BCH Codes mit kombinierter Korrektur und Erkennung

Doktorarbeit

von

M.Sc. Christian Schulz-Hanke

Universität Potsdam
Institut für Informatik und Computational Science
Arbeitsgruppe Fehlertolerantes Rechnen

Aufgabenstellung und Betreuung:
Prof. Dr. Michael Gössel

Potsdam, den 10. Dezember 2023

Soweit nicht anders gekennzeichnet, ist dieses Werk unter einem Creative-Commons-Lizenzvertrag Namensnennung 4.0 lizenziert.
Dies gilt nicht für Zitate und Werke, die aufgrund einer anderen Erlaubnis genutzt werden. Um die Bedingungen der Lizenz einzusehen, folgen Sie bitte dem Hyperlink:
https://creativecommons.org/licenses/by/4.0/legalcode.de

Schulz-Hanke, Christian
christian.schulz-hanke@cs.uni-potsdam.de
BCH Codes mit kombinierter Korrektur und Erkennung
Doktorarbeit, Institut für Informatik und Computational Science
Universität Potsdam, Dezember 2023

Online veröffentlicht auf dem
Publikationsserver der Universität Potsdam:
https://doi.org/10.25932/publishup-61794
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-617943

Mein äußerster Dank geht an Professor Gössel, welcher über den gesamten Prozess mitgeforscht, unterstützt und beraten hat. Für die herausragende Betreuung möchte ich mich herzlichst bedanken!

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, den 10. Dezember 2023

Christian Schulz-Hanke

Inhaltsverzeichnis

1 Einführung **1**
1.1 Galoisfeld $GF(2^m)$ 2

2 BCH Code **6**
2.1 Fehlermodell 7
2.2 Minimaler Hammingabstand 8
2.3 BCH Korrektur 11
2.3.1 Ermittlung der Anzahl der Fehlerstellen 11
2.3.1.1 Determinanten zur Bestimmung der Fehleranzahl 12
2.3.1.2 Allgemeine Bestimmung der Fehleranzahl 15
2.3.1.3 Determinanten bei höheren Fehlern 18
2.3.1.4 Fehlerstellen bei 1-Bit Korrektur und weiterer Erkennung 19
2.3.1.5 Fehlerstellen bei 2-Bit Korrektur und weiterer Erkennung 23
2.3.2 Bestimmen der Fehlerstellen 25
2.3.2.1 Bestimmung der Fehlerstellen für den 1-Bit Fall 26
2.3.2.2 Bestimmung der Fehlerstellen für den 2-Bit Fall 27
2.3.2.3 Bestimmung der Fehlerstellen für den 3-Bit Fall 28
2.3.2.4 Bestimmung der Fehlerstellen für den 4-Bit Fall 32
2.3.3 Schaltung 4-Bit Korrektur 43
2.3.4 Korrektur 50
2.3.5 Zum Dekodierungsverfahren nach Berlekamp/Massey 50
2.3.5.1 Funktionsweise Berlekamp-Massey Algorithmus 53
2.3.5.2 Beispiel Berlekamp-Massey Algorithmus 54

3 Kombinierte Korrektur und Erkennung höherer Fehler **57**
3.1 1-Bit Fehlerkorrektur mit zusätzlicher Erkennung 57
3.1.1 1-Bit Fehlererkennung 58
3.1.2 1-Bit Fehlerkorrektur 59
3.1.3 Schematische Hardwareimplementierung 60
3.1.4 Beweis der Prüfgleichungen bei der 1-Bit Fehlerkorrektur 60
3.2 2-Bit Fehlerkorrektur mit zusätzlicher Erkennung 64
3.2.1 Anwendung des 1-Bit Ansatzes 64
3.2.2 Generelle 2-Bit Fehlererkennung und -korrektur 71
3.2.2.1 Schaltung der 2-Bit Korrektur 73
3.2.2.2 VHDL Implementierung 76
3.2.2.3 Softwareimplementierung und Vergleich zur Determinantenberechnung 98

3.2.2.4 Erweiterung auf andere Fehler 101
3.2.3 Alternatives Vorgehen der 2-Bit Korrektur 102
3.3 3-Bit Fehlerkorrektur mit zusätzlicher Erkennung 113
3.4 4-Bit Fehlerkorrektur mit zusätzlicher Erkennung 114
3.5 Allgemeine Fehlerkorrektur mit zusätzlicher Erkennung 114
3.6 Vergleich zum bekannten Verfahren 116
3.6.1 VHDL Vergleichsimplementierung Determinante gegen spekulative Berechnung 117

4 Zusammenfassung **119**

5 Anhang **120**
5.1 Anwendungsbereiche des vorgestellten Verfahrens 120
5.1.1 Annahmen 120
5.1.2 Wahrscheinlichkeitsverteilungen 120
5.1.3 Betrachtung des vorgestellten Korrekturansatzes 125
5.2 Skripte zur Erstellung von Graphen 127
5.3 Python Skript für BCH Codes 132
5.4 Vergleichsimplementierungen 141

Literaturverzeichnis **191**

1 Einführung

Um Fehler beim Auslesen von Speichern zu verhindern, kommen fehlerkorrigierende Codes zum Einsatz. 1-Bit bis 4-Bit Fehler korrigierende BCH Codes können dabei zur parallelen Fehlerkorrektur verwendet werden. Für die Korrektur von bis zu 3-Bit Fehlern sind effiziente Hardwareimplementierungen möglich. Die Implementierung einer parallelen 4-Bit-Korrektur ist hingegen ein höherer Aufwand. Für Anwendungen mit hohen Sicherheitsanforderungen kann es notwendig sein, Fehler mit bis zu einer bestimmten Anzahl von fehlerhaften Bits sicher zu korrigieren und zudem Fehler mit höherer Anzahl fehlerhafter Bits mit hoher Wahrscheinlichkeit zu erkennen.

Gerade bei Speichern, auf welche ein schneller Zugriff notwendig ist, ist es besonders wichtig, dass eine Korrektur in einem oder wenigen Taktzyklen abgeschlossen ist.

In der vorliegenden Arbeit wird ein neues Verfahren der Kombination von Fehlererkennung und Fehlerkorrektur für BCH-Codes vorgestellt und untersucht. Unter der Annahme, dass eine geringe Anzahl an Fehlern aufgetreten ist, wird eine spekulative Fehlerkorrektur durchgeführt. Im Nachhinein werden die spekulativen Fehlerpositionen auf Korrektheit überprüft, indem für diese Fehlerpositionen die höheren Syndromkomponenten bestimmt und mit den Syndromkomponenten des übertragenen Wortes verglichen werden. Dabei werden durch einfache Prüfgleichungen zusätzlich höhere erkennbare Fehler ausgeschlossen. Dadurch ist es möglich eine Fehlererkennung sehr schnell und mit geringem Hardwareaufwand durchzuführen.

In einem bekannten Verfahren zur BCH Korrektur nach Peterson[Pet60] wird zunächst die Anzahl der Fehlerstellen bestimmt, deren Positionen in einem zweiten Schritt abhängig von der Fehleranzahl bestimmt werden. Zur Feststellung der Fehleranzahl wird für alle durch den BCH-Code korrigierbaren Fehleranzahlen x, beginnend mit der höchsten dieser Fehleranzahlen, eine Determinante von $x!$ Termen bestimmt, bis die erste Determinante in der absteigenden Folge der Determinanten den Wert 0 annimmt. Diese Determinante gleich 0 identifiziert die Fehleranzahl. Die konkreten Fehlerpositionen werden in einem Folgeschritt bestimmt.

Da unter den üblichen Annahmen für die Fehlerwahrscheinlichkeiten Fehler mit geringer Fehleranzahl erheblich häufiger auftreten als Fehler mit vielen fehlerhaften Bits, ist dies zur Bestimmung der zu korrigierenden Fehler ein erheblicher Rechenaufwand. Die hier zuerst überprüften Fehleranzahlen stellen die höchsten korrigierbaren Anzahlen fehlerhafter Bits dar. Sie besitzen damit eine geringe oder sehr geringe Auftrittswahrscheinlichkeit.

Im Unterschied dazu wird in der vorgestellten Arbeit spekulativ eine Korrektur für eine geringe Anzahl von Fehlern parallel vorgenommen und danach in einfacher Weise überprüft, ob die nach der Korrektur aus den korrigierten Bitpositionen bestimmten höheren Syndromkomponenten mit den tatsächlich aufgetretenen Syndromkomponenten übereinstimmen. Sind sie gleich, so ist der Fehler korrekt korrigiert. Besteht ein Unterschied, so erfolgte eine Falschkorrektur und ein Fehler mit einer höheren Fehlerstellenanzahl lag vor.

Der Vorteil des vorgestellten Ansatzes ist somit eine schnelle und einfache Überprüfung auf höhere Fehleranzahlen nach der Ausführung der spekulativen Fehlerkorrektur und ein Beitrag zur effektiven Multbitfehlererkennung bei der Anwendung von BCH-Codes.

Weiterhin wird das bekannte Verfahren zur parallelen 4-Bit Korrektur vereinfacht. Die Fehlerpositionen sind als Nullstellen des entsprechenden Lokatorpolynoms bestimmt, welches für die zu korrigierenden 4-Bit Fehler ein Polynom 4-ten Grades ist. Die Nullstellen des Polynoms 4-ten Grades sind in einem aufwendigen Verfahren durch das Lösen einer Gleichung 3-ten Grades und von vier Gleichungen 2-ten Grades bestimmbar. In dieser Arbeit konnte gezeigt werden, dass sich dabei eine der Gleichungen zweiten Grades einsparen lässt. Dadurch vereinfacht sich die Realisierung in Hardware. Zum Vergleich der Geschwindigkeit und des Hardwareaufwands mit dem bisher bekannten Vorgehen wurden umfangreiche Simulationen in Software und Implementierungen in VHDL durchgeführt.

1.1 Galoisfeld $GF(2^m)$

Bei der BCH Korrektur werden Berechnungen in einem Galoisfeld durchgeführt. Ein Galoisfeld $GF(2^m)$ ist ein endlicher Körper mit 2^m Elementen. Die Elemente dieses Galoisfeldes können beispielsweise in folgenden Arten dargestellt werden:

- **Binärvektordarstellung**
  Die Elemente werden als m-elementiger Binärvektor dargestellt, Rechenoperationen werden modulo eines Modularpolynoms p (Binärvektor mit $m+1$ Elementen) vorgenommen. Hinweis: Binärvektoren werden in dieser Arbeit wie Zahlen behandelt, die niedrigste Position ist somit am weitesten Rechts. Ein Beispiel für die Binärvektordarstellung:

  $$00101$$

- **Polynomdarstellung**
  Alternativ kann ein Element eines Galoisfeldes $GF(2^m)$ als Polynom mit binären Koeffizienten (modulo dem Modularpolynom) dargestellt werden, z.B.

  $$a_m x^{m-1} + a_{m-1} x^{m-2} + \ldots + a_3 x^2 + a_2 x + a_1 \tag{1.2}$$

  In dieser Darstellung erfolgen Berechnungen dem allgemeinen Regeln für Polynome mit binären Koeffizienten. Im Vergleich entsprechen die Koeffizienten den Stellen des Binärvektors. Ein Beispiel für die Polynomdarstellung:

  $$x^5 + x^2 + 1$$

- **Exponentendarstellung**
  Eine weitere Darstellung ist als Reihe von Potenzen $\alpha^i \mod p$. Diese ermöglicht eine einfache Abbildung von Multiplikationen $\alpha^a \times \alpha^b = \alpha^{(a+b) \mod 2^m-1}$ und Divisionen $\alpha^a \div \alpha^b = \alpha^{(a-b) \mod 2^m-1}$, jedoch benötigt die Addition und Subtraktion eine andere Darstellungsform. Ein direktes Ablesen der Koeffizienten der Polynomialdarstellung bzw. der Stellen der Binärvektordarstellung ist nicht möglich. Der Wert 0 ist in Exponentendarstellung nicht darstellbar, es gibt kein i mit $\alpha^i = 0$. Ein Beispiel für die Exponentendarstellung:

  $$\alpha^5$$

Um die Zusammenhänge zwischen den Darstellungsformen zu verdeutlichen, hier ein Beispiel für das Galoisfeld $GF(2^3)$ mit dem Modularpolynom $p(x) = 1+x+x^3$

Exponent	Binär	Polynom
-	000	0
α^0	001	1
α^1	010	x
α^2	100	x^2
α^3	011	$1+x$
α^4	110	$x+x^2$
α^5	111	$1+x+x^2$
α^6	101	$1+x^2$

Man kann hier erkennen, dass die Exponentendarstellung keine Darstellungsoption für den Wert 0 hat. In Implementierungen wird dies umgangen, indem dieser Wert den ersten Wert außerhalb des Galoisfelds belegt und gesondert behandelt wird. Dies verdeutlicht folgende Tabelle:

Binär	Exponent	Exponent in Binärdarstellung
000	-	**111**
001	α^0	000
010	α^1	001
100	α^2	010
011	α^3	011
110	α^4	100
111	α^5	101
101	α^6	110

Der Wert $111 = 7$ ist kein gültiger Wert für einen Exponenten und wird in der Implementierung für den Wert 0 eingesetzt.

Rechenarten Als Operationen im Galoisfeld $GF(2^m)$ werden in dieser Arbeit die Addition, Subtraktion, Multiplikation und Division verwendet. Dabei sind die Addition und Subtraktion als elementweises XOR in der Vektordarstellung beschreibbar. Addition und Subtraktion sind im Galoisfeld $GF(2^m)$ dabei dieselbe Operation. Die Multiplikation und Division können in der Exponentendarstellung als Addition und Subtraktion der (natürlichzahligen) Exponenten modulo 2^{m-1} dargestellt werden. In der Polynomdarstellung entsprechen sie der Polynommultiplikation bzw. -division modulo p. Da die Vektordarstellung besonders für Addition und Subtraktion geeignet ist und die Exponentendarstellung besonders für Multiplikation und Division, können Elemente zwischen den Darstellungsformen umgewandelt werden. Ein Hilfsmittel zur Addition und Subtraktion in der Exponentendarstellung ist der Zechsche Logarithmus, dieser ermöglicht eine Addition durch lediglich eine Transformation durchzuführen, an Stelle einer Hin- und Rücktransformation von Exponentendarstellung zu Vektordarstellung.

Beispiele im $GF(2^4)$:

$$\begin{aligned} 1101 + 0111 &= 1010 \\ \alpha^3 \times \alpha^5 &= \alpha^8 \\ \alpha^3 \times \alpha^{14} = \alpha^{(17 \mod 15)} &= \alpha^2 \\ \alpha^3 \div \alpha^{14} = \alpha^{(-11 \mod 15)} &= \alpha^4 \end{aligned}$$

Zechscher Logarithmus Der Zechsche Logarithmus[Hub90] wird verwendet, um Additionen in der Exponentialdarstellung durchzuführen, ohne in die Binärdarstellung wechseln zu müssen. Der Vorteil zu einer Umrechnung in die Binärdarstellung ist, dass statt zwei Tabellen nur eine Tabelle notwendig ist. Das alternative Verfahren mit der Binärtransformation ist aufwendiger und verwendet eine Tabelle zur Umwandlung von Exponenten- zur Binärdarstellung, führt in dieser die Berechnung durch und verwendet anschließend erneut eine Tabelle zur Umrechnung zurück in die Exponentialdarstellung. Der Zechsche Logarithmus $Z_a(n)$ eines Wertes n zu einer Basis α ist

$$\begin{aligned} &Z_\alpha(n) = log_\alpha(1 + \alpha^n) \\ &\alpha^{Z_\alpha(n)} = 1 + \alpha^n \end{aligned} \tag{1.4}$$

Um eine Addition von α^x und α^y zu ermöglichen, kann folgende Formel verwendet werden:

$$\begin{aligned} \alpha^x + \alpha^y &= \alpha^x \times (1 + \alpha^{y-x}) \\ &= \alpha^x \times \alpha^{Z_\alpha(y-x)} \\ &= \alpha^{x + Z_\alpha(y-x)} \end{aligned} \tag{1.6}$$

Im Galoisfeld $GF(2^m)$ ist die Subtraktion identisch zur Addition, somit ist für die Subtraktion kein -1 Element benötigt, welches bei reellen Zahlen zum Einsatz kommt.

Verwendete Beziehungen im Galoisfeld $GF(2^m)$ In dieser Arbeit werden folgende Beziehungen des Galoisfelds $GF(2^m)$ in Berechnungen verwendet und werden daher gesondert hervorgehoben. Seien a, b und c Elemente eines Galoisfeldes $GF(2^m)$, $m, i, j \in \mathcal{N}$

- Addition und Subtraktion sind im Galoisfeld $GF(2^m)$ dieselbe Operation.

$$a - b = a + b \tag{1.8}$$

- Die Verdopplung jedes beliebigen Elements ist 0.

$$\begin{aligned} 2 \times a &= a + a = a - a \\ &= \vec{0} \end{aligned} \tag{1.10}$$

- Das Quadrieren einer Summe entspricht der Summe der quadrierten Summanden. Bei Potenzen, bei welchen die Basis eine Summe ist, können gerade Faktoren aus dem Exponenten

gekürzt und als Exponent jedes einzelnen Summanden hinzugefügt werden.

$$\begin{aligned}
(a+b)^2 &= a^2 + 2ab + b^2 \\
&= a^2 + b^2 \\
((a+\ldots+b)+c)^2 &= (a+\ldots+b)^2 + 2\times(a+\ldots+b)\times c + c^2 \\
&= a^2 + \ldots + b^2 + c^2 \\
(\sum(a_i))^2 &= \sum(a_i^2) \\
(\sum(a_i))^{2j} &= (\sum(a_i^2))^j
\end{aligned} \tag{1.12}$$

2 BCH Code

In diesem Kapitel wird ein begrenzter Überblick zum BCH Code gegeben. Eine umfassendere Erklärung ist beispielsweise zu finden in [Pet60][LC04][TH13].

Eigener Anteil Dieses Kapitel enthält zunächst in der Literatur bekannte Beschreibungen über BCH Codes. Die Neuheit in diesem Kapitel ist, dass ein verbesserter Ansatz zur Bestimmung der Positionen eines 4-Bit Fehlers vorgestellt wird, welcher es ermöglicht, die Lösung eines Gleichungssystems zweiten Grades einzusparen und damit die Hardware zu reduzieren. Außerdem wird für das Verfahren nach Tzschach ein Sonderfall aufgezeigt, welcher zu berücksichtigen ist.

Ein binärer BCH Code [BRC60] im engeren Sinne kann durch H-Matrix in der folgenden Form definiert werden

$$H = \begin{bmatrix} \alpha^0, & \alpha^1, & \alpha^2, & \alpha^3, & \dots & \alpha^{n-1} \\ \alpha^0, & \alpha^3, & \alpha^6, & \alpha^9, & \dots & \alpha^{(n-1)\times 3} \\ \dots & & & & & \\ \alpha^0, & \alpha^{1\times(2\times m-1)}, & \alpha^{2\times(2\times m-1)}, & \alpha^{3\times(2\times m-1)}, & \dots & \alpha^{(n-1)\times(2\times m-1)} \end{bmatrix}$$

Dabei sind $\alpha^0, \dots, \alpha^{2^M-2}$ Elemente im Galoisfeld $GF(2^M)$, n die Länge des Codes und m die Anzahl der korrigierbaren Bits. Die Exponenten von α sind modulo 2^M-1 zu interpretieren und für n gilt $n \leq 2^M-1$.

Durch einen solchen BCH Code können alle bis zu m-Bit Fehler korrigiert werden, alternativ können alle bis zu $(2\times m)$-Bit Fehler erkannt werden.

Das Fehlersyndrom wird wie folgt bestimmt:

$$w \times H = S = \begin{bmatrix} s_1 \\ s_3 \\ \dots \\ s_{2\times m-1} \end{bmatrix}$$

Das Syndrom S besteht dabei aus den m Syndromkomponenten $s_1, s_3, \dots, s_{2\times m-1} \in GF(2^M)$, welche dem Produkt einzelner Zeilen der Matrix H mit dem übertragenen Wort w entsprechen. In einem binären BCH Code gilt $s_{2\times i} = s_i^2$. Sind zwei Syndromkomponenten auseinander errechenbar, bieten sie keine zusätzliche Korrigierbarkeit oder Erkennbarkeit von Stellen. Daher sind die Zeilen der H-Matrix lediglich solche, welche ungeraden Potenzen entsprechen und die Syndromkomponenten sind lediglich solche mit ungeradem Index. Wird ein Codewort mit der H-Matrix multipliziert, ist das Syndrom der Nullvektor.

Ergibt diese Gleichung für ein übertragenes Word den Nullvektor, so wird davon ausgegangen,

dass kein Fehler aufgetreten ist. Es kann jedoch nicht ausgeschlossen werden, dass kein durch den Code unerkennbarer Fehler aufgetreten ist. Ein Fehler, welcher ein Codewort in ein Codewort überführt, ist generell nicht erkennbar. Fehlervektoren solcher Fehler sind zudem selbst Codeworte, da die Differenz zweier Codeworte auch ein Codewort ist. Die Anzahl dieser generell unerkennbaren Fehlervektoren ist zudem im BCH Code gleich der Anzahl der Codewörter minus 1 (minus 1, da der Nullvektor kein Fehlervektor ist).

Falls das Syndrom ungleich dem Nullvektor ist, ist das empfangene Wort kein Codewort. Es ist also ein Fehler aufgetreten und erkannt worden. Für jedes mögliche Codewort ist ein Fehlervektor berechenbar, durch welchen dieses fehlerhafte Wort erzeugt werden könnte. Ohne Annahmen ist es daher nicht eindeutig, welches Codewort einem übertragenen Wort zugrunde liegt. Ein solches übertragenes Wort kann potentiell aus einem beliebigen Codewort mit einem entsprechenden Fehlervektor generiert werden. Daraus ergibt sich, dass die vollständige fehlerfreie Korrektur eines Codes mit mehr als einem Codewort nicht generell möglich ist. Für ein fehlerhaftes Codewort kommt ohne Annahmen jedes Codewort als Ursprungswort in Betracht.

Üblicher Weise wird angenommen, dass Fehler mit geringer Anzahl an Bits wahrscheinlicher sind als Fehler mit hoher Anzahl an Bits. In dieser Arbeit wird ein Verfahren vorgestellt, welches die Korrektur von Fehlern mit geringer Bitzahl schneller erkennen und korrigieren kann, als die bekannten Verfahren.

Im Allgemeinen wird eine Korrektur zu jenem Codewort durchgeführt, welches die geringste Hammingdistanz zu dem übertragenen Wort besitzt. Dieses Konzept wird in den folgenden Sektionen erläutert.

2.1 Fehlermodell

Es existieren verschiedene Modelle, die Fehlerauftritte beschreiben. Für den BCH Code wird ein Fehlermodell verwendet, welches Fehler unabhängig voneinander pro Position in Betracht zieht.

Wir nehmen an, dass Fehler pro Bit zufällig, unabhängig voneinander auftreten können und Bits bei einem Fehler "kippen". Dabei wird eine 1 zu einer 0 und eine 0 zu einer 1. Die Fehlerwahrscheinlichkeit hängt nicht von der Position des Bits ab.

Dadurch ergibt sich, dass man mit der Bernoulli Verteilung die Wahrscheinlichkeiten für verschiedene Fehler unterscheiden kann. Ist die Wahrscheinlichkeit p_e dafür, dass in einem Bit bei einer Übertragung ein Fehler auftritt gegeben, z.B. $p_e = 10^{-6}$, dann kann daraus die Wahrscheinlichkeit für x-Bit Fehler p_{e_x} berechnet werden.

$$p_{e_x} = \binom{n}{x} \times p_e^x \times (1 - p_e)^{n-x}$$

Im Folgenden werden exemplarisch die Wahrscheinlichkeiten für (genau) 1-Bit, 2-Bit und 3-Bit Fehler angegeben. Für ein Wort mit n Bit ist die Wahrscheinlichkeit, dass kein Fehler auftritt entsprechend $(1 - p_e)^n$. Bei einer Wortlänge von 256 Bit ist bei $p_e = 10^{-6}$ die Wahrscheinlichkeit, dass kein Fehler auftritt bei ca. $99,97\%$. Dabei liegt die Wahrscheinlichkeit für einen 1-Bit Fehler p_{e_1} bei

$$p_{e_1} = n \times p_e \times (1 - p_e)^{n-1}$$

Bei einer Wortlänge von 256 Bit bei $p_e = 10^{-6}$ ca. $0,026\%$. Ein 2-Bit Fehler tritt mit einer Wahr-

scheinlichkeit p_{e_2} von

$$p_{e_2} = \frac{n \times (n-1)}{2} \times p_e^2 \times (1-p_e)^{n-2}$$

auf. Bei 256 Bit bei $p_e = 10^{-6}$ ist die Wahrscheinlichkeit ca. $0,00000326\%$. Ein 3-Bit Fehler tritt mit einer Wahrscheinlichkeit p_{e_3} von

$$p_{e_3} = \frac{n \times (n-1) \times (n-2)}{6} \times p_e^3 \times (1-p_e)^{n-3}$$

auf. Bei 256 Bit bei $p_e = 10^{-6}$ ist die Wahrscheinlichkeit ca. $0,00000000000276\%$.

Da die Auftrittswahrscheinlichkeit von Fehlern mit geringer Bitzahl (in üblichen Fällen) größer ist als die von Fehlern mit höherer Bitzahl, wird bei der BCH Korrektur davon ausgegangen (wie allgemein bei Fehlercodes), dass der Fehler mit der geringsten Bitzahl aufgetreten ist, da dieser am wahrscheinlichsten ist.

In bestimmten Fällen können diese Annahmen jedoch nicht zutreffen. So kann es sein, dass bei bestimmten Fehlerwahrscheinlichkeiten und Codelängen eine fehlerfreie Übertragung unwahrscheinlicher ist als mindestens ein 1-Bit Fehler. Eine Ausführliche Betrachtung dazu ist im Kapitel Anwendbarkeit von Codierungsverfahren (Kapitel 5.1) zu finden.

2.2 Minimaler Hammingabstand

In dieser Sektion wird der minimale Hammingabstand und dessen Bedeutung für den BCH Code erklärt. Dabei wird gezeigt, wie sich die Fehlerkorrektur und Fehlererkennung aus dem minimalen Hammingabstand ergeben.

Die Anzahl der fehlerhaften Stellen (bzw. Einsen im Fehlervektor) bezeichnet man als Hammingdistanz zwischen Codewort und erhaltenem Wort. Man nimmt üblicher Weise an, dass das fehlerhafte Wort den kleinsten Hammingabstand zu dem ursprünglichen Codewort unter allen möglichen Codeworten hat.

Somit wird beim Auftreten eines fehlerhaften Wortes in das Codewort korrigiert, welches den geringsten Hammingabstand zum fehlerhaften Wort hat. Wie hoch die Wahrscheinlichkeit ist, dass ein falsch korrigierter Fehler auftritt, ist somit abhängig von der Dichte des Codes, dem Anteil der Codeworte an allen Wörtern zu 2^n (binärer Code), wobei n die Länge des Codes ist. Ebenso lässt sich aussagen, je mehr Prüfbits ein Code beinhaltet, desto geringer ist die Wahrscheinlichkeit, dass ein nicht korrigierbarer Fehler auftritt.

Kann ein Code m-Bit Fehler korrigieren, so bedeutet dies, dass für jedes Codewort C und jedem bis zu m-Bit Fehler F das fehlerhafte Wort $C+F$ einen geringeren Hammingabstand zu C hat, als zu jedem anderen Codewort.

Allgemein ist es so möglich, bis zu m-Bit Fehler zu korrigieren (1-Bit Fehler, 2-Bit Fehler, ... und m-Bit Fehler werden immer korrekt korrigiert) oder bis zu $(2 \times m)$-Bit Fehler zu erkennen (es wird nur erkannt, dass ein Fehler festgestellt wurde, der genaue Fehler wird nicht festgestellt). Bei kombinierter Erkennung mit Korrektur von k Bit ($k < m$), können zusätzlich zur Korrektur von k-Bit Fehler bis $(m+(m-k))$-Bit erkannt werden.

Erklärung Dies lässt sich einfach durch den minimalen Hammingabstand zwischen Codewörtern erklären. Der minimale Hammingabstand wird bei der Konstruktion eines BCH Codes festgelegt.

$$\#Erkennungsbits = Hammingabstand_{min} - 1$$

Ist der minimale Hammingabstand z.B. 7, so bedeutet dies, dass die Anzahl der Bits 7 ist, die mindestens verändert werden müssen, um von einem Codewort zu einem anderen zu gelangen. Somit ergeben Fehler bis 6-Bit kein Codewort und können erkannt werden. Falls ein Fehler ein Codewort in ein anderes Codewort verändert, kann dieser Fehler nicht festgestellt werden. Erkannt werden können alle Fehler, welche ein Codewort in ein nicht-Codewort ändern. Daher können alle Fehler, deren Bit Anzahl kleiner als der Hammingabstand sind (also bis zu 6-Bit Fehler) immer erkennbar.

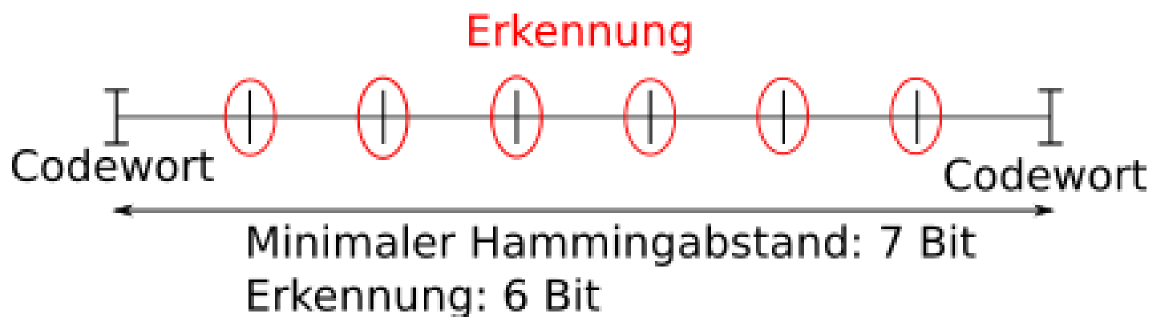

Abbildung 2.1: Fehlererkennung schematisch anhand der Hammingdistanz

Abbildung 2.1 zeigt einen minimalen Hammingabstand von 7 Bit, durch welchen Fehler mit bis zu 6 Bit immer erkannt werden. Dies ändert sich, wenn der Abstand zur Korrektur verwendet wird.

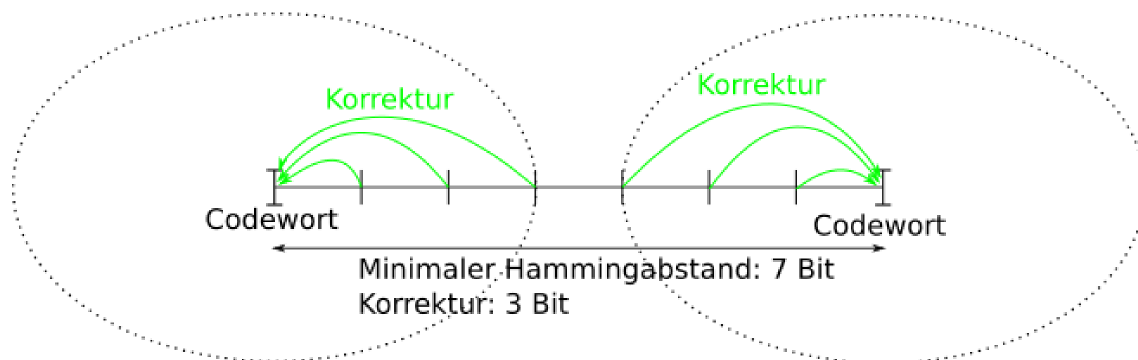

Abbildung 2.2: Fehlerkorrektur schematisch anhand der Hammingdistanz

Wird der Code zur Korrektur statt zur Erkennung verwendet, werden Fehler immer zum nächsten Codewort korrigiert. Also jenes Codewort, von welchem aus der potentielle Fehlervektor die geringste Bitzahl besitzt. Dies basiert auf der Annahme, dass Fehler mit geringer Bitanzahl wahrscheinlicher sind als Fehler mit hoher Bitanzahl. Wie in Abbildung 2.2 zu erkennen ist, können bei

einem Hammingabstand von 7 Bit alle bis zu 3-Bit Fehler immer korrigiert werden. Nehmen wir jedoch beispielsweise einen 6 Bit Fehler an, ist es nicht auszuschließen, dass ein Codewort von diesem fehlerhaften Wort einen Abstand von einem Bit besitzt (aufgrund des minimalen Hammingabstandes von 7 Bit). Man würde einen solchen Fehler falsch korrigieren, falls ein näheres anderes Codewort existiert. Bei einem minimalen Hammingabstand von 7 kann eine richtige Korrektur nur bei Fehlern bis $\frac{7-1}{2} = 3$ Bit garantiert werden.

Ist der minimale Hammingabstand gerade, so gibt es Fehler, welche nicht korrigiert werden, da diese denselben Abstand zwischen zwei Codeworten haben können. Im Falle des Hammingabstandes von 8 Bit existieren nicht eindeutig korrigierbare 4-Bit Fehler, welche jeweils einen 4-Bit Abstand von zwei Codeworten besitzen kann.

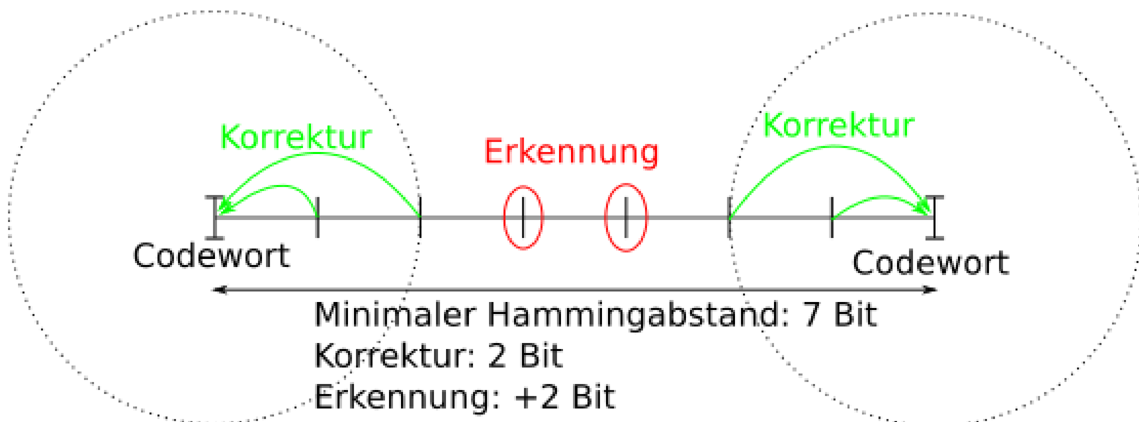

Abbildung 2.3: Fehlererkennung und Korrektur schematisch anhand der Hammingdistanz

Anstatt zu entscheiden, ob Fehler nur korrigiert oder nur erkannt werden sollen, können stattdessen beide Optionen kombiniert werden. Abbildung 2.3 zeigt eine Korrektur mit zusätzlicher Fehlererkennung. Im Vergleich zu der vorherigen Abbildung 2.2 wird hier eine Korrektur von weniger Bits gewählt als potentiell möglich. Im konkreten Beispiel mit einem minimalen Hammingabstand von 7 Bit wählen wir hier eine Korrektur von Fehlern bis 2 Bit, statt wie zuvor 3 Bit. Die 3-Bit Fehler werden in diesem Fall erkannt, aber nicht korrigiert. Da dies auf 3-Bit Fehler von allen Codewörtern zutrifft und ein 4-Bit Fehler von einem Codewort mindestens 3 Bit von jedem anderen Codewort entfernt ist (minimaler Hammingabstand von 7 Bit), können nun sowohl 3-Bit als auch 4-Bit Fehler erkannt werden. Rechnerisch ergibt sich, dass jedes Bit, welches weniger korrigiert wird, zwei Bits zusätzliche Erkennung erlaubt.

Konstruieren wir einen Code indem wir eine Bitanzahl Erkennung und Korrektur vorgeben, so kostet jedes Bit Erkennung 1 Bit minimalen Hammingabstand und jedes Bit Korrektur 2 Bit minimalen Hammingabstand. Somit ergibt sich die Formel:

$$Hammingabstand_{min} = 1 + 2 \times \#Korrekturbits + \#Erkennungsbits$$

In unserem Beispiel 2.3 ist der minimale Hammingabstand damit $1 + 2 \times 2 + 2 = 7$ Bit.

Diese Formel ist für Fehlerkorrektur und -erkennung allgemeingültig. Man kann also das Beispiel aus Abbildung 2.2 als 3-Bit Korrektur mit 0-Bit zusätzlicher Erkennung beschreiben und das Beispiel aus Abbildung 2.1 als 0-Bit Korrektur mit zusätzlicher 6-Bit Fehlererkennung.

Während die minimale Hammingdistanz per Konstruktion des Codes festgelegt wird, ist die Aufteilung in Erkennung und Korrektur flexibel. So ist es möglich, für verschiedene übertragene Worte erst im Nachhinein (nachdem die Worte bereits übertragen wurden) verschiedene Anzahlen an korrigierbaren und erkennbaren Bits zu wählen.

2.3 BCH Korrektur

In diesem Abschnitt wird der BCH Code anhand des bekannten Verfahrens der BCH Korrektur nach Okano Imai [OI87] erklärt. In diesem wird die Anzahl der aufgetreten Fehler bestimmt und diese im Anschluss korrigiert.

Ein dabei zu lösendes Problem stellt die Unterscheidung verschiedener Fehleranzahlen dar. Der bekannte generelle Ansatz berechnet absteigend Determinanten von quadratischen Matrizen mit Größe entsprechend der Anzahl der zu überprüften Fehleranzahl, um die aufgetretene Fehleranzahl festzustellen.

Anschließend werden für Fehler mit bis zu 4 Bits die Fehlerstellen durch das Lösen des Lokatorpolynoms bestimmt. Das Lokatorpolynom entspricht dabei einer Gleichung mit Grad in Höhe der Anzahl der Fehlerstellen. Dessen Nullstellen entsprechen dabei den Fehlerstellen und werden durch Lösung dieser Gleichung bestimmt. Da bei höheren als 4-Bit Fehlern Gleichungen mit Grad 5 oder höher auftreten, welche nicht mehr allgemein algebraisch lösbar sind, arbeitet das Verfahren nur mit bis zu 4-Bit Fehlern. Für Fehler höherer Bitanzahl kann der Berlekamp-Massey Algorithmus verwendet werden, welcher iterativ über die Länge des übertragenen Wortes sequentiell die Positionen auf Fehler prüft. Die üblichen Schritte zur Korrektur eines BCH Codes sind wie folgt:

1. Prüfe, ob alle Syndromkomponenten gleich dem Nullvektor sind.
   Ist dem so, kann kein Fehler erkannt werden.
   Andernfalls wird ein Fehler erkannt.

2. Bestimme die Anzahl der Fehlerbits.

3. Stelle die Positionen der Fehler durch Lösen des Lokatorpolynoms fest.

4. Korrigiere das erhaltene Wort, indem die Bits an den Fehlerstellen invertiert werden.

Diese Schritte werden im Folgenden durch eigene Abschnitte dargestellt.

Syndromkomponenten Die genannten Syndromkomponenten $s_1, s_3, s_5, ...$ werden durch dabei eine Multiplikation des zu korrigierenden Wortes w mit der H-Matrix H ermittelt $w \times H = S = [s_1, s_3, s_5, ...]$. Ist das Syndrom insgesamt der Nullvektor, so ist kein erkennbarer Fehler aufgetreten. Jedes fehlerlos übertragene Codewort ergibt bei der Multiplikation mit der H-Matrix den Nullvektor. Bei jeglichem durch den Code erkennbaren Fehler ist das Syndrom ungleich dem Nullvektor.

2.3.1 Ermittlung der Anzahl der Fehlerstellen

In diesem Unterabschnitt wird ein bekanntes Verfahren zur Erkennung der Anzahl der Fehlerstellen vorgestellt, bei welchem Determinanten in absteigender Folge berechnet werden. Nach einer

kurzen Erklärung der Determinanten wird dargestellt, wie diese zur Identifikation der Fehleranzahl genutzt werden. Zusätzlich wird exemplarisch gezeigt, wie sich die Determinanten bei höheren Fehleranzahlen verhalten. Spezifischer wird anschließend auf die Fälle von 1-Bit und 2-Bit Korrektur mit zusätzlicher Fehlererkennung erarbeitet und gezeigt, wie dieses unter Verwendung der Determinanten verwendet werden kann.

Ein großer Nachteil des beschriebenen Verfahrens ist, dass die Berechnung der Determinanten aufwändig ist. Da sich die Anzahl der Terme in exponentieller Größenordnung zu der Anzahl der zu korrigierenden/erkennenden Fehler befindet, ist das Verfahren für hohe Fehleranzahlen nicht durchführbar.

Ein weiterer Nachteil des bekannten Verfahrens ist, dass es weniger Fehler erkennen kann, als die Kapazitäten des BCH Codes hergeben. Ein BCH Code mit beispielsweise 3 Syndromkomponenten (s_1, s_3, s_5) kann bis zu 6-Bit Fehler erkennen, jedoch kann nur die Determinante für bis zur Determinante $det(M(4))$ aufgestellt werden, wodurch keine 5-Bit und 6-Bit Fehler als solche identifiziert werden können.

2.3.1.1 Determinanten zur Bestimmung der Fehleranzahl

Zur Ermittlung der Anzahl der Fehler nach Peterson [Pet60] können Determinanten der folgenden Form verwendet werden:

$$det(M(p)) = det(\begin{bmatrix} m_{1,1}, & ..., & m_{1,p} \\ ..., & ..., & ... \\ m_{p,1}, & ..., & m_{p,p} \end{bmatrix})$$

für $p > 1$, wobei:

$$\begin{aligned} m_{i,j} &= 0, \text{ falls } (j-1) > 2*(i-1) \\ m_{i,j} &= 1, \text{ falls } (j-1) = 2*(i-1) \\ m_{i,j} &= s_{(i-1)*2-(j-1)}, \text{ andernfalls} \end{aligned} \tag{2.2}$$

hier sind s_j mit ungeraden j Syndromkomponenten eines binären BCH Codes. Für gerade j gilt: $s_{2j} = s_j^2$, diese können somit aus den Syndromkomponenten berechnet werden. Zur Veranschaulichung dienen die folgenden Beispiele:

$$det(M(3)) = det(\begin{bmatrix} m_{1,1}, & m_{1,2}, & m_{1,3} \\ m_{2,1}, & m_{2,2}, & m_{2,3} \\ m_{3,1}, & m_{3,2}, & m_{3,3} \end{bmatrix}) = det(\begin{bmatrix} 1, & 0, & 0 \\ s_2, & s_1, & 1 \\ s_4, & s_3, & s_2 \end{bmatrix})$$

$$det(M(3)) = (1 \times s_1 \times s_2) + (1 \times 1 \times s_3) = s_1^3 + s_3$$

$$det(M(4)) = det(\begin{bmatrix} 1, & 0, & 0, & 0 \\ s_2, & s_1, & 1, & 0 \\ s_4, & s_3, & s_2, & s_1 \\ s_6, & s_5, & s_4, & s_3 \end{bmatrix})$$

$$det(M(4)) = s_1^3 s_3 + s_1 s_5 + s_1^6 + s_3^2$$

$$det(M(5)) = det(\begin{bmatrix} 1, & 0, & 0, & 0, & 0 \\ s_2, & s_1, & 1, & 0, & 0 \\ s_4, & s_3, & s_2, & s_1, & 1 \\ s_6, & s_5, & s_4, & s_3, & s_2 \\ s_8, & s_7, & s_6, & s_5, & s_4 \end{bmatrix})$$

$$det(M(5)) = s_1^{10} + s_1^7 s_3 + s_1^5 s_5 + s_1^3 s_7 + s_1^2 s_3 s_5 + s_1 s_3^3 + s_3 s_7 + s_5^2$$

$$det(M(7)) = det(\begin{bmatrix} 1, & 0, & 0, & 0, & 0, & 0, & 0 \\ s_2, & s_1, & 1, & 0, & 0, & 0, & 0 \\ s_4, & s_3, & s_2, & s_1, & 1, & 0, & 0 \\ s_6, & s_5, & s_4, & s_3, & s_2, & s_1, & 1 \\ s_8, & s_7, & s_6, & s_5, & s_4, & s_3, & s_2 \\ s_{10}, & s_9, & s_8, & s_7, & s_6, & s_5, & s_4 \\ s_{12}, & s_{11}, & s_{10}, & s_9, & s_8, & s_7, & s_6 \end{bmatrix})$$

Ist die Determinante $det(M(p))$ einer solchen Matrix $M(p)$ ungleich dem Nullvektor, dann existiert eine eindeutige Lösung für das lineare Gleichungssystem, welchem die Matrix entspricht. Für Details zu diesem Gleichungssystem und eine Erklärung dessen Herleitung verweisen wir auf das Buch von Peterson [Pet60]. Zusammengefasst bedeutet diese eindeutige Lösung, dass es einen eindeutigen Fehler mit p-Bit oder $p-1$-Bit gibt, welcher die Werte der in $M(p)$ verwendeten Syndromkomponenten (bis s_p) ergibt. Ein solches Ergebnis mit $det(M(p)) \neq 0$ schließt geringere Fehler kleiner $p-1$-Bit aus, jedoch bleibt offen, ob für ein größeres p nicht ebenfalls eine eindeutige Lösung existiert (welche mehr Syndromkomponenten einbeziehen würde). Eine Determinante $det(M(p)) = \vec{0}$ schließt aus, dass ein Fehler mit $p-1$ oder p Bits aufgetreten ist.

Es gilt:

- Ist $det(M(p)) = \vec{0}$, dann ist kein $(p-1)$-Bit und kein p-Bit Fehler aufgetreten.
- Andernfalls ($det(M(p)) \neq \vec{0}$) ist ein $(p-1)$-Bit oder p-Bit (oder höherer) Fehler aufgetreten.

Beispiele:

- Gilt $det(2) = s_1 = 0$, dann kann kein 1-Bit und kein 2-Bit Fehler aufgetreten sein.
- Gilt $det(3) = s_1^3 + s_3 = 0$, dann kann kein 2-Bit und kein 3-Bit Fehler aufgetreten sein.
- Gilt $det(4) = det(4) = s_1^3 s3 + s_1 s_5 + s_1^6 + s_3^2 = 0$, dann kann kein 3-Bit und kein 4-Bit Fehler aufgetreten sein.
- Gilt $det(5) = s_1 s_3^3 + s_5^2 + s_1^7 s_3 + s_1^2 s_3 s_5 + s_1^{10} + s_1^3 s_7 + s_1^5 s_5 + s_3 s_7 = 0$, dann kann kein 4-Bit und kein 5-Bit Fehler aufgetreten sein.

Die Determinante kann somit als untere Abschätzung der Fehleranzahl genutzt werden kann, indem Determinanten in absteigender Folge überprüft werden. Um einem p-Bit Fehler eindeutig zu identifizieren, muss gelten:

$$\begin{aligned} det(M(p)) &\neq \vec{0} \\ det(M(p+1)) &\neq \vec{0} \\ det(M(i)) &= \vec{0} \quad \forall i > p+1 \end{aligned} \tag{2.4}$$

Die maximale Fehlerlänge, die derartig überprüft werden kann, ist bei der Konstruktion der H-Matrix durch die Anzahl der Syndromkomponenten bestimmt.

höchste Syndromkomponente	höchste best. Determinante	korrigierbare Fehleranzahl
s_1	$det(M(2))$	1
s_3	$det(M(3))$	2
s_5	$det(M(4))$	3
s_7	$det(M(5))$	4
$\vdots$	$\vdots$	$\vdots$
s_p	$det(M(\frac{p+3}{2}))$	$\frac{p+1}{2}$

Tabelle 2.3.1.1 zeigt den Zusammenhang der höchsten enthaltenen Syndromkomponenten zur höchsten bestimmbaren Determinante und die sich ergebene erkennbare Fehleranzahl. Während durch die Determinante $det(M(\frac{p+3}{2}))$ theoretisch ein Fehler mit $\frac{p+3}{2}$-Bit oder $\frac{p+1}{2}$-Bit identifiziert wird, kann der höhere beider Fehler ohne weiteres nicht von dem niedrigeren unterschieden werden.

Um die Anzahl der Fehler zu bestimmen, können nun die Determinanten $det(M(x)) \stackrel{?}{=} 0$ für alle Werte $x \in 2..\frac{p+3}{2}$ bestimmt werden. Ist ein q-Bit Fehler aufgetreten, so ist $det(M(q+1))$ ungleich Null und alle Determinanten höherer Werte sind gleich Null.

Da eine Determinante immer zwei Fehlerwerte identifiziert, wird im Folgenden betrachtet, wie diese Eigenschaft genutzt werden kann.

Vereinfachte Gleichungen für übliche Fälle Aus den Determinanten ergeben sich für spezifische Fälle einfache Gleichungen, welche für die Erkennung niedriger Fehleranzahlen Anwendung finden.

- **0-Bit Fehler von anderen Fehlern unterscheiden**: Entspricht das Syndrom dem Nullvektor so wurde kein Fehler festgestellt.

$$S = \vec{0} \tag{2.6}$$

  Es ist jedoch nicht ausgeschlossen, dass ein unerkennbarer Fehler aufgetreten sein.

- **Bis 1-Bit Fehler von 2-Bit und 3-Bit Fehlern unterscheiden**: Ein 1-Bit Fehler kann von 2-Bit und 3-Bit Fehlern unterschieden werden, indem die Gleichung

$$s_1^3 \stackrel{?}{=} s_3 \tag{2.8}$$

  geprüft wird. Gilt die Gleichung, kann kein 2-Bit Fehler und kein 3-Bit Fehler aufgetreten sein. Gilt die Gleichung nicht, kann kein (0-Bit oder) 1-Bit Fehler aufgetreten sein. Dieses Verhalten kann wie folgt gezeigt werden.
  1-Bit Fall, die Gleichung gilt.

$$\alpha^{3i} = \alpha^{3i} \tag{2.10}$$

2-Bit Fall, Gleichung gilt nicht, da beide Fehlerstellen α^i, α^j ungleich sein müssen und $\forall i : \alpha^i \neq 0$.

$$\begin{aligned}
(\alpha^i + \alpha^j)^3 &\stackrel{?}{=} \alpha^{3i} + \alpha^{3j} \\
\alpha^{3i} + \alpha^{3j} + \alpha^{2i+j} + \alpha^{i+2j} &\stackrel{?}{=} \alpha^{3i} + \alpha^{3j} \\
\alpha^{2i+j} + \alpha^{i+2j} &\stackrel{?}{=} 0 \\
\alpha^{i+j} \times (\alpha^i + \alpha^j) &\neq 0
\end{aligned} \tag{2.12}$$

3-Bit Fall, Gleichung gilt nicht, da alle Fehlerstellen $\alpha^i, \alpha^j, \alpha^k$ paarweise verschieden sind.

$$\begin{aligned}
(\alpha^i + \alpha^j + \alpha^k)^3 &\stackrel{?}{=} \alpha^{3i} + \alpha^{3j} + \alpha^{3k} \\
\alpha^{3i} + \alpha^{3j} + \alpha^{3k} + \alpha^{2i+j} + \alpha^{i+2j} + \alpha^{2i+k} + & \\
\alpha^{i+2k} + \alpha^{2j+k} + \alpha^{j+2k} &\stackrel{?}{=} \alpha^{3i} + \alpha^{3j} + \alpha^{3k} \\
\alpha^{2i+j} + \alpha^{i+2j} + \alpha^{2i+k} + \alpha^{i+2k} + \alpha^{2j+k} + \alpha^{j+2k} &\stackrel{?}{=} 0 \\
(\alpha^i + \alpha^j)(\alpha^i + \alpha^k)(\alpha^j + \alpha^k) &\neq 0
\end{aligned} \tag{2.14}$$

- **Bis 2-Bit Fehler von 3-Bit und 4-Bit Fehlern unterscheiden**: Die in Sektion 3.2.3 vorgestellte Gleichung kann genutzt werden, um Fehler bis 2-Bit von 3-Bit und 4-Bit Fehlern zu unterscheiden.

  $$(s_1^5 + s_5) \times (s_1) \stackrel{?}{=} (s_1^3 + s_3) \times (s_3) \tag{2.16}$$

  Gilt die Gleichung, kann kein 3-Bit oder 4-Bit Fehler aufgetreten sein. Gilt die Gleichung nicht, kann kein bis zu 2-Bit Fehler aufgetreten sein. Diese Gleichung von in anderer Form ebenfalls in der Arbeit von Okano-Imai verwendet.

- **Gerade und ungerade Fehleranzahlen**: Mit einem Paritätsbit kann einfach zwischen gerader und ungerader Fehleranzahl unterschieden werden. Das Paritätsbit muss dabei Teil der vom BCH Code gesicherten Bits sein.

Durch die Verwendung der Formeln $S \stackrel{?}{=} \vec{0}$ und $(s_1^5 + s_5) \times (s_1) \stackrel{?}{=} (s_1^3 + s_3) \times (s_3)$ zusammen mit einem Paritätsbit ist es möglich, für Fehler bis 4-Bit das erwartete Korrekturverfahren eindeutig bestimmen zu können. Zusätzlich ermöglicht das Paritätsbit eine Erkennung jedes Fehlers mit ungerader Anzahl an Bits und schränkt somit die Menge der unerkennbaren Fehler weiter ein.

2.3.1.2 Allgemeine Bestimmung der Fehleranzahl

In diesem Abschnitt betrachten wir den Fall, dass alle verschiedenen Fehleranzahlen unterschieden werden sollen. Da die Aussage der Determinante $det(M(p))$ den kleinstmöglichen Fehler angibt, ist die im Verfahren von Peterson verwendete Methode, mit absteigenden p zu Prüfen. Das Konzept hier ist, dass bei einem geringen Fehler ohnehin sämtliche p überprüft werden müssen und bei einer hohen Fehlerbitanzahl so die Prüfungen für geringe p entfallen, wobei jedoch eine

hohe Bitfehleranzahl unwahrscheinlicher sind als eine geringe Bitfehlerzahl. Hier kann ein Paritätsbit zusätzlich verwendet werden, um gerade und ungerade Bitfehler einfach und eindeutig zu unterscheiden.

Ohne Parität Zunächst zeigen wir einen möglichen Algorithmus zur Berechnung ohne Paritätsbit bei einem bis n-Bit Fehler korrigierenden Code.

1. Überprüfe Syndrom auf Nullvektor. Falls dies zutrifft, wurde keine Fehler festgestellt. Beende den Algorithmus.
2. Setze $i = n+1$, wobei n der größte korrigierbare Fehler ist (um einen n-Bit Fehler von einem $(n-1)$-Bit Fehler ohne Paritätsbit zu unterscheiden, muss $det(M(n+1))$ berechnet werden)
3. Während $i > 1$:
   a) Prüfe $det(M(i)) \stackrel{?}{\neq} 0$:
      Falls dies zutrifft, nehme einen $(i-1)$-Bit Fehler an und beende den Algorithmus.
      Andernfalls setze $i = i-1$ und setze fort.
4. Falls der Algorithmus nicht beendet wurde, gebe einen nicht korrigierbaren Fehler aus.

Am Beispiel einer bis zu 3-Bit Korrektur (bzw. mindestens 6-Bit Fehlererkennung) zeigen wir nun, wie dies aussehen würde und weshalb die entsprechenden Fehler zu bestimmten Punkten angenommen werden können.

1. Prüfe, ob das Syndrom der Nullvektor ist.
   - Falls dies zutrifft, wurde kein Fehler gefunden; beende die Korrektur
   - Andernfalls setze fort
2. Prüfe, ob die Determinante für den 4-Bit und 3-Bit Fehler ungleich Null ist: $det(M(4)) \stackrel{?}{\neq} 0$
   - Falls dies zutrifft wurde ein mindestens 3-Bit Fehler festgestellt. Entsprechend der Annahme, dass nur Fehler bis 3 Bit in Betracht gezogen werden, wird dieser Fehler als 3-Bit Fehler korrigiert und ein möglicher 4-Bit oder höherer Fehler ignoriert. Der Algorithmus endet hier.
   - Andernfalls wurde kein Fehler ab 3-Bit festgestellt, setze fort
3. Prüfe, ob die Determinante für den 3-Bit und 2-Bit Fehler ungleich Null ist: $det(M(3)) \stackrel{?}{\neq} 0$
   - Falls dies zutrifft, wurde ein mindestens 2-Bit Fehler festgestellt. Da zuvor 3-Bit und höhere Fehler ausgeschlossen wurden, nehme einen 2-Bit Fehler an und korrigiere diesen. Der Algorithmus endet hier.
   - Andernfalls wurde kein 2-Bit Fehler festgestellt, setze fort
4. Prüfe, ob die Determinante für den 2-Bit und 1-Bit Fehler ungleich Null ist: $det(M(2)) = s_1 \stackrel{?}{\neq} 0$

- Falls dies zutrifft, wurde ein mindestens 1-Bit Fehler festgestellt. Da zuvor 2-Bit und höhere Fehler ausgeschlossen wurden, nehme einen 1-Bit Fehler an und korrigiere diesen.
- Andernfalls trat nicht korrigierbarer Fehler auf. Gebe einen Fehler aus.

Bei dieser Feststellung müssten für eine n-Bit Korrektur bis zu n Determinanten berechnet werden (die Determinanten $det(M(n+1))$, ..., $det(M(2))$).

Mit Parität Wird zusätzlich ein Paritätsbit verwendet, können gerade und ungerade Fehler voneinander unterschieden werden. Da bei jeder berechneten Determinante ein p-Bit bzw. $p-1$-Bit Fehler festgestellt werden kann und genau einer der beiden Fehler durch den Paritätswert auszuschließen ist, genügt es zur n-Bit Korrektur von n lediglich jede zweite Determinante zu überprüfen ($det(M(n))$, $det(M(n-2))$, ... , $det(M(2))$, je nachdem, ob n ungerade oder gerade ist).

1. Prüfe das Paritätsbit auf Korrektheit. Ist dies nicht korrekt, so muss die Anzahl der fehlerhaften Bits ungerade sein. Ist die Parität nicht korrekt:
   - Setze i auf die nächste ungerade Zahl $\leq n$ (also $i = n$, falls n ungerade, sonst $i = n-1$). i ist somit Ungerade.
   - Während $i > 1$:
     - Prüfe $det(M(i)) \overset{?}{\neq} 0$:
       Falls dies zutrifft, wurde ein $(i-1)$-Bit oder i-Bit Fehler festgestellt. Entsprechend dem Paritätsbit ist die ungerade Zahl i die Fehleranzahl. Beende den Algorithmus.
       Andernfalls setze $i = i-2$ und setze fort.
   - Prüfe $s_1 \neq \vec{0}$. Falls dies zutrifft und keine größere Fehleranzahl festgestellt wurde, nehme entsprechend der Parität einen 1-Bit Fehler an und beende den Algorithmus.
   - Gebe einen ungeraden, nicht korrigierbaren Fehler mit mehr als n Bit aus, falls kein Fehler bis hier erkannt wurde.

   Ist die Parität korrekt, so ist die Fehleranzahl gerade und es ist auch möglich, dass kein Fehler aufgetreten ist (0-Bit Fehler). Ist die Parität korrekt:
   - Überprüfe Syndrom auf Nullvektor $S = \vec{0}$. Falls dies zutrifft und die Parität korrekt ist, wurde kein Fehler festgestellt. Beende den Algorithmus.
   - Setze i auf die nächste gerade Zahl $\leq n$.
   - Während $i \geq 2$:
     - Prüfe $det(M(i)) \overset{?}{\neq} 0$:
       Falls dies zutrifft, wurde ein $(i-1)$-Bit oder i-Bit Fehler festgestellt. Entsprechend dem Paritätsbit ist die gerade Zahl i die Fehleranzahl. Beende den Algorithmus.
       Andernfalls setze $i = i-2$ und setze fort.
   - Gebe einen geraden, nicht korrigierbaren Fehler aus, falls kein Fehler bis hier erkannt wurde.

2.3.1.3 Determinanten bei höheren Fehlern

Um zu klären, wie sich die Determinanten verhalten, falls höhere Fehler auftreten, wurde exemplarisch ein Test durchgeführt (Code Anhang 5.10 und Ergebnis Anhang 5.11).

Bei diesem exemplarischen Test wird ein BCH Code mit 4 Syndromkomponenten verwendet und für aufsteigende Fehlerzahlen ausgegeben, ob die Determinanten $M(2), M(3), M(4), M(5)$ jeweils gleich oder ungleich 0 sind. Es werden pro Fehleranzahl die Determinanten der ersten 500 vorgefundenen Fehler dargestellt.

Fehler	1000
1	127

Fehler	1100
2	500

Fehler	1110	0110
3	496	4

Fehleranzahl	1111	0111	1011	1101	1110	0110
4	492	4	4	0	0	0
5	491	4	2	3	0	0
6	489	4	2	2	3	0
7	487	4	5	3	1	0
8	487	4	2	3	4	0
9	488	4	2	1	5	0
10	484	4	3	5	4	0
11	486	3	5	4	2	0
12	**484**	**1**	**2**	**9**	**3**	**1***
13	479	5	6	6	4	0
14	487	4	4	3	2	0
15	480	4	5	5	6	0
16	485	3	7	2	3	0
17	483	3	5	6	3	0
18	481	4	6	4	3	0
19	484	5	4	5	2	0
20	486	4	5	3	2	0

Tabelle 2.1: Tabellen mit Auflistung der Determinanten der ersten bis 500 Fehler für bestimmte Fehleranzahlen. Die Ergebnisse der Determinanten werden als Folge von 4 Zahlen dargestellt, eine für jede Determinante aus $M(2), M(3), M(4), M(5)$. Eine 0 besagt, dass die Determinante gleich 0 ist. Eine 1 sagt aus, dass die Determinante ungleich 0 ist. Die hervorgehobene Zeile wird im Detail erklärt.

In Tabelle 2.1 werden die Ergebnisse aus dem Anhang 5.10 zusammengefasst. In der Spalte *Fehler* bzw. *Fehleranzahl* wird die Anzahl der Fehler, für die die Ergebnisse der Determinanten dargestellt werden, aufgezählt. Die kleinen Tabellen stellen die 1-Bit, 2-Bit und 3-Bit Fehler dar, da nur wenige Ergebnisse auftreten. Im 1-Bit Fall existieren 127 1-Bit Fehler, für welche $det(M(2)) \neq 0, det(M(3)) = 0, det(M(3)) = 0, det(M(4)) = 0$ gilt, welches als 1000 dargestellt wird. Im 2-Bit Fall existieren mehr als 500 Fehler (hier werden höchstens 500 Fehler betrachtet), bei allen gilt $det(M(2)) \neq 0, det(M(3)) \neq 0, det(M(3)) = 0, det(M(4)) = 0$, dargestellt als 1100. Im 3-Bit Fall haben 496 der 500 Fehler die Form 1110, die für restlichen 4 die Form 0110. Daraus ist ersichtlich, dass es 3-Bit Fehler gibt, für welche $det(M(2))$ gleich 0 ist. Dies zeigt, aus

welchem Grund Determinanten im bekannten Verfahren in absteigender Reihenfolge berechnet werden, da die höheren Determinanten eindeutig sind. In der großen Tabelle werden nun der 4-Bit Fehler und die höheren Fehler dargestellt. Die hervorgehobene Zeile für Fehleranzahl **12** zeigen beispielsweise die für 500 Fehler berechneten Determinanten

- **484**-mal die Werte $det(M(2)) \neq 0, det(M(3)) \neq 0, det(M(3)) \neq 0, det(M(4)) \neq 0$ (Spalte 1111),
- **1**-mal $det(M(2)) = 0, det(M(3)) \neq 0, det(M(3)) \neq 0, det(M(4)) \neq 0$ (Spalte 0111),
- **2**-mal $det(M(2)) \neq 0, det(M(3)) = 0, det(M(3)) \neq 0, det(M(4)) \neq 0$ (Spalte 1011),
- **9**-mal $det(M(2)) \neq 0, det(M(3)) \neq 0, det(M(3)) = 0, det(M(4)) \neq 0$ (Spalte 1101),
- **3**-mal $det(M(2)) \neq 0, det(M(3)) \neq 0, det(M(3)) \neq 0, det(M(4)) = 0$ (Spalte 1110),
- und **1**-mal $det(M(2)) = 0, det(M(3)) \neq 0, det(M(3)) \neq 0, det(M(4)) = 0$ (Spalte 0110).

Das Ergebnis 0110 ist hier eine Besonderheit da es einen der wenigen Sonderfälle zeigt, bei welchem bei einem höheren Fehler mehr als eine einzelne Determinante gleich Null ist. Werden nicht nur für 500 Fehler Determinanten berechnet, sondern für alle, würde sich dieser Fall auch vereinzelt in anderen Fehlerzahlen vorfinden.

Generell ist aus der Tabelle ersichtlich, dass bei höheren Fehlern ($>$ 3-Bit) der Großteil aller Determinanten ungleich 0 ist. Es ist eine Tendenz erahnbar, dass die Wahrscheinlichkeit von Determinanten gleich 0 mit steigender Fehleranzahl leicht größer zu werden scheint. Determinanten mit einem Wert gleich 0 treten selten auf, und noch seltener sind Determinanten mit mehr als einem Wert gleich 0.

Da es sich jedoch nur um einen Ausschnitt aller Determinanten handelt, ist dieser Tabelle mit Vorsicht zu betrachten. Um beispielsweise zu berechnen, welche Fehleranzahlen mit welcher Wahrscheinlichkeit erkennbar sind, falls nur eine 3-Bit Korrektur durchgeführt und die höchste Determinante zur Erkennung verwendet wird, müsste alle Determinanten vollständig berechnet werden, da die angegebenen ersten 500 Werte ggf. nicht repräsentativ sind.

2.3.1.4 Fehlerstellen bei 1-Bit Korrektur und weiterer Erkennung

In diesem Abschnitt wird unter Verwendung des bekannten Verfahrens zur Erkennung von Fehlern detailliert gezeigt, wie die Determinanten im Falle einer 1-Bit Korrektur zur Identifikation höherer Fehler genutzt werden können.

Das Konzept ist, 1-Bit Fehler zu korrigieren und zu erkennen, ob ein höherer Fehler aufgetreten ist. Dabei ist es irrelevant, welcher andere Fehler aufgetreten ist. Es muss nur unterschieden werden, ob eine Übertragung fehlerlos (Syndrom gleich Nullvektor) war, ob genau ein 1-Bit Fehler aufgetreten ist oder ob ein höherer Fehler aufgetreten ist.

Es gilt hier die Annahme, dass das Syndrom S ungleich dem Nullvektor ist, ansonsten ist kein erkennbarer Fehler aufgetreten. Im Fall eines 1-Bit Fehlers muss somit grundsätzlich $det(M(2)) = s_1 \neq \vec{0}$ und $det(M(1)) = 1 \neq \vec{0}$ sein und für alle $p > 2$ muss gelten $det(M(p)) = \vec{0}$.

Ohne Parität Werden ohne Paritätsbit alle entsprechend Codekonstruktion verfügbaren Determinanten berechnet, so ergibt sich das Bild in Abbildung 2.4.

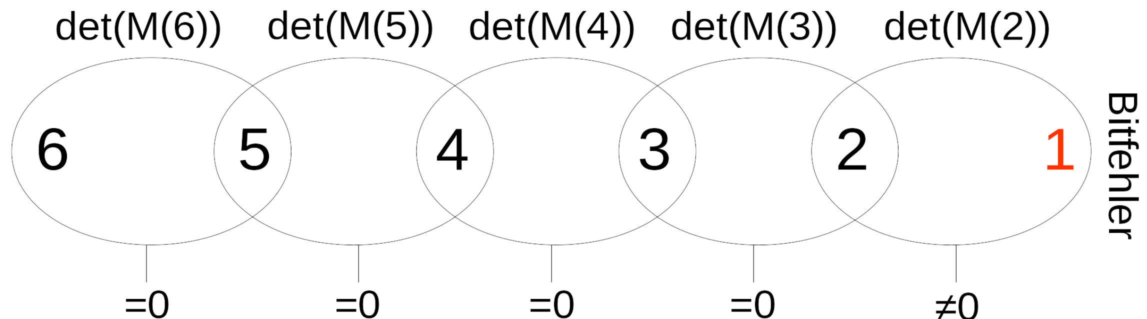

Abbildung 2.4: Identifikation eines 1-Bit Fehlers bei Erkennung bis 6-Bit Fehler. Die beiden im Kreis unter den Determinanten enthaltenen Zahlen entsprechen den durch diese Determinante erkennbare Bit-Fehler. Unter den Kreisen befinden sich die erwarteten Werte dieser Determinante im Falle eines 1-Bit Fehlers.

Ist der Wert mindestens einer der Determinanten nicht wie angegeben, kann kein 1-Bit Fehler aufgetreten sein. Der Fehler kann folgend anhand der Überlappung der höchsten beiden benachbarten Determinanten ungleich 0 identifiziert werden. Soll keine Korrektur von höheren Bitfehlern vorgenommen werden, ist nicht relevant, welche die Fehleranzahl genau aufgetreten ist. Es genügt die Erkenntnis, dass ein höherer Fehler erkannt wurde. Eine Überlappung der Determinanten ist für diesen Fall nicht notwendig, muss somit nur ungefähr die Hälfte der Determinanten berechnet werden. Dies wird in Abbildung 2.5 gezeigt.

