Verification of Pipelined Ciphers

by

Chiu Hong Lam

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

Waterloo, Ontario, Canada, 2008

Abstract

The purpose of this thesis is to explore the formal verification technique of completion functions and equivalence checking by verifying two pipelined cryptographic circuits, KASUMI and WG ciphers.

Most of current methods of communications either involve a personal computer or a mobile phone. To ensure that the information is exchanged in a secure manner, encryption circuits are used to transform the information into an unintelligible form. To be highly secure, this type of circuits is generally designed such that it is hard to analyze. Due to this fact, it becomes hard to locate a design error in the verification of cryptographic circuits. Therefore, cryptographic circuits pose significant challenges in the area of formal verification.

Formal verification use mathematics to formulate correctness criteria of designs, to develop mathematical models of designs, and to verify designs against their correctness criteria.

The results of this work can extend the existing collection of verification methods as well as benefiting the area of cryptography. In this thesis, we implemented the KASUMI cipher in VHDL, and we applied the optimization technique of pipelining to create three additional implementations of KASUMI. We verified the three pipelined implementations of KASUMI with completion functions and equivalence checking. During the verification of KASUMI, we developed a methodology to handle the completion functions efficiently based on VHDL generic parameters. We implemented the WG cipher in VHDL, and we applied the optimization techniques of pipelining and hardware re-use to create an optimized implementation of WG. We verified the optimized implementation of WG with completion functions and equivalence checking. During the verification of WG, we developed the methodology of “skipping” that can decrease the number of verification obligations required to verify the correctness of a circuit. During the verification of WG, we developed a way of applying the completion functions approach such that it can deal with a circuit that has been optimized with hardware re-use.

Acknowledgements

I would like to thank my MASc supervisor, Dr. Mark Aagaard, for his guidance and support from the beginning to the end of this research. I would like to thank Dr. Guang Gong and Dr. Anwar Hasan for their suggestions that have improved the quality of this thesis. I would like to thank my fellow researchers at the Waterloo Formal Methods Group, Vlad Ciubotariu and Hazem Shehata, for their assistance in the submission of this thesis. Finally, I would like to thank Ontario Graduate Scholarship (OGS) program for their support in this research.

To wingL

Contents

List of Tables **viii**

List of Figures **xi**

1 Introduction **1**

1.1 Overview of Background 2

1.2 Thesis Outline 3

2 Background Materials **4**

2.1 Burch-Dill Flushing 4

2.2 Completion Functions 7

2.3 The Basics of Cryptography 11

2.4 The KASUMI Cipher 12

2.5 Algebraic Terminology for Finite Fields and Normal Basis 15

2.6 The Welch-Gong Cipher 16

3 KASUMI: Design, Optimizations and Verification **21**

3.1 The First Design 21

3.2 Formulating The Specification 22

3.3 Pipelined Implementations 26

3.4 Verification Methodology 28

3.4.1 General Guidelines 28

3.4.2 Background of VHDL Features 33

3.4.3 Single Entity Environment 34

3.4.4 Multiple Entities Environment 35

3.5 Verification of KASUMI 38

4 WG: Design and Optimizations 43

4.1 The First Design 43

4.1.1 WG Core and Trace Function 44

4.1.2 Linear Feedback Shift Register 46

4.1.3 Finite State Machine 47

4.2 Formulating The Specification 48

4.2.1 Finite State Machine 50

4.2.2 Linear Feedback Shift Register 50

4.2.3 Trace Function and WG Core 52

4.3 The Optimized Implementation 53

4.3.1 Pipelining 53

4.3.2 Hardware Re-Use 55

4.3.3 Control Circuitry Modifications 64

4.4 Related Work 66

5 WG: Verification 70

5.1 The WG Core 70

5.1.1 First Verification: The Normal Basis Multiplier 71

5.1.2 Second Verification: The $(\cdot)^{2^{10}-1}$ Block 72

5.1.3 Third Verification: The WG Core 76

5.2 The Linear Feedback Shift Register 82

5.3 The Finite State Machine 84

5.3.1 Background of Model Checking 84

5.3.2 The Verification 85

5.4 Related Work 88

6 Conclusions and Future Work **90**

References **91**

List of Tables

4.1 Signal Selected Based on *load* and *init* 46

5.1 Semantics of Temporal Modal Operators 85

5.2 First Set of Properties Verified 86

5.3 Semantics of Additional Operators 86

5.4 Second Set of Properties Verified 87

5.5 Third Set of Properties Verified 87

List of Figures

2.1 Specification 5

2.2 Pipelined Implementation 5

2.3 Difficulty in The Verification of Pipelined Circuits 6

2.4 Burch-Dill Flushing 8

2.5 Flushing Example 8

2.6 Completion Functions 9

2.7 Completion Functions Example 9

2.8 Block Ciphers 12

2.9 Stream Ciphers 12

2.10 KASUMI Circuit 14

2.11 FO Circuit 14

2.12 WG Family Circuit Structure 17

2.13 LFSR Feedback Polynomial 18

2.14 Implementation of WG Transformation 19

2.15 Key Initialization Phase 20

3.1 FI Test Vectors Results 23

3.2 Fragments of Test Vectors from KASUMI Specification 24

3.3 FO Test Vectors Results 24

3.4 FL Test Vectors Results 24

3.5 KASUMI Test Vectors Results 25

3.6 Registers Locations in KASUMI 27

3.7 Registers Locations in FO 27
3.8 Area and Performance Results of Various KASUMI Designs 29
3.9 2-Stage Pipeline Verification Structure 31
3.10 First Verification Obligation 31
3.11 Second Verification Obligation 32
3.12 Third Verification Obligation 32
3.13 If-Generate Statement 33
3.14 Generic Parameters 34
3.15 Entity A: VHDL Code 36
3.16 Contrived Entity A With Completion Functions 36
3.17 Contrived Entity D 37
3.18 Stage 8 Obligation 39
3.19 Stage 7 Obligation 39
3.20 Stage 1 Obligation 41
3.21 Final Obligation 41

4.1 WG Implementation Block Diagram 44
4.2 Implementation of $(\cdot)^{2^{10}-1}$ 45
4.3 Linear Feedback Shift Register Implementation 46
4.4 Finite State Machine Implementation 47
4.5 Fragments of Test Vectors from WG Paper 49
4.6 Finite State Machine Simulation Results 51
4.7 Linear Feedback Shift Register Simulation Results 51
4.8 Trace Function Simulation Results 52
4.9 WG Test Vectors Results 53
4.10 4-Stage Implementation of $(\cdot)^{2^{10}-1}$ 54
4.11 5-Stage Implementation of WG Core 54
4.12 Initial Pipeline 56
4.13 Pipeline with Re-Use 56

4.14 Pipeline with Re-Use and Superpipelining 56

4.15 Timing Diagram of Pipeline with Re-Use and Superpipelining . . . 57

4.16 9-stage Implementation of $(\cdot)^{2^{10}-1}$ with Re-Use 58

4.17 Timing Diagram of 9-stage Implementation of $(\cdot)^{2^{10}-1}$ with Re-Use . 59

4.18 11-stage Implementation of WG Core with Re-Use 61

4.19 Timing Diagram of 11-stage Implementation of WG Core with Re-Use 62

4.20 Optimized Implementation Block Diagram of WG 64

4.21 Modified Finite State Machine Implementation 65

4.22 Area and Performance Results of Various Stream Ciphers 68

5.1 Multiplier: 1^{st} Obligation 71

5.2 Multiplier: 2^{nd} Obligation 71

5.3 Multiplier: Final Obligation 72

5.4 Stage 8 Obligation with Skipping and *d_valid*='0' 73

5.5 Stage 6 Obligation with Skipping and *d_valid*='0' 75

5.6 Stage 5 Obligation 75

5.7 Stage 3 Obligation with Skipping and *d_valid*='1' 76

5.8 Stage 1 Obligation with Skipping and *d_valid*='1' 77

5.9 Final Obligation of $(\cdot)^{2^{10}-1}$ Block 77

5.10 Stage 11 Obligation of WG Core 78

5.11 Stage 10 Obligation of WG Core 79

5.12 Stage 9 Obligation of WG Core 80

5.13 Stage 7 Obligation of WG Core 81

5.14 Stage 1 Obligation of WG Core 82

5.15 Final Obligation of WG Core 83

5.16 Verification of LFSR with *ce*='1' 83

Chapter 1

Introduction

Many aspects of our life depend vitally on digital systems. Yet, empirical studies have shown that more than 50 percent of all application specific designs do not work properly after initial design and fabrication [15]. These dysfunctional designs can cost millions of dollars and human lives. In 1994, a bug in Intel's Pentium floating point unit costs 475 millions US dollars to replace the faulty processors [12]. Also, a data type conversion bug caused the launch failure of Ariane 5 rocket in 1996. In principle, verification methods can detect and fix these bugs at the early stages of the design process.

Verification is the process of proving or disproving the correctness of a design and it traditionally relies on exhaustive simulation. However, the increasing design size and complexity make exhaustive simulation an unattractive verification method due to the unreasonable required amount of time and computing resources. Formal verification can improve the verification procedure with the application of mathematics to formulate correctness criteria of designs, to develop mathematical models of designs, and to verify designs against their correctness criteria. In exchange of verifying all possible behaviours of a circuit, formal verification suffers from the problem of limited capacity where it can only verify detailed models of small circuits or very abstract models of complex circuits. Therefore, new verification methodologies need to be developed in order to increase the probability of achieving a bug-free design.

Most of current methods of communications either involve a personal computer or a mobile phone. To ensure that the information is exchanged in a secure manner, encryption circuits are used to transform the information into an unintelligible form. To be highly secure, this type of circuits is generally designed such that it is hard to analyze. Due to this fact, it becomes hard to locate a design error in the verification of cryptographic circuits. Therefore, cryptographic circuits pose significant challenges in the area of formal verification.

The aim of this thesis is to develop new verification methodologies by verifying cryptographic circuits. The results of this work can extend the existing collection of verification methods as well as benefiting the area of cryptography. The remaining of this chapter is organized as follows. In Section 1.1, we give a brief overview of the verification technique applied in our work and the two cryptographic circuits used as our verification case studies. Section 1.2 provides an outline of the major sections in this thesis.

1.1 Overview of Background

In formal verification, the technique of combinational equivalence checking is one of the most practical developments due to its high capacity and its high degree of automation. In the design of digital circuits, the initial hardware description of a specification is referred as the golden reference model. Under the assumption that the reference model has been verified by simulation or other verification methods, equivalence checking is used to prove the equivalence between the reference model and an implementation model which is derived from optimizing the reference model. The process of equivalence verification compares the combinational logic of the two models by identifying related signals in the two models, selecting a subset of the related points to be the compare points, and verifying each compare point in the implementation model against the corresponding compare point in the reference model.

In the design of digital circuits, the optimization technique of pipelining is widely used to increase the circuit throughput by overlapping the execution of instructions. Pipelining is one of the reasons for the increase in circuit complexity. For example, the specification of a processor defines how the programmer visible parts of the processor are updated after one instruction is executed, one cycle per instruction. However, pipelined implementations of processors can have partially executed instructions in the pipeline that cause the programmer visible parts to be updated at different pipeline stages or cycles. Therefore, a proper relationship between the specification and the pipelined implementation cannot be established due to partially executed instructions in the pipeline.

The verification technique of completion functions can establish a proper relationship between the specification and the pipelined implementation. Completion functions are abstract functions used to decompose the verification of a pipeline into smaller stage-by-stage verification obligations. For each stage of a pipeline, there is a completion function that describes the effects on the programmer visible parts of completing the partially executed instruction in that stage. Thus, applying all completion functions of a pipeline has the same effect as completing all instructions in the pipeline. This verification technique can localize an implementation error and never lead to a false positive verification.

Cryptography is the study of the methods in securing information. To keep the information secret, encryption is the process used to convert comprehensible information (i.e. plaintext) into incomprehensible information (i.e. ciphertext). Decryption is the process used to recover the plaintext from the ciphertext. Together, encryption and decryption constitute a pair of algorithms which is referred as a cipher. KASUMI, also known as A5/3, is a block cipher in which encryption and decryption operations are identical with a reversal of the key schedule. It is used in the confidentiality and integrity algorithms for the third generation mobile phone system. It operates on blocks of 64 bits and outputs in block of 64 bits. WG is a synchronous stream cipher that has been designed to produce a keystream with guaranteed randomness properties such as balance, long period, large and exact linear complexity, 3-level additive autocorrelation and ideal 2-level multiplicative autocorrelation. Also, it is resistant to Time/Memory/Data tradeoff attacks, algebraic attacks and correlation attacks.

1.2 Thesis Outline

The remaining of this thesis is organized as follows.

- Chapter 2 provides the necessary background for the reader to understand the work of this thesis.
- Chapter 3 describes the implementation, optimization, and verification of the KASUMI cipher.
- Chapter 4 contains the implementation and optimization of the WG cipher.
- Chapter 5 presents the verification of the WG cipher.
- Chapter 6 includes the conclusions of this thesis and future work.

Chapter 2

Background Materials

The work of this thesis involves exploring a formal verification strategy in the verification of two cryptographic circuits. The purpose of this chapter is to provide the necessary background in order to understand the contribution of this thesis, and this chapter is organized as follows. Section 2.1 and 2.2 are used to describe two verification strategies: Burch-Dill flushing and completion functions. Section 2.3 and 2.5 provides an introduction to cryptography as well as a list of algebraic terminologies. Section 2.4 and 2.6 give a description of our two verification case studies: the KASUMI cipher and the Welch-Gong cipher.

2.1 Burch-Dill Flushing

The research of this thesis involves the use of the verification strategy known as completion functions. The verification strategy of completion functions was derived from an earlier verification strategy called Burch-Dill flushing. To understand the verification strategy of completion functions, we explain why Burch-Dill flushing was invented and how it is used to verify a pipelined design in this section.

Burch-Dill flushing [6] is an important concept which was first used in the verification of pipelined processors. The specification of a processor defines how the programmer visible parts of the processor are updated after one instruction is executed, one cycle per instruction. However, pipelined implementations of processors can have partially executed instructions in the pipeline that cause the programmer visible parts to be updated at different pipeline stages or cycles. A contrived example based on Figure 2.1 to 2.3 would demonstrate this difficulty in the verification of pipelined implementations against a non-pipelined specification.

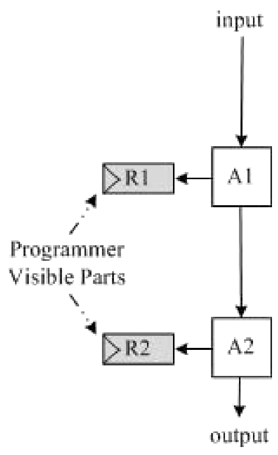


Figure 2.1: Specification

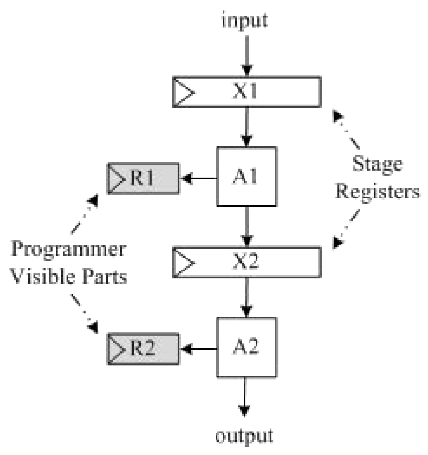


Figure 2.2: Pipelined Implementation

Figure 2.1 illustrates a non-pipelined specification which shows how the programmer visible parts (registers *R1* and *R2*) are updated once per cycle depending on the instruction at the input of the combinational block *A1* and *A2*. Figure 2.2 depicts a 2-stage implementation of the circuit shown in Figure 2.1 where the stage registers *X1* and *X2* are inserted to form the pipeline. For the 2-stage implementation of Figure 2.2, the circuit can execute two parcels of data simultaneously where the first pipeline stage writes to the register *R1* and the second pipeline stage is used to update the register *R2*. Figure 2.3 illustrates how the specification and the pipelined implementation update their programmer visible parts *R1* and *R2*. Figure 2.3 is divided into the specification side (top) and the implementation side (bottom). In addition, there is a legend for the registers (*X1*, *X2*, *R1*, *R2*) at the right end of Figure 2.3. Both the specification and the pipelined implementation begin with the registers *R1* and *R2* holding null values as shown in Figure 2.3. Then, the instruction sequence of $\langle A, B, C \rangle$ is fed, one instruction per cycle, to both the specification and the pipelined implementation. On the specification side, both programmer visible parts *R1* and *R2* are updated at the same cycle for a given instruction. On the implementation side, the registers *R1* and *R2* always have different values due to partially executed instructions in the pipeline. Note that the stage registers (*X1*, *X2*) only exist on the implementation side due to pipelining, and that their behaviour are not defined as part of the specification. Thus, the correctness of the pipelined implementation is independent of the stage registers *X1* and *X2*. Since the programmer visible parts (*R1*, *R2*) of both the specification and the pipelined implementation never hold the same value, it is impossible to draw a proper relationship between the specification and the pipelined implementation without relying on additional verification methods.

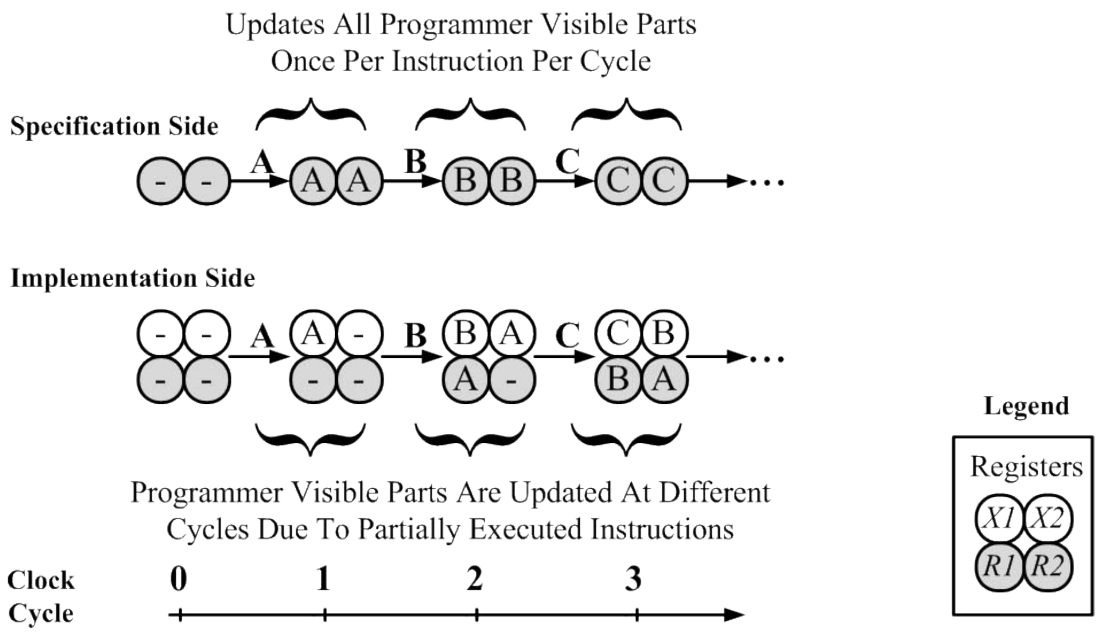

Updates All Programmer Visible Parts Once Per Instruction Per Cycle
Specification Side
A
B
C
Implementation Side
Programmer Visible Parts Are Updated At Different Cycles Due To Partially Executed Instructions
Clock Cycle
0
1
2
3
Legend
Registers
X1
X2
R1
R2

Figure 2.3: Difficulty in The Verification of Pipelined Circuits

Figure 2.4 depicts how Burch-Dill flushing can be used to establish a proper relationship between a pipelined implementation and its specification. In verifying processors, both the specification and its implementation are modeled as state machines in Figure 2.4. The transition function $N_i(\cdot\,, I)$ brings the old implementation state to its new implementation state given a set of inputs i. Similarly, $N_s(\cdot\,, I)$ returns the new specification state given its old specification state and a set of inputs i. Burch and Dill used one of the processors properties to establish a proper relationship between the specification and its pipelined implementation. All processors have an input setting I_{bubble} that causes instructions already in the pipeline to continue execution while no new instructions are initiated, and this is known sending "bubbles" down the pipeline. Thus, all instructions already in the pipeline can be completed by sending a certain number of bubbles down the pipeline, and this is called Burch-Dill flushing.

In Figure 2.4, both old and new implementation states are flushed to emulate the effects of completing all instructions already in the pipeline. Thus, all partially executed instructions are completed and updated the programmer visible parts accordingly. Then, the *proj* function extracts the programmer visible parts from the flushed states. These extracted programmer visible parts can be compared with the old and the new specification state. The implementation satisfies the specification if and only if the diagram of Figure 2.4 commutes.

In order to demonstrate the use of Burch-Dill flushing on a simple example, the last two cycles of Figure 2.3 are used to partially form Figure 2.5. Note that Figure 2.5 and Figure 2.4 have similar structure in which the square boxes denotes exactly the same states (i.e. *Old Spec*, *New Spec*, *Flushed Old Impl*, *Flushed New Impl*, *Old Impl*, *New Impl*). In Figure 2.5, the specification states (*Old*, *New*) and the implementation states (*Old*, *New*) correspond to the ones shown in the last two cycles of Figure 2.3. However, a proper relationship between the specification and the pipelined implementation cannot be established due to partially executed instructions in the pipeline. Burch-Dill flushing is used to build this relationship from the implementation side (bottom) to the specification side (top) as shown in Figure 2.5. Since the example is a 2-stage pipeline, two bubbles (denoted as $\emptyset$ in the figure) are sufficient to flush the pipeline and to emulate the effects of completing all instructions already in the pipeline.

In Figure 2.5, flushing brings the implementation states (*Old*, *New*) to their corresponding flushed states (*Old*, *New*). Then, *proj* function is applied to these flushed states in order to extract the programmer visible parts (*R1*, *R2*). In this simple example, both the implementation and the specification have a matching set of programmer visible parts (*R1*, *R2*) and it implies that the implementation is correct with respect to the specification. In the case that the set of programmer visible parts does not match, the implementation is said to be incorrect with respect to the specification.

With the idea of flushing, Burch and Dill verified a pipelined ALU and a subset of the DLX processor architecture. This section has explained the challenge in the verification of pipelined circuits, and it has demonstrated how Burch-Dill flushing is used to establish a proper relationship between a pipelined implementation and its non-pipelined specification. In the next section, the verification strategy of completion functions (based on Burch-Dill flushing) is introduced.

2.2 Completion Functions

In this section, we describe why completion functions were invented, what they are and how they are used for the verification of pipelined circuits.

Burch-Dill flushing of Section 2.1 works well in the verification of small scale pipelined circuits, e.g. in-order pipelines with small number of stages. However, Burch-Dill flushing runs into the state-space explosion problem for large scale designs that support out-of-order execution and have a large number of pipeline stages.

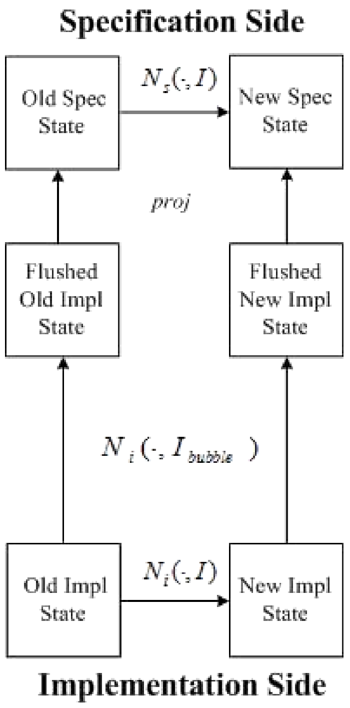


Figure 2.4: Burch-Dill Flushing

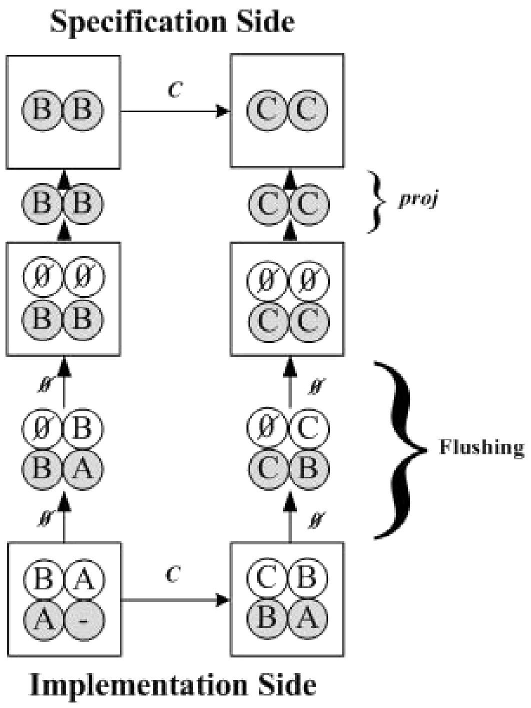


Figure 2.5: Flushing Example

Completion functions are abstract functions used to decompose Burch-Dill flushing's monolithic verification in Figure 2.4 into multiple smaller verification obligations. Thus, the completion functions approach is a solution to Burch-Dill flushing's state-space explosion problem. For each stage of a pipeline, there is a completion function that describes the effects on the programmer visible parts of completing the partially executed instruction in that stage. Thus, applying all completion functions of a pipeline has the same effect as flushing a pipeline. Hosabettu *et al.* used completion functions over Burch-Dill flushing to establish a proper relationship between a pipelined implementation states and its non-pipelined specification states. This verification technique can localize an implementation error and never lead to a false positive verification.

A contrived example in Figure 2.6 shows how completion functions are used for the verification of a 2-stage pipeline. The pipelined implementation satisfies its specification if and only if the diagram in Figure 2.6 commutes. A pipeline of n-stage requires $n+1$ verification obligations to check its correctness. In this example, the first obligation VO_1 verifies the second or last pipeline stage by comparing the state a produced by applying last stage's completion function C_2 to the state a' produced by taking an implementation step $N_i(\cdot, I)$. The second obligation VO_2 is used to verify the first pipeline stage, and it compares the state b produced by

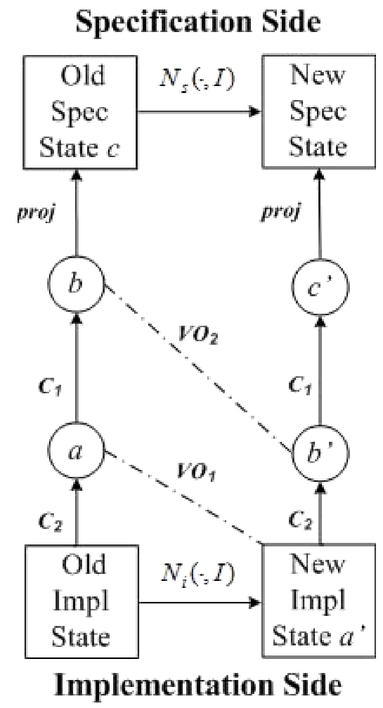


Figure 2.6: Completion Functions

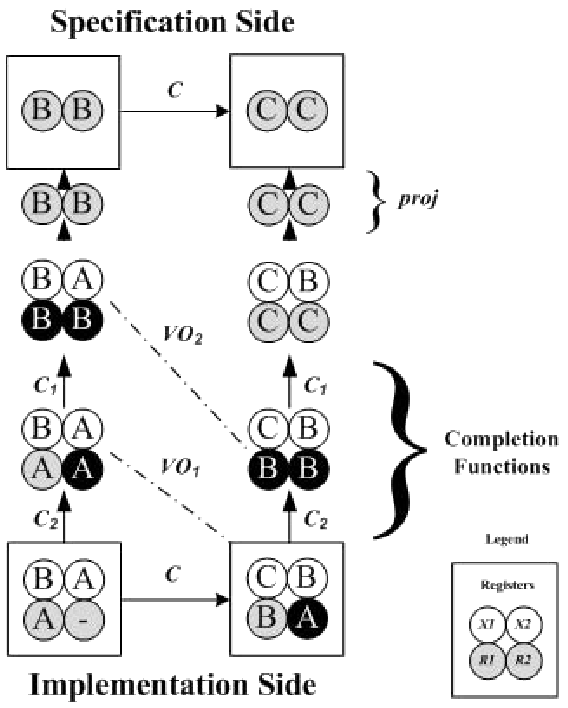


Figure 2.7: Completion Functions Example

the completion function C_1 to the state b' produced by taking an implementation step and then applying C_2. The last or third verification obligation compares the specification step $N_s(\cdot, I)$ to the completion function of the first stage C_1. Therefore, mistakes in building these completion functions can be detected with this final verification obligation. Also, implementation bugs can be localized due to the stage-by-stage verification given that the completion functions are correct.

In order to show the similarities and the differences between the completion functions approach and Burch-Dill flushing, the same example of Figure 2.5 is verified with completion functions. As depicted in Figure 2.7, the four square boxes represent the four states (*Old Impl State*, *New Impl State*, *Old Spec State*, *New Spec State*). In Figure 2.7, note that the completion functions have replaced Burch-Dill flushing in establishing a proper relationship between the implementation states and the specification states.

The completion functions approach and Burch-Dill flushing uses the same correctness criteria as follows. Given that the implementation and the specification start in any matching pair of states (i.e. *Old Impl State* matches *Old Spec State*), the implementation is said to be correct when taking an implementation step $N_i(\cdot\,, I)$ on the implementation side leads to a matching pair of states (i.e. *New Impl State* matches *New Spec State*) as taking a specification step $N_s(\cdot\,, I)$ on the specification side. In other words, the implementation is correct when the diagrams (Figures 2.5 and 2.7) commute.

