Exzellenzcluster
Cognitive Interaction Technology
Kognitronik und Sensorik
Prof. Dr.-Ing. U. Rückert

Ressourceneffiziente Hardware-Software-Kombinationen für Kryptographie mit elliptischen Kurven

zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEUR (Dr.-Ing.)

der Technischen Fakultät
der Universität Bielefeld

genehmigte Dissertation

von

Dipl.-Ing. Christoph Puttmann

Referent:	Prof. Dr.-Ing. Ulrich Rückert
Korreferent:	Prof. Dr. Joachim von zur Gathen

Tag der mündlichen Prüfung: 20.12.2013

Bielefeld / Januar 2014
DISS KS / 06

Zusammenfassung

In der heutigen Informationsgesellschaft spielt die sichere Übertragung von elektronischen Daten eine immer wichtigere Rolle. Die hierfür eingesetzten Endgeräte beschränken sich mittlerweile nicht mehr auf klassische, stationäre Computer, sondern es setzen zunehmend mobile Alltagsgegenstände (z. B. Smartphone oder Reisepass) eine sichere Datenübertragung zwingend voraus. Die Anforderungen bezüglich der Ressourcen einer Hardware-Software-Kombination variieren dabei für verschiedene Anwendungsszenarien sehr stark.

Kryptographie auf Basis von elliptischen Kurven stellt eine attraktive Alternative zu etablierten asymmetrischen Verfahren dar und wird vermehrt eingesetzt, um sicherheitskritische Daten zu ver- bzw. entschlüsseln sowie deren Integrität und Authentizität sicherzustellen.

Im Rahmen dieser Arbeit werden, am Beispiel von Algorithmen für die Kryptographie mit elliptischen Kurven, verschiedene Methoden vorgestellt, um ressourceneffiziente Hardware-Software-Kombinationen zu entwickeln. Es wird eine automatisierte Testumgebung vorgestellt, welche die systematische Entwicklung von ressourceneffizienten Hardware-Software-Kombinationen ermöglicht. Um verschiedene Implementierungen im Hinblick auf ein spezielles Anwendungsszenario miteinander vergleichen zu können, wird eine allgemeine Bewertungsmetrik eingeführt, welche die drei wesentlichen Parameter (Chipfläche, Verlustleistung, Ausführungsdauer) des Entwurfsraumes einer ASIC-Entwicklung berücksichtigt.

Basierend auf einer hierarchisch entwickelten, skalierbaren Systemarchitektur wird eine Entwurfsraumexploration für zwei exemplarische Anwendungsszenarien durchgeführt. Mit den angewandten Konzepten der Instruktionssatzerweiterung, der Parallelisierung sowie eines Coprozessor-Ansatzes wird die Ressourceneffizienz auf unterschiedlichen Hierarchieebenen der zugrundeliegenden Systemarchitektur anwendungsspezifisch optimiert. Die Ergebnisse werden mit Hilfe einer FPGA-basierten Entwicklungsumgebung prototypisch evaluiert sowie durch eine ASIC-Realisierung in einer 65-nm-CMOS-Standardzellentechnologie praktisch belegt.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Zielsetzung 3
1.2 Struktur der Arbeit 4

2 Grundlagen der Kryptographie **7**
2.1 Kryptographische Verfahren 7
2.1.1 Symmetrische Kryptographie 8
2.1.2 Asymmetrische Kryptographie 9
2.2 Elliptische Kurven 10
2.2.1 Gruppen 11
2.2.2 Körper 12
2.2.3 Elliptische Kurven in der Kryptographie 15
2.3 Sicherheit kryptographischer Verfahren 17
2.4 Stand der Technik 20
2.5 Stand der Forschung 23
2.6 Zusammenfassung 24

3 Algorithmen und Methoden **27**
3.1 Skalarmultiplikation auf elliptischen Kurven 27
3.1.1 Montgomery-Leiter 28
3.2 Punktaddition und Punktverdopplung 30
3.2.1 Koordinatentransformation 30
3.2.2 Montgomery-Leiter nach López-Dahab 33
3.3 Arithmetische Operationen im Binärkörper 35
3.3.1 Addition 36
3.3.2 Multiplikation 36

3.3.3 Quadrierung . . . 41
3.3.4 Modulo-Reduktion . . . 43
3.3.5 Invertierung . . . 44
3.4 Zusammenfassung . . . 47

4 Entwurfsraum **49**
4.1 Anwendungsszenarien . . . 49
4.1.1 Chipkarte . . . 50
4.1.2 Sicherheitsserver . . . 50
4.2 ASIC-Entwurfsablauf . . . 51
4.3 Ressourceneffizienz . . . 54
4.3.1 Chipfläche . . . 54
4.3.2 Leistungsaufnahme . . . 55
4.3.3 Rechenleistung . . . 56
4.3.4 Bewertungsmetrik . . . 57
4.3.5 Skalierungsregeln . . . 59
4.4 Hierarchischer Systementwurf . . . 60
4.4.1 Einzelkernprozessor . . . 60
4.4.2 Prozessorfeld . . . 64
4.4.3 Multiprozessorsystem . . . 66
4.5 Prototypische Realisierung . . . 68
4.5.1 RAPTOR2000 Rapid-Prototyping-Plattform . . . 69
4.6 Zusammenfassung . . . 73

5 Evaluierung der Algorithmen **75**
5.1 Software-Evaluierung . . . 75
5.1.1 Automatisierte Testumgebung . . . 76
5.1.2 Evaluierung auf Wort-Ebene . . . 77
5.1.3 Evaluierung auf Körper-Ebene . . . 78
5.2 Hardware-Evaluierung . . . 84
5.2.1 Klassische Multiplikation . . . 84
5.2.2 Karatsuba-Multiplikation . . . 86
5.2.3 Hybride Multiplikation . . . 88
5.2.4 Codegenerator . . . 91

5.3 Zusammenfassung 93

6 Hardware-Software-Kombinationen 95

6.1 Instruktionssatzerweiterung 96

6.1.1 Entwurfsmethodik 99

6.1.2 Implementierungsvarianten 101

6.1.3 Ressourceneffizienz der Instruktionssatzerweiterung 109

6.2 Parallelisierung 114

6.2.1 QuadroCore 117

6.2.2 CoreVA 122

6.2.3 Ressourceneffizienz der Parallelisierung 126

6.3 Coprozessor-Modul 129

6.3.1 Performanzanalyse des On-Chip-Netzwerks 132

6.4 Zusammenfassung 137

7 Zusammenfassung und Ausblick 139

Abkürzungsverzeichnis 145

Symbolverzeichnis 149

Algorithmenverzeichnis 151

Abbildungsverzeichnis 153

Tabellenverzeichnis 157

Literaturverzeichnis 159

Anhang 179

A Kurvenparameter 179

B Quellcode 181

B.1 Binärkörper-Multiplikation 182

B.2 Performanz-Analysator für On-Chip-Netzwerke 186

1 Einleitung

Im Zeitalter der heutigen Informationsgesellschaft spielt die sichere Übertragung von elektronischen Daten eine wichtige Rolle. So sind beispielsweise in den letzten Jahren eine Vielzahl an Anwendungen im Bereich des elektronischen Handels, dem so genannten E-Commerce, entstanden. Hierzu zählen unter anderem das Online-Shopping und Online-Banking. Wie aus Abbildung 1.1 zu erkennen ist, nutzen immer mehr Menschen das Internet, um auch vertrauliche Informationen oder personenbezogene Daten zu übermitteln. Damit diese Informationen vor dem Mitlesen durch unbefugte Personen geschützt sind, werden kryptographische Verfahren eingesetzt. Neben dem Ver- bzw. Entschlüsseln von Daten bietet die so genannte asymmetrische Kryptographie zusätzlich die Möglichkeit der Authentifizierung sowie der Integritätsprüfung. Hierdurch kann neben dem Mitlesen geheimer Informationen auch die Manipulation verhindert bzw. erkannt werden (Integrität) sowie die Zustellung der Nachricht an ausschließlich berechtigte Personen (Authentifizierung) gewährleistet werden. Diese Eigenschaften ermöglichen weitere Anwendungen, wie

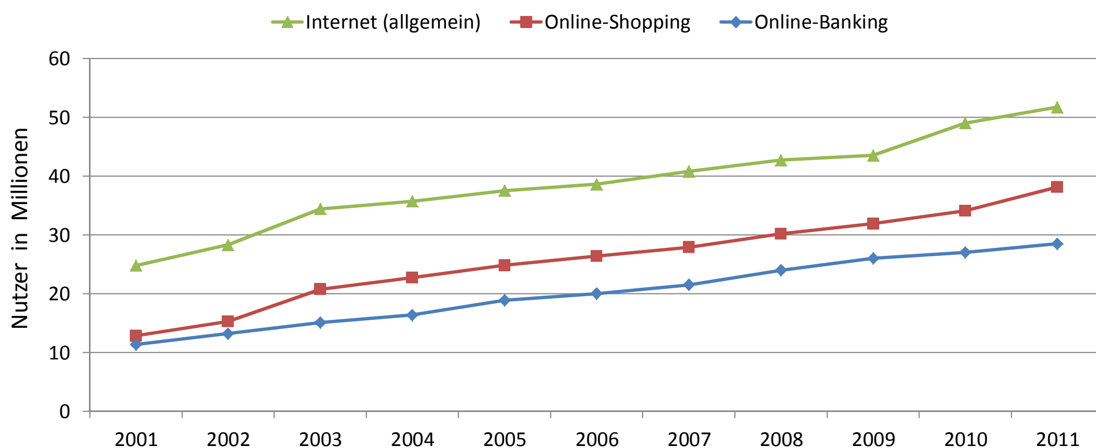


Abbildung 1.1: Internetnutzung in Deutschland [BDB11][BVH11][TNS11]

beispielsweise den sicheren Austausch elektronischer Nachrichten (E-Mail) oder das digitale Unterschreiben von Dokumenten mit Hilfe einer elektronischen Signatur. Letzteres wird zum Beispiel im Bereich E-Government bei der elektronischen Steuererklärung (ELSTER) oder dem elektronischen Entgeltnachweis (ELENA) sowie im Bereich E-Health bei der elektronischen Gesundheitskarte eingesetzt.

(a) elektronischer Personalausweis [BSI11] (b) elektronische Gesundheitskarte [BMG11]

Abbildung 1.2: Beispiele von Chipkarten mit kryptographischen Funktionen

Traditionell werden die soeben beschriebenen Anwendungen auf stationären Rechnern (*Desktop Computer*) mit großem Arbeitsspeicher, ausreichend Rechenleistung und quasi unbeschränkter Energieversorgung genutzt. Etwa ein Drittel der Internetnutzer in Deutschland verwenden heutzutage auch schon mobile Alternativen, wie z. B. tragbare Computer (*Tablet-PC*) oder Mobiltelefone (*Smartphone*) [BIT12]. In Zukunft werden immer mehr sicherheitsrelevante Anwendungen auf eingebetteten Geräten, wie beispielsweise Chipkarten (*Smartcards*), ausgeführt werden. Diese Geräte verfügen in der Regel nur über sehr begrenzte Ressourcen hinsichtlich der Rechenleistung, der Speichergröße sowie der Energieversorgung. Im Extremfall, wie z. B. bei passiven RFID[1]-Transpondern, wird zusätzlich die Kommunikationsbandbreite und der maximale Leistungsverbrauch stark eingeschränkt. Bekannte Beispiele für RFID-basierte Chipkarten mit kryptographischen Funktionen sind der elektronische Personalausweis (Abbildung 1.2a) und Reisepass. Die elektronische Gesundheitskarte (Abbildung 1.2b) wird als kontaktbehaftete Chipkarte vertrieben und enthält ebenfalls Sicherheitsmechanismen auf Basis asymmetrischer Kryptographie [BSI12a].

[1]Radio Frequency Identification

Im Jahr 1978 veröffentlichten Ronald L. Rivest, Adi Shamir und Leonard Adleman ein Verfahren zur Realisierung von asymmetrischen Kryptosystemen [RSA78]. Das nach den Entwicklern benannte RSA-Verfahren hat sich in der Vergangenheit zum Quasistandard in vielen Bereichen der asymmetrischen Kryptographie etabliert. Als Alternative zu RSA wurden in den letzten Jahren Verfahren populär, die auf elliptischen Kurven basieren. Im Gegensatz zu bisherigen asymmetrischen Verfahren bietet die Elliptische-Kurven-Kryptographie (ECC, engl. Elliptic Curve Cryptography) ein hohes Sicherheitsmaß bei vergleichsweise kurzer Schlüssellänge. Hierdurch werden weniger Rechenleistung und Speicherbedarf als bei herkömmlichen Verfahren benötigt. Kryptographie mit elliptischen Kurven eignet sich daher besonders gut für eingebettete Systeme.

1.1 Zielsetzung

Das Ziel dieser Arbeit ist die Umsetzung neuer Algorithmen für kryptographische Verfahren in ressourceneffiziente Hardware-Software-Kombinationen. Dazu werden verschiedene Varianten von Hardware-Erweiterungen für Universalprozessoren analysiert, welche sich durch eine enge Kopplung zwischen Prozessor und Hardware-Beschleuniger auszeichnen. Neben Instruktionssatzerweiterungen werden Coprozessoren auf Basis anwendungsspezifischer Hardware betrachtet. Einen Schwerpunkt bildet neben der Hardware-Software-Partitionierung die Analyse und der Entwurf neuer Algorithmen für die Arithmetik in endlichen Körpern. Diese Verfahren werden vielfach in asymmetrischen Verschlüsselungssystemen eingesetzt. Die Ergebnisse werden für zwei Anwendungsszenarien prototypisch implementiert und auf ihre praktische Einsetzbarkeit hin getestet.

Ein Großteil dieser Arbeit ist im Rahmen des von der Deutschen Forschungsgemeinschaft (DFG) geförderten Projektes "Krypto-Hardware" entstanden [PSRG08]. Das in den Jahren 2004 bis 2009 unter der Kennung RU 477/8 geförderte Projekt ermöglichte eine enge Kooperation mit der Arbeitsgruppe *Computer Security (Cosec)* von Prof. Dr. Joachim von zur Gathen am Bonn-Aachen International Center for Information Technology (B-IT).

1.2 Struktur der Arbeit

Im Anschluss an diese Einleitung, ist die Arbeit in sechs weitere Kapitel unterteilt, welche im Folgenden zusammenfassend beschrieben werden:

Kapitel 2 führt zunächst allgemein in das Thema der Kryptographie ein. Es werden besonders die Unterschiede zwischen symmetrischen und asymmetrischen Verfahren erläutert. Im Anschluss werden die mathematischen Grundlagen beschrieben, welche zum Verständnis der später betrachteten Algorithmen und deren Optimierungen notwendig sind. Außerdem wird die Sicherheit kryptographischer Verfahren betrachtet sowie ein Überblick über den aktuellen Stand der Forschung und Technik gegeben.

Kapitel 3 stellt sämtliche Algorithmen vor, welche zur Umsetzung der kryptographischen Verfahren in dieser Arbeit herangezogen werden. Ferner werden verschiedene Methoden zur Optimierung der Algorithmen diskutiert. Die dargestellten Methoden und Algorithmen bilden die Grundlage für die in den folgenden Kapiteln beschriebenen Hardware-Software-Kombinationen.

Kapitel 4 widmet sich dem betrachteten Entwurfsraum. Zunächst werden alle in dieser Arbeit berücksichtigten Effekte des digitalen Schaltungsentwurfs erläutert. Im Anschluss wird die Ressourceneffizienz als zentrale Bewertungsmetrik dieser Arbeit definiert. Ferner werden zwei Anwendungsszenarien mit sehr unterschiedlichen Anforderungen hinsichtlich der betrachteten Ressourcen vorgestellt. Abschließend wird eine hierarchische Systemarchitektur präsentiert, welche die Basis für alle untersuchten Hardware-Software-Kombinationen darstellt.

Kapitel 5 beinhaltet die Evaluierung der in Kapitel 3 vorgestellten Algorithmen. Diese werden dazu auf der in Kapitel 4 präsentierten Systemarchitektur abgebildet. Zunächst wird eine reine Software-Lösung analysiert. Es wird eine automatisierte Testumgebung vorgestellt, welche die Evaluierung der Software-Umsetzung stark vereinfacht. Anschließend wird die Realisierung ausgewählter Funktionen als

Hardware-Implementierung untersucht. Daraus resultierend ist ein Codegenerator entstanden, welcher parametrisierbare Multipliziererschaltungen in einer Hardware-Beschreibungssprache erzeugt.

Kapitel 6 beschreibt verschiedene Hardware-Software-Kombination für Kryptographie mit elliptischen Kurven, welche auf der in Kapitel 4 eingeführten Systemarchitektur basieren. Zur effizienten Realisierung werden Instruktionssatzerweiterungen, Coprozessor-Module sowie die Parallelisierung von Algorithmen betrachtet. Mit Hilfe der Ressourceneffizienz werden schließlich die jeweiligen Implementierungsvarianten bewertet und hinsichtlich eines Anwendungsszenarios miteinander verglichen.

Kapitel 7 fasst die in dieser Arbeit behandelten Themen und alle daraus abgeleiteten Erkenntnissen zusammen. Ein Vergleich mit bereits existierenden Implementierungen verdeutlicht die Vor- bzw. Nachteile der in dieser Arbeit vorgestellten Hardware-Software-Kombinationen. Abschließend wird ein Ausblick auf zukünftige Entwicklungspotentiale gegeben.

2 Grundlagen der Kryptographie

In diesem Kapitel wird zunächst eine allgemeine Einführung in die Kryptographie vorgestellt. Dabei werden insbesondere die Unterschiede zwischen symmetrischen und asymmetrischen Verfahren erläutert. Im Anschluss wird die Mathematik der Kryptographie mit elliptischen Kurven dargestellt. Darauf folgt die Betrachtung der Sicherheit kryptographischer Verfahren. Abschließend wird ein Überblick über den aktuellen Stand der Forschung und Technik gegeben.

2.1 Kryptographische Verfahren

Der Begriff Kryptographie setzt sich aus den griechischen Wörtern *kryptos* = verborgen und *graphein* = schreiben zusammen und verkörpert demnach die Wissenschaft der Verschlüsselung von Informationen. Doch auch schon vor den Griechen wurden im dritten Jahrtausend v. Chr. die ersten kryptographischen Methoden im alten Ägypten benutzt. Seither wurde die Kryptographie in jeder Zeitepoche weiterentwickelt und meist für militärische Zwecke verwendet. Durch die Etablierung des Internets setzten sich in den neunziger Jahren schließlich kryptographische Verfahren auch im privaten Bereich durch, wie z. B. das Programm PGP[1] belegt [Zim95]. Heutzutage bildet die Kryptographie eine wichtige Voraussetzung für viele Internet-Dienste wie beispielsweise dem Online-Banking oder Online-Shopping.

Alle kryptographischen Verfahren basieren auf dem Prinzip eine Nachricht bzw. Daten im Klartext mit einer geeigneten mathematischen Funktion in einen Chiffretext umzuwandeln. Der verwendete Algorithmus muss dabei so beschaffen sein, dass es mit Hilfe von so genannten Schlüsseln einfach ist, sowohl die mathematische Funktion als auch dessen Umkehrfunktion zu berechnen. Ohne Kenntnis des

[1] Pretty Good Privacy

Schlüssels muss allerdings zumindest die Umkehrfunktion nur sehr schwer berechenbar sein. In der modernen Kryptographie wird dabei zwischen so genannten symmetrischen (Private-Key) und asymmetrischen (Public-Key) Verfahren unterschieden.

2.1.1 Symmetrische Kryptographie

Bei der symmetrischen bzw. Private-Key-Kryptographie verwendet der Empfänger zum Dechiffrieren denselben Schlüssel, den auch der Sender beim Chiffrieren der Nachricht benutzt hat. Abbildung 2.1 verdeutlicht dieses Verfahren. Alice möchte die Nachricht m verschlüsseln und an den Empfänger Bob schicken. Hierzu wird der Klartext m beim Sender mit Hilfe des geheimen Schlüssels K und einer geeigneten Chiffrierungsfunktion E in den Chiffretext $c = E(K, m)$ umgewandelt. Die so verschlüsselte Nachricht c wird nun an den Empfänger Bob geschickt. Dieser muss zur Dechiffrierung die entsprechende Umkehrfunktion D sowie denselben geheimen Schlüssel K verwenden, den auch Alice zum Chiffrieren verwendet hat. Nach dem Entschlüsseln kann Bob die Nachricht $m = D(K, c)$ wieder im Klartext lesen.

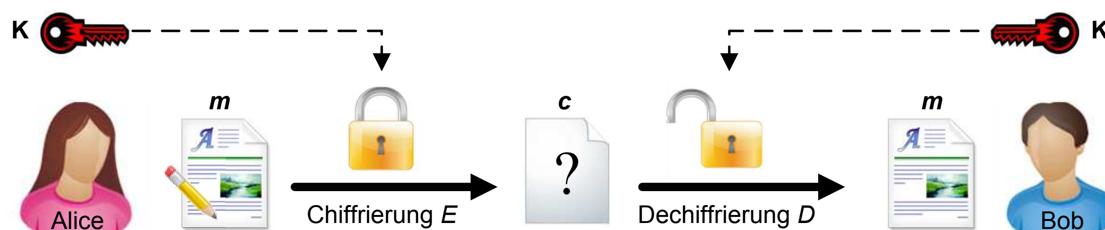


Abbildung 2.1: Verschlüsselte Kommunikation nach dem Private-Key-Verfahren

Zu den bekanntesten Verfahren in der symmetrischen Kryptographie gehören der Data Encryption Standard (DES) [NIST77] und dessen Nachfolger, der Advanced Encryption Standard (AES) [NIST01]. Symmetrische Verfahren besitzen eine hohe Sicherheit (siehe Abschnitt 2.3), solange ein Angreifer den gemeinsam verwendeten, geheimen Schlüssel (Private-Key) nicht herausfinden kann. Genau dieses ist jedoch auch gleichzeitig der Nachteil von symmetrischen Verfahren, denn damit zwei Kommunikationspartner einen geheimen Schlüssel aushandeln können, wird bereits ein sicherer Kommunikationskanal benötigt. Eine Lösung für dieses Problem bietet die asymmetrische Kryptographie.

2.1.2 Asymmetrische Kryptographie

Bei der asymmetrischen bzw. Public-Key-Kryptographie besitzt jeder Teilnehmer ein Schlüsselpaar, bestehend aus einem geheimen Schlüssel und einem öffentlichen Schlüssel, dem so genannten Public-Key. Um eine Nachricht zu verschlüsseln, wird der öffentliche Schlüssel des Empfängers verwendet. Dieser kann die Nachricht dann mit seinem geheimen Schlüssel wieder entschlüsseln. Abbildung 2.2 verdeutlicht dieses Verfahren. Alice verschlüsselt den Klartext m mit Hilfe der Chiffrierfunktion E und Bobs öffentlichem Schlüssel P_B. Der Chiffretext $c = E(P_B, m)$ kann nun über ein unsicheres Medium zum Empfänger Bob geschickt werden. Dieser benutzt zur Entschlüsselung seinen geheimen Schlüssel S_B und die Umkehrfunktion D, um wieder den Klartext $m = D(S_B, c)$ zu erhalten. Dieses asymmetrische Verfahren ist sicher, solange Bob niemanden seinen geheimen Schlüssel S_B verrät.

