsr_1_o_fsr[0]`⁶ and `fsr_4_o_fsr`.

Example 11.3.3

```
gap> Manager("phdGapExamples/FSRtoVHDL/exGRAIN.g");
OutputTextFile(/home/.../phdGapExamples/FSRtoVHDL/testGRAIN/Grain.tsv) done!
template ready? [yes/no]
separating tables
[ [ "x", "x", "x", "x", "x", "run" ],
  [ "load", "x", "x", "x", "x", "x" ],
  ... OMITTED FOR BREVITY ...
[ [ "x", "x", "x", "x", "x", "run" ],
  [ "x", "x", "x", "x", "x", "x" ],
  [ "x", "x", "x", "x", "x", "x" ],
  [ "x", "x", "x", "x", "x", "x" ],
  [ "x", "x", "run", "x", "x", "init" ],
  ... OMITTED FOR BREVITY ...
```



⁶ the value `[0]` is extracted from the second table, omitted from this example

Key 11.3: The top-level datapath: connections conditions and multiplexers

The configuration (Key 11.2) is used to extract the connections (internal signals), external step conditions, and multiplexers (inference). If possible, the multiplexers already available inside FSRs are used. Sometimes, additional datapath multiplexers are inferred.

After separating the tables, the Manager proceeds as follows, using the functions and methods from Tables 11.2 and 11.6:

- extract external step conditions using `Extract_extcondition`
- write three packages: the `field_pkg.vhd` and `fsr_pkg` (Subsection 11.2.1), and the new package `dp_pkg.vhd`.
- for each FSR in the `fsrlist`:
 - call `Get_fsr_portmap`: the only additional data needed for the port map is the external step condition. The positional port map is generated in the same order as the entity ports for the FSR object in question. The strings for signal names are the same as those listed in Key 11.2. The control signals follow similar naming convention: `fsr_FSRid_fsr_en`, `fsr_FSRid_fsr_load`, and `fsr_FSRid_c_ext`.
 - call `Write_fsr`, as was explained in Subsection 11.2.2
- extract conditions for the inferred multiplexers using `Extract_muxcondition`: how many multiplexers per target signal.
- obtain all internal datapath signals and their VHDL datatypes using `Get_dp_signals`. These signals will be declared in the architecture of the top-level datapath module. The list includes all FSR source and target signals (using the naming convention in Key 11.2), and three signals per inferred multiplexer, namely `targetname_mux_#`, `targetname_mux_#_src1`, and `targetname_mux_#_src2`, where `targetname` is one of the target signals and `#` is replaced by the consecutive number of the stacked multiplexers⁷. The VHDL data types for the FSR signals are obtained using `Get_fsr_strings` (`FSRid`) and match the type definitions in `fsr_pkg.vhd`. If a multiplexer is needed for the top-level output port, its VHDL data type matches the type definition in `dp_pkg.vhd`.
- finally obtain the combinational assignments (method `Get_dp_assignments`) using the information from all three tables⁸ to determine the r.h.s. for the assignments. There is one assignment per target signal when the condition is always or uses an FSR internal multiplexer, three assignments per each inferred datapath multiplexer (one for each multiplexer source, the actual multiplexer using a `when/else` statement, and the assignment of the multiplexer output to the target signal), and one assignment for each target signal coordinate when *connectors* are used.
- generate the VHDL of the top-level datapath module, using the functions `Write_dp_entity` and `Write_dp_arch`

⁷ wider multiplexers are implemented using 2/1 multiplexers, hence enumeration is needed

⁸ conditions, connectors, and multiplexer conditions

Key 11.4: The structure of the FSR objects and the port map for component instantiations

The port map is obtained from the FSR object structure (Key 11.1) and the external step condition. The positional port map will be generated in the same order as the entity ports for the FSR object in question. It will include the clock signal and the signals `fsr_FSRid_fsr_en` and `fsr_FSRid_fsr_load` for the (N)LFSRs. The presence of `fsr_FSRid_i_ext` and its control signal `fsr_FSRid_c_ext` depends on the given external step condition. The VHDL data types are defined in `fsr_pkg` for the FSR signals, and in `dp_pkg.vhd` for the top-level ports.

Example 11.3.5 *Grain – the top-level datapath* \longleftrightarrow

This example shows a small portion of the generated Grain datapath: signal declarations, component instantiation and combinational assignments for the top-level output port `o_data_1` and for the lfsr named `fsr_1`. The `o_data_1` output has an inferred multiplexer. Remaining VHDL code is removed for brevity. The full datapath for the Grain example, including the `dp_pkg.vhd`, the entity and entire architecture, is shown in Appendix D.1 in Example D.1.1.

VHDL Example 11.3.5

```
signal o_data_1_mux_1: top_out_1;
signal o_data_1_mux_1_src1: top_out_1;
signal o_data_1_mux_1_src2: top_out_1;
signal fsr_1_i_fsr: fsr_1_input;
signal fsr_1_i_ext: fsr_1_ext;
... OMITTED FOR BREVITY ....
begin

--submodules
m_fsr_1: entity work.fsr_1(main) port map (fsr_1_i_fsr, fsr_1_i_ext, clk,
    fsr_1_fsr_en, fsr_1_load, fsr_1_c_ext, fsr_1_o_fsr);
... OMITTED FOR BREVITY ....

--assignments
o_data_1 <= o_data_1_mux_1;
o_data_1_mux_1 <= o_data_1_mux_1_src1 when mux_1_cntl = '1' else o_data_1_mux_1_src2;
o_data_1_mux_1_src1 <= fsr_4_o_fsr;
o_data_1_mux_1_src2 <= o_data_1_zero;
fsr_1_i_fsr <= i_data_1;
fsr_1_i_ext <= fsr_3_o_fsr;
... OMITTED FOR BREVITY ....
```

\longleftarrow

Key 11.5: The structure of the FSR objects and the internal signals for the top-level datapath

For each FSR, at most three signals (Key 11.1) need to be declared in the top-level datapath module: `fsr_FSRid_i_fsr`, `fsr_FSRid_i_ext` and `fsr_FSRid_o_fsr`, i.e., the signals that also appear as the sources and targets in the spreadsheet template. There are three signals per each inferred multiplexer, namely `targetname_mux_#`, `targetname_mux_#_src1`, and `targetname_mux_#_src2`, where `targetname` is one of the target signals and `#` is replaced by the consecutive number of the stacked multiplexers.

11.4 Summary and conclusion

As was mentioned in Chapter 9, there are two *architectural decisions – automated design generation flows* through the automation framework. The first *architectural decisions – automated design generation flow* is focusing on synthesis for ciphers based on feedback shift registers. It consists of GAP packages FSR and FSRtoVHDL, whereby FSRtoVHDL relies on the package GAPtoVHDL. The key insights for the synthesis of FSR based circuits are recalled in Table 11.7.

The FSRtoVHDL package adopts the *field identifier* `Fid` from the GAPtoVHDL package and uses it in the same way (Key 10.1). In addition, an *FSR identifier* `FSRid` is used to uniquely identify FSR objects. Both identifiers are an abstraction to simplify the datapath generation. FSRtoVHDL generates three VHDL packages: `field_pkg.vhd`, `fsr_pkg.vhd`, and `dp_pkg.vhd`. They include clusters of type definitions formatted in the same manner: every field in the design has its own data type and constants, e.g., the zero element. Every FSR in the design has data types for its ports `i_fsr`, `i_ext`, and `o_fsr` (Key 11.1). Similarly, the top-level inputs and outputs are defined as a type and the synthesis uses the port identifier for datapath generation.

The synthesis of FSR-based systems relies on their modelling as a collection of FSRs and on structural similarities between the FSR objects (Keys 11.1 and 11.2). The FSRs can have at most three data ports. The configuration of the FSRs is provided as a spreadsheet, which is used to infer all internal datapath signals. The configuration must use the *connectors*, e.g., `merge`, `select`, etc., to specify how to use the individual (coordinates of) internal signals. The configuration is used to extract the connections, external step conditions, and multiplexers (Keys 11.3-11.5).

The FSRtoVHDL package contains an interactive function called `Manager`. The user must understand how to model the cipher as a collection of FSR objects and how to capture this information in a setup file. `Manager` is invoked with the setup file as its argument; it parses the setup file and writes an empty *spreadsheet template file* `*.tsv`. At this point, the *Manager* halts, and waits for the user instruction: user stores the configuration, then types “yes” into the GAP prompt. No further user interaction is required. The `Manager` function reads back the configuration and begins with synthesis for the top-level datapath.

The setup file for the Grain examples presented in this chapter (Examples 11.2.4 and 11.3.1-11.3.5) contains only 16 lines of GAP code (Example 11.3.2). The `Manager` generates 14 VHDL files for the implementation of entire Grain datapath.

Classification of the feedback expression (Key 10.4) is used to determine how to generate the FSR datapath. The FSRtoVHDL package has no means of generating submodules such as extension field multipliers. This limitation is bypassed by generating black-box feedbacks for NLFSRs and FILFUNs over extension fields. This is also the only chunk of generated code that is not synthesizable.

One option to include this functionality is to build a database of basic building blocks. Another solution is to synthesize the submodules on the fly. The latter is implemented as a part of the CIRCUIT package (Chapters 12-16), which can write arbitrary datapaths over arbitrary fields.

Key 11.1: The structure of the FSR objects	✓✓	Section 11.2
FSR objects are structured very consistently: each FSR has two data inputs, the loading input <code>i_fsr</code> and the external step input <code>i_ext</code> , and one output called <code>o_fsr</code> . Their VHDL data types are defined in <code>fsr_pkg.vhd</code> and easily accessible via the the <code>FSRid</code> .		
Key 11.2: The top-level datapath and configuration template	✓✓	Section 11.3
The top-level datapath is specified with a spreadsheet template that lists all possible sources as columns and all possible targets as rows. The possible sources are the top-level data inputs and the <code>o_fsr</code> outputs of all FSRs, and possible targets are the top-level outputs and both the <code>i_fsr</code> and the <code>i_ext</code> inputs of all FSRs in the cipher, see Key 11.1. The naming convention follows <code>fsr_FSRid_i_fsr</code> , <code>fsr_FSRid_i_ext</code> , and <code>fsr_FSRid_o_fsr</code> .		
Key 11.3: The top-level datapath: connections conditions and multiplexers	✓✓	Section 11.3
The configuration (Key 11.2) is used to extract the connections (internal signals), external step conditions, and multiplexers (inference). If possible, the multiplexers are already available inside FSRs are used. Sometimes, additional datapath multiplexers are inferred.		
Key 11.4: The structure of the FSR objects and the port map for component instantiations	✓✓	Section 11.3
The port map is obtained from the FSR object structure (Key 11.1) and the the external step condition. The positional port map will be generated in the same order as the entity ports for the FSR object in question. It will include the clock signal and the signals <code>fsr_FSRid_fsr_en</code> and <code>fsr_FSRid_fsr_load</code> for the (N)LFSRs. The presence of <code>fsr_FSRid_i_ext</code> and its control signal <code>fsr_FSRid_c_ext</code> depends on the given external step condition. The VHDL data types are defined in <code>fsr_pkg</code> for the FSR signals, and in <code>dp_pkg.vhd</code> for the top-level ports.		
Key 11.5: The structure of the FSR objects and the internal signals for the top-level datapath	✓✓	Section 11.3
For each FSR, at most three signals (Key 11.1) need to be declared in the top-level datapath module: <code>fsr_FSRid_i_fsr</code> , <code>fsr_FSRid_i_ext</code> and <code>fsr_FSRid_o_fsr</code> , i.e., the signals that also appear as the sources and targets in the spreadsheet template. There are three signals per each inferred multiplexer, namely <code>targetname_mux_#</code> , <code>targetname_mux_#_src1</code> , <code>targetname_mux_#_src2</code> , where <code>targetname</code> is one of the target signals and <code>#</code> is replaced by the consecutive number of the stacked multiplexers.		

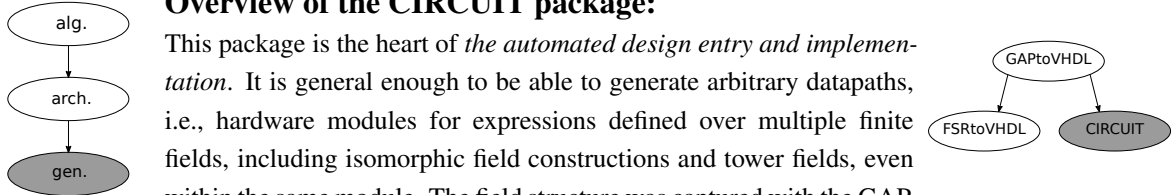
Notes: ✓✓ - solved

Table 11.7: Summary of key insights to the FSRtoVHDL package - listed chronologically

Chapter 12

CIRCUIT package: generating arbitrary datapaths

12.1 Overview



Overview of the CIRCUIT package:

This package is the heart of *the automated design entry and implementation*. It is general enough to be able to generate arbitrary datapaths, i.e., hardware modules for expressions defined over multiple finite fields, including isomorphic field constructions and tower fields, even within the same module. The field structure was captured with the GAP

objects `SignalDomain`, `SIGNAL`, and `SignalPkg`. They provide a way to systematically enter and build all the data types needed in a particular design, and allow a finite field to be viewed as a vector space. The `SignalPkg` is, for all practical purposes, just a list of `SIGNAL`s, i.e., data types, sorted following a top-down modular approach to hardware design: the submodules are expected for the subfields. The index at which a data type is stored is used as a *field identifier* `Fid`. The first step towards design automation is the functional description of the algorithm. It allows variable binding, classification of expression(s) given by the algorithm, and signal and submodule extraction (but not submodule generation). It provides the starting point for the automation, but not automation itself. The functional description, together with the `SignalPkg` and submodule instructions, is then transformed into a VHDL-ready design, that serves as a basis for the generation of VHDL files. The submodules are extracted and generated on the fly, using `FFCSA` package methods, listed as submodule instructions. The `CIRCUIT` package generates the entire datapath, including all submodules. For the generated modules and all their submodules, the testbenches, testvectors and simulation scripts are generated as well.

12.2 Roadmap

The `CIRCUIT` package is large and will be explained in many chapters (Chapters 12-16). The boxed “overview” sections and “summary of key insights” sections at the beginning and at the end of each chapter help to follow different stages through the design flow diagram from Figure 12.1. The part of the design flow in the magnifying glass will be used as orientation as well. As was already mentioned, the magnifying glass shows some intersection with *the architectural decisions*. The second *architectural decisions – automated design generation flow* involving the `CIRCUIT`

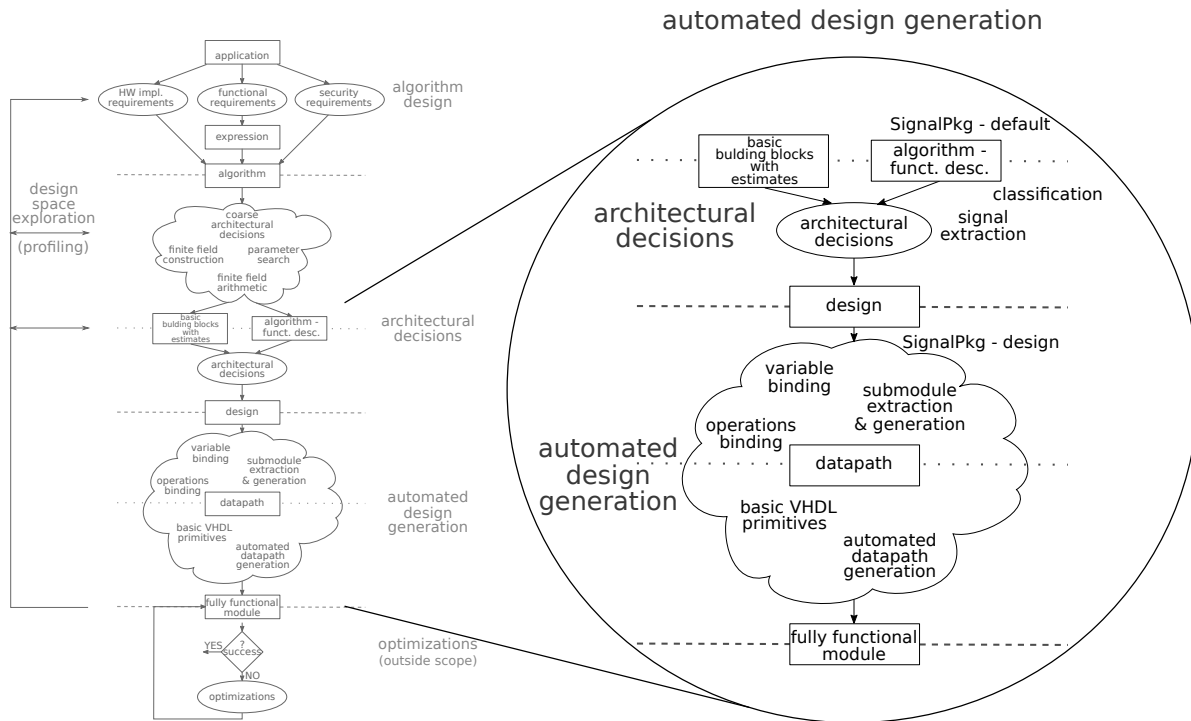


Figure 12.1: Design flow: *the automated design generation*

package will explain why the transition is not as strict as shown in the design flow diagram. Tables 12.1 and 12.2 show an overview of examples in Chapters 12-16, examples moved to appendix, and related examples (columns Related Ex.). The first column gives the subsection in which the example can be found, the second column “Ex.” the example number, the third column the example title and short description.

The structure of the remainder of Part IV is as follows:

- **CIRCUIT package part 1: arbitrary finite fields - Chapter 13**
 - Motivation (Section 13.1)
 - The signal domains and signals (Section 13.2)
 - The signal package - keeping it all in one place (Section 13.3)
 - Summary of key insights (Section 13.4)
- **CIRCUIT package part 2: functional description of the algorithm - Chapter 14**
 - The functional description of the algorithm (Section 14.1)
 - The datapath based on classification of expressions (Section 14.2)
 - Summary summary of key insights (Section 14.3)
- **CIRCUIT package part 3: VHDL-ready design - Chapter 15**
 - The initial design (Section 15.1)

- Top-down processing (Section 15.2)
- Bottom-up processing (Section 15.3)
- Connectors for the tower field elements and vectors (Section 15.4)
- Putting it all together (Section 15.5)
- Summary of key insights (Section 15.6)
- CIRCUIT package part 4: generating VHDL for the datapath - Chapter 16
 - VHDL packages (Section 16.1)
 - VHDL (sub)modules (Section 16.2)
 - Testbench generation (Section 16.3)
 - Switching the the field structure for profiling (Section 16.4)
 - Summary and conclusion (Section 16.5)

Section	Ex.	Title and keywords	GAP	VHDL	Related Ex.
CIRCUIT package part 1: arbitrary finite fields - Chapter13					
Section13.2 The signal domains and signals	13.2.1	Using different tower field constructions: why storing a construction trail across is necessary			D.2.1 D.2.2 D.2.4
Appendix D.2	D.2.1	Example 13.2.1 - continued	✓		
Section13.2 The signal domains and signals	13.2.2	SIGNAL for an expression defined over \mathbb{F}_{2^8}: why both, finite field (element) and vector space (vector) SIGNALs are needed			
Appendix D.2	D.2.2	SignalDomain and SIGNAL objects: example showing field elements and “just vectors”	✓		
Appendix D.2	D.2.3	SignalPkg example for \mathbb{F}_{2^4}	✓		D.5.3
Section13.3 The signal package	13.3.1	SignalPkg example for the tower field construction $\mathbb{F}_{((2^2)^2)^2}$: this SignalPkg will be used for the running example (13.1)	✓		D.2.4 13.3.2 15.2.1
Section13.3 The signal package	13.3.2	SignalPkg example for the tower field construction $\mathbb{F}_{((2^2)^2)^2}$ - continued: graphical representation of SignalPkg from Example 13.3.1			
Appendix D.2	D.2.4	SignalPkg example for different tower field constructions of $\mathbb{F}_{((2^2)^2)^2}$: both constructions from Example 13.2.1	✓		
CIRCUIT package part 2: functional description of the algorithm - Chapter14					
Appendix D.3	D.3.1	Functional description for the running example (13.1) expression defined over the tower field $\mathbb{F}_{((2^2)^2)^2}$	✓		

Table 12.1: Examples in Chapters 13 and 14

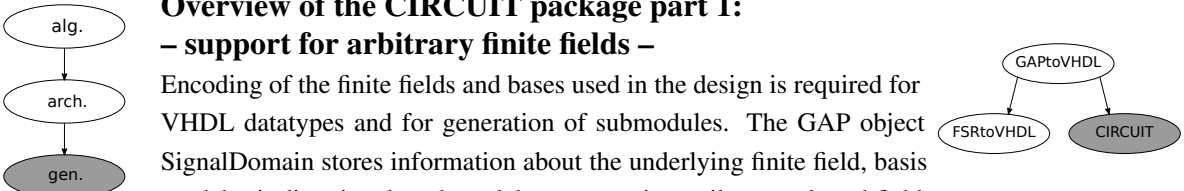
Section	Ex.	Title and keywords	GAP	VHDL	Related Ex.
CIRCUIT package part 3: VHDL-ready design - Chapter15					
Section 15.2 Top-down processing	15.2.1	Datapath synthesis for an expression over $\mathbb{F}_{((2^2)^2)^2}$ – top-down processing			15.3.1
Section 15.3 Bottom-up processing	15.3.1	Datapath synthesis for an expression over $\mathbb{F}_{((2^2)^2)^2}$ – bottom-up processing			
Appendix D.4	D.4.1	Example 15.3.1 continued: the VHDL-ready design	✓		
Section 15.3 Bottom-up processing	15.3.2	Declarations for additive constants in $\mathbb{F}_{((2^2)^2)^2}$: using directed graph for the constant		✓	
Section 15.4 Connectors	15.4.1	Connectors for the tower field elements and vectors in $\mathbb{F}_{((2^2)^2)^2}$	✓		16.1.1
Section 15.5 Putting it all together	15.5.1	Datapath synthesis for an expression over $\mathbb{F}_{((2^2)^2)^2}$ – revisited: summary of Ex. 15.2.1 and 15.3.1			
CIRCUIT package part 4: Generating VHDL for the datapath - Chapter16					
Appendix D.5	D.5.1	A full field_pkg.vhd example for $\mathbb{F}_{((2^2)^2)^2}$		✓	
Section 16.1 VHDL packages	16.1.1	The connectors_pkg.vhd for the type conversion functions: declarations		✓	16.2.1
Section 16.2 VHDL (sub)modules	16.2.1	Example of connectors and input registers in the architecture: type conversions		✓	
Section 16.2 VHDL (sub)modules	16.2.2	Example of architecture body for an expression defined over $\mathbb{F}_{((2^2)^2)^2}$: for the running example (13.1)		✓	
Section 16.2 VHDL (sub)modules	16.2.3	Example of registered <i>vector</i> output: submodule Fid2_mult for \mathbb{F}_{2^2} mult		✓	
Appendix D.5	D.5.2	Squaring for a polynomial basis: steps for writing the datapath, partial binding (Key 14.2)	✓		
Appendix D.5	D.5.3	A \mathbb{F}_{2^4} multiplier - full example: illustrates <i>AlgDesignWriteTop</i> , full binding (Key 14.2), the difference between <i>element</i> and <i>vector</i> ports	✓	✓	

Table 12.2: Examples in Chapters 15 and 16

Chapter 13

CIRCUIT package part 1: arbitrary finite fields

13.1 Motivation	163
13.2 The signal domains and signals	167
13.3 The signal package - keeping it all in one place	175
13.4 Summary of key insights	183



**Overview of the CIRCUIT package part 1:
– support for arbitrary finite fields –**

Encoding of the finite fields and bases used in the design is required for VHDL datatypes and for generation of submodules. The GAP object SignalDomain stores information about the underlying finite field, basis used, basis direction, length, and the construction trail across the subfields.

This amount of information is needed to capture tower field constructions and to distinguish isomorphisms. The GAP object SIGNAL allows to view a finite field as a vector space. It holds the SignalDomain and and the “vector (space)” parameters length, direction, and an (optional) interpretation basis. SIGNAL allows for the distinction of (field) *elements* and *vectors*. All the VHDL data types used in the design are the SIGNAL objects translated into VHDL. The SignalPkg object holds all SignalDomains and SIGNALs. The sorting of SIGNALs follows the top-down modular approach to hardware design: the submodules are expected for the subfields.

13.1 Motivation

13.1.1 The running example

The following example is used to explain some challenges related to implementing an arbitrary datapath, and to show that a new package with capabilities beyond GAPtoVHDL and FSRtoVHDL is needed. Consider the expression with indeterminates $a, b, c \in \mathcal{F}$ and two arbitrary nonzero coefficients $\gamma_1, \gamma_2 \in \mathcal{F}$:

$$z = \gamma_1 a \cdot b \cdot c + a^2 + \gamma_2 \tag{13.1}$$

Expression (13.1) is given in ANF (equation (10.1)), and there is no restriction on γ_1 and γ_2 being different. The expression (13.1) is considered over three different finite fields $\mathcal{F} \cong \mathbb{F}_q$, $q \geq 2$:

- \mathbb{F}_2 , discussed in example 13.1.1,
- \mathbb{F}_{2^8} , discussed in example 13.1.2, and
- $\mathbb{F}_{((2^2)^2)^2}$, discussed in example 13.1.3.

Expression (13.1) defined over $\mathbb{F}_{((2^2)^2)^2}$ (example 13.1.3) will be used as a running example throughout the remainder of Part IV.

Example 13.1.1 *Expression defined over \mathbb{F}_2*

When $a, b, c \in \mathbb{F}_2$, squaring is idempotent, i.e., $a^2 = a$ (recall the generalization of Fermat’s little theorem in equation (3.2)). Furthermore, since $\gamma_1, \gamma_2 \in \mathbb{F}_2$ are nonzero, the expression simplifies to

$$z = a \cdot b \cdot c + a + 1 \quad (13.2)$$

The \mathbb{F}_2 addition is implemented as an XOR gate and multiplication as an AND gate. All the ports and signals are 1 bit wide and implemented in VHDL as `std_logic`. The circuit schematic is shown in Figure 13.2(a).

Example 13.1.2 *Expression defined over \mathbb{F}_{2^8}*

When expression (13.1) is defined over \mathbb{F}_{2^8} , there are no simplifications. The field elements are represented using a basis of length 8, e.g., a polynomial basis. Then all the ports and signals are 8 bits wide and implemented in VHDL as `std_logic_vector(0 to 7)`. This example corresponds to the right side of the diagram in Figure 13.1.

The \mathbb{F}_{2^8} addition is implemented as bit-wise XOR, but submodules are needed for multiplication M , squaring SQ , and multiplication with constant $\times \gamma_1$. γ_1 is called *multiplicative constant* and γ_2 an *additive constant*, to differentiate between constants that infer submodules and constants that only require bit-wise XOR gates. The circuit schematic is shown in Figure 13.2(b).

Example 13.1.3 *Expression defined over $\mathbb{F}_{((2^2)^2)^2}$*

When expression (13.1) is defined over $\mathbb{F}_{((2^2)^2)^2}$, there are no simplifications.

Recall example 7.3.1 from Subsection 7.3.1. An element $X \in \mathbb{F}_{((2^2)^2)^2}$ is represented using a “per-level” polynomial basis of the form $B_{\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}} = \{1, \rho\}$, where ρ is a root of the extension defining polynomial $f_3(x)$: $X = x_0 + x_1\rho$, where $x_0, x_1 \in \mathbb{F}_{(2^2)^2}$. The 8-bit wide signals must now be represented as two coordinates of four bits each. Following Example 7.3.1, it is clear that elements in $\mathbb{F}_{(2^2)^2}$ must be represented as two coordinates of two bits each. Each of these two coordinates can be represented as `std_logic_vector(0 to 1)`. On each level of the tower, the elements will

be represented as arrays/vectors of length two, e.g., `array(0 to 1)`, but for $\mathbb{F}_{(2^2)^2}$ and $\mathbb{F}_{((2^2)^2)^2}$, the following question, shown graphically in Figure 13.1, arises: *array of ???*, i.e., *array of what?*. It is clear that the *what* will be the type of the subfield, but it is not yet clear how to capture this information in GAP.

The $\mathbb{F}_{((2^2)^2)^2}$ addition is implemented as bit-wise XOR, but submodules are needed for multiplication, squaring, and multiplication with constants. Because the multiplier is working over a tower field $\mathbb{F}_{((2^2)^2)^2}$, it will need multipliers in $\mathbb{F}_{(2^2)^2}$, which in turn will need multiplications in \mathbb{F}_{2^2} . Similarly, squarers and multipliers with constants will also need submodules.

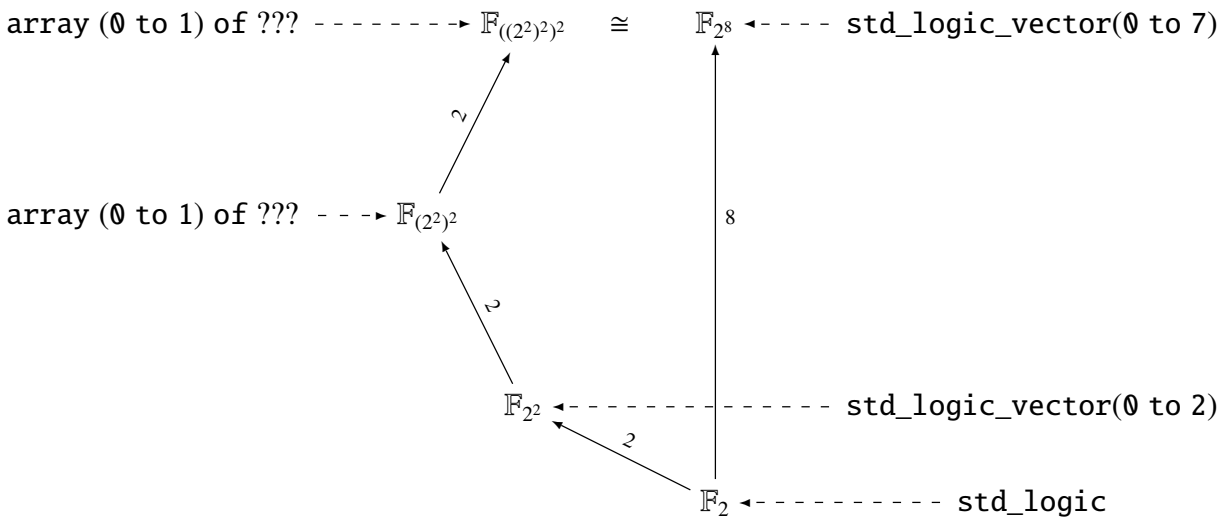


Figure 13.1: Construction of finite fields $\mathbb{F}_{((2^2)^2)^2}$ and \mathbb{F}_{2^8} and VHDL signals

13.1.2 Analysis

Examples 13.1.1, 13.1.2, and 13.1.3 identify the following problems:

- (a) are there algebraic simplifications (example 13.1.1),
- (b) how to obtain the submodules (examples 13.1.2 and 13.1.3)
- (c) which data type should be used for the VHDL signals? (example 13.1.3).

The problem (a) was addressed with miscellaneous methods from the FSR package `ReduceMonomialsOverField`¹ and `DegreeOfPolynomialOverField`. These methods are the first example of connecting an expression to a particular finite field.

¹ CIRCUIT package expects expressions to already have all their exponents reduced, i.e., no further simplifications like the one in example 13.1.1 are possible

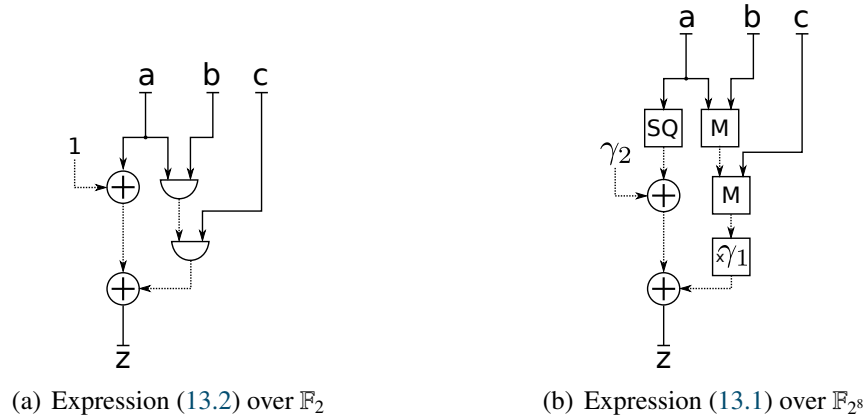


Figure 13.2: Circuit schematics (top-level modules) for expression (13.1): solid lines represent the ports and dashed lines the internal signals, boxes show the submodules needed: SQ - squarer, M - multiplier, $\times\gamma_1$ - multiplication with a constant γ_1

FSR objects operate over one field, stored as the attribute `UnderlyingField`: feedbacks and filtering functions are defined over that field. This is binding *per hardware module*, not per expression, although each FSR object has only one expression. The first step in addressing problems (b) and (c) is to *bind an expression to a finite field*. This binding is achieved by introducing a new GAP object² that will hold both the expression and the finite field over which it is defined.

To obtain the submodules, the underlying finite field and the basis used must be known (problem (b)). There are two options: (i.) fetch the appropriate submodule from a library, and (ii.) generate the submodule on the fly. The current CIRCUI package generates the submodules on the fly, using the FFCSA package methods listed in Table 7.5 to obtain the expressions for submodule implementation. The FFCSA methods for generating submodule expressions require the finite field and/or basis as input, but they do not store this information. Furthermore, as shown in Example 13.1.3, submodules must be generated for different levels of the tower field, implying more than one finite field is needed per design (problem(c)). Hence, there is no strict separation between problems (b) and (c).

Key 13.1: How to obtain the (sub)modules - binding expressions to finite fields

The first step in addressing this problem is to *bind the expressions to the finite fields over which they are defined*. The submodules are generated for the subfield operations. Knowing the subfield, its basis and direction, the expressions for the submodules are generated on the fly, using the FFCSA methods.

² AlgFunctionality, Section 14.1

13.2 The signal domains and signals

The encoding of the finite fields (and bases) used in the design is required for the implementation of VHDL ports and signals and for the submodules, as was identified in example in Section 13.1. In the case of FSR objects, the encoding of the finite field, i.e., which data type to use for the VHDL signals (motivation problem (c)), was addressed by storing a single `UnderlyingField` attribute. All ports and signals for the FSR objects were of elements or arrays of elements of the type given by the `UnderlyingField`. That means that as soon as the FSR object is created, so is the “world” it lives in.

The CIRCUIT package is intended to be more general and hence very powerful. Within the same top-level module, it should be possible to implement expressions

- defined over different finite fields, e.g., \mathbb{F}_{2^3} and \mathbb{F}_{2^8}
- defined over a tower field, e.g., $\mathbb{F}_{((2^2)^2)^2}$
- defined over several isomorphic finite fields using different bases, e.g., $\text{PB}_{\mathbb{F}_{2^8}/\mathbb{F}_2}$ and $\text{NB}_{\mathbb{F}_{2^8}/\mathbb{F}_2}$, including different tower field constructions

The field structure was captured by creating GAP objects `SignalDomain` (Subsection 13.2.1), `SIGNAL` (Subsection 13.2.2) and `SignalPkg` (Section 13.3). These objects provide a way to systematically enter and build all the data types needed in a particular design.

13.2.1 The SignalDomain

The `SignalDomain` object serves as an umbrella for `UnderlyingField`, with additional information about the basis and direction used. From a mathematical perspective, it seems strange to talk about the length³ or direction of a field element, two concepts that are very natural to a hardware designer. Similarly, from a hardware perspective, it is not common to talk about a basis; the basis is merely the mathematical interpretation of the signals.

The basis and direction must be fixed before writing the VHDL. Furthermore, they are mandatory for generating the submodules on the fly, which makes them the most important information. The length is simply the number of basis elements and is stored as attribute `Length` for convenience. Unless stated otherwise, this work assumes direction “to”. All attributes of `SignalDomain` are summarized in Table 13.1.

The last piece of information is the tower field construction itself, which is stored as a list of all previous `SignalDomain` objects. It forms the construction trail across the subfields, stored as the `SubSGDtower` attribute. New notation is introduced for `SignalDomain`: \mathcal{S}_r , where r is a counter, starting from 1, i.e., \mathcal{S}_1 for a `SignalDomain` object with \mathbb{F}_2 as the underlying finite field. Because multiple `SignalDomain` objects can have the same `UnderlyingField` attribute, as will be shown in Example 13.2.1, the usual notation with field size, e.g., $\mathbb{F}_{((2^2)^2)^2}$ and $\mathcal{S}_{((2^2)^2)^2}$, is not

SignalDomain \mathcal{S}_r	
attribute	comment
UnderlyingField	the finite field \mathbb{F}_q
UnderlyingBasis	basis B used for representation of elements of \mathbb{F}_q
BasisDirection	basis direction “to” or “downto”
Length	length of the basis B (degree of extension)
SubSGDtower	a list of SignalDomain objects for all subfields, ordered by increasing degree over the prime field, -1 when $q = 2$

Table 13.1: Main functionality of the CIRCUIT package - the SignalDomain attributes

enough to uniquely identify a particular signal domain. Example 13.2.1 illustrates why the tower field construction information is mandatory for successful VHDL implementation.

Key 13.2: The SignalDomain: the bases and the field construction trail

Encoding of the finite fields and bases used in the design is required for VHDL data types and for generation of submodules. The GAP object SignalDomain stores information about the underlying finite field, basis used, basis direction, length, and the construction trail across the subfields. This amount of information is needed to capture tower field constructions and to distinguish isomorphisms.

Example 13.2.1 Using different tower field constructions \longleftrightarrow

The diagram in Figure 13.3 shows how different constructions of the finite field $\mathbb{F}_{((2^2)^2)^2}$. The first and the last level of the tower are constructed in the same manner, using the same (only) irreducible polynomial and the same basis for the representation of the elements. The middle level, however, is constructed in two different ways, yielding two different constructions. The construction on the left branch of the diagram in Figure 13.3 shows the extension field defining polynomial $f_{2,1}(x) = x^2 + \lambda x + 1$, and the construction on the right branch of the extension field defining polynomial $f_{2,2}(x) = x^2 + \lambda x + \lambda$. The two polynomials have different roots $\mu_1 \neq \mu_2$, where $f_{2,1}(\mu_1) = 0$ and $f_{2,2}(\mu_2) = 0$. The distinction between the left and the right branch is very important: the submodules for computation using basis B_1 will differ from the submodules for computation using B_2 . For clarity, all the constants are given w.r.t. a reference field defining polynomial RDP, listed in Table D.1.

Reference finite field	Reference field defining polynomial - RDP $p_i(x)$	Root of RDP $p_i(x)$	Constants w.r.t. root of RDP
\mathbb{F}_{2^8}	$p_3(x) = x^8 + x^4 + x^3 + x^2 + 1$	$p_3(\nu) = 0$	$\nu_3 = \nu^{15}$
\mathbb{F}_{2^4}	$p_2(x) = x^4 + x + 1$	$p_2(\mu) = 0$	$\mu_1 = \mu^6, \mu_2 = \mu^7$
\mathbb{F}_{2^2}	$p_1(x) = x^2 + x + 1$	$p_1(\lambda) = 0$	

Table 13.2: Reference polynomials and their roots

³ length of the basis

The `UnderlyingField` attribute stores the information about the degree of extension for each level, e.g., `AsField(AsField(GF(2^2), GF(2^4)), GF(2^8))`, but not about the “per-level” bases (PLB) used. This information is captured by `SubSGDtower`. The GAP Example D.2.1 in Appendix D.2 shows the two tower field constructions from diagram in Figure 13.3 captured in GAP as `SignalDomain` objects.

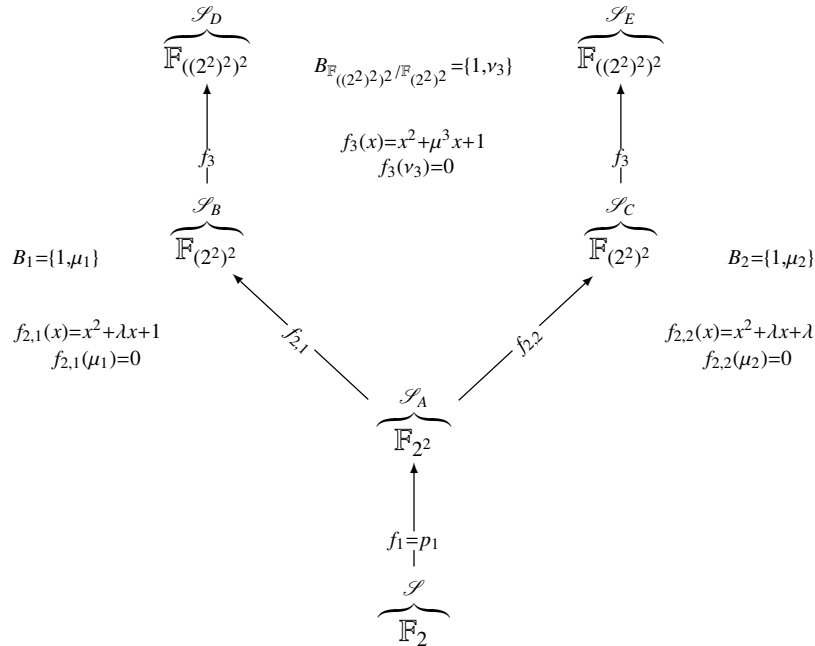


Figure 13.3: Two different tower field constructions

←

13.2.2 The finite field as a vector space and the SIGNAL object

`SIGNAL`⁴ is a new GAP object, that defines field elements or vectors. A vector is an array of given length, and its coordinates belong to a field. The simple case is an arbitrary vector (“just vector”), and a more complex case is a vector that belongs to a vector space (interpretation of a finite field). The object `SIGNAL` is a wrapper for `SignalDomain`, and stores additional information that allows for the distinction between elements and vectors. All VHDL data types needed for the datapath are `SIGNAL` objects, translated into VHDL.

While it is easy for the designer to switch between an input as an element of the finite field \mathbb{F}_{q^m} and vector space \mathbb{F}_q^m , for successful VHDL implementation this information has to be captured in GAP. The `SignalDomain` can be implicitly treated as a vector space, but it does not hold “just vectors”.

⁴ using the capital letters to differentiate the GAP object `SIGNAL` from a VHDL signal

The design can require signals that are vectors, without the notion of a vector space, e.g., the shift register for an FSR object or the input to function $h(x)$ in Grain [46].

■ **Discussion:** In the case of tower field construction, e.g., $\mathbb{F}_{((2^2)^2)^2}$, the problem is the type of the lower level, i.e., *array of what*, `array(0 to 1)` of ????. The problem was explained in more detail in motivation Examples 13.1.2 and 13.1.3, and represented graphically in the diagram in Figure 13.1. When the finite field is constructed as a single extension, e.g., \mathbb{F}_{2^8} , the ports and signals are defined as `std_logic_vector(0 to 7)`, which is equivalent to `array (0 to 7) of std_logic`. VHDL array (vector) types have coordinates⁵ that belong to some given *domain*. For \mathbb{F}_{2^m} the coordinates belong to `std_logic` and the array itself is of type `std_logic_vector(0 to m-1)`. The answer to the question *array of what?* is the given *domain*. In the case of the finite field \mathbb{F}_{q^m} , $q = 2^p$ and $p, m > 1$, the domain is \mathbb{F}_q and the datatype for implementing the field elements of \mathbb{F}_{q^m} an array (vector) type of length m . That is, the data type defines vectors of length m over \mathbb{F}_q , i.e., \mathbb{F}_q^m rather than \mathbb{F}_{q^m} . ■

Key 13.3: The SIGNAL: finite field as a vector space and “just vectors”

For the designer it is easy to switch between an input as element of the finite field \mathbb{F}_{q^m} and vector space \mathbb{F}_q^m , but for successful VHDL implementation this information has to be captured in GAP. Furthermore, the given algorithm can require “just vectors”, that are not elements of a given finite field, i.e., vector space, however, these vectors have coordinates from a given finite field, i.e., a given SignalDomain. The SignalDomain was embedded into another GAP object called SIGNAL, which holds the SignalDomain and the “vector (space)” parameters length m , the direction, and an (optional) interpretation basis. They allow a distinction of (field) *elements* and *vectors*. All the VHDL data types used in the design are SIGNAL objects, translated into VHDL.

There are two categories of SIGNALs, *elements* and *vectors*. The SIGNAL object is shown in Table 13.3, and columns mentioned below refer to this table. Let \mathcal{S}_r be the SignalDomain for \mathbb{F}_q , $q \geq 2$, and \mathcal{S}_1 will be used for the special case $q = 2$. The following distinction (and notation) of SIGNAL objects is based on the length m :

- an *element* is a SIGNAL of length $m = 1$ with the signal domain \mathcal{S}_r , denoted:

$$\mathcal{F}(\mathcal{S}_r, 1, -, -) \sim \mathcal{F}_r \quad (13.3)$$

It is a SIGNAL representation, i.e., embedding, of the SignalDomain \mathcal{S}_r , and hence an element of the finite field \mathbb{F}_q , represented by \mathcal{S}_r :

- for $q = 2$, the \mathcal{F}_1 represents \mathcal{S}_1 and hence elements of \mathbb{F}_2 (column 1).
- for $q = q'^p$ and $q' \geq 2$ and $p > 1$, \mathcal{F}_r represents \mathcal{S}_r and hence elements of the finite field \mathbb{F}_q , which is an extension the finite field $\mathbb{F}_{q'}$ (column 2).

⁵ word “coordinate” is preferred to the word “component” to distinguish between vector coordinates and the hardware components implemented as submodules

- a *vector* is a SIGNAL of length $m > 1$ over the signal domain \mathcal{S}_r , denoted:

$$\mathcal{V}(\mathcal{S}_r, m, d, B) \quad (13.4)$$

The coordinates of $\mathcal{V}(\mathcal{S}_r, m, d, B)$ belong to the finite field \mathbb{F}_q of the signal domain \mathcal{S}_r , but the vector itself can be either a vector space representation of the next extension (rn for r -next) \mathcal{S}_{rn} for \mathbb{F}_{q^m} with direction d and basis B , or a “just vector” over \mathbb{F}_q . If it is a vector space representation of a finite field \mathbb{F}_{q^m} , the basis is set to B , otherwise this component is $-$. Again, the distinction is made based on \mathbb{F}_q of the \mathcal{S}_r :

- for $q = 2$, $\mathcal{V}(\mathcal{S}_1, m, d, B)$ represents vectors over \mathbb{F}_2 (column 3)
- for $q > 2$, $\mathcal{V}(\mathcal{S}_r, m, d, B)$ represents vectors over an extension field \mathbb{F}_q (column 4)

Both *element* columns in Table 13.3 consider vectors of length 1 over the domain specified in the first half of Table 13.3: there is no notion of basis or direction (hence $-$ in rows 9 and 10). The two *vector (space)* columns consider the actual vectors of length m , with coordinates from the SignalDomain \mathcal{S} specified in the first half of the table. The row “vector (space)” \mathbb{F}_q^m , is implicitly defined by the underlying finite field \mathbb{F}_q (row 2) of its signal domain \mathcal{S}_r (row 1) and by the length m (row 8). This interpretation is also valid for the first two columns, with elements viewed as vectors of length $m = 1$. The “just vectors”, which do not belong to a vector space, also do not have a basis, hence the last row is the only row without the “always exists” checkmark \checkmark . When the vectors represent finite field elements, the basis of that field is considered as an *interpretation* of the vector coordinates (row 10, note [7] in Table 13.3).

VHDL has no notion of basis as such, and the interpretation basis is used by other GAP methods to generate the expressions for implementation of submodules and for the testbenches (note [4] in Table 13.3). The direction is only needed for the VHDL (row 9, note [3] in Table 13.3), and for some FFCSA package methods that have adopted the notion of direction from VHDL.

Table 13.4 shows three levels of a tower field construction $\mathbb{F}_{q'} \rightarrow \mathbb{F}_q \rightarrow \mathbb{F}_{q^m}$, where $q = q'^p$, the corresponding signal domains are $\mathcal{S}_{rp} \rightarrow \mathcal{S}_r \rightarrow \mathcal{S}_{rn}$. The notation rp stands for “ r previous” and rn for “ r next”. This general setting corresponds to column 4 in Table 13.3. The *element* \mathcal{F}_{rn} is the embedding of the signal domain \mathcal{S}_{rn} into a SIGNAL. The *vector* \mathcal{V}_{rn} is the vector space representation of the signal domain \mathcal{S}_{rn} ; but its coordinates belong to the subfield, i.e., \mathcal{S}_r . The UnderlyingBasis (row 5 in Table 13.3) of an *element* SIGNAL is the interpretation basis (row 10 in Table 13.3) of the *vector* SIGNAL obtained when viewing the finite field as a vector space. The following abbreviation is used for vector space notation:

$$\mathcal{V}(\mathcal{S}_r, m, d, B) \sim \mathcal{V}_{rn} \quad (13.5)$$

Similarly, when direction d is fixed, parameters can be omitted from “just vector” notations:

$$\mathcal{V}(\mathcal{S}_r, m, d, -) \sim \mathcal{V}(\mathcal{S}_r, m) \sim \mathcal{V}_r^m \quad (13.6)$$

The first column in Table 13.3 shows elements of the ground field \mathbb{F}_2 and has the same entries in both halves of the table. When viewing finite fields as vector spaces, there is an overlap between an *element* \mathcal{F}_r and *vector* \mathcal{V}_r in columns 2, 3, and 4 in Table 13.3 for the following cases:

	SIGNAL object and its interpretation		always exists	SIG. type	notes
	(field) element	vector (space)			
<i>attribute or component^c</i>	\mathcal{S}_1	\mathcal{S}_r			
signal domain notation					
UnderlyingField	\mathbb{F}_2	\mathbb{F}_q	\mathbb{F}_2	\mathbb{F}_q	[1]
Length	1	p	1	p	[2]
BasisDirection	-	to/downto	-	to/downto	[3]
UnderlyingBasis	-	$B_{\mathbb{F}_q/\mathbb{F}_d}$	-	$B_{\mathbb{F}_q/\mathbb{F}_d}$	[4]
SubSGDtower*	-1	$\mathcal{S}_{rp} \dagger$	-1	$\mathcal{S}_{rp} \dagger$	[5]
vector (space)					
length - m	\mathbb{F}_2^1	\mathbb{F}_q^1	\mathbb{F}_2^m	\mathbb{F}_q^m	[6]
basisdir ^c	1	1	m	m	[7]
basis ^c	-	-	to/downto	to/downto	[3]
	-	-	$B_{\mathbb{F}_{2m}/\mathbb{F}_2}$	$B_{\mathbb{F}_m/\mathbb{F}_q}$	[4, 7]
signal notation					
(possible) abbreviations	$\mathcal{F}(\mathcal{S}_r, 1, -, -)$	$\mathcal{F}(\mathcal{S}_r, 1, -, -)$	$\mathcal{V}(\mathcal{S}_1, m, d, B)$	$\mathcal{V}(\mathcal{S}_r, m, d, B)$	
	(13.3)	(13.3)	(13.5), (13.6)	(13.5), (13.6)	
VHDL \ddagger definition	(13.8)	(13.9), (13.10)	(13.11)	(13.12)	

SIGNAL

\mathcal{S}

Notes:

- [1] the UnderlyingField $\mathbb{F}_q, q \geq 2$
- [2] degree of extension $p = [\mathbb{F}_q : \mathbb{F}_q]$
- [5] SubSGDtower - the construction trail across the subfields, stored as a list of SignalDomain objects, with -1 for the prime fields only showing the last (highest) \mathcal{S} in SubSGDtower
- [6] vector or finite field as vector space
- [7] interpretation of coordinates if finite field \mathbb{F}_{q^m} viewed as vector space of dimension m, \mathbb{F}_q^m rp stands for “r previous”
- \dagger VHDL definitions in upcoming Section 13.3
- \ddagger VHDL definitions in upcoming Section 13.3

Table 13.3: GAP object SIGNAL with corresponding SignalDomain

finite field	$\mathbb{F}_{q'}$	\rightarrow	\mathbb{F}_q	\rightarrow	\mathbb{F}_{q^m}
signal domains	\mathcal{S}_{rp}	\rightarrow	\mathcal{S}_r	\rightarrow	\mathcal{S}_{rn}
elements	\mathcal{F}_{rp}		$\mathcal{F}(\mathcal{S}_r, 1, -, -) \sim \mathcal{F}_r$		$\mathcal{F}(\mathcal{S}_{rn}, 1, -, -) \sim \mathcal{F}_{rn}$
vectors	\mathcal{V}_{rp}		$\mathcal{V}(\mathcal{S}_{rp}, m, d, B) \sim \mathcal{V}_r$		$\mathcal{V}(\mathcal{S}_r, m, d, B) \sim \mathcal{V}_{rn}$

Table 13.4: The SIGNALs for three levels of a tower field

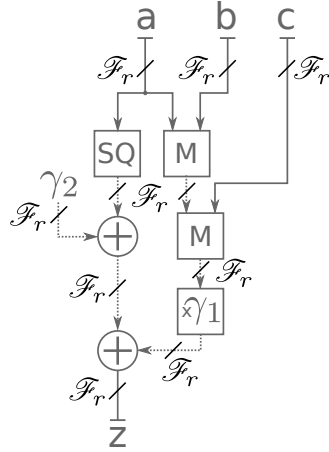
- $q = 2^p$ for column 2 and $m = p$ for column 3
- $p = m$ for column 2 and $q = q'$ for column 4

In both cases above, the *element* \mathcal{F}_r is the embedding of the signal domain \mathcal{S}_r into a SIGNAL, and the *vector* \mathcal{V}_r is the vector representation of the signal domain \mathcal{S}_r . For the first item above, $q = 2^p$, the subfield is the prime field, i.e., $q' = 2$ with the corresponding $\mathcal{S}_{rp} = \mathcal{S}_1$. For the second case, the subfield is itself an extension field, i.e., $q' > 2$, and its corresponding signal domain is \mathcal{S}_{rp} . Both the *element* \mathcal{F}_r and the vector space \mathcal{V}_r represent \mathcal{S}_r for $q = q'^m$.

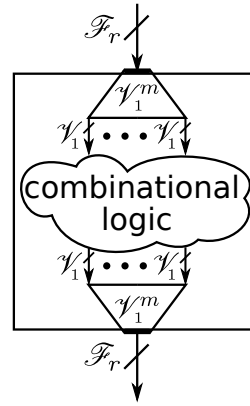
Because of this overlap, the implementation of a SIGNAL as both, a finite field (*element*) and a vector space (*vector*) seems redundant. The dual representation was chosen for automated submodule generation and checking purposes: there is a binding between the expressions defining a datapath and the ports of the hardware module. Recall Example 13.1.2, implementing expression (13.1): the ports and internal signals in the top-level module in Figure 13.4(a) are elements \mathcal{F}_r , where \mathcal{S}_r is the signal domain for \mathbb{F}_2^m , $m = 8$. Then, the submodules for squaring, multiplication and multiplication by a constant must be generated. The submodules have to implement *one expression per output coordinate*; hence, the submodule ports are declared as vectors $\mathcal{V}(\mathcal{S}_1, m, d, B) \sim \mathcal{V}_r$. Knowing the length of the output port allows for a correctness check: the number of expressions must match the Length (row 8) of the SIGNAL used for modelling the output port. For the top-level module, one expression, namely expression (13.1) is needed for an element output port of length 1. For a submodule, e.g., SQ, m expressions to drive m coordinates must be generated for a successful implementation. A submodule with annotations is shown in Figure 13.4(b) in Example 13.2.2.

Example 13.2.2 SIGNAL for an expression defined over \mathbb{F}_{2^8}

Example 13.1.2, implementing expression (13.1), is expanded to explain why both, a finite field (*element*) and a vector space (*vector*) SIGNALs are needed. Figure 13.4(a) shows the same circuit schematic as figure Figure 13.2, but annotated with *element* SIGNALs \mathcal{F}_r^1 , where \mathcal{S}_r is the signal domain for \mathbb{F}_2^m , $m = 8$. A submodule, e.g., squarer SQ, is shown in Figure 13.4(b): the submodule input and output port are *vector* SIGNALs $\mathcal{V}_r \sim \mathcal{V}_1^m$, which are internally shown as m SIGNALs \mathcal{V}_1 . From the submodule's point of view, the ports are vectors in a vector space (interpretation of a finite field).



(a) Top-level module with \mathcal{F}_r^1



(b) Submodule with \mathcal{F}_r^1 , \mathcal{V}_1^m and \mathcal{V}_1

Figure 13.4: Circuit schematics annotated with SIGNAL and *sub-signal* showing the elements and the vectors for: (a) top-level module for expression (13.1) and (b) a submodule with 1 input port and 1 output port

Key 13.4: How to obtain the (sub)modules - correctness check based on the output port length

The submodules have to implement one expression per output coordinate. Hence, the number of (generated) expressions must match the Length of the SIGNAL(s) used for modelling the output port(s) of the (sub)module.

13.2.3 Methods for SignalDomain and SIGNAL objects

There are several methods implemented for the SignalDomain and SIGNAL objects, listed in Table 13.5. SGDToSIGNAL embeds the SignalDomain \mathcal{S}_r into the SIGNAL \mathcal{F}_r . ExtractSIGNALType takes a SIGNAL sig and returns a list that contains the UnderlyingSignalDomain (rows 2-6 in Table 13.3), the Length, direction and basis (rows 8-10 in Table 13.3), i.e., the rows with \checkmark in the column “SIGNAL type” in Table 13.3. The information captured by the SIGNAL type is used to compare and manipulate the lists containing SIGNALS. It is also used as a short way of accessing the vital information stored by the SIGNAL. SIGNAL objects have some components not listed in Table 13.3, e.g., label, which are not used for the comparison. Example D.2.2 in Appendix D.2 shows the SignalDomain and SIGNAL objects for two signal domains and three signals for elements \mathcal{F}_1 and for the “just vector” \mathcal{V}_2^5 .

Recall the tower field construction $\mathbb{F}_{q'} \rightarrow \mathbb{F}_q \rightarrow \mathbb{F}_{q^m}$, where $q = q'^p$ and $q' \geq 2$ and $p, m > 1$, shown in the first two rows of Table 13.4. The expanded notation for *elements* and *vectors* in the last two rows nicely shows that the \mathcal{F}_r is the *element* representation of the “current” signal domain, but *vector* $\mathcal{V}(\mathcal{S}_r, m, d, B)$ is the vector space representation of the “next” signal domain \mathcal{S}_{rn} . The *vector* is sorted by its coordinates, i.e., the “current” signal domain \mathcal{S}_r . All SIG-

object	method	description	notes
SignalDomain	\ = SGDToSIGNAL	comparing two SignalDomain objects embed SignalDomain into SIGNAL	
SIGNAL	ExtractSIGNALType	return the <i>SIGNAL type</i>	[1]
	SameSIGNALType	comparing the <i>types</i> of two SIGNALs	[1]
	DuplicateFreeSIGNALList	remove duplicates	[2,5]
	SortedSIGNALList	first sort by Length, second by <i>F</i>	[2,3,4,5]

Notes:

- | | |
|--|--|
| [1] see Table 13.3 column <i>SIGNAL type</i> | [2] based on the SIGNAL type |
| [3] $F = \text{UnderlyingField}(\text{UnderlyingSignalDomain}(sig))$
for each SIGNAL <i>sig</i> | [4] adds any missing signals to the list |
| | [5] see Key 13.5 |

Table 13.5: Main functionality of the CIRCUIT package - methods for the SignalDomain and the SIGNAL

NALs with UnderlyingSignalDomain \mathcal{S}_r are sorted together, which implies the order $\mathcal{F}_r, \mathcal{V}_{rn}$, because $1 < m$. Assuming the list of SIGNALs contains the *element* and *vector* SIGNALs for all three fields (rows 3 and 4 in Table 13.4), the order produced by SortedSIGNALList is $\mathcal{V}_{rp}, \mathcal{F}_{rp}, \mathcal{V}_r, \mathcal{F}_r, \mathcal{V}_{rn}, \mathcal{F}_{rn}$.

13.3 The signal package - keeping it all in one place

To keep all the field(s) information in one place, a new object, called SignalPkg, is created. It stores all SIGNALs used in the design as a list. The SignalPkg is parsed and used as input for the GAPtoVHDL function Write_field_pkg listed in Table 10.1: every signal type, encountered in the SignalPkg object, becomes a VHDL data type.

The SignalPkg has three components to hold three lists, SignalDomainList, SIGNALList, and SIGNALTypeList, and methods for the creation and entry of signals (see Table 16.1):

- SignalPkg() is the constructor call for a signal package with signal domain \mathcal{S}_1 and the signal \mathcal{F}_1 for elements of the finite field \mathbb{F}_2
- DefaultSignalPkg(m) adds the signal for the elements of the finite field \mathbb{F}_{2^m} using the polynomial basis with direction “to”, and the signal for the “just vectors”⁶ of length $2m - 1$ and direction “to” with coordinates from \mathbb{F}_2 , i.e., it returns a signal package with ordered SIGNALs $\mathcal{F}_1, \mathcal{V}_r, \mathcal{V}_1^{2m-1}, \mathcal{F}_r$, where \mathcal{S}_r is the signal domain for \mathbb{F}_{2^m} .
- AddFieldToSignalPkg(pkg, sgd, B, dir): let \mathbb{F}_q be an extension of degree m over $\mathbb{F}_{q'}$ and B the basis of $\mathbb{F}_q/\mathbb{F}_{q'}$. Let $sgd = \mathcal{S}_{rp}$ be the signal domain of the base field $\mathbb{F}_{q'}$ and \mathcal{S}_r the signal domain of the extension \mathbb{F}_q . AddFieldToSignalPkg adds both the *vector* \mathcal{V}_r (with interpretation basis B) and the *element* \mathcal{F}_r to the signal package pkg .

⁶ in the case two-step classic multiplication is used

- `AddVectorToSignalPkg(pkg, sgd, m, dir)` adds the *vector* \mathcal{V}_r^m (without any interpretation basis) to the signal package *pkg*, where $sgd = \mathcal{S}_r$.
- `AddTowerFieldToSignalPkg(pkg, Blist, dirlist)` adds the following:
 - all the isomorphic finite fields with TFB with direction “to” (recall equation (7.1) in Section 7.3.1) - all of them added over the signal domain \mathcal{S}_1
 - all levels of the tower field construction using the “per-level” bases from *Blist*, starting with the signal domain \mathcal{S}_1 , and then using each added level as the new signal domain for the next level

All of the adding methods ensure that the lists in a given signal package are duplicate free and sorted. The signals are sorted first by their attribute `Length`, and then by the `UnderlyingField` of their signal domains (`SortedSIGNALList` in Table 13.5). Sorting by the `UnderlyingField` follows the *top-down modular approach to hardware design: the submodules are expected for the subfields* (recall Example D.2.2). The index at which the signal is stored in `SIGNALList` is used as a *field identifier* `Fid`. The field identifier serves as input to numerous methods defined for the `SignalPkg` object (Table 13.6), and, more importantly, for *binding*, as will be explained in detail in Section 14.1. The methods shown in the right half of Table 13.6 are used to traverse the `SignalPkg` and form the framework needed for the extraction and generation of the submodules. The use of method `AddTowerFieldToSignalPkg` is shown in Example 13.3.1, and the use of methods listed in the right half of the Table 13.6 is shown in Example 13.3.2.

<i>pkg</i>	- a <code>SignalPkg</code> object	<i>sgd</i>	- a <code>SignalDomain</code> object
<i>B</i> , <i>Blist</i>	- basis, list of bases	<i>dir</i> , <i>dirlist</i>	- direction (string), list of directions
<i>Fid</i>	- field identifier (integer)	<i>m</i>	- degree of extension/length

SignalPkg methods	
<code>SignalPkg()</code>	<code>FieldIDToSubID^[1](<i>pkg</i>, <i>Fid</i>)</code>
<code>DefaultSignalPkg(<i>m</i>)</code>	<code>VectorIDToSubID^[2](<i>pkg</i>, <i>Fid</i>)</code>
<code>AddFieldToSignalPkg(<i>pkg</i>, <i>sgd</i>, <i>B</i>, <i>dir</i>)</code>	<code>FieldIDToVectorID(<i>pkg</i>, <i>Fid</i>)</code>
<code>AddVectorToSignalPkg(<i>pkg</i>, <i>sgd</i>, <i>m</i>, <i>dir</i>)</code>	<code>VectorIDToSGDFieldID^[3](<i>pkg</i>, <i>Fid</i>)</code>
<code>AddTowerFieldToSignalPkg(<i>pkg</i>, <i>Blist</i>, <i>dirlist</i>)</code>	<code>TFieldIDToFieldID(<i>pkg</i>, <i>Fid</i>)</code>
<code>Print(<i>pkg</i>)</code> , <code>PrintDownto(<i>pkg</i>)</code>	<code>PrintAll(<i>pkg</i>)</code> , <code>PrintAllDownto(<i>pkg</i>)</code>

Notes:

- | | |
|-------------------------------|--|
| [1] - Sub stands for subfield | [3] - for vector $\mathcal{V}_{\mathcal{S}_r}^m / \mathcal{V}_{\mathcal{S}_r}$, return the signal $\mathcal{F}_{\mathcal{S}_r}^1$ |
| [2] - Sub stands for subspace | corresponding to the signal domain \mathcal{S}_r |

Table 13.6: Main functionality of the CIRCUIT package - methods for the `SignalPkg`

Key 13.5: The signal package SignalPkg - keeping it all in one place

The SignalPkg object holds all *SignalDomains* and SIGNALs used in the design. The *SignalDomains* are duplicate free, the SIGNALs are duplicate free and sorted. The sorting of SIGNALs follows *the top-down modular approach to hardware design: the submodules are expected for the subfields*. The index at which the signal is stored in the list is used as a *field identifier* Fid, which is used for binding expressions to finite fields.