The completion functions approach differs from Burch-Dill flushing in the way it establishes the relationship from the implementation side to the specification side as shown in Figure 2.7. Since Burch-Dill flushing runs into the state-space explosion problems for pipelines with large number of stages, the completion functions decompose the 2-stage pipeline verification of Figure 2.5 into 3 smaller verification obligations as shown in Figure 2.7.

In Figure 2.7, the first verification obligation VO_1 is used to verify the correctness of the second stage of the pipeline in Figure 2.2. It compares the value of the register *R2* produced by the completion function C_2 to the one produced by taking an implementation step as shown in Figure 2.7. The completion function C_2 completes the instruction A in the stage register *X2* and updates the register *R2* accordingly. The implementation step updates the register *R2* according to the circuit shown in Figure 2.2. Thus, the first verification obligation VO_1 verifies if the second pipeline stage is equivalent to the completion function C_2.

Similarly, the next verification obligation VO_2 of Figure 2.7 is used to verify the correctness of the first pipeline stage in Figure 2.2. It compares the values of the register set (*R1*, *R2*) generated by the completion function C_1 to the values created by taking an implementation step followed by the completion function C_2 in Figure 2.7. Since VO_2 verifies the correctness of the first pipeline stage, the completion functions C_1 is applied to the stage register *X1* to emulate the effects of completing the instruction B as shown in Figure 2.7. The implementation step updates the register *R1* according to the circuit shown in Figure 2.2 and it is followed by the completion function C_2 to update the register *R2*. Thus, the second verification obligation $V0_2$ verifies if the first pipeline stage is correct with respect to the behaviour given by the completion function C_1.

Up to this point of the verification, it has been shown that the pipeline stages are functionally correct with respect to their completion functions (C_1 or C_2). However, these completion functions can have design bugs buried in them. Finally, the last or third verification obligation compares the specification step to the completion function of the first pipeline stage C_1. Therefore, any bugs in the completion functions are caught with this final verification obligation.

Completion functions have been investigated under several verification techniques. Hosabettu et al. invented completion functions and verified complex out-of-order processors in an interactive theorem prover [10]. Berezin et al. applied completion functions to symbolic model checking in verifying an abstract model of Tomasulos algorithm [5]. Velev used completion functions and automated first-order decision procedures to verify abstract models of out-of-order processors [18]. Aagaard et al. combined completion functions and equivalence checking for the verification of a 32-bit OpenRISC processor and a Sobel edge-detector circuit at the register- transfer-level [3].

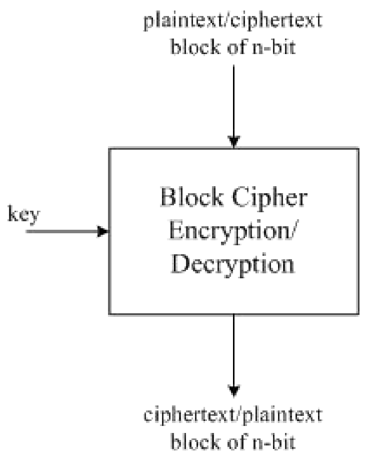
This section has covered the need for completion functions, how they are applied in the verification of a 2-stage pipeline, their similarities and their differences to Burch-Dill flushing, and how they have been used in other work.

2.3 The Basics of Cryptography

To better understand the two cryptographic circuits, this section provides an introduction to cryptography. This section describes the following: cryptography, encryption, decryption, symmetric key algorithms, asymmetric key algorithms, key scheduling, block ciphers, and stream ciphers. Section 2.4 describes the KASUMI cryptographic algorithm, and Section 2.6 provides the background of the Welch-Gong cipher.

Cryptography is the study of the methods in securing information. To keep the information secret, encryption is the process used to convert comprehensible information (i.e. plaintext) into incomprehensible information (i.e. ciphertext). Decryption is the process used to recover the plaintext from the ciphertext. Together, encryption and decryption constitute a pair of algorithms which is referred as a cipher. The operations of a cipher are not just determined by the algorithm, but they also depend on a key. When a different key is used for the encryption/decryption of the same plaintext/ciphertext, the cipher produces a corresponding ciphertext/plaintext for each key. Thus, keys make it more difficult in establishing the relationship between the plaintext and the ciphertext.

Depending on the type of key used, a cipher can be categorized as a symmetric key or an asymmetric key algorithm. In a symmetric key algorithm, the same key is used for both encryption and decryption. The key used to encrypt a plaintext differs from the key used to decrypt it in an asymmetric algorithm. Certain types of ciphers use an algorithm, known as the key schedule, to compute smaller keys from the input key. Then, the smaller keys are used during encryption/decryption. The contribution of this thesis involves the verification of two symmetric key algorithms.


plaintext/ciphertext block of n-bit
key
Block Cipher Encryption/ Decryption
ciphertext/plaintext block of n-bit
Figure 2.8: Block Ciphers

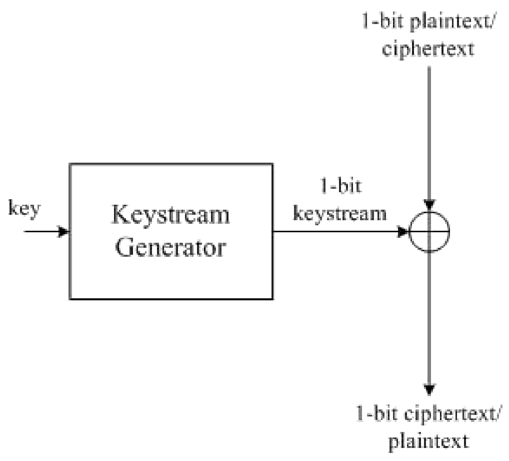


Figure 2.9: Stream Ciphers

Hence, this section focuses on symmetric key algorithms and any information relevant to the understanding of this thesis.

A symmetric key algorithm can be classified as a block cipher or a stream cipher depending on the type of input data. As the name suggested, block ciphers have the characteristic of operating on block of n-bit. Figure 2.8 illustrates the general structure of a block cipher. A block cipher takes a plaintext/ciphertext of n-bit and a key, and it computes the corresponding ciphertext/plaintext of n-bit. In contrast, stream ciphers encrypt/decrypt one bit at a time. Figure 2.9 shows the general structure of a stream cipher. Given a key, the keystream generator of the stream cipher produces a pseudorandom sequence of bits (keystream). During encryption, the keystream is XORed to the plaintext in a bitwise fashion. Similarly, decryption recovers the original plaintext by XORing the same keystream bits to its corresponding ciphertext bits. Thus, it usually requires the sender and the receiver to be synchronized in producing an identical keystream for accurate encryption and decryption. This type of cipher is known as synchronous stream cipher.

2.4 The KASUMI Cipher

In this section, we provide a description of the algorithm that we implemented in our first verification case study: the KASUMI cipher. Chapter 3 describes our implementation, optimization, and verification of the KASUMI algorithm.

KASUMI, also known as A5/3, is a block cipher in which encryption and decryption operations are identical with a reversal of the key schedule. Therefore, its implementation size can be reduced by nearly half which leads to its use in the confidentiality and integrity algorithms for the third generation mobile phone system. It operates on blocks of 64 bits and outputs in block of 64 bits.

In Figure 2.10, the algorithm begins by dividing a 64-bit input i into two 32-bit strings L_0 and R_0 which are then introduced into the first round of the cipher. KASUMI has a total of eight rounds in which all even rounds are identical and all odd rounds are the same. A 128-bit input key is used to derive the subkeys KL_i, KO_i, KI_i through XOR and bitwise permutations of the input key for each round $i = 1 \ldots 8$, and this is called the key schedule. At the end of the final round, the algorithm terminates with a 64-bit output.

Every round of the cipher in Figure 2.10 sends two 32-bit inputs L_{i-1} and R_{i-1} through the FL and FO subfunctions to compute two 32-bit outputs L_i and R_i defined as:

$$L_i = \begin{cases} R_{i-1} \oplus FO(FL(L_{i-1}, KL_i), KO_i, KI_i) & i = 1, 3, 5, 7 \\ R_{i-1} \oplus FL(FO(L_{i-1}, KO_i, KI_i), KL_i) & i = 2, 4, 6, 8 \end{cases} \quad (2.1)$$

$$R_i = L_{i-1} \qquad i = 1 \ldots 8 \quad (2.2)$$

The FL subfunction splits a 32-bit input into two 16-bit strings L and R, and it divides the 32-bit subkey KL_i into two 16-bit subkeys $KL_{i,1}$ and $KL_{i,2}$. A 32-bit output is obtained through the concatenation of the two 16-bit strings L' and R' defined as:

$$R' = R \oplus ROL(L \cap KL_{i,1}) \qquad i = 1 \ldots 8 \quad (2.3)$$

$$L' = L \oplus ROL(R' \cup KL_{i,2}) \qquad i = 1 \ldots 8 \quad (2.4)$$

In Figure 2.11, the FO subfunction starts by splitting a 32-bit input into two 16-bit strings L_0 and R_0 which are then fed into three identical rounds formed mainly by the FI subfunction. The 48-bit subkeys KO_i and KI_i are each divided into their three corresponding 16-bit subkeys $KO_{i,j}$ and $KI_{i,j}$ for $i = 1 \ldots 8$ and $j = 1 \ldots 3$. Each round of the FO subfunction is defined as:

$$R_j = FI(L_{j-1} \oplus KO_{i,j}, KI_{i,j}) \oplus R_{j-1} \qquad i = 1 \ldots 8, j = 1 \ldots 3 \quad (2.5)$$

$$L_j = R_{j-1} \qquad i = 1 \ldots 8, j = 1 \ldots 3 \quad (2.6)$$

The FI subfunction applies many bitwise permutations, truncations, and XOR to obscure the relationship between the input and the output. Its detailed description is not necessary to understand the remaining chapters of this thesis, and the reader can obtain that information in the specification of KASUMI [1].

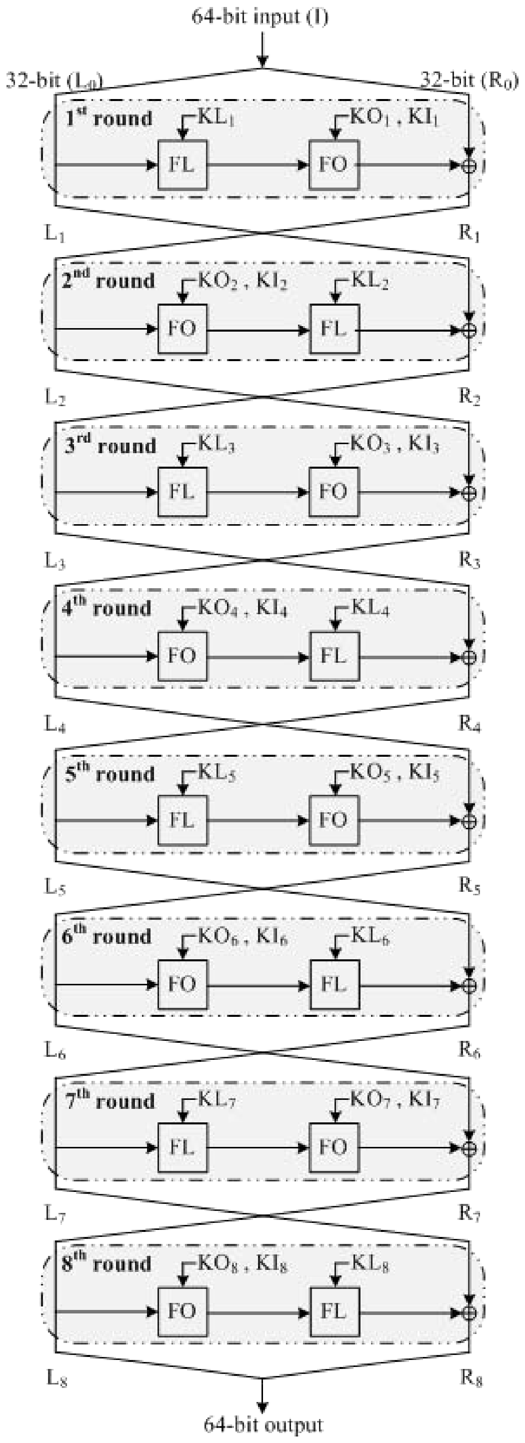


Figure 2.10: KASUMI Circuit

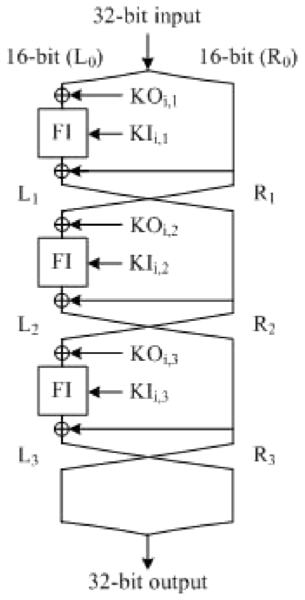


Figure 2.11: FO Circuit

2.5 Algebraic Terminology for Finite Fields and Normal Basis

In order to understand the description of the next cryptographic circuit, the reader is required to have some knowledge about abstract algebra. In this section, the following terms are defined: field, finite field, extension field, basis, primitive polynomial, polynomial basis, and normal basis.

Field: a field is an algebraic structure in which the operations of addition, subtraction, multiplication and division are define with the same rules that hold for normal arithmetic.

Finite field: finite field, also known as Galois field, is a field that contains finitely many elements. $\mathbb{F}_2$ denotes the finite field with two elements, 0 and 1.

Extension field: let L be a field. If K is a subset of L which is closed with respect to the field operations of addition and multiplication in L and the additive and multiplicative inverses of every element in K are in K, then L is an extension field of K. $\mathbb{F}_{2^m}$ denotes the extension field of $\mathbb{F}_2$ with 2^m elements, where each element is represented as a m-bit binary vector. Thus, a field $\mathbb{F}_{2^m}$ can be viewed as an m-dimensional vector space defined over $\mathbb{F}_2$.

Basis: basis is a set of vectors that, in a linear combination, can express every vector in a given vector space and such that no element of the set can be expressed as a linear combination of the others.

Primitive polynomial: let the set of vectors $\{1, \beta, \beta^2, \ldots, \beta^{m-1}\}$ be the basis for the field $\mathbb{F}_{2^m}$. Then, a polynomial with coefficients in $\mathbb{F}_2$ is a primitive polynomial if its root is $\beta \in \mathbb{F}_{2^m}$, and it has to be the smallest degree polynomial having β as a root.

Polynomial basis: let $\beta \in \mathbb{F}_{2^m}$ be the root of a primitive polynomial of degree m over $\mathbb{F}_2$. Then, the set of vectors $\{1, \beta, \beta^2, \ldots, \beta^{m-1}\}$ is called the polynomial basis or the canonical basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$.

Normal Basis: let the set of elements $P = \{\gamma, \gamma^2, \gamma^4, \ldots, \gamma^{2^{m-1}}\}$ be the basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$. Then, the basis P is called a normal basis and γ is called a normal element.

2.6 The Welch-Gong Cipher

This section describes the circuit structure proposed by Gong and Nawaz [14], and it explains how the cipher is initialized prior its use for encryption/decryption. Chapter 4-5 describes our implementation, optimization, and verification of the Welch-Gong cipher.

WG is a synchronous stream cipher that has been designed to produce a keystream with guaranteed randomness properties such as balance, long period, large and exact linear complexity, 3-level additive autocorrelation and ideal 2-level multiplicative autocorrelation. Also, it is resistant to Time/Memory/Data tradeoff attacks, algebraic attacks and correlation attacks.

Figure 2.12 shows the general circuit structure of the WG family cipher. It consists of a linear feedback shift register, also called LFSR, followed by a WG transformation block. The mathematical definition of the WG transformation $f(x)$ is described here:

$$f(x) = Tr_1^m(t(x+1)+1), x \in \mathbb{F}_{2^m} \tag{2.7}$$

The trace function $Tr_1^m(x)$ from $\mathbb{F}_{2^m} \rightarrow \mathbb{F}_2$ is defined as:

$$Tr_1^m(x) = \sum_{i=0}^{m-1} x^{2^{mi}}, x \in \mathbb{F}_{2^m} \tag{2.8}$$

The function $t(x)$ exists only if $m \neq 0 \, mod \, 3$, and it is defined as:

$$t(x) = x + x^{q_1} + x^{q_2} + x^{q_3} + x^{q_4}, x \in \mathbb{F}_{2^m} \tag{2.9}$$

For $k \in \mathbb{N}$ and $m = 3k-1$, q_i's are defined as:

$$\begin{aligned} q_1 &= 2^k + 1 \\ q_2 &= 2^{2k-1} + 2^{k-1} + 1 \\ q_3 &= 2^{2k-1} - 2^{k-1} + 1 \\ q_4 &= 2^{2k-1} + 2^k - 1 \end{aligned} \tag{2.10}$$

and for $m = 3k-2$, they are defined as:

$$\begin{aligned} q_1 &= 2^{k-1} + 1 \\ q_2 &= 2^{2k-2} + 2^{k-1} + 1 \\ q_3 &= 2^{2k-2} - 2^{k-1} + 1 \\ q_4 &= 2^{2k-1} - 2^{k-1} + 1 \end{aligned} \tag{2.11}$$

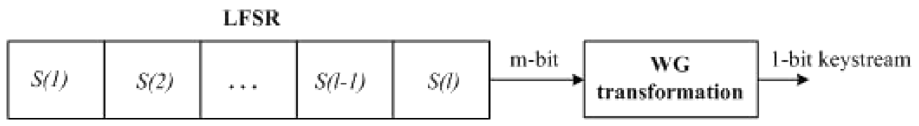


Figure 2.12: WG Family Circuit Structure

Various implementations of the WG cipher exist, and they have a different level of security depending on their design parameters. The number of LFSR stages l, the feedback polynomial of the LFSR, the number of bits m used for the WG transform computation, and the basis used to represent each field element affect the implementation complexity and security. A detailed design analysis of the WG family ciphers and how to select these design parameters are not relevant to the remaining chapters of this thesis, and the reader can find this information in Nawaz's PhD thesis [13].

The implementation presented in [14] is one of the two verification case studies of our thesis and it is described in this section. Figure 2.13 illustrates the 11-stage LFSR($l = 11$) over $\mathbb{F}_{2^{29}}(m = 29)$ in which its feedback polynomial $p(x)$ is defined as:

$$p(x) = \gamma \cdot x^{11} + x^{10} + x^8 + x^5 + x^2 + x + 1 \tag{2.12}$$

where $\beta \in \mathbb{F}_{2^{29}}$ and the set $\{1, \beta, \beta^2, \ldots, \beta^{m-1}\}$ forms the polynomial basis of $\mathbb{F}_{2^{29}}$. Also, β is the root of the primitive polynomial $g(x)$ defined as:

$$\begin{aligned} g(x) =& x^{29} + x^{28} + x^{24} + x^{21} + x^{20} + x^{19} + x^{18} + x^{17} + \\ & x^{14} + x^{12} + x^{11} + x^{10} + x^7 + x^6 + x^4 + x + 1 \end{aligned} \tag{2.13}$$

over $\mathbb{F}_2$. Note that $S_i \in \mathbb{F}_{2^{29}}$ and the feedback polynomial of the LFSR includes a normal basis multiplication with the element γ which is defined, in the polynomial form, as:

$$\begin{aligned} \gamma =& \beta^1 + \beta^2 + \beta^3 + \beta^5 + \beta^6 + \beta^7 + \beta^{10} + \beta^{11} + \beta^{12} + \beta^{13} + \\ & \beta^{14} + \beta^{15} + \beta^{16} + \beta^{17} + \beta^{20} + \beta^{23} + \beta^{24} + \beta^{26} + \beta^{27} \end{aligned} \tag{2.14}$$

γ is a normal element and it is used to define the normal basis of $\{\gamma^{2^0}, \gamma^{2^1}, \gamma^{2^2}, \ldots, \gamma^{2^{28}}\}$. For $m = 29$, $t(x)$ is defined as:

$$t(x) = x + x^{2^{10}+1} + x^{2^{19}+2^9+1} + x^{2^{19}-2^9+1} + x^{2^{19}+2^{10}-1}, x \in \mathbb{F}_{2^{29}} \tag{2.15}$$

and the WG transformation becomes:

$$f(x) = Tr_1^{29}(t(x+1)+1), x \in \mathbb{F}_{2^{29}} \tag{2.16}$$

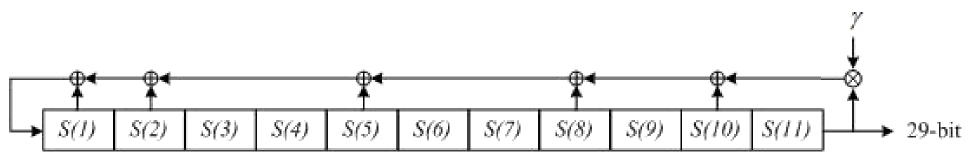


Figure 2.13: LFSR Feedback Polynomial

The WG transformation can be implemented in both the normal basis and the polynomial basis representation. However, the normal basis representation offers the following advantages:

- From the definition of a normal basis $\{\gamma, \gamma^2, \gamma^4, \ldots, \gamma^{2^{m-1}}\}$ in $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$, shifting the bits of an element x cyclically to right by i position computes x^{2^i}. Thus, the squaring operation consists of rewiring the bits of a field element in terms of hardware. This operation comes up multiple times in the definition of $t(x)$.
- In normal basis representation, the all ones vector represent 1. Therefore, the operation of adding 1 to a field element becomes inverting all the bits of that element. This operation occurs twice in the definition of the WG transformation, and it requires $2m$ inverters given a the field $\mathbb{F}_{2^m}$.
- In normal basis representation, the trace function $Tr_1^m(x)$ of any normal basis elements is one. Therefore, the trace function of a field element is done by adding all the bits of that element over $\mathbb{F}_2$.

Nawaz [13] has showed that equation (2.15) can be written in order to decrease the implementation area of WG as follows:

$$\begin{aligned} t(x) = &\ x + x^{2^{10}+1} + x^{2^{19}+2^9+1} + \ x^{2^{19}-2^9+1} + x^{2^{19}+2^{10}-1} \\ = &\ x + x^{2^{10}+1} + x^{2^{19}+2^9+1} + \ x^{(2^{10}-1)\cdot 2^9+1} + x^{2^{19}+(2^{10}-1)} \end{aligned} \tag{2.17}$$

Figure 2.14 shows the implementation of the WG transformation for $\mathbb{F}_{2^{29}}$ according to the properties of its normal basis representation. $(x)^{-1}$ denotes the normal basis inversion of x in $\mathbb{F}_{2^{29}}$. $x >> i$ denotes cyclic shift of x, i positions to the right where $c \geq 0$. $x \otimes y$ means normal basis multiplication of x and y in $\mathbb{F}_{2^{29}}$. $x \oplus y$ means bitwise XOR of x and y. Finally, $\underline{\oplus}(x)$ means XORing all 29 bits of x over $\mathbb{F}_2$.

The WG implementation described above can be used with key sizes of 80, 96, 112 and 128 bits. In addition, it can take either a 32 or a 64 bits Initialization Vector (IV). For the purpose of this thesis, only a 80-bit key and a 32-bit IV are

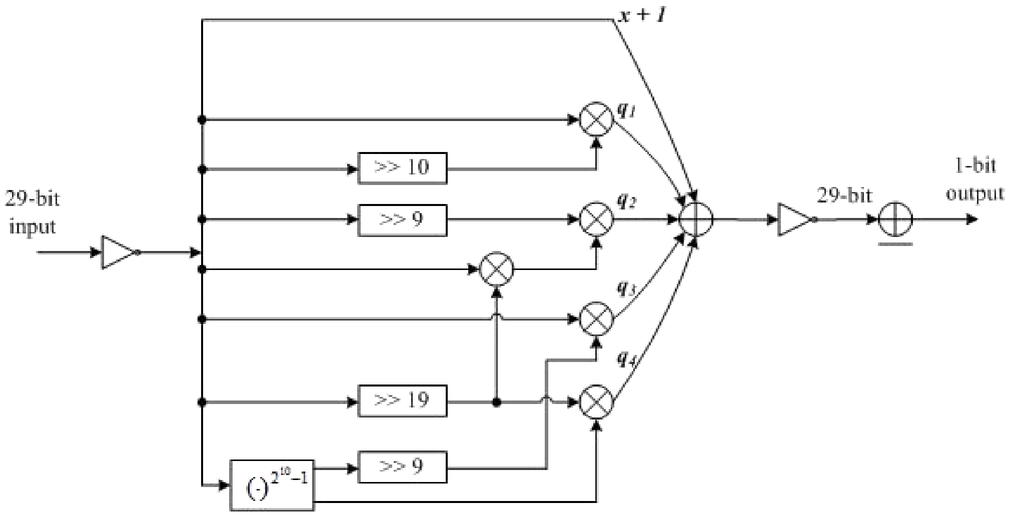


Figure 2.14: Implementation of WG Transformation

considered. The first step in initializing the cipher is to load the key bits and IV bits into the LFSR. Each stage i of the LFSR is denoted by *s(i)* or more precisely $S_{1,...,29}(i)$ where $1 \leq i \leq 11$. Similarly, the key bits are denoted as $k_{1,...,j}$ where $1 \leq j \leq 80$ and the IV bits as $IV_{1,...,k}$ where $1 \leq k \leq 32$. The 80-bit key and the 32-bit IV are loaded into the LFSR as described:

$S_{1,...,16}(1) = k_{1,...,16}$ $\quad S_{1,...,16}(2) = k_{17,...,32}$ $\quad S_{1,...,16}(3) = k_{33,...,48}$

$S_{1,...,16}(4) = k_{49,...,64}$ $\quad S_{1,...,16}(5) = k_{65,...,80}$ $\quad S_{1,...,16}(9) = k_{1,...,16}$

$S_{1,...,16}(10) = k_{17,...,32} \oplus 1$ $\quad S_{1,...,16}(11) = k_{33,...,48}$

$S_{17,...,24}(1) = IV_{1,...,8}$ $\quad S_{17,...,24}(2) = IV_{9,...,16}$ $\quad S_{17,...,24}(3) = IV_{17,...,24}$

$S_{17,...,24}(4) = IV_{25,...,32}$

To complete the LFSR loading phase, all the undefined bits of the LFSR are set to zero. Once the LFSR has been loaded with the key and IV, the circuit is run for 44 clock cycles with the additional connection shown in Figure 2.15. Once the key has been initialized, the feedback connection in Figure 2.15 is disconnected from the LFSR and the first bit of the keystream is given by the 1-bit output of the WG transformation after one clock cycle.

This section has given the description of the Welch-Gong cipher, its mathematical formulation, the circuit structure of its 29-bit implementation based on normal basis, and how to initialize the cipher with the key and the initialization vector.

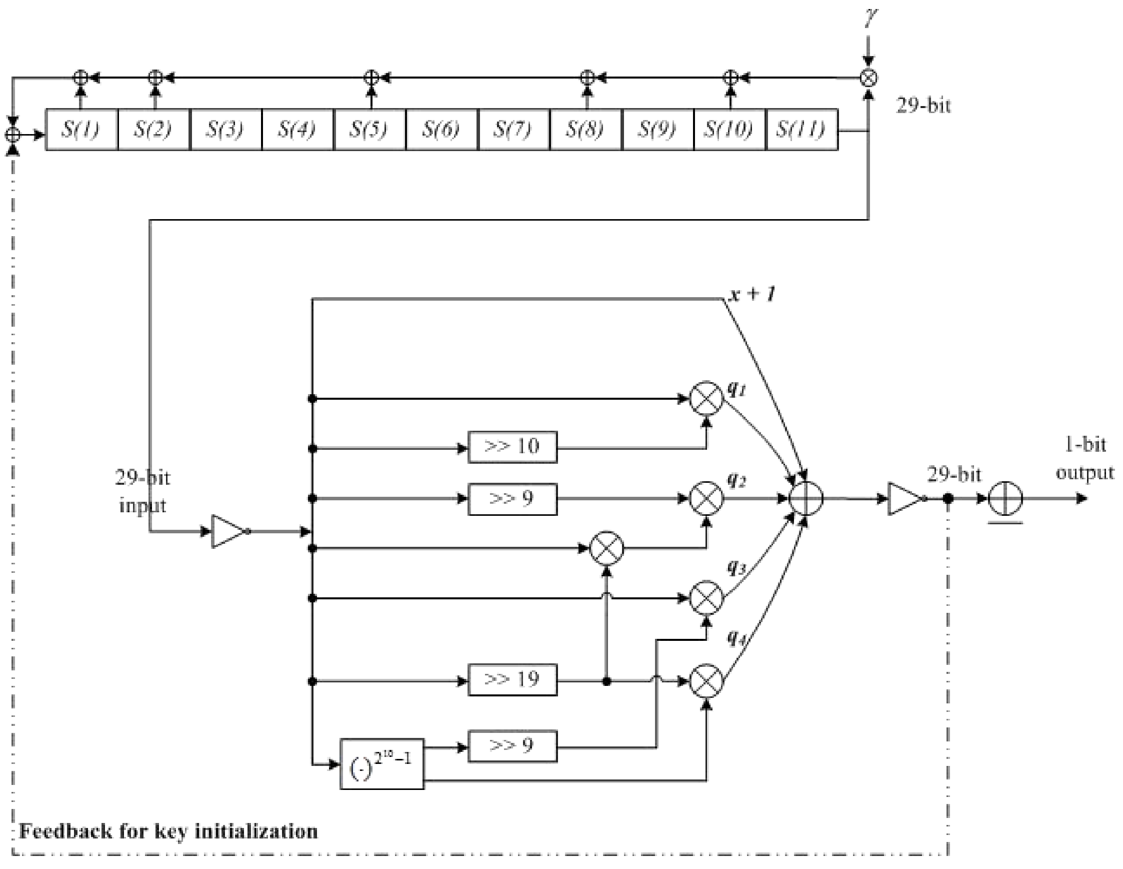


Figure 2.15: Key Initialization Phase

Chapter 3

KASUMI: Design, Optimizations and Verification

The contribution of this thesis is to verify two cryptographic circuits using the verification strategy of completion functions at the register transfer level. The goal of this chapter is to show how completion functions are used in the verification of the KASUMI cipher, and this chapter is organized as follows. Section 3.1 describes the design process of our non-pipelined KASUMI implementation, and Section 3.2 explains how RTL simulation is used to verify this non-pipelined implementation. Section 3.3 illustrates how the optimization of pipelining produces three pipelined implementations of KASUMI. Section 3.4 introduces the verification methodologies used to verify these three pipelined implementations, and Section 3.5 shows the results of our verification.

