Masterarbeit

Design and Implementation of a Secure Network Adapter for an Network-On-Chip

Georg Wagner, BSc

Institut für Technische Informatik
Technische Universität Graz
Vorstand: Univ.-Prof. Dipl.-Inform. Dr. techn. Kay Römer

Begutachter:	Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger
Betreuer:	Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger
	Dipl.-Ing. Dr. techn. Manuel Menghin, BSc

Graz, im September 2014

Kurzfassung

Die heutige Technologie erlaubt es bereits sehr kleine Strukturen auf einem IC zu erstellen. Gleichzeitig mit den immer kleiner werdenden Strukturen steigt natürlich auch die Anzahl der Komponenten die ein fertiger IC besitzen kann. Je mehr Komponenten ein IC besitzt umso mehr stellt sich die Frage wie die einzelnen Komponenten effizient miteinander kommunizieren können.

Eine Möglichkeit hingegen bietet ein Network-On-Chip (NoC). Dieses kann das Problem lösen indem es Datenpakete von einer Node im Netzwerk zur nächsten weiter schickt. Dadurch können alle Komponenten die mit dem Netzwerk verbunden sind gleichzeitig Daten senden und empfangen. Dadurch können Daten schneller ans Ziel gelangen und durch den Aufbau des NoCs wird auch weniger Platz auf dem IC verbraucht.

Allerdings haben NoCs noch einige Probleme. So werden NoCs meist ohne jegliche Sicherheitsmechanismen konzipiert, denn ein NoC soll nur sehr wenig Platz am Silizium verbrauchen. Fehler können durch diverse Umwelteinflüsse, z.B. kosmische Strahlung, hochenergetisches Licht, etc. hervorgerufen werden, aber gezielte Attacken auf das NoC können nicht ausgeschlossen werden. Dies wirft natürlich Fragen bezüglich der Sicherheit eines NoC auf und warum zumindest keine rudimentären Gegenmaßnahmen implementiert wurden.

Diese Arbeit beschäftigt sich mit der Thematik wie man ein NoC sicher gegen Fehler von außen machen kann und zeigt mögliche Sicherheitsmaßnahmen für ein solches. Dabei werden verschiedene Sicherheitsmechanismen wie Parity Bits oder CRC auf ihre Tauglichkeit sowie ihre Geschwindigkeit in einem NoC anhand einer für ein NoC angepassten Applikation überprüft. Dazu wird der von Schelle et. al. [SG04] entwickelte Network-on-Chip Emulator (NoCem) verwendet.

Abstract

Today's technology allows the creation of very small structures on an integrated circuit (IC). Simultaneously to the structures getting smaller naturally the number of components a final IC has increases. The more components an IC has the more the question has to be asked how they can communicate efficiently with each other.

A Network-On-Chip (NoC) can solve this problem by sending data packets from one node in the network to the next one. Thus all components that are connected to the network can send and receive data simultaneously and the data can reach its destination faster and because of the structure of a NoC less space on the IC is used.

However do NoCs have several problems. A NoC is usually designed without any security measurements, because a NoC shall takes as few space as possible on the silicon. Errors can be introduced by environmental influences, e.g. cosmic rays, high energetic light, etc. besides direct attacks on the NoC can not be excluded. This prompts questions regarding the security of a NoC and why at least no rudimentary counter measures are implemented.

This thesis focuses on the main topic how to secure a NoC against faults introduced from the outside and shows possible counter measures on such. Thereby different techniques like parity bits or CRC are tested for usability as well as speed in a NoC, based on a fitted application for NoCs. Therefore the Network-on-Chip emulator (NoCem), which was developed by Schelle et. al. [SG04], is used.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....	....
date	(signature)

Danksagung

Diese Diplomarbeit wurde am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Recht herzlich bedanke ich mich bei meinem Betreuer und langjährigen Schulkollegen Herrn, Bakk.-Techn. Dipl.-Ing. Dr. Techn. Johannes Grinschgl für seinen Vorschlag dieses Themas.

Mein besonderer Dank gilt auch meinem weiteren Betreuer Herrn Dipl.-Ing. Dr. techn. Manuel Menghin, BSc für seine Geduld, die er im Laufe dieser Arbeit aufbrachte.

Des weiteren bedanke ich mich auch bei meinem Gutachter Herrn Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger.

Ich bedanke mich auch bei allen Personen am ITI für die vielen guten Ideen und konstruktiven Diskussionen.

An dieser Stelle bedanke ich mich auch recht herzlich bei meinen Eltern, meinem Bruder, sowie meinen Großeltern, dass sie mich all die Jahre sehr tatkräfig unterstützt haben und ohne die ich es nicht so weit gebracht hätte.

Mein besonderer Dank gilt auch Herrn Prof. Mag. Helmut Schmid für das Korrekturlesen und den guten Input für die Arbeit.

Zu guter Letzt bleiben noch meine Freunde, Studienkollegen und Arbeitskollegen, bei denen ich mich dafür bedanke, dass sie meist ein offenes Ohr sowie den ein oder anderen Ratschlag hatten.

Graz, im September 2014 Name des Diplomanden

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Todays technology allows the creation of very small structures in an integrated circuit. This makes it easier to pack more components or even while systems on a chip (SoC).

Although increasing the number of components in an IC arises several problems. Especially when it comes to the communication between components. The simplest solution for this problem would be to connect each component with each other. Resulting in a very high amount of connections, depending on the number of components of course. Thus this solution may not be suitable for that many connections.

Another solution would be to use a bus based system to connect the components with each other. Bus based systems tend to have another downside. Now that the components are not connected to each other directly, the bus has to manage which component is allowed to send data and which components have to wait until the data is read from the bus, meaning the more data has to be send on the bus the slower it may get.

A Network-on-Chip (NoC) on the other hand can solve the problems that direct connections or bus systems arise. A NoC connects only nodes with each other. So one node would only be connected to four other nodes, resulting in a network where all components are connected to each other. Very much like a computer network a NoC would route packets of data from one node to the next until the packet has reached its destination. This allows the components to send as much data as the want, while having the benefit of not having direct connection with all the components.

Figure 1.1 shows the transition from connecting all components via wires to a NoC. It can be seen that the NoC needs only as many connections as there are components, while the wire only connections need much more wires. The wires only connection will always grow by the formula shown in 1.1, where W is the number of wires needed to connect n components with each other. On the other hand a NoC will have fewer connections which can be seen by formula 1.2. For a prove of this equation refer to appendix A.

$$W = \frac{n \cdot (n-1)}{2} \tag{1.1}$$

$$W = 2 \cdot n \cdot m - n - m \tag{1.2}$$

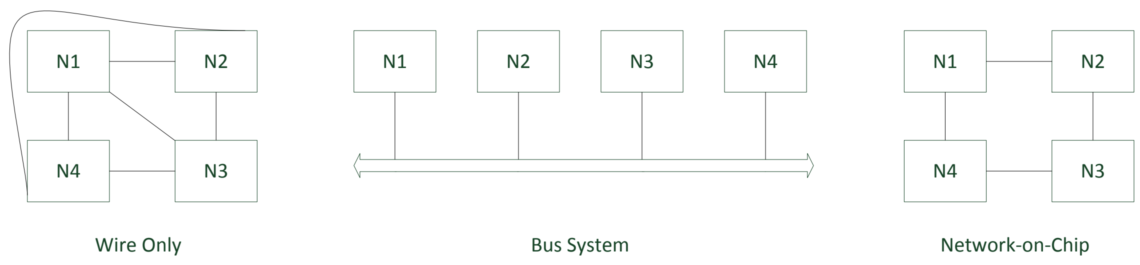


Figure 1.1: From only wires to a Network-on-Chip

As a NoC is a general purpose network it can be used for a variety of tasks. It can be used to connect different top level modules with each other and either reduce the wires that have to be routed and thus probably the size of the used silicone, or replace simple bus systems and gain more speed on the connection between different nodes on the bus.

Another example would be to use an NoC together with an algorithm that can be pipelined as pipeline. Each node of the NoC then acts as a stage of the pipeline and the connected modules are different stages of the algorithm that can be processed.

Probably another advantage of the NoC is its network adapter or access point. The access point works as an interface between the NoC and the connected module(s) and takes care of the data leaving the module and going into the NoC and also the data that is received by the NoC and going back to the module. Therefore it uses internal buffers and the module itself does not take care if the NoC has available capacities and can process the data or not. This also makes the module itself a slightly smaller and the evaluation can probably be done in less time than normal.

Using an NoC in a design may have many advantages but also a few disadvantages. Current implementations of different NoCs do not provide any security features. Packets that are routed to the wrong node may cause severe damage because a secret key was retrieved or an application fails.

Only little research has been done in this security area. NoCs are a relatively new design paradigm and were introduced by Dally et al. in [DT01] in the year 2002. Only a few papers have been published that tackle the possibility of faults in an NoC and what a fault may cause and mostly they focus on the NoC alone [FKCC06], [CDBZ99].

1.2 Goal

As written in section 1.1, not that much research on how to secure an NoC against fault induced attacks has been done yet. Many possibilities to secure the NoC lie in the access point itself where the data has not been corrupted yet.

Therefore the following points are defined:

- Evaluate some freely available NoC implementations
- Design and implement secure access points for an NoC
- Create an NoC specific fault injection framework
- Evaluate the secure access points for size, speed and reliability

The first evaluation phase should choose different NoC implementations that are freely available and see whether they are suitable for this thesis. The evaluation should focus on different parameters like size on the FPGA, documentation, readability of the code as well as the possibilities to extend the existing code. As a result of this point a selection of the NoC that will be used in this thesis is made.

The second point focuses on the design and implementation of an access point for the NoC. Several countermeasures will be implemented, the design must be very modular to insert or replace counter measures to have the same code base for the later evaluation.

The creation of the NoC specific fault injection framework focuses on the hardware as well as on the firmware. The modular fault injector introduced by Grinschgl et al. [GKS$^+$11] should be placed inside the hardware design as well as the control of sending and receiving data to and from the NoC. On the firmware side a library to control these actions has to be implemented.

Last but not least the secure access points have to be evaluated for size, speed and reliability. The NoC itself is a general purpose NoC and does not have a specific application running on top of it out of the box. As written in section 1.1 it is possible to use the NoC for pipelining an algorithm. As a possible use case for this the Advanced Encryption standard is chosen, because of its different rounds that can be run on different nodes of the NoC.

1.3 Structure

This thesis has the following structure: The chapter related work describes the work already done by other persons on which this thesis builds up. This is followed by a design chapter where the basic design of the implementation and tools is show. The implementation chapter shows details about the implementation. The design and implementation leads to the results chapters where the results of this thesis are shown. Last but not least a conclusion and future work chapter gives a short overview about this thesis and describes work that can still be done.

Chapter 2

Related Work

2.1 Network-on-Chip

In recent years a new design paradigm has become very popular. Replacing global wiring and simple bus systems in integrated circuits with so called Network-on-Chips (NoC). An NoC works very similarly to a normal computer network. While a bus can only have one sender and multiple receivers an NoC can have multiple senders and multiple receivers.

The first proposal of a NoC design paradigm was made by Dally et al. in 2001 [DT01]. Their network consists of four ports called North, South, East and West, together with a lightweight description of a datagram.

Each NoC consists of several building blocks. In today's literature the following components can be found:

- Network Adapter - implementation of the interface by which cores are connected to the NoC
- Routing Node - implement the routing strategy
- Link - physical connection that connects the nodes and provides the bandwidth

With these components it is possible to create a whole NoC [DT01], [BM06]

Bjeeregaard et al. mentions that the term NoC is used for a very broad sense ranging from gate level physical implementation, across system layout aspects and applications, to design methodologies and tools in [BM06]. This lies in the widespread adoption of network technology and abstracted models for networked communications. It is also possible to easily adapt the OSI model of layered network communication to an NoC as done by Benini and Micheli [BDM01]. Figure 2.1 shows this link to the OSI model of layered network communication.

Since the first paper on NoC in 2002 by Dally et al. [DT01] a wide variety of proposals and implementations has been made. Salminen et al. covers most of these proposals and implementations in [SKH] where he tries to compare the different proposals with each other, which proves to be a very difficult point, because basic properties of the NoC can vary very heavily from each other, but still Salminen et al. compared 44 different NoC proposals with each other.

While Dally et al. only propose a simple mesh network in their paper [DT01], many different layouts are possible.

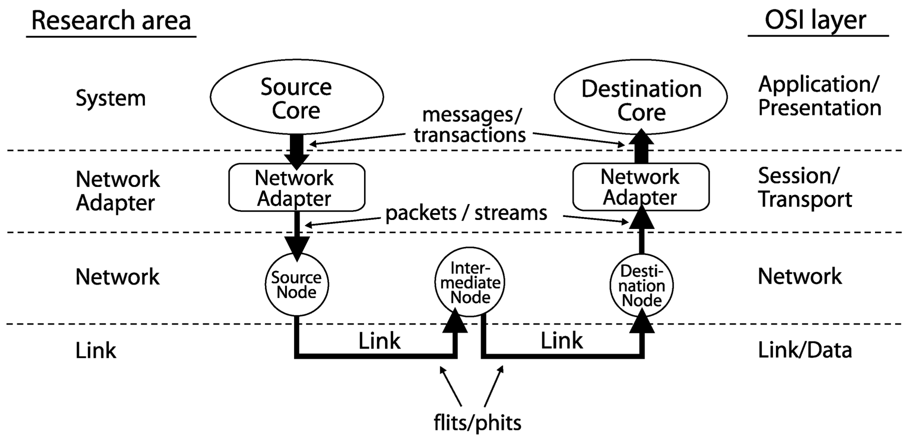


Figure 2.1: The flow of data from source to sink through the NoC components with an indication of types of datagrams and research area, [BM06]

2.1.1 Attacking a NoC

Sterpone et al. focuses on how to inject faults in a specific way to test the NoC's behaviour in [SSR12]. The approach used is very similar to the one in this thesis, although Sterpone et al. uses the net list and the FPGA configuration memory to apply different types of fault into the NoC. The results of Sterpone et al.'s work provide a new test method and identifiy areas in the NoC that are not secure, but do not give any solution on how to make them more secure.

On the other hand Frantz et al. focuses more on the crosstalk in a routers switch in [FKCC06]. Frantz et al. can either simulate Single Event Upsets (SEU) as described in [KCR06] or crosstalk, where Frantz et al. uses the Maximum Aggressor Fault model (MAF) as described in [CDBZ99]. As an approach to gain results Frantz et al. uses a golden model of the NoC and also a faulty model and checks for errors there.

Kang et al. proposes a different type of router model to overcome the problem of induced errors in an NoC [KKD10]. His network router adds CRC to different parts and therefore gets a very good error coverage.

Error Correcting Codes

Bertozzi et al. describes a very common design paradigm for fault tolerant circuits in [BBM05]. Basically the standard approach is to provide double or even triple redundancy in the circuit. Due to of the large overhead that this method bears it may not be useful in an area and power constricted environment.

Bertozzi et al. also states that Error Correcting Codes (ECC) may be useful as they can be carried out in either hardware or software. Still the circuit to correct the error may use more space on the silicone.

Therefore another solution, a two-rail code, can be used. According to [BBM05] the bus lines are doubled in this case and a complementary signal is transferred on the doubled lines. The packet layout proposed by Bertozzi et al. can be found in 2.2.

The packet layout seen in figure 2.2 is a layout that can be expected from a Hamming Weight code, which is also used in [BBM05]. Bertozzi et al. also describes a version

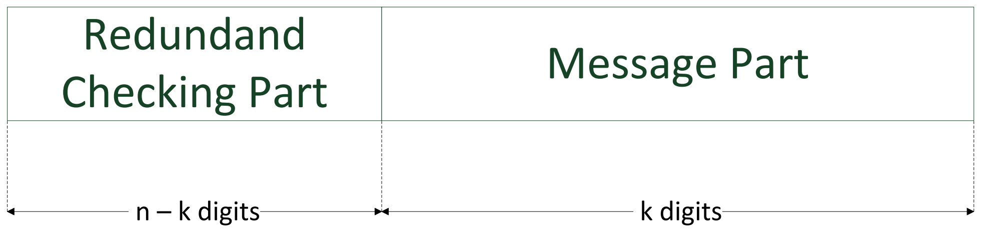


Figure 2.2: Packet layout proposed by [BBM05]

of the Hamming Weight decoder which consists of an EXOR-Tree and an optional error correcting stage [BBM05]. Figure 2.3 shows the Hamming Weight decoder.

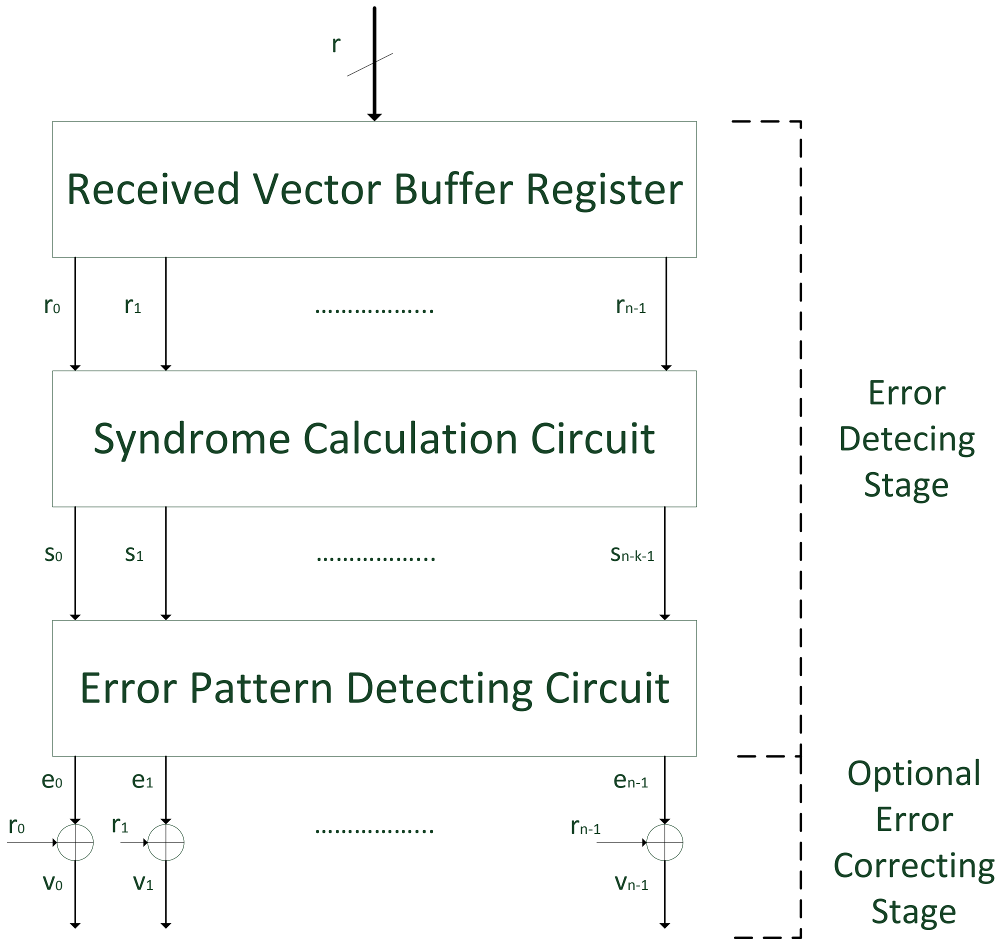

r
Received Vector Buffer Register
r0
r1
rn-1
Syndrome Calculation Circuit
s0
s1
sn-k-1
Error Pattern Detecting Circuit
e0
e1
en-1
r0
r1
rn-1
v0
v1
vn-1
Error Detecing Stage
Optional Error Correcting Stage

Figure 2.3: Hamming Weight decoder as proposed by [BBM05] with additional correction stage

Adaptive Routing Algorithms

Kohler et al. proposes adaptive routing algorithms to overcome the problem of a static fault in [KSR10]. Therefore the network is reconfigured if an error is found. This adaptive routing algorithm has some requirements. It must not create deadlocks and it must not create livelocks. Although network reconfiguration is very prone to those two effects, Jos et al. [DLPP05] and Lysne et al. [LPD05] provide a theory and a methodology that solves this problem.

The adaptive routing introduced by Kohler et al. in [KSR10] cannot be used when transient faults occur. According to [KSR10] the adaptive routing is not fast enough to adapt the routing before the error may disappear again.

In order to make the adaptive routing to work, Kohler et al. uses a few bits of the packet and replaces them by an 8 bit CRC. Figure 2.4 shows the original and the new packet. It is very clear to see that one packet is 128 bits big and the payload data is 96 bits. By adding the CRC their payload is reduced to only 88 bits while the CRC takes up 8 bits. Thus the CRC alone needs about 10% space of the original payload size.

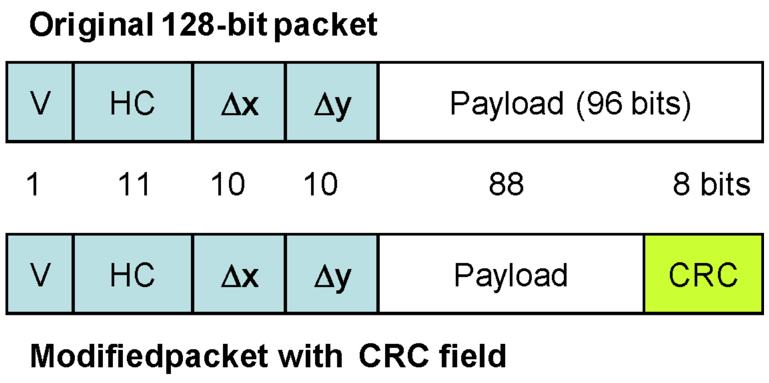


Figure 2.4: Original data packet and data packet with CRC, [KSR10]

2.2 Security Measures For the NoC and the Access Point

A wide variety of measures to determine that an error occurred during the execution does exist. It could be as easy as implementing the same circuit but in negative logic, or process the input n times and check the results. If they differ, a fault must have occurred and actions have to be taken.