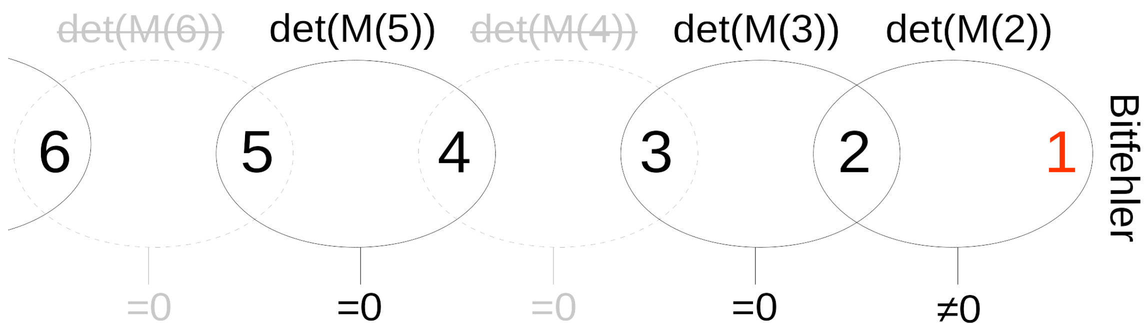

Abbildung 2.5: Identifikation eines 1-Bit Fehlers unter Berechnung nur notwendiger Determinanten bei ungerader maximal zu erkennenden Fehleranzahl. Ausgegraute Matrizen müssen nicht berechnet werden, da ihre Werte durch andere Matrizen abgedeckt werden. $det(M(3))$ muss berechnet werden, da sonst ein 1-Bit Fehler nicht von einem 2-Bit Fehler unterschieden werden kann.

Die Überlappung der Determinanten für den 2-Bit Fehler ist notwendig, da $det(M(2)) \neq \vec{0}$ auf einen 1-Bit oder 2-Bit Fehler hinweist und $det(M(3)) = \vec{0}$ benötigt wird, um den 2-Bit Fehler auszuschließen. Die Determinante $det(M(3)) = s_1^3 + s_3$ mit 0 zu vergleichen entspricht dem bekannten Verfahren, $s_1^3 + s_3 = 0$ bzw. $s_1^3 = s_3$ zu überprüfen, um einen 1-Bit Fehler von einem 2-Bit

Fehler unterscheiden zu können. Ist die höchste zu erkennende Fehleranzahl ungerade, so kann für gerade $p > 2$ auf alle $det(M(p))$ verzichtet werden.

Abbildung 2.6: Identifikation eines 1-Bit Fehlers unter Berechnung nur notwendiger Determinanten bis zum 6-Bit Fall. Ist die maximal zu überprüfende Fehleranzahl gerade, muss es eine zusätzliche Überschneidung zweier Matrizen geben.

Ist die höchste zu erkennende Fehlerzahl p gerade, muss zusätzlich $det(M(p)) = \vec{0}$ überprüft werden. In Abbildung 2.6 werden Fehler bis 6-Bit erkannt. In diesem Fall ist es theoretisch irrelevant, ob $det(M(5))$ oder $det(M(4))$ berechnet wird. Praktisch betrachtet ist $det(M(4))$ mit weniger Aufwand berechenbar und somit zu bevorzugen. Im allgemeinen Fall gerader maximal zu erkennender Fehleranzahl müssen hier Determinanten $det(M(p))$ mit ungeradem $p > 3$ nicht berechnet werden.

Parität Wird zusätzlich ein Paritätsbit verwendet, so kann man gerade von ungeraden Fehleranzahlen unterscheiden. Somit ist es möglich, anhand der $det(M(2)) \neq \vec{0}$ und der Parität den 1-Bit und 2-Bit Fehler zu unterscheiden. Dies ermöglicht es, auf die Berechnung der $det(M(3))$ zu verzichten, falls diese nicht aufgrund einer Ungeraden Höchstfehleranzahl benötigt wird. **Soll lediglich ein 1-Bit Fehler korrigiert werden, ist eine Berechnung der Determinanten nur notwendig, falls bei einem Fehler (Syndrom nicht Nullvektor) eine ungerade Parität vorliegt, andernfalls ist die Fehlerzahl gerade und wird nicht korrigiert.**

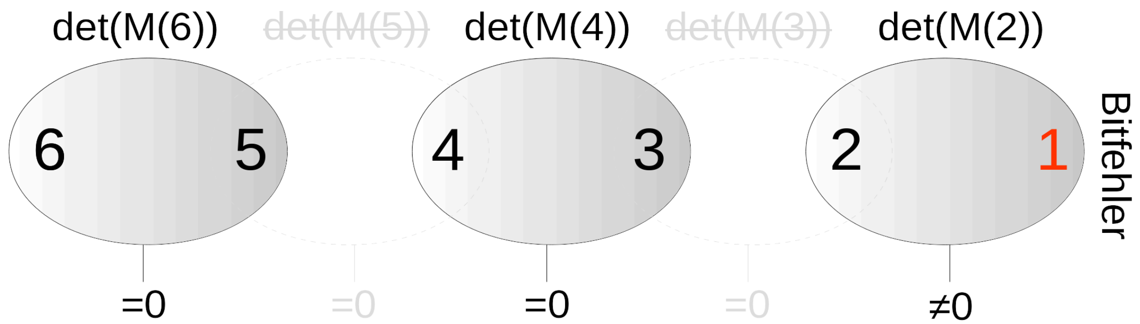

Abbildung 2.7: Identifikation eines 1-Bit Fehlers unter Berechnung nur notwendiger Determinanten bis zum 6-Bit Fall unter Verwendung der Parität. In diesem Fall kann zwischen den beiden Fehler, welche eine Determinante identifiziert mittels Parität unterschieden werden. Als Schema kann dies für gerade maximale Fehleranzahlen verwendet werden.

Bei gerader Parität und Syndrom ungleich Nullvektor kann kein 1-Bit Fehler aufgetreten sein und ein höherer Fehler wurde festgestellt. Eine Berechnung der Determinanten ist in diesem Fall unnötig. Sollen höhere Fehler ebenfalls korrigiert werden, kann die Parität genutzt werden, um die jeweils 2 durch eine Determinante abgedeckten Fehler zu unterscheiden (der Fehler wird durch die größte Determinante ungleich 0 identifiziert). Falls kein erkannter Fehler aufgetreten ist, ist das Syndrom der Nullvektor. Ist ein Fehler aufgetreten und die Parität ist gerade (eine gerade Fehleranzahl), so ist klar, dass ein vom 1-Bit Fehler verschiedener Fehler aufgetreten sein muss.

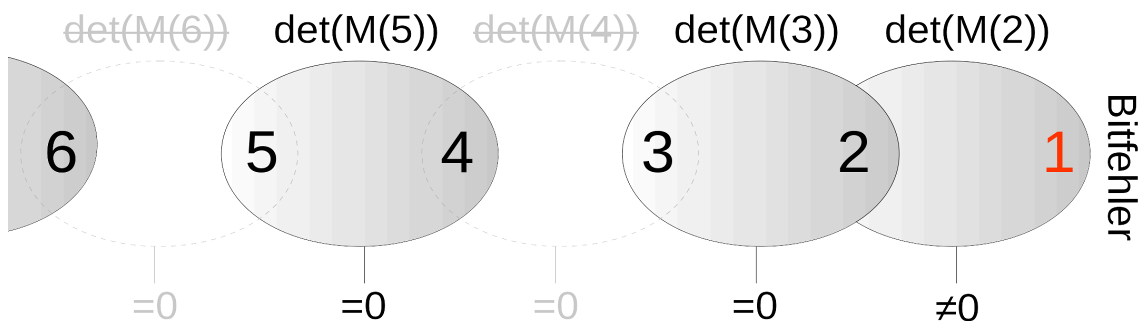

Abbildung 2.8: Identifikation eines 1-Bit Fehlers unter Verwendung der Parität, wobei die höchste zu erkennende Fehleranzahl ungerade ist. Es findet erneut eine Überschneidung statt.

Bei einer ungeraden maximal zu erkennenden Fehleranzahl wird eine Fehleranzahl erneut durch zwei Determinanten abgedeckt. In Abbildung 2.8 beispielsweise die 2-Bit Fehler. Diese doppelt abgedeckte Stelle kann durch Wahl anderer Determinanten an einer anderen Stelle auftreten. Da Determinanten kleinerer Matrizen allgemein einfacher zu berechnen sind, ist es effizienter, die ungeraden Determinanten ab $det(M(3))$ zu berechnen.

2.3.1.5 Fehlerstellen bei 2-Bit Korrektur und weiterer Erkennung

Im Falle einer 2-Bit Korrektur mit zusätzlicher Erkennung höherer Fehleranzahlen muss identifiziert werden, ob ein 1-Bit Fehler, ein 2-Bit Fehler oder ein Fehler mit höherer Bitzahl vorliegt. Unter den höheren Fehlern wird nicht unterschieden, welche Fehleranzahl der Fehler hat, es ist nur erforderlich festzustellen, dass der Fehler mehr als 2-Bit enthalten muss. Generell muss $det(M(2)) = s_1 \neq \vec{0}$ gelten, welches sowohl bei 1-Bit als auch bei 2-Bit Fehlern eintritt.

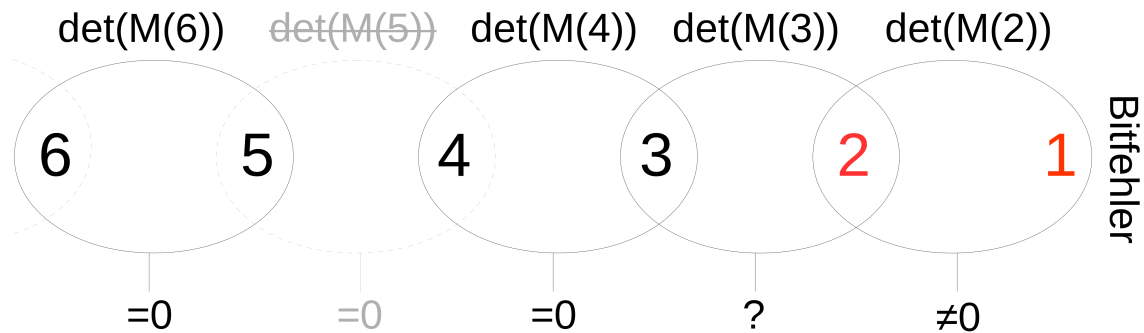

Abbildung 2.9: Vereinfachtes Schema 2-Bit Korrektur mit Erkennung weiterer Bitfehler. der Wert von $det(M(3))$ entscheidet, ob ein 1-Bit Fehler ($det(M(3)) = 0$) oder 2-Bit Fehler ($det(M(3)) \neq 0$) vorliegt.

Ohne Parität Um ohne Parität 1-Bit Fehler von 2-Bit Fehlern zu unterscheiden, ist nun zusätzlich der Wert von $det(M(3))$ zu berechnen. Dies entspricht dem bekannten Verfahren zur Unterscheidung von 1-Bit und 2-Bit Fehlern, bei welchem $s_1^3 = s_3$ geprüft wird. Für diesen gibt es nun folgende Möglichkeiten:

- Bei $det(M(3)) = 0$ ist kein 2-Bit Fehler oder 3-Bit Fehler aufgetreten, daher nehmen wir einen 1-Bit Fehler an und testen auf 4-Bit und höhere Fehler. Wie in Abbildung 2.10 abgebildet, ist in diesem Fall der 3-Bit Fehler ausgeschlossen, Determinanten müssen erst ab dem 4-Bit Fehler ($det(M(5))$) überprüft werden.

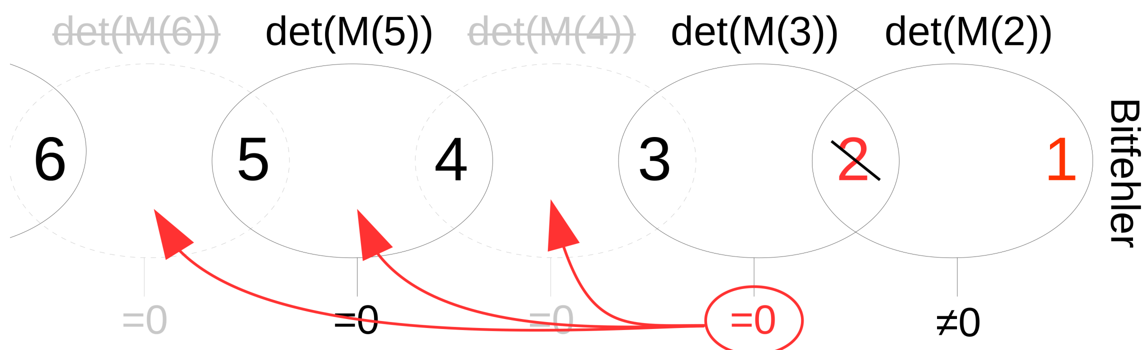

Abbildung 2.10: Identifikation eines 1-Bit Fehlers im Falle der 2-Bit Korrektur. Ist $det(M(3)) = 0$ festgestellt worden, sind nur noch ungerade Matrizen ab $det(M(5))$ zu testen. Ist die maximal erkennbare Fehleranzahl ungerade, ist es somit möglich die Berechnung einer Determinante zu sparen

- Bei $det(M(3)) \neq 0$ ist mindestens ein 2-Bit Fehler aufgetreten. In diesem Fall muss geprüft werden, ob ein 3-Bit oder höherer Fehler aufgetreten ist. Da $det(M(3)) \neq 0$ auch auf einen 3-Bit Fehler hinweisen kann, muss in diesem Fall $det(M(4)) = 0$ geprüft werden. Ist die maximale Fehleranzahl ungerade, muss im Falle eines 2-Bit Fehlers eine zusätzliche Determinante mehr überprüft werden, als im Falle eines 1-Bit Fehlers.

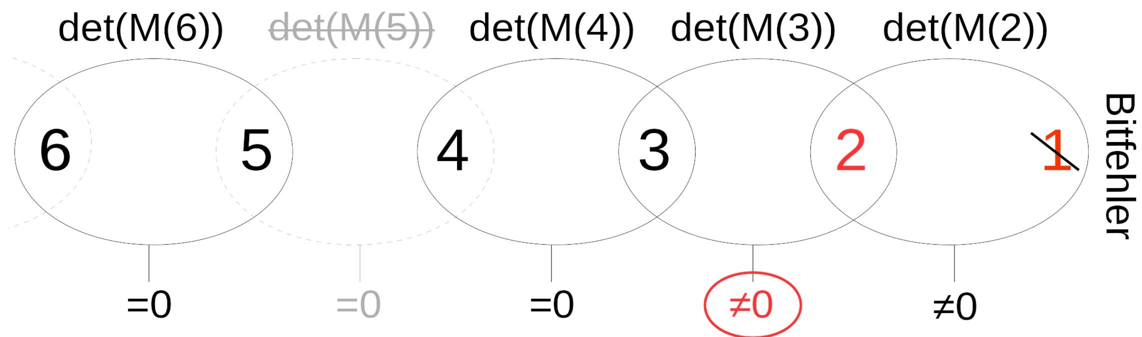

Abbildung 2.11: Identifikation eines 2-Bit Fehlers im Falle der 2-Bit Korrektur. Durch $det(M(3)) \neq 0$ ist der 1-Bit Fehler auszuschließen.

Wie bereits erläutert, ist entsprechend der Überlappung der Abdeckung der Determinanten bei 1-Bit Fehlern eine ungerade maximale Fehlerzahl optimal (keine Überlappungen), im Falle eines 2-Bit Fehlers ist eine gerade maximale Fehlerzahl optimal. Da bei der Konstruktion eines BCH-Codes entschieden werden muss, welche Fehlerzahl maximal betrachtet wird, stellt sich nun die Frage, ob eine gerade oder ungerade Zahl geeigneter ist.

- Ist die maximal erkennbare Fehlerzahl gerade, kann ein vereinfachtes Verfahren verwendet werden (siehe Abbildung 2.9), welches sowohl für den 1-Bit als auch für den 2-Bit Fehler verwendbar ist. Dies erlaubt es, die Berechnung aller Determinanten parallel anzustoßen, ohne das Ergebnis von $det(M(3))$ zu benötigen.

- Ist die maximal erkennbare Fehlerzahl ungerade, kann im Falle eines 1-Bit Fehlers nach Abbildung 2.10 eine Determinante weniger berechnet werden. Da üblicher Weise 1-Bit Fehler häufiger auftreten als 2-Bit Fehler, ist üblicher Weise mit einer Ersparnis zu rechnen. Allerdings ist erst nach Berechnung von $det(M(3))$ klar, ob mit einem 1-Bit Fehler zu rechnen ist. Im Falle eines 2-Bit Fehlers müsste zusätzlich $det(M(4)) = 0$ geprüft werden.

Mit Parität Ist zusätzlich ein Paritätsbit gegeben, so können gerade von ungeraden Fehleranzahlen unterschieden werden. Ist das Syndrom ungleich dem Nullvektor, kann anhand des Paritätsbits zu Beginn entschieden werden, ob ein 1-Bit oder 2-Bit Fehler angenommen wird. Bei einer geraden Fehleranzahl kommt kein 1-Bit Fehler in Betracht, bei einer ungeraden Fehleranzahl kann kein 2-Bit Fehler eingetreten sein.

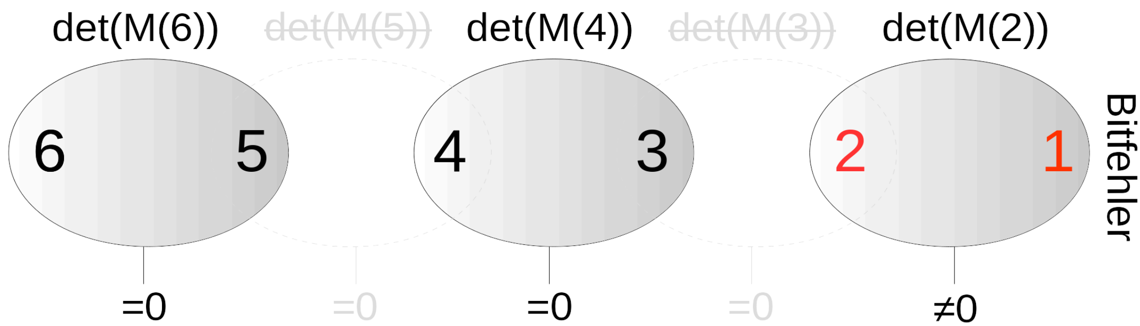

Abbildung 2.12: Identifikation eines 2-Bit Fehlers im Falle der 2-Bit Korrektur mit Parität. Anhand der Parität kann zwischen den Fehleranzahlen, welche durch Determinanten abgedeckt werden, unterschieden werden.

Ist die maximal erkennbare Fehleranzahl gerade, kann das Schema in Abbildung 2.12 verwendet werden. Ist diese hingegen ungerade, so kann wie in Abbildung 2.13 verfahren werden.

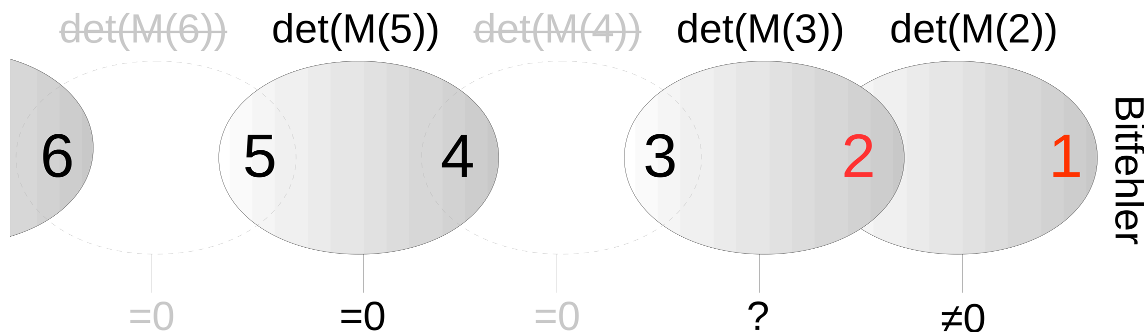

Abbildung 2.13: Identifikation eines 2-Bit Fehlers im Falle der 2-Bit Korrektur mit Parität bei ungerader maximaler Fehleranzahl. Bei gerader Parität wird ein 2-Bit Fehler erwartet, wodurch $det(M(3)) \neq 0$ gelten muss. Ist die Parität ungerade wird ein 1-Bit Fehler erwartet, bei welchem $det(M(3)) = 0$ gilt. Tritt eine dieser Bedingungen nicht ein, z.B. gerade Parität und $det(M(3)) = 0$ oder ungerade Parität und $det(M(3)) \neq 0$, dann ist ein höherer Fehler eingetreten.

2.3.2 Bestimmen der Fehlerstellen

In diesem Abschnitt wird gezeigt, wie die Positionen der fehlerhaften Bits im BCH Code bestimmt werden können, wenn die Anzahl der Fehlerstellen bekannt ist. Zu den bekannten Korrekturansätzen wird ein Konzept nach Okano Imai gezeigt, mit welchem im Galoisfeld $GF(2^m)$ bei 3 Syndromkomponenten mit Breite m eine Tabelle mit Eingangs- und Ausgangswortsbreite m verwendet werden kann, um die 3 Fehlerstellen zu bestimmen. In dem Verfahren nach Okano Imai wird die 4-Bit Korrektur auf eine Berechnung von 4 Gleichungen zweiten Grades und eine Gleichung dritten Grades zurückgeführt. Zu diesen Gleichungen sind Implementierungen in Hardware bekannt. In der vorliegenden Arbeit konnte gezeigt werden, dass eine Gleichung zweiten Grades eingespart werden kann, was auch die Hardwareimplementierung vereinfacht.

Jeder Fehlerposition i ist im Galoisfeld ein eindeutiger Wert α^i zugeordnet, welcher in der ersten Zeile der H-Matrix abzulesen ist. Die Exponenten i von α^i sind modulo $2^m - 1$ zu interpretieren,

wobei $n = 2^m - 1$ die Länge des Codes ist. Sei k die maximal zu korrigierende Anzahl an Fehlern. Bei einem t-Bit Fehler sind die Fehlerposition ist durch Lösen der folgenden Gleichungen bestimmbar.

$$\begin{aligned} \alpha^{i_1} + \alpha^{i_2} + ... + \alpha^{i_t} &= s_1 \\ \alpha^{3\times(i_1)} + \alpha^{3\times(i_2)} + ... + \alpha^{3\times(i_t)} &= s_3 \\ &... \\ \alpha^{(k\times 2+1)\times i_1} + \alpha^{(k\times 2+1)\times i_2} + ... + \alpha^{(k\times 2+1)\times i_t} &= s_{2\times k+1} \end{aligned} \tag{2.18}$$

Die Werte $i_1, ..., i_t$ sind hier die Positionen der Fehler. Dieses Gleichungssystem ergibt sich aus der Multiplikation des Fehlervektors r mit der H-Matrix $r \times H = S$. Ein Lösungsverfahren besteht darin, ein Lokatorpolynom $p(x)$ zu bilden, dessen Nullstellen die Lösungen des Gleichungssystems sind.

$$p(x) = (x + \alpha^{i_1}) \times (x + \alpha^{i_2}) \times ... \times (x + \alpha^{i_t}) \tag{2.19}$$

Die Nullstellen sind somit durch

$$0 = (x + \alpha^{i_1}) \times (x + \alpha^{i_2}) \times ... \times (x + \alpha^{i_t}) \tag{2.20}$$

bestimmt. Die Werte der Nullstellen sind $\alpha^{i_1}, ..., \alpha^{i_t}$.

Da im Falle der Fehlerkorrektur die Werte $\alpha_1^i, ..., \alpha_t^i$ unbekannt sind, muss zur Ermittlung dieser Stellen diese Nullstellengleichung derartig umgeformt werden, dass alle Werte $\alpha_1^i, ..., \alpha_k^i$ durch Syndromkomponenten dargestellt werden. Nach solchen Umformungen ist es möglich, aus den Syndromen die Werte der Nullstellen und somit die Fehlerstellen zu berechnen. Vereinfachte Lösungen für dieses Problem existieren für bis zu den 4-Bit Fall.

An dieser Stelle weisen wir darauf hin, dass allgemeine Lösungen für algebraische Gleichungen bis 4ten Grades möglich sind, aber nach dem Satz von Abel-Ruffini [Bew02] nicht allgemein für Gleichungen 5ten Grades oder höher.

Da die Syndromkomponenten immer eindeutig einem Fehler entsprechend, kann eine große Tabelle verwendet werden, welche jedes Syndrom einer Fehlerkombination zuweist. Eine solche Tabelle würde jedoch für hohe Fehlerzahlen unplausibel groß werden, da jede Kombination an Syndromkomponentenwerten in eine Kombination an Fehlerwerten überführt werden muss.

Die folgenden Erklärungen folgen zunächst den Ansätzen von Okano Imai.

2.3.2.1 Bestimmung der Fehlerstellen für den 1-Bit Fall

Für den 1-Bit Fehler ist das Lokatorpolynom nach Umformung

$$p(x) = s_1 + x \tag{2.21}$$

mit der Nullstelle

$$x = s_1 = \alpha^i \tag{2.22}$$

Da der Wert von α^X für jede verschiedene Fehlerstelle X verschieden ist, kann somit die Position des Fehlers i anhand einer Tabelle durch die Syndromkomponente s_1 eindeutig bestimmt werden.

2.3.2.2 Bestimmung der Fehlerstellen für den 2-Bit Fall

Im Falle eines 2-Bit Fehlerfalls ist das Lokatorpolynom

$$p(x) = (x+\alpha^i)(x+\alpha^j) \tag{2.23}$$

Für den 2-Bit Fehlerfall ist das Gleichungssystem der Syndromkomponenten umfangreicher als bei 1-Bit Fehlern:

$$\begin{aligned} \alpha^i + \alpha^j &= s_1 \\ \alpha^{3i} + \alpha^{3j} &= s_3 \end{aligned} \tag{2.25}$$

Das Lokatorpolynom kann wie folgt umgeformt werden. Dabei gilt im binären Fall für alle w: $2 \times w = w + w = 0$, somit für die Umformung $2s_3 = 0$.

$$\begin{aligned} 0 &= (x+\alpha^i) \times (x+\alpha^j) \\ &= x^2 + x \times (\alpha^i + \alpha^j) + \alpha^i \times \alpha^j \\ &= x^2 + x \times s_1 + \alpha^{i+j} \times \frac{s_1}{s_1} \\ &= x^2 + x \times s_1 + \alpha^{i+j} \times \frac{\alpha^i + \alpha^j}{s_1} \\ &= x^2 + x \times s_1 + \frac{\alpha^{i+2j} + \alpha^{2i+j}}{s_1} \\ &= x^2 + x \times s_1 + \frac{\alpha^{i+2j} + \alpha^{2i+j} + 2s_3}{s_1} \\ &= x^2 + x \times s_1 + \frac{\alpha^{i+2j} + \alpha^{2i+j} + \alpha^{3i} + \alpha^{3j} + s_3}{s_1} \\ &= x^2 + x \times s_1 + \frac{(\alpha^i + \alpha^j)^3 + s_3}{s_1} \\ &= x^2 + x \times s_1 + \frac{s_1^3 + s_3}{s_1} = 0 \end{aligned} \tag{2.27}$$

Diese Umformungen sind nur möglich, falls α^i und α^j verschieden sind (somit $s_1 \neq 0$). Durch $x = s_1 \times z$ ergibt sich nach Okano Imai [OI87]

$$\begin{aligned} 0 &= (s_1 \times z)^2 + (s_1 \times z) \times s_1 + \frac{s_1^3 + s_3}{s_1} \\ 0 &= s_1^2 \times z^2 + s_1^2 \times z + \frac{s_1^3 + s_3}{s_1} \\ 0 &= z^2 + z + \frac{s_1^3 + s_3}{s_1^3} \\ 0 &= z^2 + z + (1 + \frac{s_3}{s_1^3}) \end{aligned} \tag{2.29}$$

$$z^2 + z = (1 + \frac{s_3}{s_1^3}) = C$$
$$(z+1) \times z = (1 + \frac{s_3}{s_1^3})$$

Durch die Konstruktion der Gleichung existieren für jedes $C = 1 + \frac{s_3}{s_1^3}$ zwei Lösungen z_1 und z_2. Falls z_1 eine Lösung ist, ist $z_2 = z_1 + 1$ die zweite Lösung.

$$\begin{aligned}(z_1+1) \times z_1 &= (z_2+1) \times z_2 = (1 + \frac{s_3}{s_1^3}) \\ (z_1+1) \times z_1 &= ((z_1+1)+1) \times (z_1+1) \\ (z_1+1) \times z_1 &= (z_1+0) \times (z_1+1) \\ (z_1+1) \times z_1 &= z_1 \times (z_1+1)\end{aligned} \tag{2.31}$$

Daher lässt sich für jedes C eine Tabelle mit je einem z_1 generieren, welches das Gleichungssystem löst. Anschließend kann mittels $s_1 \times z_1 = \alpha^i$ die fehlerhafte Stelle berechnen. Die zweite Stelle ist daher $\alpha^j = s_1 \times (z_1 + 1) = \alpha^i + s_1$. Anschließend lassen sich erneut mit einer Tabelle die Werte für i und j herausfinden.

Aus der Arbeit meines Kollegen Herrn Nordmann geht folgende Vereinfachung hervor. Da die beiden Lösungen mit $z_2 = z_1 + 1$ bis auf das $+1$ identisch sind, kann im binären Fall das letzte Bit weggelassen werden, um den Tabelleninhalt einer Implementierung klein zu halten. Wenn ein Wert aus der Tabelle ausgelesen wird, wird nun jeweils 0 und 1 an den Wert angehangen, um beide Fehlerwerte z_1 und z_2 zu erzeugen. Die Reihenfolge beider Fehler spielt hier keine Rolle.

Der Vorteil dieses Verfahrens gegenüber einer vollständigen Tabelle ist, dass statt einer Tabelle, welche auf ein volles Wort als Eingabe zwei volle Worte als Ausgabe gibt, kann stattdessen nur ein Wort in polynomieller Länge (C) als Eingabe einen Wert in polynomieller Länge (z_1) erzeugen. Dies stellt für eine Hardwareimplementierung eine große Ersparnis dar.

2.3.2.3 Bestimmung der Fehlerstellen für den 3-Bit Fall

Die Positionen i, j, k von 3-Bit Fehlern werden im binären Fall durch die Syndromkomponenten s_1, s_3, s_5 bestimmt. Die Syndromkomponenten hängen mit den Fehlerstellen wie folgt zusammen:

$$\begin{aligned}\alpha^i + \alpha^j + \alpha^k &= s_1 \\ \alpha^{3i} + \alpha^{3j} + \alpha^{3k} &= s_3 \\ \alpha^{5i} + \alpha^{5j} + \alpha^{5k} &= s_5\end{aligned} \tag{2.33}$$

Das Lokatorpolynom für diesen Fall ist wie folgt:

$$\begin{aligned}0 &= (x+\alpha^i) \times (x+\alpha^j) \times (x+\alpha^k) \\ &= x^3 + (\alpha^i + \alpha^j + \alpha^k) \times x^2 + (\alpha^{i+j} + \alpha^{j+k} + \alpha^{k+i}) \times x + \alpha^{i+j+k} \\ &= x^3 + \sigma_1 \times x^2 + \sigma_2 \times x + \sigma_3\end{aligned} \tag{2.35}$$

wobei für $\sigma_1, \sigma_2, \sigma_3$ nach Okano-Imai [OI87] gilt:

$$\begin{aligned} \sigma_1 &= \alpha^i + \alpha^j + \alpha^k = s_1 \\ \sigma_2 &= \alpha^{i+j} + \alpha^{j+k} + \alpha^{k+i} = \frac{s_1^2 \times s_3 + s_5}{s_1^3 + s_3} \\ \sigma_3 &= \alpha^{i+j+k} = s_1^3 + s_3 + \frac{s_1 \times (s_1^2 \times s_3 + s_5)}{s_1^3 + s_3} \end{aligned} \tag{2.37}$$

Diese Berechnung ist nur möglich, falls $s_1^3 + s_3 \neq 0$. Wir nehmen dies zunächst an, dass $s_1^3 + s_3 \neq 0$ (der Sonderfall wird am Ende des Abschnitts behandelt). Wie im 2-Bit Fall muss das Lokatorpolynom umgeformt werden, damit eine einfache Tabelle erzeugbar ist. Mit $x = y + \sigma_1$ ergibt sich

$$\begin{aligned} 0 &= (y+\sigma_1)^3 + \sigma_1 \times (y+\sigma_1)^2 + \sigma_2 \times (y+\sigma_1) + \sigma_3 \\ &= y^3 + \sigma_1^2 y + \sigma_1 y^2 + \sigma_1^3 + \sigma_1 y^2 + \sigma_1^3 + \sigma_2 y + \sigma_2\sigma_1 + \sigma_3 \\ &= y^3 + \sigma_1^2 y + \sigma_2 y + \sigma_2\sigma_1 + \sigma_3 \\ &= y^3 + (\sigma_2 + \sigma_1^2) \times y + \sigma_2\sigma_1 + \sigma_3 \\ &= y^3 + \eta \times y + \delta \end{aligned} \tag{2.39}$$

Wobei $\eta = \sigma_2 + \sigma_1^2$ und $\delta = \sigma_2\sigma_1 + \sigma_3$. Zunächst wird $\eta \neq 0$ vorausgesetzt.

$$\begin{aligned} 0 &= y^3 + \eta \times y + \delta \\ &= \frac{y^3}{\eta^{\frac{3}{2}}} + \frac{\eta \times y}{\eta^{\frac{3}{2}}} + \frac{\delta}{\eta^{\frac{3}{2}}} \\ &= \left(\frac{y}{\eta^{\frac{1}{2}}}\right)^3 + \frac{y}{\eta^{\frac{1}{2}}} + \frac{\delta}{\eta^{\frac{3}{2}}} \\ &= z^3 + z + C \end{aligned} \tag{2.41}$$

Wobei

$$\begin{aligned} z &= \frac{y}{\eta^{\frac{1}{2}}} = \frac{x + \sigma_1}{(\sigma_2 + \sigma_1^2)^{\frac{1}{2}}} \\ C &= \frac{\delta}{\eta^{\frac{3}{2}}} = \frac{\sigma_2\sigma_1 + \sigma_3}{(\sigma_2 + \sigma_1^2)^{\frac{3}{2}}} \end{aligned} \tag{2.43}$$

Für den Fall $\eta = 0$ ist eine solche Division nicht möglich. Dieser Fall wird weiter unten geklärt. Anhand der Gleichung $z^3 + z = C$ kann wiederum eine Tabelle erstellt werden, in welcher für jedes C die drei Lösungen z_1, z_2, z_3 für z eingetragen sind. Diese Tabelle stellen wir hier durch die Funktion $t_3(C) = [z_1, z_2, z_3]$ dar. Daher ergeben sich die Werte x_1, x_2, x_3 für x wie folgt aus den z

Werten einer solchen Tabelle:

$$\begin{aligned} x_1 &= z_1 \times (\sigma_2 + \sigma_1^2)^{\frac{1}{2}} + \sigma_1 \\ x_2 &= z_2 \times (\sigma_2 + \sigma_1^2)^{\frac{1}{2}} + \sigma_1 \\ x_3 &= z_3 \times (\sigma_2 + \sigma_1^2)^{\frac{1}{2}} + \sigma_1 \end{aligned} \tag{2.45}$$

Die Positionen der Fehlerstellen i, j, k können durch folgenden Zusammenhang bestimmt werden.

$$\alpha^i = x_1, \quad \alpha^j = x_2, \quad \alpha^k = x_3 \tag{2.46}$$

Tabelle mit zwei Werten

Anstelle der Tabelle t_3, welche 3 Werte zu einem C ausgibt, kann die Ausgabe der Tabelle auf 2 Werte (z_1, z_2) reduziert werden. Berechnet man aus diesen x_1, x_2, so gilt durch die Syndromgleichung:

$$\begin{aligned} s_1 &= \alpha^i + \alpha^j + \alpha^k \\ s_1 &= x_1 + x_2 + x_3 \\ x_3 &= x_1 + x_2 + s_1 \end{aligned} \tag{2.48}$$

Die dritte Fehlerstelle ist somit aus den anderen beiden Fehlerstellen berechenbar.

Eine Tabelle kann durch eine XOR Verknüpfung der Bits (mit Negation) einzelner Positionen der Eingabe berechnet werden. Ist die Länge der Eingabe M (entsprechend dem $GF(2^M)$), so kann jedes Bit der Ausgabe der Tabelle durch eine Schaltung der Tiefe um $log_2(M)$ XOR-Gattern bestimmt werden. Jedes Bit der Ausgabe kann dabei parallel und unabhängig voneinander berechnet werden, dies benötigt beispielsweise eine OR Schaltung über die Eingaben, welche dieses Bit auf 1 setzen. Die Tiefe einer solchen Schalt würde höchstens M Stufen betragen ($log_2(2^M)$). Die derartige Reduktion der Tabelle auf 2 statt 3 Ausgaben (bis auf eventuelle Optimierungen, welche dies ermöglicht) bewirkt keine Einsparung von Laufzeit. Eine Einsparung an Fläche der Schaltung zur Bestimmung von z_3 ist jedoch zu erwarten.

Tabelle mit einem Wert

Statt eine Tabelle mit 2 oder 3 Werten pro Konstante zu erzeugen, kann eine Tabelle mit lediglich einer Ausgabe verwendet werden. Die Tabelle wird in diesem Fall verwendet, um einen festen Wert z_1 auszugeben. Aus diesem Wert z_1 wird wie zuvor eine Fehlerstelle berechnet.

$$\alpha^i = z_1 \times (\sigma_2 + \sigma_1^2)^{\frac{1}{2}} + \sigma_1 \tag{2.50}$$

Diese Fehlerstelle kann nun genutzt werden, um die restlichen Stellen wie einen 2-Bit Fehler zu

berechnen.

$$
\begin{aligned}
s'_1 &= s_1 + \alpha^i = \alpha^j + \alpha^k \\
s'_3 &= s_3 + \alpha^{3i} = \alpha^{3j} + \alpha^{3k} \\
s'_5 &= s_5 + \alpha^{5i} = \alpha^{5j} + \alpha^{5k} \\
&\vdots \\
s'_{2m-1} &= s_{2m-1} + \alpha^{(2m-1)\times i} = \alpha^{(2m-1)\times j} + \alpha^{(2m-1)\times k}
\end{aligned}
\tag{2.52}
$$

Trat der 3-Bit Fehler an den Stellen i, j, k auf, so entsprechen die berechneten Werte s'_1, s'_3, s'_5 den Syndromkomponenten eines 2-Bit Fehlers an den Stellen j, k. In einem Folgeschritt kann somit ein Verfahren zur Korrektur von 2-Bit Fehlern verwendet werden, um die Fehlerstellen j, k aus s'_1, s'_3, s'_5 zu berechnen und den 3-Bit Fehler an den Stellen i, j, k durch diese zu korrigieren.

Dieses Verfahren kann iterativ verwendet werden, um durch Berechnung einer Fehlerstelle einen einfacheren Algorithmus zu nutzen, um die nächste Fehlerstelle zu berechnen.

Zur Effizienz muss jedoch gesagt werden, dass dieses Verfahren allgemein langsamer ist, als die Verwendung einer Tabelle mit 3 Werten pro Eingabe. Bei diesem Ansatz muss eine erste Fehlerstelle vollständig bestimmt werden, bevor die weiteren Fehlerstellen durch eine zusätzliche 2-Bit Korrektur bestimmt werden können. Die Laufzeit der Tabelle ist identisch, da alle 3 Werte getrennt voneinander berechnet werden können und die Laufzeit der Tabelle weiterhin logarithmisch von der Länge und Anzahl der Eingaben abhängt. In einer Hardwarestruktur wäre es durch die Vereinfachung möglich, einen Teil des Platzes zu sparen, da es theoretisch möglich ist, existierende Hardware zur 2-Bit Korrektur zu nutzen.

Betrachtung $s_1^3 + s_3 = 0$:
Falls die $s_1^3 + s_3 = 0$ gilt, ist eine Division durch $s_1^3 + s_3$ nicht möglich. In diesem Fall gilt

$$
\begin{aligned}
0 &= s_1^3 + s_3 \\
s_1^3 &= s_3 \\
(\alpha^i + \alpha^j + \alpha^k)^3 &= \alpha^{3i} + \alpha^{3j} + \alpha^{3k} \\
\alpha^{3i} + \alpha^{3j} + \alpha^{3k} + \alpha^{2i+j} + \alpha^{2i+k} + \alpha^{2j+i} + \alpha^{2j+k} + \alpha^{2k+i} + \alpha^{2k+j} &= \alpha^{3i} + \alpha^{3j} + \alpha^{3k} \\
\alpha^{2i+j} + \alpha^{2i+k} + \alpha^{2j+i} + \alpha^{2j+k} + \alpha^{2k+i} + \alpha^{2k+j} &= 0 \\
(\alpha^i + \alpha^j) \times (\alpha^i + \alpha^k) \times (\alpha^j + \alpha^k) &= 0
\end{aligned}
\tag{2.54}
$$

Im 3-Bit Fehlerfall müssen i, j, k verschieden sein, somit muss jede Summe $\alpha^x + \alpha^y \forall x, y \in \{i, j, k\}, x \neq y$ ungleich 0 sein. Das bedeutet, dass $s_1^3 + s_3 = 0$ nur gelten kann, wenn kein 3-Bit Fehler aufgetreten ist. Sollte somit $s_1^3 + s_3 = 0$ eintreten, kann die Bestimmung der Fehlerpositionen abgebrochen werden. Dies wird in den Fällen relevant, in welchen die Anzahl der Fehler nicht im Voraus bestimmt wurde.

Ausnahme $\eta = 0$:
Falls $\eta = \sigma_2 + \sigma_1^2 = 0$ gilt, war eine Division durch $\eta^{\frac{1}{2}}$ nicht möglich.

$$\begin{aligned} 0 &= \eta \\ &= \sigma_2 + \sigma_1^2 \\ &= \left(\frac{s_1^2 \times s_3 + s_5}{s_1^3 + s_3}\right) + s_1^2 \end{aligned} \tag{2.56}$$

Es gilt dann

$$\begin{aligned} 0 &= y^3 + \eta \times y + \delta \\ &= y^3 + \delta \\ y &= \sqrt[3]{\delta} \end{aligned} \tag{2.58}$$

Somit kann y direkt durch die dritte Wurzel berechnet werden. Für die Fehlerstellen x ergibt sich somit:

$$\begin{aligned} x &= \sqrt[3]{\delta} + \sigma_1 \\ &= \sqrt[3]{\sigma_2\sigma_1 + \sigma_3} + \sigma_1 \\ &= \sqrt[3]{\left(\frac{s_1^2 \times s_3 + s_5}{s_1^3 + s_3}\right) \times s_1 + \left(s_1^3 + s_3 + \frac{s_1 \times (s_1^2 \times s_3 + s_5)}{s_1^3 + s_3}\right)} + s_1 \\ &= \sqrt[3]{2 \times \frac{s_1 \times (s_1^2 \times s_3 + s_5)}{s_1^3 + s_3} + s_1^3 + s_3} + s_1 \\ &= \sqrt[3]{s_1^3 + s_3} + s_1 \end{aligned} \tag{2.60}$$

2.3.2.4 Bestimmung der Fehlerstellen für den 4-Bit Fall

Der folgende Abschnitt beschäftigt sich mit der Lösung von Gleichungen vierten Grades im Körper $GF(2^m)$ mit dem Ziel, eine effiziente Hardwareimplementierung oder Softwarelösung zu ermöglichen. Ziel dieser Untersuchung ist es, das zunächst beschriebene Verfahren zur Bestimmung von 4-Bit Fehler zu vereinfachen.

Es ist bekannt, dass eine Gleichung vierten Grades im Körper der reellen Zahlen unter Verwendung einer Hilfsgleichung dritten Grades und Gleichungen zweiten Grades algebraisch lösbar ist. In den Grundzügen wird von Okano Imai [OI87] ein Ansatz zur Lösung von Gleichungen vierten Grades für das Galoisfeld $GF(2^m)$ auf etwa einer Seite beschrieben. Diese Grundüberlegungen werden zunächst detailliert ausgearbeitet und gezeigt, dass eine Gleichung zweiten Grades eingespart werden kann, was auch zu einer Hardwareeinsparung führt. Dabei wird auch gezeigt, dass bei der Bestimmung der Koeffizienten des Lokatorpolynoms nach Wicker zusätzlich ein Sonderfall zu beachten ist.

Insgesamt wird im 4-Bit Fall im $GF(2^m)$ ein Lokatorpolynom gebildet, zu welchem anschließend die Nullstellen bestimmt werden, um die Fehlerstellen i, j, k, l zu erhalten. Zunächst werden die Koeffizienten des Lokatorpolynoms aus den Syndromkomponenten bestimmt, anschließend

wird nach Okano Imai eine Aufteilung in zwei quadratische Gleichungen durch Lösung einer Gleichung dritten Grades und zwei Gleichungen zweiten Grades vorgenommen. Die Lösungen dieser quadratischen Gleichungen bilden zusammen die Lösungen des Lokatorpolynoms vierten Grades.

Der Abschnitt ist wie folgt gegliedert. Zunächst wird beschrieben, wie das Lokatorpolynom aus den Syndromkomponenten bestimmt werden kann. Im Anschluss wird gezeigt, wird das Verfahren zur Lösung des Lokatorpolynoms dargestellt. Es werden unterschiedliche Lösungsverfahren zur Bestimmung der Nullstellen des Lokatorpolynoms vierten Grades entwickelt und verglichen. Das Gesamtverfahren wird somit algoritmisch beschrieben.

Bestimmung des Lokatorpolynoms Nach Okano-Imai [OI87] ist das Lokatorpolynom bei einem 4-Bit Fehler

$$\begin{aligned} p(x) &= (x+\alpha^i)(x+\alpha^j)(x+\alpha^k)(x+\alpha^l) \\ &= x^4+\sigma_1 x^3+\sigma_2 x^2+\sigma_3 x+\sigma_4 = 0 \end{aligned} \tag{2.62}$$

wobei gilt

$$\begin{aligned} \sigma_1 &= \alpha^i+\alpha^j+\alpha^k+\alpha^l \\ \sigma_2 &= \sum_{x<y} \alpha^x\alpha^y \\ \sigma_3 &= \sum_{x<y<z} \alpha^x\alpha^y\alpha^z \\ \sigma_4 &= \alpha^i\alpha^j\alpha^k\alpha^l \end{aligned} \tag{2.64}$$

In dieser Darstellung sind die Nullstellen von $p(x)$ die Werte $\alpha^i, \alpha^j, \alpha^k, \alpha^l$. Dieses Lokatorpolynom kann zusätzlich in einer anderen Variante dargestellt werden:

$$\begin{aligned} p'(x) &= (1+x\times\alpha^i)(1+x\times\alpha^j)(1+x\times\alpha^k)(1+x\times\alpha^l) \\ &= 1+\sigma'_1 x+\sigma'_2 x^2+\sigma'_3 x^3+\sigma'_4 x^4 = 0 \end{aligned} \tag{2.66}$$

wobei gilt

$$\begin{aligned} \sigma'_1 &= \alpha^i+\alpha^j+\alpha^k+\alpha^l = \sigma_1 \\ \sigma'_2 &= \sum_{x<y} \alpha^x\alpha^y = \sigma_2 \\ \sigma'_3 &= \sum_{x<y<z} \alpha^x\alpha^y\alpha^z = \sigma_3 \\ \sigma'_4 &= \alpha^i\alpha^j\alpha^k\alpha^l = \sigma_4 \end{aligned} \tag{2.68}$$

Hier sind die Nullstellen von $p'(x)$ gleich $\alpha^{-i}, \alpha^{-j}, \alpha^{-k}, \alpha^{-l}$. Wir betrachten im Folgenden die erste Variante des Lokatorpolynoms aus Gleichung 2.62. Nach Wicker[Wic95] gilt für die Koeffi-

zienten des Lokatorpolynoms:

$$\begin{aligned}
\sigma_1 &= \alpha^i + \alpha^j + \alpha^k + \alpha^l = s_1 \\
\sigma_2 &= \sum_{x<y} \alpha^x \alpha^y = \frac{s_1(s_7 + s_1^7) + s_3(s_1^5 + s_5)}{s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5)} \\
\sigma_3 &= \sum_{x<y<z} \alpha^x \alpha^y \alpha^z = (s_1^3 + s_3) + s_1 + \sigma_2 \\
\sigma_4 &= \alpha^i \alpha^j \alpha^k \alpha^l = \frac{(s_5 + s_1^2 s_3) + (s_1^3 + s_3)\sigma_2}{s_1}
\end{aligned} \tag{2.70}$$

Im Fall, dass $s_1 = 0$ gilt, ist die Berechnung von σ_4 derartig nicht möglich. Dieser Fall $s_1 = 0$ kann ab 3-Bit Fehlern auftreten, wenn sich alle α^x der Fehlerstellen zu 0 aufaddieren. Es wurde im Rahmen dieser Arbeit festgestellt, dass ein solches $s_1 = 0$ im Fall eines 4-Bit Fehlers auftreten kann und diese Formel nach Wicker eine Fallentscheidung benötigt, um generell verwendbar zu sein. Eine alternative, allgemeingültige Formel zur Darstellung von σ_4 durch die Syndromkomponenten nach Tzschach [TH13] (entsprechend der Darstellung von Okano Imai [OI87]) kann stattdessen genutzt werden:

$$\begin{aligned}
A &= s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5) \\
\sigma_1 &= s_1 \\
\sigma_2 &= \frac{s_1(s_7 + s_1^7) + s_3(s_1^5 + s_5)}{A} \\
\sigma_3 &= \frac{s_1(s_1^3 s_5 + s_1 s_7) + s_3(s_1^6 + s_3^2)}{A} \\
\sigma_4 &= \frac{s_1^3(s_1^7 + s_7) + s_3(s_1^7 + s_1 s_3^2 + s_7) + s_5(s_1^5 + s_1^2 s_3 + s_5)}{A}
\end{aligned} \tag{2.72}$$

Vergleich der Formeln zur Berechnung des Lokatorpolynoms Diese Formel für σ_4 berechnet sich anders als die Formel nach Wicker und benötigt keine Division durch s_1. Im Folgenden wird gezeigt, dass diese Formeln äquivalent sind im Fall $s_1 \neq 0$.

$$\begin{aligned}
\sigma_2 &= \frac{s_1(s_7 + s_1^7) + s_3(s_1^5 + s_5)}{s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5)} = \frac{s_1(s_7 + s_1^7) + s_3(s_1^5 + s_5)}{A} \\
\sigma_{4,\text{wicker}} &= \frac{(s_5 + s_1^2 s_3) + (s_1^3 + s_3)\sigma_2}{s_1} = \frac{(s_5 + s_1^2 s_3) + (s_1^3 + s_3)\frac{s_1(s_7 + s_1^7) + s_3(s_1^5 + s_5)}{s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5)}}{s_1} \\
&= \frac{(s_5 + s_1^2 s_3)}{s_1} + \frac{(s_1^3 + s_3)(s_1(s_7 + s_1^7) + s_3(s_1^5 + s_5))}{s_1(s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5))} \\
&= \frac{(s_5 + s_1^2 s_3)(s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5)) + (s_1^3 + s_3)(s_1(s_7 + s_1^7) + s_3(s_1^5 + s_5))}{s_1(s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5))} \\
&= \frac{(s_5 + s_1^2 s_3)(s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5))}{s_1(s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5))} + \frac{(s_1^3 + s_3)(s_1(s_7 + s_1^7) + s_3(s_1^5 + s_5))}{s_1(s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5))} \\
&\vdots
\end{aligned} \tag{2.74}$$

$$
\begin{aligned}
&= \frac{(s_5)(s_3(s_1^3+s_3)+s_1(s_1^5+s_5)) + (s_1^2 s_3)(s_3(s_1^3+s_3)+s_1(s_1^5+s_5))}{s_1(s_3(s_1^3+s_3)+s_1(s_1^5+s_5))} \\
&\quad + \frac{(s_1^3)(s_1(s_7+s_1^7)+s_3(s_1^5+s_5)) + (s_3)(s_1(s_7+s_1^7)+s_3(s_1^5+s_5))}{s_1(s_3(s_1^3+s_3)+s_1(s_1^5+s_5))} \\
&= \frac{(s_5 s_3(s_1^3+s_3)) + (s_5 s_1(s_1^5+s_5)) + (s_1^2 s_3^2(s_1^3+s_3)) + (s_1^3 s_3(s_1^5+s_5))}{s_1(s_3(s_1^3+s_3)+s_1(s_1^5+s_5))} \\
&\quad + \frac{(s_1^4(s_7+s_1^7)) + (s_1^3 s_3(s_1^5+s_5)) + (s_3 s_1(s_7+s_1^7)) + (s_3^2(s_1^5+s_5))}{s_1(s_3(s_1^3+s_3)+s_1(s_1^5+s_5))} \\
&= \frac{s_5 s_3 s_1^3 + s_5 s_3^2 + s_5 s_1^6 + s_5^2 s_1 + s_1^5 s_3^2 + s_1^2 s_3^3 + s_1^8 s_3 + s_1^3 s_3 s_5}{s_1(s_3(s_1^3+s_3)+s_1(s_1^5+s_5))} \\
&\quad + \frac{s_1^4 s_7 + s_1^{11} + s_1^8 s_3 + s_1^3 s_3 s_5 + s_3 s_1 s_7 + s_3 s_1^8 + s_3^2 s_1^5 + s_3^2 s_5}{s_1(s_3(s_1^3+s_3)+s_1(s_1^5+s_5))} \\
&= \frac{s_1^6 s_5 + s_1 s_5^2 + s_1^2 s_3^3 + s_1^8 s_3 + s_1^3 s_3 s_5 + s_1^4 s_7 + s_1^{11} + s_1 s_3 s_7}{s_1(s_3(s_1^3+s_3)+s_1(s_1^5+s_5))} \\
&= \frac{\mathbf{s}_1}{\mathbf{s}_1} \times \frac{s_1^5 s_5 + s_5^2 + s_1 s_3^3 + s_1^7 s_3 + s_1^2 s_3 s_5 + s_1^3 s_7 + s_1^{10} + s_3 s_7}{s_3(s_1^3+s_3)+s_1(s_1^5+s_5)} \\
&= \frac{s_1^3(s_7+s_1^7) + s_3(s_1 s_3^2 + s_1^7 + s_7) + s_5(s_1^5 + s_5 + s_1^2 s_3)}{s_3(s_1^3+s_3)+s_1(s_1^5+s_5)} \\
&= \frac{s_1^3(s_1^7+s_7) + s_3(s_1^7 + s_1 s_3^2 + s_7) + s_5(s_1^5 + s_1^2 s_3 + s_5)}{A} = \sigma_{4,\text{tzschach}}
\end{aligned}
$$

Die Formeln sind somit außer dem $s_1 = 0$ Fall der Formel nach Wicker mathematisch äquivalent. Eine Überprüfung der Korrektheit der Formeln kann durch Einsetzen der variablen Fehlerstellen $\alpha^i, \alpha^j, \alpha^k, \alpha^l$ geschehen. Dabei werden die Syndromkomponenten in die spezifischen Gleichungen der Fehlerstellen im 4-Bit Fall aufgeschlüsselt und die Formeln für $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ mit diesen nach Wicker bzw. Tzschach ausgerechnet. Das Ergebnis muss gleich der Formeln der Werte von $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ entsprechend dem Lokatorpolynom sein 2.64 sein. Bei Divisionen sind dabei die Fälle zu betrachten, in denen der Divisor 0 wird.

Aus Implementierungssicht stellt sich nun die Frage, ob eine der beiden Formeln effizienter zu berechnen ist. Jenseits des Problemfalls $s_1 = 0$ baut die Formel für σ_4 nach Wicker auf den vorherigen Berechnungen von σ_2 auf. Je nach Implementierung könnte dies einen Vorteil gegenüber den längeren Gleichungen nach Okano Imai ergeben. Daher sollen beide Optionen vergleichen werden. In der Formel nach Tzschach 2.72 wird ein Wert A berechnet, welcher in allen Koeffizienten σ zur Berechnung verwendet wird. Hingegen in der Formel nach Wicker 2.70 wird σ_2 in der Berechnung der Folgewerte verwendet. Der Wert A ist der Divisor in der Berechnung von σ_2, die Berechnung von σ_2 ist somit aufwändiger. Bei einer parallelen Implementierung könnten Divisor und Dividend aufgrund nahezu identischer Struktur nahezu gleich schnell berechnet werden (die exponenzierten Werte von s_1 können durch Tabellen ausgelesen werden), der zusätzliche Aufwand sollte somit mit der Division einzuschätzen sein. Für σ_4 ist die Berechnung nach Tzschach aufwändiger, der Dividend besteht aus drei Summanden, welche jeweils aus Produkten einer Syndromkomponente mit einer weiteren Summe bestehen. Diese Formel ist daher aufwändiger zu

berechnen, als A. Die Formel nach Wicker kann wie folgt aufgebrochen werden:

$$\begin{aligned}\sigma_4 &= \frac{(s_5 + s_1^2 s_3) + (s_1^3 + s_3)\sigma_2}{s_1} \\ &= \frac{(s_5 + s_1^2 s_3)}{s_1} + \frac{(s_1^3 + s_3)}{s_1}\sigma_2 \\ &= s_1 s_3 + \frac{s_5}{s_1} + \frac{(s_1^3 + s_3)}{s_1}\sigma_2\end{aligned} \tag{2.76}$$

Hier ist zu erwarten, dass die Teile der Formel $s_1 s_3 + \frac{s_5}{s_1}$ und $\frac{(s_1^3+s_3)}{s_1}$ schneller zu berechnen sind, als *sigma*$_2$ und A. Nach Berechnung von σ_2 muss noch eine Multiplikation und eine Addition durchgeführt werden. Die erwarteten längsten Pfade in den Berechnungen stufen wir wie folgt ein:

- Nach dem Verfahren nach Wicker :

$$(s_1(s_7 + s_1^7) + X)/Y * Z + W$$

  wobei $X = s_3(s_1^5 + s_5)$, $Y = s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5)$, $Z = (s_1^3 + s_3)/s_1$ und $W = s_1 s_3 + s_5/s_1$.

- Nach dem Verfahren nach Tzschach :

$$(s_3(s_1 s_3^2 + X) + Y + Z)/A$$

  wobei $X = s_1^7 + s_7$, $Y = s_1^3(s_1^7 + s_7)$, $Z = s_5(s_1^2 s_3 + s_1^5 + s_5)$ und $A = s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5)$

Die Berechnung der Variablen X, Y, Z, W, A ist dabei jeweils nicht Teil des längsten Pfades, hier kann erwartet werden, dass die Werte dieser Variablen innerhalb des längsten Pfades bereits zur Verfügung stehen.

Diese Untersuchung genügt nicht, um den Schluss zu ziehen, dass eine Variante schneller zu berechnen ist. Beide Vorgehen erscheinen bei dieser Betrachtung plausibel. Eine exakte Bestimmung der schnellsten Variante erfordert den Vergleich konkreter Implementierungen, welche wir an dieser Stelle nicht geben. Hier besteht durchaus die Möglichkeit, dass die Schnelligkeit des Verfahrens von den konkreten Fehlern abhängt. Wir verfolgen hier somit vorerst beide Formeln und verwenden eine Fallunterscheidung $s_1 \stackrel{?}{=} 0$ für die Formel nach Wicker (2.70). Diese Fallunterscheidung erfolgt folgendermaßen. Ist $s_1 \neq 0$, werden die Formeln nach Wicker (2.70) verwendet. Falls $s_1 = 0$, verwenden wir die Formel nach Tzschach (2.72) und setzen $s_1 = 0$ wie folgt ein.

$$\begin{aligned}\sigma_4 &= \frac{s_1^3(s_1^7 + s_7) + s_3(s_1^7 + s_1 s_3^2 + s_7) + s_5(s_1^5 + s_1^2 s_3 + s_5)}{s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5)} \\ \sigma_{4,s_1=0} &= \frac{0(0 + s_7) + s_3(0 + 0 s_3^2 + s_7) + s_5(0 + 0 s_3 + s_5)}{s_3(0 + s_3) + 0(0 + s_5)} \\ &= \frac{s_7 s_3 + s_5^2}{s_3^2}\end{aligned} \tag{2.78}$$

In Fall $s_1 = 0$ ist die Formel für σ_4 besonders kurz. Zusammenfassend werden nun die Verfahren zur Bestimmung der Koeffizienten des Lokatorpolynoms kurz dargestellt.

Bestimmung der Koeffizienten des Lokatorpolynoms nach Wicker mit Fallunterscheidung

$$\begin{aligned}\sigma_1 &= s_1 \\ \sigma_2 &= \frac{s_1(s_7+s_1^7)+s_3(s_1^5+s_5)}{s_3(s_1^3+s_3)+s_1(s_1^5+s_5)} \\ \sigma_3 &= (s_1^3+s_3)+s_1+\sigma_2 \\ \sigma_4 &= \begin{cases} \frac{s_7s_3+s_5^2}{s_3^2}, & \text{falls } s_1=0 \\ \frac{(s_5+s_1^2s_3)+(s_1^3+s_3)\sigma_2}{s_1}, & \text{sonst} \end{cases}\end{aligned} \tag{2.80}$$

Bestimmung der Koeffizienten des Lokatorpolynoms nach Tzschach

$$\begin{aligned}A &= s_3(s_1^3+s_3)+s_1(s_1^5+s_5) \\ \sigma_1 &= s_1 \\ \sigma_2 &= \frac{s_1(s_7+s_1^7)+s_3(s_1^5+s_5)}{A} \\ \sigma_3 &= \frac{s_1(s_1^3s_5+s_1s_7)+s_3(s_1^6+s_3^2)}{A} \\ \sigma_4 &= \frac{s_1^3(s_1^7+s_7)+s_3(s_1^7+s_1s_3^2+s_7)+s_5(s_1^5+s_1^2s_3+s_5)}{A}\end{aligned} \tag{2.82}$$

Im Folgenden betrachten wir die Lösung des Lokatorpolynoms.

Lösen des Lokatorpolynoms In diesem Abschnitt wird eine detaillierte Beschreibung der von Okano Imai [OI87] kurz dargestellten Verfahrensschritte gegeben. Dabei wird das Lokatorpolynom vierten Grades in zwei Polynome zweiten Grades unterteilt.

$$\begin{aligned}p(x) &= (x^2+px+q)(x^2+p'x+q') = 0 \\ &= x^4+(p+p')x^3+(pp'+q+q')x^2+(pq'+p'q)x+qq' \\ \sigma_1 &= p+p' \\ \sigma_2 &= pp'+q+q' \\ \sigma_3 &= pq'+p'q \\ \sigma_4 &= qq'\end{aligned} \tag{2.84}$$

Da das Lokatorpolynom aus einem Produkt der vier Linearfaktoren besteht, welche den Nullstellen entsprechen, ist diese Unterteilung in zwei Gleichungen zweiten Grades immer möglich. Die Lösung der Gleichung vierten Grades kann damit auf die Lösung der Gleichungen zweiten Grades zurückgeführt werden, welche die Lösungen des Lokatorpolynoms sind.

Es existieren drei verschiedene Möglichkeiten zur Faktorisierung in zwei Gleichungen zweiten Grades. Eine Hilfsgleichung dritten Grades wird verwendet, um eine dieser drei Faktorisierungen

zu bestimmen. Der gesuchte Wert dieser Hilfsgleichung ist dabei ein Produkt pp', welches ermöglicht, im Anschluss Werte für q, q', p, p' zu ermitteln. Die Werte für pp' ergeben sich aus den Faktorisierungen des Lokatorpolynoms:

Aufteilung A

$$\begin{aligned} p(x) &= ((x+\alpha^i)(x+\alpha^j))((x+\alpha^k)(x+\alpha^l)) \\ &= (x^2+(\alpha^i+\alpha^j)x+\alpha^i\alpha^j)(x^2+(\alpha^k+\alpha^l)x+\alpha^k\alpha^l) \\ p_a &= (\alpha^i+\alpha^j) \qquad p'_a = (\alpha^k+\alpha^l) \\ A &= p_a p'_a = (\alpha^i+\alpha^j)(\alpha^k+\alpha^l) \end{aligned} \tag{2.86}$$

Aufteilung B

$$\begin{aligned} p(x) &= ((x+\alpha^i)(x+\alpha^k))((x+\alpha^j)(x+\alpha^l)) \\ &= (x^2+(\alpha^i+\alpha^k)x+\alpha^i\alpha^k)(x^2+(\alpha^j+\alpha^l)x+\alpha^j\alpha^l) \\ p_b &= (\alpha^i+\alpha^k) \qquad p'_b = (\alpha^j+\alpha^l) \\ B &= p_b p'_b = (\alpha^i+\alpha^k)(\alpha^j+\alpha^l) \end{aligned} \tag{2.88}$$

Aufteilung C

$$\begin{aligned} p(x) &= ((x+\alpha^i)(x+\alpha^l))((x+\alpha^k)(x+\alpha^j)) \\ &= (x^2+(\alpha^i+\alpha^l)x+\alpha^i\alpha^l)(x^2+(\alpha^k+\alpha^j)x+\alpha^k\alpha^j) \\ p_c &= (\alpha^i+\alpha^l) \qquad p'_c = (\alpha^k+\alpha^j) \\ C &= p_c p'_c = (\alpha^i+\alpha^l)(\alpha^k+\alpha^j) \end{aligned} \tag{2.90}$$

Diese Hilfsgleichung dritten Grades setzt sich wie folgt zusammen.

$$h(\lambda) = (\lambda+A)(\lambda+B)(\lambda+C) = 0 \tag{2.92}$$

Die Nullstellen dieser Gleichung sind die Werte A, B, C. Die Funktion h ist dabei wie folgt durch die Koeffizienten $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ des Lokatorpolynoms vierten Grades darstellbar.

$$\begin{aligned} h(\lambda) &= (\lambda+A)(\lambda+B)(\lambda+C) \\ &= \lambda^3+(A+B+C)\lambda^2+(AB+AC+BC)\lambda+ABC \\ (A+B+C) &= 0 \\ h(\lambda) &= \lambda^3+(AB+AC+BC)\lambda+ABC \\ &= \lambda^3+\eta\lambda+\delta \\ \eta &= AB+AC+BC \quad = \sigma_2^2+\sigma_1\sigma_3 \\ \delta &= ABC \quad = \sigma_3^2+\sigma_1^2\sigma_4+\sigma_1\sigma_2\sigma_3 \end{aligned} \tag{2.94}$$

Diese Hilfsgleichung kann durch das Verfahren zur Lösung von Gleichungen dritten Grades gelöst

werden (siehe Abschnitt 2.3.2.3). Die Lösungen A, B, C von $h(\lambda) = 0$ ergeben die möglichen Werte für den Wert pp'. Für die Folgeschritte wird lediglich eine Lösung pp' benötigt. Zur einfachen Darstellung nehmen wir in unserer Erklärung an, diese berechnete Lösung sei A mit

$$\begin{aligned} A &= (\alpha^i + \alpha^j)(\alpha^k + \alpha^l) = pp' \\ p' &= \frac{A}{p} \end{aligned} \tag{2.96}$$

Dadurch können q, q', p, p' für die Faktorisierung $p(x) = (x^2 + px + q)(x^2 + p'x + q') = 0$ berechnet werden. Es gilt:

$$\begin{aligned} \sigma_1 &= p + p' \\ \sigma_1 + p + p' &= 0 \\ \sigma_1 + p + \frac{pp'}{p} &= 0 \\ p_{(1,2)}^2 + \sigma_1 p_{(1,2)} + pp' &= 0 \end{aligned} \tag{2.98}$$

Die beiden Lösungen p_1 und p_2 entsprechen den Werten p und p', aus der Gleichung geht jedoch nicht hervor, wie diese Werte zugeordnet sind. Ob p_1 der Wert p oder der Wert p' ist, muss noch somit bestimmt werden.

$$\begin{aligned} q + q' + pp' &= \sigma_2 \\ qq' &= \sigma_4 \\ q' &= \frac{\sigma_4}{q} \\ q + \frac{\sigma_4}{q} + A &= \sigma_2 \\ q_{(1,2)}^2 + (A + \sigma_2) q_{(1,2)} + \sigma_4 &= 0 \end{aligned} \tag{2.100}$$

Die Werte q_1 und q_2 entsprechen den Werten q und q', auch hier ist die Zuordnung durch die Berechnung nicht vorgegeben. Um die Werte p_1, p_2, q_1, q_2 im Anschluss den Werten p, p', q, q' zuzuordnen, wird die Gleichung $\sigma_3 = pq' + p'q$ verwendet. Eine mögliche Belegung kann wie folgt bestimmt werden:

- Setze $p = p_1, p' = p_2$
- Teste, ob $\sigma_3 = pq_2 + p'q_1$ gilt
- Falls der Test erfolgreich ist, dann gilt $q = q_1, q' = q_2$ ansonsten gilt $q = q_2, q' = q_1$.

Somit wurden p, p', q, q' bestimmt und es können die Lösungen der beiden Teilgleichungen

$$\begin{aligned} x^2 + px + q &= 0 \\ x^2 + p'x + q' &= 0 \end{aligned} \tag{2.102}$$

gelöst werden, um die vier Fehlerstellen i, j, k, l zu erhalten. Hierzu wird das Verfahren zum Lösung quadratischer Gleichungen verwendet, deren Lösung bereits beschrieben wurde.

Algorithmus zur Bestimmung der Fehlerstellen nach Okano Imai Der Gesamtvorgang zur Berechnung der Fehlerstellen eines 4-Bit Fehlers nach Okano Imai ist somit:

- Berechne die Syndromkomponenten s_1, s_3, s_5, s_7
- Berechne $A = s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5)$, $\sigma_1 = s_1$,
  $\sigma_2 = (s_1(s_7 + s_1^7) + s_3(s_1^5 + s_5))/A$,
  $\sigma_3 = (s_1(s_1^3 s_5 + s_1 s_7) + s_3(s_1^6 + s_3^2))/A$,
  $\sigma_4 = (s_1^3(s_1^7 + s_7) + s_3(s_1^7 + s_1 s_3^2 + s_7) + s_5(s_1^5 + s_1^2 s_3 + s_5))/A$
- Berechne ein $\lambda = pp'$ mit $\lambda^3 + (\sigma_2^2 + \sigma_1\sigma_3)\lambda + (\sigma_3^2 + \sigma_1^2\sigma_4 + \sigma_1\sigma_2\sigma_3) = 0$
- Berechne die Lösungen $p = p_1, p' = p_2$ von $p_{(1,2)}^2 + \sigma_1 p_{(1,2)} + pp' = 0$
- Berechne die Lösungen q_1, q_2 von $q_{(1,2)}^2 + (pp' + \sigma_2)q_{(1,2)} + \sigma_4 = 0$
- Wenn $\sigma_2 = pq_2 + p'q_1$ gilt dann ist $q = q_1, q' = q_2$, sonst gilt $q = q_2, q' = q_1$.
- Berechne die Lösungen i, j für $x^2 + px + q = 0$
- Berechne die Lösungen k, l für $x^2 + p'x + q' = 0$

Durch diese Berechnungen ergeben sich die Lösungen i, j, k, l des Lokatorpolynoms vierten Grades. Dabei müssen 1 Gleichung dritten Grades und 4 Gleichungen zweiten Grades gelöst werden.