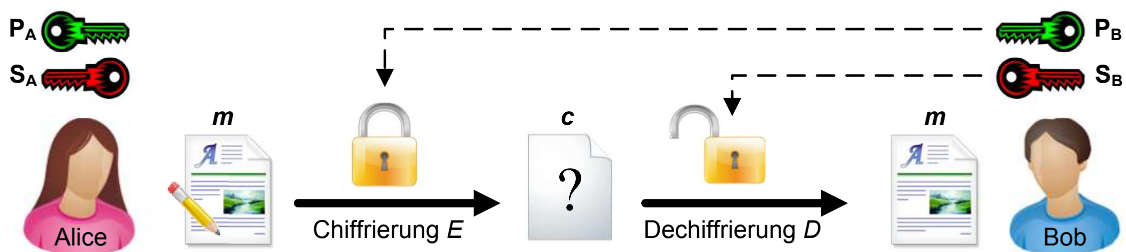


Abbildung 2.2: Verschlüsselte Kommunikation nach dem Public-Key-Verfahren

Mit asymmetrischen Verfahren ist es daher möglich, über ein unsicheres Medium sicher zu kommunizieren, ohne sich vorher auf einen gemeinsam verwendeten, geheimen Schlüssel einigen zu müssen. Im Vergleich zur symmetrischen Kryptographie besitzen asymmetrische Verfahren allerdings einen höheren Rechenaufwand, so dass man in der Praxis eine Kombination aus beiden Verfahren einsetzt. Hierbei wird nur zu Beginn einer Kommunikation ein geheimer Sitzungsschlüssel mittels asymmetrischer Kryptographie vereinbart. Die folgenden Kommunikationsdaten werden anschließend mit symmetrischen Verfahren und dem ausgehandelten Sitzungsschlüssel ver- bzw. entschlüsselt.

Die beschriebene Methode zur sicheren Schlüsselvereinbarung und das Senden verschlüsselter Nachrichten (vgl. Abbildung 2.2) gewährleisten die Vertraulichkeit, d. h. die gesendeten Informationen können nur von berechtigten Personen verarbeitet

werden. Die asymmetrische Kryptographie bietet darüber hinaus Methoden zur Überprüfung der Integrität und Authentizität. Mit Hilfe von digitalen Signaturen und Zertifikaten können Nachrichten vor Manipulation durch Dritte geschützt werden (Integrität) sowie die Identität des Senders verifiziert werden (Authentizität) [NIST94].

Die Theorie der asymmetrischen Kryptographie wurde 1976 von den Amerikanern Whitfield Diffie und Martin Hellman vorgestellt [DH76]. Die beschriebene Methode zum Aushandeln eines Sitzungsschlüssels ist daher auch als Diffie-Hellmann-Verfahren bekannt. Eine praktische Implementierung der asymmetrischen Kryptographie wurde 1978 von Ronald L. Rivest, Adi Shamir und Leonard Adleman veröffentlicht [RSA78]. Das RSA-Verfahren basiert auf dem Faktorisierungsproblem großer Zahlen. Hierbei wird angenommen, dass die Multiplikation zweier Primzahlen eine relativ einfache Rechenoperation, die Faktorisierung des Produktes jedoch eine sehr schwer zu berechnende Operation darstellt. Alternative Implementierungen von asymmetrischen Verfahren, wie beispielsweise die Kryptographie mit elliptischen Kurven, beruhen auf dem Diskreten-Logarithmus-Problem (DLP). Eine Erläuterung zum diskreten Logarithmus sowie zur Sicherheit von RSA bzw. von Verfahren, die auf elliptischen Kurven basieren, wird in Kapitel 2.3 vorgestellt. Zunächst wird im folgenden Abschnitt das allgemeine Prinzip von elliptischen Kurven in der Kryptographie eingeführt.

2.2 Elliptische Kurven

In der Mathematik werden elliptische Kurven schon seit dem 19. Jahrhundert erforscht. Sowohl in der Zahlentheorie, als auch in praktischen Anwendungen, wie beispielsweise der Faktorisierung großer Zahlen, werden elliptische Kurven benutzt. Den speziellen Einsatz von elliptischen Kurven in der Kryptographie entwickelten Neal Koblitz [Kob87] und Victor Miller [Mil85] unabhängig voneinander im Jahr 1985.

Im Folgenden werden einige mathematische Grundbegriffe eingeführt, die zum Verständnis der später beschriebenen Algorithmen benötigt werden. Die Definitionen sind möglichst allgemein verständlich gehalten. Tiefergehende Erklärungen sowie

Herleitungen und Beweise können in [Was03][CFA+05][HMV04][Buc08] nachgelesen werden.

2.2.1 Gruppen

Unter dem Begriff Gruppe versteht man in der Mathematik ein Tupel $(G, \circ)$ aus einer nicht leeren Menge G, auf der eine innere Verknüpfung $\circ$ definiert ist. Eine innere Verknüpfung sei hierbei die Abbildung $\circ : X \times X \to X$, die jedem Paar (x_1, x_2) von Elementen aus der Menge X ein Element $x_1 \circ x_2$ zuordnet. Ferner müssen folgende Bedingungen für die Gruppe $(G, \circ)$ gelten:

- Die Menge G ist unter der Verknüpfung $\circ$ abgeschlossen, d. h. für alle Elemente $a, b \in G$ gilt, dass auch die Verknüpfung $a \circ b$ ein Element aus G ist
- Es gilt das Assoziativgesetz: $(a \circ b) \circ c = a \circ (b \circ c) \quad \forall a, b, c \in G$
- Es existiert ein neutrales Element e, so dass gilt: $a \circ e = e \circ a = a \quad \forall a \in G$
- Es gibt ein inverses Element a^{-1}, so dass gilt: $a \circ a^{-1} = a^{-1} \circ a = e \quad \forall a \in G$

Falls zusätzlich noch das Kommutativgesetz gilt, d. h. $a \circ b = b \circ a \quad \forall a, b \in G$, so nennt man die Gruppe $(G, \circ)$ eine abelsche Gruppe.

Ein bekanntes Beispiel für eine abelsche Gruppe ist die Menge der ganzen Zahlen $\mathbb{Z}$ mit der Verknüpfung $+$ als übliche Addition. Diese Gruppe $(\mathbb{Z}, +)$ hat unendlich viele Elemente. Die Anzahl der Elemente einer Gruppe $(G, \circ)$ wird als Ordnung der Gruppe bezeichnet. Für kryptographische Zwecke kommen so genannte endliche Gruppen zum Einsatz, bei denen die zugrunde liegende Menge G aus einer endlichen Anzahl von Element besteht, also die Ordnung der Gruppe beschränkt ist.

Neben der Gruppenordnung ist ebenfalls die Ordnung eines Gruppenelementes a definiert als die kleinste positive ganze Zahl r für die gilt, dass $a^r = e$, also das Element a der Gruppe G r-mal mit sich selbst verknüpft das neutrale Element e ergibt.

Eine besondere Art von endlichen Gruppen sind zyklische Gruppen. Als zyklische Gruppen bezeichnet man solche Gruppen $(G', \circ)$, die ein Element g besitzen, aus

dem mittels der Verknüpfung $\circ$ alle anderen Elemente der Gruppe erzeugt werden können. Es gibt also für jedes Element a aus G' eine positive, ganze Zahl i, so dass die i-fache Verknüpfung von g mit sich selbst, also $g^i = g \circ g \cdots g = a$, das Element a erzeugt. Das Element g wird als Generator der zyklischen Gruppe $(G', \circ)$ bezeichnet, d. h. jedes Element in G' lässt sich durch g und der Verknüpfung $\circ$ erzeugen.

2.2.2 Körper

Ein Körper ist in der Mathematik als ein Tripel $(K, +, \cdot)$ aus der nicht leeren Menge K mit den beiden inneren Verknüpfungen Addition $(+)$ und Multiplikation $(\cdot)$ definiert, welches die folgenden Bedingungen erfüllt:

- Die Menge K bildet durch die additive Verknüpfung $+$ eine abelsche Gruppe mit 0 als neutralem Element
- Die Menge $K \setminus \{0\}$, d. h. K ohne das Element 0, bildet durch die multiplikative Verknüpfung $\cdot$ ebenfalls eine abelsche Gruppe mit 1 als neutralem Element
- Es gilt das Distributivgesetz: $c \cdot (a + b) = c \cdot a + c \cdot b \quad \forall a, b, c \in K$

Genau wie bei den Gruppen gibt es Körper mit endlich bzw. mit unendlich vielen Elementen. Ein bekanntes Beispiel für einen Körper mit unendlich vielen Elementen ist die Menge der rationalen Zahlen $\mathbb{Q}$ mit der üblichen Addition und Multiplikation als Verknüpfungen. Ein Körper mit endlich vielen Elementen wird auch als Restklassenkörper bezeichnet. Beispiele für Restklassenkörper sind alle Primkörper $\mathbb{Z}_p = \{0, 1, 2, 3, \cdots, p-1\}$ mit p als Primzahl und den Verknüpfungen Addition modulo p und Multiplikation modulo p.

Die Charakteristik eines Körpers K beschreibt die Ordnung des neutralen Elements der Multiplikation (1-Element) bezüglich der Addition, d. h. die kleinste natürliche Zahl n, so dass gilt: $\underbrace{1 + 1 + \cdots + 1}_{n \text{ mal}} = 0$, wobei 0 das neutrale Element der Addition ist.

Körper der Charakteristik 2 besitzen folglich nur die Elemente $\{0, 1\}$ und werden daher auch Binärkörper genannt (vgl. Tabelle 2.1). Die additive Verknüpfung in einem Binärkörper entspricht einer logischen `XOR`-Operation und die Multiplikation kann durch eine logische `AND`-Verknüpfung berechnet werden.

+	**0**	**1**
0	0	1
1	1	0

·	**0**	**1**
0	0	0
1	0	1

Tabelle 2.1: Additive und multiplikative Verknüpfung im Binärkörper

Allgemein kann gezeigt werden, dass zu jeder Primzahl p und jeder natürlichen Zahl n ein eindeutig bestimmter endlicher Körper mit p^n Elementen existiert. Diese Art von Körper spielt in der Kryptographie eine wichtige Rolle und wird generell mit $GF(p^n)$ gekennzeichnet. In Erinnerung an den französischen Mathematiker Évariste Galois steht die Abkürzung GF für Galois-Körper (engl. Galois Field).

Für kryptographische Anwendungen hat es sich in der Praxis als sinnvoll erwiesen, entweder Primkörper der Form $GF(p)$ mit einer großen Primzahl p oder Körpererweiterungen der Form $GF(2^n)$ mit einer großen natürlichen Zahl n zu verwenden. Primkörper eignen sich aufgrund ihrer einfachen Modulo-Arithmetik gut für Software-Implementierungen. Binärkörper hingegen sind wegen ihrer zweiwertigen Grundmenge besonders geeignet für Hardware-Implementierungen. Da der Fokus dieser Arbeit auf Hardware-Implementierungen liegt, werden in den folgenden Betrachtungen stets Binärkörper verwendet. Die vorgestellten Konzepte lassen sich aber ebenso auf Primkörper oder gänzlich andere endliche Körper übertragen.

2.2.2.1 Binärkörper

Wie im vorherigen Abschnitt beschrieben, ist für jeden Körper eine additive und multiplikative Verknüpfung definiert. Diese werden im Folgenden als Addition bzw. Multiplikation bezeichnet und bilden die Grundlage der arithmetischen Operationen in endlichen Körpern. Die Subtraktion wird mit Hilfe der Addition wie folgt definiert: $a - b = a + (-b) \ \forall a, b \in GF(p^n)$. Hierbei ist $-b$ das eindeutige Element des Körpers $GF(p^n)$ für das gilt: $b + (-b) = 0$. Ähnlich wird die Division von Körperelementen über die Multiplikation mit dem inversen Element b^{-1} so definiert, dass $a/b = a \cdot b^{-1} \ \forall a, b \in GF(p^n)$ mit $b \neq 0$, wobei b^{-1} das eindeutige Element des Körpers $GF(p^n)$ ist, für das gilt: $b \cdot b^{-1} = 1$.

Der endliche Körper $GF(2^n)$ kann als n-dimensionaler Vektorraum über dem Binärkörper $GF(2)$ dargestellt werden. Die zur Darstellung verwendete Basis bestimmt

dabei die Komplexität der arithmetischen Operationen auf dem erweiterten Binärkörper $GF(2^n)$. Eine Darstellung in Polynombasis hat sich für Körper kleiner Charakteristik im Bereich der Kryptographie als sehr effizient herausgestellt [HMV04]. Hierbei wird die Basis für $GF(2^n)$ mit Hilfe eines irreduziblen Polynoms $f(x)$ vom Grad n über $GF(2)$ bestimmt. Die Elemente von $GF(2^n)$ sind dann Polynome vom Grad höchstens $n-1$ über $GF(2)$:

$$GF(2^n) = \{a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_2x^2 + a_1x + a_0 : a_i \in \{0,1\}\} \tag{2.1}$$

Alle arithmetischen Operationen sind über die Polynomarithmetik, gefolgt von einer Modulo-Reduktion mit dem irreduziblen Polynom $f(x)$, definiert. Aufgrund der Körpereigenschaften ist in Binärkörpern die Subtraktion identisch mit der Addition, so dass für die Polynome $A, B \in GF(2^n)$ mit $A = \sum_{i=0}^{n-1} a_i x^i$ und $B = \sum_{i=0}^{n-1} b_i x^i$ gilt:

$$A + B = A - B = \sum_{i=0}^{n-1} (a_i + b_i)x^i \tag{2.2}$$

Da die Addition bzw. Subtraktion den Grad des Ergebnispolynoms nicht erhöhen kann, ist hier keine Modulo-Reduktion notwendig. Ferner folgt aus Gleichung 2.2, dass $+B = -B$ ist und somit allgemein gelten muss:

$$A + A = 0 \quad \forall A \in GF(2^n) \tag{2.3}$$

Bei der Multiplikation der zwei Polynome $A, B \in GF(2^n)$ hingegen kann das Ergebnispolynom einen Grad von maximal $2n-2$ erreichen, so dass hier eine Modulo-Reduktion mit Hilfe des irreduziblen Polynoms $f(x)$ notwendig ist:

$$A \cdot B = \sum_{i=0}^{2n-2} c_i x^i \mod f(x) \tag{2.4}$$

$$\text{wobei } c_i = \sum_{k=0}^{i} a_k \cdot b_{i-k} \quad \text{mit } a_i, b_i = 0 \text{ für alle } i \geq n$$

Die Quadrierung bildet einen Sonderfall der Multiplikation und lässt sich für Binärkörper durch Einsetzen von Gleichung 2.3 in Gleichung 2.4 wie folgt vereinfachen:

$$A \cdot A = A^2 = \sum_{i=0}^{n-1} (a_i) x^{2i} \mod f(x) \tag{2.5}$$

Die unter anderem zur Division benötigte Invertierung (vgl. Algorithmus 10) kann mit Hilfe der Multiplikation und der Quadrierung berechnet werden. Detaillierte Informationen zu den implementierten Algorithmen für die Binärkörper-Arithmetik werden in Kapitel 3.3 vorgestellt.

2.2.3 Elliptische Kurven in der Kryptographie

In der Kryptographie werden elliptische Kurven über endlichen Körpern verwendet, welche auf der so genannten Weierstrass-Gleichung basieren, die in affiner Darstellung wie folgt definiert ist:

$$F(x, y) : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6 \tag{2.6}$$

Die Variablen x, y sowie die Parameter a_1, a_2, a_3, a_4, a_6 müssen dabei Elemente eines gegebenen Körpers K sein. Eine weitere Voraussetzung ist, dass die elliptische Kurve keine Singularität aufweist, was gleichbedeutend mit der Nebenbedingung $\Delta(F) \neq 0$ ist, d. h. die Diskriminante ungleich von Null sein muss. Die elliptische Kurve E besteht dann aus der Menge aller Punkte, welche die Gleichung 2.6 erfüllen, und einem Punkt $\mathcal{O}$ im Unendlichen:

$$E(K) = \{(x, y) \in K \times K \mid y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6\} \cup \{\mathcal{O}\} \tag{2.7}$$

Die allgemeine Weierstrass-Gleichung 2.6 kann für Körper der Charakteristik 2 (Binärkörper) in die folgende, vereinfachte Form transformiert werden:

$$F(x, y) : y^2 + xy = x^3 + ax^2 + b \quad \text{mit } a, b \in GF(2^n), \Delta(F) = b \neq 0 \tag{2.8}$$

Äquivalent zu Gleichung 2.7 bildet nun die Menge aller Punkte, die Gleichung 2.8 erfüllen, zusammen mit dem Punkt $\mathcal{O}$ im Unendlichen die elliptische Kurve E über dem Binärkörper mit Erweiterungsgrad n:

$$E(GF(2^n)) = \{(x, y) \in GF(2^n) \times GF(2^n) \mid y^2 + xy = x^3 + ax^2 + b\} \cup \{\mathcal{O}\} \quad (2.9)$$

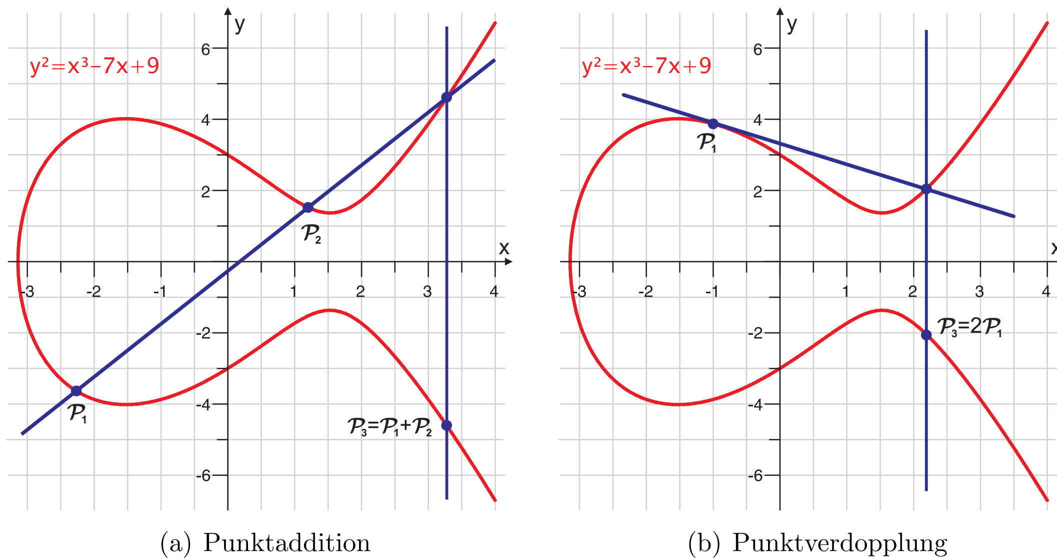
Die Anzahl der Punkte auf der elliptischen Kurve E über dem Körper $GF(2^n)$ wird als Ordnung der elliptischen Kurve $\#E$ bezeichnet. Diese kann nach dem Satz von Hasse [Sil09] durch $2^n + 1 - 2\sqrt{2^n} \leq \#E \leq 2^n + 1 + 2\sqrt{2^n}$ sehr leicht abgeschätzt werden. Die Berechnung der exakten Anzahl ist mit Hilfe verschiedener Methoden, z. B. dem Schoof-Algorithmus, in asymptotischer Laufzeit möglich [HMV04].

Die additive Gruppenstruktur (vgl. Abschnitt 2.2.1) für elliptische Kurven über Binärkörpern $E(GF(2^n))$ ist wie folgt definiert:

- Neutraler Punkt $\mathcal{O}$: $\mathcal{P} + \mathcal{O} = \mathcal{P} \qquad \forall\, \mathcal{P} = (x, y) \in E$
- Inverser Punkt $-\mathcal{P}$: $\mathcal{P} + (-\mathcal{P}) = \mathcal{O} \qquad \forall\, \mathcal{P} = (x, y) \in E \mid -\mathcal{P} = (x, x + y)$
- Verknüpfung $+$: $\mathcal{P}_1 + \mathcal{P}_2 = \mathcal{P}_3 \qquad \mathcal{P}_i = (x_i, y_i) \in E \mid i = \{1, 2, 3\}$

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \\ \lambda &= \begin{cases} \dfrac{y_1 + y_2}{x_1 + x_2} & \text{falls } \mathcal{P}_1 \neq \mathcal{P}_2 \\ x_1 + \dfrac{y_1}{x_1} & \text{falls } \mathcal{P}_1 = \mathcal{P}_2 \end{cases} \end{aligned} \quad (2.10)$$

Diese arithmetische Operation wird als Punktaddition bezeichnet, falls zwei verschiedene Punkte ($\mathcal{P}_1 \neq \mathcal{P}_2$) auf der elliptischen Kurve miteinander verknüpft werden bzw. Punktverdopplung, falls $\mathcal{P}_1$ gleich $\mathcal{P}_2$, also $\mathcal{P}_3 = \mathcal{P}_1 + \mathcal{P}_1 = 2\mathcal{P}_1$, ist. Für elliptische Kurven über den reellen Zahlen $E(\mathbb{R})$ lassen sich die Punktaddition und die Punktverdopplung geometrisch mittels der Sekanten- bzw. Tangenten-Methode (siehe Abbildung 2.3) darstellen. Hierbei legt man bei der Punktaddition eine Sekante durch die Punkte $\mathcal{P}_1, \mathcal{P}_2$ bzw. bei der Punktverdopplung eine Tangente an den Punkt $\mathcal{P}_1$. Diese Gerade schneidet die elliptische Kurve in einem weiteren



(a) Punktaddition (b) Punktverdopplung

Abbildung 2.3: Additive Verknüpfung von Punkten auf einer elliptischen Kurve

Punkt, welcher an der Abszisse gespiegelt das Ergebnis der additiven Verknüpfung ergibt.

Basierend auf der additiven Verknüpfung wird nun die Punktmultiplikation $k \cdot \mathcal{P}$ als wiederholte Punktaddition des Punktes $\mathcal{P}$ mit sich selbst definiert:

$$k \cdot \mathcal{P} = \underbrace{\mathcal{P} + \mathcal{P} + \ldots + \mathcal{P}}_{k \text{ mal}} \quad \forall \mathcal{P} \in E(GF(2^n)),\ k \in \mathbb{N} \tag{2.11}$$

Die Punktmultiplikation $k\mathcal{P}$ wird oft auch Skalarmultiplikation genannt, da der Punkt $\mathcal{P}$ der elliptischen Kurve mit dem skalaren Wert k multipliziert wird. Das Ergebnis der Multiplikation entspricht wiederum einem Punkt auf der elliptischen Kurve E über dem Körper $GF(2^n)$. Diese Operation bildet die Basis für die Kryptographie mit elliptischen Kurven (vgl. Abschnitt 2.3).