These measures are good, but unfortunately not really suitable for this thesis or probably for an NoC anyway. They may blow up the whole design and also the area on the silicon. Hence an NoC should be as small as possible such countermeasures are not evaluated in this thesis. This just leaves another type of countermeasures like parity bits that can be used and evaluated using NoCs. The following sections describe the countermeasures in a bit more detail.

2.2.1 Security for the NoC

Dally et al. propose a very common way to protect the NoC against faults during the production by providing spare bits on network links and network buffers. After the test laser fuses are blown or configuration registers are set, so that these extra lines can be used or not [DT01].

Dally et al. also explain in [DT01] that this adds additional delay to the NoC because of the rewiring of wires and if the communication requires additional security, it should be moved into the access point.

2.2.2 XOR-Table (Parity Bits)

A single parity bit can be calculated by XOR-ing all the input data. This will lead to a result that is either logical „0“ or „1“. The problem that arises when using only one parity bit is that only one error can be detected. So if a bit is flipped, this can be detected. whereas two bit flips cannot be detected any more and the result my seem to be correct.

2.2.3 Hamming Codes

Hamming Codes were introduced by R.W. Hamming in [Ham50]. While a hamming code is very similar to a parity bit, it offers a few more advantages than a simple parity bit. Hamming Codes calculate multiple parity bits and place them in the middle of the data (at positions which are 2^n where n increased for each run). It makes them very effective to detect single bit flips, but also multiple bit flips can now be detected and single bit flips can be corrected.

2.2.4 Cyclic Redundancy Check (CRC)

Peterson and Weldon describe CRC in their book Error-correcting Codes [PW72] as well as Lin et al. in [LC04].

In general CRC is a non secure digest function over a data word. It handles the data via a mathematical operation over $GF(2)$ (polynomial). Then it performs a division by the creation polynomial $GF(x)$. The remainder of this division then is the final CRC code and can be used as input for the next run.

2.2.5 Reed-Solomon (RS) Codes

Based on Hamming's work in [Ham50] Irving Reed and Gus Solomon published their new work [RS60]. It is a non binary-cyclic code. The code itself is made up of sequences and creates blocks to code them.

2.2.6 Secure Hash Algorithms (SHA-3)

SHA-3 is the new secure hash algorithm that will be standardized by the NIST in FIPS-202 very soon [NIS]. The algorithm uses a sponge function to process the message step by step. Therefore a message can be very large. Only a small part of it is processed at each step until the whole message has been processed.

The absorbed data is then XOR-ed and used the first output is squeezed out. This procedure then is repeated with the rest of the data until the whole input is adsorbed. See [BDPA11] for more details.

2.3 Different Ways of Fault Injection

In order to make changes to the system, different ways of fault injection have to be found. In this thesis it is not possible to open the chip itself and analyse its internal structure to apply the attacks on the right spot. This means that a non-invasive fault injection system must be used.

Several ways are known to create faults. One would be the usage of the partial reconfiguration functions of an FPGA as described in [dARGG06], which limits the choice of FPGAs a lot. For example an FPGA from XILINX [1] could be used.

Another way of manipulating the circuit would be the installation of faulty or modified logic elements or elements that can be controlled from the outside. Logic elements that can be controlled from the outside would be mutators or saboteurs as used by Boue in [BPC98].

Ziade et al. describes possible ways of fault injections in [ZAV04]. Ziade et al. categorizes faults in two different groups: hardware/physical faults and software faults and their probable causes.

This thesis uses saboteurs to manipulate the hardware. If not activated, a saboteur does not have any effects and the circuit works as expected. Once they are activated they inject their signals directly into the hardware.

On the other hand mutators or mutants are modified versions of the original logic element. Mutants do not have any effects on the circuit if they are not activated and just sit there and wait. If they are activated, they replace the original logic element or module entirely.

2.4 Modular Fault Injector (MFI)

The used MFI was introduced by Grinschgl et al. in [GKS$^+$11]. Their design features the following points

- Fully modular fault injector design [GKS$^+$11]
- Multi-bit fault injection to support fault attack emulation [GKS$^+$11]
- Online-testing support [GKS$^+$11]

The whole MFI can be added very easily to the whole design. It uses a General Purpose Input/Output (GPIO) communication interface which can be easily included in a design and it also supports a big selection of architectures. Grinschgl et al. [GKS$^+$11] also provide a software library for XILINX [Xil] to easily set the pins of the GPIO and to configure/control the MFI.

[1]Xilinx Virtex IV `http://www.xilinx.com/support/index.html/content/xilinx/en/supportNav/silicon_devices/fpga/virtex-5.htm`, last visit January 2014

Further the MFI consists of the following modules:

- Fault Injection Controller (FIC)
- Saboteurs
- Fault pattern support

In Grinschgl et al. [GKS$^+$11] design the FIC is the main part of the MFI. Its main purpose is the control of the mode and the activation time of the connected saboteurs.

The saboteurs that are used can be classified as unidirectional serial simple saboteurs [GKS$^+$11]. Only one direction is possible and that the saboteur is directly connected to the signal it should affect.

The fault pattern support allows the programmer of the firmware to change the pattern of an attack. Therefore the location of the saboteurs on the silicon has to be known. Different patterns can then be applied [GKS$^+$11].

After a proper placement of all the saboteurs, triggers, the fault injector and the GPIO components it is easy to control the MFI from a firmware only. No needs to change the hardware if different types of errors need to be checked, which makes it a very versatile tool for verification.

2.5 Advanced Encryption Standard (AES)

AES is the successor of the Data Encryption Standard (DES). It was released by the National Institute for Standards and Technology (NIST) in October 2000 in the standard [AES01] paper.

A similar implementation was done by Yang et al. [YBLB09], although Yang et al. did not have a close look at the security issues that can occur by porting an algorithm over to an NoC.

In general the AES is a symmetric block cypher that can either have key sizes of 128, 192 or 256 bits while a block of data is always 128 bits and is based on the design principle known as substitution-permutation network.

The algorithm operates on a $4x4$ field, also called state array. Depending on the key size the state array is computed 10 times for a key size of 128 bits, 12 times for 198 bit keys and 14 times for 256 bit keys.

In detail the AES algorithm defined in [AES01] works as described in listing 2.1. The input array `in` contains the $4x4$ array of input data ($N_b = 4$). The encrypted output is saved in `out` which contains the $4x4$ array ($N_b = 4$). The input `w` contains the key schedule as described in section 5.2 in the [AES01] and contains the original key plus the keys for all the other N_r rounds. N_r equals the rounds the AES algorithm has. If a 128 bit key is used $N_r = 10$, for a 192 bit key $N_r = 12$ and for a 256 bit key $N_r = 14$.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
        byte state[4,Nb]

        state = in

        AddRoundKey(state, w[0, Nb−1])

        for round = 1 step 1 to Nr1
                SubBytes(state)
                ShiftRows(state)
                MixColumns(state)
                AddRoundKey(state, w[round*Nb, (round+1)*Nb−1])
        end for

        SubBytes(state)
        ShiftRows(state)
        AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb−1])

        out = state
end
```

Listing 2.1: Pseudocode for AES [AES01]

For all the above methods, except AddRoundKey which uses a different RCon-Table, an inverse method does exist to decipher the bit stream. Deciphering a complete text using the NoC is not covered in this thesis.

2.5.1 Attack Methods to Reconstruct the Secret Key from Use Case AES Implementation

Takahashi et al. proposes a very efficient way to retrieve the key from AES with known plain texts and faulty cypher texts in [TFYpt]. Therefore the assumption is made that random faults are injected into the 9th round key. It has to be considered that the faults are not injected into bytes of the same row in the 9th round. Additionally the attacker has to be able to obtain pairs of correct and faulty outputs from the same input.

Figure 2.5 shows how a fault induced into the 9th round in the second column into the bits one and two is distributed throughout the key scheduling process. The faulty bits affect also the other bits of the key in the same round.

Due to the definition of AES the fault is propagated to the 10th round. This is done in two different ways. The 10th round key is affected directly by the 9th round key and indirectly by the last row of the 9th round key as this is used to calculate a starting value for the 10th round.

The main approach of Takahashi et al.'s attack is to set the faulty state calculated by the faulty output equal to the correct state calculated by the correct output before the faulty keys were added into the round operation. Both states are correct at this point. An entire key can be retrieved one by one calculating the equations yielded by the correct faulty states [TFYpt].

Another way to attack AES would be a reduction of its runtime from ten to one round. Then the reconstruction of the secret key is very easy too. Bouillaguet et al. [BDD+10] describes the procedures in his paper „Low Data Complexity Attacks on AES“.

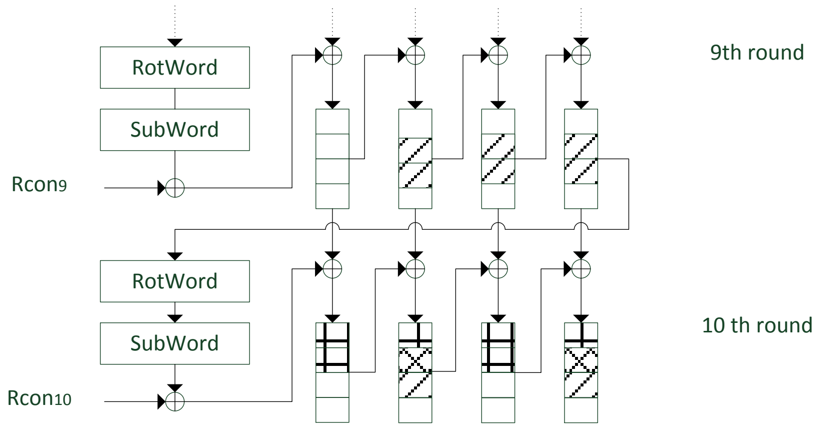


Figure 2.5: AES key scheduling process with fault induced into $K^9_{1,1}$ and $K^9_{1,2}$, [TFYpt]

The attack on an AES that is reduced to only one round is very simple. According to Bouillaguet et al. it only takes 2^{12} encryptions if some special features of the AES are taken into account. If they are not taken into account, the number of encryptions raises to 2^{16} [BDD+10]. Although on hardware that is available today it should be only a matter of seconds.

Bouillaguet et al. uses for the simplest attack two plain texts and calculates the cypher texts using the faulty algorithm. First apply $SR^{-1} \cdot MC^{-1}$ to the output difference. This leads to only the output differences of all S-boxes. The now calculated S-box differences are equal to the plain text difference in their respective bytes. Next calculate all 2^8 inputs that are able to create the input difference. Find the pairs that are suggesting the correct output difference. This should be two pairs for each Sbox. The last step is trail and error. Encrypt the input using the faulty algorithm and see which combination causes the correct output. In the worst case there are 2^{16} tries. For more information on how to improve the speed on this attack please refer to [BDD+10].

Chapter 3

Evaluation of existing NoC Implementations

3.1 Criteria for the Evaluation of existing Network-On-Chip Implementations

In the last years many proposals and implementations on NoC have been made. This has led to a wide variety of existing tools. It also makes it difficult to choose a sufficient implementation. Salmien et al. [SKDH08] gives an overview about NoC proposals. Salmien et al. examines about 60 different NoC proposals, which shows that there is already a wide variety of NoC proposals around. Only about 65% of the analysed proposals and implementations in [SKDH08] can be synthesized and used on real hardware.

In order to choose the right implementation for this thesis, two different approaches of an NoC implementation are evaluated, none of them is mentioned in [SKDH08]. The first implementation is the so called Atlas [MCM⁺] while the second one is the so called Network-On-Chip Emulator (NoCem) [SG04]. Both implementations are freely available and can be downloaded from the internet.

For this evaluation a Xilinx Virtex 5 series ml507 evaluation board is used. The board utilizes a Xilinx Virtex 5 FPGA. In the later part of this thesis the evaluation board is used to attack the NoC.

The following subsections show the evaluation criteria and give a short description where the focus of the criteria lies.

3.1.1 Features

Beside the creation of the NoC several other features can come with the tools used in this evaluation. This section describes the provided features in detail and if they can be used in this thesis or not.

3.1.2 Usability

This section describes the usability of the NoC. It describes how the NoC can be configured and how different settings can be achieved by using different settings.

3.1.3 Configuration Options

The configuration options are probably a very important point in this evaluation. They show which settings can be used for the NoC. It also shows the parameters of the settings that can be applied.

3.1.4 Routing Algorithms

As it may not be clear how the routing algorithms work, the following sections provide a short description on the routing algorithms.

XY-Routing

The XY routing algorithm is first described in [RLP06a]. It proposes that each router knows its position in the network, basically its X and Y coordinate. Depending on that coordinate the packet is sent to the right output port. For example a packet is injected into a $4x4$ mesh at the router with the position $(x, y) = (0, 0)$ and the destination of that packet is router $(x, y) = (3, 3)$. The packet is first sent into the X direction. The output port is the east port of the router. This is done until the X coordinate of the current router and the X coordinate of the destination router are the same. Now the packet traverses the network in Y direction. Therefore it is sent on to the north output port of the current router. This step is repeated until the Y coordinate of the router matches the Y coordinate of the packet. The packet has now received its destination router and can be extracted from the network. Figure 3.1 shows two examples for this algorithm.

To describe this algorithm in a more generic way: if the X coordinate of the current router is smaller than the X coordinate of the destination router, send the packet to the east port. If the X coordinate of the current router is bigger than the X coordinate of the destination router, send the packet to the west port. Do so until the X coordinate of the current router is equal to the X coordinate of the destination router. Now traverse the network in Y direction. The same algorithm is then executed to traverse the network in Y direction, except that if the Y coordinate of the current router is bigger than the Y coordinate of the destination router, the packet is sent to the north port of the router and if the Y coordinate is smaller, the packet is sent to the south port. Again, if the Y coordinates from the current router are equal to the one in the packet, the destination router is found and the packet can be extracted from the network.

West-first and West-first Non Minimal

The west-first routing algorithm is an adaptive routing algorithm [RLP06b] and is an extension of the XY routing algorithm. Basically it works in the same way as the XY routing algorithm with the exception that all packets have to go as far west as necessary. It is not allowed to route packets to the west after the first west step. Figure 3.2 shows the allowed routing directions for west-first routing.

Cypher and Gravano Routing (CG)

This algorithm is first introduced in [CG94] and is a routing algorithm used in torus networks. This algorithm is used within computer networks but can be ported to an NoC.

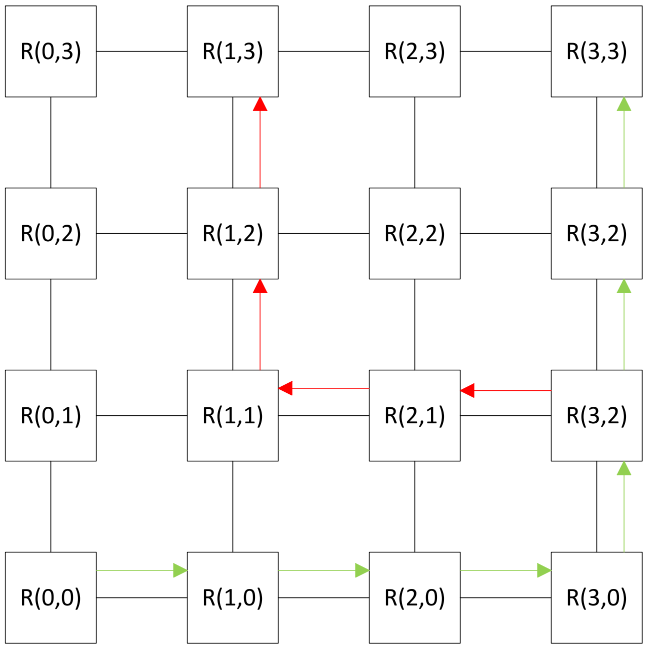


Figure 3.1: Example of the XY Routing algorithm. The green arrows show the path of a packet that is sent from R(0,0) to R(3,3) and the red arrows show the path of a packet sent from R(3,2) to R(1,3).

For the routing algorithm three queues are defined: an injection queue, a delivery queue and a standard queue. New packets can only be placed in empty injection queues in their source node and only removed from a delivery queue at the destination node. A standard queue can be accessed directly from all of the node's input ports [CG94].

The routing algorithm now specifies which movements between the queues are allowed. To do so a set of queues to which the packet may be moved can be specified. Cypher et al. call them a waiting set. Only if a queue is in a packets waiting set the packet is moved to that queue [CG94].

3.1.5 Test Bench

Within this point of the evaluation the existence and the completeness of a test bench should be evaluated. First a check for the existence of a test bench is done. If one exists, further evaluation is done.

For an existing test bench additional tests are done. First it is checked if it can be run very easily, or if it is hard to run the test bench. For example it could be possible to use some exotic Tcl/Tk libraries that have to be installed on the system.

After the test bench can be executed the completeness of the test bench is evaluated. Therefore, the test bench is run and the results of this run are evaluated.

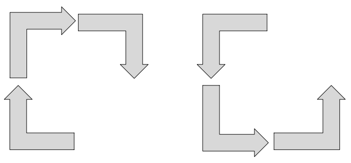


Figure 3.2: Allowed routing directions in west first routing

3.1.6 Connections to other IPs

At some point during development it may be useful to connect the NoC directly to a CPU. This allows the CPU to transfer data directly to the NoC without making any big detours. Although not used in this masters thesis this feature may be useful for other projects.

3.1.7 Size on the FPGA

An FPGA only offers a limited amount of logic gates. Therefore, it is necessary to find out how much space the NoC needs. This also leads to knowledge about the best configuration for the later fault injection experiments.

3.1.8 Readability Of the Source Code

This section has its focus on the readability of the source code. Some writers tend to write very hard to understand source code. The aim of this section is to evaluate the source code. Points like the readability of the code, naming of variables and the quality of the comments are in the close focus.

3.1.9 Extensibility

The extensibility of the NoC is very closely linked to the readability of the source code. An extension like a parity bit generator and checker can only be added if the source code is very well understandable.

3.2 Evaluation of the Atlas NoC Tool

The Atlas Tool is developed by Moares et al. and is available from [MCM$^+$]. The tool is first introduced in [MACM11]. It can be used for a wide variety of tasks like the generation of a Network-on-Chip (NoC), traffic generation for this NoC. The tool is also capable of doing a simulation and a power and performance analysis of the configured NoC.

Atlas is designed to support the designer of an NoC to quickly do power and performance analysis of the designed network. It doesn't provide features to test the NoC against faults.

3.2.1 Features

Atlas comes with a wide variety of features, although only the first two features are of interest for the master thesis. They are the NoC Generation feature and the Traffic

Generation Feature. In the thesis several Networks-on-Chip are created and fed with traffic from the Traffic Generator.

- NoC Generation
- Traffic Generation
- Performance Analysis
- Power Analysis

The NoC Generator can also configure more than one type of NoC. The configurable networks are:

- Hermes
- Hermes TB
- Hermes TU
- Hermes SR
- Hermes CRC
- Mercury

The Hermes NoC is first introduced in [MCM$^+$04]. It is a router that consists of five ports (North, South, East, West and an Local Port) and uses basically an XY routing algorithm with wormhole switching. Wormhole switching is defined in a way that a packet is split into smaller flit that are routed through the network. When a flit head flit arrives the router routes it and reserves the resource for all other flits until a tail flit arrives. In that way it looks like a worm that is travelling through the network.

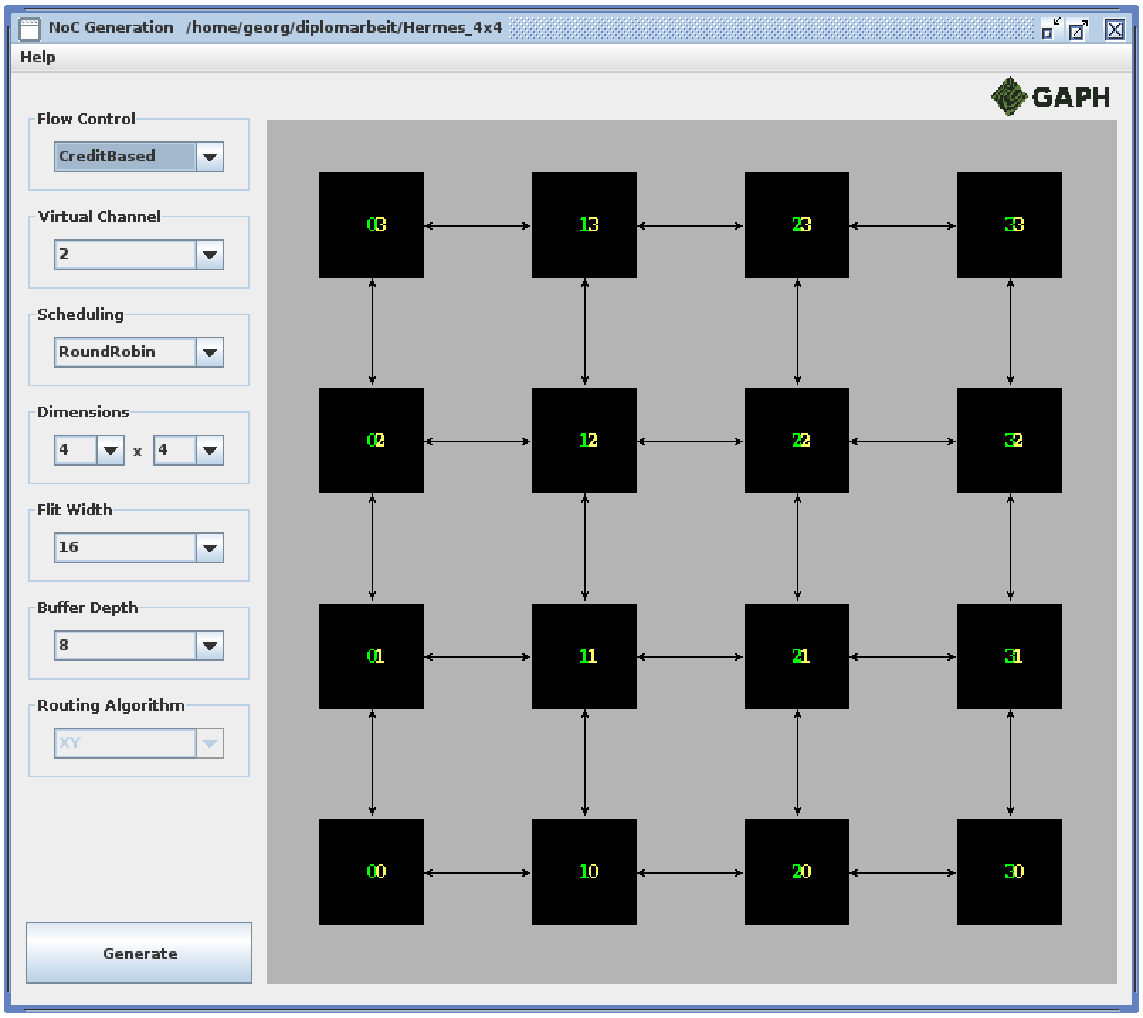
With Atlas it is also possible to create saboteurs. This can only be done when a Hermes CRC network is created and then it uses the Maximum Aggressor Fault Model (MAF) to simulate errors on the connections between routers, [CDBZ99].

3.2.2 Usability

The usability of the Atlas tool is very good. It guides the user through the process of configuring the NoC with a Graphical User Interface (GUI) and supports the user when it comes to creating the traffic. Finally the tool can also support the user when it comes to power and performance analysis, but this is not a point of the thesis.

The NoC Generation is where all parameters for the NoC can be set. Changes in the layout of the network, for example the change from the standard 3x3 network to a 4x4 network are also shown to the user so that he can imagine the look of the network even better. Figure 3.3 shows an example of the GUI.

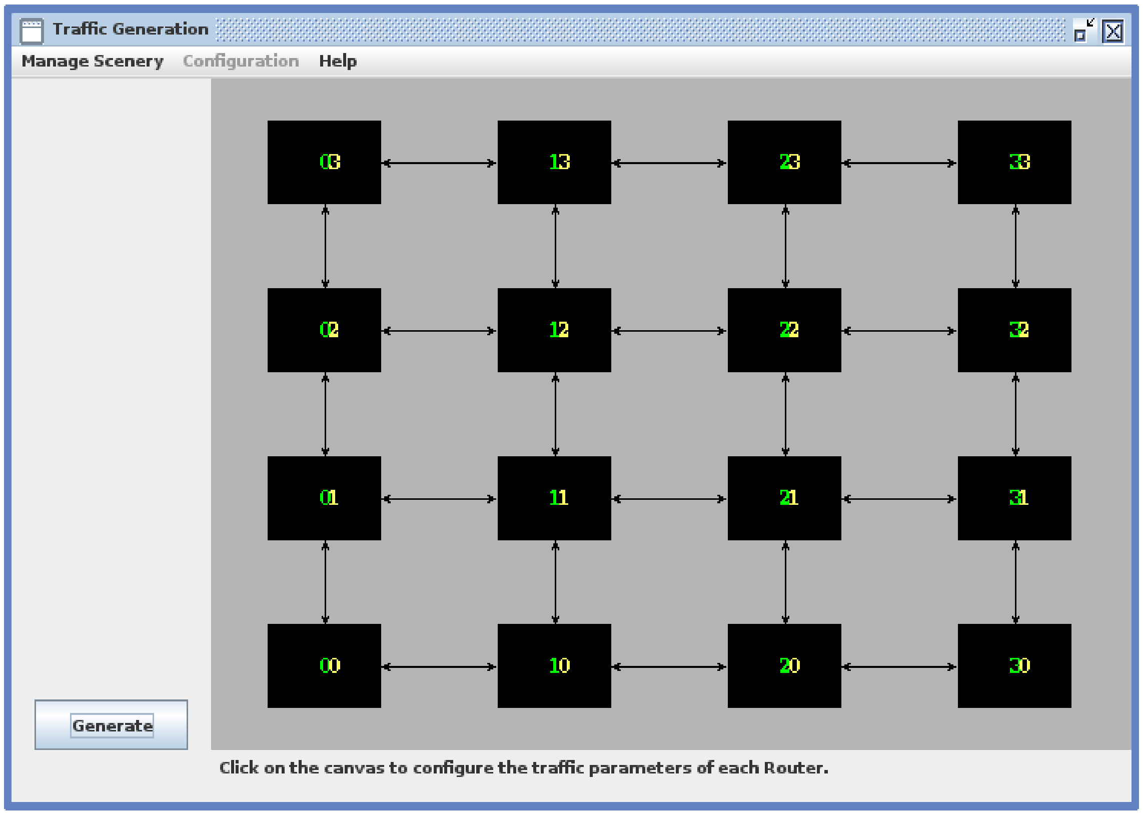
The traffic generation is quite simple as well. The user can choose between the types of traffic he wants to create and can set a lot of parameters to create the traffic. It is possible to create the traffic for each router individually. At the end of this process the traffic is generated and can be used in the test bench. Figure 3.4 shows an example of the GUI that is used to create the traffic and 3.5 shows the settings for an individual router.


NoC Generation /home/georg/diplomarbeit/Hermes_4x4

Flow Control: CreditBased
Virtual Channel: 2
Scheduling: RoundRobin
Dimensions: 4 x 4
Flit Width: 16
Buffer Depth: 8
Routing Algorithm: XY

Figure 3.3: Example of the configuration of an NoC

Unfortunately Atlas does not make any requests on the size of the NoC that should be created. It doesn't check if the NoC could be too big to fit on a certain Field Gate Array. This has to be found out by synthesis of the NoC and may lead to long times running the synthesis just to find out if the NoC fits.



Click on the canvas to configure the traffic parameters of each Router.

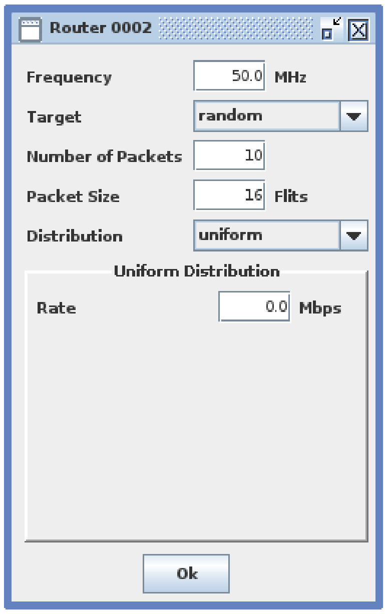
Figure 3.4: Example of the traffic generation screen.

Another downside of Atlas is that most of the source code is documented and some of the variables used in the program code are in portuguese language. This leads to the problem that the source code cannot be understood as easily as intended by the writers of Atlas. Also the fact that some variables are named in Portuguese makes it hard to read and understand the code.

Although Atlas provides a save feature for the NoC configured by the user it seems that Atlas doesn't read the settings back in a correct way. Whenever the GUI is opened in order to alter settings in the NoC Generation tool then the tool will display its default settings and not the saved settings. This seems to be a bug in the GUI as the traffic generation shows the network in the way it was configured.

3.2.3 Configuration Options

Atlas provides a lot of configuration options for the creation of an NoC. A summary of all the parameters that can be configured for an NoC can be found in Table 3.1.



Frequency 50.0 MHz
Target random
Number of Packets 10
Packet Size 16 Flits
Distribution uniform
Uniform Distribution
Rate 0.0 Mbps

Figure 3.5: Example of an individual router traffic generation setup.

Parameters	Hermes	Hermes TB	Hermes TU	Hermes SR	Hermes CRC	Mercury
Topology	2D Mesh	2D Torus	1D Torus	2D Mesh	2D Mesh	2D Mesh
Virtual Channels	1, 2 or 4	1	2	1 or 4	2	1
Flit Width	8, 16, 32 or 64			16	16	8, 16, 32 or 64
Buffer Depth	4, 8, 16 or 32					
Routing Algorithm	XY or West-first	West-first non minimal	XY	XY	XY	CG
Scheduling Algorithm	Round Robin, Priority	Round Robin	Round Robin	Age Based	Round Robin	Round Robin
CRC	No	No	No	No	Yes	No
QoS	Yes	No	No	Yes	No	No

Table 3.1: Configuration options for the NoC Generation [MCM+]

3.2.4 Test Bench

Atlas provides a large test bench. The test bench is written in SystemC and is used to test the network with the provided traffic from the tool. The simulation can also be started from the Atlas tool itself which opens up a small window with a progress bar. Nothing else can be seen.

Unfortunately it was not possible to run the test bench because ModelSim is not able to compile SystemC code as the destination system should be a RedHat Linux distribution and the test system uses a SuSe Linux distribution.

3.2.5 Routing Algorithms

Atlas provides several routing algorithms. The routing algorithms provided by Atlas are

- XY
- West-first
- West-first non minimal
- Unidirectional XY
- CG

A detailed description on how the routing algorithms work can be found in section 3.1.4.

3.2.6 Connections To Other IPs

The Atlas Tool doesn't provide any common interfaces to other IP cores. So the connections to a Processor Local Bus (PLB) for example have to be written if the connection to a micro processor is desired for example.

3.2.7 Size on the FPGA

As the NoC Generator can create a variety of networks with a lot of varying parameters the size on an FPGA can vary very much. In the test synthesis that is done it could be seen that a network with only four routers (2x2 Mesh) could fit very easily in the biggest configuration on the FPGA while a 4x4 Mesh is very hard to fit onto the FPGA and depends on the configuration.

3.2.8 Readability of the Source Code

The source code is very well structured. Each module has its own file and only covers its functionality.

A problem that occurs in the source code is the language. The comments in the source as well as some variables are kept in portuguese language. This makes it very hard to understand the source code. Probably replacing those comments and variables into English may help.

3.2.9 Extensibility

It is very hard to tell how good the extensibility of atlas and the generated NoCs are. This is basically because of the language they used to write the source code in. The code is written in English and Portuguese.

The Java part of the tool can be easily extended. It is object oriented and doesn't use any foreign language in the code. Besides it is well structured and can be understood very easily.

The VHDL code is harder to understand. Some parts of it are written in a foreign language, which makes it difficult to understand which program blocks or variables are used for. Besides the language gap other problems arise.

The VHDL code is generated from the Java part. It creates all the necessary connections between the routers. A later change of the connections is only possible if the whole network is generated by the tool again.

Also a change of the behaviour of some routers or parts of a router can probably only be done if some big changes in the Java part are made. Although the source code is very clean structured a single router can probably not be exchanged so easily. Also depending on the position of the router in the network several files have to be rewritten to make this possible.

3.3 Evaluation of the Network-On-Chip Emulator (NoCem)

NoCem was first introduced by Schelle et al. in [SG04]. NoCem only provides a configuration file and is very easy to configure once the different options are understood. NoCem provides different configuration options. It is possible to add virtual channels (VC), create different layouts like mesh, 2D- or 3D- torus, etc.

3.3.1 Features

Compared to Atlas, NoCem provides hardly any features. NoCem comes with a basic configuration and test bench and can be configured by the user. The configuration file provides a wide variety of configuration options. Although no configurations have names, it is still possible to create nearly as many different configurations as Atlas.

3.3.2 Usability

The usability of NoCem is very good. Although NoCem does not provide a GUI to configure the network, it provides configuration options that can be changed with a simple text editor. This has the advantage that NoCem can be reconfigured very fast and because the files for all different types of the network are included it can be synthesized without copying any files or write any additional source code. An example for the configuration of NoCem can be seen in listing 3.1.