Algorithmus zur Bestimmung der Fehlerstellen nach Okano Imai und Wicker mit Fallunterscheidung Wird die in dieser Arbeit vorgeschlagene Anpassung des Verfahrens nach Wicker verwendet, wird durch eine Fallunterscheidung entschieden, wie die Koeffizienten des Lokatorpolynoms im 4-Bit Fehlerfall bestimmt werden:

- Berechne die Syndromkomponenten s_1, s_3, s_5, s_7
- Berechne $\sigma_1 = s_1$, $\sigma_2 = \frac{s_1(s_7+s_1^7)+s_3(s_1^5+s_5)}{s_3(s_1^3+s_3)+s_1(s_1^5+s_5)}$, $\sigma_3 = (s_1^3 + s_3) + s_1 + \sigma_2$
- Falls $s_1 \neq 0$, berechne $\sigma_4 = \frac{(s_5+s_1^2 s_3)+(s_1^3+s_3)\sigma_2}{s_1}$ sonst berechne $\sigma_4 = \frac{s_7 s_3+s_5^2}{s_3^2}$
- Berechne ein $\lambda = pp'$ mit $\lambda^3 + (\sigma_2^2 + \sigma_1\sigma_3)\lambda + (\sigma_3^2 + \sigma_1^2\sigma_4 + \sigma_1\sigma_2\sigma_3) = 0$
- Berechne die Lösungen $p = p_1, p' = p_2$ von $p_{(1,2)}^2 + \sigma_1 p_{(1,2)} + pp' = 0$
- Berechne die Lösungen q_1, q_2 von $q_{(1,2)}^2 + (pp' + \sigma_2)q_{(1,2)} + \sigma_4 = 0$
- Wenn $\sigma_2 = pq_2 + p'q_1$ gilt dann ist $q = q_1, q' = q_2$, sonst gilt $q = q_2, q' = q_1$.
- Berechne die Lösungen i, j für $x^2 + px + q = 0$
- Berechne die Lösungen k, l für $x^2 + p'x + q' = 0$

Die Lösungen i, j, k, l entsprechen den Lösungen des Lokatorpolynoms vierten Grades. Hier werden als Teilprobleme ebenfalls 1 Gleichung dritten Grades und 4 Gleichungen zweiten Grades gelöst.

Alternative Herangehensweise In diesem Abschnitt stellen wir eine **eigene** Erweiterung der bekannten Verfahren vor, welche die Anzahl der zu berechnenden quadratischen Gleichungen um eins reduziert. Dieses Verfahren wurde in [SH22] veröffentlicht. Statt q, q' und p, p' jeweils über eine Gleichung zweiten Grades zu berechnen, kann nach folgendem Ansatz umgeformt werden, so dass nur eine Gleichung zweiten Grades zur Bestimmung von q, q', p, p' zu lösen ist.

$$\begin{aligned}
\sigma_2 &= pp' + q + q' \\
q' &= \sigma_2 + pp' + q \\
\sigma_3 &= pq' + p'q \\
\sigma_3 &= p(\sigma_2 + pp' + q) + p'q \\
\sigma_3 &= p(\sigma_2 + pp') + (p' + p)q \\
\sigma_3 &= p(\sigma_2 + pp') + \sigma_1 q \\
q &= \frac{p(\sigma_2 + pp') + \sigma_3}{\sigma_1} \\
q &= (\sigma_2 + pp') + q' \\
\sigma_3 &= pq' + p'q \\
\sigma_3 &= pq' + p'(\sigma_2 + pp' + q') \\
\sigma_3 &= (p + p')q' + p'(\sigma_2 + pp') \\
\sigma_3 &= \sigma_1 q' + p'(\sigma_2 + pp') \\
q' &= \frac{p'(\sigma_2 + pp') + \sigma_3}{\sigma_1}
\end{aligned} \tag{2.104}$$

Entweder werden q, q' parallel berechnet, oder q' kann durch $q' = q + (\sigma_2 + pp')$ aus q berechnet werden. Es gibt damit folgende Möglichkeiten q, q' aus p, p' zu berechnen:

1.

$$\begin{aligned}
q &= \frac{p(\sigma_2 + pp') + \sigma_3}{\sigma_1} \\
q' &= \frac{p'(\sigma_2 + pp') + \sigma_3}{\sigma_1}
\end{aligned} \tag{2.106}$$

2.

$$\begin{aligned}
q &= \frac{p(\sigma_2 + pp') + \sigma_3}{\sigma_1} \\
q' &= \sigma_2 + pp' + q
\end{aligned} \tag{2.108}$$

3.

$$\begin{aligned}
q' &= \frac{p'(\sigma_2 + pp') + \sigma_3}{\sigma_1} \\
q &= (\sigma_2 + pp') + q'
\end{aligned} \tag{2.110}$$

Dies erspart das Lösen einer der quadratischen Gleichungen und die anschließende Zuordnung von $p_{(1,2)}$ zu $q_{(1,2)}$. Diese Gleichung ist nur verwendbar, falls $\sigma_1 \neq 0$ gilt. Dies entspricht der Fallunterscheidung, welche im Verfahren zur Koeffizientenberechnung nach Wicker vorgeschlagen wird.

Für den Sonderfall $\sigma_1 = 0$ ergibt sich

$$\begin{aligned} \sigma_1 = s_1 = p + p' &= 0 \\ p &= p' \\ pp' &= pp' \\ p &= \sqrt{pp'} \end{aligned} \tag{2.112}$$

Somit sind p, p' direkt aus pp' ableitbar. Die Werte für q und q' müssen nun über die quadratische Gleichung bestimmt werden.

$$q_{(1,2)}^2 + (pp' + \sigma_2)q_{(1,2)} + \sigma_4 = 0 \tag{2.114}$$

Dabei kann frei gewählt werden, ob $q = q_1, q' = q_2$ oder ob $q = q_2, q' = q_1$ als Zuordnung gewählt wird. Im Folgenden wird die Veränderung in beiden zuvor beschriebenen Verfahren (2.3.2.4, 2.3.2.4) angewendet und der veränderte Ablauf beschrieben.

Vereinfachter Ablauf der Bestimmung der Fehlerstellen Als Modifikation des bekannten Ansatzes stellen wir somit folgenden Ablauf vor. Mit diesem Vorgehen wird eine quadratische Gleichung bei der Berechnung der Fehlerstellen eingespart.

- Berechne die Syndromkomponenten s_1, s_3, s_5, s_7
- Berechne $A = s_3(s_1^3 + s_3) + s_1(s_1^5 + s_5)$, $\sigma_1 = s_1$,
  $\sigma_2 = (s_1(s_7 + s_1^7) + s_3(s_1^5 + s_5))/A$,
  $\sigma_3 = (s_1(s_1^3 s_5 + s_1 s_7) + s_3(s_1^6 + s_3^2))/A$,
  $\sigma_4 = (s_1^3(s_1^7 + s_7) + s_3(s_1^7 + s_1 s_3^2 + s_7) + s_5(s_1^5 + s_1^2 s_3 + s_5))/A$
- Berechne ein $\lambda = pp'$ mit $\lambda^3 + (\sigma_2^2 + \sigma_1\sigma_3)\lambda + (\sigma_3^2 + \sigma_1^2\sigma_4 + \sigma_1\sigma_2\sigma_3) = 0$
- Falls $s_1 \neq 0$:
  - Berechne die Lösungen $p = p_1, p' = p_2$ von $p_{(1,2)}^2 + \sigma_1 p_{(1,2)} + pp' = 0$
  - Berechne parallel $q = \frac{p(\sigma_2 + pp') + \sigma_3}{\sigma_1}$ und $q' = \frac{p'(\sigma_2 + pp') + \sigma_3}{\sigma_1}$
- Sonst
  - $p = p' = \sqrt{pp'}$
  - Berechne die Lösungen $q = q_1, q' = q_2$ von $q_{(1,2)}^2 + (pp' + \sigma_2)q_{(1,2)} + \sigma_4 = 0$
- Berechne die Lösungen i, j für $x^2 + px + q = 0$
- Berechne die Lösungen k, l für $x^2 + p'x + q' = 0$

Zweiter vereinfachter Ablauf der Bestimmung der Fehlerstellen

- Berechne die Syndromkomponenten s_1, s_3, s_5, s_7
- Berechne $\sigma_1 = s_1$, $\sigma_2 = \frac{s_1(s_7+s_1^7)+s_3(s_1^5+s_5)}{s_3(s_1^3+s_3)+s_1(s_1^5+s_5)}$, $\sigma_3 = (s_1^3+s_3)+s_1+\sigma_2$
- Falls $s_1 \neq 0$, berechne $\sigma_4 = \frac{(s_5+s_1^2 s_3)+(s_1^3+s_3)\sigma_2}{s_1}$ sonst berechne $\sigma_4 = \frac{s_7 s_3+s_5^2}{s_3^2}$
- Berechne ein $\lambda = pp'$ mit $\lambda^3 + (\sigma_2^2+\sigma_1\sigma_3)\lambda + (\sigma_3^2+\sigma_1^2\sigma_4+\sigma_1\sigma_2\sigma_3) = 0$
- Falls $s_1 \neq 0$:
  - Berechne die Lösungen $p = p_1, p' = p_2$ von $p_{(1,2)}^2 + \sigma_1 p_{(1,2)} + pp' = 0$
  - Berechne parallel $q = \frac{p(\sigma_2+pp')+\sigma_3}{\sigma_1}$ und $q' = \frac{p'(\sigma_2+pp')+\sigma_3}{\sigma_1}$
- Sonst
  - $p = p' = \sqrt{pp'}$
  - Berechne die Lösungen $q = q_1, q' = q_2$ von $q_{(1,2)}^2 + (pp'+\sigma_2)q_{(1,2)} + \sigma_4 = 0$
- Berechne die Lösungen i, j für $x^2+px+q=0$
- Berechne die Lösungen k, l für $x^2+p'x+q'=0$

2.3.3 Schaltung 4-Bit Korrektur

Folgende Schaltungen entstammen der Arbeit von Okano Imai [OI87]. In dieser wurden Schaltungen für die Berechnung der 4-Bit Fehlerpositionen und der benötigten Komponenten erstellt. Zusätzlich wird aus eigener Arbeit eine Vereinfachung der Schaltung dargestellt, welche eine Gleichung zweiten Grades einspart.

Werte werden hier in der Form der Exponentialdarstellung verwendet, jeder Wert x wird als Exponent i der Basis α dargestellt, wobei $\alpha^i \mod (2^m-1) = x$. Ist $x = 0$ ein Nullelement, existiert kein solches i, in der Hardwareschaltung wird dieses Nullelement durch den nicht verwendeten Vektor $i = (111...111)$ dargestellt. Dieser Vektor wird nicht durch andere Exponenten benötigt, da durch die Berechnung $\alpha^i \mod (2^m-1)$ im Galoisfeld $GF(2^m)$ die Belegung $(111...111) \mod (2^m-1) = (2^m-1) \mod (2^m-1) = 0$ außerhalb des Modulo liegt und somit durch den Wert $i = 0$ abgedeckt werden kann. Der Vektor $(111...111)$ wird daher verwendet, um das Nullelement darzustellen. Diese Verwendung erfordert Sonderbetrachtungen in den Additions- bzw. Multiplikationsoperationen.

Eine Alternative wäre hier, einen weiteren binären Kanal mitzuführen, welcher $x = 0$ darstellt.

Multiplikation und Division können in der Exponentialdarstellung durch Additionsoperatoren dargestellt werden und sind daher effizient. Additionen erfordern eine Umwandlung der Werte durch eine Tabelle und sind daher aufwändiger.

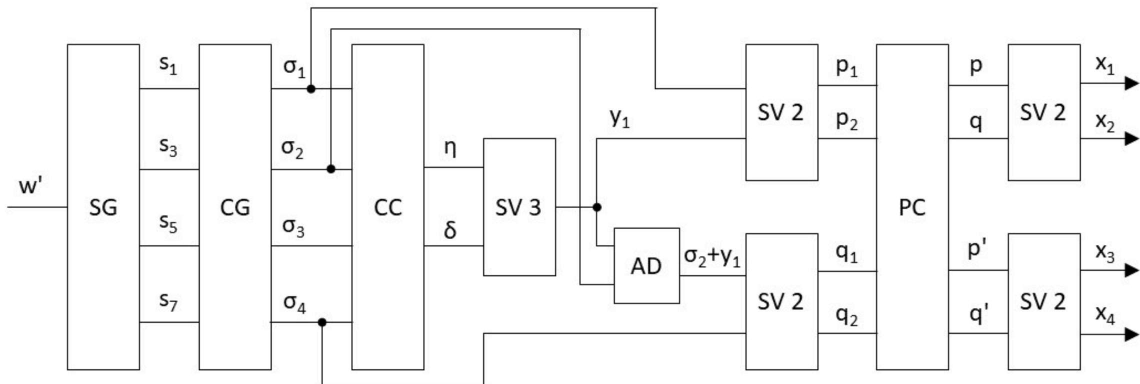

Abbildung 2.14: Nach Okano-Imai: Schaltung zur Bestimmung der Fehlerstellen eines 4-Bit Fehlers

In der Abbildung 2.14, der Schaltung zur Lösung eines 4-Bit Fehlers, werden die folgenden Elemente verwendet.

- **SG** Schaltung zur Bestimmung der Syndromkomponenten s_1, s_3, s_5, s_7.
- **CG** Schaltung zur Berechnung der Koeffizienten $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ im 4-Bit Fall.
- **CC** Schaltung zur Berechnung der Werte $\eta = \sigma_2^2 + \sigma_1\sigma_3$ und $\delta = \sigma_3^2 + \sigma_1^2\sigma_4 + \sigma_1\sigma_2\sigma_3$, der Koeffizienen der Hilfsgleichung dritten Grades.
- **SV 3** Schaltung zur Lösung der Hilfsgleichung dritten Grades Es wird lediglich die Lösung Y_1 verwendet.
- **AD** Addierer im Galoisfeld, siehe Abbildung 2.17.
- **SV 2** Schaltung Zur Lösung einer Gleichung zweiten Grades.
- **PC** Schaltung zur Bestimmung eines korrekten Paars p, q und p', q'.

Die Schaltung zur Lösung des 4-Bit Fehlers geht nach dem zuvor erklärten Prinzip vor. Zunächst werden durch **SG** die Syndromkomponenten s_1, s_3, s_5, s_7 aus dem übertragenen Wort w' berechnet. Im Anschluss werden die Koeffizienten $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ durch **CG** aus diesen berechnet. Aus diesen wird die Hilfsgleichung dritten Grades erzeugt **CC** und gelöst **SV 3**. Im Anschluss werden die Werte p_1, p_2, q_1, q_2 durch Gleichungen 2ten Grades durch **SV 2** Schaltungen berechnet. Aus diesen werden in **PC** p, q, p', q' bestimmt und durch Gleichungen 2ten Grades **SV 2** die Lösungen $x1, x_2, x_3, x_4$ bestimmt.

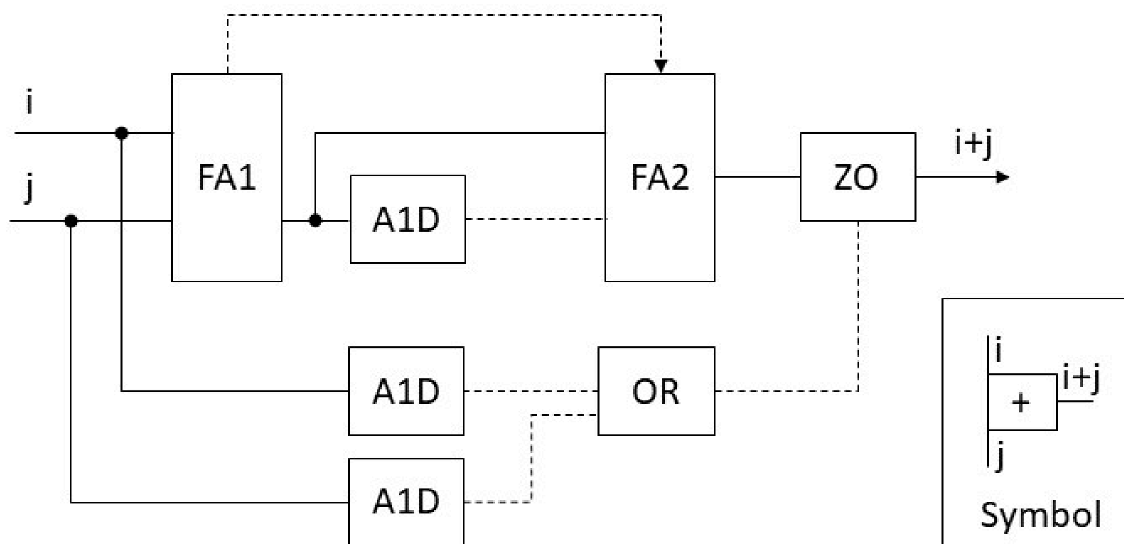

Abbildung 2.15: Nach Okano-Imai: **+** Multiplizierer im Galoisfeld. Durchgezogene Kanäle haben eine Breite von m Bit, gestrichelte Kanäle eine Breite von einem Bit

Der Multiplizierer im Galoisfeld, Abbildung 2.15, verwendet die folgenden Elemente.

- **FA1**, **FA2** Volladdierer mit Übertrag. Der zweite Volladdierer **FA2** erhält durch **A1D** ein einzelnes Bit als zweiten zu addierenden Wert (dieses wird an niedrigster Stelle verwendet) und den Übertrag des **FA1**.

- **A1D** Prüfelement auf Nullelement. Die Schaltung besitzt m Eingänge und einen Ausgang. Die Schaltung entspricht einem *AND*-Baum über die Eingänge. Der Ausgang ist 1 genau dann, wenn der Eingangsvektor $'11..11'$ ist.

- **OR** Ein OR Gatter.

- **ZO** Leitet den m-Bit Eingang weiter, falls der ein Bit Eingang 0 ist. Ansonsten wird der Nullvektor ausgegeben.

Eine Multiplikation wird durch Additionen dargestellt, die Elemente **FA1**, **FA2** stellen Volladdierer dar. **A1D** prüft ob ein Wert 0 ist (also mit dem Wert $'11...11'$ belegt ist). Die Schaltung benutzt somit den ersten Volladdierer, um i und j aufzuaddieren und den zweiten, um die $\mod 2^m - 1$ Operation zu implementieren. Durch diese müssen für Ergebnisse ab $2^m - 1$ der Wert $2^m - 1$ abgezogen werden, oder wie einfacher möglich, das Komplement, der Wert 1 ohne Überlauf aufaddiert werden, um einen Korrekten Wert zu erhalten. Dies ist in einem von zwei Fällen nötig. Entweder, wenn es einen Überlauf gab oder wenn der Wert genau der Vektor $'11...11'$ ist. Da beides nicht gleichzeitig auftreten kann, wird ein zweiter Volladdierer verwendet, in den der Überlauf des ersten Volladdierers und falls $i + j =' 11...11'$ ist $'00...001'$ aufaddiert. Ist einer der Werte von i oder j ein Nullelement, so wird durch das Element **ZO** am Ende ein Nullelement ausgegeben.

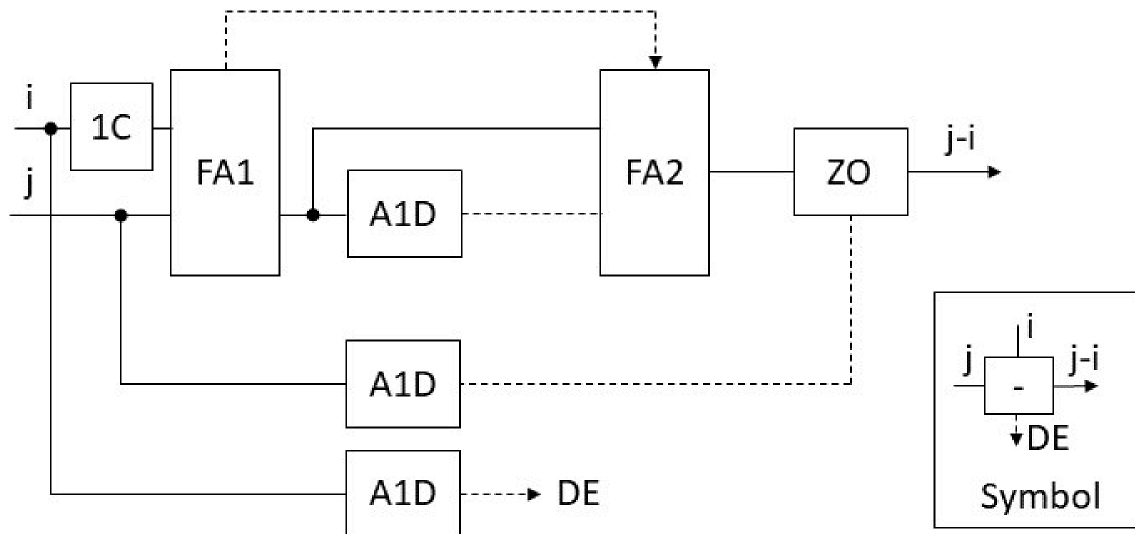

Abbildung 2.16: Nach Okano-Imai: **-** Dividierer im Galoisfeld.

Der Dividierer im Galoisfeld, Abbildung 2.16, verwendet zusätzlich zu denen des Multiplizierers folgendes Element.

- **1C** Invertierer im Einserkomplement. Aufgrund der Modulo Berechnung der Exponenten entspricht dies dem Inversen bezüglich der Multiplikation.

Der Dividierer berechnet $\log_\alpha(\frac{\alpha^j}{\alpha^i}) = j - i$ und wird ähnlich des Multiplizierers konzipiert, wobei der Eingang i durch **1C** invertiert wird. Dies sorgt für die Berechnung von $j-i$. Allerdings wird im Fall, dass i ein Nullelement ist, über DE ein Fehler ausgegeben.

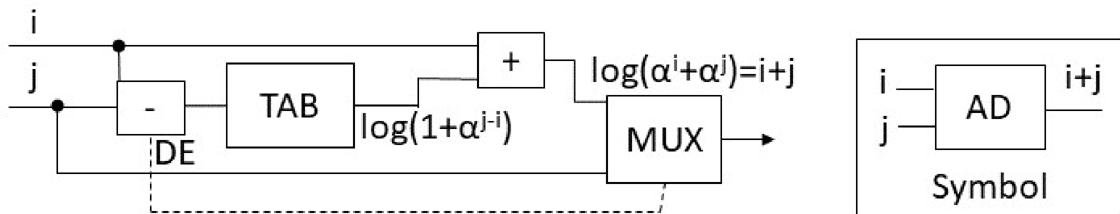

Abbildung 2.17: Nach Okano-Imai: **AD** Addierer im Galoisfeld.

Der Addierer im Galoisfeld, Abbildung 2.17 verwendet folgende Elemente.

- **+** Multiplizierer im Galoisfeld, siehe Abbildung 2.15.
- **-** Dividierer im Galoisfeld, siehe Abbildung 2.16.
- **TAB** Tabelle des Zechschen Logarithmus. Zu jeder Eingabe x wird $\log_\alpha(1+\alpha^x)$ ausgegeben. Als kombinatorische Schaltung implementierbar.
- **MUX** Multiplexer zwischen zwei Eingängen, ist der Eingang mit einem Bit 0, wird der obere Kanal ausgegeben, ansonsten der untere. Hier gilt: Ist $DE = 0$, so wird $i + j$ ausgegeben, ansonsten j.

Der Addierer muss nun die Exponentialdarstellung in eine Addition in Vektordarstellung überführen. Hier wird der Zechsche Logarithmus[Hub90] verwendet, um die Berechnung zu vereinfachen. Zunächst wird $j-i$ berechnet, dieser wird über eine Tabelle **TAB** zu $\log(1+\alpha^{j-i})$ überführt, also dem Nachfolger von $j-i$ entsprechend der Vektordarstellung. Wird dieser Wert mit i multipliziert, ergibt sich

$$\begin{aligned} &\alpha^i \times \alpha^{\log(1+\alpha^{j-i})} \\ &= \alpha^i \times (1+\alpha^{j-i}) \\ &= \alpha^i + \alpha^{j-i+i} \\ &= \alpha^i + \alpha^j \end{aligned} \tag{2.116}$$

Es ergibt sich, dass dies der Addition entspricht. Falls i ein Nullelement ist, wird ein Multiplexer **MUX** verwendet, um j auszugeben. Im Fall, dass j ein Nullelement ist, ergibt die Tabelle den Exponenten zum Wert 1, somit wird insgesamt i ausgegeben.

Eine Alternative wäre die Umrechnung beider Inputs i und j zur Vektordarstellung durch jeweils eine Tabelle, dann eine Addition $\mod 2^m-1$ und anschließend eine Rückumwandelung in die Exponentialdarstellung durch eine weitere Tabelle. Hier stellt sich die Frage, welche Variante in Hardware schneller ist und welche günstiger in der Herstellung ist.

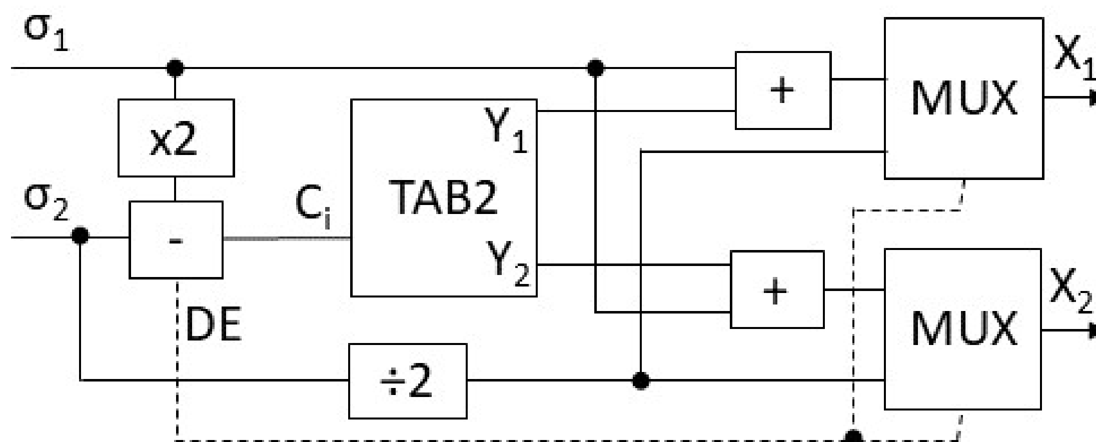

Abbildung 2.18: Nach Okano-Imai: **SV 2** Schaltung zur Lösung einer Gleichung zweiten Grades.

Die Schaltung zur Lösung von Gleichungen zweiten Grades, Abbildung 2.18 verwendet die folgenden Elemente.

- **-** Dividierer im Galoisfeld, siehe Abbildung 2.16.
- **+** Multiplizierer im Galoisfeld, siehe Abbildung 2.15.
- **x2** Quadrierer im Galoisfeld. Für Exponenten entspricht dies dem Verdoppeln (modulo beachten) Als shiften (Positionen der einzelnen Bits werden eine Stelle höher geleitet) und modulo (unter Berücksichtigung des Übertrages) implementierbar.

- **÷2** Wurzel im Galoisfeld. Für Exponenten entspricht dies dem Halbieren (modulo beachten). Kann als niedriger shiften und modulo (unter Berücksichtigung des Übertrages) implementiert werden.
- **TAB2** C Tabelle für 2-Bit Korrektur mit zwei Ausgaben. Da $Y_2 = Y_1$ XOR $'00.001'$ gilt, müssen nicht beide Werte in der Tabelle gespeichert werden, es ist ausreichend, die Ausgabe zu kopieren und die letzte Bitposition zu invertieren.
- **MUX** Multiplexer zwischen zwei Eingängen, ist der Eingang mit einem Bit 0, wird der obere Kanal ausgegeben, ansonsten der untere.

Diese Schaltung erwartet die Koeffizienten des entsprechenden Polynoms zweiten Grades als Eingabe. Während dieses in der Korrektur von 2-Bit Fehlern verwendet werden kann, haben wir dort ein anderes Verfahren dargestellt. Dies liegt daran, dass im Falle einer 2-Bit Fehlerkorrektur die Syndromkomponenten s_1, s_3 zur Verfügung stehen und eine Berechnung der Koeffizienten nicht notwendig ist, da C aus den Syndromkomponenten effizient berechnet werden kann. Diese Schaltung hier ist jedoch trotzdem relevant, da zur Lösung des Lokatorpolynoms vierten Grades Hilfsgleichungen zweiten Grades gelöst werden müssen, bei welchen lediglich die Koeffizienten der quadratischen Gleichung vorhanden sind. Es wird gerechnet $C = \sigma_2 \div \sigma_1^2$ und aus diesem werden per Tabelle die Hilfswerte Y_1 und Y_2 ausgegeben, welche per Multiplikation mit σ_1 (entspricht s_1) in die Fehlerstellen umgewandelt werden. Ist eine Division nicht möglich (σ_1 ist Nullelement), so sind beide Lösungen $\sqrt{\sigma_2}$.

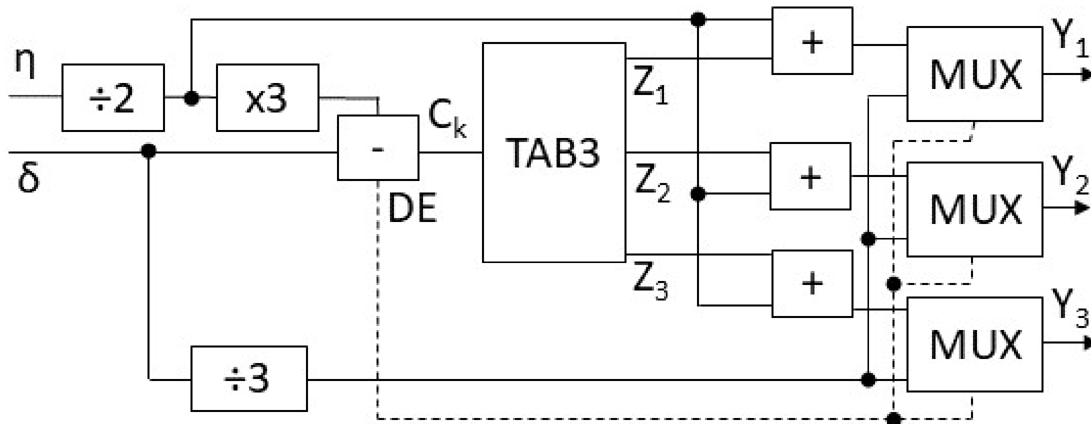

Abbildung 2.19: Nach Okano-Imai: **SV 3** Schaltung zur Lösung der Hilfsgleichung dritten Grades.

Die Schaltung zur Lösung der Hilfsgleichung dritten Grades, Abbildung 2.19 verwendet die folgenden Elemente.

- **-** Dividierer im Galoisfeld, siehe Abbildung 2.16.
- **+** Multiplizierer im Galoisfeld, siehe Abbildung 2.15.
- **x3** Kubischer Exponenzierer (hoch drei) im Galoisfeld. Für Exponenten entspricht dies dem Verdreifachen (modulo beachten). als shiften und addieren (modulo beachten) implementierbar.

- **÷2** Wurzel im Galoisfeld. Für Exponenten entspricht dies dem Halbieren (modulo beachten). Kann als niedriger shiften und modulo (unter Berücksichtigung des Übertrages) implementiert werden.

- **÷3** Dritte Wurzel im Galoisfeld. Für Exponenten entspricht dies dem Dritteln (modulo beachten). Eine Tabelle kann verwendet werden, um eine Division im Galoisfeld zu vermeiden.

- **TAB3** C Tabelle für 3-Bit Korrektur mit drei Ausgaben.

- **MUX** Multiplexer zwischen zwei Eingängen, ist der Eingang mit einem Bit 0, wird der obere Kanal ausgegeben, ansonsten der untere.

Diese Schaltung erhält die Werte η und δ der Hilfsgleichung und berechnet die Ergebnisse dieser. Im Allgemeinen Fall kann eine Gleichung dritten Grades drei Koeffizienten besitzen, diese muss umgeformt werden, bevor dieses Verfahren verwendet werden kann. Die Vorgehensweise ist hier $C_k = \frac{\delta}{\eta^{3\div 2}}$ zu berechnen und als Eingabe der Tabelle **TAB3** zu verwenden, um die drei verallgemeinerten Lösungen Z_1, Z_2, Z_3 zu erhalten. Diese werden mit $\sqrt{\eta}$ multipliziert, um die Fehlerpositionen Y_1, Y_2, Y_3 zu erhalten. Ist $\eta^{3\div 2}$ das Nullelement, kann die Division nicht durchgeführt werden, in dem Fall wird $Y_1 = Y_2 = Y_3 = \sqrt[3]{\delta}$ ausgegeben.

Beschreibung der vereinfachten Schaltung Wird das in dieser Arbeit entwickelte, alternative Vorgehen (Paragraph 2.3.2.4) zur Berechnung der 4-Bit Korrektur verwendet, so ändert sich die Korrekturschaltung wie folgt:

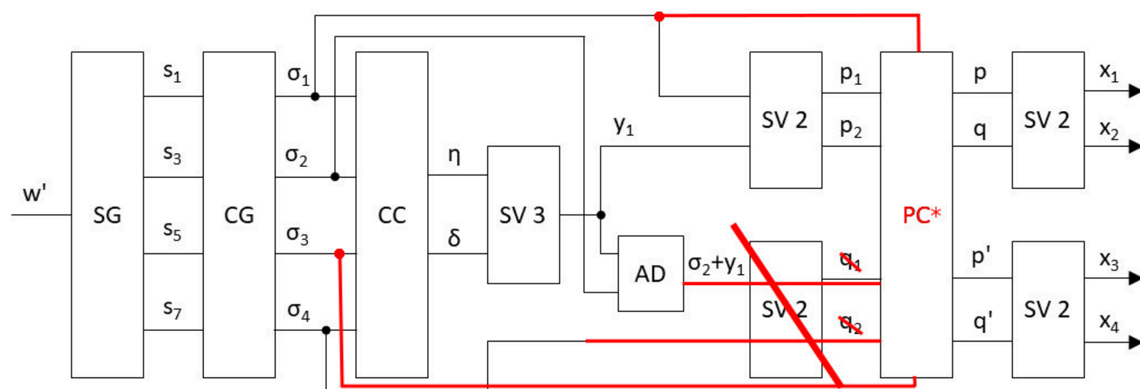

Abbildung 2.20: 4-Bit Korrektur entsprechend eigener Änderungen.

Es entfällt eine Schaltung **SV 2**, welche q_0, q_1 aus $q_{(0,1)}^2 + (pp' + \sigma_2)q_{(0,1)} + \sigma_4 = 0$ berechnet. Hier wird eine Komponente $SV2$ gespart, die Logik von **PC*** bestimmt somit q, q' direkt. Dazu wird die Formel $q = \frac{p_1(\sigma_2+y_1)+\sigma_3}{\sigma_1}, q' = \frac{p_2(\sigma_2+y_1)+\sigma_3}{\sigma_1}$ genutzt, welche im Falle $s_1 = 0$ jedoch nicht verwendbar ist. Da $s_1 = 0$ bei 4-Bit Fehlern auftreten kann, wird somit eine Fallunterscheidung durchgeführt (nicht in der Abbildung 2.20 abgebildet). Eine vollständige Schaltung kann somit wie folgt aussehen:

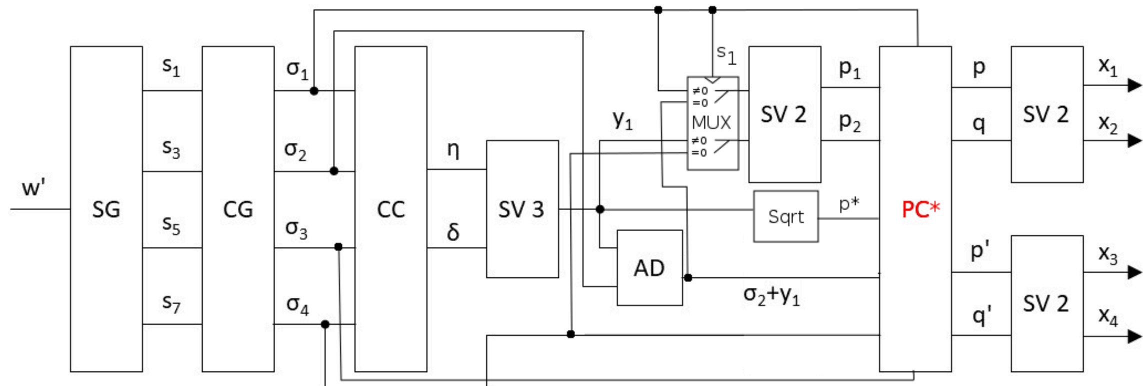

Abbildung 2.21: 4-Bit Korrektur entsprechend eigener Änderungen mit Fallunterscheidung.

Die Änderung der Schaltung findet somit durch die Komponenten **MUX**, **Sqrt** und **PC*** statt:

- **MUX** erhält $s_1, \sigma_1, \sigma_2 + y_1, y_1, \sigma_4$ als Eingabe und schaltet die Ausgänge (für **SV 2**) σ_1 und y_1, falls $s_1 \neq 0$; $\sigma_2 + y_1, \sigma_4$ falls $s_1 = 0$.
- **Sqrt** Berechnet die Wurzel von y_1 per Tabelle. Im Fall $s_1 = 0$ entspricht diese $\sqrt{y_1} = p = p'$. Die Ausgabe nennen wir $p*$
- **PC*** bestimmt $q = \frac{p_1(\sigma_2+y_1)+\sigma_3}{\sigma_1}, q' = \frac{p_2(\sigma_2+y_1)+\sigma_3}{\sigma_1}, p = p_1, p' = p_2$, falls $s_1 \neq 0$. Ansonsten setzt **PC*** $p = p*, p' = p*, q = p_1, q' = p_2$.

Das Konzept im Falle $s_1 = 0$ ist, dass die Schaltung **SV 2** zur Bestimmung von p_1, p_2 umfunktioniert wird, so dass diese nun q_1, q_2 ausgibt. Da $p = p'$ gilt, ist irrelevant welche von q_1, q_2 den Werten q, q' zugewiesen werden.

2.3.4 Korrektur

Im Anschluss an die Erkennung der Fehlerpositionen werden diese korrigiert. Da hier mit Binärvektoren gearbeitet werden, genügt es, die entsprechenden Bitpositionen zu invertieren.

2.3.5 Zum Dekodierungsverfahren nach Berlekamp/Massey

Der State-of-the-Art Algorithmus zur Dekodierung des BCH Codes wurde durch Berlekamp[Ber68] erfunden. Eine effiziente Implementierung mit Shiftregistern wurde durch Massey[Mas69] entwickelt. Das Verfahren von Berlekamp kann als iteratives Näherungsverfahren beschrieben werden, welches nach einer bestimmten Anzahl von Iterationen ein präzises Lokatorpolynom erzeugt. Dieser ermöglicht es somit, große Fehleranzahlen iterativ zu bestimmen, ohne Determinanten zu berechnen. Im Anschluss müssen aus dem Lokatorpolynom die Positionen der Fehler bestimmt werden. Hierzu wird in der Regel eine Chien Search verwendet, welche in diesem Abschnitt erklärt wird.

Die Anzahl der Iterationen des Berlekamp-Massey-Algorithmus hängt von der Anzahl der aufgetretenen Fehler ab und wird bis zur maximal korrigierbaren Fehleranzahl fortgesetzt. Nach der höchsten erkannten Fehleranzahl ändert sich das Lokatorpolynom in Folgeschritten nicht mehr.

Das Verfahren ähnelt dem Verfahren mit der Determinantenberechnung, beide iterieren über alle korrigierbaren Fehleranzahlen und bestimmen nicht direkt die Positionen der Fehler, sondern ein Zwischenergebnis, welches weiter verarbeitet werden muss.

Zunächst stellen wir den Algorithmus dar, eine Erklärung der verwendeten Ausdrücke und eine detaillierte Berechnung anhand von Beispielen erfolgt im Anschluss.

Die Kurzfassung des Algorithmus ist wie folgt.

1. Setze $k = 0, \quad \Lambda = 1, \quad T = 1, \quad \Delta = S_1$
2. Wiederhole während $k < t$:
   a) Setze $\Lambda' = \Lambda, \quad \Delta' = \Delta, \quad T' = T, \quad k = k+1$
   b) Setze $\Lambda = \Lambda' + x \cdot \Delta' \cdot T'$
   c) Falls $\Delta' \neq 0$ und $deg(\Lambda') \leq k$: Setze $T = x \cdot \frac{\Lambda'}{\Delta'}$
      Sonst setze $T = x^2 \cdot T'$
   d) Setze Δ auf den Wert des Koeffizienten von x^{2k+1} in $\Lambda(1+S(x))$
3. Interpretiere Λ als Lokatorpolynom und berechne die Fehlerstellen.

Wobei

- t die Anzahl der Syndromkomponenten ist ($(2t)$-Bit Fehler sind erkennbar bzw. Fehler bis t-Bit korrigierbar)
- k eine Zählervariable ist, welche von 1 bis t läuft.
- Λ ein Polynom in Abhängigkeit von x ist, welches iterativ zum Lokatorpolynom entwickelt wird.
- Δ jeweils ein Koeffizient aus dem Produkt von Λ und $(1+S(x))$ ist, welcher als nächster Wert für die Entwicklung von Λ verwendet wird. Δ ist 0, falls es keinen zugehörigen Koeffizienten gibt.
- T ist im Regelfall das x-fache des vorangehenden Λ Wertes dividiert durch den vorangehenden Δ Wert. Dieser bildet multipliziert mit x und dem neuen Wert von Δ die Änderung des Λ in jedem Schritt. Vereinfacht kann man die Entwicklung von Λ daher wie folgt darstellen.

  $$\Lambda = \Lambda' + x^2 \times \Lambda'' \times \frac{\Delta'}{\Delta''}$$

  Ist $\Delta'' = 0$ oder der Grad von Λ'' größer $k-1$ ($\deg(\Lambda'') > k-1$), dann

  $$\Lambda = \Lambda' + x^4 \times \Lambda''' \times \frac{\Delta'}{\Delta'''}$$

  Sollte Δ''' hier wieder 0 sein oder der Grad von Λ''' größer als $k-2$, so kann dieser Vorgang fortgesetzt werden.

- $S(x)$, aus welchem Δ gebildet wird, ist ein Polynom der folgenden Form

$$S(x) = S_1x + S_2x^2 + S_3x^3 + ... + S_{2t}x^{2t} + S_{2t+1}x^{2t+1} + ...$$

dieses Polynom hat unendlichen Grad, jedoch sind nur die Koeffizienten zu x bis höchstens x^{2t} relevant.

Chien search

Um aus einem Lokatorpolynom die einzelnen Fehlerstellen zu berechnen, kann das Verfahren Chien Search[Chi64] verwendet werden.

Verallgemeinert setzt die Chien Search iterativ jede mögliche Fehlerposition in das Lokatorpolynom ein und prüft auf eine Nullstelle.

$$\begin{aligned} p(\alpha^0) &\stackrel{?}{=} 0 \\ p(\alpha^1) &\stackrel{?}{=} 0 \\ p(\alpha^2) &\stackrel{?}{=} 0 \\ &\vdots \\ p(\alpha^{n-1}) &\stackrel{?}{=} 0 \end{aligned} \tag{2.118}$$

Genauer besteht ein entscheidender Punkt darin, dass jeder Schritt die Summanden des Vorgängerschritt verwendet. Betrachten wir den Schritt von Position i zu Position $i+1$ bei einem Lokatorpolynom vom Grad k.

$$\begin{aligned} p(\alpha^i) &= (\alpha^i)^k + \sigma_1(\alpha^i)^{k-1} + \sigma_2(\alpha^i)^{k-2} + ... + \sigma_{k-1}\alpha^i + \sigma_k \\ p(\alpha^{i+1}) &= (\alpha^{i+1})^k + \sigma_1(\alpha^{i+1})^{k-1} + \sigma_2(\alpha^{i+1})^{k-2} + ... + \sigma_{k-1}\alpha^{i+1} + \sigma_k \\ p(\alpha^{i+1}) &= (\alpha^i)^k \times \alpha^k + \sigma_1(\alpha^i)^{k-1} \times \alpha^{k-1} + \sigma_2(\alpha^i)^{k-2} \times \alpha^{k-2} + ... + \sigma_{k-1}\alpha^i \times \alpha + \sigma_k \end{aligned} \tag{2.120}$$

Hier können wir sehen, dass durch einen Schritt jeder Summand mit einem Wert α^j multipliziert wird, wobei $j = k -$ **Position des Summanden** ist. Die Summanden $Q_k, ..., Q_1$ werden mit jedem Schritt wie folgt multipliziert:

$$Q_{i'} = Q_i\alpha^i \tag{2.122}$$

Für alle $i \in (k, ..., 1)$ und $Q_{i'}$ der Folgewert des Summanden ist. Der Chien Search Algorithmus funktioniert somit wie folgt:

1. Berechne die Initialwerte $Q_k = 1$, $Q_{k-1} = \sigma_1$, $Q_{k-2} = \sigma_2$, ..., $Q_1 = \sigma_k$.
2. Wiederhole für $j = 0..n$
   a) Prüfe, ob $\sum_{i=1..k}(Q_i) + \sigma_k \stackrel{?}{=} 0$
   b) Falls, gebe j als Fehlerposition aus und setze fort
   c) Berechne parallel $Q_i = Q_i \times \alpha^i$ für $i \in (k, ..., 1)$

Der Chien Search Algorithmus berechnet mit diesem Vorgehen iterativ die Fehlerstellen, wobei Teilergebnisse jedes Schritts weiter verwendet werden. Werden Worte übertragen, können somit die Positionen in Reihenfolge der Übertragung korrigiert werden.

2.3.5.1 Funktionsweise Berlekamp-Massey Algorithmus

Aus dem Buch von Wicker[Wic95] geht folgende Erklärung hervor. Ist bei einem BCH Code ein v-Bit Fehler aufgetreten und die Positionen der Fehler sind $i_1, ..., i_v$, so setzen sich die Syndromkomponenten wie folgt zusammen.

$$S_j = \sum_{l=0}^{v} (\alpha^{i_l})^j = \sum_{l=0}^{v} (X_l)^j, \qquad j = 1, ..., 2t$$

$X_1, ..., X_v$ sind somit die α Werte an den entsprechenden Stellen der Fehler. Betrachten wir das Polynom

$$\lambda(x) = \prod_{l=1}^{v} (1 - X_l x) = \lambda_v x^v + \lambda_{v-1} x^{v-1} + ... + \lambda_1 x + \lambda_0$$

sind hier die Nullstellen die Inversen der Werte $X_1, ..., X_v$, da es im Produkt für jedes $x = \frac{1}{X_p}$ einen Faktor gibt, welcher $(1 - X_p \times \frac{1}{X_p}) = (1 - 1) = 0$ ergibt. Die Koeffizienten von $\lambda(x)$ sind dabei:

$$\begin{aligned}
\lambda_0 &= 1 \\
\lambda_1 &= \sum_{i=1}^{v} X_i = X_1 + X_2 + ... + X_{v-1} + X_v \\
\lambda_2 &= \sum_{i<j}^{v} X_i X_j = X_1 X_2 + X_1 X_3 + ... + X_1 X_v + X_2 X_3 + X_2 X_4 + ... + X_{v-2} X_{v-1} + X_{v-2} X_v + X_{v-1} X_v \\
\lambda_3 &= \sum_{i<j<k}^{v} X_i X_j X_k = X_1 X_2 X_3 + X_1 X_2 X_4 + ... + X_{v-3} X_{v-2} X_v + X_{v-3} X_{v-1} X_v + X_{v-2} X_{v-1} X_v \\
\lambda_4 &= \sum_{i<j<k<l}^{v} X_i X_j X_k X_l = X_1 X_2 X_3 X_4 + X_1 X_2 X_3 X_5 + ... + X_{v-4} X_{v-2} X_{v-1} X_v + X_{v-3} X_{v-2} X_{v-1} X_v \\
&... \\
\lambda_0 &= \sum_{i=1}^{v} X_i = X_1 \cdot X_2 \cdot X_3 \cdot ... \cdot X_{x-2} \cdot X_{v-1} \cdot X_v
\end{aligned} \tag{2.124}$$

Dies ergibt sich dadurch, dass das Produkt aller $(1 - X_l x)$ bei einem Koeffizienten zu x^i aus i verschiedenen Produkten aus $X_l x$ bestehen muss. Daher besteht z.B. λ_3 aus der Summe aller Werte $1^{v-3} \cdot X_i x \cdot X_j x \cdot X_l x = X$ (jeweils ohne das x^3, da nur der Koeffizient gesucht ist) aus $\lambda(x)$. Man

beachte, dass im binären Fall + und - die gleiche Operation darstellen.

$$\begin{aligned}\lambda(x) = \prod_{l=1}^{v} \cdot (1 - X_l x) = \\ &1^v + \\ &x \cdot (1^{v-1} \cdot X_1 + 1^{v-1} \cdot X_2 + \ldots + 1^{v-1} \cdot X_v) + \\ &x^2 \cdot (1^{v-2} \cdot X_1 X_2 + 1^{v-2} \cdot X_1 X_3 + \ldots + 1^{v-2} \cdot X_{v-1} X_v) + \\ &x^3 \cdot (1^{v-3} \cdot X_1 X_2 X_3 + 1^{v-3} \cdot X_1 X_2 X_4 + \ldots + 1^{v-3} \cdot X_{v-2} X_{v-1} X_v) + \\ &\ldots \\ &x^v \cdot (X_1 \cdot X_2 \cdot X_3 \cdot \ldots \cdot X_v)\end{aligned} \tag{2.126}$$

Betrachten wir das Produkt $\Sigma(x) = \lambda(x) \cdot (1 + S(x))$, so ist dieses ausgeschrieben

$$\begin{aligned}\lambda(x) \cdot (1 + S(x)) &= (1 + S_1 x + S_2 x^2 + \ldots + S_{2t} x^{2t})(1 + \lambda_1 x + \lambda_2 x^2 + \ldots + \lambda_v x^v) \\ &= \lambda(x) \cdot 1 + \lambda(x) \cdot S_1 x + \lambda(x) \cdot S_2 x^2 + \ldots + \lambda(x) \cdot S_{2t} x^{2t}\end{aligned} \tag{2.128}$$

Die Koeffizienten von x können hier nicht direkt abgelesen werden, da $\lambda(x)$ Potenzen von x beinhaltet. Nach der Erklärung von Wicker[Wic95] sind nun die Koeffizienten von $\Sigma(x)$ für ungerade Potenzen von x gleich 0, falls es sich um einen bis zu t-Bit Fehler handelt. Der Berlekamp Algorithmus verwendet nun eine Näherung Λ von $\lambda(x)$. Durch die Eigenschaft, dass die Koeffizienten für $\Sigma(x)$ für ungerade Potenzen von x gleich 0 ist, kann bei $\Lambda \cdot (1 + S(x))$ iterativ die Differenz Δ der jeweiligen Koeffizienten gebildet werden. Diese Differenz Δ wird anschließend verwendet, um die Näherung Λ zu verbessern. Für eine detaillierte Erklärung verweisen wir den Leser an dieser Stelle auf das Buch von Berlekamp[Ber68].

2.3.5.2 Beispiel Berlekamp-Massey Algorithmus

In diesem Abschnitt folgen wir Beispiel 9-3 aus dem Buch von Wicker[Wic95] und stellen dieses in ausführlich und detailliert dar. Wir betrachten hier den 2-Bit Fehlerkorrigierenden BCH Code mit dem Modularpolynom $p(x) = 1 + x^2$. Die H-Matrix dieses Codes sieht wie folgt aus:

$$H = \begin{bmatrix} 00001 & 00101 & 10011 & 11100 & 01101 & 11010 & 1 \\ 00010 & 01011 & 00111 & 11000 & 11011 & 10101 & 0 \\ 00100 & 10110 & 01111 & 10001 & 10111 & 01010 & 0 \\ 01000 & 01001 & 01100 & 11111 & 00011 & 01110 & 1 \\ 10000 & 10010 & 11001 & 11110 & 00110 & 11101 & 0 \\ & & & & & & \\ 00010 & 10110 & 10000 & 11001 & 00111 & 11011 & 1 \\ 01111 & 10111 & 00010 & 10110 & 10000 & 11001 & 0 \\ 00001 & 10010 & 01111 & 10111 & 00010 & 10110 & 1 \\ 00111 & 11011 & 10001 & 01011 & 01000 & 01100 & 1 \\ 10000 & 11001 & 00111 & 11011 & 10001 & 01011 & 0 \end{bmatrix} \tag{2.129}$$

Gegeben ist das übertragene Wort $r = (0010000110011000000000000000000)$ Somit ist

$$H \times r^T = \begin{pmatrix} 0\\0\\1\\0\\0\\ \\0\\1\\0\\1\\0 \end{pmatrix} + \begin{pmatrix} 1\\0\\1\\0\\0\\ \\1\\1\\0\\0\\0 \end{pmatrix} + \begin{pmatrix} 0\\1\\1\\0\\1\\ \\1\\1\\1\\1\\0 \end{pmatrix} + \begin{pmatrix} 0\\0\\1\\1\\1\\ \\0\\0\\1\\0\\0 \end{pmatrix} + \begin{pmatrix} 0\\1\\1\\1\\0\\ \\0\\0\\1\\0\\1 \end{pmatrix} = \begin{pmatrix} 1\\0\\1\\0\\0\\ \\0\\1\\1\\0\\1 \end{pmatrix} \tag{2.130}$$

Daher ist $S_1 = 10100$ und $S_3 = 01101$. Daraus folgt, dass

$$\begin{aligned} S_2 = S_1^2 =& 1010000 + 101000000 \bmod 100101 = \\ & 100010000 \bmod 100101 = \\ & 000111000 \bmod 100101 = \\ & 11101 \\ S_4 = S_2^2 =& 11101 + 1110100 + 11101000 + 111010000 \bmod 100101 = \\ & 101010001 \bmod 100101 = \\ & 001111001 \bmod 100101 = \\ & 000110011 \bmod 100101 = \\ & 10110 \end{aligned} \tag{2.132}$$

Eine einfachere Art der Berechnung ist möglich, indem man die α Werte verwendet. Diese entsprechen den Spalten der oberen 5 Zeilen der H-Matrix (Spalten des zu S_1 zugehörigen Teiles der H-Matrix). So ist $r = \alpha^2 + \alpha^7 + \alpha^8 + \alpha^{11} + \alpha^{12}$, $S_1 = 10100 = \alpha^7$, $S_3 = 01101 = \alpha^8$. Dadurch ergibt sich $S_2 = S_1^2 = (\alpha^7)^2 = \alpha^{14}$ und $S_4 = S_2^2 = \alpha^{28}$.

Nun können wir den Berlekamp Algorithmus mit diesen Werten verwenden:

1. Setze $k = 0, \quad \Lambda = 1, \quad T = 1, \quad \Delta = S_1$
2. Wiederhole während $k < t$:
   a) Setze $\Lambda' = \Lambda, \quad \Delta' = \Delta, \quad T' = T, \quad k = k + 1$
   b) Setze $\Lambda = \Lambda' + x \cdot \Delta' \cdot T'$
   c) Falls $\Delta' \neq 0$ und $deg(\Lambda') \leq k$: Setze $T = x \cdot \frac{\Lambda'}{\Delta'}$
      Sonst setze $T = x^2 \cdot T'$
   d) Setze Δ auf den Wert des Koeffizienten von x^{2k+1} in $\Lambda(1 + S(x))$
3. Interpretiere Λ als Lokatorpolynom und berechne die Fehlerstellen.

Dabei ist $t = 2$ gleich der Anzahl der Syndromkomponenten, $S(x) = \alpha^7 x + \alpha^{14} x^2 + \alpha^8 x^3 + \alpha^{28} x^4$ Diesen werden wir nun Schrittweise verfolgen:

- (1.) Setze $k = 0, \quad \Lambda = 1, \quad T = 1, \quad \Delta = \alpha^7$
- (2.) Prüfe $0 < 2$
- (2.a) Setze $\Lambda' = 1, \quad \Delta' = \alpha^7, \quad T' = 1, \quad k = 1$
- (2.b) Setze $\Lambda = 1 + x \cdot \alpha^7 \cdot 1 \quad = 1 + \alpha^7 x$
- (2.c) Prüfe $\Delta' = \alpha^7 \neq 0$ und $deg(\Lambda') = 0 \leq 1$,
  Daher setze $T = x \cdot \frac{1}{\alpha^7} = \alpha^{31-7}x = \alpha^{24}x$
- (2.d) Wir berechnen den Koeffizienten von x^3 in $\Lambda(1 + S(x))$:

$$\begin{aligned}\Lambda(1+S(x)) &= (1+\alpha^7 x)(1+S(x)) \\ &= (1+\alpha^7 x)\cdot 1 + (1+\alpha^7 x)\cdot(\alpha^7 x) + (1+\alpha^7 x)\cdot(\alpha^{14}x^2) \\ &\quad + (1+\alpha^7 x)\cdot(\alpha^8 x^3) + (1+\alpha^7 x)\cdot(\alpha^{28}x^4) + \ldots \\ &= \ldots + (\alpha^7 x)\cdot(\alpha^{14}x^2) + (1)\cdot(\alpha^8 x^3) + \ldots \\ &= \ldots + (\alpha^7 \cdot \alpha^{14} + \alpha^8)x^3 + \ldots\end{aligned} \tag{2.134}$$

  Der Koeffizient ist somit $\alpha^{7+14} + \alpha^8$. Addieren wir die Bitwerte aufeinander, ergibt sich $\Delta = 11000 + 01101 = 10101 = \alpha^{22}$.
- **Zwischenstand** $k = 1, \quad \Lambda = 1 + \alpha^7 x, \quad T = \alpha^{24}x, \quad \Delta = \alpha^{22}$
- (2.) Prüfe $1 < 2$
- (2.a) Setze $\Lambda' = 1 + \alpha^7 x, \quad \Delta' = \alpha^{22}, \quad T' = \alpha^{24}x, \quad k = 2$
- (2.b) Setze $\Lambda = (1 + \alpha^7 x) + x \cdot \alpha^{22} \cdot \alpha^{24}x = 1 + \alpha^7 x + \alpha^{22+24-31}x^2 = 1 + \alpha^7 x + \alpha^{15}x^2$
- (2.c) Prüfe $\Delta' = \alpha^{22} \neq 0$ und $deg(\Lambda') = 1 \leq 2$,
  Daher setze $T = x \cdot \frac{1+\alpha^7 x}{\alpha^{22}} = \alpha^{31-22}x + \alpha^{31+7-22}x^2 = \alpha^9 x + \alpha^{16}x^2$
- (2.d) Wir berechnen den Koeffizienten von x^5 in $\Lambda(1 + S(x))$:

$$\begin{aligned}\Lambda(1+S(x)) &= (1+\alpha^7 x+\alpha^{15}x^2)(1+S(x)) \\ &= \ldots + (1+\alpha^7 x+\alpha^{15}x^2)\cdot(\alpha^8 x^3) + (1+\alpha^7 x+\alpha^{15}x^2)\cdot(\alpha^{28}x^4)\end{aligned} \tag{2.136}$$

  Der Koeffizient von x^5 ist somit $\alpha^{15+8} + \alpha^{7+28}$. Addieren wir die Bitwerte aufeinander, ergibt sich $\Delta = 01111 + 10000 = 11111 = \alpha^{15}$.
- **Zwischenstand** $k = 2, \quad \Lambda = 1 + \alpha^7 x + \alpha^{15}x^2, \quad T = \alpha^9 x + \alpha^{16}x^2, \quad \Delta = \alpha^{15}$
- (2.) Prüfe $2 < 2$, daher endet die Wiederholung
- (3.) Λ wird als Lokatorpolynom interpretiert: $p(x) = 1 + \alpha^7 x + \alpha^{15}x^2$

Somit wurde das Lokatorpolynom als $p(x) = 1 + \alpha^7 x + \alpha^{15}x^2$ bestimmt. Die Fehlerstellen müssen zu diesem in einem gesonderten Schritt berechnet werden. Wie zuvor beschrieben, kann dazu Beispielsweise der Chien Search Algorithmus verwendet werden.

3 Kombinierte Korrektur und Erkennung höherer Fehler

In diesem Kapitel werden Verfahren vorgestellt, welche auf einfache Art und Weise BCH Korrektur und zusätzliche Mehrbiterkennung in einem Prozess kombinieren.

Eigener Anteil zur Erkennung höherer Fehler Kernaspekt dieser Arbeit ist es, die Erkennung höherer Fehler in BCH Codes mit besonderem Augenmerk auf Hardwareimplementierung zu vereinfachen. Neu ist hier, dass zunächst eine Korrektur von Bitfehlern erfolgt, welche am wahrscheinlichsten sind. Unter Verwendung dieser Korrekturwerte wird anschließend überprüft, ob ein höherer Fehler vorliegt. Dabei werden für die korrigierbaren Fehleranzahlen spekulative Fehlerpositionen berechnet, und aus diesen die spekulativen höheren Fehlersyndrome ermittelt und mit den tatsächlich aufgetretenen Fehlersyndromen auf Übereinstimmung verglichen. Insbesondere wird untersucht, welcher Zeitaufwand bei einer Berechnung in Software und welches Delay und Fläche in einer Implementierung in Hardware auftreten. Erkenntnisse dieser Arbeit wurden in den Publikationen [SH20], [SH21] und [SH22] veröffentlicht.

Unter der Voraussetzung, dass die Wahrscheinlichkeit für das Auftreten eines einzelnen Bitfehlers in einem übertragenen Codewort gering ist und mehrere fehlerhafte Bits unabhängig voneinander auftreten können, ist die Wahrscheinlichkeit für eine geringe Fehleranzahl nach Bernoulli wesentlich höher ist als bei einer hohen Fehleranzahl. Dadurch ist bei Fehler mit einer bestimmten Auftrittswahrscheinlichkeit z.B. von 10^{-6} bei einer Wortlänge von 256 Bit die Wahrscheinlichkeit für einen 1-Bit Fehler höher als alle anderen Fehler zusammen. Aus diesem Grund stellen wir ein Verfahren vor, durch welches Fehler mit geringer Bitzahl effizienter korrigierbar und von anderen, höheren Fehlern unterscheidbar sind.

3.1 1-Bit Fehlerkorrektur mit zusätzlicher Erkennung

Unter der Annahme, dass Fehler mit einer geringen Anzahl an Bits häufiger vorkommen, wird in diesem Abschnitt eine spezielle Erkennung für 1-Bit Fehler vorgestellt, mit dem Ziel, häufiger auftretende Fehler effizienter korrigieren zu können und trotzdem höhere Fehler von diesen unterscheiden zu können.

Hierbei wird im Gegensatz zu der Prüfung der Fehleranzahl durch Determinanten (Beschrieben in Abschnitt 2.3.1) mit aufsteigenden p vorgegangen, da für den 1-Bit Fall ohnehin alle Determinanten berechnet werden müssen. Das vorgestellte Verfahren hat dabei den Vorteil, dass die Überprüfung durch eine Vereinfachung der Determinanten wesentlich schneller berechenbar ist, als in dem üblichen Verfahren.

3.1.1 1-Bit Fehlererkennung

Ist das Syndrom ungleich dem Nullvektor, so ist ein mindestens ein 1-Bit Fehler aufgetreten. Ist zusätzlich $s_1 = \vec{0}$, dann ist ein mindestens 3-Bit Fehler aufgetreten, da alle 1-Bit und 2-Bit Fehler per Konstruktion durch die erste Syndromkomponente des BCH Code erkannt werden können. Ist dies der Fall, oder eine Determinante für ein höheres p wird ungleich dem Nullvektor, so tritt ein höherer Fehler auf und die 1-Bit Korrektur kann abgebrochen werden.

Da in jedem Schritt eine Gleichung $det(M(p)) = \vec{0}$ entsteht, kann diese zur Vereinfachung der Determinanten für folgende p (in steigender Folge) verwendet werden. Rechnerisch kann also die Gleichung $det(M(p)) = 0$ auf die folgenden Determinanten addiert werden, um eine kompakte Form der Folgedeterminanten zu erreichen.

Durch eine Reihe von Umformungen und Vereinfachungen ergibt sich, dass

$$det(M(p)) = (s_p + s_1^p)^{(p-1)/2}$$

gilt. Die Herleitung ist im Induktionsschritt von Kapitel 3.1.4 zu finden. Dies bedeutet, dass die Überprüfung der Determinante für ein p

$$det(M(p)) = 0$$

unter der Annahme eines 1-Bit Fehlers durch

$$s_p = s_1^p$$

ersetzt werden kann.

Durch diese Erkenntnis kann unter der Annahme eines 1-Bit Fehlers festgestellt werden, dass kein (p)-Bit oder (p-1)-Bit Fehler aufgetreten ist.

Die allgemeine Vorgehensweise unseres Ansatzes ist es, die folgenden Gleichungen zu überprüfen.

$$\begin{aligned} s &\overset{?}{\neq} 0 \\ s_1 &\overset{?}{\neq} 0 \\ s_3 &\overset{?}{=} s_1^3 \\ s_5 &\overset{?}{=} s_1^5 \\ s_7 &\overset{?}{=} s_1^7 \\ s_9 &\overset{?}{=} s_1^9 \\ &\dots \end{aligned} \tag{3.2}$$

Werden diese Gleichungen erfolgreich bis inklusive $s_p \overset{?}{=} s_1^p$ geprüft, so ist entweder ein 1-Bit Fehler aufgetreten oder mindestens ein $(p+1)$-Bit Fehler. Bei einem BCH Code, welcher bis zu n-Bit Fehler korrigieren kann, können die Gleichungen bis $s_q \overset{?}{=} s_1^q$ aufgestellt werden, wobei $q = n * 2 - 1$. Ein solcher n-Bit korrigierender BCH Code kann somit zur 1-Bit Korrektur bei gleichzeitiger

Erkennung von bis zu $(n*2-1)$-Bit Fehlern verwendet werden. Sind alle Gleichungen erfüllt, wurde ein 1-Bit Fehler festgestellt und ein möglicher unerkannter Fehler müsste mindestens $(n*2)$-Bit besitzen.

Hingegen, falls sich für einen Fehler mit $s \neq \vec{0}$ und $s_1 \neq \vec{0}$ beispielsweise ergibt, dass

$$\begin{aligned} s_3 &= s_1^3 \\ s_5 &= s_1^5 \\ s_7 &\neq s_1^7 \end{aligned} \tag{3.4}$$

ist kein 1-Bit Fehler aufgetreten. Allerding, da $det(M(7)) \neq 0$ gilt (alle vorhergehenden Reduktionen der Determinanten gelten weiterhin), kann gefolgert werden, dass ein mindestens 6-Bit, 7-Bit oder höherer Fehler aufgetreten ist. So kann zusätzlich zu der Aussage, dass kein 1-Bit Fehler aufgetreten ist, zudem noch eine untere Schranke für den aufgetretenen Fehler angegeben werden. Dies ist allerdings nur ein Zugewinn, falls mindestens $s_3 = s_1^3$, ansonsten ist ein 2-Bit, 3-Bit oder höherer Fehler aufgetreten, was bereits durch den nicht-Auftritt des 1-Bit Fehlers gegeben ist (falls das Syndrom nicht Null ist).

3.1.2 1-Bit Fehlerkorrektur

Um einen erkannten 1-Bit Fehler zu korrigieren, wird das Bit an der x-ten Stelle invertiert (die erste Stelle ist $x = 0$), wobei $\alpha^x = s_1$. Bei einer 1-Bit Fehlerkorrektur ist α^x die x-ten Spalte der H-Matrix. Tritt bei einem Codewort in der x-ten Stelle ein Fehler auf, so muss das Syndrom gleich der x-ten Spalte der H-Matrix sein. Dies kann durch eine Tabelle realisiert werden, da für verschiedene x die Werte für α^x verschieden sind.

3.1.3 Schematische Hardwareimplementierung

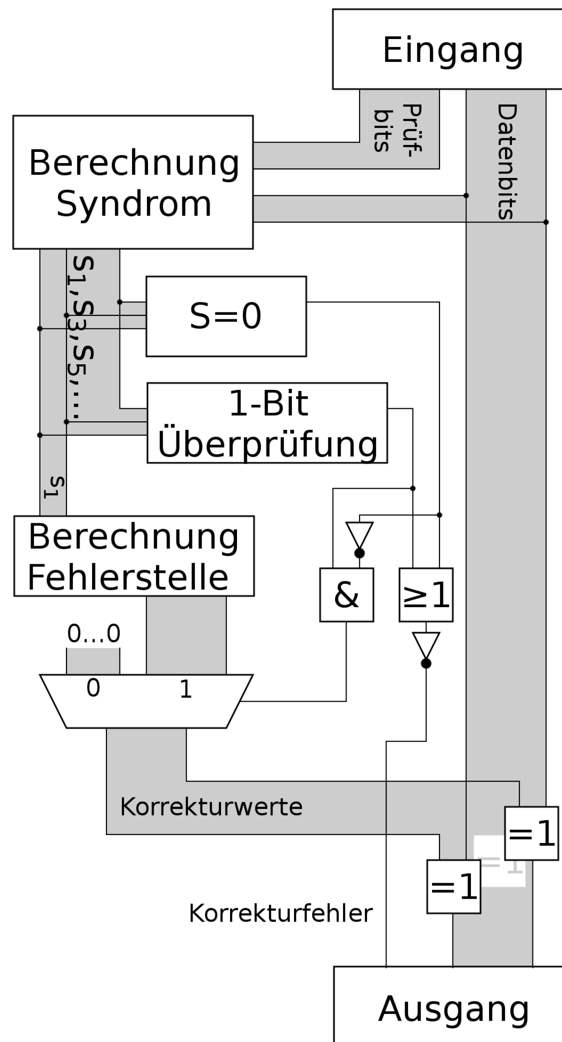

Abbildung 3.1: Schematische Hardwareimplementierung

Abbildung 3.1 zeigt eine schematische Hardwareimplementierung der 1-Bit Korrektur mit zusätzlicher Fehlerüberprüfung. Dabei findet eine Korrektur statt, falls ein 1-Bit Fehler erkannt wurde. Falls ein höherer Fehler festgestellt wird, wird dies lediglich als Korrekturfehler ausgegeben.