2.3 Sicherheit kryptographischer Verfahren

Die Sicherheit von kryptographischen Verfahren, welche auf elliptischen Kurven basieren, beruht auf dem so genannten Diskreten-Logarithmus-Problem (ECDLP). Dieses besagt, dass die Skalarmultiplikation $\mathcal{Q} = k\mathcal{P}$ für einen Punkt $\mathcal{P}$ auf der

elliptischen Kurve $E(GF(2^n))$ und einer natürlichen Zahl k nach Gleichung 2.11 sehr einfach zu berechnen ist, es hingegen aber sehr schwierig ist, aus zwei vorgegebenen Punkten $\mathcal{Q}, \mathcal{P} \in E(GF(2^n))$ wieder die Zahl k zu ermitteln. Im einfachsten Fall benötigt man zur Berechnung der Skalarmultiplikation nur $\log(k)$ Punktadditionen bzw. Punktverdopplungen. Zur Berechnung des diskreten Logarithmus k vom Punkt $\mathcal{Q}$ zur Basis $\mathcal{P}$ könnte man durch naives Ausprobieren (engl. brute force) sämtliche Skalarmultiplikationen $i\mathcal{P}$ für $i \in [0, r]$ berechnen und mit $\mathcal{Q}$ vergleichen. Diese Brute-Force-Methode ist auch als Enummeration bekannt und benötigt im Mittel $r/2$ Punktadditionen, wobei r der Ordnung des Punktes $\mathcal{P}$ entspricht [HMV04]. Es existieren verschiedene Verfahren, welche schneller als die Enummerations-Methode den diskreten Logarithmus berechnen können, wie z. B. die Pollard-ρ-Methode oder der Babystep-Giantstep-Algorithmus [Koç09]. Es ist jedoch bis heute kein Verfahren bekannt, welches das Diskrete-Logarithmus-Problem auf nicht-singulären elliptischen Kurven mit subexponentiellem Aufwand lösen kann [MOV96][Yan13].

Dieses ist ein großer Vorteil der Kryptographie mit elliptischen Kurven gegenüber den klassischen asymmetrischen Verfahren, wie dem RSA-Verfahren. RSA basiert auf dem Faktorisierungsproblem großer Zahlen. Demnach ist es für zwei große Primzahlen p und q sehr leicht das Produkt $m = pq$ zu berechnen. Allerdings ist es schwierig, das Produkt m wieder in seine Primfaktoren p und q zu zerlegen. Im Gegensatz zum Diskreten-Logarithmus-Problem auf elliptischen Kurven existieren für das Faktorisierungsproblem verschiedene Algorithmen, welche in subexponentieller Laufzeit die Primfaktorzerlegung berechnen. Bekannte Methoden sind hierfür das Quadratische-Sieb bzw. Zahlkörpersieb [Len00].

Symmetrische Verfahren **NIST Standard**		***Asymmetrische Verfahren***		***Lebenszeit*** **[Jahr]**
		RSA	**ECC**	
Skipjack	80 Bit	1024 Bit	163 Bit	≤ 2010
Triple-DES	112 Bit	2048 Bit	233 Bit	≤ 2030
AES (small)	128 Bit	3072 Bit	283 Bit	
AES (medium)	192 Bit	8192 Bit	409 Bit	> 2030
AES (large)	256 Bit	15360 Bit	571 Bit	

Tabelle 2.2: Lebenszeit und Schlüssellänge äquivalenter Sicherheitsniveaus [NIST07]

Aus der unterschiedlichen Komplexität zum Lösen des Faktorisierungsproblems und des Diskreten-Logarithmus-Problems resultieren verschiedene Schlüssellängen für das gleiche Sicherheitsniveau. Wie Tabelle 2.2 zeigt, hat zum Beispiel das RSA-Verfahren mit einem 1024-Bit-langen Schlüssel das gleiche Sicherheitsniveau wie Verfahren, die auf elliptischen Kurven basieren und einen Schlüssel mit 163 Bit Länge verwenden. Sicherheitsniveau bedeutet in diesem Zusammenhang, dass es mit vertretbaren Mitteln für einen gewissen Zeitraum nicht möglich ist, das kryptographische Verfahren zu brechen und somit die geschützten Daten zu entschlüsseln. Vertretbare Mittel meint hierbei die Rechenleistung C, welche ein potentieller Angreifer aufbringen kann. Je nach Budget kann diese Rechenleistung von einem modernen Computer über Cluster- bzw. Grid-Rechnerverbände bis hin zu anwendungsspezifischen Hardware-Lösungen sehr stark variieren [TOSK12]. Der Zeitraum, in dem sensible Daten gegenüber Angreifern sicher geschützt sein sollen, wird auch als Lebenszeit T_L bezeichnet. Soll die Sicherheit für eine lange Lebenszeit garantiert werden, muss das Gesetzt von Moore [Moo65] beachtet werden, da sich die Rechenleistung C etwa alle 18 Monate verdoppelt.

Zum Verschlüsseln großer Datenmengen oder kontinuierlicher Datenströme werden meist symmetrische Verfahren, wie zum Beispiel DES oder AES, eingesetzt. Bis heute sind zum Brechen symmetrischer Verfahren keine Algorithmen bekannt, die

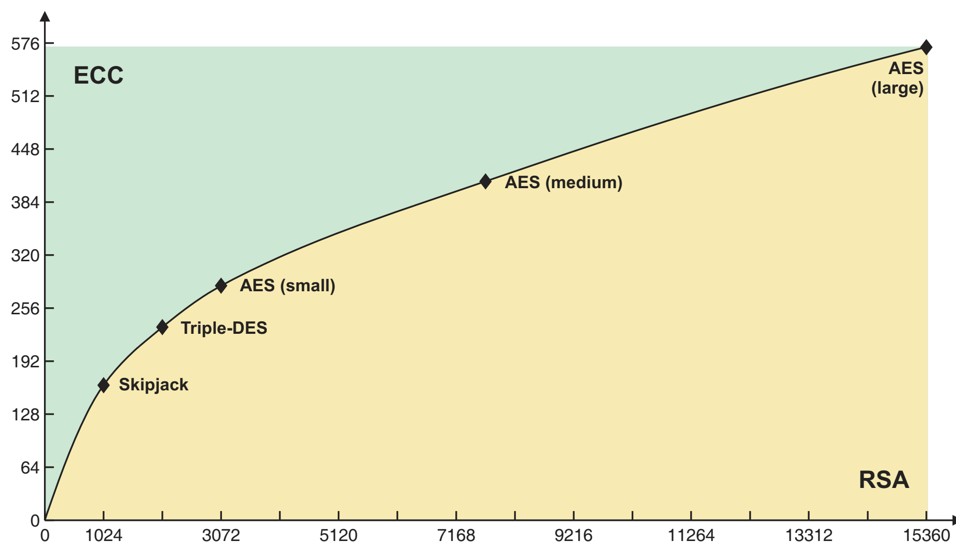


Abbildung 2.4: Schlüssellängen [Bit] äquivalenter Sicherheitsniveaus (vgl. Tabelle 2.2)

eine wesentlich bessere Komplexität als Brute-Force-Angriffe aufweisen [LV01]. Die Sicherheit für ein symmetrisches Verfahren mit k-Bit-Schlüssellänge ist daher gewährleistet wenn gilt: $2^k/C > T_L$, d. h. die Rechenleistung ist nicht ausreichend alle möglichen Schlüssel in der Lebenszeit auszuprobieren. Um einen gemeinsamen, geheimen Schlüssel für symmetrische Verfahren auszutauschen, werden in der Regel asymmetrische Verfahren verwendet. Die gesamte Verschlüsselung ist aber immer nur so sicher wie das schwächste Glied der Kette. Daher müssen die asymmetrischen Verfahren zum Schlüsselaustausch mindestens genauso sicher sein wie die symmetrischen Verfahren zur Verschlüsselung (vgl. Tabelle 2.2).

In Abbildung 2.4 sind für verschiedene symmetrische Verfahren die äquivalenten Schlüssellängen der asymmetrischen Verfahren auf Basis von RSA bzw. ECC dargestellt. Es ist deutlich zu erkennen, dass die Kryptographie mit elliptischen Kurven deutlich kürzere Schlüssellängen benötigt, um das gleiche Sicherheitsniveau wie RSA-basierte Verfahren zu erlangen. Beispielsweise wird zum Austausch eines Sitzungsschlüssels für eine symmetrische AES-Verschlüsselung mit einem 128-Bit-Schlüssel (AES small) beim RSA-Verfahren eine asymmetrische Schlüssellänge von mindestens 3072 Bit gefordert, während die Elliptische-Kurven-Kryptographie mit einer Schlüssellänge von 256 Bit das gleiche Sicherheitsniveau gewährleistet.

2.4 Stand der Technik

Das Kryptosystem $S(n, f(t), a, b, G, r, h)$, welches auf elliptischen Kurven über endlichen Körpern $GF(2^n)$ basiert, ist durch seine Parameter wie folgt eindeutig definiert:

n	Erweiterungsgrad des Binärkörpers $GF(2^n)$
$f(t)$	Irreduzierbares Polynom vom Grad n
a, b	Parameter der elliptischen Kurve E nach Gleichung 2.8
G	Basispunkt auf der elliptischen Kurve: $G = (x_G, y_G) \in E(GF(2^n))$
r	Prime Ordnung des Basispunktes G
h	Ganzzahliger Kofaktor: $h = \#E(GF(2^n))/r$

Die Wahl der Parameter (siehe Anhang A) beeinflussen dabei stark die kryptographisch relevanten Eigenschaften, wie beispielsweise die Sicherheit des Kryptosystems. Aus diesem Grund haben verschiedene Organisationen die Parameter für Kryptosysteme, die auf festgelegten Körpern basieren, standardisiert. Diese Standards erhöhen die Sicherheit und gewähren Interoperabilität zwischen unterschiedlichen Implementierungen. Tabelle 2.3 gibt einen Überblick der wichtigsten Standards, die von folgenden Organisationen herausgegeben werden:

ANSI	–	American National Standards Institute
IEEE	–	Institute of Electrical and Electronics Engineers
NIST/FIPS	–	National Institute of Standards and Technology/
		Federal Information Processing Standard
ISO/IEC	–	International Organization for Standardization/
	–	International Electrotechnical Commission
SECG	–	Standards for Efficient Cryptography Group

Standard	**Titel**	**Stand**
ANSI X9.62	The Elliptic Curve Digital Signature Algorithm	2005
ANSI X9.63	Key Agreement and Key Transport	2011
FIPS 186-3	Digital Signature Standard	2009
IEEE P1363	Standard Specifications for Public-Key Cryptography	2011
ISO/IEC 11770	Key Management	2010
ISO/IEC 14888	Digital Signatures with Appendix	2008
ISO/IEC 15946	Cryptographic Techniques based on Elliptic Curves	2009
ISO/IEC 18033	Encryption Algorithms	2010
SEC 1	Elliptic Curve Cryptography	2009
SEC 2	Recommended Elliptic Curve Domain Parameters	2010

Tabelle 2.3: Überblick an Standards zur Kryptographie mit elliptischen Kurven

In Deutschland existieren ebenfalls Empfehlungen des Bundesamtes für Sicherheit in der Informationstechnik (BSI) zur Umsetzung der in Tabelle 2.3 genannten Standards [BSI12b]. Neben der Festlegung der Parameter eines Kryptosystemes werden in den oben genannten Standards außerdem Protokolle für verschiedene Anwendungen spezifiziert. Die bekanntesten, standardisierten Protokolle, die auf elliptischen Kurven basieren, sind in Tabelle 2.4 dargestellt.

Protokoll	**Bedeutung**	**Kategorie**
ECDSA	Elliptic Curve Digital Signature Algorithm	Digitale Signatur
ECPVS	Elliptic Curve Pintsov-Vanstone Signatures	
ECNR	Elliptic Curve Nyberg-Rueppel	
ECDH	Elliptic Curve Diffie-Hellman	Schlüsselaustausch
ECMQV	Elliptic Curve Menezes-Qu-Vanstone	
ECIES	Elliptic Curve Integrated Encryption Scheme	Verschlüsselung

Tabelle 2.4: Überblick an Protokollen zur Kryptographie mit elliptischen Kurven

Die genannten Standards bzw. Protokolle finden bereits vereinzelt Anwendung in der Praxis. So gibt es viele Software-Implementierungen, die neben den klassischen asymmetrischen RSA-Verfahren auch Verfahren auf Basis elliptischer Kurven optional unterstützen. Dazu zählen sowohl frei verfügbare Software-Bibliotheken, wie z. B. OpenSSL oder GnuPG, sowie kommerziell erhältliche Produkte, beispielsweise Entwicklungen der Firmen SECUDE oder cryptovision GmbH. Einige Unternehmen, z. B. Certicom oder Elliptic Technologies, bieten neben Software-Lösungen

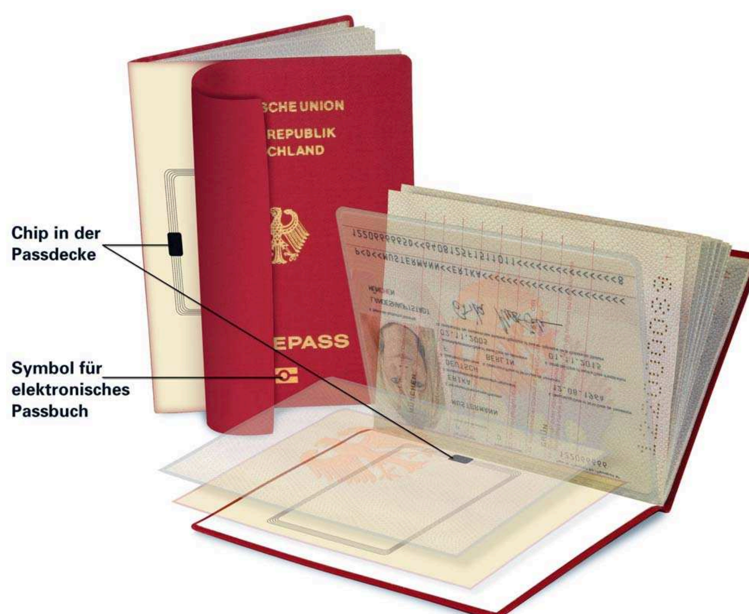


Abbildung 2.5: Elektronischer Reisepass mit integriertem Mikroprozessor für Kryptographie mit elliptischen Kurven [BSI11]

auch Hardware-Implementierungen in Form von IP[2]-Cores an, welche unter anderem in Mikroprozessorsystemen bekannter Hersteller wie ARM, Freescale Semiconductor oder Texas Instruments integriert werden. Dedizierte Hardware-Bausteine in Form von ASICs[3] werden z. B. im Bereich der Trusted Platform Modules (TPM) sowie für Chipkarten und RFID-Tags ebenfalls angeboten. Hier sind hauptsächlich Konzerne wie NXP Semiconductors, Atmel Corporation oder Infineon Technologies beteiligt. Der deutsche Reisepass beispielsweise enthält ein RFID-Tag mit dem Krypto-Prozessor SLE66CLX641P von Infineon Technologies, der unter anderem die Protokolle ECDH und ECDSA nach dem ANSI X9.62 Standard mit einer Schlüssellänge von 224 Bit verarbeitet.

2.5 Stand der Forschung

Weltweit existieren zahlreiche Forschergruppen, die sich mit Fragestellungen aus dem Bereich der Kryptographie beschäftigen. Viele Gruppen beschäftigen sich mit der Optimierung von Algorithmen auf Software-Ebene. Führende Arbeiten auf diesem Gebiet stammen aus der Arbeitsgruppe von Çetin Kaya Koç an der Oregon State University [RHK03][TK03][EYK06][Koç09][SK10][TOSK12]. Daneben gibt es einige Forscher, die sich mit der Umsetzung von kryptographischen Algorithmen in effiziente Hardware-Implementierungen beschäftigen. Hier sind in erster Linie die Arbeiten der Forschergruppe von Christof Paar an der Ruhr-Universität Bochum zu nennen [BP01][WP06][GGK+06][FBB+10][PP10][DP12]. Am Institut für angewandte Informationsverarbeitung und Kommunikation (IAIK) der Technischen Universität Graz sind unter der Leitung von Karl-Christian Posch interessante Arbeiten auf dem Spezialgebiet der Instruktionssatzerweiterung für kryptographische Anwendungen entstanden [GE04][GPT04][GIP+06][PPH11][MHH12]. Des Weiteren beschäftigen sich diverse Forschungsabteilungen aus der Industrie mit elliptischen Kurven in der Kryptographie. Vor allem die Sun Microsystems Laboratories waren hierbei sehr engagiert [GPW+04][GWZ+05], aber auch andere Firmen, wie z. B. Intel [GGK08][GK10] oder Microsoft [Mon05][LS08][PSNB11][BKK+12], betreiben Forschung auf diesem Gebiet.

[2] Intellectual Property
[3] Application-Specific Integrated Circuit

Im Gegensatz zu den genannten Forschungsarbeiten besteht der Mehrwert dieser Arbeit darin, dass eine Metrik zur Bestimmung der Ressourceneffizienz (vgl. Kapitel 4.3) entwickelt wird, die es erlaubt, die implementierten Hardware-Software-Kombinationen anwendungsspezifisch zu gewichten und so besser untereinander vergleichen zu können. Ferner wird eine hierarchische, skalierbare Systemarchitektur vorgestellt, die es ermöglicht, Hardware-Software-Kombinationen für verschiedene Anwendungsanforderungen schnell zu evaluieren. Die dazu gehörige Entwicklungsumgebung gestattet darüber hinaus eine komfortable Entwurfsraumexploration.

2.6 Zusammenfassung

Die moderne Kryptographie ist unterteilt in symmetrische und asymmetrische Verfahren. Während symmetrischen Verfahren wie DES oder AES zur reinen Daten Ver- bzw. Entschlüsselung eingesetzt werden, ermöglicht die asymmetrische Kryptographie weitere Verfahren, wie beispielsweise den sicheren Schlüsselaustausch oder digitale Signaturen. Gegenüber dem RSA-Algorithmus, der sich momentan als Standard in der asymmetrischen Kryptographie etabliert hat, existieren neue Ansätze, die auf elliptischen Kurven basieren. Die Elliptische-Kurven-Kryptographie bietet bei deutlich kürzerer Schlüssellänge das gleiche Sicherheitsniveau wie RSA-basierte Verfahren (vgl. Abbildung 2.4) und eignet sich daher besonders für eingebettete Systeme.

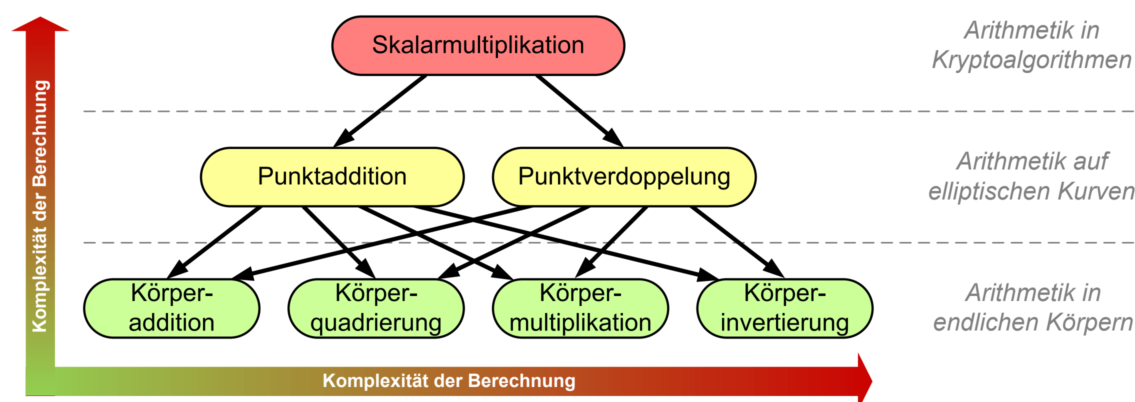


Abbildung 2.6: Arithmetik der Elliptischen-Kurven-Kryptographie über Binärkörpern

Die Sicherheit der Kryptographie mit elliptischen Kurven basiert auf dem Diskreten-Logarithmus-Problem. Die mathematische Grundlage hierzu bilden arithmetische Operationen in endlichen Körpern. Wie in Abbildung 2.6 verdeutlicht ist, werden auf der untersten Hierarchieebene Addition, Quadrierung, Multiplikation und Invertierung von Elementen eines endlichen Körpers benötigt. Im Fall von Binärkörpern der Form $GF(2^n)$ ist die Addition identisch mit der Subtraktion (vgl. Gleichung 2.2) und kann durch eine einfache `XOR`-Verknüpfung realisiert werden (vgl. Tabelle 2.1). Die Komplexität für die Berechnung der Körper-Addition ist daher gering. Die Invertierung hingegen ist zwar sehr rechenaufwändig, wird aber unter bestimmten Voraussetzungen (siehe Abschnitt 3.2) im Vergleich zu den anderen arithmetischen Operation nur selten benötigt. Die Quadrierung, als Spezialfall der Multiplikation, kann im Binärkörper $GF(2^n)$ ebenfalls relativ einfach berechnet werden (vgl. Gleichung 2.5). Dagegen stellt die Multiplikation wiederum eine rechenintensive Operation dar, welche recht häufig ausgeführt werden muss. Eine effiziente Methode zur Berechnung der Körper-Multiplikation (vgl. Abschnitt 3.3.2) bestimmt daher wesentlich die Leistungsfähigkeit der darauf aufbauenden Operationen. Diese sind als Punktaddition und Punktverdopplung bekannt (vgl. Abbildung 2.3) und bilden auf der mittleren Hierarchieebene die arithmetischen Operationen auf Punkten der zugrunde liegenden elliptischen Kurve. Aufbauend auf der Punkt-Arithmetik befindet sich auf der obersten Hierarchieebene die Skalarmultiplikation. Diese ist die Grundlage für viele, auf elliptischen Kurven basierende, kryptographische Algorithmen.

3 Algorithmen und Methoden

Wie bereits im vorherigen Kapitel erwähnt wurde, stellt die Skalarmultiplikation eine fundamentale Operation für viele kryptographische Algorithmen dar. In diesem Kapitel werden verschiedene Methoden und Algorithmen vorgestellt, die eine effiziente Berechnung der Skalarmultiplikation ermöglichen. Die im Folgenden erläuterten, algorithmischen Betrachtungen folgen strukturell der in Abbildung 2.6 dargestellt Hierarchie von oben nach unten, d. h. es wird zunächst ein Algorithmus beschrieben, welcher die Skalarmultiplikation auf Basis der Punktaddition und Punktverdopplung möglichst effizient umsetzt. Anschließend wird eine Koordinatentransformation vorgestellt, welche es wiederum ermöglicht, die Punktaddition und Punktverdopplung mit möglichst wenig rechenintensiven Operationen im Grundkörper durchzuführen. Abschließend werden Algorithmen für alle benötigten arithmetischen Operationen im Grundkörper diskutiert.