3.1 The First Design

The goal of this chapter is to explore completion functions in the verification of pipelined implementations of KASUMI against its non-pipelined specification at the register transfer level. Therefore, we first need to build a non-pipelined implementation of KASUMI such that it can be used as the specification.

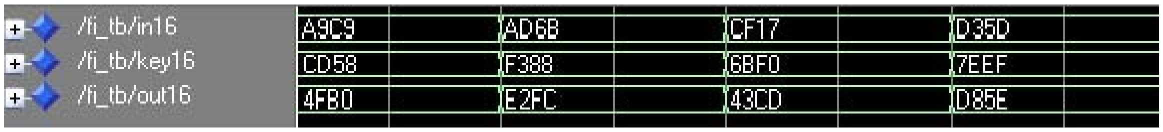
The circuit structure of the KASUMI cipher and its circuit operation were discussed in Section 2.4. Our research work began with the design of a purely combinational or non-pipelined implementation of the KASUMI cipher using the VHDL hardware description language. Since the KASUMI circuit has an highly recursive and modular structure formed by several components (*FI*, *FO*, *FL*), we have taken a bottom-up modular design approach. We started by implementing the *FI* component because it is the lowest level component in the KASUMI circuit. Then, the *FO* component was built using a composition of XOR gates, bitwise permutations and multiple *FI* components. The *FL* component was implemented independently of the other components following the description in Section 2.4. With the implementation of the *FO* and the *FL* components, we constructed all even and odd rounds of KASUMI. Our first VHDL implementation of KASUMI was completed by connecting each round of the cipher with its corresponding subkey that was generated by the key scheduling algorithm described in the specification of KASUMI [1].

This first implementation was synthesized using Mentor Graphics Precision RTL synthesis tool. On an Altera Stratix II series field programmable gate arrays (FPGA) device EP2S15F484C, our combinational implementation of KASUMI has an area of 4748 logic elements and a performance of 9 MHz when registers are inserted at the inputs and at the outputs. The verification of this combinational implementation is described in the next section.

3.2 Formulating The Specification

Various verification technologies and strategies can be applied for the verification of a circuit at different level of abstraction. Our work focuses on exploring the verification strategy of completion functions in the verification of pipelined implementations with respect to a combinational specification at the register transfer level. Therefore, we are required to verify the correctness of our combinational implementation of KASUMI because it will be used as the specification for the verification of the pipelined implementations of KASUMI.

Since the main focus of our work is to verify pipelined circuits and a set of test vectors are provided in the specification of KASUMI [2], we have used conventional RTL simulation for the verification of the combinational implementation of KASUMI. In [2], various sets of test vectors are given for all three major components (*FI*, *FL*, *FO*) as well as for the whole KASUMI cipher. Thus, the combinational implementation is defined as correct if and only if it generates the expected outputs given its associated set of test vectors. However, there are still possibilities for a subtle bug to be undetected because our RTL simulation does not cover all possible inputs to the KASUMI cipher.

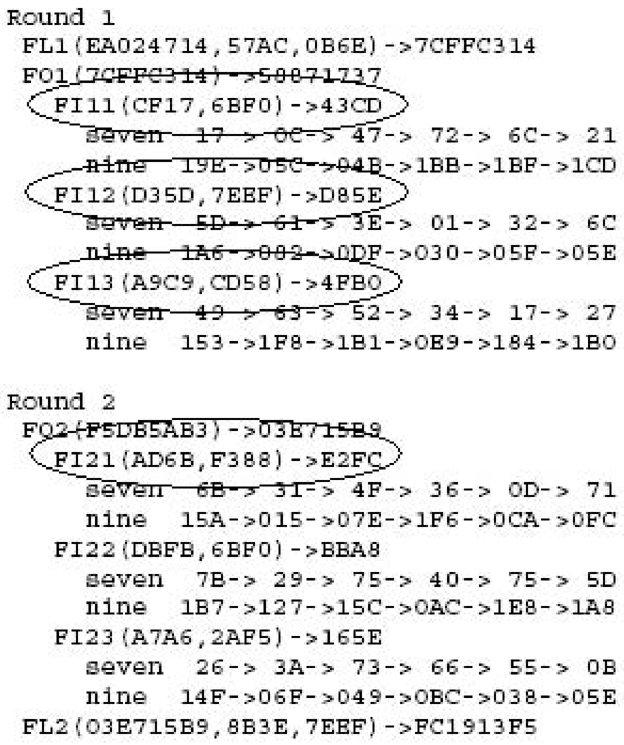


```
/fi_tb/in16    A9C9  AD6B  CF17  D35D
/fi_tb/key16   CD58  F388  6BF0  7EEF
/fi_tb/out16   4FB0  E2FC  43CD  D85E
```

Figure 3.1: FI Test Vectors Results

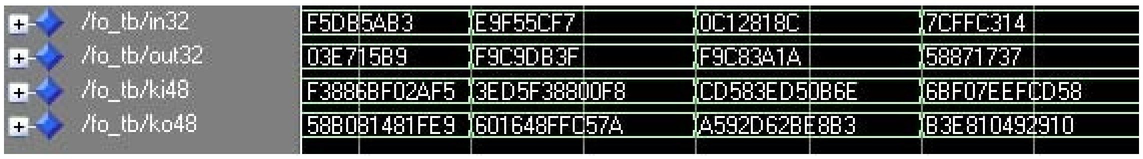
Similar to our modular bottom-up design approach, we verified the combinational KASUMI, components by components, starting from the smallest building blocks. This simulation methodology would localize an implementation bug to a component. For all of the RTL simulation runs, we have used Mentor Graphics ModelSim. Figure 3.1 shows the simulation results of the FI component in which $in16$ is the 16-bit input, $key16$ is the 16-bit subkey KL and $out16$ is the 16-bit output. Figure 3.2 illustrates fragments of test vectors in the specification of KASUMI [2], and the circled test vectors correspond to the waveform shown in Figure 3.1. Notice that all vectors holds hexadecimal values and that only a subset of test vectors from the specification of KASUMI[2] are presented here. Figure 3.3 depicts the verification of the FO component, where $in32$ and $out32$ are the 32-bit input and ouput, $ki48$ and $ko48$ are the 48-bit subkeys KI and KO respectively. Figure 3.4 illustrates the test vectors results of the FL component in which $in32$ is the 32-bit input, $out32$ is the 32-bit output and $kl32$ is the 32-bit subkey KL. Finally, we have verified that our combinational implementation of KASUMI produces the expected 64-bit output $out64$ given its corresponding 64-input $in64$ and 128-bit key $key128$ as shown in Figure 3.5. Through the key scheduling algorithm, the 128-bit key $key128$ has properly transformed into three sets (KL, KI, KO) of eight subkeys for all eight rounds of the KASUMI cipher as shown in Figure 3.5. For all three sets (KL, KI, KO), the subkeys of the first round are denoted by (7) and the subkeys of the eighth round are denoted by (0).

Our combinational implementation of KASUMI was verified through various sets of test vectors, and no bugs were found. Therefore, this implementation can serve as a specification for the pipelined implementations of KASUMI that are described in the next section, when using the verification strategy of completion functions.



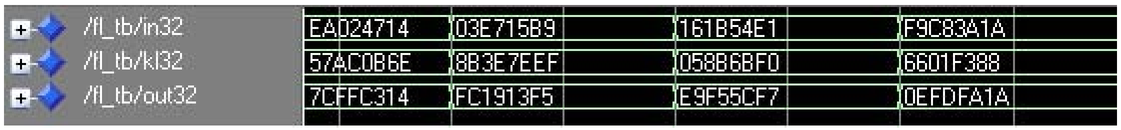
```
Round 1
 FL1(EA024714,57AC,0B6E)->7CFFC314
 FO1(7CFFC314)->58871737
  FI11(CF17,6BF0)->43CD
    seven  17-> 0C-> 47-> 72-> 6C-> 21
    nine  19E->05C->04B->1BB->1BF->1CD
  FI12(D35D,7EEF)->D85E
    seven  5D-> 61-> 3E-> 01-> 32-> 6C
    nine  1A6->082->0DF->030->05F->05E
  FI13(A9C9,CD58)->4FB0
    seven  49-> 63-> 52-> 34-> 17-> 27
    nine  153->1F8->1B1->0E9->184->1B0

Round 2
 FO2(F5DB5AB3)->03E715B9
  FI21(AD6B,F388)->E2FC
    seven  6B-> 31-> 4F-> 36-> 0D-> 71
    nine  15A->015->07E->1F6->0CA->0FC
  FI22(DBFB,6BF0)->BBA8
    seven  7B-> 29-> 75-> 40-> 75-> 5D
    nine  1B7->127->15C->0AC->1E8->1A8
  FI23(A7A6,2AF5)->165E
    seven  26-> 3A-> 73-> 66-> 55-> 0B
    nine  14F->06F->049->0BC->038->05E
 FL2(03E715B9,8B3E,7EEF)->FC1913F5
```

Figure 3.2: Fragments of Test Vectors from KASUMI Specification



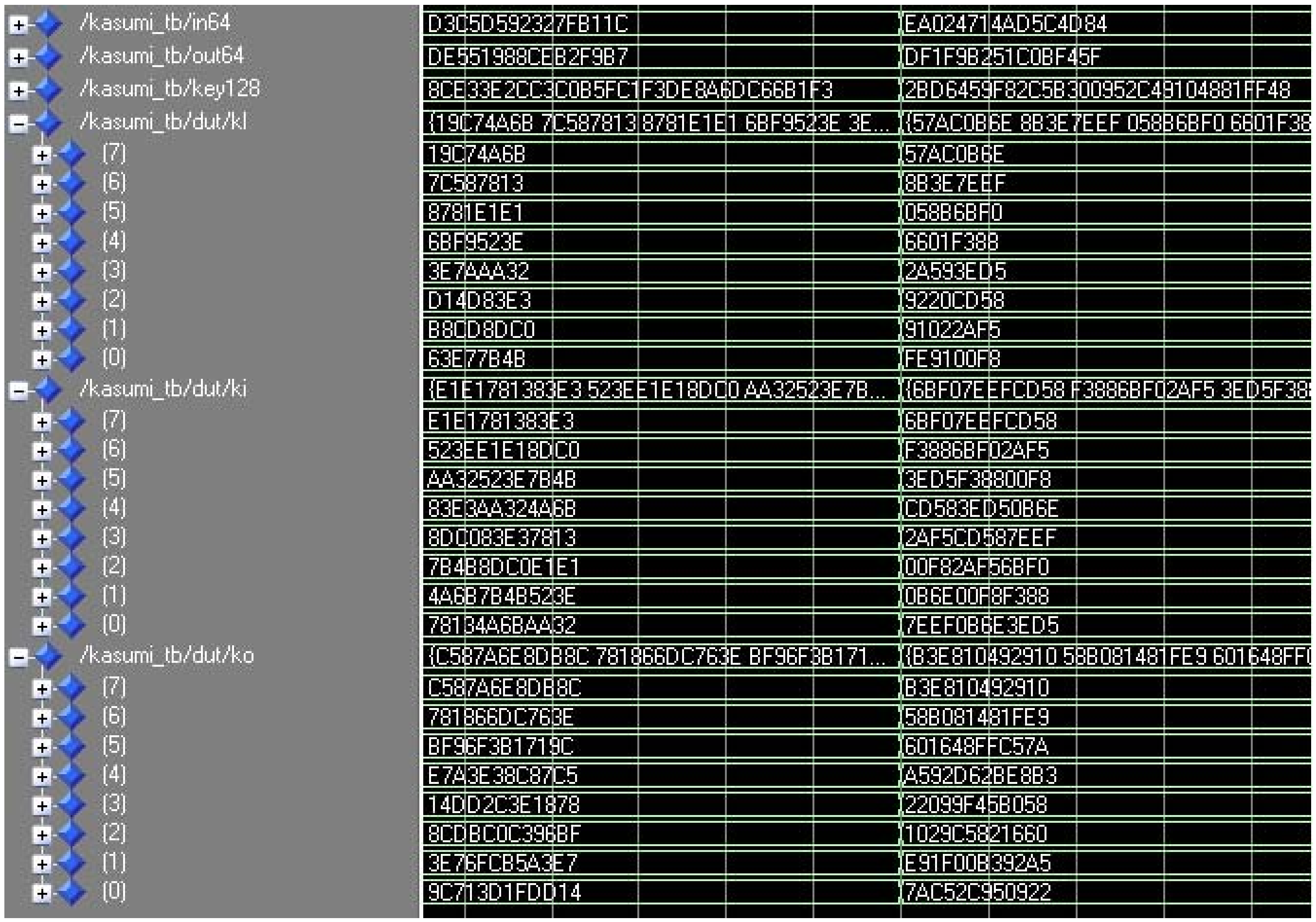
```
/fo_tb/in32    F5DB5AB3      E9F55CF7      0C12818C      7CFFC314
/fo_tb/out32   03E715B9      F9C9DB3F      F9C83A1A      58871737
/fo_tb/ki48    F3886BF02AF5  3ED5F38800F8  CD583ED50B6E  6BF07EEFCD58
/fo_tb/ko48    58B081481FE9  601648FFC57A  A592D62BE8B3  B3E810492910
```

Figure 3.3: FO Test Vectors Results



```
/fl_tb/in32    EA024714  03E715B9  161B54E1  F9C83A1A
/fl_tb/kl32    57AC0B6E  8B3E7EEF  058B6BF0  6601F388
/fl_tb/out32   7CFFC314  FC1913F5  E9F55CF7  0EFDFA1A
```

Figure 3.4: FL Test Vectors Results



```
/kasumi_tb/in64      D3C5D592327FB11C                                    EA024714AD5C4D84
/kasumi_tb/out64     DE551988CEB2F9B7                                    DF1F9B251C0BF45F
/kasumi_tb/key128    8CE33E2CC3C0B5FC1F3DE8A6DC66B1F3                    2BD6459F82C5B300952C49104881FF48
/kasumi_tb/dut/kl    {19C74A6B 7C587813 8781E1E1 6BF9523E 3E...          {57AC0B6E 8B3E7EEF 058B6BF0 6601F38
  (7)                19C74A6B                                            57AC0B6E
  (6)                7C587813                                            8B3E7EEF
  (5)                8781E1E1                                            058B6BF0
  (4)                6BF9523E                                            6601F388
  (3)                3E7AAA32                                            2A593ED5
  (2)                D14D83E3                                            9220CD58
  (1)                B8CD8DC0                                            91022AF5
  (0)                63E77B4B                                            FE9100F8
/kasumi_tb/dut/ki    {E1E1781383E3 523EE1E18DC0 AA32523E7B...            {6BF07EEFCD58 F3886BF02AF5 3ED5F38
  (7)                E1E1781383E3                                        6BF07EEFCD58
  (6)                523EE1E18DC0                                        F3886BF02AF5
  (5)                AA32523E7B4B                                        3ED5F38800F8
  (4)                83E3AA324A6B                                        CD583ED50B6E
  (3)                8DC083E37813                                        2AF5CD587EEF
  (2)                7B4B8DC0E1E1                                        00F82AF56BF0
  (1)                4A6B7B4B523E                                        0B6E00F8F388
  (0)                78134A6BAA32                                        7EEF0B6E3ED5
/kasumi_tb/dut/ko    {C587A6E8DB8C 781866DC763E BF96F3B171...            {B3E810492910 58B081481FE9 601648FF
  (7)                C587A6E8DB8C                                        B3E810492910
  (6)                781866DC763E                                        58B081481FE9
  (5)                BF96F3B1719C                                        601648FFC57A
  (4)                E7A3E38C87C5                                        A592D62BE8B3
  (3)                14DD2C3E1878                                        22099F45B058
  (2)                8CDBC0C396BF                                        1029C5821660
  (1)                3E76FCB5A3E7                                        E91F00B392A5
  (0)                9C713D1FDD14                                        7AC52C950922
```

Figure 3.5: KASUMI Test Vectors Results

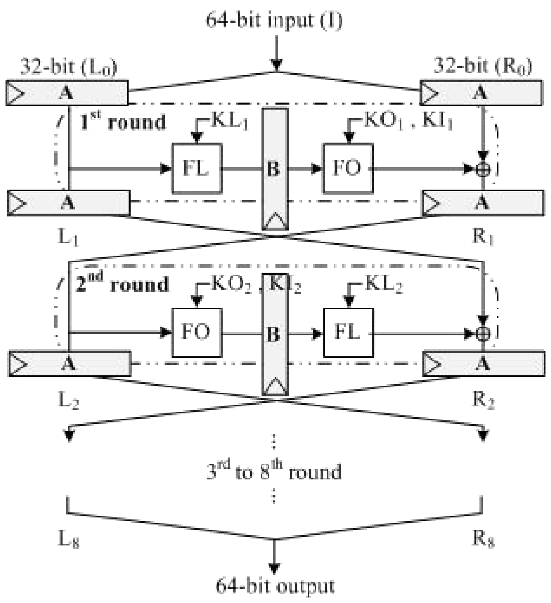
3.3 Pipelined Implementations

In Section 3.1 and 3.2, we have explained how we obtained the combinational specification of KASUMI. In this section, we describe how the optimization technique of pipelining was used to create three pipelined implementations of KASUMI.

KASUMI was our first case study in exploring the verification strategy of completion functions at the RTL abstraction level. Therefore, we have used pipelining only to create three pipelined implementations of KASUMI, without any additional optimizations. For pure datapath circuits, the optimization technique of pipelining simply inserts registers at various locations of the implementation to divide the circuit into pipeline stages. In our second verification case study, we have explored the completion functions further by applying more sophisticated optimizations to the Welch-Gong cipher.

Since the KASUMI circuit is formed by eight rounds of *FL* and *FO* components, it was natural to insert registers at the beginning of each round so that all pipeline stages have an equal amount of gate delay. Thus, our first pipelined implementation of KASUMI consists of a 8-stage pipeline where the set of registers A are inserted at the beginning of each round of the cipher as shown in Figure 3.6. Note that both *FL* and *FO* are purely combinational circuits and that the set of registers B are not present in the 8-stage implementation of KASUMI. For our second pipelined implementation of KASUMI, we have further divided each pipeline stage of the 8-stage implementation into two pipeline stages by inserting the set of registers B between the *FL* and the *FO* components as shown in Figure 3.6. Thus, our second pipelined implementation has 16 stages in which each stage contains either the *FL* or the *FO* components. For our third pipelined implementation of KASUMI, we have divided the *FO* component of the 16-stage implementation of KASUMI into a 3-stage pipeline by inserting the set of registers C as depicted in Figure 3.7. Hence, our third pipelined implementation of KASUMI has 32 stages in which each stage contains either the *FL* or the *FI* components.

The area and performance of our pipelined implementations are shown in Figure 3.8, where LUT denotes lookup table. All of our area and performance results were synthesized on an Altera Stratix II series FPGA device EP2S15F484C using Mentor Graphics PrecisionRTL. Thus, our 8-stage implementation of KASUMI has an area of 4518 LUTs with a throughput of 3.9 Gbps. In Figure 3.8, we have included the latest(2005) optimized implementations of KASUMI by Kitsos *et al.* [11]. All of their implementations were synthesized onto the Xilinx FPGA device XCV300E-8BG432. Their four pipelined implementations are divided into two main



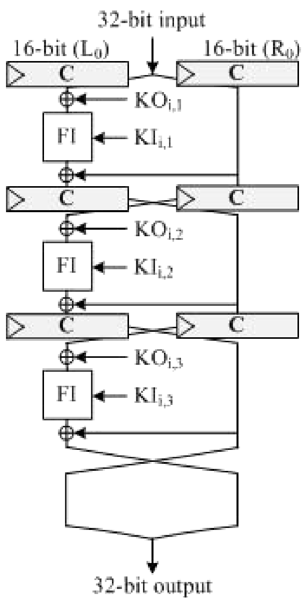


Figure 3.6: Registers Locations in KASUMI Figure 3.7: Registers Locations in FO

architectures: $8R$ and $2R$. In the $8R$ architecture, Kitsos *et al.* [11] have defined each round of the KASUMI circuit to be a pipeline stage (similar to our 8-stage implementation). For the $2R$ architecture, the optimization of area re-use has been applied to reduce the number of pipline stages to two, where one stage is an odd round of KASUMI and the second stage is an even round of the cipher. Each of these two architectures are further divided into two different implementations, where *Comb* denotes that the *FI* component is implemented with combinational logic and *ROM* means that the *FI* block is implemented as lookup table with read-only memory (ROM). In Figure 3.8, both $8R\_ROM$ and $2R\_ROM$ require less LUTs for their implementations because their *FI* components are implemented with read-only memory. $8R\_ROM$ uses 2752 bytes of read-only memory, and $2R\_ROM$ uses 688 bytes of read-only memory. Finally, note that our 8-stage implementation has only half the throughput of the 8R_Comb implementation. This difference is due to the fact that Kitsos *et al.* have used Double Edge Trigger (DET) pipelining in which the data are transferred between two successive registers in both rising and falling edges of the clock signal. For our pipelined implementations of KASUMI, we have used conventional single edge-triggered registers in which the data are transferred between two successive registers only at rising edges of the clock signal. Given the same circuit, replacing its single edge-triggered registers with double edge-triggered ones would double its throughput because data are processed at both rising and falling edges of the clock signal instead of being processed only at rising edges of the clock signal. In other words, a double edge-triggered pipeline would give two outputs per cycle (one at rising edge, one at falling edge of the clock signal) given that its single edge-triggered version produces one output per cycle.

In this section, we have provided descriptions of our pipelined implementations of KASUMI. Next section describes the verification methodology that we have applied with the verification of these three pipelined implementations.

3.4 Verification Methodology

In the verification of the three pipelined implementations of KASUMI, we have devised a set of guidelines in using the completion functions efficiently. This section describes our verification methodologies, and it is organized as follows. Section 3.4.1 provides the general steps of our verification, and a contrived example is used to show how completion functions are applied in our verification. Section 3.4.2 describes two important VHDL features used in our verification methodology. Section 3.4.3 shows how our methodology fits into a single entity design environment. Then, we show the advantages of our methodology by applying it to a multiple entities design environment in Section 3.4.4.

3.4.1 General Guidelines

For the verification of our pipelined implementations of KASUMI, we have combined the verification technology of equivalence checking with the verification strategy of completion functions. A disadvantage of combinational equivalence checking is that it cannot verify pipelined implementations against a non-pipelined specification as it is limited to comparing the next-state equations of compare points based only on the combinational circuitry driving the points. By applying the verification strategy of completion functions with equivalence checking, pipelined implementations can be verified against its non-pipelined specification. The general outline of this verification has the following steps:

1. Create a first purely combinational implementation of the circuit as described in Section 3.1. This is usually also the first step in designing circuits.

2. Verify the correctness of the combinational implementation using suitable verification technologies and strategies as explained in Section 3.2. This ver-

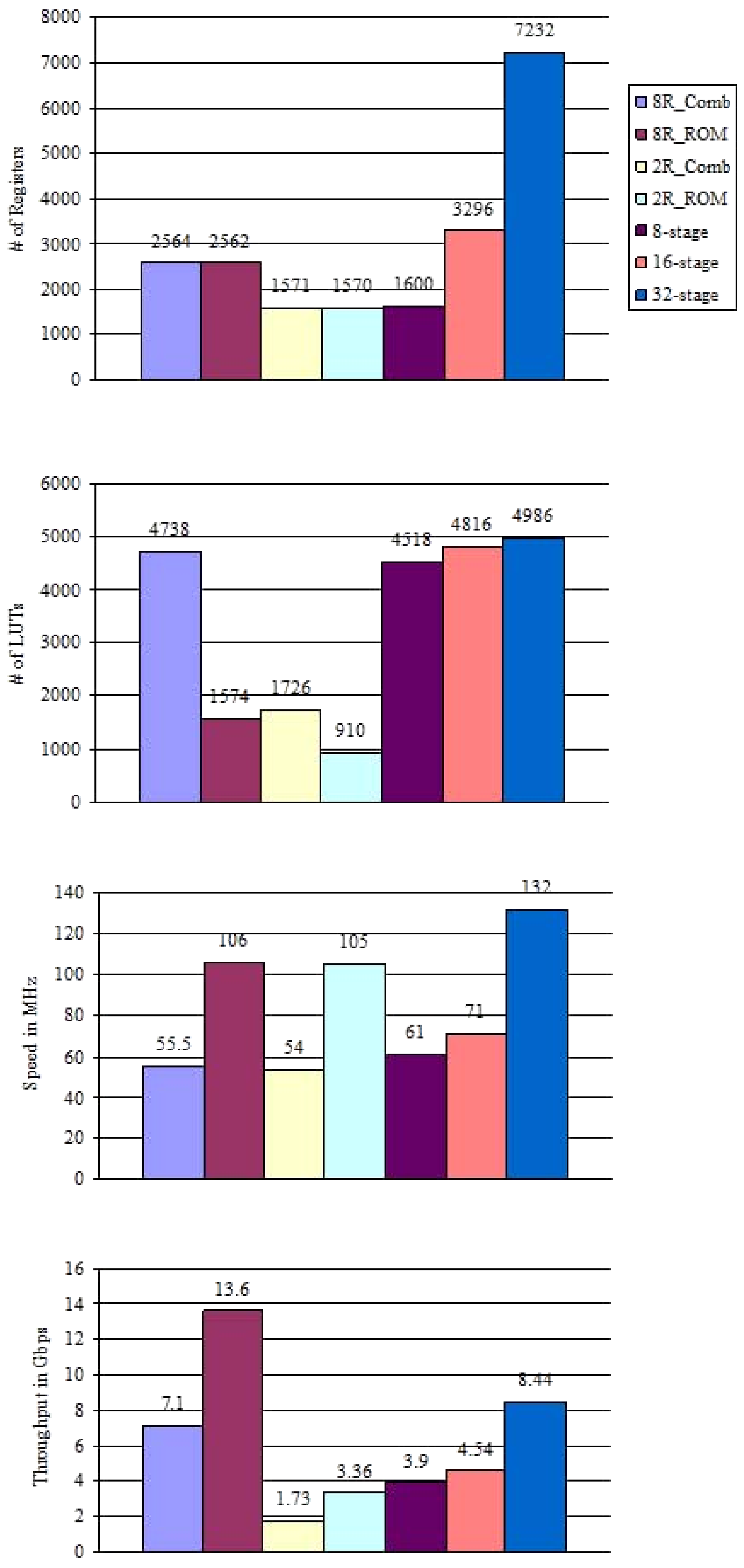


Figure 3.8: Area and Performance Results of Various KASUMI Designs

ified combinational implementation is used as the specification if there are additional optimized implementations of the same circuit to be verified.

3. Optimize the combinational implementation with optimization techniques such as pipelining or area re-use to achieve desired area/performance (described in Section 3.3).

4. For each pipeline stage of the optimized design, build its completion function such that it describes the effects on the programmer visible parts upon completing the instruction in that pipeline stage. Depending on which pipeline stage is under verification, the completion functions are either connected to the stage registers or to the outputs of the stage combinational circuitry. This is explained in this section.

5. Using the completion functions, verify the optimized design stage-by-stage starting from the last pipeline stage. The verification of each pipeline stage requires one equivalence check. The optimized design is correct if the equivalence checker returns true when verifying the completion function of the first stage against the specification (verified purely combinational implementation). Therefore, there are $n+1$ verification obligations for a n-stage pipeline.

To demonstrate the last two steps of our methodology mentioned above, we have used the 2-stage verification example shown in Figure 3.9. Note that Figure 3.9 has already appeared in Section 2.2 for the general explanation of the completion functions approach that is used for the verification of the 2-stage implementation in Figure 2.2. This section describes the details of applying the completion functions approach in combination with equivalence checking.