3.1.4 Beweis der Prüfgleichungen bei der 1-Bit Fehlerkorrektur

In diesem Abschnitt wird bewiesen, dass für den 1-Bit Fehler die Determinante $det(M(p))$ für ungerade $p > 1$ den Wert

$$det(M(p)) = (s_p + s_1^p)^{(p-1)/2}$$

annimmt. Diese Aussage erlaubt es, die Gleichung $s_p \stackrel{?}{=} s_1^p$ zu verwenden, um 1-Bit Fehler von höheren Fehlern zu unterscheiden. Um bei Ungleichheit eine Aussage über den mindestens stattdessen aufgetretenen Fehler zu treffen, müssen alle Gleichungen für kleinere p erfüllt sein, da diese zu der Vereinfachung vorausgesetzt werden.

Induktionsanfang n=3

$$\begin{aligned} &det(\begin{bmatrix} 1, & 0, & 0 \\ s_2, & s_1, & 1 \\ s_4, & s_3, & s_2 \end{bmatrix}) = 1 \times (s_1 \times s_2 - 1 \times s_3) \\ &= s_1 \times s_1^2 - s_3 = s_1^3 + s_3 \end{aligned} \tag{3.6}$$

Dies entspricht dem Wert der Formel:

$$\begin{aligned} det(M(3)) &= (s_1^3 + s_3)^{(3+1)/2-1} \\ &= (s_1^3 + s_3)^{2-1} \\ &= s_1^3 + s_3 \end{aligned} \tag{3.8}$$

Gilt $s_1^3 + s_3 \neq 0$, muss ein mindestens 2-Bit Fehler aufgetreten sein. Falls $s_1^3 + s_3 = 0$, somit $s_1^3 = s_3$ gilt, dann konnte kein 2-Bit oder 3-Bit Fehler festgestellt werden.

Induktionsschritt Wir leiten nun die Determinante für die Erkennung eines $(p-1)$-Bit und p-Bit Fehlers bei erwartetem 1-Bit Fehler Fehler her (p ist ungerade). Angenommen es tritt kein q-Bit Fehler mit $q < p-1$ jenseits des 1-Bit Fehlers auf (für die Werte $q \in [3, 5, 7, ..., p-2]$, gezeigt durch Vorgängerwerte der Induktion), daher gilt $det(M(q)) = 0$. Entsprechend der Induktionsannahme für jedes solches q:

$$\begin{aligned} &(s_q + s_1^q)^{(q+1)/2-1} = 0 \\ &s_q + s_1^q = 0 \\ &s_q = s_1^q \end{aligned} \tag{3.10}$$

Da die Matrix p Spalten hat, ist der höchste Index einer Syndromkomponente $2 \times p - 2$. Da im Galoisfeld GF(2^m) generell $s_{2q} = s_q^2$ gilt, können gerade Syndromkomponenten durch kleinere Syndromkomponenten dargestellt werden. Durch den Vorgängerschritt der Induktion wurde gezeigt, dass für $q < p-1$ die Gleichung $s_q = s_1^q$ gilt. Dies beinhaltet alle geraden Syndromkomponenten bis auf die mit dem größten Index. Für den größten Index von Syndromkomponenten der Matrix gilt $s_{2p-2} = s_{p-1}^2$. Da $p-1$ gerade ist ergibt sich $s_{2p-2} = s_{(p-1)\div 2}^4 = s_1^{2p-2}$. Somit gilt für alle s_q in der Matrix mit geraden q: $s_q = s_1^q$.

Zur Visualisierung hier eine exemplarische Darstellung der Matrix für $p = 7$

$$\begin{bmatrix} 1, & 0, & 0, & 0, & 0, & 0, & 0 \\ s_1^2, & s_1, & 1, & 0, & 0, & 0, & 0 \\ s_1^4, & s_1^3, & s_1^2, & s_1, & 1, & 0, & 0 \\ s_1^6, & s_1^5, & s_1^4, & s_1^3, & s_1^2, & s_1, & 1 \\ s_1^8, & s_7, & s_1^6, & s_1^5, & s_1^4, & s_1^3, & s_1^2 \\ s_1^{10}, & s_9, & s_1^8, & s_7, & s_1^6, & s_1^5, & s_1^4 \\ s_1^{12}, & s_{11}, & s_1^{10}, & s_9, & s_1^8, & s_7, & s_1^6 \end{bmatrix} \tag{3.11}$$

Durch die Faktorenregel für Determinanten können Zeilen und Spalten wie folgt multipliziert werden. Jede gerade Spalte (beginnend bei der zweiten Spalte) q wird mit s_1^q multipliziert (Exponenten sind $2,4,6,8,...$). Jede ungerade Spalte o wird durch s_1^{p-o} geteilt (Exponenten sind $p-1, p-3, p-4$ bzw. in verkehrter Reihenfolge $0,2,4,6,8,...$). Da p ungerade ist, ergibt sich, dass sich alle Faktoren insgesamt zu 1 auf multiplizieren. Die Determinante der Matrix bleibt somit unverändert.
Zusätzlich wird jede Zeile o mit $s_1^{p+1-2\times o}$ multipliziert (Exponenten sind $p-1, p-3, p-5, ..., 4, 2, 0, 2, ..., -p+5, -p+3, -p+1$). Diese Faktoren multipliziert sich ebenfalls zu 1 auf.
Zur Veranschaulichung hier exemplarisch die Faktoren für $p = 7$
Faktoren der Spalten:

$$[s_1^{-6}, s_1^2, s_1^{-4}, s_1^4, s_1^{-2}, s_1^6, s_1^0]$$

Und als Faktoren der Zeilen (von oben nach unten):

$$[s_1^6, s_1^4, s_1^2, s_1^0, s_1^{-2}, s_1^{-4}, s_1^{-6}]$$

Die Matrix für $p = 7$ sieht exemplarisch nach diesem Schritt wie folgt aus:

$$\begin{bmatrix} 1, & 0, & 0, & 0, & 0, & 0, & 0 \\ 1, & s_1^7, & 1, & 0, & 0, & 0, & 0 \\ 1, & s_1^7, & 1, & s_1^7, & 1, & 0, & 0 \\ 1, & s_1^7, & 1, & s_1^7, & 1, & s_1^7, & 1 \\ 1, & s_7, & 1, & s_1^7, & 1, & s_1^7, & 1 \\ 1, & s_9 \times s_1^{-2}, & 1, & s_7, & 1, & s_1^7, & 1 \\ 1, & s_{11} \times s_1^{-4}, & 1, & s_9 \times s_1^{-2}, & 1, & s_7, & 1 \end{bmatrix} \tag{3.12}$$

Allgemein ergibt sich, dass jede ungerade Spalte nur noch aus den Elementen 0 und 1 besteht und jedes Element in geraden Spalten entweder 0 ist, oder die Summe der Indexe der Syndromkomponente mal ihrer Exponenten gleich p ist.
Es ergibt sich im Folgenden, dass die Gesamtdeterminante gleich der Unterdeterminante ist, welche durch zeilenweise Entwicklung aller ungeraden Spalten (bis auf die letzte) nach der obersten 1 ist, da alle anderen Unterdeterminanten in zeilenweiser Entwicklung den Wert 0 ergeben. Dies kann wie folgt bewiesen werden.

Betrachten wir alle ungeraden Spalten a bis auf die erste, auf welche eine die weitere ungerade

Spalte b folgt (z.B. $b = a+2$). Die Werte der Spalte a sind nun in der Zeile i:

$$\begin{aligned} & s_{(i-1)\times 2-(a-1)} / s_1^{p-a} \times s_1^{p+1-2\times i} \\ & = s_1^{i\times(2-2)+p\times(-1+1)+a\times(-1+1)+(-2+1+1)} = 1 \end{aligned} \tag{3.14}$$

Falls $(a-1) <= 2\times(i-1)$, sonst 0. Die Spalten a und b sind nun bis auf die Zeile $i = (a+1)/2$ identisch. In dieser Zeile steht in a eine **1** ($(a-1) = 2\times((a+1)/2-1)$), in b eine **0** ($(a+2-1) > 2\times((a+1)/2-1)$).

Nehmen wir an, die Determinante wird zeilenweise (von oben nach unten) entwickelt. Wir nehmen weiterhin an, dass bisher keine 2 Spalten der aktuellen Untermatrix identisch sind, da entsprechend der Rechenregel für Determinanten diese Unterdeterminante gleich 0 wäre. Angenommen die Entwicklung der Determinante wird nun in Zeile $(a+1)/2$ (gegeben ein solches a und b) durchgeführt. Die Spalten a und b unterscheiden sich lediglich in der zu entwickelnden Zeile um einen Wert. Wird in dieser Zeile nicht nach Spalte a oder b entwickelt, ergibt sich, dass in der resultierenden Unterdeterminante die Spalten a und b identisch sein müssen. Diese Unterdeterminante wäre somit 0. Da der Wert der Spalte b an Position $(a+1)/2$ den Wert 0 beträgt, ergibt die Entwicklung nach dieser ebenfalls 0. Die einzige Unterdeterminante, welche nicht sicher 0 ergibt, erfordert in Zeile $(a+1)/2$ somit eine Entwicklung nach Spalte a.

Da dies für alle ungeraden Spalten a bis auf die letzte Spalte gilt und der Wert $(a+1)/2$ für ungerade a somit abgerundet die erste Hälfte der Zeilen betrifft, wurde bewiesen, dass es genügt, zunächst alle ungeraden Spalten (bis auf die letzte, bei welcher keine Folgespalte b existiert) nach der ersten auftretenden 1 zu entwickeln, da alle anderen Entwicklungen 0 ergeben.

Durch die Entwicklung entfallen die ersten $(p-1)/2$ Zeilen und die ungeraden Spalten bis auf die letzte. Die Determinante dieser Entwicklung ist somit gleich der Determinante der gesamten Matrix. Allgemein besitzt die entwickelte $(p+1)/2\times(p+1)/2$ Matrix folgende Form

$$\begin{bmatrix} s_1^p, & ..., & ..., & ..., & s_1^p, & 1 \\ s_p, & s_1^p & ..., & ..., & s_1^p, & 1 \\ s_{p+2}\times s_1^{-2}, & s_p, & s_1^p, & ... & s_1^p, & 1 \\ ..., & ..., & ..., & ..., & ..., & 1 \\ s_{(p-1)\times 2-3}\times s_1^{p-((p-1)\times 2-3)}, & ..., & s_{p+2}\times s_1^{-2}, & s_p, & s_1^p, & 1 \\ s_{(p-1)\times 2-1}\times s_1^{p-/(p-1)\times 2-1)}, & ..., & ..., & s_{p+2}\times s_1^{-2}, & s_p, & 1 \end{bmatrix} \tag{3.15}$$

Zur Visualisierung hier eine exemplarische Darstellung der entwickelten Matrix für $p = 7$ nach diesem Schritt:

$$\begin{bmatrix} s_1^7, & s_1^7, & s_1^7, & 1 \\ s_7, & s_1^7, & s_1^7, & 1 \\ s_9\times s_1^{-2}, & s_7, & s_1^7, & 1 \\ s_{11}\times s_1^{-4}, & s_9\times s_1^{-2}, & s_7, & 1 \end{bmatrix} \tag{3.16}$$

Jede Spalte i der reduzierten Matrix besitzt i-mal das Element s_1^p (oder 1 bei der letzten Spalte). Bei einer spaltenweisen Entwicklung (links nach rechts) ergibt sich, dass eine Entwicklung der Spalte i nach einer Zeile $j > i+1$ in den obersten 2 Zeilen der Unterdeterminante folgende Form

ergibt:

$$\begin{bmatrix} s_1^p, & s_1^p, & \dots, & s_1^p, & 1 \\ s_1^p, & s_1^p, & \dots, & s_1^p, & 1 \end{bmatrix} \tag{3.17}$$

Da diese Spalten identisch sind, ist die Unterdeterminante 0.
Es ergibt sich, dass bei einer spaltenweisen Entwicklung in Spalten i jeweils $i-1$ Zeilen zu 0 entwickeln, damit verbleiben für jede Spalte $(i+1)-(i-1)=2$ mögliche Zeilen, deren Entwicklungen nicht 0 sein müssen. Die Gesamtdeterminante ergibt sich somit, durch die spaltenweise Entwicklung der Werte $[s_1^p, s_p]$. In Folgespalten ergibt sich dies ebenfalls, da die beiden zu entwickelnden Zeilen bis auf das erste Element identisch sind, nach dessen Spalte entwickelt wird. Es ergibt sich durch die letzte Spalte ein Faktor von 1, der mit den Entwicklungen multipliziert wird. Die Unterdeterminante ist somit für die beiden Entwicklungen einer Spalte äquivalent.
In jedem der $(p+1)/2-1$ Schritte ergibt sich somit $s_1^p \times v + s_p \times v = (s_1^p + s_p) \times v$, wobei v die in beiden Fällen gleiche Unterdeterminante ist.
Bei $(p+1)/2-1$ Schritten, welche jeweils nach $[s_1^p, s_p]$ entwickelt werden, ergibt sich, dass der Wert der Determinante äquivalent zu

$$(s_1^p + s_p)^{(p+1)/2-1} \times 1$$

sein muss.

Somit wurde bewiesen, dass im Falle eines 1-Bit Fehlers für ungerade $p > 1$ gilt:

$$det(M(p)) = (s_p + s_1^p)^{(p-1)/2}$$

3.2 2-Bit Fehlerkorrektur mit zusätzlicher Erkennung

In diesem Abschnitt wird die Erkennung und Korrektur von 2-Bit Fehlern betrachtet. Als ersten Schritt wird die Anwendung des 1-Bit Ansatzes demonstriert und weshalb dieser für den 2-Bit Fehlerfall nur begrenzt verwendbar ist.

Im Anschluss wird ein neuer Ansatz der spekulativen Fehlerberechnung vorgestellt, welcher eine effiziente 2-Bit Fehlerkorrektur mit zusätzlicher Erkennung ermöglicht. Es wird dazu gezeigt, dass dieses Verfahren auf die Fehlerkorrektur höherer Fehler erweiterbar ist. Für dieses Verfahren wird zusätzlich eine schematische Schaltung und ein VHDL Entwurf vorgestellt. Für Beispielfälle wird ein Vergleich von Softwareimplementierungen zwischen der Determinantenberechnung und dem neuen Ansatz der spekulativen Fehlerberrechnung gezeigt. Überprüfung der Effizient in Software. Die detaillierte Überprüfung der Sinnhaftigkeit des Verfahrens in Software wurde für eine große Anzahl stochastisch implementierter Fehler durchgeführt. Insbesondere wurden die beiden Verfahren für verschiedene Größen des Galoisfeldes verglichen, was interessanter Weise unterschiedliche Ergebnisse ergibt.

3.2.1 Anwendung des 1-Bit Ansatzes

Auch im 2-Bit Fall lassen sich die Determinanten für die höheren Fehler, unter der Annahme, dass ein 2-Bit Fehler vorliegt, ebenfalls vereinfachen. Im 2-Bit Fall muss zunächst $s_1 \neq 0$ gelten.

Zudem gilt im Gegensatz zum 1-Bit Fehler $0 \neq s_1^3 + s_3$. Die Determinanten $det(M(4))$ und aufwärts werden verwendet, um die jeweils folgenden Determinanten zu vereinfachen.

Die Anzahl der Terme der Determinanten $det(M(p))$ steigt im 2-Bit Fehlerfall faktoriell zu p. Die gekürzten Determinanten ergeben sich exemplarisch wie folgt:

- Ein 2 oder 3-Bit Fehler: $0 \neq det(M(3)) = s_1^3 + s_3$
- Kein 3 oder 4-Bit Fehler: $0 = det(M(4)) = s_1^3 s_3 + s_1^6 + s_3^2 + s_5 s_1$
- Kein 4 oder 5-Bit Fehler: $0 = det(M(5)) = s_1^3 s_7 + s_5^2 + s_3 s_7 + s_1^7 s_3$
- Kein 5 oder 6-Bit Fehler: $0 = det(M(6)) = s_1^{15} + s_5^3 + s_3^5 + s_1^5 s_5^2 + s_1 s_7^2 + s_1^7 s_3 s_5$
- Kein 6 oder 7-Bit Fehler: $0 = det(M(7)) = s_1^{15} s_3^2 + s_1^9 s_5 s_7 + s_1^9 s_3 s_9 + s_1^3 s_3^3 s_9 + s_1^9 s_3^4 + s_3^7 +$
$s_1^5 s_3^2 s_5^2 + s_1^{14} s_7 + s_1^{12} s_9 + s_1^2 s_3 s_7 s_9 + s_1^5 s_7 s_9 + s_1^3 s_9^2 + s_1^3 s_7 s_{11} + s_7^3 + s_5^2 s_{11} + s_3 s_9^2 + s_3 s_7 s_{11} + s_1^7 s_3 s_{11} +$
$s_3^2 s_9 s_5 s_1 + s_1^3 s_5 s_7 s_3^2 + s_1^2 s_5^2 s_3^3 + s_5^3 s_1^6$
- Kein 8 oder 9-Bit Fehler: $0 = det(M(9)) = s_1^{17} s_3 s_7 s_9 + s_1^{15} s_3^7 + s_1^{16} s_3 s_5^2 s_7 + s_1^{17} s_5 s_7^2 + s_1^5 s_3^3 s_9 s_{13} +$
$s_1^{29} s_7 + s_1^{25} s_{11} + s_1^5 s_5^3 s_7 s_9 + s_1^{11} s_3^4 s_{13} + s_1^{21} s_3^5 + s_1^{12} s_3^2 s_7 s_{11} + s_1^{13} s_3^4 s_{11} + s_1^2 s_5^5 s_9 + s_1^2 s_3^4 s_5^3 s_7 + s_1^5 s_3^2 s_5^5 +$
$s_1^5 s_3^3 s_5^3 s_7 + s_1^{11} s_3 s_5^3 s_7 + s_1 s_3 s_5^5 s_7 + s_1 s_3^5 s_5^4 + s_1^{12} s_5^3 s_9 + s_1^{18} s_9^2 + s_1^{18} s_7 s_{11} + s_1^4 s_5^2 s_{11}^2 + s_1 s_7^5 + s_1 s_3 s_7^2 s_9^2 +$
$s_1^{24} s_3^4 + s_1^{12} s_3^8 + s_1^2 s_3^3 s_5 s_9 s_{11} + s_1^7 s_3^2 s_5^2 s_{13} + s_1^2 s_3 s_5^3 s_7 s_9 + s_3^3 s_9^3 + s_3 s_5^3 s_9^2 + s_3^2 s_5^2 s_9 s_{11} + s_1^{11} s_5^5 + s_1 s_5^7 +$
$s_1^{15} s_7^3 + s_1^{15} s_3^2 s_{15} + s_1^9 s_9^3 + s_1^9 s_7 s_9 s_{11} + s_1^9 s_5 s_7 s_{15} + s_1^9 s_3 s_{11} s_{13} + s_1^9 s_3 s_9 s_{15} + s_1^{16} s_7 s_{13} + s_1^3 s_3^3 s_{11} s_{13} +$
$s_1^3 s_3^3 s_9 s_{15} + s_1^8 s_5^2 s_7 s_{11} + s_1^{14} s_9 s_{13} + s_1^2 s_3^7 s_{13} + s_1^9 s_5^4 s_7 + s_1^{11} s_7 s_9^2 + s_1^{13} s_5 s_9^2 + s_1^7 s_7 s_{11}^2 + s_1^7 s_7 s_9 s_{13} +$
$s_1^{17} s_3^2 s_{13} + s_1^9 s_3^4 s_{15} + s_1^2 s_5^4 s_7^2 + s_1^3 s_5^3 s_7 s_{11} + s_1^{11} s_5 s_7 s_{13} + s_5^3 s_7^3 + s_1^{12} s_3 s_7^3 + s_3^7 s_{15} + s_1^2 s_3^4 s_{11}^2 + s_1 s_3^5 s_9 s_{11} +$
$s_1 s_3^4 s_7^2 s_9 + s_1^5 s_3^2 s_5^2 s_{15} + s_1^{11} s_3 s_9 s_{13} + s_1^{14} s_7 s_{15} + s_1^{12} s_{11} s_{13} + s_1^{12} s_9 s_{15} + s_1^4 s_3 s_7 s_{11}^2 + s_1^4 s_3 s_7 s_9 s_{13} +$
$s_1^2 s_7 s_9^3 + s_1^2 s_7^3 s_{13} + s_1^4 s_5 s_9^3 + s_1^2 s_3 s_9 s_{11}^2 + s_1^2 s_3 s_7 s_{11} s_{13} + s_1^2 s_3 s_9^2 s_{13} + s_1^2 s_3 s_7 s_9 s_{15} + s_1^5 s_9 s_{11}^2 + s_1^5 s_9^2 s_{13} +$
$s_1^5 s_7 s_{11} s_{13} + s_1^5 s_7 s_9 s_{15} + s_1^3 s_{11}^3 + s_1^3 s_9^2 s_{15} + s_1^3 s_7 s_{13}^2 + s_1^3 s_7 s_{11} s_{15} + s_9^4 + s_7 s_9^2 s_{11} + s_7^2 s_{11}^2 + s_7^3 s_{15} +$
$s_5^2 s_{13}^2 + s_5^2 s_{11} s_{15} + s_3 s_{11}^3 + s_3 s_9^2 s_{15} + s_3 s_7 s_{13}^2 + s_3 s_7 s_{11} s_{15} + s_1^7 s_3 s_{15} s_{11} + s_3^2 s_{15} s_9 s_5 s_1 + s_1^3 s_5 s_{15} s_7 s_3^2 +$
$s_1^2 s_5^2 s_{15} s_3^3 + s_1^7 s_3 s_{13}^2 + s_3^2 s_{13} s_{11} s_5 s_1 + s_3^2 s_{13} s_9 s_5 s_1^3 + s_1^5 s_5 s_{13} s_7 s_3^2 + s_1^4 s_5^2 s_{13} s_3^3 + s_3^3 s_{11}^2 s_1^5 + s_1^7 s_{11} s_9^2 +$
$s_1 s_5 s_{11} s_9 s_7 s_3 + s_1^{11} s_5 s_{11} s_9 + s_1^9 s_{11} s_7 s_3^3 + s_1^{17} s_3 s_{11} s_5 + s_1^5 s_3^2 s_9^2 s_7 + s_1 s_5 s_9 s_7^3 + s_1^9 s_5 s_9 s_7 s_3^2 + s_1^{14} s_9 s_5^2 s_3 +$
$s_1^{27} s_9 + s_1 s_5 s_7^3 s_3^3 + s_1^{33} s_3 + s_5^3 s_{15} s_1^6 + s_5^3 s_{13} s_1^8 + s_1^{15} s_{11} s_5^2 + s_1^2 s_5^4 s_{11} s_3$
- Kein 10 oder 11-Bit Fehler: $0 = det(M(11)) = s_1^5 s_3^{10} s_7 s_{13} + s_1^{15} s_3^{10} s_5^2 + s_1^{31} s_3^3 s_5^3 + s_1^{29} s_3^7 s_5 +$
$s_1^{31} s_3 s_5 s_7 s_9 + s_1^{31} s_3^2 s_5 s_{13} + s_1^3 s_3^7 s_5^3 s_7 s_9 + s_1^{15} s_3^3 s_7 s_9 s_{15} + s_1^3 s_3 s_5^4 s_7 s_9 s_{13} + s_1^3 s_3^3 s_5^2 s_9^2 s_{15} + s_1^3 s_3^3 s_5^2 s_7 s_{13}^2 +$
$s_1^{23} s_5 s_7 s_9 s_{11} + s_1^{10} s_5^4 s_7^2 s_{11} + s_1^{15} s_3 s_5^3 s_7 s_{15} + s_1^{11} s_3^2 s_5 s_9 s_{11} s_{13} + s_1^{19} s_5 s_7^2 s_{17} + s_1^7 s_3^3 s_9 s_{13} s_{17} + s_1^{24} s_5 s_{11} s_{15} +$
$s_1^{18} s_3^4 s_7 s_9^2 + s_1^{27} s_{13} s_{15} + s_1^{27} s_{11} s_{17} + s_1^{17} s_7 s_9^2 s_{13} + s_1^{17} s_5 s_9^2 s_{15} + s_1^{17} s_3 s_9 s_{13}^2 + s_1^{10} s_5^3 s_{15}^2 + s_1^{16} s_3^2 s_5^3 s_7 s_{11} +$
$s_1^{16} s_3^7 s_9^2 + s_1^{18} s_3 s_5 s_7^2 s_{15} + s_1^{39} s_7 s_9 + s_1^{14} s_3^4 s_5^4 s_9 + s_1^{26} s_5^4 s_9 + s_1^{21} s_3^6 s_7 s_9 + s_1^{15} s_3^4 s_{11} s_{17} + s_1^4 s_3^4 s_5^3 s_7 s_{17} +$
$s_1^3 s_5^5 s_7 s_9 s_{11} + s_1^{19} s_7^3 s_{15} + s_1^7 s_3^2 s_5^5 s_{17} + s_1^9 s_3^5 s_7 s_9 s_{15} + s_1^7 s_3^3 s_5^3 s_7 s_{17} + s_1^5 s_3^5 s_7 s_{13} s_{15} + s_1^{28} s_3^3 s_9^2 + s_1^{16} s_3^5 s_9 s_{15} +$
$s_1^3 s_3 s_5^5 s_7 s_{17} + s_1^{19} s_3 s_5^4 s_{13} + s_1^{12} s_3^2 s_5^2 s_7 s_9 s_{11} + s_1^{10} s_3 s_7^2 s_{13} s_{15} + s_1^4 s_3^5 s_7^2 s_{11}^2 + s_1^{11} s_3^3 s_7 s_{13} s_{15} + s_1^3 s_5^6 s_9 s_{13} +$
$s_1^{20} s_{11}^2 s_{13} + s_1^{20} s_9^2 s_{17} + s_1^4 s_3^4 s_5^2 s_7 s_{11}^2 + s_1^6 s_7^2 s_{11}^2 s_{13} + s_1^3 s_7^3 s_9^2 s_{13} + s_1^3 s_7^5 s_{17} + s_1^3 s_5^3 s_{11}^2 s_{15} + s_1^3 s_5^3 s_7 s_{15}^2 +$
$s_1^{18} s_3^7 s_7 s_9 + s_1^{36} s_3 s_7 s_9 + s_1^{10} s_3^4 s_5^3 s_9^2 + s_1^{14} s_3 s_5^4 s_7 s_{11} + s_1^7 s_3^{10} s_9^2 + s_1^{14} s_3^7 s_9 s_{11} + s_1^{16} s_3 s_5^2 s_{11} s_{15} + s_1^{16} s_3 s_7 s_9^2 s_{11} +$
$s_1^{39} s_3 s_{13} + s_1^{13} s_3^{14} + s_1^{47} s_3 s_5 + s_1^5 s_3^{12} s_7^2 + s_1^{55} + s_1^{52} s_3 + s_1^{19} s_3^{12} + s_1^{14} s_3^{10} s_{11} + s_1^{16} s_3^4 s_7 s_9 s_{11} + s_1^{19} s_5^2 s_{13}^2 +$
$s_1^{15} s_3^2 s_7 s_9^3 + s_1^{17} s_3^4 s_{13}^2 + s_1^2 s_3^7 s_5^5 s_7 + s_1^2 s_3^{11} s_5^4 + s_1^{13} s_3^8 s_7 s_{11} + s_1^2 s_3^8 s_5^4 s_9 + s_1^2 s_3 s_7^2 s_9^4 + s_1^2 s_3 s_7^3 s_9^2 s_{11} +$
$s_1^{13} s_3^2 s_5^4 s_7 s_9 + s_1^5 s_3^{10} s_5^4 + s_1^8 s_3^2 s_5 s_7^2 s_{11}^2 + s_1^5 s_3^8 s_5 s_7^3 + s_1^{21} s_5^2 s_9 s_{15} + s_1^{13} s_3^9 s_{15} + s_1^{19} s_3^2 s_{13} s_{17} + s_1^8 s_3 s_5^4 s_9 s_{15} +$
$s_1^{16} s_3^3 s_{15}^2 + s_1^{13} s_5^5 s_{17} + s_1^{15} s_5 s_9^2 s_{17} + s_1^{15} s_5 s_7 s_{13} s_{15} + s_1^{10} s_7^3 s_{11} s_{13} + s_1 s_3^{11} s_5 s_7 s_9 + s_1^5 s_3^3 s_5^5 s_7 s_9 + s_1^{13} s_3^4 s_5^3 s_{15} +$
$s_1 s_3^{15} s_9 + s_1 s_3^3 s_5^5 s_9 s_{11} + s_1^9 s_3^{10} s_7 s_9 + s_1^9 s_3^{12} s_5^2 + s_1^{17} s_3^{11} s_5 + s_1^{21} s_3^3 s_5^5 + s_1 s_3^{18} + s_1^{18} s_3^{10} s_7 + s_1^{36} s_3^2 s_{13} +$
$s_1 s_3^4 s_5^4 s_{11}^2 + s_1 s_3^{14} s_5 s_7 + s_1 s_3^3 s_5^4 s_7 s_9^2 + s_1^{27} s_3 s_5^5 + s_1^{23} s_3^4 s_5 s_{15} + s_1^{12} s_3^3 s_7 s_9^3 + s_1^{24} s_3 s_5^2 s_9^2 + s_1^{22} s_3^6 s_{15} +$

$$s_1^{37}s_3s_{15}+s_1^2s_3s_5^4s_7^3s_9+s_1s_3s_5^7s_7s_9+s_1s_3^5s_5s_7s_9^3+s_1^3s_3^{10}s_7s_{15}+s_1^{26}s_3^3s_5s_{15}+s_1s_3^3s_9^5+s_1s_3^3s_5s_9^2s_{11}^2+$$
$$s_1s_3^6s_7^3s_{15}+s_1^2s_3^{14}s_{11}+s_1^4s_3^6s_5^3s_9^2+s_1^{10}s_3^3s_7^2s_{11}^2+s_1^{16}s_5s_7s_9^3+s_1^{10}s_9^5+s_1^{10}s_5s_9^3s_{13}+s_1^{10}s_3s_9^3s_{15}+$$
$$s_1^{17}s_7^3s_{17}+s_1^4s_5^4s_9^2s_{13}+s_1^{11}s_3^{10}s_7^2+s_1^4s_3^4s_5^6s_9+s_1^3s_5^7s_{17}+s_1^3s_3^4s_7^2s_9s_{17}+s_1^{10}s_3^4s_9s_{11}s_{13}+s_1^{11}s_5^4s_7s_{17}+$$
$$s_1^{19}s_5^4s_7s_9+s_1^2s_3^2s_7^2s_{11}^3+s_1^2s_3s_7^4s_{11}^2+s_1s_3^2s_5^6s_9^2+s_1^9s_3^8s_9s_{13}+s_1^6s_3^4s_{11}s_{13}^2+s_1^4s_3^6s_7s_{13}^2+s_1^{13}s_5^4s_7s_{15}+$$
$$s_1^9s_3^7s_5s_9s_{11}+s_1s_3^4s_5^7s_7+s_1^9s_3^8s_5^3s_7+s_1s_3^{10}s_5^3s_9+s_1^5s_5^5s_7s_9^2+s_1^9s_3^9s_5^2s_9+s_1^9s_3^{10}s_5s_{11}+s_1^{23}s_5^2s_7s_{15}+$$
$$s_1^{14}s_5^3s_{11}s_{15}+s_1s_5^5s_7s_{11}^2+s_1^9s_3s_5^5s_9^2+s_1s_3^{12}s_7s_{11}+s_1s_3^{13}s_{15}+s_1^{25}s_3^3s_5^2s_{11}+s_1^2s_3^{11}s_9s_{11}+s_1^{14}s_3s_7^3s_{17}+$$
$$s_1^{20}s_3^3s_{13}^2+s_1^9s_3s_5^2s_{11}^3+s_1s_3^9s_7s_9s_{11}+s_1s_3^9s_5s_7s_{15}+s_1^{18}s_{11}^2s_{15}+s_1^{18}s_7s_{13}s_{17}+s_1^{22}s_3s_{15}^2+s_1^{20}s_5s_{15}^2+$$
$$s_1^4s_5^4s_7^2s_{17}+s_1^4s_3^7s_{13}s_{17}+s_1^{23}s_3^4s_7s_{13}+s_1^2s_5^3s_7^3s_{17}+s_1^2s_3^9s_{11}s_{15}+s_1^2s_3^3s_7^2s_{15}^2+s_1^9s_3^3s_7s_{15}^2+s_1s_3^5s_5^2s_7s_{11}^2+$$
$$s_1s_3^7s_7s_{13}^2+s_1^3s_7^3s_9s_{11}^2+s_1^3s_7s_9^5+s_1s_5^5s_9^2s_{11}+s_1s_3^2s_5^3s_{11}^3+s_1^5s_5s_9^5+s_1^5s_3s_5s_9^3s_{15}+s_1s_3^3s_5^3s_{15}^2+s_1^3s_3^5s_9s_{13}s_{15}+$$
$$s_1^3s_3^5s_9s_{11}s_{17}+s_1^3s_3^5s_7s_{15}^2+s_1^3s_3s_7s_9^3s_{15}+s_1^4s_7^3s_{15}^2+s_1^4s_7^3s_{13}s_{17}+s_1^4s_7s_9^2s_{13}^2+s_1s_5^2s_{11}^4+s_1s_9^6+$$
$$s_1s_7s_9^4s_{11}+s_1s_5^2s_9^2s_{11}s_{15}+s_1s_3s_9^2s_{11}^3+s_1s_3s_9^4s_{15}+s_1s_3s_7s_9^2s_{13}^2+s_1s_3s_7s_9^2s_{11}s_{15}+s_1^4s_3^3s_5^7s_7+$$
$$s_1s_3^{11}s_7^3+s_3^7s_7^3s_{13}+s_1^{10}s_3^5s_5^2s_9s_{11}+s_1^{22}s_3^3s_9s_{15}+s_1^2s_3^5s_7^2s_9s_{15}+s_1s_3^8s_7^3s_9+s_3^3s_5^6s_7s_9+s_3^4s_5^2s_7s_{13}^2+$$
$$s_3^{10}s_5^2s_{15}+s_1^{10}s_3s_5^7s_7+s_1^{16}s_3s_5s_7s_9s_{15}+s_1^{14}s_3^8s_{17}+s_1^4s_3^3s_5s_9s_{13}s_{15}+s_1^4s_3^3s_5s_9s_{11}s_{17}+s_1^4s_3^4s_{13}^3+$$
$$s_1^4s_3^4s_{11}^2s_{17}+s_1^{21}s_3s_5s_{11}s_{15}+s_1^4s_3s_5^3s_7s_9s_{17}+s_1^3s_3s_5^4s_7^2s_{15}+s_1^7s_3s_5^3s_{15}^2+s_1^2s_3s_5^2s_9^3s_{13}+s_1^2s_3^2s_7^2s_9^2s_{15}+$$
$$s_1^2s_3^2s_7^3s_{13}^2+s_1^{17}s_3^3s_5^4s_9+s_1^3s_3^{11}s_5^2s_9+s_1^{11}s_3s_5s_7^3s_{15}+s_1^{31}s_3^2s_9^2+s_1^{20}s_3^8s_{11}+s_1s_3^7s_5^3s_9^2+s_1s_3^{11}s_5^2s_{11}+$$
$$s_3s_5^9s_7+s_1^{32}s_3s_5^4+s_3^3s_5^7s_{11}+s_1s_3^3s_5^4s_7^2s_{11}+s_3s_5^5s_7s_9s_{11}+s_3s_5^6s_{11}^2+s_1^2s_3^{10}s_5s_7s_{11}+s_1^2s_3^6s_7s_{13}s_{15}+$$
$$s_1^5s_3^{10}s_9s_{11}+s_1^3s_3^4s_7^2s_{13}^2+s_1^9s_3^3s_5^2s_7s_9s_{11}+s_1^{19}s_3s_5s_{13}s_{15}+s_1^{30}s_3s_{11}^2+s_1^2s_3s_5s_7^3s_{11}s_{13}+s_1^2s_3s_5s_7^3s_9s_{15}+$$
$$s_3s_5^5s_9^3+s_3^3s_5^3s_9^2s_{13}+s_1^2s_3s_5s_7^2s_9s_{11}^2+s_3^4s_5^2s_9s_{11}s_{13}+s_3^7s_5^2s_{11}s_{13}+s_3^7s_5^2s_9s_{15}+s_1^{17}s_3s_{11}^2s_{13}+s_3s_5^2s_9s_{11}^3+$$
$$s_3s_5^2s_9^3s_{15}+s_3s_5s_7^2s_9^2s_{15}+s_3s_5s_7^3s_{13}^2+s_3s_5s_7^2s_{11}^3+s_3s_5s_7^3s_{11}s_{15}+s_3^5s_5^2s_{15}^2+s_3^2s_7^2s_{11}^2s_{13}+s_3^2s_5^2s_9s_{15}^2+$$
$$s_3^2s_7^3s_{13}s_{15}+s_1^5s_3^6s_5^5s_7+s_1^{15}s_5^4s_9s_{11}+s_1^{13}s_3^3s_5^3s_9^2+s_1^9s_3^7s_5^2s_{15}+s_1^3s_3^{14}s_5^2+s_1^{27}s_3^6s_5^2+s_1^2s_3^5s_5^5s_{13}+$$
$$s_1^{35}s_5^4+s_1^{23}s_3^3s_5s_9^2+s_1^{10}s_5^5s_7s_{13}+s_1^2s_3^4s_5s_7^2s_{11}^2+s_1^7s_3^4s_5s_9^2s_{13}+s_1^3s_3^9s_5^2s_{15}+s_1^{25}s_3s_5s_{11}^2+s_1^3s_3^{10}s_5^3s_7+$$
$$s_1^{13}s_5^3s_7s_9s_{11}+s_1^{19}s_3s_7s_{13}^2+s_1^{17}s_3s_7s_9s_{19}+s_1^7s_3^6s_5^3s_{15}+s_1^{19}s_3^7s_{15}+s_1^{15}s_3^7s_{19}+s_1^{13}s_3s_5^3s_7s_{17}+s_1^{16}s_3s_5^2s_7s_{19}+$$
$$s_1^{17}s_5s_7^2s_{19}+s_1^5s_3^3s_{11}s_{15}^2+s_1^5s_3^3s_{11}s_{13}s_{17}+s_1^5s_3^3s_9s_{15}s_{17}+s_1^5s_3^3s_9s_{13}s_{19}+s_1^{29}s_7s_{19}+s_1^{25}s_{11}s_{19}+$$
$$s_1^4s_3^6s_{11}^3+s_1^5s_5^2s_7s_9s_{11}s_{13}+s_1^5s_5^3s_9s_{11}s_{15}+s_1^5s_5^3s_7s_{11}s_{17}+s_1^5s_5^3s_7s_9s_{19}+s_1^{11}s_3^4s_{15}s_{17}+s_1^{11}s_3^4s_{13}s_{19}+$$
$$s_1^4s_5^5s_9s_{17}+s_1^{28}s_5^4s_7+s_1^7s_3^8s_5^2s_7^2+s_1^2s_3^2s_5^4s_7s_9s_{11}+s_1^{18}s_5^6s_7+s_1^{16}s_5^2s_7s_9s_{13}+s_1^{10}s_3^7s_9s_{15}+s_1^{10}s_3^{10}s_{15}+$$
$$s_1^{36}s_3s_5s_{11}+s_1^{32}s_3s_5s_{15}+s_1^{33}s_{11}^2+s_1^{21}s_5^5s_9+s_1^{23}s_3^5s_{17}+s_1^{21}s_3^5s_{19}+s_1^{40}s_{15}+s_1^2s_3^2s_5^5s_{11}^2+s_1^{10}s_3^3s_7^3s_{15}+$$
$$s_1^{12}s_3^2s_7s_{11}s_{19}+s_1^{14}s_3^4s_7s_{11}^2+s_1^{26}s_3^4s_{17}+s_1^{13}s_3^4s_{11}s_{19}+s_1^2s_5^4s_{11}^3+s_1^{10}s_3^4s_{11}^3+s_1^2s_3^2s_5^2s_9s_{13}s_{15}+s_1^2s_3^2s_5^2s_9s_{11}s_{17}+$$
$$s_1^2s_5^5s_9s_{19}+s_1^2s_3^4s_5^3s_7s_{19}+s_1^2s_3s_7^5s_{15}+s_1^{23}s_3s_7s_9s_{13}+s_1^{30}s_7s_9^2+s_1^4s_5^4s_7s_9s_{15}+s_1^4s_3^8s_7^2s_{13}+s_1^5s_3^2s_5^5s_{19}+$$
$$s_1^{17}s_5^2s_{13}s_{15}+s_1^{14}s_5^3s_9s_{17}+s_1^5s_5^2s_7^2s_{13}^2+s_1^5s_3^3s_5^3s_7s_{19}+s_1^{13}s_3^4s_{13}s_{17}+s_1^{17}s_3^7s_{17}+s_1^{11}s_3s_5^3s_7s_{19}+s_1^3s_3^5s_5^4s_{17}+$$
$$s_1^{14}s_3^5s_{11}s_{15}+s_1s_3s_5^5s_7s_{19}+s_1s_3^5s_5^4s_{19}+s_1^{12}s_5^3s_9s_{19}+s_1^4s_5^3s_7^2s_{11}^2+s_1^{11}s_3^6s_{11}s_{15}+s_1^{31}s_7s_{17}+s_1^3s_3^4s_9^2s_{11}^2+$$
$$s_1^{19}s_5s_9s_{11}^2+s_1^9s_3^2s_5^2s_{13}s_{17}+s_1s_3^6s_5^2s_{13}^2+s_1^{18}s_9^2s_{19}+s_1^{18}s_7s_{11}s_{19}+s_1^4s_5^2s_{13}^2s_{15}+s_1^4s_5^2s_{11}^2s_{19}+s_1^{20}s_7s_{11}s_{17}+$$
$$s_1s_7^4s_{13}^2+s_1s_7^5s_{19}+s_1s_3s_7^2s_9^2s_{19}+s_1^{16}s_5^6s_9+s_1^8s_5^8s_7+s_3^6s_5^6s_7+s_3^{10}s_5^5+s_1^4s_5^7s_7s_9+s_1^2s_3^4s_5^5s_7s_9+$$
$$s_1^4s_5^8s_{11}+s_1^4s_3^{10}s_5^2s_{11}+s_1^5s_5^5s_7^2s_{11}+s_1^3s_3^8s_5^2s_7s_{11}+s_3^4s_5^5s_7s_{11}+s_5^7s_9s_{11}+s_1^{14}s_3s_5s_9s_{11}s_{13}+s_3^2s_5^5s_{11}s_{13}+$$
$$s_1^{14}s_5s_7s_9^2s_{11}+s_1^{12}s_5s_7s_9s_{11}^2+s_1^8s_5s_7^2s_{13}s_{15}+s_1^{19}s_5s_7s_9s_{15}+s_1^7s_5^3s_9s_{11}s_{13}+s_1^{11}s_3^2s_5s_7s_{13}^2+s_1^7s_5^3s_7s_9s_{17}+$$
$$s_1^{10}s_5s_9^2s_{11}^2+s_1^3s_3^8s_5^3s_{13}+s_1^{26}s_5s_9s_{15}+s_5^5s_7^3s_9+s_1^2s_5^4s_9s_{11}s_{13}+s_1^{10}s_5^2s_9s_{11}s_{15}+s_1^{10}s_5^2s_7s_{11}s_{17}+$$
$$s_1^4s_5^5s_{13}^2+s_1^{12}s_3s_5s_9s_{13}^2+s_1^6s_5s_9s_{11}^2s_{13}+s_1^6s_5s_9^3s_{17}+s_3^7s_5s_7s_{11}^2+s_1^5s_5s_7^3s_{11}s_{13}+s_1^3s_5s_7^3s_{13}^2+s_5^3s_9^2s_{11}^2+$$
$$s_5s_7^2s_9^4+s_5^4s_9s_{11}s_{15}+s_5s_7^3s_9^2s_{11}+s_5s_7^4s_{11}^2+s_5s_7^5s_{15}+s_3^2s_5s_{11}^4+s_3^2s_5s_9^2s_{13}^2+s_5^5s_{15}^2+s_1^{12}s_3^3s_5s_7s_{11}^2+$$
$$s_1^{20}s_3s_5^2s_7s_{15}+s_1^{14}s_3s_7s_9s_{11}^2+s_1^{13}s_3s_5^2s_7s_{11}^2+s_3^7s_7s_9^3+s_3^{10}s_7s_9^2+s_3^{11}s_{11}^2+s_3^{14}s_{13}+s_1^{24}s_3^4s_{19}+$$
$$s_1^{12}s_3^8s_{19}+s_3^{10}s_5s_7s_{13}+s_3^2s_5^3s_7^3s_{13}+s_1^2s_3s_5^5s_7^2s_{11}+s_1^8s_3s_{11}^4+s_1^4s_3^2s_5^2s_9s_{13}^2+s_1^2s_3^3s_5s_{11}s_{13}s_{15}+s_1^2s_3^3s_5s_9s_{11}s_{19}+$$
$$s_1^9s_3^3s_5^2s_9^3+s_1^3s_3^5s_5^2s_9^3+s_1^7s_3^2s_5^2s_{15}s_{17}+s_1^7s_3^2s_5^2s_{13}s_{19}+s_1^{18}s_3^5s_{11}^2+s_1s_3^{10}s_{11}s_{13}+s_3^5s_9^3s_{13}+s_1^{10}s_3s_9s_{11}^3+$$
$$s_1^2s_3s_5^2s_7s_9s_{11}s_{13}+s_1^2s_3s_5^3s_7s_9s_{19}+s_1^{12}s_3s_7s_{11}^3+s_1^6s_3s_7s_9s_{13}s_{17}+s_1^6s_3s_7s_{11}^2s_{17}+s_1s_3^7s_9s_{11}s_{13}+$$
$$s_1^3s_3s_7^2s_9^2s_{17}+s_3s_5^2s_7s_9s_{11}s_{15}+s_3s_5^2s_7s_9s_{13}^2+s_3^3s_{11}^3s_{13}+s_3^3s_7s_{13}^3+s_3^3s_9^3s_{19}+s_3^3s_7s_9s_{15}^2+s_3s_5^3s_9^2s_{19}+$$
$$s_3^2s_5^2s_{11}s_{13}s_{15}+s_3^2s_5^2s_9s_{11}s_{19}+s_1^3s_3^8s_7^4+s_1^7s_3^4s_7^3s_{15}+s_1^5s_3^7s_7s_9s_{13}+s_1^2s_3^8s_7s_{11}^2+s_1^{20}s_7s_{13}s_{15}+s_1^5s_5^6s_9s_{11}+$$
$$s_1^5s_3^7s_5s_9s_{15}+s_1^5s_3^{10}s_5s_{15}+s_1^{11}s_5^5s_{19}+s_1s_5^7s_{19}+s_1^{15}s_9^3s_{13}+s_1^7s_3^2s_7s_9s_{11}s_{15}+s_1^{19}s_3s_7s_9s_{17}+s_1^3s_3^7s_7s_9s_{15}+$$
$$s_1^7s_3^2s_7s_9s_{13}^2+s_1^{15}s_7^3s_{19}+s_1^{15}s_3^2s_{17}^2+s_1^{15}s_3^2s_{15}s_{19}+s_1^9s_{11}^3s_{13}+s_1^9s_9s_{11}s_{13}^2+s_1^9s_7s_{11}s_{13}s_{15}+s_1^9s_7s_{11}^2s_{17}+$$

$$s_1^9 s_9^2 s_{13} s_{15} + s_1^9 s_9^3 s_{19} + s_1^9 s_7 s_9 s_{13} s_{17} + s_1^9 s_7 s_9 s_{11} s_{19} + s_1^9 s_5 s_7 s_{17}^2 + s_1^9 s_5 s_7 s_{15} s_{19} + s_1^9 s_3 s_{13} s_{15}^2 +$$
$$s_1^9 s_3 s_{11} s_{15} s_{17} + s_1^9 s_3 s_{13}^2 s_{17} + s_1^9 s_3 s_{11} s_{13} s_{19} + s_1^9 s_3 s_9 s_{17}^2 + s_1^9 s_3 s_9 s_{15} s_{19} + s_1^{16} s_7 s_{15} s_{17} + s_1^{16} s_7 s_{13} s_{19} +$$
$$s_1^3 s_3^3 s_{13} s_{15}^2 + s_1^3 s_3^3 s_{11} s_{15} s_{17} + s_1^3 s_3^3 s_{13}^2 s_{17} + s_1^3 s_3^3 s_{11} s_{13} s_{19} + s_1^3 s_3^3 s_9 s_{17}^2 + s_1^3 s_3^3 s_9 s_{15} s_{19} + s_1^2 s_3^{10} s_5 s_9^2 +$$
$$s_1^{14} s_5 s_7^3 s_{15} + s_1^2 s_3 s_5^6 s_7 s_{13} + s_1^{18} s_3 s_5^2 s_7 s_{17} + s_1^4 s_5^4 s_7 s_{11} s_{13} + s_1^{14} s_3^2 s_7 s_{11} s_{17} + s_1^{16} s_9 s_{13} s_{17} + s_1^8 s_5^2 s_7 s_{11} s_{19} +$$
$$s_1^2 s_3^8 s_5 s_{11} s_{13} + s_1^{14} s_{11} s_{15}^2 + s_1^{14} s_9 s_{15} s_{17} + s_1^{14} s_{11} s_{13} s_{17} + s_1^{14} s_9 s_{13} s_{19} + s_1^2 s_3 s_5^2 s_7^2 s_{11} s_{15} + s_1^2 s_3^7 s_{15} s_{17} +$$
$$s_1^2 s_3^7 s_{13} s_{19} + s_1^9 s_5^4 s_7 s_{19} + s_1^{11} s_7 s_9^2 s_{19} + s_1^{13} s_5 s_9^2 s_{19} + s_1^7 s_9 s_{13}^3 + s_1^7 s_9^2 s_{13} s_{17} + s_1^7 s_7 s_{13}^2 s_{15} + s_1^7 s_9 s_{11}^2 s_{17} +$$
$$s_1^7 s_7 s_{11}^2 s_{19} + s_1^7 s_7 s_{11} s_{15}^2 + s_1^7 s_7 s_9 s_{15} s_{17} + s_1^7 s_7 s_{11} s_{13} s_{17} + s_1^7 s_7 s_9 s_{13} s_{19} + s_1^{17} s_3^2 s_{15} s_{17} + s_1^{17} s_3^2 s_{13} s_{19} +$$
$$s_1^9 s_3^4 s_{17}^2 + s_1^9 s_3^4 s_{15} s_{19} + s_1 s_7^3 s_9^2 s_{15} + s_1^{16} s_5^2 s_7^2 s_{15} + s_5^6 s_7 s_9^2 + s_1^8 s_3^4 s_5^2 s_7 s_9^2 + s_1^4 s_3^2 s_5 s_7 s_{11}^3 + s_1^8 s_3^3 s_5 s_7 s_{13}^2 +$$
$$s_1^6 s_7 s_9 s_{11}^3 + s_1^6 s_7^3 s_{13} s_{15} + s_1^{13} s_7 s_{11}^2 s_{13} + s_1^5 s_3^2 s_7 s_{11}^2 s_{15} + s_1^{13} s_7 s_9^2 s_{17} + s_1^3 s_5^2 s_7 s_9 s_{11} s_{15} + s_1^{11} s_7 s_{11} s_{13}^2 +$$
$$s_1^3 s_5^2 s_7 s_9 s_{13}^2 + s_1^{11} s_7 s_9 s_{13} s_{15} + s_1^{11} s_7 s_9 s_{11} s_{17} + s_1^3 s_3^2 s_7 s_{13}^3 + s_1^3 s_3^2 s_7 s_9 s_{15}^2 + s_5^2 s_7 s_9^3 s_{11} + s_5^2 s_7^3 s_9 s_{15} +$$
$$s_3^2 s_7 s_9 s_{11}^3 + s_5^3 s_7^2 s_{11} s_{15} + s_1^4 s_3^2 s_5 s_9^3 s_{13} + s_1^{12} s_5^2 s_{11}^3 + s_1^2 s_5^4 s_7^2 s_{19} + s_1^8 s_3^3 s_5 s_{11}^3 + s_3^8 s_5 s_{13}^2 + s_1^6 s_5^2 s_{11}^2 s_{17} +$$
$$s_1^2 s_3 s_5^3 s_9^2 s_{17} + s_1^{11} s_5 s_{13}^3 + s_1^3 s_5^3 s_7 s_{11} s_{19} + s_1^{11} s_5 s_7 s_{15} s_{17} + s_1^{11} s_5 s_7 s_{13} s_{19} + s_5^3 s_7^3 s_{19} + s_3^4 s_5^2 s_7 s_{11} s_{15} +$$
$$s_1^4 s_3^3 s_7 s_9 s_{11} s_{15} + s_1^2 s_3^3 s_9 s_{11}^2 s_{13} + s_1^2 s_3^3 s_9^3 s_{17} + s_3 s_5^6 s_7 s_{15} + s_1^4 s_3^3 s_7 s_9 s_{13}^2 + s_1^{12} s_3 s_7^3 s_{19} + s_3^4 s_5^2 s_{11}^3 +$$
$$s_3^3 s_5^4 s_{11} s_{15} + s_3^7 s_{17}^2 + s_3^7 s_{15} s_{19} + s_1^2 s_3^4 s_{13}^2 s_{15} + s_1^2 s_3^4 s_{11}^2 s_{19} + s_1 s_3^5 s_{11} s_{13} s_{15} + s_1 s_3^5 s_9 s_{11} s_{19} + s_1^9 s_3^2 s_9^2 s_{11}^2 +$$
$$s_1 s_3^4 s_9^2 s_{11} s_{13} + s_1^9 s_3 s_5^2 s_9^2 s_{15} + s_1^{13} s_3 s_9 s_{13} s_{17} + s_1^{11} s_9 s_{11}^2 s_{13} + s_1^{11} s_9^3 s_{17} + s_1 s_3^4 s_7^2 s_9 s_{19} + s_1^{13} s_5 s_7 s_{13} s_{17} +$$
$$s_1^5 s_3^2 s_5^2 s_{17}^2 + s_1^5 s_3^2 s_5^2 s_{15} s_{19} + s_1^{11} s_3 s_{11} s_{13} s_{17} + s_1^{11} s_3 s_{11} s_{15}^2 + s_1^{11} s_3 s_9 s_{15} s_{17} + s_1^{11} s_3 s_9 s_{13} s_{19} + s_1^2 s_5^4 s_7 s_{13}^2 +$$
$$s_1^{14} s_7 s_{17}^2 + s_1^{14} s_7 s_{15} s_{19} + s_1^{12} s_{13} s_{15}^2 + s_1^{12} s_{13}^2 s_{17} + s_1^{12} s_{11} s_{15} s_{17} + s_1^{12} s_{11} s_{13} s_{19} + s_1^{12} s_9 s_{17}^2 + s_1^{12} s_9 s_{15} s_{19} +$$
$$s_1 s_3^4 s_9 s_{11}^3 + s_5^2 s_9^5 + s_5^2 s_7^2 s_9 s_{11}^2 + s_1^4 s_7 s_9 s_{11}^2 s_{13} + s_1^4 s_7 s_9^3 s_{17} + s_1^4 s_3 s_9 s_{13}^3 + s_1^4 s_3 s_9 s_{11}^2 s_{17} + s_1^4 s_3 s_9^2 s_{13} s_{17} +$$
$$s_1^2 s_9 s_{11}^4 + s_1^2 s_9^2 s_{11}^2 s_{13} + s_1^2 s_9^3 s_{13}^2 + s_1^2 s_9^4 s_{17} + s_1^4 s_7 s_{11}^4 + s_1^4 s_3 s_7 s_{13}^2 s_{15} + s_1^4 s_3 s_7 s_{11} s_{15}^2 + s_1^4 s_3 s_7 s_{11} s_{13} s_{17} +$$
$$s_1^4 s_3 s_7 s_{11}^2 s_{19} + s_1^4 s_3 s_7 s_9 s_{15} s_{17} + s_1^4 s_3 s_7 s_9 s_{13} s_{19} + s_1^2 s_7 s_{11}^3 s_{13} + s_1^2 s_7 s_9 s_{11}^2 s_{15} + s_1^2 s_7 s_9^2 s_{11} s_{17} + s_1^2 s_7 s_9^3 s_{19} +$$
$$s_1^2 s_7^2 s_{11}^2 s_{17} + s_1^2 s_7^3 s_{15} s_{17} + s_1^2 s_7^3 s_{13} s_{19} + s_1^4 s_5 s_{11}^3 s_{13} + s_1^4 s_5 s_9^3 s_{19} + s_1^2 s_3 s_{11} s_{13}^3 + s_1^2 s_3 s_9 s_{13}^2 s_{15} + s_1^2 s_3 s_7 s_{13} s_{15}^2 +$$
$$s_1^2 s_3 s_7 s_{13}^2 s_{17} + s_1^2 s_3 s_{11}^3 s_{17} + s_1^2 s_3 s_9 s_{11}^2 s_{19} + s_1^2 s_3 s_7 s_{11} s_{15} s_{17} + s_1^2 s_3 s_7 s_{11} s_{13} s_{19} + s_1^2 s_3 s_9^2 s_{15} s_{17} +$$
$$s_1^2 s_3 s_9^2 s_{13} s_{19} + s_1^2 s_3 s_7 s_9 s_{17}^2 + s_1^2 s_3 s_7 s_9 s_{15} s_{19} + s_1^5 s_{11} s_{13}^3 + s_1^5 s_{11}^3 s_{17} + s_1^5 s_9 s_{13}^2 s_{15} + s_1^5 s_9 s_{11}^2 s_{19} +$$
$$s_1^5 s_9^2 s_{15} s_{17} + s_1^5 s_9^2 s_{13} s_{19} + s_1^5 s_7 s_{13} s_{15}^2 + s_1^5 s_7 s_{11} s_{15} s_{17} + s_1^5 s_7 s_{13}^2 s_{17} + s_1^5 s_7 s_{11} s_{13} s_{19} + s_1^5 s_7 s_9 s_{17}^2 +$$
$$s_1^5 s_7 s_9 s_{15} s_{19} + s_1^3 s_{13}^4 + s_1^3 s_{11} s_{13}^2 s_{15} + s_1^3 s_{11}^2 s_{15}^2 + s_1^3 s_{11}^3 s_{19} + s_1^3 s_9^2 s_{17}^2 + s_1^3 s_9^2 s_{15} s_{19} + s_1^3 s_7 s_{15}^3 + s_1^3 s_7 s_{13}^2 s_{19} +$$
$$s_1^3 s_7 s_{11} s_{17}^2 + s_1^3 s_7 s_{11} s_{15} s_{19} + s_{11}^5 + s_9^2 s_{11} s_{13}^2 + s_9^2 s_{11}^2 s_{15} + s_9^4 s_{19} + s_7 s_9^2 s_{15}^2 + s_7 s_{11}^2 s_{13}^2 + s_7^2 s_{13}^2 s_{15} +$$
$$s_7 s_{11}^3 s_{15} + s_7 s_9^2 s_{11} s_{19} + s_7^2 s_{11}^2 s_{19} + s_7^3 s_{17}^2 + s_7^3 s_{15} s_{19} + s_5^2 s_{15}^3 + s_5^2 s_{13}^2 s_{19} + s_5^2 s_{11} s_{17}^2 + s_5^2 s_{11} s_{15} s_{19} +$$
$$s_3 s_{13}^4 + s_3 s_{11} s_{13}^2 s_{15} + s_3 s_{11}^2 s_{15}^2 + s_3 s_{11}^3 s_{19} + s_3 s_9^2 s_{17}^2 + s_3 s_9^2 s_{15} s_{19} + s_3 s_7 s_{15}^3 + s_3 s_7 s_{13}^2 s_{19} + s_3 s_7 s_{11} s_{17}^2 +$$
$$s_3 s_7 s_{11} s_{15} s_{19} + s_1^7 s_3 s_{19} s_{15} s_{11} + s_3^2 s_{19} s_{15} s_9 s_5 s_1 + s_1^3 s_5 s_{19} s_{15} s_7 s_3^2 + s_1^2 s_5^2 s_{19} s_{15} s_3^3 + s_1^7 s_3 s_{19} s_{13}^2 +$$
$$s_3^2 s_{19} s_{13} s_{11} s_5 s_1 + s_3^2 s_{19} s_{13} s_9 s_5 s_1^3 + s_1^5 s_5 s_{19} s_{13} s_7 s_3^2 + s_1^4 s_5^2 s_{19} s_{13} s_3^3 + s_3^3 s_{19} s_{11}^2 s_1^5 + s_1^7 s_{19} s_{11} s_9^2 +$$
$$s_1 s_5 s_{19} s_{11} s_9 s_7 s_3 + s_1^{11} s_5 s_{19} s_{11} s_9 + s_1^9 s_{19} s_{11} s_7 s_3^3 + s_1^{17} s_3 s_{19} s_{11} s_5 + s_1^5 s_3^2 s_{19} s_9^2 s_7 + s_1 s_5 s_{19} s_9 s_7^3 +$$
$$s_1^9 s_5 s_{19} s_9 s_7 s_3^2 + s_1^{14} s_{19} s_9 s_5^2 s_3 + s_1^{27} s_{19} s_9 + s_1 s_5 s_{19} s_7^3 s_3^3 + s_1^{33} s_3 s_{19} + s_1^7 s_3 s_{17}^2 s_{11} + s_3^2 s_{17}^2 s_9 s_5 s_1 +$$
$$s_1^3 s_5 s_{17}^2 s_7 s_3^2 + s_1^2 s_5^2 s_{17}^2 s_3^3 + s_3^2 s_{17} s_{15} s_{11} s_5 s_1 + s_3^2 s_{17} s_{15} s_9 s_5 s_1^3 + s_1^5 s_5 s_{17} s_{15} s_7 s_3^2 + s_1^4 s_5^2 s_{17} s_{15} s_3^3 +$$
$$s_3^2 s_{17} s_{13}^2 s_5 s_1 + s_3^2 s_{17} s_{13} s_{11} s_5 s_1^3 + s_3^2 s_{17} s_{13} s_9 s_5 s_1^5 + s_1^7 s_5 s_{17} s_{13} s_7 s_3^2 + s_1^6 s_5^2 s_{17} s_{13} s_3^3 + s_3^3 s_{17} s_{11}^2 s_1^7 +$$
$$s_1^9 s_{17} s_{11} s_9^2 + s_1^3 s_5 s_{17} s_{11} s_9 s_7 s_3 + s_1^{13} s_5 s_{17} s_{11} s_9 + s_1^{11} s_{17} s_{11} s_7 s_3^3 + s_1^{19} s_3 s_{17} s_{11} s_5 + s_1^7 s_3^2 s_{17} s_9^2 s_7 +$$
$$s_1^3 s_5 s_{17} s_9 s_7^3 + s_1^{11} s_5 s_{17} s_9 s_7 s_3^2 + s_1^{16} s_{17} s_9 s_5^2 s_3 + s_1^{29} s_{17} s_9 + s_1^3 s_5 s_{17} s_7^3 s_3^3 + s_1^{35} s_3 s_{17} + s_1^7 s_3 s_{15}^3 + s_3^2 s_{15}^2 s_{13} s_5 s_1 +$$
$$s_3^2 s_{15}^2 s_{11} s_5 s_1^3 + s_1^4 s_3 s_{15}^2 s_9^2 + s_1^7 s_3^3 s_{15}^2 s_9 + s_3^2 s_{15}^2 s_7^2 s_5 + s_1^7 s_3^6 s_{15}^2 + s_3^3 s_{15} s_{13}^2 s_1^5 + s_1 s_5 s_{15} s_{13} s_{11} s_7 s_3 +$$
$$s_1^{11} s_5 s_{15} s_{13} s_{11} + s_1 s_5 s_{15} s_{13} s_9^2 s_3 + s_1^3 s_5 s_{15} s_{13} s_9 s_7 s_3 + s_1^{13} s_5 s_{15} s_{13} s_9 + s_1^9 s_3^2 s_{15} s_{13} s_7 s_5 + s_1^7 s_{15} s_{11}^3 +$$
$$s_1^9 s_{15} s_{11}^2 s_9 + s_1^3 s_3 s_{15} s_{11}^2 s_7 s_5 + s_1^{11} s_3^3 s_{15} s_{11} s_9 + s_3^2 s_{15} s_{11} s_7^3 s_1^2 + s_3^2 s_{15} s_{11} s_7 s_5^3 s_1 + s_3^6 s_{15} s_{11} s_7 s_1^4 +$$
$$s_3^4 s_{15} s_{11} s_7 s_1^{10} + s_1^8 s_3^2 s_{15} s_{11} s_5^3 + s_3^7 s_{15} s_{11} s_5 s_1^3 + s_1^7 s_{15} s_9^2 s_5^3 + s_1^4 s_3^6 s_{15} s_9^2 + s_3^4 s_{15} s_9 s_7^2 s_5 + s_1^6 s_{15} s_9 s_7 s_5^3 s_3 +$$
$$s_3^4 s_{15} s_9 s_7 s_5 s_1^7 + s_1^4 s_3^4 s_{15} s_9 s_5^3 + s_1^{13} s_3 s_{15} s_9 s_5^3 + s_3^2 s_{15} s_7^3 s_5^2 s_1^3 + s_1^3 s_5^6 s_{15} s_7 + s_3^5 s_{15} s_7 s_5^3 s_1^3 + s_1^{12} s_5^5 s_{15} s_3 +$$
$$s_1^{18} s_3^4 s_{15} s_5^2 + s_1^7 s_{13}^2 s_{11}^2 + s_1 s_5 s_{13}^2 s_{11} s_9 s_3 + s_1^3 s_5 s_{13}^2 s_{11} s_7 s_3 + s_1^8 s_{13}^2 s_9^2 s_3 + s_3^6 s_{13}^2 s_9 s_1^2 + s_3^2 s_{13}^2 s_7^2 s_1^9 +$$
$$s_1^{14} s_{13}^2 s_5^3 + s_1^7 s_3^2 s_{13} s_{11}^2 s_7 + s_1 s_5 s_{13} s_{11} s_9^2 s_7 + s_1 s_5 s_{13} s_{11} s_9 s_7 s_3^3 + s_1^{10} s_5 s_{13} s_{11} s_9 s_7 + s_1^3 s_5^2 s_{13} s_{11} s_9 s_3^3 +$$
$$s_1^8 s_5^4 s_{13} s_{11} s_3 + s_3^2 s_{13} s_{11} s_5^2 s_1^{15} + s_1^{17} s_5 s_{13} s_{11} s_3^3 + s_1^{26} s_5 s_{13} s_{11} + s_1 s_5 s_{13} s_9^4 + s_1^3 s_5 s_{13} s_9^3 s_7 + s_3^8 s_{13} s_9^2 +$$
$$s_1^{19} s_5 s_{13} s_9^2 + s_3^2 s_{13} s_9 s_7 s_5^3 s_1^5 + s_1^{11} s_{13} s_9 s_5^2 s_3^4 + s_1^6 s_{13} s_9 s_3^9 + s_1^{22} s_3^5 s_{13} s_5 + s_1^5 s_3 s_{11}^3 s_9 s_5 + s_1^{19} s_3 s_{11}^3 +$$

$$
\begin{aligned}
&s_1s_5s_{11}^2s_9^3 + s_1^3s_5s_{11}^2s_9^2s_7 + s_1^6s_{11}^2s_9s_5^3s_3 + s_3^2s_{11}^2s_9s_5^2s_1^8 + s_1^3s_3^7s_{11}^2s_9 + s_1^{21}s_{11}^2s_9s_3 + s_1^6s_{11}^2s_7^2s_5^2s_3 + \\
&s_1^{11}s_{11}^2s_7s_5^3 + s_1^{23}s_3s_{11}^2s_7 + s_1^{19}s_3^3s_{11}^2s_5 + s_1^{15}s_{11}^2s_3^6 + s_1^2s_5s_{11}s_9^3s_7s_3 + s_1^{16}s_{11}s_9^2s_5^2 + s_3^8s_{11}s_9^2s_1^2 + s_1^{18}s_5s_{11}s_9^2s_3 + \\
&s_3^7s_{11}s_9s_7s_1^7 + s_1^{17}s_{11}s_9s_5^3s_3 + s_1^3s_3^9s_{11}s_7^2 + s_1^{16}s_{11}s_5^5s_3 + s_1^7s_5^5s_{11}s_3^4 + s_1^{13}s_9^4s_3^2 + s_1^6s_9^3s_7s_5^3 + s_1^4s_3^8s_9^3 + \\
&s_1^{19}s_3^3s_9^3 + s_1^{14}s_3^2s_9^2s_7s_5^2 + s_3^8s_9^2s_7s_1^6 + s_1^{24}s_3^2s_9^2s_7 + s_1^6s_5^5s_9^2s_3^2 + s_1^{17}s_3^5s_9^2s_5 + s_1^{12}s_3^4s_9s_7s_5^3 + s_3^8s_9s_7s_5s_1^{10} + \\
&s_1^{22}s_3^4s_9s_7s_5 + s_1^8s_3^6s_9s_5^4 + s_3^3s_7s_5^4s_1^{19} + s_1^{16}s_3^9s_7s_5 + s_1^{13}s_5^6s_3^4 + s_1^{22}s_3^6s_5^3 + s_1^{38}s_3^4s_5 + s_5^3s_{19}s_{15}s_1^6 + s_5^3s_{19}s_{13}s_1^8 + \\
&s_1^{15}s_{19}s_{11}s_5^2 + s_5^3s_{17}^2s_1^6 + s_5^3s_{17}s_{15}s_1^8 + s_5^3s_{17}s_{13}s_1^{10} + s_1^{17}s_{17}s_{11}s_5^2 + s_1^5s_5^2s_{15}^2s_7s_3 + s_1^{15}s_{15}s_5^5 + s_1^{15}s_{13}^2s_{11}s_3 + \\
&s_1^5s_5^2s_{13}^2s_{11}s_3 + s_1^5s_5^3s_{13}^2s_9 + s_1^{15}s_{13}s_{11}s_7s_3^3 + s_1^5s_5^2s_{13}s_{11}s_7s_3^3 + s_1^2s_7^3s_{13}s_5^2s_3^3 + s_1^{21}s_{13}s_7s_5s_3^3 + s_1^{13}s_3^4s_5^2s_{13}s_7 + \\
&s_1^8s_7^2s_{13}s_5^4 + s_1^2s_7^2s_{13}s_5s_3^7 + s_1^{15}s_{11}^3s_7 + s_1^7s_3s_5s_{11}^3s_7 + s_1^{17}s_{11}s_9^3 + s_1^3s_7^2s_{11}s_9^3 + s_1^{25}s_7^2s_{11}s_5 + s_3^5s_9^2s_5^2s_1^{12} + \\
&s_1^{13}s_7^2s_9^2s_5^2 + s_1^2s_5^4s_{19}s_{11}s_3 + s_1^4s_5^4s_{17}s_{11}s_3 + s_1^3s_3^8s_{15}s_{13} + s_1^3s_3^5s_{15}s_{11}^2 + s_1^3s_3^4s_{15}s_9^2s_7 + s_3^8s_{15}s_7^2s_1^2 + \\
&s_3^3s_{15}s_5^6s_1 + s_1^3s_3^4s_{13}s_{11}^2s_5 + s_1^{11}s_3^5s_{13}s_9s_7 + s_1^9s_3^6s_{13}s_5^3 + s_1s_5^2s_{11}^2s_9s_7s_3^2 + s_1^{21}s_3s_{11}s_5^4 + s_3^2s_9^3s_5^4s_1^2 + \\
&s_1^3s_5s_9s_7^2s_3^8 + s_1^3s_3s_9s_5^8 + s_3^{10}s_9s_5s_1^{11} + s_3^8s_7^3s_1^{10} + s_1^7s_5^6s_{11}s_7 + s_1^9s_3s_5^5s_{11}s_7 + s_3^8s_7^3s_5^2
\end{aligned}
$$

Der 11 bzw. 10-Bit Fall $det(M(11))$ besteht bei ausmultiplizierter Determinante aus der Summe von 2248 Produkten, welche durch Vereinfachungen auf 876 Produkte reduziert werden kann.