3.1 Skalarmultiplikation auf elliptischen Kurven

Es existieren viele unterschiedliche Verfahren zur Berechnung der Skalarmultiplikation auf elliptischen Kurven [Mül98]. In dieser Arbeit wurde verstärkt der Montgomery-Leiter-Algorithmus [Mon87][LD99] betrachtet, da dieser einen guten Kompromiss hinsichtlich Performanz, Sicherheit und benötigten Ressourcen darstellt [HMV04]. So benötigt der Montgomery-Leiter-Algorithmus im Gegensatz zu anderen Verfahren beispielsweise keinen zusätzlichen Speicher, um auf vorab berechnete Teilergebnisse zurückgreifen zu können. Dieses ist besonders für eingebettete Systeme wichtig, da Speicherplatz dort meist eine knappe und teure Ressource darstellt. Weiterhin ist der Montgomery-Leiter-Algorithmus von Natur aus sicher gegenüber Angriffen,

Algorithmus 1: *Montgomery-Leiter zur Berechnung der Skalarmultiplikation auf elliptischen Kurven*

Eingabe : Ein Punkt $\mathcal{P}$ auf der elliptischen Kurve E sowie der Skalar k als binäre Repräsentation $(k_{i-1}k_{i-2}\dots k_1k_0)_2$
Ausgabe : $k \cdot \mathcal{P}$

```
1  Q1 ← P, Q2 ← 2P, u ← (i − 1)
2  while k_u = 0 do
3      u ← u − 1
4  end while
5  for v from u − 1 downto 0 do
6      if k_v = 1 then
7          Q1 ← Q1 + Q2, Q2 ← 2Q2
8      else
9          Q2 ← Q1 + Q2, Q1 ← 2Q1
10     end if
11 end for
12 return Q1
```

die auf Analyse der Leistungsaufnahme (SPA[1]) oder der Ausführungszeit (TA[2]) beruhen. Schlussendlich lässt sich der Montgomery-Leiter-Algorithmus gut parallelisieren, was einer effizienten Hardware-Umsetzung sehr entgegenkommt (vgl. Kapitel 6.3).

3.1.1 Montgomery-Leiter

Algorithmus 1 stellt die Montgomery-Leiter dar. Als Eingabe wird ein Punkt $\mathcal{P}$ auf der elliptischen Kurve E sowie der skalare Wert k benötigt. Der Algorithmus berechnet iterativ die Skalarmultiplikation $k\mathcal{P}$ und weist die logarithmische Laufzeitkomplexität $O(\log_2 i)$ auf. Die Iteration in der `for`-Schleife beginnt beim höchstwertigsten Bit, das einer logischen Eins in der binären Repräsentation von k entspricht, und bricht beim niederwertigsten Bit (k_0) ab. Innerhalb jedes Iterationsschrittes wird sowohl eine Punktaddition als auch eine Punktverdopplung berechnet. Die Ergebnisse werden in Abhängigkeit des Binärkoeffizienten k_v in den Zwischenwerten

[1] Simple Power Analysis
[2] Timing Analysis

$\mathcal{Q}_1$ und $\mathcal{Q}_2$ gespeichert. Dadurch, dass in jeder Iteration, unabhängig vom aktuellen Binärkoeffizienten k_v, die gleichen Operationen (Punktaddition und Punktverdopplung) durchgeführt werden und lediglich die Zuweisung der Teilergebnisse vom Wert k_v abhängt, ist der Montgomery-Leiter-Algorithmus sicher gegenüber SPA- und TA-basierte Seitenkanalattacken.

Am Beispiel $k = 13$, bzw. $k = (\texttt{00001101})_2$ in binärer Schreibweise, soll Abbildung 3.1 die Montgomery-Leiter nach Algorithmus 1 veranschaulichen. Die `while`-Schleife in Algorithmus 1 sorgt zunächst dafür, dass alle führenden Nullen in der binären Repräsentation von k ignoriert werden. Die höchstwertigste, logische Eins befindet sich an Position k_3 und bildet den Startpunkt des binären Entscheidungsbaumes in Abbildung 3.1. Das linke Element eines jeden Knotens entspricht dem Wert $\mathcal{Q}_1$ aus Algorithmus 1, das rechte Element eines Knotens dem Wert $\mathcal{Q}_2$. Gemäß der `for`-Schleife wird eine Punktaddition und Punktverdopplung berechnet und in Abhängigkeit von $k_2 = 1$ die Zwischenwerte $\mathcal{Q}_1 = \mathcal{Q}_1 + \mathcal{Q}_2 = 3\mathcal{P}$ (Punktaddition) und $\mathcal{Q}_2 = 2\mathcal{Q}_2 = 4\mathcal{P}$ (Punktverdopplung) zugewiesen. Im nächsten Iterationsschritt gilt $k_1 = 0$, so dass $\mathcal{Q}_2 = \mathcal{Q}_1 + \mathcal{Q}_2 = 7\mathcal{P}$ und $\mathcal{Q}_1 = 2\mathcal{Q}_1 = 6\mathcal{P}$ berechnet wird. Schließlich werden in der letzten Iteration ($k_0 = 1$) die Werte $\mathcal{Q}_1 = \mathcal{Q}_1 + \mathcal{Q}_2 = 13\mathcal{P}$ und $\mathcal{Q}_2 = 2\mathcal{Q}_2 = 14\mathcal{P}$ ermittelt, so dass $\mathcal{Q}_1 = 13\mathcal{P}$ als Ergebnis dieser Beispielrechnung zurückgegeben wird.

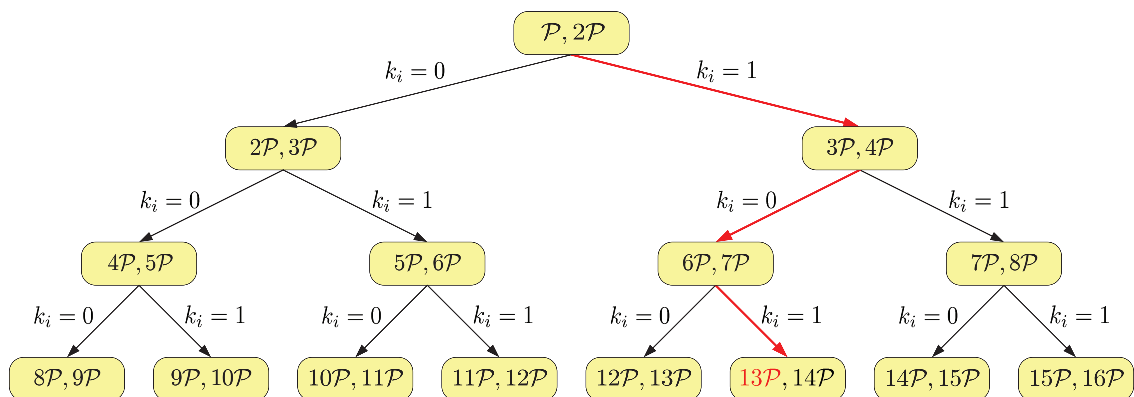


Abbildung 3.1: Beispiel einer Skalarmultiplikation auf Basis der Montgomery-Leiter

3.2 Punktaddition und Punktverdopplung

Wie die hierarchische Struktur in Abbildung 2.6 veranschaulicht, basiert die Skalarmultiplikation auf der Addition und Verdopplung von Punkten auf der zugrunde liegenden elliptischen Kurve. Wenn die Skalarmultiplikation nach Algorithmus 1 mit Hilfe der Montgomery-Leiter berechnet wird, bedeutet dieses, dass in jedem Iterationsschritt sowohl eine Punktaddition als auch eine Punktverdopplung durchgeführt werden muss. Aufgrund der logarithmischen Laufzeitkomplexität benötigt die Skalarmultiplikation $k\mathcal{P}$ im Binärkörper $GF(2^n)$ maximal n Iterationsschritte, nämlich genau dann, wenn der Binärkoeffizient $k_{n-1} = 1$ ist.

Bei der Berechnung der Punktaddition bzw. der Punktverdopplung spielt die Wahl des verwendeten Koordinatensystems eine entscheidende Rolle. In Kapitel 2.2.3 wurde bereits die Berechnung im affinen Koordinatensystem vorgestellt. Wie in Gleichung 2.10 zu erkennen ist, benötigt die Berechnung der Punktaddition bzw. der Punktverdopplung in affinen Koordinaten jeweils eine Invertierung und zwei Multiplikationen im entsprechenden Grundkörper. Additionen und Quadrierungen im Grundkörper können bei dieser Abschätzung vernachlässigt werden, da diese im Vergleich zur Invertierung und Multiplikation um ein Vielfaches einfacher zu berechnen sind [HMV04]. Folglich kann im Binärkörper $GF(2^n)$ eine Skalarmultiplikation in affinen Koordinaten nach dem Montgomery-Leiter-Algorithmus bis zu n Invertierungen und $2n$ Multiplikationen im Körper $GF(2^n)$ erfordern. Weiterhin ist allgemein anerkannt, dass die Berechnung der Invertierung um mindestens ein Zehnfaches rechenintensiver als eine Multiplikation ist [HHM00]. Es wird daher versucht, die Skalarmultiplikation mit möglichst wenig Invertierungen durchzuführen. Dieses kann durch eine Transformation in ein projektives Koordinatensystem erreicht werden.

3.2.1 Koordinatentransformation

Alle Punkte (x, y) eines affinen Koordinatensystems können durch Substitution von $x = X/Z^i$ und $y = Y/Z^j$ mit geeigneten Werten i, j in ein projektives Koordinatensystem transformiert werden, so dass äquivalente Punkte durch ein Tripple (X, Y, Z) repräsentiert werden. Wählt man beispielsweise $i = j = 0$, so wird $Z = 1$ gesetzt und

alle Punkt in der affinen Repräsentation sind identisch mit der projektiven Repräsentation. Für die Punktaddition bzw. Punktverdopplung in Binärkörpern hat sich das Koordinatensystem nach López-Dahab mit $i = 1$ und $j = 2$ als besonders effizient herausgestellt [LD98]. Die Transformation der affinen Weierstrass-Gleichung 2.8 ergibt nach López-Dahab die folgende projektive Form:

$$F(X, Y, Z) : Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4 \quad \text{mit } a, b \in GF(2^n), b \neq 0 \tag{3.1}$$

Es gelten im projektiven Raum weiterhin die Eigenschaften der additiven Gruppenstruktur (vgl. Kapitel 2.2.1), d. h. der neutrale Punkt $\mathcal{O}$ besitzt die Koordinaten $(1, 0, 0)$ und der inverse Punkt $-\mathcal{P}$ die Koordinaten $(X, X + Y, Z)$. Die additive Verknüpfung der Punkte $\mathcal{P}_1 + \mathcal{P}_2 = \mathcal{P}_3$, $\mathcal{P}_i = (X_i, Y_i, Z_i) \in E \mid i = 1, 2, 3$ lässt sich im projektiven Raum (vgl. Gleichung 2.10) wie folgt berechnen:

$$\begin{array}{c} \mathcal{P}_1 = \mathcal{P}_2 \\ \text{(Punktverdopplung)} \end{array} \quad \begin{aligned} X_3 &= {X_1}^4 + b \cdot {Z_1}^4 \\ Y_3 &= b \cdot {Z_1}^4 \cdot Z_3 + X_3 \cdot (a \cdot Z_3 + {Y_1}^2 + b \cdot {Z_1}^4) \\ Z_3 &= {X_1}^2 \cdot {Z_1}^2 \end{aligned} \tag{3.2}$$

$$\begin{array}{c} \mathcal{P}_1 \neq \mathcal{P}_2 \\ \text{(Punktaddition)} \end{array} \quad \begin{aligned} X_3 &= E^2 + J + I \\ Y_3 &= J \cdot K + Z_3 \cdot L \\ Z_3 &= H^2 \end{aligned} \tag{3.3}$$

mit

$$\begin{array}{llll} A = Y_2 \cdot {Z_1}^2, & D = X_1 \cdot Z_2, & G = Z_1 \cdot Z_2, & J = E \cdot H \\ B = Y_1 \cdot {Z_2}^2, & E = A + B, & H = F \cdot G, & K = F^2 \cdot C \cdot G + X_3 \\ C = X_2 \cdot Z_1, & F = C + D, & I = F^2 \cdot (H + a \cdot G^2), & L = F^2 \cdot A + X_3 \end{array}$$

Wie aus den Gleichungen 3.2 und 3.3 ersichtlich wird, benötigt man keine rechenintensive Inversion zur Berechnung der Punktaddition bzw. Punktverdopplung im projektiven Raum. Unter der Annahme, dass im Allgemeinen der Kurvenparameter $a = \{0, 1\}$ gewählt wird [NIST94], kann eine Punktverdopplung mit 4 Multiplikationen im Grundkörper $GF(2^n)$ berechnet werden. Die Punktaddition benötigt entsprechend 13 Multiplikationen im Grundkörper $GF(2^n)$. Die Anzahl der zur Punktaddition benötigten Multiplikationen kann jedoch noch weiter reduziert

werden, indem affine und projektive Koordinaten zur Berechnung gemischt werden [AMRK02]. Liegt der Punkt $\mathcal{P}_1 = (X_1, Y_1, Z_1)$ in projektiven Koordinaten und der Punkt $\mathcal{P}_2 = (x_2, y_2)$ in affinen Koordinaten vor, so kann die Punktaddition $\mathcal{P}_1 + \mathcal{P}_2 = \mathcal{P}_3$, mit $\mathcal{P}_1 \neq \mathcal{P}_2$, $\mathcal{P}_3 = (X_3, Y_3, Z_3)$ wie folgt vereinfacht werden:

$$(X_1, Y_1, Z_1) + (x_2, y_2) = (F, (D + C^2) \cdot G + H, C^2) \tag{3.4}$$

mit

$$\begin{aligned} &A = y_2 \cdot {Z_1}^2 + Y_1, && C = Z_1 \cdot B, && E = B^2 \cdot (C + a \cdot {Z_1}^2), && G = F + x_2 \cdot C^2 \\ &B = x_2 \cdot Z_1 + X_1, && D = A \cdot C, && F = A^2 + D + E, && H = (x_2 + y_2) \cdot C^4 \end{aligned}$$

Auf diese Weise kann die benötigte Anzahl an Grundkörper-Multiplikationen von 13 (Gleichung 3.3) auf 8 (Gleichung 3.4) reduziert werden. In Tabelle 3.1 sind die Kosten einer Punktaddition und einer Punktverdopplung in den bekanntesten Koordinatensystemen zusammengefasst. Es sei nochmals daran erinnert, dass die Reduktion der Multiplikationen (M) im Grundkörper das primäre Ziel bei der Auswahl einer geeigneten Repräsentation darstellt, da die Addition (A) und Quadrierung (Q) vergleichsweise einfach, gegenüber der rechenintensiven Multiplikation, berechnet werden kann [HHM00]. Wie aus Tabelle 3.1 ersichtlich wird, bildet das projektive López-Dahab-Koordinatensystem, unter Berücksichtigung der gemischten Koordinaten für die Punktaddition, die beste Repräsentation zur Durchführung von arithmetischen Punkt-Operationen auf elliptischen Kurven über Binärkörpern.

Koordinaten	**Punktaddition**	**Punktverdopplung**
Affine* (Standard)	12 M + 1 Q + 8 A	12 M + 1 Q + 8 A
Projektive (Standard)	14 M + 1 Q + 7 A	7 M + 4 Q + 5 A
Projektive (Jacobi)	14 M + 5 Q + 7 A	5 M + 5 Q + 4 A
Projektive (López-Dahab)	13 M + 6 Q + 7 A	4 M + 5 Q + 4 A
Gemischte (Standard)	11 M + 2 Q + 8 A	7 M + 4 Q + 5 A
Gemischte (Jacobi)	10 M + 3 Q + 7 A	5 M + 5 Q + 4 A
Gemischte (López-Dahab)	8 M + 5 Q + 9 A	4 M + 5 Q + 4 A

* 1 Invertierung = 10 Multiplikationen (M) in $GF(2^n)$

Tabelle 3.1: Kosten der Punkt-Operationen in verschiedenen Koordinatensystemen

3.2.2 Montgomery-Leiter nach López-Dahab

Mit Hilfe der projektiven López-Dahab-Koordinaten kann das Montgomery-Leiter-Verfahren (vgl. Algorithmus 1) so angepasst werden, dass keine Invertierung für die Punktaddition bzw. Punktverdopplung mehr benötigt wird. Des Weiteren konnte von López-Dahab gezeigt werden, dass es zur Berechnung der Skalarmultiplikation $k\mathcal{P}$ ausreichend ist, nur die x-Koordinate des Punktes $\mathcal{P}$ innerhalb der Punktaddition bzw. Punktverdopplung zu betrachten [LD99]. Die entsprechende y-Koordinate kann aus den Ergebnissen $k\mathcal{P}$, $(k+1)\mathcal{P}$ und dem Ausgangspunkt $\mathcal{P}$ rekonstruiert werden. Für die zur Berechnung der Skalarmultiplikation $k\mathcal{P}$, $\mathcal{P} = (x, y)$ notwendige Punkt-Operation $\mathcal{P}_1 + \mathcal{P}_2 = \mathcal{P}_3$, $\mathcal{P}_i = (X_i, Y_i, Z_i)$ ergeben sich folgende Formeln:

$$\begin{array}{cl} \mathcal{P}_1 = \mathcal{P}_2 & X_3 = {X_1}^4 + b \cdot {Z_1}^4 \\ \text{(Punktverdopplung)} & Z_3 = {X_1}^2 \cdot {Z_1}^2 \end{array} \tag{3.5}$$

$$\begin{array}{cl} \mathcal{P}_1 \neq \mathcal{P}_2 & X_3 = x \cdot Z_3 + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1) \\ \text{(Punktaddition)} & Z_3 = (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2 \end{array} \tag{3.6}$$

Wie aus den Gleichungen 3.5 und 3.6 ersichtlich wird, benötigt man nur noch 2 Multiplikationen, 4 Quadrierungen und eine Addition im Grundkörper zur Punktverdopplung bzw. 4 Multiplikationen, eine Quadrierung und 2 Additionen zur Punktaddition. Durch die Berechnung der Punktaddition bzw. Punktverdopplung nach López-Dahab kann der Montgomery-Leiter-Algorithmus weiter optimiert werden. In Algorithmus 2 sind die entsprechenden Anpassungen dargestellt. In Zeile 2 wird zunächst die affine x-Koordinate des Ausgangspunktes $\mathcal{P} = (x, y)$ in die projektive Form (X, Z) transformiert. Äquivalent zu Zeile 1 in Algorithmus 1 entspricht nach dieser Initialisierung (X_1, Z_1) der x-Koordinate des Punktes $\mathcal{P}$ und (X_2, Z_2) der x-Koordinate des Punktes $2\mathcal{P}$. Innerhalb der `for`-Schleife werden die Punktverdopplung gemäß Gleichung 3.5 und die Punktaddition gemäß Gleichung 3.6 berechnet. Im Vergleich zur affinen Variante der Montgomery-Leiter werden innerhalb der `for`-Schleife keinerlei Invertierungen benötigt. Eine Invertierung wird erst bei der Koordinaten-Rücktransformation und Rekonstruktion der y-Koordinate benötigt. Diese Berechnung ist in den Zeilen 15 und 16 dargestellt. Aus der projekti-

Algorithmus 2: *Montgomery-Leiter nach López-Dahab zur Berechnung der Skalarmultiplikation auf elliptischen Kurven*

Eingabe : Ein Punkt $\mathcal{P} = (x, y)$ auf der elliptischen Kurve E mit Kurvenparameter b sowie der Skalar k in Binärform $(k_{i-1}k_{i-2} \dots k_1k_0)_2$
Ausgabe : $k \cdot \mathcal{P} = (x_k, y_k)$

```
1  if k = 0 or x = 0 then return (0,0)
2  X1 ← x, Z1 ← 1, X2 ← x^4 + b, Z2 ← x^2, u ← (i − 1)
3  while k_u = 0 do
4      u ← u − 1
5  end while
6  for v from u − 1 downto 0 do
7      if k_v = 1 then
8          (X1, Z1) ← (X1, Z1) + (X2, Z2), (X2, Z2) ← (X2, Z2) + (X2, Z2)
9      else
10         (X2, Z2) ← (X2, Z2) + (X1, Z1), (X1, Z1) ← (X1, Z1) + (X1, Z1)
11     end if
12 end for
13 if Z1 = 0 then return (0,0)
14 if Z2 = 0 then return (x, x + y)
15 x_k ← (x · X1 · Z2) · (x · Z1 · Z2)^−1
16 y_k ← y+(x+x_k)·([X1+(x·Z1)]·[X2+(x·Z2)]+Z1·Z2·(x^2+y))·(x·Z1·Z2)^−1
17 return (x_k, y_k)
```

ven x-Koordinate der Punkte $k\mathcal{P} = (X_1, Z_1)$ und $(k+1)\mathcal{P} = (X_2, Z_2)$ sowie dem affinen Ausgangspunkt $\mathcal{P} = (x, y)$ wird das affine Ergebnis der Skalarmultiplikation $k\mathcal{P} = (x_k, y_k)$ berechnet. Hierzu sind eine Invertierung, 10 Multiplikationen (M), eine Quadrierung (Q) und 6 Additionen (A) im Grundkörper $GF(2^n)$ notwendig.