```
constant NOCEM_TYPE                  : integer := NOCEM_CHFIFO_VC_TYPE;
constant NOCEM_CHFIFO_TYPE           : integer := NOCEM_CHFIFO_VC_TYPE;
constant NOCEM_TOPOLOGY_TYPE         : integer := NOCEM_TOPOLOGY_MESH;
constant NOCEM_FIFO_IMPLEMENTATION   : integer := NOCEM_FIFO_LUT_TYPE;

constant NOCEM_NUM_AP                : integer := 4;
constant NOCEM_NUM_COLS              : integer := 2;
constant NOCEM_NUM_ROWS              : integer := NOCEM_NUM_AP / NOCEM_NUM_COLS;

constant NOCEM_DW                    : integer := 8;
constant NOCEM_AW                    : integer := 2;


constant NOCEM_NUM_VC                : integer := 2;
constant NOCEM_VC_ID_WIDTH           : integer := NOCEM_NUM_VC;

constant NOCEM_CHFIFO_DEPTH          : integer := 4;
constant NOCEM_MAX_PACKET_LENGTH     : integer := 8;
```

Listing 3.1: Configuration example of NoCem.

3.3.3 Configuration Options

NoCem provides a wide variety of configuration options. The configuration options are a little bit different if a microprocessor is used or not. A summary of the configuration parameters can be found in table 3.2

Parameters	without microprocessor	with microprocessor
Topology	Mesh, Torus, double Torus	
Grid Configuration	Rectangle or Square	
NoC dataword size	1-256 bits	
Packet Control word size	1-256bits	
Packet Length	2, 4, 8, 16 datawords	
Virtual Channels	2 or 4	
Channel FIFO length	2, 4, 8, 16	
Routing Algorithm	XY	
Scheduling Algorithm	Round Robin	
CRC	No	
QoS	No	
Microprocessor Datasize	-	4 bytes
Peripheral Bus	-	OPB, PLB
Processor Type	-	Microblaze, PPC

Table 3.2: Configuration options for the NoC Generation or NoCem

3.3.4 Test Bench

NoCem does provide a test bench in VHDL. This test bench is very small and only instantiates the NoC and drives the clock for the simulation of the NoC.

The test bench also provides so called exercisers which can create data for the simulation process. Although very well designed, those exercisers are kind of hard to understand and due to the inflexibility of VHDL it is even better to write a test bench that is more flexible.

3.3.5 Routing Algorithms

Although not really stated in any of the documents regarding NoCem the assumption can be made that it uses an XY routing algorithm. This assumption can be confirmed by a simple simulation. No other routing algorithm is implemented in this version of NoCem.

3.3.6 Connections to other IPs

NoCem already provides two different connection IPs. One is an On-Chip Peripheral Bus (OPB) and the other one is a Processor Local Bus (PLB). They can be used to connect to either a Microblaze or a Power Pc (PPC) softcore CPU.

3.3.7 Size on the FPGA

The size of the NoC depends on its configuration. For example there a configuration that requires virtual channels may require more space on the FPGA than a configuration without the virtual channels. Although having virtual channels in the NoC may speed up the routing process, it is not necessary to have them.

3.3.8 Readability of the Source Code

The source code is very readable. Each module has its own source file and only uses this one file for the implementation.

A minor downside is the heavy usage of generate statements in the code. This makes it harder to read the code because the configuration has to be either known or looked up in order to know which path is taken in the code, but the configuration of the NoC can be remembered very easy.

3.3.9 Extensibility

NoCem is very easy to extend. The source code is very organized and packed into small modules. NoCem also provides a good documentation how packets can be sent via the network and how they can be injected into the network. This makes the process of writing an access point very easy.

3.4 Conclusion

Making a decision for either implementation was not easy. Both have very unique features. Atlas on the one hand uses a GUI for the whole configuration process while NoCem does the configuration with a simple configuration file but it would be possible to write a GUI for the configuration of NoCem, as well.

Table 3.3 shows a comparison of the two implementations with focus on the size of the NoC. It can be seen that NoCem needs more space on the FPGA than the implementations provided by Atlas.

The decision finally was made for NoCem. It provides the better features and is easier to configure. Besides the fact that it needs more space on the FPGA, NoCem's code is readable and can be extended very easily.

Easy to configure

NoCem provides a configuration file where all the configurations can be made. If something needs to be changed only a few lines in the configuration file have to be changed. Custom changes are kept and only a recompilation of the NoC is needed to apply the changes.

Atlas only provides the GUI to configure the NoC. Changes can only be made in the GUI and result in a new generation of the NoC. This may cause problems when some custom changes to the source code are made. A new generation of the NoC will result in a loss of changes, unless they were already made in the files that are used for generation.

Readability of the Source Code

NoCem's source code is very easy to read and understand, beside the fact that the generate statements sometimes can make it a little bit difficult to know whether a specific section of code is generated or not.

Atlas on the one hand doesn't come with a documentation in English. The documentation is included in the source files and is in Portuguese. If you are not familiar with this language, no information can be extracted from the documentation. Although the source code looks clean, some variables are named in Portuguese. Again, it can't be said what a specific variable is doing without any doubts.

Test bench

Both implementations provide a test bench. NoCem provides a test bench completely in VHDL and provides modules to send packets into the NoC and extract the sent packets from the NoC again. This makes it easy to write an own test bench on top of the existing one. Atlas on the other hand provides a test bench completely written in SystemC. Unfortunately the server which should run the test bench should, was not able to compile SystemC syntax and it was impossible to run the test bench.

Environment	NoCem	Atlas					
		Hermes	Hermes CRC	Hermes SR	Hermes TB	Hermes TU [a]	Mercury
Type							
Topology	4x4 Mesh	4x4 Mesh	4x4 Mesh	4x4 Mesh	4x4 2D Torus	4x4 1D Torus	4x4 Mesh
FIFO Implementation	LUT						
# Routers	16	16	16	16	16	16	16
# Rows	4	4	4	4	4	4	4
# Columns	4	4	4	4	4	4	4
# VC	2	2		6	1	2	1
VC Width	2	16					
Fifo Depth	4	16	16	16	16	16	16
Max Packet Length	8						
Flit Width			16	16	16	16	16
Data Width	8						
Address Width	4						
Scheduling	Round Robin	Round Robin	Round Robin	Age Based	Round Robin	Round Robin	Round Robin
Routing Algorithm		XY	XY	XY	West First	XY	CG
CRC Type			Link CRC				
# of Slice Registers	23715 (52%)	6692 (14%)	2755 (6%)	24733 (55%)	3711 (8%)		15480 (35%)
# Slice LUT	27345 (61%)	21031 (46%)	9863 (22%)	74650 (166%)	13153 (29%)		26163 (58%)
# Fully used LUT-FF pairs	12454 (32%)	6035 (27%)	2513 (24%)	21338 (27%)	3364 (24%)		4575 (12%)
Number of BUFG / BUFCTRLs	2 (6%)	16 (50%)	16 (50%)	16 (50%)	16 (50%)		2 (6%)
Minimum Period	6,803 ns	10.732 ns	10.418ns	11.130 ns	7.232 ns		7.167 ns
Maximum Frequency	146,996 MHz	93.179 MHz	95.988 MHz	89.847 MHz	138.274 MHz		139.528 MHz

Table 3.3: Comparison between existing Network-on-Chip (NoC) Tools in size on a Xilinx ML507 board.

[a] Although it could be configured with Atlas the synthesis tool caused a crash during synthesis of this network

Chapter 4

Design

4.1 Extended NoC Module

As described in section 3.1 NoCem is used as a basis for this thesis. It already provides a working NoC that needs to be configured so that the application can be applied to it.

4.1.1 The Networks Access Points

An access point is either an entry point into or an exit point from the NoC. Access points need to be placed wherever data is extracted from the NoC or data is send to the NoC. Figure 4.1 shows where the access point is located. It sits right on top of the node and provides send and receive functionality.

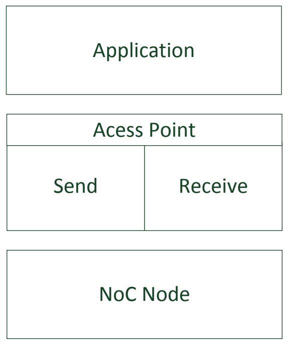


Figure 4.1: Overview: The access point is split into a send and a receive part

4.1.2 Receiving Data from the NoC

The basic state machine, used for all counter measures, in the receiving process has four different states (Init, ReInit, Idle and Data). With these states it is possible to receive data correctly. Figure 4.2 shows how the states are connected together.

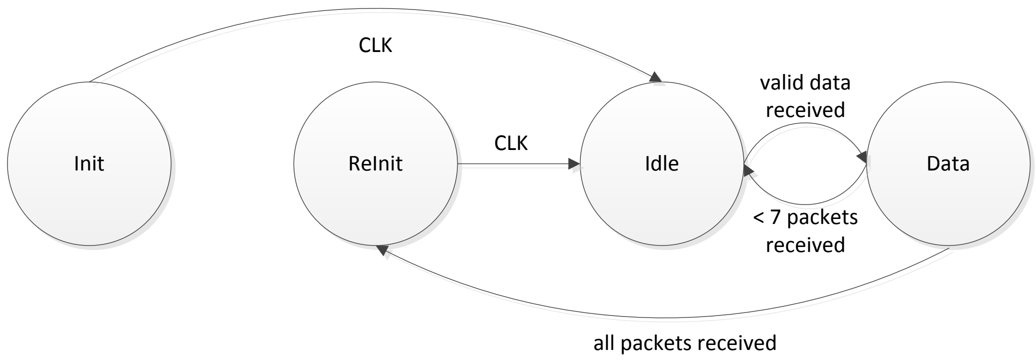


Figure 4.2: State machine used to receive data.

For the CRC countermeasure the state machine looks a little bit different. The CRC is generated for all packets that are sent and thus can only be received as the last packet. Therefore it gets an extra state where the check of the calculated and the received CRC takes place, by this time the CRC of the last packet received is calculated. The new handling of the states is shown in figure 4.3.

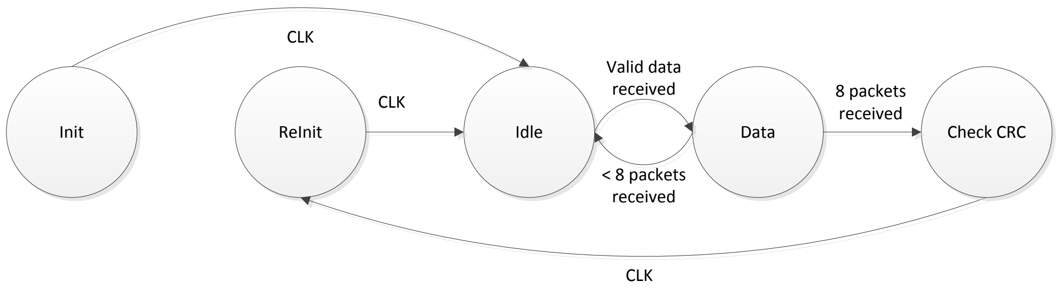


Figure 4.3: State machine used to receive data using CRC as a counter measure.

- **Init**: In this state the whole module is initialized. It is the entry state and is active when the NoC is being reset. The Init state is only entered after a reset of the NoC. The transition to the next state happens if the first CLK transition happens.
- **ReInit**: This state is entered when the whole data is received. It resets all the variables needed for the reception and then traverses right to the Idle state.
- **Idle**: This state is entered right after the Init or the ReInit states have finished. This state checks if data is available on the input lines. If data is received in this state the transition to the Data state is made.

- **Data**: Writes the data into the right registers according to their positions in the data stream. This state also sends an acknowledge to the sender. After one packet has been received the idle state is called again. If the activated countermeasure is CRC, the state Check CRC is entered right after the last packet has been received.
- **Check CRC**: In this state the received CRC is compared to the calculated CRC and the transition back to the ReInit state is made.

4.1.3 Sending Data into the NoC

The state machine for sending data contains five states (Init, ReInit, Idle, Send and WaitForReception). Figure 4.4 shows how they are linked together. The layout is very similar to the receive state machines.

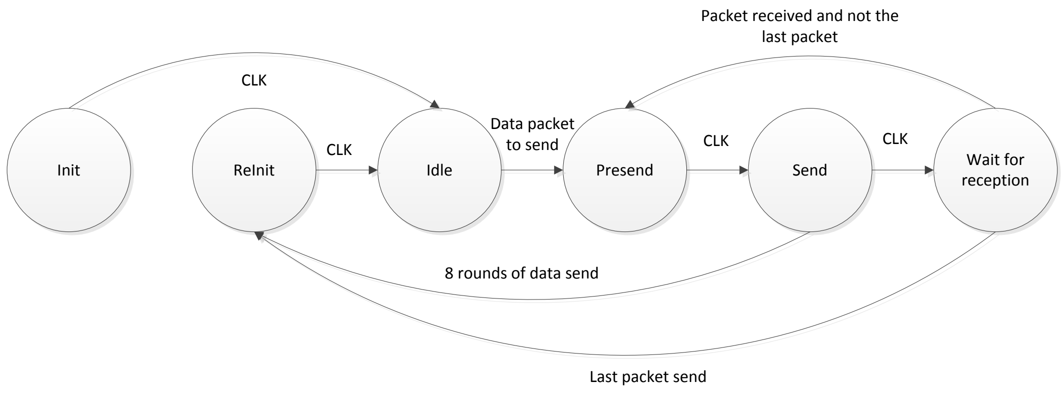


Figure 4.4: State machine used to send data.

- **Init**: In this state the whole module is initialized. It is the entry state and is active when the NoC is being reset.
- **ReInit**: This state is entered when the whole data is received. It resets all the variables needed for the send process.
- **Idle**: In this state the module waits until data is available to send. If data is available to send, it traverses to the Send state.
- **Presend**: Waits until the calculation of counter measures is finished, which normally takes one CLK cycle and then the next state is entered.
- **Send**: Prepares one packet to send and actually sends it to the destination. In the case that the 8^{th} packet should be send then the ReInit state is entered. Else the WaitForRecepton state is entered.
- **WaitForReception**: After one packet is sent this state waits until it is received by the receiver. If more packets have to be sent, then the Send state is entered again. If the send process is finished, the ReInit state is entered.

4.2 Framework for the Fault Injection System

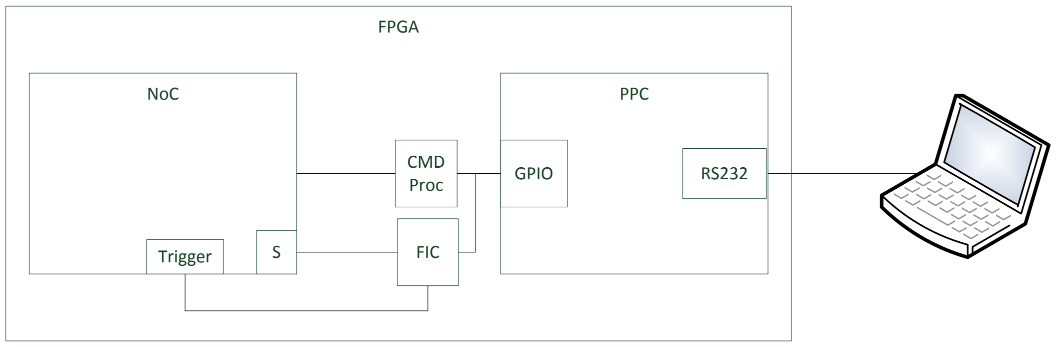


Figure 4.5: Framework for the fault injection system containing the NoC

The framework consists of several parts. It consists of the Fault Injection Controller (FIC) which can be programmed to react on different states of the NoC. The FIC gets the information about the current state of the NoC via triggers and then can decide due to its programming if the saboteurs (S) should be triggered or not. Figure 4.5 shows the setup.

In order to program the FIC and also to send data to the NoC either fixed values have to be added. The downside of this is that they have to be hard coded in the hardware and cannot be altered without a lot of effort.

An easier solution would be to add a CPU to communicate with the FIC and the NoC. As it is easier and faster to change some piece of software this seems to be the right choice. For the CPU a PowerPC is chose, simply because it is already provided by Xilinx EPS and can be easily extended and it provides a simple GPIO IP core which can be used to communicate with the FIC and NoC. The PowerPC also provides an RS323 interface which can be used to show a status on the console of a PC connected to the FPGA board.

Last but not least a mechanism to communicate with the NoC is used. Therefore the Command Processor (CMD Proc) module is used. It can parse the commands received from the PowerPC and translate them into data for the NoC.

4.2.1 Command Processor Commands

In order to control the NoC and to send and receive data from the NoC the PowerPC has to send several commands. The command processor itself does not have any buffers and thus only one command can be sent and processed. The software has to make sure that the NoC is given enough time to process the commands until a new one is sent. This is done via an acknowledge signal.

PPC, GPIO and NoC may work with different clock speeds, it may not be possible to hold a command for exactly one clock cycle. It is also not possible to tell if the command is correctly received by the NoC. Therefore a command has to be acknowledged.

The acknowledge signal contains exactly the same command code that was sent to the NoC. GPIOs already contain input and output ports that are exactly the same size. They only need to be defined in the top level design. A simple line that just makes a transition from logical low to logical high would also require a GPIO module with a width of just one bit and more overhead then the proposed solution because of the new GPIO peripheral. Sending the whole command code also adds a better flow control. The received command can be checked and a possible error can be detected.

An acknowledge is sent back to the sender by simply sending the received command back when its processing is done. This allows the application to wait until the command is processed and then sends new commands. Figure 4.6 shows how a command is sent to the NoC, processed and sent back to the PPC. It also shows the lifetime of the command on the PPC and on the NoC.

A whole set of commands is defined for this thesis. Table 4.1 shows the commands defined to interface with the NoC. With these commands it is possible to send data into the NoC and extract the encrypted data from it. Table 4.1 shows in which direction a command can be sent. It can be either sent by the PowerPC and received by the NoC or it can be sent by the NoC and received by the PowerPC. It has to be said that table 4.1 shows the command names in the direction firmware to FPGA, NoC. The names in the hardware model are different. A SEND in the firmware becomes a RECEIVE in the hardware model and vice versa.

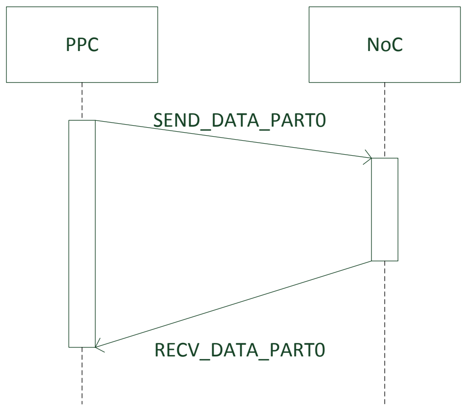


Figure 4.6: A typical command sequence to send data into the NoC

4.2.2 The Firmware

As stated in the beginning of section 4.2 PowerPC is used to control the FIC and the NoC and that implementing the commands for FIC and NoC in hardware would not be the goal. Therefore the firmware of the PowerPC is used.

The firmware contains the control of the FIC. The firmware therefore is already provided by [GKS^{+}11]. The fault injection library currently does not custom callbacks for methods that can set different fault injection patterns. This is kind of annoying because for the both attacks two different firmwares have to be written that only differ in one point, the fault injection patterns. It is much better to have one firmware that is able to

Command	Value	PPC to NoC	NoC to PPC	Description
`DO_NOTHING`	0x000	x	x	Does nothing. Clears the send command.
`KEY_PART_0`	0x001	x		First column of the key.
`KEY_PART_1`	0x002	x		Second column of the key.
`KEY_PART_2`	0x003	x		Third column of the key.
`KEY_PART_3`	0x004	x		Fourth column of the key.
`DATA_PART_0`	0x005	x		First column of the data.
`DATA_PART_1`	0x006	x		Second column of the data.
`DATA_PART_2`	0x007	x		Third column of the data.
`DATA_PART_3`	0x008	x		Fourth column of the data.
`RECV_COMPLETE`	0x009	x		No more data to send. NoC can start work.
`DATA_READY`	0x00B		x	Not used.
`ERROR_KEY`	0x00C		x	Key error.
`ERROR_DATA`	0x00D		x	Data Error.
`CALC_STARTED`	0x00E		x	Calculation was started by the NoC.
`ENCRYPTED_DATA_PART_0`	0x00F		x	Get first encrypted data column.
`ENCRYPTED_DATA_PART_1`	0x010		x	Get second encrypted data column.
`ENCRYPTED_DATA_PART_2`	0x011		x	Get third encrypted data column.
`ENCRYPTED_DATA_PART_3`	0x012		x	Get fourth encrypted data column.
`CALC_DONE`	0x013		x	Calculation is done. Data ready.
`ENCRYPTED_DATA_COMPLETE`	0x014		x	Encrypted data is read from NoC. No more data available.
`RECV_RECV_COMPLETE`	0x015		x	All data received. Can ready to start calculation.
`RECV_UNKNOWN_CMD`	0xFFF		x	Command unknown. NoC doesn't know how to process it.

Table 4.1: Command overview. Those are the commands used in the firmware. On the FPGA the commands that are receive commands in this table are send commands and vice versa.

set both patterns. Therefore the function `fic_write_pattern_cb` is introduced. It takes an callback as argument and executes the callback if one is passed to the function.

The other part of the firmware consists of two layers. One layer represents the NoC and contains methods for the command processor to communicate with it. The second layer contains the data and methods for AES. As in the hardware the AES layer is build on top of the NoC layer.

4.3 Attack Scenarios

Several parts of the NoC can be attacked but not everywhere an attack will lead to a good result. This is shown for the case of a pipelined version of AES that is run as application on the NoC. For other attacks the command set needs to be adapted on the hardware and software side.

This leaves only a limited number of attacks on AES. One is a round reduction and the other one is a more complicated attack introduced by Takahashi et al. in [TFYpt] on the round key transmission in the 9th round of the AES core. Those attacks can be executed directly on the NoC. There is no need to place saboteurs inside the access nodes or the AES cores directly.

For the round reduction the control registers of the NoC need to be attacked. With bit flips on those lines it should be possible to redirect the whole data to the extraction node where the partly encrypted data can then be extracted and evaluated.

In order to execute Takahashi et al.'s attack some changes have to be made to it. Krieg et al. proposes a possible implementation of this attack in [KGS+12]. Although it places the saboteur after the XOR gate, it would be possible to get the same results if the saboteur were placed right before the XOR gate. Figure 4.7 shows this modification of the attack.

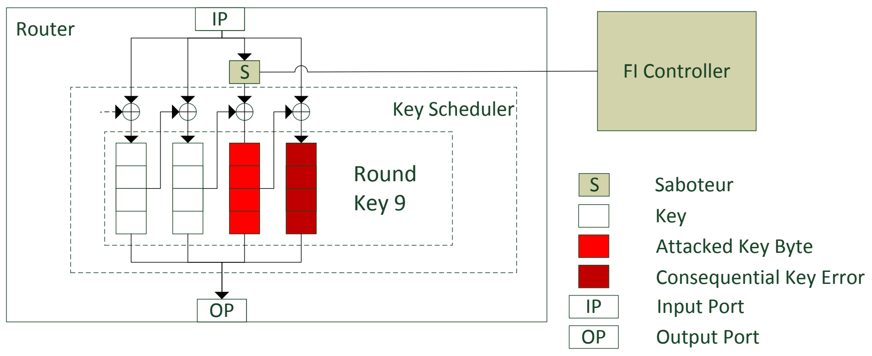


Figure 4.7: Krieg et al.'s attack modified to fit into an NoC environment

The results should then show if it is possible to retrieve the key and if the counter measures are able to detect the change in the key because of the saboteur.

4.4 Countermeasures

Counter measures are an essential part of this thesis. In this section the design of the counter measures is described, together with their effects on the NoC.

4.4.1 XOR Table

This counter measure is also known as parity bit. To calculate a parity bit over a wide range of data an XOR table, or tree seems like a good solution. It always XORs two lines next to each other. The output is half the amount of lines which are processed in the same way until one bit is left. For the 32 bits of data that are used in this thesis this process has to be repeated four times until only one bit is left. A snippet of the implementation can be seen in figure 4.8.

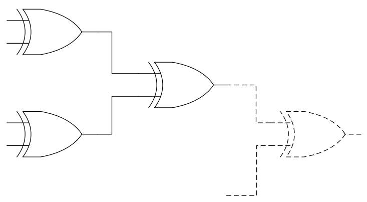


Figure 4.8: Simple parallel XOR-Table implementation.

Another way to calculate a parity bit would be to take two lines and xor them. They then are used as input for the next xor together with the next line. This is repeated until no more lines are left. The result is then the parity bit. For the 32 bits of data this has to be done exactly 32 times. A snippet of the implementation can be seen in figure 4.9.

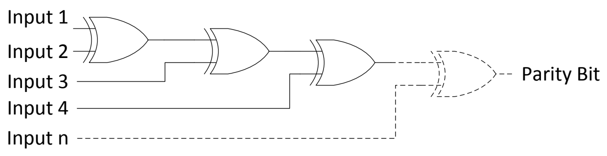


Figure 4.9: Simple serial XOR-Table implementation.

Now only one parity bit for the packet control line is submitted. The same goes for the data lines. The newly calculated parity bit is added to the data lines and sent. On the receiver side it is extracted and compared with the calculated value of the other 32 bits of data.

4.4.2 Hamming Weight