The verification of our 2-stage contrived example in Figure 3.9 requires a total of three verification obligations. The first verification obligation VO_1 is to verify the correctness of the last pipeline stage (i.e. second stage) with respect to the completion function C_2 that describes the correct behaviour. Figure 3.10 illustrates the first equivalence check which corresponds to our first verification obligation VO_1. On the left hand side of Figure 3.10, the completion function $C2$ is connected to the stage registers *X2* and emulates the effects of completing the instructions in the second pipeline stage. This corresponds to taking the register *R2* from state “-” to state “A” (through completion function C_2) in Figure 3.9. On the right hand side of Figure 3.10, the combinational block *A2* is connected to the stage registers *X2* and this circuit is the original pipelined implementation. This corresponds to taking the register *R2* from state “-” to state “A” (through taking

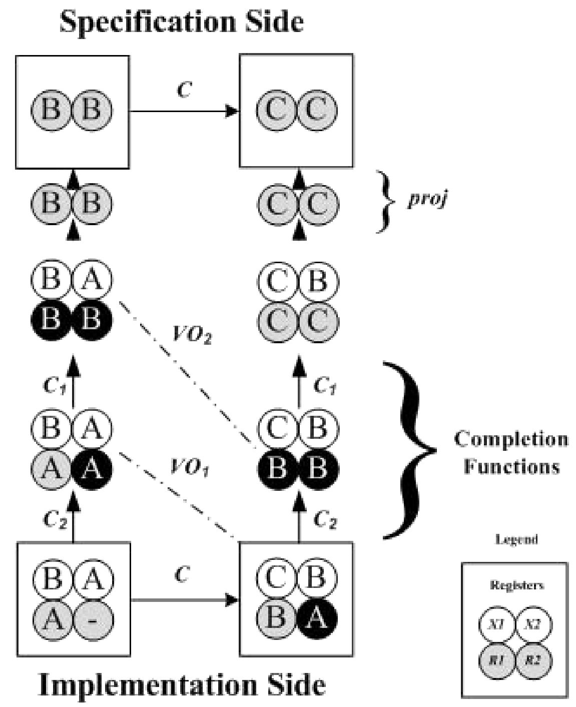


Figure 3.9: 2-Stage Pipeline Verification Structure

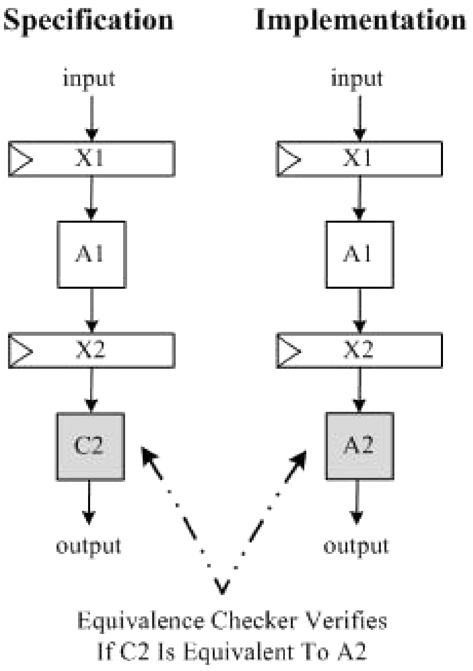


Figure 3.10: First Verification Obligation

an implementation step) in Figure 3.9. Note that the combinational block *A2* of Figure 3.10 is viewed as the implementation step. Thus, the first equivalence check verifies if the combinational block *A2* has the same behaviour as the completion function $C2$.

In Figure 3.9, the second verification obligation VO_2 is used to verify the correctness of the first pipeline stage according to the behaviour given by the completion function C_1. Similarly, Figure 3.11 shows the equivalence check that is associated to our second verification obligation VO_2. On the left hand side of Figure 3.11, the effect of completing the instruction in the first stage is achieved by connecting the completion function of the first stage (C_1) to the stage registers *X1*. This corresponds to taking the set of registers (*R1*, *R2*) from the state (A, $-$) to the state (B, B) (through completion function C_1) in Figure 3.9. On the right hand side of Figure 3.11, the completion function of the second stage (C_2) is connected to the pipelined implementation through the outputs of the combinational block *A1*. This corresponds to taking the set of registers (*R1*, *R2*) from the state (A, $-$) to the state (B, B) (through completion functions C_2 and an implementation step) in Figure 3.9. Note that the combinational block *A1* of Figure 3.11 is viewed as the implementation step in Figure 3.9. Hence, the second equivalence check verifies if

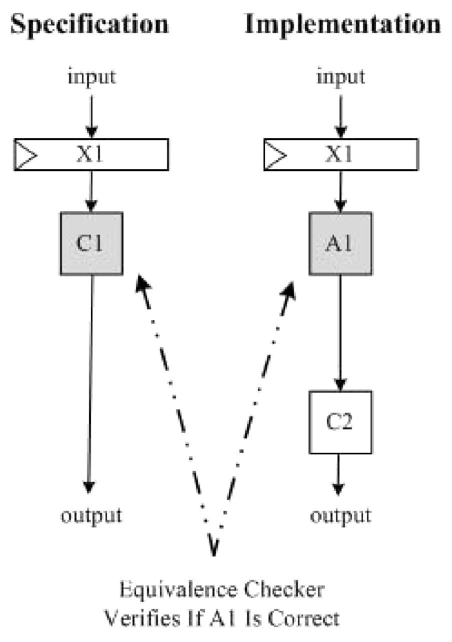


Figure 3.11: Second Verification Obligation

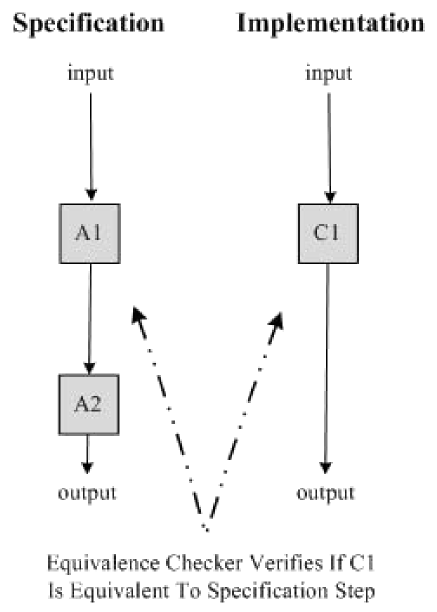

Specification
Implementation
input
input
A1
C1
A2
output
output
Equivalence Checker Verifies If C1 Is Equivalent To Specification Step

Figure 3.12: Third Verification Obligation

the combinational block *A1* is correct with respect to the behaviour given by the completion function C_1.

The previous two verification obligations (VO_1 and VO_2) have verified both pipeline stages under the assumption that the completion functions are bug-free. Hence, the third verification obligation is to verify the correctness of the completion functions with respect to the specification. Figure 3.12 depicts the equivalence run for the third verification obligation. The equivalence checker verifies if the completion function of the first stage (C_1) is equivalent to the non-pipelined specification. In Figure 3.9, this corresponds to showing that the diagram commutes by proving that the completion function C_1 is equivalent to the specification step. Thus, the verification of a 2-stage pipeline is completed with 3 verification obligations.

As mentioned earlier, different completion functions are connected to different locations of the circuit depending on the pipeline stage being verified. In our 2-stage contrived example, the completion function of the second stage C_2 is connected to the stage register *X2* for the verification of the second stage in Figure 3.10. For the verification of the first stage in Figure 3.11, the completion function C_2 is connected to the outputs of the combinational block *A1*.

In VHDL, the process of connecting the completion functions to various registers can be cumbersome because the registers buried inside a VHDL entity cannot be

```
1  if <boolean condition> generate
2   ...  hardware to be conditionally synthesized ...
3  end generate;
4  ...  hardware to be synthesized ...
```

Figure 3.13: If-Generate Statement

accessed without declaring additional output ports. It is undesirable to have all the completion functions synthesized during equivalence checking because it would increase the size of the circuit being compared. Therefore, there is a need for a methodology to efficiently control the completion functions in VHDL. This can be achieved by a combination of the "if-generate" statements and the "generic" parameters in VHDL, which are introduced in the next subsection.

3.4.2 Background of VHDL Features

This subsection provides the background of two important VHDL features, if-generate statements and generic parameters, used as part of our verification methodology. Section 3.4.3 and 3.4.4 illustrate how these two VHDL features are applied with our verification methodology.

If-generate statements are evaluated at elaboration time to conditionally create some hardware, and they are similar to #ifdef in C. Figure 3.13 shows some VHDL code fragments which include the if-generate statement. Line 1-3 of Figure 3.13 is the general structure of an if-generate statement. Line 2 is the body of the if-generate statement, and its hardware is generated if the *boolean condition* of line 1 is true. All VHDL codes outside the scope of the if-generate statements are synthesized into actual hardware as shown on line 4 of Figure 3.13. The *boolean condition* controls which completion function to be synthesized and how they would be connected to the circuit depending on the pipeline stage under verification. The VHDL generic parameters can be used as control parameters, and they are introduced next.

Generic parameters are evaluated at elaboration time, and they are analogous to #define in C. Generic parameters are constant values and they are declared as part of a VHDL entity declaration as shown in Figure 3.14. Unlike the port parameters of line 3 in Figure 3.14, the generic parameters of line 2 are not actual ports of the circuit and they would not be synthesized into any additional signals or hardware. Generic parameters are commonly used to modify the bit width of

```
entity <entity name> is
  generic( ...  list of generic parameters ...);
  port( ...  list of input and output ports ...);
end entity;
```

Figure 3.14: Generic Parameters

an implementation, such as the bit width of an adder, during hardware synthesis. Generic parameters can only be assigned as constant values. A common pitfall in using generic parameters is to assign constant signals to generic parameters. However, the VHDL compiler interprets the “signal” type to have dynamic values that can be modified during circuit operation. Generic parameters can help the verification engineer to manage the completion functions in VHDL as part of the *boolean condition*. Our methodology in using completion functions efficiently in VHDL is described next.

3.4.3 Single Entity Environment

This subsection explains our verification methodologies and recommendations in the application of completion functions in equivalence checking at the register transfer level.

In using the verification strategy of completion functions for the equivalence checking of pipelined implementations against its non-pipelined specification, the first step is to build the completion functions for each pipeline stage. We recommend the circuit designers to include the completion functions for each pipeline stage with their implementation because writing the completion functions requires an in-depth knowledge about the circuit operation and the designers already have that information. If the verification engineers are to build the completion functions, additional time resources would be spent on understanding the detailed circuit operation.

The completion functions are built for verification purpose only and should not be synthesized into actual hardware as part of the implementation. Only the associated completion functions and wire connections should be generated for its associated pipeline stage equivalence run because it saves time and memory resources. To avoid synthesizing additional hardware, we recommend the use of if-generate statements with boolean conditions formed by two suggested generic parameters: *stage* and *spec*.

stage is an integer used to specify which stage of a pipelined entity is under verification, and the associated completion functions are exclusively synthesized with if-generate statements. *spec* is a binary digit used to indicate whether the specification or the implementation is synthesized with if-generate statements during the equivalence checking of a pipeline stage as shown in Figure 3.10 to 3.12. In other words, *spec* specifies whether the associated completion functions are connected to the stage registers or to the outputs of a combinational block.

Referring back to the verification of our 2-stage example of Figure 3.9, setting *stage* to 2 and *spec* to '1' results in synthesizing the left hand side circuit $Specification$ exclusively as shown in Figure 3.10. Setting *stage* to 2 and *spec* to '0' results in synthesizing the right hand side circuit $Implementation$ exclusively as shown in Figure 3.10. Figure 3.15 shows the VHDL code (with completion functions) of the same 2-stage implementation that is used to described our verification methodology in Section 3.4.1. Note that the two generic parameters (*stage* and *spec*) are used with if-generate statements to control the hardware generation of the completion functions. Figure 3.16 illustrates which completion functions are generated and how they are connected depending on the generic parameters *stage* and *spec*. In the next section, we show the advantages of our VHDL methodology in a multiple entities design environment.

3.4.4 Multiple Entities Environment

For most of today's large digital systems, their design flow usually involve multiple designers creating multiple VHDL entities. To illustrate that our methodology supports multiple entities design environment, we created a contrived example as shown in Figure 3.17. Suppose that two designers A and B create separately their own corresponding 2-stage pipelined entities A and B. A third designer d creates 4-stage pipelined entity d, which is formed by both entities A and B, without knowing the detailed implementations of these sub-entities.

To build the completion functions for each pipeline stage of the circuit in Figure 3.17, the verification engineers are required to gain an in-depth understanding about the circuit operation of each component. However, it is intuitive to the circuit designers upon how to build the completion functions for each stage of their own entities.

In Figure 3.17, designers A and B can easily build the completion functions for each of their pipeline stage because they are fully aware about each block of

```
entity A is
 generic(stage:integer:=0;
         spec:std_logic:='0');
 port(input:in std_logic;
      output:out std_logic;
      clk:in std_logic);
end entity A;

architecture main of A is
 signal x1,x2:std_logic;
begin
 process begin
  wait until rising_edge(clk);
  x1<=input; x2<=A1(r1);
 end process;
 cfSpec:if spec='1' generate
  cf2Spec:if stage=2 generate
   output<=C2(x2);
  end generate cf2Spec;
  cf1Spec:if stage=1 generate
   output<=C1(X1);
  end generate cf1Spec;
 end generate cfSpec;
 cfImpl:if spec='0' generate
  cf2Impl:if stage=2 generate
   output<=A2(x2);
  end generate cf2Impl;
  cf1Impl:if stage=1 generate
   output<=C2(A1);
  end generate cf1Impl;
  cf0Impl:if stage=0 generate
   output<=C1(input);
  end generate cf0Impl;
 end generate cfImpl;
end architecture main;
```

Figure 3.15: Entity A: VHDL Code

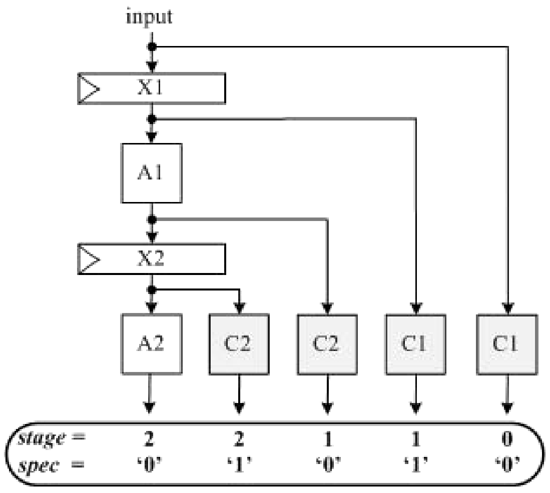


Figure 3.16: Contrived Entity A With Completion Functions

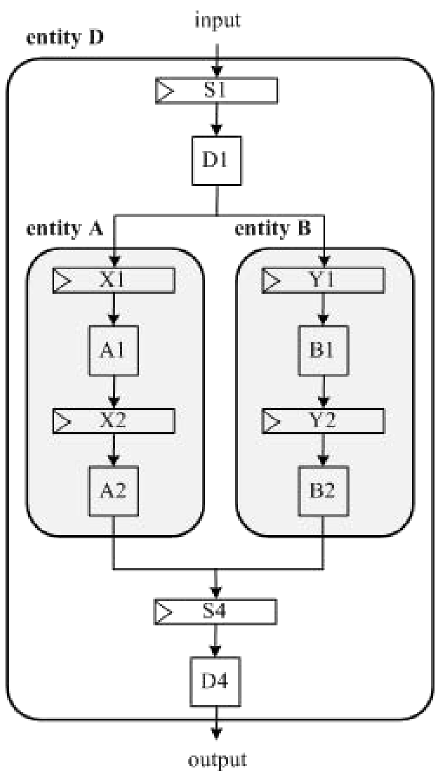


Figure 3.17: Contrived Entity D

their own entities A and B. However, the circuit designer of entity d does not know the detailed implementation of its sub-entities A and B. Therefore, our methodology suggests the use of if-generate statements with two VHDL generic parameters: *stage* and *spec*. For example, setting *stage* to 2 and *spec* to '0' of entity A would generate the path $X1 \rightarrow A1 \rightarrow X2 \rightarrow A2$ in Figure 3.16. With the addition of the two generic parameters to entities A and B, designer d can now build the completion functions of entity d simply by setting *stage* to 0 and *spec* to '0' of both sub-entities. The verification engineers can also use the completion functions of each entity by setting the two generic parameters accordingly.

Note that the two generic parameters (*stage* and *spec*) are not global parameters which can be passed from entity d to entity A or B. The two generic parameters, *stage* and *spec*, can be viewed as local variables for each entity. The reason is that the *stage* parameter of entity d is not the same as the *stage* parameter of entity A. As shown in Figure 3.17, the *stage* parameter of the 4-stage entity d are defined for values from 0 to 4 whereas the *stage* parameter of the 2-stage entity A are only defined for values from 0 to 2. Nonetheless, our methodology fits very well in the current design environment where there are multiple components and multiple designers. In the next section, we discuss the verification of the three pipelined implementations of KASUMI.

3.5 Verification of KASUMI

The purely combinational implementation of KASUMI, which was verified in Section 3.2, was optimized with pipelining and conceived three additional pipelined implementations of the cipher in Section 3.3. However, these additional implementations of KASUMI need to be verified for their correctness. In this section, we describe how the verification methodology of Section 3.4 were used to verify these additional implementations of the cipher.

Figure 3.18 to 3.21 each represents an equivalence check of the implementation *Impl* against the specification *Spec*. For each equivalence run, the equivalence checker returns true whenever the implementation is equivalent to the specification that describes the desired behaviour of the circuit. The verification of our 8-stage KASUMI began by verifying the last pipeline stage as shown in Figure 3.18. The left hand side is the 8-stage KASUMI implementation *Impl* with stage registers Ai. The right hand side is the 8-stage KASUMI specification *Spec* with the completion function of the eighth stage connected to the eighth stage registers $A8$. The equivalence checker compared the left hand side circuit *Impl* against the right hand side circuit *Spec* to verify the correctness of the last pipeline stage. In other words, it was verifying the equivalence of "8^{th} round implementation" against the "Completion function 8" and it was our first verification obligation.

The second verification obligation was to move up the pipeline and verify the correctness of the seventh stage as depicted in Figure 3.19. On the right hand side, we connected the completion function of the seventh stage to its associated stage registers $A7$ in the specification. Note that the completion function of the seventh stage is formed by the completion function of the last stage "Completion function 8" plus "Completion function 7", and it has the effect of completing the instructions in the stage registers $A7$. On the left hand side, we connected "Completion function 8" directly to the outputs of the seventh stage combinational circuitry "7^{th} round implementation". Basically, the second verification obligation compared "7^{th} round implementation" against "Completion function 7" as shown in Figure 3.19.

If the stage registers $A8$ were not removed from the left hand side *Impl* in Figure 3.19, the equivalence checker would not be able to compare the implementation against the specification due to the nature of equivalence checking. In addition, it

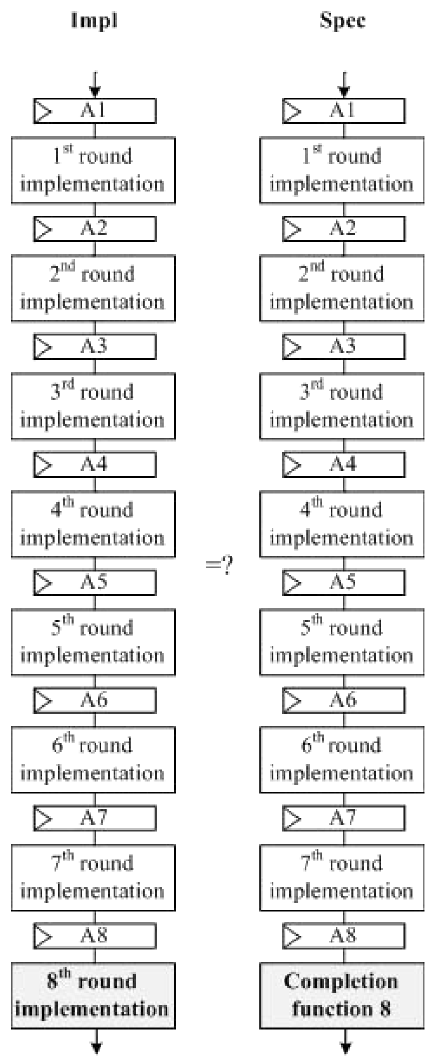


Figure 3.18: Stage 8 Obligation

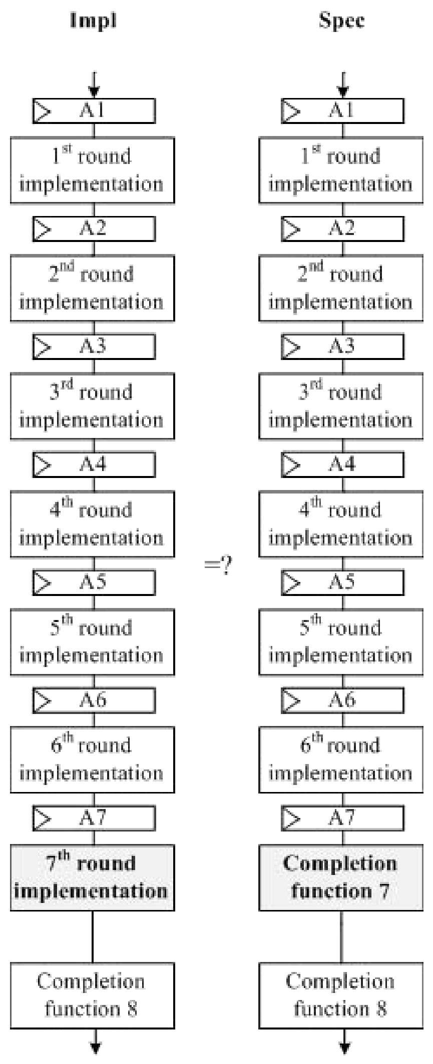


Figure 3.19: Stage 7 Obligation

is easier for the equivalence checker to solve when both the implementation and the specification have the same block "Completion function 8" as part of their circuits. By matching signal names and circuit structures on both sides (*Spec* and *Impl*), the equivalence checker applies the optimization of structural matching to recognize that "Completion function 8" is part of both *Spec* and *Impl.* Without structural matching, the equivalence checker cannot reduce its computational complexity and it would run into the state-space explosion problem. Therefore, structural matching is very important in our verification strategy.

Similarly, the remaining pipeline stages were verified in the same manner where the completion functions of each stage were connected to their corresponding stage registers *Ai* on the specification side (right hand side of figures). Also, the completion functions of the next stage were connected to the outputs of the stage combinational circuitry on the implementation side (left hand side of figures) for each stage verification obligation. Figure 3.20 shows the equivalence check for the verification of the first pipeline stage. Note that the completion function of the first stage is the composition of "Completion function 1" to "Completion function 8".

Up to this point of the verification, any design bugs would be detected and localized under the assumption that the completion functions were correctly built to describe the behaviour of the circuit. Our final verification obligation was used to verify this assumption by comparing the completion function of the first stage (left hand side) against the "verified" purely combinational implementation of KASUMI (right hand side) as shown in Figure 3.21. This final verification obligation is crucial as it would catch any bugs in writing the completion functions used for the stage-by-stage verification. Therefore, our verification would never lead to false positive results. The verification of our 8-stage KASUMI was completed with 9 verification obligations in three minutes, and no bugs were found.

Using the same verification tools and strategies, we verified the 16-stage and the 32-stage KASUMI implementations. The verification of our 16-stage KASUMI was completed with 17 verification obligations in five minutes. The verification of our 32-stage KASUMI was completed with 33 verification obligations in nine minutes. No bugs were found in both the 16-stage and the 32-stage KASUMI implementations.

In this chapter, we have covered the design and the verification of the non-pipelined specification of KASUMI (Section 3.1 and 3.2). In Section 3.3, we have introduced our three pipelined implementations of KASUMI. We explained our verification methodologies in using completion functions with equivalence checking

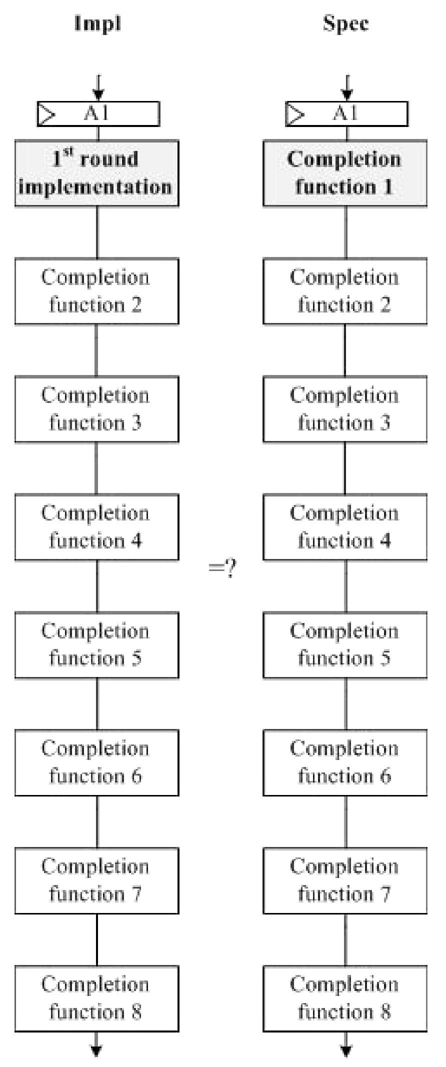


Figure 3.20: Stage 1 Obligation

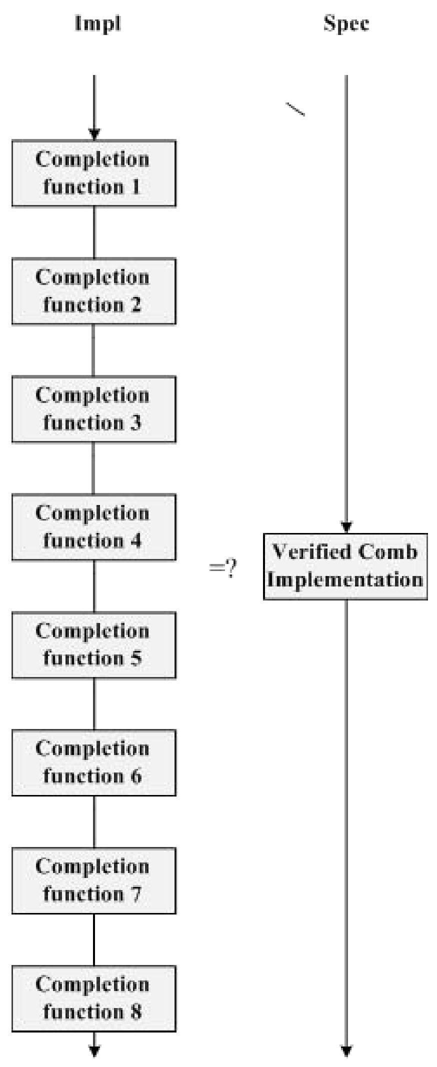


Figure 3.21: Final Obligation

in Section 3.4. In Section 3.5, we conclude the chapter with the verification results of KASUMI. Next chapter contains the design and the optimization of the Welch-Gong cipher.

Chapter 4

WG: Design and Optimizations

The purpose of this chapter is to extend our verification methodology used in the verification of KASUMI to a more sophisticated case study, the Welch-Gong cipher, and it is organized as follows. Section 4.1 describes the design process of the non-pipelined implementation of WG, and Section 4.2 explains how RTL simulation was used to verify the correctness of this first implementation. In Section 4.3, we illustrate how the optimizations of pipelining and hardware re-use were applied to create the optimized implementation of WG. Section 4.4 provides a comparison of our optimized WG implementation to other state-of-the-art stream ciphers.

4.1 The First Design

This section is used to describe our design of the non-pipelined implementation of WG. We first introduce the overall structure of our implementation then we describe each of the major components separately.

The circuit structure of WG and its circuit operation were described in Section 2.6. We implemented the circuit shown in Figure 2.15, and we added a finite state machine to generate control signals used to change the configuration of WG depending on the phase of the cipher (loading of the registers, initialization phase, keystream generation).

Figure 4.1 illustrates our 29-bit non-pipelined implementation of WG in which the four main components are: the 11-stage linear feedback shift register, the WG core, the trace function and the finite state machine. Note that we have split the WG transformation into WG core and trace function because there is a feedback from the output of the WG core block to the input of the LFSR. In the remaining

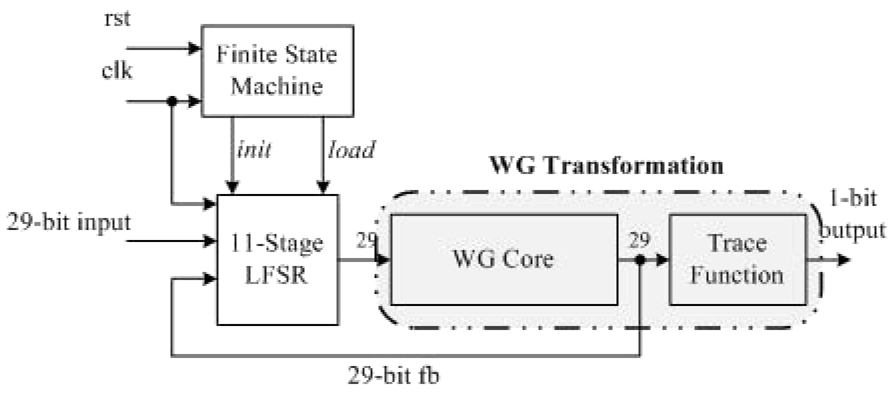


Figure 4.1: WG Implementation Block Diagram

subsections, we first describe how we designed the datapath (WG core and trace function) of the WG cipher. Then, we introduce the control circuitry (LFSR and finite state machine) used in our implementation.

4.1.1 WG Core and Trace Function

In this section, we focus on the datapath of our implementation of WG. We first introduce the WG core block then we briefly describe our implementation of the trace function.