To explain how the VHDL type definitions are written, a pair of *SIGNALs* is used:

$$signal / sub - signal \quad (13.7)$$

where the *sub - signal* corresponding to a given *signal* can be obtained with FieldIDToSubID and VectorIDToSubID (Table 13.6). The actual strings used as *signal* and *sub-signal* are obtained using the GAPtoVHDL method Get_ffe_strings in Table 10.1.

Table 13.7 contains the VHDL type definitions, assuming directions “to”. It is split into *element* (upper half) and *vector* (lower half) SIGNALs, with further distinction based on the (sub)field size (column 1 in Table 13.7). For the example $q' = 2$, $\mathcal{F}_r/\mathcal{F}_1$ gives defn. (13.9) in row 2 in Table 13.7, which is the data type for the elements of \mathbb{F}_{2^p} (column 2 in Table 13.3). For the *vector* signals, two different notations are used in case of “just vectors” (notation (13.6)) and and vector spaces (notation (13.5)), which are merged to \mathcal{V} for the purpose of *signal/sub-signal* notation and actual type definitions (13.11) and (13.12), but the distinction is listed in the comments. Column 4 links Table 13.7 to Table 13.3. The VHDL keywords `type` and `subtype` are omitted from Table 13.7 for brevity but are shown in Table 13.8. As was mentioned before, when viewing finite fields as vector spaces, there is an overlap between columns 2, 3, and 4 in Table 13.3. This overlap is clearly reflected in the VHDL definitions (13.9-13.12) in Table 13.7, and the definitions can be merged as shown in Table 13.8. Note that in both cases the overlap conditions are related to the subfield, specifically to the presence of a tower field construction:

- for $q' = 2$ the overlap conditions are $q = 2^p$ for column 2 and $m = p$ for column 3 in Table 13.3: when no tower field is present, both elements and vectors use `std_logic_vector` - defn. (13.13) in first row in Table 13.8. It merges definitions (13.9) and (13.11), i.e., $\mathcal{F}_r/\mathcal{F}_1$ and $\mathcal{V}/\mathcal{V}_1$.
- for $q' > 2$ the overlap conditions are $p = m$ for column 2 and $q = q'^m$ for column 4 in Table 13.3: in the presence of a tower field, both elements and vectors use `array` - defn. (13.14) in second line in Table 13.8. It merges definitions (13.10) and (13.12), i.e., $\mathcal{F}_r/\mathcal{F}_{rp}$ and $\mathcal{V}/\mathcal{V}_{rp}$.

Key 13.6: The VHDL type definitions and tower field constructions

In the presence of a tower field, the SIGNALs on the higher levels of the tower are defined as `array`. When no tower field is present, the SIGNALs are defined as `std_logic_vector`.

q or q'	notation (13.7)	VHDL type definition	Table 13.3 column	comments
<i>elements, $m = 1$</i>				
$q = 2$	$\mathcal{F}_1 \dagger$	\mathcal{F}_1 is std_logic (13.8)	column 1	elements of \mathbb{F}_2 [1]
$q' = 2$	$\mathcal{F}_r/\mathcal{F}_1$	\mathcal{F}_r is std_logic_vector(0 to $p-1$) (13.9)	column 2	elements of $\mathbb{F}_q, q = 2^p$ [1, 2]
$q' > 2$	$\mathcal{F}_r/\mathcal{F}_{rp}$	\mathcal{F}_r is array(0 to $p-1$) of \mathcal{F}_{rp} (13.10)	column 2	elements of $\mathbb{F}_q, q = q'^p$ [2, 3]
<i>vectors, $m > 1$</i>				
$q = 2$	$\mathcal{V}_1 \dagger$	not a vector but an element (\mathcal{F}_1 , defn.(13.8)); \mathcal{V}_1 is kept as <i>sub-signal</i> notation.		
$q' = 2$	$\mathcal{V}/\mathcal{V}_1$	\mathcal{V} is std_logic_vector(0 to $m-1$) (13.11)	column 3	$\mathcal{V} \sim \mathcal{V}_1^m$ “just vectors” \mathbb{F}_2^m $\mathcal{V} \sim \mathcal{V}_r$ vector space \mathbb{F}_2^m [1, 2, 4]
$q' > 2$	$\mathcal{V}/\mathcal{V}_{rp}$	\mathcal{V} is array(0 to $m-1$) of \mathcal{V}_{rp} (13.12)	column 4	$\mathcal{V} \sim \mathcal{V}_{rp}^m$ “just vectors” \mathbb{F}_q^m $\mathcal{V} \sim \mathcal{V}_r$ vector space \mathbb{F}_q^m [3] [2, 3, 5]

Notes:

- [1] \mathcal{S}_1 for \mathbb{F}_2 [2] \mathcal{S}_r for \mathbb{F}_q [3] \mathcal{S}_{rp} for \mathbb{F}_q [4] vector space $\mathbb{F}_2^m \cong \mathbb{F}_q, q = 2^m$ [5] vector space $\mathbb{F}_q^m \cong \mathbb{F}_q, q = q'^m$
 \dagger exception to notation (13.7) - there is no notion of *sub-signal*

Table 13.7: VHDL type definitions for different SIGNAL objects

overlap conditions	tower field	VHDL type definitions	overlap w.r.t. Table 13.3	overlap definitions w.r.t. Table 13.7
$q = 2^p$ $m = p$	\times	subtype <i>signal</i> is std_logic_vector(0 to $p-1$) (13.13)	column 2 column 3	element $\mathcal{F}_r/\mathcal{F}_1$ (defn. (13.9)) \dagger vector $\mathcal{V}/\mathcal{V}_1$ (defn. (13.11))
$q = q'^m$ $p = m$	\checkmark	type <i>signal</i> is array(0 to $m-1$) of <i>sub-signal</i> (13.14)	column 2 column 4	element $\mathcal{F}_r/\mathcal{F}_{rp}$ (defn. (13.10)) \dagger vector $\mathcal{V}/\mathcal{V}_{rp}$ (defn. (13.12))

Notes: \dagger for the appropriate \mathcal{V} notation, please refer to the comments column in the last two rows of Table 13.7

Table 13.8: VHDL type definitions for different SIGNAL objects - continued

The SignalPkg and especially the field identifier `Fid` will be used throughout the following sections. The structure of the SignalPkg is crucial for automated submodule extraction and generation (recall Key 13.5). Examples 13.3.1 and 13.3.2 show the SignalPkg for the tower field construction $\mathbb{F}_{((2^2)^2)^2}$ used for the running example (13.1). The diagram in Figure 13.5, shown in Example 13.3.2 explains the structure of the SignalPkg for $\mathbb{F}_{((2^2)^2)^2}$ graphically, and should not be skipped. A full example of the $\mathbb{F}_{((2^2)^2)^2}$ VHDL package is shown in Example D.5.1 Appendix D.

Example 13.3.1 *SignalPkg example for the tower field construction* $\mathbb{F}_{((2^2)^2)^2}$

Example 13.3.1 uses the tower field from Example 13.2.1, shown in left branch of the diagram in Figure 13.3, to demonstrate the use of method `AddTowerFieldToSignalPkg`. This tower construction will be used as the $\mathbb{F}_{((2^2)^2)^2}$ construction for the running example expression (13.1). The details of the construction are listed in Table 13.9: the bases BA, BB, BD are “per-level” bases, with two basis elements each. Besides adding all the “per-level” bases, the `AddTowerFieldToSignalPkg` also computes and adds the bases of isomorphic finite fields, obtained as was shown in equation (7.1) in Section 7.3.1:

$$\begin{aligned} TFB_{\mathbb{F}_{2^8}/\mathbb{F}_2} &= \{1, \lambda, \mu_1, \mu_1\lambda, \nu_3, \nu_3\lambda, \nu_3\mu_1, \nu_3\mu_1\lambda\} \\ TFB_{\mathbb{F}_{2^4}/\mathbb{F}_2} &= \{1, \lambda, \mu_1, \mu_1\lambda\} \end{aligned}$$

These two fields have the signal domains \mathcal{S}_4 for \mathbb{F}_{2^8} with basis $TFB_{\mathbb{F}_{2^8}/\mathbb{F}_2}$ (sgd # 4) and \mathcal{S}_3 for \mathbb{F}_{2^4} with basis $TFB_{\mathbb{F}_{2^4}/\mathbb{F}_2}$ (sgd # 3), both with direction “to”. Corresponding elements added are \mathcal{F}_4 and \mathcal{F}_3 with field identifiers `Fid=10` and `Fid=7`, and their corresponding vectors \mathcal{V}_4 and \mathcal{V}_3 with field identifiers `Fid=4` and `Fid=3` respectively. The entire output in Example 13.3.1 is returned by the method `PrintDownto` listed in Table 16.1, with *downto* ordering of all lists, to correspond with the “top-down” modular approach to hardware design.

$$\mathbb{F}_2 \xrightarrow{f_1(x)} \mathbb{F}_{2^2} \xrightarrow{f_{2,1}(x)} \mathbb{F}_{(2^2)^2} \xrightarrow{f_3(x)} \mathbb{F}_{((2^2)^2)^2} \quad \Rightarrow \quad \mathbb{K} \xrightarrow{fa} \mathbb{F}_A \xrightarrow{fb} \mathbb{F}_B \xrightarrow{fd} \mathbb{F}_D$$

Finite field	Extension defining polynomial - EDP $f_i(x)$	“per-level” PB $B_{\mathbb{F}_{q^2}/\mathbb{F}_q} = \{1, \rho\}$	Comments and GAP example labels
\mathbb{F}_{q^2}			$f_i(\rho) = 0$ field polynomial basis
$\mathbb{F}_{((2^2)^2)^2}$	$f_3(x) = x^2 + \mu^3 x + 1$	$B_{\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}} = \{1, \nu_3\}$	$f_3(\nu_3) = 0$ FD fd BD
$\mathbb{F}_{(2^2)^2}$	$f_{2,1}(x) = x^2 + \lambda x + 1$	$B_1 = \{1, \mu_1\}$	$f_{2,1}(\mu_1) = 0$ FB fb BB
\mathbb{F}_{2^2}	$f_1(x) = x^2 + x + 1$	$B_{\mathbb{F}_2/\mathbb{F}_2} = \{1, \lambda\}$	$f_1(\lambda) = 0$ FA fa BA

Table 13.9: Tower construction of $\mathbb{F}_{((2^2)^2)^2}$ - left branch of diagram in Figure 13.3

$\nu_3 = Z(2^8)^{15}$	BD := [Z(2)^0, Z(2^8)^15]
$\mu_1 = Z(2^4)^6$	BB := [Z(2)^0, Z(2^4)^6]
$\lambda = Z(2^2)$	BA := [Z(2)^0, Z(2^2)]
$TFB_{\mathbb{F}_{2^8}/\mathbb{F}_2} =$	[Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11, Z(2^8)^15, Z(2^8)^100, Z(2^8)^117, Z(2^8)^202]
$TFB_{\mathbb{F}_{2^4}/\mathbb{F}_2} =$	[Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11]

Table 13.10: Tower construction of $\mathbb{F}_{((2^2)^2)^2}$ - roots and bases in GAP native representation

In GAP Example D.2.4 in Appendix D.2 a signal package containing both branches of the diagram in Figure D.1 is shown. Example D.2.4 shows the `PrintAllDownto` listed in Table 16.1. For clarity, values of the roots and bases are given in GAP native form in Table 13.10.

Example 13.3.1

```

gap> defaultPkg := SignalPkg();;  Blist := [BA, BB, BD];;
warning: SubSGDtower is -1
gap> AddTowerFieldToSignalPkg(defaultPkg, Blist, ["to", "to", "to"]);;
gap> PrintDownto(defaultPkg);

-----
SignalPkg with
11:  elements in      AsField( AsField( GF(2^2), GF(2^4) ), GF(2^8) )      with
                                     direction to basis [ Z(2)^0, Z(2^8)^15 ]
10:  elements in      GF(2^8)      with direction to basis [ Z(2)^0, Z(2^2),
                                     Z(2^4)^6, Z(2^4)^11, Z(2^8)^15, Z(2^8)^100,
                                     Z(2^8)^117, Z(2^8)^202 ]
9:   vectors over     AsField( GF(2^2), GF(2^4) )      of length 2 and direction
                                     to - interpretation basis [ Z(2)^0, Z(2^8)^15 ]
8:   elements in      AsField( GF(2^2), GF(2^4) )      with direction to basis
                                     [ Z(2)^0, Z(2^4)^6 ]
7:   elements in      GF(2^4)      with direction to basis [ Z(2)^0, Z(2^2),
                                     Z(2^4)^6, Z(2^4)^11 ]
6:   vectors over     GF(2^2)      of length 2 and direction to - interpretation
                                     basis [ Z(2)^0, Z(2^4)^6 ]
5:   elements in      GF(2^2)      with direction to basis [ Z(2)^0, Z(2^2) ]
4:   vectors over     GF(2)        of length 8 and direction to - interpretation
                                     basis [ Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11,
                                     Z(2^8)^15, Z(2^8)^100, Z(2^8)^117, Z(2^8)^202 ]
3:   vectors over     GF(2)        of length 4 and direction to - interpretation
                                     basis [ Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11 ]
2:   vectors over     GF(2)        of length 2 and direction to - interpretation
                                     basis [ Z(2)^0, Z(2^2) ]
1:   elements in      GF(2)

```

Example 13.3.2 *SignalPkg example for the tower field construction* $\mathbb{F}_{((2^2)^2)^2}$ - continued

Figure 13.5 shows an enhanced diagram of all the signal domains and signals (both element and vector signals) in the current signal package `defaultPkg` from Example 13.3.1, i.e., the $\mathbb{F}_{((2^2)^2)^2}$ construction for the running example expression (13.1). Diagram in Figure 13.5 is explained in layers, starting with the inner layer:

1. the innermost layer are the six signal domains $\mathcal{S}_1, \dots, \mathcal{S}_6$, annotated in the overbraces of their finite fields:
 - tower field construction with extension defining polynomials is shown on the left path:

$$\mathbb{F}_2 \xrightarrow{f_1(x)} \mathbb{F}_{2^2} \xrightarrow{f_{2,1}(x)} \mathbb{F}_{(2^2)^2} \xrightarrow{f_3(x)} \mathbb{F}_{((2^2)^2)^2}$$
 - with signal domains $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_5$ and \mathcal{S}_6 (“per-level bases”)
 - their isomorphic finite fields: $\mathbb{F}_2 \rightarrow \mathbb{F}_{2^8}$ with basis $TFB_{\mathbb{F}_{2^8}/\mathbb{F}_2}$ and $\mathbb{F}_2 \rightarrow \mathbb{F}_{2^4}$ with basis $TFB_{\mathbb{F}_{2^4}/\mathbb{F}_2}$ - with signal domains \mathcal{S}_4 and \mathcal{S}_3 (there are no defining polynomials)
2. the middle layer to the left and the right of the signal domains are the *element* signals, corresponding to the 6 signal domains. The elements are signal representations of signal domains, and are always shown in the same line as the signal domain they represent, e.g., \mathcal{F}_6 is shown to the left of its `UnderlyingSignalDomain` \mathcal{S}_6 and \mathcal{F}_4 is shown to the right of its `UnderlyingSignalDomain` \mathcal{S}_4 .
 - notation (13.7) is used for elements: *signal/sub – signal*, e.g., $\mathcal{F}_r/\mathcal{F}_{rp}$. For $rp > 1$, the VHDL type definition (13.10) is used, and for $rp = 1$, i.e., *sub-signal* is \mathcal{F}_1 , the VHDL type definition (13.9).
 - the *sub – signal* is also identified with arrow \longleftarrow representing the action of `FieldIDToSubID`, e.g., `FieldIDToSubID(defaultPkg, 11)` returns 8.
3. the outermost layer are the *vector* representations of the elements in the middle layer, e.g., vector \mathcal{V}_6 is shown to the left of the element signal \mathcal{F}_6 .
 - notation (13.7) *signal/sub – signal* and abbreviation (13.5) are used for vectors: $\mathcal{V}_r/\mathcal{V}_{rp}$. For $rp > 1$, the VHDL type definition (13.12) is used, and for $rp = 1$, i.e., *sub-signal* is \mathcal{V}_1 , the VHDL type definition (13.11).
 - the *sub – signal* is also identified with arrow \longleftarrow representing the action of `VectorIDToSubID`, e.g., `VectorIDToSubID(defaultPkg, 9)` returns 6.

The aforementioned methods `FieldIDToSubID` and `VectorIDToSubID` allow a top-down walk through the diagram, with `FieldIDToSubID` descending down the middle layers (\longleftarrow on the *element* signals), and `VectorIDToSubID` descending down the outermost layers (\longleftarrow on the *vector* signals).

The diagram also shows the action of the method `FieldIDToVectorID` with \Leftarrow arrows, which allows the passage from the middle to the outermost layer by returning the field identifier of the vector representation of a given element, e.g., `FieldIDToVectorID(defaultPkg, 11)` returns 9.

Last method shown in this example is the `FieldIDToSGDFieldID`, marked with \leftarrow , which for a given vector identifies the element representative of its underlying signal domain, e.g., `FieldIDToSGDFieldID(defaultPkg, 9)` returns 8.

These methods together form the framework needed for the extraction and generation of the submodules.

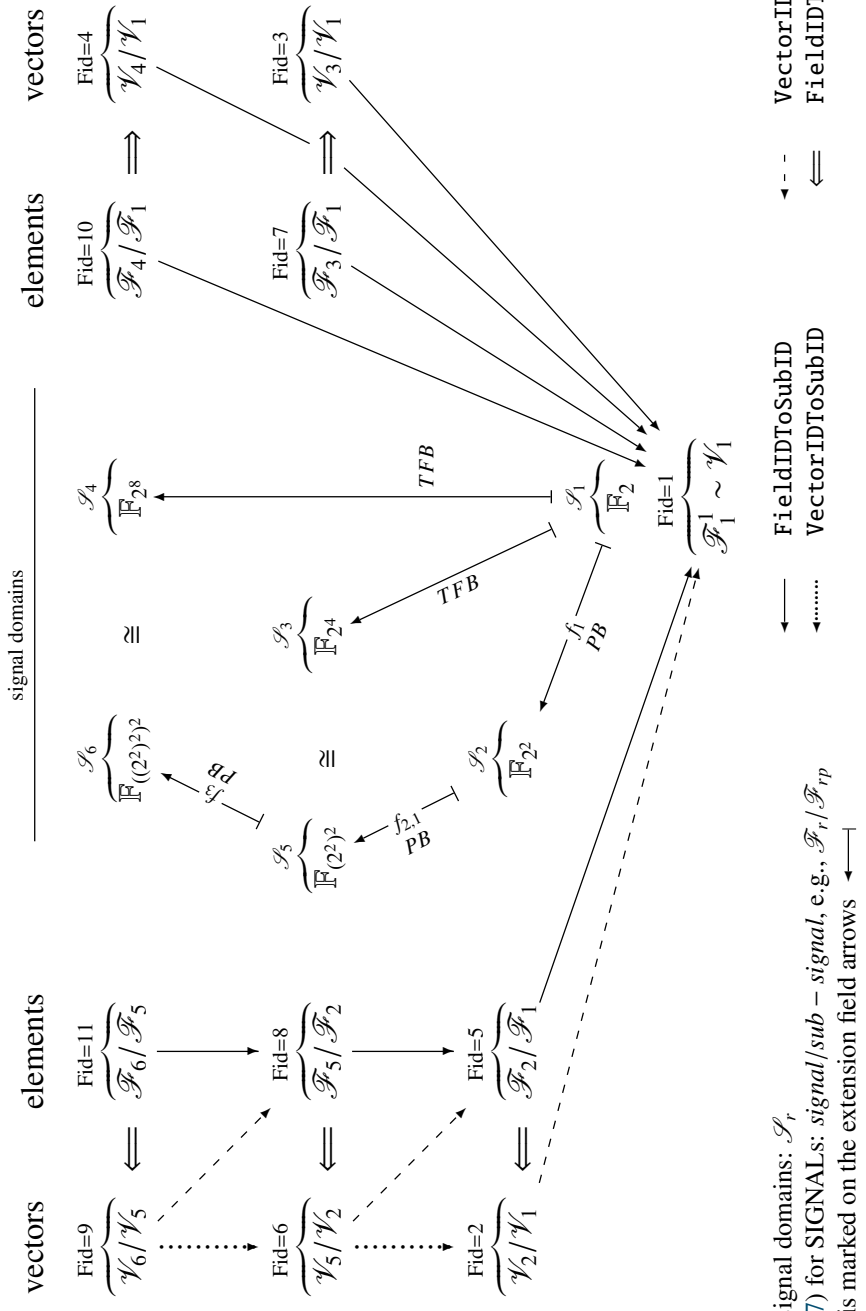


Figure 13.5: SignalPkg for the tower construction of $\mathbb{F}_{(2^2)^2}$ - left branch of diagram in Figure 13.3

13.4 Summary of key insights

In previous sections some of the key insights to datapath generation were already discussed, some solved and some of the remaining solutions partially announced. Table 13.11 shows a recap of the key insights, with ✓ indicating the ones with partial solutions and ✓✓ indicating the ones with final solutions. Because CIRCUIT uses the GAPtoVHDL functionality, the two relevant keys from Chapter 10 are repeated in Table 13.11.

The GAP objects `SignalDomain`, `SIGNAL`, and `SignalPkg` strongly fit the “math meets hardware” motif. The field structure was encoded in a way that is suitable for hardware implementations. It facilitates the synthesis of datapaths over arbitrary fields, including “just vectors” (Key 13.2 and 13.3). The `SignalPkg` is formatted to allow top-down modular synthesis (Key 13.5).

The next step

Recall Figure 12.1, showing that the threshold between *the architectural decisions* and *the automated design generation* is not strict. During the former, the decisions about the finite field and bases to be used are made. `SignalPkg`, used for encoding of this information, is shown on the top right within the magnifying glass in Figure 12.1. Next is the functional description of the algorithm, the `AlgFunctionality` object (Section 14.1). This object allows variable binding, classification of expression(s) given by the algorithm, and signal extraction (annotated around `AlgFunctionality` in Figure 12.1). `AlgFunctionality` is the starting point for automation.

Figure 12.1 also shows basic building blocks (box on the top left). Key 13.1 states that by knowing the subfield, basis, and direction (all contained within the `SIGNAL` referenced by the `Fid`), the expressions for the submodules can be generated on the fly, using the `FFCSA` package methods. Alternatively, the submodules could be fetched from a library of existing VHDL modules. Deciding how to obtain the submodules is the last architectural decision made by the user.

Key 10.3: Binding of GAP variables and VHDL signal names	✓	Section
A variable binding is a pair of strings in the form [GAP_variable, VHDL_signal]		10.2
Key 10.4: Classification of expressions defined over finite fields	✓	Section
The classification of expressions is a basis for the inference of the internal signals, the extraction and generation of the submodules, and for the automated generation of a datapath. The classification is based on the finite field over which the expression is defined.		10.3
Key 13.1: How to obtain the (sub)modules - binding expressions to finite fields		Section
The first step in addressing this problem is to <i>bind the expressions to the finite fields over which they are defined</i> . The submodules are generated for the subfield operations. Knowing the subfield, its basis and direction, the expressions for the submodules are generated on the fly, using the FFCSA methods.		13.1
Key 13.2: The SignalDomain: the bases and the field construction trail	✓✓	Section
Encoding of the finite fields and bases used in the design is required for VHDL datatypes and for generation of submodules. The GAP object SignalDomain stores information about the underlying finite field, basis used, basis direction, length, and the construction trail across the subfields. This amount of information is needed to capture tower field constructions and to distinguish isomorphisms.		13.2
Key 13.3: The SIGNAL: finite field as a vector space and “just vectors”	✓✓	Section
For the designer it is easy to switch between an input as element of the finite field \mathbb{F}_q^m and vector space \mathbb{F}_q^m , but for successful VHDL implementation this information has to be captured in GAP. Furthermore, the given algorithm can require “just vectors”, that are not elements of a given finite field, i.e., vector space, however, these vectors have coordinates from a given finite field, i.e., a given SignalDomain. The SignalDomain was embedded into another GAP object called SIGNAL, which holds the SignalDomain and the ‘vector (space)’ parameters length m , the direction, and an (optional) interpretation basis. They allow a distinction of (field) <i>elements</i> and <i>vectors</i> . All the VHDL data types used in the design are GAP SIGNAL objects, translated into VHDL.		13.2
Key 13.4: How to obtain the (sub)modules - correctness check based on the output port length		Section
The (sub)modules have to implement one expression per output coordinate. Hence the number of (generated) expressions must match the Length of the SIGNAL(s) used for modelling the output port(s) of the (sub)module.		13.2 Ex. 13.2.2
Key 13.5: The signal package SignalPkg - keeping it all in one place	✓✓	Section
The SignalPkg object holds all SignalDomains and SIGNALs used in the design. The SignalDomains are duplicate free, the SIGNALs are duplicate free and sorted. The sorting of SIGNALs follows <i>the top-down modular approach to hardware design: the submodules are expected for the subfields</i> . The index at which the signal is stored in the list is used as a <i>field identifier</i> Fid, which is used for binding of expressions to the finite fields.		13.3
Key 13.6: The VHDL type definitions and tower field constructions	✓✓	Section
In the presence of a tower field, the SIGNALs on the higher levels of the tower are defined as <code>array</code> . When no tower field is present, the SIGNALs are defined as <code>std_logic_vector</code> .		13.3

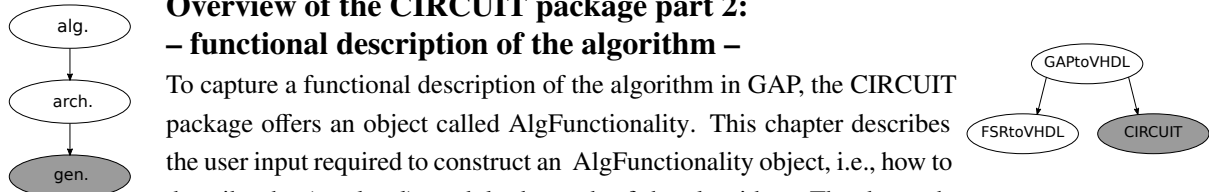
Notes: ✓ - partially solved ✓✓ - solved

Table 13.11: Summary of key insights to the CIRCUIT package - listed chronologically

Chapter 14

CIRCUIT package part 2: functional description of the algorithm

14.1 The functional description of the algorithm	185
14.2 The datapath based on classification of expressions	192
14.3 Summary of key insights	199



**Overview of the CIRCUIT package part 2:
– functional description of the algorithm –**

To capture a functional description of the algorithm in GAP, the CIRCUIT package offers an object called AlgFunctionality. This chapter describes the user input required to construct an AlgFunctionality object, i.e., how to describe the (top-level) module datapath of the algorithm. The datapath is always given with one or more ANF expressions. A SignalPkg is used to bind expressions to finite fields. The AlgFunctionality object allows for the entering of management information: entity and port names, and whether to use registered or combinational ports. The key automation tasks for this step include: variable binding, classification of expression(s) given by the algorithm, and signal and submodule extraction (but not submodule generation). The AlgFunctionality object provides the starting point for automation.

14.1 The functional description of the algorithm

The expression alone conveys only partial information about the module. Additional information is captured as (i.) *binding*, which will be explained in detail shortly, and (ii.) *management*, which specifies the following information:

- basic management: entity name, architecture name, input and output port names, i.e., strings
- technical management: registered or combinational input and output ports

In the VHDL background section (Section 3.3.4), a hardware module is presented as an entity/architecture pair, where entity describes the interface and architecture the internals. To write a VHDL module, sufficient information must be given, some of it very simple, e.g., the string for the entity

name, and some of it more complex, e.g., the VHDL data type to be used for an input port or the expression to be implemented by the architecture. The object AlgFunctionality was designed to store all the information needed to begin the automated design entry. It takes four arguments as an input to the constructor call AlgFunctionality:

- *entitylist* - a list of strings, with mandatory entity name
- *portlist* - specifies all the input and output ports and their data types
- *archlist* - provides the binding and the expressions to be implemented in VHDL
- *archtype* - implementation instructions, e.g., use registered outputs

Overview of attributes and components

Table 14.1 summarizes the attributes and components storing the information for an AlgFunctionality object. The attributes and the component *archtype* are set by the AlgFunctionality constructor call and will be explained in this section. The component *commentstr* is set to a default value "`--generated by GAP pkg: AlgFunctionality\n`" and can be easily modified. This comment is copied into the generated VHDL file. Example D.3.1 in Appendix D.3 shows the AlgFunctionality constructor call. The remaining components in Table 14.1 are initialized to -1 and will be explained in Section 14.2.

The AlgFunctionality object			
<i>attributes</i>		<i>components</i>	
FuncName	InputPorts	archtype	commentstr
EntityName	OutputPorts	classperoutput	class
ArchName	ArchBind	signalcollected	
EntityGenerics	ArchExpr	submodules	

Table 14.1: Main functionality of the CIRCUIT package - the AlgFunctionality object

Description of arguments and initialization of attributes and components

The constructor call returns a new AlgFunctionality object that represents the functional description of the top-level module for implementation of the algorithm. Below is a detailed explanation of the constructor arguments. The *argument/attribute* maps (14.1-14.3) are used to explain exactly how the attributes and components are set. The argument-attribute maps are formatted in two rows, and the relationship between the argument and attribute from Table 14.1 is given positionally: each argument gets stored as the attribute below it. This section also explains binding. A graphical representation of binding is shown in Figure 14.1. Although all binding is done at the time of the AlgFunctionality constructor call, the figure (and the description in text) are present binding as the gradual addition of new information.

Basic management

The *entitylist* is a list in the form given in argument-attribute map (14.1), whose values are used to set the *attributes* listed beneath each entry:

```
entitylist :      [ functionality,  entity_name,  arch_name,    generic_list  ]      (14.1)
attributes set :   FuncName      EntityName   ArchName     EntityGenerics
```

The first three list entries *functionality*, *entity_name*, and *arch_name* are just strings (without whitespace characters). The last entry, *generic_list*, is a list intended for future use; none of the GAP generated VHDL modules currently use generic parameters. The only mandatory string is *entity_name*, hence *entitylist* must contain at least one value, which is stored as the `EntityName` value of the `AlgFunctionality` object. In this case, all other attributes are set to default, with `FuncName` set to `blackbox`, `ArchName` to `main`, and `EntityGenerics` to an empty list `[]`.

Binding of VHDL ports to finite fields

The *portlist* contains three entries:

- *pkg* - a `SignalPkg` object
- *inlist* - input port list
- *outlist* - output port list

The input and output port lists must follow a given format, which binds the ports to the finite fields (or vectors). More precisely, it *binds the VHDL ports to SIGNAL objects and hence to the VHDL data types to be used*. Port binding is shown in Figure 14.1(b): the information conveyed by port binding is the interface (ports and their types).

Binding is again very simple, it is a pair of a string and a field identifier `Fid` w.r.t. the signal package *pkg*, namely `[VHDL*_port, Fid]`, where `*` is a placeholder for `input` or `output`. Both lists are subjected to a simple check that triggers an error in one of following two cases: (i.) one of the `VHDL*_port` values is not a string or includes whitespace characters, and (ii.) the field identifier `Fid` is not a positive integer or is bigger than the number of `SIGNALs` in the given signal package *pkg*. If the check is passed, the two lists are stored as attributes `InputPorts` and `OutputPorts`, as shown in argument-attribute map (14.2):

```
portlist :      [ pkg,      inlist,      outlist      ]      (14.2)
attributes set :          InputPorts   OutputPorts
```

■ **Implementation detail:** The `SignalPkg` is never stored as a part of `AlgFunctionality` object. The reason is the following: in many cases, the design will contain more than a single `AlgFunctionality`, and all of them must use the same signal package. The `SignalPkg` is stored only once in each design as will be discussed in more detail in Section 15.1. ■

Key 14.1: Binding of VHDL ports to the finite fields and vector(s) (spaces)

Port binding binds ports to SIGNAL objects and hence to the VHDL data types. A port binding is a pair of a string and a field identifier *Fid* in the form

[VHDL_input_port, *Fid*] and [VHDL_output_port, *Fid*]

More binding: binding between GAP variables and VHDL ports

The *archlist* contains two entries:

- *bindlist* - a list of variable binding
- *exprlist* - a list of expressions

The *bindlist* contains *variable binding* for GAP variables (for expressions) to the VHDL ports (for module). The *exprlist* contains a list of expressions to be implemented. Both lists are subjected to correctness checks before being stored as AlgFunctionality attributes ArchBind and ArchExpr, as shown in the argument-attribute map (14.3):

<i>archlist</i> :	[<i>bindlist</i> ,	<i>exprlist</i>]	(14.3)
attributes set :	ArchBind	ArchExpr	

The binding discussed here covers two cases, namely, the partial and the full GAP variable binding:

[GAP_variable*, VHDL_input_port]

where * indicates partial or full variable:

1. partial binding: binds part of a GAP variable to the VHDL port, whereby GAP variable indices translate to the VHDL port coordinates
2. full binding: binds the entire GAP variable to the VHDL port, i.e., the GAP variable indices are not used as coordinates

Partial binding is very similar to variable binding in Key 10.3 (Section 10.2), but there is no need to specify a separate binding rule for each GAP variable. The short example in Table 14.2, the binding ["a", "i_a"] translates $a_0 \cdot a_1$ to $i_a(0) \cdot i_a(1)$. This binding is used for squaring for a polynomial basis in Example D.5.2 in Appendix D.2. A single partial binding for numerous GAP variables is enough, because the GAP variables used in this way have a meaning: they are the *avec* and *bvec* variables [*a_0*, *a_1*, ...,] and [*b_0*, *b_1*, ...], generated automatically by the CIRCUIT package for a specific finite field, which prevents errors and ensures that the VHDL port used in the binding rule corresponds to a vector SIGNAL of length *m*.

partial binding		full binding	
["a", "i_a"]		["a_0", "i_a"], ["a_1", "i_b"]	
GAP	VHDL	GAP	VHDL
$a_0 \cdot a_1$	$i_a(0) \cdot i_a(1)$	$a_0 \cdot a_1$	$i_a \cdot i_b$

Table 14.2: Short examples of partial and full binding

Binding allows a designer to use the existing GAP variables x_i , a_i , b_i and d_i (Section 10.2) to specify arbitrary expressions in GAP. With full binding, the entire GAP variable and its subscript is replaced by the VHDL port name. However, a binding must be given for each GAP variable. In short example in Table 14.2, the binding ["a_0", "i_a"], ["a_1", "i_b"] translates $a_0 \cdot a_1$ to $i_a \cdot i_b$. This binding is also shown in Figure 14.1(c). Further examples of full binding are shown in Example D.3.1 in Appendix D.3, demonstrating the AlgFunctionality for the running example expression (13.1) defined over $\mathbb{F}_{((2^2)^2)}$ (see Example D.3.1), and in the multiplication module in Example D.5.3 in Appendix D.3 (see Example D.5.3(a)).

An important difference w.r.t. variable binding from Key 10.3 in Section 10.2 is that the binding for AlgFunctionality is done for the VHDL ports, not internal signals. Therefore, the *bindlist* must pass the CheckPortStrings test, which ensures that the second string in an individual binding is a valid input port name. This is a simple check, only comparing two strings: the VHDL_input_port string in a port binding (*inlist* of the *portlist* argument to the AlgFunctionality constructor), and the VHDL_input_port string in the variable binding (*bindlist* of the *archlist* argument), see Table 14.3.

argument	<i>portlist</i>	<i>archlist</i>
sub-list	<i>inlist</i>	<i>bindlist</i>
meaning	input port binding	variable binding
	[VHDL_input_port, Fid]	[GAP_variable*, VHDL_input_port]
	↑	↑
	CheckPortStrings test comparison	

Table 14.3: The CheckPortStrings test comparison

Key 14.2: Binding of GAP variables and VHDL input ports

Variable binding binds the GAP variables used in the expressions to the VHDL input ports of the hardware module implementing the datapath. A variable binding is a pair of strings in the form

[GAP_variable*, VHDL_input_port]

* indicates partial or full variable.

More binding: binding of expressions

The *exprlist* format must match the list of the output ports *outlist* from the *portlist* argument: there must be an expression list *exprlist_i* for each *o_port_i* (abbreviated for VHDL_ouput_port_i). The expression lists are *positionally bound* to the output ports by their indices within the parent lists *exprlist* and *outlist*: *exprlist_i* contains instructions on how to drive the (coordinate of) output port *o_port_i*. Figure 14.1(d) shows full positional binding.

$$\begin{array}{l} \textit{exprlist} : \quad [\textit{exprlist}_1, \quad \quad \quad \textit{exprlist}_2, \quad \quad \dots \quad \quad \quad \textit{exprlist}_n \quad] \\ \textit{outlist} : \quad [[\textit{o_port}_1, \textit{Fid}_1], \quad [\textit{o_port}_2, \textit{Fid}_2], \quad \dots \quad [\textit{o_port}_n, \textit{Fid}_n] \quad] \end{array}$$

Since each output port is bound to a SIGNAL via its field identifier *Fid* (Key 14.1), the positional binding of the expressions to the output ports also *implicitly binds the expressions to the finite fields over which they are defined*. This provides the final step to Key 10.4, i.e., the classification of expressions defined over finite fields, and the first step to Key 13.1, i.e., how to obtain the submodules. For example, if the expression includes a multiplication and it is bound to *Fid=1*, which corresponds to a SIGNAL representation of elements in \mathbb{F}_2 , then the hardware component is a simple AND gate and no multiplier submodule is needed. Figure 14.1(e) shows implicit binding.

Key 14.3: Positional binding of expressions and output ports

Positional binding of the expression lists to the output ports provides datapath instructions on how to drive the output ports:

$$\begin{array}{ccc} \textit{exprlist}_i & \Rightarrow & \textit{exprlist}_i \\ [\textit{o_port}_i, \textit{Fid}_i] & & \textit{o_port}_i \end{array}$$

Key 14.4: Implicit binding of expressions to the finite fields and vector(s) (spaces)

Positional binding of the expression lists to the output ports also *implicitly binds the expressions to the finite fields* via the field identifiers *Fid* associated with the output ports:

$$\begin{array}{ccc} \textit{exprlist}_i & \Rightarrow & \textit{exprlist}_i \\ [\textit{o_port}_i, \textit{Fid}_i] & & \textit{Fid}_i \end{array}$$

■ **Implementation detail:** The implicit binding of the expression lists to the field identifiers allows a correctness check, as described in Key 13.4 in Section 13.2: the (sub)modules have to implement one expression per output coordinate, hence the number of (generated) expressions within each *exprlist_i* must match the Length of the *Fid_i* SIGNAL used for modelling the corresponding output port of the (sub)module *o_port_i*. If the corresponding *Fid_i* SIGNAL has length 1, i.e., is an *element* output, the *exprlist_i* contains a single expression. If the *Fid_i* SIGNAL has length $m > 1$, i.e., is a *vector* output, the *exprlist_i* is a list with m expressions (recall Figure 13.4(b)). This ensures that the hardware will be able to drive (each coordinate of) each output port. ■

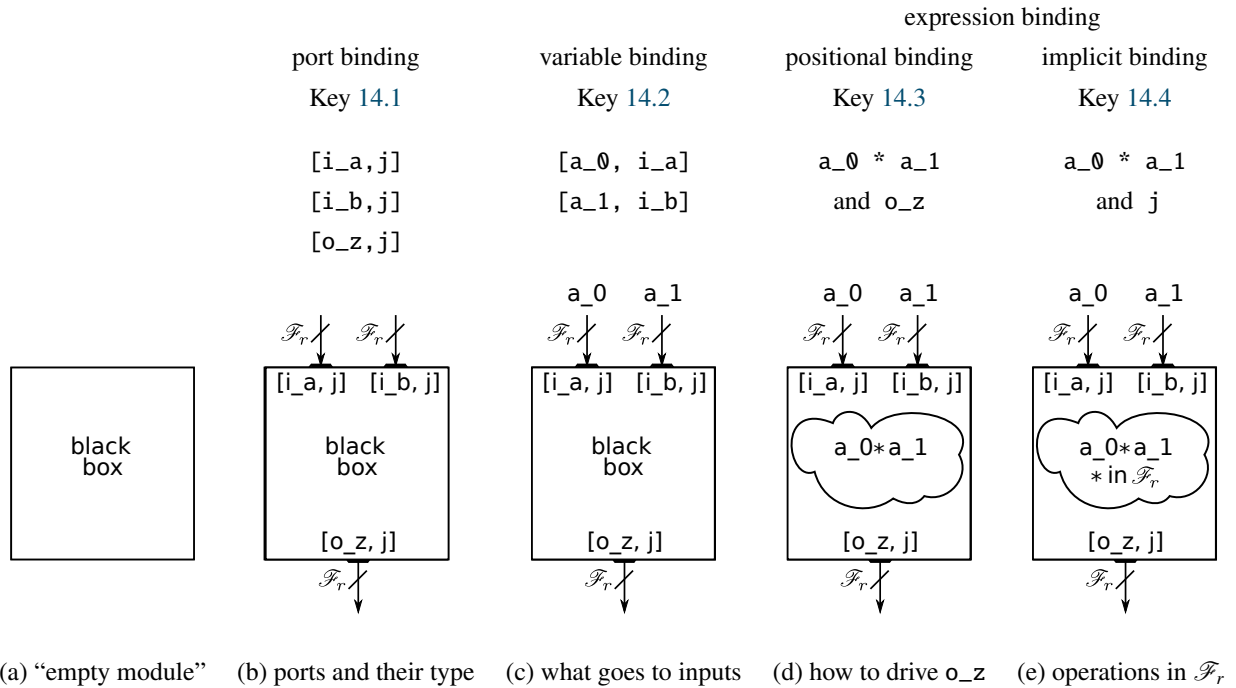


Figure 14.1: Binding for AlgFunctionality object and their effects on the hardware module

Figure 14.1 shows the effects of binding and the information that bindings convey to the automation process. The example expression is $a_0 \cdot a_1$, defined over a finite field \mathbb{F}_q . In the given SignalPkg, \mathbb{F}_q is represented by \mathcal{F}_r . Figure 14.1(b) shows the port binding, which defines the interface (port labels and \mathcal{F}_r , which become a VHDL type). Figure 14.1(c) shows the full variable binding, which binding the variables to the ports. As the algorithm specifies what a variable means, the variable binding in turn specifies “what goes on inputs” in the sense of interaction with environment. Figures 14.1(d,e) show the expression binding, with two effects. First, the positional binding of expression to the output o_z gives instructions on how to drive that output. Second, the implicit binding to the Fid=j binds the expression to \mathcal{F}_r , which in turn implies that all the operations are over \mathcal{F}_r as well. Binding the AlgFunctionality to the SignalPkg brings the awareness of the underlying field structure to the automation process, i.e., is a part of the “math meets hardware” motif.

Key 14.5: Functional description aware of finite fields and vector(s) (spaces)

Binding the AlgFunctionality to the SignalPkg brings awareness of the underlying field structure to the automation process. This is achieved by the port, variable, and expression binding.

More management: additional datapath instructions

The argument *archtype* provides further instructions:

- *porttype* - combinational or registered
- *datapathtype* - simple or complex

In Subsection 3.3.3 the circuits were classified based on the type of their input and output ports¹. The argument *porttype* provides further instructions, for combinational C or registered R ports, and with I denoting input 0 output port, the following instructions are possible: CICO, RICO, CIRO and RIRO (Figure 3.11 in Subsection 3.3.3). For writing the VHDL code for the registers, the `Write_registers` function from Table 10.3 will be used, and a list of control signals and corresponding control values must be provided, $[cntllist, cntlvals]$. Whenever a registered port is used, the entity will be generated with additional control ports for all entries from (both) *cntllist*; there is no need to include them in the *portlist*. The *archtype* list is stored as a component *archtype*.

■ **Implementation detail:** When RIRO is used, two sets of control signals and control values must be provided, e.g.,: $[["i_clk", "i_rst", "i_ce"], [1,1,1]], [["i_clk", "i_rst", "i_ceout"], [1,1,1]]$, the instructions for rising-edge registers with active- high reset and both chip-enable signals². ■

Only the the simple *datapath* is used so far, the complex datapath is reserved for future use. Among other, it will allow (some) subexpression elimination, e.g., (a_0+a_3) in the matrix U in Example 7.4.1 in Section 7.3.1, and synthesis of unrolled datapaths. The complex datapaths will make us of the GAP syntax trees, are announced for the next GAP version.

14.2 The datapath based on classification of expressions

As stated in Key 10.4 in Section 10.3, the classification of expressions is a basis for the inference of internal signals, extraction, and generation of submodules, as well as for the automated generation of a datapath. Recall equation (10.1), with non-zero coefficients $\gamma_{i_0, i_1, \dots, i_{t-1}} \in \mathbb{F}_q$, where $i_j \in \mathbb{Z}_q$ for $0 \leq j < t$:

$$f(x_0, x_1, \dots, x_{t-1}) = \sum_{\substack{\forall (i_0, i_1, \dots, i_{t-1}) \in \mathbb{Z}_q^t \\ \gamma_{i_0, i_1, \dots, i_{t-1}} \neq 0}} \gamma_{i_0, i_1, \dots, i_{t-1}} x_0^{i_0} x_1^{i_1} \cdots x_{t-1}^{i_{t-1}} \quad (14.4)$$

The 8 classes of expressions in Table 10.4 are defined in an incremental fashion: the higher the class, the more complex the expression, i.e., the more demanding its corresponding datapath w.r.t. to finite field arithmetic. The last column in Table 10.4 contains the following finite field arithmetic operations:

(\oplus) *addition*: bit-wise XOR of the terms, no submodule required

¹ all input ports are always of the same type and all output ports are always of the same type, “mixing” is prohibited

² could also be the same chip-enable for inputs and outputs

- ($\times\gamma_1$) *multiplication with a constant*: term coefficient is different from 1, requires a submodule (a matrix-vector multiplier for multiplication with a constant)
- (M) *multiplication*: a monomial contains more than one variable, requires an AND gate when defined over \mathbb{F}_2 , and a multiplier submodule when defined over an extension field.
- (E) *exponentiation*: a monomial contains a variable with an exponent and a submodule for exponentiation is required

Inspecting a class 7 expression, i.e., an expression containing all operations listed above, reveals a natural order for performing the operations. The innermost operations, i.e., the exponentiation for class 7 expressions, are performed first, followed by multiplications to obtain the monomials and terms. Additions come last. Accordingly, the datapath can be written in steps following this order. An expression is a set of instructions on how to drive (a coordinate of) an output port, and is transformed into a datapath with the following steps:

1. *variable binding*: The starting point are variables x_j , $0 \leq j < t$, which occur in the expression. Before discussing the datapath, recall the binding of GAP variable to VHDL input ports (Key 14.2). This step will be discussed in more detail in Chapter 16.
2. *exponentiation*: for each variable x_j with $i_j > 1$, that is for each $x_j^{i_j}$, perform exponentiation. The exponentiations can be done in parallel, i.e., the component instantiations for the submodules are concurrent. Assuming the variable x_j itself is (a coordinate of) an input port and is (after binding) used as the input to the submodule, a signal for the output of the submodule is needed.
3. *multiplication*: perform the multiplications to obtain all monomials, i.e., all the $x_0^{i_0} x_1^{i_1} \dots x_{t-1}^{i_{t-1}}$. To improve performance, the multiplications are always performed in a tree structure, with two factors per multiplication. A signal is needed for each (intermediate) product. Note that there can be many monomials, hence many multiplication trees, performed in parallel.
4. *multiplication with constants*: for each extension field constant $\gamma_{i_0, i_1, \dots, i_{t-1}} \in \mathbb{F}_{2^m}/\mathbb{F}_2$, with the exception of the constant term, perform the multiplications to obtain the terms, i.e., all the $\gamma_{i_0, i_1, \dots, i_{t-1}} x_0^{i_0} x_1^{i_1} \dots x_{t-1}^{i_{t-1}}$. The inputs to the ($\times\gamma$) submodules are the monomials, and new signals are needed for their outputs.
5. *additive constant*: if the expression has a non-zero constant term, its value will be defined as a VHDL constant. If there are at least two terms in the expression, the constant is treated as one of the terms. Otherwise, the constant is directly driving (a coordinate of) an output port.
6. *addition of the terms*: this step computes the final sum in the expression. It is performed in a tree structure, with two-input bit-wise XOR gates at each node. A signal is needed for each (intermediate) sum. Note that there is always only one summation per expression, and thus, per (coordinate of an) output port.
7. *drive output*: connect the sum to the appropriate (coordinate of an) output port.

The 7 steps are captured by the schematic in Figure 14.2. The horizontal dashed line at the top of Figure 14.2 represents the variables x_j , which are not a part of the hardware module. Above the dashed line, the classification of a given expression is listed, with a class 7 expression on the right. The steps are listed on the left and the signals, i.e., “results” of the performed steps, on the right. Solid vertical arrows indicate that at least one of the step inputs will undergo the action of the step, e.g., at least one of the variables in a class 7 expression will have an exponent $i_j > 1$. The inputs that do not undergo the action are simply passed on to the next step. When no solid line is present, the passing on is indicated with a dashed vertical arrow. For example, all but class 0 expressions require a binding of GAP variables x_j and the input ports of the hardware module. Similarly, the only difference between a class 6 and a class 7 expression are the multiplicative constants: the class 6 expressions have a dashed arrow for step 4. To distinguish the vertical paths for classes 3 and 4, which look exactly the same in terms of their arrows, the solid arrows in step 3 are annotated with A for AND gates and with M for the multiplier submodules. Similarly, there may be no multiplications for class 6 or 7, which is annotated with M?.

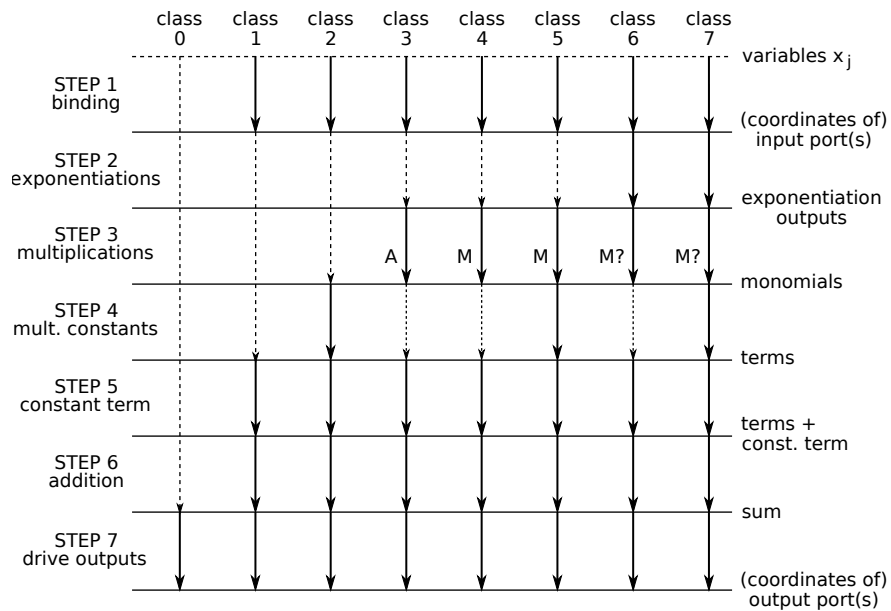


Figure 14.2: The steps of writing a datapath

Key 14.6: The datapath and classification of expressions

The 8 classes in the classification Table 10.4 are defined in an incremental fashion: the higher the class, the more complex the expression and its corresponding datapath w.r.t. the finite field arithmetic. Inspecting the highest class expression reveals a natural order for performing the operations. An expression is a set of instructions on how to drive (a coordinate of) an output port, and it is transformed into a datapath by following this order: 1. binding of GAP variables to VHDL input ports, 2. (E) exponentiations, 3. (M) multiplications (to obtain monomials), 4. ($\times\gamma$) multiplication with constants (to obtain the terms), 5. ($+\gamma$) including the additive constant to the list of terms, 6. (\oplus) addition of the terms, and 7. driving the output.

Towards datapath generation

Three methods are used to prepare the AlgFunctionality object for generation of the datapath, listed in Table 14.4. First column lists the method name and the second column the component of the AlgFunctionality object, into which the method stores its result. The third column contains a short description of the method and the fourth column additional notes.

The AlgFunctionality object		
method	AF component set	description
AlgFunctionalityClass † ^[1,2]	classperoutput, class	classify all expressions
AlgFunctionalityCollectSignals † ^[2,3]	signalcollected	all signals for all expressions
AlgFunctionalityCollectAllSubmodules † ^[4,5]	submodules	actual submodules, duplicate free
MakeAlgFunctionalityDesignReady † - run all three methods above in order in which they appear		

Notes:

- | | |
|--|--|
| [1] based on Table 10.4 in Section 10.3 | [3] for (E) outputs, (M) tree(s) nodes and edges, ($\times\gamma$) outputs, ($+\gamma$) constants, (\oplus) tree nodes and edges |
| [2] for each coordinate of each output port | [5] true submodules: (E), ($\times\gamma$), (M), but not AND |
| [4] same operation repeated more than once means one submodule (duplicate free), but instantiated multiple times | † arguments (AF, pkg), where AF is the AlgFunctionality object and pkg the corresponding SignalPkg |

Table 14.4: Main functionality of the CIRCUIT package - methods for the AlgFunctionality

AlgFunctionalityClass is using the Which_class method from the GAPtoVHDL package and returns classification based on Table 10.4 in Section 10.3: Which_class call is repeated for each coordinate of each output, i.e., for each individual ANF expression stored in ArchExpr attribute. Key 14.3 explained the implicit binding of expressions to the Fids. The expression is defined (and classified) over the UnderlyingField of the UnderlyingSignalDomain of the SIGNAL given by this Fid. A check is set in place to backtrack to all inputs and ensure there is no change in this information. For the element output ports, Which_class is run only once per port, and the result is stored in the component classperoutput. For the vector output ports of length m , Which_class is run for each of the m output coordinates, and then the highest class is stored

as the corresponding `classperoutput` component of the `AlgFunctionality` object. Afterwards, the highest class among the `classperoutput` values is stored in the `class` component. Classification plays a role for the extraction of the internal signals and submodules. The internal signals are extracted, i.e., collected by the method `AlgFunctionalityCollectSignals`, again for each expression, i.e., for each coordinate of each output port in the given `AlgFunctionality` object, and stored into the `signalcollected` component of the `AlgFunctionality` object. Example D.3.1 in Appendix D.3 shows the collected signals, printed in a human friendly fashion, using the method `PrintAlgFunctionalityCollectSignals`.

operation and encoding	description
(E) exponentiation [EXPname, e, t, j]	EXPname=EXPout_portNum_coordNum_n† is the actual string, n is a counter, e = i _j is the exponent of the current x _j ^{i_j} , t is the term number within the expression, j is the variable number within the term
(M) multiplication ‡ prefix_lvl_#_node# prefix_lvl_#_edge#	performed in tree structure, with two factors per multiplication, only edge signals are used for VHDL, the prefix has the format out_portNum_coordNum_mon t†, t is the term number within the expression
(×γ) mult. const. [MVname, c, t]	MVname=MVout_portNum_coordNum_n† is the actual string, n is a counter, c = γ _{i₀,i₁,...,i_{t-1}} is the current constant, t is the term number within the expression
(+γ) additive const. [Cname, c, t]	Cname encoding depends on the Fid, special care is needed in case of tower field constants c = γ _{i₀,i₁,...,i_{t-1}} is the current constant ct, and t is the term number within the expression (always the last term)
(⊕) addition ‡ prefix_lvl_#_node# prefix_lvl_#_edge#	performed in a tree structure, with two-input bit-wise XOR gates at each node, only edge signals are used for VHDL, the prefix has the format out_portNum_coordNum_term†

† portNum index of the output port of the module, coordNum index of coordinate (vectors only)

‡ stores a list for all trees

Table 14.5: The encoding for the inferred signals

Each (coordinate of) output is associated with an ANF expression, as given by equation (14.4). The encoding in Table 14.5 is used for the collection of signals. For each expression, signals for the steps 2–6 in in datapath Figure 14.2 are collected as follows:

2. *exponentiation* (E): for each exponentiation submodule, a signal for the output of the submodule is needed. The encoding in row 1 in Table 14.5 is a list that contains all the information needed to extract and generate the exponentiation submodule, and then place it in the datapath.
3. *multiplication* (M): as was mentioned before, to improve latency, the multiplications are always performed in a tree structure, with two factors per multiplication. A signal is needed for each (intermediate) product. Note that there can be many monomials, hence many multiplication trees, performed in parallel. For each multiplication tree, the signal names are created for the nodes and for the edges: `prefix_lvl_#_node#` and `prefix_lvl_#_edge#`. Only the edge signals are declared in VHDL; the nodes are replaced by AND gates or multipliers.

4. *multiplicative constants* ($\times\gamma$): for each extension field constant, with exception of the constant term, a signal is needed for the output of the ($\times\gamma$) submodule. The encoding in row 3 in Table 14.5 is a list that contains all the information needed to extract and generate the matrix-vector multiplier (MV) submodule, and then place it in the datapath.
5. *additive constant* ($+\gamma$): if the expression has a non-zero constant term, its value will be declared as a VHDL constant.
6. *addition of the terms* (\oplus): the final sum in the expression is performed in a tree structure, with two-input bit-wise XOR gates at each node. A signal is needed for each (intermediate) sum. Note there is always only one summation per expression, and hence per (coordinate of an) output port. The signal names are created for the nodes and the for the edges: `prefix_lvl_#_node#` and `prefix_lvl_#_edge#`. Only the edge signals are declared in VHDL, the nodes are replaced by XOR gates or multipliers.

Key 14.7: The inference of the internal VHDL signals

The connection between the classification of expressions and the datapath (Key 14.6) plays an important role for the inference of the internal VHDL signals and for the extraction of submodules. The signals are collected for all exponentiations (E), multiplications (M), more precisely for the multiplication trees to obtain the monomials, for multiplications with the extension field constants ($\times\gamma$), for the additive constants ($+\gamma$), and for the final addition (\oplus), i.e., for the addition tree.

The collected signals are used as an input to the `AlgFunctionalityCollectAllSubmodules` method. There are two key differences between collected signals and collected submodules:

- collected signals reflect the VHDL component instantiations: the number of actual submodules is less or equal to the number of component instantiations
- AND, XOR and ($+\gamma$) do not require submodules (but still need signal collection)

The method `AlgFunctionalityCollectAllSubmodules` will traverse the `signalcollected` and identify each hardware component instantiation: exponentiations (E), multiplications with constants ($\times\gamma$), and multiplications (M) in extension fields (in \mathbb{F}_2 , multiplication is the AND gate and does not require a submodule instantiation). Based on the encoding of the signals and the `Fid` of the corresponding output port, it removes all duplicates. For example, two exponentiation signals with the same exponent and for the same `Fid` require the same submodule. The submodule, i.e., the component VHDL is generated just once, but can be instantiated multiple times: this will create multiple physical copies of the same building blocks. The submodules³ found for a particular `Fid` are collected together and the `Fids` are stored as well. The encoding for the submodules is listed in Table 14.6.

³ there is always only one multiplier submodule for a given `Fid`

	submodule	encoding	description
(E)	<i>exponentiation submodule</i>	[“EXP”, e]	e is the value of the exponent
(M)	<i>multiplier</i>	“MforFid#”	# is replace by the actual Fid
($\times\gamma$)	<i>matrix-vector multiplier for multiplication with a constant</i>	[“MV”, c]	$c = \gamma$

Table 14.6: The encoding for the submodules

Methods `AlgFunctionalityClass`, `AlgFunctionalityCollectSignals`, and `AlgFunctionalityCollectAllSubmodules` rely on the `SIGNAL` and `SignalDomain` structure, as well as the assumption that the entire expression is defined over the same `SignalDomain`. A special check is set in place to backtrack from the output port (coordinate) to all the input ports (and their coordinates) that are bound to the GAP variables used in the expression (using the binding between GAP variables and VHDL input ports, see Key 14.2). This check can detect basis transitions. When viewing the finite field as a vector space, the basis transition is given by a list of expressions, one expression for each output coordinate, and will pass this check: the two vectors⁴ will have different interpretation bases, maybe even different directions, but will be defined over the same signal domain.

Key 14.8: The extraction of submodules

The collected signals (Key 14.7) are the basis for extracting the submodules, however the collected signals reflect *the VHDL component instantiations* and the number of actual submodules is less than or equal to the number of component instantiations (possible duplicates). The \mathbb{F}_2 multiplications (AND gates) and the additions (XOR gates for terms and for $(+\gamma)$) do not require submodules. Submodules are extracted for the exponentiations (E), multiplications with constants ($\times\gamma$), and multiplications (M) in extension fields. The duplicates are removed based on the encoding of the signals and the *Fid* of the corresponding output port.

Example D.3.1 in Appendix D.2 shows an `AlgFunctionality` for the implementation of running example (13.1) defined over the tower field $\mathbb{F}_{(2^2)^2}$ (Example 13.1.3 in Section 13.1). It demonstrates the collector method `AlgFunctionalityCollectSignals` and method `MakeAlgFunctionalityDesignReady`.

⁴ input port and output port

14.3 Summary of key insights

This chapter presented the AlgFunctionality object, which serves as the manual entry point for the top-level module. The datapath is given with one or more ANF expressions and SignalPkg is used to bind expressions to finite fields (Keys 14.1 - 14.5). The information conveyed by binding defines the ports and their types (Key 14.1), what goes to inputs (Key 14.1), how to drive the output (Key 14.2), and what kind of submodules are needed (Key 14.3), as the operations are bound to the Fid \mathcal{F}_r together with expressions. Binding the AlgFunctionality to the SignalPkg brings the awareness of the underlying field structure to the automation process (Key 14.5).

Binding expressions to Fids allows for their classification and for signal inference (Key 14.7). Finally, the submodules are extracted (Key 14.8).

Tables 14.7 and 14.8 summarize all key insights from Chapter 13 and 14. The tables contain checkmarks indicating the status of the Key: ✓ for previously partially solved, ✓✓ for previously solved by GAPtoVHDL and SignalPkg, and ✓ for newly partially solved, ✓✓ for newly solved by AlgFunctionality. The Keys are no longer ordered chronologically but are grouped together by concepts they describe.

The Keys from the previous recap (Table 13.11 in Section 13.4) are listed only by their title in the left half of Table 14.7. The previously fully solved Keys 13.2, 13.3, 13.5, and 13.6 are grouped together and listed first. The remaining Keys in the left half obtained their new (partial) solutions from the AlgFunctionality object and its methods (and from new Keys, which are listed under the new checkmarks ✓). Key 13.1 is only partially solved: it is missing the FFCSA package instruction for the submodule generation. The new AlgFunctionality Keys 14.1-14.5, listed in the right half from Table 14.7, are the foundation for the Key 14.6.

The prerequisite for the datapath generation (Key 14.6) are the inferred signals (Key 14.7) and the submodule extraction (Key 14.8). Both are achieved by MakeAlgFunctionalityDesignReady method (Table 14.4).

The next step

The remaining two Keys 13.1 and 14.6 will be fully solved by the GAP object AlgDesign and its methods (Section 15.1), which will make the transition from *design* (on the dashed line in Figure 14.3) to *datapath* (on the dotted line in Figure 14.3). Figure 14.3 shows the last part of *the architectural decisions* and *the automated design generation*, with the Keys listed in Tables 14.7 and 14.8 annotated on the right. Afterwards, all that is left is writing the VHDL.

Key 14.1	Binding of VHDL ports to the finite fields and vector(s) (spaces)	✓✓	Section 14.1
<i>Port binding</i> binds ports to SIGNAL objects and hence to the VHDL data types. A port binding is a pair of a string and a field identifier <code>Fid</code> [<code>VHDL_input_port, Fid</code>] and [<code>VHDL_output_port, Fid</code>]			
Key 14.2	Binding of GAP var. and VHDL in. ports	✓✓	Section 14.1
<i>Variable binding</i> binds the GAP variables used in the expressions to the VHDL input ports of the hardware module implementing the datapath. A variable binding is a pair of strings [<code>GAP_variable*, VHDL_input_port</code>]			
Key 14.3	Positional binding of expressions and output ports	✓✓	Section 14.1
<i>Positional binding</i> of the expression lists to the output ports provides datapath instructions on how to drive the output ports: $\text{exprlist}_i \Rightarrow \text{o_port}_i$			
Key 14.4	Implicit binding of expressions to the finite fields and vector(s) (speaces)	✓✓	Section 14.1
Positional binding of the expression lists to the output ports also <i>implicitly binds the expressions to the finite fields</i> via the field identifiers <code>Fid</code> associated with the output ports: $\text{exprlist}_i \Rightarrow \text{Fid}_i$			
Key 14.5	Functional description aware of finite fields and vector(s) (speaces)	✓✓	Section 14.1
Binding the AlgFunctionality to the SignalPkg brings awareness of the underlying field structure to the * automation process. This is achieved by the port, variable, and expression binding.			