Vergleicht man die Kosten zur Berechnung der Skalarmultiplikation mit Hilfe der Montgomery-Leiter in affinen Koordinaten (Algorithmus 1) mit dem Verfahren von López-Dahab in projektiven Koordinaten (Algorithmus 2), so wird ersichtlich, dass sich der Mehraufwand hinsichtlich der Koordinatentransformation lohnt. Für eine Skalarmultiplikation $k\mathcal{P}$ im Binärkörper $GF(2^n)$ wird insgesamt die folgende Anzahl an Grundkörper-Operationen (Kosten einer Invertierung = 10 M) benötigt:

$$\text{Algorithmus 1:} \quad \lfloor \log_2 k \rfloor \times (24\,\text{M} + 2\,\text{Q} + 16\,\text{A}) + 12\,\text{M} + 1\,\text{Q} + 8\,\text{A} \tag{3.7}$$

$$\text{Algorithmus 2:} \quad \lfloor \log_2 k \rfloor \times (\ 6\,\text{M} + 5\,\text{Q} + \ 3\,\text{A}) + 20\,\text{M} + 3\,\text{Q} + 7\,\text{A} \tag{3.8}$$

3.3 Arithmetische Operationen im Binärkörper

Wie in den vorherigen Kapiteln bereits erwähnt wurde, bilden die Addition, die Multiplikation mit anschließender Modulo-Reduktion und die Invertierung die grundlegenden, arithmetischen Operationen in endlichen Körpern, welche zur Berechnung der Punktaddition bzw. Punktverdopplung (vgl. Algorithmus 2) notwendig sind. Zusätzlich kann die Quadrierung von Elementen als Spezialfall der Multiplikation betrachtet werden. In diesem Abschnitt werden Algorithmen und Methoden vorgestellt, welche eine effiziente Berechnung der genannten arithmetischen Operationen im Binärkörper $GF(2^n)$ ermöglichen. Da es sich bei der Binärkörper-Arithmetik um die unterste Abstraktionsebene der Skalarmultiplikation handelt (vgl. Abbildung 2.6), muss an dieser Stelle die Datenwortbreite w der ausführenden Hardware-Architektur berücksichtigt werden. Ein Element a des Binärkörpers $GF(2^n)$ benötigt in Polynombasis-Darstellung mindestens einen n-Bit-breiten Vektor zur Speicherung der Koeffizienten (vgl. Gleichung 2.1). Bei einer vorgegebenen Datenwortbreite w können die Koeffizienten in einer geeigneten Datenstruktur, z. B. einem Array A mit der Größe $t = \lceil n/w \rceil$, wie folgte gespeichert werden:

A[t-1]		A[2]	A[1]	A[0]
$0,\cdots,0,a_{n-1},\cdots,a_{(t-1)w}$	$\cdots$	$a_{3w-1},\cdots,a_{2w}$	$a_{2w-1},\cdots,a_{w}$	$a_{w-1},\cdots,a_{0}$

Abbildung 3.2: Datenstruktur zur Speicherung der Koeffizienten eines Binärkörper-Elementes in Polynombasis-Darstellung

Die höchstwertigen $wt-n$ Bit des Array-Elementes $A[t-1]$ werden zur Speicherung der Koeffizienten nicht benötigt und mit Nullen aufgefüllt. Um beispielsweise Elemente des Binärkörpers $GF(2^{233})$ mit einer Hardware-Architektur zu bearbeiten, die eine Datenwortbreite von 32 Bit unterstützt, wird ein Array mit einer Größe von $\lceil 233/32 \rceil = 8$ Einträgen je 32 Bit als Datenstruktur verwendet, in welchem die obersten $32 \cdot 8 - 233 = 23$ Bit unbenutzt bleiben. Die Parameter $n = 233$ als Erweiterungsgrad des Binärkörpers und $w = 32$ als Datenwortbreite werden im weiteren Verlauf dieser Arbeit als Standardwerte angenommen. Die Auswahl beruht auf der Tatsache, dass viele eingebettete Prozessoren heutzutage eine Datenwortbreite von 32 Bit aufweisen und eine Schlüssellänge von 233 Bit für Kryptographie mit elliptischen Kurven bis zum Jahr 2030 als sicher angesehen wird (vgl. Tabelle 2.2).

3.3.1 Addition

Die Addition im Binärkörper stellt die einfachste arithmetische Operation dar. Sie kann durch eine bitweise `XOR`-Verknüpfung der Array-Elemente berechnet werden (vgl. Gleichung 2.2). Wie Algorithmus 3 zeigt, kann die Addition von zwei Elementen eines Binärkörpers in t Iterationsschritten durchgeführt werden.

Algorithmus 3: *Addition von zwei Elementen des Binärkörpers* $GF(2^n)$

Eingabe : $a, b \in GF(2^n)$ als Array A, B mit t Elementen der Wortbreite w
Ausgabe : $c = a + b$ als Array C mit t Elementen der Wortbreite w

1 **for** i **from** 0 **to** $t-1$ **do**
2 $\quad C[i] \leftarrow A[i]$ `xor` $B[i]$
3 **end for**
4 **return** C

3.3.2 Multiplikation

Im Vergleich zur Addition ist die Multiplikation von Elementen des Binärkörpers $GF(2^n)$ eine sehr rechenintensive Operation. Da selbst bei der Verwendung von projektiven Koordinaten die Multiplikation häufig benötigt wird (vgl. Gleichung 3.8), wirkt sich eine Optimierung des Multiplikationsalgorithmus stark auf die Rechenzeit der gesamten Skalarmultiplikation aus. Ein naives Verfahren zur Berechnung der Körper-Multiplikation in $GF(2^n)$ stellt die *Schiebe-und-Addiere*-Methode dar. Wie in Algorithmus 4 zu sehen ist, benötigt diese Methode genau n Iterationen. In jeder Iteration wird der Ergebnisvektor C um eine Stelle nach links geschoben und falls das entsprechende Bit des Multiplikators A gesetzt ist, wird der Multiplikand B aufaddiert. Die Koeffizienten von A werden hierbei vom höchstwertigsten Bit (MSB, engl. Most Significant Bit) zum niederwertigsten Bit (LSB, engl. Least Significant Bit) ausgewertet. Der *Schiebe-und-Addiere*-Algorithmus besitzt zwar eine einfache Struktur, ist in der Praxis aber ineffizient, da das Schieben des Ergebnisvektors C eine Operation auf einem Array mit $2t$ Elementen darstellt. Der Ergebnisvektor muss doppelt so viele Elemente speichern können, da die Multiplikation zweier Polynome vom Grad $n-1$ als Ergebnis ein Polynom vom Grad $2n-2$

Algorithmus 4: *Schiebe-und-Addiere-Methode zur Multiplikation in* $GF(2^n)$

Eingabe : $a, b \in GF(2^n)$ als Array A, B mit t Elementen der Wortbreite w
Ausgabe : $c = a \cdot b$ als Array C mit $2t$ Elementen der Wortbreite w

```
1 C ← 0
2 for i from n − 1 downto 0 do
3     C ← C << 1
4     if Bit i mod w von A[i/w] = 1 then C ← C xor B
5 end for
6 return C
```

produziert (vgl. Gleichung 2.4). Es ist daher im Anschluss an die Multiplikation eine Modulo-Reduktion (vgl. Algorithmus 9) mit einem irreduziblen Polynom notwendig.

Um in Algorithmus 4 die aufwendigen Schiebeoperationen innerhalb eines Arrays zu vermeiden, wird im Folgenden ein *Teile-und-Herrsche*-Verfahren angewandt, welches die Multiplikation der Polynome vom Grad $n-1$ auf mehrere Teilmultiplikationen vom Grad $n/2-1$ aufteilt. Dieses Prinzip kann rekursiv solange angewandt werden, bis die partiellen Polynome nur noch den Grad $w/2-1$ besitzen. Auf diese Weise kann das Teilergebnis in einem Datenwort der Breite w gespeichert werden, so dass kein Array mehr benötigt wird und die Schiebeoperationen über Datenwortgrenzen hinweg entfallen.

Abbildung 3.3 zeigt das *Teile-und-Herrsche*-Verfahren am Beispiel einer Multiplikation im Binärkörper $GF(2^{233})$. Die Datenwortbreite wird wieder als $w = 32$ Bit angenommen, so dass t jeweils die notwendige Größe des Arrays angibt, die zur Speicherung der Koeffizienten eines Polynoms benötigt wird. Anstatt direkt zwei Arrays der Größe $t = 8$ als Eingabe für Algorithmus 4 zu verwenden, wird die Größe der

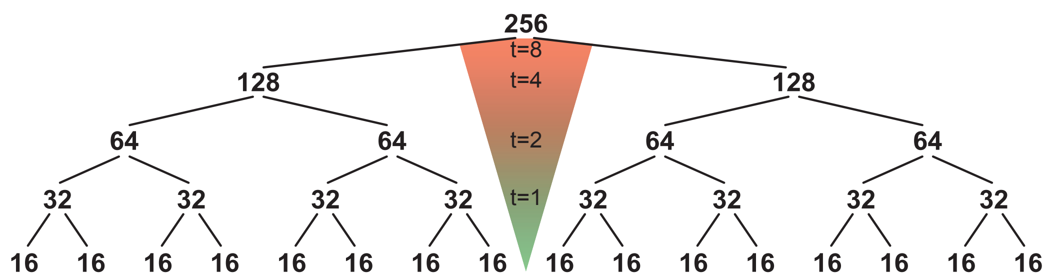


Abbildung 3.3: Reduktion der Polynomlänge nach dem Teile-und-Herrsche-Prinzip

Eingabe-Arrays rekursiv solange halbiert, bis deren Ergebnis C in einem einzigen Datenwort gespeichert werden kann.

Seien $a, b \in GF(2^n)$ in Polynombasis-Darstellung $A = \sum_{i=0}^{n-1} a_i x^i$ und $B = \sum_{i=0}^{n-1} b_i x^i$. Nach der klassischen Methode kann ein Polynom vom Grad $n-1$ durch vier Multiplikationen und einer Addition von Polynomen vom Grad $n/2 - 1$ auf folgende Weise berechnet werden:

$$\begin{aligned} A(x)B(x) &= (A_1 x^{n/2} + A_0)(B_1 x^{n/2} + B_0) \\ &= A_1 B_1 x^n + (A_1 B_0 + A_0 B_1) x^{n/2} + A_0 B_0 \end{aligned} \tag{3.9}$$

Algorithmus 5 beschreibt die rekursive Multiplikation auf Basis der klassischen Methode nach Gleichung 3.9. Die Array-Strukturen A_1, A_0, B_1, B_0 entsprechen jeweils der oberen bzw. unteren Hälfte der Eingabe-Arrays A und B (vgl. Abbildung 3.3). Die vier Multiplikationen in den Zeilen 6 – 9 von Algorithmus 5 werden durch einen rekursiven Aufruf des Algorithmus berechnet. In jeder Rekursion halbiert sich die Anzahl der Array-Elemente t, bis die Abbruchbedingung $t < t_0$ erreicht ist. Im Beispiel zu Abbildung 3.3 würde $t_0 = 1$ gesetzt, so dass das Ergebnis der Multiplikation in Zeile 3 von Algorithmus 5 in einem Datenwort der Breite $w = 32$ Bit gespeichert werden kann. Diese Multiplikation auf Wort-Ebene kann beispielsweise mit Algo-

Algorithmus 5: *Rekursive Multiplikation nach der klassischen Methode*

Eingabe : $a, b \in GF(2^n)$ als Array A, B mit t Elementen der Wortbreite w, Abbruchbedingung t_0
Ausgabe : $c = a \cdot b$ als Array C mit $2t$ Elementen der Wortbreite w

1 Array A_1, A_0, B_1, B_0 mit $t/2$ Elementen der Wortbreite w
2 Array C_3, C_2, C_1, C_0 mit t Elementen der Wortbreite w
3 **if** $t < t_0$ **then return** $A \cdot B$
4 $A_1 \leftarrow (A[t-1], \cdots, A[t/2])$, $A_0 \leftarrow (A[t/2-1], \cdots, A[0])$
5 $B_1 \leftarrow (B[t-1], \cdots, B[t/2])$, $B_0 \leftarrow (B[t/2-1], \cdots, B[0])$
6 $C_3 \leftarrow A_1 \cdot B_1$
7 $C_2 \leftarrow A_1 \cdot B_0$
8 $C_1 \leftarrow A_0 \cdot B_1$
9 $C_0 \leftarrow A_0 \cdot B_0$
10 **return** $(C_3 << wt)$ `xor` $((C_2$ `xor` $C_1) << wt/2)$ `xor` C_0

rithmus 4 durchgeführt werden. Da sich nach Gleichung 3.9 durch jede Halbierung der Polynomlänge die Anzahl der Teilmultiplikationen vervierfacht, benötigt man zur Multiplikation zweier Polynome vom Grad $n-1$ und $t = \lceil n/w \rceil$ durch rekursives Anwenden der klassischen Methode insgesamt $4^{(1+\log_2 \lceil t/t_0 \rceil)}$ Teilmultiplikationen vom Grad $w\lceil t_o/2 \rceil - 1$.

Die Anzahl an partiellen Multiplikationen kann durch Einsatz der Karatsuba-Methode jedoch noch weiter reduziert werden [KO63] [Kar95]. Im Vergleich zur klassischen Methode kann so eine rechenintensive Binärkörper-Multiplikation auf Kosten von zwei zusätzlichen Binärkörper-Additionen eingespart werden. Das Prinzip der Karatsuba-Methode ist in Gleichung 3.10 dargestellt und beruht darauf, die bereits berechneten Teilmultiplikationen A_1B_1 und A_0B_0 zur Berechnung des mittleren Teilpolynoms wieder einzubeziehen (siehe auch Kapitel 5.2.2).

$$\begin{aligned} A(x)B(x) &= (A_1x^{n/2} + A_0)(B_1x^{n/2} + B_0) \\ &= A_1B_1x^n + [(A_1 + A_0)(B_1 + B_0) + A_1B_1 + A_0B_0]x^{n/2} + A_0B_0 \end{aligned} \tag{3.10}$$

Durch rekursive Anwendung der Karatsuba-Methode nach Algorithmus 6 lässt sich die Gesamtanzahl der benötigten Teilmultiplikationen auf $3^{(1+\log_2 \lceil t/t_0 \rceil)}$ reduzieren. Für das Beispiel in Abbildung 3.3 ($n = 233$, $t = 8$, $t_0 = 1$, $w = 32$) bedeutet dieses, dass die Binärkörper-Multiplikation in $GF(2^{233})$ anstatt mit 256 Teilmultiplikationen

Algorithmus 6: *Rekursive Multiplikation nach der Karatsuba-Methode*

Eingabe : $a, b \in GF(2^n)$ als Array A, B mit t Elementen der Wortbreite w, Abbruchbedingung t_0
Ausgabe : $c = a \cdot b$ als Array C mit $2t$ Elementen der Wortbreite w

1 Array A_1, A_0, B_1, B_0 mit $t/2$ Elementen der Wortbreite w
2 Array C_2, C_1, C_0 mit t Elementen der Wortbreite w
3 **if** $t < t_0$ **then return** $A \cdot B$
4 $A_1 \leftarrow (A[t-1], \cdots, A[t/2])$, $A_0 \leftarrow (A[t/2-1], \cdots, A[0])$
5 $B_1 \leftarrow (B[t-1], \cdots, B[t/2])$, $B_0 \leftarrow (B[t/2-1], \cdots, B[0])$
6 $C_2 \leftarrow A_1 \cdot B_1$
7 $C_1 \leftarrow (A_1 + A_0) \cdot (B_1 + B_0)$
8 $C_0 \leftarrow A_0 \cdot B_0$
9 **return** $(C_2 << wt)$ `xor` $((C_2$ `xor` C_1 `xor` $C_0) << wt/2)$ `xor` C_0

nach der klassischen Methode (Algorithmus 5) nur 81 Teilmultiplikationen von Polynomen vom Grad 15 nach der Karatsuba-Methode (Algorithmus 6) benötigt.

Abschließend können die rekursiven Multiplikationsverfahren weiter optimiert werden, indem zur Multiplikation auf Wort-Ebene Algorithmus 4 durch Algorithmus 7 ersetzt wird. Algorithmus 7 [BGTZ08] arbeitet ebenfalls nach dem *Schiebe-und-Addiere*-Prinzip, allerdings wird im Gegensatz zu Algorithmus 4 der Multiplikand nicht bitweise ausgewertet, sondern in Blöcken von vorgegebener Größe s. Hierdurch beschleunigt sich die Berechnung auf Kosten von 2^s Datenworten (Array U) an zusätzlichem Speicherplatzbedarf. Zunächst wird in Zeile $1 - 5$ das Array U mit den Vielfachen vom Multiplikator B, d. h. den Werten $B(x) \cdot P(x)$ für alle Polynome $P(x)$ vom Grad kleiner $s - 1$, initialisiert. Das Array U dient anschließend als Nachschlagetabelle (LUT, engl. Look-Up Table), welche durch die entsprechenden s Bit von A adressiert wird. Die `for`-Schleife in den Zeilen $6 - 9$ bearbeitet folglich immer s Bit von A gleichzeitig und addiert den entsprechenden, vorher berechneten Wert aus U passend verschoben zum Ergebnis C. Wie man in Zeile 8 erkennen kann, geschieht die Berechnung für die obere Ergebnishälfte $C[1]$ unabhängig von der unteren Er-

Algorithmus 7: *w-Bit-Wort-Multiplikation mit Blockgröße s*

Eingabe : A, B als w-Bit-Datenwort, Blockgröße s
Ausgabe : $C = A \cdot B$ als Array C mit 2 Elementen der Wortbreite w

1 Array U mit 2^s Elementen der Wortbreite w
2 $C[0] \leftarrow 0$, $C[1] \leftarrow 0$, $U[0] \leftarrow 0$, $U[1] \leftarrow B$
3 **for** i **from** *1* **to** $2^{s-1} - 1$ **do**
4 $\quad U[2i] \leftarrow U[i] << 1$, $U[2i+1] \leftarrow U[2i]$ `xor` B
5 **end for**
6 **for** i **from** 0 **to** $\lfloor w/s \rfloor$ **do**
7 $\quad V \leftarrow U[A >> (si)$ `and` $(2^s - 1)]$
8 $\quad C[0] \leftarrow C[0]$ `xor` $V << (si)$, $C[1] \leftarrow C[1]$ `xor` $V >> (w - si)$
9 **end for**
10 $V \leftarrow (2^s - 2) \cdot (\sum_{i=0}^{2si<2w} 2^{si}) \bmod 2^w$, $U[0] \leftarrow A$
11 **for** i **from** 1 **to** $s - 1$ **do**
12 $\quad U[0] \leftarrow (U[0] << 1)$ `and` V
13 $\quad$ **if** Bit $w - i$ von $B = 1$ **then** $C[1] \leftarrow C[1]$ `xor` $U[0]$
14 **end for**
15 **return** C

gebnishälfte $C[0]$. Hierdurch kann Algorithmus 7 auch Eingabe-Polynome A, B vom Grad $w - 1$ verarbeiten, ohne ein Bit des Ergebnis-Array C über Wortgrenzen hinweg schieben zu müssen. Dieses wiederum ermöglicht es, die Abbruchbedingung für die rekursiven Multiplikationsverfahren von $t_0 = 1$ auf $t_0 = 2$ zu erhöhen, d. h. die Rekursionstiefe zu verringern. In den Zeilen $10 - 14$ muss das Ergebnis gegebenenfalls nochmal korrigiert werden, da bei der Initialisierung von U eventuell höherwertige Koeffizienten nicht berücksichtigt wurden, falls B ein Polynom vom Grad größer $w - s - 1$ entspricht.

Benutzt man die rekursive Multiplikation auf Basis der Karatsuba-Methode mit Abbruchbedingung $t_0 = 2$, so wird Algorithmus 7 genau $3^{(\log_2 t)}$ mal zur Multiplikation auf Wort-Ebene aufgerufen. Für das Beispiel in Abbildung 3.3 bedeutet dieses konkret, dass für die Körper-Multiplikation in $GF(2^n)$ mit $n = 233$, $t = 8$, $t_0 = 2$ und $w = 32$ genau 27 Binärkörper-Multiplikationen auf w-Bit-Wort-Ebene nach Algorithmus 7 notwendig sind.

3.3.3 Quadrierung

Die Quadrierung von Elementen im Binärkörper $GF(2^n)$ stellt eine Sonderform der Multiplikation dar. Wie bereits in Gleichung 2.5 gezeigt wird, kann das Element $a \in GF(2^n)$ in Polynombasis-Darstellung quadriert werden, indem die Koeffizienten a_i an Position a_{2i} geschoben werden und die restlichen Koeffizienten mit Nullen aufgefüllt werden (vgl. Abbildung 3.4). Was in Hardware direkt durch eine entsprechende Verdrahtung realisiert werden kann, ist in Software wiederum durch eine Nachschlagetabelle sehr effektiv zu implementieren. In den Zeilen $1 - 7$ von Algorithmus 8 werden für eine vorgegebene Blockgröße s alle möglichen Transformationen vorausberechnet und im Array U gespeichert. Diese Nachschlagetabelle

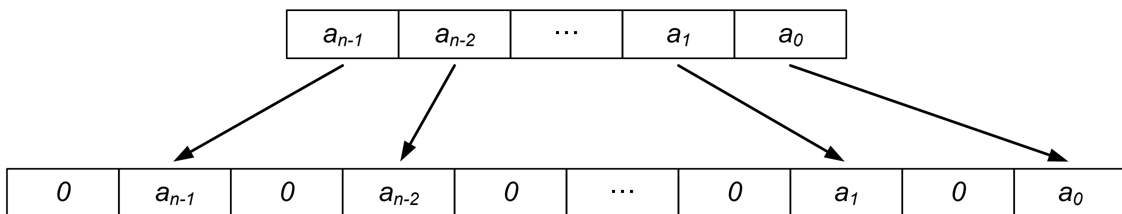


Abbildung 3.4: Quadrierung im Binärkörper $GF(2^n)$

besitzt folglich s^2 Elemente der Wortbreite $2s$ Bit. Die Zeilen 9 – 15 von Algorithmus 8 bearbeiten iterativ alle Datenworte des Eingabe-Arrays A. In der inneren `for`-Schleife werden vom aktuellen Datenwort der Breite w Bit immer zwei s-Bit-breite Blöcke nacheinander verarbeitet. Während in Zeile 12 die untere Hälfte des Datenwortes blockweise von Bit 0 bis Bit $w/2-1$ durchlaufen wird, bearbeitet Zeile 13 die obere Hälfte des Datenwortes blockweise von Bit $w/2$ bis Bit $w-1$. Diese Aufteilung ist sinnvoll, da die Koeffizienten des oberen Halbwortes in ein anderes Datenwort des Ergebnisvektors C abgelegt werden als die Koeffizienten des unteren Halbwortes. Die entsprechend ausmaskierten s-Bit-Blöcke adressieren die Tabelle U und das nachgeschlagene Ergebnis wird an die jeweilige Position im Ergebnisvektor C geschoben.