We implemented the WG core block (i.e. WG transformation without trace function) shown in Figure 2.15 of Section 2.6. Note that the WG core block is formed by a combination of simple building blocks such as inverters, bitwise shifting (re-wiring) and XOR gates. The most sophisticated components are the normal basis multiplier and the $(\cdot)^{2^{10}-1}$ block which are described next.

For all the normal basis multipliers in our implementation, we chose to use the optimal normal basis multiplier implemented by Sunar [17]. A detailed description of the implementation of the optimal normal basis multiplier is not necessary to understand our work, and the reader can obtain that information in [17].

We implemented the $(\cdot)^{2^{10}-1}$ operation with a combination of normal basis multiplications and bitwise shifting as shown in Figure 4.2. Recall that shifting the bits of an element x cyclically to right by i position computes x^{2^i} if x is expressed in normal basis representation. Our implementation of this block becomes clear once we rewrite the exponent (i.e. $2^{10}-1$) as follows. Suppose that u is an element

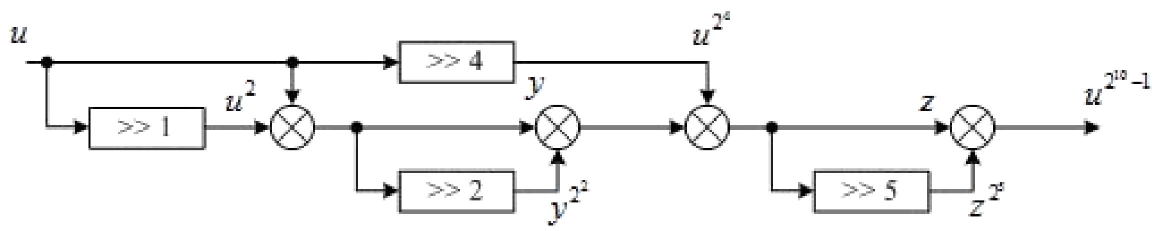


Figure 4.2: Implementation of $(\cdot)^{2^{10}-1}$

represented in the normal basis form, then:

$$\begin{aligned} u^{2^{10}-1} &= u^{2^0+2^1+2^2+2^3+2^4+2^5+2^6+2^7+2^8+2^9} \\ &= u^{(2^0+2^1+2^2+2^3+2^4)+2^5\times(2^0+2^1+2^2+2^3+2^4)} \end{aligned}$$

let $z = u^{(2^0+2^1+2^2+2^3+2^4)}$, then:

$$u^{2^{10}-1} = z \times z^{2^5}$$

where z can be rewritten as:

$$\begin{aligned} z &= u^{(2^0+2^1+2^2+2^3+2^4)} \\ &= u^{(2^0+2^1)+2^2\times(2^0+2^1)+2^4} \end{aligned}$$

let $y = u^{(2^0+2^1)}$, then:

$$z = y \times y^{2^2} \times u^{2^4}$$

Therefore, we can compute $u^{2^{10}-1}$ with the following three steps:

$$\begin{aligned} y &= u^{2^0} \times u^{2^1} \\ z &= y \times y^{2^2} \times u^{2^4} \\ u^{2^{10}-1} &= z \times z^{2^5} \end{aligned}$$

We completed the design of the datapath (i.e. WG transformation) by connecting the 29-bit output of the WG core block to the 29-bit input of the trace function as shown in Figure 4.1. For the implementation of the trace function, we XORed all 29 input bits together because the trace function of an element is simply the addition of all the bits of that element over $\mathbb{F}_2$. This section has described how we designed the datapath of our 29-bit non-pipelined implementation of WG. In the next section, we explain how we designed the control circuitry of our WG implementation.

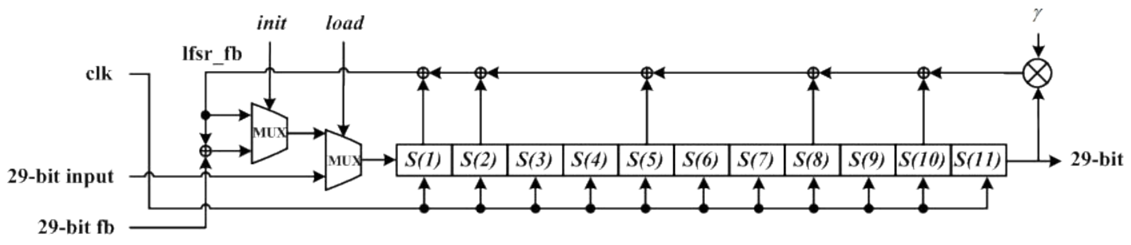


Figure 4.3: Linear Feedback Shift Register Implementation

load	*init*	WG Phase	Input of register *S(1)* becomes...
1	0	loading of the registers	29-bit input
1	1	loading of the registers	29-bit input
0	1	initialization	*lfsr_fb* $\oplus$ 29-bit *fb*
0	0	keystream generation	*lfsr_fb*

Table 4.1: Signal Selected Based on *load* and *init*

4.1.2 Linear Feedback Shift Register

In this section, we describe how we designed the control circuitry of our non-pipelined implementation of WG. We first introduce the design of the LFSR and how control signals are used to change its circuit configuration. Then, we explain how we implemented the finite state machine that is used to generate the control signals.

We implemented the LFSR in Figure 2.13 of Section 2.6, and we added two sets of multiplexers at the input of the registers *S(1)* as shown in Figure 4.3. The two sets of multiplexers are added to change the input signal of the LFSR registers *S(1)* depending on the two control signals, *init* and *load*, as shown in Figure 4.3. In Figure 4.3, note that γ is not an input to the LFSR but it is a constant number which is multiplied to the output of the LFSR $S(11)$ as mentioned in Section 2.6. The five inputs to the LFSR are: the clock signal *clk*, the 29-bit input used for the loading of the registers, the 29-bit feedback *fb* used for the initialization of WG, the two control signals *init* and *load* used for the control of the two sets of multiplexers. Table 4.1 illustrates the signal at the input of the LFSR registers *S(1)* based on the values of the two control signals, *init* and *load*, and the respective phase of the WG cipher.

For the loading of the LFSR registers *S(1)* to $S(11)$, we chose to implement serial loading instead of parallel loading because it requires less implementation area without affecting the performance of the WG cipher. In our 29-bit implementation

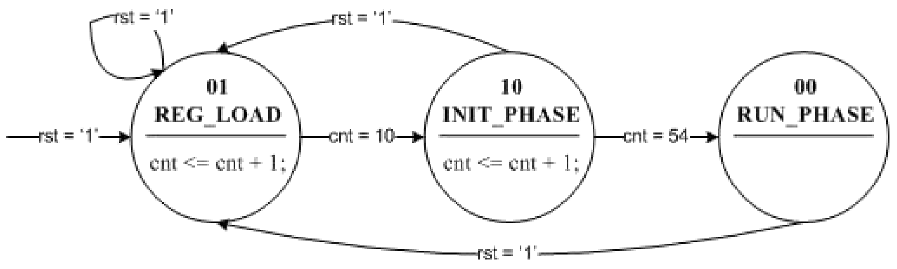


Figure 4.4: Finite State Machine Implementation

of WG, serial loading introduces 29 multiplexers at the input of the first stage LFSR registers *S(1)* and it loads the desired values into the registers *S(1)* every clock cycle until the desired values have propagated to the registers $S(11)$. Parallel loading would insert 319 (29 bits × 11 LFSR stages) multiplexers, one per register, into the LFSR and it would load all the registers (*S(1)* to $S(11)$) in one clock cycle. Although serial loading requires 11 clock cycles to load all the registers, it is an acceptable trade off because the loading of the LFSR registers only occur once at the beginning of the WG cipher operation.

4.1.3 Finite State Machine

We completed the design of the control circuitry by implementing a finite state machine used to generate the two control signals, *init* and *load*. Figure 4.4 shows the state transition diagram of our 3-state finite state machine. In our design of the finite state machine, we added the counter *cnt* to keep track of the number of clock cycles elapsed and we encoded the three states (*REG_LOAD*, *INIT_PHASE*, *RUN_PHASE*) with a 2-bit vector in which the left bit corresponds to the *init* signal and the right bit is the *load* signal.

The operation of the finite state machine begins by setting the reset signal rst to '1' as shown in Figure 4.4, then the finite state machine is initialized and enters its first state *REG_LOAD* and the counter *cnt* resets to 0. From the state encoding mentioned above, the state *REG_LOAD* is equivalent to setting *init* to '0' and *load* to '1'. Thus, the input of the LFSR registers *S(1)* becomes the 29-bit input according to Table 4.1. During the state *REG_LOAD*, the counter *cnt* is incremented every clock cycle by a value of one. When the counter *cnt* hits a value of 10, the finite state machine makes a transition to its next state *INIT_PHASE* because the LFSR has loaded all of its registers from clock cycle 0 to 10 (a total of 11 clock cycles).

From our state encoding, the state *INIT_PHASE* corresponds to assigning *init* to '1' and *load* to '0'. Hence, the input of the LFSR registers *S(1)* becomes the LFSR feedback polynomial *lfsr_fb* added with the 29-bit feedback *fb* from the WG core block as shown in Table 4.1. During the state *INIT_PHASE*, the counter *cnt* continues to count the number of clock cycles elapsed. Once the counter *cnt* reaches a value of 54, the finite state machine transfers to its next state *RUN_PHASE* because the initialization phase has been completed from clock cycle 11 to 54 (a total of 44 clock cycles).

For our finite state machine, the state *RUN_PHASE* implies that *init* is '0' and *load* is '0'. Thus, the input of the LFSR registers *S(1)* becomes the LFSR feedback polynomial *lfsr_fb* as listed in Table 4.1. During the state *RUN_PHASE*, the counter *cnt* becomes idles and the finite state machine would remain in this state unless the reset signal *rst* is set to '1'. Note that keystream generation occurs in this state, and the two other states (*REG_LOAD*, *INIT_PHASE*) are used to initialize the WG cipher. Throughout all three states of our finite state machine, setting '1' to the reset signal *rst* leads the state machine back to the state *REG_LOAD* and it resets the counter *cnt* to 0.

This first implementation was synthesized using Mentor Graphics Precision RTL synthesis tool. On an Altera Stratix II series field programmable gate arrays (FPGA) device EP2S15F484C, our combinational implementation of WG has an area of 7412 logic elements and a performance of 31 MHz when registers are inserted at the inputs and at the outputs. This section has explained how we designed both the datapath and the control circuitry of our 29-bit non-pipelined implementation of the Welch-Gong cipher. In the subsection about the WG core block and the trace function, we have covered the design of the $(\cdot)^{2^{10}-1}$ operation. In the subsection about the linear feedback shift register and the finite state machine, we have showed how the input to the LFSR is controlled by the signals generated by the finite state machine. In the next section, we have to verify the correctness of our non-pipelined implementation of WG so that it can be used as a specification in equivalence checking.

4.2 Formulating The Specification

Similar to the case study of KASUMI, we have to verify the correctness of our non-pipelined implementation of WG before it can be used as the specification for the verification of the optimized implementation of WG.

80 bit key and 32 bit IV

key = 80000000000000000000
IV = 01234567
keystream = 604DDB771960DFF05353D6E22E3E2ECD

Figure 4.5: Fragments of Test Vectors from WG Paper

For the verification of the combinational implementation of WG, we have used conventional RTL simulation because a set of test vectors are given in the WG paper [14]. These test vectors only defines the correct output keystream given a specific input key and initialization vector IV as shown in Figure 4.5. Note that all vectors are in hexadecimal values and that only a subset of test vectors from the WG paper are shown in Figure 4.5. Hence, our non-pipelined implementation of WG is defined as correct if and only if it produces an expected keystream given a key and an initialization vector. However, there are still possibilities for a subtle bug to be hidden in our implementation because our RTL simulation does not cover all possible combinations of input key and initialization vector to the WG cipher.

The test vector of Figure 4.5 is only useful in the verification of our implementation of WG as a whole system, and it does not provide any information about the internal components (e.g. linear feedback shift register) used to form the whole WG cipher. Similar to any real hardware design, our implementation of WG has various implementation-specific details that need to be verified prior to the verification of the WG implementation as a system. As an example of implementation-specific details, the design of our linear feedback shift could have been loading the registers in parallel instead of loading the registers in serial. Thus, we have defined the correctness criteria for the finite state machine as well as for the linear feedback shift register, and we have verified them in this section.

The verification of our implementation of WG is divided into four parts, one per component, and it proceeds in a bottom-up manner because it can localize an implementation bug to a component instead to the whole WG implementation. In this section, we first verify the finite state machine, the linear feedback shift register and the trace function individually. Our final RTL simulation combines the verification of the WG core with the verification of the whole WG implementation.

4.2.1 Finite State Machine

For all of the RTL simulation runs, we have used Mentor Graphics ModelSim. Figure 4.6 illustrates the simulation results of the finite state machine in which rst is the reset signal and cnt is the counter for the number of clock cycles elapsed. The finite state machine is defined as correct when it satisfies all the properties mentioned below. These properties capture the behavior of the finite state machine as shown in Figure 4.4, which includes the sequence of the states, the number of clock cycles elapsed in each state, and the transitions to the next state.

1. The finite state machine can only be initialized by setting the reset signal rst to '1', then the finite state machine would enter the *REG_LOAD* state ("01") where $init = $ '0' and $load = $ '1'.

2. When the finite state machine enters the *REG_LOAD* state, if the reset signal *rst* remains at '0', then the finite state machine remains in the *REG_LOAD* state for exactly 11 clock cycles then goes into the *INIT_PHASE* state ("10") where $init = $ '1' and $load = $ '0'.

3. When the finite state machine enters the *INIT_PHASE* state, if the reset signal *rst* remains at '0', then the finite state machine remains in the *INIT_PHASE* state for exactly 44 clock cycles then goes into the *RUN_PHASE* state ("00") where $init = $ '0' and $load = $ '0'.

4. When the finite state machine enters the *RUN_PHASE* state, if the reset signal *rst* remains at '0', then the finite state machine remains in the *RUN_PHASE*.

4.2.2 Linear Feedback Shift Register

Figure 4.7 shows the simulation results of the 1-bit linear feedback shift register in which d is the input of the LFSR, *fb* is the feedback signal from WG core, *lfsr_fb* is the LFSR feedback polynomial and *first_bit* is the input of the register *S(1)*. By replicating the 1-bit linear feedback shift register 29 times, we can obtain the 29-bit linear feedback shift register shown in Figure 4.3. The 1-bit linear feedback shift register is defined as correct when it fulfills properties mentioned below. These properties capture the description of the linear feedback shift register as shown in Figure 4.3, which includes the bit shifting of the registers, the LFSR feedback polynomial connection, and the signal selection by the mutliplexers.

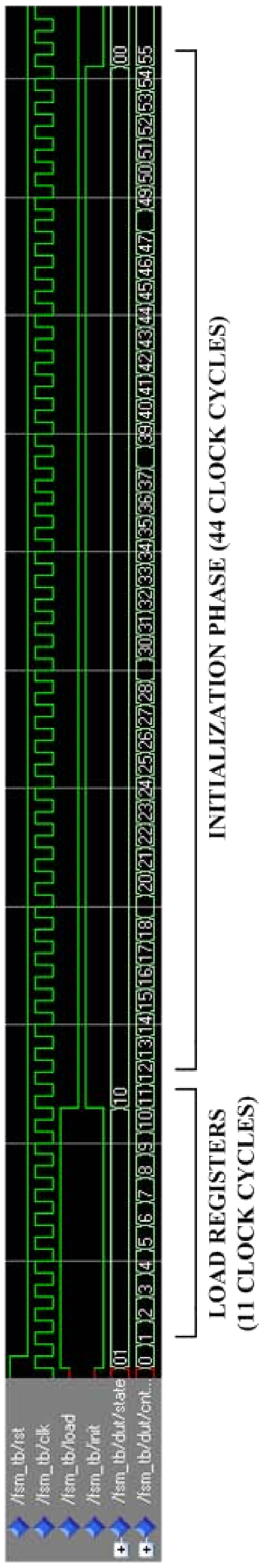


Figure 4.6: Finite State Machine Simulation Results

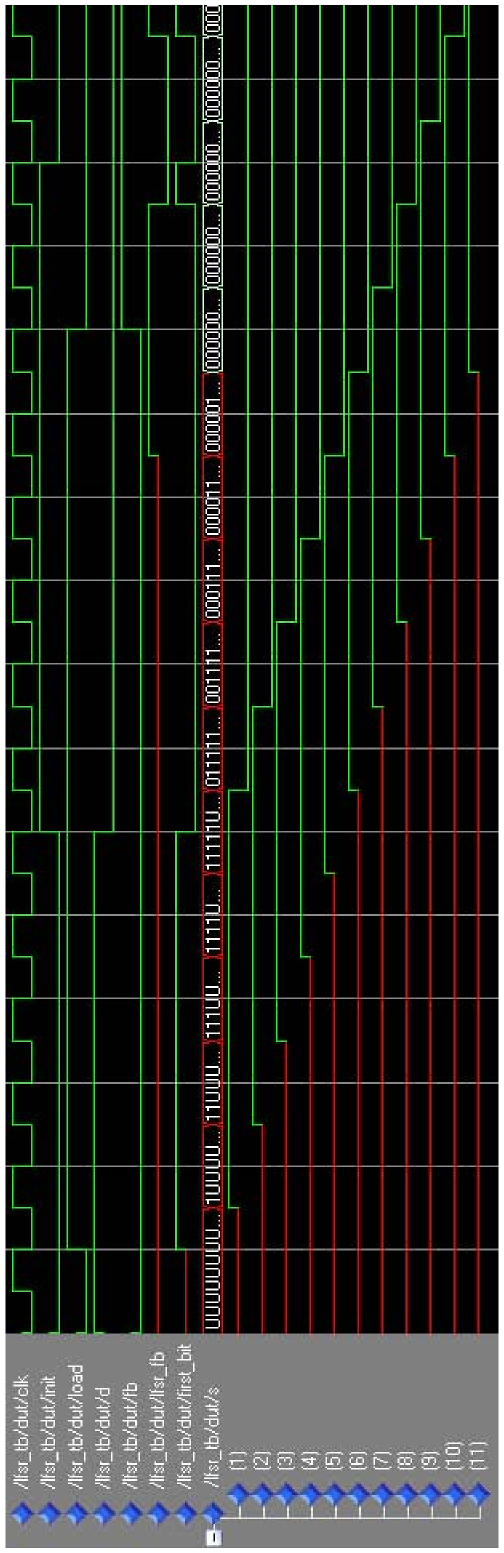


Figure 4.7: Linear Feedback Shift Register Simulation Results

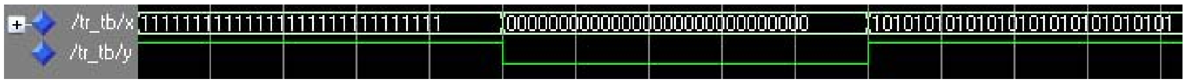


Figure 4.8: Trace Function Simulation Results

1. At each clock cycle, the data of the register *s(i)* shifts to the register *s(i+1)* for $i = 1 \dots 10$ and *s(1)* retrieves its value from the signal *first_bit*.
2. At all time, the LFSR feedback polynomial *lfsr_fb* has to match equation (2.12) of Section 2.6.
3. When *load=‘1*, the *first_bit* input to the register *s(1)* becomes the *d* input of the linear feedback shift register.
4. When *load=‘0* and *init=‘1*, the *first_bit* input to the register *s(1)* becomes the XOR of the LFSR feedback polynomial (*lfsr_fb*) and the feedback signal from WG core (*fb*).
5. When *load=‘0* and *init=‘0*, the *first_bit* input to the register *s(1)* becomes the LFSR feedback polynomial *lfsr_fb*.

4.2.3 Trace Function and WG Core

Figure 4.8 depicts the simulation results of the trace function in which x is the 29-bit input and y is the 1-bit output. The trace function is define as correct if and only if the output y is the addition (XOR) of all 29 bits of input x.

The RTL simulation results from Figure 4.6 to 4.8 have confirmed the correctness for three components of our WG implementation except the WG core block. Our fourth RTL simulation is used to verify the correctness of the WG implementation (as a system) as well as to verify the correctness of the WG core block. Figure 4.9 shows the simulation results of the non-pipelined implementation of WG in which *key* is the 80-bit input key, *init_v* is the 32-bit initialization vector, *spec_k_stream* is the 1-bit output of WG and *spec_keystream* is the output keystream of WG. Note that our simulation results match with the test vectors of Figure 4.5.

In this section, we verified the correctness of our non-pipelined implementation of WG by using multiple RTL simulation runs. Therefore, our non-pipelined im-

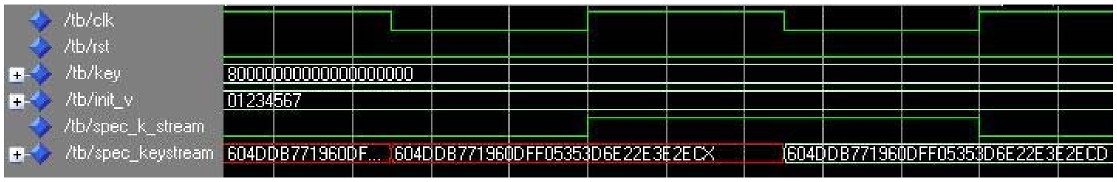


Figure 4.9: WG Test Vectors Results

plementation can be used as the specification in the verification of the optimized implementation of WG that is described in the next section.

4.3 The Optimized Implementation

To explore the completion functions in the verification of a pipelined circuit with sophisticated optimizations, we have optimized our non-pipelined implementation of WG with both optimization techniques of pipelining and hardware re-use. In this section, the first subsection shows how we have applied pipelining onto the datapath of our implementation of WG. The second subsection introduces the concept of hardware re-use and how to apply it to our implementation of WG. In the third subsection, we explain how the two optimizations (i.e. pipelining and re-use) require some modifications to be made in the control circuitry (i.e. linear feedback shift register and finite state machine) so that the whole optimized implementation of WG can operate correctly.

4.3.1 Pipelining

The optimization process began with the pipelining of the datapath (i.e. WG core and trace function) of our WG implementation. In the pipelining of a design, it is optimal for each pipeline stage to have an equal amount of delay because the operating speed would not be limited only by the slowest stage. Recall the the datapath of our WG implementation is mainly dominated by normal basis multipliers (beside the inverters and the XOR gates) as shown in Figure 2.14 and 4.2. Therefore, we have divided the datapath of our WG implementation such that each pipeline stage has approximately a delay of one normal basis multiplier.

In combination to the pipelining strategy mentioned above, we have taken a modular bottom-up approach in which we first optimize the smaller block of $(\cdot)^{2^{10}-1}$ then we proceed to the larger block of WG core. Figure 4.10 shows our 4-stage

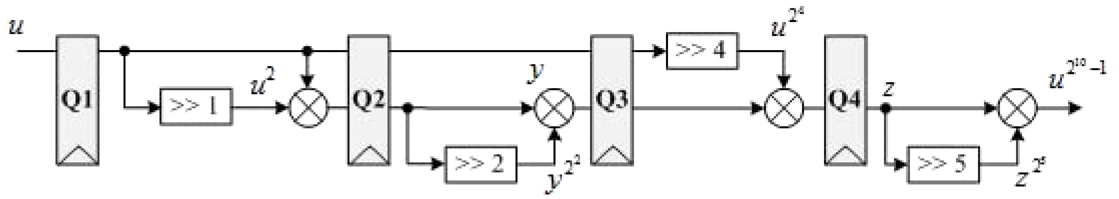


Figure 4.10: 4-Stage Implementation of $(\cdot)^{2^{10}-1}$

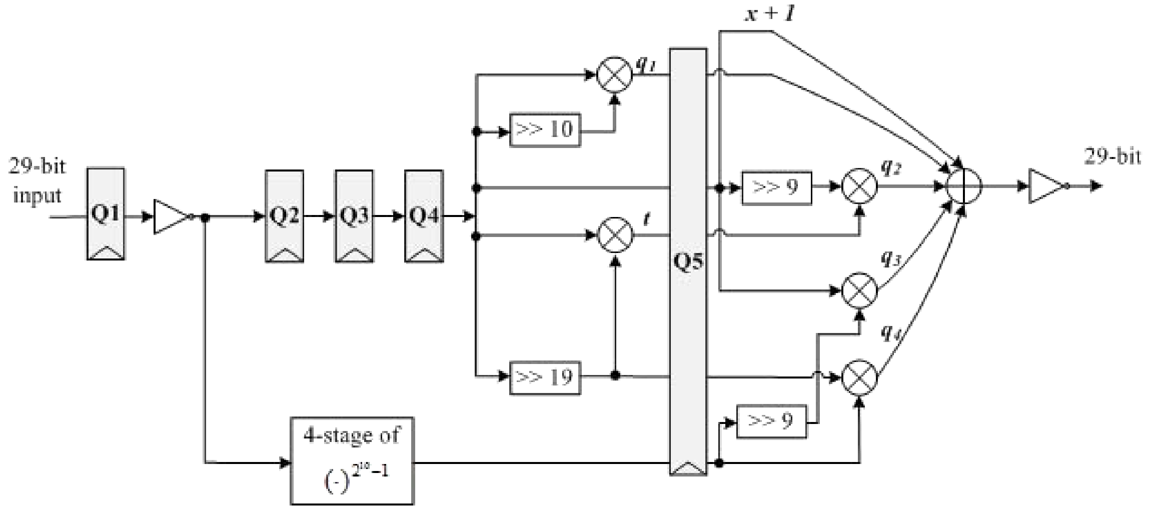


Figure 4.11: 5-Stage Implementation of WG Core

implementation of $(\cdot)^{2^{10}-1}$ in which we have inserted the registers (*Q1*, *Q2*, *Q3*, *Q4*) at the inputs of the normal basis multipliers.

By moving up one level in the hierarchy, Figure 4.11 illustrates our 5-stage implementation of WG Core. In Figure 4.11, we have extracted the registers *Q1* from the "4-stage of $(\cdot)^{2^{10}-1}$" block and relocated (retimed) them to the input of the inverter because it minimizes the number of registers used in our implementation of WG while keeping the amount of delay per pipeline stage to approximately a delay of one normal basis multiplier. If we did not relocate the registers *Q1*, then we would need to create an additional stage solely for the inverter at the input of the WG core block and it would increase the number of registers used in our implementation of WG as well as the latency of the WG core block.

The registers (*Q2*, *Q3*, *Q4*) of Figure 4.11 were inserted so that the data packets flowing in the WG core block are synchronized with the data packets flowing in the "4-stage of $(\cdot)^{2^{10}-1}$" block. In the fourth pipeline stage of Figure 4.11, the two multipliers could have been moved to any earlier stage because their inputs do not depend on the outputs of the "4-stage $(\cdot)^{2^{10}-1}$" block. However, moving these two multipliers to any earlier stage would increase the number of registers used in our implementation of WG because the outputs of these two multipliers have to be carried to the fifth stage where they are used to compute the output of the WG core block.

The registers *Q5* were inserted to build the fifth pipeline stage in which we have lumped the normal basis multiplier with the simple operations of exclusive-or and bitwise inversion. Similar to the two multipliers in the fourth pipeline stage, the multiplier used to produce the signal q_2 in the fifth stage is located in the latest possible stage so that the number of registers used in our WG implementation is minimized. Note that the multiplier (in fourth stage) used to generate the signal q_1 could have been moved to the fifth stage to save some registers, but we have kept this multiplier in the fourth stage because it would allow the optimization technique of hardware re-use to be applied onto our pipelined implementation of WG in later subsection.

In this subsection, we explained how we used pipelining to optimize the WG core block into a 5-stage pipeline. An immediate effect of changing the latency of the WG core block to five clock cycles is that the linear feedback shift register would sample the feedback signal from the WG core every six clock cycles (instead of every clock cycle when the WG core block is purely combinational) during the initialization phase of the WG cipher. Note that Figure 4.10 and 4.11 are not the finalized pipelines used in our optimized implementation of WG, and we will explain the modifications required in the control circuity (i.e. linear feedback shift register and finite state machine) to remedy the WG core latency change after we finalize the optimization of the datapath with hardware re-use in the next subsection.

4.3.2 Hardware Re-Use

We proceed to the second phase of our optimization process with the optimization technique known as hardware re-use. This optimization technique can reduce the implementation area by re-using a component in multiple pipeline stages instead of instantiating multiple instances of the same component in various pipeline stages to generate the same outputs. However, hardware re-use can only be applied to circuits that contain multiple instances of a component and it has the drawback of decreasing the throughput of the circuit if the clock speed cannot be increased further. In this subsection, we first clarify our description of hardware re-use with a simple contrived example then we proceed to the application of hardware re-use onto our pipelined implementation of WG from the previous subsection.