- newly partially solved (✓) and newly solved (✓✓) by AlgFunctionality

Key 13.2	The SignalDomain: the bases and the field construction trail	✓✓	Section 13.2
Key 13.3	The SIGNAL: finite field as a vector space and “just vectors”	✓✓	Section 13.2
Key 13.5	The signal package SignalPkg - keeping it all in one place	✓✓	Section 13.3
Key 13.6	The VHDL type definitions and tower field constructions	✓✓	Section 13.3
Key 10.3	Binding of GAP variables and VHDL signal names	✓✓ Key 14.1	Section 10.2
Key 10.4	Classification of expressions defined over finite fields	✓✓ Key 14.3, 14.1, 14.7, 14.8	Section 10.3
Key 13.1	How to obtain the (sub)modules - binding expressions to the finite fields	✓ Key 14.1, 14.3	Section 13.1
Key 13.4	How to obtain the (sub)modules - correctness check based on the output port length	✓✓	Section 13.2

- previously partially solved (✓) and solved (✓✓) by GAProVHDL and SignalPkg

Table 14.7: Summary of key insights to the CIRCUIT package with new (partial) solutions from the AlgFunctionality - grouped conceptually

Key	The datapath and classification of expressions	✓	Section
14.6	The 8 classes in the classification Table 10.4 are defined in an incremental fashion: the higher the class, the more complex the expression and its corresponding datapath w.r.t. the finite field arithmetic. Inspecting the highest class expression reveals a natural order for performing the operations. An expression is a set of instructions on how to drive (a coordinate of) an output port, and it is transformed into a datapath by following this order: 1. binding of GAP variables to VHDL input ports, 2. (E) exponentiations, 3. (M) multiplications (to obtain monomials), 4. ($\times\gamma$) multiplication with constants (to obtain the terms), 5. ($+\gamma$) including the additive constant to the list of terms, 6. (\oplus) addition of the terms, and 7. driving the output.		14.2

Key	The inference of the internal VHDL signals	✓✓	Section
14.7	The connection between the classification of the expressions and the datapath (Key 14.6) plays an important role for the inference of the internal VHDL signals and for the extraction of submodules. The signals are collected for all exponentiations (E), multiplications (M), more precisely for the multiplication trees to obtain the monomials, for multiplications with the extension field constants ($\times\gamma$), for the additive constants ($+\gamma$) and for the final addition (\oplus), i.e., for the addition tree.		14.2

Key	The extraction of submodules	✓✓	Section
14.8	The collected signals (Key 14.7) are the basis for extracting the submodules, however the collected signals reflect the <i>VHDL component instantiations</i> and the number of actual submodules is less or equal to the number of component instantiations (possible duplicates). The \mathbb{F}_2 multiplications (AND gates) and the additions (XOR gates for terms and for ($+\gamma$)) do not require submodules. Submodules are extracted for the exponentiations (E), multiplications with constants ($\times\gamma$), and multiplications (M) in extension fields. The duplicates are removed based on the encoding of the signals and the Fid of the corresponding output port.		14.2

- newly partially solved (✓) and newly solved (✓✓) by AlgFunctionality

Table 14.8: Summary of key insights to the CIRCUIT package with new (partial) solutions from the AlgFunctionality - continued

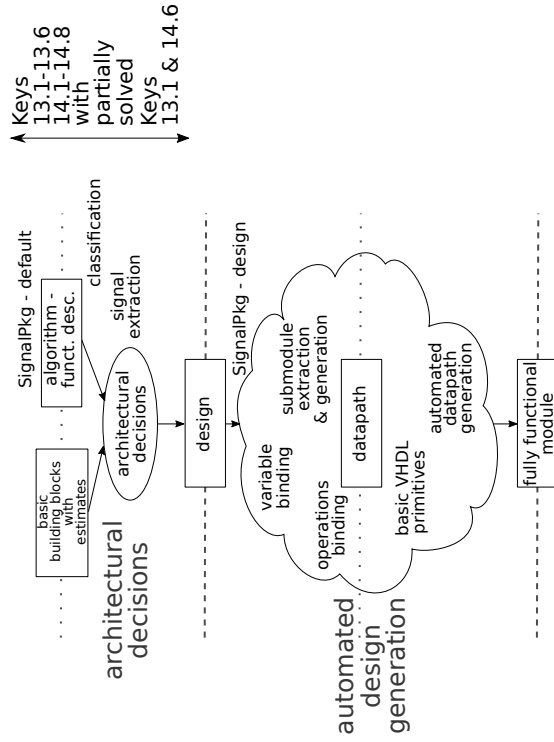
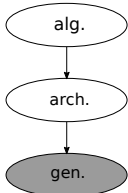


Figure 14.3: Last part of the architectural decisions and the automated design generation with key insights

Chapter 15

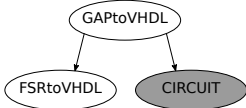
CIRCUIT package part 3: VHDL-ready design

15.1 The initial design	203
15.2 Top-down processing	207
15.3 Bottom-up processing	212
15.4 Connectors for the tower field elements and vectors	216
15.5 Putting it all together	220
15.6 Summary of key insights	222



**Overview of the CIRCUIT package part 3:
– VHDL-ready design –**

This chapter describes the AlgDesign object. It is formed from a design-ready AlgFunctionality, SignalPkg, and submodule instructions. The submodule instructions are the last user input required; the architectural decisions are now complete. AlgFunctionality is the top-level module of AlgDesign, and in a way, AlgDesign works as a wrapper. The created AlgDesign with stored AlgFunctionality and SignalPkg is called *initial design*. The CIRCUIT package implements two compilation algorithms to process the design in a top-down/bottom-up loop. The first compilation algorithm generates all submodules, required for implementation, in a top-down manner. The submodules are generated as AlgFunctionality objects of their own and added to the initial AlgDesign. The same SignalPkg is used for all submodules. FFCSA package methods will be used for the automated submodule generation; the submodule instructions have to be passed to the AlgDesign. The second compilation algorithm transforms all AlgFunctionality objects into VHDL-ready Circuit objects in a bottom-up manner. It adds the information needed for the component instantiations, declaration of constants, and possible connectors between elements/vectors of isomorphic fields in the design.



15.1 The initial design

Key 15.1: The design

A hardware design^a itself consists of a top-level module and all its submodules. The top-level is modelled as an AlgFunctionality object that together with corresponding SignalPkg and submodule instructions forms the initial AlgDesign. All the required submodules will be generated as AlgFunctionality objects of their own, using the same SignalPkg, and added to the initial AlgDesign in a top-down fashion.

^anot the process of designing

The initial AlgDesign object has four mandatory constructor arguments and two optional arguments, listed in `[]` brackets:

- *AF* - the top-level AlgFunctionality object
- *defaultPkg* - the SignalPkg
- [*designPkg*] - an alternative SignalPkg object, used in the design if given (Section 16.4)
- *sminsn* - the submodule instructions
- [*folder, comment*] - the target folder and comment for the generated VHDL files
- *writetop* - a flag that indicates whether the top-level *AF* is to be implemented or not

The top-level AF object is stored as the component `topAF`. The flag *writetop* decides if the top-level modules should be implemented or not and is stored as a property `AlgDesignWriteTop`. Assuming only the *defaultPkg* is given, it is stored as the `AlgDesignSignalPkg` attribute. As was stated in Key 15.1, all the submodules will use the stored SignalPkg¹. The SignalPkg is crucial to the automated design generation (Key 13.5). The sorting of SIGNALs follows the top-down modular approach to hardware design: the submodules are expected for the subfields.

AlgDesign has three internal data structures called `SMList`, `AFList` and the `CIRCList`:

- they have the same length as the SignalPkg²
- their initial elements are (empty) lists, accessible by the `Fid`
- they are stored as components³ of the new AlgDesign object.

The `SMList` is a list of *extracted submodules*. The initial extracted submodules are copied from the AF into the `SMList`. Recall that the design-ready top-level AF already contains⁴ extracted submodules stored by their `Fids`, if they exist.

¹ accessible via `AlgDesignSignalPkg(design)`, where *design* is the name of the AlgDesign object

² as the `SIGNALList` of the SignalPkg to be precise

³ components of the GAP object AlgDesign

⁴ stored in the component `submodules`

The `AFlist` is a list of *generated submodules*. The submodules are generated as `AlgFunctionality` objects. The `AFlist` will hold all the `AlgFunctionality` objects of the design, except for the top-level AF, stored as a component `topAF`. For the component `topAF`, there is a corresponding component `topCIRC` intended to store the top-level VHDL-ready Circuit object.

The `CIRClst` is a list of *VHDL-ready submodules*, modelled as GAP objects called `Circuit`. For now, it is sufficient to say that `Circuit` objects are generated from the `AlgFunctionality` objects and hold additional information needed to translate them into VHDL modules.

The three lists are represented graphically as ladders, with `Fid=1` for the \mathbb{F}_2 on the bottom (lowest level) and the highest `Fid` for the $\mathbb{F}_{((2^2)^2)^2}$ on the top, as shown in Figure 15.1. The *top-level* modules, i.e., the `topAF` and its corresponding `topCIRC`, are not included in the ladders. The extracted submodules from the `topAF` are shown in `SMlist` at `Fid=11`

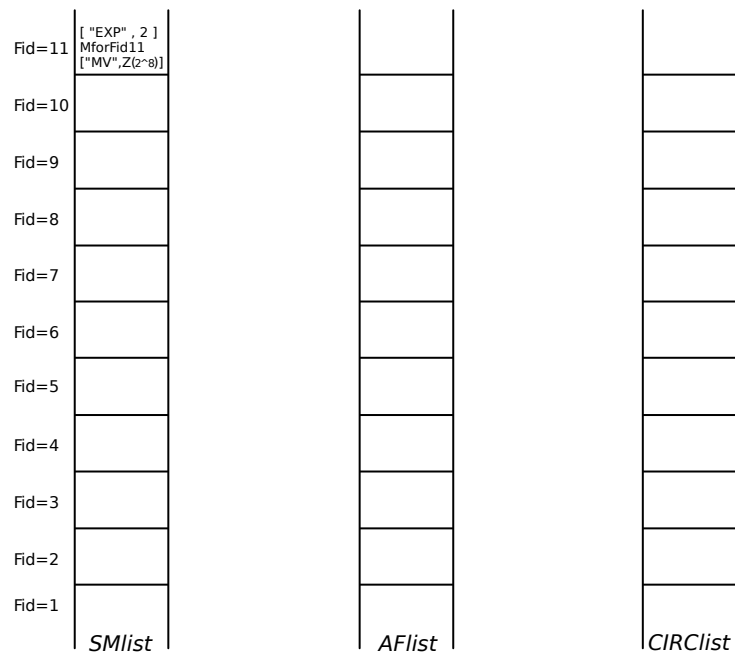


Figure 15.1: The initial `AlgDesign` lists for `SignalPkg` corresponding to $\mathbb{F}_{((2^2)^2)^2}$: graphical representation of extracted (`SMlist`), generated (`AFlist`) and VHDL-ready (`CIRClst`) submodules

The initial extracted submodules are the starting point for the processing of the *design*, which eventually produces a *datapath*.

Figure 15.2(a) shows the current stage within the dataflow diagram from Figure 12.1. The initial design (object `AlgDesign`) takes the top-level functional description of the algorithm (`AlgFunctionality`), the corresponding `SignalPkg`, and `FFCSA submodule instructions` (instead of existing basic building blocks from a library). The compilation algorithms will transform the *extracted submodules* to *generated submodules* and finally to *VHDL-ready submodules*, i.e., the *VHDL-ready datapath*, shown in Figure 15.2(b).

The internal structure of AlgDesign was shown in Figure 15.1. The middle pillar, i.e., AFlist and topAF conceptually form *design*, shaded grey in Figure 15.2(a). The rightmost pillar, i.e., CIRClst and topCIRC conceptually form *datapath*, shaded grey in Figure 15.2(b).

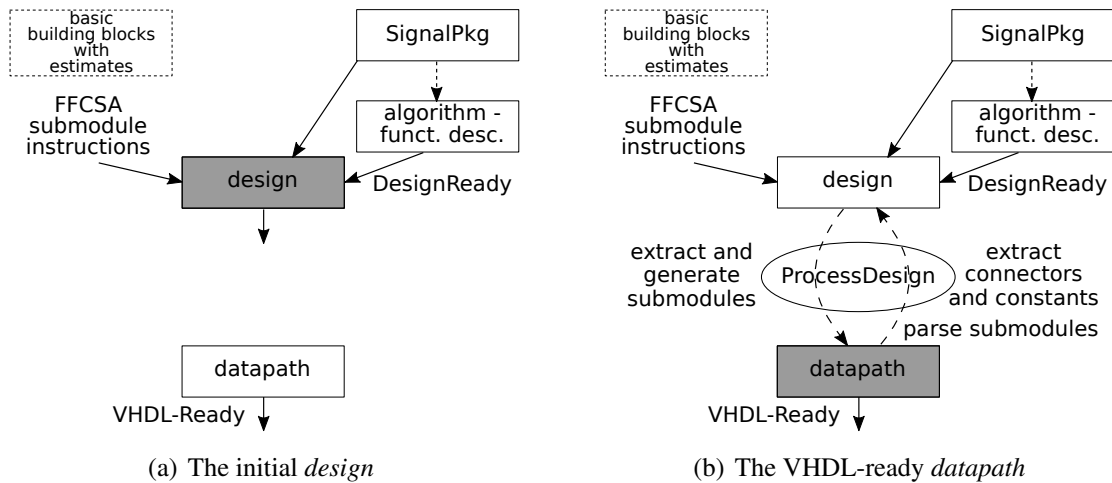


Figure 15.2: The current stage in the design-flow diagram: (a) the initial design, and (b) the VHDL-ready datapath

Key 15.2: The design and the datapath

The AlgDesign holds three lists, the SMList of submodules needed, the AFlists of AlgFunctionality objects generated for the submodules from the SMList, and the CIRClst of the VHDL-ready Circuit objects, corresponding to the generated AlgFunctionality objects. The structure of the SignalPkg object plays a crucial role for automation: it allows for the *top-down modular approach in which the submodules are generated for the subfields*. All three lists are accessible by the Fid's from the stored AlgDesignSignalPkg. There is an almost clean split of the AlgDesign object into *design* and *datapath*, with only the AlgDesignSignalPkg as their overlap:

- *design*: AFlist and topAF
- *datapath*: CIRClst and topCIRC

The entries of the SMList, together with the FFCSA submodule instructions, are used to generate the AlgFunctionality objects for the submodules. The submodule instruction format example:

```
sminsn := [ "generate", ["matrixU"], "CICO", "simple"]
```

The first entry `generate` indicates the submodules will be generated on the fly (instead of fetching them from a library). The next entry in `sminsn` is a list containing the actual “how-to”, which will

invoke the appropriate FFCSA method and generate the expressions needed to drive the submodule outputs. The last part of *sminsn* are the *archtype* instructions needed for the new AlgFunctionality.

All the submodules in the design will be generated using the same instructions, but the second entry is a list that allows for variations, e.g., one FFCSA method can be used for inversions and another for all other exponentiations. After passing a few simple checks for valid keywords, the entire *sminsn* is stored as an attribute of AlgDesign, called SubmoduleInstructions. For example, given and SMList entry ["EXP", *e*] at SMList index Fid use:

1. Fid to obtain field *F* and corresponding basis *B* from the SignalPkg
2. ChooseFieldElms on *F* to obtain a vector of GAP variables⁵ *avec* (Table 7.5)
3. FFA_exp_matrixU(*B*, *avec*, *e*) to get the expressions for the exponentiation (Section 7.4)

■ **Implementation detail:** The encoding for the extracted submodules was given in Table 14.6 in Section 14.2: possible submodules are ["EXP", *e*], "MforFid#" and ["MV", *c*], where *e* is the value of the exponent, # is replaced by the actual Fid and *c* = γ is the multiplicative constant. ■

The AlgDesign object			
<i>attributes/properties</i>	<i>components</i>		
AlgDesignSignalPkg		SMList	commentstr
SubmoduleInstructions	topAF	AFlist	folder
AlgDesignWriteTop †	topCIRC	CIRClist	filelist
		CONNlist	

Notes: † property

Table 15.1: Main functionality of the CIRCUIT package - the AlgDesign object

The attributes, properties, and components of the AlgDesign object are listed in Table 15.1. The components *folder* and *commentstr* are self-explanatory. Component *filelist* stores all the generated VHDL files, and is used to write scripts for the synthesis tools. The component *tt* CONNlist, holding *connectors* (Table 15.2), will be explained at the end of this section and in Example 15.4.1. The optional *designPkg* argument will be explained in Subsection 16.4.

■ **Implementation detail:** The submodule instructions *sminsn* will always contain the keyword "generate", but the keyword "fetch" is reserved for possible future use (recall the Key 13.1 and discussion in the motivation examples in Section 13.1: the submodules can also be fetched from a database of basic building blocks). ■

⁵ recall that the dimension of *F* = the length of *B* = the length of *avec*

15.2 Top-down processing

The first compilation algorithm is called `ProcessSMAFloop`. As the name suggests, it will retrieve extracted submodules from the `Smlist` and generate their corresponding `AlgFunctionality` objects. Each new `AlgFunctionality` can have extracted submodules of their own, which are added to the `Smlist`, and the process repeats.

Detailed description of `ProcessSMAFloop`

When the initial `AlgDesign design` is created, any extracted submodules of the top-level `AlgFunctionality` are copied into the appropriate index of the `Smlist`. If the initial `Smlist` is empty, i.e., the `topAF` is of class 0,1 or 3 (see Table 10.4 in Section 10.3), no additional processing is needed. For all other cases, the `Smlist` will have entries at the *element* type `Fids` and the `ProcessSMAFloop` will start. The new `Fids` are obtained using `SignalPkg` methods from Table 13.6 in Section 13.3 .

The loop maintains two lists of indices, the `SMidlist` for `Fids` where `Smlist[Fid]` is *not empty*, and the `AFidlist` for `Fids` where `Aflist[Fid]` was *not yet processed*. One pass of `ProcessSMAFloop` performs the following steps:

1. collect all `Fids` for which the `Smlist` is not empty into the `SMidlist` and order the list in *descending* order for *top-down* processing. Create an empty `AFidlist` to store the indices returned by the `ProcessSmlistByFieldID` method.
2. for each `SMid` in `SMidlist` call `ProcessSmlistByFieldID`, skip⁶ `SMid=1`
 - obtain `newAFid` as `FieldIDToVectorID(AlgDesignSignalPkg(design, SMid))`. By the structure of the `SignalPkg`, the submodules are *found* (extracted) for the *element* `SMids`, and the `AlgFunctionality` objects are *generated* for *vector* `newAFids`.
 - `newAFid` is used for the ports of the generated `AlgFunctionality` objects
 - `FunctName` set to the exact encoding of the submodule in `Smlist[Fid]` (Table 14.6).
 - *remove* all submodules from `Smlist[SMid]` and create their corresponding `AlgFunctionality` objects, using the `SubmoduleInstructions`
 - the generated `AlgFunctionality` objects are added to the `Aflist[newAFid]`. By the structure of the `SignalPkg`, the `AlgFunctionality` objects are always added to the *vector* indices `newAFid` of the `Aflist`.
 - return `newAFid`, which is added to the `AFidlist` of the *not yet processed* indices.
3. when all `SMids` from `SMidlist` are processed (the `Smlist` is currently also empty, because method `ProcessSmlistByFieldID` removes the submodules), proceed to the `AFidlist`. For each `AFid` from `AFidlist` call `ProcessAflistByFieldID`:
 - obtain `newSMid` as `VectorIDToSGDFieldID(AlgDesignSignalPkg(design, AFid))`. By the structure of the `SignalPkg`, the `AFid` is a *vector* type, and new `AlgFunctionality` submodules are found (extracted) for *element* `newSMids`.

⁶ submodules are not possible for \mathbb{F}_2

- call `MakeAlgFunctionalityDesignReady` for each entry of the `AFlist[AFid]`
 - if a submodule is found⁷ and is not yet listed at `Smlist[newSMid]`, then add it
4. when all `AFids` are processed, check the `Smlist`: if empty, `ProcessSMAFloop` is finished, otherwise start a new pass, i.e., return to step 1.

When `ProcessSMAFloop` is finished, the `Smlist` is empty and will not be populated again. At this point the `AFlist` contains the final *design* (Key 15.2), no more `AlgFunctionality` objects will be added. The methods described above are listed in Table 15.3 at the end of this section.

Key 15.3: Top-down processing: generate all submodules

The structure of `SignalPkg`, inherited by the `Smlist` and `AFlist`, allows a top-down processing of the initial `AlgDesign` in a `ProcessSMAFloop`. It transforms all the submodules in `Smlist` into `AlgFunctionality` objects, and adds them to the `AFlist`. Then it extracts submodules from the newly created `AlgFunctionality` objects and adds them to the `Smlist`. All the populated `Fids` in the `Smlist` are *element* type, and all the populated `Fids` in the `AFlist` *vector* type `SIGNALs`. When `ProcessSMAFloop` is finished, the `Smlist` is empty and the `AFlist` contains the final *design*.

■ **Discussion:** The `ProcessSMAFloop` uses two `SignalPkg` methods, `FieldIDToVectorID` and `VectorIDToSGDFieldID` (Table 13.6). Different compositions of these two methods gives `FieldIDToSubID` and `VectorIDToSubID`. For `SignalPkg pkg`, stored as `AlgDesignSignalPkg` and a valid field identifier `Fid`:

```
FieldIDToSubID (pkg, Fid) = VectorIDToSGDFieldID (FieldIDToVectorID (pkg, Fid))
VectorIDToSubID (pkg, Fid) = FieldIDToVectorID (VectorIDToSGDFieldID (pkg, Fid))
```

See Table 13.6 in Section 13.3 and Examples 13.3.1 and 13.3.2 for `SignalPkg` methods. As the `Smlist` is populated by the *element* `Fids`, the passes of `ProcessSMAFloop` follow the `FieldIDToSubID`: 11 → 8 → 5 in Example 15.2.1. Similarly, the `AFlist` is populated by the *vector* `Fids`, and the passes follow the `VectorIDToSubID`: 9 → 6 → 2 in Example 15.2.1.

Hence, it is possible to know for which `Fids` the submodules and their `AlgFunctionality` objects are expected. Without generating the `AlgFunctionality` objects, however, it is not possible to know if and which submodules are needed. ■

⁷ during `AlgFunctionalityCollectAllSubmodules` of `MakeAlgFunctionalityDesignReady` (Table 14.4)

Example 15.2.1 Datapath synthesis for an expression over $\mathbb{F}_{((2^2)^2)^2}$ – top-down processing

This example shows the graphical representation of the three lists in the initial AlgDesign obtained from the AF implementing the expression in equation (13.1) defined over the tower field $\mathbb{F}_{((2^2)^2)^2}$ (Example 13.1.3 in Section 13.1). The extracted submodules are shown at Fid=11 in the SMList. The passes of the ProcessSMAFloop are shown graphically:

- pass 1 of the ProcessSMAFloop is shown in Figure 15.3:
 - three submodules were removed from SMList[11], shaded grey
 - ProcessSMListByFieldID (design, 11) creates three new AlgFunctionality objects to be added at newAFid=9, i.e., AFlist[9]
 - ProcessAFlistByFieldID (design, 9) finds 7 new submodules and adds them to newSMid=8, i.e., to SMList[8]. There are 5 distinct matrix-vector multipliers MV, but the exact values of the constants are omitted from Figure 15.3

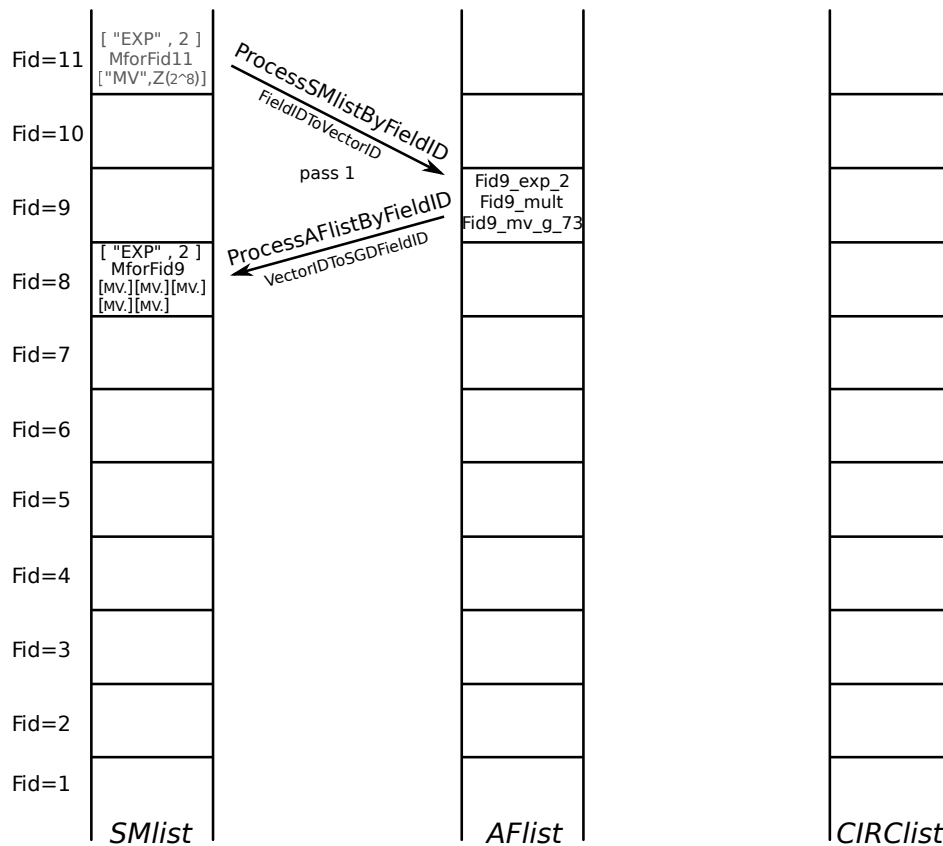


Figure 15.3: Pass 1 of the ProcessSMAFloop for the AlgDesign with the expression in equation (13.1)

- pass 2 of the ProcessSMAFloop is shown in Figure 15.4:
 - 7 submodules were removed from SMList[8], shaded grey
 - ProcessSMListByFieldID (design, 8) creates 7 new AlgFunctionality objects to be added at newAFid= 6, i.e., AFList[6]
 - ProcessAFlistByFieldID (design, 6) finds four new submodules and adds them to newSMid=5, i.e., to SMList[5]. There are 2 distinct matrix-vector multipliers MV, but the exact values of the constants are omitted from Figure 15.4

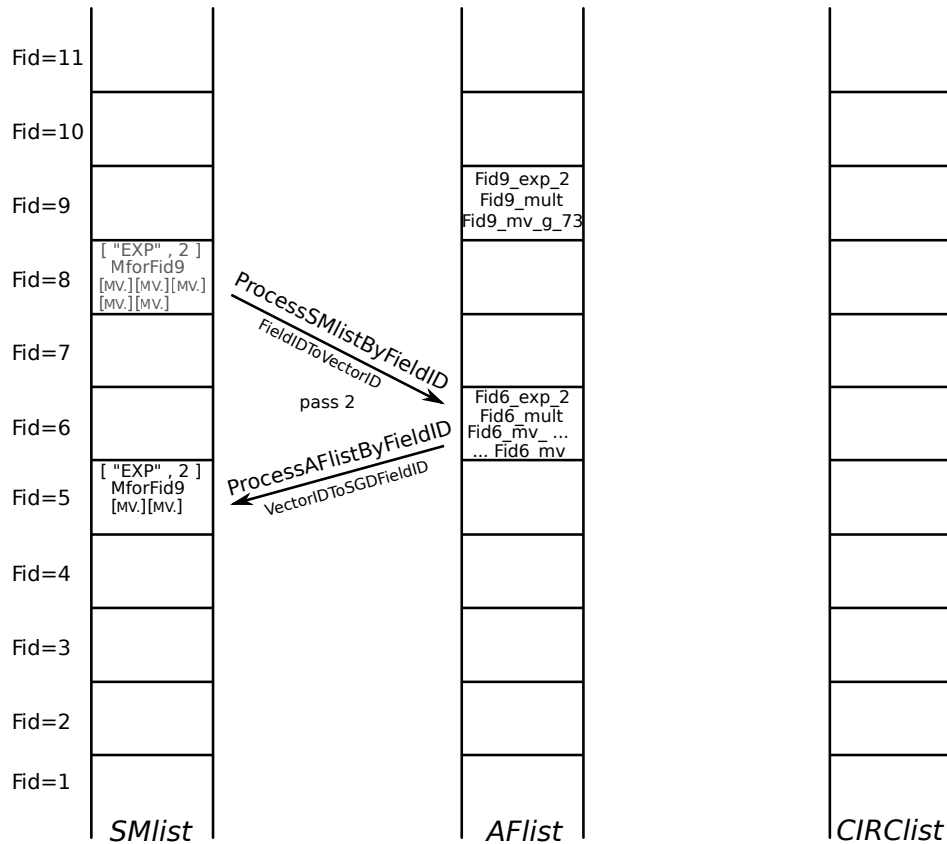


Figure 15.4: Pass 2 of the ProcessSMAFloop for the AlgDesign with the expression in equation (13.1)

- pass 3 of the ProcessSMAFloop is shown in Figure 15.5:
 - four submodules were removed from SMList[5], shaded grey
 - *ProcessSMListByFieldID* (design, 5) creates four new AlgFunctionality objects to be added at newAFid= 2, i.e., AFlist[2]
 - *ProcessAFlistByFieldID* (design, 2) terminates without adding new submodules because no submodules exist in \mathbb{F}_2

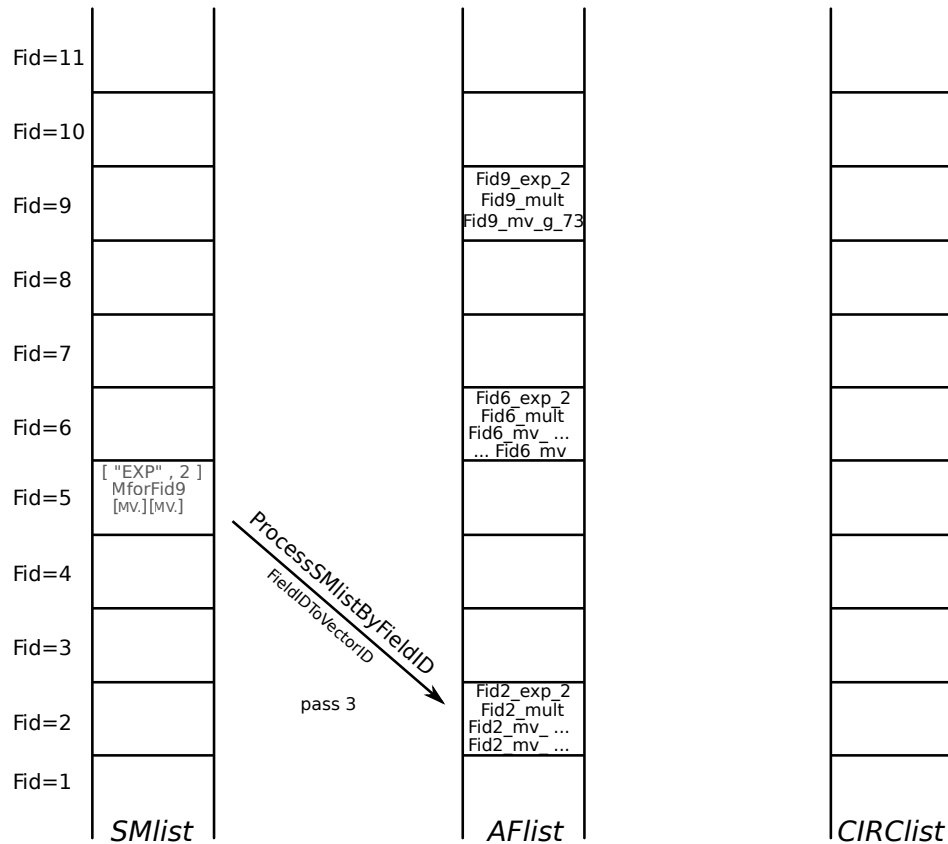


Figure 15.5: Pass 3 of the ProcessSMAFloop for the AlgDesign with the expression in equation (13.1)

15.3 Bottom-up processing

The second compilation algorithm in the CIRCUIT package is the `ProcessAFCIRCLoop`. It traverses the `AFlist` in a bottom-up fashion, gradually building a VHDL-ready datapath. The `AlgFunctionality` object does not contain all the information needed to write the VHDL code, i.e., it gives the *functional description* of an algorithm/expression/module. The `AlgDesign` object has methods to extract and generate the needed submodules but does not yet tie the submodules to the “parent” modules. This step is accomplished by another kind of binding, in this case: the binding of operations to the submodules. One last missing piece are the additive constants, which will be explained at the end of this section.

Detailed description of `ProcessAFCIRCLoop`

In the bottom-up loop `ProcessAFCIRCLoop`, all the `AlgFunctionality` objects, starting with the lowest `Fid` for which the `AFlist` is not empty, are transformed into `Circuit` objects, using the method `AFtoCIRC (design, AF)`. In *ascending* `Fid` order, `ProcessAFCIRCListByFieldID` is called for each non-empty `AFlist[Fid]`. `ProcessAFCIRCListByFieldID` which is one pass of the `ProcessAFCIRCLoop`, and proceeds as follows:

- for each `AlgFunctionality` object `AF` at `AFlist[Fid]`, call `AFtoCIRC`, which will
 - create the empty lists `parsedSmlist` and `constlist`
 - if the `AF` has submodules:
 - * for each submodule, find its corresponding `Circuit` object by its `FuncName` at a lower `Fid` and parse its `entitylist`:
$$\text{parsed submodule list : [FuncName, EntityName, ArchName, EntityGenerics]} \quad (15.1)$$

then add the parsed submodule list to `parsedSmlist`
 - * `FuncName` binds the arithmetic operation to its submodule. Recall that `FuncName` was set to submodule encoding (Table 14.6) by `ProcessSmlistByFieldID`
 - * `EntityName` and `ArchName` parameters of the parsed `entitylist` will be used directly for the VHDL component instantiation of the submodule
 - if `AF` has additive constants:
 - * transform each constant into a directed graph⁸ and add it to `constlist`. More details on digraphs will follow after Example 15.3.1
 - * while there can be only one additive constant, i.e., constant term, per ANF expression, (each coordinate of) each output can have its own constant term
 - * the directed graph is used for VHDL declaration of the corresponding constant (see declaration (15.3) and Example 15.3.2)
 - create the `Circuit CIRC` corresponding to the `AF` and add it to the `CIRClist[Fid]` :

$$\text{CIRC} = \text{Circuit}(\text{AF}, \text{parsedSmlist}, \text{constlist}) \quad (15.2)$$

⁸ using the GAP package Digraphs [35]

Key 15.4: Bottom-up processing: submodules and additive constants

The bottom-up `ProcessAFCIRCLoop` transforms each `AlgFunctionality` object in the `AFlist` into a VHDL-ready `Circuit` object by adding the missing information about its submodules, which are now available, and about any additive constants present. The `Circuit` objects are added to the `CIRClst` at the same `Fid` as their corresponding `AlgFunctionality` objects.

The `Circuit` constructor call (15.2) takes three parameters: the `AlgFunctionality` object `AF`, the list of parsed submodules `parsedSmlst`, and the list of directed graphs for all additive constants `constlist`. Since the `Circuit` is just a VHDL-ready version of the `AlgFunctionality` object, it stores the same information as the latter (Table 14.1), with an extra `ConstantsList` and with *encoded submodules* (Table 14.6 in Section 14.2) replaced by the *parsed submodule* lists (15.1). The main difference between `AlgFunctionality` and `Circuit` objects is in the submodules: for `AlgFunctionality`, the submodules are black boxes (only their functionality is known), but for `Circuit` objects, the generated submodules and their exact information (the parsed submodule lists (15.1)) are available. The combination of top-down extracting and generating submodules and the bottom-up parsing of submodules also implicitly *binds the arithmetic operations and submodules*; the submodules are identified by their *FuncName*. When the `ProcessAFCIRCLoop` terminates, the `AFlist` is empty and the `CIRClst` contains the VHDL-ready `Circuit` objects.

Key 15.5: AlgFunctionality vs Circuit

The main difference between `AlgFunctionality` and `Circuit` objects is in the submodules: for `AlgFunctionality`, the submodules are black boxes (only their functionality is known), but for `Circuit` objects, the actual submodules and their exact information, i.e., the parsed submodule lists (15.1), are available. The combination of top-down and bottom-up processing of submodules also implicitly binds the arithmetic operations and submodules using the `FuncName` attribute.

Example 15.3.1 Datapath synthesis for an expression over $\mathbb{F}_{((2^2)^2)^2}$ – bottom-up processing

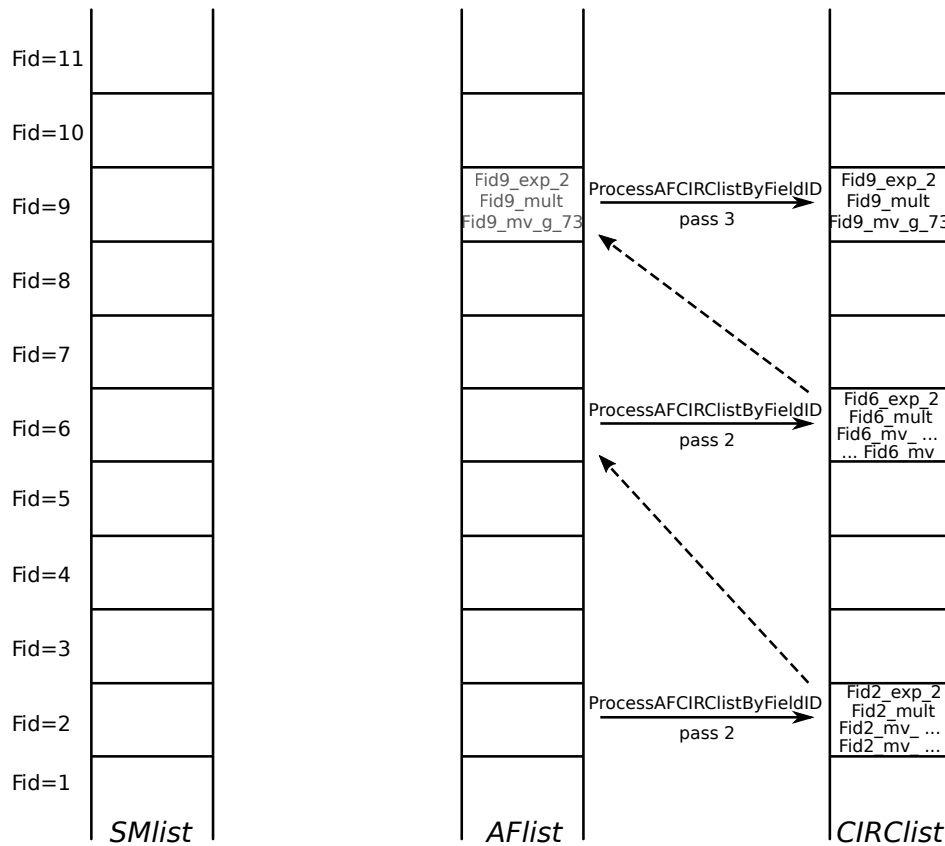


Figure 15.6: The ProcessAFCIRLoop for the AlgDesign with the expression in equation (13.1)

This example continues the Example 15.2.1. The passes of the ProcessAFCIRLoop are shown graphically in Figure 15.6, the constant graphs are omitted. For brevity, all three passes are shown in the same figure. The top-down ProcessSMAFloop shown in Example 15.2.1 took 3 passes, and the bottom-up ProcessAFCIRLoop takes 3 passes as well:

- pass 1 of the ProcessAFCIRLoop is shown at the bottom: since there are no submodules, all four AlgFunctionality objects from *AFlist*[2] are simply transformed into Circuit
- pass 2 takes the 7 AlgFunctionality objects from *AFlist*[6], parses their submodules (dashed arrow from *CIRclist*[2] to *AFlist*[6] in Figure 15.6), and creates 7 Circuit objects for *CIRclist*[6]
- pass 3 takes the 3 AlgFunctionality objects from *AFlist*[9], parses their submodules (dashed arrow from *CIRclist*[6] to *AFlist*[9] in Figure 15.6), and creates 3 Circuit objects for *CIRclist*[9]

The GAP code Example D.4.1 in Appendix D.4 shows the *CIRclist* contents after processing.

Additive constants

The additive constants from a tower field need to be represented in a form that allows for them to be specified in VHDL. Recall (sub)type definitions (13.8-13.14) in Section 13.3, especially (13.14), repeated below:

```
type signal is array (0 to m-1) of sub-signal
```

A type *signal* constant *c* can then be specified in VHDL as a tuple of coordinates c_0, \dots, c_{m-1} , which were already specified as type *sub-signal* constants:

```
constant c : signal := (c_0, ..., c_{m-1})
```

 (15.3)

In order to obtain all the information needed, a tower field constant is represented as a directed graph, where a node *c* contains the coordinates c_0, \dots, c_{m-1} as children, and each node label contains the following information: the string for the constant, the value of the constant and the Fid to identify its type in the corresponding SignalPkg used. The leaf nodes contain simple binary values for coordinates whose types are `std_logic_vector`. If the constant is not a tower field constant but belongs to a finite field constructed as one extension, the directed graph consists of one node with a single leaf. Example 15.3.2 is a continuation of Example D.3.1: Figure 15.7 shows the digraph for the constant $\gamma_2 \in \mathbb{F}_{((2^2)^2)^2}$ and the corresponding VHDL declarations are shown in VHDL Example 15.3.2.

Example 15.3.2 Declarations for additive constants in $\mathbb{F}_{((2^2)^2)^2} \longleftrightarrow$

Figure 15.7 shows the directed graph for the constant $\gamma_2 = \nu^3$, where ν is a root of the reference defining polynomial $p_3(x) = x^8 + x^4 + x^3 + x^2 + 1$ (see Table D.1 from Example 13.2.1). The exact tower field construction is specified in Example 13.3.1 and the SignalPkg `defaultPkg` shown in a diagram in Figure 13.5 in Example 13.3.2, in Section 13.3. The root node in Figure 15.7 is γ_2 with Fid=9. The next level are its two coordinates with Fid=6, and lastly the coordinates with Fid=2, which are shown two layers: the coordinates (constants) itself, and their `std_logic_vector` values `00`, `11`, `01`, `10`. The graph in Figure 15.7 is “top-down”, with $\gamma_2 \in \mathbb{F}_{((2^2)^2)^2}$. The VHDL code VHDL Example 15.3.2 declares the constants “bottom-up”, with γ_2 , called `Fid9_c_g_3` in VHDL, declared last.

VHDL Example 15.3.2

```
constant Fid9_c_g_3_0_0: ffe_2 := "00";
constant Fid9_c_g_3_0_1: ffe_2 := "11";
constant Fid9_c_g_3_1_0: ffe_2 := "01";
constant Fid9_c_g_3_1_1: ffe_2 := "10";
constant Fid9_c_g_3_1: ffe_6 := (Fid9_c_g_3_1_0, Fid9_c_g_3_1_1);
constant Fid9_c_g_3_0: ffe_6 := (Fid9_c_g_3_0_0, Fid9_c_g_3_0_1);
constant Fid9_c_g_3: ffe_9 := (Fid9_c_g_3_0, Fid9_c_g_3_1);
```

←

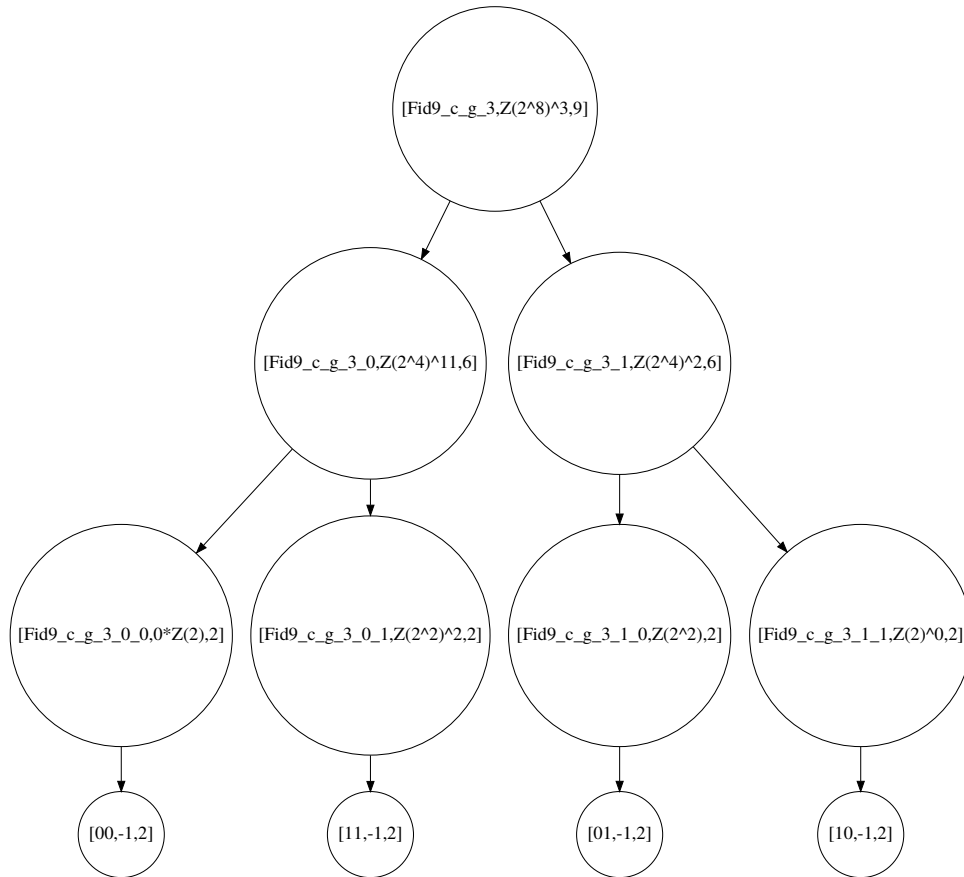


Figure 15.7: The directed graphs for the additive constant γ_2

15.4 Connectors for the tower field elements and vectors

The tower fields require some extra management. First are the aforementioned additive constants, which are transformed into directed graphs and stored as a part of the `Circuit` object. Second are the *connectors*, which are used for the type conversions between the tower field *element* and *vector SIGNALS*.

Connectors are used for conversions between different types. As the connectors are only needed for tower field constructions, they are explained on the example tower field construction used in Examples 13.1.3-15.3.2, with the exact `SignalPkg` called `defaultPkg` shown in Examples 13.3.1 and 13.3.2. Recall the “per-level” bases and the tower field bases *TFB* of the isomorphic field (equation (7.1) in Section 7.3.1):

- “per-level” bases: $B_{\mathbb{F}_{(2^2)^2}/\mathbb{F}_{(2^2)^2}}$ and $B_{\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}}$ (shown on the left side in the diagram in Figure 13.5 from Example 13.3.2)
- tower field bases *TFB*: $TFB_{\mathbb{F}_{2^8}/\mathbb{F}_2}$ and $TFB_{\mathbb{F}_{2^4}/\mathbb{F}_2}$ (shown on the right side in the diagram in Figure 13.5 from Example 13.3.2)

For each of the bases above, two SIGNAL objects are created, namely an *element* \mathcal{F}_r and a *vector* \mathcal{V}_r (with $m = 2$ for the tower field construction used in Examples 13.1.3-15.3.2).

The method `TFieldIDToFieldID` in Table 13.6 takes a `Fid` of an *element* SIGNAL with a “per-level” basis and returns the `Fid` of the *element* SIGNAL with the corresponding tower field basis *TFB* of the isomorphic field (equation (7.1) in Section 7.3.1). For example, the `Fid=11` *element* SIGNAL \mathcal{F}_6 with the “per-level” *UnderlyingBasis* $B_{\mathbb{F}_{(2^2)^2}/\mathbb{F}_{(2^2)^2}}$ (see Table 13.9 for details), has its counterpart `Fid=10` *element* SIGNAL \mathcal{F}_4 with interpretation basis $TFB_{\mathbb{F}_{2^8}/\mathbb{F}_2}$ obtained as shown in equation (7.1), i.e., `TFieldIDToFieldID(defaultPkg, 11) = 10`.

For all the SIGNALs in `defaultPkg`, that have different VHDL (sub)type declarations, a *pair of connectors* is needed between:

- *element* \mathcal{F}_r (`tffe`) and *vector* \mathcal{V}_r (`tfvec`) types w.r.t. the same tower field “per-level” basis (the *UnderlyingBasis* of the *element* signal is also the interpretation basis of the *vector* signal, and the two SIGNALs are matched using the method `FieldIDToVectorID` from Table 13.6). There are two connectors, called *element-vector connectors*, namely `tffe_to_tfvec` and its inverse `tfvec_to_tffe`, shown in the first row of Table 15.2.
- SIGNALs for the tower field “per-level” bases (`tffe` or `tfvec`) and SIGNALs for the *TFB* bases of the isomorphic finite fields (`ffe` or `fvec`):
 - *element* SIGNALs `tffe` and `ffe` that can be matched with the `TFieldIDToFieldID` method as explained above. There are two connectors, called *element-element connectors*, namely `tffe_to_ffe` and its inverse `ffe_to_tffe`, shown in the second row of Table 15.2.
 - *vector* SIGNALs `tfvec` and `fvec` that can be matched with a combination of methods `TFieldIDToFieldID` and `FieldIDToVectorID`. There are two connectors, called *vector-vector connectors*, namely `tfvec_to_fvec` and its inverse `fvec_to_tfvec`. The connectors are shown in the third row of Table 15.2, and details on methods `TFieldIDToFieldID` and `FieldIDToVectorID` in the last row of Table 15.2.

No connectors are needed for the *element* \mathcal{F}_r (`ffe`) and *vector* \mathcal{V}_r (`fvec`) types for the isomorphic finite fields, because they both have the same VHDL declaration `std_logic_vector` (see declarations (13.9) and (13.11) for $p = m$ case). The same rationale applies to why the connectors are not needed if there is no tower field in the design. The three types of connectors that are required for the VHDL implementation are shown in Table 15.2, with examples from `defaultPkg` in the last column (see Examples 13.3.1 and 13.3.2 for details on `defaultPkg`).

The connectors themselves are kept in the `CONNlist` component of the `AlgDesign` object (see Table 15.1), which follows the structure of the `SignalPkg` used (just as the `SMList`, `AFList` and `CIRCList` components). Each connector is always stored at the `Fid` of its origin, thus decoupling the connector pairs, as can be seen in Example 15.4.1. The `CONNlist` entries are obtained with the `ExtractConnectors` method, which only depends on the `SignalPkg`. Chapter 16 will discuss how the connectors are used.

<i>connector pairs</i>		example w.r.t. defaultPkg †	comments
<i>element-vector</i>	$\begin{array}{ccc} \mathcal{V}_r / \mathcal{V}_{rp} & \begin{array}{c} \xleftarrow{\text{tffe_to_tffvec}} \\ \Leftarrow \\ \xrightarrow{\text{tffvec_to_tffe}} \end{array} & \mathcal{F}_r / \mathcal{F}_{rp} \end{array}$	$\begin{array}{ccc} \text{Fid} = 9 & \begin{array}{c} \xleftarrow{\text{tffe_to_tffvec}} \\ \Leftarrow \\ \xrightarrow{\text{tffvec_to_tffe}} \end{array} & \text{Fid} = 11 \end{array}$	between <i>element</i> tffe and <i>vector</i> tffvec types w.r.t. the same tower field “per-level” basis (matched with Field-IDToVectorID, Table 13.6)
<i>element-element</i>	$\begin{array}{ccc} \mathcal{F}_{r_1} / \mathcal{F}_{rp_1} & \begin{array}{c} \xleftarrow{\text{tffe_to_ffe}} \\ \dashrightarrow \\ \xrightarrow{\text{ffe_to_tffe}} \end{array} & \mathcal{F}_{r_2} / \mathcal{F}_{rp_2} \end{array}$	$\begin{array}{ccc} \text{Fid} = 10 & \begin{array}{c} \xleftarrow{\text{tffe_to_ffe}} \\ \dashrightarrow \\ \xrightarrow{\text{ffe_to_tffe}} \end{array} & \text{Fid} = 11 \end{array}$	between <i>elements</i> for the “per-level” bases tffe and <i>elements</i> for the <i>TFB</i> bases of the isomorphic fields ffe
<i>vector-vector</i>	$\begin{array}{ccc} \mathcal{V}_{r_1} / \mathcal{V}_{rp_1} & \begin{array}{c} \xleftarrow{\text{fvec_to_tffvec}} \\ \dashrightarrow \\ \xrightarrow{\text{tffvec_to_fvec}} \end{array} & \mathcal{V}_{r_2} / \mathcal{V}_{rp_2} \end{array}$	$\begin{array}{ccc} \text{Fid} = 9 & \begin{array}{c} \xleftarrow{\text{fvec_to_tffvec}} \\ \dashrightarrow \\ \xrightarrow{\text{tffvec_to_fvec}} \end{array} & \text{Fid} = 4 \end{array}$	between <i>vectors</i> for the “per-level” bases tffvec and <i>vectors</i> for the <i>TFB</i> bases of the isomorphic fields fvec
<i>vector-vector connector details</i>			
$\begin{array}{ccc} \mathcal{V}_{r_1} / \mathcal{V}_{rp_1} & \begin{array}{c} \xleftarrow{\text{fvec_to_tffvec}} \\ \dashrightarrow \\ \xrightarrow{\text{tffvec_to_fvec}} \end{array} & \mathcal{F}_{r_1} / \mathcal{F}_{rp_1} \\ & & \dashrightarrow \\ & & \mathcal{F}_{r_2} / \mathcal{F}_{rp_2} \\ & & \xrightarrow{\text{tffvec_to_fvec}} \end{array}$			

Notes:
notation (13.7) is used: *signal/sub – signal*, i.e., $\mathcal{F}_r / \mathcal{F}_{rp}$ and $\mathcal{V}_r / \mathcal{V}_{rp}$ \Leftarrow FieldIDToVectorID
† defaultPkg for $\mathbb{F}_{(2^3)^2}$ diagram in Figure 13.5 \dashrightarrow TFieldIDToFieldID

Table 15.2: Connectors for the tower field elements and vectors

Key 15.6: Conversions between different data types

A pair of connectors is needed for conversions between different data types, when tower field constructions are used in the design: *element-vector connectors* (w.r.t. the same tower field “per-level” basis), *element-element connectors* and *vector-vector connectors* (both between the tower field “per-level” bases and the *TFB* bases of the isomorphic finite fields).

Example 15.4.1 Connectors for the tower field elements and vectors in $\mathbb{F}_{((2^2)^2)^2} \longleftrightarrow$

Below are all the connectors needed for the tower field construction used in the running example in this section. The exact tower field construction is specified in Example 13.3.1 and the SignalPkg defaultPkg shown in a diagram in Figure 13.5 in Example 13.3.2, in Section 13.3.

The output is formatted as follows: for each existing Fid_1, the *element-vector connector* (first row in Table 15.2), if it exists, is specified as Fid_1 -> tffe_to_tfvec -> Fid_2 and its pair connector is listed in the row for Fid_2 as Fid_2 -> tfvec_to_tffe -> Fid_1. Next the *element-element connectors* (second row in Table 15.2) are listed in the same format, and the *vector-vector connectors* (third row in Table 15.2) are listed last. If a connector doesn't exist, -1 is listed instead.

Example 15.4.1

```

-----
AlgDesign with CONNlist
11: 11 -> tffe_to_tfvec -> 9      11 -> tffe_to_ffe -> 10      -1
10:           -1                  10 -> ffe_to_tffe -> 11      -1
9: 9 -> tfvec_to_tffe -> 11          -1                9 -> tfvec_to_fvec -> 4
8: 8 -> tffe_to_tfvec -> 6      8 -> tffe_to_ffe -> 7          -1
7:           -1                  7 -> ffe_to_tffe -> 8          -1
6: 6 -> tfvec_to_tffe -> 8          -1                6 -> tfvec_to_fvec -> 3
5:           -1                  -1                    -1
4:           -1                  -1                    4 -> fvec_to_tfvec -> 9
3:           -1                  -1                    3 -> fvec_to_tfvec -> 6
2:           -1                  -1                    -1
1:           -1                  -1                    -1
-----

```



15.5 Putting it all together

The main methods described in this section are summarized in Table 15.3. They are not called individually, but are a part of the method called `ProcessDesign`, which performs the following steps for the given `AlgDesign` object `design`:

1. `ExtractConnectors(design)`: connectors for the tower field elements and vectors
2. `ProcessSMAFloop(design)`: top-down processing of the design
3. `ProcessAFCIRClloop(design)`: bottom-up processing of the design
4. finally, if `AlgDesignWriteTop` is true, the `topAF` is also transformed⁹ to a `Circuit` object and stored as `topCIRC`.

design - an AlgDesign object	AF - an AlgFunctionality object	Fid - field identifier
top-down processing		bottom-up processing
<code>ProcessSMListByFieldID (design, Fid)</code>		<code>AFtoCIRC (design, AF)</code>
<code>ProcessAFListByFieldID (design, Fid)</code>		<code>ProcessAFCIRClListByFieldID (design, Fid)</code>
<code>ProcessSMAFloop (design)</code>		<code>ProcessAFCIRClloop (design)</code>
<code>ProcessDesign (design)</code>		

Table 15.3: Main functionality of the CIRCUIT package - methods for the `AlgDesign`

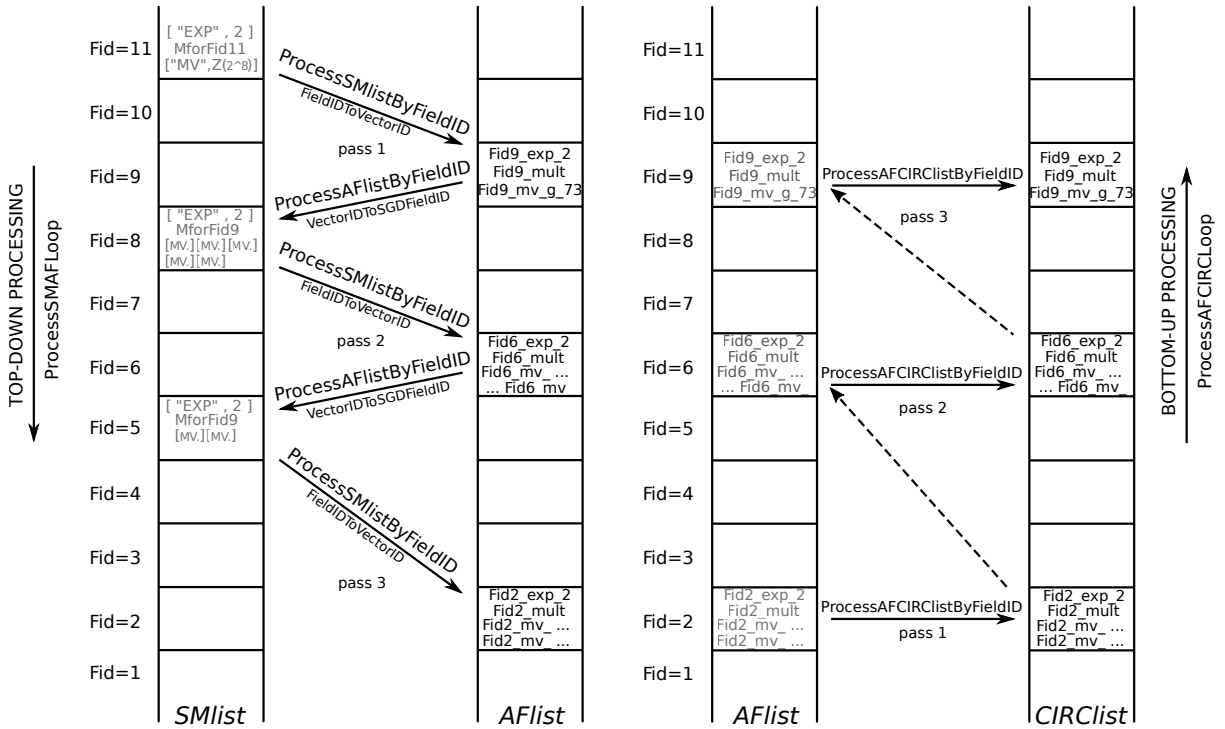
Key 15.7: Putting it all together - `ProcessDesign`

The method `ProcessDesign` takes an initial `AlgDesign`, with the functional description of the algorithm stored as `topAF`, and builds the entire *design* (`AFList` and `topAF`) in a top-down loop. It then transforms the *design* into the *datapath* (`CIRClList` and `topCIRC`) in a bottom-up loop. The remaining information, needed for the VHDL implementation, is captured as the `AlgDesignSignalPkg` attribute and as the component `CONNlist`. The `AlgDesign` object is now considered to be VHDL-ready, with the *datapath* captured in `CIRClList` and `topCIRC`.

⁹ using the `AFtoCIRC` method in (15.2)

Example 15.5.1 Datapath synthesis for an expression over $\mathbb{F}_{(2^2)^2}$ – revisited \longleftrightarrow

This example is merely a summary of Examples 15.2.1 and 15.3.1. It shows the completed ProcessSMAFloop in Figure 15.8(a) and completed ProcessAFCIRCLoop in Figure 15.8(b). The SMList, AFList and CIRCList are represented graphically as ladders, with Fid=1 for the \mathbb{F}_2 on the bottom (lowest level) and the highest Fid on the top. SMList holds extracted submodules, encoded as shown in Table 14.6, the AFList holds the generated submodules - the design, and the CIRCList holds the VHDL-ready submodules - the datapath.



(a) Top-down processing: ProcessSMAFLoop

(b) Bottom-up processing: ProcessAFCIRCLoop

Figure 15.8: The ProcessDesign for the AlgDesign with the expression in equation (13.1)

The ProcessSMAFloop is shown in Figure 15.8(a): three passes are required to finish the loop, and each pass consists of ProcessSMListByFieldID, which removes the submodules from the SMList (shaded grey), followed by the ProcessAFListByFieldID, which extracts new submodules, if they exist. No submodules are found in pass 3. The bottom-up loop is shown in Figure 15.8(b). Values of the multiplicative constants are omitted for brevity: there is one constant for Fid=11, five distinct constants for Fid=8 and two distinct constants for Fid=5. It shows three passes of the bottom-up loop, with no submodules in pass 1. After pass 3 is completed, the AFList is empty (shaded grey).



15.6 Summary of key insights

This chapter demonstrated the transformation of a functional description of the top-level module into a VHDL-ready design, i.e., AlgFunctionality to VHDL-ready AlgDesign.

As was mentioned in Key 15.2, the AlgDesign object is almost cleanly split into the *design* (AFlist and topAF) and *datapath* (CIRClst and topCIRC), with only AlgDesignSignalPkg and the CONNlist as their overlap. The method ProcessDesign takes the initial design-ready AlgDesign, with the functional description of the algorithm stored as topAF and builds the entire *design* in a top-down loop. It then transforms the *design* into the *datapath* in a bottom-up loop. When ProcessDesign terminates, AlgDesign is VHDL-ready, with the *datapath* captured in CIRClst and topCIRC. One last step remains: writing the datapath (and all packages, testbenches and testvectors) using VHDL as the chosen method of design entry.

Tables 15.4 and 15.5 summarize all Keys from Chapter 12 seen so far. The Keys from the previous recap (Table 14.7 in Section 15.1) are listed only by their title in Table 15.4: the previously fully solved Keys 13.2-14.8 are marked with ✓✓, and the new solutions from the AlgDesign object and its methods with ✓✓. The last part of *the automated design generation* is writing the VHDL for the datapath. This part will be explained in the next section (Section 16), which is organized as follows: packages, (sub)modules, and testbenches.