The hamming code is very similar to the XOR table implemented in section 4.4.1. A hamming code still calculates XOR tables, but not just one. It calculates several tables for different combinations of input bits and therefore is capable of detecting two errors and correct one of them.

Hamming Code calculation is still a little bit more difficult than calculating an XOR table. Not only are more parity bits calculated. They are also in specific spots. For the data lines the parity bits are left in place, because they do not bother, but for the packet control lines a few changes have to be made.

The Hamming Code expects the parity bits in a specific place and thus the decoder has to be aware of this. Especially if no specialised decoder for the packet control lines is used. Then the bits have to be aligned correctly prior to decoding them. Decoding is basically the same as encoding with a comparison of the calculated parity bits.

4.4.3 CRC

Because of the size of the CRC it is not possible to attach the calculated values to the same packet. Instead the CRC is calculated over all transmitted packets and then sent as the 9th packet afterwards.

Now the problem lies in the detail again. The CRC packet has to be able to contain the CRC data for the data as well as the packet control lines. Therefore the 32 data lines that are available are split up.

The CRC calculated over the packet control lines is very small. The full CRC only takes up 8 bit of space. Those 8 bits are the 8 least significant bits of the data packet.

The CRC calculated over the data lines is a little bit too big to transfer in one packet. It has 32 bits in total. Now either another packet is sent or the 24 bits in the data packet that are not used right now are filled up with 24 bits of the CRC. It does not cause any security vulnerabilities to only send 24 bits instead of 32 bits as the CRC is only used to check if the previously sent data has been corrupted or not.

4.4.4 Reed-Solomon Code

By having a closer look at RS some doubts for this algorithm seemed to be the available registers and lookup tables on the FPGA. To spend less time an implementation[1] was downloaded from `www.opencores.org`. This implementation is then used to evaluate if it would be possible to fit enough RS encoders and decoders on the NoC. Unfortunately it turned out, that already one set of encoders and decoders need most of the FPGA resources and therefore no further investigation on this counter measure will be done.

4.4.5 Secure Hash Algorithm (SHA)

As described in section 4.4.4 the same doubts are given for the SHAs. Again to save time the implementation[2] was downloaded from `www.opencores.org`. Again it showed

[1]`http://opencores.org/project,rs_dec_enc`

[2]`http://opencores.org/project,sha3`

that the implementation uses most of the resources of the FPGA and therefore no further investigation on this counter measure will be done.

4.5 Use Case: Implementing an AES Module with NoC

The AES module used in this thesis was developed by Satyanarayana Hemanth and can be obtained from [Hem10]. This implementation works as a basis. Upon further research it was seen that this version is not suitable to work with an NoC because of its design. The whole algorithm is implemented in one file and is not split up into several smaller packages. This makes it impossible to map parts of the algorithm on NoC nodes.

Therefore the AES module needs to be redesigned. All the functions that AES provide need to be packed into small distinct modules. The modules are:

- KeyExpansion
- AddRoundKey
- MixColumns
- ShiftRows
- SubBytes

By splitting the implementation into this modules it is possible to create the initial round, body rounds and final round without any problems.

4.5.1 AES Initial Round

The AES initial round module consists of a send, a receive, an add round key, shift rows, sub bytes and a key schedule module. The module itself has no own functionality except that it connects these modules. Figure 4.10 shows the data flow through this module.

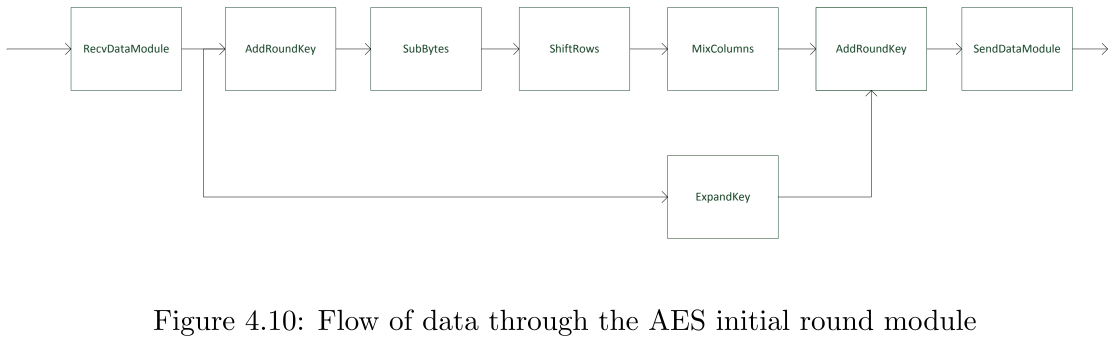


Figure 4.10: Flow of data through the AES initial round module

The initial round is different from the other rounds. It has two AddRoundKey modules inside. The first one creates a new key right after the body round module receives data and generates a key which is used to cypher the data. The second one then creates a new key with the expanded key.

4.5.2 AES Body Round

The body round module consists of the same modules as the initial round does. The only difference is that it only has one AdRoundKey module inside which is fed with data at the end of a body round. Figure 4.11 shows the data flow through this module.

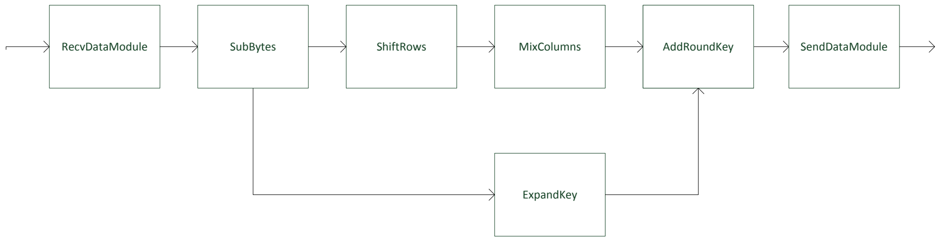


Figure 4.11: Flow of data through the AES body round module

4.5.3 AES Final Round

The final round module contains the same modules as the body round, except for the mix columns module. The last round of AES does not make use of this module and therefore it is missing here. Figure 4.12 shows the data flow through this module.

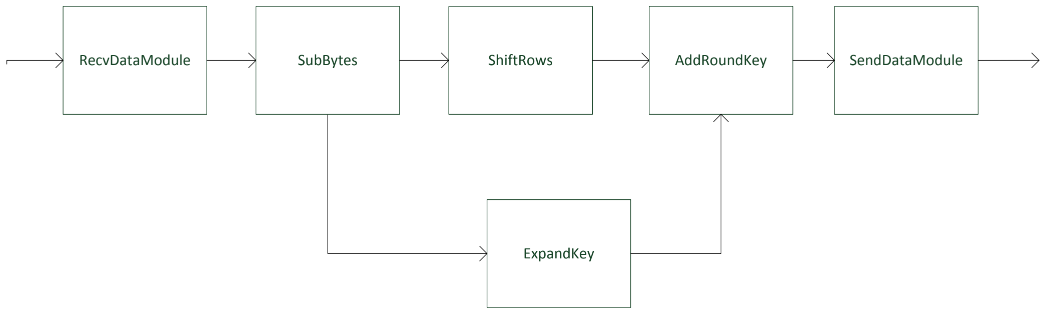


Figure 4.12: Flow of data through the AES final round module

4.5.4 Connecting the NoC Access Points to the Application

Until now it has not been possible to work with the NoC. Although it is possible to send and receive data, no data is processed. To do so the nodes must contain some AES code. As they are very specialized, the algorithm has to be split up.

A first step is already done in section 4.5. Now that all the functions are available it is easy to set up specialized NoC nodes that can do parts of the calculation. To calculate AES, three different types of nodes are needed:

- **Initial Round**: In this round all the initial calculations for AES are done.
- **Body Round**: This module does all the calculations that can be referred to as body. This core is used nine times.
- **Final Round**: AES handles the last round different. It does not contain the Mix-Column module.

4.5.5 Evaluation Tools to compute Keys from induced Faults

Until now the design has only features to attack the NoC. It has not been possible to evaluate them yet. Since the evaluation can be very calculation intensive, a set of tools is developed to help with the process of retrieving the secret key.

The main purpose of this tool is to support the process of retrieving the secret key. It should be possible to retrieve the secret key for the round reduction attack and for the modified Takahashi/Krieg attack. The calculation of the secret key for Takahashi can be very time consuming (Takahashi et al. calculated the time for the worst case to retrieve the key to be about one year [TFYpt]). Although this thesis doesn't show the whole computation of the secret key for Takahashi's method it is still possible to extend the calculation to retrieve the whole key.

Another goal is to only have one tool that can calculate the round reduced key and the Takahashi key retrieval algorithm. Multiple tools would be good enough for this but having them in just one tool has the benefit that the user does not have to search for the other tools and can just open them from the GUI.

4.5.6 Layout of the GUI

The layout of the GUI is kept very simple. It is a multiple document interface (MDI), which means that a workspace keeps track of the open dialogs. Dialogs can only be moved in that workspace and if the application is minimized all dialogs are minimized to, or if the workspace is closed all opened dialogs are closed to.

The Takahashi dialog is very simple and can be seen in 4.13. The tabwidget at the top contains the plain, cypher and faulty cypher text. Because the attack needs at least two different sets of plain, cypher and faulty cypher text it is possible to add new tabs by clicking on the `Add new P, C, C'` button. This will create a new tab with the same fields where data can be entered. By clicking on the `Delete current P, C, C'` button the currently active tab can be deleted.

The input has to have a C/C++ hex style syntax and is separated by a colon. This allows an easy parsing of the values. It is also possible to get this values directly from the firmware that runs on the PowerPC. By clicking on the `Calculate` button the calculation of the key is started.

At the bottom of the dialog another tabwidget can be found. It is filled with the calculated data. The Key-tab contains a 4x4 spreadsheet. All the calculated keys are

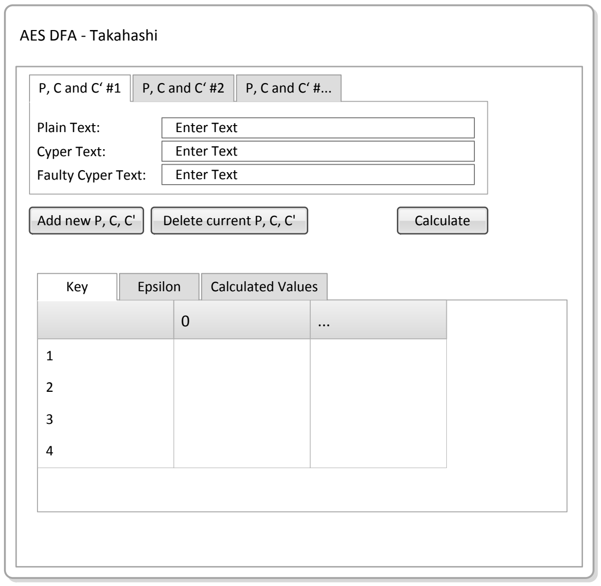


Figure 4.13: Dialog to retrieve key from AES based on Takahashi’s attack

entered in the corresponding row and column. Each value also contains their occurrence in braces right after the calculated value.

The Epsilon and Calculated Values tab are very similar. They contain a spreadsheet which holds the name and the values of the calculated data. With these intermediate values it is easier to find errors in the calculation and the results can be verified easily.

The round reduction dialog contains all the elements to successfully calculate the key from a round reduced AES. The dialog can be seen in figure 4.14. The input fields for plain and cypher texts can be found at the top of the dialog. Because two different plain texts and with that two different cypher texts are used, input boxes are provided for them.

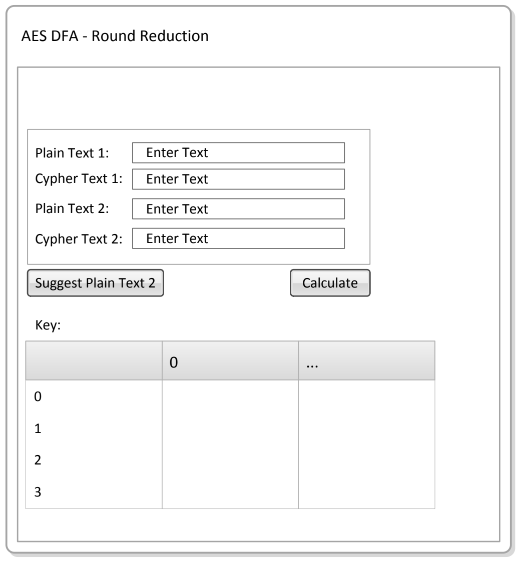


Figure 4.14: Dialog to retrieve key from AES based on a round reduction

Although it seems easy to have two different plain texts, the second plain text has to fullfill certain criteria. The two inputs need a certain distance from each other. A good suggestion for the second plain text can be made by pressing the `Suggest Plain Text 2` button. It then calculates values for the second plain text.

By pressing the `Calculate` button, the values for the new key are calculated. The calculation itself is just a rerun of the AES. Refer to section 2.5.1 for more details.

The results are shown in the spreadsheet at the bottom of the dialog. It contains the rows and columns with the calculated values.

In case of an error, or if it was not possible to calculate a key, a message box will show up, informing the user that something went wrong.

Chapter 5

Implementation

This section deals with the implementation of the diploma thesis. It shows the details of the implementation and elaborates the solutions that lead to the final results.

The first section of this chapter describes how the infrastructure is implemented. It explains in detail how the access point to the NoC is implemented as well as send and receive modules. This is followed by the implementation of the security measures that each access point has.

5.1 Implementation of the Extended NoC Infrastructure (Including the Use Case Implementation)

One very basic implementation is used for the attacks and the security measures (see table 5.1). It allows a comparison between the basic setup and the different security measures. The only thing that changes in the configuration is the size of the packet control lines. They have to be adjusted to fit the needs of the security measure. For example, it will not make any sense to transmit the additional bits that a hamming weight generates in an additional packet, because they have to be evaluated in each round and thus would introduce bigger changes to the code that would make a comparison much harder.

Table 5.1 gives an overview of the most important parameters used in the configuration. The topology of the NoC is a mesh. This layout saves some space as connections. Nodes that are at the end of the graph are not directly connected to the node on the other side (e.g. a node at the bottom left corner has no direct connection to the node at the bottom right corner).

The NoC contains 16 access points. They are aligned in four rows and four columns. A bus type of network was tried out but failed due to some implementation errors in the NoCem implementation. The first version also used virtual channels, which would allow multiple packets to be transmitted on one connection, however another bug in the NoCem implementation stopped any further investigation in using virtual channels.

The data bus is 32 bits wide. This makes it possible to transmit one row of the AES key or of the data to encrypt at once. As the AES implementation uses an 128 bit key, only four transmissions are needed to transmit the whole key. Because an 128 bit key is used, the maximum amount of data is also 128 bit. One row of data can be transmitted during one transaction.

Parameter	Value
Topology	Mesh
Type	Simple Packet Type
Topology	No Virtual Channels
Access Points	16
Columns	4
Rows	4
Data width	32 bit
Address width	4 bit
Maximum Packet Length	8 Packets
Packet Control Width	9 bit

Table 5.1: Configuration parameters of the NoC

Four rows of data and four rows of key lead to a maximum packet length of eight continuous packets. NoCem is able to have maximum packet length of 8 continuous packets. Although in the later development of this thesis the maximum packet length will be exceeded.

5.1.1 Access Points

Access points are used to inject or extract data from the NoC. They have to be written for each task as they can send data in various ways.

The basic layout of an access point used in this thesis can be seen in figure 5.1. It shows that the access points contain a send and receive unit. These two units are the same for every access point used in this thesis. Still there are some special access points that lack either a send or a receive unit. This is because those units are not needed for those access points to work (e.g. the ingress access point that only needs to send data).

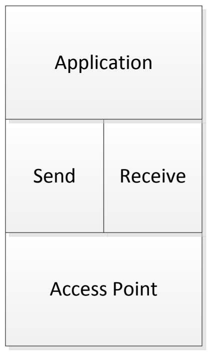


Figure 5.1: General construction of an access point

Send Module

The send module takes each row with either key or data values and tries to send it. Figure 5.2 shows the general input and output values for this module. Table 5.2 gives an overview of each pin and what it is used for. The send module implements the state machine in figure 4.4.

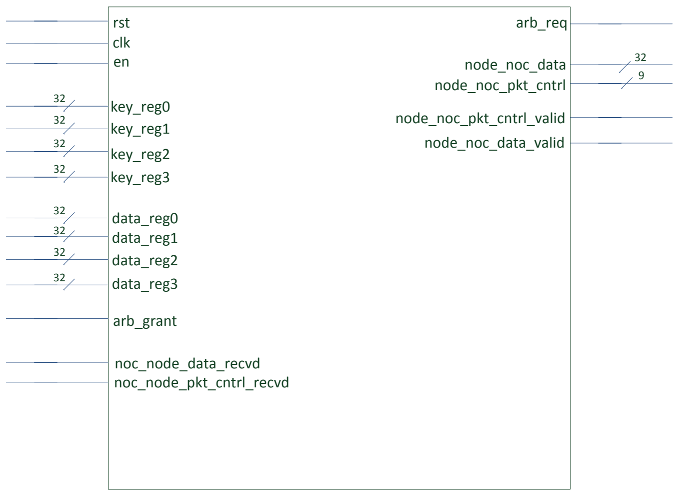


Figure 5.2: The send module of an access point

Pin	Width	Direction	Description
clk	1	in	Clock
rst	1	in	Reset
key_reg0	32	in	Row 0 of the key
key_reg1	32	in	Row 1 of the key
key_reg2	32	in	Row 2 of the key
key_reg3	32	in	Row 3 of the key
data_reg0	32	in	Row 0 of the data
data_reg1	32	in	Row 1 of the data
data_reg2	32	in	Row 2 of the data
data_reg3	32	in	Row 3 of the data
arb_grant	1	in	Set if access on the line is granted
noc_node_data_recvd	1	in	Set if the data was successfully received by the next node
noc_node_pkt_recvd	1	in	Set if the control packet was successfully received by the next node
arb_req	1	out	Set if the access point wants to send data
node_noc_data	32	out	Contains the data for the current transmission round

noc_node_pkt_cntrl	9	out	Contains the control data for the packet
node_noc_pkt_cntrl_valid	1	out	Set when the control data is valid. Next node can receive it.
node_noc_data_valid	1	out	Set when the data is valid. Next node can receive it.

Table 5.2: Send module pin description

It keeps track of the currently sent data with an internal counter. According to the value in the counter the data that relates to the counters value is sent out. Data value rows 0 to 3 are sent when the value of the counter is smaller than 3 and key row 0 to 3 are sent afterwards.

Receive Module

The receive module is built very similarily to the send module in the previous section. It only gets some additional lines and most of the lines that are inputs to the send module are now outputs and outputs of the send modules are now used as input for the receive module. Table 5.3 shows the pins and their description whereas figure 5.3 shows the general input and output variables.

Pin	Width	Direction	Description
clk	1	in	Clock
reset	1	in	Reset
noc_node_data_valid	1	in	Set if the data is valid
noc_node_pkt_cntrl_valid	1	in	Set if the control packet is valid
isSOP	1	in	Set to control the internal receive round counter
isEOP	1	in	Set to control the internal receive round counter
noc_node_data	32	in	Contains the data about to be received
key_reg0	32	out	Row 0 of the key
key_reg1	32	out	Row 1 of the key
key_reg2	32	out	Row 2 of the key
key_reg3	32	out	Row 3 of the key
data_reg0	32	out	Data 0 of the key
data_reg1	32	out	Data 0 of the key
data_reg2	32	out	Data 0 of the key
data_reg3	32	out	Data 0 of the key
node_noc_data_recvd	1	out	Set when the data was received. Tells the sending node that the data is received successfully.

node_noc_pkt_cntrl_recvd	1	out	Set when the control packet was received. Tells the sending node that the data is received successfully.
recvd_data	1	out	

Table 5.3: Receive module pin description

An incoming packet is detected by the lines noc_node_data_valid and noc_node_pkt_cntrl_valid. If both are logic high the values in noc_node_data and noc_node_pkt_cntrl are valid and the packet is received. Besides the values isSOP and isEOP indicate whether it is the beginning of a transmission (data 0) or the end of a transmission (key 3). If the data and the packet control are valid, the internal counter is increased. The isSOP and isEOP are used to set the counter to the start value or to the end value. This allows an error correction when for some reasons glitches appear and the counter is increased by accident.

Some lines like the recv_error is not available in the basic implementation (without counter measures). It is simply not needed and not connected to anything. The module also has two additional lines in the implementation which allow an activation and deactivation of countermeasures on data and packet control lines.

5.1.2 Injecting Data

Data is injected when command DATA_READY is received (see table 4.1) via the GPIO lines from the PowerPC. The whole data that is currently saved in the input buffers is then send to the NoC.

This module only contains a send module to send that data to the NoC that has been received from the PowerPC via the GPIO lines.

5.1.3 Extracting Data

Very similar to the inject module is the extract module. It receives the data from the final AES round and sends it back to the PowerPC by sending the CALC_DONE command following the data. The PowerPC then picks up the data and the application can display it.

This module contains only a receive module to receive the data from the NoC. The data is then sent back via GPIO lines to the PowerPC.

5.2 Implementing the Security Measures for the Access Points

This section describes the implementation of the security measures used in this thesis. It shows how the implementation was done and which problems have occurred.

There is one common point between all the security measures. As soon as an error is detected the error bit on the packet control line is set. This allows the extraction module

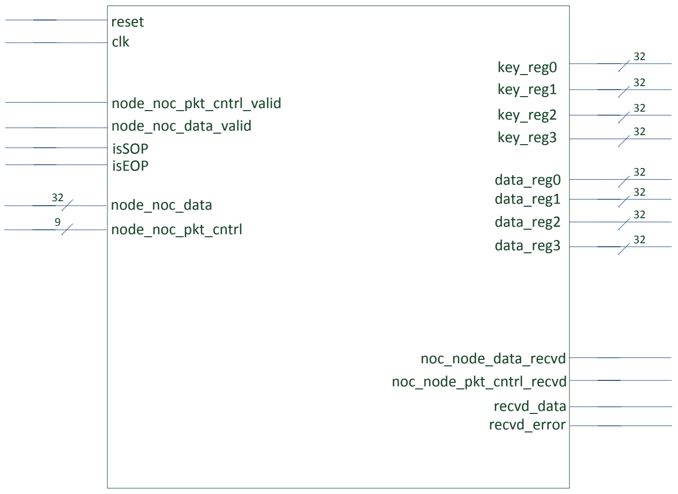


Figure 5.3: The receive module of an access point

to recognize the error and set the data to zero. This flag informs the last node that there was an error and again the extraction node will set the data to zero.

Setting the data to zero has the benefit that the attacker cannot gain any information from his attack. By showing him only zeros the only information he has gathered is that his attack had at least some impact on either data or packet control lines, but he still has no data to either do a DFA or calculate the round key using Takahashi et al.'s method.

5.2.1 Detecting the Round Reduction

A round reduction can be detected very easily, especially for AES. Each node knows which round it is calculating. The only the NoC now needs to send the current round number on the packet control lines, so that the nodes know which node was previously executed. This is an easy and straight forward implementation. Figure 5.4 shows the new control packet which is able to send the round that was currently processed.

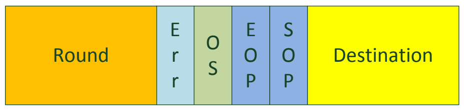


Figure 5.4: New packet containing information about the round in the „Round“ field of the packet.

5.2.2 XOR-Table

The implementation chosen for this counter measure is the parallel XOR-Table implementation as described in section 4.4.1. It is the faster implementation compared to the serial implementation an.

The implementation has an input of 32 bits. This means that the data lines can be processed without any problems, while the packet control lines cause some problems. There are not as many packet control lines as data lines do exist. To overcome this problem, and to use only one implementation of the XOR-Table, the missing amount of lines are filled with zero. Then it is no problem to use the same implementation for data and packet control lines.

Now that the parity bit is calculated it is transmitted via the packet control lines. This is the only bit that is not part of the parity bit calculation of the packet control lines. Figure 5.5 shows the new packet.

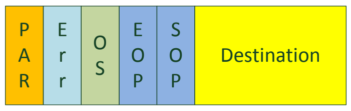


Figure 5.5: New packet containing information about the parity in the „PAR“ field of the packet.

The implementation just uses generate statements to generate the XOR tree nearly automatically. Listing 5.1 shows the generate statements used.