Um zu verdeutlichen, wie die Reduktion funktionieren kann, betrachten wir $det(M(5))$, in welcher Ersetzungen durch die Formel $0 = det(M(4))$ vorgenommen werden.

$$
\begin{aligned}
0 &= det(M(4)) = s_1^3s_3 + s_1^6 + s_3^2 + s_5s_1 \\
0 &= det(M(4)) \times s_1s_3 = s_1^4s_3^2 + s_1^7s_3 + s_3^3s_1 + s_1^2s_5s_3 \\
\mathbf{s_1^4s_3^2} &= s_1^7s_3 + s_3^3s_1 + s_1^2s_5s_3 \\
0 &= det(M(4)) \times s_1^4 = s_1^7s_3 + s_1^{10} + s_3^2s_1^4 + s_1^5s_5 \\
\mathbf{s_1^7s_3} &= s_1^{10} + s_3^2s_1^4 + s_1^5s_5 \\
det(M(5)) &= s_1^7s_3 + s_1^{10} + s_1s_3^3 + s_1^2s_3s_5 + s_1^5s_5 + s_1^3s_7 + s_5^2 + s_3s_7 \\
&= (s_1^7s_3 + s_1s_3^3 + s_1^2s_3s_5) + s_1^{10} + s_1^5s_5 + s_1^3s_7 + s_5^2 + s_3s_7 \\
&= \mathbf{s_1^4s_3^2} + s_1^{10} + s_1^5s_5 + s_1^3s_7 + s_5^2 + s_3s_7 \\
&= (s_1^4s_3^2 + s_1^{10} + s_1^5s_5) + s_1^3s_7 + s_5^2 + s_3s_7 \\
&= \mathbf{s_1^7s_3} + s_1^3s_7 + s_5^2 + s_3s_7
\end{aligned}
\tag{3.19}
$$

In der Formel für $det(M(5))$ wird dabei $s_1^7s_3 + s_1s_3^3 + s_1^2s_3s_5$ durch $\mathbf{s_1^4s_3^2}$ und $s_1^4s_3^2 + s_1^{10} + s_1^5s_5$ durch $\mathbf{s_1^7s_3}$ ersetzt.

Erklärung Wird der Ansatz zur Korrektur von 1-Bit Fehlern auf 2-Bit Fehler angewendet um zu überprüfen, ob nicht ein höherer Fehler vorliegt, so steigt die Formellänge exponentiell zu der zu überprüften Fehleranzahl. Eine Erklärung, weshalb die Länge der Determinanten im 2-Bit Fall nicht konstant ist, kann wie folgt gegeben werden. Im Falle eines beliebigen Fehlers berechnen sich die Syndromkomponenten aus der Summe der entsprechenden alpha-Werte der H-Matrix.

Für den 1-Bit Fehlerfall sieht dies wie folgt aus:

$$\begin{aligned} s_1 &= \alpha^i \\ s_3 &= \alpha^{i^3} \\ s_5 &= \alpha^{i^5} \\ s_7 &= \alpha^{i^7} \\ &\dots \end{aligned} \tag{3.21}$$

Der Wert i ist der Index der Position des Fehlers. Die Formeln, die sich für den 1-Bit Fehler ergeben, sind aus diesen Gleichungen leicht erklärbar.

$$\begin{aligned} s_3 &= s_1^3 = \alpha^{i^3} \\ s_5 &= s_1^5 = \alpha^{i^5} \\ s_7 &= s_1^7 = \alpha^{i^7} \\ &\dots \end{aligned} \tag{3.23}$$

Für den 2-Bit Fehlerfall sehen diese wie folgt aus.

$$\begin{aligned} s_1 &= \alpha^i + \alpha^j \\ s_3 &= \alpha^{i^3} + \alpha^{j^3} \\ s_5 &= \alpha^{i^5} + \alpha^{j^5} \\ s_7 &= \alpha^{i^7} + \alpha^{j^7} \\ &\dots \end{aligned} \tag{3.25}$$

In diesem Fall ergibt sich, dass es für einige x eine Differenz zwischen s_x und s_1^x gibt.

$$
\begin{aligned}
s_1^2 &= \alpha^{\mathbf{i2}} + \alpha^{\mathbf{j2}} \\
s_1^3 &= \alpha^{\mathbf{i3}} + \alpha^{\mathbf{j3}} + \alpha^{i^2}\alpha^{j} + \alpha^{i}\alpha^{j^2} \\
s_1^4 &= \alpha^{\mathbf{i4}} + \alpha^{\mathbf{j4}} \\
s_1^5 &= \alpha^{\mathbf{i5}} + \alpha^{\mathbf{j5}} + \alpha^{i^4}\alpha^{j} + \alpha^{i}\alpha^{j^4} \\
s_1^6 &= \alpha^{\mathbf{i6}} + \alpha^{\mathbf{j6}} + \alpha^{i^4}\alpha^{j^2} + \alpha^{i^2}\alpha^{j^4} \\
s_1^7 &= \alpha^{\mathbf{i7}} + \alpha^{\mathbf{j7}} + \alpha^{i^6}\alpha^{j} + \alpha^{i^5}\alpha^{j^2} + \alpha^{i^3}\alpha^{j^4} + \alpha^{i}\alpha^{j^6} + \alpha^{i^4}\alpha^{j^3} + \alpha^{i^2}\alpha^{j^5} \\
s_1^8 &= \alpha^{\mathbf{i8}} + \alpha^{\mathbf{j8}} \\
s_1^9 &= \alpha^{\mathbf{i9}} + \alpha^{\mathbf{j9}} + \alpha^{i^8}\alpha^{j} + \alpha^{i}\alpha^{j^8} \\
s_1^{10} &= \alpha^{\mathbf{i10}} + \alpha^{\mathbf{j10}} + \alpha^{i^8}\alpha^{j^2} + \alpha^{i^2}\alpha^{j^8} \\
s_1^{11} &= \alpha^{\mathbf{i11}} + \alpha^{\mathbf{j11}} + \alpha^{i^{10}}\alpha^{j} + \alpha^{i^9}\alpha^{j^2} + \alpha^{i^3}\alpha^{j^8} + \alpha^{i^8}\alpha^{j^3} + \alpha^{i^2}\alpha^{j^9} + \alpha^{i}\alpha^{j^{10}} \\
s_1^{12} &= \alpha^{\mathbf{i12}} + \alpha^{\mathbf{j12}} + \alpha^{i^8}\alpha^{j^4} + \alpha^{i^4}\alpha^{j^8} \\
s_1^{13} &= \alpha^{\mathbf{i13}} + \alpha^{\mathbf{j13}} + \alpha^{i^{12}}\alpha^{j} + \alpha^{i^4}\alpha^{j^9} + \alpha^{i^5}\alpha^{j^8} + \alpha^{i^8}\alpha^{j^5} + \alpha^{i^9}\alpha^{j^4} + \alpha^{i}\alpha^{j^{12}} \\
&\dots
\end{aligned}
\tag{3.27}
$$

Diese Differenz wird in den Gleichungen für die verschiedenen Fehleranzahlen genutzt, indem sich ergibt, dass höhere Fehlerzahlen andere Werte von Syndromkomponenten ergeben, wodurch die Gleichungen nicht erfüllt werden. Wird ein 2-Bit Fehler angenommen und entsprechend die Syndromkomponenten ersetzt, ist diese Formel allgemeingültig.

$$
\begin{aligned}
0 &= s_1^3 s_7 + s_5^2 + s_3 s_7 + s_1^7 s_3 \\
0 &= (\alpha^{i^3} + \alpha^{j^3} + \alpha^{i^2}\alpha^{j} + \alpha^{i}\alpha^{j^2}) \times (\alpha^{i^7} + \alpha^{j^7}) \\
&\quad + (\alpha^{i^5} + \alpha^{j^5})^2 + (\alpha^{i^3} + \alpha^{j^3}) \times (\alpha^{i^7} + \alpha^{j^7}) \\
&\quad + (\alpha^{i^7} + \alpha^{j^7} + \alpha^{i^6}\alpha^{j} + \alpha^{i^5}\alpha^{j^2} + \alpha^{i^3}\alpha^{j^4} + \alpha^{i}\alpha^{j^6} + \alpha^{i^4}\alpha^{j^3} + \alpha^{i^2}\alpha^{j^5}) \times (\alpha^{i^3} + \alpha^{j^3}) \\
0 &= (\alpha^{i^{10}} + \alpha^{i^7}\alpha^{j^3} + \alpha^{i^9}\alpha^{j} + \alpha^{i^8}\alpha^{j^2}) + (\alpha^{i^3}\alpha^{j^7} + \alpha^{j^{10}} + \alpha^{i^2}\alpha^{j^8} + \alpha^{i}\alpha^{j^9}) \\
&\quad + \alpha^{i^{10}} + \alpha^{j^{10}} + \alpha^{i^{10}} + \alpha^{i^3}\alpha^{j^7} + \alpha^{i^7}\alpha^{j^3} + \alpha^{j^{10}} \\
&\quad + (\alpha^{i^{10}} + \alpha^{i^3}\alpha^{j^7} + \alpha^{i^9}\alpha^{j} + \alpha^{i^8}\alpha^{j^2} + \alpha^{i^6}\alpha^{j^4} + \alpha^{i^4}\alpha^{j^6} + \alpha^{i^7}\alpha^{j^3} + \alpha^{i^5}\alpha^{j^5}) + \\
&\quad + (\alpha^{i^7}\alpha^{j^3} + \alpha^{j^{10}} + \alpha^{i^6}\alpha^{j^4} + \alpha^{i^5}\alpha^{j^5} + \alpha^{i^3}\alpha^{j^7} + \alpha^{i}\alpha^{j^9} + \alpha^{i^4}\alpha^{j^6} + \alpha^{i^2}\alpha^{j^8}) \\
0 &= 4 \times \alpha^{i^{10}} + 4 \times \alpha^{i^7}\alpha^{j^3} + 2 \times \alpha^{i^9}\alpha^{j} + 2 \times \alpha^{i^8}\alpha^{j^2} + 4 \times \alpha^{i^3}\alpha^{j^7} + 2 \times \alpha^{i^2}\alpha^{j^8} + 2 \times \alpha^{i}\alpha^{j^9} \\
&\quad + 4 \times \alpha^{j^{10}} + 2 \times \alpha^{i^6}\alpha^{j^4} + 2 \times \alpha^{i^4}\alpha^{j^6} + 2 \times \alpha^{i^5}\alpha^{j^5} \\
0 &= 0
\end{aligned}
\tag{3.29}
$$

Entsprechend der Aussage der Determinante $det(M(2)) = 0$, gilt dies auch, falls ein 1-Bit Fehler als

$$\begin{aligned} 0 &= s_1^3 s_7 + s_5^2 + s_3 s_7 + s_1^7 s_3 \\ &= \alpha^3\alpha^7 + \alpha^{5^2} + \alpha^3\alpha^7 + \alpha^7\alpha^3 \\ &= \alpha^{10} + \alpha^{10} + \alpha^{10} + \alpha^{10} = 0 \end{aligned} \tag{3.31}$$

Wird ein 3-Bit Fehler eingesetzt, ist diese Gleichung nicht allgemein erfüllt.

$$\begin{aligned} 0 &= s_1^3 s_7 + s_5^2 + s_3 s_7 + s_1^7 s_3 \\ &= (\alpha^i + \alpha^j + \alpha^k)^3 \times (\alpha^{i^7} + \alpha^{j^7} + \alpha^{k^7}) \\ &+ (\alpha^{i^5} + \alpha^{j^5} + \alpha^{k^5})^2 \\ &+ (\alpha^{i^3} + \alpha^{j^3} + \alpha^{k^3}) \times (\alpha^{i^7} + \alpha^{j^7} + \alpha^{k^7}) \\ &+ (\alpha^i + \alpha^j + \alpha^k)^7 \times (\alpha^{i^3} + \alpha^{j^3} + \alpha^{k^3}) \\ &\overset{?}{\neq} 0 \end{aligned} \tag{3.33}$$

Auch für Fehler mit höher Bitzahl ist diese Gleichung nicht allgemein gültig.

Insgesamt sind die Formeln, um einen 2-Bit Fehler von größeren Fehlern zu unterscheiden wesentlich komplizierter als jene zur Unterscheidung von 1-Bit Fehlern.

Der vorgestellte Ansatz zur 1-Bit Fehlererkennung ist somit nur begrenzt für eine 2-Bit Fehlererkennung einsetzbar. Die Frage ist nun, ob eine einfachere Erkennung im 2-Bit Fehlerfall möglich ist.

3.2.2 Generelle 2-Bit Fehlererkennung und -korrektur

Im Folgenden wird die 2-Bit Fehlerkorrektur behandelt. Bei diesem Ansatz werden unter der spekulativen Annahme, dass ein 2-Bit Fehler aufgetreten ist, aus den Syndromkomponenten s_1 und s_3 die Fehlerpositionen i und j eines 2-Bit Fehlers bestimmt. Nach der spekulativen Berechnung der Fehlerstellen werden die Syndromkomponenten der potentiellen Fehlerstellen bestimmt und mit den vorhandenen höheren Syndromkomponenten verglichen. Die Vorgehensweise ist die folgende:

1. Bestimmung der Syndromkomponenten.

2. Ist das Syndrom gleich 0, so wurde kein Fehler festgestellt und der Vorgang ist beendet.

3. Gilt $s_1^3 = s_3$, so kann ein 1-Bit Fehler, aber kein 2-Bit Fehler aufgetreten sein und es wird mit den höheren Syndromkomponenten überprüft, ob es sich um einen 1-Bit Fehler handelt. Der Vorgang wird somit mit einer 1-Bit Korrektur und der Überprüfung, ob ein höherer Fehler vorliegt, beendet.

4. Spekulative Berechnung der Fehlerpositionen i und j unter der Annahme, dass ein 2-Bit

Fehler aufgetreten ist, aus den Syndromkomponenten s_1 und s_3. Es muss gelten:

$$\begin{aligned} i &\neq j \\ \alpha^i + \alpha^j &= s_1 \\ \alpha^{3i} + \alpha^{3j} &= s_3 \end{aligned} \tag{3.35}$$

In einigen Fällen ist diese Berechnung nicht erfolgreich, z.B. falls durch null dividiert werden muss, kein Tabelleneintrag der Konstante (Erklärung siehe Abschnitt 2.3.2.2) vorhanden ist oder i und j gleich sind. Ist die Berechnung nicht erfolgreich, muss ein anderer Fehler als ein 2-Bit Fehler aufgetreten sein und der Vorgang ist beendet. Ansonsten wird vorläufig angenommen, dass ein 2-Bit Fehler an Positionen i und j aufgetreten ist.

5. Berechnung der weiteren Syndromkomponenten $s_{5,2}$, $s_{7,2}$, $s_{9,2}$, ... unter der Annahme, dass ein 2-Bit Fehler mit den spekulativ berechneten Fehlerpositionen i und j aufgetreten ist. Die Syndromkomponenten s_1 und s_3 wurden bereits zur Bestimmung von i und j verwendet und müssen somit nicht erneut aus i und j berechnet werden. Es wird also

$$\begin{aligned} s_{5,2} &= \alpha^{5i} + \alpha^{5j} \\ s_{7,2} &= \alpha^{7i} + \alpha^{7j} \\ s_{9,2} &= \alpha^{9i} + \alpha^{9j} \\ &\dots \end{aligned} \tag{3.37}$$

aus den zuvor bestimmten Werten i, j berechnet

6. Die unter Annahme eines 2-Bit Fehlers berechneten Syndromkomponenten $s_{5,2}, s_{7,2}, s_{9,2}, \dots$ werden mit den tatsächlich aufgetretenen Syndromkomponenten s_5, s_7, s_9, ... verglichen.

$$\begin{aligned} s_{5,2} &\stackrel{?}{=} s_5 \\ s_{7,2} &\stackrel{?}{=} s_7 \\ s_{9,2} &\stackrel{?}{=} s_9 \\ &\dots \end{aligned} \tag{3.39}$$

Im Falle eines 2-Bit Fehlers müssen die Syndromkomponenten des berechneten 2-Bit Fehlers den tatsächlich festgestellten Syndromkomponenten entsprechen. Andernfalls muss ein Fehler mit mehr als 2-Bit aufgetreten sein.

Zur Berechnung der Fehlerpositionen kann prinzipiell ein beliebiges Verfahren verwendet werden. Wir verweisen hier auf Abschnitt 2.3.2.2, in welchem wir das Verfahren nach Okano-Imai [OI87] vorgestellt haben. Ebenso ist ein Verfahren anwendbar, bei welchem nach einem Suchverfahren (nach Chien[Chi64]) die Nullstellen des Lokatorpolynoms zweitens Grades als Fehlerstellen bestimmt.

Weitere Aussagen Betrachten wir zunächst einen binären BCH Code mit Codeabstand $d = 11$. Dieser Code hat die Syndromkomponenten s_1, s_3, s_5, s_7, s_9. Nehmen wir an, es wurde ein durch

den Code codiertes Wort übertragen.

Falls möglich, sei aus s_1 und s_3 entsprechend der Gleichungen $s_1 = \alpha^i + \alpha^j$ und $s_3 = \alpha^{3i} + \alpha^{3j}$ durch ein beliebiges Verfahren die Fehlerstellen i und j berechnet worden. Die folgenden Gleichungen sind in diesem Fall zu prüfen:

$$\begin{aligned} \alpha^{5i} + \alpha^{5j} &\stackrel{?}{=} s_5 \\ \alpha^{7i} + \alpha^{7j} &\stackrel{?}{=} s_7 \\ \alpha^{9i} + \alpha^{9j} &\stackrel{?}{=} s_9 \end{aligned} \tag{3.41}$$

Es können die folgenden Szenarien aufgetreten sein:

- Ist das Syndrom gleich 0, ist kein Fehler oder ein mindestens 11-Bit Fehler aufgetreten.
- Ansonsten: Scheitert die Berechnung von i und j, kann kein 2-Bit Fehler aufgetreten sein.
- Ansonsten: Gelten alle Gleichungen, so ist ein 2-Bit Fehler oder ein mindestens 9-Bit Fehler aufgetreten (Code korrigiert maximal 2-Bit und erkennt zusätzlich bis zu 8-Bit Fehler).
- Ansonsten: Gilt **nicht** $\alpha^{5i} + \alpha^{5j} = s_5$, so kann kein 2-Bit Fehler aufgetreten sein und der Fehler muss ein mindestens 3-Bit Fehler sein.
- Ansonsten: Gilt $\alpha^{5i} + \alpha^{5j} = s_5$ aber **nicht** $\alpha^{7i} + \alpha^{7j} = s_7$, so kann kein 2-Bit Fehler aufgetreten sein und der Fehler muss ein mindestens 5-Bit Fehler sein.
- Ansonsten: Gilt lediglich **nicht** $\alpha^{9i} + \alpha^{9j} = s_9$, so kann kein 2-Bit Fehler aufgetreten sein und der Fehler muss ein mindestens 7-Bit Fehler sein.

Generell gilt: Ein BCH-Code mit Syndromkomponenten bis s_p besitzt Codeabstand $d = p + 2$ und kann generell bis zu $(d-1)$-Bit Fehler erkennen. Wird eine 2-Bit Fehlerkorrektur durchgeführt, können nicht alle $(d-1)$-Bit Fehler und $(d-2)$-Bit Fehler erkannt werden. Bei einer 2-Bit Fehlerkorrektur mit zusätzlicher Erkennung können somit alle bis zu $(d-3)$-Bit Fehler erkannt werden.

Gilt also $s_5 = s_{5,2}, s_7 = s_{7,2}, \ldots, s_p = s_{p,2}$, so liegt kein 1-Bit Fehler, kein 3-Bit Fehler, ... und kein bis zu $(p-1)$-Bit Fehler vor. Es ist weiterhin möglich, dass ein o-Bit Fehler oder ein höherer Fehler vorliegt. Bei einem unerkannten Fehler müsste bei einem Codeabstand von d ein mindestens $(d-2)$-Bit Fehler aufgetreten sein.

Unterscheiden sich s_5, s_7, ..., s_n von $s_{5,2}$, $s_{7,2}$, ..., $s_{p,2}$, so liegt kein 2-Bit-Fehler vor. Im Falle eines Unterschieds kann zusätzlich folgendes ausgesagt werden: Gelten $s_3 = s_{3,2}$, $s_5 = s_{5,2}$, $s_7 = s_{7,2}$, ..., $s_x = s_{x,2}$ mit $x < p$, x ungerade, aber $s_{x+2} \neq s_{x+2,2}$, dann kann kein 2-Bit Fehler aufgetreten sein und der aufgetretene Fehler muss mindestens ein x-Bit Fehler sein.

3.2.2.1 Schaltung der 2-Bit Korrektur

Die Überprüfung auf einen höheren Fehler wird durch die folgenden beiden Abbildungen veranschaulicht. Abbildung 3.2 zeigt die 2-Bit Korrektur ohne Überprüfung auf höhere Fehler. Ab-

bildung 3.3 veranschaulicht das beschriebene Verfahren mit enthaltener Überprüfung auf höhere Fehler.

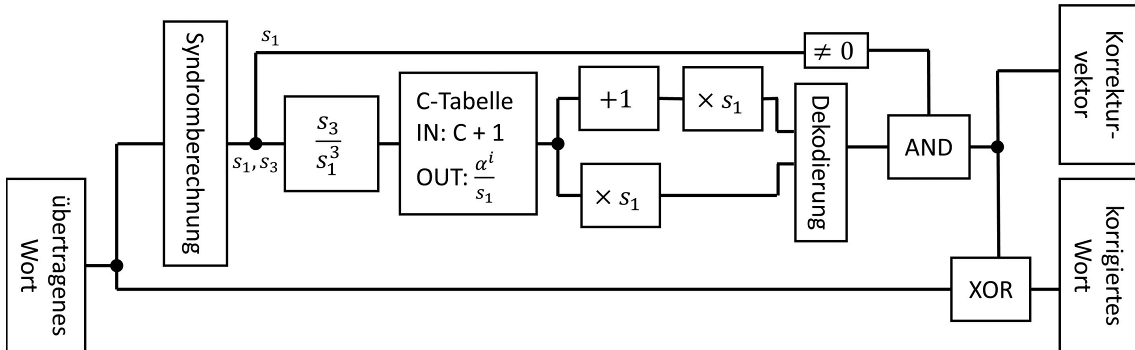

Abbildung 3.2: Schematische Schaltung der 2-Bit Dekodierung ohne Überprüfung. Die C-Tabelle gibt für den Fall eines Wertes, der keinem 2-Bit Fehler entspricht, den Nullvektor aus.

Das Schema 3.2 basiert auf einer Transformation des Lokatorpolynoms, wie in Abschnitt 2.3.2.2 beschrieben. Dabei werden durch eine Tabelle aus transformierten Syndromkomponenten die Nullstellen des Lokatorpolynoms bzw. die Fehlerpositionen abgeleitet. Diese Fehlerpositionen werden anschließend im Wort durch Invertieren (**XOR** jedes Bits mit 1) korrigiert.

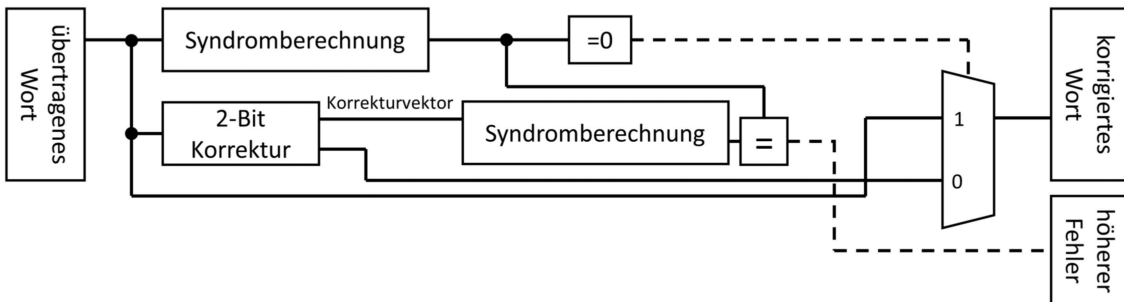

Abbildung 3.3: Schematische Schaltung der 2-Bit Dekodierung mit zusätzlicher Überprüfung auf höhere Fehler. Es wird angenommen, dass der Fehlervektor nach der Dekodierung der Fehler als Ausgabe der Korrektur verfügbar ist.

Das Schema 3.3 zeigt dazu die zusätzliche Schaltung, welche für eine zusätzliche Erkennung höherer Fehler eingesetzt werden kann. Während die Schaltung zur Korrektur von 2-Bit Fehler bekannt ist, stellt die Ergänzung um eine beliebige Fehlererkennung einen wichtigen Beitrag dieser Arbeit dar.

Die Korrektur für 2-Bit Fehler und zusätzliche Überprüfung auf höhere Fehler besteht hier aus den folgenden Schritten:

1. **Syndromberechnung**
   Aus dem übertragenen Wort werden die Syndromkomponenten bestimmt. Die Anzahl der Syndromkomponenten ist vom gewählten BCH Code abhängig, zur Korrektur von 2-Bit Fehlern werden 2 Syndromkomponenten (s_1 und s_3) benötigt. Die weiteren Syndromko-

monenten werden für die Erkennung höherer Fehler erforderlich (z.B. s_5, s_7, s_9). Die Berechnung der Syndromkomponenten des übertragenen Wortes muss lediglich einmal durchgeführt werden, eine zweite Schaltung innerhalb der 2-Bit Korrektur ist nicht notwendig. Im Falle eines 1-Bit oder 2-Bit Fehlers müssen $S \neq 0$ und spezifischer $s_1 \neq 0$ gelten. Im Fall, dass $s_1 = 0$, wird keine Korrektur durchgeführt und der Korrekturvektor wird auf den Nullvektor gesetzt.

2. **Berechnung** $C + 1 = \frac{s_3}{s_1^3}$
   Aus den Syndromkomponenten s_1 und s_3 wird ein Wert $C + 1$ berechnet, aus welchem beide Fehlerstellen bestimmt werden können. Dieser Wert $C + 1$ wird zur Bestimmung der Nullstellen des Lokatorpolynoms verwendet. Eine Berechnung des Wertes ist nicht möglich, falls $s_1^3 = 0$. In diesem Fehlerfall kann ein Default Wert (0) und der Fehlerstatus auf einem weiteren Kanal ausgegeben werden, damit die Korrektur abgebrochen werden kann (nicht in der schematischen Abbildung enthalten).
3. **Tabelle** $C + 1$ **zu** $\frac{\alpha^i}{s_1}$
   Eine Tabelle wird verwendet, um die transformierten Syndromkomponenten einer transformierten Fehlerstelle zuzuordnen. Diese Tabelle erhält $C + 1$ als Eingabe und gibt die erste Fehlerstelle in Form $\frac{\alpha^i}{s_1}$ aus.
4. **Multiplikation mit** s_1
   Durch die Multiplikation von $\frac{\alpha^i}{s_1}$ mit s_1 wird $\frac{\alpha^i}{s_1} \times s_1 = \alpha^i$ bestimmt. In dem Fall, dass der Fehler einem 1-Bit Fehler, aber keinem 2-Bit Fehler entspricht, gilt $s_1^3 = s_3$, somit $C + 1 = 1$. Für diesen Wert definieren wir das Ergebnis der C-Tabelle als den Wert 0. Durch die Folgeoperationen wird diese Ausgabe 0 durch Aufaddieren von s_1 die Fehlerstelle eines 1-Bit Fehlers und korrigiert diesen stattdessen.
5. **Berechnen der zweiten Fehlerstelle** $\alpha^j = \alpha^i + s_1$
   Die zweite Fehlerstelle α^j wird durch Addition von s_1 auf α^i oder durch Addition $\frac{\alpha^j}{s_1} = \frac{\alpha^i}{s_1} + 1$ mit anschließender Multiplikation $\frac{\alpha^j}{s_1} \times s_1 = \alpha^j$ berechnet.
6. **Dekodieren und Korrigieren**
   Bei dem Dekodieren wird aus α^i und α^j der Korrekturvektor bestimmt. Dieser Korrekturvektor weißt den Wert 1 an den Positionen i, j auf, Alle anderen Komponenten sind 0. Eine Korrektur findet durch ein **XOR** des Korrekturvektors mit dem übertragenen Wort statt. Ist $s_1 = 0$, so kann kein 2-Bit oder 1-Bit Fehler aufgetreten sein, daher wird in diesem Fall keine Korrektur durchgeführt.
7. **Überprüfung: Vergleich mit weiteren Syndromkomponenten**
   Nach der spekulativen 2-Bit Korrektur können entweder die Werte $\alpha^{5i} + \alpha^{5j}$, $\alpha^{7i} + \alpha^{7j}$, $\alpha^{9i} + \alpha^{9j}$ berechnet werden oder der erwartete Fehlervektor bestimmt (Nullvektor mit 1 an den Positionen i, j) und durch eine Syndromberechnung aus diesem das erwartete Syndrom. In beiden Fällen muss das beobachtete Syndrom den erwarteten Werten entsprechen, wenn es sich um einen 2-Bit Fehler gehandelt hat. Ist ein anderer, erkennbarer Fehler aufgetreten, so kann es keine Übereinstimmung geben. In diesem Fall kann durch die in aufsteigender Reihenfolge erste nicht übereinstimmende Syndromkomponente die Mindestanzahl aufgetretener Fehler eingrenzen.

Im Folgenden betrachten wir verschiedene Varianten, dieses Verfahren in Hardware zu implementieren und betrachten dazu die Laufzeit des *längsten Pfades*. Der *längste Pfad* sagt aus, wie viele Gatterstufen nach Anlegen der Eingabewerte durchlaufen werden müssen, bis die Ausgabe dem korrekten Wert entspricht. Um eine allgemeine Aussage treffen zu können, verwenden wir als Referenz die Länge der Codewörter N, wobei $N \leq 2^M - 1, M \in \mathbf{N}$.

3.2.2.2 VHDL Implementierung

In diesem Abschnitt stellen wir eine VHDL Implementierung des Verfahrens zur Korrektur von 2-Bit Fehlern mit zusätzlicher Überprüfung von mehrstelligen Fehlern dar. Als Beispiel wird ein BCH Code über $GF(2^4)$ gewählt. Das verwendete Galoisfeld $GF(2^4)$ wurde durch das Modularpolynom 101001 bzw. $x^5 + x^3 + 1$ generiert. Die VHDL Implementierung ist in Komponenten unterteilt, welche jeweils Eingangs- und Ausgangssignale besitzen.

Syndromberechnung Die Berechnung des Syndroms erhält als Eingang das übertragene Wort und gibt die Syndromkomponenten aus. In diesem Fall sind die Syndromkomponenten s_1, s_3, s_5. Die einzelnen Bits der Syndromkomponenten werden als XOR Gleichung über Bits des übertragenen Worts entsprechend der H-Matrix gebildet.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  --  Defines a design entity
5  entity syndromkomponenten is
6      PORT(
7      input : IN bit_vector(0 to 30);
8      s1 : OUT bit_vector(0 to 4);
9      s3 : OUT bit_vector(0 to 4);
10     s5 : OUT bit_vector(0 to 4)
11     );
12 end syndromkomponenten;
13
14 architecture behavioral of syndromkomponenten is
15 begin
16
17 s1(0) <= input(4) XOR input(6) XOR input(8) XOR input(9) XOR
       input(10) XOR input(12) XOR input(13) XOR input(17) XOR
       input(18) XOR input(19) XOR input(20) XOR input(21) XOR
       input(24) XOR input(25) XOR input(27) XOR input(30);
18 s1(1) <= input(3) XOR input(5) XOR input(7) XOR input(8) XOR
       input(9) XOR input(11) XOR input(12) XOR input(16) XOR input
       (17) XOR input(18) XOR input(19) XOR input(20) XOR input(23)
        XOR input(24) XOR input(26) XOR input(29);
19 s1(2) <= input(2) XOR input(7) XOR input(9) XOR input(11) XOR
       input(12) XOR input(13) XOR input(15) XOR input(16) XOR
```

```
    input(20) XOR input(21) XOR input(22) XOR input(23) XOR
    input(24) XOR input(27) XOR input(28) XOR input(30);
20 s1(3) <= input(1) XOR input(6) XOR input(8) XOR input(10) XOR
    input(11) XOR input(12) XOR input(14) XOR input(15) XOR
    input(19) XOR input(20) XOR input(21) XOR input(22) XOR
    input(23) XOR input(26) XOR input(27) XOR input(29);
21 s1(4) <= input(0) XOR input(5) XOR input(7) XOR input(9) XOR
    input(10) XOR input(11) XOR input(13) XOR input(14) XOR
    input(18) XOR input(19) XOR input(20) XOR input(21) XOR
    input(22) XOR input(25) XOR input(26) XOR input(28);
22
23 s3(0) <= input(2) XOR input(3) XOR input(4) XOR input(6) XOR
    input(7) XOR input(8) XOR input(9) XOR input(10) XOR input
    (13) XOR input(16) XOR input(17) XOR input(22) XOR input(24)
     XOR input(25) XOR input(27) XOR input(29);
24 s3(1) <= input(1) XOR input(3) XOR input(4) XOR input(6) XOR
    input(8) XOR input(12) XOR input(13) XOR input(14) XOR input
    (16) XOR input(17) XOR input(18) XOR input(19) XOR input(20)
     XOR input(23) XOR input(26) XOR input(27);
25 s3(2) <= input(3) XOR input(4) XOR input(5) XOR input(7) XOR
    input(8) XOR input(9) XOR input(10) XOR input(11) XOR input
    (14) XOR input(17) XOR input(18) XOR input(23) XOR input(25)
     XOR input(26) XOR input(28) XOR input(30);
26 s3(3) <= input(2) XOR input(4) XOR input(5) XOR input(7) XOR
    input(9) XOR input(13) XOR input(14) XOR input(15) XOR input
    (17) XOR input(18) XOR input(19) XOR input(20) XOR input(21)
     XOR input(24) XOR input(27) XOR input(28);
27 s3(4) <= input(0) XOR input(3) XOR input(6) XOR input(7) XOR
    input(12) XOR input(14) XOR input(15) XOR input(17) XOR
    input(19) XOR input(23) XOR input(24) XOR input(25) XOR
    input(27) XOR input(28) XOR input(29) XOR input(30);
28
29 s5(0) <= input(2) XOR input(4) XOR input(5) XOR input(6) XOR
    input(7) XOR input(8) XOR input(10) XOR input(11) XOR input
    (14) XOR input(15) XOR input(16) XOR input(21) XOR input(22)
     XOR input(24) XOR input(26) XOR input(29);
30 s5(1) <= input(1) XOR input(4) XOR input(8) XOR input(10) XOR
    input(11) XOR input(12) XOR input(13) XOR input(14) XOR
    input(16) XOR input(17) XOR input(20) XOR input(21) XOR
    input(22) XOR input(27) XOR input(28) XOR input(30);
31 s5(2) <= input(3) XOR input(4) XOR input(6) XOR input(8) XOR
    input(11) XOR input(15) XOR input(17) XOR input(18) XOR
    input(19) XOR input(20) XOR input(21) XOR input(23) XOR
    input(24) XOR input(27) XOR input(28) XOR input(29);
32 s5(3) <= input(2) XOR input(3) XOR input(4) XOR input(9) XOR
```

```
    input(10) XOR input(12) XOR input(14) XOR input(17) XOR
    input(21) XOR input(23) XOR input(24) XOR input(25) XOR
    input(26) XOR input(27) XOR input(29) XOR input(30);
33 s5(4) <= input(0) XOR input(1) XOR input(2) XOR input(4) XOR
    input(5) XOR input(8) XOR input(9) XOR input(10) XOR input
    (15) XOR input(16) XOR input(18) XOR input(20) XOR input(23)
     XOR input(27) XOR input(29) XOR input(30);
34
35 end behavioral;
```

Listing 3.1: VHDL Skript zur Berechnung der Syndromkomponenten

Berechnung von C' Bei dieser Berechnung wird zu eingehenden Syndromkomponenten s_1 und s_3 der Index $C' = log_\alpha(C-1) = log_\alpha(\frac{s_3}{s_1^3})$ bestimmt. Im Code werden die folgenden Schritte getätigt:

- Bestimmung des Exponenten i von $\alpha^i = s_3$ durch eine Tabelle $s3_ex = log_\alpha(s_3)$.
- Bestimmung des Exponenten i von $\alpha^i = s_1^{-3}$ durch eine Tabelle aus s_1: $s1_3_ex = log_\alpha(s_1^{-3})$
- Bestimmung des Exponenten i von $\alpha^i = \frac{s_3}{s_1^3}$ durch Addition $C' = s3_ex + s1_3_ex$ (Ganzzahlige Addition)

Die Indexe sind als Exponenten entsprechend der Basis α zu verstehen. Im Code wird C' vereinfacht als C bezeichnet.

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  --  Defines a design entity
6  entity c_berechnen is
7      PORT(
8      s1 : IN bit_vector(0 to 4);
9      s3 : IN bit_vector(0 to 4);
10     C :  OUT bit_vector(0 to 4)
11     );
12 end c_berechnen;
13
14 architecture behavioral of c_berechnen is
15
16 function to_exponent(input : in bit_vector(0 to 4)) return
      bit_vector is
17 variable result : bit_vector(0 to 4);
18 begin
19     case input is
```

```
20      when "00001" => result := "00000";
21      when "00010" => result := "00001";
22      when "00100" => result := "00010";
23      when "01000" => result := "00011";
24      when "10000" => result := "00100";
25      when "01001" => result := "00101";
26      when "10010" => result := "00110";
27      when "01101" => result := "00111";
28      when "11010" => result := "01000";
29      when "11101" => result := "01001";
30      when "10011" => result := "01010";
31      when "01111" => result := "01011";
32      when "11110" => result := "01100";
33      when "10101" => result := "01101";
34      when "00011" => result := "01110";
35      when "00110" => result := "01111";
36      when "01100" => result := "10000";
37      when "11000" => result := "10001";
38      when "11001" => result := "10010";
39      when "11011" => result := "10011";
40      when "11111" => result := "10100";
41      when "10111" => result := "10101";
42      when "00111" => result := "10110";
43      when "01110" => result := "10111";
44      when "11100" => result := "11000";
45      when "10001" => result := "11001";
46      when "01011" => result := "11010";
47      when "10110" => result := "11011";
48      when "00101" => result := "11100";
49      when "01010" => result := "11101";
50      when "10100" => result := "11110";
51      when others => result := "00000";
52      end case;
53      return result;
54  end to_exponent;
55
56
57  function exp_min_3_index(input : in bit_vector(0 to 4)) return
        bit_vector is
58  variable result : bit_vector(0 to 4);
59  begin
60      case input is
61      when "00001" => result := "00000";
62      when "00010" => result := "11100";
63      when "00011" => result := "10100";
```

```
64      when "00100" => result := "11001";
65      when "00101" => result := "01001";
66      when "00110" => result := "10001";
67      when "00111" => result := "11011";
68      when "01000" => result := "10110";
69      when "01001" => result := "10000";
70      when "01010" => result := "00110";
71      when "01011" => result := "01111";
72      when "01100" => result := "01110";
73      when "01101" => result := "01010";
74      when "01110" => result := "11000";
75      when "01111" => result := "11101";
76      when "10000" => result := "10011";
77      when "10001" => result := "10010";
78      when "10010" => result := "01101";
79      when "10011" => result := "00001";
80      when "10100" => result := "00011";
81      when "10101" => result := "10111";
82      when "10110" => result := "01100";
83      when "10111" => result := "11110";
84      when "11000" => result := "01011";
85      when "11001" => result := "01000";
86      when "11010" => result := "00111";
87      when "11011" => result := "00101";
88      when "11100" => result := "10101";
89      when "11101" => result := "00100";
90      when "11110" => result := "11010";
91      when "11111" => result := "00010";
92      when others => result := "00000";
93      end case;
94      return result;
95  end exp_min_3_index;
96
97  function add(v1,v2 : in bit_vector) return bit_vector is
98  constant m : integer := 5;
99  constant modulo : bit_vector(0 to 4) := "11111"; -- max val =
        30, mod is 31
100 variable carry : bit := '0';
101 variable dummy : bit := '0';
102 variable v_temp : bit_vector(0 to 5);
103 variable ret : bit_vector(0 to 4);
104
105 begin
106     --v_temp := (0 to 4 => v1, others => '0');
107     v_temp := (others => '0');
```

```
108      for i in m-1 downto 0 loop
109          v_temp(i) := (v_temp(i+1) and v2(i)) or (v1(i) and v2(i
                 )) or (v_temp(i+1) and v1(i));
110          v_temp(i+1) := v_temp(i+1) xor v1(i) xor v2(i);
111      end loop;
112      if (v_temp(0) = '1') then -- modulo
113          carry := '0';
114          for i in m-1 downto 0 loop
115              dummy := (modulo(i) and carry) or (modulo(i) and
                     not v_temp(i+1)) or (carry and not v_temp(i+1));
116              v_temp(i+1) := v_temp(i+1) xor carry xor modulo(i);
117              carry := dummy;
118          end loop;
119      end if;
120      if (v_temp = "011111") then -- modulo
121          v_temp := "000000";
122      end if;
123      ret := v_temp(1 to 5);
124      return ret;
125  end add;
126
127
128      --signal s1_ex : bit_vector(0 to 4) := (others=>'0');
129      signal s3_ex : bit_vector(0 to 4) := (others=>'0');
130      signal s1_3_ex : bit_vector(0 to 4) := (others=>'0');
131
132  begin
133
134  --s1_ex <= to_exponent(s1);
135  s3_ex <= to_exponent(s3);
136  s1_3_ex <= exp_min_3_index(s1);
137  C <= add(s3_ex,s1_3_ex);
138
139  end behavioral;
```

Listing 3.2: VHDL Skript zur Berechnung von C'

C Tabelle Die Tabelle zum Berechnen der Fehlerstellen bekommt den Wert $C' = log_{\alpha}(C-1)$ als Eingabe und gibt $ex = \frac{\alpha^i}{s_1}$ aus, wobei i eine Fehlerstelle ist. Die zweite Fehlerstelle entspricht $\frac{\alpha^i + s_1}{s_1} = \frac{\alpha^i}{s_1} + 1$. Die einzelnen Ausgangsbits werden aus einem OR über verschiedene Belegungen des Wertes C' gebildet. Dabei werden zunächst alle Werte C' und die jeweils erwünschten Ausgangssignale $ex(C')$ entsprechend den zugrundeliegenden Fehlern bestimmt. Für jedes Bit des Ausgangssignals ex wird anschließend eine Formel gebildet, welches dieses Bit auf 1 setzt, falls sie erfüllt ist. Dabei werden die Werte C', bei welchen dieses Bit 1 ist per OR als Unterformeln

kombiniert. Jede Formel eines Werts C' wird dabei als eine Kombination von AND und NOT über die einzelnen Bitstellen dargestellt. Gilt beispielsweise $C' = 11001$ dann ist diese Unterformeln:

$$C(0) \text{ AND } C(1) \text{ AND NOT } C(2) \text{ AND NOT } C(3) \text{ AND } C(4)$$

Eine Gesamtformel für ein Ausgangsbit kann wie folgt aussehen, falls Stelle 0 von ex für die Werte $C' = 11001$ und $C' = 01010$ den Wert 1 erhält, sonst 0.

$$\begin{aligned} ex(0) \leftarrow &(C(0) \text{ AND } C(1) \text{ AND NOT } C(2) \text{ AND NOT } C(3) \text{ AND } C(4)) \\ &\text{OR } (\text{ NOT } C(0) \text{ AND } C(1) \text{ AND NOT } C(2) \text{ AND } C(3) \text{ AND NOT } C(4)) \end{aligned} \tag{3.43}$$

Diese Formeln werden für jedes Ausgangsbit erzeugt.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  --  Defines a design entity
5  entity c_tabelle is
6      PORT(
7      C :  IN bit_vector(0 to 4);
8      ex : OUT bit_vector(0 to 4)
9      );
10 end c_tabelle;
11
12 architecture behavioral of c_tabelle is
13 begin
14
15
16 ex(0) <= (C(0) AND C(1) AND NOT C(2) AND NOT C(3) AND C(4)) OR
17          (NOT C(0) AND C(1) AND NOT C(2) AND NOT C(3) AND C(4))
              OR
18          (C(0) AND NOT C(1) AND NOT C(2) AND C(3) AND NOT C(4));
19
20 ex(1) <= (NOT C(0) AND C(1) AND NOT C(2) AND NOT C(3) AND C(4))
       OR
21          (NOT C(0) AND C(1) AND NOT C(2) AND C(3) AND NOT C(4))
              OR
22          (NOT C(0) AND C(1) AND C(2) AND C(3) AND NOT C(4)) OR
23          (C(0) AND NOT C(1) AND C(2) AND NOT C(3) AND C(4)) OR
24          (NOT C(0) AND NOT C(1) AND C(2) AND NOT C(3) AND C(4))
              OR
25          (C(0) AND NOT C(1) AND C(2) AND NOT C(3) AND NOT C(4))
              OR
26          (C(0) AND C(1) AND C(2) AND NOT C(3) AND NOT C(4));
27
28 ex(2) <= (NOT C(0) AND NOT C(1) AND C(2) AND C(3) AND C(4)) OR
```

```
29          (C(0) AND NOT C(1) AND NOT C(2) AND C(3) AND NOT C(4))
               OR
30          (NOT C(0) AND C(1) AND NOT C(2) AND C(3) AND NOT C(4))
               OR
31          (NOT C(0) AND C(1) AND C(2) AND C(3) AND NOT C(4)) OR
32          (NOT C(0) AND C(1) AND NOT C(2) AND C(3) AND C(4)) OR
33          (C(0) AND C(1) AND NOT C(2) AND C(3) AND NOT C(4)) OR
34          (C(0) AND NOT C(1) AND C(2) AND NOT C(3) AND NOT C(4));
35
36 ex(3) <= (C(0) AND NOT C(1) AND NOT C(2) AND C(3) AND C(4)) OR
37          (NOT C(0) AND C(1) AND NOT C(2) AND NOT C(3) AND C(4))
               OR
38          (NOT C(0) AND NOT C(1) AND C(2) AND C(3) AND C(4)) OR
39          (NOT C(0) AND C(1) AND C(2) AND NOT C(3) AND C(4)) OR
40          (NOT C(0) AND C(1) AND NOT C(2) AND C(3) AND NOT C(4))
               OR
41          (NOT C(0) AND C(1) AND NOT C(2) AND C(3) AND C(4)) OR
42          (NOT C(0) AND NOT C(1) AND C(2) AND NOT C(3) AND C(4));
43
44 ex(4) <= (C(0) AND C(1) AND NOT C(2) AND NOT C(3) AND C(4)) OR
45          (C(0) AND NOT C(1) AND NOT C(2) AND C(3) AND C(4)) OR
46          (C(0) AND NOT C(1) AND NOT C(2) AND C(3) AND NOT C(4))
               OR
47          (NOT C(0) AND C(1) AND NOT C(2) AND C(3) AND NOT C(4))
               OR
48          (NOT C(0) AND C(1) AND NOT C(2) AND C(3) AND C(4)) OR
49          (NOT C(0) AND NOT C(1) AND C(2) AND NOT C(3) AND C(4))
               OR
50          (C(0) AND NOT C(1) AND C(2) AND C(3) AND NOT C(4)) OR
51          (C(0) AND NOT C(1) AND C(2) AND NOT C(3) AND NOT C(4))
               OR
52          (C(0) AND C(1) AND C(2) AND NOT C(3) AND NOT C(4));
53
54
55 end behavioral;
```

Listing 3.3: VHDL Skript zur Implementierung der C-Tabelle

Berechnung der Fehlerstellen Die Berechnung der Fehlerstellen erhält den Exponenten *ex* von $\frac{\alpha^i}{s_1}$ (i ist eine Fehlerstelle) und s_1 als Eingabe und gibt die beiden Fehlerstellen $e1$ und $e2$ aus. Dabei wird wie folgt verfahren:

- Bestimme den Exponenten von $\frac{\alpha^i}{s_1}+1$ aus einer Tabelle durch Eingabe von *ex*, $exp_ex_p1 = log_\alpha(\alpha^{ex}+1)$.
- Bestimme den Exponenten $exp_s1 = log_\alpha(s_1)$

- Rechne die Fehlerstellen $e1 = ex + exp_s1$ und $e2 = exp_ex_p1 + exp_s1$ (Ganzzahlige Addition).

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  --  Defines a design entity
5  entity fehlerstellen is
6      PORT(
7      ex : IN bit_vector(0 to 4);
8      s1 : IN bit_vector(0 to 4);
9      e1 : OUT bit_vector(0 to 4);
10     e2 : OUT bit_vector(0 to 4)
11     );
12 end fehlerstellen;
13
14 architecture behavioral of fehlerstellen is
15
16 function add(v1,v2 : in bit_vector) return bit_vector is
17 constant m : integer := 5;
18 constant modulo : bit_vector(0 to 4) := "11111"; -- max val =
       30, mod is 31
19 variable carry : bit := '0';
20 variable dummy : bit := '0';
21 variable v_temp : bit_vector(0 to 5);
22 variable ret : bit_vector(0 to 4);
23
24 begin
25     --v_temp := (0 to 4 => v1, others => '0');
26     v_temp := (others => '0');
27     for i in m-1 downto 0 loop
28         v_temp(i) := (v_temp(i+1) and v2(i)) or (v1(i) and v2(i
               )) or (v_temp(i+1) and v1(i));
29         v_temp(i+1) := v_temp(i+1) xor v1(i) xor v2(i);
30     end loop;
31     if (v_temp(0) = '1') then -- modulo
32         carry := '0';
33         for i in m-1 downto 0 loop
34             dummy := (modulo(i) and carry) or (modulo(i) and
                   not v_temp(i+1)) or (carry and not v_temp(i+1));
35             v_temp(i+1) := v_temp(i+1) xor carry xor modulo(i);
36             carry := dummy;
37         end loop;
38     end if;
39     if (v_temp = "011111") then -- modulo
```

```
40          v_temp := "000000";
41      end if;
42      ret := v_temp(1 to 5);
43      return ret;
44  end add;
45
46  function to_exponent(input : in bit_vector(0 to 4)) return
       bit_vector is
47  variable result : bit_vector(0 to 4);
48  begin
49          case input is
50          when "00001" => result := "00000";
51          when "00010" => result := "00001";
52          when "00100" => result := "00010";
53          when "01000" => result := "00011";
54          when "10000" => result := "00100";
55          when "01001" => result := "00101";
56          when "10010" => result := "00110";
57          when "01101" => result := "00111";
58          when "11010" => result := "01000";
59          when "11101" => result := "01001";
60          when "10011" => result := "01010";
61          when "01111" => result := "01011";
62          when "11110" => result := "01100";
63          when "10101" => result := "01101";
64          when "00011" => result := "01110";
65          when "00110" => result := "01111";
66          when "01100" => result := "10000";
67          when "11000" => result := "10001";
68          when "11001" => result := "10010";
69          when "11011" => result := "10011";
70          when "11111" => result := "10100";
71          when "10111" => result := "10101";
72          when "00111" => result := "10110";
73          when "01110" => result := "10111";
74          when "11100" => result := "11000";
75          when "10001" => result := "11001";
76          when "01011" => result := "11010";
77          when "10110" => result := "11011";
78          when "00101" => result := "11100";
79          when "01010" => result := "11101";
80          when "10100" => result := "11110";
81          when others => result := "00000";
82          end case;
83      return result;
```

```
84  end to_exponent;
85
86  function exponent_plus_one(input : in bit_vector(0 to 4))
       return bit_vector is
87  variable result : bit_vector(0 to 4);
88  begin
89          case input is
90          when "00101" => result := "00011";
91          when "11010" => result := "11101";
92          when "11100" => result := "00010";
93          when "00011" => result := "00101";
94          when "10110" => result := "01111";
95          when "10101" => result := "11011";
96          when "01100" => result := "10100";
97          when "10001" => result := "10010";
98          when "11001" => result := "00100";
99          when "01001" => result := "11000";
100         when "01111" => result := "10110";
101         when "00110" => result := "01010";
102         when "01011" => result := "10111";
103         when "01010" => result := "00110";
104         when "11011" => result := "10101";
105         when "10111" => result := "01011";
106         when "11000" => result := "01001";
107         when "00000" => result := "11111";
108         when "00010" => result := "11100";
109         when "00001" => result := "01110";
110         when "01000" => result := "10011";
111         when "01110" => result := "00001";
112         when "10011" => result := "01000";
113         when "00111" => result := "10000";
114         when "01101" => result := "11110";
115         when "11110" => result := "01101";
116         when "00100" => result := "11001";
117         when "10100" => result := "01100";
118         when "10000" => result := "00111";
119         when "11101" => result := "11010";
120         when "10010" => result := "10001";
121         when others => result := "00000";
122         end case;
123     return result;
124 end exponent_plus_one;
125
126 signal exp_s1 : bit_vector(0 to 4);
127 signal exp_ex_p1 : bit_vector(0 to 4);
```

```
128
129 begin
130
131 exp_s1 <= to_exponent(s1);
132 exp_ex_p1 <= exponent_plus_one(ex);
133
134 e1 <= add(ex,exp_s1);
135
136 e2 <= add(exponent_plus_one(ex),exp_s1);
137
138 end behavioral;
```

Listing 3.4: VHDL Skript zur Berechnung der Fehlerstellen

Prüfen auf Mehrbitfehler Der Vorteil unseres Ansatzes ist nun, dass aus den Fehlerstellen geprüft wird, ob ein Fehler mit mehr als zwei Bit aufgetreten ist. Die Positionen $e1$ und $e2$ werden dazu durch Tabellen in α^{e1}, α^{e2}, $\alpha^{e1\times3}$, $\alpha^{e2\times3}$, $\alpha^{e1\times5}$, $\alpha^{e2\times5}$ überführt, die Exponenten jeweils modulo $2^m - 1$. Diese werden per XOR aufaddiert und müssen den Syndromkomponenten im Falle eines Zweibitfehlers entsprechen. Andernfalls handelt es sich um einen Fehler mit mehr als 2 Bit.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  --  Defines a design entity
5  entity mehrbit_test is
6      PORT(
7      e1 : IN bit_vector(0 to 4);
8      e2 : IN bit_vector(0 to 4);
9      s1 : IN bit_vector(0 to 4);
10     s3 : IN bit_vector(0 to 4);
11     s5 : IN bit_vector(0 to 4);
12     b1 : OUT boolean;
13     b3 : OUT boolean;
14     b5 : OUT boolean
15     );
16 end mehrbit_test;
17
18 architecture behavioral of mehrbit_test is
19
20 function from_exponent(input : in bit_vector(0 to 4)) return
      bit_vector is
21 variable result : bit_vector(0 to 4);
22 begin
23         case input is
24         when "00000" => result := "00001";
```

```
25          when "00001" => result := "00010";
26          when "00010" => result := "00100";
27          when "00011" => result := "01000";
28          when "00100" => result := "10000";
29          when "00101" => result := "01001";
30          when "00110" => result := "10010";
31          when "00111" => result := "01101";
32          when "01000" => result := "11010";
33          when "01001" => result := "11101";
34          when "01010" => result := "10011";
35          when "01011" => result := "01111";
36          when "01100" => result := "11110";
37          when "01101" => result := "10101";
38          when "01110" => result := "00011";
39          when "01111" => result := "00110";
40          when "10000" => result := "01100";
41          when "10001" => result := "11000";
42          when "10010" => result := "11001";
43          when "10011" => result := "11011";
44          when "10100" => result := "11111";
45          when "10101" => result := "10111";
46          when "10110" => result := "00111";
47          when "10111" => result := "01110";
48          when "11000" => result := "11100";
49          when "11001" => result := "10001";
50          when "11010" => result := "01011";
51          when "11011" => result := "10110";
52          when "11100" => result := "00101";
53          when "11101" => result := "01010";
54          when "11110" => result := "10100";
55          when others => result := "00000";
56          end case;
57      return result;
58  end from_exponent;
59
60
61  function exp_three(input : in bit_vector(0 to 4)) return
        bit_vector is
62  variable result : bit_vector(0 to 4);
63  begin
64      case input is
65      when "10000" => result := "11110";
66      when "11011" => result := "01011";
67      when "00111" => result := "10000";
68      when "00100" => result := "10010";
```

```
69      when "00101" => result := "00111";
70      when "00001" => result := "00001";
71      when "00011" => result := "01111";
72      when "01101" => result := "10111";
73      when "00010" => result := "01000";
74      when "11101" => result := "10110";
75      when "01110" => result := "01101";
76      when "10010" => result := "11001";
77      when "10011" => result := "10100";
78      when "01010" => result := "10001";
79      when "01000" => result := "11101";
80      when "01100" => result := "11000";
81      when "11000" => result := "11111";
82      when "10100" => result := "00101";
83      when "01111" => result := "00100";
84      when "11010" => result := "11100";
85      when "01001" => result := "00110";
86      when "11100" => result := "10011";
87      when "10101" => result := "11010";
88      when "11001" => result := "01110";
89      when "10111" => result := "00010";
90      when "11110" => result := "01001";
91      when "10001" => result := "10101";
92      when "10110" => result := "11011";
93      when "11111" => result := "01010";
94      when "00110" => result := "00011";
95      when "01011" => result := "01100";
96      when others => result := "00000";
97      end case;
98      return result;
99  end exp_three;
100
101 function exp_5(input : in bit_vector(0 to 4)) return bit_vector
        is
102 variable result : bit_vector(0 to 4);
103 begin
104     case input is
105     when "11011" => result := "00100";
106     when "00111" => result := "11000";
107     when "10010" => result := "10100";
108     when "00110" => result := "10101";
109     when "10100" => result := "01011";
110     when "00100" => result := "10011";
111     when "11101" => result := "00011";
112     when "11001" => result := "00101";
```

```
113       when "11110" => result := "01010";
114       when "00001" => result := "00001";
115       when "10110" => result := "01111";
116       when "11010" => result := "11101";
117       when "01111" => result := "11100";
118       when "00011" => result := "11010";
119       when "01000" => result := "00110";
120       when "01101" => result := "10000";
121       when "01100" => result := "11001";
122       when "00010" => result := "01001";
123       when "10001" => result := "00010";
124       when "01011" => result := "10010";
125       when "10000" => result := "11111";
126       when "10011" => result := "11011";
127       when "11111" => result := "01101";
128       when "01001" => result := "10001";
129       when "00101" => result := "01100";
130       when "01010" => result := "10111";
131       when "01110" => result := "00111";
132       when "10111" => result := "11110";
133       when "11000" => result := "01110";
134       when "10101" => result := "01000";
135       when "11100" => result := "10110";
136       when others => result := "00000";
137       end case;
138       return result;
139  end exp_5;
140
141  signal norm_e1 : bit_vector(0 to 4);
142  signal norm_e2 : bit_vector(0 to 4);
143
144  signal n_s1 : bit_vector(0 to 4);
145  signal n_s3 : bit_vector(0 to 4);
146  signal n_s5 : bit_vector(0 to 4);
147
148  begin
149
150  norm_e1 <= from_exponent(e1);
151  norm_e2 <= from_exponent(e2);
152
153  n_s1 <= (norm_e1 XOR norm_e2);
154  n_s3 <= (exp_three(norm_e1) XOR exp_three(norm_e2));
155  n_s5 <= (exp_5(norm_e1) XOR exp_5(norm_e2));
156
157  b1 <= n_s1 = s1;
```

```
158 b3 <= n_s3 = s3;
159 b5 <= n_s5 = s5;
160
161
162 end behavioral;
```

Listing 3.5: VHDL Skript zum Prüfen auf Mehrbitfehler

Dekodieren der Korrekturwerte Die beiden Fehlerstellen werden dekodiert, somit aus der Exponentialdarstellung $e1, e2$ zu Bitvektoren $e1_full, e2_full$, bei welchen die entsprechende Position des Fehlers den Wert 1 hat, alle anderen Positionen den Wert 0. Diese Bitvektoren für beide Fehlerstellen werden anschließend auf das übertragene Wort per XOR aufaddiert, wodurch die Fehler korrigiert werden. Ob korrigiert wird, entscheidet das Signal *decode*, welches den Wert 1 besitzt, falls die Fehlerüberprüfung den 2-Bit Fehler nicht widerlegen kann, sonst 0.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  --  Defines a design entity
5  entity dekodieren is
6      PORT(
7      e1 :     IN bit_vector(0 to 4);
8      e2 :     IN bit_vector(0 to 4);
9      decode : IN boolean;
10     input :  IN bit_vector(0 to 30);
11     output : OUT bit_vector(0 to 30)
12     );
13
14
15 end dekodieren;
16
17 architecture behavioral of dekodieren is
18     signal e1_full : bit_vector(0 to 30) := (others=>'0');
19     signal e2_full : bit_vector(0 to 30) := (others=>'0');
20     signal dec : bit := '0';
21 begin
22
23 dec <= '1' when decode else '0';
24
25 e1_full(0) <= dec AND ( NOT e1(0) AND NOT e1(1) AND NOT e1(2)
       AND NOT e1(3) AND NOT e1(4));
26 e1_full(1) <= dec AND ( NOT e1(0) AND NOT e1(1) AND NOT e1(2)
       AND NOT e1(3) AND e1(4));
27 e1_full(2) <= dec AND ( NOT e1(0) AND NOT e1(1) AND NOT e1(2)
       AND e1(3) AND NOT e1(4));
```

```
28 e1_full(3) <= dec AND ( NOT e1(0) AND NOT e1(1) AND NOT e1(2)
     AND e1(3) AND e1(4));
29 e1_full(4) <= dec AND ( NOT e1(0) AND NOT e1(1) AND e1(2) AND
     NOT e1(3) AND NOT e1(4));
30 e1_full(5) <= dec AND ( NOT e1(0) AND NOT e1(1) AND e1(2) AND
     NOT e1(3) AND e1(4));
31 e1_full(6) <= dec AND ( NOT e1(0) AND NOT e1(1) AND e1(2) AND
     e1(3) AND NOT e1(4));
32 e1_full(7) <= dec AND ( NOT e1(0) AND NOT e1(1) AND e1(2) AND
     e1(3) AND e1(4));
33 e1_full(8) <= dec AND ( NOT e1(0) AND e1(1) AND NOT e1(2) AND
     NOT e1(3) AND NOT e1(4));
34 e1_full(9) <= dec AND ( NOT e1(0) AND e1(1) AND NOT e1(2) AND
     NOT e1(3) AND e1(4));
35 e1_full(10) <= dec AND ( NOT e1(0) AND e1(1) AND NOT e1(2) AND
     e1(3) AND NOT e1(4));
36 e1_full(11) <= dec AND ( NOT e1(0) AND e1(1) AND NOT e1(2) AND
     e1(3) AND e1(4));
37 e1_full(12) <= dec AND ( NOT e1(0) AND e1(1) AND e1(2) AND NOT
     e1(3) AND NOT e1(4));
38 e1_full(13) <= dec AND ( NOT e1(0) AND e1(1) AND e1(2) AND NOT
     e1(3) AND e1(4));
39 e1_full(14) <= dec AND ( NOT e1(0) AND e1(1) AND e1(2) AND e1
     (3) AND NOT e1(4));
40 e1_full(15) <= dec AND ( NOT e1(0) AND e1(1) AND e1(2) AND e1
     (3) AND e1(4));
41 e1_full(16) <= dec AND ( e1(0) AND NOT e1(1) AND NOT e1(2) AND
     NOT e1(3) AND NOT e1(4));
42 e1_full(17) <= dec AND ( e1(0) AND NOT e1(1) AND NOT e1(2) AND
     NOT e1(3) AND e1(4));
43 e1_full(18) <= dec AND ( e1(0) AND NOT e1(1) AND NOT e1(2) AND
     e1(3) AND NOT e1(4));
44 e1_full(19) <= dec AND ( e1(0) AND NOT e1(1) AND NOT e1(2) AND
     e1(3) AND e1(4));
45 e1_full(20) <= dec AND ( e1(0) AND NOT e1(1) AND e1(2) AND NOT
     e1(3) AND NOT e1(4));
46 e1_full(21) <= dec AND ( e1(0) AND NOT e1(1) AND e1(2) AND NOT
     e1(3) AND e1(4));
47 e1_full(22) <= dec AND ( e1(0) AND NOT e1(1) AND e1(2) AND e1
     (3) AND NOT e1(4));
48 e1_full(23) <= dec AND ( e1(0) AND NOT e1(1) AND e1(2) AND e1
     (3) AND e1(4));
49 e1_full(24) <= dec AND ( e1(0) AND e1(1) AND NOT e1(2) AND NOT
     e1(3) AND NOT e1(4));
50 e1_full(25) <= dec AND ( e1(0) AND e1(1) AND NOT e1(2) AND NOT
```

```
    e1(3) AND e1(4));
51 e1_full(26) <= dec AND ( e1(0) AND e1(1) AND NOT e1(2) AND e1
     (3) AND NOT e1(4));
52 e1_full(27) <= dec AND ( e1(0) AND e1(1) AND NOT e1(2) AND e1
     (3) AND e1(4));
53 e1_full(28) <= dec AND ( e1(0) AND e1(1) AND e1(2) AND NOT e1
     (3) AND NOT e1(4));
54 e1_full(29) <= dec AND ( e1(0) AND e1(1) AND e1(2) AND NOT e1
     (3) AND e1(4));
55 e1_full(30) <= dec AND ( e1(0) AND e1(1) AND e1(2) AND e1(3)
    AND NOT e1(4));
56
57 e2_full(0) <= dec AND ( NOT e2(0) AND NOT e2(1) AND NOT e2(2)
    AND NOT e2(3) AND NOT e2(4));
58 e2_full(1) <= dec AND ( NOT e2(0) AND NOT e2(1) AND NOT e2(2)
    AND NOT e2(3) AND e2(4));
59 e2_full(2) <= dec AND ( NOT e2(0) AND NOT e2(1) AND NOT e2(2)
    AND e2(3) AND NOT e2(4));
60 e2_full(3) <= dec AND ( NOT e2(0) AND NOT e2(1) AND NOT e2(2)
    AND e2(3) AND e2(4));
61 e2_full(4) <= dec AND ( NOT e2(0) AND NOT e2(1) AND e2(2) AND
    NOT e2(3) AND NOT e2(4));
62 e2_full(5) <= dec AND ( NOT e2(0) AND NOT e2(1) AND e2(2) AND
    NOT e2(3) AND e2(4));
63 e2_full(6) <= dec AND ( NOT e2(0) AND NOT e2(1) AND e2(2) AND
    e2(3) AND NOT e2(4));
64 e2_full(7) <= dec AND ( NOT e2(0) AND NOT e2(1) AND e2(2) AND
    e2(3) AND e2(4));
65 e2_full(8) <= dec AND ( NOT e2(0) AND e2(1) AND NOT e2(2) AND
    NOT e2(3) AND NOT e2(4));
66 e2_full(9) <= dec AND ( NOT e2(0) AND e2(1) AND NOT e2(2) AND
    NOT e2(3) AND e2(4));
67 e2_full(10) <= dec AND ( NOT e2(0) AND e2(1) AND NOT e2(2) AND
    e2(3) AND NOT e2(4));
68 e2_full(11) <= dec AND ( NOT e2(0) AND e2(1) AND NOT e2(2) AND
    e2(3) AND e2(4));
69 e2_full(12) <= dec AND ( NOT e2(0) AND e2(1) AND e2(2) AND NOT
    e2(3) AND NOT e2(4));
70 e2_full(13) <= dec AND ( NOT e2(0) AND e2(1) AND e2(2) AND NOT
    e2(3) AND e2(4));
71 e2_full(14) <= dec AND ( NOT e2(0) AND e2(1) AND e2(2) AND e2
     (3) AND NOT e2(4));
72 e2_full(15) <= dec AND ( NOT e2(0) AND e2(1) AND e2(2) AND e2
     (3) AND e2(4));
73 e2_full(16) <= dec AND ( e2(0) AND NOT e2(1) AND NOT e2(2) AND
```

```
        NOT e2(3) AND NOT e2(4));
74  e2_full(17) <= dec AND ( e2(0) AND NOT e2(1) AND NOT e2(2) AND
        NOT e2(3) AND e2(4));
75  e2_full(18) <= dec AND ( e2(0) AND NOT e2(1) AND NOT e2(2) AND
        e2(3) AND NOT e2(4));
76  e2_full(19) <= dec AND ( e2(0) AND NOT e2(1) AND NOT e2(2) AND
        e2(3) AND e2(4));
77  e2_full(20) <= dec AND ( e2(0) AND NOT e2(1) AND e2(2) AND NOT
        e2(3) AND NOT e2(4));
78  e2_full(21) <= dec AND ( e2(0) AND NOT e2(1) AND e2(2) AND NOT
        e2(3) AND e2(4));
79  e2_full(22) <= dec AND ( e2(0) AND NOT e2(1) AND e2(2) AND e2
        (3) AND NOT e2(4));
80  e2_full(23) <= dec AND ( e2(0) AND NOT e2(1) AND e2(2) AND e2
        (3) AND e2(4));
81  e2_full(24) <= dec AND ( e2(0) AND e2(1) AND NOT e2(2) AND NOT
        e2(3) AND NOT e2(4));
82  e2_full(25) <= dec AND ( e2(0) AND e2(1) AND NOT e2(2) AND NOT
        e2(3) AND e2(4));
83  e2_full(26) <= dec AND ( e2(0) AND e2(1) AND NOT e2(2) AND e2
        (3) AND NOT e2(4));
84  e2_full(27) <= dec AND ( e2(0) AND e2(1) AND NOT e2(2) AND e2
        (3) AND e2(4));
85  e2_full(28) <= dec AND ( e2(0) AND e2(1) AND e2(2) AND NOT e2
        (3) AND NOT e2(4));
86  e2_full(29) <= dec AND ( e2(0) AND e2(1) AND e2(2) AND NOT e2
        (3) AND e2(4));
87  e2_full(30) <= dec AND ( e2(0) AND e2(1) AND e2(2) AND e2(3)
        AND NOT e2(4));
88
89  output <= input xor e1_full xor e2_full;
90
91  end behavioral;
```