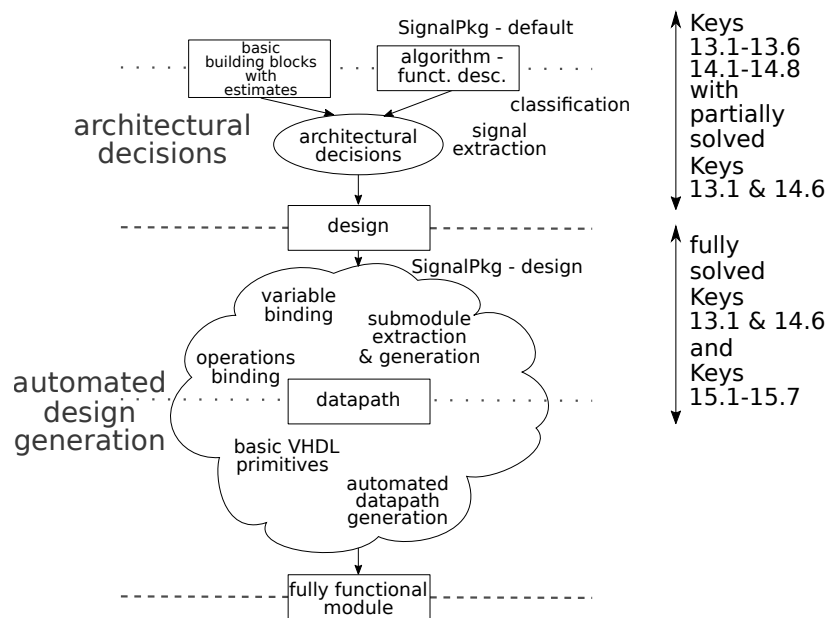


Figure 15.9: The automated design generation with key insights

Key 13.2	The SignalDomain: the bases and the field construction trail	✓✓	Section 13.2
Key 13.3	The SIGNAL: finite field as a vector space and “just vectors”	✓✓	Section 13.2
Key 13.5	The signal package SignalPkg - keeping it all in one place	✓✓	Section 13.3
Key 13.6	The VHDL type definitions and tower field constructions	✓✓	Section 13.3
Key 10.3	Binding of GAP variables and VHDL signal names	✓✓	Section 10.2
Key 14.1	Binding of VHDL ports to the finite fields and vector(s) (spaces)	✓✓	Section 14.1
Key 14.2	Binding of GAP variables and VHDL input ports	✓✓	Section 14.1
Key 14.3	Positional binding of expressions and output ports	✓✓	Section 14.1
Key 14.4	Implicit binding of expressions to the finite fields and vector(s) (speaces)	✓✓	Section 14.1
Key 13.1	How to obtain the (sub)modules - binding expressions to the finite fields	✓✓ Key 15.3	Section 13.1
Key 13.4	How to obtain the (sub)modules - correctness check based on the output port length	✓✓	Section 13.2
Key 10.4	Classification of expressions defined over finite fields	✓✓	Section 10.3
Key 14.5	Functional description aware of finite fields and vector(s) (speaces)	✓✓	Section 14.1
Key 14.6	The datapath and classification of expressions	✓✓ Key 15.2, 15.3, 15.4	Section 14.2
Key 14.7	The inference of the internal VHDL signals	✓✓	Section 14.2
Key 14.8	The extraction of submodules	✓✓	Section 14.2

- previously partially solved (✓) and solved (✓✓)

by GAPtoVHDL, SignalPkg and AlgFunctionality

- newly solved (✓✓) by AlgDesign

Table 15.4: Summary of key insights to the CIRCUIT package with new solutions from AlgDesign

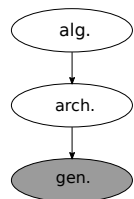
Key	The design	✓✓	Section
15.1	A hardware design itself consists of the top-level module and all its submodules. The top-level is modelled as an AlgFunctionality object that together with corresponding SignalPkg and submodule instructions forms the initial AlgDesign. All the required submodules will be generated as AlgFunctionality objects of their own, using the same SignalPkg, and added to the initial AlgDesign in a top-down fashion.		15.1
Key	The design and the datapath	✓✓	Section
15.2	The AlgDesign holds three lists, the SMList of submodules needed, the AFLists of AlgFunctionality objects generated for the submodules from the SMList, and the CIRCList of the VHDL-ready Circuit objects, corresponding to the generated AlgFunctionality objects. The structure of the SignalPkg object plays a crucial role for the automation: it allows for the <i>the top-down modular approach in which the submodules are generated for the subfields</i> . All three lists are accessible by the Fid's from the stored AlgDesignSignalPkg. There is an almost clean split of the AlgDesign object into <i>design</i> and <i>datapath</i> , with only the AlgDesignSignalPkg as their overlap: <ul style="list-style-type: none"> • <i>design</i>: AFList and topAF • <i>datapath</i>: CIRCList and topCIRC 		15.1
Key	The top-down processing: generate all submodules	✓✓	Section
15.3	The structure of SignalPkg, inherited by the SMList and AFList, allows a top-down processing of the initial AlgDesign in a ProcessSMAFloop. It transforms all the submodules in SMList into AlgFunctionality objects, and adds them to the AFList. Then it extracts new submodules from the just created AlgFunctionality objects and adds them to the SMList. All the populated Fids in the SMList are <i>element</i> type, and all the populated Fids in the AFList <i>vector</i> type SIGNALs. When ProcessSMAFloop is finished, the SMList is empty, and the AFList contains the final <i>design</i> .		15.1
Key	Bottom-up processing: submodules and additive constants	✓✓	Section
15.4	The bottom-up ProcessAFIRCLoop transforms each AlgFunctionality object in the AFList into a VHDL-ready Circuit object by adding the missing information about its submodules, which are now available, and about any additive constants present. The Circuit objects are added to the CIRCList at the same Fid as their corresponding AlgFunctionality objects.		15.1
Key	AlgFunctionality vs Circuit	✓✓	Section
15.5	The main difference between AlgFunctionality and Circuit objects is in the submodules: for AlgFunctionality, the submodules are black boxes (only their functionality is known), but for the Circuit objects, the actual submodules and their exact information (the parsed submodule list (15.1)) are available. The combination of top-down and bottom-up processing of submodules also implicitly binds the arithmetic operations and submodules using the FunctName attribute.		15.1
Key	Conversions between different data types	✓✓	Section
15.6	A pair of <i>connectors</i> is needed for conversions between different data types, when tower field constructions are used in the design: <i>element-vector connectors</i> (w.r.t. the same tower field “per-level” basis), <i>element-element connectors</i> and <i>vector-vector connectors</i> (both between the tower field “per-level” bases and the <i>TFB</i> bases of the isomorphic finite fields).		15.1
Key	Putting it all together - ProcessDesign	✓✓	Section
15.7	The method ProcessDesign takes an initial AlgDesign, with the functional description of the algorithm stored as topAF, and builds the entire <i>design</i> (AFList and topAF) in a top-down loop. It then transforms the <i>design</i> into the <i>datapath</i> (CIRCList and topCIRC) in a bottom-up loop. The remaining information, needed for the VHDL implementation, is captured as AlgDesignSignalPkg attribute and as component CONNList. The AlgDesign is now considered to be VHDL-ready, with the <i>datapath</i> captured in CIRCList and topCIRC.		15.1

Table 15.5: Summary of key insights to the CIRCUIT package with new solutions from AlgDesign - continued

Chapter 16

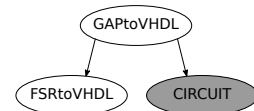
CIRCUIT package part 4: generating VHDL for the datapath

16.1	VHDL packages	226
16.2	VHDL (sub)modules	227
16.3	Testbench generation	233
16.4	Switching the field structure for profiling	233
16.5	Summary and conclusion	236



Overview of the CIRCUIT package part 4: – generating VHDL for the datapath –

The last step for a VHDL-ready AlgDesign is the VHDL code generation. The CIRCUIT package writes all VHDL packages, (sub)modules, and testbenches for the design. AlgDesign stores a SignalPkg, which holds all SIGNALs used in the design; the SIGNALs are directly translated VHDL



data types and written as VHDL package `field_pkg.vhd`. In the presence of a tower field, connectors are needed: in VHDL, the connectors are implemented as type conversion functions and written in a separate package `connectors_pkg.vhd`. The CIRCUIT package reuses the GAPtoVHDL package functions to write the field package and offers functions specialized for the connectors package. VHDL modules are generated for all Circuit objects in the design and each individual Circuit object holds all information needed for their VHDL generation. The datapath for each ANF expression is generated following the identified natural order, from exponentiations to additions. CIRCUIT contains a set of functions which complement the GAPtoVHDL functions. For each generated module a testbench is generated as well. The last feature of the CIRCUIT package is its ability to switch between a default and a design SignalPkg. This feature allows rapid design space exploration: the initial user setup is needed just once, afterwards, the hardware for different field constructions is generated automatically by switching the design SignalPkg objects.

16.1 VHDL packages

There are two packages that need to be written:

- the `field_pkg`, which is the `SignalPkg` translated to VHDL and is always written
- the `connectors_pkg` for the connectors (if they exist) from Table 15.2

The `SignalPkg`, stored as `AlgDesignSignalPkg`, is fed to the `Write_field_pkg` function from the `GAPtoVHDL` package (see Table 10.1). The VHDL data type definitions (13.8-13.14) are used. The type names contain the field identifier `Fid` from the `SignalPkg`, e.g., `ffe_11` for the `Fid=11 SIGNAL` in `defaultPkg`, as shown in Examples 13.3.1 and 13.3.2. For all the array types (recall type declaration (13.14) in Section 13.3), the "xor" functions (for the finite field addition) are added to the package. A full example of a package can be found in appendix Example D.5.1. For each type definition, the zero element `ffe_#_zero` is added as a VHDL constant. For the *vector* types, this is the constant with all coordinates set to zero. The constants are used as default values in case registered inputs and/or outputs with a reset are required by the archetype.

The `connectors_pkg` is a package of type conversion functions. The functions have the same name as their corresponding connectors in Table 15.2, and for every connector found (see Example 15.4.1), a function is needed. All other modules, including testbenches, need a use clause for the packages.

Example 16.1.1 *The connectors_pkg.vhd for the type conversion functions* \longleftrightarrow

This example a fraction of the `connectors_pkg.vhd` for the type conversion functions for the connectors listed in Example 15.4.1.

VHDL Example 16.1.1

```
package connectors_pkg is

    function fvec_to_tfvec ( st: ffe_3 ) return ffe_6;
    function fvec_to_tfvec ( st: ffe_4 ) return ffe_9;
    function tfvec_to_tffe ( st: ffe_6 ) return ffe_8;
    function tfvec_to_fvec ( st: ffe_6 ) return ffe_3;
    function ffe_to_tffe   ( st: ffe_7 ) return ffe_8;
    function tffe_to_tfvec ( st: ffe_8 ) return ffe_6;
    function tffe_to_ffe   ( st: ffe_8 ) return ffe_7;
    function tfvec_to_tffe ( st: ffe_9 ) return ffe_11;
    function tfvec_to_fvec ( st: ffe_9 ) return ffe_4;
    function ffe_to_tffe   ( st: ffe_10 ) return ffe_11;
    function tffe_to_tfvec ( st: ffe_11 ) return ffe_9;
    function tffe_to_ffe   ( st: ffe_11 ) return ffe_10;

end package;
```

\longleftarrow

16.2 VHDL (sub)modules

Each Circuit object in the *datapath*, i.e., in `CIRClst` and `topCIRC`, becomes a VHDL module. A set of GAP functions writes the following files: the module itself, the testbench file, the testvectors, and simulation scripts. The Circuit attributes and components contain all information needed: `EntityName` is used for the entity name of the module, the strings in `InputPorts` and `OutputPorts` are used as port names, and the `Fid`'s for their data types. The control list in `archtype` is used to add ports for the control signals (limited to `std_logic` only). The architecture name is given by `ArchName` and all submodules needed are available from the parsed submodule list. The architecture body is written following the steps in Figure 16.1.

<i>modules</i> functions and methods	
<code>Write_circ_arch_declarations</code> ^[1]	<code>ExpressionBinding</code> ^{[3]†}
<code>Write_circ_input_connectors</code> ^{[2]†}	<code>Write_circ_input_registers</code> ^{[2]†}
<code>Write_circ_arch_exponents</code> ^[3,4]	<code>Write_circ_arch_andtree</code> ^[3,5]
<code>Write_circ_arch_all_multree</code> ^{[3]†}	<code>Write_circ_arch_multree</code> ^[3,5]
<code>Write_circ_arch_MVs</code> ^[3,5]	<code>Add_constant_digraph</code> ^[3]
<code>Write_circ_arch_sumtree</code> ^[3]	
<code>Write_circ_ach_output</code> ^[6]	

Notes:

- [1] - entire module: all signals, constants, connectors and registers
- [2] - for all input ports
- [3] - for each expression, i.e., each (coordinate of) each output
- [4] - for each exponent in expression
- [5] - for each monomial in expression
- † - see Figure 16.1

Table 16.1: Main functionality of the CIRCUIT package - writing the VHDL (sub)modules

The function `Write_circ_arch_declarations` writes the declarations for all signals found in `signalcollected` (Table 14.1) except for the `prefix_lvl_#_node#` (Table 14.5), which are replaced by components or gates, and for all constants from `ConstantsList` (recall Example 15.3.2, showing the declarations needed for a tower field constant in VHDL Example 15.3.2). If connectors are found in `CONNlist` for the current `Fid`, additional signals are declared by simply prepending `con_` to the port names. Similarly, if `archtype` specified `RI*0`, the signals for the input registers are declared as input port names with the prefix `reg_`, e.g., `reg_i_a`.

The rest of the architecture is written following the steps in Figure 16.1:

- for each input port in `InputPorts`:
 - if needed, write input connectors and input registers (step 0)
- for each ANF expression in `ArchExpr` perform steps 1-6

- for each output port in OutputPorts perform step 7:
 - if needed, write output connectors and output registers
 - drive the outputs

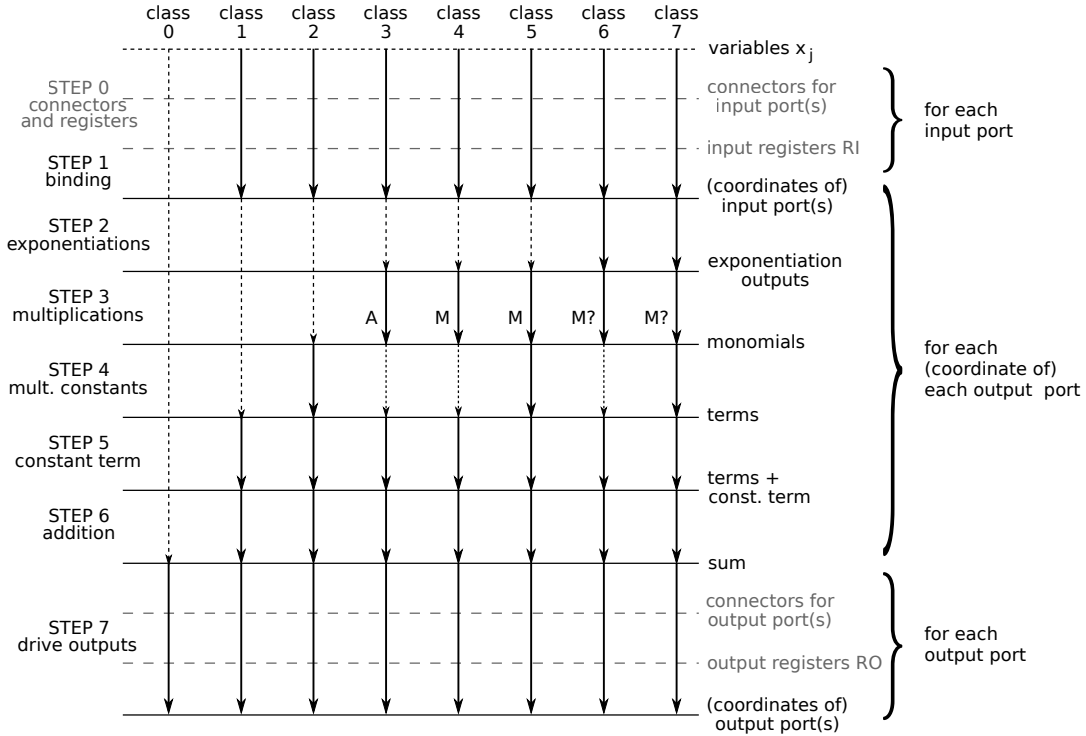


Figure 16.1: The (steps of writing a) datapath including the connectors and the registers

Step 0 and step 1 are separated in Figure 16.1, because connectors and registers are written for the input ports, but binding is performed for each ANF expression separately. Steps 1-6 are always performed for each (coordinate of) each output, i.e., one ANF expression at a time. Step 7 is performed per output, which may contain more than one expression. In the case of a *vector* output SIGNAL of length m , the individual sums (step 6) are collected in a list called *outsources*, and when all m sums are available, step 7 is performed.

Step 0, shown on top of the datapath in Figure 16.1, consists of `Write_circ_input_connectors` function, which writes the possible VHDL data type conversions (if any), followed by the function `Write_circ_input_registers` (if registered inputs RI). The signal names with the prefixes `con_` and `reg_` are used. Example 16.2.1 shows the output VHDL for an `RI*0` type top-level module, implementing the running example (13.1), defined over the tower field $\mathbb{F}_{((2^2)^2)^2}$ (Example 13.1.3 in Section 13.1).

Example 16.2.1 *Example of connectors and input registers in the architecture* \longleftrightarrow

The connectors are needed for the modules, which have e.g., *element* ports and *vector* submodules, like the running example (13.1) defined over $\mathbb{F}_{(2^2)^2}$: the ports have `Fid=11`, but all internal signals are `Fid=9`. This way, the connectors are needed only for the port signals and not for each individual submodule. The connectors are also used for the testbenches: the testvectors are written for `Fid=10` using the *TFB*, and a `ffe_to_tffe` function is needed for the top-level module with `Fid=11` ports. VHDL Example 16.2.1 is showing the input port (entity), declaration of the connector signal and one register signal, followed by the use of the type conversion functions (see VHDL Example 16.1.1) and input registers (architecture). The input registers have been specified with the following control list in GAP: `[["i_clk", "i_rst", "i_ce"], [1,1,1]]`, dictating a rising edge clock signal `i_clk`, a reset `i_rst` and a chip enable `i_ce`, both active high. Note the resetting to `ffe_9_zero` (declared in `field_pkg`). Since the control list holds for all input ports they are written in the same clocked process. The dots indicate VHDL code removed for brevity.

VHDL Example 16.2.1

```
port( i_a: in ffe_11; -- input
      ... OMITTED FOR BREVITY ....
);
end entity;

architecture main of random is
  ... OMITTED FOR BREVITY ....
  signal con_i_a : ffe_9; -- input connector
  signal reg_i_a : ffe_9; -- input reg
  ... OMITTED FOR BREVITY ....
begin
  con_i_a <= tffe_to_tfvec(i_a);
  con_i_b <= tffe_to_tfvec(i_b);
  con_i_c <= tffe_to_tfvec(i_c);

  --input regz
  inputregs : process(i_clk) begin
    if rising_edge(i_clk) then
      if i_rst = '1' then
        reg_i_a <= ffe_9_zero;
        reg_i_b <= ffe_9_zero;
        reg_i_c <= ffe_9_zero;
      else
        if i_ce = '1' then
          reg_i_a <= con_i_a;
          reg_i_b <= con_i_b;
          reg_i_c <= con_i_c;
        end if;
      end if;
    end if;
  end process;
  ... OMITTED FOR BREVITY ....
```

\longleftarrow

After step 0, for each output port j in `OutputPorts` an empty list `outsources` for the final sum(s) is created. In the case of a *vector* output, steps 1-6 are performed for each coordinate `ArchExpr[j][i]`, and in the case of an *element* output for the expression in `ArchExpr[j]`. After an expression is obtained from `ArchExpr`, it is split into a list of monomials (without coefficients). Each monomial is further split into a sublist containing all variables present in this monomial. The ordering of monomials and variables is preserved from the ANF expression (see Section 10.3 for details). The constant term is removed. This list is called `exprlist` and will be modified by steps 1-6 in Figure 16.1. Depending on the class of the current expression, some steps take no action and the `exprlist` is passed on unmodified. For example, there are no multiplicative constants (step 4) present in a class 6 expression.

When submodules are needed, the function `Write_component_inst` is called (see Table 10.3 in Section 10). All component instantiations need the following information: a label, entity and architecture name, and a port list. Labels are created on the fly, the entity and architecture names are chosen from the parsed submodule list, and the port map will be explained for each step separately.

1. *variable binding*: all GAP variables in `exprlist` are replaced as explained in Section 14.2 (Key 14.2). As the binding is done per expression, the function is called *ExpressionBinding*. First, the GAP variables are replaced with VHDL input port names (including the coordinates in the case of partial binding). If only input connectors are present, the prefix `con_` is added, if inputs are registered (RI), the prefix `reg_` is added.
2. *exponentiation*: using the Table 14.5 encoding `[EXPname, e, t, j]` the correct submodule for exponentiation to the power e is chosen from the parsed submodules. Value of t identifies the sublist in `exprlist` and the value j the variable in the sublist. The port map input is the `exprlist[t][j]` string and the output the `EXPname`. The `EXPname` also replaces the `exprlist[t][j]` entry. `exprlist` strings with no exponentiations remain unchanged. The exponentiations can be done in parallel, i.e., the component instantiations for the submodules are concurrent.
3. *multiplication*: the `exprlist` is passed on to the `Write_circ_arch_all_multree` function, which decides if a multiplier submodule or an AND gate is needed. For each sublist in `exprlist`, it then calls either `Write_circ_arch_multree` or `Write_circ_arch_andtree`. For an individual multiplication tree, the inputs are parsed from the corresponding sublist in `exprlist` for the top level. All other signals used are `prefix_lvl_#_edge#` signals in `signalcollected` (Table 14.5), which already have the tree structure. In the case of multiplier submodules, these signals are used for the port maps. The node signals are replaced by submodule instantiations or AND gates. After all multiplications are written, the sublists (monomials) in the `exprlist` are replaced by the signals for the final products.
4. *multiplication with constants*: this step is up to the encoding (Table 14.5) and submodules identical to the *exponentiations*. When `Write_circ_arch_MVs` is finished, the `exprlist` entries contain the signals that represent the whole terms.
5. *additive constant*: if the expression has a non-zero constant term, the method `Add_constant_digraph` finds the corresponding digraph from `ConstantsList` and adds the constant name from the root node of the digraph to `exprlist`. This is the final missing term.

6. *addition of the terms*: the function `Write_circ_arch_sumtree` is very similar to `Write_circ_arch_andtree`, but replaces the nodes with XOR gates. After this step, the `exprlist` contains a single entry, namely the final sum, which is copied to (the appropriate coordinate of the) `outsources` list.

The last step is performed once per output, i.e., each output has a new `outsources` list. The reason for this are the output connectors: when the output connectors are needed, the type conversion function is applied to the entire output, not individual coordinates. The signal for the connector or register has the same name as the output port, with the prefix `con_` or `reg_`. If the output is registered (RO), the register is written next and finally assigned to the output port. For registered *vector* outputs, each register coordinate is driven separately, i.e., `reg_o_z(0) <= ...` etc. For combinational *vector* outputs, each output coordinate is driven separately, i.e., `o_z(0) <= ...` etc. The r.h.s. signals for these assignments are the signals in the `outsources` list.

Example 16.2.2 *Example of architecture body for an expression defined over $\mathbb{F}_{((2^2)^2)^2}$* \longleftrightarrow

The VHDL code for the architecture body of the top-level module implementing the running example (13.1) defined over $\mathbb{F}_{((2^2)^2)^2}$ is shown in VHDL Example 16.2.2. The archtype used is CICO, hence no registers are written, but connectors are needed (registered inputs were shown in VHDL Example 16.2.1). Line 2. in VHDL Example 16.2.2 shows the type conversions (recall Example 15.4.1 and VHDL Example 16.1.1). Next line shows the component instantiation for the squarer in $\mathbb{F}_{((2^2)^2)^2}$, with entity name `Fid9_exp_2`. Lines 4-7 compute the monomial $a \cdot b \cdot c$, and line 8. completes the first term with the matrix vector multiplier. Lines 9-12 show the additions. Note line 9, which illustrates how `exprlist` was changing: the first term was modified by steps 1, 3, and 4, the second by steps 1 and 2, and the last term (a constant) by step 5 only. Line 13 shows the inverse output connector and line 14 drives the output.

\longleftrightarrow

Example 16.2.3 *Example of registered vector output* \longleftrightarrow

The VHDL code VHDL Example 16.2.3 shows example of a registered outputs RO submodule, in particular for the the \mathbb{F}_{2^2} multiplier with `FuncName MforFid6` and `EntityName Fid2_mult`. The submodule instruction control list `[["i_clk", "i_rst", "i_ceout"], [1,1,1]]` dictates a rising edge clock signal `i_clk`, a reset `i_rst` and a chip enable `i_ceout`, both active high. Note the resetting to `ffe_1_zero` (declared in `field_pkg`). On the chip enable, the output register clocks the outputs of the sums.

By the structure of the `SignalPkg` object, the submodules are written for *vector* SIGNALs, and the output has two coordinates. VHDL Example 16.2.3 shows the registers for two coordinates of the aforementioned *vector* SIGNAL. The output port however, requires only a single assignment.

\longleftrightarrow

VHDL Example 16.2.2

```

1. begin
2.   con_i_a <= tffe_to_tfvec(i_a); con_i_b <= tffe_to_tfvec(i_b); con_i_c <= tffe_to_tfvec(i_c);
3.   EXP0_0_0 : entity work.Fid9_exp_2(main) port map ( con_i_a, EXPout_0_0_0 );
4.   out_0_0_mon0_lv13_edge0 <= con_i_a; out_0_0_mon0_lv13_edge1 <= con_i_b; out_0_0_mon0_lv13_edge2 <= con_i_c;
5.   mult_0_0_0_0 : entity work.Fid9_mult(main) port map
      ( out_0_0_mon0_lv13_edge0, out_0_0_mon0_lv13_edge1, out_0_0_mon0_lv12_edge0 );
6.   out_0_0_mon0_lv12_edge1 <= out_0_0_mon0_lv13_edge2;
7.   mult_0_0_0_1 : entity work.Fid9_mult(main) port map
      ( out_0_0_mon0_lv12_edge0, out_0_0_mon0_lv12_edge1, out_0_0_mon0_lv11_edge0 );
8.   MV0_0_0 : entity work.Fid9_mv_g_73(main) port map ( out_0_0_mon0_lv11_edge0, MVout_0_0_0 );
9.   out_0_0_term_lv13_edge0 <= MVout_0_0_0; out_0_0_term_lv13_edge1 <= EXPout_0_0_0; out_0_0_term_lv13_edge2 <=
      Fid9_c_g_3;
10.  out_0_0_term_lv12_edge0 <= out_0_0_term_lv13_edge0 xor out_0_0_term_lv13_edge1;
11.  out_0_0_term_lv12_edge1 <= out_0_0_term_lv13_edge2;
12.  out_0_0_term_lv11_edge0 <= out_0_0_term_lv12_edge0 xor out_0_0_term_lv12_edge1;
13.  con_o_z <= tfvec_to_tffe (out_0_0_term_lv11_edge0);
14.  o_z <= con_o_z;
15. end;
```

VHDL Example 16.2.3

```

-- output reg
outputregs_1 : process(i_clk) begin
  if rising_edge(i_clk) then
    if i_rst = '1' then
      reg_o_z(0) <= ffe_1_zero;
      reg_o_z(1) <= ffe_1_zero;
    else
      if i_ceout = '1' then
        reg_o_z(0) <= out_0_0_term_lv11_edge0;
        reg_o_z(1) <= out_0_1_term_lv11_edge0;
      end if;
    end if;
  end if;
end process;
-- output
o_z <= reg_o_z;
```

16.3 Testbench generation

A testbench is generated for each submodule for its functionality, not for the generated submodule expressions, e.g., for multiplication, not for expressions generated by the FFCSA to implement the multiplier. This ensures a check on the submodule generation method. For example, for the `MforFid6` multiplier from Example 16.2.3, the testbench is generated as ab and then both inputs and the result are converted into the basis used by the `Fid=2` SIGNAL. The module implementing the `MforFid6` multiplier implements two expressions¹ for the two output coordinates: $a_0b_0 + a_1b_1$ and $a_0b_1 + a_1b_0 + a_1b_1$. These expressions were generated for the basis used by the `Fid=2`, but were not used to generate the testvectors. Generating the testvectors from the high level expression, such as ab provides additional confidence in the functional correctness of the automated design flow and the on the fly generated submodule expressions.

For the running example (13.1) defined over the towerfield $\mathbb{F}_{((2^2)^2)^2}$, the testbenches need connectors as described in Example 15.4.1.

16.4 Switching the field structure for profiling

This section explains the last part of the CIRCUIIT package, which facilitates the design space exploration. GAP was chosen for the implementation of the framework because of its symbolic computation, which provides an easy way to generate the submodule expressions (see Example 13.4(c) in Subsection 13.2.2). To generate the expressions, details about the field construction and about representation of the field elements used must be known. For the designer, it is very interesting to see how this information affects the synthesis results. Assuming some information is fixed to reduce the search space, e.g., the field size or the number of extensions and their degrees, the structure of the `SignalPkg` will not change if an individual parameter changes. Recall the tower field example $\mathbb{F}_{((2^2)^2)^2}$:

$$\mathbb{F}_2 \xrightarrow{f_1(x)} \mathbb{F}_{2^2} \xrightarrow{f_2(x)} \mathbb{F}_{(2^2)^2} \xrightarrow{f_3(x)} \mathbb{F}_{((2^2)^2)^2}.$$

The construction of $\mathbb{F}_{((2^2)^2)^2}$ was listed in Table 7.6: there is only one irreducible polynomial f_1 , but there are 6 irreducible polynomials $f_{2,i}$ and 120 irreducible $f_{3,j}$, yielding 720 different tower field constructions for $\mathbb{F}_{((2^2)^2)^2}$. The 720 constructions were stored in a file with a list of the extension defining polynomials (EDPs) $[f_1, f_{2,i}, f_{3,j}]$ in each line.

Example 7.3.1 in Section 7.3.1 followed by Examples 13.3.1 and 13.3.2 in Section 13.3 demonstrated the FFCSA flow for tower field bases. The method `FindEDPLAllfromEDL [2,2,2]` was used to find the extension defining polynomials for the tower field $\mathbb{F}_{((2^2)^2)^2}$, resulting in 720 combinations, and, subsequently, 720 candidates for profiling. A particular candidate given by a triplet of EDPs $[f_1, f_{2,i}, f_{3,j}]$. For DSE, each candidate EDP triplet is stored into a `ply` file. Each iteration retrieves a new candidate from the file, and for a given candidate $[f_1, f_{2,i}, f_{3,j}]$: construct all three

¹ generated on the fly

“per-level” bases to obtain the list $Blist := [B_1, B_2, B_3]$, then create an empty design `SignalPkg` called `designPkg`, and add the tower field construction using `AddTowerFieldToSignalPkg (designPkg, Blist, dirlist)`.

Acknowledgement goes to Bo Yang and Mark Aagaard for help with the following scripts. “uw tools” are developed and maintained by Mark Aagaard for teaching and research purposes.

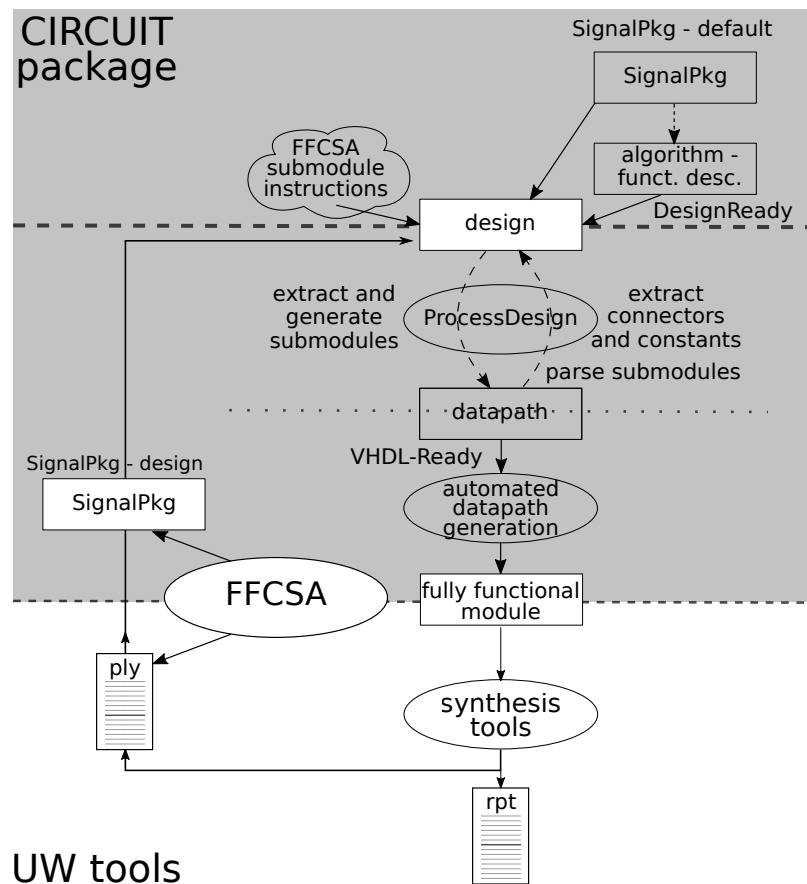


Figure 16.2: The automated design generation and the design space exploration: profiling loop invoking synthesis tools

Figure 16.2 shows the profiling loop. A set of scripts was used for the DSE: “uw tools”, a set of scripts that invoke commercial tools such as ModelSim and Design Compiler (shown in the oval shape), and the profiling loop script. The profiling loop calls GAP with a prepared setup file to build the initial `algfun` with `defaultPkg` and saves the current workspace. Henceforth, the profiling loop invokes GAP with the stored workspace and a new candidate triplet, and runs the datapath synthesis for the new parameter set. The results are stored for each candidate (`rpt` file), but the VHDL files are overwritten.

The DSE has two phases:

- the setup
 - prepare the `ply` file with candidates (using `FindEDPLAllfromEDL [2,2,2]`)
 - prepare (modify) GAP templates (e.g., construction of `defaultPkg` for Table 7.6 construction) for the an expression defined over $\mathbb{F}_{((2^2)^2)^2}$ (shown on top right in Figure 16.2: `SignalPkg - default` and `algorithm - functional description`)
- the profiling loop: a script, which reads the file of candidates (`ply` in Figure 16.2):
 - grab a new line (list of EDPs) and build the `designPkg SignalPkg` (`SignalPkg - design` on the loop in Figure 16.2):
 - create the `AlgDesign` with two `SignalPkg` inputs: `defaultPkg` and `designPkg` (Section 15.1)
 - run `WriteDesignVHDL` to write a new datapath (overwrite any existing VHDL files)
 - run synthesis tools and store the generated report with implementation results (`rpt` in Figure 16.2)

In each iteration, the `designPkg` will have the structure shown in Figure 13.5 in Example 13.3.2 in Section 13.3. i.e., the same structure as the `defaultPkg`. The design od `SignalPkg` object (Section 13.3) assures that same field structure results in the same structure of the `SignalPkg` objects, which makes them interchangeable: the tower field constructions used have the same number of *element* and *vector* SIGNALs ordered the same way. For example, an *element* of $\mathbb{F}_{((2^2)^2)^2}$ will always have `Fid=11` and its corresponding vector space will always be `Fid=9`, but the two SIGNALs will differ in at least one “per-level” basis PLB. The constructor will check if the structure of the `designPkg` is the same as the `defaultPkg`, except for the bases used, and then use the `designPkg` as `AlgDesignSignalPkg`. For each new `designPkg`, the `ProcessSMAFLoop` will generate different submodules, hence a different datapath (Section 15.2). This workflow is used for profiling and will be revisited in Part V.

16.5 Summary and conclusion

Chapters 13-16 presented the second automation mechanism. It completes the *architectural decisions – automated design generation flow*, which allows synthesis of arbitrary datapaths over arbitrary fields.

Chapter 13 introduced the necessary structures for the support of tower fields: the objects SignalDomain, SIGNAL, and SignalPkg encode the field structure in a way suitable for hardware implementations. This encoding allows to work with field elements, vectors in a vector space, or “just vectors”. Chapter 14 presented the AlgFunctionality object used for the functional description of the algorithm. It serves as the manual entry point for the top-level module. The datapath is given with one or more ANF expressions and SignalPkg is used to bind expressions to finite fields. Binding the AlgFunctionality to the SignalPkg brings the awareness of the underlying field structure to the synthesis.

Chapter 15 shows how two compilation algorithms gradually transform the initial design to a VHDL-ready datapath. They work in a top-down/bottom-up fashion, facilitated by the structure of the SignalPkg: the submodules are namely expected for the subfields. For the tower field support extra management is required. First are the additive constants, which are transformed into directed graphs for declarations in correct order. Second are the *connectors*, which are used for the type conversions between the tower field *element* and *vector* SIGNALs. Chapter 16 explains how the VHDL is generated for the VHDL-ready design; it covers the packages, (sub)module generation based on a natural order of performing operations, and testbench generation. Finally the facility that enables the design space exploration is presented: as long as the structure of the SignalPkg remains unchanged, the same setup templates can be used for various field parameters, including the field size.

Figure 16.3 shows the entire framework, placed into the hardware implementation design flow. The CIRCUIT package is fragmented, and all its components are enclosed in a shaded rectangle. All user input is shown in darker shapes. The FSR and FSRtoVHDL *architectural decisions – automated design generation flow* is included in the figure.

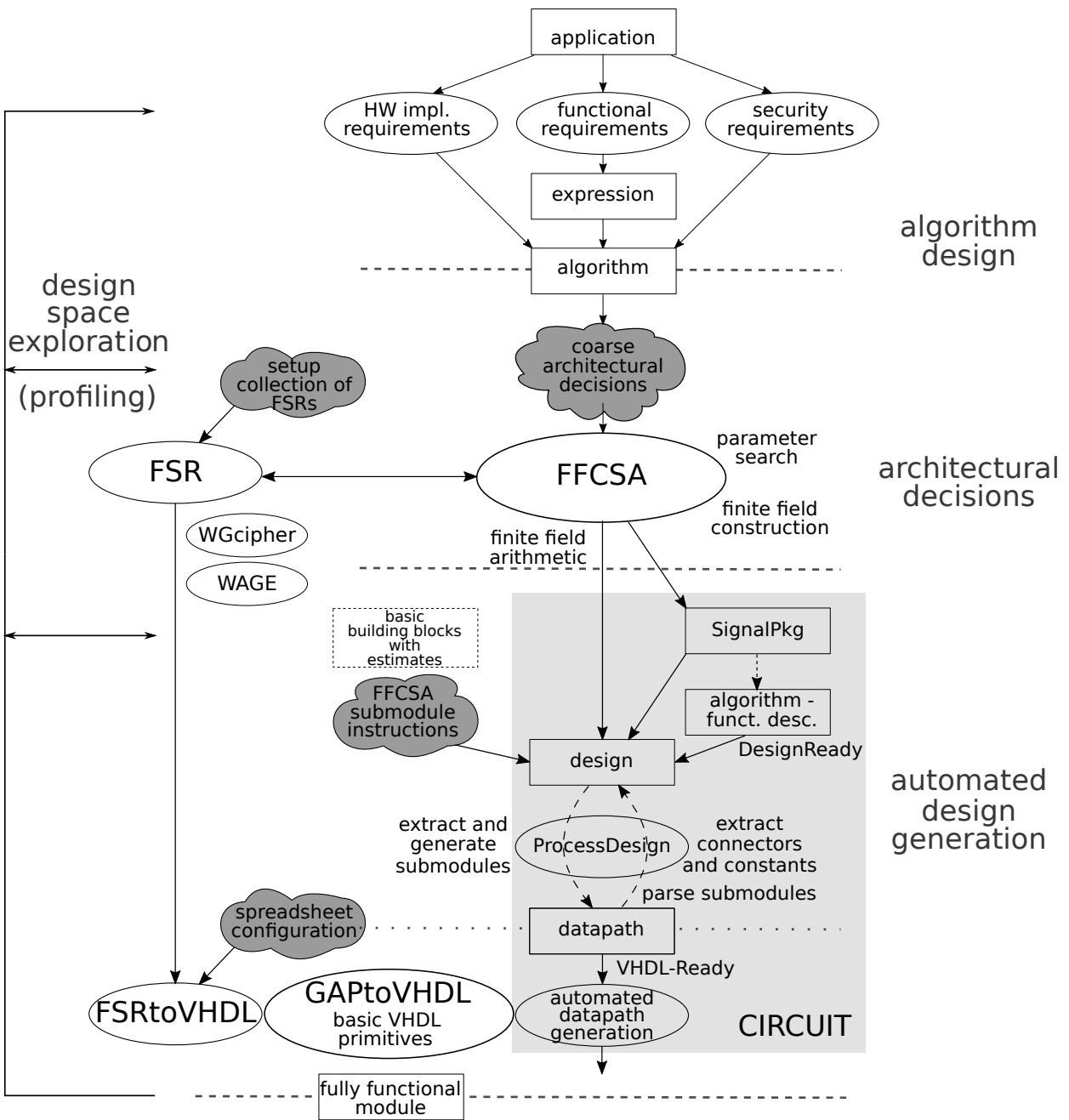


Figure 16.3: Design flow: datapath synthesis (automation framework)

Part V

Design space exploration

Part V - Outline

17 Design space exploration - overview	240
18 DSE for arithmetic modules - basic building blocks	245
19 Design space exploration of an arbitrary datapath	256
20 The WG cipher design space exploration	257

Chapter 17

Design space exploration - overview

17.1 DSE from a mathematical perspective	240
17.2 Different profiles	242

17.1 DSE from a mathematical perspective

This part of the thesis is concerned with design space exploration (DSE) from the finite field perspective. Finite field arithmetic design space is vast, it involves the choice of bases used for the representation of the field elements and algorithms for the basic building blocks, i.e., arithmetic modules.

Figure 17.1 shows the design with the automation framework and and DSE loop on the left. The FFCSA package (Chapter 7), implemented as a part of *the architectural decisions*, facilitates DSE with search methods for different parameters, such as field defining polynomials, bases, transition matrices, etc.

The search algorithms can be classified as exhaustive search (e.g., find all normal bases), reduced search space (e.g., ignore conjugates), and specialized search (e.g., find a primitive polynomial with a specified number of nonzero coefficients). Specialized search is a form of reduced search space, but the reduction criteria is different. The search algorithms produce a list of candidates, i.e., a design space.

There are two *architectural decisions* – *automated design generation flows* through the framework:

- the first flow focuses on synthesis for ciphers based on feedback shift registers. It consists of the GAP packages FSR and FSRtoVHDL (Chapters 6 and 11 respectively)
- the second flow allows for the synthesis of arbitrary expressions over arbitrary finite fields, realized by the GAP package CIRCUIT (Chapters 12-16)

Both FSRtoVHDL and CIRCUIT package rely on the package GAPtoVHDL (Chapter 10). The FFCSA package plays an important role for the second *architectural decisions* – *automated design generation flow*. Exploiting symbolic computation capabilities of GAP, the FFCSA methods can generate ANF expressions, which are used as datapath instructions for the CIRCUIT package. For example, the user specified multiplication simply as $a \cdot b$ and a submodule instruction, e.g., matrix

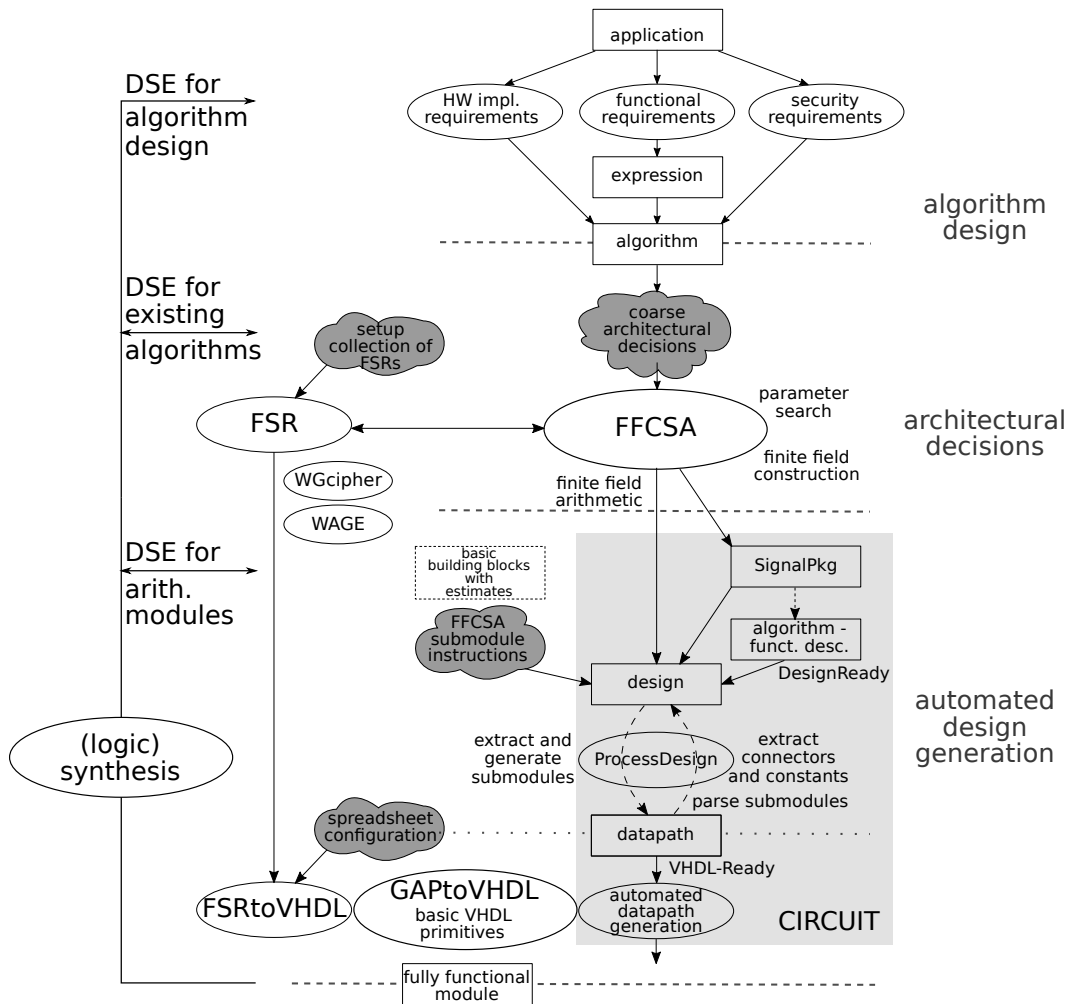


Figure 17.1: Design flow: detained design space exploration

U, and the CIRCUIT package will use the FFCSA method *FFCSA_mult_matrixU* to generate the expressions for the multiplier submodule. There is a variety of well studied algorithms for implementing finite field arithmetic, that can be added to the FFCSA package.

The lists of candidates and the FFA algorithms produced during *the architectural decisions* are the starting point for *the automated design generation*: the FSRtoVHDL and CIRCUIT packages automatically produce synthesizable VHDL code for the datapath, yielding fully functional hardware modules. A new module is generated for each candidate, then passed on to synthesis tools (shown on the DSE loop in the magnifying glass in Figure 17.1), e.g., Design Compiler. A discussion on the metrics used and the representation of the results follows in Section 17.2.

In the magnifying glass on the right side of Figure 17.1 is a simplified design flow diagram, showing the extended design space exploration loop on the left. The loop is explained from bottom-up:

1. **DSE for arithmetic modules:** as the finite field arithmetic modules are the building blocks

for all other datapaths (algorithms), this represents the lowest level DSE, the findings of which can be used for decisions in the earlier stages of the design flow, e.g., to further narrow down the search space.

2. **DSE for existing algorithms:** the aim of this DSE is to find the best field parameters (e.g., bases and finite field arithmetic modules) for the implementation of the already specified algorithm. The term “best” depends on the application, e.g., this work is focusing on lightweight cryptography, and therefore aims for a small area. One example is the WG cipher, which is a family and can be parametrized to find new instances. An example from literature is the AES block cipher (see the review in Subsection 4.1.4). There are numerous implementations of the AES S-box that use one representation for the actual implementation and then encapsulate the S-box with basis transition matrices (to ensure that the functionality is compliant with the specifications of AES).
3. **DSE for algorithm design:** the hardware considerations directly affect the design of the algorithm. One such example is the authenticated encryption scheme WAGE.

17.2 Different profiles

Design space exploration relies on selected metric(s) to choose good design options from the set under exploration. The most commonly used metrics for hardware implementations are combinational delay, clock period or frequency, area, throughput and other derived optimality metrics (Section 3.3). The throughput in terms of bits per cycle, for example, will differ for serial and parallel modules, such as multipliers. This work focuses on parallel input - parallel output (PIPO) modules (Subsection 4.2.2), hence the throughput is not of interest.

The delay and the area can be estimated theoretically or using synthesis tools to obtain technology-dependent values. The first option will be called offline profiling and the second online profiling. This distinction is similar to that between technology-independent synthesis and technology-dependent synthesis described in Section 4.3. Different terminology is used to emphasize that the proposed automation process does not plug-into the synthesis tools, but only interacts with them and uses their results.

Offline profiling

A very common estimate is the Hamming weight of the field defining polynomial, as it affects the algorithms for arithmetic operations. Past experience has shown that for small finite fields, this not a good estimate of area. The FFCSA package can produce basic building blocks, i.e., finite field arithmetic modules, such as transition or multiplication matrices. In case of a simple block, the maximum row Hamming weight is used as the estimate for the combinational delay. Similarly, the total Hamming weight can be used as the area estimate. An example of such a basic building

block is a matrix for multiplication by a constant $\gamma \in \mathbb{F}_{2^m}$. In this simple case, the Hamming weight is computed as a number of XOR gates. However, the delay could be expressed as the *depth* of the binary tree with *HW* leafs, where *HW* stands for maximum row Hamming weight, i.e., of the critical path through the building block. Furthermore, possible subexpression eliminations can reduce the actual estimated area. Much research has recently been dedicated to XOR counts, as was summarized in Subsection 4.2.3.

More complicated examples, e.g., an \mathbb{F}_{2^m} multiplier, will have AND gates in addition to XOR gates, and some algorithms can even require NOT gates. In case the of tower field constructions, the area and delay of the subfield modules must be taken into account, but is finally also expressed in terms of AND and XOR gates. To obtain a unified metric, all gates can be expressed in terms of NAND gates. This is the “gate equivalents” (GE) area metric for ASIC technologies.

As a final note, it is important to emphasize the differences between ASIC and FPGAs (Subsection 3.3.1). For the FPGAs, which can implement an arbitrary n -input Boolean function in an n -input/1-output LUT, the Hamming weights are irrelevant. Important information is the number of variables for the Boolean function realizing a single bit of output and the width of the output in bits¹.

The FFCSA package includes offline profiling methods (Table 7.4), explained in more detail in Section 7.2. The offline profiling methods facilitate the search for special elements and polynomials, and the *offline profile* is always a (set of) Hamming weight(s).

Online profiling

The aforementioned metrics can also be obtained from synthesis tools such as Design Compiler. This approach is computationally much more demanding and takes much longer, but it provides more accurate results: synthesis tools can do optimizations (e.g. common subexpression elimination), they can select different gates (e.g., using 3-input XOR gates to replace some 2-input XORs, or use special OR-AND-Invert gates), or they can optimize for different performance goals, e.g., high speed or small area. For example, (technology-aware) synthesis process will select larger gates if a particularly small target clock period is given as a constraint.

In general, online profiling provides more accurate values. Furthermore, online profiling provides different options, e.g., using the results after logic synthesis or physical synthesis, i.e., pre- and post- place and route (PAR) results. Only pre-PAR results are reported throughout this work. Switching to (more accurate) post-PAR results is trivial, but it increases the total time for profiling, as synthesis tools have more work for each instance.

This thesis explored the following two online profiles for arithmetic circuits:

- *min. lib.*: minimal library with only NOT and two-input AND, XOR, OR gates
 - *min. lib. before*: initial gate counts extracted from the original VHDL code (similar to d-XOR-counts in Subsection 4.2.3)

¹ because each output bit requires a LUT of its own

- *min. lib. after*: gate counts after synthesis tools performed optimizations, e.g., subexpression eliminations (similar to s-XOR-counts in Subsection 4.2.3)
- *full lib.*: full library, no restrictions on gates, selection of gates left to synthesis tools

The *min. lib.* profile is using a library containing a minimal set of gates for which the synthesis works: removing any of the gates triggers a library error. Especially with the presence of the OR gate, the synthesis using the *min. lib.* profile is not perfect for finite field arithmetic. However, the synthesis tools only infer a combination of NOT and OR gates to replace some of the AND gates, since $\overline{a \wedge b} = \overline{a} \vee \overline{b}$. The purpose of using the *min. lib.* profile is to get a sense of optimizations performed by the synthesis tools while not using more “exotic” gates, e.g., 3-input XOR gates, XNOR gates, etc..

In this thesis, the reports using gate counts will contain a *before* and *after* column, where *before* relates to the initial gate counts extracted from the original VHDL code, and *after* to the gate counts after synthesis tools performed their optimizations. The minimal library *before* and *after* profiles are similar to the d-XOR- and s-XOR- counts, reported by the literature (Subsection 4.2.3), where d-XOR-count stands for direct XOR count, and s-XOR-count for sequential XOR count. The latter is a theoretical estimate for optimizations, that are possible due to sequential computations or subexpression eliminations.

Since finite field addition and multiplication in the prime field \mathbb{F}_2 use the XOR and AND gates, including NOT and OR gates is a disadvantage, because it is hard to translate them back to the expressions. However, the advantages of this approach are: (i.) optimizations are left to the synthesis tools, and (ii.) the process is not limited to circuits with only XOR gates. The latter is particularly interesting for the nonlinear components in cryptographic hardware. The current research on gate counts in the literature is focusing on XOR gates only (Subsection 4.2.3).

The results of this library are reported for the example of normal basis multipliers in Section 18.1: the number of OR gates used is very small and amounts to 0.5% - 3.8% of all gates in the design.

Unless stated otherwise, the full library profile will be used. When the minimal library profile is used, it is assumed to be *min, lib. after* profile, unless stated otherwise. In many cases, only the area results are reported, because critical path delay is of lesser importance for lightweight applications.

Chapter 18

DSE for arithmetic modules - basic building blocks

18.1 Normal basis multiplication for \mathbb{F}_{2^7}	245
18.2 Polynomial basis multiplication and inversion in \mathbb{F}_{2^7} using primitive defining polynomials	248
18.3 Polynomial basis multiplication and inversion for $\mathbb{F}_{2^{14}}$ using primitive defining polynomials	254

This chapter shows several examples of profiling for basic building blocks, i.e., arithmetic modules. Each example is presented as a section on its own and is linked to GAP methods that are used in the examples.

18.1 Normal basis multiplication for \mathbb{F}_{2^7}

This example shows DSE for the normal basis multipliers for \mathbb{F}_{2^7} . It is an online profiling equivalent of the FFCSA offline profiling method `FindSmallestTNBGenerator` (Table 7.4). The method was explained in more detail in Section 7.2. Both the offline and the online approach are the solution to Example 7.1.1 in Section 7.1.

For \mathbb{F}_{2^7} there is no optimal normal basis. The defining polynomial for the reference field \mathbb{F}_{2^7} is $f_{ref}(x) = x^7 + x + 1$, with root ω . All normal elements are given w.r.t. root ω , which is also a generator of F_{2^7} . The synthesis results are represented graphically, ordered by their normal element. The normal elements, their corresponding N-polynomials, and the complexities C_T are shown in Table 18.1, ordered by the N-polynomial. The two profiles used are the minimal set of gates¹ (min.lib.) and the full CMOS 65nm library (full lib.).

■ **Implementation detail:** The `FindNormalFFEsIgnoreConjugates` method

Normal elements were found using the method `FindNormalFFEsIgnoreConjugates (F)`, listed in Table 7.4 in Section 7.1. The `IgnoreConjugates` is used to reduce the search space. The `IgnoreConjugates` implementation is checking only elements of form ω^{c_j} , where c_j is a cyclotomic coset leader (obtained using method `CCLeaders` (Table 7.2)):

$$c_j \in [0, 1, 3, 5, 7, 9, 11, 13, 15, 19, 21, 23, 27, 29, 31, 43, 47, 55, 63]$$

¹ selected from the full library CMOS 65nm

Elements $\alpha = \omega^{c_j}$ are checked using method `IsNormalFFE` (F, α), which is implemented following Theorem 3.2 in background Section 3.1.1. It generates the polynomial

$$T_\alpha(x) = \sum_{i=0}^{m-1} \sigma^i(\alpha)x^i \in \mathbb{F}_{2^m}[x]$$

and checks $\gcd(T_\alpha, x^m - 1)$. Element α is a normal element iff the two polynomials are coprime. ■

■ **Implementation detail: The ComplexityOfT method**

The complexity of normal basis, obtained as the number of non-zero entries in the multiplication table T, is one of the oldest offline metrics used [186]. The coefficients of the multiplication table T were listed in equation (3.5). The complexity of T C_T is defined as the number of non-zero entries in T and obtained with method `WeightMatrix` (Table 7.2)). Note that for the implementation $C_T - m$ XOR gates are needed: one XOR is removed for each row of T. ■

This example is listing the offline and online profiling results for the multiplication table T, matrix U multiplier and rr_mo multiplier. The matrix U multiplication was explained in Section 7.4. The rr_mo multiplier is the reduced redundancy multiplier [156]. The rr_mo multiplier is used to show the benefits of using highly optimized algorithms for finite field arithmetic. The graph in Figure 18.1 shows a much smaller area for all rr_mo multipliers w.r.t. the matrix U multipliers.

The graph in Figure 18.1 shows that the matrix U results and the rr_mo results approximately follow the same pattern. For example, the smallest area multipliers are obtained for the normal element ω^{13} , which also has the smallest offline profile, i.e., the complexity C_T . The only exceptions are the last two normal elements: ω^{19} rr_mo area is bigger than ω^{27} rr_mo area, while the ω^{19} matrix U area is smaller than ω^{27} matrix U area.

For both multipliers, the graph in Figure 18.1 nicely shows that the full library area is smaller than the minimal library area. Observing the minimal library profile gate counts *before* and *after* optimizations, listed in Table 18.2, shows the following:

- for the multiplication tables T, the before gate counts always match the theoretical (offline) value $C_T - m$, where $m = 7$. The after gate counts show how subexpression eliminations lower the count, with all except the last two normal elements being at the minimum of 10 XOR gates. Note that for the matrix-vector multipliers, like the multiplication table T module, the before counts correspond to the d-XOR-counts and the after counts to the s-XOR-counts produced by the synthesis tools.
- for the matrix U multipliers, the before counts contain the AND and XOR gates, but the after counts include some AND gates represented as NOT and OR gates. The biggest reduction in the AND count is seen for ω^{19} . On average 53% of AND gates and 71% of XOR gates remain after optimizations.
- for the rr_mo multipliers, there are no reductions in the AND counts², and no NOT and OR gates are inferred. On the average 74% of XOR gates remain after optimizations.
- Furthermore, the modules for ω^{63} , ω^{21} and ω^{43} and for ω^{19} and ω^{27} , which have the same C_T , also have the same before gate counts, but differ in the after counts for the two multipliers, indicating differences in subexpression eliminations.

² reduced redundancy by design

normal element ω^{c_j}	N-polynomial $N_j(x)$	complexity of T C_T
$\alpha_0 = \omega^{63}$	$x^7 + x^6 + 1$	21
$\alpha_1 = \omega^{21}$	$x^7 + x^6 + x^3 + x + 1$	21
$\alpha_2 = \omega^{43}$	$x^7 + x^6 + x^4 + x + 1$	21
$\alpha_3 = \omega^{31}$	$x^7 + x^6 + x^4 + x^2 + 1$	25
$\alpha_4 = \omega^{13}$	$x^7 + x^6 + x^5 + x^2 + 1$	19
$\alpha_5 = \omega^{19}$	$x^7 + x^6 + x^5 + x^3 + x^2 + x + 1$	27
$\alpha_6 = \omega^{27}$	$x^7 + x^6 + x^5 + x^4 + x^2 + x + 1$	27

Table 18.1: \mathbb{F}_{27} normal elements, N-polynomials, and complexity of T C_T

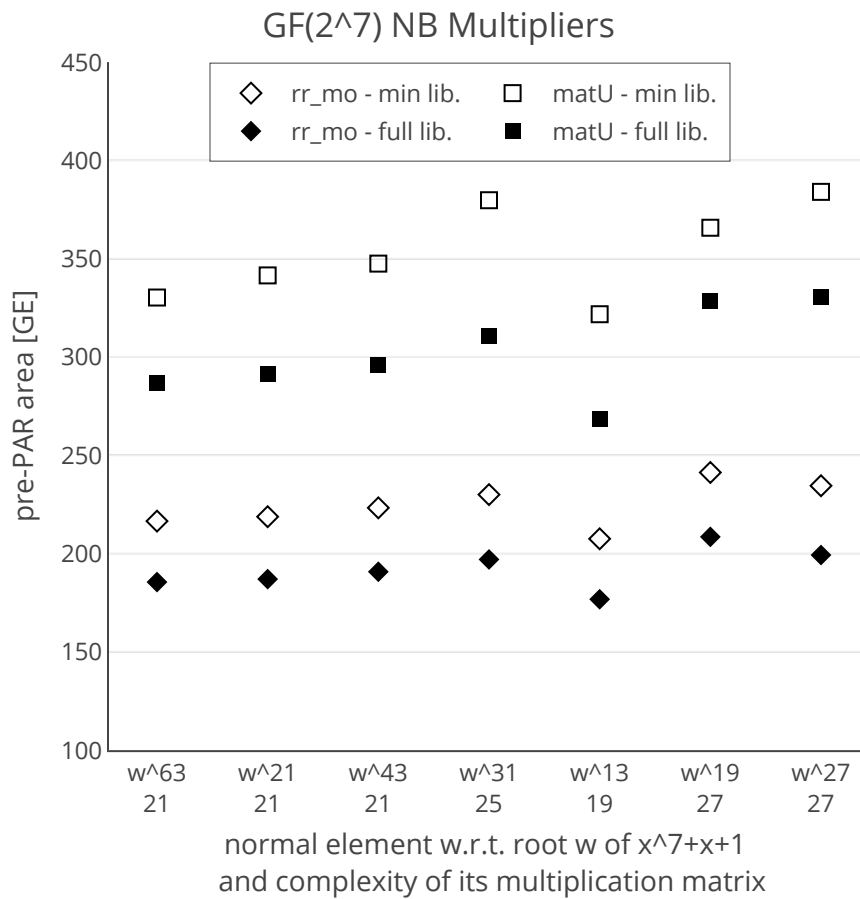


Figure 18.1: Area results for the \mathbb{F}_{27} normal basis multiplication

- examining the before and after gate counts for the ω^{19} and ω^{27} explains the aforementioned exception in the graph. The matrix U multiplier shows a bigger reduction of AND gates for ω^{19} , and the rr_mo multiplier a slightly bigger reduction of XOR gates.

normal element ω^{e_j}	C_T	Mult. table T		matrix U multiplier						rr_mo multiplier			
		before XOR	after XOR	before AND XOR		after AND XOR NOT OR				before AND XOR		after AND XOR	
ω^{63}	21	14	10	147	140	74	102	5	3	49	91	49	69
ω^{21}	21	14	10	147	140	77	104	11	2	49	91	49	70
ω^{43}	21	14	10	147	140	83	103	6	5	49	91	49	72
ω^{31}	25	18	10	175	168	86	114	7	7	49	105	49	75
ω^{13}	19	12	10	133	126	78	93	6	7	49	84	49	65
ω^{19}	27	20	12	189	182	76	118	5	1	49	112	49	80
ω^{27}	27	20	11	189	182	81	119	8	6	49	112	49	77

Table 18.2: Area results for the \mathbb{F}_{2^7} normal basis multiplication: Complexity C_T , and the *min. lib.* profile results before and after optimizations performed by the synthesis tools for the multiplication table T, the matrix U multiplier, and the rr_mo multiplier

18.2 Polynomial basis multiplication and inversion in \mathbb{F}_{2^7} using primitive defining polynomials

This example shows DSE for the polynomial basis multiplication and inversion submodules for \mathbb{F}_{2^7} . The candidates are all primitive polynomials³ of degree $m = 7$. The defining polynomial for the reference field \mathbb{F}_{2^7} is $f_{ref}(x) = x^7 + x + 1$, with root ω . The primitive polynomials are labelled $p_j(x)$ in Table 18.3. The table is organized as follows: the first column shows the root α_j of polynomial $p_j(x)$, the second column shows the polynomial $p_j(x)$ itself, and the last column the Hamming weight of the reduction matrix R, denoted “complexity of R” and C_R . The values C_R were obtained with FFCSA methods `ReductionMatrix` and `WeightMatrix` (Table 7.2). This notation was chosen to keep the formatting similar to that of the normal basis \mathbb{F}_{2^7} in Table 18.1.

The profiling loop was shown in Figure 16.2. The setup file for the multiplier is shown in Example 18.2(a): this is the initial design setup using the default `SignalPkg defaultPkg`. The loop reads the new candidate from the ply file (the next primitive polynomial `f`) and executes the template shown in Example 18.2(b). The latter constructs the field F and retrieves the root `w`, then obtains the polynomial basis, and constructs the new `SignalPkg designPkg` using the new basis, and finally continues to generate the design.