```
tmp <= datain;

g1: for i in 0 to 15 generate
        tmp1(i) <= tmp(2*i) xor tmp(2*i+1);
end generate;

g2: for i in 0 to 7 generate
        tmp2(i) <= tmp1(2*i) xor tmp1(2*i+1);
end generate;

g3: for i in 0 to 3 generate
        tmp3(i) <= tmp2(2*i) xor tmp2(2*i+1);
end generate;

g4: for i in 0 to 1 generate
        tmp4(i) <= tmp3(2*i) xor tmp3(2*i+1);
end generate;

dataout(DATA_OUT_SIZE-1 downto 0) <=  tmp4(DATA_OUT_SIZE-1 downto 0);
```

Listing 5.1: Calculation of the parity bits for the XOR-Table counter measure

The data packet also has to be extended in order to transmit the parity bit. Therefore it transmits now 33 bits, where the MSB is the parity bit.

When the packet is received those fields are extracted and the receive module has to calculate the parity bit of the received packet. If the parity bits are equal then there is a high possibility that the received packet is correct.

5.2.3 Hamming Code

The hamming code is implemented very similar to the XOR Table, except that 6 different parity bits are calculated. The generation of the parity bits can be seen in listing 5.2.

```
dout( 0) <= din( 0) xor din( 1) xor din( 3) xor din( 4) xor
            din( 6) xor din( 7) xor din( 8) xor din(10) xor
            din(11) xor din(13) xor din(15) xor din(17) xor
            din(19) xor din(21) xor din(23) xor din(25) xor
            din(26) xor din(28) xor din(30);

dout( 1) <= din( 0) xor din( 2) xor din( 3) xor din( 5) xor
            din( 6) xor din( 9) xor din(10) xor din(12) xor
            din(13) xor din(16) xor din(17) xor din(20) xor
            din(21) xor din(24) xor din(25) xor din(27) xor
            din(28) xor din(29) xor din(30);

dout( 3) <= din( 1) xor din( 2) xor din( 3) xor din( 7) xor
            din( 8) xor din( 9) xor din(10) xor din(14) xor
            din(15) xor din(16) xor din(17) xor din(22) xor
            din(23) xor din(24) xor din(25) xor din(29) xor
            din(30) xor din(31);

dout( 7) <= din( 4) xor din( 5) xor din( 6) xor din( 7) xor
            din( 8) xor din( 9) xor din(10) xor din(18) xor
            din(19) xor din(20) xor din(21) xor din(22) xor
            din(23) xor din(24) xor din(25);

dout(15) <= din(16) xor din(17) xor din(18) xor din(19) xor
            din(20) xor din(21) xor din(22) xor din(23) xor
            din(24) xor din(25);

dout(31) <= din(26) xor din(27) xor din(28) xor din(29) xor
            din(30) xor din(31);
```

Listing 5.2: Calculation of the parity bits for the hamming code counter measure

Figure 5.6 shows the new packet control lines. The hamming weight bits are added as MSBs. Leaving them in place would have resulted in a lot of changes in the NoC itself. The NoC itself expects all the bits including the OS bit in a very specific spot. Changing this order would have been a lot more work, than rearranging the bits.

Again only one encoder and decoder is implemented. They have do deal with the 32 bits of the data lines, which creates the same problem for the packet control lines. The problem was solved very similarly as in the last section.

5.2.4 Cyclic Redundancy Check

In order to have an already working implementation of a CRC a generator was used[1]. This already provides source code for a provided polynomial so that the development of

[1]OutputLogic.com

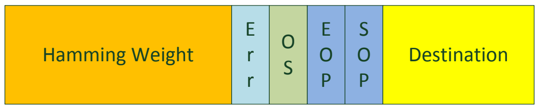


Figure 5.6: New packet containing information about the parity in the „Hamming Weight“ field of the packet.

the whole IP core can be skipped.

The polynome for the 8 bit CRC used is $CRC = 1 + x^4 + x^5 + x^8$ while the 32 bit CRC uses $CRC = 1 + x^1 + x^2 + x^4 + x^5 + x^7 + x^8 + x^{10} + x^{11} + x^{12} + x^{16} + x^{22} + x^{23} + x^{26} + x^{32}$. In order to store the internal state 8 bit registers are used. The first initialisation of the integers is with all logic highs.

The next CRC is calculated by calculating xors for each bit of the register input state variable (which is only saved during a clock high and if the CRC module is enabled). The calculation of those bits can be seen in listing 5.3.