Listing 3.6: VHDL Skript zum Dekodieren

Testbench Um die VHDL Implementierung zu testen, wurde ein Testbench erstellt, welcher alle Fehler von 1-Bit bis 3-Bit Fehlern auf das Codewort *help_word* setzt und diese durch die Korrektur laufen lässt.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4
5  entity my_entity_tb is
```

```
6  end my_entity_tb;
7
8  architecture tb of my_entity_tb is
9
10     COMPONENT syndromkomponenten
11     PORT(
12     input : IN bit_vector(0 to 30);
13     s1 : OUT bit_vector(0 to 4);
14     s3 : OUT bit_vector(0 to 4);
15     s5 : OUT bit_vector(0 to 4)
16     );
17     END COMPONENT;
18
19     COMPONENT c_berechnen
20     PORT(
21     s1 : IN bit_vector(0 to 4);
22     s3 : IN bit_vector(0 to 4);
23     C :  OUT bit_vector(0 to 4)
24     );
25     END COMPONENT;
26
27     COMPONENT c_tabelle
28     PORT(
29     C :  IN bit_vector(0 to 4);
30     ex : OUT bit_vector(0 to 4)
31     );
32     END COMPONENT;
33
34     COMPONENT fehlerstellen
35     PORT(
36     ex : IN bit_vector(0 to 4);
37     s1 : IN bit_vector(0 to 4);
38     e1 : OUT bit_vector(0 to 4);
39     e2 : OUT bit_vector(0 to 4)
40     );
41     END COMPONENT;
42
43     COMPONENT dekodieren
44     PORT(
45     e1 :     IN bit_vector(0 to 4);
46     e2 :     IN bit_vector(0 to 4);
47     input :  IN bit_vector(0 to 30);
48     output : OUT bit_vector(0 to 30)
49     );
50     END COMPONENT;
```

```
51
52      -- Inputs
53      signal input_tb : bit_vector(0 to 30) :=
            "0000000000000000000000000000000";
54
55      -- inbetween
56      signal s1_tb : bit_vector(0 to 4) := "00000";
57      signal s3_tb : bit_vector(0 to 4) := "00000";
58      signal s5_tb : bit_vector(0 to 4) := "00000";
59      signal C_tb : bit_vector(0 to 4) := "00000";
60      signal ex_tb : bit_vector(0 to 4) := "00000";
61      signal e1_tb : bit_vector(0 to 4) := "00000";
62      signal e2_tb : bit_vector(0 to 4) := "00000";
63      signal correct_corrected : boolean;
64
65      -- Outputs
66      signal output_tb : bit_vector(0 to 30) :=
            "0000000000000000000000000000000";
67      signal b1_tb : boolean := False;
68      signal b3_tb : boolean := False;
69      signal b5_tb : boolean := False;
70
71      constant help_word : bit_vector(0 to 30) :=
            "0000000000000110010000111100010";
72      constant help_invword : bit_vector(0 to 30) :=
            "1111111111111001101111000011101";
73
74  begin
75      -- connecting testbench signals with half_adder.vhd
76      u1 : entity work.syndromkomponenten port map (input =>
            input_tb, s1 => s1_tb, s3 => s3_tb, s5 => s5_tb);
77      u2 : entity work.c_berechnen port map (s1 => s1_tb, s3 =>
            s3_tb, C => C_tb);
78      u3 : entity work.c_tabelle port map (C => C_tb, ex => ex_tb
            );
79      u4 : entity work.fehlerstellen port map (ex => ex_tb, s1 =>
             s1_tb, e1 => e1_tb, e2 => e2_tb);
80      u5 : entity work.dekodieren port map (e1 => e1_tb, e2 =>
            e2_tb, decode => correct_corrected, input => input_tb,
            output => output_tb);
81      u6 : entity work.mehrbit_test port map (e1 => e1_tb, e2 =>
            e2_tb, s1 => s1_tb, s3 => s3_tb, s5 => s5_tb, b1 =>
            b1_tb, b3 => b3_tb, b5 => b5_tb);
82
83      correct_corrected <= b1_tb AND b3_tb AND b5_tb;
```

```
84
85      -- Stimulus process
86      stim_proc: process
87      begin
88      -- hold reset state for 100 ns.
89
90      --input_tb <= "0000000000000000000000000000000";
91      input_tb <= help_word;
92      wait for 100 ns;
93
94      -- 1 Bit Errors
95      for i in 30 downto 0 loop
96          input_tb <= help_word;
97          input_tb(i) <= help_invword(i);
98          wait for 10 ns;
99      end loop;
100
101     wait for 20 ns;
102
103     -- 2 Bit Errors
104     for i in 30 downto 0 loop
105         input_tb <= help_word;
106         input_tb(i) <= help_invword(i);
107         for j in i-1 downto 0 loop
108             input_tb(j) <= help_invword(j);
109             wait for 10 ns;
110             input_tb(j) <= help_word(j);
111         end loop;
112     end loop;
113
114     wait for 20 ns;
115
116     -- 3 Bit Errors
117     for i in 30 downto 0 loop
118         input_tb <= help_word;
119         input_tb(i) <= help_invword(i);
120         for j in i-1 downto 0 loop
121             input_tb(j) <= help_invword(j);
122             for k in j-1 downto 0 loop
123                 input_tb(k) <= help_invword(k);
124                 wait for 10 ns;
125                 input_tb(k) <= help_word(k);
126             end loop;
127             input_tb(j) <= help_word(j);
128         end loop;
```

```
129     end loop;
130
131     wait;
132     end process;
133
134 end tb ;
```

Listing 3.7: VHDL Testbench

Es ergibt sich, dass für das gezeigte Codewort und den Nullvektor alle 2-Bit Fehler korrigiert werden und alle 1-Bit sowie 3-Bit Fehler erkannt werden.

3.2.2.3 Softwareimplementierung und Vergleich zur Determinantenberechnung

Neben der VHDL Implementierung für eine hardwarenahe, parallele Lösung wurde eine sequentielle Softwarevariante implementiert, um exemplarisch zu vergleichen, wie sich der Ansatz zur spekulativen Berechnung mit anschließendem Vergleich im Gegensatz zur Berechnung der Determinanten verhält. Dazu wurde ein 4-Bit korrigierender BCH Code durch den Code in Anhang 5.5 erzeugt und eine Reihe von Durchläufen für verschiedene Fehler durchgeführt (Code siehe Anhang 5.9).

Dabei wird pro Fehler wie folgt vorgegangen:

- Spekulative Berechnung
  - Berechnung der Syndromkomponenten $s_1, s_3.s_5, s_7$
  - Berechnung eines Einbitfehlers durch Syndromkomponente s_1
  - Berechnung eines Zweitbitfehlers durch Syndromkomponenten s_1, s_3
  - Vergleich der Ergebnisse mit höheren Syndromkomponenten
- Determinanten im Voraus berechnen
  - Berechnung der Syndromkomponenten $s_1, s_3.s_5, s_7$
  - Berechnung der Determinanten
  - Falls 1-Bit Fehler: Berechnung eines Einbitfehlers durch Syndromkomponente s_1
  - Falls 2-Bit Fehler: Berechnung eines Zweitbitfehlers durch Syndromkomponenten s_1, s_3

Die Determinanten wurden dabei wie auf folgende Gleichungen reduziert:

- $det(2) = s1$
- $det(3) = s_1^3 + s_3$
- $det(4) = s_1^3 s3 + s_1 s_5 + s_1^6 + s_3^2$
- $det(5) = s_1 s_3^3 + s_5^2 + s_1^7 s_3 + s_1^2 s_3 s_5 + s_1^{10} + s_1^3 s_7 + s_1^5 s_5 + s_3 s_7$

Die Erwartungshaltung zu dem Vergleich ist zunächst, dass im Falle eines 2-Bit Fehlers die spekulative Berechnung schneller sein sollte, da in beiden Fällen die 2-Bit Korrektur durchzuführen ist. Im Falle eines 1-Bit, 3-Bit bzw. 4-Bit Fehlers hängt es davon ab, wie aufwändig die Berechnung der 2-Bit Korrektur gegenüber der Berechnung aller Determinanten ist.

Getestet und verglichen wurden Laufzeiten für einen bis zu 4-Bit korrigierenden BCH Code bei 2-Bit Korrektur. Der Code basiert auf dem Galoisfeld $GF(2^7)$, erzeugt durch das Modularpolynom $x^7 + x^3 + 1$ (10001001 wobei niedrigste Stelle rechts). Da die Versuche auf einem Rechner durchgeführt werden, auf welchem noch weitere Prozesse laufen, schwanken die Laufzeiten ein wenig. Da bei geringen Fehleranzahl die Fehleranzahl sehr gering ist, wird das Experiment mehrfach hintereinander wiederholt, um eine vergleichbare Zeit zu erhalten. Über diesen Gesamtdurchlauf wird dann in einem zweiten Schritt der Durchschnitt gebildet, um Schwankungen auszugleichen.

Ergebnisse 2-Bit Korrektur (2 Syndromkomponenten):
Hier werden 2 Syndromkomponenten verwendet. Es erfolgt eine 2-Bit Korrektur, keine höhere Erkennung. Für 3-Bit und 4-Bit Fehler ist es in diesem Fall möglich, dass diese nicht erkannt oder falsch korrigiert werden.

- Alle 127 1-Bit Fehler (100 Wiederholungen um ein messbares Ergebnis zu erhalten, Mittelung über 10 Durchläufe)
  - Spekulativ: 0,76 Sekunden (16514 Korrekturen pro Sekunde)
    Dies bedeutet, dass es im Durchschnitt 0,76 Sekunden gedauert hat, 100 mal jeden einzelnen 1-Bit Fehler zu konstruieren und durch den Ansatz der spekulativen Fehlerüberprüfung mit 2 Syndromkomponenten korrigieren zu lassen. Dabei wird eine spekulative 1-Bit Korrektur gestartet und getestet, ob das Ergebnis entsprechend der zweiten Syndromkomponente korrekt ist. Zusätzlich wird eine 2-Bit Korrektur durchgeführt und geprüft, ob die Berechneten Fehlerstellen das Syndrom ergeben
  - Determinante: 0,6 Sekunden (20853 Korrekturen pro Sekunde)
    Dies bedeutet, dass es im Durchschnitt 0,6 Sekunden gedauert hat, 100 mal jeden einzelnen 1-Bit Fehler zu konstruieren und durch den Ansatz der bekannten Fehlerkorrektur korrigieren zu lassen. Hierbei werden die ersten beiden vereinfachten Determinanten berechnet und anschließend eine Korrektur durchgeführt.
- Alle 8001 2-Bit Fehler (5 Wiederholungen, Mittelung über 4 Durchläufe)
  - Spekulativ: 5,62 Sekunden (7108 Korrekturen pro Sekunde)
  - Determinante: 4,43 Sekunden (9015 Korrekturen pro Sekunde)
- Alle 333375 3-Bit Fehler (Mittelung über 2 Durchläufe)
  In diesem Fall kann der Fehler nicht korrigiert werden, da ein 3-Bit Fehler jenseits der Kapazität des Codes mit 2 Syndromkomponenten liegt.
  - Spekulativ: 49,2 Sekunden (6775 Korrekturen pro Sekunde)
  - Determinante: 40,19 Sekunden (8294 Korrekturen pro Sekunde)
- Alle 10334625 4-Bit Fehler (Mittelung über 1 Durchlauf)
  Auch in diesem Fall kann der Fehler nicht korrigiert werden, da ein 4-Bit Fehler jenseits der Kapazität des Codes mit 2 Syndromkomponenten liegt.

  - Spekulativ: 28 Minuten 1 Sekunde (6145 Korrekturen pro Sekunde)
  - Determinante: 23 Minuten 20,9 Sekunden (7377 Korrekturen pro Sekunde)

Ergebnisse 2-Bit Korrektur, Erkennung bis 4-Bit Fehler (3 Syndromkomponenten): Hier werden 3 Syndromkomponenten verwendet. Es erfolgt eine 2-Bit Korrektur durch 2 Syndromkomponenten und eine zusätzliche Erkennung von Fehlern durch die dritte Syndromkomponente, somit eine Erkennung von bis zu 4-Bit Fehlern.

- Alle 127 1-Bit Fehler (100 Wiederholungen, Mittelung über 10 Durchläufe)
  - Spekulativ: 0,71 Sekunden (17837 Korrekturen pro Sekunde)
  - Determinante: 1,62 Sekunden (7825 Korrekturen pro Sekunde)
- Alle 8001 2-Bit Fehler (5 Wiederholungen, Mittelung über 4 Durchläufe)
  - Spekulativ: 6,34 Sekunden (6309 Korrekturen pro Sekunde)
  - Determinante: 7,41 Sekunden (5393 Korrekturen pro Sekunde)
- Alle 333375 3-Bit Fehler (Mittelung über 2 Durchläufe)
  - Spekulativ: 56,52 Sekunden (5897 Korrekturen pro Sekunde)
  - Determinante: 38,53 Sekunden (8651 Korrekturen pro Sekunde)
- Alle 10334625 4-Bit Fehler (Mittelung über 1 Durchlauf)
  - Spekulativ: 31 Minuten 27,12 Sekunden (5476 Korrekturen pro Sekunde)
  - Determinante: 22 Minuten 14,41 Sekunden (7744 Korrekturen pro Sekunde)

Ergebnisse 2-Bit Korrektur, Erkennung bis 6-Bit Fehler (4 Syndromkomponenten): Hier werden 4 Syndromkomponenten verwendet. Es erfolgt eine 2-Bit Korrektur durch 2 Syndromkomponenten und eine zusätzliche Erkennung von Fehlern durch zwei weitere Syndromkomponenten, somit eine Erkennung von bis zu 6-Bit Fehlern.

- Alle 127 1-Bit Fehler (100 Wiederholungen, Mittelung über 10 Durchläufe)
  - Spekulativ: 1,264 Sekunden (10047 Korrekturen pro Sekunde)
  - Determinante: 3,116 Sekunden (4075 Korrekturen pro Sekunde)
- Alle 8001 2-Bit Fehler (5 Wiederholungen, Mittelung über 4 Durchläufe)
  - Spekulativ: 7,318 Sekunden (5466 Korrekturen pro Sekunde)
  - Determinante: 9,594 Sekunden (4169 Korrekturen pro Sekunde)
- Alle 333375 3-Bit Fehler (Mittelung über 2 Durchläufe)
  - Spekulativ: 1 Minuten 3,41 Sekunden (5257 Korrekturen pro Sekunde)
  - Determinante: 1 Minuten 45,58 Sekunden (3157 Korrekturen pro Sekunde)
- Alle 10334625 4-Bit Fehler (Mittelung über 1 Durchlauf)
  - Spekulativ: 34 Minuten 48,94 Sekunden (5257 Korrekturen pro Sekunde)

- Determinante: 44 Minuten 30,23 Sekunden (3870 Korrekturen pro Sekunde)

Bei diesen Werten ist zu beachten, dass nur eine bis zu 2-Bit Fehlerkorrektur durchgeführt wurde und diese in den Zeiten enthalten ist, falls sie berechnet wird. Im Falle der spekulativen Ausführung kann dies bei 2-Bit oder höheren Fehlern auftreten (falls diese nicht fälschlich als 1-Bit Fehler identifiziert werden). Die Methode, bei welcher Determinanten berechnet werden, führt diese Korrektur nur aus, falls ein 2-Bit Fehler festgestellt wurde. Da die Determinanten in absteigendender Reihenfolge berechnet werden, verläuft die Determinantenberechnung schneller, wenn es sich um einen Fehler mit hoher Bitanzahl handelt. Die verschiedenen Werte können durch die Anzahl der Korrekturen pro Sekunde verglichen werden, da die Dauer des Durchlaufs an die Fehleranzahl gebunden ist.

Die Werte zeigen, dass im Falle einer 2-Bit Korrektur mit 2 Syndromkomponenten der Ansatz der Determinantenberechnung schneller abgeschnitten hat. Werden 3 Syndromkomponenten verwendet, also eine zusätzliche Syndromkomponente zur Erkennung höherer Fehler, ist die spekulative Fehlerberechnung bei 1-Bit und 2-Bit Fehlern schneller, bei 3-Bit und 4-Bit Fehlern langsamer. Unter Verwendung von 4 Syndromkomponenten schneidet die spekulative Fehlerberechnung schneller ab, als die Berechnung der Determinanten. Es ist erkennbar, dass die Laufzeiten in der Regel langsamer werden, je mehr Syndromkomponenten für die Fehlererkennung hinzukommen.

Während bei keiner oder geringer Korrektur ein Vorteil bei der Berechnung der Determinanten zu sehen ist, steigt der Aufwand der spekulativen Berechnung bei höheren Fehlererkennungen langsamer.

Diese Berechnung stellt nur einen Einzelfall dar, zeigt aber, dass der Vorteil einer spekulativen Korrektur bereits bei 4 Syndromkomponenten zu sehen ist. Es ist davon auszugehen, dass je mehr Syndromkomponenten, desto langsamer wird die Berechnung der Determinanten. Die Anzahl der Summanden einer Determinante ist die Fakultät der Breite der Matrix. Bei einer Breite von x Zeilen und Spalten ist die Anzahl der Summanden wie folgt

$$(x)(x-1)(x-2)...(2) = x! \tag{3.44}$$

Die Größenordnung der Fakultät lässt sich durch die Stirlingformel abschätzen:

$$n! \sim \sqrt{2\pi n}(\frac{n}{e})^n, n \to \inf \tag{3.45}$$

Die Laufzeit zur Berechnung der Determinanten befindet sich somit in exponentieller Laufzeit zu der Anzahl der korrigierbaren Fehler, wobei die Laufzeit zur Überprüfung der spekulativen Berechnungen linear zu der Anzahl der korrigierenden Fehler verhält. Die lineare Länge im Fall der spekulativen Berechnung kommt dadurch, dass die zu prüfende Gleichung, welche bei einem weiteren Bit hinzu kommt, konstant bei derselben Gleichung bleibt.

3.2.2.4 Erweiterung auf andere Fehler

In diesem Abschnitt zeigen wir, wie der Ansatz zur spekulativen Fehlerkorrektur für eine höhere Fehlerkorrektur angewendet werden kann. Generell ist eine Anwendung für beliebige Fehlerkorrektur möglich, dazu werden die entsprechenden Gleichungen der Syndromkomponenten genutzt.

$$
\begin{aligned}
s_{x,2} &= \alpha^{i_1 \times x} + \alpha^{i_2 \times x} \\
s_{x,3} &= \alpha^{i_1 \times x} + \alpha^{i_2 \times x} + \alpha^{i_3 \times x} \\
s_{x,4} &= \alpha^{i_1 \times x} + \alpha^{i_2 \times x} + \alpha^{i_3 \times x} + \alpha^{i_4 \times x} \\
&\dots \\
s_{x,y} &= \sum_{z=1}^{y} \alpha^{i_z \times x}
\end{aligned} \tag{3.47}
$$

$s_{x,y}$ steht hier für die Syndromkomponente s_x im Falle einer y-Bit Fehlerkorrektur, $\{i_1, ..., i_y\}$.

Das Vorgehen für eine y-Bit Fehlerkorrektur mit zusätzlicher Erkennung funktioniert nun wie folgt.

1. Spekulative Ausführung der y-Bit Fehlerkorrektur unter der Annahme, dass ein y-Bit Fehler aufgetreten ist.
2. Berechnung und Vergleich der Syndromkomponenten $s_{x,y}$ entsprechend der oberen Gleichungen 3.47 für alle x ab $x = 2 \times y + 1$.

Wurden alle Syndromkomponenten bis zu (inklusive) einer Syndromkomponente s_x erfolgreich überprüft, so ist ein y-Bit Fehler oder mindestens $(x - y + 2)$-Bit Fehler aufgetreten sein. Gelten andernfalls Gleichungen ab einer Syndromkomponente s_x nicht, so ist ein mindestens $(x - y)$-Bit Fehler aufgetreten.

Für den 2-Bit Fall $y = 2$ ergeben sich die Werte der vorangegangenen Abschnitte. Es kann zusammengefasst werden, dass jede verwendete Syndromkomponente entweder für ein zusätzliches Bit Korrektur oder für 2 zusätzliche Bits Fehlererkennung verwendet werden kann. Die Aufteilung muss nicht bei der Konstruktion vorgenommen werden, sondern kann bei Erhalt eines fehlerhaften Wortes entschieden werden.

Hier darf nicht der Trugschluss entstehen, man könne erst eine Fehlererkennung ohne Korrektur machen und im Anschluss den Fehler mit einer möglichst hohen Korrektur korrigieren. Ein Fehler mit hoher Bitzahl, welcher mit einer hohen Fehlererkennung erkannt wird, wird bei einer hohen Fehlerkorrektur potentiell trotzdem falsch zugeordnet. Beträgt beispielsweise der minimale Codeabstand 7, so können Fehler bis 6 Bit erkannt werden. Wird jedoch eine 2-Bit Korrektur verwendet, so liegen 5-Bit und 6-Bit Fehler näher an einem anderen Codewort, so dass diese potentiell einem 2-Bit oder 1-Bit Fehler entsprechen würden. Eine Unterscheidung ist mangels Information über die Anzahl der Fehler nicht möglich.

3.2.3 Alternatives Vorgehen der 2-Bit Korrektur

Im vorherigen Abschnitt wurden 2-Bit Fehler korrigiert, indem deren Positionen ermittelt wurden und diese mit höheren Syndromkomponenten verglichen wurden, um Fehler mit mehr Bitstellen auszuschließen. In diesem Ansatz wird ein alternatives Vorgehen vorgestellt, bei welchem bereits vor der Berechnung die ein 1-Bit und 2-Bit Fehler von einem 3-Bit und 4-Bit Fehler unterschieden werden kann.

Zunächst wird erneut das Verfahren von Okano Imai verwendet, um aus einem Lokatorpolynom die Positionen der Fehler allein durch Syndromkomponenten zu berechnen.

Im Falle eines 2-Bit Fehlers gilt:

$$
\begin{aligned}
s_1 &= \alpha^i + \alpha^j \\
s_3 &= \alpha^{3i} + \alpha^{3j} \\
s_5 &= \alpha^{5i} + \alpha^{5j} \\
s_7 &= \alpha^{7i} + \alpha^{7j} \\
&\dots
\end{aligned}
\tag{3.49}
$$

sowie

$$
\begin{aligned}
s_1 &= \alpha^i + \alpha^j \\
s_1^3 &= \alpha^{3i} + \alpha^{3j} + \alpha^{2i+j} + \alpha^{i+2j} \\
s_1^5 &= \alpha^{5i} + \alpha^{5j} + \alpha^{4i+j} + \alpha^{i+4j} \\
s_1^7 &= \alpha^{7i} + \alpha^{7j} + \alpha^{6i+j} + \alpha^{i+6j} + \alpha^{4i+3j} + \alpha^{3i+4j} + \alpha^{5i+2j} + \alpha^{2i+5j} \\
&\dots
\end{aligned}
\tag{3.51}
$$

Daraus ergibt sich die Differenz aus s_x zu s_1^x wie folgt:

$$
\begin{aligned}
s_3 + s_1^3 &= \alpha^{2i+j} + \alpha^{i+2j} \\
s_5 + s_1^5 &= \alpha^{4i+j} + \alpha^{i+4j} \\
s_7 + s_1^7 &= \alpha^{6i+j} + \alpha^{i+6j} + \alpha^{4i+3j} + \alpha^{3i+4j} + \alpha^{5i+2j} + \alpha^{2i+5j} \\
&\dots
\end{aligned}
\tag{3.53}
$$

Zur Verbesserung wird nun erneut das Verfahren von Okano und Imai[OI87] zur Bestimmung der Positionen eines 2-Bit Fehler betrachtet.

$$
\begin{aligned}
0 &= (x + \alpha^i) \times (x + \alpha^j) \\
&= x^2 + x \times (\alpha^i + \alpha^j) + \alpha^i \times \alpha^j \\
&= x^2 + x \times s_1 + \alpha^{i+j} \times \frac{s_1}{s_1} \\
&= x^2 + x \times s_1 + \alpha^{i+j} \times \frac{\alpha^i + \alpha^j}{s_1} \\
&= x^2 + x \times s_1 + \frac{\alpha^{i+2j} + \alpha^{2i+j}}{s_1} \\
&= x^2 + x \times s_1 + \frac{\alpha^{i+2j} + \alpha^{2i+j} + 2s_3}{s_1} \\
&= x^2 + x \times s_1 + \frac{\alpha^{i+2j} + \alpha^{2i+j} + \alpha^{3i} + \alpha^{3j} + s_3}{s_1} \\
&= x^2 + x \times s_1 + \frac{(\alpha^i + \alpha^j)^3 + s_3}{s_1} \\
&= x^2 + x \times s_1 + \frac{s_1^3 + s_3}{s_1} = 0
\end{aligned}
\tag{3.55}
$$

Statt des Multiplizierens mit $\frac{s_1}{s_1}$ wird stattdessen $\frac{s_3}{s_3}$ verwendet. Diese Vorgehensweise setzt voraus, dass $s_3 \neq 0$ ist. Dadurch entwickelt sich die Gleichung wie folgt:

$$\begin{aligned}
0 &= x^2 + x \times (\alpha^i + \alpha^j) + \alpha^i \times \alpha^j \\
&= x^2 + x \times s_1 + \alpha^{i+j} \times \frac{s_3}{s_3} \\
&= x^2 + x \times s_1 + \alpha^{i+j} \times \frac{\alpha^{3i} + \alpha^{3j}}{s_3} \\
&= x^2 + x \times s_1 + \frac{\alpha^{i+4j} + \alpha^{4i+j}}{s_3} \\
&= x^2 + x \times s_1 + \frac{\alpha^{i+4j} + \alpha^{4i+j} + 2s_5}{s_3} \\
&= x^2 + x \times s_1 + \frac{\alpha^{i+4j} + \alpha^{4i+j} + \alpha^{5i} + \alpha^{5j} + s_5}{s_1} \\
&= x^2 + x \times s_1 + \frac{(\alpha^i + \alpha^j)^5 + s_5}{s_3} \\
&= x^2 + x \times s_1 + \frac{s_1^5 + s_5}{s_3} = 0
\end{aligned} \tag{3.57}$$

Insgesamt sehen die verschiedenen Gleichungen wie folgt aus

$$\begin{aligned}
0 &= x^2 + x \times (\alpha^i + \alpha^j) + \alpha^i \times \alpha^j \\
0 &= x^2 + x \times s_1 + \frac{s_1^5 + s_5}{s_3} \\
0 &= x^2 + x \times s_1 + \frac{s_1^3 + s_3}{s_1}
\end{aligned} \tag{3.59}$$

Diese Gleichungen können verglichen werden, um bis zu 4-Bit Fehler abzudecken (unter der Voraussetzung, dass $s_3 \neq 0$, $s_1 \neq 0$):

$$\alpha^i \times \alpha^j = \alpha^{i+j} = \frac{s_1^5 + s_5}{s_3} = \frac{s_1^3 + s_3}{s_1} \tag{3.60}$$

Da in der zusätzlichen Berechnung eine weitere Syndromkomponente benutzt wird, liegt die Vermutung nahe, dass die Gleichheit bei Fehlern von mehr als 2-Bit Fehlern wohlmöglich nicht gilt. Um die verschiedenen Fälle zu trennen bezeichnen wir im Folgenden $s_{x,y}$ als die Syndromkomponente x unter Annahme eines y-Bit Fehlers. Ist $s_1 = 0$, so wurde durch diese Syndromkomponenten allein kein erkennbarer Fehler festgestellt. In diesem Fall kann kein 2-Bit Fehler aufgetreten sein, da die H-Matrix keine duplizierten α-Werte der Syndromkomponente enthält. Ist $s_3 = 0$ und es gibt keine duplizierten Spalten der zugehörigen Zeile der H-Matrix, dann kann ebenfalls kein 2-Bit Fehler aufgetreten sein. Den sonstigen Fall betrachten wir als Sonderfall in Unterabschnitt 3.2.3. In den folgenden Abschnitten beweisen wir, dass die Ungleichheit gelten muss, wenn ein 3-Bit bzw. 4-Bit Fehler eingetreten ist. Wir verwenden eine umgestellte Form der Gleichung, um

die Überprüfung zu machen.

$$(s_1^5 + s_5) \times (s_1) = (s_1^3 + s_3) \times (s_3) \tag{3.61}$$

Tritt kein Fehler auf, so gilt die Gleichheit, da beide Seiten 0 sind.

Sonderfall $s_3 = 0$ Ist die Länge des Codes durch 3 teilbar beispielsweise 15 ($2^4 - 1$), so besteht der Teil der H-Matrix, welcher für die Syndromkomponenten von s_3 verantwortlich ist, aus wiederholt derselben Folge an α^i.

Beispiel: Modularpolynom 11001

$$H = \begin{bmatrix} \alpha^0, & \alpha^1, & \alpha^2, & \alpha^3, & ... & \alpha^{n-1} \\ \alpha^0, & \alpha^3, & \alpha^6, & \alpha^9, & ... & \alpha^{(n-1)\times 3} \end{bmatrix}$$

$$= \left[\begin{array}{c|c|c} 00011 & 11010 & 11001 \\ 00100 & 01111 & 01011 \\ 01000 & 11110 & 10110 \\ 10001 & 11101 & 01100 \\ \hline 01100 & 01100 & 01100 \\ 00110 & 00110 & 00110 \\ 00101 & 00101 & 00101 \\ 10111 & 10111 & 10111 \end{array}\right] \tag{3.63}$$

In der zweiten Zeile der H-Matrix ist erkennbar, dass sich die ersten fünf α Werte drei mal wiederholen. So ist sowohl in der fünften Spalte, als auch in der zehnten Spalte der Wert dieser Zeile $\alpha^{12} = \alpha^{12} = 0011$ identisch. Tritt ein Fehler an zwei solchen Stellen auf, addieren sich diese im Syndrom zu dem Nullvektor auf.

In solchen Fällen kann daher ein 2-Bit Fehler mit $s_3 = 0$ auftreten.

Anwendung der Gleichung Aus den vorhergehenden Überlegungen ergibt sich die folgende Gleichung, welche zu prüfen ist:

$$(s_1^5 + s_5) \times (s_1) \stackrel{?}{=} (s_1^3 + s_3) \times (s_3) \tag{3.64}$$

Diese Gleichung kann benutzt werden, um verschiedene Fehlerfälle vom 2-Bit Fehlerfall zu unterscheiden. Im Folgenden beweisen wir, dass

- Im Falle eines 0-Bit Fehlers die Gleichung gilt
- Im Falle eines 1-Bit Fehlers die Gleichung gilt
- Im Falle eines 2-Bit Fehlers die Gleichung gilt
- Im Falle eines 3-Bit Fehlers die Gleichung nicht gilt
- Im Falle eines 4-Bit Fehlers die Gleichung nicht gilt

Es kann also geschlossen werden, dass kein 1-Bit oder 2-Bit Fehler eingetreten ist, falls die Gleichung nicht gilt. Im Falle eines 3-Bit Fehlers oder 4-Bit Fehlers muss diese Ungleichheit gelten,

somit sind durch die Gleichung 3-Bit und 4-Bit Fehler von 0,1 und 2-Bit Fehlern unterscheidbar. Der Beweis des 0-Bit Falles ist trivial, da alle Syndromkomponenten 0 sind.

Beweis der Gleichung im 1-Bit Fehlerfall Im Fall einer einzelnen Fehlerstelle sind die Syndromkomponenten wie folgt:

$$\begin{aligned} s_{1,1} &= \alpha^{i} \\ s_{3,1} &= \alpha^{3i} \\ s_{5,1} &= \alpha^{5i} \\ s_{1,1}^{3} &= s_{3,1} = \alpha^{3i} \\ s_{1,1}^{5} &= s_{5,1} = \alpha^{5i} \end{aligned} \tag{3.66}$$

Setzen wir diese Werte in die Gleichung ein, ergibt sich folgendes.

$$\begin{aligned} (s_1^5 + s_5) \times (s_1) &\stackrel{?}{=} (s_1^3 + s_3) \times (s_3) \\ (s_1^5 + s_1^5) \times (s_1) &\stackrel{?}{=} (s_1^3 + s_1^3) \times (s_3) \\ 0 \times (s_1) &\stackrel{?}{=} 0 \times (s_3) \\ 0 &= 0 \end{aligned} \tag{3.68}$$

Somit gilt die Gleichung $(s_1^5 + s_5) \times (s_1) = (s_1^3 + s_3) \times (s_3)$ im Falle eines 1-Bit Fehlers.

Beweis der Gleichung im 2-Bit Fehlerfall In diesem Abschnitt wird gezeigt, dass im Fall eines 2-Bit Fehlers die Gleichheit der Gleichung gelten muss.

Im 2-Bit Fall ergeben sich die Syndromkomponenten wie folgt:

$$\begin{aligned} s_{1,2} &= \alpha^{i} + \alpha^{j} \\ s_{3,2} &= \alpha^{3i} + \alpha^{3j} \\ s_{5,2} &= \alpha^{5i} + \alpha^{5j} \\ s_{1,2}^{3} &= (\alpha^{2i} + \alpha^{2j}) \times (\alpha^{i} + \alpha^{j}) \\ &= \alpha^{3i} + \alpha^{3j} + \alpha^{2i+j} + \alpha^{2j+i} \\ s_{1,2}^{5} &= (\alpha^{4i} + \alpha^{4j}) \times (\alpha^{i} + \alpha^{j}) \\ &= \alpha^{5i} + \alpha^{5j} + \alpha^{4i+j} + \alpha^{4j+i} \end{aligned} \tag{3.70}$$

Dazu führen wir den folgenden Beweis für den 2-Bit Fehlerfall:

$$(s_{1,2}^5 + s_{5,2}) \times (s_{1,2}) \stackrel{?}{=} (s_{1,2}^3 + s_{3,2}) \times (s_{3,2}) \tag{3.71}$$

$$
\begin{aligned}
(s_{1,2}^5 + s_{5,2}) \times s_{1,2} &= (s_{5,2} + \alpha^{5i} + \alpha^{5j} + \alpha^{4i+j} + \alpha^{4j+i}) \times s_{1,2} \\
&= (\alpha^{4i+j} + \alpha^{4j+i}) \times (\alpha^i + \alpha^j) \\
&= (\alpha^{5i+j} + \alpha^{5j+i} + \alpha^{4i+2j} + \alpha^{4j+2i})
\end{aligned}
\tag{3.73}
$$

$$
\begin{aligned}
(s_{1,2}^3 + s_{3,2}) \times s_{3,2} &= (s_{3,2} + \alpha^{3i} + \alpha^{3j} + \alpha^{2i+j} + \alpha^{2j+i}) \times s_{3,2} \\
&= (\alpha^{2i+j} + \alpha^{2j+i}) \times (\alpha^{3i} + \alpha^{3j} + \alpha^{3k}) \\
&= (\alpha^{5i+j} + \alpha^{5j+i} + \alpha^{4i+2j} + \alpha^{4j+2i}) \\
&= (s_{1,2}^5 + s_{5,2}) \times (s_{1,2})
\end{aligned}
\tag{3.75}
$$

Es gilt somit im Falle eines 2-Bit Fehlers $(s_1^5 + s_5) \times (s_1) = (s_1^3 + s_3) \times (s_3)$.

Beweis der Gleichung im 3-Bit Fehlerfall In diesem Abschnitt wird gezeigt, dass die Gleichung im Fall eines 3-Bit Fehlers nicht gelten kann. Im 3-Bit Fall ergeben sich die Syndromkomponenten wie folgt:

$$
\begin{aligned}
s_{1,3} &= \alpha^i + \alpha^j + \alpha^k \\
s_{3,3} &= \alpha^{3i} + \alpha^{3j} + \alpha^{3k} \\
s_{5,3} &= \alpha^{5i} + \alpha^{5j} + \alpha^{5k} \\
s_{1,3}^3 &= (\alpha^{2i} + \alpha^{2j} + \alpha^{2k}) \times (\alpha^i + \alpha^j + \alpha^k) \\
&= \alpha^{3i} + \alpha^{3j} + \alpha^{3k} + \alpha^{2i+j} + \alpha^{2i+k} + \alpha^{2j+i} + \alpha^{2j+k} + \alpha^{2k+i} + \alpha^{2k+j} \\
s_{1,3}^5 &= (\alpha^{4i} + \alpha^{4j} + \alpha^{4k}) \times (\alpha^i + \alpha^j + \alpha^k) \\
&= \alpha^{5i} + \alpha^{5j} + \alpha^{5k} + \alpha^{4i+j} + \alpha^{4i+k} + \alpha^{4j+i} + \alpha^{4j+k} + \alpha^{4k+i} + \alpha^{4k+j}
\end{aligned}
\tag{3.77}
$$

Dazu führen wir den folgenden Beweis für den 3-Bit Fehlerfall:

$$
(s_{1,3}^5 + s_{5,3}) \times (s_{1,3}) \stackrel{?}{=} (s_{1,3}^3 + s_{3,3}) \times (s_{3,3}) \tag{3.78}
$$

$$
\begin{aligned}
&(s_{1,3}^5 + s_{5,3}) \times s_{1,3} \\
&\quad= (s_{5,3} + \alpha^{5i} + \alpha^{5j} + \alpha^{5k} + \alpha^{4i+j} + \alpha^{4i+k} + \alpha^{4j+i} + \alpha^{4j+k} + \alpha^{4k+i} + \alpha^{4k+j}) \times s_{1,3} \\
&\quad= (\alpha^{4i+j} + \alpha^{4i+k} + \alpha^{4j+i} + \alpha^{4j+k} + \alpha^{4k+i} + \alpha^{4k+j}) \times (\alpha^i + \alpha^j + \alpha^k) \\
&\quad= (\alpha^{5i+j} + \alpha^{5i+k} + \alpha^{4j+2i} + \cancel{\alpha^{4j+k+i}} + \alpha^{4k+2i} + \cancel{\alpha^{4k+j+i}}) \\
&\quad+ (\alpha^{4i+2j} + \cancel{\alpha^{4i+k+j}} + \alpha^{5j+i} + \alpha^{5j+k} + \cancel{\alpha^{4k+i+j}} + \alpha^{4k+2j}) \\
&\quad+ (\cancel{\alpha^{4i+j+k}} + \alpha^{4i+2k} + \cancel{\alpha^{4j+i+k}} + \alpha^{4j+2k} + \alpha^{5k+i} + \alpha^{5k+j})
\end{aligned}
\tag{3.80}
$$

$$
\begin{aligned}
&(s_{1,3}^3 + s_{3,3}) \times s_{3,3} \\
&\quad = (s_{3,3} + \alpha^{3i} + \alpha^{3j} + \alpha^{3k} + \alpha^{2i+j} + \alpha^{2i+k} + \alpha^{2j+i} + \alpha^{2j+k} + \alpha^{2k+i} + \alpha^{2k+j}) \times s_{3,3} \\
&\quad = (\alpha^{2i+j} + \alpha^{2i+k} + \alpha^{2j+i} + \alpha^{2j+k} + \alpha^{2k+i} + \alpha^{2k+j}) \times (\alpha^{3i} + \alpha^{3j} + \alpha^{3k}) \\
&\quad = (\alpha^{5i+j} + \alpha^{5i+k} + \alpha^{2j+4i} + \alpha^{2j+k+3i} + \alpha^{2k+4i} + \alpha^{2k+j+3i}) \\
&\quad + (\alpha^{2i+4j} + \alpha^{2i+k+3j} + \alpha^{5j+i} + \alpha^{5j+k} + \alpha^{2k+i+3j} + \alpha^{2k+4j}) \\
&\quad + (\alpha^{2i+j+3k} + \alpha^{2i+4k} + \alpha^{2j+i+3k} + \alpha^{2j+4k} + \alpha^{5k+i} + \alpha^{5k+j})
\end{aligned}
\tag{3.82}
$$

$$
\begin{aligned}
(s_{1,3}^5 + s_{5,3}) \times s_{1,3} &\stackrel{?}{=} (s_{1,3}^3 + s_{3,3}) \times s_{3,3} \\
&(\cancel{\alpha^{5i+j}} + \cancel{\alpha^{5i+k}} + \cancel{\alpha^{4j+2i}} + \cancel{\alpha^{4k+2i}} + \cancel{\alpha^{4i+2j}} + \cancel{\alpha^{5j+i}} \\
&+ \cancel{\alpha^{5j+k}} + \cancel{\alpha^{4k+2j}} + \cancel{\alpha^{4i+2k}} + \cancel{\alpha^{4j+2k}} + \cancel{\alpha^{5k+i}} + \cancel{\alpha^{5k+j}}) \\
&\stackrel{?}{=} (\cancel{\alpha^{5i+j}} + \cancel{\alpha^{5i+k}} + \cancel{\alpha^{2j+4i}} + \alpha^{2j+k+3i} + \cancel{\alpha^{2k+4i}} + \alpha^{2k+j+3i} \\
&+ \cancel{\alpha^{2i+4j}} + \alpha^{2i+k+3j} + \cancel{\alpha^{5j+i}} + \cancel{\alpha^{5j+k}} + \alpha^{2k+i+3j} + \cancel{\alpha^{2k+4j}} \\
&+ \alpha^{2i+j+3k} + \cancel{\alpha^{2i+4k}} + \alpha^{2j+i+3k} + \cancel{\alpha^{2j+4k}} + \cancel{\alpha^{5k+i}} + \cancel{\alpha^{5k+j}}) \qquad (3.84) \\
&0 \stackrel{?}{=} (\alpha^{2j+k+3i} + \alpha^{2k+j+3i} + \alpha^{2i+k+3j} + \alpha^{2k+i+3j} + \alpha^{2i+j+3k} + \alpha^{2j+i+3k}) \\
&0 \stackrel{?}{=} \alpha^{i+j+k} \times ((\alpha^i + \alpha^j) \times (\alpha^i + \alpha^k) \times (\alpha^j + \alpha^k)) \\
&0 \stackrel{?}{=} (\alpha^i + \alpha^j) \times (\alpha^i + \alpha^k) \times (\alpha^j + \alpha^k) \\
&0 \stackrel{?}{=} (\alpha^i + \alpha^j) \\
&\alpha^i \neq \alpha^j
\end{aligned}
$$

Da von 3 fehlerhaften Bits ausgegangen wird, müssen die Fehler α^i, α^j und α^k paarweise verschieden und ungleich 0 sein. Die Division durch α^{i+j+k} (welches ungleich 0 ist) und wahlweise $(\alpha^i + \alpha^k)$ und $(\alpha^j + \alpha^k)$ zeigt, dass zwei Fehlerstellen identisch sein müssten, damit die Gleichung gilt. Dies widerspricht der Grundannahme, bewiest somit, dass die Ungleichheit $(s_1^5 + s_5) \times (s_1) \neq (s_1^3 + s_3) \times (s_3)$ im Falle eines 3-Bit Fehlers gilt. Also kann diese Gleichung verwendet werden, um unter der Annahme eines 2-Bit Fehler zu testen, ob ein 3-Bit Fehler aufgetreten ist, wenn nicht gleichzeitig $s_1 = 0, s_3 = 0$. Für höhere Fehler führen wir noch die folgenden Beweise.

Beweis der Gleichung im 4-Bit Fehlerfall In diesem Abschnitt wird gezeigt, dass die Gleichung bei einem 4-Bit Fehler nicht gelten kann. Im 4-Bit Fall ergeben sich die Syndromkomponenten wie

folgt:

$$
\begin{aligned}
s_{1,4} &= \alpha^{i} + \alpha^{j} + \alpha^{k} + \alpha^{l} \\
s_{3,4} &= \alpha^{3i} + \alpha^{3j} + \alpha^{3k} + \alpha^{3l} \\
s_{5,4} &= \alpha^{5i} + \alpha^{5j} + \alpha^{5k} + \alpha^{5l} \\
s_{1,4}^{3} &= (\alpha^{2i} + \alpha^{2j} + \alpha^{2k} + \alpha^{2l}) \times (\alpha^{i} + \alpha^{j} + \alpha^{k} + \alpha^{l}) \\
&= \alpha^{3i} + \alpha^{3j} + \alpha^{3k} + \alpha^{3l} + \alpha^{2i+j} + \alpha^{2i+k} + \alpha^{2i+l} + \alpha^{2j+i} \\
&+ \alpha^{2j+k} + \alpha^{2j+l} + \alpha^{2k+i} + \alpha^{2k+j} + \alpha^{2k+l} + \alpha^{2l+i} + \alpha^{2l+j} + \alpha^{2l+k} \\
s_{1,4}^{5} &= (\alpha^{4i} + \alpha^{4j} + \alpha^{4k} + \alpha^{4l}) \times (\alpha^{i} + \alpha^{j} + \alpha^{k} + \alpha^{l}) \\
&= \alpha^{5i} + \alpha^{5j} + \alpha^{5k} + \alpha^{5l} + \alpha^{4i+j} + \alpha^{4i+k} + \alpha^{4i+l} + \alpha^{4j+i} + \alpha^{4j+k} + \alpha^{4j+l} \\
&+ \alpha^{4k+i} + \alpha^{4k+j} + \alpha^{4k+l} + \alpha^{4l+i} + \alpha^{4l+j} + \alpha^{4l+k}
\end{aligned}
\tag{3.86}
$$

Dazu führen wir den folgenden Beweis für den 4-Bit Fehlerfall:

$$
(s_{1,4}^{5} + s_{5,4}) \times (s_{1,4}) \stackrel{?}{=} (s_{1,4}^{3} + s_{3,4}) \times (s_{3,4})
\tag{3.87}
$$

$$
\begin{aligned}
&(s_{1,4}^{5} + s_{5,4}) \times s_{1,4} \\
&= (\alpha^{4i+j} + \alpha^{4i+k} + \alpha^{4i+l} + \alpha^{4j+i} + \alpha^{4j+k} + \alpha^{4j+l} \\
&+ \alpha^{4k+i} + \alpha^{4k+j} + \alpha^{4k+l} + \alpha^{4l+i} + \alpha^{4l+j} + \alpha^{4l+k}) \times s_{1,4} \\
&= (\alpha^{4i+j} + \alpha^{4i+k} + \alpha^{4i+l} + \alpha^{4j+i} + \alpha^{4j+k} + \alpha^{4j+l} \\
&+ \alpha^{4k+i} + \alpha^{4k+j} + \alpha^{4k+l} + \alpha^{4l+i} + \alpha^{4l+j} + \alpha^{4l+k}) \times (\alpha^{i} + \alpha^{j} + \alpha^{k}\alpha^{l}) \\
&= (\alpha^{5i+j} + \alpha^{5i+k} + \alpha^{5i+l} + \alpha^{4j+2i} + \cancel{\alpha^{4j+k+i}} + \cancel{\alpha^{4j+l+i}} \\
&+ \alpha^{4k+2i} + \cancel{\alpha^{4k+j+i}} + \cancel{\alpha^{4k+l+i}} + \alpha^{4l+2i} + \cancel{\alpha^{4l+j+i}} + \cancel{\alpha^{4l+k+i}}) \\
&+ (\alpha^{4i+2j} + \cancel{\alpha^{4i+k+j}} + \cancel{\alpha^{4i+l+j}} + \alpha^{5j+i} + \alpha^{5j+k} + \alpha^{5j+l} \\
&+ \cancel{\alpha^{4k+i+j}} + \alpha^{4k+2j} + \cancel{\alpha^{4k+l+j}} + \cancel{\alpha^{4l+i+j}} + \alpha^{4l+2j} + \cancel{\alpha^{4l+k+j}}) \\
&+ (\cancel{\alpha^{4i+j+k}} + \alpha^{4i+2k} + \cancel{\alpha^{4i+l+k}} + \cancel{\alpha^{4j+i+k}} + \alpha^{4j+2k} + \cancel{\alpha^{4j+l+k}} \\
&+ \alpha^{5k+i} + \alpha^{5k+j} + \alpha^{5k+l} + \cancel{\alpha^{4l+i+k}} + \cancel{\alpha^{4l+j+k}} + \alpha^{4l+2k}) \\
&+ (\cancel{\alpha^{4i+j+l}} + \cancel{\alpha^{4i+k+l}} + \alpha^{4i+2l} + \cancel{\alpha^{4j+i+l}} + \cancel{\alpha^{4j+k+l}} + \alpha^{4j+2l} \\
&+ \cancel{\alpha^{4k+i+l}} + \cancel{\alpha^{4k+j+l}} + \alpha^{4k+2l} + \alpha^{5l+i} + \alpha^{5l+j} + \alpha^{5l+k})
\end{aligned}
\tag{3.89}
$$

$$
\begin{aligned}
(s_{1,4}^3 + s_{3,4}) \times s_{3,4} \\
&= (\alpha^{2i+j} + \alpha^{2i+k} + \alpha^{2i+l} + \alpha^{2j+i} + \alpha^{2j+k} + \alpha^{2j+l} + \alpha^{2k+i} + \alpha^{2k+j} \\
&+ \alpha^{2k+l} + \alpha^{2l+i} + \alpha^{2l+j} + \alpha^{2l+k}) \times s_{3,4} \\
&= (\alpha^{2i+j} + \alpha^{2i+k} + \alpha^{2i+l} + \alpha^{2j+i} + \alpha^{2j+k} + \alpha^{2j+l} + \alpha^{2k+i} + \alpha^{2k+j} \\
&+ \alpha^{2k+l} + \alpha^{2l+i} + \alpha^{2l+j} + \alpha^{2l+k}) \times (\alpha^{3i} + \alpha^{3j} + \alpha^{3k} + \alpha^{3l}) \\
&= (\alpha^{5i+j} + \alpha^{5i+k} + \alpha^{5i+l} + \alpha^{2j+4i} + \alpha^{2j+k+3i} + \alpha^{2j+l+3i} \\
&+ \alpha^{2k+4i} + \alpha^{2k+j+4i} + \alpha^{2k+l+3i} + \alpha^{2l+4i} + \alpha^{2l+j+3i} + \alpha^{2l+k+3i}) \\
&+ (\alpha^{2i+4j} + \alpha^{2i+k+3j} + \alpha^{2i+l+3j} + \alpha^{5j+i} + \alpha^{5j+k} + \alpha^{5j+l} \\
&+ \alpha^{2k+i+3j} + \alpha^{2k+4j} + \alpha^{2k+l+3j} + \alpha^{2l+i+3j} + \alpha^{2l+4j} + \alpha^{2l+k+3j}) \\
&+ (\alpha^{2i+j+3k} + \alpha^{2i+4k} + \alpha^{2i+l+3k} + \alpha^{2j+i+3k} + \alpha^{2j+4k} + \alpha^{2j+l+3k} \\
&+ \alpha^{5k+i} + \alpha^{5k+j} + \alpha^{5k+l} + \alpha^{2l+i+3k} + \alpha^{2l+j+3k} + \alpha^{2l+4k}) \\
&+ (\alpha^{2i+j+3l} + \alpha^{2i+k+3l} + \alpha^{2i+4l} + \alpha^{2j+i+3l} + \alpha^{2j+k+3l} + \alpha^{2j+4l} \\
&+ \alpha^{2k+i+3l} + \alpha^{2k+j+3l} + \alpha^{2k+4l} + \alpha^{5l+i} + \alpha^{5l+j} + \alpha^{5l+k})
\end{aligned}
\tag{3.91}
$$

$$
\begin{aligned}
(s_{1,4}^5 + s_{5,4}) \times s_{1,4} &\stackrel{?}{=} (s_{1,4}^3 + s_{3,4}) \times s_{3,4} \\
&(\cancel{\alpha^{5i+j}} + \cancel{\alpha^{5i+k}} + \cancel{\alpha^{5i+l}} + \cancel{\alpha^{4j+2i}} + \cancel{\alpha^{4k+2i}} + \cancel{\alpha^{4l+2i}} + \cancel{\alpha^{4i+2j}} \\
&+ \cancel{\alpha^{5j+i}} + \cancel{\alpha^{5j+k}} + \cancel{\alpha^{5j+l}} + \cancel{\alpha^{4k+2j}} + \cancel{\alpha^{4l+2j}} + \cancel{\alpha^{4i+2k}} + \cancel{\alpha^{4j+2k}} + \cancel{\alpha^{5k+i}} \\
&+ \cancel{\alpha^{5k+j}} + \cancel{\alpha^{5k+l}} + \cancel{\alpha^{4l+2k}} + \cancel{\alpha^{4i+2l}} + \cancel{\alpha^{4j+2l}} + \cancel{\alpha^{4k+2l}} + \cancel{\alpha^{5l+i}} \\
&+ \alpha^{5l+j} + \cancel{\alpha^{5l+k}}) \\
&\stackrel{?}{=} (\cancel{\alpha^{5i+j}} + \cancel{\alpha^{5i+k}} + \cancel{\alpha^{5i+l}} + \cancel{\alpha^{2j+4i}} + \alpha^{2j+k+3i} + \alpha^{2j+l+3i} \\
&+ \cancel{\alpha^{2k+4i}} + \alpha^{2k+j+3i} + \alpha^{2k+l+3i} + \cancel{\alpha^{2l+4i}} + \alpha^{2l+j+3i} + \alpha^{2l+k+3i} \\
&+ \cancel{\alpha^{2i+4j}} + \alpha^{2i+k+3j} + \alpha^{2i+l+3j} + \cancel{\alpha^{5j+i}} + \cancel{\alpha^{5j+k}} + \cancel{\alpha^{5j+l}} \\
&+ \alpha^{2k+i+3j} + \cancel{\alpha^{2k+4j}} + \alpha^{2k+l+3j} + \alpha^{2l+i+3j} + \cancel{\alpha^{2l+4j}} + \alpha^{2l+k+3j} \\
&+ \alpha^{2i+j+3k} + \cancel{\alpha^{2i+4k}} + \alpha^{2i+l+3k} + \alpha^{2j+i+3k} + \cancel{\alpha^{2j+4k}} + \alpha^{2j+l+3k} \\
&+ \cancel{\alpha^{5k+i}} + \cancel{\alpha^{5k+j}} + \cancel{\alpha^{5k+l}} + \alpha^{2l+i+3k} + \alpha^{2l+j+3k} + \cancel{\alpha^{2l+4k}} \\
&+ \alpha^{2i+j+3l} + \alpha^{2i+k+3l} + \cancel{\alpha^{2i+4l}} + \alpha^{2j+i+3l} + \alpha^{2j+k+3l} + \cancel{\alpha^{2j+4l}} \\
&+ \alpha^{2k+i+3l} + \alpha^{2k+j+3l} + \cancel{\alpha^{2k+4l}} + \cancel{\alpha^{5l+i}} + \cancel{\alpha^{5l+j}} + \cancel{\alpha^{5l+k}}) \\
0 &\stackrel{?}{=} (\alpha^{2j+k+3i} + \alpha^{2j+l+3i} + \alpha^{2k+j+3i} + \alpha^{2k+l+3i} + \alpha^{2l+j+3i} + \alpha^{2l+k+3i} \\
&+ \alpha^{2i+k+3j} + \alpha^{2i+l+3j} + \alpha^{2k+i+3j} + \alpha^{2k+l+3j} + \alpha^{2l+i+3j} + \alpha^{2l+k+3j} \\
&+ \alpha^{2i+j+3k} + \alpha^{2i+l+3k} + \alpha^{2j+i+3k} + \alpha^{2j+l+3k} + \alpha^{2l+i+3k} + \alpha^{2l+j+3k} \\
&+ \alpha^{2i+j+3l} + \alpha^{2i+k+3l} + \alpha^{2j+i+3l} + \alpha^{2j+k+3l} + \alpha^{2k+i+3l} + \alpha^{2k+j+3l}) \\
&= \sum_{a,b,c \in \{i,j,k,l\}, a \neq b \neq c} \alpha^{3a+2b+c} \\
0 &\stackrel{?}{=} (\alpha^i + \alpha^j) \times (\alpha^i + \alpha^k) \times (\alpha^i + \alpha^l) \times (\alpha^j + \alpha^k) \times (\alpha^j + \alpha^l) \times (\alpha^k + \alpha^l) \\
0 &\stackrel{?}{=} (\alpha^i + \alpha^j) \\
&\alpha^i \neq \alpha^j
\end{aligned}
\tag{3.93}
$$

Wie im Fall des 3-Bit Fehlers ergibt sich bei dem 4-Bit Fehler auch, dass einige α Summanden mit verschiedenen Exponenten übrig bleiben, nachdem beide Seiten der Gleichung gekürzt wurden. Im Gegensatz zu dem 3-Bit Fall kommen in den übrigen Exponenten nicht alle verschiedenen Fehlerstellen, sondern nur 3 vor. Diese lassen sich ähnlich in Produkte aus jeweils zwei α hoch zwei verschiedenen Fehlerstellen zusammenfassen. Da wir von 4 fehlerhaften Bits ausgehen, müssen α^i, α^j, α^k und α^l paarweise verschieden und ungleich 0 sein. Jeder der geklammerten Werte (z.B. $(\alpha^i + \alpha^k)$) ist dabei ungleich 0. Die Gleichheit kann somit nicht gelten, da ein Produkt von Werten ungleich 0 selbst ungleich 0 ist. Somit wurde bewiesen, dass die Ungleichheit $(s_1^5 + s_5) \times (s_1) \neq (s_1^3 + s_3) \times (s_3)$ im Falle eines 4-Bit Fehlers gilt. Also kann diese Gleichung verwendet werden, um unter der Annahme eines 2-Bit Fehler zu testen, ob ein 4-Bit Fehler aufgetreten ist, wenn nicht gleichzeitig $s_1 = 0, s_3 = 0$.

Erweiterung des Ansatzes für mehr als 2-Bit Fehler Bei einem BCH Code kann per Konstruktion pro Syndromkomponente 2 fehlerhafte Bits erkannt werden. Soll bei m Syndromkomponenten gleichzeitig eine n-Bit Korrektur gemacht werden, sind insgesamt Fehler bis $n+2\times(m-n)$ erkennbar. Im gegebenen Fall können wir mit 3 Syndromkomponenten (s_1, s_3, s_5) 2-Bit korrigieren und also Fehler bis 4-Bit erkennen. Dies spiegelt sich im Vergleich der obigen Gleichungen wider.

Die Frage ist nun, wie sich dieser Ansatz auf Erkennung zusätzlicher Bit-Fehler erweitern lässt, da jede Syndromkomponente eine Erkennung von 2 weiteren Bit-Fehlern bewirken kann. Hier eine längere Liste der potenzierten ersten Syndromkomponenten:

$$\begin{aligned} s_3+s_1^3 &= \alpha^{2i+j}+\alpha^{i+2j} \\ s_5+s_1^5 &= \alpha^{4i+j}+\alpha^{i+4j} \\ s_7+s_1^7 &= \alpha^{6i+j}+\alpha^{i+6j}+\alpha^{4i+3j}+\alpha^{3i+4j}+\alpha^{5i+2j}+\alpha^{2i+5j} \\ &\ldots \end{aligned} \tag{3.95}$$

Geht man nun allgemein an das bekannte Schema, und verwenden Syndrom s_x so ergibt sich:

$$\begin{aligned} \alpha^{i+j} &= \alpha^{i+j}\times\frac{s_{x-2}}{s_{x-2}} \\ &= \frac{\alpha^{i(x-1)+j}+\alpha^{i+j(x-1)}}{s_{x-2}} \\ &= \frac{\alpha^{ix}+\alpha^{jx}+\alpha^{i(x-1)+j}+\alpha^{i+j(x-1)}+s_x}{s_{x-2}} \end{aligned} \tag{3.97}$$

Falls nun $s_1^x=\alpha^{ix}+\alpha^{jx}+\alpha^{i(x-1)+j}+\alpha^{i+j(x-1)}$, dann lässt sich die Gleichung allgemein wie folgt zusammenfassen:

$$\alpha^{i+j}=\frac{s_1^x+s_x}{s_{x-2}} \tag{3.99}$$

Dieses gilt für alle $x\in 2^n+1, n\in\mathbb{N}_+$, also $x\in\{3,5,9,17,33,\ldots\}$. Hier ist nun die Frage, wie restlichen Syndromkomponenten verwendet werden können, um α^{i+j} darzustellen. Als Demonstration des Vorgehens verwenden wir nun s_7:

$$\begin{aligned} \alpha^{i+j} &= \alpha^{i+j}\times\frac{s_5}{s_5} \\ &= \frac{\alpha^{i6+j}+\alpha^{i+j6}}{s_5} \\ &= \frac{\alpha^{i7}+\alpha^{j7}+\alpha^{i6+j}+\alpha^{i+j6}+s_7}{s_5} \\ &= \frac{s_1^7+s_7+\alpha^{4i+3j}+\alpha^{3i+4j}+\alpha^{5i+2j}+\alpha^{2i+5j}}{s_5} \\ &= \frac{s_1^7+s_7+\alpha^{2(i+j)}\times(\alpha^{3i}+\alpha^{3j})+\alpha^{3(i+j)}\times(\alpha^{i}+\alpha^{j})}{s_5} \\ &= \frac{s_1^7+s_7+\alpha^{2(i+j)}\times s_3+\alpha^{3(i+j)}\times s_1}{s_5} \end{aligned} \tag{3.101}$$

Nun kann in diese Gleichungen das zuvor berechnete α^{i+j} eingesetzt werden.

Diese Herangehensweise kann verwendet werden, um jeweils eine Vergleichsformel für eine bestimmte Syndromkomponente zu bestimmen. Werden alle diese Formeln gegeneinander verglichen, können somit pro Formel 2 weitere Bitfehler (entsprechend der Syndromkomponenten, welche verwendet wurden) ausgeschlossen werden. Es muss allerdings immer beachtet werden, welche Randfälle sich durch die Division ergeben.

3.3 3-Bit Fehlerkorrektur mit zusätzlicher Erkennung

Das vorgestellte Verfahren aus Abschnitt 3.2.2 kann erweitert werden, um 3-Bit Fehler abzudecken. Um eine Korrektur von 3-Bit Fehler durchzuführen, wird ein Code mit mindestens 3 Syndromkomponenten benötigt, zusätzliche Syndromkomponenten können entsprechend des Ansatzes zur zusätzlichen Erkennung höherer Fehler verwendet werden.

Das Vorgehen ist wie folgt: Unter der spekulativen Annahme, dass ein 3-Bit Fehler aufgetreten ist, werden die Fehlerstellen i, j, k bestimmt. Zur Bestimmung kann z.B. das Verfahren nach Okano-Imai [OI87], wie in Abschnitt 2.3.2.3 beschrieben, verwendet werden. Scheitern die Berechnungsschritte (Division durch 0, kein Tabelleneintrag), kann kein 3-Bit Fehler aufgetreten sein. Es muss somit gelten:

$$\begin{aligned} i \neq j &\neq k \\ s_1 &= \alpha^i + \alpha^j + \alpha^k \\ s_3 &= \alpha^{i^3} + \alpha^{j^3} + \alpha^{k^3} \\ s_5 &= \alpha^{i^5} + \alpha^{j^5} + \alpha^{k^5} \end{aligned} \tag{3.103}$$

Darauf folgend werden die verschiedenen Syndromkomponenten überprüft, ob kein Fehler mit mehr als 3 Bit aufgetreten ist.

$$\begin{aligned} s_7 &\stackrel{?}{=} \alpha^{i^7} + \alpha^{j^7} + \alpha^{k^7} \\ s_9 &\stackrel{?}{=} \alpha^{i^9} + \alpha^{j^9} + \alpha^{k^9} \\ &\dots \end{aligned} \tag{3.105}$$

Die Gleichungen für die Syndromkomponenten s_1, s_3 und s_5, welche in die Berechnung der Fehlerstellen eingeflossen sind, müssen dabei nicht getestet werden.

Gelten die Syndromgleichungen für s_7 und s_9, so ist der Fehler entweder ein 3-Bit Fehler oder es muss mindestens ein 8-Bit Fehler aufgetreten sein. Es kann kein 4-Bit, 5-Bit, 6-Bit oder 7-Bit Fehler aufgetreten sein.

Allgemein gilt, wenn die die Syndromgleichungen bis inklusive s_n gelten, kann kein Fehler zwischen 4 und $n-2$ Bit aufgetreten sein. Der Fehler muss dann ein 3-Bit Fehler oder ein Fehler mit mindestens $(n-1)$-Fehler sein.

3.4 4-Bit Fehlerkorrektur mit zusätzlicher Erkennung

Das vorgestellte Verfahren kann ebenfalls auf 4-Bit Fehler erweitert werden. Zur Korrektur wird dabei ein BCH Code mit mindestens 4 Syndromkomponenten benötigt. Weitere Syndromkomponenten werden für eine zusätzliche Fehlererkennung verwendet.

Der Vorgang ist im Falle der 4-Bit Korrektur wie folgt: Unter der spekulativen Annahme, dass ein 4-Bit Fehler aufgetreten ist, werden die 4 Fehlerstellen i, j, k, l bestimmt. Das Verfahren zur Bestimmung der Fehlerstellen kann beliebig gewählt werden, z.B. die Lösung des Lokatorpolynoms nach Okano Imai. Es gilt:

$$\begin{aligned} & i \neq j \neq k \neq l \\ & s_1 = \alpha^i + \alpha^j + \alpha^k + \alpha^l \\ & s_3 = \alpha^{i^3} + \alpha^{j^3} + \alpha^{k^3} + \alpha^{l^3} \\ & s_5 = \alpha^{i^5} + \alpha^{j^5} + \alpha^{k^5} + \alpha^{l^5} \\ & s_7 = \alpha^{i^7} + \alpha^{j^7} + \alpha^{k^7} + \alpha^{l^7} \end{aligned} \tag{3.107}$$

Im Anschluss wird überprüft, ob die weiteren Syndromkomponenten den berechneten Fehlerstellen entsprechen.

$$\begin{aligned} s_9 &\stackrel{?}{=} \alpha^{i^9} + \alpha^{j^9} + \alpha^{k^9} + \alpha^{l^9} \\ s_{11} &\stackrel{?}{=} \alpha^{i^{11}} + \alpha^{j^{11}} + \alpha^{k^{11}} + \alpha^{l^{11}} \\ &\ldots \end{aligned} \tag{3.109}$$

Gelten alle Gleichungen bei Syndromkomponenten bis inklusive s_n, ist ein 4-Bit Fehler oder mindestens ein (nicht berücksichtigter) $(n-2)$-Bit Fehler aufgetreten. Gilt mindestens eine Gleichung nicht, kann kein 4-Bit Fehler aufgetreten sein.

Gelten beispielsweise alle Gleichungen bei Syndromkomponenten bis inklusive s_{11}, so ist ein 4-Bit oder mindestens 9-Bit Fehler aufgetreten.

Gelten alle Gleichungen bis zu (inklusive) einer Syndromkomponente s_n, die Gleichung für die folgende Syndromkomponente s_{n+2} jedoch nicht, so ist ein mindestens $(n-2)$-Bit Fehler aufgetreten. Ein Fehler bis $(n-3)$-Bit ist ausgeschlossen, da alle Gleichungen für Syndromkomponenten bis s_n gelten, ein 4-Bit Fehler ist ausgeschlossen, weil die Gleichung für s_{n+2} nicht gilt.

Gelten beispielsweise alle Gleichungen bis s_{11}, aber nicht die Gleichung für die Syndromkomponente s_{13}, so ist ein mindestens 9-Bit Fehler aufgetreten. In solchen Fällen, in denen eine Gleichung nicht zutrifft, kann kein 4-Bit Fehler aufgetreten sein.

3.5 Allgemeine Fehlerkorrektur mit zusätzlicher Erkennung

Zusätzlich zu den demonstrierten Verfahren bis zur 4-Bit Korrektur ist da vorgestellte Verfahren zur Fehlerkorrektur mit zusätzlicher Erkennung auf beliebige höhere Korrekturen mit zusätzlicher Erkennung erweiterbar.

Der allgemeine Ansatz für m-Bit Fehler lautet wie folgt: Unter der spekulativen Annahme, dass ein m-Bit Fehler aufgetreten ist, bestimme die m Fehlerstellen $i_1, i_2, \ldots i_m$. Gegeben sei ein Code mit

$q > m$ Syndromkomponenten (mindestens m Syndromkomponenten werden benötigt, um m Fehler zu korrigieren). Die letzte Syndromkomponente ist s_p mit $p = 2q - 1$. Hierzu kann ein beliebiges Verfahren zur Bestimmung von m-Bit Fehlern benutzt werden. Das Lösen eines Lokatorpolynoms für $m > 4$ ist nicht mehr allgemein algebraisch lösbar, daher muss ein anderes Verfahren, z.B. das Suchverfahren von Chien, verwendet werden. Es gilt:

$$\begin{aligned} &\forall j,k \in \{1,2,...,m\}, j \neq k : i_j \neq i_k \\ s_1 &= \sum_{n=1}^{m} \left(\alpha^{i_n}\right) \\ s_3 &= \sum_{n=1}^{m} \left(\alpha^{i_n 3}\right) \\ &\dots \\ s_{2m-1} &= \sum_{n=1}^{m} \left(\alpha^{i_n 2m-1}\right) \end{aligned} \tag{3.111}$$

Nach der Bestimmung der m Fehlerstellen erfolgt die Überprüfung auf höhere Fehlerstellen. Hier beginnend ab der Syndromkomponente s_{2m+1}, da die Syndromkomponenten bis s_{2m-1} bereits bei der Bestimmung verwendet wurde:

$$\begin{aligned} s_{2m+1} &\stackrel{?}{=} \sum_{n=1}^{m} \left(\alpha^{i_n 2t+1}\right) \\ s_{2m+3} &\stackrel{?}{=} \sum_{n=1}^{m} \left(\alpha^{i_n 2t+3}\right) \\ &\dots \\ s_{2q-1} &\stackrel{?}{=} \sum_{n=1}^{m} \left(\alpha^{i_n 2q-1}\right) \end{aligned} \tag{3.113}$$

Gelten alle Gleichungen bis s_r, so ist ein m-Bit Fehler oder ein (nicht berücksichtigter) Fehler mit mindestens $(r + 1 - m)$-Bit aufgetreten. Ein m-Bit Fehler ist ausgeschlossen, falls eine Gleichung nicht gilt.

Gelten die Gleichungen bis zu einem s_r, aber die Gleichung für die Syndromkomponente s_{r+2} nicht, so ist ein Fehler mit mindestens $(r + 1 - m)$-Bit aufgetreten. Man beachte, dass s_r die t-te Syndromkomponente ist mit $r = 2t - 1$.

Vereinfachte Darstellung Es ergibt sich, dass der erarbeitete Ansatz wie folgt vereinfacht dargestellt werden kann.

- Führe eine spekulative Fehlerkorrektur für alle Korrekturverfahren parallel durch
- Berechne pro Fehlervektor das spekulative Fehlersyndrom (beispielsweise auch durch Anwendung der H-Matrix)
- Vergleiche das spekulative Fehlersyndrom mit dem aufgetretenen Fehlersyndrom
-

3.6 Vergleich zum bekannten Verfahren

In diesem Abschnitt wird dargestellt, wie sich das vorgestellte Verfahren im Gegensatz zu den bekannten Verfahren unterscheidet.

In diesem Kapitel wurde ein Verfahren zur Korrektur und zusätzlicher Erkennung unter Verwendung einer spekulativen Fehlerkorrektur vorgestellt. Dabei werden verschiedene Verfahren zur spekulativen Korrektur von beispielsweise 1-Bit, 2-Bit, 3-Bit und 4-Bit Fehlern durchgeführt, die Ergebnisse gegen erkennbare Fehler überprüft und eine entsprechende Korrektur oder Ausgabe eines erkannten Fehlers, welcher nicht korrigiert werden kann, durchgeführt.

Das vorgestellte Verfahren zur spekulativen Fehlerberechnung ermöglicht es, die Kapazität des BCH Codes insoweit auszuschöpfen, dass alle entsprechend der Anzahl der Syndromkomponenten erkennbaren Fehler erkannt und von den zu korrigierenden Fehlern unterschieden werden können. Das bekannte Verfahren zur Bestimmung der Fehleranzahl durch die Berechnung der Determinanten besitzt keine vollständige Abdeckung aller erkennbaren Fehler, da die Determinanten nur für entsprechend des Codeabstandes korrigierbare Fehler erzeugt werden können.

Generell ist ein entscheidender Vorteil des vorgestellten Verfahrens, das der Schritt zur Feststellung der Fehleranzahl übersprungen werden kann, höhere Fehler jedoch trotzdem erkannt werden können, ohne die Anzahl der Fehler festzustellen. Somit wird der Schritt des Berechnens der Determinanten übersprungen und stattdessen eine Überprüfung der spekulativen Fehlerstellen durchgeführt. Dies wurde exemplarisch für bis zu 4-Bit Fehler demonstriert.

Im Vergleich zu der Berechnung der Determinanten für alle erkennbaren Fehler stellt dies eine enorme Verbesserung dar, da die Formeln der Determinanten fakultativ wachsen. Zudem wird im bekannten Verfahren mit der größten Determinante begonnen, welche dem höchsten erkennbaren Fehler entspricht. Es werden somit die unwahrscheinlichsten Fälle zuerst getestet. Das vorgestellte Verfahren ermöglicht es, die zu korrigierenden Fehler zu unterscheiden und eine Überprüfung auf höhere Fehler im Anschluss durchzuführen. Dabei können verschieden Fehlerfälle parallel überprüft werden.

Im Vergleich zum Berlekamp-Massey Algorithmus wird ein Ansatz zur direkten Bestimmung der Lokatorpolynome bis vierten Grades statt der iterativen Herangehensweise verwendet. Der Berlekamp-Massey Algorithmus muss für die Erkennung höherer Fehler das Lokatorpolynom über weitere Iterationen bestimmen, während der vorgestellte Ansatz eine schnelle, parallele Überprüfung auf höhere Fehler bietet.

Das Verfahren kann analog auch für eine größere Anzahl von Bitfehlern durchgeführt werden. So kann beispielsweise eine 3-Bit Korrektur unter Verwendung der Fehlersyndromkomponenten s_1, s_3 und s_5 erfolgen. Anschließend kann bestimmt werden, ob die aus den 3 bestimmten Bitpositionen berechneten Syndromkomponenten $s_7, s_9, \ldots$ korrekt sind.

3.6.1 VHDL Vergleichsimplementierung Determinante gegen spekulative Berechnung

Um das vorgestellte Verfahren der spekulativen Fehlerberechnung gegen die Bestimmung der Fehleranzahl im Voraus vergleichen zu können, werden in diesem Abschnitt Vergleichsimplementierungen in VHDL vorgestellt.

Ausgegangen von einer 1-Bit Korrektur mit zusätzlicher Erkennung ist der Unterschied beider Verfahren die Berechnung der Determinanten bzw. Berechnung der höheren Syndromkomponenten unter Annahme eines 1-Bit Fehlers. Zu beiden Teilverfahren wurde VHDL Code für jeweils 4 und 5 Syndromkomponenten ausgearbeitet. Diese VHDL Implementierungen enthalten daher lediglich die Determinantenberechnung und Prüfung auf höhere Fehler.

In den Implementierungen wurden BCH Codes über dem Galoisfeld $GF(2^5)$ mit dem Modularpolynom $p(x) = x^7 + x^3 + 1$ verwendet. Bei 4 Syndromkomponenten kann dieser Code neben einer 1-Bit Korrektur alle bis zu 7-Bit Fehler erkennen. Bei 5 Syndromkomponenten kann dieser Code neben einer 1-Bit Korrektur alle bis zu 9-Bit Fehler erkennen.

Im Anhang dieser Arbeit sind dazu die Varianten mit 5 Syndromkomponenten zu finden (Anhang 5.6 und Anhang 5.7). Zum Testen der Korrektheit wurde zusätzlich ein Testbench erzeugt, welcher sich ebenfalls im Anhang befindet (Anhang 5.8).

Die Implementierungen wurden mittels Genus Synthesis Solution 21.10-p002_1 synthetisiert, jeweils mit den identischen Einstellungen:

- library = ixc013ng_stdcell_slow_1p08V_125C.lib
- syn_generic_effort = medium
- syn_map_effort = high

Dabei ergeben nach der Synthese folgende Ergebnisse für verwendete Fläche und Laufzeit folgende Werte:

Name	Cell Area	Data Path (ps)
Determinante 5 Syndromkomponenten	19695	10641
Determinante 4 Syndromkomponenten	6566	6406
Spekulativ 5 Syndromkomponenten	2948	4831
Spekulativ 4 Syndromkomponenten	2537	4447

Die Tabelle zeigt jeweils welche Fläche (Cell Area) und Dauer des längsten Pfades (Data Path) die Vergleichsimplementierungen der Determinantenberechnung und spekulativen Fehlerbestimmung für einen BCH Code mit jeweils 4 und 5 Syndromkomponenten bei der Synthese ergeben haben. In der ersten Zeile ist beispielsweise zeigt die Berechnung von Determinanten eines BCH Code mit 5 Syndromkomponenten eine Fläche (Cell Area) von 19695 Einheiten und einen längsten Pfad (Data Path) von 10641 ps.

Betrachtet man die Werte in Relation zueinander, ist ersichtlich, dass die Ergebnisse der spekulativen Berechnung der Fehlerpositionen schneller sind ($4831/10641 = 45\%$ für 5 Syndromkomponenten / $4447/6406 = 69\%$ für 4 Syndromkomponenten) und weniger Fläche ($2948/19695 = 15\%$ für 5 Syndromkomponenten / $2537/6566 = 39\%$ für 4 Syndromkomponenten) auf einem Chip benötigen würde, als das Berechnen der Determinanten im Voraus.

Anhand dieser Implementierungen ist zu sehen, dass die Bestätigung der Fehlerstellen in diesem Fall wesentlich weniger Fläche und Laufzeit benötigt, als eine Berechnung von Determinanten zur Fehlerbestimmung.

Jenseits der in dieser Auswertung verglichenen Implementierung wird die Prognose aufgestellt, dass der Vorteil der spekulativen Berechnung größer wird, je mehr Syndromkomponenten insgesamt verwendet werden. Dieser Effekt schwächt sich jedoch ab, je mehr Fehler korrigiert werden sollen.

4 Zusammenfassung

In dieser Arbeit wird auf Grundlage des BCH Codes untersucht, wie eine Fehlerkorrektur mit einer Erkennung höherer Fehleranzahlen kombiniert werden kann.

Mit dem Verfahren der 1-Bit Korrektur mit zusätzlicher Erkennung höherer Fehler wurde ein Ansatz entwickelt, welcher die Erkennung zusätzlicher Fehler durch das parallele Lösen einfacher Gleichungen der Form $s_x \stackrel{?}{=} s_1^x$ durchführt. Die Anzahl dieser Gleichungen ist linear zu der Anzahl der zu überprüfenden höheren Fehler.

In dieser Arbeit wurde zusätzlich für bis zu 4-Bit Korrekturen mit zusätzlicher Erkennung höherer Fehler ein weiterer allgemeiner Ansatz vorgestellt. Dabei werden parallel für alle korrigierbaren Fehleranzahlen spekulative Fehlerkorrekturen durchgeführt. Aus den bestimmten Fehlerstellen werden spekulative Syndromkomponenten erzeugt, durch welche die Fehlerstellen bestätigt und höhere erkennbare Fehleranzahlen ausgeschlossen werden können.

Die vorgestellten Ansätze unterscheiden sich von dem in [Pet60] entwickelten Ansatz, bei welchem die Anzahl der Fehlerstellen durch die Berechnung von Determinanten in absteigender Reihenfolge berechnet wird, bis die erste Determinante 0 bildet. Bei dem bekannten Verfahren ist durch die Berechnung der Determinanten eine faktorielle Anzahl an Berechnungen in Relation zu der Anzahl zu überprüfender Fehler durchzuführen. Im Vergleich zu dem bekannten sequentiellen Verfahrens nach Berlekamp Massey besitzen die Berechnungen im vorgestellten Ansatz simple Gleichungen und können parallel durchgeführt werden.

Bei dem bekannten Verfahren zur parallelen Korrektur von 4-Bit Fehlern ist eine Gleichung vierten Grades im $GF(2^m)$ zu lösen. Dies erfolgt, indem eine Hilfsgleichung dritten Grades und vier Gleichungen zweiten Grades parallel gelöst werden. In der vorliegenden Arbeit wurde gezeigt, dass sich eine Gleichung zweiten Grades einsparen lässt, wodurch sich eine Vereinfachung der Hardware bei einer parallelen Realisierung der 4-Bit Korrektur ergibt.

Die erzielten Ergebnisse wurden durch umfangreiche Simulationen in Software und Hardwareimplementierungen überprüft.

5 Anhang

5.1 Anwendungsbereiche des vorgestellten Verfahrens

In diesem Abschnitt beschäftigen wir uns mit der Frage, welche Wortlängen und Fehlerwahrscheinlichkeiten für das vorgestellte Verfahren in Betracht kommen und welche Grenzen hier gelten.

Im vorgestellten Verfahren werden Fehler unter der Annahme korrigiert, dass Fehler mit einer geringen Anzahl fehlerhafter Bits wahrscheinlicher sind, als Fehler mit einer hohen Anzahl an fehlerhaften Bits. Im Folgenden wird gezeigt, dass diese Annahme nur bis zu bestimmten Kombinationen von Wortlängen und Fehlerwahrscheinlichkeit gilt.

Es wird weiterhin gezeigt, dass die Verteilung der Fehler sich bei hohen Werten für Wortlänge und Fehlerwahrscheinlichkeit verschiebt, wodurch z.B. ein 4-Bit Fehler wahrscheinlicher sein kann als eine fehlerfreie Übertragung.

5.1.1 Annahmen

In Bezug auf den in dieser Arbeit verwendeten BCH Code beschränkt sich die Betrachtung auf Fehler, welche Änderung des Bitwertes einzelner Bits darstellen. Dies bedeutet eine Änderung des Wertes von 0 zu 1 bzw. von 1 zu 0.

Es wird zudem angenommen, dass die betrachteten Bitfehler mit einer Fehlerwahrscheinlichkeit $p(e)$ auftreten, welche unabhängig von der Position des Bits ist.

Die Länge n der übertragenen Wörter wird als gleich für alle Wörter angenommen.

5.1.2 Wahrscheinlichkeitsverteilungen

Im Folgenden wird betrachtet, wie sich Fehlerauftritte anhand der Auftrittswahrscheinlichkeit $p(e)$ eines Fehlers an einer Bitstelle und der Wortlänge n verhalten. Es wird gezeigt, dass die grundlegende Annahme, dass niedrigere Fehleranzahlen wahrscheinlicher sind, als höhere Fehleranzahlen nicht allgemeingültig ist.

Der Programmcode, mit welchen die Berechnungen in diesem Abschnitt durchgeführt wurden, kann im Anhang gefunden werden.

Die Tabellen 5.1, 5.2, 5.3 (Script 5.1) zeigen die Auftrittswahrscheinlichkeit eines Fehlers entsprechend den verschiedenen Wortlängen. Ist n die Wortlänge, k die Fehleranzahl und $p(e)$ die Wahrscheinlichkeit eines Fehlers einer beliebigen Position, werden die Werte nach der Binomialverteilung wie folgt berechnet

$$\binom{n}{k} \times (p(e))^k \times (1 - p(e))^{n-k} \tag{5.1}$$

Tabelle 5.1 zeigt zu einem Fehler mit der Wahrscheinlichkeit $p(e) = 10^{-6}$ die Wahrscheinlichkeit, mit welcher verschiedene Fehleranzahlen auftreten. Die verschiedenen Spalten zeigen die verschiedenen Wortlängen, die Zeilen die verschiedenen Fehleranzahlen. 0-Bit steht für die Wahrscheinlichkeit, dass kein Fehler auftritt. Zur Übersicht werden nur bis 7 Prozentstellen angezeigt.

Als Beispiel in Spalte 2, Zeile 2 steht die Wahrscheinlichkeit 0.00320% eines genau 1-Bit Fehlers bei einer 32 Bit Wortlänge. Dieser Wert wird bestimmt durch

$$\binom{32}{1} \times (10^{-6})^1 \times (1-10^{-6})^{32-1} = 0.00320\% \tag{5.2}$$

Fehler	16	32	64	128	256	512	1024
0-Bit	99.99840%	99.99680%	99.99360%	99.98720%	99.97440%	99.94881%	99.89765%
1-Bit	0.00160%	0.00320%	0.00640%	0.01280%	0.02559%	0.05117%	0.10230%
2-Bit	0.00000%	0.00000%	0.00000%	0.00000%	0.00000%	0.00001%	0.00005%
3-Bit	0.00000%	0.00000%	0.00000%	0.00000%	0.00000%	0.00000%	0.00000%
4-Bit	0.00000%	0.00000%	0.00000%	0.00000%	0.00000%	0.00000%	0.00000%
Fehler	2^{15}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
0-Bit	96.77630%	35.04362%	12.28056%	1.50812%	0.02274%	0.00001%	0.00000%
1-Bit	3.17117%	36.74594%	25.75422%	6.32552%	0.19079%	0.00009%	0.00000%
2-Bit	0.05195%	19.26546%	27.00527%	13.26559%	0.80024%	0.00073%	0.00000%
3-Bit	0.00057%	6.73376%	18.87805%	18.54665%	2.23765%	0.00407%	0.00000%
4-Bit	0.00000%	1.76521%	9.89753%	19.44758%	4.69269%	0.01708%	0.00000%
5-Bit	0.00000%	0.37019%	4.15132%	16.31381%	7.87303%	0.05730%	0.00000%
6-Bit	0.00000%	0.06470%	1.45099%	11.40418%	11.00730%	0.16023%	0.00000%
7-Bit	0.00000%	0.00969%	0.43471%	6.83322%	13.19085%	0.38402%	0.00000%
8-Bit	0.00000%	0.00127%	0.11396%	3.58258%	13.83161%	0.80535%	0.00001%
9-Bit	0.00000%	0.00015%	0.02655%	1.66960%	12.89199%	1.50128%	0.00004%
10-Bit	0.00000%	0.00002%	0.00557%	0.70028%	10.81459%	2.51874%	0.00013%

Tabelle 5.1: Fehlerwahrscheinlichkeit n-Bit Fehler bei bestimmter Wortlänge für $p(e) = 10^{-6}$. Die Spalten sind die verschiedenen Wortlängen, die Zeilen die Fehleranzahlen.

Man kann hier erkennen, wie die Fehlerwahrscheinlichkeit, beispielsweise eines 1-Bit Fehlers, zunächst mit steigender Wortlänge wächst und ab einer gewissen Wortlänge wieder abnimmt. Die Tabelle zeigt, dass beispielsweise bei einer Wortlänge von 2^{20} die Wahrscheinlichkeit eines 1-Bit Fehler höher ist als die Wahrscheinlichkeit für eine fehlerfreie Übertragung. Bei einer Wortlänge von 2^{25} ist die Fehlerwahrscheinlichkeit für Übertragungen, die weniger als einen 8-Bit Fehler aufweisen, so gering, dass sie vernachlässigt werden kann.

Fehler	16	32	64	128	256	512	1024
0-Bit	18.53020%	3.43368%	0.11790%	0.00014%	0.00000%	0.00000%	0.00000%
1-Bit	32.94258%	12.20865%	0.83841%	0.00198%	0.00000%	0.00000%	0.00000%
2-Bit	27.45215%	21.02601%	2.93445%	0.01395%	0.00000%	0.00000%	0.00000%
3-Bit	14.23445%	23.36224%	6.73836%	0.06509%	0.00000%	0.00000%	0.00000%
4-Bit	5.14022%	18.81958%	11.41777%	0.22602%	0.00001%	0.00000%	0.00000%
5-Bit	1.37072%	11.70996%	15.22370%	0.62282%	0.00003%	0.00000%	0.00000%
6-Bit	0.27922%	5.85498%	16.63330%	1.41865%	0.00013%	0.00000%	0.00000%
7-Bit	0.04432%	2.41634%	15.31319%	2.74722%	0.00053%	0.00000%	0.00000%
8-Bit	0.00554%	0.83901%	12.12295%	4.61686%	0.00184%	0.00000%	0.00000%
9-Bit	0.00055%	0.24859%	8.38130%	6.83979%	0.00563%	0.00000%	0.00000%
10-Bit	0.00004%	0.06353%	5.12190%	9.04373%	0.01545%	0.00000%	0.00000%

Tabelle 5.2: Fehlerwahrscheinlichkeit n-Bit Fehler bei bestimmter Wortlänge für $p(e) = 0.1$.

Wie man an der Tabelle 5.2 erkennen kann, ist dies zudem von der Fehlerwahrscheinlichkeit abhängig. Je höher die Fehlerwahrscheinlichkeit, desto schneller verschieben sich die Verteilungen der Fehler. Bei einer Wortlänge von 16 Bit ist die Wahrscheinlichkeit eines genau 1-Bit Fehlers mit 32.94258% bereits höher als die einer fehlerlosen Übertragung. Die Formel dazu lautet $\binom{16}{1} \times (0.1)^1 \times (1-0.1)^{16-1} = 32.94258\%$.

Fehler	2	4	8	16	32	64
0-Bit	4.00000%	0.16000%	0.00026%	0.00000%	0.00000%	0.00000%
1-Bit	32.00000%	2.56000%	0.00819%	0.00000%	0.00000%	0.00000%
2-Bit	64.00000%	15.36000%	0.11469%	0.00000%	0.00000%	0.00000%
3-Bit	-	40.96000%	0.91750%	0.00002%	0.00000%	0.00000%
4-Bit	-	40.96000%	4.58752%	0.00031%	0.00000%	0.00000%
5-Bit	-	-	14.68006%	0.00293%	0.00000%	0.00000%
6-Bit	-	-	29.36013%	0.02150%	0.00000%	0.00000%
7-Bit	-	-	33.55443%	0.12284%	0.00000%	0.00000%
8-Bit	-	-	16.77722%	0.55276%	0.00000%	0.00000%
9-Bit	-	-	-	1.96538%	0.00000%	0.00000%
10-Bit	-	-	-	5.50306%	0.00000%	0.00000%

Tabelle 5.3: Fehlerwahrscheinlichkeit n-Bit Fehler bei bestimmter Wortlänge für $p(e) = 0.8$

Tabelle 5.3 zeigt hier einen Fall, in welchem die Wahrscheinlichkeit eines Fehlers pro Bit mit 80 Prozent sehr hoch liegt. Auch diese Tabelle zeigt eine derartige Verteilung, bei welcher hohe Fehleranzahlen wahrscheinlich sind. Man beachte, dass hier sehr geringe Wortlängen gezeigt werden, wodurch einige Fehler nicht auftreten können.

Abbildung 5.1 zeigt diese Daten als Graphen (Erzeugt durch Script 5.2). Jeder Graph steht für eine bestimmte Fehlerwahrscheinlichkeit pro Bit und stellt von links nach rechts die Wahrscheinlichkeit verschiedener Fehleranzahlen an. Jede Kurve stellt die Wahrscheinlichkeitsverteilung zu einer bestimmten Wortlänge dar. Die Verteilung der einzelnen Kurven nimmt ungefähr die Form

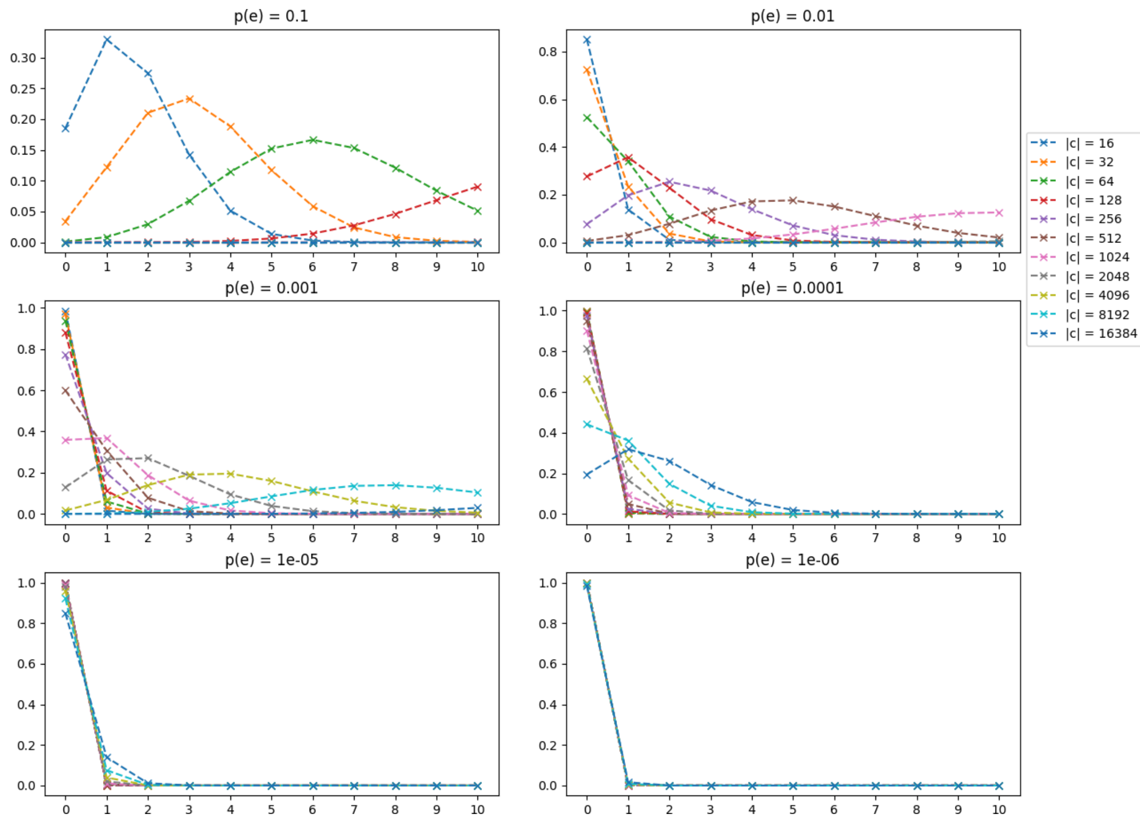

Abbildung 5.1: Graphen zu verschiedenen Fehlerwahrscheinlichkeiten, in welchen Kurven für unterschiedliche Wortlängen die Wahrscheinlichkeit der Fehler von 0 Bit bis 10 Bit dargestellt wird. Die Wahrscheinlichkeiten nehmen jeweils grob die Form einer Gaußschen Glockenkurve an.

einer Gaußschen Glockenkurve an, diese ist jedoch nur bei einer hohen Kombination aus Fehlerwahrscheinlichkeit und Wortlänge ablesbar. Ansonsten tritt eine Häufung der Werte bei dem 0-Bit Fehler auf. Diese Das Maximum dieser Verteilung wandert mit steigender Wortlänge in Richtung der höheren Fehlerzahlen.

Die Abbildung 5.2 zeigt diese Daten auf eine andere Art (Skript 5.3). In dieser wird für die verschiedenen Wortlängen 16, 32, 64, 128, 256, 512, 1024 und 2048 Bit dargestellt, welche Bitfehler akkumuliert ab 0-Bit zusammen 99.95% aller Fehler darstellen. Pro Wortlänge sind die verschiedenen Fehlerwahrscheinlichkeiten des Kanals durch Farben getrennt dargestellt. Ein Balken stellt also dar, dass wenn für eine gewisse Fehlerwahrscheinlichkeit und Wortlänge Fehlerwahrscheinlichkeiten aufsummiert werden, ab welchem aufaddierten Fehler die 99.95% überschritten werden. In dieser wird somit dargestellt, bis zu welcher Bitanzahl Fehler zu erwarten sind. Unter der Annahme, dass 99.95 Prozent aller Fehler erkannt werden sollen, ergeben beispielsweise bei einer unabhängigen Fehlerwahrscheinlichkeit von $p(e) = 0.1$ bei einer Wortlänge von 16 Bit die Bitfehler von 0-Bit bis 8-Bit zusammen die geforderten 99.95% aller Fälle. Alle möglichen Fehler mit 9 Bit oder mehr können somit nur weniger als 0.05% ergeben und werden daher vernachlässigt. In Abbildung 5.2 ist nun beispielsweise ersichtlich, dass bei $p(e) = 10^{-6}$ Wortlängen bis inklusive 256 Bit nur 0-Bit und 1-Bit Fehler zu betrachten sind. Bei den Wortlängen 512, 1024 und 2048

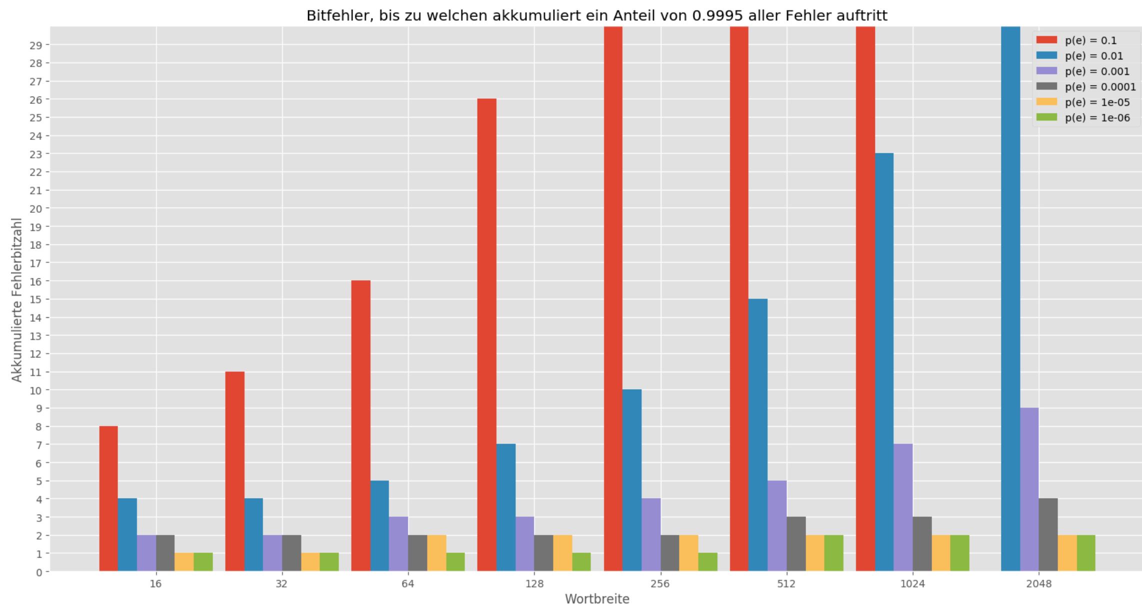

Abbildung 5.2: Kumulative Bitfehler ab 0-Bit Fehlern, welche zusammen mit 99.95% auftreten. Die horizontale Skala gruppiert für Wortlängen jeweils die Auftrittswahrscheinlichkeiten eines einzelnen Fehlers. Die vertikale Skala gibt die Anzahl der Fehler an, welche von 0-Bit beginnend zusammen mindestens 99.95% besitzen. Der Graph ist aus Platzgründen bei 30-Bit abgeschnitten. Der Fall $p(e) = 0.1$ für 2048 wurde hier wegen einer Zahlenbereichsüberschreitung (Skriptfehler) nicht dargestellt, würde aber ebenfalls die Höhe des Graphen ausfüllen.

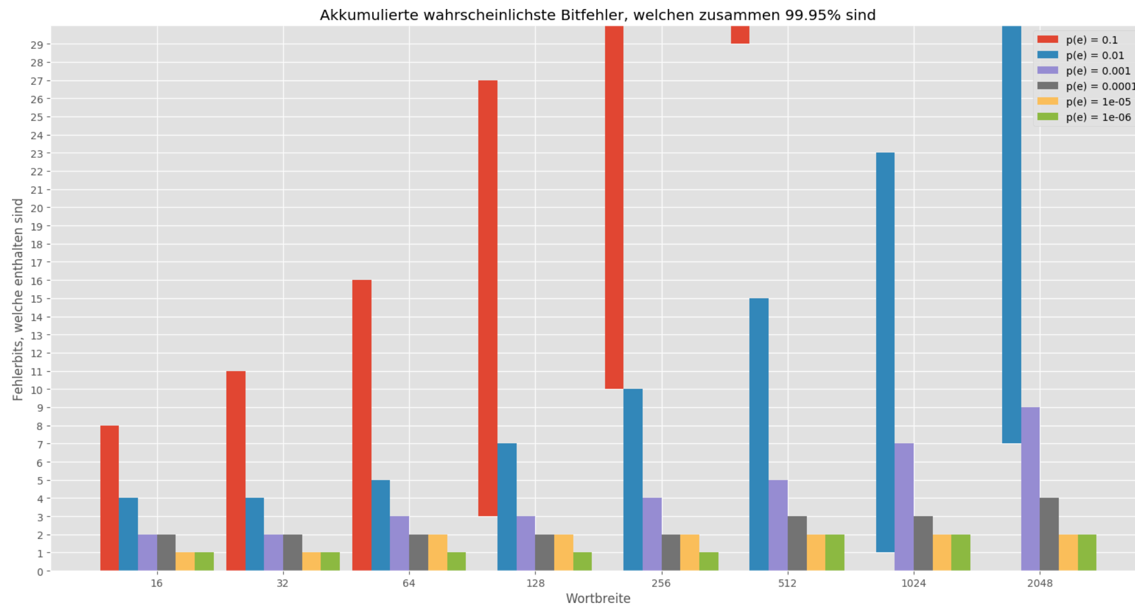

Abbildung 5.3: Zu verschiedenen Wortlängen und Fehlerwahrscheinlichkeiten wir hier gezeigt, welche Bitfehler mit höchster Wahrscheinlichkeit zusammen 99.95% aller auftreten Fälle umfassen.

sind zusätzlich die 2-Bit Fehler zu erwarten.

Abbildung 5.3 zieht nun in Betracht, dass 0-Bit Fehler ggf. unwahrscheinlich sind. (Skript 5.4). In dieser werden die Fehler entsprechend den höchsten Wahrscheinlichkeiten akkumuliert, welche bis diese mindestens 99.95% ausmachen. Sind die Auftrittswahrscheinlichkeit für fehlerfreie Übertragungen bzw. niedrige Fehleranzahlen gering, kann es vorkommen, dass diese nicht in den wahrscheinlichsten 99.95% der Fehler enthalten sind und somit nicht in den abgebildeten Balken enthalten sind. Dies ist in den Fällen zu sehen, in den ein Balken nicht bei dem 0-Bit Fehler beginnt, sondern erst bei einem höheren Fehler. Dabei zeigt sich erneut, dass es entsprechend der Wortlänge und Fehlerwahrscheinlichkeit pro Bit Fälle auftreten, bei welchen nicht mit einer fehlerfreien Übertragung zu rechnen ist.

5.1.3 Betrachtung des vorgestellten Korrekturansatzes

Im vorgestellten Verfahren besitzen Codeworte gewisse Hammingdistanzen zueinander. Tritt ein Fehler auf, würde das Codewort mit der geringsten Hammingdistanz zum Fehlerhaften Wort gesucht werden. Ist die Distanz im Korrekturbereich, würde zum nächsten Codewort korrigiert werden, ansonsten würde ein erkannter Fehler ausgegeben werden.

Damit Fehler korrigiert oder erkannt werden, müssen diese kleiner als die größte Anzahl generell erkennbarer Fehlerstellen. Dies ist wahrscheinlich, wenn die akkumulierte Wahrscheinlichkeit bis inklusive dieses größten generell erkennbaren Fehlers nahezu 100% ausmacht. Liegt der überwiegende Prozentteil der Fehlerwahrscheinlichkeit über dieser Anzahl, so kann es passieren, dass Fehler falsch korrigiert oder nicht erkannt werden.

Während eine hohe Codelänge Korrekturbits einspart und ermöglicht, mehr Bits zu korrigieren, erhöht sie die Wahrscheinlichkeit für mehr Fehler in einem übertragenen Wort. Es hängt daher

vom Anwendungsfall ab, welche Häufigkeit von fehlerhaften Übertragungen bzw. welcher Korrekturaufwand akzeptabel ist. Die Anzahl der zu korrigierenden Bits und Wortlänge kann dazu problemspezifisch gewählt werden.

Soll für einen bestimmten Übertragungskanal eine Kodierung vorgenommen werden, kann wie folgt verfahren werden:

1. Gegeben die Bitfehlerwahrscheinlichkeit $p(e)$ des Kanals, wähle eine Wortlänge n.
2. Berechne über steigende Fehleranzahlen k nach der Binomialverteilung $\binom{n}{k} \times (p(e))^k \times (1-p(e))^{n-k}$ die Fehlerwahrscheinlichkeiten.
3. Entscheide anhand der Auftrittswahrscheinlichkeiten für bestimmte Fehler, wie viele Bits korrigiert und wie viele erkannt werden sollen.

Dieses Vorgehen kann geprüft werden, wie die Auftrittswahrscheinlichkeit bestimmter Fehler eines Codes ist. Dies ermöglicht, die Anzahl der Korrekturbits auf die Wahrscheinlichkeit der Fehler anzupassen.

5.2 Skripte zur Erstellung von Graphen