³all irreducible polynomials for this field are also primitive

Example 18.2(a)

```

K := GF(2);; x := X(K, "x");;
defaultPkg := SignalPkg();
AddFieldToSignalPkg(defaultPkg , Basis(GF(2^m)), "to");

mul := a_0*b_0;;
inputports := [ ["i_a", 3], ["i_b",3] ];;
outputports := [ ["o_z", 3] ];;
bindlist := [ ["a_0", "i_a"], ["b_0", "i_b"] ];;
exprlist := [mul];;

entitylist := ["mult", "multiply", "main"];;
portlist := [defaultPkg, inputports, outputports];;
archlist := [bindlist, exprlist];;
archtype := ["CICO", "simple"];;
sminsn := ["generate", ["matrixU"], "CICO", "simple"];;

AF := AlgFunctionality(entitylist, portlist, archlist, archtype);;
MakeAlgFunctionalityDesignReady(AF, defaultPkg);;

design := AlgDesign(AF, defaultPkg, sminsn, [folder, commentstr] , 0);;

basistype := "PB"; basisdir := "to";
SaveWorkspace(Concatenation(folder, "/savedesign1"));

```

Example 18.2(b)

```

# new field params
F := FieldExtension(K, f);; w := RootOfDefiningPolynomial(F);;
# retrieve basis call
basisused := Concatenation(basistype, basisdir);
cmd := LookupDictionary(dictBases , basisused );;
cmd := Concatenation(cmd, "(", String(F), ",", String(w), ");");
B := EvalString(cmd);

# design pkg
designPkg := SignalPkg();
AddFieldToSignalPkg(designPkg , B, "to");

if CanSwitchSignalPkg(defaultPkg, designPkg) then
  design := AlgDesign(algfun, designPkg, sminsn, [folder, commentstr] , 0);;
  nrfiles:= WriteDesignVHDL(design);
  Print(design!.filelist);
  ... OMITTED FOR BREVITY ....

```

root of $p_j(x)$ ω^{e_j}	prim. poly $p_j(x)$	complexity of R † C_R
$\alpha_0 = \omega^1$	$x^7 + x + 1$	19
$\alpha_1 = \omega^{63}$	$x^7 + x^6 + 1$	34
$\alpha_2 = \omega^{11}$	$x^7 + x^3 + 1$	21
$\alpha_3 = \omega^{29}$	$x^7 + x^4 + 1$	22
$\alpha_4 = \omega^{55}$	$x^7 + x^4 + x^3 + x^2 + 1$	35
$\alpha_5 = \omega^9$	$x^7 + x^5 + x^4 + x^3 + 1$	26
$\alpha_6 = \omega^{47}$	$x^7 + x^6 + x^5 + x^4 + 1$	27
$\alpha_7 = \omega^5$	$x^7 + x^3 + x^2 + x + 1$	34
$\alpha_8 = \omega^{21}$	$x^7 + x^6 + x^3 + x + 1$	32
$\alpha_9 = \omega^{43}$	$x^7 + x^6 + x^4 + x + 1$	34
$\alpha_{10} = \omega^{31}$	$x^7 + x^6 + x^4 + x^2 + 1$	35
$\alpha_{11} = \omega^3$	$x^7 + x^5 + x^3 + x + 1$	27
$\alpha_{12} = \omega^{13}$	$x^7 + x^6 + x^5 + x^2 + 1$	34
$\alpha_{13} = \omega^{23}$	$x^7 + x^5 + x^2 + x + 1$	32
$\alpha_{14} = \omega^{27}$	$x^7 + x^6 + x^5 + x^4 + x^2 + x + 1$	34
$\alpha_{15} = \omega^{19}$	$x^7 + x^6 + x^5 + x^3 + x^2 + x + 1$	30
$\alpha_{16} = \omega^{15}$	$x^7 + x^5 + x^4 + x^3 + x^2 + x + 1$	35
$\alpha_{17} = \omega^7$	$x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + 1$	28

† the number of XOR gates for implementation is $C_R - m$

Table 18.3: \mathbb{F}_{2^7} primitive polynomials, their roots, and complexity of R, where R is the reduction matrix

root of $p_j(x)$ ω^{c_j}	C_R	matrix U multiplication						matrix U inversion					
		before		after				before		after			
		AND	XOR	AND	XOR	NOT	OR	AND	XOR	AND	XOR	NOT	OR
ω	19	70	63	59	55	3	2	1013	422	148	291	43	14
ω^{63}	34	105	98	70	65	12	10	1008	395	172	288	34	24
ω^{11}	21	73	66	58	53	6	4	1057	425	140	291	37	20
ω^{29}	22	76	69	60	55	9	7	985	381	177	269	43	36
ω^{55}	35	120	113	81	76	16	18	1081	431	169	324	33	27
ω^9	26	104	97	82	13	17	71	1070	432	164	304	37	32
ω^{47}	27	99	92	75	72	12	19	1115	440	141	300	36	23
ω^5	34	118	111	78	79	9	10	1097	428	180	307	32	38
ω^{21}	35	114	107	78	78	10	18	1097	442	158	301	37	28
ω^{43}	34	120	113	84	78	13	17	1175	450	157	332	36	24
ω^{31}	35	129	122	92	77	18	33	1046	425	158	303	37	20
ω^3	27	102	95	76	70	7	13	1041	430	143	304	34	21
ω^{13}	34	123	116	91	77	19	24	1072	413	163	277	46	33
ω^{23}	32	118	111	73	82	11	13	1026	410	148	287	33	24
ω^{27}	34	118	111	79	78	12	18	1102	431	162	308	30	27
ω^{19}	30	114	107	87	15	24	74	1115	444	143	317	19	13
ω^{15}	35	131	124	77	84	9	15	1083	432	157	305	41	21
ω^7	28	109	102	97	71	15	31	1010	415	142	291	41	27

Table 18.4: Area results for the \mathbb{F}_{27} polynomial basis multiplication and inversion: Complexity C_R , and the *min. lib.* profile results before and after optimizations performed by the synthesis tools for matrix U multiplication and inversion in \mathbb{F}_{27}

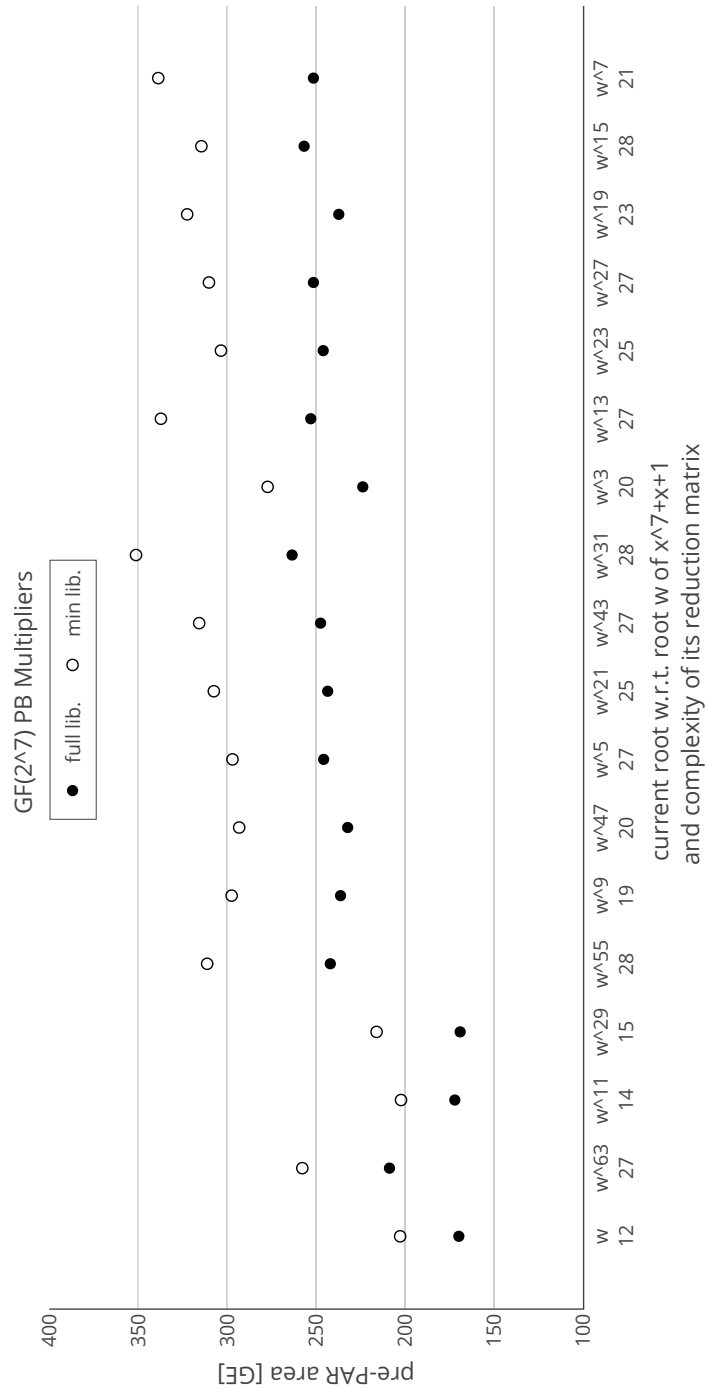


Figure 18.2: Area results for the \mathbb{F}_{2^7} polynomial basis multiplication

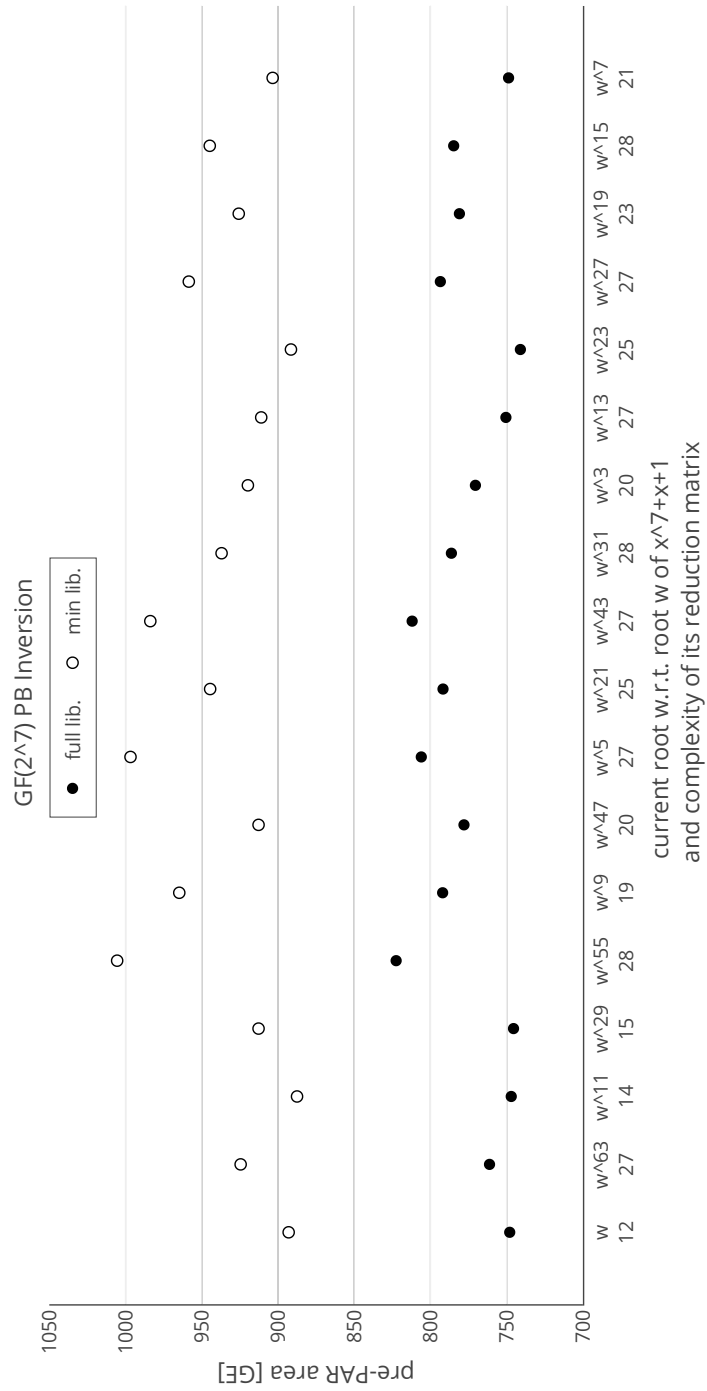


Figure 18.3: Area results for the \mathbb{F}_{2^7} polynomial basis inversion

18.3 Polynomial basis multiplication and inversion for $\mathbb{F}_{2^{14}}$ using primitive defining polynomials

This example shows DSE for the polynomial basis multiplication and inversion submodules for $\mathbb{F}_{2^{14}}$. The candidates are all primitive polynomials of degree $m = 14$. The same workflow was used as in Section 18.2. Templates shown in Example 18.2(a) and Example 18.2(b) can be reused for multipliers over arbitrary \mathbb{F}_{2^m} with no need to change any other value than m . The SignalPkg will be constructed in the same way for every field size, so the Fid does not change either. Thus, the structure of SignalPkg enables a batch mode DSE for several m , provided that all the ply files are available.

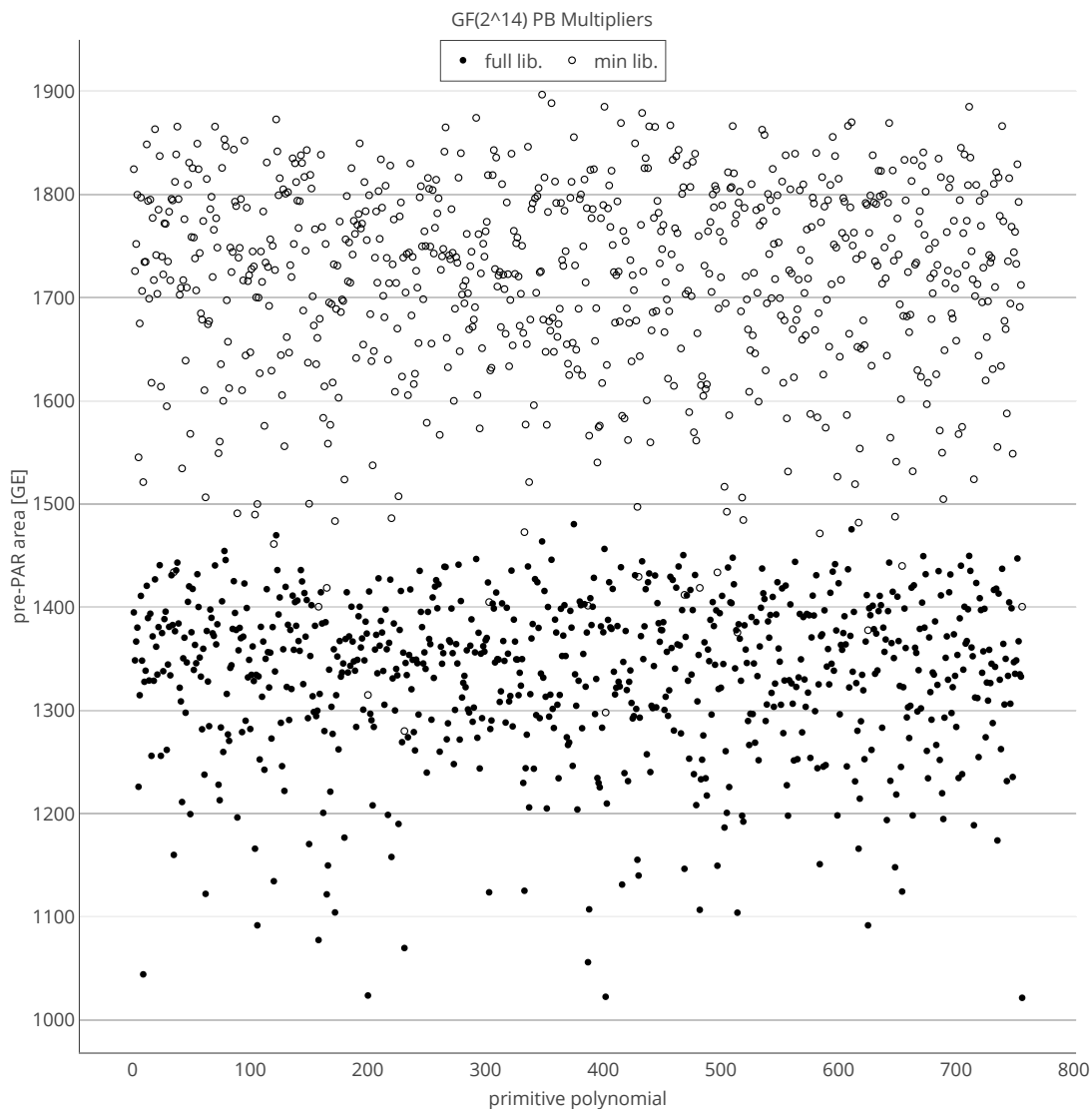


Figure 18.4: Area results for the $\mathbb{F}_{2^{14}}$ polynomial basis multiplication

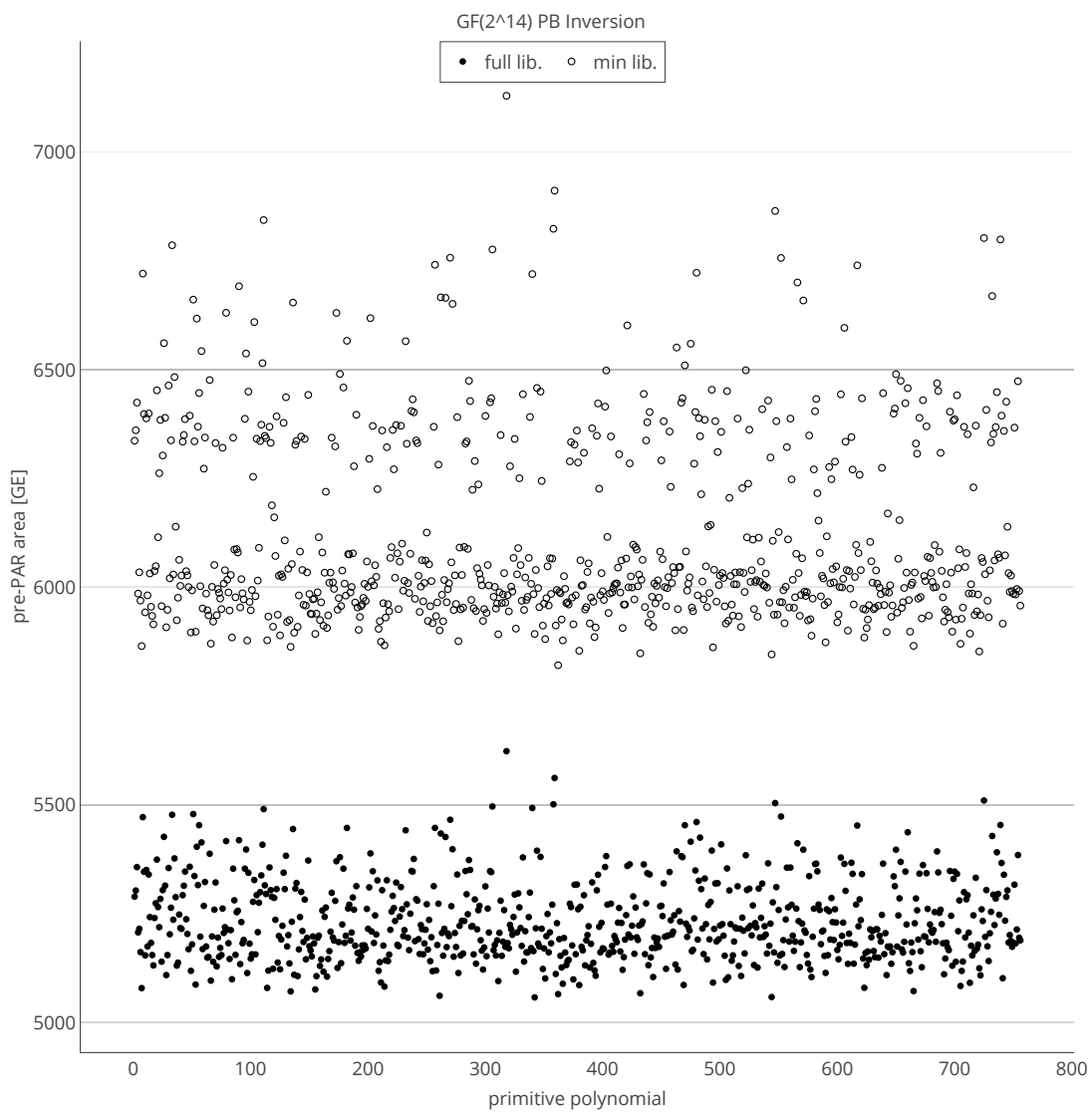


Figure 18.5: Area results for the $\mathbb{F}_{2^{14}}$ polynomial basis inversion

Chapter 19

Design space exploration of an arbitrary datapath

This chapter provides the synthesis results for the running example expression from equation (13.1), with fixed constants and 720 different tower field constructions for $\mathbb{F}_{((2^2)^2)^2}$. For details on the field construction, refer to Example 13.2.1. The implementation results are shown graphically in Figure 19.1. The profiling loop was explained in Figure 16.2 in Subsection 16.4. Figure 19.1 illustrates the importance of design space exploration and the impact of the choice of parameters such as defining polynomials. The area results range from 791–917 GE and the delays from 1.00–1.47 ns.

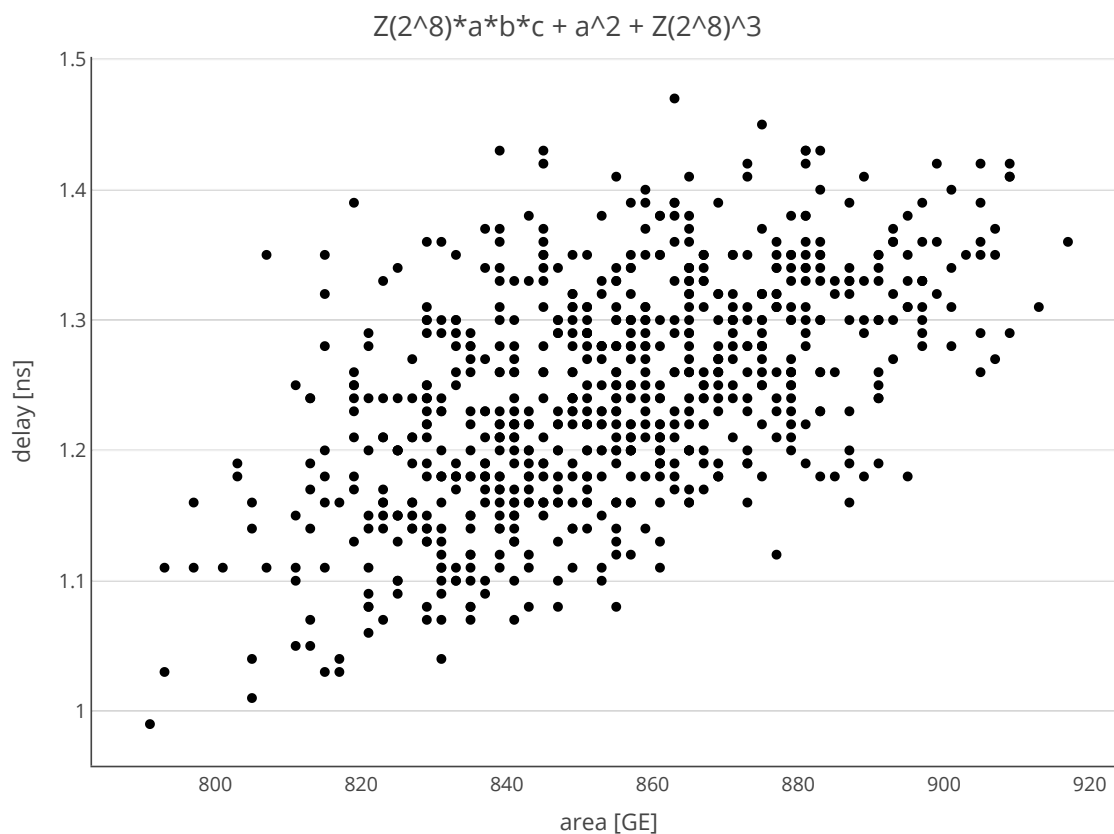


Figure 19.1: Implementation results for module implementing the expression in equation (13.1): DSE for 720 different tower field constructions for $\mathbb{F}_{((2^2)^2)^2}$

Chapter 20

The WG cipher design space exploration

20.1 Introduction	257
20.2 WGP and constant array implementations	258
20.3 WGT and the trace	259
20.4 The LFSR polynomial	260

20.1 Introduction

Cipher families like WG stream cipher can be parametrized to fit a vast range of applications. The bigger instances, e.g., WG-16, are intended for high throughput applications. The smaller instances, i.e., over small finite fields, are suitable for lightweight cryptography. A thorough discussion on lightweight cryptography was presented in Subsection 4.1.1. It is therefore desirable to have both, the estimates of hardware implementation cost and security analysis as early in the design cycle as possible, as will be seen in Chapter 22. Assuming the finite field is fixed, the design space exploration for the WG cipher can be roughly divided into two categories:

- representation of the field elements and its impact on the building blocks: when constant array is used, the building block is the constant array itself (Section 20.2)
- the LFSR polynomial: the non-zero coefficients and the constant term γ (Section 20.4)

The aforementioned constant arrays are used for the implementation of the WG permutation, discussed in Section 20.2. The WG transformation is implemented as the trace applied to the output of the WG permutation constant array; in most cases, this is the best implementation option, because the WG permutation value is used during the key initialization phase (recall K_{iAWG} in Table 8.1 in Section 8.1). Section 20.3 shows offline profiling for the trace function.

This chapter will demonstrate the cooperation of FFCSA (Chapter 7), FSR (Chapter 6), and FSR-toVHDL (Chapter 11) packages for profiling.

20.2 WGP and constant array implementations

WGP and constant array implementations are performed with the FFCSA package methods for the parameter search (e.g., to obtain the field polynomials and different bases), the FSR package to model the WGP, and the FSRtoVHDL package to obtain their corresponding hardware modules.

Previous work in [113] has shown that the best design for small WG instances is the constant array architecture¹. The WG permutation is implemented as constant arrays, indexed by the sequence element produced by clocking the LFSR, and can easily be generated by the FSRtoVHDL package as shown in Example 11.2.5 in Subsection 11.2.1. When the WGP values are not needed at all, the entire WG transformation can be implemented as a constant array. Such examples were shown in [113] for parallel implementations, where one WGP was used for key initialization and multiple WGT instances for the running phase. Online WGP profiling is shown in Example 20.2.1. This workflow was used extensively in [119, 120] for finite fields up to $\mathbb{F}_{2^{16}}$, and for the WAGE design exploration (Section 22.2) for finite fields \mathbb{F}_{2^7} and \mathbb{F}_{2^8} . Example 20.2.1 shows the WGP implementation results for 16 primitive polynomials for \mathbb{F}_{2^8} .

Mathematical parameters			WGP	
ω_i	$f_i(x)$	C_R	A [GE]	t_{cp} [ns]
$\omega_0 = \omega^1$	$x^8 + x^4 + x^3 + x^2 + 1$	36	542	1.2
$\omega_1 = \omega^{13}$	$x^8 + x^5 + x^3 + x + 1$	41	565	1.4
$\omega_2 = \omega^{31}$	$x^8 + x^5 + x^3 + x^2 + 1$	34	582	1.1
$\omega_3 = \omega^{59}$	$x^8 + x^6 + x^3 + x^2 + 1$	40	518	1.3
$\omega_4 = \omega^{37}$	$x^8 + x^6 + x^4 + x^3 + x^2 + x + 1$	37	517	1.0
$\omega_5 = \omega^{23}$	$x^8 + x^6 + x^5 + x + 1$	38	513	1.0
$\omega_6 = \omega^{19}$	$x^8 + x^6 + x^5 + x^2 + 1$	39	518	1.0
$\omega_7 = \omega^7$	$x^8 + x^6 + x^5 + x^3 + 1$	44	573	1.1
$\omega_8 = \omega^{127}$	$x^8 + x^6 + x^5 + x^4 + 1$	37	510	1.2
$\omega_9 = \omega^{53}$	$x^8 + x^7 + x^2 + x + 1$	40	574	1.1
$\omega_{10} = \omega^{29}$	$x^8 + x^7 + x^3 + x^2 + 1$	38	556	1.0
$\omega_{11} = \omega^{47}$	$x^8 + x^7 + x^5 + x^3 + 1$	40	567	1.4
$\omega_{12} = \omega^{43}$	$x^8 + x^7 + x^6 + x + 1$	40	515	1.5
$\omega_{13} = \omega^{61}$	$x^8 + x^7 + x^6 + x^3 + x^2 + x + 1$	41	584	1.2
$\omega_{14} = \omega^{11}$	$x^8 + x^7 + x^6 + x^5 + x^2 + x + 1$	38	514	1.3
$\omega_{15} = \omega^{91}$	$x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1$	38	581	1.3

Table 20.1: Pre-PAR CMOS 65nm implementation results for the PB constant array WGP for \mathbb{F}_{2^8} with decimation exponent $d = 19$, using $f_{ref}(x) = x^8 + x^4 + x^3 + x^2 + 1$, where $f_{ref}(\omega) = 0$

¹ a lookup table based design

Example 20.2.1 *Online profiling of WGP constant array implementation for \mathbb{F}_{2^8}* \longleftrightarrow

Table 20.1 shows the pre-PAR implementation results using the full CMOS 65nm library, for the normal basis WGP constant array implementation. This example uses decimation exponent $d = 19$, based on optimal parameters in [50].

The FFCSA method FindPrimitivePolyAll was used for the parameter search (Table 7.4). The polynomials were sorted by the power of their root ω_i (column 1) w.r.t. ω , where ω is the root of the reference defining polynomial $f_{ref}(x) = x^8 + x^4 + x^3 + x^2 + 1$. Column 2 lists the primitive polynomial, which is also the minimal polynomial f_i of ω_i . For additional reference, column 3 shows C_R , the complexity of the reduction matrix, obtained with the ReductionMatrix method. Last two columns show the online implementation results: the area [GE] and the critical path delay [ns]. The hardware modules were generated with the FSRtoVHDL package.

The smallest area WGP constant array is obtained for the polynomial basis $PB = \{1, \omega_8, \omega_8^2, \dots, \omega_8^7\}$ with defining polynomial $f_8(x) = x^8 + x^6 + x^5 + x^4 + 1$ at 510 GEs. Note that reciprocals are included in the profiling: polynomial $f_0(x) = x^8 + x^4 + x^3 + x^2 + 1$ has area 542 GE. Average² area of a WGP table is 546 GE. Furthermore, the lowest area WGP does not have the smallest theoretical C_R .

\longleftarrow

20.3 WGT and the trace

While the trace in equation (3.6) is independent of the basis used, the basis must be known for efficient implementation in hardware. Following equation (3.6) directly requires the implementation of the modules for all conjugates and the XOR gates to sum them up, yielding a large hardware area. Instead, the FFCSA package was used to obtain the ANF expression for the trace by representing the output as an expression containing only the coordinates of the field element w.r.t. to the given basis. Trace is also a perfect example of offline profiling that can be performed early on during *the architectural decisions*, and is not expected to change significantly after being implemented. Differences arise when the synthesis tools use arbitrary gates such as 3-input XORs. Offline trace profiling is shown in Example 20.3.1. This workflow was used extensively in [119, 120] for finite fields up to $\mathbb{F}_{2^{16}}$.

Example 20.3.1 *Offline profiling of the trace expression* \longleftrightarrow

This example shows simplest use of FFCSA package for the offline profiling: the trace expressions were obtained for 18 primitive polynomials of degree 7, using the FFA_trace method (Table 7.5), and the Hamming weights with the WeightPolynomial method (Table 7.2). The number of 2-input XOR gates for the implementation is Hamming weight - 1.

² averages were computed before the area results were rounded up

Example 20.3.1

```

x^7+x+Z(2)^0 -> a_0 -> hamming weight = 1
x^7+x^3+Z(2)^0 -> a_0 -> hamming weight = 1
x^7+x^3+x^2+x+Z(2)^0 -> a_0+a_5 -> hamming weight = 2
x^7+x^4+Z(2)^0 -> a_0+a_3+a_6 -> hamming weight = 3
x^7+x^4+x^3+x^2+Z(2)^0 -> a_0+a_3+a_5+a_6 -> hamming weight = 4
x^7+x^5+x^2+x+Z(2)^0 -> a_0+a_5 -> hamming weight = 2
x^7+x^5+x^3+x+Z(2)^0 -> a_0 -> hamming weight = 1
x^7+x^5+x^4+x^3+Z(2)^0 -> a_0+a_3+a_5+a_6 -> hamming weight = 4
x^7+x^5+x^4+x^3+x^2+x+Z(2)^0 -> a_0+a_3+a_6 -> hamming weight = 3
x^7+x^6+Z(2)^0 -> a_0+a_1+a_2+a_3+a_4+a_5+a_6 -> hamming weight = 7
x^7+x^6+x^3+x+Z(2)^0 -> a_0+a_1+a_2+a_3+a_4+a_6 -> hamming weight = 6
x^7+x^6+x^4+x+Z(2)^0 -> a_0+a_1+a_2+a_4 -> hamming weight = 4
x^7+x^6+x^4+x^2+Z(2)^0 -> a_0+a_1+a_2+a_4+a_5 -> hamming weight = 5
x^7+x^6+x^5+x^2+Z(2)^0 -> a_0+a_1+a_2+a_4 -> hamming weight = 4
x^7+x^6+x^5+x^3+x^2+x+Z(2)^0 -> a_0+a_1+a_2+a_4+a_5 -> hamming weight = 5
x^7+x^6+x^5+x^4+Z(2)^0 -> a_0+a_1+a_2+a_3+a_4+a_5+a_6 -> hamming weight = 7
x^7+x^6+x^5+x^4+x^2+x+Z(2)^0 -> a_0+a_1+a_2+a_3+a_4+a_6 -> hamming weight = 6
x^7+x^6+x^5+x^4+x^3+x^2+Z(2)^0 -> a_0+a_1+a_2+a_3+a_4+a_5+a_6 -> hamming weight = 7

```



20.4 The LFSR polynomial

Work on WG-16 in [8] analyzed the LFSR with the feedback polynomial $\ell_1(x) = x^{32} + x^{25} + x^{16} + x^7 + \tau$, $\tau = \omega^{2743} \in \mathbb{F}_{2^{16}}$, where ω is a root of the field defining polynomial $x^{16} + x^5 + x^3 + x^2 + 1$. As WG-16 is not a lightweight instance of the WG stream cipher family, the implementation in [8] aimed for a high frequency design, not small area.

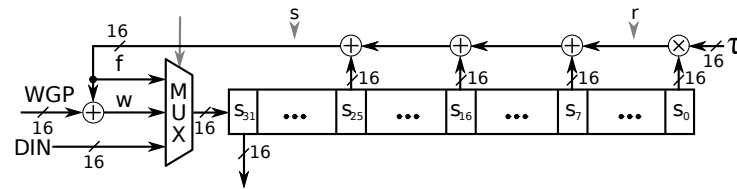
During the design space exploration reported in [8], the LFSR became the critical component in the design. First, feedback pipelining (retiming) was used to fix the problem. Then, the number of terms in the LFSR feedback was increased at no cost to hardware. Next, using the FFCSA method (Table 7.4 in Section 7.1) `FindPrimitivePolyExtraTapsFixedPoly•([of,] F, f, t)` new feedback polynomials were found, where F is $\mathbb{F}_{2^{16}}$, and t the number of extra terms. The number of extra terms was estimated from the maximum row Hamming weight of the matrix-vector multiplier for the constant term τ ; it was set to $t=5$ to balance the theoretical critical path of the two parts of the feedback. Two methods were used, as noted by the •:

- `FindPrimitivePolyExtraTapsFixedPolyFixedGamma`: with argument $f = \ell_1(x)$, including the constant term τ (τ will not change)
- `FindPrimitivePolyExtraTapsFixedPoly`: with argument $f = \ell_1(x) - \tau$, and allow new constants

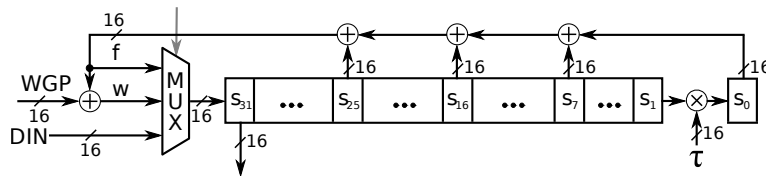
The final polynomials used in [8] were:

$$\begin{aligned} \ell_1(x) &= x^{32} + x^{25} + x^{16} + x^7 + \omega^{2743} \\ \ell_2(x) &= x^{32} + x^{25} + x^{23} + x^{22} + x^{16} + x^{15} + x^{12} + x^{10} + x^7 + \omega^{2743} \\ \ell_3(x) &= x^{32} + x^{25} + x^{16} + x^7 + \omega^{24319} \\ \ell_4(x) &= x^{32} + x^{30} + x^{25} + x^{24} + x^{22} + x^{16} + x^{14} + x^7 + x^2 + \omega^{24319} \end{aligned}$$

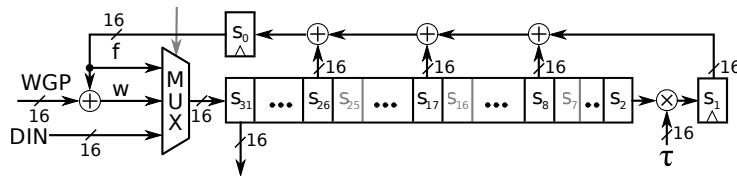
The best post-PAR CMOS 65nm implementation results are reported in Table 4.3 in Subsection 3.2.4. Note that in [8], the LFSRs were implemented manually, not using FSRtoVHDL.



(a) LFSR L0 with original (*no-retiming*) feedback



(b) LFSR L1 with feedback *retimed once*



(c) LFSR L2 with feedback *retiming twice*

Figure 20.1: Schematics for the LFSR with original feedback and its *retimed* versions

Three possible hardware implementations of the same LFSR were explored, and can be distinguished by feedback *retiming*:

- the *no-retiming* version with entire feedback computation separated from the shift registers (L0 in Figure 20.1(a))
- feedback computation *retimed once*, by *retiming* applied to multiplication by the constant τ (L1 in Figure 20.1(b))
- feedback computation *retimed twice*, by *retiming* applied to both feedback and multiplication by the constant τ (L2 in Figure 20.1(c))

Part VI

WAGE

Part VI - Outline

21 WAGE - overview	264
22 WAGE algorithm design - impact of profiling	265
23 Hardware design of the WAGE datapath	272

Chapter 21

WAGE - overview

Disclaimer 21.1: The WAGE authenticated encryption scheme

The WAGE authenticated encryption scheme is a joint work of the members of the ComSec Lab (will be referred to as the WAGE team), listed in alphabetical order: Mark Aagaard, Riham AlTawy, Guang Gong, Kalikinkar Mandal, Raghvendra Rohit, and Nusa Zidaric (author of this thesis). Subsection 3.2.8 is a background section, summarizing the specifications of the LWC candidate WAGE [6], as submitted to the LWC [5] (joint work). Part VI consists of two chapters:

- Chapter 22, which shows the impact of hardware implementations during the *algorithm design* of WAGE. My contributions to the WAGE design are the parameter search, (automated) hardware implementations during the algorithm design of WAGE, and contribution to the loading sequence and input ports, and tag extraction sequence and output ports.
- Chapter 23, which explains the *hardware design* of the WAGE datapath in depth. My contribution is the hardware design (and implementation) for the WAGE datapath. However, the hardware design slowly evolved concurrently with the algorithm design and in constant communication with the entire WAGE team. A short summary of the joint work of the WAGE team with no contribution of the author of this thesis is presented in Section 23.1

The presented text also in Part VI overlaps to some degree with the text in WAGE's submission document [6] (joint work), and in [188]. Acknowledgement goes to Marat Sattarov for help with synthesis scripts, developed by Mark Aagaard (reuse of the design flow for the WG cipher constant array implementations, used in Marat Sattarov's MASc thesis [120]).

Chapter 22

WAGE algorithm design - impact of profiling

22.1 Introduction	265
22.2 The size of the underlying finite field	266
22.3 The remaining nonlinear components	268
22.4 The LFSR feedback	269
22.5 The WAGE permutation hardware area estimate	270
22.6 Summary	271

22.1 Introduction

This section summarizes different options explored during the design of the WAGE algorithm, with a focus on the hardware estimates. WAGE was designed for the NIST LWC competition. A thorough discussion on lightweight cryptography was presented in Subsection 4.1.1. For lightweight cryptography, area is more important than delay. The design process targeted a small implementation area for the WAGE permutation, with the threshold set to approximately 2000GE. The final design, i.e., the permutation that met this criterion, is shown in Algorithm 1 in Section 3.2.8. The area metrics used are a mix of estimates and synthesis results:

- estimates: pen and paper analysis using 3.75GE for a 1-bit register and 2.00GE for a 2-input XOR gate.
- synthesis results: pre-PAR implementation results using CMOS 65nm

As the key and nonce are set to 128 bits, assuming internal state of at least 256 bits, the area for the state is estimated at 960GE already taking up a large portion of the 2000GE threshold. The overall structure of WAGE was fixed to an LFSR with two decimated WG permutations $WGP-16(X^d)$, WGP for short, and additional nonlinear elements for faster confusion and diffusion. From previous work [68, 113], it is known that for small fields, the WGP implemented as a constant array, i.e. as a look-up table, is smaller than the WGP implemented with basic arithmetic blocks implementing multiplications and exponentiations to the powers of two. However, the WGP is not stored in

hardware as a memory array, but rather as a net of AND, OR, and NOT gates, derived by the synthesis tools. The WGP module is small enough to be successfully optimized by the synthesis tools. For the DSE of the WG cipher family, see Chapter 20.

22.2 The size of the underlying finite field

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

The first option considered for the underlying finite field was \mathbb{F}_{2^8} , due to its efficiency in software. The choice of the underlying finite field was based on the area of the WGP constant array implementation for polynomial bases obtained using primitive defining polynomials, which was already discussed in Section 20. More specifically, Table 20.1 in Section 20 showed that the average area for the WGP constant array implementations is 546GE, which raises the total area of the design over 1000GE with only two WGP tables (without any additional nonlinear components). The implementation results for the normal bases \mathbb{F}_{2^8} WGP are similar, with 16 normal elements and an average area of 531GE, just slightly lower.

The next candidate field was \mathbb{F}_{2^7} . Two options were explored: normal bases and polynomial bases for the representation of the field elements.

Normal bases for \mathbb{F}_{2^7}

Table 22.1 shows the implementation results for the normal basis WGP constant array implementations, using decimation exponent $d = 13$, based on optimal parameters in [50].

The FFCSA methods for search (Table 7.4) and normal bases (Table 7.3) were used. The search was reduced to ignore the conjugates, i.e., `FindNormalFFEsBIgnoreConjugates` was used. The normal elements found are listed in the first column of Table 22.1 as n_i , and are given as a power of ω , where ω is the root of the reference defining polynomial $f_{ref}(x) = x^7 + x + 1$. Column 2 lists the minimal polynomial f_i of n_i , which is used as the defining polynomial when this normal element/basis is selected for the implementation. The normal bases were generated with `GenerateNB` and Column 4 shows C_T , the complexity of the multiplication table for multiplication with the normal element, obtained with the `ComplexityOfT` method. The last two columns show the implementation results: the area [GE] and the critical path delay [ns]. The hardware modules were generated with the `FSRtoVHDL` package.

Mathematical parameters			WGP	
n_i	$f_i(x)$	C_T	A [GE]	t_{cp} [ns]
$n_0 = \omega^{63}$	$x^7 + x^6 + 1$	21	258	0.8
$n_1 = \omega^{21}$	$x^7 + x^6 + x^3 + x + 1$	21	261	0.8
$n_2 = \omega^{43}$	$x^7 + x^6 + x^4 + x + 1$	21	251	0.7
$n_3 = \omega^{31}$	$x^7 + x^6 + x^4 + x^2 + 1$	25	248	0.7
$n_4 = \omega^{13}$	$x^7 + x^6 + x^5 + x^2 + 1$	19	264	0.9
$n_5 = \omega^{19}$	$x^7 + x^6 + x^5 + x^3 + x^2 + x + 1$	27	270	0.7
$n_6 = \omega^{27}$	$x^7 + x^6 + x^5 + x^4 + x^2 + x + 1$	27	256	0.8

Table 22.1: Pre-PAR CMOS 65nm implementation results for the NB constant array WGP for \mathbb{F}_{2^7} with decimation exponent $d = 13$, using $f_{ref}(x) = x^7 + x + 1$, where $f_{ref}(\omega) = 0$

Mathematical parameters			WGP		ω_i mult.
ω_i	$f_i(x)$	C_R	A [GE]	t_{cp} [ns]	A [GE]
$\omega_0 = \omega^1$	$x^7 + x + 1$	19	258	0.7	2
$\omega_1 = \omega^{11}$	$x^7 + x^3 + 1$	21	247	0.6	16
$\omega_2 = \omega^5$	$x^7 + x^3 + x^2 + x + 1$	34	245	0.7	10
$\omega_3 = \omega^{29}$	$x^7 + x^4 + 1$	22	243	0.7	23
$\omega_4 = \omega^{55}$	$x^7 + x^4 + x^3 + x^2 + 1$	35	255	0.7	22
$\omega_5 = \omega^{23}$	$x^7 + x^5 + x^2 + x + 1$	32	258	0.7	24
$\omega_6 = \omega^3$	$x^7 + x^5 + x^3 + x + 1$	27	261	0.7	6
$\omega_7 = \omega^9$	$x^7 + x^5 + x^4 + x^3 + 1$	26	264	0.7	116
$\omega_8 = \omega^{15}$	$x^7 + x^5 + x^4 + x^3 + x^2 + x + 1$	35	251	0.7	19
$\omega_9 = \omega^{63}$	$x^7 + x^6 + 1$	34	270	0.9	14
$\omega_{10} = \omega^{21}$	$x^7 + x^6 + x^3 + x + 1$	32	248	0.7	28
$\omega_{11} = \omega^{43}$	$x^7 + x^6 + x^4 + x + 1$	34	261	0.7	29
$\omega_{12} = \omega^{31}$	$x^7 + x^6 + x^4 + x^2 + 1$	35	265	0.7	27
$\omega_{13} = \omega^{13}$	$x^7 + x^6 + x^5 + x^2 + 1$	34	257	0.8	16
$\omega_{14} = \omega^{19}$	$x^7 + x^6 + x^5 + x^3 + x^2 + x + 1$	30	257	0.7	26
$\omega_{15} = \omega^{47}$	$x^7 + x^6 + x^5 + x^4 + 1$	27	259	0.9	31
$\omega_{16} = \omega^{27}$	$x^7 + x^6 + x^5 + x^4 + x^2 + x + 1$	34	254	0.7	20
$\omega_{17} = \omega^7$	$x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + 1$	28	255	0.8	14

Table 22.2: Pre-PAR CMOS 65nm implementation results for the PB constant array WGP for \mathbb{F}_{2^7} with decimation exponent $d = 13$, using $f_{ref}(x) = x^7 + x + 1$, where $f_{ref}(\omega) = 0$. The last column contains implementation results for the multiplication with ω_i

Polynomial bases for \mathbb{F}_{2^7}

The pre-PAR CMOS 65nm implementation results for constant array implementations of WGP using the decimation exponent $d = 13$ are listed in Table 22.2: they show 18 primitive¹ polynomials of degree 7, denoted $f_i(x)$. Each of the f_i has a different root ω_i and a different polynomial basis. The profiling workflow is exactly the same as for the \mathbb{F}_{2^8} WGP shown in Example 20.2.1. The theoretical estimate included is the Hamming weight of the $m \times (2m - 1)$ reduction matrix obtained for f_i .

The smallest area for WGP-16(X^d) was found for $f_3(x) = x^7 + x^4 + 1$. Considering the constant term for the LFSR feedback, the smallest combined area of two WGPs and a constant term was found for the polynomial $f_2(x) = x^7 + x^3 + x^2 + x + 1$. The implementation results for the matrix-vector multiplier for multiplication with ω_i are shown in the last column (area only, delay is negligible). Note that the constant term multiplier amounts to only 2% of the combined area.

22.3 The remaining nonlinear components

Normal bases for \mathbb{F}_{2^7}

In Section 22.2, both polynomial and normal bases were considered. Normal bases are attractive because the exponentiations to powers of 2 are cyclic shifts, and hence implemented for free. For nonlinear components, the Gold exponents (see 9.2.59 in [15]) of the form x^{2^i+1} were considered. These exponents can be implemented with a cyclic shift and a multiplication. Table 22.3 lists the implementation results for the rr_mo multipliers [156] and for a whole module $x^{33} = x^{2^5+1}$, which is a permutation. The rr_mo multipliers were already discussed in Section 18.1. The x^{33} modules were implemented using the CIRCUIT package with the x^{33} set as the expression input (Chapter 14). Because x^{2^5} requires only a cyclic shift, which is free in hardware, the multiplier was directly compared to the exponentiation module for x^{33} . Area-wise, both modules were very close, with the multipliers having the average area of 189 GE and the x^{33} modules 192 GE. Their size was, on the average, 73% and 74% of the size of an average WGP, respectively. The smallest combined area of two WGP modules and

- two multipliers is achieved for $f_4(x) = x^7 + x^6 + x^5 + x^2 + 1$ with 882GE
- two x^{33} modules is achieved for $f_3(x) = x^7 + x^6 + x^4 + x^2 + 1$ with 772GE

Two alternative implementations were tested for the x^{33} modules: the constant array implementation and the canonical disjunctive normal form (CDNF) implementation. The constant array was translated to CDNF using Logic Friday [189], which is using Espresso for minimization. For example, the $f_3(x) = x^7 + x^6 + x^4 + x^2 + 1$ CDNF x^{33} module achieves an area of 267GE, almost twice the value reported in Table 22.3, and the constant array implementation is even smaller, with

¹ all irreducible polynomials for \mathbb{F}_{2^7} are also primitive

Mathematical parameters			rr_mo multiplier		x^{33} module	
n_i	$f_i(x)$	C_T	A [GE]	t_{cp} [ns]	A [GE]	t_{cp} [ns]
$n_0 = \omega^{63}$	$x^7 + x^6 + 1$	21	185	0.4	182	0.4
$n_1 = \omega^{21}$	$x^7 + x^6 + x^3 + x + 1$	21	187	0.4	143	0.3
$n_2 = \omega^{43}$	$x^7 + x^6 + x^4 + x + 1$	21	191	0.3	195	0.6
$n_3 = \omega^{31}$	$x^7 + x^6 + x^4 + x^2 + 1$	25	197	0.4	138	0.4
$n_4 = \omega^{13}$	$x^7 + x^6 + x^5 + x^2 + 1$	19	177	0.3	156	0.4
$n_5 = \omega^{19}$	$x^7 + x^6 + x^5 + x^3 + x^2 + x + 1$	27	208	0.4	184	0.4
$n_6 = \omega^{27}$	$x^7 + x^6 + x^5 + x^4 + x^2 + x + 1$	27	199	0.4	164	0.4

Table 22.3: Pre-PAR CMOS 65nm implementation results for the NB multipliers and x^{33} modules for \mathbb{F}_{2^7} , using the reference defining polynomial $f_{ref}(x) = x^7 + x + 1$, where $f_{ref}(\omega) = 0$

213GE. In general, the CDNF and constant array implementation yield similar results because the underlying process of minimization is similar. The outputs of Logic Friday are seven expressions, one for each 7-input/1-output truth table, while the constant array implementation is a 7-input/7-output table; the latter could also give more room for optimizations.

Polynomial bases for \mathbb{F}_{2^7}

Alternatively, 7-bit Sboxes were used to try to further reduce the implementation area. The final candidates were implemented and based on a security-area tradeoff. An Sbox with only 58GE was chosen for WAGE; the Sbox SB details are listed in Table 3.1 in the Subsection 3.2.8, and for more details refer to [6]. Note that the SB is defined over \mathbb{F}_2^7 and does not depend on the basis; the same SB can be used for both, the polynomial and the normal bases. As the SB does not contain any operations that would benefit from the use of a normal basis, the normal bases were discarded. The final design uses two WGP and four Sboxes SB, and the output of each is XORed back to one of the (7-bit) LFSR stages; these XOR gates count towards the row “other XORs” in Table 22.4.

The area of the implemented Sboxes varied from 55 to 65GE, which is much smaller than the proposed Gold exponents. The small area advantage of the Sboxes lies in their iterative structure, which allows an unrolled implementation with a small number of gates. For example, if the 7-bit Sboxes were implemented using constant arrays, their implementation area would be comparable to the area of the normal bases multipliers and x^{33} modules in Table 22.3.

22.4 The LFSR feedback

Exhaustive search is performed to find the best primitive polynomials for the LFSR based on the delay and area of the matrix for multiplication with a constant. For lightweight cryptography, the

area is more important than delay, so the primary search criterion is the total Hamming weight of the matrix, rather than the maximum row Hamming weight. The offline profiling for the smallest area constant term is shown in Example 7.2.1 in Section 7.2. The profiling using the FFCSA methods `ProfileGamma` and `FindSmallestAreaGamma` (Table 7.4) for WAGE parameters is shown in Example 7.2.1(d).

The online profiling flow used the FFCSA method `MatrixMultByConstExpression` (Table 7.5) to obtain expressions for the matrix-vector multiplier and the CIRCUIT package to generate its datapath. The online profiling results for the constant term are listed in the last column in Table 22.2, labelled “ ω_i mult.”.

Finally, the candidates for the LFSR polynomial over \mathbb{F}_{27} were found as follows. The LFSR polynomial should have degree 37 and 10 tap positions. The 10 tap positions ensure that 70 state bits enter the feedback, which is more than the rate of 64 bits. Two positions were fixed: namely, y^{19} (to avoid stage S_{18} , which has a WGP), and the constant term ω . The remaining 8 taps were placed evenly in the two halves of the WAGE state, with symmetry after y^{18} . The GAP function `IsPrimitivePolynomial(F, poly)` (Section 66.4-12 in the GAP reference manual [34]) was used to decide if the candidate poly is primitive. A total of 68 primitive polynomials were found. The candidate list was further reduced to 36 primitive polynomials by disallowing the terms y^{36} and y^{35} . Stacking the taps towards the lower stages of the LFSR is beneficial for parallel implementations. For example, if y^{36} was present in a two-way parallel implementation, the second feedback needs to be computed using $y^{36} \cdot y = y^{37}$, which requires adding the entire first feedback to the second one, resulting in a long combinational path. The candidate list was further reduced based on the security analysis (for details see Section 5 in [6]).

22.5 The WAGE permutation hardware area estimate

Table 22.4 shows the estimates for the area of the WAGE permutation. The area cost for the LFSR is estimated as shown in the first 3 rows of Table 22.4, with $37 \times 7 \times (3.75\text{GE}) = 971\text{GE}$ for the internal state, $10 \times 7 \times (2\text{GE}) = 140\text{GE}$ for the feedback XORs, and synthesized matrix-vector multiplier for the constant term. The pen-and-paper analysis in Table 22.4 is the first estimate, not considering the hardware needed to accommodate the mode, e.g., loading multiplexers etc., hence using the Hamming weight is a fair estimate.

Thorough area analysis showing the impact of the mode and new implementation results of WAGE are reported in [188].

Component	Estimate per unit [GE]	Count	Estimate per component [GE]
wage_lfsr registers	3.75	37×7	971
wage_lfsr feedback XORs	2.00	10×7	140
wage_lfsr feedback ω	10 [†]	1	10
SB	58 [†]	4	232
WGP	245 [†]	2	490
lfsr_c	45 [†]	1	45
Other XORs	2.00	8×7	112
WAGE permutation - Total estimated area			2000
pre-PAR CMOS 65nm implementation area results			
WAGE permutation (Chapter 23)			2051 [†]

[†] pre-PAR CMOS 65nm implementation results

Table 22.4: WAGE permutation hardware area estimate [187]

22.6 Summary

This chapter presented DSE during the WAGE algorithm design. The threshold for WAGE permutation was set to 2000GE. The parameter search was focused on two finite fields \mathbb{F}_{2^8} and \mathbb{F}_{2^7} . Finite field \mathbb{F}_{2^8} was discarded due to the large area of the WGP constant array implementations. DSE for \mathbb{F}_{2^7} was performed for normal and polynomial bases. Normal bases \mathbb{F}_{2^7} options were discarded due to the large area of the normal bases multiplier and exponentiation module. The last parameter set was the LFSR feedback polynomial.

The candidate field parameters were generated using the FFCSA search methods (Table 7.4) and methods for bases (Table 7.3). The normal bases rr_mo multipliers [156] were implemented using a stand-alone GAP script, and the exponentiation modules were implemented using the CIRCUIT package. The constant term for the LFSR polynomial was selected in two steps using offline and online profiling. The offline profiling used the FFCSA methods (Table 7.4). The online profiling used the FFCSA method `MatrixMultByConstExpression` (Table 7.5) to obtain expressions for the MV multiplier and the CIRCUIT package to synthesize its datapath.

Chapter 23

Hardware design of the WAGE datapath

23.1 Short summary of the joint work	272
23.2 Hardware design of the WAGE datapath	273
23.3 The WAGE datapath and the FSRtoVHDL package	277
23.4 Summary	279

23.1 Short summary of the joint work

My contribution is the hardware design (and implementation) for WAGE datapath. While other aspects of the algorithm had a direct impact on the datapath, the datapath design affected the overall algorithm and input/output protocol. This section summarizes aspects of WAGE that the author of this thesis was not directly involved with and that impacted the datapath. As WAGE is intended to be lightweight, certain hardware design principles were followed: both encryption and decryption are implemented in one module, which has a single input and a single output port, e.g., the input port is shared by the key, nonce, associated data, and the message. Furthermore, each port is paired with a valid bit to denote when the data is valid. This enables stalling: the hardware waits in an idle state signalling to the environment that it is ready to receive. The environment is responsible for possible padding and for control signals: an `i_mode` signal to distinguish encryption and decryption, and the domain separator signal `i_dom_sep` to indicate which data is on the input port, e.g., a nonce or associated data (see Figure 3.10 in Subsection 3.2.8). More details on the interaction between the environment and the `WAGE_cipher` can be found in [6], in the Interface protocol and Protocol timing. WAGE uses the unified sponge duplex mode for sLiSCP [61] to provide the AEAD functionality, as shown in Figure 3.10. To support this mode, `WAGE_cipher` operates in one of six phases, that were identified as: (I.) loading, (II.) permutation, (III.) absorbing during initialization, processing associated data and finalization, (IV.) (absorbing during) encryption, (V.) (replacing during) decryption, and (VI.) tag extraction. The six phases were used for both FSM and datapath design. For example, during encryption, one clock cycle is spent in phase (IV.) to encrypt and output the ciphertext and absorb the ciphertext, followed by the 111 clock cycles in phase (II.), the permutation.

23.2 Hardware design of the WAGE datapath

The WAGE input and output port positions, as well as the loading and tag regions, evolved together with the WAGE datapath. The loading and tag extracting are presented in Appendix B.1.

The behaviour of the six phases is explained with help of the dataflow diagrams (DFD). The dataflow diagram was chosen for the representation of the WAGE hardware because it captures both the behaviour and the hardware components used. The data flow is indicated with arrows and the grey thick horizontal lines indicate a clock cycle boundary: the time is incremented by moving downward vertically. All stages are registers, and an arrow crossing the clock cycle boundary marks the updating of the corresponding register (stage). Stage numbers are shown on top of all diagrams, e.g., 36 is indicating the stage S_{36} register¹. The stages $S_{32} \rightarrow S_{21}$ are omitted for brevity. Some paths through the DFDs are shaded to provide an example of activity, e.g., loading through D_0 in Figure 23.1; paths that are not shaded are also active. Whenever there is a different behaviour, i.e., different path driving the same stage, control circuitry is needed.

The six phases are explained out of order, to emphasize the similarities and reuse of hardware: the rounds use the same hardware, but in different clock cycles. In general, reuse saves hardware area, but there is a tradeoff for control circuitry (more multiplexers, more complicated FSM).

The loading and tag extraction are discussed in detail in appendix Chapter B.1; both take advantage of the shifting nature of an LFSR. Three clock cycles of loading (phase(I.)) are shown in the DFD in Figure 23.1. The data inputs (re)used for loading are D_9, D_5, D_4, D_3 and D_0 for the state registers $S_{36}, S_{27}, S_{18}, S_{16}$ and S_8 , respectively. All the remaining registers simply shift as they would during a normal LFSR operation. The DFD shows the annotated loading path through D_0 : the new data is first stored into S_8 in the first cycle, then shifted to S_7 in the second cycle, and then to S_6 in the third cycle. Old data in stages S_{19}, S_{17}, S_9 and S_0 is garbage and is overwritten; this approach allows state registers without reset signals, hence smaller hardware area.

The DFD in Figure 23.2 shows replacing (phase(V.)), with the XOR gates for decryption, the output path o_data , and the domain separator added to stage S_0 . The path through D_0 is again shaded: note the similarity between this shaded path and the shaded loading path in the first clock cycle of the DFD in Figure 23.1. This similarity indicates that the loading and replacing are similar enough that the replacing multiplexer can be reused for loading. Also note that during the replacing clock cycle (Figure 23.2), only the rate stages are updated, while all the other stages remain unchanged; special care is needed for the chip enable control signal, as not all stages shift under the same conditions.

Another path shaded in the DFD in Figure 23.2 is the decryption $D_1 \rightarrow \text{XOR} \rightarrow O_1$. This path is also shown in the DFD in Figure 23.3, highlighting the absorbing during encryption (phase(IV.)): the same XOR gate is used for both decrypting and encrypting. The other absorbing clock cycles for initialization, processing associated data, and finalization (phase(III.)), are similar to the cycle shown in the DFD in Figure 23.3 and not shown in a separate DFD of their own. The only difference is that in phase (III.), the output XORs and o_data are not used.