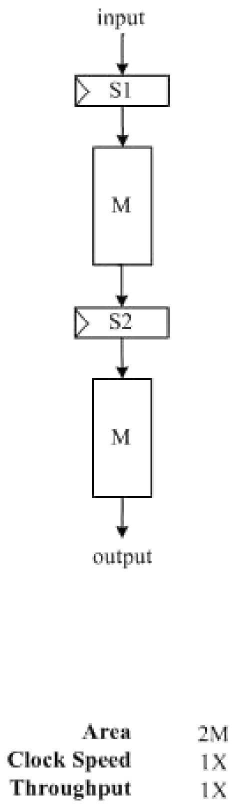


Figure 4.12: Initial Pipeline

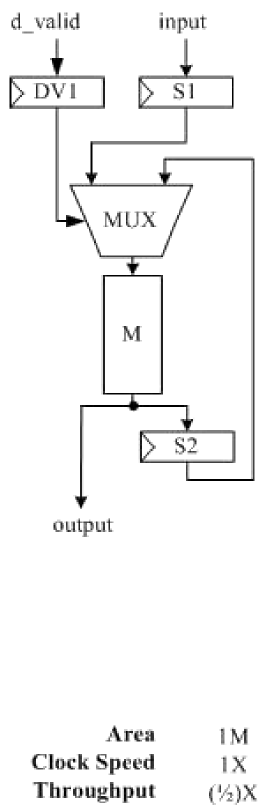


Figure 4.13: Pipeline with Re-Use

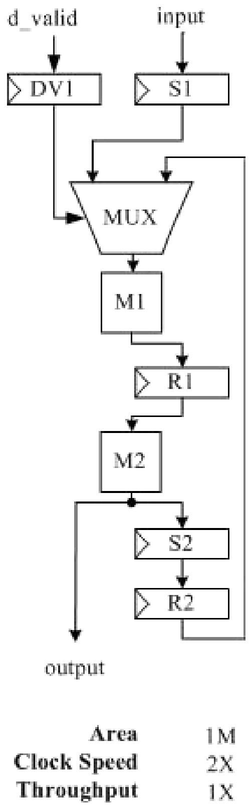


Figure 4.14: Pipeline with Re-Use and Superpipelining

2-Stage Pipeline Example

Figure 4.12 depicts the initial 2-stage pipeline in which both stages have their own combinational block M and the stages are delimited by the stage registers $S1$ and $S2$. This initial pipeline consumes an area of two blocks M and it has a throughput of $1X$ at a clock speed of $1X$. Since the 2-stage pipeline of Figure 4.12 has two instances of the component M, the optimization of hardware re-use can be applied to reduce the implementation area to one block M as shown in Figure 4.13.

In Figure 4.13, one block M is used in both the first and the second pipeline stage to compute the same output as the initial pipeline. When a data packet first enters the pipeline, the control signal *d_valid* is asserted so that the multiplexer feeds the stage register $S1$ (data packet) into the block M as shown in Figure 4.13. When a data packet reaches the second pipeline stage, the control signal must be de-asserted so that the multiplexer feeds the stage register $S2$ (data packet) back into the block M again to compute the output and no new data packet enters the pipeline. Since each data packet occupies the block M for two consecutive clock

clk cycle	0	1	2	3	4	5
input	**A**	$\emptyset$	B	$\emptyset$	C	$\emptyset$
d_valid	1	0	1	0	1	0
S1		**A**	$\emptyset$	B	$\emptyset$	C
DV1		1	0	1	0	1
R1			**A**	$\emptyset$	B	**A**
S2				**A**	$\emptyset$	B
R2					**A**	$\emptyset$
output						**A**

Figure 4.15: Timing Diagram of Pipeline with Re-Use and Superpipelining

cycles, the throughput has decreased from one output per clock cycle to one output per every two clock cycles at the same clock speed of $1X$. Thus, hardware re-use has reduced the implementation area from two to one block M and decreased the throughput from $1X$ to $(1/2)X$ at a clock speed of $1X$. It is possible to restore the throughput of the circuit back to $1X$ if the re-used block M can be further pipelined (i.e. superpipelining) as depicted in Figure 4.14.

In Figure 4.14, we have inserted the stage registers $R1$ to split the block M into two pipeline stages ($M1$, $M2$) in which both stage has a delay of $(1/2)M$ so that the clock speed is doubled to $2X$. Although this pipeline with re-use samples an input per every two clock cycles to avoid contention of the signals at the input of the block M, it can still achieve a throughput of $1X$ because the clock speed has been increased to $2X$ instead of the original clock speed of $1X$. Figure 4.15 shows the timing diagram of the pipeline with re-use and superpipelining in which an input is fed into the pipeline once per every two clock cycles. Since there is a data packet in the first pipeline stage ($S1$) whenever there is a data packet in the third pipeline stage ($S2$), as depicted in clock cycle 3 of the timing diagram in Figure 4.15, the registers $R2$ (instead of $S2$) are added and fed back into the multiplexer to avoid the contention of these two data packets (data packet B in $S1$ and data packet A $S2$) at the input of the block M. In comparison to the initial pipeline of Figure 4.12, the pipeline with re-use and superpipelining has reduced the implementation area from two to one block M while keeping the throughput at $1X$ by increasing the clock speed from $1X$ to $2X$. For the second phase of the optimization of our pipelined WG implementation, we have applied hardware re-use and superpipelining as described next.

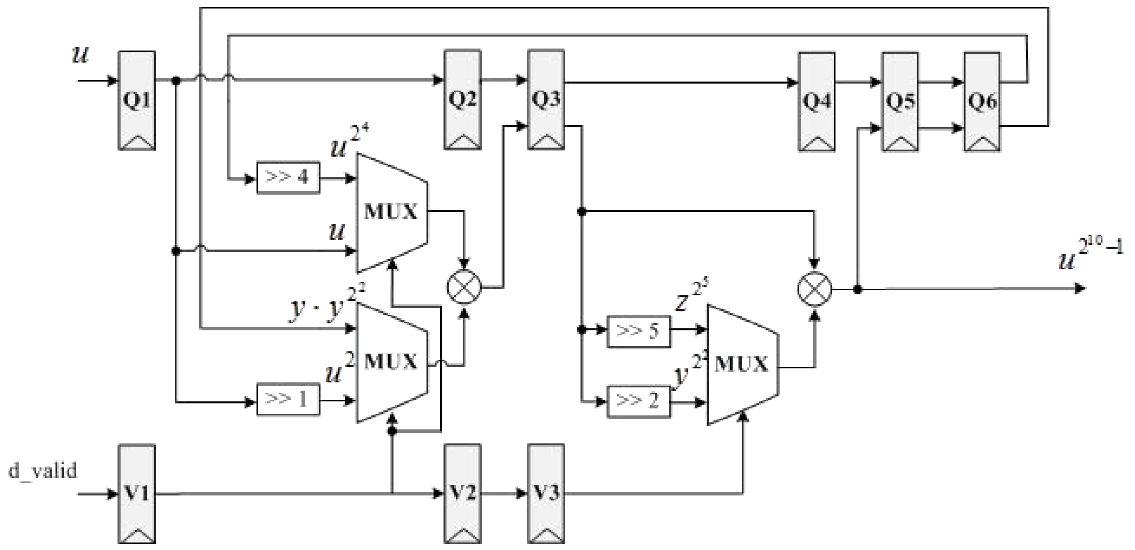


Figure 4.16: 9-stage Implementation of $(\cdot)^{2^{10}-1}$ with Re-Use

Pipelined WG with Re-Use

The optimization goal of applying hardware re-use to our pipelined WG implementation is to decrease the implementation area while keeping the same throughput in generating the WG keystream. We began our second optimization by superpipelining all normal basis multipliers into two balanced (i.e. similar delay) pipeline stages so that the clock speed of our WG implementation can be roughly doubled to maintain the throughput after the application of hardware re-use. The remaining of this section first shows how hardware re-use has been applied to the $(\cdot)^{2^{10}-1}$ block then it explains how the WG core has been optimized with hardware re-use.

Figure 4.16 illustrates the circuit structure of the $(\cdot)^{2^{10}-1}$ block after hardware re-use has been used to reduce its area from four to two normal basis multipliers. Since we have inserted one stage register within each multiplier (not shown in Figure 4.16) to superpipeline it into two balanced stages, we have added the stage registers $Q2$ and $Q4$ for the synchronization of the data packets outside and inside of the two multipliers. This implementation of the $(\cdot)^{2^{10}-1}$ has a latency of nine clock cycles because the output $u^{2^{10}-1}$ is computed after a data packet makes its second passage through the second multiplier located between the stage registers $Q3$ and $Q5$. The interactions between the data valid registers ($V1$, $V2$, $V3$) and the two multiplexers can be explained with the timing diagram in Figure 4.17.

In Figure 4.17, the data packet A enters the pipeline at clock cycle 0 and reaches the output at clock cycle 9 while other data packets (B, C, D, E) are fed into the pipeline with a *d_valid* of ‘1’ every other clock cycle. A stage-by-stage description

clk cycle	0	1	2	3	4	5	6	7	8	9
u	**A**	$\emptyset$	B	$\emptyset$	C	$\emptyset$	D	$\emptyset$	E	$\emptyset$
d_valid	1	0	1	0	1	0	1	0	1	0
Q1		**A**	$\emptyset$	B	$\emptyset$	C	$\emptyset$	D	$\emptyset$	E
V1		1	0	1	0	1	0	1	0	1
Q2			**A**	$\emptyset$	B	$\emptyset$	C	**A**	D	B
V2			1	0	1	0	1	0	1	0
Q3				**A**	$\emptyset$	B	$\emptyset$	C	**A**	D
V3				1	0	1	0	1	0	1
Q4					**A**	$\emptyset$	B	$\emptyset$	C	**A**
Q5						**A**	$\emptyset$	B	$\emptyset$	C
Q6							**A**	$\emptyset$	B	$\emptyset$
$u^{2^{10}-1}$										**A**

Figure 4.17: Timing Diagram of 9-stage Implementation of $(\cdot)^{2^{10}-1}$ with Re-Use

of our pipelined implementation of the $(\cdot)^{2^{10}-1}$ block with re-use is provided here because it helps the reader to understand the verification of this block in Chapter 5. Throughout our stage-by-stage description, we use the data packet A of Figure 4.17 as reference and it begins as follow.

- **Stages 1 and 2:** in clock cycle 1 (i.e. stage 1), the data valid register $V1$ is '1' so that the multiplexer selects the sigals u and u^2 to be fed into the multiplier in Figure 4.16. In clock cycle 2 (i.e. stage 2), the computation of $y = u \times u^2$ is completed and matches with the first multiplication in Figure 4.10.

- **Stage 3 and 4:** in clock cycle 3 (i.e. stage 3), the data valid register $V3$ is '1' so that the multiplexer selects the signal y^{2^2} to be multiplied with the signal y (output of register $Q3$) in Figure 4.16. In clock cycle 4 (i.e. stage 4), the computation of $y \times y^{2^2}$ is completed and matches with the second multiplication in Figure 4.10.

- **Stage 5:** in clock cycle 5, the data packet A is temporarily stored in the stage registers $Q5$ because the input of the first multiplier (between the registers $Q1$ and $Q3$) is being occupied with the data packet C (stored in $Q1$)

- **Stages 6 and 7:** in clock cycle 6 (i.e. stage 6), the data valid register $V1$ is '0' so that the multiplexer selects the sigals u^{2^4} and $y \times y^{2^2}$ to be fed into the multiplier in Figure 4.16. In clock cycle 7 (i.e. stage 7), the computation of $z = u^{2^4} \times y \times y^{2^2}$ is completed and matches with the third multiplication in Figure 4.10.

- **Stages 8 and 9:** in clock cycle 8 (i.e. stage 8), the data valid register $V3$ is '0' so that the multiplexer selects the signal z^{2^5} to be multiplied with the signal z (output of register $Q3$) in Figure 4.16. In clock cycle 9 (i.e. stage 9), the computation of $u^{2^{10}-1} = z \times z^{2^5}$ is completed and matches with the fourth multiplication in Figure 4.10.

By moving up one level in the hierarchy of the datapath, Figure 4.18 shows the circuit structure of the WG core after hardware re-use has condensed the original five multipliers down to three multipliers plus multipliers buried in the $(\cdot)^{2^{10}-1}$ block. Similar to the original pipelined WG core block, we have extracted the registers ($Q1$, $V1$) from the "9-stage of $(\cdot)^{2^{10}-1}$" block and retimed them to the input of the inverter. The stage registers $Q2$ to $Q9$ are added for the synchronization of the data packets outside and inside of the "9-stage of $(\cdot)^{2^{10}-1}$" block. The latency of this WG core block (with re-use) is eleven clock cycles since the 29-bit output is generated after a data packet makes its second passage through the multipliers located between the stage registers $Q7$ and $Q9$. Similar to the "9-stage of $(\cdot)^{2^{10}-1}$" block, we provide a stage-by-stage description of the circuit in Figure 4.18 along with its timing diagram in Figure 4.19.

In Figure 4.19, the data packet A enters the WG core pipeline at clock cycle 0 and produces an output at clock cycle 11 while additional data packets (B, C, D, E, F) are fed into the pipeline with a *d_valid* of '1' every other clock cycle. Throughout our stage-by-stage description, we use the data packet A of Figure 4.19 as reference and it begins as follow.

- **Stages 1 to 6:** from clock cycle 1 to 6, the data packet A is inverted then propagated through the stage registers $Q1$ to $Q6$ and the first six pipeline stages of the $(\cdot)^{2^{10}-1}$ block.

- **Stages 7 and 8:** in clock cycle 7, the data valid register $V7$ is '1' so that the two multiplexers select the signals originated from the stage registers $Q7$ to be fed into the two multipliers in Figure 4.18. In clock cycle 8, the computation of both q_1 and t are completed and match with the first two multiplications located in the fourth stage of the pipeline in Figure 4.11.

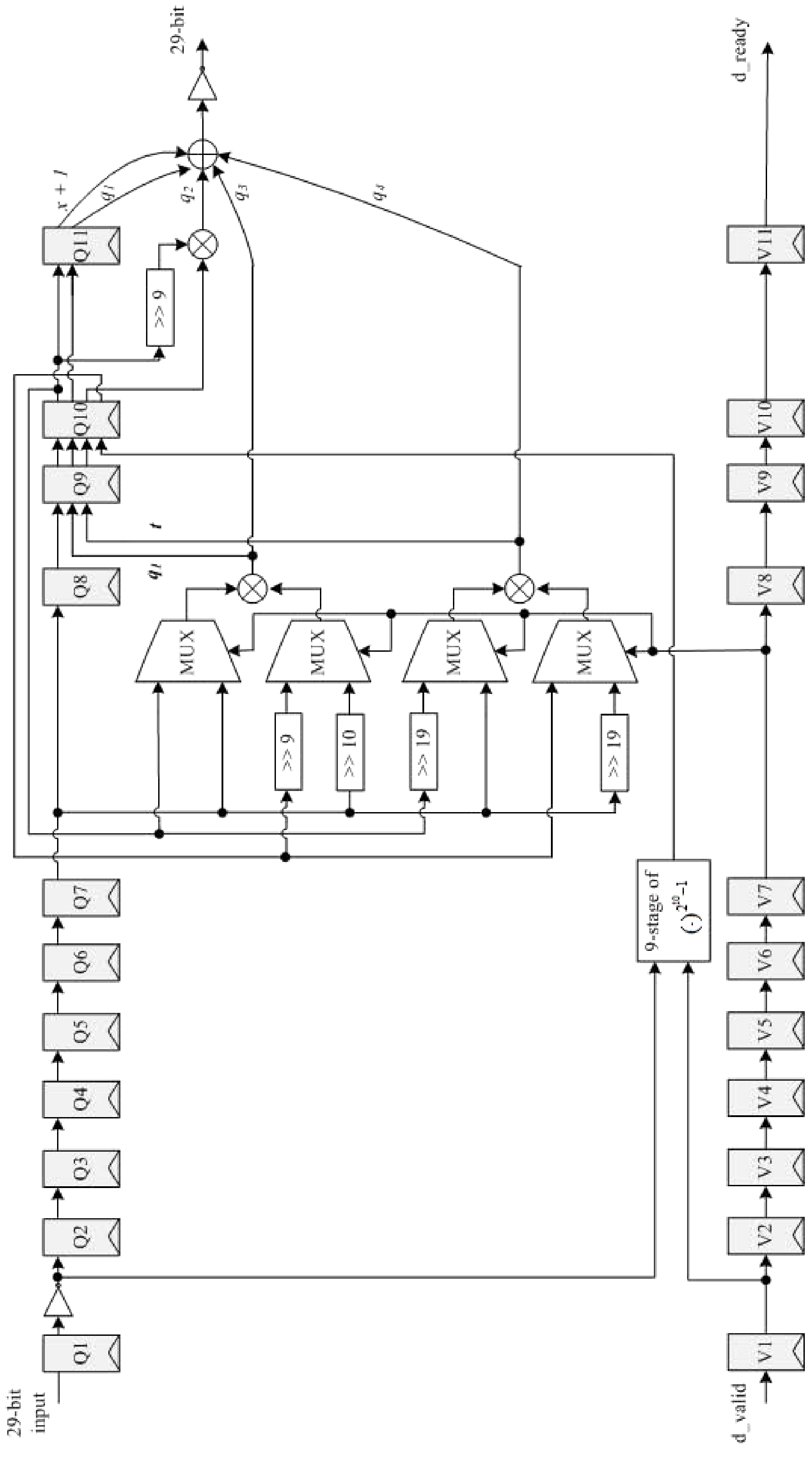


Figure 4.18: 11-stage Implementation of WG Core with Re-Use

clk cycle	0	1	2	3	4	5	6	7	8	9	10	11
29-bit input	**A**	∅	B	∅	C	∅	D	∅	E	∅	F	∅
d_valid	1	0	1	0	1	0	1	0	1	0	1	0
Q1		**A**	∅	B	∅	C	∅	D	∅	E	∅	F
V1		1	0	1	0	1	0	1	0	1	0	1
Q2			**A**	∅	B	∅	C	∅	D	∅	E	∅
V2			1	0	1	0	1	0	1	0	1	0
Q3				**A**	∅	B	∅	C	∅	D	∅	E
V3				1	0	1	0	1	0	1	0	1
Q4					**A**	∅	B	∅	C	∅	D	∅
V4					1	0	1	0	1	0	1	0
Q5						**A**	∅	B	∅	C	∅	D
V5						1	0	1	0	1	0	1
Q6							**A**	∅	B	∅	C	∅
V6							1	0	1	0	1	0
Q7								**A**	∅	B	∅	C
V7								1	0	1	0	1
Q8									**A**	∅	B	**A**
V8									1	0	1	0
Q9										**A**	∅	B
V9										1	0	1
Q10											**A**	∅
V10											1	0
Q11												**A**
V11												1

Figure 4.19: Timing Diagram of 11-stage Implementation of WG Core with Re-Use

- **Stage 9:** in clock cycle 9, the data packet A is temporarily stored in the stage registers $Q9$ because the inputs of the two multipliers (between the registers $Q7$ and $Q9$) are being occupied with the data packet B (stored in $Q7$).

- **Stages 10 and 11:** in clock cycle 10, the data valid register $V7$ is '0' so that the two multiplexers select the signals originated from the stage registers $Q10$ to be fed into the two re-used multipliers in Figure 4.18. Meanwhile, the stage registers $Q10$ are directly fed into the third multiplier located near the output of the WG core block. In clock cycle 11, the computations of all three multipliers (q_2, q_3, q_4) are completed and match with the last three multiplications located in the fifth stage of the pipeline in Figure 4.11.

Up to this point of our optimization process, the datapath (i.e. WG core and trace function) of our optimized implementation of WG is finalized and is shown in Figure 4.20. Since the trace function is simply the XOR of all 29 bits of its input, we did not apply any optimizations to this component.

In field programmable gate arrays designs, critical paths can be first approximated in terms of lookup tables (LUTs) because they are the basic building blocks. Recall that all normal basis multipliers in our optimized implementation of WG have been pipelined into two balanced stages. By observing the synthesized field programmable gate arrays schematic, both pipeline stages of the multiplier have a delay of two LUTs.

In order to maintain the delay of each pipeline stage to approximately a delay of two LUTs, the stage register $Q12$ are inserted at the input of the trace function to shorten the critical path from the stage register $Q11$ (inside the 11-stage WG core) to the 1-bit output of the whole WG implementation in Figure 4.20. The final critical path of our WG implementation is buried within the "11-Stage LFSR block, and this path begins from the stage registers (inside the multiplier) through a XOR gate then back to the input of the registers $S(1)$ as shown in Figure 4.3 of Section 4.1.2. We did not further pipeline (i.e. shorten) this critical path because it involves the feedback signal within the linear feedback shift register and pipelining it further(i.e. increasing its latency) would decrease the throughout of the linear feedback shift register without receiving a proportional increase in clock speed to maintain the current throughput.

Due to pipelining and hardware re-use of the WG core block, three control signals (*ce*, *d_valid*, *d_ready*) are added to maintain the data synchronization between the "11-Stage LFSR" and the "11-Stage WG Core" as shown in Figure 4.20. The generation of these three control signals are explained in the next section.

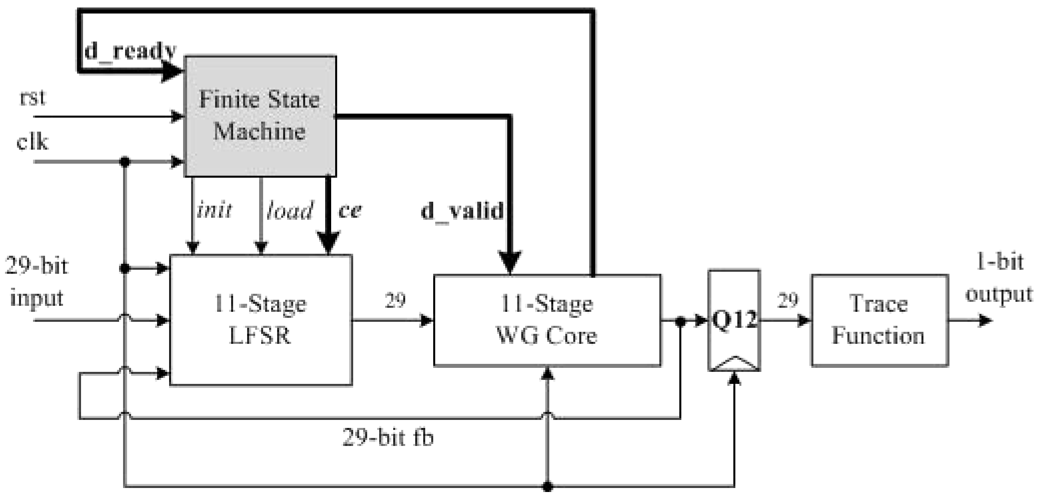


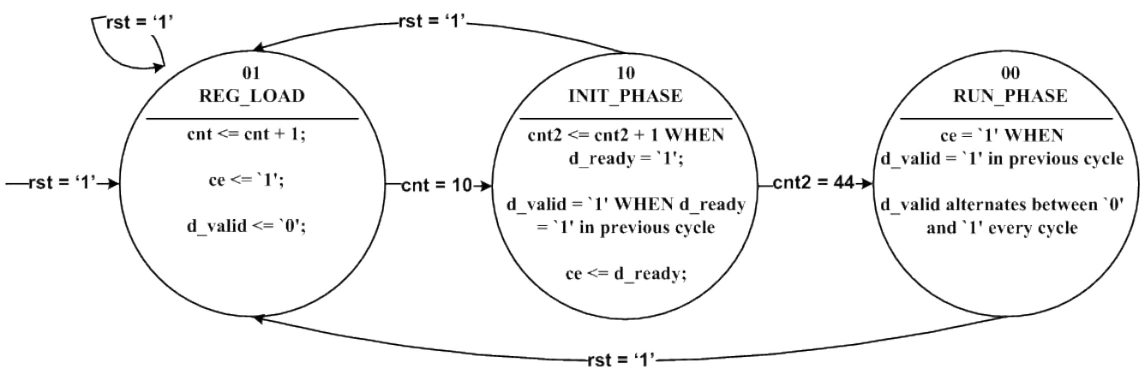
Figure 4.20: Optimized Implementation Block Diagram of WG

4.3.3 Control Circuitry Modifications

In this section, we first explain why the three control signals (*ce*, *d_valid*, *d_ready*) of Figure 4.20 have been added to our implementation of WG then we describe how the generation of these three control signals has been incorporated into the finite state machine.

In Figure 4.20, we have added chip-enable *ce* to all registers inside the "11-Stage LFSR" to control its input sampling rate. During the initialization phase of the WG cipher, the "11-Stage LFSR" can no longer sample its input every clock cycle because pipelining has increased the latency of the WG core block that feeds the signal *fb* back into the input of the linear feedback shift register. During the keystream generation of the WG cipher, all registers inside the "11-Stage LFSR can no longer shift every clock cycle because hardware re-use has decreased the data rate of the WG core block to one data packet per every two clock cycles as mentioned in Section 4.3.2. In Figure 4.20, we have added a data valid signal *d_valid* for the control of the multiplexers used in the hardware re-use of the normal basis multipliers inside the WG core block as mentioned in Section 4.3.2. As shown in Figure 4.18, the signal *d_ready* is simply a delayed version of the data valid signal *d_valid* and it is used for the generation of the chip-enable *ce* inside the finite state machine that is described next.

Figure 4.21 shows the original implementation of the finite state machine with additional modifications made for the three control signals (*ce*, *d_valid*, *d_ready*). The finite state machine remains the same in terms of the reset signal behaviour, the number of states, and the state encoding. In Figure 4.21, we have added the



rst = '1'
rst = '1'
01
REG_LOAD
cnt <= cnt + 1;
ce <= `1';
d_valid <= `0';
rst = '1'
cnt = 10
10
INIT_PHASE
cnt2 <= cnt2 + 1 WHEN d_ready = `1';
d_valid = `1' WHEN d_ready = `1' in previous cycle
ce <= d_ready;
cnt2 = 44
00
RUN_PHASE
ce = `1' WHEN d_valid = `1' in previous cycle
d_valid alternates between `0' and `1' every cycle
rst = '1'

Figure 4.21: Modified Finite State Machine Implementation

behaviour of the two control signals (*ce*, *d_valid*) in each state and we have changed the transition condition of the *INIT_PHASE* state by adding a second counter *cnt2* that is explained in this section.

During the state *REG_LOAD*, the loading of the linear feedback shift register is not affected by the two optimizations made to the WG core block. In the implementation of this state, *ce* is set to '1' and *d_valid* is set to '0' because there is no valid data fed to the WG core block and the linear feedback shift register shifts every clock cycle to load the key and the initialization vector into the registers.

When the finite state machine makes it transition to the state *INIT_PHASE*, it sets *d_valid* to '1' for one clock cycle because there is valid data (output of the LFSR has been loaded) to be fed into the 11-stage WG core. During that same clock cycle, all data valid registers inside the 11-stage WG core have a value of '0' because a data valid of '0' has been fed in the previous 11 clock cycles of loading (as described earlier). Since pipelining increased the WG core latency to 11 clock cycles, the number of clock cycles in the initialization phase increased from 44 to $44 \times (1 + 11)$ clock cycles. To save implementation area of the finite state machine, we have added a data packet counter *cnt2* instead of the expensive process of counting the large number of clock cycles elapsed in the initialization phase. This data packet counter *cnt2* is incremented by one whenever the finite state machine receives a *d_ready* of '1' from the WG core block. The implementation of our data packet counting is independent of the WG core latency because the first *d_valid* of '1' is propagated through the WG core and returned to the finite state machine as the signal *d_ready* to update the counter *cnt2* then sent back to the WG core as the next *d_valid* of '1'. Inside the finite state machine, the signal *d_ready* is also used to set the chip-enable *ce* to '1' because the LFSR can only be clocked when a data packet exits the 11-stage WG core (i.e. data packet arrives at the input of the LFSR). In Figure 4.21, the transition condition from state *INIT_PHASE* to state *RUN_PHASE* translates into processing 44 data packets through the WG core block (i.e. $cnt2 = 44$).

When the finite state machine makes it transition to the state *RUN_PHASE*, it sets *d_valid* to '1' because there is valid data (output of the LFSR has been initialized) to be fed into the 11-stage WG core. Since the optimization of hardware re-use has decreased the data rate of our WG implementation to one data packet per every two clock cycles, the finite state machine outputs a *d_valid* which alternates between '0' and '1' every clock cycle (i.e. one data valid per every two clock cycles). During the first clock cycle in the state *RUN_PHASE*, the chip-enable *ce* cannot have a value of '1' because a data packet (output of the LFSR) needs one clock cycle to travel from the output of the LFSR (register $S(11)$) to the input of register $S(1)$ as shown in Figure 4.3 (due to the additional stage register buried in the multiplier). Hence, the chip-enable *ce* is the signal *d_valid* with a delay of one clock cycle.

This optimized implementation was synthesized using Mentor Graphics Precision RTL synthesis tool. On an Altera Stratix II series field programmable gate arrays (FPGA) device EP2S15F484C, our pipelined implementation with re-use of WG has an area of 4184 LUTs (3740 registers) and a throughput of 109 Mbps at a clock speed of 218 MHz. In the next section, we compare our optimized implementation of WG to other state-of-the-art stream ciphers.

4.4 Related Work

Since the WG algorithm is a fairly recent stream cipher and no other implementation of WG has been previously published, we have compared the area and performance of our optimized WG implementation against three state-of-the-art stream ciphers in this section.

The stream ciphers Grain [9], MICKEY-128 [4] and Trivium [7] are the finalists in the eSTREAM project organized by European Network of Excellence for Cryptology (ECRYPT). One of the goals of the eSTREAM project was to identify a new and secure stream cipher that is suitable for hardware implementations with limited resources such as area, power and memory. To have a low complexity in hardware, all three stream ciphers (Grain, MICKEY-128, Trivium) are based on linear and non-linear feedback shift registers along with simple bitwise operations such as XOR and AND operations.