```
lfsr_c(0) <= lfsr_q(0) xor lfsr_q(3) xor lfsr_q(4) xor lfsr_q(6) xor
             data_in(0) xor data_in(3) xor data_in(4) xor data_in(6);

lfsr_c(1) <= lfsr_q(1) xor lfsr_q(4) xor lfsr_q(5) xor lfsr_q(7) xor
             data_in(1) xor data_in(4) xor data_in(5) xor data_in(7);

lfsr_c(2) <= lfsr_q(2) xor lfsr_q(5) xor lfsr_q(6) xor data_in(2) xor
             data_in(5) xor data_in(6);

lfsr_c(3) <= lfsr_q(3) xor lfsr_q(6) xor lfsr_q(7) xor data_in(3) xor
             data_in(6) xor data_in(7);

lfsr_c(4) <= lfsr_q(0) xor lfsr_q(3) xor lfsr_q(6) xor lfsr_q(7) xor
             data_in(0) xor data_in(3) xor data_in(6) xor data_in(7);

lfsr_c(5) <= lfsr_q(0) xor lfsr_q(1) xor lfsr_q(3) xor lfsr_q(6) xor
             lfsr_q(7) xor data_in(0) xor data_in(1) xor data_in(3)
             xor data_in(6) xor data_in(7);

lfsr_c(6) <= lfsr_q(1) xor lfsr_q(2) xor lfsr_q(4) xor lfsr_q(7) xor
             data_in(1) xor data_in(2) xor data_in(4) xor data_in(7);