¹ all registers are 7 bits wide

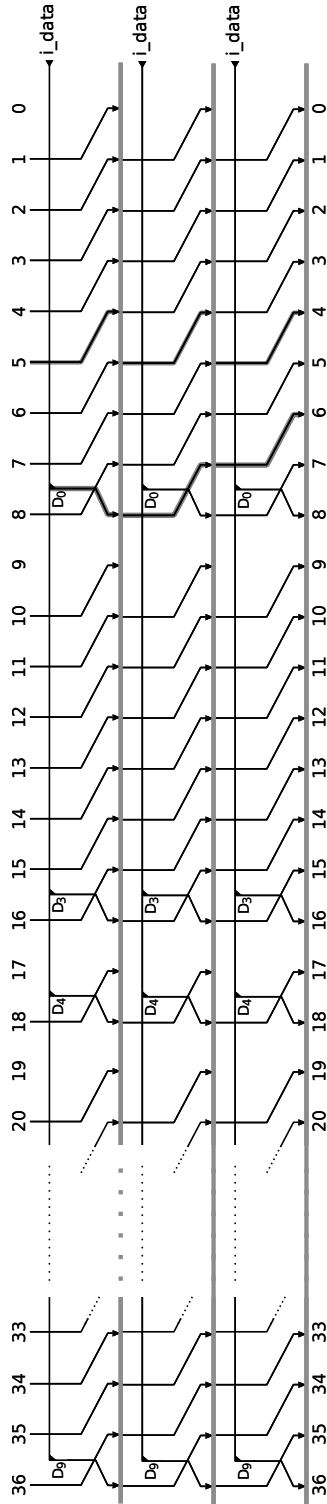


Figure 23.1: Dataflow diagram for loading, showing 3 clock cycles and the path $D_0 \rightarrow S_8 \rightarrow S_7 \rightarrow S_6$ and $S_{11} \rightarrow S_{10}$ shaded

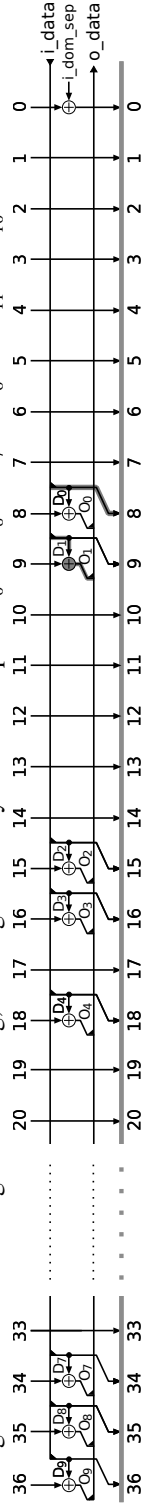


Figure 23.2: Dataflow diagram for replacing (1 clock cycle only), the paths $D_0 \rightarrow S_8$ and $D_1 \rightarrow XOR \rightarrow O_1$ shaded

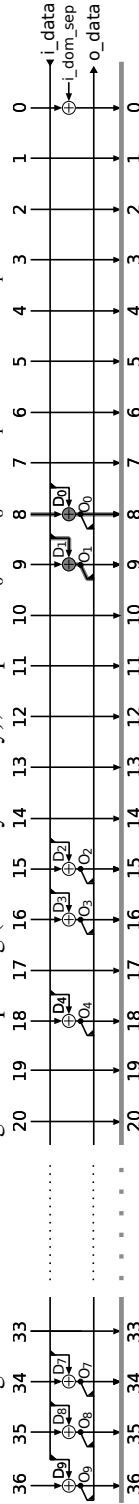


Figure 23.3: Dataflow diagram for absorbing (1 clock cycle only), and the path $D_1 \rightarrow XOR \rightarrow O_1$ and $S_8 \rightarrow XOR \rightarrow S_8$ shaded

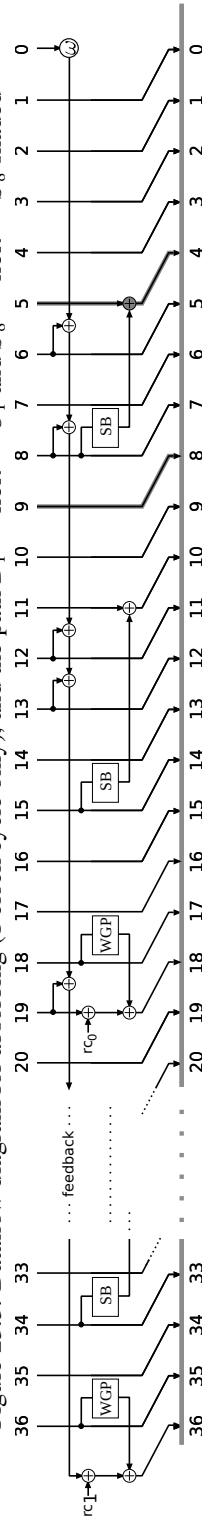


Figure 23.4: Dataflow diagram for permutation (1 clock cycle only), and the paths $S_5 \rightarrow S_4$ and $S_9 \rightarrow S_8$ shaded

Figure 23.4 shows the DFD for one round of permutation (phase (II.)). This clock cycle will be repeated $111\times$. The DFD shows the feedback, the SB, and the WGP components. The shaded path $S_5 \rightarrow \text{XOR} \rightarrow S_4$, compared to the path $S_5 \rightarrow S_4$ in the loading DFD in Figure 23.1, indicates a multiplexer is needed to turn off the SB during the loading. This is also required during the tag extraction (phase (VI.)), not shown in a DFD. The shaded $S_9 \rightarrow S_8$ path in the permutation DFD and the shaded $S_8 \rightarrow \text{XOR} \rightarrow S_8$ path in the absorbing DFD in Figure 23.3 also requires a multiplexer.

The just identified multiplexers are shown in the circuit schematic in Figure 23.5 for stages S_0, \dots, S_{10} . The grey line represents the path during WAGE permutation (phase(II.)). The additional hardware for the entire `wage_1fsr`, which is required to support the permutation in the mode, is listed below, with examples referring to Figure 23.5 and to the six phases.

- For each data input D_k , there is a corresponding 7-bit data output O_k ; Figure 23.5 shows D_1, O_1 and D_0, O_0 .
- 10 XOR gates must be added to the S_r stages to accommodate absorbing during initialization, processing associated data and finalization (phase (III.)), absorbing during encryption (phase (IV.)), and replacing during decryption (phase (V.)). These XORs are shown at stages S_9, S_8 in Figure 23.5).
- 10 multiplexers to switch between absorbing and normal operation. Figure 23.5 shows the **Amux1** at S_9 and the **Amux0** at S_8 . They choose whether to shift in data from the previous stage (phases (I.), (II.), and (VI.)) or to absorb new data into the S_r stages, while the remaining stages hold their previous values (phases (III.) and (IV.)). Loading behaviour is shown in DFD in Figure 23.1: the data is shifted $S_{10} \rightarrow S_9$, hence the left path through **Amux1** is active. The absorbing behaviour is shown in the DFD in Figure 23.3: the XORed value is passed along on the right path through **Amux1**.
- An XOR and a multiplexer are needed to add the domain separator into the internal state (**Amux** at S_0 in Figure 23.5). The XOR is shown in the DFDs in Figures 23.2 and 23.3: the S_0 receives its value through the right path of **Amux** in Figure 23.5.
- To replace the contents of the S_r stages, 10 multiplexers are added. They allow switching between replacing (phase (V.)) and all other phases. An example of a replace multiplexer is **Rmux1** at stage S_9 in Figure 23.5. The replacing DFD in Figure 23.2 shows $D_1 \rightarrow S_9$, i.e., the right path through **Rmux1**.
- Instead of additional multiplexers for loading, the existing **Rmuxk**, $k = 9, 5, 4, 3, 0$, multiplexers are now controlled by `replace` or `load` and labelled **RLmuxk**. An example is **RLmux0** on S_8 in Figure 23.5. Clock cycle 1 in the loading DFD in Figure 23.1 and the replacing DFD in Figure 23.2 show the same path $D_0 \rightarrow S_8$: the active right path through **RLmux0** disconnects this (loading) region from S_9 , as can easily be seen in the loading DFD in Figure 23.1.
- During phases (III.), (IV.) and (V.) all non-input stages must keep their previous values, hence an enable signal `1fsr_en` is needed.

- Three 7-bit AND gates to turn off the inputs D_6 , D_3 and D_1 (see AND at D_1 in Figure 23.5). The output O_1 is used for both the tag extraction (phase (VII.)), as well as ciphertext and plaintext (phases (IV.) and (V.)). The AND gates allow to turn off the input for tag extraction.
- 4 multiplexers are needed to turn off the SB during loading and tag extraction (SBmux at S_4). The non-linear inputs are used only during permutation (phase (II.)). The permutation DFD in Figure 23.4 shows the path $S_5 \rightarrow \text{XOR} \rightarrow S_4$, which is the lower path through the SBmux in Figure 23.5. The upper path allows normal shifting of the LFSR, i.e., $S_5 \rightarrow S_4$, as shown in the loading DFD in Figure 23.1. The `lfsr_en` ensures the behaviour shown in DFDs in Figures 23.3 and 23.2, namely the stage will keep its value, i.e., $S_4 \rightarrow S_4$.

Control signals for multiplexers

The extra circuitry described above (and shown in Figure 23.5) needs the following control signals, which are set by the FSM:

- for $Rmuxk$, $RLmuxk$, and $Amuxk$ multiplexer control: `load`, `absorb`, and `replace`. The control signal is always interpreted as follows: a value of 0 denotes the left input to the mux and a value of 1 the right input.
- for $SBmux$ multiplexer control: `sb_off`. A signal value of 0 selects the bottom mux input, and the value 1 the top mux input.
- for the AND gate: `is_tag`

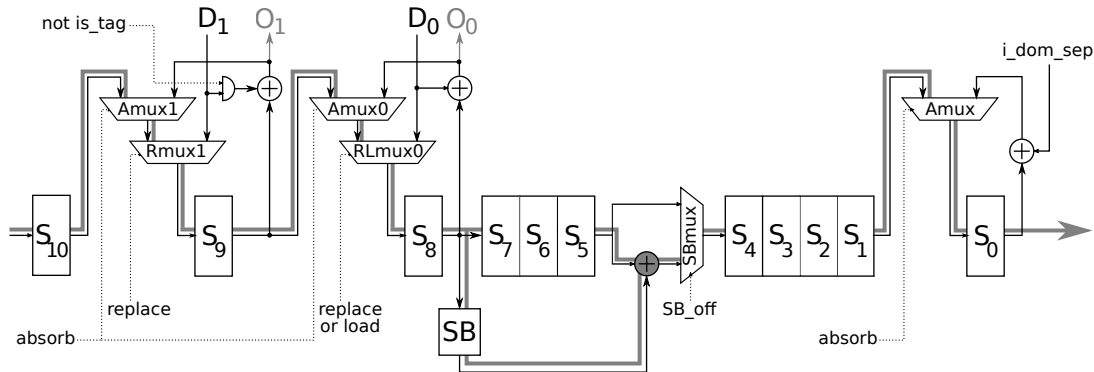


Figure 23.5: Small region of WAGE permutation: stages S_0, \dots, S_{10} with multiplexers, XOR and AND gates for the sponge mode

The control signals are listed in Table 23.1. A final multiplexer is needed to decide whether or not the O_k outputs are sent to the environment. Instead of showing this multiplexer, an extra “output” column is included in Table 23.1. The value of control signals is determined by the interface signals `i_mode` and `i_dom_sep`, and the current phase. These interface signals are used by the FSM to generate the datapath control signals. The - in the table means “dont care”.

Interface signals		Phase	Output	Datapath control					
i_mode	i_dom_sep			load	sb_off	is_tag	absorb	replace	
-	--	loading	I	×	1	1	0	0	0
-	--	permutation	II	×	0	0	0	0	0
-	00	absorbing during: initialization	III	×	0	0	0	1	0
-	01	processing of associated data							
-	00	finalization							
0	10	encryption (absorbing)	IV	✓	0	0	0	1	0
1	10	decryption (replacing)	V	✓	0	0	0	0	1
-	--	tag extraction	VI	✓	0	1	1	0	0

Table 23.1: Control table for WAGE

23.3 The WAGE datapath and the FSRtoVHDL package

The FSRtoVHDL package was successful at implementing WG stream cipher datapaths and the Grain datapath. The WAGE datapath revealed some drawbacks. The WGP component of WAGE can be implemented using constant arrays, as was shown in Example 11.2.5 in Section 11.2. The first problem encountered is the SB. Both the SB and the LFSR are too complex to be implemented using FSRtoVHDL.

Modelling SB with FSR and FSRtoVHDL

Table 23.2 shows the SB fraction from Table 3.1. The SB looks simple, both R and Q need only AND and NOT gates, and the last layer only NOT gates. Since AND and NOT can be represented as multiplication and modulo 2 addition of constant 1 in \mathbb{F}_2 arithmetic, they can also easily be represented as a monomial. For example, $\bar{x}_3 \oplus (x_5 \wedge x_6)$ becomes $1 + x_3 + x_5x_6$.

To model the SB as an FSR object, 19 FILFUN elements are needed: 3 FILFUNs for each layer of R , followed by 3 FILFUNs for Q , and a final FILFUN for the second NOT gate. For example, bit 1 of R and bit 0 of Q' can be modelled with $1 + x_0 + x_1x_2$, as shown in the Implementation detail 22.3.1. Note the modification to Q : the first NOT gate in the last layer was added to Q , hence the notation Q' . Special care is needed when selecting the appropriate inputs, as is shown in the comments. The indices in the comments refer to a 7-bit vector from a previous layer.

Implementation detail 22.3.1

```
R1 := FILFUN(GF(2), 1+x_0+x_1*x_2); # 1 + x_3 + x_5*x_6 # [3,5,6]
Q0 := FILFUN(GF(2), 1+x_0+x_1*x_2); # x_0+x_2*x_3 ADD NOT # [1,3,4]
```

Not only is the process tedious, special care must be taken when using the spreadsheet template to enter the permutation P correctly. Furthermore, as the FSRtoVHDL package simply enumerates the FSRs in the design, the process becomes very error prone. As all the FSRs in the design must be added at once, and there are 4 SB in WAGE, using FSR and FSRtoVHDL to implement the SB is infeasible.

SB Q	$Q(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \rightarrow (x_0 \oplus (x_2 \wedge x_3), x_1, x_2, \bar{x}_3 \oplus (x_5 \wedge x_6), x_4, \bar{x}_5 \oplus (x_2 \wedge x_4), x_6)$
SB P	$P(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \rightarrow (x_6, x_3, x_0, x_4, x_2, x_5, x_1)$
SB R	$R(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \rightarrow (x_6, \bar{x}_3 \oplus (x_5 \wedge x_6), x_0 \oplus (x_2 \wedge x_3), x_4, x_2, \bar{x}_5 \oplus (x_2 \wedge x_4), x_1)$
SB	$(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \leftarrow R^5(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$
	$(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \leftarrow Q(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$
	$(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \leftarrow (\bar{x}_0, x_1, \bar{x}_2, x_3, x_4, x_5, x_6)$

Table 23.2: Specification parameters of WAGE SB

Modelling LFSR with FSR and FSRtoVHDL

Recall from Subsection 8.2 how to model the WAGE state using the FSR package: the state is fragmented into a collection of short LFSRs without feedback, the LFSR feedback is modelled as a FILFUN, and some stages are left as standalone GAP variables:

[s36, s35, s34, s33_30, s29, s28, s27, s26_24, s23_19, s18, s17, s16, s15, s14_11, s10, s9, s8, s7_5, s4_1, s0]

To achieve implementation of registers for the standalone stages, they must be modelled as FSR objects as well, e.g., $s8 := \text{LFSR}(F, y)$ and $s0 := \text{LFSR}(F, y)$. The two LFSRs already shown in Subsection 8.2 are $s7_5 := \text{LFSR}(F, y^3)$ and $s4_1 := \text{LFSR}(F, y^4)$. This process results in 20 LFSR objects that are added to the spreadsheet template by the FSRtoVHDL *Manager* function. Followed by the 10 data inputs and outputs D_i, O_i and the multiplexers to accommodate the mode (Section 23.2), LFSR is not suitable for FSRtoVHDL implementation either.

Future work

The CIRCUIT package has a placeholder `complex` as a future datapath instruction. Among other, it will allow unrolled datapaths, which will significantly simplify the implementation of Sboxes. Another feature, currently not supported by the CIRCUIT package, is implementation of Boolean gates beyond the AND, XOR, and NOT set. The complex datapaths will make use of the GAP syntax trees, announced for the next GAP version. A different solution is available in the GUI based RunFein [174]. RunFein is a framework for block ciphers with access to a library of cryptographic kernels, which can be parameterized by the user and selected layer by layer.

The problem of a large spreadsheet template is an interface issue. The FSRtoVHDL package can be extended to support a clustered setup file and hierarchical *Manager* function, which would allow generating datapaths independently and incrementally building new modules. This feature would allow for the implementation of the SB, and address the problem of complexity of the spreadsheet template.

To solve the problem of fragmentation of the WAGE LFSR, the structure of the FSR package has to be extended with a new type of extended (N)LFSR objects. They would have a set of control flags for each stage of the FSR to decide if the stage:

- has its own load input
- has its own chip enable
- has its own external step input
- is an output

This approach would allow the WAGE LFSR feedback to be implemented as a part of the LFSR instead of as a separate FILFUN. Furthermore, the absorbing stages with data inputs and outputs could be updated, i.e., clocked, independently. An external step input for each stage would enable to plug in the nonlinear components directly. Each stage can already be used as an output stage, so the last item is not an extension.

23.4 Summary

This chapter presented the hardware design of the WAGE datapath. The behaviour analysis using the DFDs outlines the similarities and reuse of hardware. It explains the additional hardware needed to support the permutation in the mode, and the reuse of multiplexers and data inputs and outputs. The chapter concluded with an applicability analysis of the FSRtoVHDL package to the WAGE datapath. This case study revealed some drawbacks of the automation framework and proposed possible solutions with extensions to the FSR, FSRtoVHDL, and CIRCUIT packages.

Part VII

Conclusion

Chapter 24

Conclusion and future work

Summary and conclusion

This thesis presented a novel framework for the synthesis of arbitrary datapaths over arbitrary finite fields. The framework is implemented in GAP from the ground-up. In introduction a design flow with four phases, *the algorithm design*, *the architectural decisions*, *the automated design generation*, and *the design space exploration* was described and shown in Figure 1.1. Figure 24.1 shows design flow with solution, i.e., the design flow with the developed automation framework. Figure 24.1 was already presented in detail in Section 17.1.

The automation framework covers two parts of the design flow: *the architectural decisions* and *the automated design generation*. There are two different paths through the *architectural decisions – automated design generation*, one specialized for FSR-based ciphers and one specialized for arbitrary circuits over arbitrary finite fields.

At the heart of *the architectural decisions* is the package FFCSA (Finite Field Construction, Search and Algorithms). The purpose of the FFCSA package is twofold: (i.) it enables design space exploration by providing various constructions and search capabilities, and (ii.) it can generate the expressions for the synthesis of submodules.

In the light of NIST Lightweight Cryptography Project (LWC), this work focuses mainly on small finite fields. An important part of this thesis is WAGE, a hardware oriented authenticated encryption scheme, which is currently a round 2 candidate in the ongoing NIST LWC standardization process. The automation framework was used to aid the parameter selection during the design of WAGE algorithm. The parameter selection for WAGE was aimed at balancing the security and hardware implementation area, using hardware implementation results for many design decisions, for example field size, representation of field elements, etc. The design space exploration for the FSRtoVHDL package was demonstrated on both WAGE and WG cipher family. The latter was also the motivation for the design of the FSR and FSRtoVHDL packages.

The synthesis for FSR-based datapaths is provided by the FSR package and its sister package FSRtoVHDL. It relies on their modelling as a collection of FSRs and on structural similarities between the FSR objects. The LFSR and NLFSR GAP objects differ only in the degree of the multivariate polynomial used to define their feedback. A FILFUN (filtering function) is an FSR object without feedback, shifting, or storing, whose functionality is defined by a multivariate polynomial. The justification for modelling filtering functions as FSR objects is twofold: (i.) filtering functions are similar to (NLFSR) feedback functions, and (ii.) FSRs with output filters are common, hence they will be used together. The FSR-based system is modelled as a collection of FSR objects.

The user input to the framework is the setup file containing FSR equations and top-level interface signals. The framework generates a spreadsheet template, the user completes the configuration for the connections between the FSRs, and FSRtoVHDL writes the entire datapath as VHDL-ready for synthesis by conventional tools, such as Synopsys Design Compiler. The configuration must use the connectors (e.g., merge, select, etc.), which allow users to specify how individual (coordinates of) internal signals are connect. The configuration is used to infer internal signals, external step conditions, and datapath multiplexers. The basic building blocks of this datapath are the FSR submodules.

The classification of the feedback expression is used to determine how to generate the FSR submodule. The FSRtoVHDL package has no means of generating building blocks like extension field multipliers. This limitation is bypassed by generating black-box feedback for NLFSRs and FILFUNs over extension fields. A possible solution is enabling the use of CIRCUIT for the FSR-toVHDL.

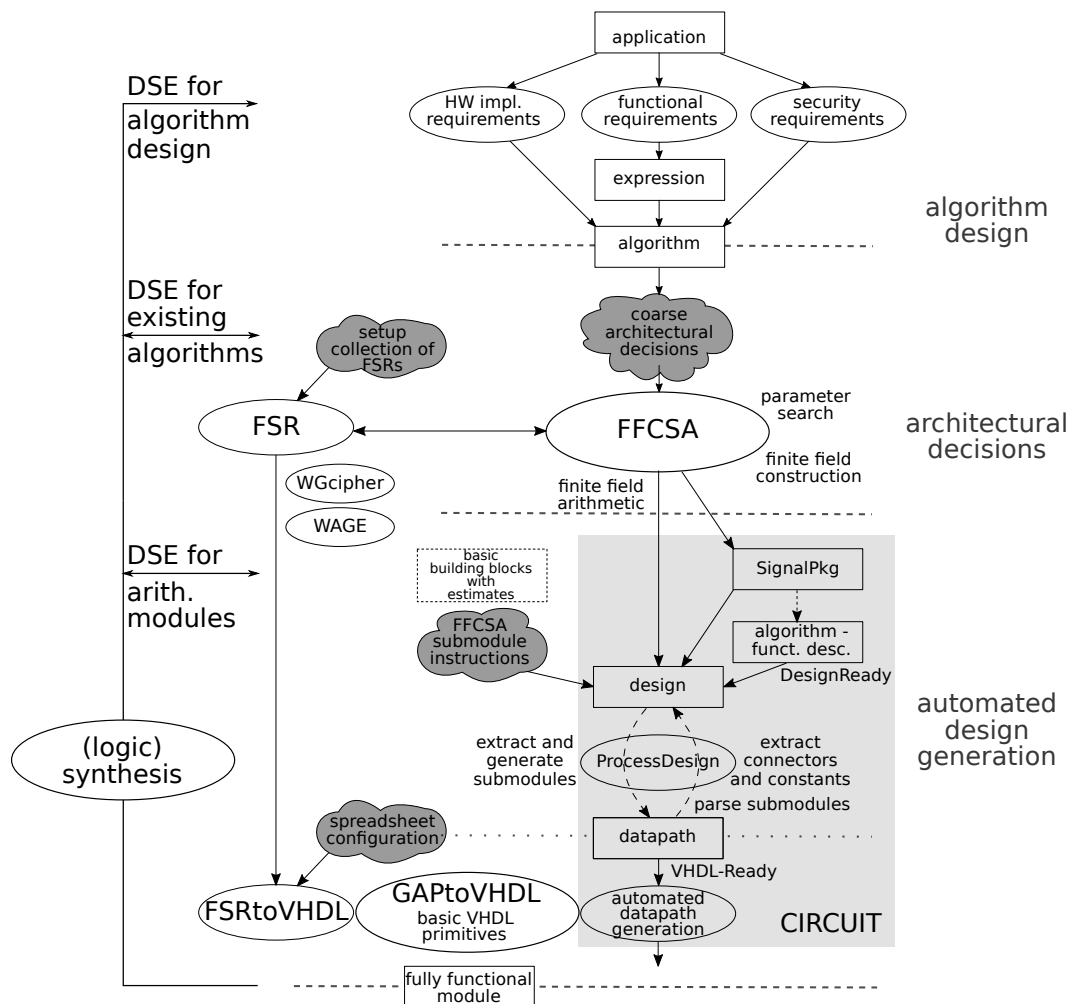


Figure 24.1: Design flow: datapath synthesis (automation framework) and detailed design space exploration

The FSR–FSRtoVHDL tandem was used successfully to generate the datapath for Grain stream cipher, and for the design space exploration of the WG stream cipher family. The analysis of the WAGE datapath revealed two drawbacks. First, while it is possible to generate the datapath of the SB Sbox, it is very impractical, as the spreadsheet configuration file grows out of proportions quickly. However, this drawback is an interface issue. Second, the nonlinear layer of WAGE causes the fragmentation of the LFSR, which must be modelled as a collection of shift registers (modelled as LFSRs without feedback) and filters (modelled as FILFUNs). This drawback can be addressed by adding a new type of extended (N)LFSR objects to the FSR package, with the following features: (i.) each stage can have its own external step input, and (ii.) each stage can be clocked independently of others. This functionality would allow users to plug the nonlinear components directly into the LFSR and provide mode support for absorbing and replacing.

The second *architectural decisions – automated design generation flow*, specialized for arbitrary circuits over arbitrary finite fields, is realized with the CIRCUIT package. User input for datapath synthesis is a mathematical description of the field parameters and the expressions in algebraic normal form, along with submodule instructions to invoke appropriate FFCSA methods. The most distinctive feature of the CIRCUIT package is the encoding of the underlying field structure with the GAP objects SignalDomain, SIGNAL and SignalPkg. SIGNAL allows to view the finite fields as vector spaces, to use “just vectors”, and provides support for tower fields. The SignalPkg, which is an ordered collection of all SIGNALs needed for datapath synthesis, follows the top-down modular approach: the submodules are expected for the subfields. The distinction between a field element and a vector from the corresponding vector space enables the on the fly generation of the submodules. The submodules are generated using the FFCSA package methods. The CIRCUIT package contains two compilation algorithms that gradually refine the design until a VHDL-ready datapath is produced. The last step is the generation of synthesizable VHDL code.

The first compilation algorithm extracts and generates all the submodules needed in the design in a top-down fashion. The second compilation algorithm works bottom-up to bind all the generated submodules to the operations they implement, i.e., to connect them into their parent modules. It also fills in the last bits, such as the declarations of additive constants or connectors between elements/vectors of isomorphic fields in the design.

The framework offers a systematic approach to design space exploration from the perspective of finite field arithmetic. The FFCSA search methods are used to produce a candidate set, and for each candidate, the datapath is synthesized using the automation framework. The design space exploration using the CIRCUIT package was demonstrated on examples of basic building blocks (finite field multiplication and inversion examples), and for an arbitrary datapath over an arbitrary field (the running example expression over $\mathbb{F}_{((2^2)^2)^2}$). The design space exploration for the FRStoVHDL package was demonstrated on the WG cipher family.

In the light of High-Level Synthesis, the proposed framework is described as follows: (i.) the entry point is GAP, (ii.) the framework is domain specific, and is aware of the field structure, (iii.) submodules are generated on the fly instead of retrieved from a library, and (iv.) it synthesizes datapaths without control. The design space exploration is focusing on different field parameters, rather than different hardware optimizations. In particular, this thesis is the first work to provide

general support for tower fields in design automation for finite field arithmetic.

The presented framework resides in the intersection of two areas, namely finite field arithmetic and hardware design, with applications to cryptography. The “math meets hardware” motif is present throughout the entire thesis. The framework allows for rapid design space exploration under different field parameters, with simple user inputs and minimal involvement. For example, not only a single finite field, but several fields can be processed in one go, without any further involvement of the user.

Finally, rapid automated design space exploration and datapath synthesis for cryptography are becoming mandatory, especially in the fast evolving fields, such as Internet of Things.

Future work

The FFCSA methods used to generate the submodules in this thesis rely on the universal algorithm for multiplication. This algorithm was used as a proof of concept. The reduced redundancy Massey-Omura multiplier was used as secondary algorithm on a small finite field, and compared with the circuits generated by the universal algorithm for multiplication. The results show a big room for optimizations. Future work involves adding different algorithms to the FFCSA package. Extending the FFCSA package is an elegant way of extending the reach of the CIRCUIT package. In turn, such extensions enable rapid design space exploration for arithmetic modules, i.e., the basic building blocks, which could yield many interesting results. Possible directions are observing the impact of field parameters on implementation results, comparison of online profiling results to theoretical results, the impact of optimizations performed by the synthesis tools, etc., and finally searching for patterns that can be extended to large finite fields, for which the exhaustive approach is infeasible. Furthermore, the automation framework is not limited to multipliers alone, but supports other arithmetic operations, such as inversions, and arbitrary datapaths. The latter opens the possibility of design space exploration for many algorithms using finite field arithmetic, and is not limited to cryptography alone.

The CIRCUIT package currently supports expressions given in algebraic normal form. Future work includes adding support for Boolean operators, e.g., OR gate, and for more general expressions and distributive expressions in other forms. Another item on the list is adding support for unrolled datapaths, and full support for sequential designs including the control circuitry. The next big step in the CIRCUIT package design is adding the capability of representing the datapath as a directed graph, which opens many new hardware design options, such as pipelining, and hardware optimizations, such as retiming. This addition would open a board area for future research, namely the design space exploration for finite field arithmetic from a hardware perspective.

Bibliography

- [1] D. Ratter, “FPGAs on Mars”, Xcell Journal, no. 50, Fall 2004, Xilinx, Available at <https://www.xilinx.com/publications/archives/xcell/Xcell150.pdf> [Accessed: April 4, 2020]
- [2] L. Tran, “Tiny Microchips Enable Extreme Science”, NASA’s Goddard Space Flight Center, July 2016, Available at <https://phys.org/news/2016-07-tiny-microchips-enable-extreme-science.html> [Accessed: April 4, 2020]
- [3] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.10.2*; 2019, Available at <https://www.gap-system.org>
- [4] M. Robshaw, “The eSTREAM Project”, in *New Stream Cipher Designs: The eSTREAM Finalists*, M. Robshaw, O. Billet, Eds., Lecture Notes in Computer Science, vol. 4986, pp 1-6, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008
- [5] NIST Lightweight Cryptography Standardization Project, January 2017, Available at <https://csrc.nist.gov/Projects/Lightweight-Cryptography> [Accessed: April 4, 2020]
- [6] M. Aagaard, R. AlTawy, G. Gong, K. Mandal, R. Rohit, N. Zidaric, “WAGE: An Authenticated Cipher”, Round 2 Submission to *NIST Lightweight Cryptography Standardization Project*, Available at <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/wage-spec-round2.pdf> [Accessed: April 4, 2020]
- [7] Y. Nawaz, G. Gong, “The WG Stream Cipher”, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/033, 2005, Available at http://www.ecrypt.eu.org/stream/p2ciphers/wg/wg_p2.pdf, [Accessed: April 4, 2020]
- [8] X. Fan, N. Zidaric, M. Aagaard, and G. Gong, “Efficient Hardware Implementation of the Stream Cipher WG-16 with Composite Field Arithmetic”, In Proceedings of the 3rd international workshop on Trustworthy embedded devices (TrustED ’13), ACM, pp. 21–34
- [9] N. Zidaric, “Hardware Implementations of the WG-16 Stream Cipher with Composite Field Arithmetic” MASc thesis, University of Waterloo, Waterloo, ON, CA, 2014

- [10] NIST Lightweight Cryptography Standardization Project - Round 2 Candidates, Announced August 30, 2019, Available at <https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates> [Accessed: April 4, 2020]
- [11] R. Lidl and H. Niederreiter, *Finite fields*, Encyclopedia of Mathematics and its Applications, Vol.20, Cambridge University Press, 1996
- [12] A. Menezes, I. Blake, S. Gao, R. Mullin, S. Vanstone, and T. Yaghoobian, *Applications of Finite Fields*, Boston: Kluwer Academic Publishers, 1993
- [13] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, New York: Springer-Verlag New York, 2004
- [14] S.W. Golomb and G. Gong, *Signal Design for Good Correlation: For Wireless Communication, Cryptography, and Radar*, Cambridge: Cambridge University Press, 2005
- [15] G.L. Mullen and D.Panario, *Handbook of Finite Fields*, Boca Raton, FL: CRC Press, 2013
- [16] G.L. Mullen, C. Mummert, *Finite Fields and Applications*, Providence, RI: American Mathematical Society, 2007
- [17] E. Artin, *Galoissche theorie*, Notre Dame Mathematical Lectures, 2. Aufgabe, Zürich und Frankfurt/Main: Verlag Harri Deutsch, 1968
- [18] F.J. MacWilliams, N.J.A. Sloane, *The Theory of Error-Correcting Codes*, Amsterdam: North-Holland, 1977
- [19] H. Cohen, G. Frey, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, Boca Raton FL: Chapman & Hall/CRC, 2006
- [20] J-P. Deschamps, J.L. Imaña in G. D. Sutter , *Hardware Implementation of Finite-Field Arithmetic*, New York: McGraw-Hill, 2009
- [21] J. Grabmeier, E. Kaltofen, V. Weispfenning, *Computer Algebra Handbook: Foundations, Applications, Systems*, Berlin: Springer-Verlag Berlin and Heidelberg GmbH & Co, 2003
- [22] J. von zur Gathen, J. Gerhard, *Modern Computer Algebra*, Cambridge: Cambridge University Press, 1999
- [23] A. Heck, *Introduction to Maple*, New York: Springer-Verlag New York, 2003
- [24] Maple: <https://www.maplesoft.com/>
- [25] Mathematica: <https://www.wolfram.com/mathematica/>
- [26] A. Sorgatz, S. Wehmeier, “Towards high-performance symbolic computing: using MuPAD as a problem solving environment”, *Mathematics and Computers in Simulation*, vol.: 49, no: 3, August, pp. 235-246, 1999

- [27] MATLAB: <https://www.mathworks.com/>
- [28] SageMath: <http://www.sagemath.org/>
- [29] Sage Reference Manual, Sage 9.1 Reference Manual: Cryptography, Available at <http://doc.sagemath.org/html/en/reference/cryptography/index.html> [Accessed: April 4, 2020]
- [30] N. Galbreath, *Symbolic Linear Feedback Shift Registers*, 2005-08-24, Available at <http://library.wolfram.com/infocenter/MathSource/5717/> [Accessed: April 4, 2020]
- [31] T. Breuer, S. Linton, “The GAP 4 Type System: Organising Algebraic Algorithms”, Proceedings of the 1998 international symposium on Symbolic and algebraic computation - ISSAC 98, pp. 38-45, ACM, 1998
- [32] A. Hulpke, “Using GAP”, tutorial presented at 2000 international symposium on Symbolic and algebraic computation - ISSAC 2000, St. Andrews 2000, Available at <https://www.math.colostate.edu/~hulpke/paper/gap4tut.pdf> [Accessed: April 4, 2020]
- [33] The GAP Package *GAPDoc - A Meta Package for GAP Documentation, Version 1.6.2; 2018*, Available at <https://www.gap-system.org/Packages/gapdoc.html>
- [34] GAP - Reference Manual, Release 4.10.2, 19-Jun-2019, Available at <https://www.gap-system.org/Manuals/doc/ref/chap0.html>
- [35] The GAP Package *Digraphs - Graphs, digraphs, and multidigraphs in GAP, Version 1.0.1; 2019*, Available at <https://gap-packages.github.io/Digraphs/>
- [36] N. Zidaric, M. Aagaard, G. Gong, “Rapid Hardware Design for Cryptographic Modules with Filtering Structures over Small Finite Fields” in: L. Budaghyan, F. Rodríguez-Henríquez, Eds., *Arithmetic of Finite Fields, WAIFI 2018, Lecture Notes in Computer Science*, vol. 11321, pp. 128-145, Springer, Cham
- [37] L.Chen and G. Gong, *Communication System Security*, Boca Raton FL: CRC Press, 2012
- [38] H. Wu, “ACORN: A Lightweight Authenticated Cipher (v1)”, Submission to *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*, 2014, (v3 is in *CAESAR final portfolio*), Available at <http://competitions.cr.yp.to/round1/acornv1.pdf> [Accessed: April 4, 2020]
- [39] CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness, January 2014, Available at <https://competitions.cr.yp.to/caesar.html> [Accessed: April 4, 2020]

- [40] M. Hell, T. Johansson, W. Meier, J. Sönnerup, “Grain-128AEAD - A lightweight AEAD stream cipher”, Submission to *NIST Lightweight Cryptography Standardization Project*, Available at <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/grain-128aead-spec-round2.pdf> [Accessed: April 4, 2020]
- [41] Xilinx, Appl. Note XAPP 052, July 7, 1996 (Version 1.1), Available at https://www.xilinx.com/support/documentation/application_notes/xapp052.pdf [Accessed: April 4, 2020]
- [42] M. A. Hasan, ECE-720 (Topic 2), Lecture notes, “Selected Topics in Cryptographic Computations”, University of Waterloo, Winter 2017
- [43] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, *Handbook of applied cryptography*, Boca Raton FL: CRC Press, 1996
- [44] T. Good and M. Benaissa, “ASIC Hardware Performance”, in *New Stream Cipher Designs: The eSTREAM Finalists*, M. Robshaw, O. Billet, Eds., Lecture Notes in Computer Science, vol. 4986, pp. 267-293, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008
- [45] M. Hell, T. Johansson, A. Maximov, and W. Meier, “The Grain Family of Stream Ciphers”, in *New Stream Cipher Designs: The eSTREAM Finalists*, M. Robshaw, O. Billet, Eds., Lecture Notes in Computer Science, vol. 4986, pp. 179-190, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008
- [46] M. Hell, T. Johansson, W. Meier, “Grain - A Stream Cipher for Constrained Environments”, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/010, 2005, Available at <http://www.ecrypt.eu.org/stream/ciphers/grain/grain.pdf> [Accessed: April 4, 2020]
- [47] C. De Cannière and B. Preneel, “Trivium”, in *New Stream Cipher Designs: The eSTREAM Finalists*, M. Robshaw, O. Billet, Eds., Lecture Notes in Computer Science, vol. 4986, pp. 224-243, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008
- [48] G. Gong, A.M. Youssef, “Cryptographic Properties of the Welch-Gong Transformation Sequence Generators”, in *IEEE Transactions on Information Theory*, vol. 48, no. 11, pp. 2837-2846, November 2002
- [49] Y. Nawaz, G. Gong, “WG: A family of stream ciphers with designed randomness properties”, *Information Sciences*, vol. 178, no. 7, pp. 1903-1916, ACM, 2008
- [50] K. Mandal, G. Gong, X. Fan, M. Aagaard, “Optimal parameters for the WG stream cipher family”, *Cryptography and Communications*, vol. 6, no. 2, pp. 117-135, 2014
- [51] P. Rogaway, “Authenticated-Encryption with Associated-Data”, *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 98-107, ACM, 2002, Available at <https://dl.acm.org/doi/pdf/10.1145/586110.586125> [Accessed: April 4, 2020]

- [52] H. Wu, B. Preneel, “AEGIS: A Fast Authenticated Encryption (v1.1)”, *CAESAR final portfolio*, 2016, Available at <https://competitions.cr.yp.to/round3/aegisv11.pdf> [Accessed: April 4, 2020]
- [53] M. Dworkin, “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC”, NIST Special Publication 800-38D, November, 2007. Available at <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf> [Accessed: April 4, 2020]
- [54] T. Krovetz, P. Rogaway, “OCB (v1.1)”, *CAESAR final portfolio*, 2016, Available at <https://competitions.cr.yp.to/round3/ocbv11.pdf> [Accessed: April 4, 2020]
- [55] M. Ågren, M. Hell, T. Johansson, W. Meier, “Grain-128a: a new version of Grain-128 with optional authentication”, *International Journal of Wireless and Mobile Computing*, vol. 5, no. 1, pp. 48-59, ACM, 2011
- [56] CAESAR call for submissions, January 2014, Available at <https://competitions.cr.yp.to/caesar-call.html> [Accessed: April 4, 2020]
- [57] C. Dobraunig, M. Eichlseder, F. Mendel, M. Schl affer, “Ascon v1.2”, *CAESAR final portfolio*, 2016, Available at <https://competitions.cr.yp.to/round3/asconv12.pdf> [Accessed: April 4, 2020]
- [58] J. Jean, I. Nikolić, T. Peyrin, Y. Seurin, “Deoxys v1.41”, *CAESAR final portfolio*, 2016, Available at <https://competitions.cr.yp.to/round3/deoxysv141.pdf> [Accessed: April 4, 2020]
- [59] E. Andreeva, A. Bogdanov, N. Datta, A. Luykx, B. Mennink, M. Nandi, E. Tischhauser, K. Yasuda, “COLM v1”, *CAESAR final portfolio*, 2016, Available at <https://competitions.cr.yp.to/round3/colmv1.pdf> [Accessed: April 4, 2020]
- [60] “Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process”, *NIST Lightweight Cryptography Standardization Project*, Available at <https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf>
- [61] R. AlTawy, R. Rohit, M. He, K. Mandal, G. Yang, G. Gong, “sLiSCP: Simeck-based permutations for lightweight sponge cryptographic primitives”, in: C. Adams, J. Camenisch, Eds., *Selected Areas in Cryptography, SAC 2017, Lecture Notes in Computer Science*, vol. 10719, pp. 129-150, Springer, Cham
- [62] M.J.S. Smith, *Application-specific integrated circuits*, Reading, MA: Addison-Wesley, 1997
- [63] I. Kuon, J. Rose, “Measuring the Gap Between FPGAs and ASICs”, in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203 - 215, February 2007

- [64] A. Rahman, S. Das, A. Chandrakasan, and R. Reif, “Wiring Requirement and Three-Dimensional Integration Technology for Field Programmable Gate Arrays”, in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 1, pp. 44-54, February 2003
- [65] Pong P. Chu, *RTL Hardware Design Using VHDL: coding for Efficiency, Portability, and Scalability*, Hoboken, N.J: Wiley-IEEE Press, 2006
- [66] K.A. McKay, L. Bassham, M. S. Turan, N. Mouha, “Report on Lightweight Cryptography”, NISTIR 8114, March 2017, Available at <https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf> [Accessed: April 4, 2020]
- [67] L. Batina, J. Lano, N. Mentens, S.B. Örs, B. Preneel, I. Verbauwhede, “Energy, Performance, Area versus Security Trade-offs for Stream Ciphers”, *SASC - The State of the Art of Stream Ciphers*, October 14-15, Brugge, Belgium: Special Workshop hosted by the ECRYPT Network of Excellence, pp. 302-310, 2004
- [68] M.D. Aagaard, G. Gong, R.K. Mota, “Hardware Implementations of the WG-5 Cipher for Passive RFID Tags”, in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2013*, pp. 29-34, June 2013
- [69] S. Kilts, *Advanced FPGA design - Architecture, Implementation, and Optimization*, Hoboken, N.J: Wiley-IEEE Press, 2007
- [70] M. Aagaard, ECE-327/627, Course notes, “Digital Systems Engineering”, University of Waterloo, Winter 2020
- [71] N. Zidaric, M. Aagaard, “Throughput as a defining characteristic in register-transfer-level design”, *to be submitted*
- [72] “IEEE Standard for VHDL Language Reference Manual”, Developed by the Design Automation Standards Committee of the IEEE Computer Society, IEEE Std 1076TM-2019, September 2019
- [73] S. Brown, Z. Vranesic, *Fundamentals of digital logic with VHDL design, 2nd ed.*, New York: McGraw-Hill Higher Education, 2005
- [74] M. C. McFarland, A. C. Parker, R. Camposano, “The high-level synthesis of digital systems”, in *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301-318, February 1990
- [75] D.D. Gajski, L. Ramachandran, “Introduction to High-Level Synthesis”, in *IEEE Design & Test of Computers*, vol. 11, no. 4, pp. 44 - 54, October 1994
- [76] P. Coussy, D.D. Gajski, M. Meredith, A. Takach, “An Introduction to High-Level Synthesis”, in *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8 - 17, August 2009

- [77] M.C. McFarland. 1986. “Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions”. in Proceedings of the 23rd ACM/IEEE Design Automation Conference (DAC), pp. 474-480, June-July 1986
- [78] R. Nane, V.M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis , Y.T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, K. Bertels, “A Survey and Evaluation of FPGA High-Level Synthesis Tools”, in IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 10, pp. 1591 - 1604, October 2016
- [79] S. Lahti, P. Sjövall, J. Vanne and T. D. Hämmäläinen, “Are We There Yet? A Study on the State of High-Level Synthesis”, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 5, pp. 898-911, May 2019
- [80] M.C. McFarland, T.J. Kowalski. “Incorporating bottom-up design into hardware synthesis”, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 9, no. 9, pp. 938-950, September 1990
- [81] E.D. Lagnese, D.E. Thomas, “Architectural partitioning for system level synthesis of integrated circuits”, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 10, no. 7, pp. 847-860, July 1991
- [82] P.-C. Kao, C.-K. Hsieh, A.-H. Wu, “An RTL design-space exploration method for high-level applications”, Proceedings of the 2001 Asia and South Pacific Design Automation Conference, ASP-DAC 2001, pp. 162-167, January 2001, Available at <https://dl.acm.org/doi/pdf/10.1145/370155.370313> [Accessed: April 4, 2020]
- [83] S. Liu, F. C. Lau, B. C. Schäfer, “Accelerating FPGA Prototyping through Predictive Model-Based HLS Design Space Exploration”, 2019 56th ACM/IEEE Design Automation Conference (DAC), pp. 1-6, June 2019
- [84] M. M. Ziegler, H. Liu, G. Gristede, B. Owens, R. Nigaglioni, L. P. Carloni, “A synthesis-parameter tuning system for autonomous design-space exploration”, 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1148-1151, March 2016
- [85] L. Batina, J. Lano, N. Mentens, S. B. Örs, B. Preneel, I. Verbauwhede, “Energy, Performance, Area Versus Security Trade-offs for Stream Ciphers”, in SACS - The State of the Art of Stream Ciphers: Workshop Record, Brugge, Belgium, October 2004, pp. 302–310, Available at: <http://www.ecrypt.eu.org/stvl/sasc/record.html> [Accessed: April 4, 2020]
- [86] S. Kumar, K. Lemke, C. Paar, “Some Thoughts about Implementation Properties of Stream Ciphers”, in SACS - The State of the Art of Stream Ciphers: Workshop Record, Brugge, Belgium, October 2004, pp. 311–319, Available at: <http://www.ecrypt.eu.org/stvl/sasc/record.html> [Accessed: April 4, 2020]

- [87] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, L. Uhsadel, “A Survey of Lightweight-Cryptography Implementations”, in *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 522-533, November-December 2007
- [88] A. Bogdanov, G. Leander, L.R. Knudsen, C. Paar, A. Poschmann, M.J. Robshaw, Y. Seurin, C. Vikkelsoe, “PRESENT - An Ultra-Lightweight Block Cipher”. in: P. Paillier, I. Verbauwhede, Eds., *Cryptographic Hardware and Embedded Systems – CHES 2007*, CHES 2007, *Lecture Notes in Computer Science*, vol. 4727, pp. 450-466, Springer, Berlin, Heidelberg, 2007
- [89] C. Rolfes, A. Poschmann, G. Leander, C. Paar, “Ultra-Lightweight Implementations for Smart Devices – Security for 1000 Gate Equivalents”, in: G. Grimaud, FX. Standaert, Eds., *Smart Card Research and Advanced Applications, CARDIS 2008*, *Lecture Notes in Computer Science*, vol 5189, pp. 89-103, Springer, Berlin, Heidelberg, 2008
- [90] D. Engels, X. Fan, G. Gong, H. Hu, E.M. Smith, “Hummingbird: Ultra-Lightweight Cryptography for Resource-Constrained Devices”, in: R. Sion *et al.* Eds., *Financial Cryptography and Data Security, FC 2010*, *Lecture Notes in Computer Science*, vol 6054., pp. 3-18, Springer, Berlin, Heidelberg, 2010
- [91] M. Xiao, X. Shen, Y. Yang, J. Wang, “Low power implementation of hummingbird cryptographic algorithm for RFID tag”, 2010 10th IEEE International Conference on Solid-State and Integrated Circuit Technology, Shanghai, 2010, pp. 581-583
- [92] A. Moradi, A. Poschmann, S. Ling, C. Paar, H. Wang, “Pushing the Limits: A Very Compact and a Threshold Implementation of AES”, in: K.G. Paterson, Eds., *Advances in Cryptology – EUROCRYPT 2011*, *EUROCRYPT 2011*, *Lecture Notes in Computer Science*, vol 6632, pp. 69-88, Springer, Berlin, Heidelberg, 2011
- [93] S. Mathew, S. Satpathy, V. Suresh, M. Anders, H. Kaul, A. Agarwal, S. Hsu, G. Chen, R. Krishnamurthy, “340 mV–1.1 V, 289 Gbps/W, 2090-Gate NanoAES Hardware Accelerator With Area-Optimized Encrypt/Decrypt $GF(2^4)^2$ Polynomials in 22 nm Tri-Gate CMOS”, in *IEEE Journal of Solid-State Circuits*, vol. 50, no. 4, pp. 1048-1058, April 2015
- [94] L. Batina, A. Das, B. Ege, E.B. Kavun, N. Mentens, C. Paar, “Dietary Recommendations for Lightweight Block Ciphers: Power, Energy and Area Analysis of Recently Developed Architectures”, in: M. Hutter, JM. Schmidt, Eds., *Radio Frequency Identification, RFIDSec 2013*, *Lecture Notes in Computer Science*, vol. 8262, pp. 103 - 112, Springer, Berlin, Heidelberg, 2013
- [95] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, L. Wingers, “The SIMON and SPECK Families of Lightweight Block Ciphers”, 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp.1-6, July 2015

- [96] G. Yang, B. Zhu, V. Suder, M. Aagaard, G. Gong, “The Simeck Family of Lightweight Block Ciphers”, in: T. Güneysu, H. Handschuh, Eds., *Cryptographic Hardware and Embedded Systems – CHES 2015*, CHES 2015, Lecture Notes in Computer Science, vol. 9293, pp. 307-329, Springer, Berlin, Heidelberg, 2015
- [97] S. Banik, S. K. Pandey, T. Peyrin, S. M. Sim, Y. Todo, Y. Sasaki, “GIFT: A Small Present”, in: W. Fischer, N. Homma, Eds., *Cryptographic Hardware and Embedded Systems – CHES 2017*, CHES 2017, Lecture Notes in Computer Science, vol 10529, pp 321-345, Springer, Cham, 2017
- [98] W. Diehl, F. Farahmand, A. Abdulgadir, J.P. Kaps, K. Gaj, “Face-off between the CAESAR Lightweight Finalists: ACORN vs. Ascon”, 2018 International Conference on Field-Programmable Technology (FPT), Naha, Okinawa, Japan, 2018, pp. 330-333
- [99] H. Groß, E. Wenger, C. Dobraunig and C. Ehrenhöfer, "Suit up! – Made-to-Measure Hardware Implementations of ASCON", 2015 Euromicro Conference on Digital System Design, Funchal, 2015, pp. 645-652
- [100] J.W. Yu, M.D. Aagaard, “Benchmarking and Optimizing AES for Lightweight Cryptography on ASICs”, NIST LWC workshop 2019, Available at <https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2019/documents/papers/benchmarking-and-optimizing-aes-for-lwc-on-asics-lwc2019.pdf> [Accessed: April 4, 2020]
- [101] P. Hamalainen, T. Alho, M. Hannikainen and T. D. Hamalainen, “Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core”, 9th EUROMICRO Conference on Digital System Design (DSD’06), Dubrovnik, 2006, pp. 577-583
- [102] A. Biryukov, L. Perrin, “State of the Art in Lightweight Symmetric Cryptography”, Cryptology ePrint Archive, Report 2017/511, Available at <https://eprint.iacr.org/2017/511> [Accessed: April 4, 2020]
- [103] G. Gong, “Securing Internet-of-Things”, in: N. Zincir-Heywood, G. Bonfante, M. Debbabi, J. Garcia-Alfaro, Eds., *Foundations and Practice of Security, FPS 2018*, Lecture Notes in Computer Science, vol 11358, pp. 3-16, Springer, Cham, 2019
- [104] F.K. Gürkaynak, 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”, eSTREAM, ECRYPT Stream Cipher Project, Report 2006/015, (2006) Available at <http://www.ecrypt.eu.org/stream/papersdir/2006/015.pdf> [Accessed: April 4, 2020]
- [105] D. Hwang, M. Chaney, S. Karanam, N. Ton, K. Gaj, “Comparison of FPGA-Targeted Hardware Implementations of eSTREAM Stream Cipher Candidates”, in *SACS - The State of the Art of Stream Ciphers: Workshop Record*, Lausanne, Switzerland, February 2008,

- pp.151-162, Available at http://ece.gmu.edu/~kgaj/publications/conferences/GMU_SASC_2008.pdf [Accessed: April 4, 2020]
- [106] P. Bulens, K. Kalach, F. Standaert, J. Quisquater, “FPGA Implementations of eSTREAM Phase-2 Focus Candidates with Hardware Profile”, eSTREAM, ECRYPT Stream Cipher Project, Report 2007/024 (SASC 2007), Available at <http://www.ecrypt.eu.org/stream/papersdir/2007/024.pdf> [Accessed: April 4, 2020]
- [107] K. Gaj, G. Southern, R. Bachimanchi, “Comparison of hardware performance of selected Phase II eSTREAM candidates”, eSTREAM, ECRYPT Stream Cipher Project, Report 2007/026 (SASC 2007), Available at <http://www.ecrypt.eu.org/stream/papersdir/2007/026.pdf> [Accessed: April 4, 2020]
- [108] T. Good, W. Chelton, M. Benaissa, “Review of stream cipher candidates from a low resource hardware perspective”, eSTREAM, ECRYPT Stream Cipher Project, Report 2006/016, Available at <http://www.ecrypt.eu.org/stream/papersdir/2006/016.pdf> [Accessed: April 4, 2020]
- [109] P. Kitsos, N. Sklavos, G. Provelengios, A.N. Skodras, “FPGA-based performance analysis of stream ciphers ZUC, Snow3g, Grain V1, Mickey V2 Trivium and E0”, *Microprocessors & Microsystems*, vol. 37, no. 2, pp. 235-245, March, 2013
- [110] M. Rogawski, “Hardware evaluation of eSTREAM Candidates: Grain, Lex, Mickey128, Salsa20 and Trivium”, eSTREAM, ECRYPT Stream Cipher Project, Report 2007/025 (SASC 2007), Available at <http://www.ecrypt.eu.org/stream/papersdir/2007/025.pdf> [Accessed: April 4, 2020]
- [111] H. El-Razouk, A. Reyhani-Masoleh, 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> [Accessed: April 4, 2020]
- [112] H. El-Razouk, A. Reyhani-Masoleh, G. Gong. “New Hardware Implementations of WG(29;11) and WG-16 Stream Ciphers Using Polynomial Basis,” in *IEEE Transactions on Computers*, vol. 64, no. 7, pp. 2020-2035, July 2015
- [113] G. Yang, X. Fan, M. Aagaard, G. Gong, “Design Space Exploration of the Lightweight Stream Cipher WG-8 for FPGAs and ASICs”, in *Proceedings of The 8th Workshop on Embedded Systems Security (WESS’13)*, ACM, Article No. 8, September 2013, Available at <https://dl.acm.org/doi/pdf/10.1145/2527317.2527325> [Accessed: April 4, 2020]
- [114] X. Fan, 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> [Accessed: April 4, 2020]

- [115] N. Zidaric, M. Aagaard, G. Gong, “Hardware Optimizations and Analysis for the WG-16 Cipher with Tower Field Arithmetic”, in IEEE Transactions on Computers, vol. 68, no. 1, pp. 67-82, Jan. 2019
- [116] A. Karatsuba and Y. Ofman, “Multiplication of Multidigit Numbers on Automata”, Soviet Physics-Doklady, vol. 7, no. 7, pp 595-596, January 1963
- [117] M.-J.O. Saarinen, D.Engels “A Do-It-All-Cipher for RFID: Design Requirements (Extended Abstract)”, Cryptology ePrint Archive, Report 2012/317, 2012, Available at <https://eprint.iacr.org/2012/317.pdf> [Accessed: April 4, 2020]
- [118] Z. Liu, Q. Zhang, C. Ma, C. Li, J. Jing, “HPAZ: A High-throughput Pipeline Architecture of ZUC in Hardware”, 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, 2016, pp. 269-272
- [119] M. Aagaard, G. Gong, K. Mandal, M. Sattarov, N. Zidaric, “WG-lite Design Space Exploration”, *work in progress*
- [120] M. Sattarov, “Hardware Implementations of the Lightweight Welch-Gong Stream Cipher Family using Polynomial Bases”, MASC thesis, University of Waterloo, Waterloo, ON, CA, 2019
- [121] D.H. Green, I.S. Taylor, “Irreducible polynomials over composite Galois fields and their applications in coding techniques”, in Proceedings of the Institution of Electrical Engineers, vol.121, no.9,pp. 935-939, September 1974
- [122] T. Itoh, S. Tsuji, “A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases”, Information and Computation, vol. 78, no. 3, pp. 171-177, September 1988
- [123] I.S. Hsu, T.K. Truong, I.S. Reed, N. Glover, “A VLSI architecture for performing finite field arithmetic with reduced table lookup”, Linear Algebra and its Applications, vol. 98, pp. 249-262, January 1988
- [124] M. Morii and M. Kasahara, “Efficient construction of gate circuit for computing multiplicative inverses over $GF(2^m)$ ”, Transactions of the IEICE, vol. E72, no. 1, pp. 37-42, January 1989
- [125] 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
- [126] 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
- [127] 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

- [128] 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
- [129] C. Paar, “Low complexity parallel multipliers for Galois fields $GF((2^n)^4)$ based on special types of primitive polynomials”, Proceedings of 1994 IEEE International Symposium on Information Theory, pp. 98, Trondheim, Norway, June-July 1994
- [130] C. Paar, “A new architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields”, in IEEE Transactions on Computers, vol. 45, no. 7, pp. 856-861, July 1996
- [131] 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, 1994
- [132] C. Paar, P. Soria-Rodriguez, “Fast Arithmetic Architectures for Public-Key Algorithms over Galois Fields $GF((2^n)^m)$ ”, in: W. Fumy, Eds., Advances in Cryptology – EUROCRYPT ’97, EUROCRYPT 1997, Lecture Notes in Computer Science, vol. 1233, pp. 363-378, Springer, Berlin, Heidelberg, 1997
- [133] J. Guajardo, C. Paar, “Itoh-Tsuji Inversion in Standard Basis and Its Application n Cryptography and Codes”, Designs, Codes and Cryptography, vol. 25, no. 2, pp. 207-216, February 2002
- [134] G. Harper, A. Menezes, S. Vanstone, “Public-Key Cryptosystems with Very Small Key Lengths”, in: R.A. Rueppel, Eds., Advances in Cryptology - EUROCRYPT ’92, EUROCRYPT 1992. Lecture Notes in Computer Science, vol 658, pp. 163-173 , May 1992, Springer, Berlin, Heidelberg, 1992
- [135] E. Savas, 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> [Accessed: April 4, 2020]
- [136] C.K. Koc, B. Sunar, “Low-Complexity Bit-Parallel Canonical and Normal Basis Multipliers for a Class of Finite Fields”, in IEEE Transactions on Computers, vol. 47, no. 3, pp. 353-356, March 1998
- [137] B. Sunar, C.K. Koc, “An efficient optimal normal basis type II multiplier”, in IEEE Transactions on Computers, vol. 50, no. 1, pp. 83-87, January 2001
- [138] B. Sunar, E. Savas, C.K. Koc, “Constructing Composite Field Representations for Efficient Conversion”, in IEEE Transactions on Computers, vol. 52, no. 11, pp. 1391-1398, November 2003

- [139] 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 <https://www.gsma.com/aboutus/wp-content/uploads/2014/12/snow3gspec.pdf> [Accessed: April 4, 2020]
- [140] V. Rijmen, “Efficient Implementation of the Rijndael S-box”, Available at <http://www.networkdls.com/Articles/sbox.pdf> [Accessed: April 4, 2020]
- [141] A. Rudra, P.K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, P. Rohatgi, “Efficient Rijndael Encryption Implementation with Composite Field Arithmetic”, in: Ç.K. Koç, Eds., Cryptographic Hardware and Embedded Systems – CHES 2001, CHES 2001, Lecture Notes in Computer Science, vol. 2162, pp. 171-184, Springer, Berlin, Heidelberg, 2007
- [142] A. Satoh, S. Morioka, K. Takano, S. Munetoh, “A Compact Rijndael Hardware Architecture with S-Box Optimization”, in: C. Boyd, Eds., Advances in Cryptology – ASIACRYPT 2001, ASIACRYPT 2001, Lecture Notes in Computer Science, vol. 2248, pp. 239-254 Springer, Berlin, Heidelberg, 2001
- [143] N. Mentens, L. Batina, B. Preneel, I. Verbauwhede, “A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-Box”, in: A. Menezes, Eds., Topics in Cryptology – CT-RSA 2005, CT-RSA 2005, Lecture Notes in Computer Science, vol. 3376, pp. 323-333, Springer, Berlin, Heidelberg, 2005
- [144] D. Canright, “A Very Compact S-Box for AES”, in: J.R. Rao, B. Sunar, Eds., Cryptographic Hardware and Embedded Systems – CHES 2005, CHES 2005, Lecture Notes in Computer Science, vol. 3659, pp. 441-455, Springer, Berlin, Heidelberg, 2005
- [145] S. Nikova, V. Rijmen, M. Schlaeffler, “Using Normal Bases for Compact Hardware Implementations of the AES S-Box”, in: R. Ostrovsky, R. De Prisco, I. Visconti, Eds., Security and Cryptography for Networks, SCN 2008, Lecture Notes in Computer Science, vol. 5229, pp. 236-245, Springer, Berlin, Heidelberg, 2008
- [146] Y. Nogami, K. Nekado, T. Toyota, N. Hongo, Y. Morikawa, “Mixed Bases for Efficient Inversion in $\mathbb{F}_{(2^2)^2}$ and Conversion Matrices of SubBytes of AES”, in: S. Mangard, FX. Standaert, Eds., Cryptographic Hardware and Embedded Systems – CHES 2010, CHES 2010, Lecture Notes in Computer Science, vol. 6225, pp. 234-247, Springer, Berlin, Heidelberg, 2010
- [147] A. Bonnecaze, P. Liardet, and A. Venelli, “AES side-channel countermeasure using random tower field constructions”, *Designs, Codes and Cryptography*, vol. 69, no. 3, pp. 331-349, December 2013
- [148] A. Reyhani-Masoleh, M. Taha, D. Ashmawy, “New Area Record for the AES Combined S-Box/Inverse S-Box,” 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH), Amherst, MA, 2018, pp. 145-152

- [149] A. Pradeep, V. Mohanty, A. M. Subramaniam, C. Rebeiro, “Revisiting AES SBox Composite Field Implementations for FPGAs”, in *IEEE Embedded Systems Letters*, vol. 11, no. 3, pp. 85-88, September 2019.
- [150] F. Rodríguez-Henríquez, N.A. Saqib, A. Díaz-Pérez, Ç.K. Koç, *Cryptographic Algorithms on Reconfigurable Hardware*, New York: Springer, 2006
- [151] H. Fan, M.A. Hasan, “A survey of some recent bit-parallel $GF(2^n)$ multipliers”, *Finite Fields and Their Applications*, vol. 32, pp. 5-13, March 2015
- [152] E. Mastrovito, “VLSI Architectures for Computations in Galois Fields” , PhD thesis, Linköping University, Linköping, Sweden, 1991
- [153] R. Azarderakhsh, A. Reyhani-Masoleh, “A Modified Low Complexity Digit-Level Gaussian Normal Basis Multiplier”, in: M.A. Hasan, Hellesteth T., Eds., *Arithmetic of Finite Fields, WAIFI 2010, Lecture Notes in Computer Science*, vol. 6087, pp. 25-40, Springer, Berlin, Heidelberg, 2010
- [154] J.L. Massey and J.K. Omura, *Computational Method and Apparatus for Finite Field Arithmetic*, US Patent No. 4587627, 1986
- [155] S. Kwon, K. Gaj, C.H. Kim, C.P. Hong, “Efficient Linear Array for Multiplication in $GF(2^m)$ Using a Normal Basis for Elliptic Curve Cryptography”, in: M. Joye, JJ. Quisquater, Eds., *Cryptographic Hardware and Embedded Systems – CHES 2004, CHES 2004, Lecture Notes in Computer Science*, vol. 3156, pp.76-91, Springer, Berlin, Heidelberg, 2004
- [156] A. Reyhani-Masoleh, M.A. Hasan, “A New Construction of Massey-Omura Parallel Multiplier over $GF(2^m)$ ”, in *IEEE Transactions on Computers*, vol. 51, no. 5, pp. 511-520, May 2002
- [157] A. Reyhani-Masoleh, M.A. Hasan, “Efficient Digit-Serial Normal Basis Multipliers over Binary Extension Fields”, *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 3, pp. 575-592, August 2004
- [158] A. Reyhani-Masoleh, M. A. Hasan, “Low complexity word-level sequential normal basis multipliers”, in *IEEE Transactions on Computers*, vol. 54, no. 2, pp. 98-110, February 2005
- [159] M. Olofsson, “VLSI Aspects on Inversion in Finite Fields”, PhD thesis, Linköping University, Linköping, Sweden, 2002
- [160] C. Beierle, T. Kranz, G. Leander, “Lightweight Multiplication in $GF2^n$ with Applications to MDS Matrices”, in: M. Robshaw, J. Katz, Eds., *Advances in Cryptology – CRYPTO 2016, CRYPTO 2016, Lecture Notes in Computer Science*, vol. 9814, pp. 625-653, Springer-Verlag, Berlin, Heidelberg, 2016

- [161] L. Kölsch, “XOR-Counts and Lightweight Multiplication with Fixed Elements in Binary Finite Fields”, in: Y. Ishai, V. Rijmen, Eds., *Advances in Cryptology – EUROCRYPT 2019, EUROCRYPT 2019, Lecture Notes in Computer Science*, vol. 11476, pp. 285-312, Springer, Cham, 2019
- [162] S.M. Sim, K. Khoo, F. Oggier, T. Peyrin, “Lightweight MDS involution matrices”, in: G. Leander, Eds., *Fast Software Encryption, FSE 2015, Lecture Notes in Computer Science*, vol. 9054, pp. 471-493. Springer, Heidelberg, 2015
- [163] T. Kranz, G. Leander, K. Stoffelen, F. Wiemer, “Shorter Linear Straight-Line Programs for MDS Matrices”, *IACR Transactions on Symmetric Cryptology*, vol. 2017, no. 4, pp. 188-211, December 2017
- [164] J. Jean, T. Peyrin, S.M. Sim, J. Tourteaux, “Optimizing implementations of lightweight building blocks”, *IACR Transactions on Symmetric Cryptology*, vol. 2017, no. 4, pp. 130-168, December 2017
- [165] M. Liu, S.M. Sim, “Lightweight MDS generalized circulant matrices”, in: T. Peyrin, Eds., *Fast Software Encryption, FSE 2016, Lecture Notes in Computer Science*, vol. 9783, pp. 101-120. Springer, Berlin, Heidelberg 2016
- [166] K. Khoo, T. Peyrin, A.Y. Poschmann, H. Yap, “FOAM: Searching for Hardware-Optimal SPN Structures and Components with a Fair Comparison”, in: L. Batina, M. Robshaw, Eds., *Cryptographic Hardware and Embedded Systems – CHES 2014, CHES 2014, Lecture Notes in Computer Science*, vol. 8731, pp. 433-450, Springer, Berlin, Heidelberg, 2014
- [167] R. Brayton, A. Mishchenko A., “ABC: An Academic Industrial-Strength Verification Tool”, in: T. Touili, B. Cook, P. Jackson, Eds., *Computer Aided Verification, CAV 2010, Lecture Notes in Computer Science*, vol. 6174, pp. 24-40, Springer, Berlin, Heidelberg, 2010
- [168] C. Pilato, F. Ferrandi, “Bambu: A modular framework for the high level synthesis of memory-intensive applications”, *2013 23rd International Conference on Field programmable Logic and Applications, Porto, 2013*, pp. 1-4, September 2013
- [169] R. Nane, V.M. Sima, B. Olivier, R. Meeuws, Y. Yankova, K. Bertels, “DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler”, *22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, 2012*, pp. 619-622, August 2012
- [170] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, T. Czajkowski, “LegUp: high-level synthesis for FPGA-based processor/accelerator systems”, In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11)*, ACM, New York, pp. 33-36, February 2011
- [171] P. Coussy, G. Lhairech-Lebreton, D. Heller, E. Martin, “GAUT – A Free and Open Source High-Level Synthesis Tool”, Available at <https://past.date-conference.com/files/file/10-ubooth/ub-1.3-p05.pdf> [Accessed: April 4, 2020]

- [172] G. Lhairech-Lebreton, P. Coussy, D. Heller, E. Martin, “Bitwidth-aware high-level synthesis for designing low-power DSP applications”, 2010 17th IEEE International Conference on Electronics, Circuits and Systems, Athens, 2010, pp. 531-534, December 2010
- [173] C. Andriamisaina, P. Coussy, E. Casseau and C. Chavet, “High-Level Synthesis for Designing Multimode Architectures”, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 29, no. 11, pp. 1736-1749, November 2010.
- [174] A. Khalid, G. Paul, A. Chattopadhyay, *Domain Specific High-Level Synthesis for Cryptographic Workloads*, Singapore: Springer Singapore, 2019
- [175] A. Khalid, M. Hassan, G. Paul, A. Chattopadhyay, “RunFein: a rapid prototyping framework for Feistel and SPN-based block ciphers”, Journal of Cryptographic Engineering, vol. 6, no. 4, pp.299-323, November 2016
- [176] A. Khalid, G. Paul, A. Chattopadhyay, F. Abediostad, S.I.U. Din, M. Hassan, B. Biswas, P. Ravi, “RunStream: A High-Level Rapid Prototyping Framework for Stream Ciphers”, ACM Transactions on Embedded Computing Systems, vol. 15, no. 3, Article 61, June 2016, Available at <https://dl.acm.org/doi/pdf/10.1145/2891412> [Accessed: April 4, 2020]
- [177] S. Browning, P. Weaver, “Designing Tunable, Verifiable Cryptographic Hardware Using Cryptol”, in: D. Hardin, Eds., Design and Verification of Microprocessor Systems for High-Assurance Applications, pp 89-143, Springer, Boston, MA, 2010
- [178] *Cryptol: The Language of Cryptography*, Galois inc., 2018, Available at: <https://cryptol.net/files/ProgrammingCryptol.pdf> [Accessed: April 4, 2020]
- [179] K. Gaj, J.P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, B. Y. Brewster, “ATHENa - Automated Tool for Hardware Evaluation: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs”, 2010 International Conference on Field Programmable Logic and Applications, Milano, 2010, pp. 414-421, August-September 2010
- [180] ATHENa: <https://cryptography.gmu.edu/athena/>
- [181] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, M.U. Sharif, K. Gaj, “A Universal Hardware API for Authenticated Ciphers”, 2015 International Conference on ReConfigurable Computing and FPGAs (ReConFig), Mexico City, 2015, pp. 1-8, December 2015
- [182] F. Farahmand, A. Ferozpuri, W. Diehl, K. Gaj, “Minerva: Automated Hardware Optimization Tool”, International Conference on ReConfigurable Computing and FPGAs (ReConFig), Cancun, 2017, pp. 1-8, December 2017
- [183] E. Homsirikamol and K. Gaj, “Toward a New HLS-Based Methodology for FPGA Benchmarking of Candidates in Cryptographic Competitions: The CAESAR Contest Case Study”, 2017 International Conference on Field Programmable Technology (ICFPT), Melbourne, VIC, 2017, pp. 120-127, December 2017

- [184] P. Banerjee, M. Haldar, A. Nayak, V. Kim, V. Saxena, S. Parkes, D. Bagchi, S. Pal, N. Tripathi, D. Zaretsky, R. Anderson, J. Uribe, “Overview of a compiler for synthesizing MATLAB programs onto FPGAs”, in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 12, no. 3, pp. 312-324, March 2004
- [185] T. Herlestam, “On Functions of Linear Shift Register Sequences”, in: F. Pichler F, Eds., Advances in Cryptology – EUROCRYPT ’85, EUROCRYPT 1985, Lecture Notes in Computer Science, vol 219. pp. 119-129, Springer Berlin Heidelberg, 1986
- [186] R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, R.M. Wilson, “ Optimal Normal Bases in $GF(p^n)$ ”, Discrete Applied Mathematics, vol. 22, no. 2, pp. 149-161, North-Holland, 1989
- [187] M. Aagaard, R. AlTawy, G. Gong, K. Mandal, R. Rohit, N. Zidaric, “WAGE: An Authenticated Cipher”, Round 1 Submission to *NIST Lightweight Cryptography Standardization Project*, <https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/wage-spec.pdf>
[Accessed: April 4, 2020]
- [188] M. D. Aagaard, M. Sattarov, N. Zidaric, “Hardware Design and Analysis of the ACE and WAGE Ciphers”, NIST LWC workshop 2019, arXiv:1909.12338v2, Available at <https://arxiv.org/abs/1909.12338v2> [Accessed: April 4, 2020]
- [189] Logic Friday v1.1.4 <https://web.archive.org/web/20131022021257/http://www.sontrak.com/>
- [190] G.B. Agnew, R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone , “An implementation for a fast public-key cryptosystem”, Journal of Cryptology, vol. 3, no. 2, pp. 63-79 , Springer, 1991

Part VIII

Appendix

Part VIII - Outline

A	Additional mathematical background	304
B	WAGE in more detail	307
C	The architectural decisions phase	311
D	The automated design entry and implementation phase	325

Appendix A

Additional mathematical background

Definition A.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 $\text{ord}(g) = r$. The order of an element must divide

the order of the group, i.e. for $\forall g \in \mathcal{G} : \text{ord}(g) \mid |G|$.