Algorithmus 8: *Quadrierung von zwei Elementen des Binärkörpers* $GF(2^n)$

Eingabe : $a \in GF(2^n)$ als Array A mit t Elementen der Wortbreite w, Blockgröße s für die gilt: $w \bmod s = 0$

Ausgabe : $c = a^2$ als Array C mit $2t$ Elementen der Wortbreite w

1 Array U mit s^2 Elementen der Wortbreite $2s$
2 **for** i **from** 0 **to** s^2-1 **do**
3 $\quad U[i] \leftarrow 0$
4 $\quad$ **for** j **from** 0 **to** $s-1$ **do**
5 $\quad\quad$ **if** Bit j von $(i_{s-1}i_{s-2},\ldots i_1 i_0)_2 = 1$ **then** $U[i] \leftarrow U[i]$ `xor` $(1 << 2j)$
6 $\quad$ **end for**
7 **end for**
8 $C \leftarrow 0$
9 **for** i **from** 0 **to** $t-1$ **do**
10 $\quad$ **for** j **from** 0 **to** $w/2s$ **do**
11 $\quad\quad p = js,\ q = w/2 + p$
12 $\quad\quad C[2i{+}0] \leftarrow C[2i{+}0]$ `xor` $((U[(A[i]$ `and` $((2^s{-}1) << p)) >> p]) << 2p)$
13 $\quad\quad C[2i{+}1] \leftarrow C[2i{+}1]$ `xor` $((U[(A[i]$ `and` $((2^s{-}1) << q)) >> q]) << 2p)$
14 $\quad$ **end for**
15 **end for**
16 **return** C

3.3.4 Modulo-Reduktion

Die Binärkörper-Multiplikation bzw. Binärkörper-Quadrierung erzeugen bei der Eingabe eines Polynoms vom Grad $n-1$ als Ausgabe ein Polynom vom Grad $2n-2$. Entsprechend weist der Algorithmus 7 bzw. Algorithmus 8 als Eingabe ein Array mit t Elementen der Wortbreite w Bit und als Ausgabe ein Array mit $2t$ Elementen der Wortbreite w Bit auf. Das endgültige Ergebnis der Multiplikation bzw. Quadrierung im Binärkörper $GF(2^n)$ erhält man erst nach Durchführung einer Modulo-Reduktion mit einem irreduziblen Polynom $f(x)$. In [KSFV08] werden mehrere Algorithmen zur allgemeinen Berechnung der Modulo-Reduktion in $GF(2^n)$ vorgestellt. Für den Einsatz in der Kryptographie werden elliptische Kurven mit speziellen Parametern (vgl. Tabelle 2.3) eingesetzt. Die vom *National Institute of Standards and Technology* festgelegten Parameter [NIST94] beinhalten für die jeweilige elliptische Kurve auch das entsprechende irreduzible Polynom (vgl. Anhang A). Dieses Polynom ist häufig nur dünn besetzt (Trinom bzw. Pentanom), so dass der Algorithmus zur Berechnung der Modulo-Reduktion stark vereinfacht werden

Algorithmus 9: *Modulo-Reduktion mit* $f(x) = x^{233} + x^{74} + 1$, $w = 32$

Eingabe : Koeffizienten des binären Polynoms $a(x)$ vom Grad höchstens 464 als Array A mit $2t$ Elementen der Wortbreite w
Ausgabe : $c(x) = a(x) \bmod f(x)$ als Array C mit t Elementen der Wortbreite w

1 $C \leftarrow A$
2 **for** i **from** 15 **to** 8 **do**
3 $\quad V \leftarrow C[i]$
4 $\quad C[i-8] \leftarrow C[i-8]$ `xor` $(V << 23)$
5 $\quad C[i-7] \leftarrow C[i-7]$ `xor` $(V >> 9)$
6 $\quad C[i-5] \leftarrow C[i-5]$ `xor` $(V << 1)$
7 $\quad C[i-4] \leftarrow C[i-4]$ `xor` $(V >> 31)$
8 **end for**
9 $V \leftarrow C[7] >> 9$
10 $C[0] \leftarrow C[0]$ `xor` V
11 $U[2] \leftarrow C[2]$ `xor` $(V << 10)$
12 $C[3] \leftarrow C[3]$ `xor` $(V >> 22)$
13 $C[7] \leftarrow C[7]$ `and` 511
14 **return** $(C[7], C[6], \cdots, C[0])$

kann. Auf Basis der NIST-Parameter werden in [HMV04] optimierte Algorithmen für verschiedene Grundkörper beschrieben. Algorithmus 9 zeigt die Berechnung der Modulo-Reduktion im Binärkörper $GF(2^{233})$ mit dem irreduziblen Polynom $f(x) = x^{233} + x^{74} + 1$.

3.3.5 Invertierung

Jedes Element $a \in GF(2^n)$ besitzt genau eine multiplikative Inverse $a^{-1} \in GF(2^n)$, so dass gilt: $a \cdot a^{-1} = 1$. Die Berechnung der multiplikativen Inversen ist eine sehr rechenintensive Operation, die jedoch durch Verwendung von projektiven Koordinaten bei der Berechnung der Skalarmultiplikation nur einmalig benötigt wird (vgl. Algorithmus 2). Die multiplikative Inverse a^{-1} kann mit Hilfe des kleinen Satzes von Fermat im endlichen Körper $GF(2^n)$ wie folgt ermittelt werden:

$$a^{-1} = a^{2^n-2} = (a^{2^{n-1}-1})^2 \tag{3.11}$$

Der Term innerhalb der Klammer in Gleichung 3.11 kann hinsichtlich des Exponenten n in folgendes Produkt aufgeteilt werden:

$$a^{2^{(i+j)}-1} = (a^{2^i-1})^{2^j} \cdot a^{2^j-1} \tag{3.12}$$

Hierdurch ist es möglich, mit Hilfe von Additionsketten [Knu97] die multiplikative Inverse iterativ durch Multiplikationen und Quadrierungen im endlichen Körper $GF(2^n)$ zu berechnen [IT88]. Die hierzu benutzte Additionskette $K(m)$ stellt eine endliche, monoton steigende Folge $(1, \cdots, m)$ von positiven, ganzen Zahlen dar, welche mit 1 beginnt und alle folgenden Kettenglieder aus der Summe zweier Vorgänger berechnet. Die Auswahl der Summanden u_s,v_s zur Berechnung des nächsten Kettengliedes wird in der Sequenz $S(K)$ festgelegt. Die Länge l einer Additions-

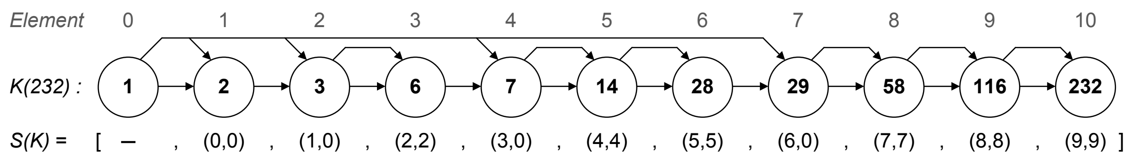


Abbildung 3.5: Additionskette $K(m)$ für $m = 232$ mit minimaler Länge $l(K) = 10$

kette K für eine natürliche Zahl m gibt an, wie viele Additionen benötigt werden, um von dem Startwert 1 zur gewünschten Zahl m zu gelangen. Abbildung 3.5 zeigt eine Additionskette mit minimaler Länge $l(K) = 10$, welche für die Berechnung der multiplikativen Inversen im endlichen Körper $GF(2^{233})$ genutzt wird.

Auf Basis von Gleichung 3.12 und der für den endlichen Körper $GF(2^n)$ entsprechenden Additionskette $K(n-1)$ kann die multiplikative Inverse $a^{-1} \in GF(2^n)$ nach Algorithmus 10 berechnet werden [GN04]. Hierbei werden für die Variablen i und j in Gleichung 3.12 jeweils die Werte aus der Additionskette $K(n-1)$ benutzt, deren Elemente über das Tupel (u_s, v_s) gemäß der Sequenz $S(K)$ in der jeweiligen Iteration s vorgegeben werden. Algorithmus 10 benötigt zur Berechnung der Inversen $a^{-1} \in GF(2^n)$ genau $l(K)$ Multiplikationen und $n-1$ Quadrierungen im endlichen Körper $GF(2^n)$, welche jeweils von der Modulo-Reduktion mit dem entsprechenden irreduziblen Polynom $f(x)$ gefolgt sind.

In Tabelle 3.2 ist die iterative Berechnung entsprechend der äußeren `for`-Schleife (Zeilen 3 – 9 von Algorithmus 10) am Beispiel des endlichen Körpers $GF(2^{233})$ dargestellt. Die erste Zeile in Tabelle 3.2 entspricht der Initialisierung gemäß Zeile 2 in

Algorithmus 10: *Invertierung in $GF(2^n)$ mit Hilfe von Additionsketten*

Eingabe : $a \in GF(2^n)$ als Array A mit t Elementen der Wortbreite w,
irreduzibles Polynom $f(x)$,
Additionskette $K(n-1)$ mit minimaler Länge l und
zugehöriger Berechnungssequenz $S(K) = (u_s, v_s)$
Ausgabe : $a^{-1} \in GF(2^n)$ als Array C mit t Elementen der Wortbreite w

```
1  l Arrays (C_0, ··· , C_l) mit je t Elementen der Wortbreite w
2  C_0 ← A
3  for s from 1 to l do
4      T ← C_{u_s}^2 mod f(x)
5      for v from 1 to v_s do
6          T ← T^2 mod f(x)
7      end for
8      C_s ← T · C_{v_s} mod f(x)
9  end for
10 C_0 ← C_l^2 mod f(x)
11 return C_0
```

s	$\mathbf{u_s}$	$\mathbf{v_s}$	**i**	**j**	**i+j**	**Berechnung von** $\mathbf{C_s}$
0	–	–	0	1	1	$a = a^{2^{1}-1}$
1	0	0	1	1	2	$(a^{2^{1}-1})^{2^{1}} \cdot a^{2^{1}-1} = a^{2^{2}-1}$
2	1	0	2	1	3	$(a^{2^{2}-1})^{2^{1}} \cdot a^{2^{1}-1} = a^{2^{3}-1}$
3	2	2	3	3	6	$(a^{2^{3}-1})^{2^{3}} \cdot a^{2^{3}-1} = a^{2^{6}-1}$
4	3	0	6	1	7	$(a^{2^{6}-1})^{2^{1}} \cdot a^{2^{1}-1} = a^{2^{7}-1}$
5	4	4	7	7	14	$(a^{2^{7}-1})^{2^{7}} \cdot a^{2^{7}-1} = a^{2^{14}-1}$
6	5	5	14	14	28	$(a^{2^{14}-1})^{2^{14}} \cdot a^{2^{14}-1} = a^{2^{28}-1}$
7	6	0	28	1	29	$(a^{2^{28}-1})^{2^{1}} \cdot a^{2^{1}-1} = a^{2^{29}-1}$
8	7	7	29	29	58	$(a^{2^{29}-1})^{2^{29}} \cdot a^{2^{29}-1} = a^{2^{58}-1}$
9	8	8	59	59	116	$(a^{2^{59}-1})^{2^{59}} \cdot a^{2^{59}-1} = a^{2^{116}-1}$
10	9	9	116	116	232	$(a^{2^{116}-1})^{2^{116}} \cdot a^{2^{116}-1} = a^{2^{232}-1}$

Tabelle 3.2: Iterationsschritte gemäß Algorithmus 10 am Beispiel $GF(2^{233})$

Algorithmus 10. Die folgenden Zeilen der Tabelle repräsentieren die Berechnungen innerhalb eines Schleifendurchlaufes der äußeren `for`-Schleife. Die erste Spalte entspricht demnach der jeweiligen Iteration $1 < s < l$, wobei l die Länge der zugrunde liegenden Additionskette $K(m)$ darstellt und in diesem Beispiel $m = n-1 = 232$ und $l = 10$ ist. Spalten 2 und 3 geben die zur aktuellen Berechnung benutzen Elemente der Additionskette gemäß der Sequenz $S(K) = (u_s, v_s)$ wieder. In Spalte 4, 5 und 6 sind die Werte der Elemente u_s und v_s sowie deren Summe angegeben. Die Spalte $i + j$ entspricht dabei den Werten der Additionskette $K(232)$. Die Berechnung von C_s wird mit den jeweiligen Werten i und j nach Gleichung 3.12 durchgeführt. Wie man aus Tabelle 3.2 erkennen kann, ist nach 10 Iterationen die Potenz $a^{2^{232}-1}$, $a \in GF(2^{233})$ berechnet. Dieser Wert wird abschließend noch einmal quadriert (Zeile 10 in Algorithmus 10), so dass nach dem kleinen Satz von Fermat (vgl. Gleichung 3.11) dieser Wert schließlich der multiplikative Inversen $a^1 \in GF(2^{233})$ entspricht:

$$[a^{2^{232}-1}]^2 = a^{2^{233}-2} = a^{-1} \tag{3.13}$$

3.4 Zusammenfassung

Im Rahmen dieser Arbeit wurden verschiedene Methoden betrachtet, um die Berechnung der Skalarmultiplikation zu optimieren. Die Auswahl und Funktionsweise der dazu eingesetzten Algorithmen wurde in diesem Kapitel vorgestellt. Abbildung 3.6 fasst die algorithmischen Optimierungen für jede arithmetische Hierarchieebene der Skalarmultiplikation zusammen.

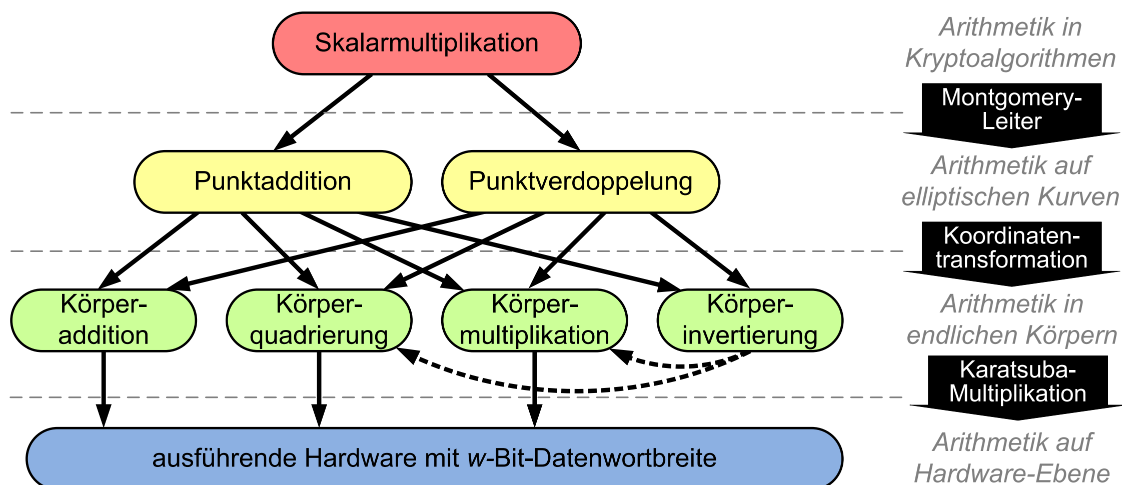


Abbildung 3.6: Algorithmische Optimierung auf verschiedenen Arithmetik-Ebenen

Auf höchster Hierarchieebene wurde die Skalarmultiplikation mit Hilfe der Montgomery-Leiter (vgl. Algorithmus 1) auf Punktadditionen und Punktverdopplungen abgebildet. Der Montgomery-Leiter-Algorithmus bietet dabei nicht nur eine logarithmische Laufzeit, sondern gleichzeitig inhärenten Schutz gegenüber Seitenkanalangriffen, welche auf Analyse der Leistungsaufnahme (Simple Power Analysis) oder der Ausführungszeit (Timing Analysis) beruhen.

Um die rechenintensive Invertierung im Binärkörper auf Ebene der Punktarithmetik zu vermeiden, wurde eine Koordinatentransformation durchgeführt. Durch Verwendung von projektiven Koordinaten nach López-Dahab konnte ebenfalls die benötigte Anzahl an Additionen, Multiplikationen und Quadrierungen im Binärkörper stark reduziert werden (vgl. Tabelle 3.1). Mit Hilfe von Additionsketten konnte die zur Koordinaten-Rücktransformation benötigte Invertierung durch Körper-Multiplikationen bzw. Körper-Quadrierungen berechnet werden.

Die Algorithmen 3 – 10 bilden die arithmetischen Operationen im Grundkörper effizient auf die ausführende Hardware ab. Die besonders häufig benötigte und rechenintensive Multiplikation im Binärkörper wurde durch Anwendung der Karatsuba-Methode (vgl. Algorithmus 6) optimiert.

Alle vorgestellten Algorithmen und Methoden werden in Kapitel 5.1 auf einem Mikroprozessor mit 32-Bit-Datenwortbreite implementiert und die Auswirkung der verschiedenen Optimierungsschritte analysiert.

4 Entwurfsraum

In dieser Arbeit werden ressourceneffiziente Hardware-Software-Kombinationen für Kryptographie mit elliptischen Kurven untersucht. Die Bewertung der Ressourceneffizienz für verschiedene Hardware-Software-Kombinationen hängt dabei stark von dem Einsatzgebiet der Zielanwendung ab. Deshalb werden in diesem Kapitel zunächst zwei exemplarische Anwendungsszenarien mit unterschiedlichen Anforderungen an die Ressourcen vorgestellt. Anschließend wird der Begriff *Ressourceneffizienz* als Bewertungsmaß für unterschiedliche Implementierungen definiert. In Abschnitt 4.4 werden mehrere Implementierungsvarianten vorgestellt, die auf einem hierarchischen Entwicklungskonzept basieren. Angefangen von einem simplen Einzelkernprozessor bis hin zu einem komplexen Multiprozessorsystem werden aufeinander aufbauende Systeme entwickelt. Die verschiedenen Implementierungsvarianten bilden die Grundlage des betrachteten Entwurfsraumes. Die vorgestellte Systemarchitektur kann auf diversen Hierarchieebenen durch Hardware-Beschleuniger, wie z. B. Instruktionssatzerweiterungen oder Coprozessor-Module, anwendungsspezifisch optimiert werden. Abschließend wird in diesem Kapitel eine ausgewählte Implementierungsvariante mit Hilfe einer Rapid-Prototyping-Plattform realisiert. Hierbei werden die Vorteile einer FPGA[1]-basierten Emulation erörtert.

4.1 Anwendungsszenarien

Algorithmen, welche asymmetrische Verfahren der Kryptographie abbilden, stellen die Zielanwendung dieser Arbeit dar. Jedoch unterscheiden sich die Anforderungen dieser Algorithmen bezüglich der Hardware-Ressourcen *Chipfläche*, *Energieverbrauch* und *Rechenleistung* je nach Einsatzgebiet erheblich. In Abbildung 4.1 sind

[1] Field Programmable Gate Array

Anforderungen an zwei Anwendungsszenarien dargestellt. Diese Anwendungsszenarien werden im Folgenden näher beschrieben.

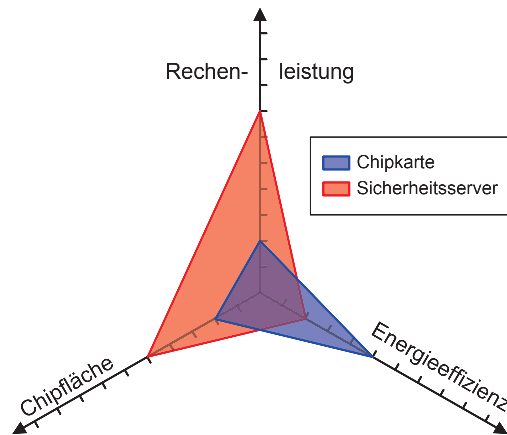


Abbildung 4.1: Anforderungen an die Ressourcen zweier Anwendungsszenarien

4.1.1 Chipkarte

Das Anwendungsszenario *Chipkarte* bzw. *Smartcard* zeichnet sich dadurch aus, dass es sich bei dem Einsatzgebiet meist um mobile Geräte, wie beispielsweise Mobiltelefone handelt. Diese Geräte weisen eine begrenzte Energieversorgung in Form eines Batteriebetriebes auf und müssen möglichst kostengünstig angeboten werden, um sich am Markt durchzusetzen. Die Anforderungen bei diesem Anwendungsszenario sind folglich eine hohe Energieeffizienz sowie eine kleine Chipfläche, da diese proportional zu den Herstellungskosten ist. Die Rechenleistung hingegen spielt nur eine untergeordnete Rolle. Sie wirkt sich auf die benötigte Ausführungszeit der kryptographischen Algorithmen aus. Da der Datendurchsatz bei einem Mobiltelefon als nicht sehr hoch angenommen wird, ergeben sich im Vergleich zur Energieeffizienz und der Chipfläche nur geringe Anforderungen an die Rechenleistung.

4.1.2 Sicherheitsserver

Hinsichtlich eines Client/Server-Systems repräsentiert das Anwendungsszenario Chipkarte in den meisten Fällen die Client-Seite. Die Anforderungen an ein System, wel-

ches auf Server-Seite agiert, sind hierzu genau entgegengesetzt. Das Anwendungsszenario *Sicherheitsserver* stellt ein System dar, bei dem möglichst viele Daten in möglichst kurzer Zeit verarbeitet werden müssen. Solche Systeme werden zum Beispiel bei Anbietern von Online-Banking benötigt. Hier steht die Leistungsfähigkeit, d. h. die Minimierung der Ausführungszeit, im Vordergrund. Die Rechenleistung eines solchen Systems fällt also viel stärker ins Gewicht als die Energieeffizienz und die Chipfläche. Zudem handelt es sich bei diesem Anwendungsszenario um stationäre Geräte mit fester Energieversorgung und es werden im Vergleich zur Chipkarte deutlich höhere Herstellungskosten (Chipfläche) akzeptiert.

4.2 ASIC-Entwurfsablauf

Der Entwurf von anwendungsspezifischen, integrierten Schaltkreisen (ASIC[2]) umfasst eine Reihe von Schritten, deren Ablauf in Abbildung 4.2 veranschaulicht wird. Der dargestellte Ablauf bezieht sich auf den Entwurf digitaler Schaltungen und wurde absichtlich auf die vier wesentlichen Schritte *Implementierung*, *Logik-Synthese*, *Place & Route* und *Tape-Out* reduziert. In der Realität beinhaltet jeder Schritt eine Vielzahl weiterer Unterpunkte, die nicht immer sequentiell miteinander verknüpft sind, sondern sich gegenseitig stark beeinflussen und iterativ durchlaufen werden. Der Fokus in diesem Abschnitt liegt jedoch nicht auf einer möglichst detaillierten Beschreibung der einzelnen Entwurfsschritte, sondern viel mehr auf dem Einfluss potentieller Optimierungsmöglichkeiten und der Analysegenauigkeit hinsichtlich der Bestimmung des Ressourcenverbrauches von *Chipfläche* und *Verlustleistung*.