lfsr_c(7) <= lfsr_q(2) xor lfsr_q(3) xor lfsr_q(5) xor data_in(2) xor
             data_in(3) xor data_in(5);
```

Listing 5.3: Calculation of the 8 bit CRC counter measure. lfsr _q is the output while lfsr_c is the input of the register.

5.3 Implementation of the Fault Injection System

The whole implementation of the NoC and the fault injection system is done on a Xilinx Virtex 5 Evaluation Board (ml506) which is provided by the University. The board already comes with a PowerPC core which only needs to be configured for the needs of this thesis.

The following already provided peripherals are used besides the PowerPC CPU:

- plb_v46
- ppc440
- xps_bram_if_cntlr_bram
- xps_bram_if_cntlr
- DDR2_SDRAM
- jtagppc_cntrl_inst
- proc_sys_reset
- DIP_Switches_8Bit
- RS232_Uart_1
- RS232_Uart_2
- clock_generator
- several GPIO peripherals to connect the NoC

The only custom made peripheral is the aesnoc peripheral. It contains the NoC, the Access Nodes and the Fault Injection System. The aesnoc peripheral can communicate via the GPIO ports with the PowerPC and send and receive data.

To communicate with the NoC several GPIO modules are used. Table 5.4 shows the used GPIO peripherals, configuration and purpose. The GPIO peripheral allows data to be received as well as sent back. Therefore the aesnoc peripheral makes heavy use of this with its command processor. Control commands are received via the gpio_AES_cntrl peripheral. If it is a pure control command, then it is sent back on the same GPIO peripheral. On the other hand if it sends data to the NoC, the command is setd back via the gpio_AES_in GPIO.

The gpio_AES_out peripheral acts a little bit differently and creates an exception. Because it is needed to send the processed data back, the command is sent back via the gpio_AES_cntrl peripheral.

5.3.1 The Fault Injection System

The fault injection system used in this thesis is first introduced by Grinschgl et al. in [GKS+11]. It consists of a Fault Injection Controller (FIC), Triggers and Saboteur.

The FIC is connected directly to the gpio_FIC peripheral and is controlled directly by the firmware. To the FIC, triggers are connected. They are used to inform the FIC about the next action it should do according to its programming and if it should activate the saboteurs that are placed in the VHDL code.

GPIO Name	Address Start	Address End	Size	Purpose
gpio_AES_cntrl	0xA0002000	0xA0002FFF	4kB	Is used for control purposes. Commands from and to the NoC are send via this GPIO peripheral.
gpio_AES_in	0xA0000000	0xA0000FFF	4 kB	Data that is used for the processing of the AES algorithm.
gpio_AES_out	0xA0001000	0xA0001FFF	4 kB	Processed data from the NoC.
gpio_DEBUG	0xA0004000	0xA0004FFF	4 kB	For debugging purposes only.
gpio_AES_reset	0xA0003000	0xA0003FFF	4 kB	Reset of the NoC and its application.
gpio_FIC	0xA0008000	0xA0008FFF	4 kB	Fault Injection control mechanism.

Table 5.4: GPIO peripherals used on the FPGA board to communicate with the NoC.

5.4 The Firmware

The firmware itself consists of a few modules and resembles the commands from section 4.2.1. The firmware is used to talk to the NoC and send and receive the data.

The base layer of the modules is the NoC module. It contains the basic functionality to send commands to the NoC. It holds the basic structures for the GPIO modules that are implemented in the PowerPC core.

On top of the NoC module sits the AES module. It holds all the values from the NoC and as well as the key, plain and cypher text. The AES module also knows which commands it has to send via the NoC module to the NoC.

5.5 Attack Scenarios

5.5.1 Round Reduction

For the round reduction attack only one trigger is needed. This trigger has to listen on the arb_req line between the InitialRound and the first body round. The arb_req line is always activated when data is sent. This trigger is then used to notify the FIC about the beginning of the transaction and the FIC activates the saboteurs according to its programming. Figure 5.7 shows the placement of the trigger and the saboteurs together with the FIC. The saboteurs are only placed on the address lines of node_noc_pkt_cntrl. So it is possible to reroute the data to every node of the NoC.

Although there is a trigger used for the activation, no trigger for deactivating the saboteurs is used. Because the transmission of the data between the initial round and the first body round requires all transmitted data to be rerouted.

For this attack several saboteurs are placed. One on each address line, allowing a better configuration of the attack. This allows to reroute the data to each access node in the NoC.

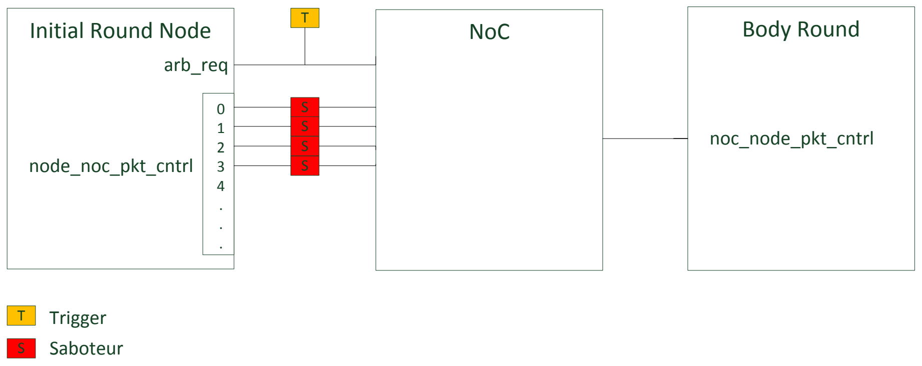


Figure 5.7: Placement of Trigger and Saboteur and their connection to the FIC for a round reduction attack on AES

5.5.2 Attacking the Key Schedule

The attack on the key schedule is a little bit more sophisticated. It needs more triggers and more saboteurs, compared to the attack implemented in section 5.5.1. This is because the attack on the key schedule is only possible if the timing is correct. The attack that is implemented needs to inject a fault in the third row key between the transmission from the eight to the ninth round. The placement of the triggers and saboteurs is shown in figure 5.8.

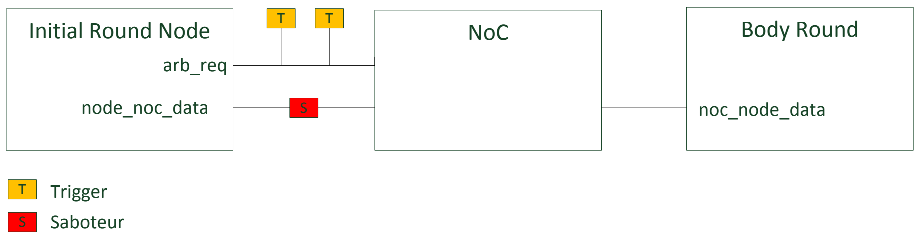


Figure 5.8: Placement of Trigger and Saboteur and their connection to the FIC to implement Takahashi et al.'s attack

To fulfil this task two triggers are placed in the design. One is used to activate the fault injection, while the other is used to deactivate the fault injection. The faults should only be injected in one packet of the transmission and not into all of them as in the previous attack.

This attack places the used saboteurs on the data lines. By placing them there they are able to inject faults into the data lines and alter the third row of the ninth round key, which will then also have some effects on the fourth row.

Finding the right packet is very easy with this system. The trigger already implements a counter that only needs to be configured correctly. As it is known that the seventh transmitted packet is the one containing the right key it is easy to set the counter to this value and let the trigger take care of it. If the trigger had not implemented an internal counter, this counter would have been another implementation that is necessary to implement.

Still having the counter implemented in the trigger has a few big advantages. First there is no need to implement the counter and second the counter is already configurable and can be set to any value.

5.6 Use Case Implementation: Implementing an AES Module with NoC

The original version of the AES code used in this thesis is provided by [Hem10] and can be downloaded from the OpenCores website at `http://opencores.org/project,aes_crypto_core`. It already provides a complete AES128 encoder and decoder. Although the implementation contains all the elements, the implementation only has one top level design where all the modules have to be extracted from. Thus the implementation provided by [Hem10] can only be used as a template.

In a first step the implementation is analysed to see which code represents the later modules and where they can be found in the source code. This is mainly done by looking at the code and verifying it by using a simulator to gain a better understanding of the code. After the code is fully understood, it is easy to separate the code in the desired modules.

With this knowledge the next step could be taken. The source is being split up into several modules that represent their functions in the AES specification [AES01].

The key_expander has already been a module of its own in the original code and is basically taken as it is, with a minor change inside the code. The other modules got an interface that fits their needs. In general they get all the data and keys needed and have the data as output again.

Now that all the original code is separated into modules, it is easy to arrange the modules in a way that each round of AES can be mapped onto an access node of the NoC. The sequence of the modules can be seen in figure 4.10, 4.11 and 4.12 from the design section of this thesis.

The modules are provided access to the NoC so that they can communicate with each other. They are connected directly to the send and receive lines of the access point. The data at the access points is there for at least one clock cycle. This is enough time to calculate the output and save it in the send module.

Chapter 6

Results

6.1 Testenvironment

The test environment consists of the FPGA[1] and a PC that is connected to the FPGA development board via a serial interface, which allows communication between the FPGA development board and the computer. Debug outputs and data can be sent from PC to the FPGA and vice versa. The computer does not need any additional software except for a terminal program that can access the serial interface of the PC. A detailed overview of the environment can be found in figure 6.1.

Inside the FPGA the NoC and a PowerPC processor can be found. The PowerPC processor is used to execute the test program and takes commands from the serial interface or write data to the serial interface for debugging purposes. The PowerPC is also responsible for sending commands to the NoC. These commands are processed by the CMD Proc module. It checks whether it is a valid command and if the data is available and sends it to the NoC. It is also possible to send data processed by the NoC back to the PowerPC. The PowerPC is also responsible to configure the Fault Injection Controller (FIC). The FIC is responsible to evaluate the data from the trigger and then activate or deactivate the saboteur inside the NoC.

All the analysis of the data is done on the PC. This keeps the PowerPC code fast and slim. Besides the FPGA can be used for other tasks and only the PC has to stay on during the analysis process. It is also possible to speed up the code of the evaluation software by using multiple CPUs for the calculation, which is another reason to shift the evaluation from the FPGA to the PC.

6.2 Comparison in Size

This section deals with the comparison of the different sizes of the counter measurements compared to the NoC and AES application without any counter measurements.

To get the results for this section, the program ISE from Xilinx [Xil] is used to synthesize the NoC and the AES. To speed up the generation of the results, the PowerPC is not synthesized into the project. The differences between the counter measures can be seen

[1]Xilinx Virtex IV `http://www.xilinx.com/support/index.html/content/xilinx/en/supportNav/silicon_devices/fpga/virtex-5.htm`, last visit January 2014

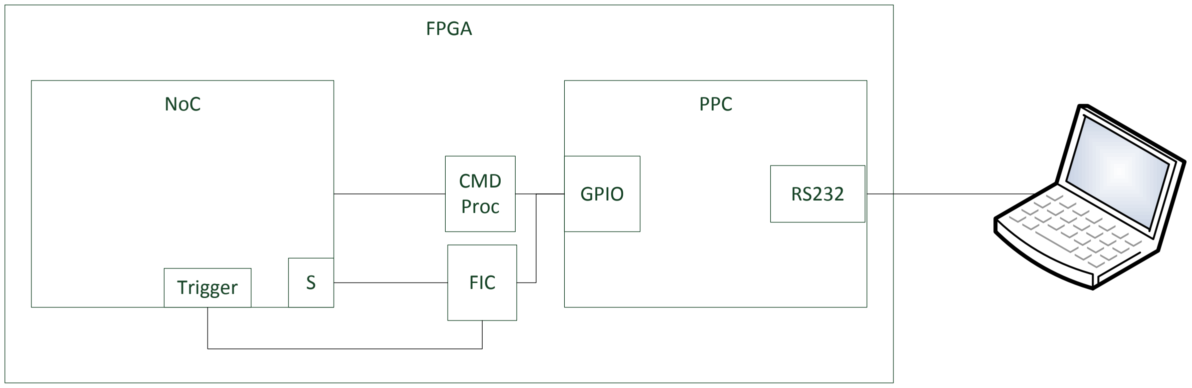


Figure 6.1: Testenvironment setup consisting of the NoC with triggers and saboteurs, control logic and PowerPC (PPC) and the connection to a PC via a serial interface

without the PowerPC core, too. With the PowerPC included in the project the synthesis project can take several hours as 99% of the FPGA are used.

Table 6.1 shows the results of the synthesis process for the NoC and AES implementation without the PowerPC core and is taken as a reference for all the other counter measurements. This shows that there is plenty of room for counter measurements.

	Available	Used	%
Slice Register	44800	18159	40.533
Slice LUTs	44800	28696	64.054
Used as Logic	44800	28696	64.054
Unused Flip Flop	33322	15163	45.504
Unused LUT	33322	4626	13.883
Fully used LUT-FF pairs	33322	13533	40.613
Block RAM/FIFO	148	36	24.324

Table 6.1: Synthesis result from ISE for the whole project without any counter measures installed (PowerPC is not included)

6.2.1 XOR-Table

The first counter measure implemented is the XOR-Table or parity bit. It is a relative small counter measure. Depending on the size of the data and packet lines it only uses a few XOR gates. Table 6.2 shows the results of the synthesis reports. In total three different systems were created: one with the counter measure on the data and packet lines, one with the counter measures on the data lines and a last one with the counter measures on the packet lines.

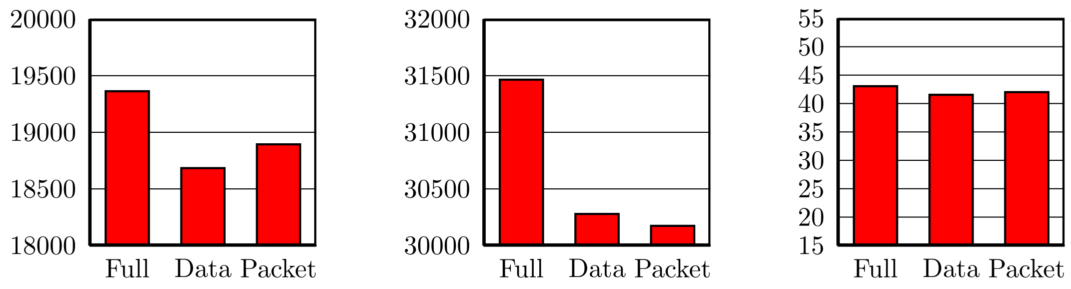
The interesting part can be seen in figure 6.2(a) and 6.2(c), because there are only a few packet lines and 32 data lines one would expect that the counter measure on the data

	Packet		Data		All	
	size	%	size	%	size	%
Slice Register	18903	42.194	18693	41.725	19372	43.241
Slice LUTs	30183	67.373	30288	67.607	31474	70.254
Used as Logic	30183	67.373	30288	67.607	31474	70.254
Unused Flip Flop	16112	46.015	16015	46.142	16710	46.311
Unused LUT	4832	13.800	4420	12.735	4608	12.771
Fully used LUT-FF pairs	14071	40.186	14273	41.123	14764	40.918
Block RAM/FIFO	36	24.324	63	24.324	36	24.324

Table 6.2: Synthesis result from ISE for the XOR table counter measure (PowerPC is not included)

lines would need more registers. In this case the implementation of the counter measure was not optimized for the packet lines. Thus the packet lines were extended to 32 bits, which happens on every access node.

Figure 6.2(b) shows the utilization of the LUTs available in the FPGA. Nothing really exciting could be seen here, except that the synthesizing tool has problems with the counter measures applied to both data and packet lines.



(a) Absolute registers used in the FPGA

(b) Absolute utilization of LUTs available in the FPGA

(c) Utilization of the FPGA

Figure 6.2: FPGA utilization with XOR-Table countermeasure.

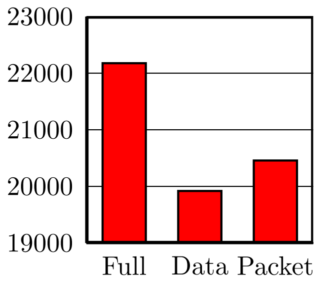
6.2.2 Hamming Codes

The second counter measure implemented was the Hamming Code counter measure. Table 6.3 shows the synthesis results of this counter measure. Again no special implementation for the packet lines was done, which can be seen in the synthesis results.

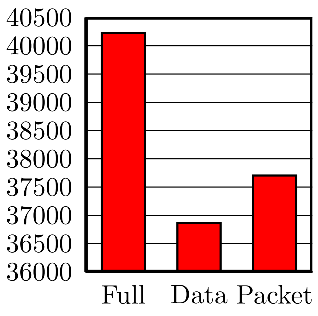
A reason for this can be found in the implementation, which was not really adjusted for the needs of the fewer packet lines that are available. The Hamming Weight implementation takes 32 bits of data. As the packet lines are usually smaller than 32 bits, the missing bits have to be inserted. This can explain the higher utilization when applying this counter measure on the packet lines only. This effect can be seen very well in figure 6.3(a), 6.3(b), 6.3(c).

	Packet		Data		All	
	size	%	size	%	size	%
Slice Register	20476	45.705	19938	44.504	22195	49.542
Slice LUTs	37723	84.203	36883	82.328	40246	89.835
Used as Logic	37723	84.203	36883	82.328	40246	89.835
Unused Flip Flop	22474	52.326	22139	52.615	23821	51.767
Unused LUT	5227	12.169	5194	12.344	5770	12.539
Fully used LUT-FF pairs	15249	35.504	14744	35.041	16425	35.694
Block RAM/FIFO	-	-	-	-	-	-

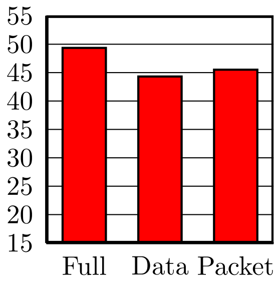
Table 6.3: Synthesis result from ISE for the Hamming Weight counter measure (PowerPC is not included)



(a) Absolute registers used in the FPGA



(b) Absolute utilization of LUTs available in the FPGA



(c) Utilization of the FPGA

Figure 6.3: FPGA utilization with Hamming Weight countermeasure.

6.2.3 Cyclic Redundancy Check

As a third counter measure CRC was implemented. Table 6.4 shows the synthesis results of the CRC counter measure. Compared to the Hamming Weight and XOR-Table some differences can be seen.

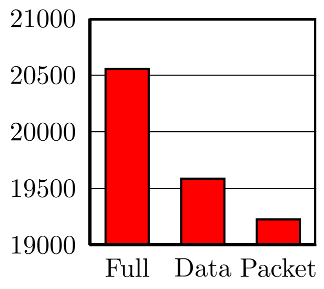
First, the synthesis of the counter measure just applied on the packet lines and on the data lines shows a picture that would have been expected. The difference this time is that two different implementations for the CRC are used. On the data lines a CRC-32 is used while on the packet lines only a CRC-8 is used.

Second, the counter measure applied on data and packet lines is very big. It uses more than 45% of the FPGAs resources. Together with the PowerPC this counter measure uses about 99.5% of the resources available on the FPGA and its synthesizing takes about 3 hours compared to the others which can be synthesized in about 30 minutes.

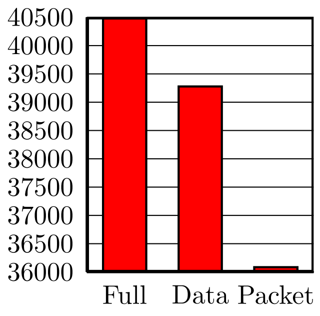
Again figures 6.4(a), 6.4(b), 6.4(c) give a graphical overview. It can be clearly seen that the counter measure only applied to data lines needs more space than only on the packet lines.

	Packet		Data		All	
	size	%	size	%	size	%
Slice Register	19235	42.935	19595	43.739	20564	45.902
Slice LUTs	36104	80.589	39297	87.717	40498	90.397
Used as Logic	36104	80.589	39297	87.717	40498	90.397
Unused Flip Flop	21857	53.190	24348	55.408	24854	54.723
Unused LUT	4988	12.139	4646	10.572	4920	10.832
Fully used LUT-FF pairs	14247	34.671	14949	34.019	15644	34.444
Block RAM/FIFO	-	-	-	-	-	-

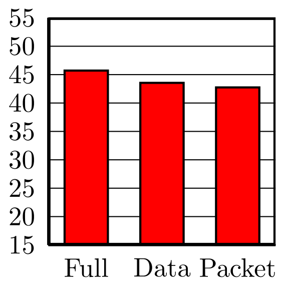
Table 6.4: Synthesis result from ISE for the CRC counter measure (PowerPC is not included)



(a) Absolute registers used in the FPGA



(b) Absolute utilization of LUTs available in the FPGA



(c) Utilization of the FPGA

Figure 6.4: FPGA utilization with CRC countermeasure.

6.2.4 Reed-Solomon Codes

Reed-Solomon codes are rather big. They were considered as a possible counter measurement, but unfortunately one set of encoder and decoder is already very big. Table 6.5 shows the utilization of one single set of encoder and decoder. It can be seen that one set nearly uses about 40% of the FPGA resources. Therefore this counter measure was not implemented due to the size of it.

Figure 6.5(a) shows that NoC and one single set of Reed-Solomon encoder and decoder nearly utilize the same amount of registers. This means that only one Reed-Solomon encoder and decoder can be fit into the whole design, requiring to give up all the benefits of the NoC, by routing all data and packet control lines through this one set of encoder and decoder. Figure 6.5(b) shows that there are plenty of LUTs available.

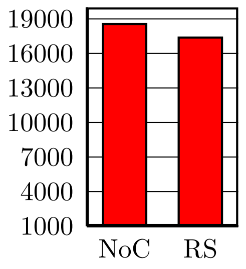
6.2.5 Secure Hash Algorithm (SHA)

Although this counter measure is very small (see table 6.6), it is not very suitable for the needs in this thesis. On one hand it needs 512 bits of data to calculate the SHA-3, on the other hand it takes very long to compute the SHA-3.

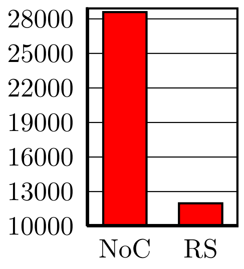
512 bits of data can hardly be provided. A total of 128 bits of data and 128 bits for

	Reed-Solomon	
	size	%
Slice Register	16978	37.897
Slice LUTs	12073	26.949
Used as Logic	12073	26.949
Unused Flip Flop	4786	21.990
Unused LUT	9691	44.578
Fully used LUT-FF pairs	7287	33.482
Block RAM/FIFO	2	1.351

Table 6.5: Synthesis result from ISE for the ReedSolomon encoder and decoder only



(a) Absolute registers used in the FPGA for NoC and ReedSolomon (RS)



(b) Absolute utilization of LUTs available in the FPGA for NoC and RS

Figure 6.5: FPGA utilization compared NoC with RS

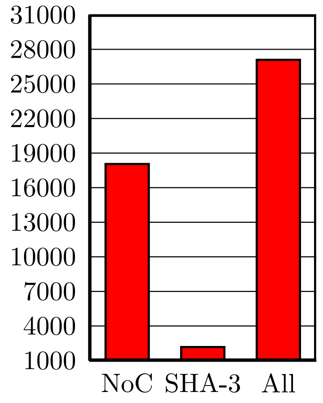
the key can be provided, but that would not be enough and the rest of 256 bits need to be filled in, which takes up additional space.

The column All in table 6.6 shows the outcome of the results for twelve SHA-3 cores used in the FPGA. This calculation was done by multiplying one SHA-3 core by twelve. Many fields cannot be calculated with this method, but the most important ones can, in order to get a rough estimation of the size to expect.

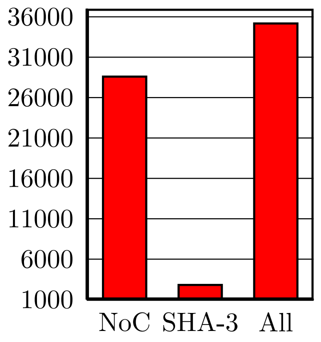
	SHA-3		All	
	size	%	size	%
Slice Register	2267	5.060	27204	60.720
Slice LUTs	2939	6.560	35268	78.720
Used as Logic	2938	6.558	35256	79.056
Unused Flip Flop	1282	36.123	-	-
Unused LUT	610	17.188	-	-
Fully used LUT-FF pairs	1657	46.689	-	-
Block RAM/FIFO	-	-	-	-

Table 6.6: Sythesis result from ISE for the SHA-3 encoder and decoder only

To prove this, figure 6.6(a) and 6.6(b) show that twelve of this SHA-3 cores take even more space than the NoC. Considering that the NoC already takes up more than 40% space without any counter measures and that the PowerPC takes up another 50% it can easily be seen that there is not enough space to fit twelve SHA-3 cores in there. Therefore it is not suitable in this NoC.



(a) Absolute registers used in the FPGA for NoC and SHA-4 (All refer to all the SHA-3 cores needed in the FPGA)



(b) Absolute utilization of LUTs available in the FPGA for NoC and SHA-3 (All refer to all the SHA-3 cores needed in the FPGA)

Figure 6.6: FPGA utilization compared NoC with SHA-3

6.2.6 Round Reduction Counter Measure

As a last counter measure this round reduction counter measure was implemented. This counter measure relies on the fact that each node knows which round of the AES it has to compute. Therefore this counter measure is only valid for this special application and probably not usable in a different scenario where no rounds or sequences exist.

Table 6.7 shows the synthesis results of this counter measure. This counter measure can only be applied to the nodes itself and not to the data and packet lines, those rows are removed from the table.

6.2.7 Direct Comparison of the Counter Measures

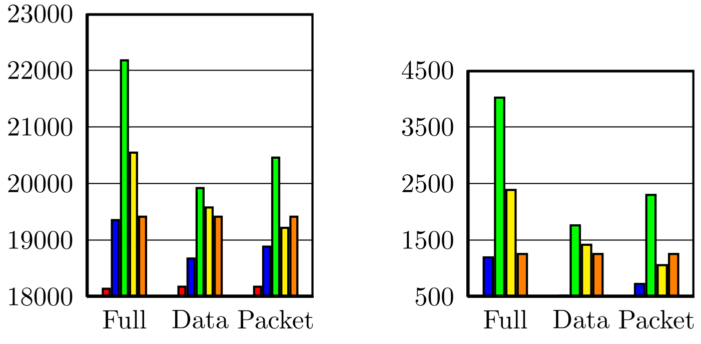
The direct comparisons in figure 6.7(a) and 6.7(b) show some interesting results. It was expected that the counter measures were deployed in a way that their size is ascending. Surprisingly it turns out that the Hamming Weight counter measure requires more space than the CRC counter measure.

The Hamming Weight counter measure needs more space on the FPGA than the CRC, which can be explained with the implementation of the two counter measures. While the Hamming Weight counter measure adds additional data and packet lines to the NoC, CRC only transmits an additional packet. All the additional data and packet lines have to be routed throughout the entire design and thus create this overhead.

	All	
	size	%
Slice Register	19432	43.375
Slice LUTs	32069	71.583
Used as Logic	32069	71.583
Unused Flip Flop	17688	47.651
Unused LUT	5051	13.607
Fully used LUT-FF pairs	14381	38.742
Block RAM/FIFO	36	24.324

Table 6.7: Synthesis result from ISE for the Round Reduction counter measure (PowerPC is not included)

This effect can also be seen in figure 6.8(a) and 6.8(b). Although the utilization of the LUTs shows that Hamming Weight and CRC counter measure nearly utilize the same amount of LUTs.



(a) Absolute values (b) Difference

Figure 6.7: Differences in registers utilized on the FPGA (red = no counter measure, blue = XOR Table, green = Hamming Weight, yellow = CRC, orange = Round Reduction)

Figure 6.7(a), 6.7(b), 6.8(a) and 6.8(b) also show that the counter measure against the round reduction needs almost as much space as the XOR-Table counter measurement applied to both data and packet control lines. Each node knows its exact round, it cannot be applied on data and packet control lines. Although the figures show this, the utilized registers and LUTs stay the same for this counter measurement.

For this special application it is a perfectly good counter measure. It is small and does not take that much more space. Although it is not possible to protect against Takahashi et al. [TFYpt] attack, still it may work well for a different application that is only based on different rounds that need to be watched over.

The rest of the results show the expected result. The XOR-Table only adds a few logic elements to each node, while the CRC counter measure needs more logic elements to compute the results.

Reed-Solomon and SHA-3 are not included in this section, because they are both too big to fit into the FPGA together with the NoC. For a detailed explanation see section

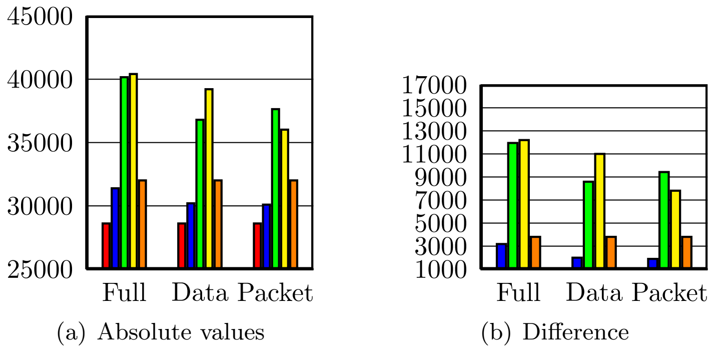


Figure 6.8: Differences in LUTs utilized on the FPGA (red = no counter measure, blue = XOR Table, green = Hamming Weight, yellow = CRC, orange = Round Reduction)

6.2.4 and section 6.2.5

6.3 Comparison in Speed

This speed comparison aims to show the differences in speed between the counter measures. One the one hand it will show how fast the whole NoC can be synthesized with the counter measure and on the other hand it will show how much effort, or how much overhead it takes to transmit the whole data (in terms of packets).

The maximum clock frequency (see table 6.8 is automatically determined during the synthesis process by Xilinx ISE. This should just give a rough estimation about how fast the NoC together with the applied counter measure can be theoretically run and if there are any counter measures that may slow the whole project down. However for the emulation together with the PowerPC only clock frequency of 25 MHz is used.

The second part then shows how much a particular counter measure slows down the transfer of packets over the network. In order to transmit a packet to compute AES, eight packets are used. A good counter measure shall not increase this value a lot.

Clock Frequency	MHz
All	143.625

Table 6.8: Clock frequency for NoC without counter measure.

6.3.1 XOR-Table

Again the first counter measure that is analysed is the XOR-Table or parity bit. Table 6.9 shows the maximum clock periods for this counter measure and figure 6.9 shows the comparison between them.

If the values are compared to table 6.8, it can easily be seen that the NoC can operate nearly at the same speed as without the counter measure. It also shows that it depends on the gates that are used how fast the counter measure can act. Each gate has a setup and hold time. Especially this counter measure has a cascade of logic gates and thus has to be a little bit slower.

Clock period	MHz
All	123.220
Data	143.972
Packet	143.805

Table 6.9: Clock period for NoC with XOR-Table counter measure.

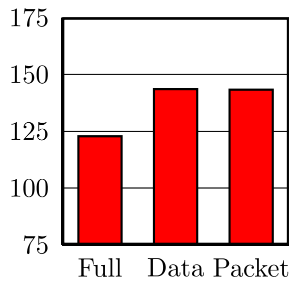


Figure 6.9: Timing measurements for XOR-Table counter measure generated with Xilinx ISE

Regarding the packets that have to be transmitted, this counter measure does not add any additional packets. The additional bits can easily be coded into the existing data and packet lines by extending them.

6.3.2 Hamming Weight

The Hamming Weight counter measure surprises with its speed. Table 6.10 shows that the maximum speed for the counter measure applied only on the packet lines can increase the clock period quite a bit. This may only be because the routing algorithms of Xilinx ISE can place the needed elements better in this case compared to the others. In total the counter measure is slower than the NoC without counter measures (table 6.8).

Figure 6.10 shows the comparison between the counter measure applied on either all data and packet lines, or either data or packet lines and shows kind of an expected result. The more registers and LUTs are used within this counter measurement, the slower it gets.

Clock period	MHz
All	128.811
Data	134.277
Packet	145.256

Table 6.10: Clock period for NoC with Hamming Weight counter measure.

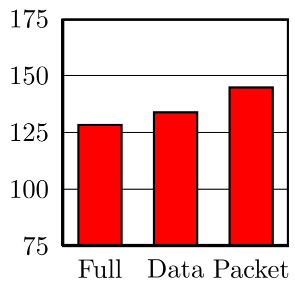


Figure 6.10: Timing measurements for Hamming Weight counter measure generated with Xilinx ISE

Again this counter measure does not need any additional packets to be transmitted through the network and the existing lines can be extended to transport the additional information. Although this makes the whole NoC a little bit bigger, it is still fast.

6.3.3 Cyclic Redundancy Check

This counter measure showed some unexpected clock period behaviours. It does not really depend on where this counter measure is applied to (data or packet lines, or both). The clock period stays nearly constant, which is good, compared to the other counter measures. Table 6.11 shows the absolute clock periods, while figure 6.11 shows the timings compared to each other.

Clock period	MHz
All	145.256
Data	145.256
Packet	143.061

Table 6.11: Clock period for NoC with CRC counter measure.

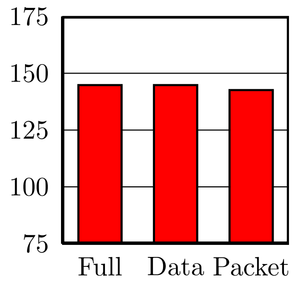


Figure 6.11: Timing measurements for CRC counter measure generated with Xilinx ISE

This counter measure unfortunately has to transmit an additional packet. Therefore the CRC can be calculated over all the data sent, which can increase the security of the NoC. CRC needs a clock, while the others don't, another packet needs to be added at the end and no intermediate values are transmitted.

The last packet contains the CRC information which is divided into two parts. 8 bits are reserved for the packet line CRC and the other bits are used for the CRC calculated on the data lines.

It doesn't matter that one additional packet is transmitted. It can be seen from the Hamming Weight counter measure that packing everything into the data and packet lines may not be the best solution.

6.3.4 Reed-Solomon Codes

Due to the fact that the Reed-Solomon code core already needs so much space, no further investigation into this topic was made. Only with a simulator the values could have been extracted, but then still issues that the simulator has would have to be taken into account and the question if this version of the NoC runs on real hardware, cannot be answered.

6.3.5 Secure Hash Algorithm (SHA)

Similar to section 6.3.4 no further investigation for this counter measure was done, although in this case it would be possible to fit enough cores into the NoC to calculate the SHA-3, it would take a very long time to transmit additional packets.

From [BDPA11] it can be estimated that SHA-3 needs at least 13 rounds to be computed. During this time no further data can be processed by the NoC. In a bigger application this could lead to a congestion very quickly.

Another problem is the size of data. SHA-3 needs 512 bits of data, which simply cannot be provided by the NoC with this application.

6.3.6 Round Reduction

Interesting results are shown by the round reduction counter measure. Table 6.12 shows the clock period for this counter measure. It is much faster than the actual NoC without any counter measures applied (compare with table 6.8). This may be because of the placement of the different cores. The cores are nearly the same, except they now know which round they belong to. Still some more registers have to be used to save this data, but unfortunately this can be synthesized better to create a faster NoC.

Clock period	MHz
All	150.928

Table 6.12: Clock period for NoC with round reduction check counter measure.

Nothing really interesting can be said about the packet transfer. The actual round is transmitted in the packet lines, only a comparison between the transmitted round and the round the core is aware of has to be made.