Definition A.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 $\text{ord}(g) = n$. That is $g^n = e$.

Definition A.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 A.4 [Definition 1.43 in [11]] If \mathcal{R} is an arbitrary ring and there exists a positive integer n such that $nr = 0$ for every $r \in R$, then the least such positive integer n is called characteristic of R and R is said to have (positive) characteristic n . If no such positive integer n exists, R is said to have characteristic 0.

Definition A.5 [Definition 2.37 in [19]] Let K be a field. A vector space V over K is a commutative group for a first operation denoted by “+”, together with a scalar multiplication from $K \times V$ into V , which sends (λ, x) on λx , and such that for all $x, y \in V$ and $\lambda, \mu \in K$:

- $\lambda(x + y) = \lambda x + \lambda y$
- $(\lambda + \mu)x = \lambda x + \mu x$
- $(\lambda\mu)x = \lambda\mu x$
- $1x = x$

An element $x \in V$ is called a vector and $\lambda \in K$ a scalar.

A.0.1 Polynomials

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

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

Their sum is defined as

$$g(x) + h(x) = \sum_{i=0}^{\max\{n,m\}} (a_i + b_i) x^i \tag{A.1}$$

and their product as

$$g(x) \cdot h(x) = \sum_{k=0}^{m+n} c_k x^k \quad \text{where} \quad c_k = \sum_{i+j=k} a_i b_j \tag{A.2}$$

The set of polynomials $\mathcal{K}[x]$ over field \mathcal{K} , with addition and multiplication defined as above, is a (commutative) polynomial ring (see A.3) with additive identity $f_0(x) = 0$ and multiplicative identity $f_1(x) = 1$ (Definition 1.48 in [11]).

In Section 3.2 in [14], the finite field \mathcal{F}/\mathcal{K} , where $[\mathcal{F} : \mathcal{K}] = m$ is constructed as follows: let $f \in \mathcal{K}[x]$ be a polynomial of degree m , which is irreducible over \mathcal{K} , and let α be a root of f , i.e., $f(\alpha) = 0$. Let the set F contain polynomials with degree at most $m - 1$ and coefficients from \mathcal{K} :

$$F = \left\{ \sum_{i=0}^{m-1} a_i \alpha^i ; a_i \in \mathcal{K} \right\}$$

The addition and multiplication of polynomials in F are defined as in equations (A.1) and (A.2) respectively¹, with the following modification: the product $c(\alpha) = g(\alpha) \cdot h(\alpha)$ needs to be *reduced* modulo $f(\alpha)$. This is achieved using the division algorithm, which produces two polynomials $q(\alpha), r(\alpha)$, such that

$$c(\alpha) = q(\alpha) \cdot f(\alpha) + r(\alpha) \quad \text{where} \quad \deg(r(\alpha)) < m \quad (\text{A.3})$$

Since $f(\alpha) = 0$, the final product is the remainder of $g(\alpha) \cdot h(\alpha)$ divided by $f(\alpha)$:

$$c(\alpha) = r(\alpha) \quad (\text{A.4})$$

Then the set F together with addition $g(\alpha) + h(\alpha)$, as defined in equation (A.1), and multiplication $g(\alpha) \cdot h(\alpha) = r(\alpha)$, as defined in equations (A.2)-(A.4), forms a finite field $\mathcal{F} = (F, +, \cdot)$ (Theorem 3.5 in [14]).

A.0.2 Miscellaneous

Theorem A.1 [Theorem 6 in [18]] *All finite fields of order p^m are isomorphic. [Two fields \mathcal{F}, \mathcal{G} are said to be isomorphic if there is a one-to-one mapping from \mathcal{F} onto \mathcal{G} which preserves addition and multiplication.]*

¹now using the root α

Appendix B

WAGE in more detail

Disclaimer B.1: The WAGE authenticated encryption scheme

The WAGE authenticated encryption scheme is a joint work of the members of the ComSec Lab, listed in alphabetical order: Mark Aagaard, Riham AlTawy, Guang Gong, Kalikinkar Mandal, Raghvendra Rohit, and Nusa Zidaric (author of this thesis). Sections B.2 and B.1 are added for the completeness of this thesis: they contain additional text from the WAGE submission document [6] (joint work).

B.1 WAGE loading and tag extraction

Previous sections 22.2-22.4 were discussing profiling and the WAGE permutation. This section is related to the WAGE permutation in a mode. The rate S_r and the data inputs D_k and outputs O_k , $k = 0, \dots, 9$, were specified in Section 3.2.8.

The 128-bit key K and 128-bit nonce N are divided into 7-bit tuples. In software we work with bytes, and since WAGE is using 7-bit tuples, we have “left-over” bits k_{63} and n_{63} ; instead of shifting all remaining key and nonce bits by 1, the bits n_{63} and k_{63} are put into the last key block \widehat{K}_{18}^* , which makes the loading phase and key absorption efficient for the software implementation.

Loading regions. Recall the data inputs D_k , $k = 0, \dots, 9$, in the shift register as shown in Figure 3.9. In order to minimize the hardware overhead, the data inputs D_k are reused for loading: instead of XORing the D_k with previous stage content, the D_k data is fed directly into the corresponding stage, and the stages without D_k inputs are loaded by shifting.

There are 10 D_k inputs available to load 37 stages: the LFSR is divided into loading regions. For example, the loading region S_8, \dots, S_0 can be loaded through the data input D_0 and has length 9, hence will require 9 shifts for loading. There are two other loading regions of length 9, namely region S_{27}, \dots, S_{19} (loaded through D_5) and region S_{36}, \dots, S_{28} (loaded through D_9). The remaining 10 consecutive stages were split into two regions, one of length 8 and another of length 2: the region of length 8 are the stages S_{16}, \dots, S_9 , loaded through D_3 , and the region of length 2 the stages S_{18}, S_{17} , loaded through D_4 .

WAGE contains SB inputs to the LFSR, that need to be disconnected during loading. For example, the loading region S_8, \dots, S_0 has a nonlinear input from the SB. The two WGP are automatically disabled by loading through D_9 and D_4 .

Loading sequence. The five loading regions, annotated with D_k used for loading, are listed below in a way that reflects their respective lengths. The \widehat{K}_i and \widehat{N}_i tuples on the right show the contents of the stages S_j after the loading is complete. The notations on the top denote the 64-bit loading blocks KN_t . They are formed by lumping together tuples appearing in the same column. For example, during the first shift we load the 64-bit block $KN_0 = \widehat{N}_0 || \widehat{K}_1 || 0^7 || 0^7 || \widehat{K}_0$ and during the last shift the block $KN_8 = \widehat{N}_{16} || \widehat{K}_{17} || \widehat{K}_{18}^* || \widehat{N}_{17} || \widehat{K}_{16}$.

$$\begin{array}{rcl}
S_{36}, S_{35}, S_{34}, S_{33}, S_{32}, S_{31}, S_{30}, S_{29}, S_{28} & \leftarrow^{D_9} & KN_8 \quad KN_7 \quad KN_6 \quad KN_5 \quad KN_4 \quad KN_3 \quad KN_2 \quad KN_1 \quad KN_0 \\
S_{27}, S_{26}, S_{25}, S_{24}, S_{23}, S_{22}, S_{21}, S_{20}, S_{19} & \leftarrow^{D_5} & \widehat{N}_{16}, \widehat{N}_{14}, \widehat{N}_{12}, \widehat{N}_{10}, \widehat{N}_8, \widehat{N}_6, \widehat{N}_4, \widehat{N}_2, \widehat{N}_0 \\
S_{18}, S_{17} & \leftarrow^{D_4} & \widehat{K}_{17}, \widehat{K}_{15}, \widehat{K}_{13}, \widehat{K}_{11}, \widehat{K}_9, \widehat{K}_7, \widehat{K}_5, \widehat{K}_3, \widehat{K}_1 \\
S_{16}, S_{15}, S_{14}, S_{13}, S_{12}, S_{11}, S_{10}, S_9 & \leftarrow^{D_3} & \widehat{K}_{18}^*, N_{15} \\
S_8, S_7, S_6, S_5, S_4, S_3, S_2, S_1, S_0 & \leftarrow^{D_0} & \widehat{N}_{17}, \widehat{N}_{13}, \widehat{N}_{11}, \widehat{N}_9, \widehat{N}_7, \widehat{N}_5, \widehat{N}_3, \widehat{N}_1 \\
& & \widehat{K}_{16}, \widehat{K}_{14}, \widehat{K}_{12}, \widehat{K}_{10}, \widehat{K}_8, \widehat{K}_6, \widehat{K}_4, \widehat{K}_2, \widehat{K}_0
\end{array}$$

The entire loading process for regions S_{18}, \dots, S_9 and S_8, \dots, S_0 is shown in Table B.1. The table shows the shifting of data through the registers in 9 shifts. The first column shows which KN_t is sent to the D_k inputs during the shift $t + 1$. The stages are shown in the second row of Table B.1, and the values “-” in the table denote the old, unknown values, which will be overwritten by the specified \widehat{K}_i and \widehat{N}_i blocks by the time the loading is finished. The state of stages S_{18}, \dots, S_0 after shifting 9 times, i.e. after the loading is finished, is visible from the last row.

KN_t block	shift count	D_4 S_{18}, S_{17}	D_3 $S_{16}, S_{15}, S_{14}, S_{13}, S_{12}, S_{11}, S_{10}, S_9$	D_0 $S_8, S_7, S_6, S_5, S_4, S_3, S_2, S_1, S_0$
KN_0	1	- -	- - - - - - - -	\widehat{K}_0 - - - - - - - -
KN_1	2	- -	\widehat{N}_1 - - - - - - - -	$\widehat{K}_2, \widehat{K}_0$ - - - - - - - -
KN_2	3	- -	$\widehat{N}_3, \widehat{N}_1$ - - - - - - - -	$\widehat{K}_4, \widehat{K}_2, \widehat{K}_0$ - - - - - - - -
KN_3	4	- -	$\widehat{N}_5, \widehat{N}_3, \widehat{N}_1$ - - - - - - - -	$\widehat{K}_6, \widehat{K}_4, \widehat{K}_2, \widehat{K}_0$ - - - - - - - -
KN_4	5	- -	$\widehat{N}_7, \widehat{N}_5, \widehat{N}_3, \widehat{N}_1$ - - - - - - - -	$\widehat{K}_8, \widehat{K}_6, \widehat{K}_4, \widehat{K}_2, \widehat{K}_0$ - - - - - - - -
KN_5	6	- -	$\widehat{N}_9, \widehat{N}_7, \widehat{N}_5, \widehat{N}_3, \widehat{N}_1$ - - - - - - - -	$\widehat{K}_{10}, \widehat{K}_8, \widehat{K}_6, \widehat{K}_4, \widehat{K}_2, \widehat{K}_0$ - - - - - - - -
KN_6	7	- -	$\widehat{N}_{11}, \widehat{N}_9, \widehat{N}_7, \widehat{N}_5, \widehat{N}_3, \widehat{N}_1$ - - - - - - - -	$\widehat{K}_{12}, \widehat{K}_{10}, \widehat{K}_8, \widehat{K}_6, \widehat{K}_4, \widehat{K}_2, \widehat{K}_0$ - - - - - - - -
KN_7	8	\widehat{N}_{15} -	$\widehat{N}_{13}, \widehat{N}_{11}, \widehat{N}_9, \widehat{N}_7, \widehat{N}_5, \widehat{N}_3, \widehat{N}_1$ - - - - - - - -	$\widehat{K}_{14}, \widehat{K}_{12}, \widehat{K}_{10}, \widehat{K}_8, \widehat{K}_6, \widehat{K}_4, \widehat{K}_2, \widehat{K}_0$ - - - - - - - -
KN_8	9	$\widehat{K}_{18} \widehat{N}_{15}$	$\widehat{N}_{17}, \widehat{N}_{13}, \widehat{N}_{11}, \widehat{N}_9, \widehat{N}_7, \widehat{N}_5, \widehat{N}_3, \widehat{N}_1$	$\widehat{K}_{16}, \widehat{K}_{14}, \widehat{K}_{12}, \widehat{K}_{10}, \widehat{K}_8, \widehat{K}_6, \widehat{K}_4, \widehat{K}_2, \widehat{K}_0$

Table B.1: Loading into the shift register through data inputs D_4 , D_3 and D_0

Tag extraction regions. The tag is extracted in a similar fashion, from the positions that were loaded with nonce tuples. For example, the state region S_{16}, \dots, S_9 , which was loaded through D_3 , is extracted through the output that belongs to the D_1 input. Similarly, the state region S_{18}, S_{17}

is extracted through the output belonging to the D_3 input and the region S_{36}, \dots, S_{28} through the output belonging to the D_6 input. The longest tag extraction region is also of length 9. Similar to KN_t for the loading, the 7-bit tuples extracted during shift $t + 1$ are lumped into a tag-extract block TE_t .

B.2 WAGE permutation

The finite field \mathcal{F} is defined using the primitive polynomial $f(x) = x^7 + x^3 + x^2 + x + 1$. Field elements are represented using the polynomial basis $\text{PB} = \{1, \omega, \dots, \omega^6\}$, $f(\omega) = 0$. For $a \in \mathcal{F}$ the conversion to binary and HEX is as follows:

$$[a]_{\text{PB}} = (a_0, a_1, a_2, a_3, a_4, a_5, a_6) \rightarrow [a]_b = (0, a_0, a_1, a_2, a_3, a_4, a_5, a_6) \rightarrow [a]_{\text{hex}} = (h_1, h_0)$$

Table B.2 shows some examples of the conversion to HEX.

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	16^1	16^0
$a \in \mathcal{F}$	0	a_0	a_1	a_2	a_3	a_4	a_5	a_6	h_1	h_0
1	0	1	0	0	0	0	0	0	4	0
ω	0	0	1	0	0	0	0	0	2	0
$1 + \omega$	0	1	1	0	0	0	0	0	6	0
$1 + \omega^6$	0	1	0	0	0	0	0	1	4	1

Table B.2: Examples of conversion of the field elements to HEX

An Sbox representation of WGP is given in Table B.3 in a row-major order. The 7-bit finite field elements are represented in HEX as shown in Table B.2. The HEX representation of SB is provided in Table B.4. The round constants are listed in Table B.5. The interpretation of the round constants rc_j^i , where $j = 0, 1$ and $i = 0, \dots, 110$, in Table B.5 is as follows:

$$[rc_j^i]_{\text{hex}} \rightarrow [rc_j^i]_b = (0, r_{j,0}, r_{j,1}, r_{j,2}, r_{j,3}, r_{j,4}, r_{j,5}, r_{j,6}) \rightarrow [r_j]_{\text{PB}} = (r_{j,0}, r_{j,1}, r_{j,2}, r_{j,3}, r_{j,4}, r_{j,5}, r_{j,6})$$

00	12	0a	4b	66	0c	48	73	79	3e	61	51	01	15	17	0e
7e	33	68	36	42	35	37	5e	53	4c	3f	54	58	6e	56	2a
1d	25	6d	65	5b	71	2f	20	06	18	29	3a	0d	7a	6c	1b
19	43	70	41	49	22	77	60	4f	45	55	02	63	47	75	2d
40	46	7d	5c	7c	59	26	0b	09	03	57	5d	27	78	30	2e
44	52	3b	08	67	2c	05	6b	2b	1a	21	38	07	0f	4a	11
50	6a	28	31	10	4d	5f	72	39	16	5a	13	04	3c	34	1f
76	1e	14	23	1c	32	4e	7b	24	74	7f	3d	69	64	62	6f

Table B.3: Hex representation of WGP-16(X^d)

2e	1c	6d	2b	35	07	7f	3b	28	08	0b	5f	31	11	1b	4d
6e	54	0d	09	1f	45	75	53	6a	5d	61	00	04	78	06	1e
37	6f	2f	49	64	34	7d	19	39	33	43	57	60	62	13	05
77	47	4f	4b	1d	2d	24	48	74	58	25	5e	5a	76	41	42
27	3e	6c	01	2c	3c	4e	1a	21	2a	0a	55	3a	38	18	7e
0c	63	67	56	50	7c	32	7a	68	02	6b	17	7b	59	71	0f
30	10	22	3d	40	69	52	14	36	44	46	03	16	65	66	72
12	0e	29	4a	4c	70	15	26	79	51	23	3f	73	5b	20	5c

Table B.4: Hex representation of SB

Round i	Round constant (rc_1^i, rc_0^i)											
0 - 9	(3f, 7f)	(0f, 1f)	(03, 07)	(40, 01)	(10, 20)	(04, 08)	(41, 02)	(30, 60)	(0c, 18)	(43, 06)		
10 - 19	(50, 21)	(14, 28)	(45, 0a)	(71, 62)	(3c, 78)	(4f, 1e)	(13, 27)	(44, 09)	(51, 22)	(34, 68)		
20 - 29	(4d, 1a)	(66, 73)	(5c, 39)	(57, 2e)	(15, 2b)	(65, 4a)	(79, 72)	(3e, 7c)	(2f, 5f)	(0b, 17)		
30 - 39	(42, 05)	(70, 61)	(1c, 38)	(47, 0e)	(11, 23)	(24, 48)	(49, 12)	(32, 64)	(6c, 59)	(5b, 36)		
40 - 49	(56, 2d)	(35, 6b)	(6d, 5a)	(7b, 76)	(5e, 3d)	(37, 6f)	(0d, 1b)	(63, 46)	(58, 31)	(16, 2c)		
50 - 59	(25, 4b)	(69, 52)	(74, 3a)	(6e, 5d)	(3b, 77)	(4e, 1d)	(33, 67)	(4c, 19)	(53, 26)	(54, 29)		
60 - 69	(55, 2a)	(75, 6a)	(7d, 7a)	(7f, 7e)	(1f, 3f)	(07, 0f)	(01, 03)	(20, 40)	(08, 10)	(02, 04)		
70 - 79	(60, 41)	(18, 30)	(06, 0c)	(21, 43)	(28, 50)	(0a, 14)	(62, 45)	(78, 71)	(1e, 3c)	(27, 4f)		
80 - 89	(09, 13)	(22, 44)	(68, 51)	(1a, 34)	(66, 4d)	(39, 73)	(2e, 5c)	(2b, 57)	(4a, 15)	(72, 65)		
90 - 99	(7c, 79)	(5f, 3e)	(17, 2f)	(05, 0b)	(61, 42)	(38, 70)	(0e, 1c)	(23, 47)	(48, 11)	(12, 24)		
100 - 109	(64, 49)	(59, 32)	(36, 6c)	(2d, 5b)	(6b, 56)	(5a, 35)	(76, 6d)	(3d, 7b)	(6f, 5e)	(1b, 37)		
110	(46, 0d)											

Table B.5: Round constants of WAGE

Appendix C

The architectural decisions phase

C.1 FSR package: feedback shift registers

Example C.1.1 *A simple LFSR over an extension field - continued* \longleftrightarrow

The TEX writing functions come in two versions: the finite field elements can be represented in a chosen basis or as powers of generator. Recall Example 6.2.2: first run from GAP code shown in Example 6.2.2(a) can be written using `WriteTEXRunFSR` (Table C.1) and `WriteTEXRunFSRByGenerator` (Table C.2). The two functions produce the same information using different representation. Generated *.tex files were included directly, with only table labels set manually. The package FSR offers a lot of other formatting functions, some of them used for captions of Tables C.1 and C.2. Another helper function is `WriteTEXElementTableByGenerator`, that writes the elements of the field as a table in a chosen basis and as power of a chosen generator, see Table C.3.

Furthermore, the created FSR objects can be represented graphically using automatically generated tikz code. The LFSR is shown in Figure C.1.

step	state				sequence
num	\mathcal{S}_3	\mathcal{S}_2	\mathcal{S}_1	\mathcal{S}_0	\mathcal{S}_0
0	0000	0110	1101	1000	1000
1	1011	0000	0110	1101	1101
2	1100	1011	0000	0110	0110
3	0111	1100	1011	0000	0000
4	1100	0111	1100	1011	1011
5	1010	1100	0111	1100	1100

Table C.1: LFSR with feedback $y^4 + y^3 + y + \alpha$ over \mathbb{F}_{2^4} with basis $B = [\beta_i] = [1, \alpha^7, \alpha^{14}, \alpha^6]$ where $\alpha = \omega^1 + \omega^2$ and ω is a root of $x^4 + x^3 + 1$.

The whole sequence: 1000, 1101, 0110, 0000, 1011, 1100

step num	state				sequence
	S_3	S_2	S_1	S_0	S_0
0	0	α	α^5	1	1
1	α^2	0	α	α^5	α^5
2	α^9	α^2	0	α	α
3	α^{11}	α^9	α^2	0	0
4	α^9	α^{11}	α^9	α^2	α^2
5	α^3	α^9	α^{11}	α^9	α^9

Table C.2: LFSR with feedback $y^4 + y^3 + y + \alpha$ over \mathbb{F}_{2^4} where generator where $\alpha = \omega^1 + \omega^2$ and ω is a root of $x^4 + x^3 + 1$.

The whole sequence: $1, \alpha^5, \alpha, 0, \alpha^2, \alpha^9$

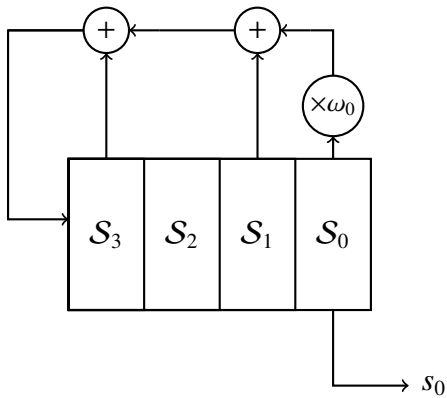


Figure C.1: LFSR with feedback $y^4 + y^3 + y + \alpha$ over \mathbb{F}_{2^4} where generator where $\alpha = \omega^1 + \omega^2$ and ω is a root of $x^4 + x^3 + 1$ and constant(s) $\omega_0 = \alpha$.

elm	given basis B				
order	β_0	β_1	β_2	β_3	α^i
-	0	0	0	0	0
1	1	0	0	0	1
3	1	1	0	1	α^5
3	0	1	0	1	α^{10}
15	0	1	1	0	α
15	1	0	1	1	α^2
5	1	0	1	0	α^3
15	1	1	1	0	α^4
5	0	0	0	1	α^6
15	0	1	0	0	α^7
15	0	0	1	1	α^8
5	1	1	0	0	α^9
15	0	1	1	1	α^{11}
5	1	1	1	1	α^{12}
15	1	0	0	1	α^{13}
15	0	0	1	0	α^{14}

Table C.3: Element table for \mathbb{F}_{2^4} using basis $B = [\beta_i] = [1, \alpha^7, \alpha^{14}, \alpha^6]$ with generator α where $\alpha = \omega^1 + \omega^2$ and ω is a root of $x^4 + x^3 + 1$.

C.2 FFCSA package part 1: additional mathematical background

This section contains additional definitions and theorems, used for the implementation of FFCSA package methods. This section is ordered in the same way as the FFCSA methods are introduced in Tables 7.2 - 7.5.

C.2.1 Cyclotomic coset leaders

Definition C.1 [Definition 3.18 in [14]] A (cyclotomic) coset C_s modulo $p^m - 1$ is defined to be

$$C_s = \{s, sp, \dots, sp^{m_s-1}\},$$

where m_s is the smallest positive integer such that $s \equiv sp^{m_s} \pmod{p^m - 1}$. The subscript s is chosen as the smallest integer in C_s , and s is called the coset leader of C_s .

Method CCLeaders returns the set $\Gamma_p(m) = \{\text{all coset leaders in } \mathbb{Z}_{p^m-1}\}$, obtained by definition C.1.

C.2.2 Polynomial Φ function

Definition C.2 [Definition 2.1.111 in [15]] For $f \in \mathbb{F}_q[x]$, the finite field polynomial Φ function, $\Phi_q(f)$, denotes the number of polynomials over \mathbb{F}_q , which are of smaller degree than degree of f and which are relatively prime to f .

Method PolyPhi(F, f) is implemented using properties of Φ_q and the existing GAP method Factors(PolynomialRing(F), f) (Sections 66.10-1 and Ch. 66.15-1 in [34]) to obtain factorization $f = \prod_i f_i^{c_i}$, where f_i are irreducible, $\deg(f_i) = n_i$ and c_i is the multiplicity of f_i . Properties of Φ_q used (see page 9 in [12]):

1. if $\gcd(f_i, f_j) = 1$ then $\Phi_q(f_i f_j) = \Phi_q(f_i) \Phi_q(f_j)$
2. if f_i is irreducible of degree n_i then $\Phi_q(f_i^{c_i}) = q^{n_i c_i} - q^{n_i(c_i-1)}$

C.2.3 Number of irreducible and primitive polynomials

The number of monic irreducible and primitive polynomials by methods NrIPoly(F, f), where $F = \mathbb{F}_q$:

- `NrMonicIrreduciblePoly` - the number of monic irreducible polynomials of degree m over \mathbb{F}_q (Theorem 3.1.2 in [15]):

$$I_q(m) = \frac{1}{m} \sum_{d|m} \mu(d) q^{\frac{m}{d}} \quad (\text{C.1})$$

where μ is the Möbius function (equation (C.3))

- `NrPrimitivePoly` - the number of primitive polynomials¹ of degree m over \mathbb{F}_q (Theorem 4.1.3 in [15]):

$$\frac{1}{m} \Phi(q^m - 1) \quad (\text{C.2})$$

where Φ denotes the Euler function²

- `NrIrreducibleNotPrimitivePoly` is computed as the difference of the previous two

Möbius function, Definition 2.1.22 in [15]:

$$\mu(d) = \begin{cases} 1 & \text{if } d = 1, \\ (-1)^k & \text{if } d = d_1 d_2 \dots d_k, \text{ where } d_i \text{ are distinct primes,} \\ 0 & \text{otherwise, i.e., if } p^2 \text{ divides } d \text{ for some prime } p. \end{cases} \quad (\text{C.3})$$

Note: using the existing GAP method `MoebiusMu` (see Section 15.5-3 in [34]).

Euler function, Definition 2.1.43 in [15]:

The number of positive integers $e \leq n$, such that $\gcd(n, e) = 1$, is denoted by $\Phi(n)$, and is the Euler function.

C.2.4 Number of normal elements and bases

Methods `NrNormalFFE*(F)`, where $F = \mathbb{F}_{q^m}$, are implemented as follows:

- `NrNormalFFE` - the number of normal elements in \mathbb{F}_{q^m} over \mathbb{F}_q (Corollary 5.2.8 in [15]):

$$\Phi_q(x^m - 1) \quad (\text{C.4})$$

where Φ denotes the polynomial Φ function (Section C.2.2)

- `NrNormalFFEIgnoreConjugates` - the number of normal bases of \mathbb{F}_{q^m} over \mathbb{F}_q (Corollary 4.14 in [12]):

$$\frac{1}{m} \Phi_q(x^m - 1) \quad (\text{C.5})$$

where Φ denotes the polynomial Φ function (Section C.2.2)

¹ monic by definition

² note to be mistaken for the Φ function in Section C.2.2

C.2.5 Matrices for the matrix-vector multipliers

Let $A, B, C \in \mathcal{F}$ and let $B_{\mathcal{F}/\mathcal{K}} = \{\alpha^{(0)}, \alpha^{(1)}, \dots, \alpha^{(m-1)}\}$ be an arbitrary basis of \mathcal{F}/\mathcal{K} , where $m = [\mathcal{F} : \mathcal{K}]$. The following are the representation of A w.r.t. basis $B_{\mathcal{F}/\mathcal{K}}$, its vector form and notation for the i -th coordinate of A , $0 \leq i \leq m - 1$:

$$A = \sum_{i=0}^{m-1} a_i \alpha^{(i)} \quad [A]_B = [a_0, a_1, \dots, a_{m-1}] \quad [A]_{B,(i)} = a_i$$

where $a_i \in \mathcal{K}$, $i = 0, 1, \dots, m - 1$. If there is no ambiguity, i.e., just one basis, the subscript B can be removed.

- **MatrixMultByConst**(B, ffe) returns a $m \times m$ matrix M whose columns are $[\gamma \alpha^{(i)}]$, where $ffe = \gamma$
- **TransitionMatrix**($B1, B2$) for $B1 = \{\alpha^{(0)}, \alpha^{(1)}, \dots, \alpha^{(m-1)}\}$ and $B2 = \{\beta^{(0)}, \beta^{(1)}, \dots, \beta^{(m-1)}\}$, return a $m \times m$ matrix M whose columns are $[\alpha^{(i)}]_{B2}$, $i = 0, \dots, m - 1$
- **ReductionMatrix Δ** (f), where $\deg(f) = m$, returns a matrix w.r.t. polynomial f with following cases:
 - direction “to”: using polynomial basis $B = \{1, x, \dots, x^{m-1}\}$,
 - * **ReductionMatrix**(f) returns $m \times (m - 1)$ matrix R , whose columns are elements $[x^m], [x^{m+1}], \dots [x^{2m-2}]$
 - * **ReductionMatrixIR**(f) computes matrix R then returns a $m \times (2m - 1)$ matrix of the form $I|R$ - columns of identity matrix I , followed by the columns of R
 - direction “downto”: using polynomial basis $B = \{x^{m-1}, \dots, x, 1\}$,
 - * **ReductionMatrixDownto**(f) returns $m \times (m - 1)$ matrix R , whose columns are elements $\dots [x^{2m-2}], \dots, [x^{m+1}], [x^m]$
 - * **ReductionMatrixRI**(f) computes matrix R then returns a $m \times (2m - 1)$ matrix of the form $R|I$ - columns of R , followed by the columns of identity matrix I

Matrices for Massey-Omura multiplication

MatrixM(B), **MatrixMi**(B, i), where B is a normal basis of length³ m , returns $m \times m$ matrices M and coordinate matrices M_i , $i = 0, \dots, m - 1$. The discussion below follows the Slide Set 12 in [42], but can also be found in [190]. Let $B = \{\beta, \beta^2, \dots, \beta^{2^{m-1}}\}$. The matrices are computed as follows:

³ which is also the degree of extension $m = [\mathbb{F}_{2^m} : \mathbb{F}_2]$

$$M = \begin{bmatrix} \beta^{2^0+2^0} & \beta^{2^0+2^1} & \dots & \beta^{2^0+2^{m-1}} \\ \beta^{2^1+2^0} & \beta^{2^1+2^1} & \dots & \beta^{2^1+2^{m-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \beta^{2^{m-1}+2^0} & \beta^{2^{m-1}+2^1} & \dots & \beta^{2^{m-1}+2^{m-1}} \end{bmatrix} \quad (\text{C.6})$$

Method `MatrixM(B)` returns the matrix from equation (C.6) in the form where each matrix entry is a vector of coefficients w.r.t. basis B , i.e., $[\beta^{2^j+2^k}]$.

Matrix M can be written in terms of smaller matrices, which are obtained for a specific coordinate, i.e., a specific basis element:

$$M = M_0\beta + M_1\beta^2 + \dots + M_{m-1}\beta^{2^{m-1}} \quad (\text{C.7})$$

Method `MatrixMi(B, i)` returns the matrix from equation (C.6) in the form where each matrix entry is the i -th coefficient w.r.t. basis B , i.e., $[\beta^{2^j+2^k}]_{(i)}$.

C.2.6 Conditions for existence of optimal normal bases type I and II

From Theorem 5.3.6 in [15] and Section 8.6 in [20], the following conditions on m and q are used to determine if an optimal normal basis of type I or II exists for the finite field $\mathbb{F}_{q^m}/\mathbb{F}_q$:

1. `ISONBI` - $m + 1$ is a prime and q is a primitive element⁴ modulo $m + 1$.
2. `ISONBII` - $2m + 1$ is prime and either
 - 2 is a primitive element modulo $2m + 1$, or
 - $2m + 1 \equiv 3 \pmod{4}$ and multiplicative order of 2 modulo $2m + 1$ is m

C.2.7 Dual bases

Definition C.3 [Definition 5.1.1 in [15]] Two ordered bases of \mathbb{F}_{q^m} over \mathbb{F}_q $B_1 = \{\alpha_i\}_0^{m-1}$ and $B_2 = \{\beta_i\}_0^{m-1}$ are dual if for $i, j = 0, \dots, m - 1$

$$\text{Tr}_{\mathbb{F}_q}^{\mathbb{F}_{q^m}}(\alpha_i\beta_j) = \delta_{ij} = \begin{cases} 0 & ; i \neq j \\ 1 & ; i = j \end{cases}$$

An ordered basis is self-dual if it is dual with itself.

Method `IsDualBasisPair(B1, B2)` follows definition C.3.

⁴the multiplicative order of q modulo $m + 1$ is m

Finding a dual basis to normal basis

Theorem C.1 [Theorem 4.7 in [12]] Let $NB = \{\alpha, \alpha^q, \alpha^{q^2}, \dots, \alpha^{q^{m-1}}\}$ be a normal basis of $\mathbb{F}_{q^m}/\mathbb{F}_q$. Let

$$t_i = \text{Tr}_{\mathbb{F}_{q^m}/\mathbb{F}_q}(\alpha\alpha^{q^i}) \quad \text{and} \quad N(x) = \sum_{i=0}^{m-1} t_i x^i. \quad (\text{C.8})$$

Furthermore, let

$$D(x) = \sum_{i=0}^{m-1} d_i x^i, \quad d_i \in \mathbb{F}_q, \quad (\text{C.9})$$

be the unique polynomial such that

$$N(x)D(x) \equiv 1 \pmod{(x^m + 1)}. \quad (\text{C.10})$$

Then, the dual basis of N is generated by

$$\beta = \sum_{i=0}^{m-1} d_i \alpha^{q^i}. \quad (\text{C.11})$$

Conjugates of element β generate a basis DB , which is dual w.r.t. NB and is itself a normal basis:

$$DB = \{\beta, \beta^q, \beta^{q^2}, \dots, \beta^{q^{m-1}}\} \quad (\text{C.12})$$

Method `FindInvCyc(F, N)` takes the polynomial $N(x)$ and finds its inverse $D(x)$ w.r.t. cyclotomic polynomial $x^m - 1$, see equation(C.10). For a normal basis B of $\mathbb{F}_{q^m}/\mathbb{F}_q$, where $F = \mathbb{F}_{q^m}$, the method `GeneratorOfDBtoNB(F, ffe)` performs the following steps:

1. recreate the normal basis B from the normal element $ffe = \alpha$ as $\{\alpha, \alpha^q, \dots, \alpha^{q^{m-1}}\}$, where $m = [\mathbb{F}_{q^m} : \mathbb{F}_q]$
2. form the polynomial $N(x)$ from basis B elements as shown in equation (C.8)
3. obtain polynomial $D(x)$ as shown in equation (C.9) using method `FindInvCyc(F, N)`, where $F = \mathbb{F}_{q^m}$ and $N = N(x)$
4. return β from equation (C.11), formed from coefficients of $D(x)$

For a normal basis B of $\mathbb{F}_{q^m}/\mathbb{F}_q$, the method `GenerateDBtoNB(F, ffe)` returns the basis DB in equation (C.12) by calling `GenerateNB(F, GeneratorOfDBtoNB(F, elm))`.

Finding a dual basis to polynomial basis

Theorem C.2 [Slide set 12 in [42], Theorem 5.1.12 in [15]] Let α be a root of a monic irreducible polynomial f of degree m over \mathbb{F}_q , and let $PB = \{1, \alpha, \alpha^2, \dots, \alpha^{m-1}\}$ be the corresponding polynomial basis of $\mathbb{F}_{q^m}/\mathbb{F}_q$. The polynomial f can be rewritten as:

$$f(x) = (x - \alpha) \left(\sum_{i=0}^{m-1} g_i x^i \right) \quad \text{where } g_i \in \mathbb{F}_{q^m} \quad \text{for } i = 0, \dots, m-1. \quad (\text{C.13})$$

The dual basis of PB is obtained from coefficients g_i and the derivative of $f(x)$ evaluated for α , that is $f'(\alpha)$:

$$DB = \left\{ \frac{g_0}{f'(\alpha)}, \dots, \frac{g_{m-1}}{f'(\alpha)} \right\} \quad (\text{C.14})$$

The method `GenerateDBtoPB(F, ffe)`, where $F = \mathbb{F}_{q^m}$ and $ffe = \alpha$, performs the following steps:

1. obtain f as minimal polynomial of ffe (using `MinimalPolynomial`, see Section .66.8-1 in [34])
2. compute $(\sum_{i=0}^{m-1} g_i x^i)$ to obtain the coefficients g_i
3. compute $f'(\alpha)$ (using `Derivative` and `Value`, see Sections 66.6-5 and Ch. 66.7-1 in [34])
4. return the dual basis DB from equation (C.14)

`GenerateDBtoPBdownto(F, ffe)` method proceeds exactly the same, but uses the reverse order of basis elements in step 4.

■ **Implementation detail:** Arguments F, ffe

The FFCSA package does not create any new GAP objects and because of that, there is no way of checking if a basis B , used as an argument, is a normal basis or not. Instead, the basis B is recreated for the finite field F from the normal element ffe . The same goes for the polynomial basis, which has an extra restriction: it can use the “to” or “downto” direction. ■

C.2.8 Optimal normal bases

Method `FindONB \diamond Generator**(F)`, where \diamond is listing both type I (ONBI) and type II (ONBII) optimal normal basis, is currently implemented as exhaustive search. It is a profiling method, that computes `ComplexityOfT` for the multiplication table (equation (3.5)) for each normal basis of $F = \mathbb{F}_{q^m}/\mathbb{F}_q$, and compares it to the theoretical minimum $2m - 1$, where m is the number of basis elements. Details on `ComplexityOfT` can be found in Section 18.1. As a normal basis method, it suffers from FFCSA package limitations C.2.10(1.), namely the restriction to $F \leq \mathbb{F}_{2^{16}}$. When this limitation is circumvented, the proper methods based on remaining⁵ part of the Theorem 5.3.6 in [15] for finding the ONB generators can be implemented.

⁵ not listed in Section C.2.6

C.2.9 Tower field bases - general case

This Section describes a general case for a tower field construction and tower field basis TFB , derived from the “per-level” bases PLB , which were introduced in Section 7.3.1 in Chapter 7 for example $\mathbb{F}_{((2^2)^2)^2}$. The tower field basis as given in equation (7.1), can be generalized to an arbitrary tower field construction. For $m = n_1 \cdots n_k$ and a construction from equation (3.3):

$$\mathbb{F}_p = \mathcal{K}_0 \subset \mathcal{K}_1 \subset \cdots \subset \mathcal{K}_{k-1} \subset \mathcal{K}_k = \mathbb{F}_{(\dots((p^{n_1})^{n_2})\dots)^{n_k}} \cong \mathbb{F}_{p^m}$$

For k extensions, there are k “per-level” bases of the following form: $PLB_j = \{u_{j,0}, \dots, u_{j,n_j-1}\}$ for $\mathcal{K}_j/\mathcal{K}_{j-1}$, $j = 1, \dots, k$. Then the elements of the $TFB_{\mathbb{F}_{p^m}/\mathbb{F}_p}$ are m distinct products of the “per-level” basis elements:

$$t_{i_1, i_2, \dots, i_k} = \prod_{j=1}^k u_{j, i_j}, \quad \text{where } i_j \in \{0, \dots, n_j - 1\} \quad (\text{C.15})$$

Since $m = n_1 \cdots n_k$ and, by construction, each “per-level basis” has n_j elements, there are n_j possible choices for u_{j, i_j} , $j = 1, \dots, k$, and hence exactly m distinct products t_{i_1, i_2, \dots, i_k} . It can be shown that the TFB elements are linearly independent following the proof for Theorem 1.84 in [11]. Following the construction in a “top-down” fashion, as was show in diagram in Figure 7.2 for the example $TFB_{\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_2}$, an element $A \in \mathcal{K}_k/\mathcal{K}_0$ can be expressed in terms of “per-level” bases $PLB_k, PLB_{k-1}, \dots, PLB_1$:

$$\begin{aligned} A &= \sum_{i_k=0}^{n_k-1} a_{i_k} u_{k, i_k}, & a_{i_k} &\in \mathcal{K}_{k-1} \\ &= \sum_{i_k=0}^{n_k-1} \left(\sum_{i_{k-1}=0}^{n_{k-1}-1} a_{i_{k-1}, i_k} u_{k-1, i_{k-1}} \right) u_{k, i_k}, & a_{i_{k-1}, i_k} &\in \mathcal{K}_{k-2} \\ & & \dots & \\ &= \sum_{i_k=0}^{n_k-1} \left(\sum_{i_{k-1}=0}^{n_{k-1}-1} \left(\dots \left(\sum_{i_2=0}^{n_2-1} a_{i_2, \dots, i_{k-1}, i_k} u_{2, i_2} \right) \dots \right) u_{k-1, i_{k-1}} \right) u_{k, i_k}, & a_{i_2, \dots, i_{k-1}, i_k} &\in \mathcal{K}_1 \\ &= \sum_{i_k=0}^{n_k-1} \left(\sum_{i_{k-1}=0}^{n_{k-1}-1} \left(\dots \left(\sum_{i_2=0}^{n_2-1} \left(\sum_{i_1=0}^{n_1-1} a_{i_1, i_2, \dots, i_{k-1}, i_k} u_{1, i_1} \right) u_{2, i_2} \right) \dots \right) u_{k-1, i_{k-1}} \right) u_{k, i_k}, & a_{i_1, i_2, \dots, i_{k-1}, i_k} &\in \mathcal{K}_0 \end{aligned} \quad (\text{C.16})$$

It is possible to show that all coefficients $a_{i_{k-1}, i_k, \dots, i_1} \in \mathcal{K}_0$ must be 0 for $A = 0$. Since each PLB_j is a basis, it holds that its linear combination is 0 only when the corresponding coefficients are 0. Then, following the “bottom-up” approach and starting with the innermost bracket:

$$\begin{aligned} 0 &= a_{i_2, \dots, i_{k-1}, i_k} = \sum_{i_1=0}^{n_1-1} a_{i_1, i_2, \dots, i_{k-1}, i_k} u_{1, i_1} &\Leftrightarrow \forall a_{i_1, i_2, \dots, i_{k-1}, i_k} &= 0 \\ & & \dots & \\ 0 &= A = \sum_{i_k=0}^{n_k-1} a_{i_k} u_{k, i_k} &\Leftrightarrow \forall a_{i_k} &= 0 \end{aligned}$$

Removing the brackets from last line in equation (C.16) gives the linear combination of the *TFB* elements, namely:

$$\begin{aligned}
A &= \sum_{i_k=0}^{n_k-1} \sum_{i_{k-1}=0}^{n_{k-1}-1} \cdots \sum_{i_2=0}^{n_2-1} \sum_{i_1=0}^{n_1-1} a_{i_1, i_2, \dots, i_{k-1}, i_k} u_{1, i_1} u_{2, i_2} \cdots u_{k-1, i_{k-1}} u_{k, i_k} \\
&= \sum_{i_k=0}^{n_k-1} \sum_{i_{k-1}=0}^{n_{k-1}-1} \cdots \sum_{i_2=0}^{n_2-1} \sum_{i_1=0}^{n_1-1} a_{i_1, i_2, \dots, i_{k-1}, i_k} \prod_{j=1}^{j=k} u_{j, i_j} \\
&= \sum_{i_k=0}^{n_k-1} \sum_{i_{k-1}=0}^{n_{k-1}-1} \cdots \sum_{i_2=0}^{n_2-1} \sum_{i_1=0}^{n_1-1} a_{i_1, i_2, \dots, i_{k-1}, i_k} t_{i_1, i_2, \dots, i_{k-1}, i_k} \tag{C.17}
\end{aligned}$$

The last line in equation (C.17) is a linear combination of the *TFB* basis elements with coefficients $a_{i_1, i_2, \dots, i_{k-1}, i_k} \in \mathcal{K}_0$. These are the same coefficients as in equation (C.16), and setting $A = 0$ again implies $a_{i_1, i_2, \dots, i_{k-1}, i_k} = 0$ for all $a_{i_1, i_2, \dots, i_{k-1}, i_k}$, hence m elements $t_{i_1, i_2, \dots, i_{k-1}, i_k}$ are linearly independent and form the basis $TFB_{\mathbb{F}_{p^m}/\mathbb{F}_p}$.

C.2.10 FFCSA package limitations

1. for finite fields $F \leq \mathbb{F}_{2^{16}}$, GAP is using Zech's logarithm representation of field elements (see IsFFE and Z Sections 59.1-1 and 59.1-2 in [34]), for larger fields polynomial representation is used. The existing GAP method `Conjugates` is using IsFFE, which affects the normal basis methods implemented in FFCSA package
2. GAP uses a set of precomputed Conway polynomials up to $m = 409$, however, some polynomials are missing in between, e.g., $m=128$. Some methods in the automation framework rely on these polynomials, and for the missing m a warning is displayed - these cases were skipped in all examples. A list of precomputed polynomials is available at <http://www.math.rwth-aachen.de/~Frank.Luebeck/data/ConwayPol/CP2.html>

C.3 FFCSA package part 2: additional examples

Example C.3.1 $\mathbb{F}_{(2^4)^4}$ tower field basis using a mixed basis \longleftrightarrow

The “mixed basis” method allows to choose a different type of basis at each level. Example C.3.1 shows a normal basis for the first level and a polynomial basis with direction “downto” for the second level of extension. This information was passed on to the FFCSA method with an instruction list following the *edpl*:

```
GenerateTFBfromEDPLwithMB(edpl, [{"NB", "to"}, {"PB", "downto"}]).
```

Then both individual bases are shown, namely $B1 = NB_{\mathbb{F}_{2^4}/\mathbb{F}_2}$ and $B2 = PB_{\mathbb{F}_{(2^4)^4}/\mathbb{F}_{2^4}}$. Last, a manual check is comparing the obtained TFB1 basis elements with the products of $B1$ and $B2$ basis elements.

Example C.3.1

```
gap> K := GF(2);; listall := FindEDPLAllfromEDL([4, 4]);;
gap> f1 := listall[1][2]; f2 := listall[1][14097];;
gap> F1 := FieldExtension(K, f1);; lambda := RootOfDefiningPolynomial(F1);;
gap> F2 := FieldExtension(F1, f2);; mu := RootOfDefiningPolynomial(F2);;
gap> edpl := [f1, f2];
[ x^4+x^3+x^2+x+Z(2)^0, x^4+Z(2^4)^7*x^2+Z(2^2)*x+Z(2)^0 ]
gap> TFB1 := GenerateTFBfromEDPLwithMB(edpl, [{"NB", "to"}, {"PB", "downto"}]);
Basis( GF(2^16), [ Z(2^16)^26922, Z(2^16)^40029, Z(2^16)^708, Z(2^16)^53136, Z(2^16)^
22317, Z(2^16)^35424, Z(2^16)^61638, Z(2^16)^48531, Z(2^16)^17712, Z(2^16)^30819,
Z(2^16)^57033, Z(2^16)^43926, Z(2^4)^3, Z(2^4)^6, Z(2^4)^12, Z(2^4)^9 ] )
gap> B1 := GenerateNB(F1, lambda);
Basis( GF(2^4), [ Z(2^4)^3, Z(2^4)^6, Z(2^4)^12, Z(2^4)^9 ] )
gap> B2 := GeneratePBdownto(F2, mu);
Basis( AsField( GF(2^4), GF(2^16) ), [ Z(2^16)^13815, Z(2^16)^9210, Z(2^16)^4605,
Z(2)^0 ] )
gap> for i in [1..4] do
> for j in [1..4] do
>   Print(TFB1[4*(i-1)+j] = B2[i]*B1[j], "\t");
> od;
> od;
true   true   true   true   true   true   true   true   true   true   true
      true   true   true   true   true
```

In Example C.3.1, the *FindEDPLAllfromEDL* method found 16320 polynomials of degree 4, that can be used as EDP f_2 for construction of $\mathbb{F}_{(2^4)^4}/\mathbb{F}_{2^4}$. Number of candidates for $\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$ is even bigger, at 32640 monic irreducible polynomials of degree 2. Number of candidates can be found using the *NrPoly(F, m)* methods (Table 7.2). \longleftrightarrow

Example C.3.2 *Multiplication expressions for a polynomial basis - two-step classic multiplication* \longleftrightarrow

This example shows in more detail how the two-step classic method was used to obtain the ANF expressions for the coordinates of the product. It is a continuation of Example 7.4.1(b) in Section 7.4. The irreducible polynomial used in this example is $f(x) = x^4 + x + 1$ with root α , yielding $PB = \{1, \alpha, \alpha^2, \alpha^3\}$. Expressions used for reduction are $\alpha^4 = \alpha + 1$, $\alpha^5 = \alpha^2 = \alpha$ and $\alpha^6 = \alpha^3 + \alpha^2$.

The two-step classic multiplication is a well known, school-book multiplication method, and can be found in numerous sources, which is why further details are omitted. `FFA_mult_2stepClassic(f, aVec, bVec, "to")` calls method `FFA_mult_convolution(vec1, vec2)`, to obtain `dexpr`, followed by `ReductionMatrixExpressionDelta(f, dexpr)`. Convolution, produces a vector of length $2m - 1$, which is why variables `d_i` are created by `ChooseFieldElmsDelta`, listed in Table 7.5. Note also that `FFA_mult_2stepClassic` call needs the direction:

- direction "to": expressions for the multiplication are obtained by $[I|R] \cdot [d_0, \dots, d_{2m-2}]^T$
- direction "downto": expressions for the multiplication are obtained by $[R|I] \cdot [d_{2m-2}, \dots, d_0]^T$

where R is the $m \times (m-1)$ reduction matrix, and I the $m \times m$ identity matrix. The two reduction matrices are computed by `ReductionMatrixDelta(f)` method listed in Table 7.2. Both `ReductionMatrixExpression` calls in Example C.3.2 use the `ReductionMatrixIR` call; the matrix produced this way is listed last.

Example C.3.2

```

gap> dexpr := FFA_mult_convolution(aVec, bVec);;
gap> for i in dexpr do Display(i); od;
a_0*b_0
a_0*b_1+a_1*b_0
a_0*b_2+a_1*b_1+a_2*b_0
a_0*b_3+a_1*b_2+a_2*b_1+a_3*b_0
a_1*b_3+a_2*b_2+a_3*b_1
a_2*b_3+a_3*b_2
a_3*b_3
gap> mult3 :=ReductionMatrixExpression(f, dexpr);;
gap> for i in mult3 do Display(i); od;
a_0*b_0+a_1*b_3+a_2*b_2+a_3*b_1
a_0*b_1+a_1*b_0+a_1*b_3+a_2*b_2+a_2*b_3+a_3*b_1+a_3*b_2
a_0*b_2+a_1*b_1+a_2*b_0+a_2*b_3+a_3*b_2+a_3*b_3
a_0*b_3+a_1*b_2+a_2*b_1+a_3*b_0+a_3*b_3
gap> mult3 = mult2sc;
true
gap> R := ReductionMatrixExpression(f, dvec);;
gap> for i in R do Display(i); od;
d_0+d_4
d_1+d_4+d_5
d_2+d_5+d_6
d_3+d_6
gap> IR := ReductionMatrixIR(f);;
gap> for i in IR do Display(VecToString(i)); od;
1000100
0100110
0010011

```

```

0001001
gap> mult2sc := FFA_mult_2stepClassic(f, avec, bvec, "to");
gap> for i in mult2sc do Display(i); od;
a_0*b_0+a_1*b_3+a_2*b_2+a_3*b_1
a_0*b_1+a_1*b_0+a_1*b_3+a_2*b_2+a_2*b_3+a_3*b_1+a_3*b_2
a_0*b_2+a_1*b_1+a_2*b_0+a_2*b_3+a_3*b_2+a_3*b_3
a_0*b_3+a_1*b_2+a_2*b_1+a_3*b_0+a_3*b_3
gap> mult = mult2sc;
true

```

←

Example C.3.3 *Squaring expressions for a polynomial basis - matrix U method* ↔

The following simple example shows how to compute the squaring expressions for a polynomial basis. The irreducible polynomial used in this example is $f(x) = x^4 + x + 1$ with root α , yielding $PB = \{1, \alpha, \alpha^2, \alpha^3\}$. The expressions for the squaring are obtained using the squaring method `FFA_sq_matrixU` (see Table 7.5 in Section 7 and Section 7.4).

Example C.3.3

```

gap> K := GF(2);; x := X(K, "x");; f := x^4+x+1;;
gap> F := FieldExtension(K, f);;
gap> PB := GeneratePB(F, RootOfDefiningPolynomial(F));;
gap> ChooseFieldElms(F);

variables
[ "a_0", "a_1", "a_2", "a_3" ]
[ "b_0", "b_1", "b_2", "b_3" ]
[ "d_0", "d_1", "d_2", "d_3", "d_4", "d_5", "d_6" ]
gap> sq := FFA_sq_matrixU(PB, avec);;
gap> for i in sq do Display(i); od;
a_0+a_2
a_2
a_1+a_3
a_3

```

←

Appendix D

The automated design entry and implementation phase

D.1 The FSRtoVHDL package

Example D.1.1 *Grain – top-level datapath: full example* \longleftrightarrow

This example shows the package `dp_pkg.vhd` in VHDL Example D.1.1(a) with type definitions for the top-level data inputs and outputs. Next are the top-level module entity `Grain_dp.vhd` in VHDL Example D.1.1(b) and architecture `Grain_dp_arch.vhd` in VHDL Example D.1.1(c).

VHDL Example D.1.1(a)

```
-----  
--- generated by GAPtoVHDL package  
--- using FSRtoVHDL package  
--- dp_pkg  
--- test for Grain  
-----  
library ieee;  
use ieee.std_logic_1164.all;  
use std.textio.all;  
use work.field_pkg.all;  
use work.fsr_pkg.all;  
  
package dp_pkg is  
  
-- top inputs  
  subtype top_in_1 is ffe;  
  constant i_data_1_zero : ffe := ffe_zero;  
  subtype top_in_2 is ffe;  
  constant i_data_2_zero : ffe := ffe_zero;  
  
-- top outputs  
  subtype top_out_1 is ffe;  
  constant o_data_1_zero : ffe := ffe_zero;  
  
end package;
```

```

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.field_pkg.all;
use work.fsr_pkg.all;
use work.dp_pkg.all;

entity Grain_dp is

    port(
        -- external data inputs
        i_data_1: in top_in_1;
        i_data_2: in top_in_2;
        -- control signals
        clk:      in std_logic; -- clock input
        fsr_1_fsr_en: in std_logic; -- fsr enable signal
        fsr_1_load:  in std_logic; -- load control signal
        fsr_1_c_ext: in std_logic; -- init control signal
        fsr_2_fsr_en: in std_logic; -- fsr enable signal
        fsr_2_load:  in std_logic; -- load control signal
        fsr_2_c_ext: in std_logic; -- init control signal
        -- 2/1 HW mux control signals
        mux_1_cntl:  in std_logic;
        mux_2_cntl:  in std_logic;
        -- external data outputs
        o_data_1:    out top_out_1
    );end entity;

```

```

--- Grain_dp_arch architecture
--- test for Grain
-----

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.field_pkg.all;
use work.fsr_pkg.all;
use work.dp_pkg.all;

architecture main of Grain_dp is
    signal o_data_1_mux_1: top_out_1;
    signal o_data_1_mux_1_src1: top_out_1;
    signal o_data_1_mux_1_src2: top_out_1;
    signal fsr_1_i_fsr: fsr_1_input;
    signal fsr_1_i_ext: fsr_1_ext;
    signal fsr_2_i_fsr: fsr_2_input;

```

```

signal fsr_2_i_ext_mux_1: fsr_2_ext;
signal fsr_2_i_ext_mux_1_src1: fsr_2_ext;
signal fsr_2_i_ext_mux_1_src2: fsr_2_ext;
signal fsr_2_i_ext: fsr_2_ext;
signal fsr_3_i_fsr: fsr_3_input;
signal fsr_3_i_ext: fsr_3_ext;
signal fsr_4_i_fsr: fsr_4_input;
signal fsr_4_i_ext: fsr_4_ext;
signal fsr_1_o_fsr: fsr_1_output;
signal fsr_2_o_fsr: fsr_2_output;
signal fsr_3_o_fsr: fsr_3_output;
signal fsr_4_o_fsr: fsr_4_output;
begin

--submodules
m_fsr_1: entity work.fsr_1(main) port map (fsr_1_i_fsr, fsr_1_i_ext, clk,
    fsr_1_fsr_en, fsr_1_load, fsr_1_c_ext, fsr_1_o_fsr);
m_fsr_2: entity work.fsr_2(main) port map (fsr_2_i_fsr, fsr_2_i_ext, clk,
    fsr_2_fsr_en, fsr_2_load, fsr_2_c_ext, fsr_2_o_fsr);
m_fsr_3: entity work.fsr_3(main) port map (fsr_3_i_fsr, fsr_3_i_ext, fsr_3_o_fsr);
m_fsr_4: entity work.fsr_4(main) port map (fsr_4_i_fsr, fsr_4_i_ext, fsr_4_o_fsr);

--assignments
o_data_1 <= o_data_1_mux_1;
o_data_1_mux_1 <= o_data_1_mux_1_src1 when mux_1_cntl = '1' else o_data_1_mux_1_src2;
o_data_1_mux_1_src1 <= fsr_4_o_fsr;
o_data_1_mux_1_src2 <= o_data_1_zero;
fsr_1_i_fsr <= i_data_1;
fsr_1_i_ext <= fsr_3_o_fsr;
fsr_2_i_fsr <= i_data_2;
fsr_2_i_ext <= fsr_2_i_ext_mux_1;
fsr_2_i_ext_mux_1 <= fsr_2_i_ext_mux_1_src1 when mux_2_cntl = '1' else
    fsr_2_i_ext_mux_1_src2;
fsr_2_i_ext_mux_1_src1 <= fsr_1_o_fsr(0);
fsr_2_i_ext_mux_1_src2 <= fsr_2_ext_zero;
fsr_3_i_fsr(0) <= fsr_1_o_fsr(1);
fsr_3_i_fsr(1) <= fsr_1_o_fsr(2);
fsr_3_i_fsr(2) <= fsr_1_o_fsr(3);
fsr_3_i_fsr(3) <= fsr_1_o_fsr(4);
fsr_3_i_fsr(4) <= fsr_2_o_fsr(1);
fsr_3_i_ext <= fsr_2_o_fsr(0);
fsr_4_i_fsr <= fsr_4_o_fsr;
fsr_4_i_ext <= fsr_1_o_fsr(0);

end architecture;

```

←

D.2 The CIRCUIT package part 1

Example D.2.1 Using different tower field constructions - continued \longleftrightarrow

The diagram in Figure 13.3 shows how different constructions of the finite field $\mathbb{F}_{((2^2)^2)^2}$. The first and the last level of the tower are constructed in the same manner, using the same (only) irreducible polynomial and the same basis for the representation of the elements. The middle level, however, is constructed in two different ways, yielding two different constructions. The construction on the left branch of the diagram in Figure 13.3 is using the extension field defining polynomial $f_{2,1}(x) = x^2 + \lambda x + 1$, and the construction on the right branch of the extension field defining polynomial $f_{2,2}(x) = x^2 + \lambda x + \lambda$. The two polynomials have different roots $\mu_1 \neq \mu_2$, where $f_{2,1}(\mu_1) = 0$ and $f_{2,2}(\mu_2) = 0$. For clarity, all the constants are given w.r.t. a reference field defining polynomial RDP, listed in Table D.1.