Wie in Abbildung 4.2 dargestellt ist, beginnt der Entwurfsablauf typischerweise mit einer Spezifikation. In dieser Entwurfsphase werden die Anforderungen an den ASIC sowie die Systemarchitektur festgelegt. Das Optimierungspotential ist in diesem Entwurfsschritt am größten, da man hier in der Regel noch viele Freiheiten beim Entwurf der Systemarchitektur besitzt. Beispielsweise können durch Variation der Pipelinetiefe oder des Grads der Parallelisierung die Eigenschaften des ASICs beeinflusst werden. Eine genaue Analyse des erwarteten Ressourcenverbrauches ist

[2]Application Specific Integrated Circuit

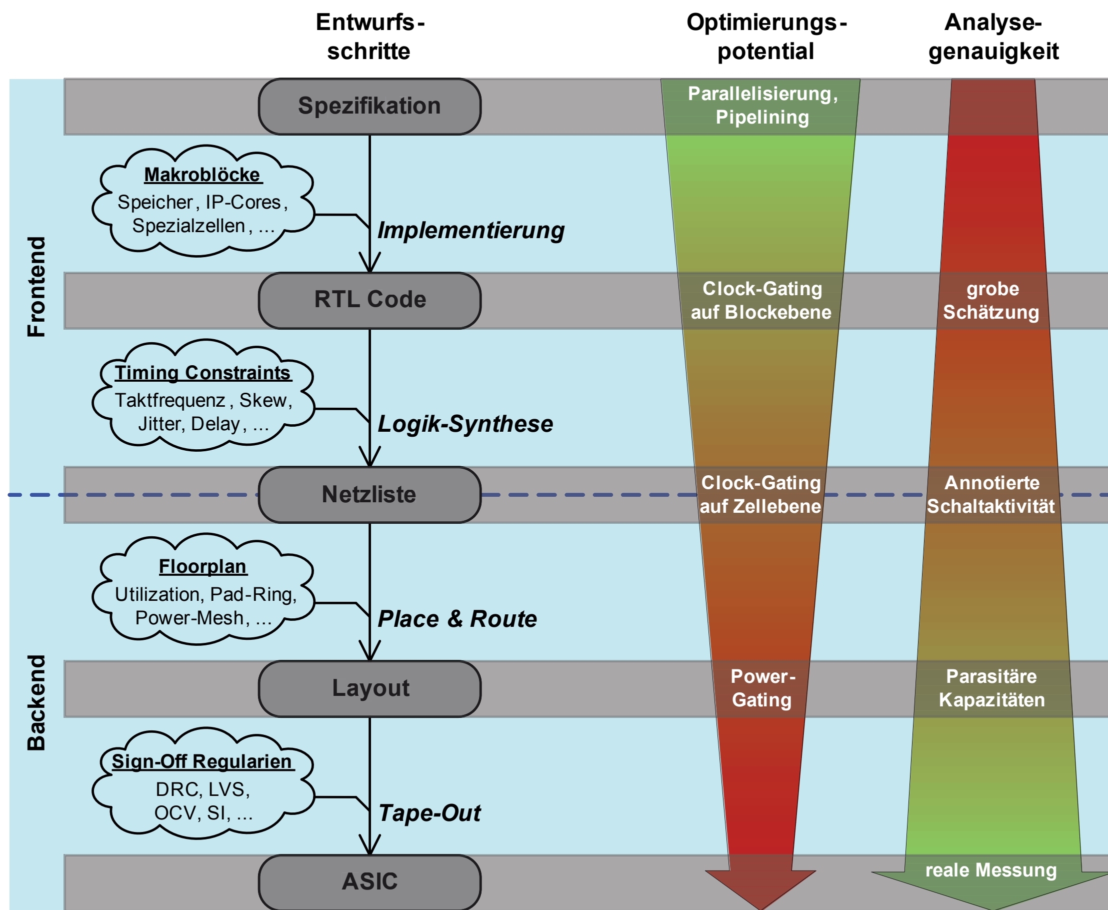


Abbildung 4.2: Vereinfachter Ablauf des digitalen Schaltungsentwurfs

auf dieser Systemebene sehr schwierig und kann nur durch Expertenwissen bzw. Erfahrungswerte abgeschätzt werden [Rab09].

Durch die Implementierung wird das spezifizierte System in eine abstrakte Hardware-Beschreibung auf Register-Transfer-Ebene (RTL, engl. Register Transfer Level) umgesetzt. Hierbei können Makroblöcke, wie z. B. Speicher oder IP-Cores, integriert werden. Der resultierende RTL-Code kann simulativ oder formal verifiziert werden und es können erste Analysen hinsichtlich der Schaltaktivitäten sowie der Verteilung von sequentieller und kombinatorischer Logik durchgeführt werden. Auf der Register-Transfer-Ebene hat man immer noch recht großen Einfluss auf die Optimierung der Ressourcen. Durch Abschaltung des Taktes (engl. clock gating) in

einem Schaltungsblock, dessen Funktionalität nicht permanent benötigt wird, kann zum Beispiel die Verlustleistung reduziert werden [CB95].

Der Synthese-Schritt bildet die Verhaltensbeschreibung des RTL-Codes auf eine ausgewählte Zieltechnologie ab. Das Synthese-Werkzeug versucht bei der Übersetzung die vorgegebenen Grenzwerte hinsichtlich des zeitlichen Verhaltens (engl. timing constraints) einzuhalten. Als Ergebnis entsteht eine Netzliste, die alle benötigten Logik-Gatter sowie eventuelle Makroblöcke und deren Verbindungen beinhaltet. Die Netzliste kann wiederum simuliert werden, um einerseits die korrekte Funktionalität zu überprüfen und andererseits alle Schaltaktivitäten auf Gatter-Ebene zu protokollieren. Hierdurch kann die Genauigkeit der Verlustleistungsanalyse erhöht werden. Das Optimierungspotential hingegen verringert sich, da auf Netzlisten-Ebene hauptsächlich das Synthese-Werkzeug die Optimierungen, wie beispielsweise das Clock-Gating auf Zellebene, durchführt.

Die Netzliste bildet die Schnittstelle zwischen den so genannten Frontend-Schritten und den entsprechenden Backend-Entwurfsschritten. Während im Frontend-Bereich hauptsächlich die Funktionalität und das zeitliche Verhalten der Schaltung implementiert wird, widmet sich der Backend-Bereich im Wesentlichen der Umsetzung, der in der Netzliste instanziierten Komponenten, in eine geometrische Strukturbeschreibung. Dieser Schritt wird Platzieren und Verdrahten (engl. place & route) genannt und erzeugt, unter Berücksichtigung der vorgegebenen geometrischen Grenzen (engl. floorplan), das so genannte Layout. Auf der Layout-Ebene ist die Analysegenauigkeit am größten, da hier die geometrische Struktur aller Gatter und Verbindungsleitungen festgelegt ist. Dieses beinhaltet im Gegensatz zur Netzliste zusätzlich die Gatter-Dichte (engl. utilization), den Taktbaum, die Spannungsversorgung (engl. power mesh) sowie eventuell vorhandene Ein- bzw. Ausgangstreiber (engl. pad ring). Auch auf Layout-Ebene kann eine Simulation durchgeführt werden und sämtliche Schaltaktivitäten können protokolliert werden. Zusammen mit den extrahierten parasitären Kapazitäten ergibt sich die größte Genauigkeit bei der Analyse des Ressourcenverbrauches. Als Optimierung auf Layout-Ebene bietet sich das Abschalten der Versorgungsspannung von temporär nicht benötigten Funktionsblöcken (engl. power gating) an.

Als letzter Schritt im Entwurfsablauf muss das Layout diversen Überprüfungen, wie zum Beispiel dem DRC (engl. Design Rule Check), standhalten. Anschließend kann das Layout an die Fertigung übergeben werden (engl. tape-out).

4.3 Ressourceneffizienz

Die Ressourceneffizienz (RE) stellt eine Metrik zur Bewertung unterschiedlicher Realisierungsvarianten dar. Mit Hilfe der Ressourceneffizienz ist es möglich, verschiedene Implementierungen hinsichtlich einer bestimmten Anwendung miteinander zu vergleichen. Die Ressourcen, welche zur Berechnung dieser Metrik betrachtet werden, entsprechen den Entwurfsparametern *Chipfläche* (A), *Leistungsaufnahme* (P) und *Rechenleistung* (T). Was genau unter diesen Parametern zu verstehen ist, soll im Folgenden erklärt werden.

4.3.1 Chipfläche

Die Chipfläche stellt diejenige Fläche dar, welche benötigt wird, um einen Schaltungsentwurf in einer bestimmten Halbleitertechnologie abzubilden. In dieser Arbeit werden ausschließlich digitale Schaltungsentwürfe betrachtet, die auf eine CMOS[3]-Technologie abgebildet werden. Ausgehend von einer abstrakten Beschreibung der Hardware, z. B. auf Register-Transfer-Ebene, werden Synthese-Werkzeuge benutzt, um den Schaltungsentwurf auf die Gatter-Ebene zu übersetzen (vgl. Abbildung 4.2). Die Chipfläche nach dem Synthese-Schritt beinhaltet die Summe der einzelnen Flächen aller instanziierten Gatter. Diese Flächenangabe kann durchaus mit anderen Flächen, die auf Synthese-Werten basieren, verglichen werden, sie entspricht aber nicht der wirklichen Chipfläche. Um beispielsweise einen Schaltungsentwurf mit existierenden Produkten vergleichen zu können, müssen weitere Backend-Schritte berücksichtigt werden. Man spricht dann nicht mehr von der Synthese-Fläche, sondern bezeichnet die Chipfläche als Layout. Die Layout-Fläche entspricht der realen geometrischen Chipfläche und berücksichtigt die erzielte Gatter-Dichte sowie die Zellen des Pad-Rings.

[3]Complementary Metal Oxide Semiconductor

Die Chipfläche ist eine sehr kostbare Ressource im wörtlichen Sinne, denn sie ist in den meisten Fällen direkt proportional zu den Herstellungskosten. Andererseits kann auf einer größeren Chipfläche auch mehr Funktionalität oder Speicher untergebracht werden, was die Rechenleistung im Allgemeinen verbessert, aber wiederum auch die Leistungsaufnahme erhöht.

4.3.2 Leistungsaufnahme

Die Leistungsaufnahme eines CMOS-Gatters setzt sich aus der statischen und der dynamischen Verlustleistung wie folgt zusammen:

$$P_{\text{total}} = \underbrace{P_{\text{last}} + P_{\text{quer}}}_{\text{dynamisch}} + \underbrace{P_{\text{leck}}}_{\text{statisch}} \tag{4.1}$$

Unter Verlustleistung versteht man die Umsetzung von elektrischer Energie in Wärme. Der dynamische Anteil ist von der Schaltaktivität der Transistoren abhängig, wogegen der statische Anteil auch im Ruhezustand zum Tragen kommt und hauptsächlich von der Fertigungstechnologie beeinflusst wird.

Die statische Verlustleistung wird durch Leckströme innerhalb der Transistoren verursacht. Leckströme fließen ungewollt sowohl im gesperrten als auch im leitenden Zustand eines Feldeffekttransistors und haben drei Hauptursachen [SPG02]. Aufgrund der Dotierung fließen Leckströme an den pn- bzw. np-Übergängen zwischen Drain und Substrat (engl. junction leakage). Weiterhin nimmt mit zunehmender Miniaturisierung die Isolationsschicht zwischen dem Gate und dem Kanal ab, so dass dort ebenfalls ein Leckstrom (engl. gate oxide leakage) fließt. Selbst im gesperrten Zustand fließt aufgrund der diffundierenden Ladungsträger ein Leckstrom zwischen Drain und Source (engl. subthreshold leakage).

Die dynamische Verlustleistung entsteht nur dann, wenn sich der Zustand eines CMOS-Gatters durch einen Schaltvorgang ändert und besteht aus zwei Komponenten [Vee08]. Zum einen fließt bei jedem Schaltvorgang ein Querstrom, da für eine kurze Zeit sowohl der p-Kanal als auch n-Kanal Transistor gleichzeitig leiten. Dieser Querstrom erzeugt die Verlustleistung P_{quer}, dessen Höhe von der Schaltdauer, d. h. der Flankensteilheit am Gatter-Eingang und den physikalischen Eigenschaften der

Transistoren, abhängt. Zum anderen muss bei jedem Schaltvorgang die interne Lastkapazität C_L eines CMOS-Gatters umgeladen werden, was die Verlustleistung P_{last} verursacht. Dieser Anteil stellt in derzeitigen CMOS-Schaltungen ca. 90 % der gesamten Verlustleistung dar, so dass sich P_{total} wie folgt abschätzen lässt [CB95]:

$$P_{\text{total}} \approx \frac{1}{2} \cdot \alpha \cdot C_L \cdot {U_{DD}}^2 \cdot f \tag{4.2}$$

Hierbei bezeichnet α die Schaltaktivität bei der entsprechenden Taktfrequenz f, d. h. die Anzahl der Umladungen von C_L pro Takt. Ist beispielsweise $\alpha = 1$ bedeutet dieses, dass sich in jedem Takt das Ausgangssignal eines CMOS-Gatters ändert. Durch Glitches kann die Schaltaktivität auch größer als Eins werden, in der Regel ist α aber kleiner als Eins, da sich nicht in jedem Takt der Zustand eines CMOS-Gatters ändert. Dieses gilt natürlich nicht für den Taktbaum, dessen Treiber-Gatter in jedem Takt zweimal ihren Zustand ändern. Allerdings kann durch Clock-Gating die Schaltaktivität des Taktbaumes reduziert werden. Andere Optimierungsstrategien, wie z. B. Power-Gating, Multi-V_{th} Zellen oder das dynamische Anpassen der Versorgungsspannung und der Taktfrequenz (DVFS[4]), helfen sowohl die dynamische als auch die statische Verlustleistung weiter zu senken [ITRS07].

4.3.3 Rechenleistung

Die Rechenleistung ergibt sich aus der Zeit, welche benötigt wird, um eine bestimmte Aufgabe zu bearbeiten. Die Aufgabe stellt meist ein algorithmisches Problem dar, welches durch eine sinnvolle Hardware-Software-Partitionierung auf eine Systemarchitektur abgebildet werden muss. Hierbei kann sowohl eine reine Softwarelösung auf ein Prozessorsystem abgebildet werden, als auch eine reine Hardwarelösungen in Form eines IP-Cores realisiert werden. Häufig werden aber nur besonders rechenintensive Teilprobleme auf dedizierte Hardware-Blöcke, z. B. einen Coprozessor, ausgelagert und der Rest des Algorithmus wird als Softwarelösung auf einem Mikroprozessor ausgeführt. Das zur Bewertung der Rechenleistung betrachtete Problem wird auch als Benchmark-Programm bezeichnet. Die benötigte Ausführungszeit des Benchmark-Programmes hängt von der gewählten Hardware-

[4]Dynamic Voltage and Frequency Scaling

Software-Partitionierung und deren Umsetzung ab. Beim Software-Teil spielt die Qualität der Programmierung sowie die Güte des Übersetzers (engl. compiler) eine große Rolle, um einen Algorithmus von einer Hochsprache, beispielsweise ANSI-C, auf die Architektur des Prozessors effizient abzubilden. Die Optimierungsmethoden des Übersetzers können hierbei erheblich die Ausführungszeit beeinflussen. Beim Hardware-Teil bestimmen in erster Linie die maximal erreichbare Taktfrequenz und der Grad der Parallelisierung die Ausführungsdauer. Die maximal erreichbare Taktfrequenz ergibt sich aus dem kritischen Pfad des Schaltungsentwurfs. Ähnlich wie beim Software-Programm spielt hier die Qualität der Hardware-Beschreibung (RTL-Code) und die Güte des Synthese-Werkzeuges eine entscheidende Rolle.

Bei der Ausführungsdauer eines Benchmark-Programmes wird nicht nur die Berechnung betrachtet, sondern auch der Kommunikationsaufwand bewertet. So wirken sich beispielsweise die Speicherbandbreite oder die Synchronisation zwischen Prozessoren eines Multiprozessorsystems ebenfalls auf die Rechenleistung aus.

Die Rechenleistung kann durch Hinzufügen von funktionalen Hardware-Blöcken, beispielsweise in Form von Instruktionssatzerweiterungen oder Coprozessoren, erhöht werden. Ferner kann auch der Grad der Parallelität, zum Beispiel durch eine Multiprozessorarchitektur, vergrößert werden, um die Rechenleistung zu verbessern, sofern der sequentielle Anteil im Benchmark-Programm klein ist. Zur Erhöhung der Taktfrequenz kann weiterhin die Anzahl der Pipeline-Stufen vergrößert werden. Diese Maßnahmen verbessern die Rechenleistung zumeist auf Kosten einer größeren Chipfläche.

4.3.4 Bewertungsmetrik

In der Literatur existieren bereits einige Metriken zur Bewertung von verschiedenen Realisierungsvarianten [CB95][SS05][GP08]. Die meisten davon betrachten allerdings jeweils nur zwei Kriterien. Am bekanntesten darunter sind die folgenden Bewertungsmaße:

- Fläche × Ausführungszeit (ATP, engl. Area Time Product)
- Leistungsaufnahme × Ausführungszeit (PDP, engl. Power Delay Product)
- Energieverbrauch × Ausführungszeit (EDP, engl. Engery Delay Product)

Der Nachteil dieser Metriken besteht darin, dass jeweils ein Kriterium bei der Bewertung nicht berücksichtigt wird. So wird beispielsweise beim Power-Delay-Produkt und beim Energy-Delay-Produkt die Chipfläche nicht berücksichtigt. Es könnten daher zwei Realisierungsvarianten das gleiche Power-Delay-Produkt besitzen, obwohl die Chipfläche und damit auch die Herstellungskosten der beiden Varianten sehr unterschiedlich sein kann. In [SJD04] wurde diese Problematik erörtert und das Power-Delay-Area-Produkt (PDAP) bzw. das Energy-Delay-Area-Produkt (EDAP) als Bewertungsmetrik vorgeschlagen.

Im Rahmen dieser Arbeit wird zur Bewertung der Ressourceneffizienz (RE) der Kehrwert eines gewichteten Produktes mit den Faktoren Chipfläche (A), Leistungsaufnahme (P) und Ausführungszeit (T) in folgender Form verwendet:

$$\mathrm{RE} = \frac{1}{A^{c_A} \cdot P^{c_P} \cdot T^{c_T}} \quad \text{mit } c_A, c_P, c_T \in \{0, 1, 2\} \tag{4.3}$$

Der Kehrwert wird zur Berechnung herangezogen, um dem intuitivem Verständnis von Effizienz gerecht zu werden. Es wird so mathematisch sichergestellt, dass eine kleinere Chipfläche, eine geringere Leistungsaufnahme bzw. eine kürzere Ausführungszeit zu einer Vergrößerung der Ressourceneffizienz beitragen. Ferner soll durch die Exponenten c_A, c_P und c_T eine Gewichtung hinsichtlich des betrachteten Anwendungsszenarios möglich sein. Sind die Kriterien Leistungsaufnahme bzw. Energieverbrauch für ein Einsatzgebiet beispielsweise wichtiger als die Ressource Chipfläche, so kann dieses durch die Exponenten c_P und c_T berücksichtigt werden. Durch die Einschränkung bei der Wahl der Exponenten auf die Werte $0, 1$ und 2 soll sichergestellt werden, dass die Ressourceneffizienz nicht willkürlich durch beliebig große Gewichtungsexponenten verfälscht wird. Jedes Kriterium kann somit sehr stark in der Ressourceneffizienz berücksichtigt werden (Exponent $= 2$), normal berücksichtigt (Exponent $= 1$) oder gar nicht berücksichtigt (Exponent $= 0$) werden. Durch geeignete Wahl der Exponenten können mit der Ressourceneffizienz nach Gleichung 4.3 weiterhin die traditionellen Metriken bzw. deren Kehrwert berechnet werden, aber auch anwendungsspezifisch gewichtete Bewertungen unter Berücksichtigung aller drei Kriterien (Chipfläche, Leistungsaufnahme, Ausführungszeit) vorgenommen werden.

4.3.5 Skalierungsregeln

Um Schaltungsentwürfe, die auf CMOS-Technologien mit unterschiedlichen Strukturgrößen abgebildet wurden, miteinander zu vergleichen, können die Skalierungsregeln nach [FDN+01] angewandt werden. Dabei werden zwei Modelle unterschieden. Bei dem idealisierten Modell wird davon ausgegangen, dass sowohl alle geometrischen Abmessungen eines Transistors als auch die Versorgungsspannung um den Faktor s reduziert werden. Bei gleichzeitiger Erhöhung der Dotierung und Ladungsdichte um den Faktor s bleibt das elektrische Feld im Transistor E konstant. Man spricht bei diesem Modell deshalb auch von *constant field scaling*. Im Laufe der Zeit stellte sich allerdings heraus, dass die Versorgungsspannung nicht im gleichen Maße wie die Strukturgröße verkleinert werden kann. Das allgemeine Modell geht daher davon aus, dass die geometrischen Abmessungen des Transistors weiterhin mit dem Faktor s reduziert werden, während die Versorgungsspannung nur um den Faktor E/s reduziert wird. Als Folge dessen steigt das elektrische Feld ebenfalls um den Faktor E an. Das allgemeine Modell ist daher auch als *generalized field scaling* bekannt. Die Skalierungsfaktoren für die hier betrachteten Ressourcen Chipfläche, Leistungsaufnahme und Rechenleistung sind in Tabelle 4.1 dargestellt. Die Skalierung der Rechenleistung bezieht sich hier auf die Erhöhung der Taktfrequenz um Faktor s, da die Gatter-Laufzeit (engl. gate delay) allgemein mit dem Faktor $1/s$ skaliert [FDN+01].

Ressource	**Skalierungsfaktor**	
	(ideal)	**(allgemein)**
Chipfläche	$1/s^2$	$1/s^2$
Leistungsaufnahme	$1/s^2$	E^2/s^2
Rechenleistung	s	s

Tabelle 4.1: Skalierungsfaktoren für die Ressourcen Chipfläche, Leistungsaufnahme und Rechenleistung

4.4 Hierarchischer Systementwurf

Die Festlegung der Systemarchitektur ist eine wichtige Entwurfsentscheidung, die sehr großen Einfluss auf die Ressourceneffizienz hat. In diesem Kapitel werden verschiedene Systemarchitekturen vorgestellt, die einem hierarchischen Entwicklungskonzept folgen. Das Konzept bietet, von einem simplen Einzelkernprozessorsystem über verschiedene Prozessorfeld-Architekturen bis hin zu einem komplexen Multiprozessorsystem, eine Vielzahl an Realisierungsmöglichkeiten. Durch die modulare, hierarchische Entwicklung können die einzelnen Komponenten fast beliebig miteinander kombiniert und skaliert werden. Einheitliche Schnittstellen ermöglichen außerdem eine einfache Erweiterung der Systemarchitektur durch anwendungsspezifische Hardware-Beschleuniger [NPPR06]. Die im Folgenden vorgestellten Systemvarianten sollen als Grundlage für eine Implementierung dienen. Entsprechend den Anforderungen der Zielanwendung kann so eine Systemvariante ausgewählt werden und als Ausgangspunkt für weitere anwendungsspezifische Optimierungen dienen.

4.4.1 Einzelkernprozessor

Die unterste Ebene der hierarchischen Systemarchitektur bildet ein Mikroprozessor als flexible Datenverarbeitungseinheit. Im Rahmen dieser Arbeit kommt der am Fachgebiet Schaltungstechnik der Universität Paderborn entwickelte Mikroprozessor namens S-Core [Has99][Lan05] zum Einsatz. Der S-Core wurde von Grund auf in der Hardware-Beschreibungssprache VHDL[5] entwickelt und ist binärkompatibel zum Motorola M-Core [Mot98]. Dieses bringt zwei entscheidende Vorteile mit sich. Einerseits können durch die Binärkompatibilität bereits für den M-Core existierende Werkzeuge zur Software-Entwicklung (Assembler, Compiler, Linker, usw.) genutzt werden. Die GNU-Compiler-Collection (GCC) stellt beispielsweise eine frei verfügbare Werkzeugkollektion bereit, die den Motorola M-Core unterstützt [GCC04]. Andererseits ermöglicht der Zugriff auf den VHDL-Code sämtliche Freiheiten für spätere Hardware-Optimierungen, wie zum Beispiel Instruktionssatzerweiterungen. In Abbildung 4.3 ist der schematische Aufbau des S-Core-Prozessorkerns skizziert. Der S-Core stellt einen Einzelkernprozessor mit klassischer RISC[6]-Architektur dar.