Both Grain and Trivium have the additional feature of increasing the throughput by duplicating the feedback circuitry of their shift registers d times. In doing so, their shift registers can be shifted by d positions per clock cycle and output d bits of the keystream per clock cycle instead of shifting by 1 position per clock cycle and outputting 1 bit of the keystream per clock cycle. This parallelization factor d has a maximum value depending on the cipher structure. Grain has a maximum parallelization factor of 16, and Trivium has a maximum parallelization factor of 64.

The area and performance of our optimized WG implementation is shown in Figure 4.22, where LUT denotes lookup table. All of our area and performance results were synthesized on an Altera Stratix II series FPGA device EP2S15F484C using Mentor Graphics PrecisionRTL. Thus, our optimized implementation of WG has an area of 4184 LUTs with a throughput of 109 Mbps at a clock speed of 218 MHz. In Figure 4.22, we have included the implementations of the three ciphers (Grain, MICKEY-128, Trivium) by Gaj *et al.* [8]. All of their implementations were synthesized onto the Xilinx Spartan 3 FPGA family devices using Synopsys tools.

As shown in Figure 4.22, the throughput of our optimized WG implementation is half of what Grain and Trivium can achieve when they have a parallelization factor of 1 (i.e. basic architecture). If we did not apply hardware re-use to our WG implementation, we would achieve a throughput of 218 Mbps because the data rate would return to one data packet per every clock cycle instead of one data packet per every two clock cycles. Compared with the basic architecture of the three ciphers, the WG cipher can achieve a competitive throughput. However, both Grain and Trivium can greatly increase their throughputs when they parallelize their operations with their respective maximum factor d of 16 and 64 as shown in Figure 4.22.

In Figure 4.22, the area of our WG implementation greatly exceeds the implementation area of all other three ciphers. This is due to the fact that the WG cipher is based on normal basis multiplication whereas the other three ciphers are based on simple shifting as well as bitwise XOR and AND operations. From our synthesis results, a 29-bit normal basis multiplier in our WG implementation costs approximately 600 LUTs. In parallelized architecture, both Grain and Trivium have a greater implementation area than their basic architecture counterpart because additional hardware was inserted to increase the throughput as mentioned earlier.

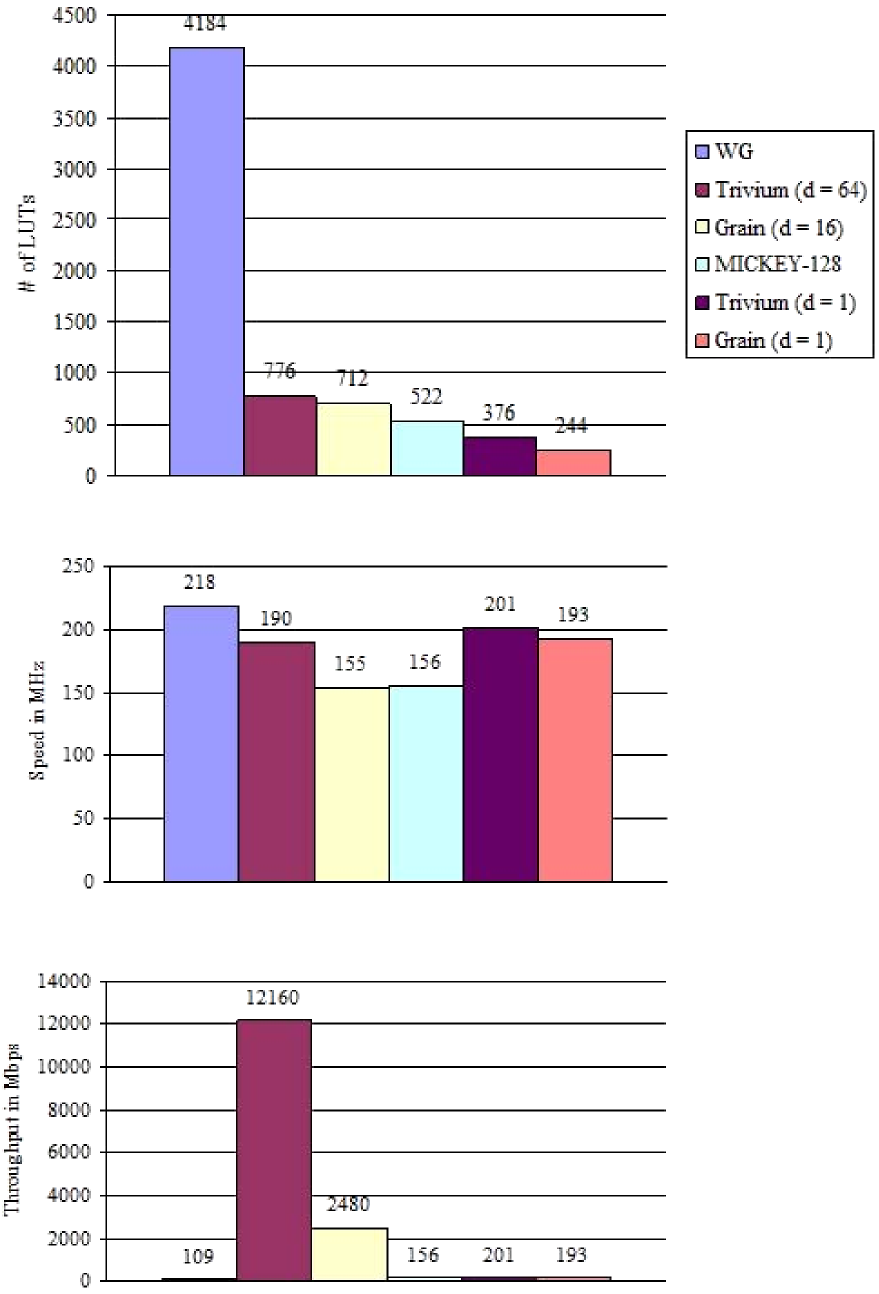


Figure 4.22: Area and Performance Results of Various Stream Ciphers

Although our optimized implementation of WG has a large implementation area, it offers a high level of security with proven mathematical properties. As mentioned in Section 2.6, various architectures of the WG cipher exist depending on their design parameters. By lowering the level of security and decreasing the number of LFSR stages as well as the number of bits used in the multiplication, the WG cipher can achieve a higher throughput with less implementation area.

In this chapter, we have covered the design and the verification of the non-pipelined implementation of WG (Section 4.1 and 4.2). In Section 4.3, we have showed how the optimizations of pipelining and hardware re-use were applied to form our optimized implementations of WG. In Section 4.4, we conclude the chapter with a comparison of our optimized WG implementation against three state-of-the-art stream ciphers. The next chapter contains the verification of the Welch-Gong cipher.

Chapter 5

WG: Verification

The purpose of this chapter is to verify our optimized implementation of WG and to develop a method of applying the completion functions approach such that it can deal with a circuit that has been optimized with hardware re-use. Since all of our optimizations were applied to the WG core block, Section 5.1 first describes how completion functions were used to verify this block. In Section 5.2, we verify the linear feedback shift register in which a chip-enable signal was added to accommodate the optimization made to the WG core block. During the verification of both the WG core block and the linear feedback shift register, several assumptions were made about the control signals (*d_valid*, *ce*) and these assumptions were confirmed with the verification of the finite state machine by model checking in Section 5.3. Section 5.4 is used to describe related work.

5.1 The WG Core

Similar to the verification of KASUMI, we have used completion functions in combination with equivalence checking for the verification of the optimized datapath (i.e. WG core) of WG. All verification methodologies suggested in the KASUMI chapter apply in this section as well. Since the WG core block consists of the $(\cdot)^{2^{10}-1}$ block which is formed by normal basis multipliers, we have taken a bottom-up modular verification approach in which we begin with the verification of the multipliers followed by the verification of the $(\cdot)^{2^{10}-1}$ block then the verification of the WG core. In conducting our verification in a bottom-up modular way, we have developed a new methodology that can decrease the number of verification obligations and we refer to it as "skipping" (explained later in this section).

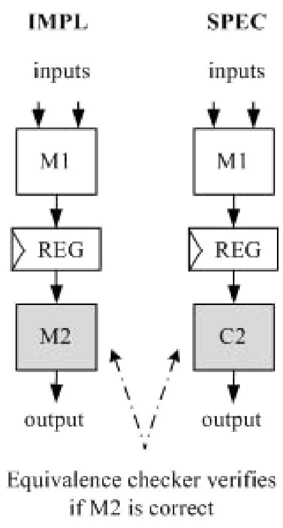


Figure 5.1: Multiplier: 1st Obligation

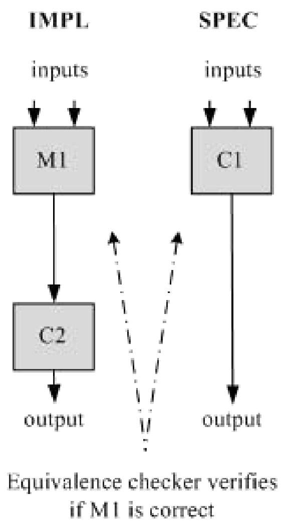


Figure 5.2: Multiplier: 2nd Obligation

5.1.1 First Verification: The Normal Basis Multiplier

We began the verification with the smallest building block (i.e. multiplier) because it helps the reader to understand the methodology of "skipping" in later sections. Since only pipelining was used to optimize the normal basis multiplier, this verification is identical to the one of KASUMI except that the number of stages differs. In Figure 5.1 to 5.3, we have used the block $M1$ to represent the first pipeline stage of the multiplier, the block $M2$ to represent the second stage and REG to denote the stage registers.

Figure 5.1 illustrates the first equivalence check used to verify the correctness of the second pipeline stage $M2$ with respect to the completion function of the second stage $C2$ of the multiplier. In Figure 5.2, the equivalence checker verifies that the first pipeline stage $M1$ has the behaviour described by the completion function of the first stage $C1$. Since we have verified our non-pipelined implementation of WG with RTL simulations in Section 4.2, we can use our non-pipelined implementation of the normal basis multiplier as the specification (denoted as $SPEC$) in the final verification obligation. The final verification of the normal basis multiplier is to verify that the completion function of the first stage $C1$ is equivalent to the non-pipelined implementation of the multiplier $SPEC$ as depicted in Figure 5.3.

Since the 2-stage normal basis multiplier has been verified here, we would not need to verify this pipeline again when it is used to form a larger component such as the $(\cdot)^{2^{10}-1}$ block. However, we would still need to verify the connection to the inputs and outputs of the pipelined multipliers in order to prove the correctness of

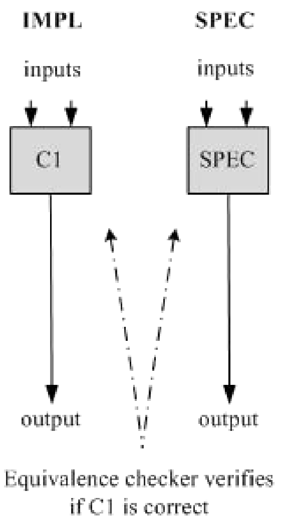


Figure 5.3: Multiplier: Final Obligation

the larger component. This is the fundamental concept of our methodology known as "skipping" (skip internal stages of a verified pipeline), and it is demonstrated in the verification of the $(\cdot)^{2^{10}-1}$ block described the next subsection.

5.1.2 Second Verification: The $(\cdot)^{2^{10}-1}$ Block

Moving up one level in the hierarchy of the datapath of WG, the verification of the $(\cdot)^{2^{10}-1}$ block can be used to show how completion functions are used to verify a circuit which has been optimized with hardware re-use and to demonstrate the methodology of "skipping".

In Section 4.3.2, we have provided a stage-by-stage description of our 9-stage implementation of the $(\cdot)^{2^{10}-1}$ block. If we apply the same verification methodology used in the verification of KASUMI, we would need ten verification obligations (one per stage and one final obligation) to completely verify the correctness of the $(\cdot)^{2^{10}-1}$ block. Since the 2-stage normal basis multiplier within the $(\cdot)^{2^{10}-1}$ block has already been verified in Section 5.1.1, we do not need to verify this multiplier again. Therefore, this fact allows us to "skip" the verification of the internal stages of the multipliers and decreases the number of verification obligations of the $(\cdot)^{2^{10}-1}$ block from ten to six as shown in this section.

Similar to the verification of KASUMI, we began the verification of the $(\cdot)^{2^{10}-1}$ block from its last pipeline stage (i.e. 9^{th} stage). As mentioned in the stage-by-stage description of this pipeline in Section 4.3.2, the 9^{th} pipeline stage is formed by

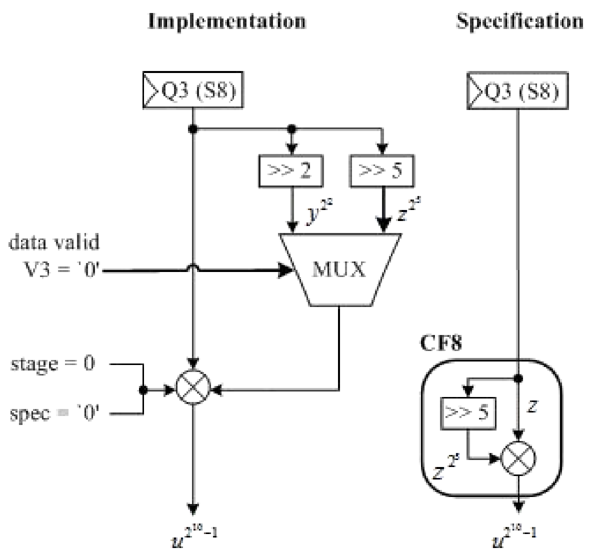


Figure 5.4: Stage 8 Obligation with Skipping and *d_valid*='0'

the second stage of the normal basis multiplier. In Section 5.1.1, we have already verified the correctness of the 2-stage normal basis multiplier. Therefore, we skipped the verification of the 9th pipeline stage of the $(\cdot)^{2^{10}-1}$ block. For the same reason, we skipped the verification of the 2nd, 4th and 7th pipeline stage of the $(\cdot)^{2^{10}-1}$ block. Although we skipped the verification of the internal stage of the multiplier, we still need to verify if the input and output connections of the multipliers are correct. To do so, we verify the 1st, 3rd, 5th, 6th and 8th pipeline stage of the $(\cdot)^{2^{10}-1}$ block. Similar to the case study of KASUMI, we have used the VHDL generic parameter *stage* to specify which pipeline stage is under verification and the *spec* to indicate whether the specification or the implementation is synthesized during equivalence checking. Throughout these verification obligations, we set both VHDL generic parameters *stage* and *spec* to 0 so that the multiplier block becomes its purely combinational form as shown in Figure 5.4 to 5.8. In doing so, we can verify the input and output connections of the multiplier while skipping the internal stage of the multiplier.

We proceed to our first equivalence check used for the verification of the 8th stage as shown in Figure 5.4. On the specification side (right hand side) of Figure 5.4, the completion function of the 8th stage ($CF8$) simply describes the desired behaviour of the 8th stage and it does not provide any description about the multiplexer (on the left hand side) because the multiplexer is an implementation-specific detail added for the optimization of hardware re-use. With the presence of this multiplexer, the equivalence checker cannot prove the equivalence between the circuit on the implementation side and the circuit on the specification side. To remedy this problem, we have to make the assumption on the implementation side that the data valid register $V3$ is '0' so that the multiplexer feeds the signal z^{2^5} to the multiplier in Figure 5.4. This assumption matches with the stage-by-stage description of this pipeline in Section 4.3.2 and the timing diagram of Figure 4.17 because the data valid register $V3$ should be '0' when a valid data packet reaches the 8^{th} pipeline stage. In the verification of the 3^{rd} pipeline stage shown in Figure 5.7, we made a different assumption that the data valid register $V3$ is '1' because the re-used multiplier now takes the signal y^{2^2} as input according to its stage-by-stage description in Section 4.3.2. Thus, completion functions can handle the optimization of hardware re-use only if we make assumptions about the data valid registers during equivalence checking.

Similarly, the remaining verification obligations proceed in the same manner where the equivalence checker verifies the equivalence between the circuit on the implementation side (left hand side) and the circuit on the specification side (right hand side). For the implementation side to be equivalent to the specification side, we have to make an assumption about the associated data valid registers Vi so that the multiplexers (added on the implementation side for hardware re-use) feed the appropriate signals into the multipliers.

Figure 5.5 depicts the verification of the 6^{th} pipeline stage and the second verification obligation in which we made the assumption that the data valid register $V1$ is '0' so that it matches with the stage-by-stage description in Section 4.3.2. Figure 5.6 shows our third verification obligation used to verify the correctness of the 5^{th} pipeline stage. Since the 5^{th} pipeline stage of the $(\cdot)^{2^{10}-1}$ was added to store the data packets and to avoid the contention of these data packets at the inputs of the multiplexers, we simply connect the completion function of the next stage (6^{th} stage) to the stage register $Q5$ on the implementation side. In Figure 5.7, we have made the assumption that the data valid register $V3$ is '1' so that the equivalence checker can compare the implementation against the specification in the verification of the 3^{rd} pipeline stage. Figure 5.8 illustrate the verification of the 1^{st} pipeline stage in which we have assumed that the data valid register $V1$ is '1'.

The previous five verification obligations have verified the pipeline stages under the assumption that the completion functions are bug-free and the data valid signals are generated in a specific manner. To verify the correctness of the completion functions, we can simply check the equivalence between the completion function of the 1^{st} stage and the specification of the $(\cdot)^{2^{10}-1}$ block (i.e. non-pipelined implementation of Figure 4.2 in Section 4.1) as shown in Figure 5.9. To verify the

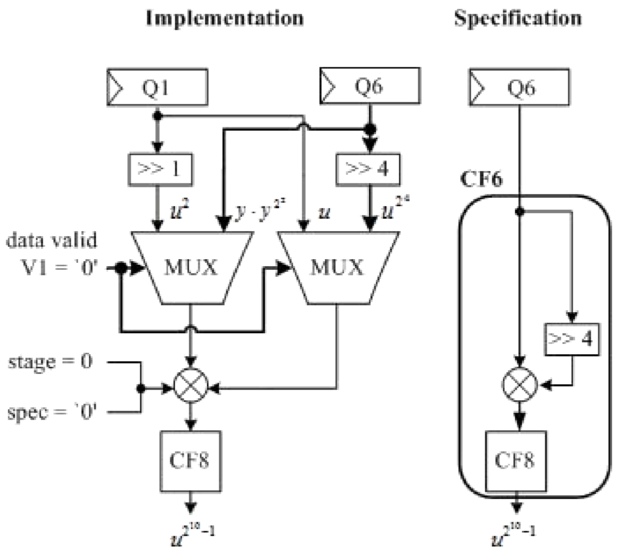


Figure 5.5: Stage 6 Obligation with Skipping and *d_valid*='0'

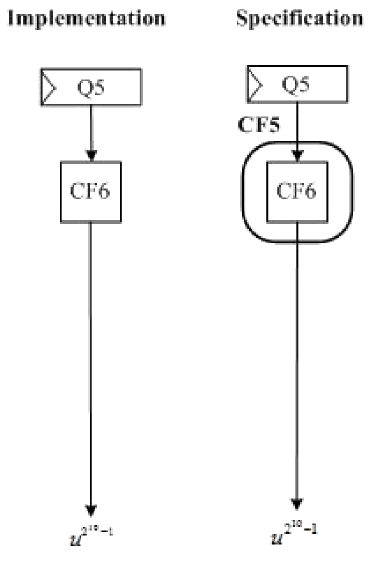


Figure 5.6: Stage 5 Obligation

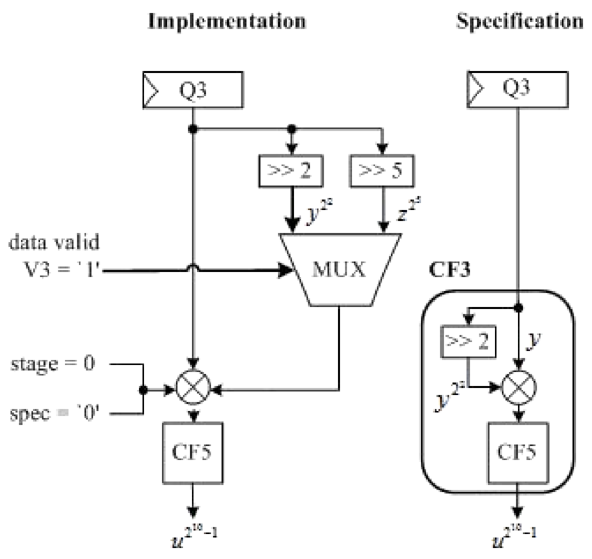


Figure 5.7: Stage 3 Obligation with Skipping and *d_valid*='1'

assumptions about the data valid signals, we have to verify the finite state machine because the data valid signals are generated by the finite state machine. Prior to the verification of the finite state machine in Section 5.3, we first go up one level in the hierarchy and verify the correctness of the WG core block in the next subsection.

5.1.3 Third Verification: The WG Core

To complete the verification of the datapath of WG, the verification of the WG core is provided here. Similar to the verification of the $(\cdot)^{2^{10}-1}$ block, the verification of the WG core began with the last pipeline stage and ends with the first pipeline stage.

Figure 5.10 shows our first verification obligation used to verify the correctness of the 11^{th} stage of the WG core. The completion function $CF11$ on the specification side (right hand side) is used to define to correct behaviour of the 11^{th} stage. The circuit on the implementation side (left hand side) is the circuit under verification. The equivalence checker compares the implementation side against the specification side to verify the correctness of the 11^{th} stage. Since all normal basis multipliers were verified in Section 5.1.1, we skipped the verification of all three multipliers by instantiating all three multipliers on both the implementation and specification sides in their combinational form (by setting the generic parameters *stage* and *spec* to 0). Although we skipped the verification of the multipliers, we did not skip the verification of the output connections of all three multipliers as shown in Figure

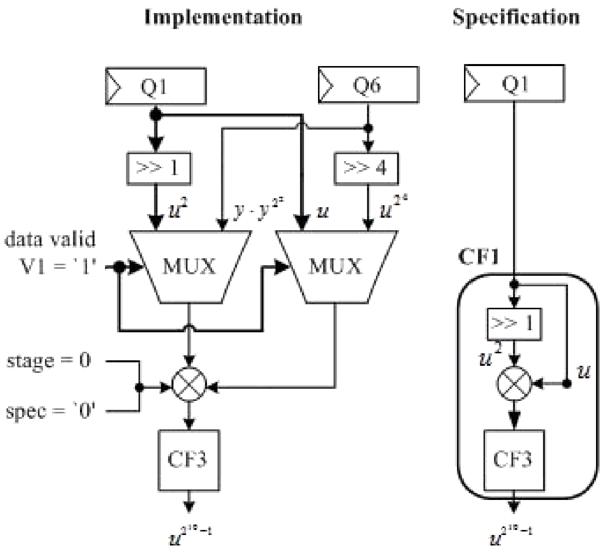


Figure 5.8: Stage 1 Obligation with Skipping and *d_valid*='1'

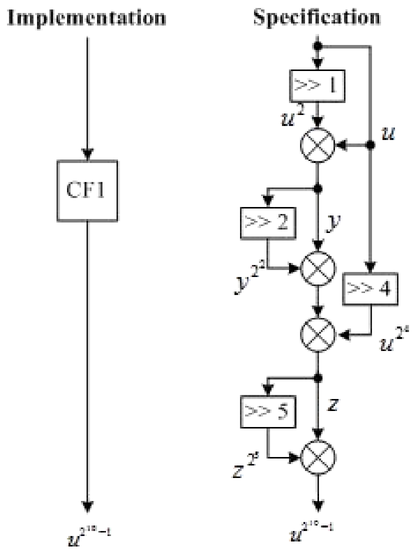


Figure 5.9: Final Obligation of $(\cdot)^{2^{10}-1}$ Block

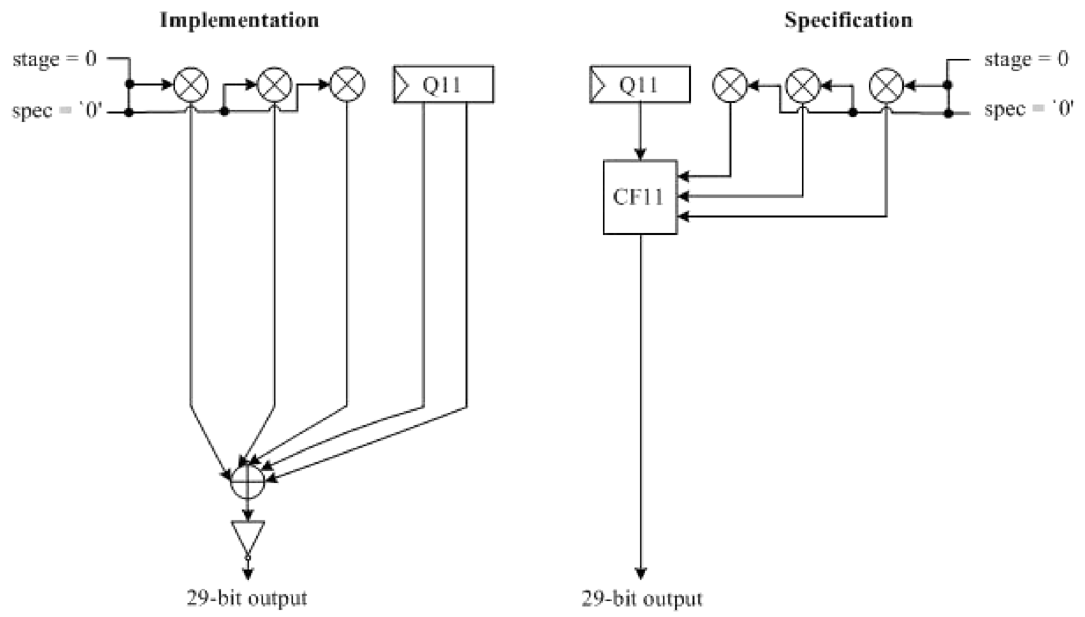


Figure 5.10: Stage 11 Obligation of WG Core

5.10. The verification of the input connections of all three multipliers is included in the next verification obligation.

Similar to the previous verification obligation, the circuit on the implementation side of Figure 5.11 is defined as correct if the equivalence checker proves that the circuit on the implementation side is equivalent to the completion function $CF10$ on the specification side which is used to define the behaviour of the 10^{th} pipeline stage. Due to the optimization of hardware re-use, the inputs of all three multipliers on the implementation side can either be from the stage registers $Q7$ or $Q10$ depending on the value of the data valid register $V7$ fed to the multiplexers. Similar to the verification of the $(\cdot)^{2^{10}-1}$ block, we have to make an assumption about the value of the data valid register $V7$ and this assumption has to match with the stage-by-stage description of this pipeline in Section 4.3.2. For this verification obligation, we made the assumption that the data valid register $V7$ is '0' so that the stage registers $Q10$ are fed to all three multipliers. By doing so, the equivalence checker can prove the equivalence between the circuit on the implementation side and the completion function $CF10$ on the specification side. In this verification obligation, we verified that the connections to the inputs of all three multipliers are correct. As explained in Section 3.5 of the KASUMI chapter, we have connected the completion function $CF11$ to the output of the combinational circuitry on the implementation side because the equivalence checker can apply structural matching to decrease its computational complexity. Note that the completion function $CF11$ is internally

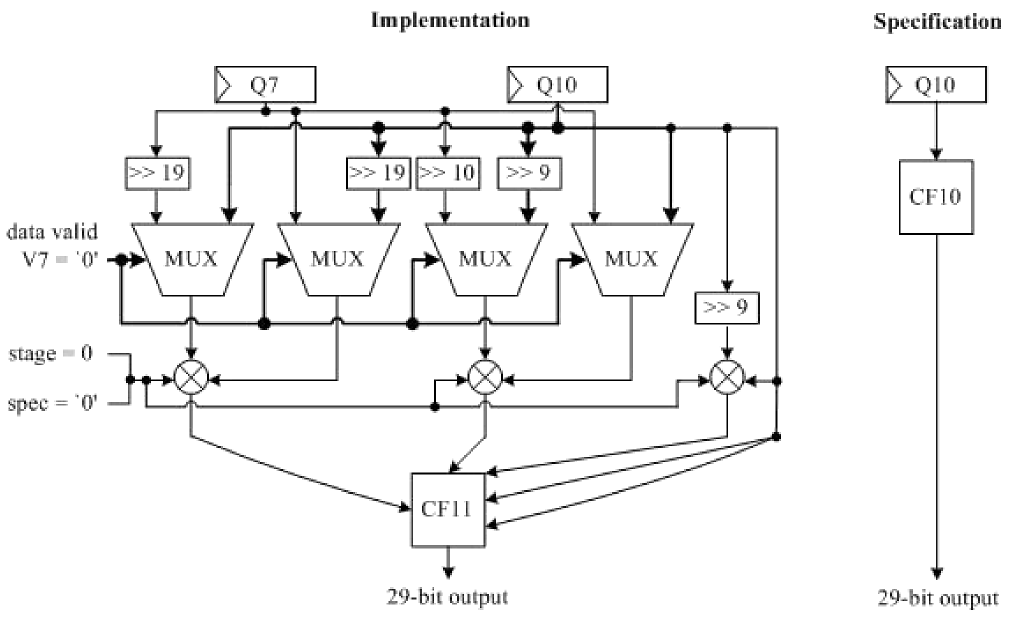


Figure 5.11: Stage 10 Obligation of WG Core

part of the completion function $CF10$ but it is not shown in Figure 5.11. Next, we verify the correctness of the 9^{th} pipeline stage of the WG core block.