```
import math
def bincoeff(x,y): # from x draw y
    if y == x:
        return 1
    elif y == 1:
        return x
    elif y > x:
        return 0
    else:
        a = math.factorial(x)
        b = math.factorial(y)
        c = math.factorial(x-y)
        return a // (b * c)

data={}
for p in [0.8,0.7,0.6,0.5,0.4,0.3,0.2,0.1,0.01,0.001,0.0001,10**-5,10**-6]:
    print "p=",p
    rp = 1-p
    data[p]={}
    for n in [16,32,64,128,256,512,1024,2048,4096]:
        #print "n=",n
        for b in [0,1,2,3,4,5,6,7,8,9,10]:
            bc = bincoeff(n,b)
            wp = (p ** b)
            try:
                wrp = (rp ** (n-b))
            except:
                wrp = 0
            res = bc * wp * wrp
            #print "b=",b,"->",res
            data[p][b].append(res)

    keys = data[p].keys()
    keys.sort()
    print "   \t"+", ".join(["16","32","64","128","256","512","1024","2048","4096"])
    for x in keys:
        print str(x)+":\t"+", ".join([str("{:.10%}".format(res)) for res in data[p][x]])
    print ""
```

Listing 5.1: Skript für die Tabellen verschiedener Fehler bei gegebener Fehlerwahrscheinlichkeit eines 1-Bit Fehler mit verschiedenen Wortlängen

```
1  import math
2  import matplotlib.pyplot as plt
3  prob = [0.1,0.01,0.001,0.0001,10**-5,10**-6]
4  wlen = [16,32,64,128,256,512,1024,2048,4096,8192,16384]
5  errs = [0,1,2,3,4,5,6,7,8,9,10]
6  def bincoeff(x,y): # from x draw y
7      if y == x:
8          return 1
9      elif y == 1:
10         return x
11     elif y > x:
12         return 0
13     else:
14         a = math.factorial(x)
15         b = math.factorial(y)
16         c = math.factorial(x-y)
17         return a // (b * c)
18 i=1
19 revdata={}
20 for p in prob:
21     print "p=",p
22     rp = 1-p
23     revdata[p]={}
24     for n in wlen:
25         res = []
26         for b in errs:
27             bc = bincoeff(n,b)
28             wp = (p ** b)
29             try:
30                 wrp = (rp ** (n-b))
31             except:
32                 wrp = 0
33             res.append(bc * wp * wrp)
34         revdata[p][n] = res
35
36     keys = revdata[p].keys()
37     keys.sort()
38     print "  \t"+", ".join([str(x) for x in errs])
39     for x in keys:
40         print str(x)+":\t"+", ".join([str("{:.10%}".format(res)) for res in
               revdata[p][x]])
41     print ""
42     fig = plt.subplot(3,2,i)
43     i+=1
44     plt.title("p(e) = "+str(p))
45     for x in keys:
46         vec = revdata[p][x]
47         plt.plot(errs,vec,'x--',label="|c| = "+str(x))
48     plt.xticks(range(0,11))
49 plt.legend(loc='upper left',bbox_to_anchor=(1,3))
50 plt.show()
```

Listing 5.2: Skript für die Erstellung von Graphen zu den Fehlerverteilungen

```
1  import math
2  import matplotlib.pyplot as plt
3  plt.style.use('ggplot')
4  prob = [0.1,0.01,0.001,0.0001,10**-5,10**-6]
5  wlen = [16,32,64,128,256,512,1024,2048]
6  wlen_str = [str(x) for x in wlen]
7  req_perc = 0.9995
8  max_errors = 220 # There may be an Overflow error for higher values!
9  max_shown_errors = 30
10
11 def bincoeff(x,y): # from x draw y
12     if y == x:
13         return 1
14     elif y == 1:
15         return x
16     elif y > x:
17         return 0
18     else:
19         a = math.factorial(x)
20         b = math.factorial(y)
21         c = math.factorial(x-y)
22         return a // (b * c)
23
24 i=1
25 width=0.15
26 revdata={}
27 values={}
28 for p in prob:
29     print "p=",p,"\tbenoetigt ",req_perc
30     rp = 1-p
31     revdata[p]={}
32     values[p]=[]
33     maxerrs = -1
34     broken = False
35
36     for n in wlen:
37         #print "n=",n
38         num_errs=0
39         acc_perc=0
40         if not broken: maxerrs += 1
41         while acc_perc < req_perc and num_errs < max_errors:
42             bc = bincoeff(n,num_errs)
43             wp = (p ** num_errs)
44             try:
45                 wrp = (rp ** (n-num_errs))
46             except:
47                 wrp = 0
48             acc_perc += bc * wp * wrp
49             num_errs += 1
50         if num_errs < max_errors:
51             revdata[p][n] = num_errs
52             values[p].append(num_errs)
53         else:
54             revdata[p][n] = -1
```

```
55                  values[p].append(-1)
56                  broken = True
57          if not broken: maxerrs += 1
58
59          keys = revdata[p].keys()
60          keys.sort()
61          for x in keys:
62              print str(x)+":\t"+str(revdata[p][x])
63          print ""
64          pos = [v+width*i for v in range(maxerrs)]
65          i+=1
66          plt.bar(pos,values[p][0:maxerrs],width,label="p(e) = "+str(p))
67  plt.title("Bitfehler, bis zu welchen akkumuliert "+str("{:.2%}".format(
        req_perc))+" aller Fehler auftritt")
68  plt.ylabel('Akkumulierte Fehlerbitzahl')
69  plt.xlabel('Wortbreite')
70  pos = [v+width*3.5 for v in range(len(wlen_str))]
71  plt.ylim(top=max_shown_errors)
72  plt.yticks(range(max_shown_errors))
73  plt.xticks(pos, wlen_str)
74  plt.legend(loc='best')
75  plt.show()
```

Listing 5.3: Skript für die Erstellung von Graphen zu akkumulierten Fehlerverteilungen

```
1  import math
2  import matplotlib.pyplot as plt
3  plt.style.use('ggplot')
4  prob = [0.1,0.01,0.001,0.0001,10**-5,10**-6]
5  wlen = [16,32,64,128,256,512,1024,2048]
6  wlen_str = [str(x) for x in wlen]
7  req_perc_max = 0.9999
8  req_perc = 0.9995
9  max_errors = 220 # There may be an Overflow error for higher values!
10 max_shown_errors = 30
11
12 def bincoeff(x,y): # from x draw y
13     if y == x:
14         return 1
15     elif y == 1:
16         return x
17     elif y > x:
18         return 0
19     else:
20         a = math.factorial(x)
21         b = math.factorial(y)
22         c = math.factorial(x-y)
23         return a // (b * c)
24
25 i=1
26 width=0.15
27 revdata={}
28 values={}
29 bottoms={}
30 for p in prob:
31     print "p=",p,"\tbenoetigt ",req_perc
32     rp = 1-p
33     revdata[p]={}
34     values[p]=[]
35     bottoms[p]=[]
36     maxerrs = -1
37     broken = False
38
39     for n in wlen:
40         num_errs=0
41         res = {}
42         acc_perc=0
43         while acc_perc < req_perc_max and num_errs < max_errors:
44             bc = bincoeff(n,num_errs)
45             wp = (p ** num_errs)
46             try:
47                 wrp = (rp ** (n-num_errs))
48             except:
49                 wrp = 0
50             val = bc * wp * wrp
51             acc_perc += val
52             res[num_errs]=val
53             num_errs += 1
54
```

```
55            sort_res = sorted(res.items(), key=lambda item: item[1]) #Sort by
                  keys ascending
56            fin = []
57            acc_perc=0
58            while acc_perc < req_perc and len(sort_res)>0:
59                merr,mperc = sort_res.pop() # Get last (biggest)
60                fin.append(merr)
61                acc_perc += mperc
62            if len(sort_res)>0 or acc_perc >= req_perc:
63                els = sorted(fin)
64                revdata[p][n] = els
65                if len(els) != els[-1]-els[0]+1:
66                    values[p].append(-1)
67                    bottoms[p].append(els[0])
68                else:
69                    values[p].append(len(els))
70                    bottoms[p].append(els[0])
71            else:
72                revdata[p][n] = 0
73                values[p].append(0)
74                bottoms[p].append(0)
75        keys = revdata[p].keys()
76        keys.sort()
77        myid=0
78        for x in keys:
79            if type(revdata[p][x]) is list:
80                print str(x)+":\t"+str(revdata[p][x][0])+".."+str(revdata[p][x
                      ][-1])
81            else:
82                print str(x)+":\t Not Calculated!"
83            myid+=1
84        print ""
85
86        pos = [v+width*i for v in range(len(values[p]))]
87        i+=1
88        plt.bar(pos,values[p],width,label="p(e) = "+str(p),bottom=bottoms[p])
89    plt.title("Akkumulierte wahrscheinlichste Bitfehler, welchen zusammen "+str("
        {:.2%}".format(req_perc))+" sind")
90    plt.ylabel('Fehlerbits, welche enthalten sind')
91    plt.xlabel('Wortbreite')
92    pos = [v+width*3.5 for v in range(len(wlen_str))]
93    plt.ylim(top=max_shown_errors)
94    plt.yticks(range(max_shown_errors))
95    plt.xticks(pos, wlen_str)
96    plt.legend(loc='best')
97    plt.show()
```

Listing 5.4: Skript für die Erstellung von Graphen zu genauen akkumulierten Fehlerverteilungen

5.3 Python Skript für BCH Codes

```
1  #!/usr/bin/env python
2
```

```
3   import json, sys
4   import copy
5
6   alpha_table = []
7
8   # Full modulo multiplication, only use for generating the alpha table!
9   def mult_full(a,b,p):
10      la=len(a)
11      lb=len(b)
12      res = [0]*(la+lb-1)
13      for i in range(lb):
14          if b[i] == 1:
15              for j in range(la):
16                  if a[j] == 1:
17                      res[i+j] = 1 - res[i+j]
18      for i in range(la-1):
19          if res[i]==1:
20              for j in range(len(p)):
21                  if p[j]=='1': res[j+i] = 1 - res[j+i]
22      res = res[la-1:]
23      return res
24
25  def build_alpha_table(p):
26      global alpha_table
27      alpha_table = []
28      a = [0]*(len(p)-1)
29      b = [0]*(len(p)-1)
30      a[-1] = 1
31      b[-2] = 1
32      alpha_table.append(a)
33      count = 2**(len(p)-1)
34      for i in range(1,count-1):
35          a = mult_full(a,b,p)
36          alpha_table.append(a)
37
38  def get_exponent_table(v):
39      global alpha_table
40      if v in alpha_table:
41          return alpha_table.index(v)
42      return None
43
44  def get_value_table(e):
45      global alpha_table
46      if e is None: return [0 for x in alpha_table[0]]
47      c = e % len(alpha_table)
48      return alpha_table[c]
49
50  def add(*items):
51      if len(items) == 0: return 0
52      a = get_value_table(items[0]) if items[0] is None or type(items[0]) is
            int else items[0]
53      for item in items[1:]:
54          b = get_value_table(item) if item is None or type(item) is int else
                item
55          a = [ 0 if a[x]==b[x] else 1 for x in range(len(a))]
```

```
56      return a
57
58  def add_two(a,b):# len(a) == len(b) required
59      if a is None or b is None: return None
60      return [ 0 if a[x]==b[x] else 1 for x in range(len(a))]
61
62  def mult(*items):
63      if len(items) == 0: return 0
64      a = items[0] if items[0] is None or type(items[0]) is int else
            get_exponent_table(items[0])
65      if a is None: return get_value_table(None)
66      for item in items[1:]:
67          b = item if item is None or type(item) is int else get_exponent_table
                (item)
68          if b is None: return get_value_table(None)
69          a += b
70      return get_value_table(a)
71
72  def mult_two(a,b):
73      if a is None or b is None: return None
74      if not 1 in a: return a
75      if not 1 in b: return b
76      ea = get_exponent_table(a)
77      eb = get_exponent_table(b)
78      try: ne = ea + eb
79      except:
80          print "Error! Multiplication of "+str(a)+"*"+str(b)+" failed because
                of non-primitive polynom!"
81          sys.exit(2)
82      return get_value_table(ne)
83
84  def div(a,b):
85      if a is None or b is None: return None
86      if not 1 in a: return a # 0 / X
87      if not 1 in b
88          return None
89      ea = get_exponent_table(a)
90      eb = get_exponent_table(b)
91      try: ne = ea - eb
92      except:
93          print "Error! Division of "+str(a)+"/"+str(b)+" failed because of non
                -primitive polynom!"
94          sys.exit(2)
95      return get_value_table(ne)
96
97  def exp(a,b):
98      if a is None or b is None: return None
99      if not 1 in a: return a # 0 ^ b
100     ea = get_exponent_table(a)
101     if ea is None:
102         print "Error! Polynom might be non-primitive!"
103         sys.exit(2)
104     if type(b) == int:
105         return get_value_table(ea*b)
106     elif type(b) == float:
```

```
107            res_exp = float(ea)*b
108            if res_exp.is_integer():
109                return get_value_table(int(res_exp))
110            else:
111                modu = 2**len(a)-1
112                modx = float(modu)*b
113                res_expn = res_exp
114                while (modx > 0 and res_expn < modu and res_expn >= 0):
115                    res_expn += modx
116                    if res_expn.is_integer():
117                        return get_value_table(int(res_expn))
118                return get_value_table(int(res_exp))
119
120  def sqrt(a):
121      if not 1 in a: return a
122      ea = get_exponent_table(a)
123      if ea is None:
124          print "Error! Polynom might be non-primitive!"
125          sys.exit(2)
126      if ea % 2 == 0:
127          return get_value_table(ea/2)
128      else:
129          modu = 2**len(a)-1
130          nv = (ea + modu)/2
131          return get_value_table(nv)
132
133  def sqrt3(a):
134      if not 1 in a: return a
135      ea = get_exponent_table(a)
136      if ea is None:
137          print "Error! Polynom might be non-primitive!"
138          sys.exit(2)
139      if ea % 3 == 0:
140          return get_value_table(ea/3)
141      else:
142          modu = 2**len(a)-1
143          if (ea + modu) % 3 == 0:
144              nv = (ea + modu)/3
145              return get_value_table(nv)
146          else:
147              nv = (ea + 2*modu)/3
148              return get_value_table(nv)
149
150  def isZero(a):
151      return not 1 in a
152
153  def to_ex(a):
154      return get_exponent_table(a)
155
156  def to_val(a):
157      return get_value_table(a)
158
159  def int_to_bin(a, vector_length=5):
160      return ("{0:0"+str(vector_length)+"b}").format(a)[::-1]
161
```

```
162 def load_json(filename):
163     fp = open(filename)
164     text = fp.read()
165     fp.close()
166     data = json.loads(text)
167     return data
168
169 def alpha_matrix_from_h(h,cb,sl):
170     le = len(h[0])
171     res = [ [ [ h[i*sl+j][k] for j in range(sl) ] for k in range(le) ] for i
            in range(cb) ]
172     return res
173
174 def print_h(h,rm,cm):
175     row_count = 0
176     for row in h:
177         line = ""
178         col_count = 0
179         for col in row:
180             if col_count >= cm:
181                 line += " "
182                 col_count = 0
183             line += str(col)+" "
184             col_count += 1
185         if row_count >= rm:
186             print ""
187             row_count = 0
188         print "\t"+line
189         row_count += 1
190
191 def build_polynom_table(polynom,pl,length):
192     cur = [0]*pl
193     cur[-1] = 1
194     pt = [cur]
195     for i in range(length-1):
196         next = cur[1:]+[0]
197         if cur[0] == 1:
198             for x in range(pl):
199                 next[x] = (next[x] + polynom[x+1]) % 2
200         cur = next
201         pt.append(cur)
202     for x in pt[1:]:
203         if x == pt[0]:
204             print polynom," is not primitive."
205     return pt
206
207 def calc_c_3_bit(synd):
208     synd_add = add(exp(synd[0],3),synd[1])
209     sigma_1 = synd[0]
210     sigma_2 = div(add(mult(exp(synd[0],2),synd[1]),synd[2]),synd_add)
211     # (s_1^2*s_3 + s_5) / (s_1^3 + s_3)
212     sigma_3 = add(add(exp(synd[0],3),synd[1]),mult(synd[0],sigma_2))
213     # s_1^3 + s_3 + s_1 * sigma_2
214     n = add(exp(sigma_1,2),sigma_2)
215     delta = add(mult(synd[0],sigma_2),sigma_3)
```

```
216        c = div(delta,sqrt(exp(n, 3)))
217
218        return c
219
220    def reduce_table(table):
221        keystab = table[0].keys()
222        keys = [json.loads(x) for x in keystab]
223        print "\n".join([str(x) for x in keys])
224        max_len = len(keys[0])
225        reduced_keys = keys
226        is_unique = len(reduced_keys) == len(set([str(x) for x in reduced_keys]))
227        def rec_red(t,l,reduce_list=[]):
228            if len(t) < 1: return []
229            if len(t) != len(set([str(x) for x in t])): return []
230            if l == 1: return [([t],reduce_list)]
231            result = []
232            for i in range(l):
233                st = copy.deepcopy(t)
234                for x in st: x.pop(i)
235                if len(st) != len(set([str(x) for x in st])):
236                    continue
237                else : result += rec_red(st,l-1,reduce_list+[i])
238            if len(result) == 0: return [([t],reduce_list)]
239            return result
240        reduced = rec_red(keys,max_len)
241        print reduced
242        print "\n".join([str(x) for x in reduced])
243        reduce_lines = None
244        best_keys = None
245        for (red,li) in reduced:
246            if len(red) < max_len:
247                max_len = len(red)
248                reduce_lines = li
249                best_keys = red
250        if best_keys is not None:
251            new_table = {}
252            for x,i in enumerate(keystab):
253                 new_table[str(x)] = best_keys[i]
254            table = [new_table,]
255        return table,reduce_lines
256
257    def generate_3_bit_table(s_polynom,data,verbose=False):
258        h = data["H"]
259        cb = data["checksyndroms"]
260        sl = data["syndromlength"]
261        pol = data["polynom"]
262        if sl < 2:
263            print "Cannot create C_table if there is no s_3"
264            sys.exit(1)
265        cwl = len(h[0])
266        build_alpha_table(pol)
267        dh = alpha_matrix_from_h(h,cb,sl)
268        table = {}
269        reverse_table = [{}]
270        endit = False
```

```
271         for i in range(cwl-2):
272             if endit: break
273             for j in range(i+1,cwl-1):
274                 if endit: break
275                 for k in range(j+1,cwl):
276                     synd = [add(add(dh[l][i],dh[l][j]),dh[l][k]) for l in range(
                            len(dh))]
277                     check_index = 1
278                     check_value = 1 + check_index*2 # 3
279                     tab = reverse_table[check_index-1]
280                     synd_add = add(exp(synd[0],3),synd[1])
281                     sigma_1 = synd[0]
282                     sigma_2 = div(add(mult(exp(synd[0],2),synd[1]),synd[2]),
                            synd_add)
283                     # (s_1^2*s_3 + s_5) / (s_1^3 + s_3)
284                     sigma_3 = add(add(exp(synd[0],3),synd[1]),mult(synd[0],
                            sigma_2))
285                     # s_1^3 + s_3 + s_1 * sigma_2
286                     n = add(exp(sigma_1,2),sigma_2)
287                     delta = add(mult(synd[0],sigma_2),sigma_3)
288                     c = div(delta,sqrt(exp(n, 3)))
289                     alpha_i = get_value_table(i)
290                     alpha_j = get_value_table(j)
291                     alpha_k = get_value_table(k)
292                     z_i = div(add(alpha_i,sigma_1),sqrt(n))
293                     z_j = div(add(alpha_j,sigma_1),sqrt(n))
294                     z_k = div(add(alpha_k,sigma_1),sqrt(n))
295                     if verbose: print "i,j,k =",i,j,k," -> c =",c,
                            get_exponent_table(c)
296                     if str(c) in tab:
297                         tab_res = tab[str(c)]
298                     else:
299                         tab[str(c)] = [z_i,z_j,z_k]
300         return reverse_table
301
302     def generate_2_bit_table(s_polynom,data):
303         h = data["H"]
304         cb = data["checksyndroms"]
305         sl = data["syndromlength"]
306         pol = data["polynom"]
307         if sl < 2:
308             print "Cannot create C_table if there is no s_3"
309             sys.exit(1)
310         cwl = len(h[0])
311         build_alpha_table(pol)
312         dh = alpha_matrix_from_h(h,cb,sl)
313         table = {}
314         reverse_table = [{}]
315         for i in range(cwl-1):
316             for j in range(i+1,cwl):
317                 synd = [add(dh[k][i],dh[k][j]) for k in range(len(dh))]
318                 check_index = 1
319                 check_value = 1 + check_index*2 # 3
320                 tab = reverse_table[check_index-1]
321                 # Key of Table: C = s_3 / s_1^3
```

```
322                 sexp = exp(synd[0],check_value) # s_1 ^ 3
323                 sdiv = div(synd[check_index],sexp) # s_3 / s_1^3
324                 # Value of Table: z_1, z_2 = (a^i / s_1) (a^j / s_1)
325                 # Calculating inside exponents y_i = i - i(s_1)
326                 alph_s1 = get_exponent_table(synd[0])
327                 y_i = (i-alph_s1)%len(alpha_table)
328                 y_j = (j-alph_s1)%len(alpha_table)
329                 if str(sdiv) in tab and str(tab[str(sdiv)]) not in [str(y_i),str(
                        y_j)]:
330                     print "Overwriting Key",str(sdiv),"from",tab[str(sdiv)],"to",
                            y_i
331                 if y_i < y_j: tab[str(sdiv[:-1])] = y_i
332                 else: tab[str(sdiv[:-1])] = y_j
333         return reverse_table
334 
335     def construct_h_matrix(polynom,num_corr):
336         if type(polynom) != list:
337             polynom = polynom.tolist()[0]
338         pl = len(polynom)-1
339         print "Modular Polynom length = ",pl
340         length = 2**pl - 1
341         print "Length of code = ",length
342         message_len = length - num_corr * pl
343         print "Message length = ",message_len
344         pt = build_polynom_table(polynom,pl,length)
345         result = []
346         used_mults = []
347         factor = 1
348         for i in range(num_corr):
349             lines = []
350             for n in range(length):
351                 ind = (n)*factor % length
352                 r = pt[ind]
353                 row = 0
354                 for el in r:
355                     if len(lines) > row:
356                         lines[row].append(el)
357                     else:
358                         lines.append([el,])
359                     row += 1
360             result += lines
361             if num_corr != i:
362                 if factor == 1:
363                     factor = 2
364                 used_mults.append(factor)
365                 for factor_new in range(factor+1,length):
366                     found = False
367                     for x in used_mults:
368                         mf = x*2
369                         while mf != 0 and mf < factor_new:
370                             mf = mf * mf
371                         if mf == factor_new:
372                             found = True
373                             break
374                     if not found:
```

```
375                     break
376                 factor = factor_new
377         return result
378 
379 def all_to_json_file(data):
380     result = ""
381     result += "{\n \"H\":[ \n\t"+",\n\t".join([str(x) for x in data["H"]])+"\
            n]," + "\n"
382     if "reverse_table" in data: result += "\"reverse_table\":[\n\t"+ \
383     ",\n".join("{ \n\t"+",\n\t".join(["\""+str(x)+"\":"+str(rev_elem[x]) for
            x in rev_elem])+"\n}" for rev_elem in data["reverse_table"])+ \
384     "\n]," + "\n"
385     result += "\"reverse_table_3\":[\n\t"+ \
386     ",\n".join("{ \n\t"+",\n\t".join(["\""+str(x)+"\":"+str(rev_elem[x]) for
            x in rev_elem])+"\n}" for rev_elem in data['reverse_table_3'])+ \
387     "\n]," + "\n"
388     result += " \"checksyndroms\":"+str(data["checksyndroms"])+"," + "\n"
389     result += " \"syndromlength\":"+str(data["syndromlength"])+"," + "\n"
390     result += " \"polynom\":\""+str(data["polynom"])+"\"" + "\n"
391     result += "}" + "\n"
392     save = True
393     if save:
394         try:
395             fp = open(str(data["polynom"])+"_rt3.json",'w')
396             fp.write(result)
397             fp.close()
398             print "Written to "+str(data["polynom"])+"_rt.json"
399         except:
400             print "Could not write "+data["polynom"]+"_rt.json"
401             print result
402     else:
403         print result
404 
405 if __name__ == "__main__":
406     if len(sys.argv) >= 2:
407         s_polynom = sys.argv[1]
408     else:
409         s_polynom = raw_input("Polynom: ")
410     if len(sys.argv) >= 3:
411         num_corr = sys.argv[2]
412     else:
413         num_corr = int(raw_input("Correcting N bits: "))
414     pol = [int(x) % 2 for x in s_polynom]
415     f1 = len(pol)-1
416     print "Gallois Feld 2 ^",f1
417     length = 2**f1 - 1
418     maxcorr = length / f1
419     mat = construct_h_matrix(pol,num_corr)
420     print "H = "
421     print_h(mat,5,5)
422     data = {
423         'H' : mat,
424         'checksyndroms' : num_corr,
425         'syndromlength' : f1,
426         'polynom' : s_polynom
```

```
427        }
428        data['reverse_table'] = generate_2_bit_table(s_polynom, data)
429        print "2-Bit Reverse Table:"
430        print "\"reverse_table\":[\n\t"+ \
431        ",\n".join("{ \n\t"+",\n\t".join(["\""+str(x)+"\":"+str(rev_elem[x]) for
               x in rev_elem])+"\n}" for rev_elem in data["reverse_table"])+ \
432        "\n]," + "\n"
433        reverse_table_3 = generate_3_bit_table(s_polynom, data)
434        print "\"reverse_table_3\":[\n\t"+ \
435        ",\n".join("{ \n\t"+",\n\t".join(["\""+str(x)+"\":"+str(rev_elem[x]) for
               x in rev_elem])+"\n}" for rev_elem in reverse_table_3)+ \
436        "\n]," + "\n"
437        data['reverse_table_3'] = reverse_table_3
438        print "3-Bit Reverse Table:"
439        print "\"reverse_table_3\":[\n\t"+ \
440        ",\n".join("{ \n\t"+",\n\t".join(["\""+str(x)+"\":"+str(rev_elem[x]) for
               x in rev_elem])+"\n}" for rev_elem in data["reverse_table_3"])+ \
441        "\n]," + "\n"
442        all_to_json_file(data)
```

Listing 5.5: Skript zur Erstellung von BCH Codes in Python

5.4 Vergleichsimplementierungen

```
1  --  log(i) in GF(2^5)
2
3  library IEEE;
4  use IEEE.STD_LOGIC_1164.ALL;
5
6  --  Defines a design entity
7  entity determinants is
8   Port (
9           signal s1 : in bit_vector(0 to 4);
10          signal s3 : in bit_vector(0 to 4);
11          signal s5 : in bit_vector(0 to 4);
12          signal s7 : in bit_vector(0 to 4);
13          signal s9 : in bit_vector(0 to 4);
14 --       signal det3 : out bit_vector(0 to 4);
15 --       signal det4 : out bit_vector(0 to 4);
16 --       signal det5 : out bit_vector(0 to 4)
17      signal is_onebit : OUT boolean
18  );
19 end determinants;
20
21 architecture behavioral of determinants is
22
23 function mult(v1,v2 : in bit_vector) return bit_vector is
24 constant m : integer := 5;
25 variable dummy : bit;
26 variable v_temp : bit_vector(0 to 4);
27 variable ret : bit_vector(0 to 4);
28
29 begin
```

```
30        v_temp := (others=>'0');
31        for i in 0 to m-1 loop
32            dummy := v_temp(0);
33            v_temp(0) := v_temp(1);
34            v_temp(1) := v_temp(2) xor dummy;
35            v_temp(2) := v_temp(3);
36            v_temp(3) := v_temp(4);
37            v_temp(4) := dummy;
38            for j in 0 to m-1 loop
39                v_temp(j) := v_temp(j) xor (v1(j) and v2(i));
40            end loop;
41        end loop;
42        ret := v_temp;
43        return ret;
44    end mult;
45
46    function add(v1,v2 : in bit_vector) return bit_vector is
47    constant m : integer := 4;
48    variable ret : bit_vector(0 to 4);
49
50    begin
51        for i in m downto 0 loop
52            ret(i) := v1(i) xor v2(i);
53        end loop;
54        return ret;
55    end add;
56
57    --old outdated
58
59    function mult_old(v1,v2 : in bit_vector) return bit_vector is
60    constant m : integer := 5;
61    variable dummy : bit;
62    variable v_temp : bit_vector(0 to 4);
63    variable ret : bit_vector(0 to 4);
64
65    begin
66        v_temp := (others=>'0');
67        for i in 0 to m-1 loop
68            dummy := v_temp(4);
69            v_temp(4) := v_temp(3);
70            v_temp(3) := v_temp(2) xor dummy;
71            v_temp(2) := v_temp(1);
72            v_temp(1) := v_temp(0);
73            v_temp(0) := dummy;
74            for j in 0 to m-1 loop
75                v_temp(j) := v_temp(j) xor (v1(j) and v2(m-i-1));
76            end loop;
77        end loop;
78        ret := v_temp;
79        return ret;
80    end mult_old;
81
82    function add_old(v1,v2 : in bit_vector) return bit_vector is
83    constant m : integer := 5;
84    constant modulo : bit_vector(0 to 4) := "11111"; -- max val = 30, mod is 31
```

```
85  variable carry : bit := '0';
86  variable dummy : bit := '0';
87  variable v_temp : bit_vector(0 to 5);
88  variable ret : bit_vector(0 to 4);
89
90  begin
91      --v_temp := (0 to 4 => v1, others => '0');
92      v_temp := (others => '0');
93      for i in m-1 downto 0 loop
94          v_temp(i) := (v_temp(i+1) and v2(i)) or (v1(i) and v2(i)) or (v_temp(
                i+1) and v1(i));
95          v_temp(i+1) := v_temp(i+1) xor v1(i) xor v2(i);
96      end loop;
97      if (v_temp(0) = '1') then -- modulo
98          carry := '0';
99          for i in m-1 downto 0 loop
100             dummy := (modulo(i) and carry) or (modulo(i) and not v_temp(i+1))
                    or (carry and not v_temp(i+1));
101             v_temp(i+1) := v_temp(i+1) xor carry xor modulo(i);
102             carry := dummy;
103         end loop;
104     end if;
105     if (v_temp = "011111") then -- modulo
106         v_temp := "000000";
107     end if;
108     ret := v_temp(1 to 5);
109     return ret;
110 end add_old;
111
112 function exp2(input : in bit_vector(0 to 4)) return bit_vector is
113 variable result : bit_vector(0 to 4);
114 begin
115         case input is
116         when "00001" => result := "00001";
117         when "00010" => result := "00100";
118         when "00100" => result := "10000";
119         when "01000" => result := "10010";
120         when "10000" => result := "11010";
121         when "01001" => result := "10011";
122         when "10010" => result := "11110";
123         when "01101" => result := "00011";
124         when "11010" => result := "01100";
125         when "11101" => result := "11001";
126         when "10011" => result := "11111";
127         when "01111" => result := "00111";
128         when "11110" => result := "11100";
129         when "10101" => result := "01011";
130         when "00011" => result := "00101";
131         when "00110" => result := "10100";
132         when "01100" => result := "00010";
133         when "11000" => result := "01000";
134         when "11001" => result := "01001";
135         when "11011" => result := "01101";
136         when "11111" => result := "11101";
137         when "10111" => result := "01111";
```

```
138            when "00111" => result := "10101";
139            when "01110" => result := "00110";
140            when "11100" => result := "11000";
141            when "10001" => result := "11011";
142            when "01011" => result := "10111";
143            when "10110" => result := "01110";
144            when "00101" => result := "10001";
145            when "01010" => result := "10110";
146            when "10100" => result := "01010";
147        when others => result := "00000";
148        end case;
149        return result;
150    end exp2;
151
152    function exp3(input : in bit_vector(0 to 4)) return bit_vector is
153    variable result : bit_vector(0 to 4);
154    begin
155            case input is
156            when "00001" => result := "00001";
157            when "00010" => result := "01000";
158            when "00100" => result := "10010";
159            when "01000" => result := "11101";
160            when "10000" => result := "11110";
161            when "01001" => result := "00110";
162            when "10010" => result := "11001";
163            when "01101" => result := "10111";
164            when "11010" => result := "11100";
165            when "11101" => result := "10110";
166            when "10011" => result := "10100";
167            when "01111" => result := "00100";
168            when "11110" => result := "01001";
169            when "10101" => result := "11010";
170            when "00011" => result := "01111";
171            when "00110" => result := "00011";
172            when "01100" => result := "11000";
173            when "11000" => result := "11111";
174            when "11001" => result := "01110";
175            when "11011" => result := "01011";
176            when "11111" => result := "01010";
177            when "10111" => result := "00010";
178            when "00111" => result := "10000";
179            when "01110" => result := "01101";
180            when "11100" => result := "10011";
181            when "10001" => result := "10101";
182            when "01011" => result := "01100";
183            when "10110" => result := "11011";
184            when "00101" => result := "00111";
185            when "01010" => result := "10001";
186            when "10100" => result := "00101";
187        when others => result := "00000";
188        end case;
189        return result;
190    end exp3;
191
192    function exp4(input : in bit_vector(0 to 4)) return bit_vector is
```

```
193 variable result : bit_vector(0 to 4);
194 begin
195         case input is
196         when "00001" => result := "00001";
197         when "00010" => result := "10000";
198         when "00100" => result := "11010";
199         when "01000" => result := "11110";
200         when "10000" => result := "01100";
201         when "01001" => result := "11111";
202         when "10010" => result := "11100";
203         when "01101" => result := "00101";
204         when "11010" => result := "00010";
205         when "11101" => result := "01001";
206         when "10011" => result := "11101";
207         when "01111" => result := "10101";
208         when "11110" => result := "11000";
209         when "10101" => result := "10111";
210         when "00011" => result := "10001";
211         when "00110" => result := "01010";
212         when "01100" => result := "00100";
213         when "11000" => result := "10010";
214         when "11001" => result := "10011";
215         when "11011" => result := "00011";
216         when "11111" => result := "11001";
217         when "10111" => result := "00111";
218         when "00111" => result := "01011";
219         when "01110" => result := "10100";
220         when "11100" => result := "01000";
221         when "10001" => result := "01101";
222         when "01011" => result := "01111";
223         when "10110" => result := "00110";
224         when "00101" => result := "11011";
225         when "01010" => result := "01110";
226         when "10100" => result := "10110";
227     when others => result := "00000";
228     end case;
229     return result;
230 end exp4;
231
232 function exp5(input : in bit_vector(0 to 4)) return bit_vector is
233 variable result : bit_vector(0 to 4);
234 begin
235         case input is
236         when "00001" => result := "00001";
237         when "00010" => result := "01001";
238         when "00100" => result := "10011";
239         when "01000" => result := "00110";
240         when "10000" => result := "11111";
241         when "01001" => result := "10001";
242         when "10010" => result := "10100";
243         when "01101" => result := "10000";
244         when "11010" => result := "11101";
245         when "11101" => result := "00011";
246         when "10011" => result := "11011";
247         when "01111" => result := "11100";
```