6.3.7 Direct Comparison of the Counter Measures

The comparison in figure 6.12 shows the very interesting results for the speed comparison. It shows the comparison between all counter measurement timings compared to the NoC.

It can be seen very clearly that the optimized CRC counter measure is as fast as the NoC without any counter measure and that the round reduction counter measure is a little bit faster than both of them.

Also the difference in the maximum clock period of the counter measure between XOR-Table, Hamming Weight and the NoC without counter measurements can be seen very well.

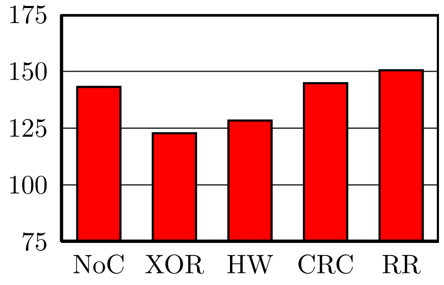


Figure 6.12: Overall overview of all timings in the counter measures, (XOR = XOR-Table, HW = Hamming Weight, CRC = Cyclic Redundancy Check, RR = Round Reduction

According to the additional packets, only the CRC counter measure introduces an additional packet that has to be transmitted, while all the other counter measurements can transmit the additional bits in the same packet.

6.4 Results of the Attacks

This section shows the results of the attacks and the counter measures. This makes it easier to compare the results.

The results are determined in the following way: First a set of plain text and key values is determined. This stays the same throughout the whole evaluation and does not change. Then the correct cypher text is calculated. This is the value that should be returned if the NoC worked properly.

After the determination of the cypher text, the attacks can be applied one by one to the NoC. Their results are then saved and run through the decypher tool to see if the secret key can be retrieved or not. It would be tedious to do this for all the values that are received, especially since they are mostly the same.

It is also necessary to see that the counter measure does not have any effects on the calculation. Therefore each sub section contains some one run with the data and no counter measure applied.

4C	69	74	74
6C	65	20	6D
69	73	73	20
6D	75	66	66

Table 6.13: Plain text

00	01	02	03
04	05	06	07
08	09	0A	0B
0C	0D	0E	0F

Table 6.14: Key

ac	22	83	b4
a9	7b	7f	51
7f	2f	a3	19
73	a4	17	e4

Table 6.15: Cypher text

Table 6.13 and 6.14 show the used plain text and key. Table 6.15 shows the encrypted data. The plain text and the key are used in all evaluations. The result from the last table is the expected result. This result can be retrieved if no attacks or faults have happened.

a0	00	ea	09
9d	7f	63	5b
fa	60	5d	5b
da	3d	21	9f

Table 6.16: Cypher text with activated round reduction attack

7b	1f	e4	e3
7e	6d	d5	cd
e4	16	a7	8e
11	99	56	08

Table 6.17: Cypher text with attack on the 9th round key schedule

In order to have comparable data, the attacks were applied to the NoC and the application started. The results of the round reduction attack are found in table 6.16 and the results of the attack on the 9th round key schedule are found in table 6.17. Once again both tables are used as reference for all the counter measures.

6.4.1 XOR-Table

The results of the the XOR-Table contain no real surprise. A simple parity bit can only detect one bit error. All of the attacks change more than one bit and therefore it is very hard to detect an error with this method.

ac	22	83	b4
a9	7b	7f	51
7f	2f	a3	19
73	a4	17	e4

Table 6.18: Cypher text (data lines, no attack)

ac	22	83	b4
a9	7b	7f	51
7f	2f	a3	19
73	a4	17	e4

Table 6.19: Cypher text (packet lines, no attack)

Table 6.18 shows the counter measure applied on the data lines and table 6.19 shows the counter measure applied on the packet lines, both with no attack. The result is in table 6.15. This means that the counter measure does not have any effects on the NoC if it is run normally.

a0	00	ea	09
9d	7f	63	5b
fa	60	5d	5b
da	3d	21	9f

Table 6.20: Cypher text (packet lines, round reduction)

7b	1f	e4	e3
7e	6d	d5	cd
e4	16	a7	8e
11	99	56	08

Table 6.21: Cypher text (packet lines, 9th round key schedule)

Table 6.20 and 6.21 show an expected behaviour. The cypher text contains errors and is similar to the results in table 6.16 and 6.17. The attack itself changes more than one bit, which cannot be detected by this counter measure. The counter measure thinks that everything was fine and therefore this counter measure does not work with the round reduction attack.

a0	00	ea	09
9d	7f	63	5b
fa	60	5d	5b
da	3d	21	9f

Table 6.22: Cypher text (data lines, round reduction)

7b	1f	e4	e3
7e	6d	d5	cd
e4	16	a7	8e
11	99	56	08

Table 6.23: Cypher text (data lines, 9th round key schedule)

Very similar are the results for the counter measure applied to the data lines only. Table 6.22 and 6.23 show that the same values are calculated as before and again the counter measure cannot detect whether an error occurred or not.

a0	00	ea	09
9d	7f	63	5b
fa	60	5d	5b
da	3d	21	9f

Table 6.24: Cypher text (packet and data lines, round reduction)

7b	1f	e4	e3
7e	6d	d5	cd
e4	16	a7	8e
11	99	56	08

Table 6.25: Cypher text (packet and data lines, 9th round key schedule)

Although already proven that it is not possible to protect the NoC against an attack with this counter measure it is shown that it is not possible if counter measures are applied on data and packet lines at the same time. The result can be seen in table 6.24 and 6.25.

6.4.2 Hamming Codes

The Hamming Weight works very similarly to the parity bit in the previous section. Only this time several parity bits are calculated. Table 6.26 and 6.27 show the results of the normal run without any attacks.

The results for the Hamming Weight show some interesting behaviour. It was not possible to run it completely on the FPGA. It would always freeze. Each synthesis process showed a different unwanted behaviour. This may be due to some timing constraints that were not met and some signals need longer than expected and thus the NoC is not able to process the data correctly.

On the other hand, the simulation does not show this behaviour. Therefore the results are generated from the FPGA as well as they could be created. Only the cypher text with counter measures on data lines and packet lines had to be gathered from the simulation.

ac	22	83	b4
a9	7b	7f	51
7f	2f	a3	19
73	a4	17	e4

Table 6.26: Cypher text (data lines, no attack)

ac	22	83	b4
a9	7b	7f	51
7f	2f	a3	19
73	a4	17	e4

Table 6.27: Cypher text (packet lines, no attack)

a0	00	ea	09
9d	7f	63	5b
fa	60	5d	5b
da	3d	21	9f

Table 6.28: Cypher text (packet lines, round reduction)

7b	1f	e4	e3
7e	6d	d5	cd
e4	16	a7	8e
11	99	56	08

Table 6.29: Cypher text (packet lines, 9th round key schedule)

Unfortunately this counter measure shows an interesting behaviour on the data lines. The error cannot be detected and so the data can be extracted as in the comparison tables. Table 6.28 and 6.29 show the data of this experiment.

00	00	00	00
00	00	00	00
00	00	00	00
00	00	00	00

Table 6.30: Cypher text (packet lines, round reduction, generated with ModelSim)

It can be seen by comparing table 6.30, which shows the results of the counter measure calculated with the simulator and table 6.28, where the results are calculated on the FPGA. This may be due to timing issues on the FPGA that the simulator does not handle yet. Still this would require some more investigation. It shows that the results can be calculated correctly.

a0	00	ea	09
9d	7f	63	5b
fa	60	5d	5b
da	3d	21	9f

Table 6.31: Cypher text (data lines, round reduction)

00	00	00	00
00	00	00	00
00	00	00	00
00	00	00	00

Table 6.32: Cypher text (data lines, 9th round key schedule)

As it comes to the counter measure on the data lines, it is clear why it can't help if the attack is on the packet lines. Table 6.31 shows clearly that data can be retrieved and that it contains the data from table 6.15.

On the other hand table 6.32 shows the results of the counter measure to the data lines with an attack on the key schedule, which happens to transfer data on the data lines. Thus the attack can be detected and the corrupted data can be sorted out. It is just for security reasons that the NoC returns all zeros.

a0	00	ea	09
9d	7f	63	5b
fa	60	5d	5b
da	3d	21	9f

Table 6.33: Cypher text (packet and data lines, round reduction)

00	00	00	00
00	00	00	00
00	00	00	00
00	00	00	00

Table 6.34: Cypher text (packet and data lines, 9th round key schedule)

Last but not least the counter measure is applied to both data and packet lines. This shows exactly the actually expected result. Table 6.33 only has the packet lines attacked at an early stage and thus it has been seen in table 6.28 that it is not possible to protect the data.

If the data is processed until the attack of the 9th round key, the counter measure again behaves as expected. The results are shown in table 6.33.

6.4.3 Cyclic Redundancy Check

The CRC shows the expected behaviour with the counter measure on data and packet lines. Table 6.35 and 6.36 show that the values are correct and prove that the counter measure does not have any effects on the data to be processed.

ac	22	83	b4
a9	7b	7f	51
7f	2f	a3	19
73	a4	17	e4

Table 6.35: Cypher text (data lines, no attack)

ac	22	83	b4
a9	7b	7f	51
7f	2f	a3	19
73	a4	17	e4

Table 6.36: Cypher text (packet lines, no attack)

00	00	00	00
00	00	00	00
00	00	00	00
00	00	00	00

Table 6.37: Cypher text (packet lines, round reduction)

7b	1f	e4	e3
7e	6d	d5	cd
e4	16	a7	8e
11	99	56	08

Table 6.38: Cypher text (packet lines, 9th round key schedule)

The results for the counter measure applied on the packet lines together with the round reduction attack show the expected behaviour. Table 6.37 shows the expected result of the attack, the output is all zeros.

Table 6.38 shows the same results but this time the key schedule is attacked. This attack does not have any effects on the packet lines and thus it is not possible to detect the injected fault and the NoC computes the wrong cypher text.

a0	00	ea	09
9d	7f	63	5b
fa	60	5d	5b
da	3d	21	9f

Table 6.39: Cypher text (data lines, round reduction)

00	00	00	00
00	00	00	00
00	00	00	00
00	00	00	00

Table 6.40: Cypher text (data lines, 9th round key schedule)

The results in table 6.39 are similar to the ones presented before. Due to the fact that the synthesis didn't work that well the results generated for the counter measure just on the data lines are equal to the ones generated for just the counter measure on the packet lines.

The cypher text in table 6.40 shows exactly the expected behaviour. The counter measure worked for the data lines and so the injected fault can be detected.

00	00	00	00
00	00	00	00
00	00	00	00
00	00	00	00

Table 6.41: Cypher text (packet and data lines, round reduction)

00	00	00	00
00	00	00	00
00	00	00	00
00	00	00	00

Table 6.42: Cypher text (packet and data lines, 9th round key schedule)

The counter measure applied on the data and packet lines only concludes the previous results. Table 6.42 and 6.41 show the results of the counter measure with both kinds of attacks.

6.4.4 Round Reduction Counter Measure

This result is just for completeness. The result for keeping track of the round number is in table 6.43. Again the counter measure does not have any effects on the proper calculation of the result.

ac	22	83	b4
a9	7b	7f	51
7f	2f	a3	19
73	a4	17	e4

Table 6.43: Cypher text (packet lines, no attack)

00	00	00	00
00	00	00	00
00	00	00	00
00	00	00	00

Table 6.44: Cypher text (packet lines, round reduction)

7b	1f	e4	e3
7e	6d	d5	cd
e4	16	a7	8e
11	99	56	08

Table 6.45: Cypher text (packet lines, 9th round key schedule)

The results for this counter measure and the attack on the packet lines work as expected. The results in table 6.44 show exactly this result. It can be seen that the round number already protects very well against this attack.

It is also very clear that this attack cannot help against an attack on the round key schedule. This is documented in table 6.45. Therefore it is a very valid result and shows that this counter measure works for this specific reason.

Again it has to be mentioned that this counter measure is just here for completeness. It should only show that even easy looking counter measures can prevent the stealing of data and that it depends heavily on the case how they shall be used or which level of security is needed.

6.4.5 Reconstructing the Secret Key in One Round AES

The algorithm to reconstruct the secret key is proposed by Bouillaguet et al. in [BDD+10]. The reconstruction is easier with two known plain- and cyphertext pairs another pair is generated. Table 6.46 shows the new plaintext and table 6.47 shows the round reduced version.

9b	b4	36	08
73	f1	0a	3f
c7	03	c4	2f
62	30	71	73

Table 6.46: 2nd plain text

bf	4d	e2	d5
73	d2	22	1d
a3	4d	ec	45
5f	aa	81	03

Table 6.47: 2nd cypher text

In the first step the input difference is calculated. The result of this operation is found in table 6.48.

The output difference in table 6.49 is the Sbox difference of the output. Therefore the difference between the two cypher texts is calculated followed by $SR^{-1} \circ MC^{-1}$. This allows to handle each Sbox independently and the problems with the MixColumn can be overcome. By shifting the rows to their original places only the Sbox needs to be examined later on and no difficult calculations have to be made.

Now the 2^8 possible input values to create the input difference need to be found. After they are found search through the 2^8 values to find out which values can create the Sbox difference, by subsituting the input values by their Sbox equivalent and XORing those values. This should result in two suggestions for each byte of k_0. The key consists of sixteen bytes and two values for each byte are available which means that only 2^{16} train encryptions would be needed in the worst case to retrieve the secret key shown in table 6.14.

d7	dd	42	7c
1f	94	2a	52
ae	70	b7	0f
0f	45	17	15

Table 6.48: Input difference

3d	90	20	de
b0	6f	09	05
65	bd	3d	c7
70	9b	97	e9

Table 6.49: Sbox difference

6.4.6 Reconstructing the 9th Round Key

In order to prove the counter measures to work correctly it can be tried to reconstruct the 9th round key. In order to do so, the calculations provided in [TFYpt] have to be executed. To be able to retrieve the key correctly more than one result is needed. This second result has to use different faults than the first one. The first result was created by setting all values of the third round key to 0xFF. Thus another result must not use this values for the third round key. Then the unique keys as proposed in [TFYpt] can be calculated without any problems. If no second result exists, the complexity of this problem rises. A single calculated key can provide two results for the 9th round key but only one of them is the correct one. Still it would be possible to construct the correct key by guessing and brute force but it will be more time consuming.

With all the implemented counter measures the 9th round attack from Takahashi can be partly avoided and only parts of the secret key can be extracted rather than the whole one.

00	01	02	03
04	05	06	07
08	09	0A	0B
0C	0D	0E	0F

Table 6.50: Key used to create results

54	99	32	d1
f0	85	57	68
0f	7a	a8	97
a1	c5	d2	45

Table 6.51: Corrupted key in the 9th round (Red shows the injected fault, while yellow is just a continuous effect of the error)

54	99	32	d1
f0	85	57	68
0f	7a	a8	97
be	c5	97	4e

Table 6.52: Calculated key for the 9th round (Red fields show the recalculated 9th round key)

The table 6.50 shows the original key that is used for the cypher algorithm. No modifications have been made here so far. It is the standard key.

Table 6.51 shows the change of the values in the 9th round after the attack. Notice that the column in red is the one attacked. The yellow one is just the error distribution due to the key expansion algorithm of AES.

Last but not least table 6.52 shows the calculated values for the 9th round key. It is not that difficult any more to calculate the rest of the values, but unfortunately it should just show that the error has an effect and that the key can be reconstructed. Cells coloured in red are the values that could be reconstructed using [TFYpt].

As a matter of fact it is not that difficult to reconstruct the original key whenever the last subkey has been correctly calculated. Dusart et al. describe how to do this in [DLV03].

Chapter 7

Conclusion and Future Work

Security in NoCs is becoming more and more important as NoCs are being used for more security relevant applications. Due to the fact that NoCs are usually designed without counter measures it is easily possible to attack them right now.

This thesis has shown that it is possible to secure an NoC against fault induced attacks by applying different types of countermeasures on the network adapter and therefore check whether the data has been corrupted or not. Therefore, counter measures like a parity bit up to the complexity of an secure hash algorithm have been examined.

A problem that arises when it comes to counter measures is the fact, that they add overhead to each node of the NoC which then can make the whole system undesirably big or slow. Depending on the counter measurement. Counter measures without any registers would only add up in size, but most likely not in speed (due to the propagation delay time of the logic elements), while a counter measure that has to use registers would also need more time to calculate.

The different countermeasures have been evaluated for size and their speed. It could be seen that countermeasures that may sound promising in the first place were not as good as expected and that probably adding an additional transmission cycle for data may be helpful quite a lot.

Unfortunately some counter measures were not able to fit into the NoC. After a first evaluation round they used too much space already. Therefore they couldn't be emulated on the FPGA due to its limited size.

With all the implemented counter measures the 9th round attack from Takahashi can be partly avoided and only parts of the secret key can be extracted rather than the whole one.

The results of the 9th round attack are just for a prove of concept. The tool can only crack a few values of the key but not all of them. It would be nice to implement the rest of the attack to get the whole key out of it, but for a prove of concept just parts of the key are good enough.

As future work the NoC extension should be improved to discard packages where a fault has been detected. Also different use cases should be evaluated to improve the designed and implemented counter measures.

Appendix A

Appendix A

Proove of Connections in a Mesh NoC

In order to prove that the number of connections in a NoC are smaller or equal to the number of connections. Therefore two different approaches do exist. The NoC examined has a mesh layout and consists of $n \times m$ nodes. The first approach in equation A.4 defines that each router only has half the connections. Routers usually share a connection. If they are split up and each router has one half of the connection the following observations can be made: First the corners of the NoC are examined. A corner only has two full connections and four corners do exist within a mesh layout. Thus the number of connections in the corners can be defined as

$$CornerNode = 4 \cdot \frac{2}{2} \tag{A.1}$$

Next the surrounding nodes are examined. It shows that a node that is not a corner block, but at the border of the NoC has three connections and is mirrored on the other side. In one direction only $n-2$ nodes with three connections can exist. The two nodes substracted are the corner nodes. Thus the following equation can be defined

$$BorderNode = 2 \cdot \frac{3}{2} \cdot (n-2) \tag{A.2}$$

The same observation is made for the m direction.

Finally the inner core is examined. Each node here has four connections and in n and m direction the border, which is 2, needs to be substracted. This can be expressed in the following equation

$$CoreNode = \frac{4}{2} \cdot (n-2) \cdot (m-2) \tag{A.3}$$

Figure A.1 shows how the half connections are used. Now those three equations for corners, borders and core nodes are put together, which leads to the equation in A.4.

$$(n-2) \cdot (m-2) \cdot \frac{4}{2} + (n-2) \cdot \frac{3}{2} \cdot 2 + (m-2) \cdot \frac{3}{2} \cdot 2 + 4 \cdot \frac{2}{2} \tag{A.4}$$

Another approach would be to count the number of connections differently. Figure A.2 shows this count strategy. A node is defined with only two connections. One in n and the other one in m direction.

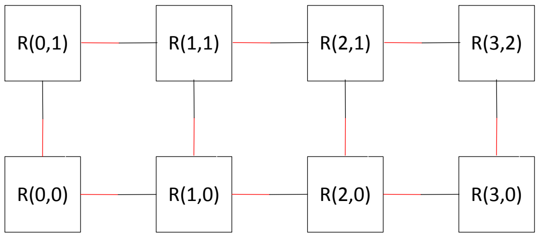


Figure A.1: Count strategy for node connections (half connections). To show one half connection the line between to nodes has two colors.

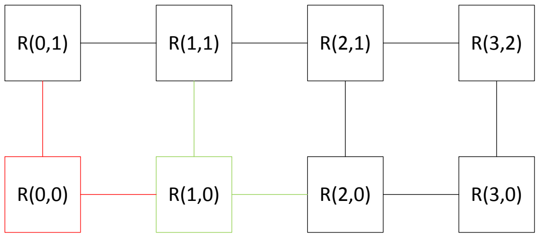


Figure A.2: Count strategy for node connections. Red and green show the example of the counting. This leaves only a few connections out.

If started with a network that has 1×0 or 1×1 nodes it can be seen that there are no nodes that have two connections. This allows the assumption that there are only

$$Connections = (n-1) + (m-1) \tag{A.5}$$

in the network. Now a bigger network with 2×2 nodes is examined. In n direction only one node exists that has the two connections that should be counted. By rotating the search pattern, the same applies to the m direction. Therefore the number of such nodes can be extracted as

$$Nodes = (n-1) \cdot (m-1) \tag{A.6}$$

To get the number of connections the equation needs to multiplied by 4. Now the equation is still not correct. If an 3×2 network is examined, the number of calculated connections would be too big. Again counting the nodes that apply to the counting rules should help. It can be seen that there are $n-1$ nodes that apply in n direction and only one node (again $m-1$) in m direction. If the formula $(n-1) \cdot (m-1) \cdot 4$ is applied the result is 8 connections. However, the number of connections in a 3×2 network is 7.

In order to get the correct result the approach is changed a little bit. Again the same counting method is applied, but now the nodes with two connections are counted again and multiplied by the number of connections. Now in the n direction exactly $n-1$ connections and in m direction exactly $m-1$ connections have not been counted. If everything is put together the following formula can be created:

$$(n-1)\cdot(m-1)\cdot 2+(n-1)+(m-1) \tag{A.7}$$

In order two show that the two thesis are equal the two equations in A.4 and A.7 are simplified and the result compared. See the calculation in A.8.

$$\begin{gathered}(n-1)\cdot(m-1)\cdot 2+(n-1)+(m-1)=\\ =(n-2)\cdot(m-2)\cdot\frac{4}{2}+(n-2)\cdot\frac{3}{2}\cdot 2+(m-2)\cdot\frac{3}{2}\cdot 2+4\cdot\frac{2}{2}\\ 2\cdot n\cdot m-2n-2m+4+n+m-2=2\cdot n\cdot m-4\cdot n-4\cdot m+8+3\cdot n+3\cdot m-12+4\\ 2\cdot n\cdot m-n-m=2\cdot n\cdot m-n-m\end{gathered} \tag{A.8}$$

It can be seen that the two equations are equal and that either of them can be used to represent the number of connections in a mesh type network.

Proove that a NoC always has less Connections than a full Mesh

In order to show that a NoC has less connections than a full mesh the upper boundary of the growth is shown. As it is already known the NoC always has $2\cdot n\cdot m-n-m$ connections. Therefore the upper boundary of a NoC would be

$$f(NoC)\in\mathcal{O}(n\cdot m) \tag{A.9}$$

On the other side the full mesh grows with

$$f(Mesh)=\frac{(n\cdot m)(n\cdot m-1)}{2}=frac(n\cdot m)^2-n\cdot m2\rightarrow f(Mesh)\in\mathcal{O}((n\cdot m)^2) \tag{A.10}$$

In a final conflusion it can be determined that the number of connections in the NoC will always be smaller than the number of connections in the mesh.

$$f(NoC)<f(Mesh) \tag{A.11}$$

Bibliography

[AES01] Federal information processing standards publication (FIPS 197). Advanced Encryption Standard (AES), 2001.

[BBM05] Davide Bertozzi, Luca Benini, and Giovanni De Micheli. Error control schemes for on-chip communication links: the energy-reliability tradeoffs. *IEEE TCAD*, 24(6):2005, 2005.

[BDD$^+$10] Charles Bouillaguet, Patrick Derbez, Orr Dunkelman, Nathan Keller, Vincent Rijmen, and Pierre-Alain Fouque. Low data complexity attacks on aes. *IACR Cryptology ePrint Archive*, 2010:633, 2010.

[BDM01] Luca Benini and Giovanni De Micheli. Powering networks on chips: Energy-efficient and reliable interconnect design for socs. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 33–38, New York, NY, USA, 2001. ACM.

[BDPA11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The keccak reference. Submission to NIST (Round 3), 2011.

[BM06] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1), June 2006.

[BPC98] Jrome Bou, Philippe Ptillon, and Yves Crouzet. Mefisto-l: A vhdl-based fault injection tool for the experimental assessment of fault tolerance. In *FTCS*, pages 168–173. IEEE Computer Society, 1998.

[CDBZ99] M. Cuviello, S. Dey, Xiaoliang Bai, and Yi Zhao. Fault modeling and simulation for crosstalk in system-on-chip interconnects. In *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on*, pages 297 –303, 1999.

[CG94] Robert Cypher and Luis Gravano. Storage-efficient, deadlock-free packet routing algorithms for torus networks. *IEEE Trans. on Computers*, 43:1376–1385, 1994.

[dARGG06] David de Andrs, Juan Carlos Ruiz, Daniel Gil, and Pedro J. Gil. Fades: a fault emulation tool for fast dependability assessment. In George A. Constantinides, Wai-Kei Mak, Phaophak Sirisuk, and Theerayod Wiangtong, editors, *FPT*, pages 221–228. IEEE, 2006.

[DLPP05] Jos Duato, Olav Lysne, Ruoming Pang, and Timothy M. Pinkston. Part i: A theory for deadlock-free dynamic network reconfiguration. *IEEE Transactions on Parallel and Distributed Systems*, 16(5):412–427, 2005.

[DLV03] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on a.e.s. *IACR Cryptology ePrint Archive*, 2003:10, 2003.

[DT01] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *DAC*, pages 684–689. ACM, 2001.

[FKCC06] Arthur Pereira Frantz, Fernanda Lima Kastensmidt, Luigi Carro, and Érika Cota. Evaluation of seu and crosstalk effects in network-on-chip switches. In *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design*, SBCCI '06, pages 202–207, New York, NY, USA, 2006. ACM.

[GKS$^+$11] Johannes Grinschgl, Armin Krieg, Christian Steger, Reinhold Weiss, Holger Bock, and Josef Haid. Modular fault injector for multiple fault dependability and security evaluations. In *Proceedings of the 2011 14th Euromicro Conference on Digital System Design*, DSD '11, pages 550–557, Washington, DC, USA, 2011. IEEE Computer Society.

[Ham50] R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Techincal Journal*, 29:147–160, 1950.

[Hem10] Satyanarayana Hemanth. Aes128 `http://opencores.org/project,aes_crypto_core`, last visit august 2013. [online], October 2010.

[KCR06] Fernanda Lima Kastensmidt, Luigi Carro, and Ricardo Reis. *Fault-Tolerance Techniques for SRAM-Based FPGAs (Frontiers in Electronic Testing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[KGS$^+$12] Armin Krieg, Johannes Grinschgl, Christian Steger, Reinhold Weiss, Holger Bock, and Josef Haid. Power-modes: Power-emulator- and model-based dependability and security evaluations. *ACM Trans. Reconfigurable Technol. Syst.*, 5(4):19:1–19:21, December 2012.

[KKD10] Young Hoon Kang, Taek-Jun Kwon, and J. Draper. Fault-tolerant flow control in on-chip networks. In *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*, pages 79–86, 2010.

[KSR10] A. Kohler, G. Schley, and M. Radetzki. Fault tolerant network on chip switching with graceful performance degradation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(6):883–896, 2010.

[LC04] Shu Lin and Daniel J. Costello. *Error Control Coding, Second Edition.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[LPD05] Olav Lysne, Timothy Mark Pinkston, and Jos Duato. Part ii: A methodology for developing deadlock-free dynamic network reconfiguration processes. *IEEE Trans. Parallel Distrib. Syst.*, 16(5):428–443, 2005.

[MACM11] Aline Mello, Alexandre Amory, Ney Calazans, and Fernando Moraes. A noc generation and evaluation framework. DATE Conference, 2011.

[MCM$^+$] F. Moraes, N. Calazans, A. Mello, L. Mller, and E. Moreno. Atlas - an environment for noc generation and evaluation `https://corfu.pucrs.br/redmine/projects/atlas` , last visit january, 19th 2014. [online].

[MCM$^+$04] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integr. VLSI J.*, 38(1):69–93, October 2004.

[NIS] NIST. Sha-3 keccak `http://csrc.nist.gov/groups/ST/hash/sha-3/sha-3_standardization.html`, visited january 2014. [online].

[PW72] W.W. Peterson and E.J. Weldon. *Error-correcting Codes*. MIT Press, 1972.

[RLP06a] V. Rantala, T. Lehtonen, and J. Plosila. *Network on Chip Routing Algorithms*. TUCS technical report. Turku Centre for Computer Science, 2006.

[RLP06b] Ville Rantala, Teijo Lehtonen, and Juha Plosila. Network on chip routing algorithms, 2006.

[RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *J. Soc. Indust. Appl. Math.*, 8:300–304, 1960.

[SG04] G. Schelle and D. Grnwald. Onchip interconnect exploration for multicore processors utilizing fpgas. In *Proceedings of 2nd Workshop on Architecture Research using FPGA Platforms*, 2004.

[SKDH08] Erno Salminen, Ari Kulmala, and Timo D. Hamalainen. Survey of network-on-chip proposals. March 2008.

[SKH] Erno Salminen, Ari Kulmala, and Timo D. Hmlinen. On network-on-chip comparison.

[SSR12] Luca Sterpone, Davide Sabena, and Matteo Sonza Reorda. A new fault injection approach for testing network-on-chips. In Rainer Stotzka, Michael Schiffers, and Yannis Cotronis, editors, *PDP*, pages 530–535. IEEE, 2012.

[TFYpt] J. Takahashi, T. Fukunaga, and K. Yamakoshi. Dfa mechanism on the aes key schedule. In *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*, pages 62–74, Sept.

[Xil] Xilinx. Xilinx virtex iv `http://www.xilinx.com/support/index.html/content/xilinx/en/supportNav/silicon_devices/fpga/virtex-5.htm`, last visit january 2014. [online].

[YBLB09] Yoon Seok Yang, Jun Ho Bahn, Seung Eun Lee, and Nader Bagherzadeh. Parallel and pipeline processing for block cipher algorithms on a network-on-chip. In Shahram Latifi, editor, *ITNG*, pages 849–854. IEEE Computer Society, 2009.

[ZAV04] Haissam Ziade, Rafic A. Ayoubi, and Raoul Velazco. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186, 2004.