According to the stage-by-stage description of the WG core pipeline in Section 4.3.2, the 9^{th} pipeline stage was added for the storage of the data packets to avoid the contention of the data packets at the inputs of the multiplexers. As shown in Figure 5.12, the circuit on the implementation side is simply the stage registers $Q9$ connected to the completion function of the 10^{th} pipeline stage because there is no computation other than storage in this stage. Similar to the normal basis multiplier, the "9-stage of $(\cdot)^{2^{10}-1}$" block was already verified in Section 5.1.2 and we skipped its verification by instantiating this block on both the implementation and specification sides in their combinational form (by setting the generic parameters *stage* and *spec* to 0). Although we skipped the verification of the "9-stage of $(\cdot)^{2^{10}-1}$" block, we did not skip the verification of its output connections as shown in Figure 5.12. The verification of the input connections of the "9-stage of $(\cdot)^{2^{10}-1}$" block is included in the verification of the 1^{st} pipeline stage described later in this section. In Figure 5.12, the equivalence checker proved the equivalence between the circuit on the implementation side and the completion function $CF9$ on the specification side. Next, we verify the correctness of the 8^{th} pipeline stage of the WG core block.

According to the stage-by-stage description of the WG core pipeline in Section 4.3.2, the 8^{th} pipeline stage is formed by the second stage of the normal basis multiplier (which was already verified). For this reason, we skipped the verification

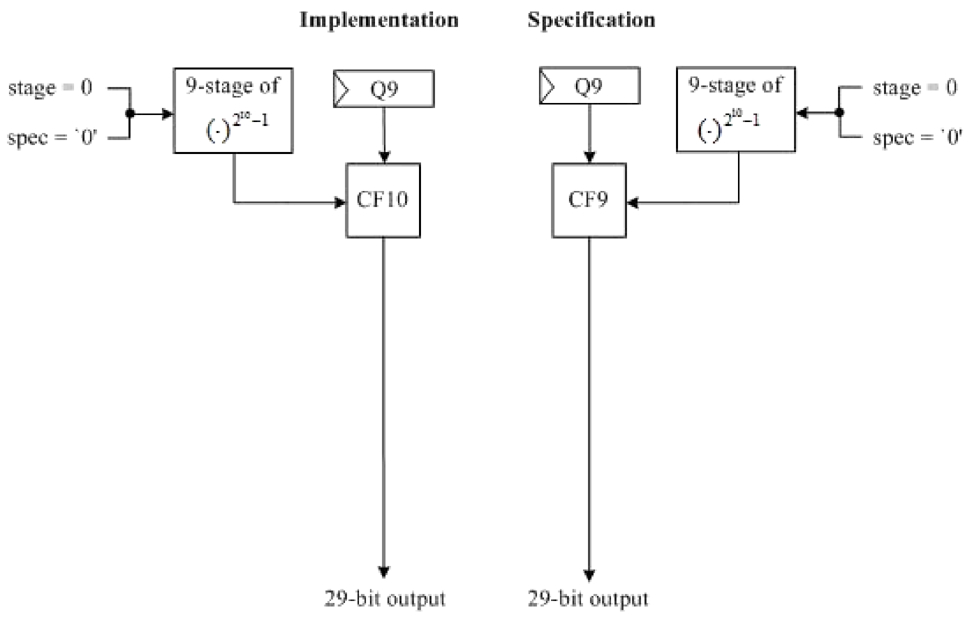


Figure 5.12: Stage 9 Obligation of WG Core

of the 8th pipeline stage of the WG core. In Figure 4.18 of Section 4.3.2, the stage registers $Q8$ (outside to the multipliers) are used to synchronize the data packets flowing inside and outside of the multipliers. Although the methodology of “skipping” saved the verification of the 8th pipeline stage, the circuit designer or the verification engineer still needs to verify that the stage registers $Q8$ (outside of the multipliers) are correctly connected to the other stage registers so that the data packets in the pipeline are synchronized correctly. This simple verification was done by VHDL code review. The user can choose to not skip this verification and pursue a stage-by-stage verification strategy without making use of “skipping”. Next, we verify the correctness of the 7th pipeline stage of the WG core block.

Similar to the verification of the 10th pipeline stage of the WG core, we made the assumption that the data valid register $V7$ is ‘1’ to select the correct input of the multiplier and we connected the completion function $CF9$ to the output of the combinational circuitry so that the equivalence checker can apply structural matching to decrease its computational complexity as shown in Figure 5.13. Note that the “9-stage of $(\cdot)^{2^{10}-1}$” block still exist on both the implementation and specification sides because the input connections of this block have not been verified and the semantics of this block have not been captured by any completion functions yet (to be shown in the verification of the 1st pipeline stage). In Figure 5.13, the equivalence checker proved the equivalence between the implementation and the specification. Next, we verify the correctness of the 6th pipeline stage of the WG

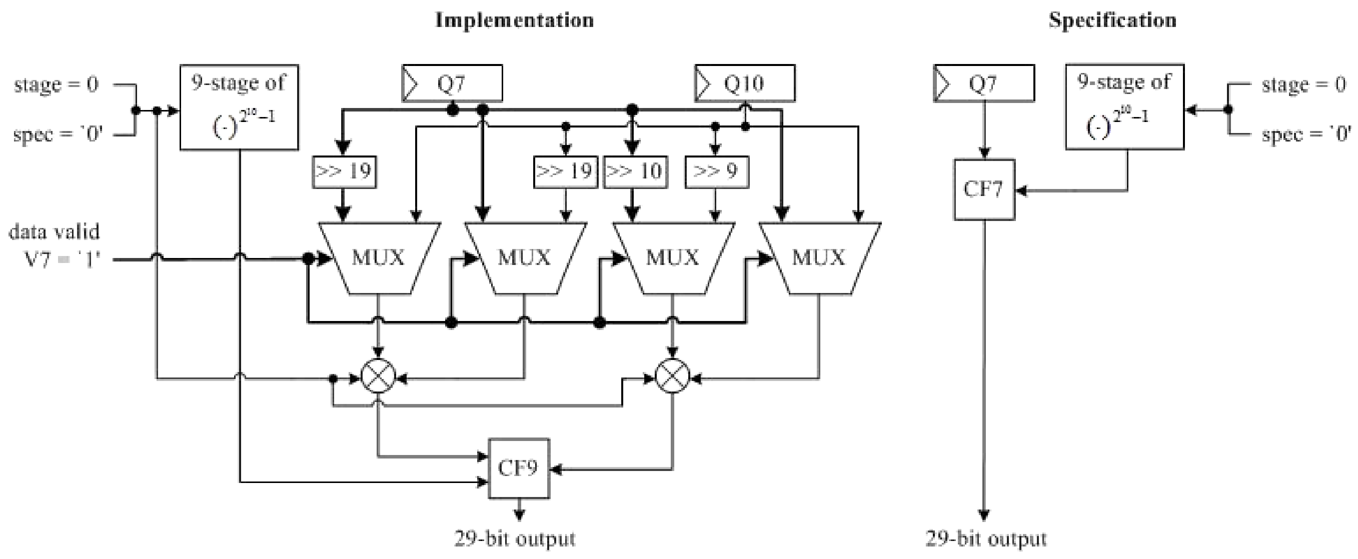


Figure 5.13: Stage 7 Obligation of WG Core

core block.

Similar to the verification of the 8^{th} pipeline stage of the WG core, we skipped the equivalence verification of the 6^{th} pipeline stage and we used VHDL code review to verify that the stage registers $Q6$ are correctly connected to the other stage registers so that the data packets in the pipeline are synchronized correctly. This is because the 6^{th} pipeline stage of the WG core is formed by the 6^{th} stage of the "9-stage of $(\cdot)^{2^{10}-1}$" block (which was already verified). Similarly, we verified the 2^{nd} to 5^{th} pipeline stage of the WG core in the same manner because these pipeline stages are formed by the 2^{nd} to 5^{th} pipeline stage of the "9-stage of $(\cdot)^{2^{10}-1}$" block. Next, we verify the correctness of the 1^{st} pipeline stage of the WG core block.

Figure 5.14 depicts the equivalence check used in the verification of the 1^{st} pipeline stage of the WG core. By checking the equivalence between the completion function of the first stage $CF1$ on the specification side and the circuit on the implementation side, we verified that the input connections of the "9-stage of $(\cdot)^{2^{10}-1}$" block are correct. Although "skipping" was applied to the "9-stage of $(\cdot)^{2^{10}-1}$" block, the completion function of the first stage $CF1$ still captures the semantics of the "skipped" components (shown in Figure 5.14) because the completion function $CF1$ needs to be verified in the final verification obligation as discussed next.

The previous five verification obligations have verified the pipeline stages of the WG core under the assumption that the completion functions are bug-free and the data valid signals are generated in a specific manner. To verify the correctness of the

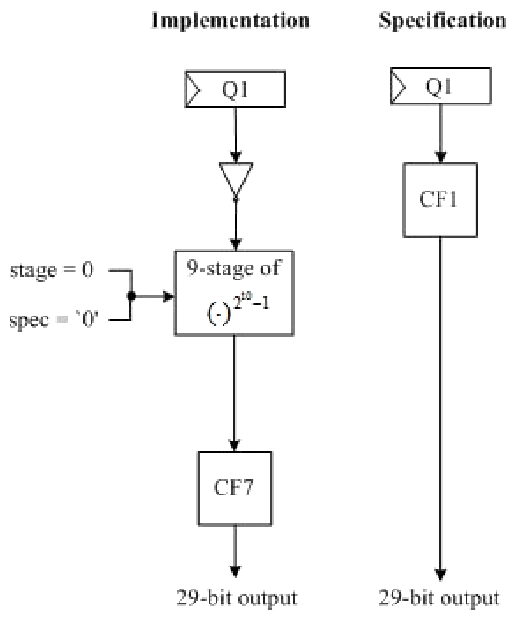


Figure 5.14: Stage 1 Obligation of WG Core

completion functions, we can simply check the equivalence between the completion function of the 1^{st} stage and the specification *SPEC* of the WG core (i.e. non-pipelined implementation in Section 4.1.1) as shown in Figure 5.15. Prior the to verification of the assumptions about the data valid signals in Section 5.3, we first verify the correctness of the linear feedback shift register with the addition of the chip-enable signal in the next section.

5.2 The Linear Feedback Shift Register

In our first non-pipelined WG implementation of Section 4.1, the linear feedback shift register shifts and feeds a data packet per clock cycle to the WG core. Due to the application of hardware re-use onto the WG core in Section 4.3.2, the WG core can only process one data packet per every two clock cycles instead of the original one data packet per clock cycle. To change the rate at which the linear feedback shift register feeds data packets to the WG core, a chip-enable signal was added to all registers inside the linear feedback shift register in Section 4.3.3. In this section, we describe how we have used equivalence checking to verify the correctness of the modified linear feedback shift register with chip-enable.

Figure 5.16 shows the equivalence check used in the verification of the linear feedback shift register with chip-enable. On the specification side of Figure 5.16,

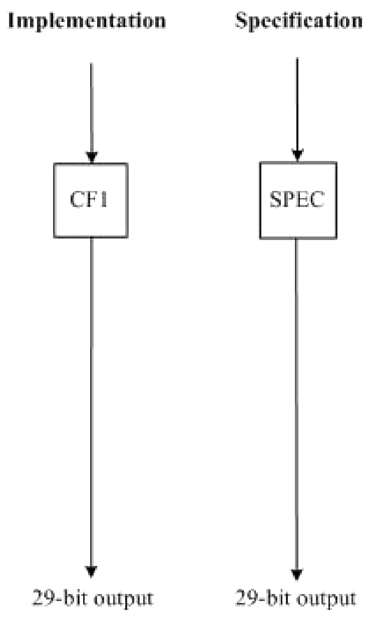


Figure 5.15: Final Obligation of WG Core

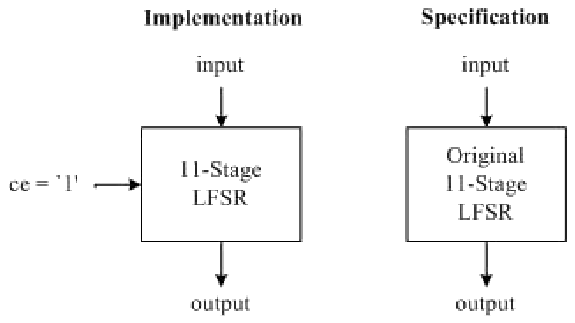


Figure 5.16: Verification of LFSR with *ce*='1'

we used the original linear feedback shift register of Section 4.1.2 because its correctness was already verified with RTL simulations in Section 4.2.2. Similar to the verification of the WG core with hardware re-use, we have to make an assumption about the chip-enable *ce* of the linear feedback shift register on the implementation side so that the equivalence checker can verify its correctness with respect to the specification on the right hand side. As shown in Figure 5.16, we made the assumption that the chip-enable *ce* is '1' because the linear feedback shift register with chip-enable would shift and this is equivalent to the original linear feedback shift register. If we made the assumption that the chip-enable *ce* is '0', the linear feedback shift register with chip-enable would not shift and the equivalence checker would conclude that the implementation is not equivalent to the specification in Figure 5.16.

Recall that there is a normal basis multiplier within the linear feedback shift register as shown in Figure 4.3. This multiplier is purely combinational in the original linear feedback shift register but becomes a 2-stage pipeline in the linear feedback shift register with chip-enable due to pipelining. Since the pipelined normal basis multiplier was already verified in Section 5.1.1, we instantiated the multiplier in its purely combinational form on both the implementation and specification side of Figure 5.16 by setting the VHDL generic parameters *stage* and *spec* to ‘0’. By doing so, the equivalence checker can decrease its computational complexity with structural matching. Up to this point, the verification of our optimized implementation of WG is correct if the assumptions about the data valid and chip-enable signals are correct. To verify that our assumptions are correct, we verify the finite state machine that generates these two control signals in the next section.

5.3 The Finite State Machine

Throughout the verification of our optimized WG implementation, we made assumptions about the data valid and chip-enable signals based on the fact that these signals are correctly generated by the finite state machine. To verify the generation of these two control signals by the finite state machine, we used a method known as model checking. In this section, we first provide background information about model checking then we describe how it was used to verify the finite state machine of our optimized WG implementation.

5.3.1 Background of Model Checking

Model checking is an automatic formal verification technique used to verify finite state systems. In the verification of hardware designs, these systems in hardware description language are first translated into state transition graphs then these graphs are traversed to check if they satisfy certain properties. In the case that the system does not satisfy a property, model checking provides a counterexample to show why the property does not hold. In our verification, we used linear temporal logic formulas to express the properties that we verified.

Linear temporal logic (LTL) formulas are formed by propositional variables, logical connectives and temporal modal operators. Propositional variables are Boolean variables used to represent a proposition (i.e. property). To link multiple propositions together, the following five logical connectives are used: negation $\neg$, con-

Temporal Operator	LTL Formula	Meaning of Formula
next X	X a	a is true at the next state
globally G	G a	a is true globally on the path
eventually F	F a	a is eventually true somewhere on the path
until U	a U b	a is true until b is true, and b is true at current or future state

Table 5.1: Semantics of Temporal Modal Operators

junction $\wedge$, disjunction $\vee$, and material implication $\Rightarrow$. By adding temporal modal operators, the aspect of time is added to these propositions. Given two propositional variables a and b, the semantics of the temporal modal operators are shown in Table 5.1. Note that a path is a sequence of states that describe the behaviour of the circuit. A path satisfies a linear temporal logic formula if and only if the initial position of that path satisfies this formula.

5.3.2 The Verification

The verification of the finite state machine began by checking the VHDL code of our optimized WG implementation into IBM RuleBase verification tool which can handle model checking. Similar to conventional simulation, a set of environment stimulus needs to be defined before verifying if our finite state machine satisfies certain properties. As shown in Figure 4.20, there are two environment variables that can affect the behavior of the finite state machine: the clock signal *clk* and the reset signal *rst*. Under normal operation of the WG cipher, the clock signal is always active and the reset signal is only asserted on the first clock cycle to initialize the finite state machine then it is de-asserted for the remaining clock cycles. Therefore, we defined (i.e. modeled) these two environment variables as described above.

To verify that the data valid *d_valid* and chip-enable *ce* are correctly generated by the finite state machine as described in Section 4.3.3, we devised 19 linear temporal logic formulas (i.e. properties) and divided them intro three sets. The first set is used to verify whether the finite state machine makes transition from state to state in the desired sequence. The second set is used to verify if the finite state machine remains in each state for a correct number of clock cycles. The third set is used to verify whether the finite state machine generates the correct data valid *d_valid* and chip-enable *ce* in each state. In the rest of this section, all

Property	Formula
1	$rst \Rightarrow \mathrm{X}\ reg\_load$
2	$reg\_load \Rightarrow reg\_load\ \mathrm{U}\ init\_phase$
3	$init\_phase \Rightarrow init\_phase\ \mathrm{U}\ run\_phase$
4	$run\_phase \Rightarrow \mathrm{X}\ run\_phase$

Table 5.2: First Set of Properties Verified

Operator	Formula	Meaning of Formula
prev	prev a	a is true at the previous cycle
next_a[$i..j$]	next_a[$i..j$] a	a is true for all cycles from i^{th} to j^{th} cycle
next[i]	next[i] a	a is true at the i^{th} cycle

Table 5.3: Semantics of Additional Operators

linear temporal logic formulas are evaluated globally but we did not include the temporal modal operator G at the beginning of each formula and we have replaced the proposition (state = *state_name*) by *state_name* for readability purpose.

Table 5.2 shows the first set of properties used to verify that the state machine makes state transitions in the correct sequence. These properties have shown that the reset signal *rst* is asserted to initialize the state machine at the state *reg_load* (property 1) then it remains in this state until it enters the state *init_phase* (property 2). Once the state machine enters the state *init_phase*, it remains in this state until it enters the state *run_phase* (property 3) where the state machine remains in this state (property 4) for the remaining clock cycles to generate the keystream. This sequence of state transitions matches with the description provided in Section 4.2.1.

To represent complex properties (i.e. formulas) in a simple form, we used three additional RuleBase built-in operators: prev, next_a[$i..j$], and next[i]. Given a proposition a and two integers i and j, the semantics of these operators are shown in Table 5.3.

Table 5.4 illustrates the second set of properties used to verify that the state machine remains in each state for a correct number of cycles. These properties have demonstrated that the state machine remains in the state *reg_load* for at least 11 cycles (property 5) and makes a transition to the state *init_phase* at the 12^{th} cycle (property 6) so that the 11-stage linear feedback shift register has the exact number of cycles required to load all of its registers. Once the state machine enters the state *init_phase*, it remains in this state for at least 528 cycles (property 7) and

Property	Formula
5	$reg\_load \wedge \text{prev } \neg reg\_load \Rightarrow \text{next\_a}[1..10]\ reg\_load$
6	$reg\_load \wedge \text{prev } \neg reg\_load \Rightarrow \text{next}[11]\ init\_phase$
7	$init\_phase \wedge \text{prev } \neg init\_phase \Rightarrow \text{next\_a}[1..527]\ reg\_load$
8	$init\_phase \wedge \text{prev } \neg init\_phase \Rightarrow \text{next}[528]\ run\_phase$

Table 5.4: Second Set of Properties Verified

Property	Formula
9	$reg\_load \Rightarrow ce \wedge \neg d\_valid \wedge load \wedge \neg init$
10	$init\_phase \Rightarrow \neg load \wedge init$
11	$init\_phase \wedge \text{prev } \neg init\_phase \Rightarrow d\_valid \wedge \neg ce$
12	$init\_phase \wedge d\_valid \Rightarrow \text{next\_a}[1..10]\ \neg ce$
13	$init\_phase \wedge d\_valid \Rightarrow \text{next}[11]\ ce$
14	$init\_phase \wedge d\_valid \Rightarrow \text{next\_a}[1..11]\ \neg d\_valid$
15	$init\_phase \wedge d\_valid \Rightarrow \text{next}[12]\ d\_valid$
16	$run\_phase \Rightarrow \neg load \wedge \neg init$
17	$run\_phase \wedge \text{prev } \neg run\_phase \Rightarrow d\_valid \wedge \neg ce$
18	$run\_phase \wedge d\_valid \Rightarrow \text{X } \neg d\_valid \wedge ce$
19	$run\_phase \wedge \neg d\_valid \Rightarrow d\_valid \wedge \neg\ ce$

Table 5.5: Third Set of Properties Verified

makes a transition to the state *run_phase* at the 529^{th} cycle (property 8) so there are exactly 528 cycles for the initialization of the cipher. The finite state machine remains in the state *run_phase* until keystream generation is fully completed. The number of cycles in each state matches with the description provided in Section 4.3.3

Table 5.5 depicts the third set of properties used to verify that the state machine generates the correct control signals (*d_valid*, *ce*, *load*, *init*) in each state. Throughout all cycles of the state *reg_load*, the state machine generates a chip-enable of '1', a data valid of '0', a *load* signal of '1', and a *init* signal of '0' (property 9) so that the linear feedback shift register loads itself one stage per cycle.

Throughout all cycles of the state *init_phase*, the state machine generates a *load* of '0' and an *init* of '1' (property 10) so that the feedback of the WG core is connected to the input of the linear feedback shift register. In the first cycle of the state *init_phase*, the state machine generates a data valid of '1' and a chip-enable of '0' (property 11) because there is valid data to be fed from the linear feedback shift register to the WG core and the linear feedback shift register cannot shift until it receives data back from the WG core. Since every valid data packet needs 12 cycles to travel from the WG core back (latency of 11 cycles) to the linear feedback shift register (latency of 1 cycle), the chip-enable is de-asserted for 11 cycles (property 12) then re-asserted at the 12^{th} cycle (property 13) so that the linear feedback shift register can shift the data packet into its first stage. Once the data valid signal is asserted, it is de-asserted for 12 cycles (property 14) because a data packet needs to travel through the WG core to the linear feedback shift register. The data valid signal is re-asserted on the 13^{th} cycle (property 15) because another valid data packet in the linear feedback shift register is ready to be fed to the WG core again.

Throughout all cycles of the state *run_phase*, the state machine generates a *load* of '0' and an *init* of '0' (property 16) so that the input of the linear feedback shift register becomes the original feedback polynomial. In the first cycle of the state *run_phase*, the state machine generates a data valid of '1' and a chip-enable of '0' (property 17) because there is valid data to be fed to the WG core and the output bit of the linear feedback shift register has not arrived to its own input yet(due to the pipelined multiplier in the feedback polynomial). During the state *run_phase*, both the data valid and chip-enable signals toggle between '0' and '1' every cycle (property 18 and 19) because the WG can only handle a data rate of one data packet per every two clock cycles.

All of the properties mentioned above were satisfied by our optimized implementation of WG. Therefore, the verification of our optimized WG implementation was completed and no bugs were found.

5.4 Related Work

Since ciphers were invented to encrypt information for high security applications, most of the research in the area of cryptography has been focused on breaching the security of ciphers. In the past, various formal verification techniques have been used to verify security protocols. However, there is not much research which involves the hardware verification for cryptography.

In 2008, Slobodová published the first work [16] which is related to the verification of hardware for cryptography. Their work verified the Advanced Encryption Standard (AES) by using Symbolic Trajectory Evaluation.

In this chapter, we began the verification of our optimized WG implementation in Section 5.1 by verifying the WG core (i.e. datapath) with completion functions and equivalence checking under the assumption that the data valid signals were generated correctly. In Section 5.2, we verified that the linear feedback shift register with chip-enable is equivalent to the original linear feedback shift register under the assumption that the chip-enable signal was generated correctly. To complete the verification of our optimized WG implementation, we used model checking to confirm that the finite state machine did generate the control signals correctly. The next chapter provides the conclusions of this thesis as well as future work.

Chapter 6

Conclusions and Future Work

In this thesis, we explored the verification technique of completion functions by the verification of two ciphers: KASUMI and WG.

In Chapter 3, we first designed a non-pipelined implementation of KASUMI then we used RTL simulations to verify its correctness. With the optimization technique of pipelining, we created three additional implementations of KASUMI: 8-stage, 16-stage, and 32-stage. We verified these three pipelined implementations by using completion functions and equivalence checking, and their correctness was defined with respect to the non-pipelined specification.

During the verification of KASUMI, we developed a methodology to handle the completion functions efficiently in VHDL. This methodology use the VHDL "if-generate" and generic parameters to control the generation of the completion functions in hardware. The *stage* generic parameter specifies which stage is under verification, and the *spec* parameter indicate whether the specification or the implementation is synthesized. This methodology offers two advantages. First, it avoids the cumbersome process of connecting the completion functions to the register buried inside VHDL sub-entities. Second, a hardware designer or a verification engineer can instantiate the completion functions of a component simply by specifying the values of the generic parameters. The drawback of this methodology is that it consumes the time of hardware designer to build these completion functions.

In the case study of the WG cipher, we aimed to explore the completion functions with more sophisticated circuits. Similar to the case study of KASUMI, we first designed a non-pipelined implementation of WG then we used RTL simulations to verify its correctness in Chapter 4. With the optimization technique of pipelining and hardware re-use, we created an optimized implementation of WG.

In the verification of WG in Chapter 5, we developed two verification methodologies: skipping and completion functions dealing with hardware re-use. The methodology of skipping makes use of the fact that the verification of the internal stages of a verified sub-component can be skipped during the verification of a block which uses this sub-component. This methodology offers the advantage of decreasing the number of verification obligations in proving the correctness of a circuit.

The optimization technique of hardware re-use requires multiplexers to be added at the input of the re-used circuitry. In using completion functions and equivalence checking to verify a circuit optimized with hardware re-use, the completion functions or the specification does not provide any description about these multiplexers because they are implementation-specific details. By making assumptions about the select signals of the multiplexers, equivalence checking can prove that the implementation has the same functionality as the specification. These assumptions need to be verified in order to have a complete proof of correctness. In the case study of WG, we used model checking to verify that the finite state machine generates the select signals of the multiplexers correctly. From the results in this thesis, we derived research topics for future work and they are described below.

Completion Functions in Verilog: similar to programming languages, there exists many different hardware description languages with different features. In the industry, the two most widely-used hardware description languages are: VHDL and Verilog. In this thesis, we developed a methodology to handle completion functions based on some VHDL features such as "if-generate" and generic parameters. For a methodology to be widely-used, it has to support both VHDL and Verilog. Therefore, there is a need to explore completion function in Verilog.

Automatic Use of Completion Functions: in a report, it is estimated that between 40 to 70 percent of the total development effort is consumed by verification tasks [15]. In this thesis, we verified the correctness of both pipelines in a systematic way. The verification begins at the last pipeline stage and it ends at the first pipeline stage. For each pipeline stage, we first build its completion function then we use this completion function to verify the correctness of that pipeline stage. Other than building the completion functions of each pipeline stage, most steps in the verification can be automated. To decrease verification effort and time, heuristics should be developed to automate the use of completion functions.

References

[1] *3GPP TS 35.202: "3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 2: KASUMI Specification"*. 13, 22

[2] *3GPP TS 35.203: "3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 3: Implementor's Test Data"*. 22, 23

[3] Mark Aagaard, Vlad C. Ciubotariu, Jason T. Higgins, and Farzad Khalvati. Combining equivalence verification and completion functions. In *FMCAD*, pages 98–112, 2004. 11

[4] Steve Babbage and Matthew Dodd. The stream cipher mickey-128 2.0. available at http://www.ecrypt.eu.org/stream/p3ciphers/mickey/mickey128_p3.pdf. 66

[5] Sergey Berezin, Edmund M. Clarke, Armin Biere, and Yunshan Zhu. Verification of out-of-order processor designs using model checking and a light-weight completion function. *Formal Methods in System Design*, 20(2):159–186, 2002. 11

[6] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. pages 68–80. Springer-Verlag, 1994. 4

[7] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In *ISC*, pages 171–186, 2006. 66

[8] Kris Gaj, Gabriel Southern, and Ramakrishna Bachimanchi. Comparison of hardware performance of selected phase ii estream candidates. THE STATE

OF THE ART OF STREAM CIPHERS-SASC 2007, Ruhr University Bochum, Germany January 31 - February 1, 2007. 67

[9] Martin Hell, Thomas Johansson, and Willi Meier. Grain: a stream cipher for constrained environments. *IJWMC*, 2(1):86–93, 2007. 66

[10] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. Formal verification of a complex pipelined processor. *Form. Methods Syst. Des.*, 23(2):171–213, 2003. 10

[11] Paris Kitsos, Michalis D. Galanis, and Odysseas G. Koufopavlou. An fpga implementation of the gprs encryption algorithm 3 (gea3). *Journal of Circuits, Systems, and Computers*, 14(2):217–232, 2005. 26, 27

[12] Thomas Kropf. *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. 1

[13] Yassir Nawaz. *Design of Stream Ciphers and Cryptographic Properties of Nonlinear Functions.* PhD thesis, University of Waterloo, 2007. 17, 18

[14] Yassir Nawaz and Guang Gong. The wg stream cipher. 16, 17, 49

[15] Prakash Rashinkar, Peter Paterson, and Leena Singh. *System-on-a-chip verification: methodology and techniques.* Kluwer Academic Publishers, Norwell, MA, USA, 2000. 1, 91

[16] Anna Slobodová. Formal verification of hardware support for advanced encryption standard. 2008. 88

[17] Berk Sunar and Çetin Kaya Koç. An efficient optimal normal basis type ii multiplier. *IEEE Trans. Computers*, 50(1):83–87, 2001. 44

[18] Miroslav N. Velev. Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In *DATE*, pages 28–35, 2002. 11