```
248             when "11110" => result := "01010";
249             when "10101" => result := "01000";
250             when "00011" => result := "11010";
251             when "00110" => result := "10101";
252             when "01100" => result := "11001";
253             when "11000" => result := "01110";
254             when "11001" => result := "00101";
255             when "11011" => result := "00100";
256             when "11111" => result := "01101";
257             when "10111" => result := "11110";
258             when "00111" => result := "11000";
259             when "01110" => result := "00111";
260             when "11100" => result := "10110";
261             when "10001" => result := "00010";
262             when "01011" => result := "10010";
263             when "10110" => result := "01111";
264             when "00101" => result := "01100";
265             when "01010" => result := "10111";
266             when "10100" => result := "01011";
267         when others => result := "00000";
268         end case;
269         return result;
270     end exp5;
271
272     function exp6(input : in bit_vector(0 to 4)) return bit_vector is
273     variable result : bit_vector(0 to 4);
274     begin
275             case input is
276             when "00001" => result := "00001";
277             when "00010" => result := "10010";
278             when "00100" => result := "11110";
279             when "01000" => result := "11001";
280             when "10000" => result := "11100";
281             when "01001" => result := "10100";
282             when "10010" => result := "01001";
283             when "01101" => result := "01111";
284             when "11010" => result := "11000";
285             when "11101" => result := "01110";
286             when "10011" => result := "01010";
287             when "01111" => result := "10000";
288             when "11110" => result := "10011";
289             when "10101" => result := "01100";
290             when "00011" => result := "00111";
291             when "00110" => result := "00101";
292             when "01100" => result := "01000";
293             when "11000" => result := "11101";
294             when "11001" => result := "00110";
295             when "11011" => result := "10111";
296             when "11111" => result := "10110";
297             when "10111" => result := "00100";
298             when "00111" => result := "11010";
299             when "01110" => result := "00011";
300             when "11100" => result := "11111";
301             when "10001" => result := "01011";
302             when "01011" => result := "00010";
```

```
303            when "10110" => result := "01101";
304            when "00101" => result := "10101";
305            when "01010" => result := "11011";
306            when "10100" => result := "10001";
307        when others => result := "00000";
308        end case;
309        return result;
310    end exp6;
311
312    function exp7(input : in bit_vector(0 to 4)) return bit_vector is
313    variable result : bit_vector(0 to 4);
314    begin
315            case input is
316            when "00001" => result := "00001";
317            when "00010" => result := "01101";
318            when "00100" => result := "00011";
319            when "01000" => result := "10111";
320            when "10000" => result := "00101";
321            when "01001" => result := "10000";
322            when "10010" => result := "01111";
323            when "01101" => result := "11001";
324            when "11010" => result := "10001";
325            when "11101" => result := "00010";
326            when "10011" => result := "11010";
327            when "01111" => result := "00110";
328            when "11110" => result := "00111";
329            when "10101" => result := "01010";
330            when "00011" => result := "01001";
331            when "00110" => result := "11110";
332            when "01100" => result := "11011";
333            when "11000" => result := "01011";
334            when "11001" => result := "00100";
335            when "11011" => result := "11101";
336            when "11111" => result := "01100";
337            when "10111" => result := "01110";
338            when "00111" => result := "10100";
339            when "01110" => result := "10010";
340            when "11100" => result := "10101";
341            when "10001" => result := "11111";
342            when "01011" => result := "10110";
343            when "10110" => result := "01000";
344            when "00101" => result := "10011";
345            when "01010" => result := "11000";
346            when "10100" => result := "11100";
347        when others => result := "00000";
348        end case;
349        return result;
350    end exp7;
351
352    function exp8(input : in bit_vector(0 to 4)) return bit_vector is
353    variable result : bit_vector(0 to 4);
354    begin
355            case input is
356            when "00001" => result := "00001";
357            when "00010" => result := "11010";
```

```
358          when "00100" => result := "01100";
359          when "01000" => result := "11100";
360          when "10000" => result := "00010";
361          when "01001" => result := "11101";
362          when "10010" => result := "11000";
363          when "01101" => result := "10001";
364          when "11010" => result := "00100";
365          when "11101" => result := "10011";
366          when "10011" => result := "11001";
367          when "01111" => result := "01011";
368          when "11110" => result := "01000";
369          when "10101" => result := "01111";
370          when "00011" => result := "11011";
371          when "00110" => result := "10110";
372          when "01100" => result := "10000";
373          when "11000" => result := "11110";
374          when "11001" => result := "11111";
375          when "11011" => result := "00101";
376          when "11111" => result := "01001";
377          when "10111" => result := "10101";
378          when "00111" => result := "10111";
379          when "01110" => result := "01010";
380          when "11100" => result := "10010";
381          when "10001" => result := "00011";
382          when "01011" => result := "00111";
383          when "10110" => result := "10100";
384          when "00101" => result := "01101";
385          when "01010" => result := "00110";
386          when "10100" => result := "01110";
387      when others => result := "00000";
388      end case;
389      return result;
390  end exp8;
391
392  function exp9(input : in bit_vector(0 to 4)) return bit_vector is
393  variable result : bit_vector(0 to 4);
394  begin
395          case input is
396          when "00001" => result := "00001";
397          when "00010" => result := "11101";
398          when "00100" => result := "11001";
399          when "01000" => result := "10110";
400          when "10000" => result := "01001";
401          when "01001" => result := "00011";
402          when "10010" => result := "01110";
403          when "01101" => result := "00010";
404          when "11010" => result := "10011";
405          when "11101" => result := "11011";
406          when "10011" => result := "00101";
407          when "01111" => result := "10010";
408          when "11110" => result := "00110";
409          when "10101" => result := "11100";
410          when "00011" => result := "00100";
411          when "00110" => result := "01111";
412          when "01100" => result := "11111";
```

```
413         when "11000" => result := "01010";
414         when "11001" => result := "01101";
415         when "11011" => result := "01100";
416         when "11111" => result := "10001";
417         when "10111" => result := "01000";
418         when "00111" => result := "11110";
419         when "01110" => result := "10111";
420         when "11100" => result := "10100";
421         when "10001" => result := "11010";
422         when "01011" => result := "11000";
423         when "10110" => result := "01011";
424         when "00101" => result := "10000";
425         when "01010" => result := "10101";
426         when "10100" => result := "00111";
427     when others => result := "00000";
428     end case;
429     return result;
430 end exp9;
431
432 function exp10(input : in bit_vector(0 to 4)) return bit_vector is
433 variable result : bit_vector(0 to 4);
434 begin
435         case input is
436         when "00001" => result := "00001";
437         when "00010" => result := "10011";
438         when "00100" => result := "11111";
439         when "01000" => result := "10100";
440         when "10000" => result := "11101";
441         when "01001" => result := "11011";
442         when "10010" => result := "01010";
443         when "01101" => result := "11010";
444         when "11010" => result := "11001";
445         when "11101" => result := "00101";
446         when "10011" => result := "01101";
447         when "01111" => result := "11000";
448         when "11110" => result := "10110";
449         when "10101" => result := "10010";
450         when "00011" => result := "01100";
451         when "00110" => result := "01011";
452         when "01100" => result := "01001";
453         when "11000" => result := "00110";
454         when "11001" => result := "10001";
455         when "11011" => result := "10000";
456         when "11111" => result := "00011";
457         when "10111" => result := "11100";
458         when "00111" => result := "01000";
459         when "01110" => result := "10101";
460         when "11100" => result := "01110";
461         when "10001" => result := "00100";
462         when "01011" => result := "11110";
463         when "10110" => result := "00111";
464         when "00101" => result := "00010";
465         when "01010" => result := "01111";
466         when "10100" => result := "10111";
467     when others => result := "00000";
```

```
468      end case;
469      return result;
470  end exp10;
471
472  function exp11(input : in bit_vector(0 to 4)) return bit_vector is
473  variable result : bit_vector(0 to 4);
474  begin
475          case input is
476          when "00001" => result := "00001";
477          when "00010" => result := "01111";
478          when "00100" => result := "00111";
479          when "01000" => result := "00100";
480          when "10000" => result := "10101";
481          when "01001" => result := "11100";
482          when "10010" => result := "10000";
483          when "01101" => result := "00110";
484          when "11010" => result := "01011";
485          when "11101" => result := "10010";
486          when "10011" => result := "11000";
487          when "01111" => result := "00101";
488          when "11110" => result := "11010";
489          when "10101" => result := "11011";
490          when "00011" => result := "10100";
491          when "00110" => result := "10011";
492          when "01100" => result := "10111";
493          when "11000" => result := "00010";
494          when "11001" => result := "11110";
495          when "11011" => result := "01110";
496          when "11111" => result := "01000";
497          when "10111" => result := "00011";
498          when "00111" => result := "10001";
499          when "01110" => result := "01001";
500          when "11100" => result := "01100";
501          when "10001" => result := "10110";
502          when "01011" => result := "01101";
503          when "10110" => result := "11001";
504          when "00101" => result := "01010";
505          when "01010" => result := "11101";
506          when "10100" => result := "11111";
507      when others => result := "00000";
508      end case;
509      return result;
510  end exp11;
511
512  function exp12(input : in bit_vector(0 to 4)) return bit_vector is
513  variable result : bit_vector(0 to 4);
514  begin
515          case input is
516          when "00001" => result := "00001";
517          when "00010" => result := "11110";
518          when "00100" => result := "11100";
519          when "01000" => result := "01001";
520          when "10000" => result := "11000";
521          when "01001" => result := "01010";
522          when "10010" => result := "10011";
```

```
523         when "01101" => result := "00111";
524         when "11010" => result := "01000";
525         when "11101" => result := "00110";
526         when "10011" => result := "10110";
527         when "01111" => result := "11010";
528         when "11110" => result := "11111";
529         when "10101" => result := "00010";
530         when "00011" => result := "10101";
531         when "00110" => result := "10001";
532         when "01100" => result := "10010";
533         when "11000" => result := "11001";
534         when "11001" => result := "10100";
535         when "11011" => result := "01111";
536         when "11111" => result := "01110";
537         when "10111" => result := "10000";
538         when "00111" => result := "01100";
539         when "01110" => result := "00101";
540         when "11100" => result := "11101";
541         when "10001" => result := "10111";
542         when "01011" => result := "00100";
543         when "10110" => result := "00011";
544         when "00101" => result := "01011";
545         when "01010" => result := "01101";
546         when "10100" => result := "11011";
547     when others => result := "00000";
548     end case;
549     return result;
550 end exp12;
551
552 function exp13(input : in bit_vector(0 to 4)) return bit_vector is
553 variable result : bit_vector(0 to 4);
554 begin
555         case input is
556         when "00001" => result := "00001";
557         when "00010" => result := "10101";
558         when "00100" => result := "01011";
559         when "01000" => result := "11010";
560         when "10000" => result := "10111";
561         when "01001" => result := "01000";
562         when "10010" => result := "01100";
563         when "01101" => result := "01010";
564         when "11010" => result := "01111";
565         when "11101" => result := "11100";
566         when "10011" => result := "10010";
567         when "01111" => result := "11011";
568         when "11110" => result := "00010";
569         when "10101" => result := "00011";
570         when "00011" => result := "10110";
571         when "00110" => result := "11101";
572         when "01100" => result := "00111";
573         when "11000" => result := "10000";
574         when "11001" => result := "11000";
575         when "11011" => result := "10100";
576         when "11111" => result := "11110";
577         when "10111" => result := "10001";
```

```
578            when "00111" => result := "01101";
579            when "01110" => result := "11111";
580            when "11100" => result := "00100";
581            when "10001" => result := "00110";
582            when "01011" => result := "00101";
583            when "10110" => result := "10011";
584            when "00101" => result := "01110";
585            when "01010" => result := "01001";
586            when "10100" => result := "11001";
587        when others => result := "00000";
588        end case;
589        return result;
590    end exp13;
591
592    function exp14(input : in bit_vector(0 to 4)) return bit_vector is
593    variable result : bit_vector(0 to 4);
594    begin
595            case input is
596            when "00001" => result := "00001";
597            when "00010" => result := "00011";
598            when "00100" => result := "00101";
599            when "01000" => result := "01111";
600            when "10000" => result := "10001";
601            when "01001" => result := "11010";
602            when "10010" => result := "00111";
603            when "01101" => result := "01001";
604            when "11010" => result := "11011";
605            when "11101" => result := "00100";
606            when "10011" => result := "01100";
607            when "01111" => result := "10100";
608            when "11110" => result := "10101";
609            when "10101" => result := "10110";
610            when "00011" => result := "10011";
611            when "00110" => result := "11100";
612            when "01100" => result := "01101";
613            when "11000" => result := "10111";
614            when "11001" => result := "10000";
615            when "11011" => result := "11001";
616            when "11111" => result := "00010";
617            when "10111" => result := "00110";
618            when "00111" => result := "01010";
619            when "01110" => result := "11110";
620            when "11100" => result := "01011";
621            when "10001" => result := "11101";
622            when "01011" => result := "01110";
623            when "10110" => result := "10010";
624            when "00101" => result := "11111";
625            when "01010" => result := "01000";
626            when "10100" => result := "11000";
627        when others => result := "00000";
628        end case;
629        return result;
630    end exp14;
631
632    function exp15(input : in bit_vector(0 to 4)) return bit_vector is
```

```
633  variable result : bit_vector(0 to 4);
634  begin
635          case input is
636          when "00001" => result := "00001";
637          when "00010" => result := "00110";
638          when "00100" => result := "10100";
639          when "01000" => result := "00011";
640          when "10000" => result := "01010";
641          when "01001" => result := "10101";
642          when "10010" => result := "00101";
643          when "01101" => result := "11110";
644          when "11010" => result := "10110";
645          when "11101" => result := "01111";
646          when "10011" => result := "01011";
647          when "01111" => result := "10011";
648          when "11110" => result := "10001";
649          when "10101" => result := "11101";
650          when "00011" => result := "11100";
651          when "00110" => result := "11010";
652          when "01100" => result := "01110";
653          when "11000" => result := "01101";
654          when "11001" => result := "00111";
655          when "11011" => result := "10010";
656          when "11111" => result := "10111";
657          when "10111" => result := "01001";
658          when "00111" => result := "11111";
659          when "01110" => result := "10000";
660          when "11100" => result := "11011";
661          when "10001" => result := "01000";
662          when "01011" => result := "11001";
663          when "10110" => result := "00100";
664          when "00101" => result := "11000";
665          when "01010" => result := "00010";
666          when "10100" => result := "01100";
667      when others => result := "00000";
668      end case;
669      return result;
670  end exp15;
671
672      signal d4_1, d4_2, d4_3, d4_4, d4_5 : bit_vector(0 to 4) := "00000";
673
674  begin
675
676  --   det3 <= add(exp3(s1),s3);
677  --   det4 <= add(mult(exp3(s1),s3),add(mult(s1,s5),add(exp6(s1),exp2(s3))));
678  --   det5 <= add(mult(s1,exp3(s3)),add(exp2(s5),add(mult(exp7(s1),s3),add(mult
         (exp2(s1),mult(s3,s5)),add(exp10(s1),add(mult(exp3(s1),s7),add(mult(exp5(
         s1),s5),mult(s3,s7))))))));
679  -- det6 <= add(mult(s1,mult(exp3(s3),s5)),add(mult(s1,mult(s5,s9)),add(mult(
         exp5(s1),mult(s3,s7)),add(mult(exp3(s1),mult(s5,s7)),add(mult(exp6(s1),s9
         ),add(mult(exp2(s1),mult(exp2(s3),s7)),add(mult(exp5(s1),exp2(s5)),add(
         mult(s1,exp2(s7)),add(mult(exp6(s1),exp3(s3)),add(mult(exp4(s1),mult(exp2
         (s3),s5)),add(mult(exp12(s1),s3),add(mult(exp2(s3),s9),add(mult(exp8(s1),
         s7),add(mult(exp9(s1),exp2(s3)),add(mult(exp3(s1),mult(s3,s9)),add(exp5(
         s3),add(exp15(s1),add(mult(exp7(s1),mult(s3,s5)),add(exp3(s5),mult(exp2(
```

```
        s1),mult(s3,exp2(s5))))))))))))))))))))))
680
681
682     is_onebit <= ("00000" /= s1) AND ("00000" /= add(exp3(s1),s3)) AND ( "00000
            " = add(mult(exp3(s1),s3),add(mult(s1,s5),add(exp6(s1),exp2(s3))))) AND
             ( "00000" = add(mult(s1,exp3(s3)),add(exp2(s5),add(mult(exp7(s1),s3),
            add(mult(exp2(s1),mult(s3,s5)),add(exp10(s1),add(mult(exp3(s1),s7),add(
            mult(exp5(s1),s5),mult(s3,s7))))))))) AND ( "00000" = add(mult(s1,mult(
            exp3(s3),s5)),add(mult(s1,mult(s5,s9)),add(mult(exp5(s1),mult(s3,s7)),
            add(mult(exp3(s1),mult(s5,s7)),add(mult(exp6(s1),s9),add(mult(exp2(s1),
            mult(exp2(s3),s7)),add(mult(exp5(s1),exp2(s5)),add(mult(s1,exp2(s7)),
            add(mult(exp6(s1),exp3(s3)),add(mult(exp4(s1),mult(exp2(s3),s5)),add(
            mult(exp12(s1),s3),add(mult(exp2(s3),s9),add(mult(exp8(s1),s7),add(mult
            (exp9(s1),exp2(s3)),add(mult(exp3(s1),mult(s3,s9)),add(exp5(s3),add(
            exp15(s1),add(mult(exp7(s1),mult(s3,s5)),add(exp3(s5),mult(exp2(s1),
            mult(s3,exp2(s5)))))))))))))))))))))) );
683
684   end behavioral;
```

Listing 5.6: VHDL Vergleichsimplementierung Determinante

```
 1  --  log(i) in GF(2^5)
 2
 3  library IEEE;
 4  use IEEE.STD_LOGIC_1164.ALL;
 5
 6  --  Defines a design entity
 7  entity exponents is
 8   Port (
 9      signal i,j : in bit_vector(0 to 4);
10          signal s1 : in bit_vector(0 to 4);
11          signal s3 : in bit_vector(0 to 4);
12          signal s5 : in bit_vector(0 to 4);
13          signal s7 : in bit_vector(0 to 4);
14          signal s9 : in bit_vector(0 to 4);
15      signal is_onebit : OUT boolean
16   );
17  end exponents;
18
19  architecture behavioral of exponents is
20
21  function add(v1,v2 : in bit_vector) return bit_vector is
22  constant m : integer := 4;
23  variable ret : bit_vector(0 to 4);
24
25  begin
26      for i in m downto 0 loop
27          ret(i) := v1(i) xor v2(i);
28      end loop;
29      return ret;
30  end add;
31
32  function exp3(input : in bit_vector(0 to 4)) return bit_vector is
33  variable result : bit_vector(0 to 4);
34  begin
```

```
35          case input is
36          when "00001" => result := "00001";
37          when "00010" => result := "01000";
38          when "00100" => result := "10010";
39          when "01000" => result := "11101";
40          when "10000" => result := "11110";
41          when "01001" => result := "00110";
42          when "10010" => result := "11001";
43          when "01101" => result := "10111";
44          when "11010" => result := "11100";
45          when "11101" => result := "10110";
46          when "10011" => result := "10100";
47          when "01111" => result := "00100";
48          when "11110" => result := "01001";
49          when "10101" => result := "11010";
50          when "00011" => result := "01111";
51          when "00110" => result := "00011";
52          when "01100" => result := "11000";
53          when "11000" => result := "11111";
54          when "11001" => result := "01110";
55          when "11011" => result := "01011";
56          when "11111" => result := "01010";
57          when "10111" => result := "00010";
58          when "00111" => result := "10000";
59          when "01110" => result := "01101";
60          when "11100" => result := "10011";
61          when "10001" => result := "10101";
62          when "01011" => result := "01100";
63          when "10110" => result := "11011";
64          when "00101" => result := "00111";
65          when "01010" => result := "10001";
66          when "10100" => result := "00101";
67      when others => result := "00000";
68      end case;
69      return result;
70  end exp3;
71
72  function exp5(input : in bit_vector(0 to 4)) return bit_vector is
73  variable result : bit_vector(0 to 4);
74  begin
75          case input is
76          when "00001" => result := "00001";
77          when "00010" => result := "01001";
78          when "00100" => result := "10011";
79          when "01000" => result := "00110";
80          when "10000" => result := "11111";
81          when "01001" => result := "10001";
82          when "10010" => result := "10100";
83          when "01101" => result := "10000";
84          when "11010" => result := "11101";
85          when "11101" => result := "00011";
86          when "10011" => result := "11011";
87          when "01111" => result := "11100";
88          when "11110" => result := "01010";
89          when "10101" => result := "01000";
```

```
90          when "00011" => result := "11010";
91          when "00110" => result := "10101";
92          when "01100" => result := "11001";
93          when "11000" => result := "01110";
94          when "11001" => result := "00101";
95          when "11011" => result := "00100";
96          when "11111" => result := "01101";
97          when "10111" => result := "11110";
98          when "00111" => result := "11000";
99          when "01110" => result := "00111";
100         when "11100" => result := "10110";
101         when "10001" => result := "00010";
102         when "01011" => result := "10010";
103         when "10110" => result := "01111";
104         when "00101" => result := "01100";
105         when "01010" => result := "10111";
106         when "10100" => result := "01011";
107     when others => result := "00000";
108     end case;
109     return result;
110 end exp5;
111
112 function exp7(input : in bit_vector(0 to 4)) return bit_vector is
113 variable result : bit_vector(0 to 4);
114 begin
115         case input is
116         when "00001" => result := "00001";
117         when "00010" => result := "01101";
118         when "00100" => result := "00011";
119         when "01000" => result := "10111";
120         when "10000" => result := "00101";
121         when "01001" => result := "10000";
122         when "10010" => result := "01111";
123         when "01101" => result := "11001";
124         when "11010" => result := "10001";
125         when "11101" => result := "00010";
126         when "10011" => result := "11010";
127         when "01111" => result := "00110";
128         when "11110" => result := "00111";
129         when "10101" => result := "01010";
130         when "00011" => result := "01001";
131         when "00110" => result := "11110";
132         when "01100" => result := "11011";
133         when "11000" => result := "01011";
134         when "11001" => result := "00100";
135         when "11011" => result := "11101";
136         when "11111" => result := "01100";
137         when "10111" => result := "01110";
138         when "00111" => result := "10100";
139         when "01110" => result := "10010";
140         when "11100" => result := "10101";
141         when "10001" => result := "11111";
142         when "01011" => result := "10110";
143         when "10110" => result := "01000";
144         when "00101" => result := "10011";
```

```
145            when "01010" => result := "11000";
146            when "10100" => result := "11100";
147        when others => result := "00000";
148        end case;
149        return result;
150    end exp7;
151
152    function exp9(input : in bit_vector(0 to 4)) return bit_vector is
153    variable result : bit_vector(0 to 4);
154    begin
155            case input is
156            when "00001" => result := "00001";
157            when "00010" => result := "11101";
158            when "00100" => result := "11001";
159            when "01000" => result := "10110";
160            when "10000" => result := "01001";
161            when "01001" => result := "00011";
162            when "10010" => result := "01110";
163            when "01101" => result := "00010";
164            when "11010" => result := "10011";
165            when "11101" => result := "11011";
166            when "10011" => result := "00101";
167            when "01111" => result := "10010";
168            when "11110" => result := "00110";
169            when "10101" => result := "11100";
170            when "00011" => result := "00100";
171            when "00110" => result := "01111";
172            when "01100" => result := "11111";
173            when "11000" => result := "01010";
174            when "11001" => result := "01101";
175            when "11011" => result := "01100";
176            when "11111" => result := "10001";
177            when "10111" => result := "01000";
178            when "00111" => result := "11110";
179            when "01110" => result := "10111";
180            when "11100" => result := "10100";
181            when "10001" => result := "11010";
182            when "01011" => result := "11000";
183            when "10110" => result := "01011";
184            when "00101" => result := "10000";
185            when "01010" => result := "10101";
186            when "10100" => result := "00111";
187        when others => result := "00000";
188        end case;
189        return result;
190    end exp9;
191
192    begin
193
194      is_onebit <= (s1 = add(i,j)) AND (s3 = add(exp3(i),exp3(j)) ) AND (s5 = add
           (exp5(i),exp5(j))) AND (s7 = add(exp7(i),exp7(j))) AND (s9 = add(exp9(i
           ),exp9(j)));
195
196    end behavioral;
```

Listing 5.7: VHDL Vergleichsimplementierung Syndrom

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity syndromkomponenten is
5      PORT(
6      input : IN bit_vector(0 to 30);
7      s1 : OUT bit_vector(0 to 4);
8      s3 : OUT bit_vector(0 to 4);
9      s5 : OUT bit_vector(0 to 4);
10     s7 : OUT bit_vector(0 to 4);
11     s9 : OUT bit_vector(0 to 4)
12     );
13 end syndromkomponenten;
14
15 architecture behavioral of syndromkomponenten is
16 begin
17 s1(0) <= input(4) XOR input(6) XOR input(8) XOR input(9) XOR input(10) XOR
       input(12) XOR input(13) XOR input(17) XOR input(18) XOR input(19) XOR
       input(20) XOR input(21) XOR input(24) XOR input(25) XOR input(27) XOR
       input(30);
18 s1(1) <= input(3) XOR input(5) XOR input(7) XOR input(8) XOR input(9) XOR
       input(11) XOR input(12) XOR input(16) XOR input(17) XOR input(18) XOR
       input(19) XOR input(20) XOR input(23) XOR input(24) XOR input(26) XOR
       input(29);
19 s1(2) <= input(2) XOR input(7) XOR input(9) XOR input(11) XOR input(12) XOR
       input(13) XOR input(15) XOR input(16) XOR input(20) XOR input(21) XOR
       input(22) XOR input(23) XOR input(24) XOR input(27) XOR input(28) XOR
       input(30);
20 s1(3) <= input(1) XOR input(6) XOR input(8) XOR input(10) XOR input(11) XOR
       input(12) XOR input(14) XOR input(15) XOR input(19) XOR input(20) XOR
       input(21) XOR input(22) XOR input(23) XOR input(26) XOR input(27) XOR
       input(29);
21 s1(4) <= input(0) XOR input(5) XOR input(7) XOR input(9) XOR input(10) XOR
       input(11) XOR input(13) XOR input(14) XOR input(18) XOR input(19) XOR
       input(20) XOR input(21) XOR input(22) XOR input(25) XOR input(26) XOR
       input(28);
22
23 s3(0) <= input(2) XOR input(3) XOR input(4) XOR input(6) XOR input(7) XOR
       input(8) XOR input(9) XOR input(10) XOR input(13) XOR input(16) XOR input
       (17) XOR input(22) XOR input(24) XOR input(25) XOR input(27) XOR input
       (29);
24 s3(1) <= input(1) XOR input(3) XOR input(4) XOR input(6) XOR input(8) XOR
       input(12) XOR input(13) XOR input(14) XOR input(16) XOR input(17) XOR
       input(18) XOR input(19) XOR input(20) XOR input(23) XOR input(26) XOR
       input(27);
25 s3(2) <= input(3) XOR input(4) XOR input(5) XOR input(7) XOR input(8) XOR
       input(9) XOR input(10) XOR input(11) XOR input(14) XOR input(17) XOR
       input(18) XOR input(23) XOR input(25) XOR input(26) XOR input(28) XOR
       input(30);
26 s3(3) <= input(2) XOR input(4) XOR input(5) XOR input(7) XOR input(9) XOR
       input(13) XOR input(14) XOR input(15) XOR input(17) XOR input(18) XOR
```

```
      input(19) XOR input(20) XOR input(21) XOR input(24) XOR input(27) XOR
      input(28);
27  s3(4) <= input(0) XOR input(3) XOR input(6) XOR input(7) XOR input(12) XOR
      input(14) XOR input(15) XOR input(17) XOR input(19) XOR input(23) XOR
      input(24) XOR input(25) XOR input(27) XOR input(28) XOR input(29) XOR
      input(30);
28
29  s5(0) <= input(2) XOR input(4) XOR input(5) XOR input(6) XOR input(7) XOR
      input(8) XOR input(10) XOR input(11) XOR input(14) XOR input(15) XOR
      input(16) XOR input(21) XOR input(22) XOR input(24) XOR input(26) XOR
      input(29);
30  s5(1) <= input(1) XOR input(4) XOR input(8) XOR input(10) XOR input(11) XOR
      input(12) XOR input(13) XOR input(14) XOR input(16) XOR input(17) XOR
      input(20) XOR input(21) XOR input(22) XOR input(27) XOR input(28) XOR
      input(30);
31  s5(2) <= input(3) XOR input(4) XOR input(6) XOR input(8) XOR input(11) XOR
      input(15) XOR input(17) XOR input(18) XOR input(19) XOR input(20) XOR
      input(21) XOR input(23) XOR input(24) XOR input(27) XOR input(28) XOR
      input(29);
32  s5(3) <= input(2) XOR input(3) XOR input(4) XOR input(9) XOR input(10) XOR
      input(12) XOR input(14) XOR input(17) XOR input(21) XOR input(23) XOR
      input(24) XOR input(25) XOR input(26) XOR input(27) XOR input(29) XOR
      input(30);
33  s5(4) <= input(0) XOR input(1) XOR input(2) XOR input(4) XOR input(5) XOR
      input(8) XOR input(9) XOR input(10) XOR input(15) XOR input(16) XOR input
      (18) XOR input(20) XOR input(23) XOR input(27) XOR input(29) XOR input
      (30);
34
35  s7(0) <= input(3) XOR input(5) XOR input(7) XOR input(8) XOR input(10) XOR
      input(15) XOR input(16) XOR input(19) XOR input(22) XOR input(23) XOR
      input(24) XOR input(25) XOR input(26) XOR input(28) XOR input(29) XOR
      input(30);
36  s7(1) <= input(1) XOR input(6) XOR input(7) XOR input(10) XOR input(13) XOR
      input(14) XOR input(15) XOR input(16) XOR input(17) XOR input(19) XOR
      input(20) XOR input(21) XOR input(25) XOR input(27) XOR input(29) XOR
      input(30);
37  s7(2) <= input(1) XOR input(3) XOR input(4) XOR input(6) XOR input(11) XOR
      input(12) XOR input(15) XOR input(18) XOR input(19) XOR input(20) XOR
      input(21) XOR input(22) XOR input(24) XOR input(25) XOR input(26) XOR
      input(30);
38  s7(3) <= input(2) XOR input(3) XOR input(6) XOR input(9) XOR input(10) XOR
      input(11) XOR input(12) XOR input(13) XOR input(15) XOR input(16) XOR
      input(17) XOR input(21) XOR input(23) XOR input(25) XOR input(26) XOR
      input(28);
39  s7(4) <= input(0) XOR input(1) XOR input(2) XOR input(3) XOR input(4) XOR
      input(6) XOR input(7) XOR input(8) XOR input(12) XOR input(14) XOR input
      (16) XOR input(17) XOR input(19) XOR input(24) XOR input(25) XOR input
      (28);
40
41  s9(0) <= input(1) XOR input(2) XOR input(3) XOR input(8) XOR input(9) XOR
      input(11) XOR input(13) XOR input(16) XOR input(20) XOR input(22) XOR
      input(23) XOR input(24) XOR input(25) XOR input(26) XOR input(28) XOR
      input(29);
42  s9(1) <= input(1) XOR input(2) XOR input(4) XOR input(6) XOR input(9) XOR
      input(13) XOR input(15) XOR input(16) XOR input(17) XOR input(18) XOR
```

```
        input(19) XOR input(21) XOR input(22) XOR input(25) XOR input(26) XOR
        input(27);
43  s9(2) <= input(1) XOR input(3) XOR input(6) XOR input(10) XOR input(12) XOR
        input(13) XOR input(14) XOR input(15) XOR input(16) XOR input(18) XOR
        input(19) XOR input(22) XOR input(23) XOR input(24) XOR input(29) XOR
        input(30);
44  s9(3) <= input(3) XOR input(5) XOR input(6) XOR input(7) XOR input(8) XOR
        input(9) XOR input(11) XOR input(12) XOR input(15) XOR input(16) XOR
        input(17) XOR input(22) XOR input(23) XOR input(25) XOR input(27) XOR
        input(30);
45  s9(4) <= input(0) XOR input(1) XOR input(2) XOR input(4) XOR input(5) XOR
        input(8) XOR input(9) XOR input(10) XOR input(15) XOR input(16) XOR input
        (18) XOR input(20) XOR input(23) XOR input(27) XOR input(29) XOR input
        (30);
46  end behavioral;
47
48  entity testbench is
49  end testbench;
50
51  architecture tb of testbench is
52      component syndromkomponenten is
53      PORT(
54          input : IN bit_vector(0 to 30);
55          s1 : OUT bit_vector(0 to 4);
56          s3 : OUT bit_vector(0 to 4);
57          s5 : OUT bit_vector(0 to 4);
58          s7 : OUT bit_vector(0 to 4);
59          s9 : OUT bit_vector(0 to 4)
60      );
61      end component;
62
63      component exponents is
64       Port (
65              signal i,j : in bit_vector(0 to 4);
66              signal s1 : in bit_vector(0 to 4);
67              signal s3 : in bit_vector(0 to 4);
68              signal s5 : in bit_vector(0 to 4);
69              signal s7 : in bit_vector(0 to 4);
70          signal s9 : in bit_vector(0 to 4);
71          signal is_onebit : OUT boolean
72       );
73      end component;
74
75      component determinants is
76       Port (
77              signal s1 : in bit_vector(0 to 4);
78              signal s3 : in bit_vector(0 to 4);
79              signal s5 : in bit_vector(0 to 4);
80              signal s7 : in bit_vector(0 to 4);
81          signal s9 : in bit_vector(0 to 4);
82      --  signal det3 : out bit_vector(0 to 4);
83      --  signal det4 : out bit_vector(0 to 4);
84      --  signal det5 : out bit_vector(0 to 4)
85          signal is_onebit : OUT boolean
86       );
```

```
87      end component;
88
89      -- from exponent to base e -> v, where v = alpha^e
90      function to_base(input : integer) return bit_vector is
91      variable result : bit_vector(0 to 4);
92      begin
93          case input is
94          when 0 => result := "00001";
95          when 1 => result := "00010";
96          when 2 => result := "00100";
97          when 3 => result := "01000";
98          when 4 => result := "10000";
99          when 5 => result := "01001";
100         when 6 => result := "10010";
101         when 7 => result := "01101";
102         when 8 => result := "11010";
103         when 9 => result := "11101";
104         when 10 => result := "10011";
105         when 11 => result := "01111";
106         when 12 => result := "11110";
107         when 13 => result := "10101";
108         when 14 => result := "00011";
109         when 15 => result := "00110";
110         when 16 => result := "01100";
111         when 17 => result := "11000";
112         when 18 => result := "11001";
113         when 19 => result := "11011";
114         when 20 => result := "11111";
115         when 21 => result := "10111";
116         when 22 => result := "00111";
117         when 23 => result := "01110";
118         when 24 => result := "11100";
119         when 25 => result := "10001";
120         when 26 => result := "01011";
121         when 27 => result := "10110";
122         when 28 => result := "00101";
123         when 29 => result := "01010";
124         when 30 => result := "10100";
125         when others => result := "11111";
126         end case;
127         return result;
128     end to_base;
129
130     function compare(a,b: bit_vector(0 to 4)) return bit is
131     begin
132         if a = b then
133             return('1');
134         else
135             return('0');
136         end if;
137     end function compare;
138
139     -- Inputs
140     signal input_tb : bit_vector(0 to 30) := "0000000000000000000000000000000
            ";
```

```
141
142     -- inbetween
143     signal in_i, in_j : bit_vector(0 to 4) := "00000";
144     signal s1_tb : bit_vector(0 to 4) := "00000";
145     signal s3_tb : bit_vector(0 to 4) := "00000";
146     signal s5_tb : bit_vector(0 to 4) := "00000";
147     signal s7_tb : bit_vector(0 to 4) := "00000";
148     signal s9_tb : bit_vector(0 to 4) := "00000";
149
150     -- Outputs
151     signal out_det : BOOLEAN;
152     signal out_exp : BOOLEAN;
153     constant empty_small : bit_vector(0 to 4) := "00000";
154     constant empty_small_inv : bit_vector(0 to 4) := "11111";
155     constant help_word : bit_vector(0 to 30) := "
            0000000000000000000000000000000";
156     constant help_invword : bit_vector(0 to 30) := "
            1111111111111111111111111111111";
157
158 begin
159     -- connecting testbench signals with half_adder.vhd
160     u1 : entity work.syndromkomponenten port map (input => input_tb, s1 =>
            s1_tb, s3 => s3_tb, s5 => s5_tb, s7 => s7_tb, s9 => s9_tb);
161     u2 : entity work.exponents port map (i => in_i, j => in_j, s1 => s1_tb,
            s3 => s3_tb, s5 => s5_tb, s7 => s7_tb, s9 => s9_tb, is_onebit =>
            out_exp);
162     u3 : entity work.determinants port map (s1 => s1_tb, s3 => s3_tb, s5 =>
            s5_tb, s7 => s7_tb, s9 => s9_tb, is_onebit => out_det);
163
164     -- Stimulus process
165     stim_proc: process
166     begin
167     -- hold reset state for 100 ns.
168     input_tb <= help_word;
169     in_i <= empty_small_inv;
170     in_j <= empty_small;
171     wait for 100 ns;
172
173     -- 1 Bit Errors
174     for i in 30 downto 0 loop
175         in_i <= s1_tb;
176         input_tb <= help_word;
177         input_tb(i) <= help_invword(i);
178         wait for 10 ns;
179     end loop;
180
181     wait for 20 ns;
182
183     -- 2 Bit Errors
184     for i in 30 downto 1 loop
185         in_i <= to_base(i);
186         input_tb <= help_word;
187         input_tb(i) <= help_invword(i);
188         for j in i-1 downto 0 loop
189             in_j <= to_base(j);
```

```
190                 input_tb(j) <= help_invword(j);
191                 wait for 10 ns;
192                 input_tb(j) <= help_word(j);
193             end loop;
194         end loop;
195
196         wait for 20 ns;
197
198         -- 3 Bit Errors
199         for i in 30 downto 0 loop
200             in_i <= to_base(i);
201             input_tb <= help_word;
202             input_tb(i) <= help_invword(i);
203             for j in i-1 downto 0 loop
204                 in_j <= to_base(j);
205                 input_tb(j) <= help_invword(j);
206                 for k in j-1 downto 0 loop
207                     input_tb(k) <= help_invword(k);
208                     wait for 10 ns;
209                     input_tb(k) <= help_word(k);
210                 end loop;
211                 input_tb(j) <= help_word(j);
212             end loop;
213         end loop;
214
215         wait;
216         end process;
217 end tb ;
```

Listing 5.8: VHDL Vergleichsimplementierung Testbench

```
1  #!/usr/bin/env python
2  import json, sys
3
4  alpha_table = []
5
6  def mult_full(a,b,p):
7      la=len(a)
8      lb=len(b)
9      res = [0]*(la+lb-1)
10     for i in range(lb):
11         if b[i] == 1:
12             for j in range(la):
13                 if a[j] == 1:
14                     res[i+j] = 1 - res[i+j]
15     for i in range(la-1):
16         if res[i]==1:
17             for j in range(len(p)):
18                 if p[j]=='1': res[j+i] = 1 - res[j+i]
19     res = res[la-1:]
20     return res
21
22 def build_alpha_table(p):
23     global alpha_table
24     alpha_table = []
```

```
25        a = [0]*(len(p)-1)
26        b = [0]*(len(p)-1)
27        a[-1] = 1
28        b[-2] = 1
29        alpha_table.append(a)
30        count = 2**(len(p)-1)
31        for i in range(1,count-1):
32            a = mult_full(a,b,p)
33            alpha_table.append(a)
34
35    def get_exponent_table(v):
36        global alpha_table
37        if v in alpha_table:
38            return alpha_table.index(v)
39        return None
40
41    def get_value_table(e):
42        global alpha_table
43        if e is None: return [0 for x in alpha_table[0]]
44        c = e % len(alpha_table)
45        return alpha_table[c]
46
47    def add(*items):
48        if len(items) == 0: return 0
49        a = get_value_table(items[0]) if items[0] is None or type(items[0]) is
              int else items[0]
50        for item in items[1:]:
51            b = get_value_table(item) if item is None or type(item) is int else
                  item
52            a = [ 0 if a[x]==b[x] else 1 for x in range(len(a))]
53        return a
54
55    def mult(*items):
56        if len(items) == 0: return 0
57        a = items[0] if items[0] is None or type(items[0]) is int else
              get_exponent_table(items[0])
58        if a is None: return get_value_table(None)
59        for item in items[1:]:
60            b = item if item is None or type(item) is int else get_exponent_table
                  (item)
61            if b is None: return None
62            a += b
63        return get_value_table(a)
64
65    def div(a,b):
66        if not 1 in a: return a
67        if not 1 in b:
68            return None
69        ea = get_exponent_table(a)
70        eb = get_exponent_table(b)
71        try: ne = ea - eb
72        except:
73            print "Error! Division of "+str(a)+"/"+str(b)+" failed because of non
                  -primitive polynom!"
74            sys.exit(2)
```

```
75      return get_value_table(ne)
76
77  def exp(a,b):
78      if not 1 in a: return a # 0 ^ b
79      ea = get_exponent_table(a)
80      if ea is None:
81          print "Error! Polynom might be non-primitive!"
82          sys.exit(2)
83      return get_value_table(ea*b)
84
85  def isZero(a):
86      return not 1 in a
87
88  def load_json(filename):
89      fp = open(filename)
90      text = fp.read()
91      fp.close()
92      data = json.loads(text)
93      return data
94
95  def alpha_matrix_from_h(h,cb,sl):
96      le = len(h[0])
97      res = [ [ [ h[i*sl+j][k] for j in range(sl) ] for k in range(le) ] for i
            in range(cb) ]
98      return res
99
100 def error_iterator(values,maxval,count):
101     if values is None:
102         return range(count), True
103     else:
104         mymax = maxval
105         ind = len(values)-1
106         todos = [ind]
107         while ind >= 0 and values[ind] == mymax-1:
108             mymax -= 1
109             ind -= 1
110             todos.append(ind)
111         if ind < 0:
112             return None, False
113         ind = todos.pop()
114         next = values[ind]+1
115         values[ind] = next
116         next += 1
117         while len(todos) > 0:
118             ind = todos.pop()
119             values[ind] = next
120             next += 1
121     return values, True
122
123 def oneBitCorrection(synd,reverse_table):
124     return reverse_table[str(synd[0])][0] if str(synd[0]) in reverse_table
            else None
125
126 def oneBitCorrection_full(synd,reverse_table):
127     s = str(synd[0])
```

```
128        if s in reverse_table:
129            x,sy = reverse_table[s]
130            if str(sy) == str(synd[1:]):
131                return reverse_table[str(synd[0])]
132        return None
133
134    def oneBitCheck(synd):
135        if len(synd) < 1: return False
136        s_a = synd[0]
137        for i in range(1,len(synd)):
138            index = i*2+1
139            if str(exp(synd[0],index)) != str(synd[i]):
140                return False
141        return True
142
143    def twoBitCorrection(synd,reverse_table):
144        check_index = 1
145        check_value = 1 + check_index*2
146        tab = reverse_table[check_index-1]
147        sexp = exp(synd[0],check_value)
148        sdiv = div(synd[check_index],sexp)
149        if sdiv is None:
150            return None, None
151        index=str(sdiv[:-1])
152        if index not in tab:
153            return None, None
154        item=tab[index]
155        first_error=mult(get_value_table(item),synd[0])
156        second_error = add(synd[0],first_error)
157        return get_exponent_table(first_error), get_exponent_table(second_error)
158
159    def twoBitCheck_new(synd,e_1,e_2):
160        if len(synd) < 2 or e_1 is None or e_2 is None: return False
161        ve_1 = get_value_table(e_1)
162        ve_2 = get_value_table(e_2)
163        for i in range(len(synd)):
164            index = i*2+1
165            res = add(exp(ve_1,index),exp(ve_2,index))
166            if str(res) != str(synd[i]):
167                return False
168        return True
169
170    def threeBitCorrection(synd,reverse_table):
171        check_index = 1
172        check_value = 1 + check_index*2
173        tab = reverse_table[check_index-1]
174        sigma_2 = div(add(mult(exp(synd[0],2),synd[1]),synd[2]),add(exp(synd
               [0],3),exp(synd[0],2)))# (s_1^2*s_3 + s_5) / (s_1^3 + s_2)
175        sigma_3 = add(add(exp(synd[0],3),synd[1]),mult(synd[0],sigma_2)) # s_1^3
               + s_3 + s_1 * sigma_2
176        n = add(exp(synd[0],2),sigma_2)
177        delta = add(mult(synd[0],sigma_2),sigma_3)
178        c = div(delta,exp(n, 3/2))
179        if sdiv is None:
180            return None, None, None
```

```
181       index=str(sdiv[:-1])
182       if index not in tab:
183           return None, None, None
184       item=tab[index]
185       first_error=mult(get_value_table(item),synd[0])
186       second_error = add(synd[0],first_error)
187       return get_exponent_table(first_error), get_exponent_table(second_error)
188
189   def test_determinant_4(synd):
190       s1 = synd[0]
191       s3 = synd[1]
192       s5 = synd[2]
193       s7 = synd[3]
194       det5 = add(mult(s1,exp(s3,3)),exp(s5,2),mult(exp(s1,7),s3),mult(exp(s1,2)
              ,s3,s5),exp(s1,10),mult(exp(s1,3),s7),mult(exp(s1,5),s5),mult(s3,s7))
195       if not isZero(det5):
196           return 4
197       det4 = add(mult(exp(s1,3),s3),mult(s1,s5),exp(s1,6),exp(s3,2))
198       if not isZero(det4):
199           return 3
200       det3 = add(exp(s1,3),s3)
201       if not isZero(det3):
202           return 2
203       det2 = s1
204       if not isZero(det2):
205           return 1
206       return 0
207
208   def test_determinant_3(synd):
209       s1 = synd[0]
210       s3 = synd[1]
211       s5 = synd[2]
212       det4 = add(mult(exp(s1,3),s3),mult(s1,s5),exp(s1,6),exp(s3,2))
213       if not isZero(det4):
214           return 3
215       det3 = add(exp(s1,3),s3)
216       if not isZero(det3):
217           return 2
218       det2 = s1
219       if not isZero(det2):
220           return 1
221       return 0
222
223   def test_determinant_2(synd):
224       s1 = synd[0]
225       s3 = synd[1]
226       det3 = add(exp(s1,3),s3)
227       if not isZero(det3):
228           return 2
229       det2 = s1
230       if not isZero(det2):
231           return 1
232       return 0
233
234   def test_syndrom_4(synd,e_1,e_2):
```

```
235        s1 = synd[0]
236        s3 = synd[1]
237        s5 = synd[2]
238        s7 = synd[3]
239        if isZero(s1) and isZero(s3) and isZero(s5) and isZero(s7):
240            return 0
241        if s3 == exp(s1,3) and s5 == exp(s1,5) and s7 == exp(s1,7):
242            return 1
243        if e_1 is not None and e_2 is not None:
244            v_1 = get_value_table(e_1)
245            v_2 = get_value_table(e_2)
246            ns1 = add(v_1,v_2)
247            ns3 = add(exp(v_1,3),exp(v_2,3))
248            ns5 = add(exp(v_1,5),exp(v_2,5))
249            ns7 = add(exp(v_1,7),exp(v_2,7))
250            if s1 == ns1 and s3 == ns3 and s5 == ns5 and s7 == ns7:
251                return 2
252        return 10
253
254    def test_syndrom_3(synd,e_1,e_2):
255        s1 = synd[0]
256        s3 = synd[1]
257        s5 = synd[2]
258        if isZero(s1) and isZero(s3) and isZero(s5):
259            return 0
260        if s3 == exp(s1,3) and s5 == exp(s1,5):
261            return 1
262        s7 = synd[3]
263        if e_1 is not None and e_2 is not None:
264            v_1 = get_value_table(e_1)
265            v_2 = get_value_table(e_2)
266            ns1 = add(v_1,v_2)
267            ns3 = add(exp(v_1,3),exp(v_2,3))
268            ns5 = add(exp(v_1,5),exp(v_2,5))
269            if s1 == ns1 and s3 == ns3 and s5 == ns5:
270                return 2
271        return 10
272
273    def test_syndrom_2(synd,e_1,e_2):
274        s1 = synd[0]
275        s3 = synd[1]
276        if isZero(s1) and isZero(s3):
277            return 0
278        if s3 == exp(s1,3):
279            return 1
280        if e_1 is not None and e_2 is not None:
281            v_1 = get_value_table(e_1)
282            v_2 = get_value_table(e_2)
283            ns1 = add(v_1,v_2)
284            ns3 = add(exp(v_1,3),exp(v_2,3))
285            if s1 == ns1 and s3 == ns3:
286                return 2
287        return 10
288
289    def main(s_polynom, data, test_syndrome=True, error_count=2,debug=False,
```

```
        use_count=4):
290     h = data["H"]
291     cb = data["checksyndroms"]
292     sl = data["syndromlength"]
293     pol = data["polynom"]
294     twoBit_reverse_table = data["reverse_table"]
295     cwl = len(h[0])
296     build_alpha_table(pol)
297     dh = alpha_matrix_from_h(h,cb,sl)
298     oneBit_reverse_table = {}
299     for i in range(cwl):
300         val = [dh[k][i] for k in range(len(dh))]
301         oneBit_reverse_table[str(val[0])]=(i,val[1:])
302     if debug:
303         print "Size: "+str(cwl)
304         print "H = \n\t","\n\t".join([str(x) for x in h])
305         print "DH = \n\t","\n\t".join([str(x) for x in dh])
306         print str(len(dh))+" Syndrome Components"
307     total = 0
308     test_determinant = test_determinant_4
309     test_syndrom = test_syndrom_4
310     if use_count == 3:
311         test_determinant = test_determinant_3
312         test_syndrom = test_syndrom_3
313     elif use_count == 2:
314         test_determinant = test_determinant_2
315         test_syndrom = test_syndrom_2
316     numBits = error_count
317     if True:
318         if debug: print "--- "+("Syndrome" if test_syndrome else "Determinant
                ")+" Checking "+str(numBits)+"-Bit Errors ---"
319         correctedCount = 0
320         incorrectCount = 0
321         errs, can_continue = error_iterator(None,cwl,numBits)
322         while can_continue:
323             synd = [dh[k][errs[0]] for k in range(len(dh))]
324             for j in errs[1:]:
325                 synd = [add(synd[k],dh[k][j]) for k in range(len(dh))]
326             total += 1
327             if not "1" in str(synd):
328                 undetectedCount +=1
329             else:
330                 if test_syndrome:
331                     e_1_a = oneBitCorrection(synd,oneBit_reverse_table)
332                     e_1, e_2 = twoBitCorrection(synd,twoBit_reverse_table)
333                     identifiedNumber = test_syndrom(synd,e_1,e_2)
334                     if identifiedNumber != numBits:
335                         incorrectCount += 1
336                     else:
337                         correctedCount += 1
338                 else:
339                     identifiedNumber = test_determinant(synd)
340                     if identifiedNumber == 1:
341                         e_1 = oneBitCorrection(synd,oneBit_reverse_table)
342                     elif identifiedNumber == 2:
```

```
343                         e_1, e_2 = twoBitCorrection(synd,twoBit_reverse_table
                                )
344                     if identifiedNumber != numBits:
345                         incorrectCount += 1
346                     else:
347                         correctedCount += 1
348                 errs, can_continue = error_iterator(errs,cwl,numBits)
349         if debug: print "Total:",str(total), \
350                 "\n\tCorrect:",str(correctedCount), \
351                 "\n\tIncorrect:",str(incorrectCount)
352 
353 if __name__ == "__main__":
354     use_synd = True
355     errs = 2
356     reps = 1
357     debug = False
358     if len(sys.argv) > 1:
359         use_synd = True if sys.argv[1] == '1' else False
360     if len(sys.argv) > 2:
361         errs = int(sys.argv[2])
362     if len(sys.argv) > 3:
363         reps = int(sys.argv[3])
364     if len(sys.argv) > 4:
365         use_count = int(sys.argv[4])
366     s_polynom = "10001001"
367     data = load_json(s_polynom+"_rt3.json")
368     for i in range(reps):
369         main(s_polynom,data,use_synd,errs,debug,use_count)
370     if debug: print "exiting."
```

Listing 5.9: Pythonskript zum Vergleich des neuen Ansatzes gegen die Berechnung von Determinanten in Software

```
 1 def gen_determinants(synd):
 2     s1 = synd[0]
 3     s3 = synd[1]
 4     s5 = synd[2]
 5     s7 = synd[3]
 6     det5 = add(mult(s1,exp(s3,3)),exp(s5,2),mult(exp(s1,7),s3),mult(exp(s1,2)
           ,s3,s5),exp(s1,10),mult(exp(s1,3),s7),mult(exp(s1,5),s5),mult(s3,s7))
 7     det4 = add(mult(exp(s1,3),s3),mult(s1,s5),exp(s1,6),exp(s3,2))
 8     det3 = add(exp(s1,3),s3)
 9     det2 = s1
10     return det2, det3, det4, det5
11 
12 def main(s_polynom,data,error_count=2,debug=False):
13     h = data["H"]
14     cb = data["checksyndroms"]
15     sl = data["syndromlength"]
16     pol = data["polynom"]
17     twoBit_reverse_table = data["reverse_table"]
18     cwl = len(h[0])
19     build_alpha_table(pol)
20     dh = alpha_matrix_from_h(h,cb,sl)
21     oneBit_reverse_table = {}
```

```
22      for i in range(cwl):
23          val = [dh[k][i] for k in range(len(dh))]
24          oneBit_reverse_table[str(val[0])]=(i,val[1:])
25      if debug:
26          print "Size: "+str(cwl)
27          print "H = \n\t","\n\t".join([str(x) for x in h])
28          print "DH = \n\t","\n\t".join([str(x) for x in dh])
29          print str(len(dh))+" Syndrome Components"
30      total = 0
31      numBits = error_count
32      if True:
33          counter = 0
34          errs, can_continue = error_iterator(None,cwl,numBits)
35          print str(error_count),"\tM(2)\tM(3)\tM(4)\tM(5)"
36
37          while can_continue and counter < 500:
38              counter+=1
39              synd = [dh[k][errs[0]] for k in range(len(dh))]
40              for j in errs[1:]:
41                  synd = [add(synd[k],dh[k][j]) for k in range(len(dh))]
42              if "1" in str(synd):
43                  det2, det3, det4, det5 = gen_determinants(synd)
44                  d = ("0" if isZero(det2) else "1") + ("0" if isZero(det3)
                        else "1") + ("0" if isZero(det4) else "1") + ("0" if
                        isZero(det5) else "1")
45                  sys.stdout.write(d+" ")
46                  if counter % 20 == 0:
47                      sys.stdout.write("\n")
48                      sys.stdout.flush()
49
50              errs, can_continue = error_iterator(errs,cwl,numBits)
51
52          sys.stdout.write("\n")
53          sys.stdout.flush()
54
55
56  if __name__ == "__main__":
57      use_synd = True
58      max_errs = 20
59      debug = False
60      s_polynom = "10001001"
61      data = load_json(s_polynom+"_rt3.json")
62      for errs in range(1,max_errs+1):
63          main(s_polynom,data,errs,debug)
64      if debug: print "exiting."
```

Listing 5.10: Determinante für höhere Fehler

```
1  1        M(2)     M(3)     M(4)     M(5)
2  1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
       1000 1000 1000 1000 1000
3  1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
       1000 1000 1000 1000 1000
4  1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
       1000 1000 1000 1000 1000
```

```
 5  1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
        1000 1000 1000 1000 1000
 6  1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
        1000 1000 1000 1000 1000
 7  1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
        1000 1000 1000 1000 1000
 8  1000 1000 1000 1000 1000 1000 1000
 9  2         M(2)      M(3)      M(4)      M(5)
10  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
11  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
12  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
13  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
14  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
15  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
16  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
17  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
18  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
19  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
20  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
21  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
22  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
23  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
24  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
25  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
26  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
27  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
28  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
29  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
30  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
31  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
32  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
        1100 1100 1100 1100 1100
33  1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
```

```
         1100 1100 1100 1100 1100
34   1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
         1100 1100 1100 1100 1100
35
36   3        M(2)      M(3)      M(4)      M(5)
37   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
38   1110 1110 1110 1110 1110 1110 1110 1110 1110 0110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
39   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
40   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
41   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
42   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
43   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
44   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
45   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
46   1110 1110 1110 1110 0110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
47   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
48   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
49   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 0110 1110 1110
         1110 1110 1110 1110 1110
50   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
51   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
52   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
53   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
54   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
55   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
56   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
57   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
58   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
59   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
60   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110
         1110 1110 1110 1110 1110
61   1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 0110 1110 1110 1110
         1110 1110 1110 1110 1110
```

```
62
63  4          M(2)      M(3)      M(4)      M(5)
64  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
65  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
66  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
67  1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1011 1111 1111
68  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
69  0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
70  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        0111 1111 1111 1111 1111
71  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
72  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
73  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
74  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
75  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
76  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
77  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
78  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
79  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
80  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
81  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
82  1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
83  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
84  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
85  1111 1011 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
86  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
87  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
88  1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
89
90  5          M(2)      M(3)      M(4)      M(5)
91  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
```

```
         1111 1111 1111 1111 1111
 92  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
 93  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
 94  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
 95  1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1101 1111 1111
         1111 1111 1111 1111 1111
 96  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
 97  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1011 1111 1111 1111 1111
 98  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
 99  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
100  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
101  1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
102  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111
         1111 1111 1111 1111 1111
103  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
104  1111 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
105  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
106  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 0111 1111 1111 1111
107  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
108  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
109  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
110  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
111  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
112  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1101 1111 1111 1111 1111
113  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
114  1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
115  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
116
117  6          M(2)      M(3)      M(4)      M(5)
118  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
119  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111
         1111 1111 1111 1111 1111
```

```
120  1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
121  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
122  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1011
123  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1011
124  1111 1111 1110 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 0111
125  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
126  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
127  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
128  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
129  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
130  1111 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
131  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
132  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
133  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
134  1111 1111 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
135  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
136  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
137  1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
138  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
139  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
140  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
141  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
142  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
143
144  7          M(2)       M(3)       M(4)       M(5)
145  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
146  1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
147  1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111
         1111 1111 1111 1111 1111
148  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
```

```
          1111 1111 1111 1111 1111
149  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
150  1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
151  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
152  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
153  1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111
          1111 1111 1111 1011 1111
154  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
155  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
156  1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
157  1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
158  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
159  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
160  1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
161  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111
          1111 1111 1111 1111 1111
162  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
163  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
164  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
165  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
166  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          0111 1111 1111 1111 1111
167  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111
          1111 1111 1111 1111 1111
168  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
169  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1011
          1111 1111 1111 1111 1111
170
171  8       M(2)     M(3)     M(4)     M(5)
172  1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
173  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
174  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
175  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
          1111 1111 1111 1111 1111
176  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111
          1111 1111 1111 1111 1111
```

```
177  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
178  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
179  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111
         1111 1111 1111 1111 1111
180  1111 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
181  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
182  1111 1111 1111 1110 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
183  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
184  1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
185  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
186  1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 0111
187  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
188  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111
         1111 1110 1111 1111 1111
189  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
190  1111 1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
191  1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1101 1111
192  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
193  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
194  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111
         1111 1111 1111 1111 1111
195  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
196  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
197
198  9        M(2)    M(3)    M(4)    M(5)
199  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
200  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
201  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
202  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
203  1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
204  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1110 1111 1111
205  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111
```

```
         1111 1111 1111 1111 1111
206  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
207  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111
         1111 1111 1111 1111 1111
208  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
209  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
210  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
211  1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
212  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
213  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
214  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
215  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 0111 1111 1111 1111
216  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1110 1011 1111
217  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 1111
         1111 1111 1111 1111 1111
218  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
219  0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
220  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1011 1111
221  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1101
222  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
223  1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
224
225  10       M(2)     M(3)     M(4)     M(5)
226  1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111
         1111 1111 1111 1111 1111
227  1111 1110 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
228  1111 1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1011
         1111 1111 1111 1111 1111
229  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1101 1111
230  1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
231  1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
232  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
233  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1110
```

```
234  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 0111 1111 1111 1111
235  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
236  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
237  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
238  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
239  1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 0111 1111 1111 1111
240  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
241  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
242  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
243  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
244  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
245  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
246  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111
        1111 1111 1111 1111 1111
247  1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
248  1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1110 1111 1111
        1111 1111 1111 1111 1111
249  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
250  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111
        1111 1111 1111 1111 1111
251
252  11      M(2)    M(3)    M(4)    M(5)
253  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
254  1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
255  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
256  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
257  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
258  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
259  1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1011 1111 1111 1111 1111
260  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111
        1111 1111 1111 1111 1111
261  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
262  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
```

```
       1111 1111 1111 1111 1111
263 1111 1111 1111 1101 1111 1110 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
264 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
265 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
266 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
267 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111
       1111 1111 1111 1111 1111
268 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111
       1111 1111 1111 1111 1111
269 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111
       1111 1111 1111 1111 1111
270 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
271 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 0111 1111
272 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
273 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1011
274 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
275 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
276 1111 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
277 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
278
279 12       M(2)      M(3)      M(4)      M(5)
280 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1110 1111 1111
281 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
282 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
283 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
284 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
285 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1101 1111
286 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
287 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
288 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111
       1111 1111 1111 1111 1111
289 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111
       1111 1111 1111 1111 1111
290 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
```

291 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1110
1111 1111 1111 0110 1111
292 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
293 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111
1111 1111 1111 1111 1111
294 1111 1111 1111 1101 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
295 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
296 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
297 1111 1110 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
298 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
299 1111 1011 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1101 1111 1111 1111
300 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
301 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
302 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
303 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
304 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
305
306 13 M(2) M(3) M(4) M(5)
307 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111
1110 1111 1111 1111 1111
308 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
309 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1101 1011
310 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
311 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
312 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111
1111 1011 1111 1111 1111
313 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
314 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1110 1111 1111
1111 1111 1111 1111 1111
315 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
316 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
317 1011 1111 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
318 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111 1101 1111 1111 1111 1111
1111 1111 1111 1111 1111
319 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111

```
       0111 1111 1111 1111 1111
320  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1110 1111 1111 1111 1111
321  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1101 1111 1111 1111 1111
322  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
323  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1101 1111 1111 1111 1111
324  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
325  1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
326  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
327  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111
       1111 1111 1111 1111 1111
328  1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
329  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
330  1011 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
331  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
332
333  14        M(2)     M(3)     M(4)     M(5)
334  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
335  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1011 1111 1111 1111 1111
336  1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
337  1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
338  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
339  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
340  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
341  1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
342  1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
343  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
344  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111
       1111 1111 1111 1111 1111
345  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1101 1111 1111
346  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 1111 1111
347  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
       1111 1111 1111 0111 1111
```

```
348  1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1110 1111
349  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
350  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
351  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1110 1111 1111
352  1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
353  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
354  1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
355  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
356  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
357  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
358  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
359
360  15      M(2)    M(3)    M(4)    M(5)
361  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1110 1111
362  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
363  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
364  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111
        1111 1111 1111 1111 1111
365  1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
366  1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
367  1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
368  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111
        1111 1101 1111 1111 1111
369  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
370  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1011 1111 1111 1111
371  1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111
        1111 1111 0111 1111 1111
372  1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
373  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
374  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111
        1111 1110 1111 1111 1111
375  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111
        1111 1111 1111 1111 1111
376  1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
```

```
        1111 1111 1111 1111 1111
377  1111 1111 1111 1111 1111 1111 1111 1110 1111 1011 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
378  1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
379  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
380  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
381  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1101
382  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
383  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
384  1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
385  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1011
386
387  16    M(2)   M(3)   M(4)   M(5)
388  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 0111 1111 1111
389  1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111
        1111 1111 1111 1111 1111
390  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
391  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
392  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
393  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1011
394  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110
        1111 1111 1111 1111 1111
395  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
396  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
397  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
398  1111 1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111
        1111 1111 1111 1111 0111
399  1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
400  1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
401  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
402  1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1101 1111 1111 1111 1111
403  1111 1011 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
404  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
```

405 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
406 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111
1111 1111 1111 1111 1111
407 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
408 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
409 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
410 1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
411 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111
1111 1111 1111 1111 1111
412 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
413
414 17 M(2) M(3) M(4) M(5)
415 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
416 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
417 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
418 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
419 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
420 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
421 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
422 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
423 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111 1011
1111 1111 1111 1111 1111
424 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
425 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
426 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111
1111 1111 1111 1111 1111
427 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111
1111 1111 1110 1111 1111
428 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
429 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
430 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111
431 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
0111 1111 1111 1111 1111
432 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111
1111 1111 1111 1111 1111
433 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111

```
        1111 1111 1111 1111 1111
434 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
435 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
436 1101 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
437 1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111
        1111 1111 1111 1111 1111
438 1111 1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
439 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
440
441 18      M(2)     M(3)     M(4)     M(5)
442 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
443 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
444 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
445 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
446 1111 1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
447 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
448 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
449 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
450 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111
        1111 1111 1111 1111 1111
451 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
452 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
453 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
454 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1011
        1111 1111 1111 1111 0111
455 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111
        1111 1111 1111 1111 1111
456 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
457 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
458 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
459 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
460 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
        1111 1111 1111 1111 1111
461 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0111 1111
        1111 1111 1111 1111 1111
```

```
462  1111 1111 1111 1111 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
463  1111 1111 1110 1111 1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
464  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
465  1111 1111 1111 1011 1111 1111 1111 1110 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
466  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1011
467
468  19        M(2)      M(3)      M(4)      M(5)
469  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1011 1111 1111
470  1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 0111
471  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
472  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
473  1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1101 1111 1111 1111 1111
         1111 1111 1111 1111 1111
474  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
475  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
476  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
477  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
478  1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
479  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
480  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
481  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 1111 1111
         1111 1111 1111 1111 1111
482  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
483  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
484  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1011 1111 1110 1111
485  1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1110 1111 1111 1111
486  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
487  1101 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111 1111
         1111 1111 1111 1111 1111
488  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
489  1111 1111 1111 1101 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111
         1111 1111 1111 1111 1111
490  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
```

```
         1111 1111 1111 1111 1111
491  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
492  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
493  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 0111 1011
494
495  20       M(2)     M(3)     M(4)     M(5)
496  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
497  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
498  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 0111 1111 1111 1111
499  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 1111
         1111 1111 1111 1111 1111
500  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
501  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
502  1111 1111 1101 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1101
503  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
504  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
505  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 0111 1111 1111
         1111 1111 1111 1111 1111
506  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
507  1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
508  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
509  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1011 1111 1111 1111
510  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
511  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
512  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
513  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
514  1111 1110 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
515  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
516  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1111 1111 1111
         1111 1111 1111 1111 1111
517  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
518  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1011 1111 1111
         1111 1111 1111 1111 1111
```

```
519  1111 1111 1111 1111 1111 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
520  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
         1111 1111 1111 1111 1111
```

Listing 5.11: Determinante für höhere Fehler, die Zahlenpaare aus 4 Ziffern stellen jeweils die vier berechenbaren Determinanten dar. Der Wert 1 steht für eine Determinante ungleich 0. Hier werden pro Fehleranzahl die ersten bis zu 500 Fehler aufgeführt.

Literaturverzeichnis

[Ber68] Elwyn Berlekamp. *Algebraic coding theory*. World Scientific, 1968.

[Bew02] Jörg Bewersdorff. *Algebra für Einsteiger*. Springer, 2002.

[BRC60] Raj Chandra Bose and Dwijendra K Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and control*, 3(1):68–79, 1960.

[Chi64] Robert Chien. Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes. *IEEE Transactions on information theory*, 10(4):357–363, 1964.

[Hub90] Klaus Huber. Some comments on Zech's logarithms. *IEEE Transactions on Information Theory*, 36(4):946–950, 1990.

[LC04] Shu Lin and Daniel J Costello. Error Control Coding, 2004.

[Mas69] James Massey. Shift-register synthesis and BCH decoding. *IEEE transactions on Information Theory*, 15(1):122–127, 1969.

[OI87] Hirokazu Okano and Hideki Imai. A construction method of high-speed decoders using rom. *IEEE transactions on computers*, (10):1165–1171, 1987.

[Pet60] Wesley Peterson. Encoding and error-correction procedures for the Bose-Chaudhuri codes. *IRE Transactions on Information Theory*, 6(4):459–470, 1960.

[SH20] Christian Schulz-Hanke. Fast BCH 1-bit error correction combined with fast multi-bit error detection. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–5. IEEE, 2020.

[SH21] Christian Schulz-Hanke. BCH 2-Bit and 3-Bit Error Correction with Fast Multi-Bit Error Detection. In *Architecture of Computing Systems: 34th International Conference, ARCS 2021, Virtual Event, June 7–8, 2021, Proceedings 34*, pages 201–212. Springer, 2021.

[SH22] Christian Schulz-Hanke. Vereinfachung der Bestimmung von 4-Bit Fehlern für BCH Codes. In *TuZ'22: 34. GI / GMM / ITG-Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen*, 2022.

[TH13] Hans Tzschach and Gerhard Hasslinger. *Codes für den störungssicheren Datentransfer*. Oldenbourg Wissenschaftsverlag, 2013.

[Wic95] Stephen B Wicker. *Error control systems for digital communication and storage*, volume 1. Prentice hall Englewood Cliffs, 1995.