Reference finite field	Reference field defining polynomial - RDP $p_i(x)$	Root of RDP $p_i(x)$	Constants w.r.t. root of RDP
\mathbb{F}_{2^8}	$p_3(x) = x^8 + x^4 + x^3 + x^2 + 1$	$p_3(\nu) = 0$	$\nu_3 = \nu^{15}$
\mathbb{F}_{2^4}	$p_2(x) = x^4 + x + 1$	$p_2(\mu) = 0$	$\mu_1 = \mu^6, \mu_2 = \mu^7$
\mathbb{F}_{2^2}	$p_1(x) = x^2 + x + 1$	$p_1(\lambda) = 0$	

Table D.1: Reference polynomials and their roots

The distinction between the left and the right branch is very important: the submodules for computation using basis B_1 will differ from the submodules for computation using B_2 . The `UnderlyingField` attribute stores the information about the degree of extension for each level, e.g., `AsField(AsField(GF(2^2), GF(2^4)), GF(2^8))`, but not about the “per-level” bases used. This information is captured by `SubSGDtower`.

The GAP Example D.2.1 shows the two tower field constructions from diagram in Figure 13.3. As the counter r for the signal domains used in this example will change significantly when this example will be used in the future, to avoid confusion, the subscripts r are labelled with letters, rather than numbers, e.g., $\mathcal{S}_1 \rightarrow \mathcal{S}$, $\mathcal{S}_2 \rightarrow \mathcal{S}_A$, ... Hence the fields, EDPs and bases are distinguished with an alphabet letter: A, B, C, D, E. Same letters are used for the corresponding bases and `SignalDomain` objects `sgd`:

$$\begin{array}{l}
 \text{The left branch: } \mathbb{F}_2 \xrightarrow{f_1(x)} \mathbb{F}_{2^2} \xrightarrow{f_{2,1}(x)} \mathbb{F}_{(2^2)^2} \xrightarrow{f_3(x)} \mathbb{F}_{((2^2)^2)^2} \quad \Rightarrow \quad \mathbb{K} \xrightarrow{fa} \mathbb{FA} \xrightarrow{fb} \mathbb{FB} \xrightarrow{fd} \mathbb{FD} \\
 \text{The right branch: } \mathbb{F}_2 \xrightarrow{f_1(x)} \mathbb{F}_{2^2} \xrightarrow{f_{2,2}(x)} \mathbb{F}_{(2^2)^2} \xrightarrow{f_3(x)} \mathbb{F}_{((2^2)^2)^2} \quad \Rightarrow \quad \mathbb{K} \xrightarrow{fa} \mathbb{FA} \xrightarrow{fc} \mathbb{FC} \xrightarrow{fe} \mathbb{FE}
 \end{array}$$

The top-level signal domains `sgdD` and `sgdE` are compared at the end of the example: they only differ in the third entry of their `SubSGDtower` attributes, all other attributes, including the `UnderlyingField` attribute, are the same.

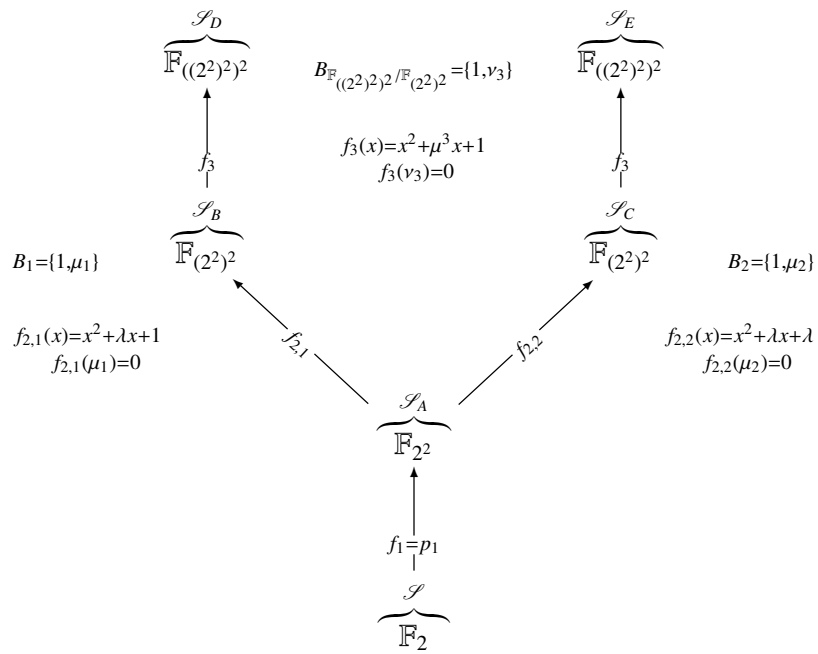


Figure D.1: Two different tower field constructions

```

gap> K := GF(2);; x := X(K, "x");;
gap> sgd := SignalDomain(K);;
warning: SubSGDtower is -1
gap> fa := x^2+x+Z(2)^0;; FA := FieldExtension(K, fa);;
gap> BA := GeneratePB(FA, RootOfDefiningPolynomial(FA));;
gap> sgdA := SignalDomain(sgd, BA, "to");;
gap> fb := x^2+Z(2^2)*x+Z(2)^0;; FB := FieldExtension(FA, fb);;
gap> BB := GeneratePB(FB, RootOfDefiningPolynomial(FB));;
gap> sgdB := SignalDomain(sgdA, BB, "to");;
gap> fc := x^2+Z(2^2)*x+Z(2^2);; FC := FieldExtension(FA, fc);;
gap> BC := GeneratePB(FC, RootOfDefiningPolynomial(FC));;
gap> sgdC := SignalDomain(sgdA, BC, "to");;
gap> fd := x^2+Z(2^4)^3*x+Z(2)^0;; FD := FieldExtension(FB, fd);;
gap> BD := GeneratePB(FD, RootOfDefiningPolynomial(FD));;
gap> sgdD := SignalDomain(sgdB, BD, "to");;
gap> fe := x^2+Z(2^4)^3*x+Z(2)^0;; FE := FieldExtension(FC, fe);;
gap> BE := GeneratePB(FE, RootOfDefiningPolynomial(FE));;
gap> sgdE := SignalDomain(sgdC, BE, "to");;
gap> sgdD;
< signal domain AsField( AsField( GF(2^2), GF(2^4) ), GF(2^8) ) with basis [ Z(2)^0,
  Z(2^8)^15 ] and direction to of length 2 >
gap> sgdE;
< signal domain AsField( AsField( GF(2^2), GF(2^4) ), GF(2^8) ) with basis [ Z(2)^0,
  Z(2^8)^15 ] and direction to of length 2 >
gap> for i in SubSGDtower(sgdD) do Print(i, "\n"); od;
signal domain GF(2) of length 1
signal domain GF(2^2) with basis [ Z(2)^0, Z(2^2) ] and direction to of length 2
signal domain AsField( GF(2^2), GF(2^4) ) with basis [ Z(2)^0, Z(2^4)^6 ] and
direction to of length 2
gap> for i in SubSGDtower(sgdE) do Print(i, "\n"); od;
signal domain GF(2) of length 1
signal domain GF(2^2) with basis [ Z(2)^0, Z(2^2) ] and direction to of length 2
signal domain AsField( GF(2^2), GF(2^4) ) with basis [ Z(2)^0, Z(2^4)^7 ] and
direction to of length 2
gap> sgdD = sgdE;
false
gap> UnderlyingField(sgdD) = UnderlyingField(sgdE);
true
gap> UnderlyingBasis(sgdD) = UnderlyingBasis(sgdE);
true
gap> Length(sgdD) = Length(sgdE); BasisDirection(sgdD) = BasisDirection(sgdE);
true
true

```

←

Example D.2.2 *SignalDomain and SIGNAL objects* \longleftrightarrow

This example specifies two signal domains and three signals:

- signal domain \mathcal{S}_1 sgd1 for \mathbb{F}_2 , for \mathcal{F}_1 elements sig1 and sig3
- signal domain \mathcal{S}_2 sgd2 for \mathbb{F}_{2^4} , for “just vector” \mathcal{V}_2^5 labelled sig2

The signal domain sgd1 belongs to the prime field \mathbb{F}_2 , hence has -1 for SubSGDtower (see warning). The signal domain sgd2 stores only sgd1 as its SubSGDtower, unlike sgdB, sgdC, sgdD and sgdE shown in Example 13.2.1. sig2 represents a vector of length 5 with coordinates from \mathcal{S}_2 . It has no interpretation basis (“NA”) but it needs a direction (for VHDL). For manipulation in GAP the signal domain sgd2 is sufficient. For the actual implementation, a signal \mathcal{V}_2 , for the representation of \mathcal{S}_2 as a vector space, would be added by the “list” methods from Table 13.5 (see note [4]). Rest of Example D.2.2 shows the SIGNAL types and their comparisons. All three signals also have labels.

Example D.2.2

```

gap> K := GF(2);; x := X(K,"x");; f := x^4 + x^3 + 1;; F := FieldExtension(K, f);;
gap> w := RootOfDefiningPolynomial(F);; NB := GenerateNB(F, w);;
gap> sgd1 := SignalDomain(K);; sig1 := SIGNAL(sgd1, "sig1");
warning: SubSGDtower is -1
< SIGNAL with label "sig1" of length 1 over GF(2) >
gap> sgd2 := SignalDomain(sgd1, NB, "to");;
gap> SubSGDtower(sgd1); SubSGDtower(sgd2);
-1
[ < signal domain GF(2) of length 1 > ]
gap> sig2 := SIGNAL(sgd2, "sig2", 5);
< SIGNAL with label "sig2" of length 5 over GF(2^4) with direction to >
gap> UnderlyingField(UnderlyingSignalDomain(sig2));
GF(2^4)
gap> UnderlyingSignalDomain(sig2);
< signal domain GF(2^4) with basis [ Z(2^4)^7, Z(2^4)^14, Z(2^4)^13, Z(2^4)^11 ]
and direction to of length 4 >
gap> WhichBasis(sig2); WhichBasisDir(sig2); Length(sig2);
"NA"
"to"
5
gap> SameSIGNALType(sig1, sig2);
false
gap> sig3 := SGDToSIGNAL(sgd1); SameSIGNALType(sig1, sig3);
< SIGNAL with label "" of length 1 over GF(2) >
true
gap> ExtractSIGNALType(sig1);
[ < signal domain GF(2) of length 1 >, "NA", "NA", 1 ]
gap> ExtractSIGNALType(sig2);
[ < signal domain GF(2^4) with basis [ Z(2^4)^7, Z(2^4)^14, Z(2^4)^13, Z(2^4)^11 ]
and direction to of length 4 >, "NA", "to", 5 ]
gap> ExtractSIGNALType(sig3);
[ < signal domain GF(2) of length 1 >, "NA", "NA", 1 ]

```

\longleftarrow

Example D.2.3 *SignalPkg* example for \mathbb{F}_{2^4} \longleftrightarrow

This is the first example of signal package objects, showing simple entry methods and a detailed print method called `PrintAllDownto`. It demonstrates the `DefaultSignalPkg(m)` call, which builds the finite field \mathbb{F}_{2^m} using the (GAP pre-stored) irreducible polynomial $f(x) = x^4 + x + 1$ and polynomial basis $\{1, \omega, \omega^2, \omega^3\}$, where ω is the root the defining polynomial, i.e., $f(\omega) = 0$. In the GAP code Example D.2.3, ω is `Z(2^4)`. Default direction used for all array types is “to”. The highest field identifier `Fid=4` signal $\mathcal{F}_2^1/\mathcal{F}_1^1$ represents the elements of the finite field \mathbb{F}_{2^4} . Its vector equivalent is signal $\mathcal{V}_1^4/\mathcal{V}_1^1$ with `Fid=2`, see `FieldIDToVectorID`. Note that for the four signals in the `defaultPkg`, there are only two signal domains, corresponding to the two finite fields, \mathbb{F}_2 and \mathbb{F}_{2^4} .

Example D.2.3

```

gap> m := 4;; defaultPkg := DefaultSignalPkg(m);
warning: SubSGDtower is -1
warning: SubSGDtower is -1
gap> PrintDownto(defaultPkg);

-----
SignalPkg with
4:      elements in      GF(2^4) with direction to basis [ Z(2)^0, Z(2^4), Z(2^4)^2,
                                Z(2^4)^3 ]
3:      vectors over     GF(2)   of length 7 and direction to
2:      vectors over     GF(2)   of length 4 and direction to - interpretation basis
                                [ Z(2)^0, Z(2^4), Z(2^4)^2, Z(2^4)^3 ]
1:      elements in      GF(2)
-----

gap> FieldIDToVectorID(defaultPkg, 4); FieldIDToSubID(defaultPkg, 4);
2
1
gap> for i in SignalDomainList(defaultPkg) do Print(i, "\n"); od;
signal domain GF(2) of length 1
signal domain GF(2^4) with basis [ Z(2)^0, Z(2^4), Z(2^4)^2, Z(2^4)^3 ] and
direction to of length 4

```

\longleftrightarrow

Example D.2.4 *SignalPkg example for different tower field constructions of $\mathbb{F}_{((2^2)^2)^2} \longleftrightarrow$*

This example shows a signal package containing both branches of the diagram in Figure D.1; it is a continuation of example Example 13.3.1. For clarity, values of the roots and bases are given in GAP native form in Table 13.10.

$$\mathbb{F}_2 \xrightarrow{f_1(x)} \mathbb{F}_{2^2} \xrightarrow{f_{2,1}(x)} \mathbb{F}_{(2^2)^2} \xrightarrow{f_3(x)} \mathbb{F}_{((2^2)^2)^2} \Rightarrow \mathbb{K} \xrightarrow{fa} FA \xrightarrow{fb} FB \xrightarrow{fd} FD$$

Finite field	EDP	Polynomial basis	Comments and GAP example labels
\mathbb{F}_{q^2}	$f_i(x)$	$B_{\mathbb{F}_{q^2}/\mathbb{F}_q} = \{1, \rho\}$	$f_i(\rho) = 0$ field polynomial basis
$\mathbb{F}_{((2^2)^2)^2}$	$f_3(x)$	$B_{\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}} = \{1, \nu_3\}$	$f_3(\nu_3) = 0$ FD fd BD
$\mathbb{F}_{(2^2)^2}$	$f_{2,1}(x)$	$B_1 = \{1, \mu_1\}$	$f_{2,1}(\mu_1) = 0$ FB fb BB
\mathbb{F}_{2^2}	$f_1(x)$	$B_{\mathbb{F}_{2^2}/\mathbb{F}_2} = \{1, \lambda\}$	$f_1(\lambda) = 0$ FA fa BA

Table D.2: Tower construction of $\mathbb{F}_{((2^2)^2)^2}$ - left branch of diagram in Figure D.1 (see Example 13.3.1)

$$\mathbb{F}_2 \xrightarrow{f_1(x)} \mathbb{F}_{2^2} \xrightarrow{f_{2,2}(x)} \mathbb{F}_{(2^2)^2} \xrightarrow{f_3(x)} \mathbb{F}_{((2^2)^2)^2} \Rightarrow \mathbb{K} \xrightarrow{fa} FA \xrightarrow{fc} FC \xrightarrow{fe} FE$$

Finite field	EDP	Polynomial basis	Comments and GAP example labels
\mathbb{F}_{q^2}	$f_i(x)$	$B_{\mathbb{F}_{q^2}/\mathbb{F}_q} = \{1, \rho\}$	$f_i(\rho) = 0$ field polynomial basis
$\mathbb{F}_{((2^2)^2)^2}$	$f_3(x)$	$B_{\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}} = \{1, \nu_3\}$	$f_3(\nu_3) = 0$ FE fe BE
$\mathbb{F}_{(2^2)^2}$	$f_{2,2}(x)$	$B_2 = \{1, \mu_2\}$	$f_{2,2}(\mu_2) = 0$ FC fc BC
\mathbb{F}_{2^2}	$f_1(x)$	$B_{\mathbb{F}_{2^2}/\mathbb{F}_2} = \{1, \lambda\}$	$f_1(\lambda) = 0$ FA fa BA

Table D.3: Tower construction of $\mathbb{F}_{((2^2)^2)^2}$ - right branch of diagram in Figure D.1

$\nu_3 = Z(2^8)^{15}$	BD := [Z(2)^0, Z(2^8)^15]
$\mu_1 = Z(2^4)^6$	BB := [Z(2)^0, Z(2^4)^6]
$\lambda = Z(2^2)$	BA := [Z(2)^0, Z(2^2)]
$TFB_{1, \mathbb{F}_{2^8}/\mathbb{F}_2} =$	[Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11, Z(2^8)^15, Z(2^8)^100, Z(2^8)^117, Z(2^8)^202]
$TFB_{1, \mathbb{F}_{2^4}/\mathbb{F}_2} =$	[Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11]

Table D.4: Tower construction of $\mathbb{F}_{((2^2)^2)^2}$ - roots and bases in GAP native representation for the left branch of diagram in Figure D.1 (see Example 13.3.1)

$\nu_3 = Z(2^8)^{15}$	BE := [Z(2)^0, Z(2^8)^15]
$\mu_2 = Z(2^4)^7$	BC := [Z(2)^0, Z(2^4)^7]
$\lambda = Z(2^2)$	BA := [Z(2)^0, Z(2^2)]
$TFB_{2, \mathbb{F}_{2^8}/\mathbb{F}_2} =$	[Z(2)^0, Z(2^2), Z(2^4)^7, Z(2^4)^12, Z(2^8)^15, Z(2^8)^100, Z(2^8)^134, Z(2^8)^219]
$TFB_{2, \mathbb{F}_{2^4}/\mathbb{F}_2} =$	[Z(2)^0, Z(2^2), Z(2^4)^7, Z(2^4)^12]

Table D.5: Tower construction of $\mathbb{F}_{((2^2)^2)^2}$ - roots and bases in GAP native representation for the right branch of diagram in Figure D.1

Example 13.3.2

```
gap> Blist := [BA, BC, BE];;
gap> AddTowerFieldToSignalPkg(defaultPkg, Blist, ["to", "to", "to"]);;
gap> PrintAllDownto(defaultPkg);
```

SignalPkg with

```
19:      elements in      AsField( AsField( GF(2^2), GF(2^4) ), GF(2^8) ) with direction
to basis [ Z(2)^0, Z(2^8)^15 ]
18:      elements in      GF(2^8) with direction to basis [ Z(2)^0, Z(2^2), Z(2^4)^7,
Z(2^4)^12, Z(2^8)^15, Z(2^8)^100, Z(2^8)^134,
Z(2^8)^219 ]
17:      elements in      AsField( AsField( GF(2^2), GF(2^4) ), GF(2^8) ) with direction
to basis [ Z(2)^0, Z(2^8)^15 ]
16:      elements in      GF(2^8) with direction to basis [ Z(2)^0, Z(2^2), Z(2^4)^6,
Z(2^4)^11, Z(2^8)^15, Z(2^8)^100, Z(2^8)^117,
Z(2^8)^202 ]
15:      vectors over     AsField( GF(2^2), GF(2^4) )      of length 2 and direction to -
interpretation basis [ Z(2)^0, Z(2^8)^15 ]
14:      elements in      AsField( GF(2^2), GF(2^4) )      with direction to basis
[ Z(2)^0, Z(2^4)^7 ]
13:      vectors over     AsField( GF(2^2), GF(2^4) )      of length 2 and direction to -
interpretation basis [ Z(2)^0, Z(2^8)^15 ]
12:      elements in      GF(2^4) with direction to basis [ Z(2)^0, Z(2^2), Z(2^4)^7,
Z(2^4)^12 ]
11:      elements in      AsField( GF(2^2), GF(2^4) )      with direction to basis
[ Z(2)^0, Z(2^4)^6 ]
10:      elements in      GF(2^4) with direction to basis [ Z(2)^0, Z(2^2), Z(2^4)^6,
Z(2^4)^11 ]
9:      vectors over     GF(2^2) of length 2 and direction to - interpretation basis
[ Z(2)^0, Z(2^4)^7 ]
8:      vectors over     GF(2^2) of length 2 and direction to - interpretation basis
[ Z(2)^0, Z(2^4)^6 ]
7:      elements in      GF(2^2) with direction to basis [ Z(2)^0, Z(2^2) ]
6:      vectors over     GF(2)   of length 8 and direction to - interpretation basis
[ Z(2)^0, Z(2^2), Z(2^4)^7, Z(2^4)^12, Z(2^8)^15,
Z(2^8)^100, Z(2^8)^134, Z(2^8)^219 ]
5:      vectors over     GF(2)   of length 8 and direction to - interpretation basis
[ Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11, Z(2^8)^15,
Z(2^8)^100, Z(2^8)^117, Z(2^8)^202 ]
4:      vectors over     GF(2)   of length 4 and direction to - interpretation basis
[ Z(2)^0, Z(2^2), Z(2^4)^7, Z(2^4)^12 ]
3:      vectors over     GF(2)   of length 4 and direction to - interpretation basis
[ Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11 ]
2:      vectors over     GF(2)   of length 2 and direction to - interpretation basis
[ Z(2)^0, Z(2^2) ]
1:      elements in      GF(2)
```

DETAILS
signal domains:

- 10: signal domain $\text{AsField}(\text{AsField}(\text{GF}(2^2), \text{GF}(2^4)), \text{GF}(2^8))$ with basis $[Z(2)^0, Z(2^8)^{15}]$ and direction to of length 2
- 9: signal domain $\text{AsField}(\text{GF}(2^2), \text{GF}(2^4))$ with basis $[Z(2)^0, Z(2^4)^7]$ and direction to of length 2
- 8: signal domain $\text{GF}(2^8)$ with basis $[Z(2)^0, Z(2^2), Z(2^4)^7, Z(2^4)^{12}, Z(2^8)^{15}, Z(2^8)^{100}, Z(2^8)^{134}, Z(2^8)^{219}]$ and direction to of length 8
- 7: signal domain $\text{GF}(2^4)$ with basis $[Z(2)^0, Z(2^2), Z(2^4)^7, Z(2^4)^{12}]$ and direction to of length 4
- 6: signal domain $\text{AsField}(\text{AsField}(\text{GF}(2^2), \text{GF}(2^4)), \text{GF}(2^8))$ with basis $[Z(2)^0, Z(2^8)^{15}]$ and direction to of length 2
- 5: signal domain $\text{AsField}(\text{GF}(2^2), \text{GF}(2^4))$ with basis $[Z(2)^0, Z(2^4)^6]$ and direction to of length 2
- 4: signal domain $\text{GF}(2^8)$ with basis $[Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^{11}, Z(2^8)^{15}, Z(2^8)^{100}, Z(2^8)^{117}, Z(2^8)^{202}]$ and direction to of length 8
- 3: signal domain $\text{GF}(2^4)$ with basis $[Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^{11}]$ and direction to of length 4
- 2: signal domain $\text{GF}(2^2)$ with basis $[Z(2)^0, Z(2^2)]$ and direction to of length 2
- 1: signal domain $\text{GF}(2)$ of length 1

signals:

- 19: SIGNAL with label "" of length 1 over $\text{AsField}(\text{AsField}(\text{GF}(2^2), \text{GF}(2^4)), \text{GF}(2^8))$
- 18: SIGNAL with label "" of length 1 over $\text{GF}(2^8)$
- 17: SIGNAL with label "" of length 1 over $\text{AsField}(\text{AsField}(\text{GF}(2^2), \text{GF}(2^4)), \text{GF}(2^8))$
- 16: SIGNAL with label "" of length 1 over $\text{GF}(2^8)$
- 15: SIGNAL with label "" of length 2 over $\text{AsField}(\text{GF}(2^2), \text{GF}(2^4))$ with direction to with interpretation basis $[Z(2)^0, Z(2^8)^{15}]$
- 14: SIGNAL with label "" of length 1 over $\text{AsField}(\text{GF}(2^2), \text{GF}(2^4))$
- 13: SIGNAL with label "" of length 2 over $\text{AsField}(\text{GF}(2^2), \text{GF}(2^4))$ with direction to with interpretation basis $[Z(2)^0, Z(2^8)^{15}]$
- 12: SIGNAL with label "" of length 1 over $\text{GF}(2^4)$
- 11: SIGNAL with label "" of length 1 over $\text{AsField}(\text{GF}(2^2), \text{GF}(2^4))$
- 10: SIGNAL with label "" of length 1 over $\text{GF}(2^4)$
- 9: SIGNAL with label "" of length 2 over $\text{GF}(2^2)$ with direction to with interpretation basis $[Z(2)^0, Z(2^4)^7]$
- 8: SIGNAL with label "" of length 2 over $\text{GF}(2^2)$ with direction to with interpretation basis $[Z(2)^0, Z(2^4)^6]$
- 7: SIGNAL with label "" of length 1 over $\text{GF}(2^2)$
- 6: SIGNAL with label "" of length 8 over $\text{GF}(2)$ with direction to with interpretation basis $[Z(2)^0, Z(2^2), Z(2^4)^7, Z(2^4)^{12}, Z(2^8)^{15}, Z(2^8)^{100}, Z(2^8)^{134}, Z(2^8)^{219}]$
- 5: SIGNAL with label "" of length 8 over $\text{GF}(2)$ with direction to with interpretation basis $[Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^{11}, Z(2^8)^{15}, Z(2^8)^{100}, Z(2^8)^{117}, Z(2^8)^{202}]$
- 4: SIGNAL with label "" of length 4 over $\text{GF}(2)$ with direction to with interpretation basis $[Z(2)^0, Z(2^2), Z(2^4)^7, Z(2^4)^{12}]$
- 3: SIGNAL with label "" of length 4 over $\text{GF}(2)$ with direction to with interpretation basis $[Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^{11}]$
- 2: SIGNAL with label "" of length 2 over $\text{GF}(2)$ with direction to with interpretation basis $[Z(2)^0, Z(2^2)]$

1: SIGNAL with label "" of length 1 over GF(2)

signal types (sgd w.r.t. SignalPkg) + sub-signal:

```
19: [ sgd # 10, NA, NA, 1]          sub = 14
18: [ sgd # 8, NA, NA, 1]          sub = 1
17: [ sgd # 6, NA, NA, 1]          sub = 11
16: [ sgd # 4, NA, NA, 1]          sub = 1
15: [ sgd # 9, Basis( AsField( AsField( GF(2^2), GF(2^4) ), GF(2^8) ), [ Z(2)^0,
    Z(2^8)^15 ] ), to, 2]          sub = 9
14: [ sgd # 9, NA, NA, 1]          sub = 7
13: [ sgd # 5, Basis( AsField( AsField( GF(2^2), GF(2^4) ), GF(2^8) ), [ Z(2)^0,
    Z(2^8)^15 ] ), to, 2]          sub = 8
12: [ sgd # 7, NA, NA, 1]          sub = 1
11: [ sgd # 5, NA, NA, 1]          sub = 7
10: [ sgd # 3, NA, NA, 1]          sub = 1
9: [ sgd # 2, Basis( AsField( GF(2^2), GF(2^4) ), [ Z(2)^0, Z(2^4)^7 ] ), to, 2]
    sub = 2
8: [ sgd # 2, Basis( AsField( GF(2^2), GF(2^4) ), [ Z(2)^0, Z(2^4)^6 ] ), to, 2]
    sub = 2
7: [ sgd # 2, NA, NA, 1]          sub = 1
6: [ sgd # 1, Basis( GF(2^8), [ Z(2)^0, Z(2^2), Z(2^4)^7, Z(2^4)^12, Z(2^8)^15,
    Z(2^8)^100, Z(2^8)^134, Z(2^8)^219 ] ), to, 8]          sub = 1
5: [ sgd # 1, Basis( GF(2^8), [ Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11, Z(2^8)^15,
    Z(2^8)^100, Z(2^8)^117, Z(2^8)^202 ] ), to, 8]          sub = 1
4: [ sgd # 1, Basis( GF(2^4), [ Z(2)^0, Z(2^2), Z(2^4)^7, Z(2^4)^12 ] ), to, 4]
    sub = 1
3: [ sgd # 1, Basis( GF(2^4), [ Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11 ] ), to, 4]
    sub = 1
2: [ sgd # 1, Basis( GF(2^2), [ Z(2)^0, Z(2^2) ] ), to, 2]          sub = 1
1: [ sgd # 1, NA, NA, 1]          sub = -1
```

←

D.3 The CIRCUIT package part 2

Example D.3.1 *Functional description for the running example (13.1) expression defined over the tower field $\mathbb{F}_{((2^2)^2)^2} \hookrightarrow$*

The Example D.3.1 shows the AlgFunctionality object AF, for the implementation of running example (13.1) defined over the tower field $\mathbb{F}_{((2^2)^2)^2}$ (Example 13.1.3 in Section 13.1):

$$z = \gamma_1 a \cdot b \cdot c + a^2 + \gamma_2$$

where $\gamma_1 = \nu$ and $\gamma_2 = \nu^3$ and ν is a root of the reference defining polynomial $p_3(x) = x^8 + x^4 + x^3 + x^2 + 1$ (see Table D.1 from Example 13.2.1). The exact tower field construction is specified in Example 13.3.1 and the SignalPkg defaultPkg shown in a diagram in Figure 13.5 in Example 13.3.2, in Section 13.3.

For the created AlgFunctionality object AF, the collector method AlgFunctionalityCollectSignals is called, followed by the method PrintAlgFunctionalityCollectSignals, which shows the collected signals, printed in a human friendly fashion. For each edge, the start and end node are printed as well.

At the end of the example, the method MakeAlgFunctionalityDesignReady is called, and the contents of the submodules component shown. The *Fid* of the output port is stored together with submodules from this field.

Example D.3.1

```
gap> random := Z(2^8)*a_0*a_1*a_2+a_0^2+Z(2^8)^3;;
gap> entitylist := ["random"];; ct := ["CICO", "simple" ];;
gap> inputports := [ ["i_a", 11], ["i_b", 11], ["i_c", 11] ];;
gap> outputports := [ ["o_z", 11] ];;
gap> portlist := [defaultPkg, inputports, outputports];;
gap> bindlist := [ ["a_0", "i_a"], ["a_1", "i_b"], ["a_2", "i_c"] ];;
gap> exprlist := [random];;
gap> archlist := [bindlist, exprlist];;
gap> AF := AlgFunctionality(entitylist, portlist, archlist, ct);;
gap> AlgFunctionalityCollectSignals(AF, defaultPkg);;
gap> PrintAlgFunctionalityCollectSignals(AF);
```

DETAILS for algfun:

```
<
AlgFunctionality of type [ "CICO", "simple", -1, -1, -1 ] :
- FunctName: blackbox
- EntityName: random
- ArchName: main
- InputPorts: [ "i_a", 11 ], [ "i_b", 11 ], [ "i_c", 11 ],
- OutputPorts: [ "o_z", 11 ],
inputport binding: [ [ "a_0", "i_a" ], [ "a_1", "i_b" ], [ "a_2", "i_c" ] ]
implementing: [ [ Z(2^8)*a_0*a_1*a_2+a_0^2+Z(2^8)^3 ] ]
>
```

```
----- output =1, coordinate =1 siglist -----
```

```

expcount= 1, IsEmpty(explist)? false
explist= [ [ "EXPout_0_0", 2, 1, 0 ] ]
IsEmpty(multree)? false
depth = 3

```

printing for lvl 3

```

[ "out_0_0_mon0_lvl3_node0", "out_0_0_mon0_lvl3_node1", "out_0_0_mon0_lvl3_node2" ]
edge:          out_0_0_mon0_lvl3_edge0
                <out_0_0_mon0_lvl3_node0 , out_0_0_mon0_lvl2_node0>
edge:          out_0_0_mon0_lvl3_edge1
                <out_0_0_mon0_lvl3_node1 , out_0_0_mon0_lvl2_node0>
edge:          out_0_0_mon0_lvl3_edge2
                <out_0_0_mon0_lvl3_node2 , out_0_0_mon0_lvl2_node1>

```

printing for lvl 2

```

[ "out_0_0_mon0_lvl2_node0", "out_0_0_mon0_lvl2_node1" ]
edge:          out_0_0_mon0_lvl2_edge0
                <out_0_0_mon0_lvl2_node0 , out_0_0_mon0_lvl1_node0>
edge:          out_0_0_mon0_lvl2_edge1
                <out_0_0_mon0_lvl2_node1 , out_0_0_mon0_lvl1_node0>

```

printing for lvl 1

```

[ "out_0_0_mon0_lvl1_node0" ]
edge:          out_0_0_mon0_lvl1_edge0
                <out_0_0_mon0_lvl1_node0 , 0>

```

```

depth = 1
MVount= 1, IsEmpty(MVlist)? false
MVlist= [ [ "MVout_0_0", Z(2^8), 0 ] ]
ccount= 1, IsEmpty(constlist)? false
constlist= [ [ "out_0_0_c_g_219", Z(2^8)^3, 2 ] ]
number of sumlvl=3
IsEmpty(sumtree)? false
depth = 3

```

printing for lvl 3

```

[ "out_0_0_term_lvl3_node0", "out_0_0_term_lvl3_node1", "out_0_0_term_lvl3_node2" ]
edge:          out_0_0_term_lvl3_edge0
                <out_0_0_term_lvl3_node0 , out_0_0_term_lvl2_node0>
edge:          out_0_0_term_lvl3_edge1
                <out_0_0_term_lvl3_node1 , out_0_0_term_lvl2_node0>
edge:          out_0_0_term_lvl3_edge2
                <out_0_0_term_lvl3_node2 , out_0_0_term_lvl2_node1>

```


printing for lvl 2

```
[ "out_0_0_term_lvl2_node0", "out_0_0_term_lvl2_node1" ]  
edge:          out_0_0_term_lvl2_edge0  
                <out_0_0_term_lvl2_node0 , out_0_0_term_lvl1_node0>  
edge:          out_0_0_term_lvl2_edge1  
                <out_0_0_term_lvl2_node1 , out_0_0_term_lvl1_node0>
```

printing for lvl 1

```
[ "out_0_0_term_lvl1_node0" ]  
edge:          out_0_0_term_lvl1_edge0  
                <out_0_0_term_lvl1_node0 , 0>
```

end of DETAILS

```
gap> MakeAlgFunctionalityDesignReady(AF, defaultPkg);;
```

```
gap> AFsubmodules;
```

```
[ [ 11 ], [ [ [ "EXP", 2 ], "MforFid11", [ "MV", Z(2^8) ] ] ] ]
```



D.4 The CIRCUIT package part 3

Example D.4.1 *Datapath synthesis for an expression over $\mathbb{F}_{((2^2)^2)^2}$ – bottom-up processing continued* \longleftrightarrow

This example continues the Example 15.3.1, the bottom-up processing of the design, which resulted in a VHDL-ready design. The GAP code Example D.4.1 shows the CIRClst contents (listed by their class, archetype, and the first two entries of the parsed submodule list (15.1), namely `FunctName` and `EntityName`) after processing, followed by the AFlist and SMList. The `PrintCIRClst` method only displays the class and archetype of each Circuit object, and their `FunctName` and `EntityName`, which are also the first two entries of each parsed submodule list (15.1) and will appear as such in Circuits at higher Fids.

Example D.4.1

```
gap> PrintCIRClst(design);

-----
AlgDesign with CIRClst

11:
10:
9:
class = 7      [ "CICO", "simple", -1, -1, -1 ]      [[ "EXP", 2 ], Fid9_exp_2]
class = 5      [ "CICO", "simple", -1, -1, -1 ]      [MforFid11, Fid9_mult]
class = 2      [ "CICO", "simple", -1, -1, -1 ]      [[ "MV", Z(2^8) ], Fid9_mv_g_73]
8:
7:
6:
class = 7      [ "CICO", "simple", -1, -1, -1 ]      [[ "EXP", 2 ], Fid6_exp_2]
class = 5      [ "CICO", "simple", -1, -1, -1 ]      [MforFid9, Fid6_mult]
class = 2      [ "CICO", "simple", -1, -1, -1 ]      [[ "MV", Z(2^4)^7 ], Fid6_mv_g_7]
class = 2      [ "CICO", "simple", -1, -1, -1 ]      [[ "MV", Z(2^2) ], Fid6_mv_g_5]
class = 2      [ "CICO", "simple", -1, -1, -1 ]      [[ "MV", Z(2^4)^9 ], Fid6_mv_g_9]
class = 2      [ "CICO", "simple", -1, -1, -1 ]      [[ "MV", Z(2^4)^2 ], Fid6_mv_g_2]
class = 2      [ "CICO", "simple", -1, -1, -1 ]      [[ "MV", Z(2^4) ], Fid6_mv_g_1]
5:
4:
3:
2:
class = 1      [ "CICO", "simple", -1, -1, -1 ]      [[ "EXP", 2 ], Fid2_exp_2]
class = 3      [ "CICO", "simple", -1, -1, -1 ]      [MforFid6, Fid2_mult]
class = 1      [ "CICO", "simple", -1, -1, -1 ]      [[ "MV", Z(2^2) ], Fid2_mv_g_1]
class = 1      [ "CICO", "simple", -1, -1, -1 ]      [[ "MV", Z(2^2)^2 ], Fid2_mv_g_2]
1:
-----
gap> PrintAFlist(design); PrintSMList(design);

-----
AlgDesign with empty AFlist

-----
```

AlgDesign with empty SList



D.5 The CIRCUIT package part 4

Example D.5.1 A *full* field_pkg.vhd *example for* $\mathbb{F}_{((2^2)^2)^2} \longleftrightarrow$

This the example of the field_pkg.vhd for the signal package defaultPkg used in Examples 13.3.1 and 13.3.2.

VHDL Example D.5.1

```
-----  
--- generated by GAPtoVHDL package  
--- field_pkg  
--- exampleTF8.g  
-----  
library ieee;  
use ieee.std_logic_1164.all;  
use std.textio.all;  
  
package field_pkg is  
  
-----  
--- field_pkg params for a binary field  
--- with field identifier Fid = 1  
  
    subtype ffe_1 is std_logic;  
  
--- element 1 in chosen basis:  
    constant ffe_1_one: ffe_1 := '1';  
--- element 0 in chosen basis:  
    constant ffe_1_zero: ffe_1 := '0';  
-----  
-----  
--- field_pkg params for vector  
--- with identifier Fid = 2  
--- of length 2  
--- and direction "to"  
--- with interpretation basis  
--- [ Z(2)^0, Z(2^2) ]  
--- over sub-id = 1  
  
    constant ffe_2_dim : natural := 2;  
    subtype ffe_2 is std_logic_vector(0 to ffe_2_dim - 1);  
  
--- element 1 in chosen basis:  
    constant ffe_2_one: ffe_2 := "10";  
--- element 0 in chosen basis:  
    constant ffe_2_zero: ffe_2 := "00";  
-----  
-----  
--- field_pkg params for vector
```

```

--- with identifier Fid = 3
--- of length 4
--- and direction "to"
--- with interpretation basis
--- [ Z(2)^0, Z(2^2), Z(2^4), Z(2^4)^6 ]
--- over sub-id = 1

constant ffe_3_dim : natural := 4;
subtype ffe_3 is std_logic_vector(0 to ffe_3_dim - 1);

--- element 1 in chosen basis:
constant ffe_3_one: ffe_3 := "1000";
--- element 0 in chosen basis:
constant ffe_3_zero: ffe_3 := "0000";
-----

-----
--- field_pkg params for vector
--- with identifier Fid = 4
--- of length 8
--- and direction "to"
--- with interpretation basis
--- [ Z(2)^0, Z(2^2), Z(2^4), Z(2^4)^6, Z(2^8)^7, Z(2^8)^92, Z(2^8)^24, Z(2^8)^109 ]
--- over sub-id = 1

constant ffe_4_dim : natural := 8;
subtype ffe_4 is std_logic_vector(0 to ffe_4_dim - 1);

--- element 1 in chosen basis:
constant ffe_4_one: ffe_4 := "10000000";
--- element 0 in chosen basis:
constant ffe_4_zero: ffe_4 := "00000000";
-----

-----
--- field_pkg params for elements of field
--- with identifier Fid = 5
--- for defining polynomial
--- x^2+x+Z(2)^0
--- and direction "to" basis
--- [ Z(2)^0, Z(2^2) ]
--- over sub-id = 1

constant ffe_5_dim : natural := 2;
subtype ffe_5 is std_logic_vector(0 to ffe_5_dim - 1);

--- element 1 in chosen basis:
constant ffe_5_one: ffe_5 := "10";
--- element 0 in chosen basis:
constant ffe_5_zero: ffe_5 := "00";
-----

```

```

-----
--- field_pkg params for vector
--- with identifier Fid = 6
--- of length 2
--- and direction "to"
--- with interpretation basis
--- [ Z(2)^0, Z(2^4) ]
--- over sub-id = 2

    constant ffe_6_dim : natural := 2;
    type ffe_6 is array(0 to ffe_6_dim - 1) of ffe_2;

--- element 0:
    constant ffe_6_zero: ffe_6 := ( others => ffe_2_zero );

    function "xor" ( a, b : ffe_6 ) return ffe_6;
-----

-----
--- field_pkg params for elements of field
--- with identifier Fid = 7
--- for defining polynomial
--- x^4+x+Z(2)^0
--- and direction "to" basis
--- [ Z(2)^0, Z(2^2), Z(2^4), Z(2^4)^6 ]
--- over sub-id = 1

    constant ffe_7_dim : natural := 4;
    subtype ffe_7 is std_logic_vector(0 to ffe_7_dim - 1);

--- element 1 in chosen basis:
    constant ffe_7_one: ffe_7 := "1000";
--- element 0 in chosen basis:
    constant ffe_7_zero: ffe_7 := "0000";
-----

-----
--- field_pkg params for elements of field
--- with identifier Fid = 8
--- for defining polynomial
--- x^2+x+Z(2^2)
--- and direction "to" basis
--- [ Z(2)^0, Z(2^4) ]
--- over sub-id = 5

    constant ffe_8_dim : natural := 2;
    type ffe_8 is array(0 to ffe_8_dim - 1) of ffe_5;

--- element 0:
    constant ffe_8_zero: ffe_8 := ( others => ffe_5_zero );

```

```

function "xor" ( a, b : ffe_8 ) return ffe_8;
-----

-----
--- field_pkg params for vector
--- with identifier Fid = 9
--- of length 2
--- and direction "to"
--- with interpretation basis
--- [ Z(2)^0, Z(2^8)^7 ]
--- over sub-id = 6

constant ffe_9_dim : natural := 2;
type ffe_9 is array(0 to ffe_9_dim - 1) of ffe_6;

--- element 0:
constant ffe_9_zero: ffe_9 := ( others => ffe_6_zero );

function "xor" ( a, b : ffe_9 ) return ffe_9;
-----

-----
--- field_pkg params for elements of field
--- with identifier Fid = 10
--- for defining polynomial
--- x^8+x^4+x^3+x^2+Z(2)^0
--- and direction "to" basis
--- [ Z(2)^0, Z(2^2), Z(2^4), Z(2^4)^6, Z(2^8)^7, Z(2^8)^92, Z(2^8)^24, Z(2^8)^109 ]
--- over sub-id = 1

constant ffe_10_dim : natural := 8;
subtype ffe_10 is std_logic_vector(0 to ffe_10_dim - 1);

--- element 1 in chosen basis:
constant ffe_10_one: ffe_10 := "10000000";
--- element 0 in chosen basis:
constant ffe_10_zero: ffe_10 := "00000000";
-----

-----
--- field_pkg params for elements of field
--- with identifier Fid = 11
--- for defining polynomial
--- x^2+x+Z(2^4)^7
--- and direction "to" basis
--- [ Z(2)^0, Z(2^8)^7 ]
--- over sub-id = 8

constant ffe_11_dim : natural := 2;
type ffe_11 is array(0 to ffe_11_dim - 1) of ffe_8;

```

```

--- element 0:
    constant ffe_11_zero: ffe_11 := ( others => ffe_8_zero );

    function "xor" ( a, b : ffe_11 ) return ffe_11;
-----

end package;

package body field_pkg is
function "xor" ( a, b : ffe_6 ) return ffe_6 is
    variable z : ffe_6;

    begin
        for i in ffe_6'range loop
            z(i) := a(i) xor b(i);
        end loop;
        return z;
end function;

function "xor" ( a, b : ffe_8 ) return ffe_8 is
    variable z : ffe_8;

    begin
        for i in ffe_8'range loop
            z(i) := a(i) xor b(i);
        end loop;
        return z;
end function;

function "xor" ( a, b : ffe_9 ) return ffe_9 is
    variable z : ffe_9;

    begin
        for i in ffe_9'range loop
            z(i) := a(i) xor b(i);
        end loop;
        return z;
end function;

function "xor" ( a, b : ffe_11 ) return ffe_11 is
    variable z : ffe_11;

    begin
        for i in ffe_11'range loop
            z(i) := a(i) xor b(i);
        end loop;
        return z;
end function;

end package body;

```


Example D.5.2 *Squaring for a polynomial basis* ↔

The following example is a continuation of Example C.3.3 and shows the implementation of the squaring module. The CIRCUIT package contains debug switches; one of them was used to show the steps of writing a datapath. Also note the use of partial binding between GAP variables and VHDL input ports (recall Key 14.2).

Example D.5.2

```
gap> K := GF(2);; x := X(K, "x");; m := 4;; f := x^4+x+1;;
gap> defaultPkg := DefaultSignalPkg(m);; PrintDownto(defaultPkg);

-----
SignalPkg with
4:      elements in      GF(2^4) with direction to basis
          [ Z(2)^0, Z(2^4), Z(2^4)^2, Z(2^4)^3 ]
3:      vectors over    GF(2)  of length 7 and direction to
2:      vectors over    GF(2)  of length 4 and direction to - interpretation basis
          [ Z(2)^0, Z(2^4), Z(2^4)^2, Z(2^4)^3 ]
1:      elements in      GF(2)
-----
gap> sq := FFA_sq_matrixU(f, "PB", "to");;

variables
[ "a_0", "a_1", "a_2", "a_3" ]
[ "b_0", "b_1", "b_2", "b_3" ]
[ "d_0", "d_1", "d_2", "d_3", "d_4", "d_5", "d_6" ]

gap> entitylist := ["square"];; ct := ["CICO", "simple"];;
gap> inputports := [{"i_a", 2}];; outputports := [{"o_z", 2}];;
gap> portlist := [defaultPkg, inputports, outputports];;
gap> bindlist := [{"a", "i_a"}];; exprlist := sq;;
gap> archlist := [bindlist, exprlist];;
gap> AF := AlgFunctionality(entitylist, portlist, archlist, ct);;
gap> MakeAlgFunctionalityDesignReady(AF, defaultPkg);;
gap> folder := "phdGapExamples/circuits/exampleSQ";; commentstr := "exampleSQ.g";;
gap> sminsn := ["generate", ["matrixU"], "CICO", "simple"];;
gap> design := AlgDesign(AF, defaultPkg, sminsn, [folder, commentstr] , 1);;
gap> nrfiles:= WriteDesignVHDL(design);;
no submodules exist, write circuits
OutputTextFile(/home/sonce/phdGapExamples/circuits/exampleSQ/field_pkg.vhd) done
available submodules: [ [ 2 ], [ [ ] ] ]##

### now writing coordinate=0 of output=0 with expr=      a_0+a_2 ###
#### STEP 1 - binding using rules: [ [ "a", "i_a" ] ]
```

```

*expr list after binding: [ [ "i_a(0)" ], [ "i_a(2)" ] ]

*expr list after EXP: [ [ "i_a(0)" ], [ "i_a(2)" ] ]

*expr list after multrees: [ [ "i_a(0)" ], [ "i_a(2)" ] ]

*expr list after MVs: [ [ "i_a(0)" ], [ "i_a(2)" ] ]

*expr list after constants: [ "i_a(0)", "i_a(2)" ]

#### STEP 6 - found sumtree
*expr list after sum: out_0_0_term_lvl1_edge0

### now writing coordinate=1 of output=0 with expr= a_2 ###
#### STEP 1 - binding using rules: [ [ "a", "i_a" ] ]

*expr list after binding: [ [ "i_a(2)" ] ]

... OMITTED FOR BREVITY ....

#### STEP 7 - writing outputs
*for output=1 with outsources: [ "out_0_0_term_lvl1_edge0", "i_a(2)",
  "out_0_2_term_lvl1_edge0", "i_a(3)" ]

OutputTextFile(/home/sonce/phdGapExamples/circuits/exampleSQ/square.vhd) done
OutputTextFile(/home/sonce/phdGapExamples/circuits/exampleSQ/square_tb.vhd) done

variables
[ "a_0", "a_1", "a_2", "a_3" ]
[ "b_0", "b_1", "b_2", "b_3" ]
[ "d_0", "d_1", "d_2", "d_3", "d_4", "d_5", "d_6" ]
OutputTextFile(/home/sonce/phdGapExamples/circuits/exampleSQ/square_stim.tv) done
OutputTextFile(/home/sonce/phdGapExamples/circuits/exampleSQ/square_spec.tv) done
OutputTextFile(/home/sonce/phdGapExamples/circuits/exampleSQ/square_tb.sim) done
OutputTextFile(/home/sonce/phdGapExamples/circuits/exampleSQ/square.uwp) done

```



Example D.5.3 A \mathbb{F}_{2^4} multiplier - full example \longleftrightarrow

This is a full example of a basic building block, \mathbb{F}_{2^4} multiplier, from SignalPkg setup to generating the VHDL files. The signal package always contains two entries for any given finite field: the field itself and the field as a vector space. This example is using the SignalPkg shown in Example D.2.3 in Section D.2. The SIGNAL corresponding to the finite field \mathbb{F}_{2^4} with the defining polynomial $f(x) = x^4 + x + 1$ and polynomial basis has `Fid=4`, and the SIGNAL corresponding to its vector space has the `Fid=2`. Example D.5.3 demonstrates several CIRCUIT package concepts:

- the effect of `AlgDesignWriteTop` (the last argument to the `AlgDesign` constructor)
- full binding between GAP variables and VHDL input ports (recall Key 14.2)
- the difference between *element* (`Fid=4`) and *vector* (`Fid=2`) ports (the vector ports are used only in last example, i.e., Example D.5.3(f))

The code in Example D.5.3(a) is using *element* ports, that is all input and output ports have `Fid=4`. The expression is simply set to `a_0*b_0`, using full binding `[["a_0", "i_a"], ["b_0", "i_b"]]`. The `topswitch` is set to 1, setting the `AlgDesignWriteTop` property to true. The datapath produced by the `ProcessDesign` method is printed with the `PrintCIRCListAll` method, which shows the actual multiplier submodule at `Fid=2`:

- functionality is “multiplier for `Fid=4`”, `FuncName: MforFid4`
- the entity name reflect the `Fid=2`: `EntityName: Fid2_mult`
- submodule ports are set to `Fid=2`
- the four expressions for the output vector of length 4 were obtained using the `FFA_mult_matrixU` method from the `FFCSA` package (see Example 7.4.1(a))

In Example D.5.3(b) the `AlgDesign` is “reinitialized” and the `WriteDesignVHDL` function is called. The design files are printed at the end of the example. The example also shows the generated testbench files and scripts (for the purpose of this text, the long paths in the `OutputTextFile` lines were shorten manually with ...).

■ **Implementation detail:** The reason why the constructor has to be called again is the top-down `ProcessSMAFloop` (see Section 15.1). The loop needs a starting point, the initial entries in the `SMList`, and when the loop terminates, the `SMList` is empty. Calling `ProcessDesign` but not writing the VHDL files in never used, hence no actual reinitialization method needed. ■

The VHDL Example D.5.3(c) is showing the VHDL module `multiply.vhd` with ports of type `ffe_4`, internal signals of type `ffe_2` and multiplier `Fid2_mult`. The `multiply` module is basically just a wrapper for the `Fid2_mult` module. The component instantiation uses the `Write_component_inst` function from the `GAPtoVHDL` package (Table 10.3), and the comment above the component instantiation clearly shows the `portmap` exactly as it was used for the function call.

Then, the example is rerun for the `topswitch 0`, see Example D.5.3(d). The only difference is the number of VHDL files generated: the top-level `AlgFunctionality` is not translated to VHDL, but all of its submodules are (in this case, there is a single submodule `Fid2_mult`). Multiplier `Fid2_mult` has all ports of type `ffe_2`, and all internal signals of type `ffe_1`; just as the submodule `Fid2_mult` generated in Example D.5.3(b).

VHDL Example D.5.3(e) shows a small fraction of the architecture that drives the output for coordinate 0 of the product, using the expression `--expr=a_0*b_0+a_1*b_3+ a_2*b_2+a_3*b_1`. The computation is done monomial by monomial, with lines like `--["i_a(0)", "i_b(0)"]` showing the result of the partial binding set for the multiplier automatically by the `ProcessSMAFloop`. This class 3 expression underwent steps 1 - binding, 3 - and gates, 5 - no additive constant was found, 6 - final sum and 7- drive the outputs. Note that the produced VHDL code is systematic, but not really human-friendly.

As was just mentioned, partial binding was used for the submodule `Fid2_mult` VHDL Example D.5.3(e). For the top-level module, the `bindlist` in VHDL Example D.5.3(a) uses full binding, which is also reflected in the commented line `--["i_a", "i_b"]` in VHDL Example D.5.3(c).

The Examples D.5.3(a)-D.5.3(d) used the following submodule instructions: "generate" and ["matrixU"], see `sminsn`. This tells `ProcessSMAFLoop` to generate submodules using the "matrixU" methods, in this case the `FFA_mult_matrixU` (see Section 7.4 Example 7.4.1 and GAP Example 7.4.1(a)). In Example D.5.3(f), the *vector* ports `Fid=2` are used. The length of the output port is 4 and the `FFA_mult_matrixU` method generates 4 expressions, hence the correctness check is again passed (see Key 13.4 Section 13.2 and its solution Key 14.3 , Section 14.1). The only difference between the VHDL modules generated in Example D.5.3(d) and this example are the two entity names, their architectures are exactly the same.

Example D.5.3(a)

```

gap> m:=4;; x := X(GF(2), "x");;
gap> defaultPkg := DefaultSignalPkg(m);
< SignalPkg with
1:  elements in      GF(2)
2:  vectors over    GF(2)    of length 4 and direction to - interpretation basis
                                Basis( GF(2^4), [ Z(2)^0, Z(2^4), Z(2^4)^2, Z(2^4)^3 ] )
3:  vectors over    GF(2)    of length 7 and direction to
4:  elements in      GF(2^4)>

gap> # parapemets that will change
gap> mul := a_0*b_0;;
gap> inputports := [ ["i_a", 4], ["i_b",4] ];;
gap> outputports := [ ["o_z", 4] ];;
gap> bindlist := [ ["a_0", "i_a"], ["b_0", "i_b"] ];;
gap> exprlist := [mul];;
gap> # parapemets that will NOT change
gap> entitylist := ["mult", "multiply", "main"];;
gap> portlist := [defaultPkg, inputports, outputports];;
gap> archlist := [bindlist, exprlist];;
gap> archtype := ["CICO", "simple"];;
gap> AF := AlgFunctionality(entitylist, portlist, archlist, archtype);;
gap> MakeAlgFunctionalityDesignReady(AF, defaultPkg);;
gap> folder := "phdGapExamples/circuits/exampleM1a";; commentstr := "ex6.2.MUL1a.g";;
gap> sminsn := ["generate", ["matrixU"], "CICO", "simple"];;
gap>
gap> # toposwitch = 1
gap> design := AlgDesign(AF, defaultPkg, sminsn, [folder, commentstr] , 1);;
gap> ProcessDesign(design);;

variables
[ "a_0", "a_1", "a_2", "a_3" ]
[ "b_0", "b_1", "b_2", "b_3" ]
[ "d_0", "d_1", "d_2", "d_3", "d_4", "d_5", "d_6" ]
gap> AlgDesignWriteTop(design); AlgDesignArchExprTop(design);
true
< AlgDesign for top implementing:
[ [ a_0*b_0 ] ]
>

```

```

gap> PrintCIRlistAll(design);

-----
AlgDesign with CIRlist

4:
3:
2:
class = 3      < Circuit of type [ "CICO", "simple", -1, -1, -1 ] :
- FuncName: MforFid4
- EntityName: Fid2_mult
- ArchName: main
- InputPorts: [ "i_a", 2 ], [ "i_b", 2 ],
- OutputPorts: [ "o_z", 2 ],
inputport binding: [ [ "a", "i_a" ], [ "b", "i_b" ] ]
implementing: [ [ a_0*b_0+a_1*b_3+a_2*b_2+a_3*b_1,
a_0*b_1+a_1*b_0+a_1*b_3+a_2*b_2+a_2*b_3+a_3*b_1+a_3*b_2,
a_0*b_2+a_1*b_1+a_2*b_0+a_2*b_3+a_3*b_2+a_3*b_3,
a_0*b_3+a_1*b_2+a_2*b_1+a_3*b_0+a_3*b_3 ] ]
>

constants: [ ]

1:
-----

```

```
gap> # reinitialize and write VHDL
gap> design := AlgDesign(AF, defaultPkg, sminsn, [folder, commentstr] , 1);
gap> WriteDesignVHDL(design);

variables
[ "a_0", "a_1", "a_2", "a_3" ]
[ "b_0", "b_1", "b_2", "b_3" ]
[ "d_0", "d_1", "d_2", "d_3", "d_4", "d_5", "d_6" ]
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/field_pkg.vhd) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/Fid2_mult.vhd) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/Fid2_mult_tb.vhd) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/Fid2_mult_stim.tv) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/Fid2_mult_spec.tv) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/Fid2_mult_tb.sim) done !
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/Fid2_mult.uwp) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/multiply.vhd) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/multiply_tb.vhd) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/multiply_stim.tv) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/multiply_spec.tv) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/multiply_tb.sim) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1a/multiply.uwp) done!
gap> Print(design!.filelist);
[ "field_pkg.vhd", "Fid2_mult.vhd", "multiply.vhd" ]
```

```

entity multiply is
    port(
        i_a:    in ffe_4;    -- input
        i_b:    in ffe_4;    -- input
        o_z:    out ffe_4    -- output
    );
end entity;

architecture main of multiply is
-- SKIP: multiplication tree nodes
-- multiplication tree edges

    signal out_0_0_mon0_lv12_edge0, out_0_0_mon0_lv12_edge1: ffe_2;
    signal out_0_0_mon0_lv11_edge0: ffe_2;

begin

--output=0, idx=0, --expr=a_0*b_0
--[ "i_a", "i_b" ]

    out_0_0_mon0_lv12_edge0 <= i_a;
    out_0_0_mon0_lv12_edge1 <= i_b;

    -- mult_0_0_0_0 with portmap = [ "out_0_0_mon0_lv12_edge0", "out_0_0_mon0_lv12_edge1", "out_0_0_mon0_lv11_edge0" ]

    mult_0_0_0_0 : entity work.Fid2_mult(main) port map ( out_0_0_mon0_lv12_edge0, out_0_0_mon0_lv12_edge1,
                                                         out_0_0_mon0_lv11_edge0 );

    -- output

    o_z    <= out_0_0_mon0_lv11_edge0;

end;

```

```
gap> # reinitialize with top switch = 0 and write VHDL
gap> folder := "phdGapExamples/circuits/exampleM1d";;
gap> design := AlgDesign(AF, defaultPkg, sminsn, [folder, commentstr] , 0);;
gap> WriteDesignVHDL(design);;

variables
[ "a_0", "a_1", "a_2", "a_3" ]
[ "b_0", "b_1", "b_2", "b_3" ]
[ "d_0", "d_1", "d_2", "d_3", "d_4", "d_5", "d_6" ]
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1d/field_pkg.vhd) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1d/Fid2_mult.vhd) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1d/Fid2_mult_tb.vhd) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1d/Fid2_mult_stim.tv) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1d/Fid2_mult_spec.tv) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1d/Fid2_mult_tb.sim) done !
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1d/Fid2_mult.uwp) done!
gap> Print(design!.filelist);
[ "field_pkg.vhd", "Fid2_mult.vhd" ]
```



```

--output=0, idx=0, --expr=a_0*b_0+a_1*b_3+a_2*b_2+a_3*b_1
--[ "i_a(0)", "i_b(0)" ]
    out_0_0_mon0_lv12_edge0 <= i_a(0);
    out_0_0_mon0_lv12_edge1 <= i_b(0);
    out_0_0_mon0_lv11_edge0 <= out_0_0_mon0_lv12_edge0 and out_0_0_mon0_lv12_edge1;

--[ "i_a(1)", "i_b(3)" ]
    out_0_0_mon1_lv12_edge0 <= i_a(1);
    out_0_0_mon1_lv12_edge1 <= i_b(3);
    out_0_0_mon1_lv11_edge0 <= out_0_0_mon1_lv12_edge0 and out_0_0_mon1_lv12_edge1;

--[ "i_a(2)", "i_b(2)" ]
    out_0_0_mon2_lv12_edge0 <= i_a(2);
    out_0_0_mon2_lv12_edge1 <= i_b(2);
    out_0_0_mon2_lv11_edge0 <= out_0_0_mon2_lv12_edge0 and out_0_0_mon2_lv12_edge1;

--[ "i_a(3)", "i_b(1)" ]
    out_0_0_mon3_lv12_edge0 <= i_a(3);
    out_0_0_mon3_lv12_edge1 <= i_b(1);
    out_0_0_mon3_lv11_edge0 <= out_0_0_mon3_lv12_edge0 and out_0_0_mon3_lv12_edge1;
    out_0_0_mon0_lv11_edge0, "out_0_0_mon1_lv11_edge0", "out_0_0_mon2_lv11_edge0", "out_0_0_mon3_lv11_edge0" ]
    out_0_0_term_lv13_edge0 <= out_0_0_mon0_lv11_edge0;
    out_0_0_term_lv13_edge1 <= out_0_0_mon1_lv11_edge0;
    out_0_0_term_lv13_edge2 <= out_0_0_mon2_lv11_edge0;
    out_0_0_term_lv13_edge3 <= out_0_0_mon3_lv11_edge0;
    out_0_0_term_lv12_edge0 <= out_0_0_term_lv13_edge0 xor out_0_0_term_lv13_edge1;
    out_0_0_term_lv12_edge1 <= out_0_0_term_lv13_edge2 xor out_0_0_term_lv13_edge3;
    out_0_0_term_lv11_edge0 <= out_0_0_term_lv12_edge0 xor out_0_0_term_lv12_edge1;

```

```

gap> # new parapepets
gap> ChooseFieldElms(GF(2^4));; B := Basis(GF(2^4));;

variables
[ "a_0", "a_1", "a_2", "a_3" ]
[ "b_0", "b_1", "b_2", "b_3" ]
[ "d_0", "d_1", "d_2", "d_3", "d_4", "d_5", "d_6" ]
gap> mul := FFA_mult_matrixU(B, avec, bvec);;
gap> inputports := [ ["i_a", 2], ["i_b",2] ];;
gap> outputports := [ ["o_z",2] ];;
gap> bindlist := [ ["a", "i_a"], ["b", "i_b"] ];;
gap> exprlist := mul;;
gap> # parapepets that did NOT change
gap> entitylist := ["mult", "multiply", "main"];;
gap> portlist := [defaultPkg, inputports, outputports];;
gap> archlist := [bindlist, exprlist];;
gap> archtype := ["CICO", "simple"];;
gap> AF := AlgFunctionality(entitylist, portlist, archlist, archtype);;
gap> MakeAlgFunctionalityDesignReady(AF, defaultPkg);;
gap> folder := "phdGapExamples/circuits/exampleM1f";;
gap> sminsn := ["generate", ["matrixU"], "CICO", "simple"];;
gap> design := AlgDesign(AF, defaultPkg, sminsn, [folder, commentstr] , 1);;
gap> AlgDesignWriteTop(design); AlgDesignArchExprTop(design);
true
< AlgDesign for top implementing:
[ [ a_0*b_0+a_1*b_3+a_2*b_2+a_3*b_1,
a_0*b_1+a_1*b_0+a_1*b_3+a_2*b_2+a_2*b_3+a_3*b_1+a_3*b_2,
a_0*b_2+a_1*b_1+a_2*b_0+a_2*b_3+a_3*b_2+a_3*b_3,
a_0*b_3+a_1*b_2+a_2*b_1+a_3*b_0+a_3*b_3 ] ]
>
gap> WriteDesignVHDL(design);;
no submodules exist, write circuits!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1f/field_pkg.vhd) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1f/multiply.vhd) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1f/multiply_tb.vhd) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1f/multiply_stim.tv) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1f/multiply_spec.tv) done!
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1f/multiply_tb.sim) done !
OutputTextFile(/home/ ... /phdGapExamples/circuits/exampleM1f/multiply.uwp) done!
gap> Print(design!.filelist);
[ "field_pkg.vhd", "multiply.vhd" ]

```