[5] Very High Speed Integrated Circuit Hardware Description Language
[6] Reduced Instruction Set Computing

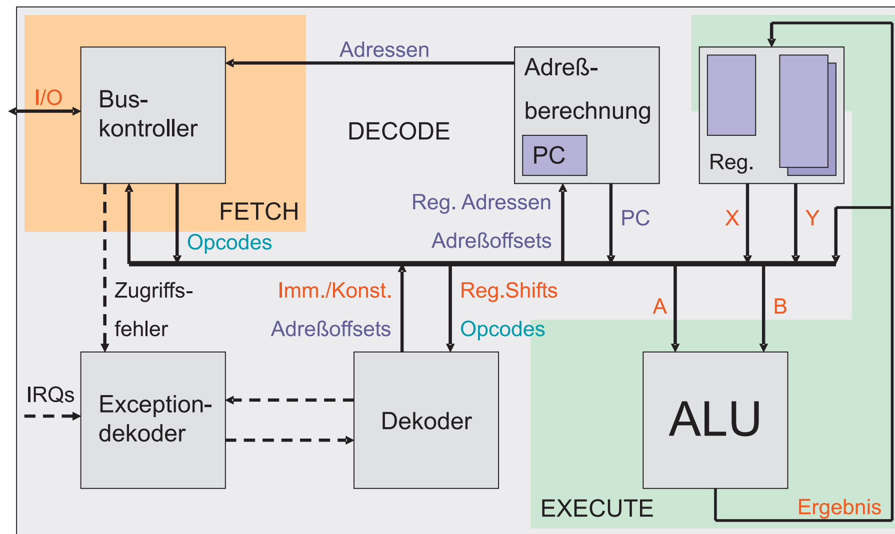


Abbildung 4.3: Schematischer Aufbau eines S-Core-Einzelprozessorkerns [Lan05]

Bei dem S-Core-Mikroprozessor handelt es sich um eine Zwei-Adress-Maschine mit typischem Load/Store-Verhalten, d. h. alle Befehle mit Ausnahme der Lade- und Speicherbefehle arbeiten nur auf Registerwerten. Die Speicheranbindung ist über einen 32-Bit-breiten Prozessorbus (I/O) nach dem Von-Neumann-Prinzip [Fly72] realisiert. Der Buscontroller steuert die Zugriffe auf den Prozessorbus und initiiert als Bus-Master den Transfer von Instruktionen bzw. Daten. Es werden sowohl Wort- (32 Bit), Halbwort- (16 Bit) und Byte-weise (8 Bit) Speicherzugriffe im Big-Endian-Format unterstützt. Der Buscontroller bildet die erste von insgesamt drei Pipeline-Stufen, den so genannten *fetch*-Block. In der zweiten Pipeline-Stufe, dem *decode*-Block, werden die geladenen Instruktionen dekodiert und die Speicheradresse des nächsten Befehles berechnet. Schließlich wird in der dritten Pipeline-Stufe (*execute*-Block) der dekodierte Befehl in der ALU[7] ausgeführt.

Der S-Core verfügt über zwei Registerbänke, die jeweils sechzehn 32-Bit-Register umfassen. Die zusätzliche Registerbank erlaubt einen schnellen Kontextwechsel im Programmablauf und kann beispielsweise für Interrupt-Behandlungen oder Multi-Threading genutzt werden. Weiterhin besitzt der S-Core 13 Kontrollregister, wovon zwei als direkte Ein- bzw. Ausgaberegister benutzt werden können. Das so genannte *Global Control Register* (GCR) ist direkt mit einem 32-Bit-Ausgangsport verbun-

[7] Arithmetic Logic Unit

den, wogegen das *Global Status Register* (GSR) mit einem 32-Bit-Eingangsport verknüpft ist. Über diese beiden Kontrollregister ist der Prozessor in der Lage, direkt Steuerdaten auszugeben (GCR) oder Statusinformationen einzulesen (GSR).

Der Befehlssatz des S-Cores umfasst insgesamt 99 Instruktionen. Alle Instruktionen haben eine feste Wortbreite von 16 Bit, wodurch eine hohe Codedichte erreicht wird. Hierdurch kann nicht nur Speicherplatz für den Programmcode eingespart werden, sondern auch die Speicherbandbreite reduziert werden, da mit einem 32-Bit-Speicherzugriff immer zwei Instruktionen parallel geladen werden. Trotz der geringen Instruktionswortbreite sind noch 11 % des Befehlssatzes ungenutzt und können für Instruktionssatzerweiterungen genützt werden (vgl. Abschnitt 6.1). Die Verteilung der Instruktionen in einzelne Funktionsklassen ist in Abbildung 4.4 dargestellt.

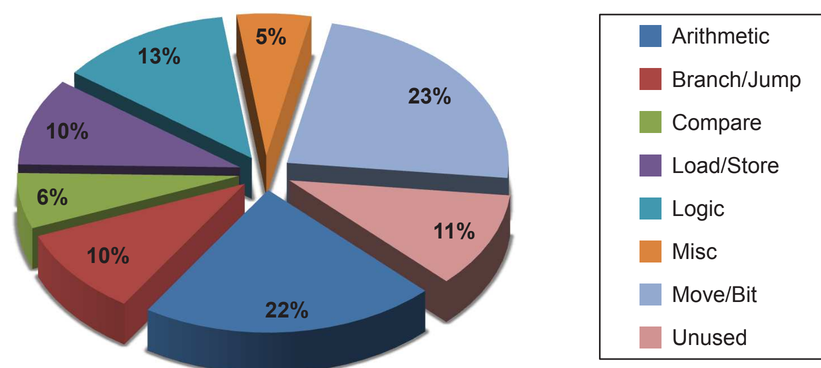


Abbildung 4.4: Verteilung der Instruktionen des S-Core-Mikroprozessors

Wie man aus Abbildung 4.4 erkennen kann, bilden die arithmetischen und logischen Instruktionen zusammen mit den Befehlen zum Kopieren bzw. zur Bitextraktion die größten Funktionsklassen. Der Befehlssatz des S-Cores ist daher besonders für eingebettete Systeme geeignet, wo die effiziente Auswertung und Manipulation einzelner Bits sehr wichtig ist [Put04]. Fast alle Instruktionen benötigen nur einen Takt zur Ausführung. Ausnahmen bilden Sprünge und Speicherzugriffe, welche in zwei Takten bearbeitet werden, sowie die Multiplikation und Division, die iterativ berechnet werden. Eine Multiplikation wird nach dem Booth-Verfahren [PH05] in maximal 18 Takten ausgeführt, eine Division benötigt bis zu 37 Takte.

Auf der nächsten Stufe des hierarchischen Entwicklungskonzeptes wird der S-Core durch zusätzliche Komponenten zu einem Mikroprozessorsystem erweitert [Put05]. Abbildung 4.5 stellt den schematischen Aufbau dieses Mikroprozessorsystems dar. Mit Hilfe des Buscontrollers kann der S-Core als Master über den Prozessorsystembus auf verschiedene Slave-Module zugreifen. Der parametrisierbare Adressdekoder steuert die Arbitrierung und teilt jedem Slave-Modul, das am Prozessorbus angeschlossen ist, einen konfigurierbaren Adressbereich zu. Der S-Core kann so nach dem MMIO[8]-Prinzip [HH07] auf sämtliche Slave-Module zugreifen.

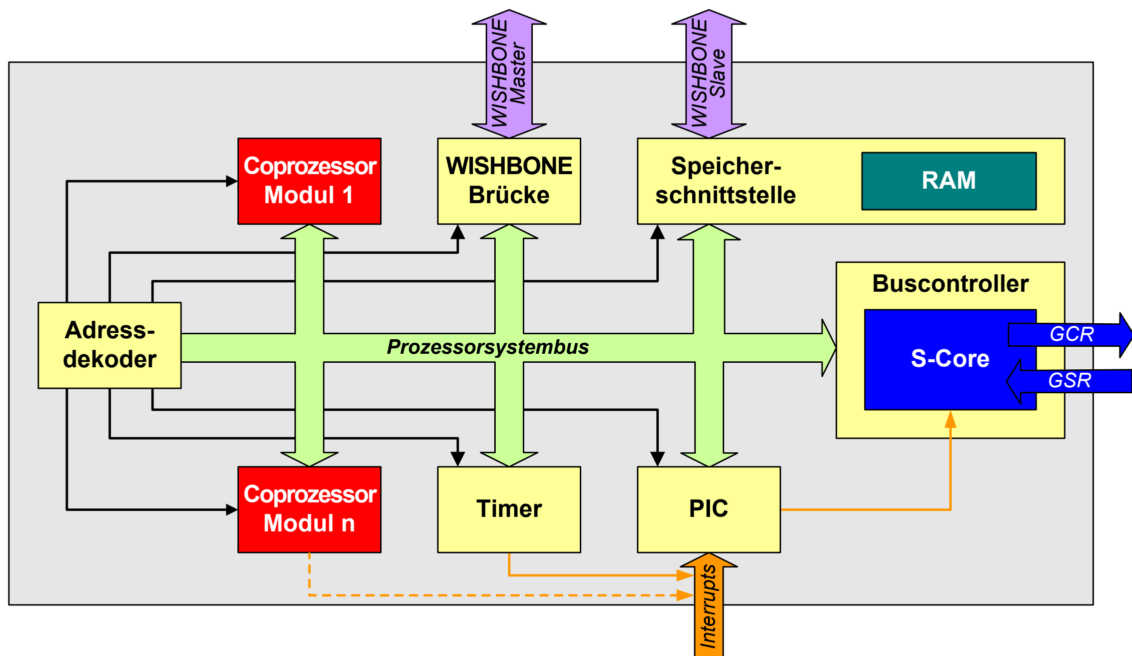


Abbildung 4.5: Prozessorsystem auf Basis des S-Core-Mikroprozessors

Die Grundversion des Mikroprozessorsystems nach Abbildung 4.5 besteht, neben dem S-Core samt Buscontroller und dem Adressdekoder, aus einem programmierbaren Interruptcontroller (PIC), einem Timer-Modul, der Speicheranbindung und einer WISHBONE-Brücke. Über die WISHBONE-Brücke kann das Prozessorsystem mit externen Komponenten auf der nächsten Hierarchieebene erweitert werden. Die Anbindung erfolgt dabei über das frei verfügbare WISHBONE-Protokoll [OCO02]. Durch Verwendung dieses standardisierten Bussystems ist eine universelle Erweiterungsmöglichkeit mit kompatiblen IP-Cores gegeben. Über die Speicherschnitt-

[8]Memory Mapped Input/Output

stelle kann von zwei Seiten auf den lokalen Speicher (RAM[9]) zugegriffen werden. Einerseits greift natürlich der S-Core über den Prozessorsystembus auf den Speicher zu, andererseits kann aber auch von externen Komponenten über die zweite WISHBONE-Schnittstelle auf den Speicher zugegriffen werden, um z. B. während der Initialisierungsphase den Speicher mit Programmcode zu füllen.

Der programmierbare Interruptcontroller erlaubt die Priorisierung und Maskierung von Interrupt-Signalen. Der S-Core unterscheidet zwischen normalen und schnellen Interrupts. Schnelle Interrupts haben eine höhere Priorisierung und nutzen die zusätzliche Registerbank des S-Cores, so dass ein Sichern der regulären Registerinhalte im Speicher entfallen kann. Der programmierbare Interruptcontroller unterstützt bis zu 32 Interrupt-Quellen. Eine davon ist bereits durch das Timer-Modul reserviert. Das Timer-Modul stellt einen programmierbaren Zähler dar, welcher auch als Zeitgeber verwendet werden kann. Mit Hilfe des Timer-Moduls lassen sich beispielsweise die Ausführungszeiten bestimmter Programmabschnitte taktgenau bestimmen, was zur Ermittlung der Rechenleistung (vgl. Abschnitt 4.3.3) nötig ist.

Ferner können dieser Grundversion eine beliebige Anzahl von anwendungsspezifischen Coprozessor-Modulen hinzugefügt werden. Wie alle anderen Module auch, werden die Coprozessor-Module an den Prozessorsystembus angeschlossen und sind im Adressraum des S-Cores eingeblendet. Der Prozessor kann auf diese Weise Daten an die Coprozessor-Module senden. Die Daten werden vom Coprozessor bearbeitet und anschließend wird das Ergebnis dem S-Core zurückgemeldet. Diese Rückmeldung kann entweder aktiv vom Coprozessor durch Auslösen eines Interrupts geschehen oder der S-Core überprüft selber, ob ein Ergebnis am Coprozessor abrufbar ist (Polling-Methode).

4.4.2 Prozessorfeld

Auf der nächst höheren Hierarchieebene werden mehrere S-Core-Prozessorsysteme zu einem Prozessorfeld zusammen geschaltet. Der schematische Aufbau dieses Prozessorfeldes, das im Folgenden auch *Cluster* genannt wird, ist in Abbildung 4.6 dargestellt. Das darin gezeigt S-Core-Subsystem mit integriertem lokalen Speicher

[9] Random Access Memory

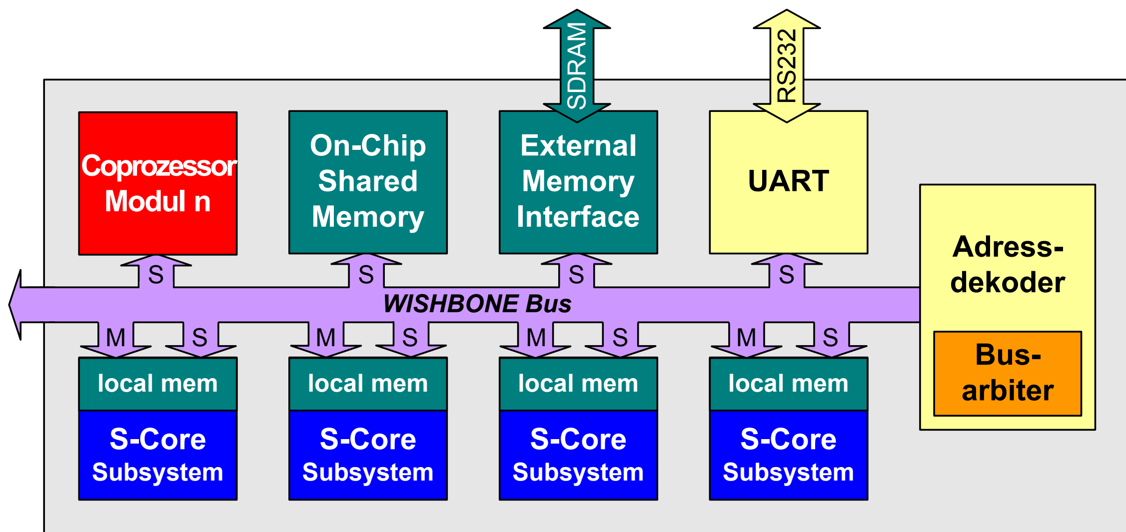


Abbildung 4.6: Prozessorfeld auf Basis des S-Core-Mikroprozessorsystems

entspricht dem zuvor beschriebenen Mikroprozessorsystem aus Abbildung 4.5. Als zentrale On-Chip-Kommunikationsstruktur kommt wiederum der WISHBONE-Bus zum Einsatz. Über ihn können die S-Core-Subsysteme sowohl untereinander als auch mit anderen Slave-Modulen kommunizieren. Der Adressdekoder steuert die Zugriffe auf den WISHBONE-Bus. Im Gegensatz zum Einzelprozessorsystem (vgl. Abbildung 4.5) handelt es sich hierbei allerdings um ein Multi-Master-Kommunikationssystem, d. h. es kann vorkommen, dass mehrere S-Core-Subsysteme gleichzeitig einen Buszugriff starten wollen. Durch den Busarbiter im Adressdekoder wird eine faire Zugriffssteuerung nach dem *Round-Robin*-Prinzip gewährleistet. Neben den einzelnen S-Core-Subsystemen besteht das Prozessorfeld aus einem gemeinsam genutzten Speicher (shared memory), einer Schnittstelle zur Anbindung von externem Speicher sowie einer seriellen Schnittstelle (UART[10]). Der gemeinsame On-Chip-Speicher kann zum Datenaustausch zwischen den S-Core-Subsystemen nach dem *Producer-Consumer*-Prinzip auf Basis von Semaphoren [Tan07] genutzt werden. Zum Auslagern von größeren Datenmengen steht die externe Speicherschnittstelle zur Verfügung. Die serielle Schnittstelle dient dem Anschluss von Ein- bzw. Ausgabegeräten. So können die S-Core-Mikroprozessoren Statusinformationen, z. B. in Form von Textnachrichten, über die serielle Schnittstelle auf einem externen Anzeigemodul

[10]Universal Asynchronous Receiver/Transmitter

ausgeben. Dieses erleichtert die Fehlersuche in komplexen Programmen. Auch auf der Prozessorfeld-Hierarchieebene besteht die Möglichkeit Coprozessor-Module an den WISHBONE-Bus zu koppeln. Im Gegensatz zur Mikroprozessorsystem-Ebene können im Prozessorfeld alle S-Core-Subsysteme von den Coprozessor-Modulen profitieren. Allerdings teilen sich die S-Core-Subsysteme dabei nicht nur den Coprozessor, sondern auch die verfügbare Bandbreite des Kommunikationsbusses.

4.4.3 Multiprozessorsystem

Auf der obersten Hierarchieebene werden mehrere Prozessorfelder über ein On-Chip-Netzwerk (NoC, engl. Network-on-Chip) [BDM02][JT03] miteinander verbunden. Das daraus entstehende, NoC-basierte Multiprozessor-System-on-Chip (MPSoC) ist in Abbildung 4.7 dargestellt. Jedes Prozessorfeld wird über eine so genannte *Switch-Box* an das On-Chip-Netzwerk angebunden. Die Switch-Box fungiert zum einen als Router und propagiert die Daten vom Sender zum Empfänger. Zum anderen bildet die Switch-Box eine aktive Brücke zwischen dem WISHBONE-Bus des Prozessorfeldes und dem On-Chip-Netzwerk. Der Datentransport vom Sender zum Empfänger geschieht Paket-orientiert [DT01]. Der gemeinsame Speicher des Prozessorfeldes wird hier zur temporären Speicherung von ein- und ausgehenden Datenpaketen ge-

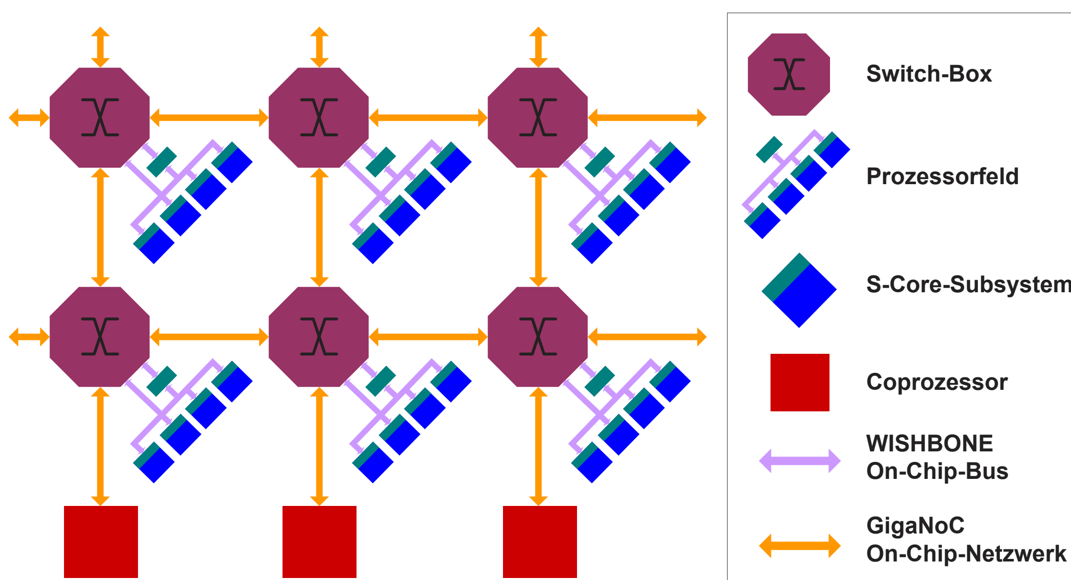


Abbildung 4.7: Multiprozessor-System-on-Chip auf Basis des S-Core-Prozessorfeldes

nutzt und wird daher auch als Paket-Buffer bezeichnet. Die Switch-Box verwaltet den Paket-Buffer und ist in der Lage Datenpakete autonom zu verschicken bzw. zu empfangen. Ein S-Core des Prozessorfeldes muss lediglich über den WISHBONE-Bus einen Befehl inklusive den Zielkoordinaten zum Versenden von Daten an die Switch-Box absetzen. Die Switch-Box übernimmt die komplette Flusskontrolle des Datenversands. Über weitere Befehle können die Prozessoren den Status des Datenversands bzw. das Eintreffen neuer Datenpakete abfragen. Um Daten effizient über das On-Chip-Netzwerk vom Sender zum Empfänger zu transportieren, werden die Pakete in kleine Einheiten, so genannte *Flits* (flow control digits) [ZJ04], unterteilt. Ein Flit beinhaltet jeweils einen Teil des Datenpaketes, der mit zusätzlichen Informationen für das Routing erweitert wird. Flits können bidirektional in einem Takt von Switch-Box zu Switch-Box übertragen werden, sofern die FIFO[11]-Buffer der empfangenden Switch-Box nicht voll sind. Die Buffer befinden sich jeweils am Eingang einer Switch-Box. Für jeden Ausgang existiert dabei ein eigener FIFO-Buffer, so dass nach dem Konzept der *Virtual-Output-Queues* das so genannte *Head-of-Line-Blocking* verhindert werden kann [PY05]. Dieses Problem kann auftreten, wenn nur ein FIFO-Buffer pro Ausgang verwendet würde und ein eingehendes Flit auf einen blockierten Ausgang weitergeleitet werden soll. In diesem Fall müssten alle nachfolgenden Flits am entsprechenden Eingang warten, selbst wenn sie auf einen anderen, nicht blockierten Ausgang weitergeleitet werden könnten. Die Weiterleitung von Flits basiert auf einem statischen XY-Routing, d. h. alle Flits werden, bezogen auf die in Abbildung 4.7 gezeigte Gitter-Topologie, zunächst in X-Richtung und danach in Y-Richtung weitergeleitet.

Die vorgestellte MPSoC-Architektur mit NoC-basierter Kommunikationsinfrastruktur wurde sehr generisch entwickelt [PNPR07][NPPR06], so dass eine Anpassung des Systems an die Anforderungen verschiedener Zielanwendungen möglich ist. Durch die reguläre Struktur kann das Gesamtsystem leicht skaliert, d. h. vergrößert bzw. verkleinert, werden. Weiterhin ist es auch auf dieser Hierarchieebene möglich Coprozessor-Module direkt an das On-Chip-Netzwerk anzubinden. Auf diese Weise können die Coprozessor-Module von allen Prozessorelementen der gesamten MPSoC-Architektur gemeinsam verwendet werden.

[11]First In First Out

4.5 Prototypische Realisierung

Die im vorherigen Abschnitt beschriebene Systemarchitektur wurde auf verschiedene Technologien abgebildet. Als Zielplattform wird eine CMOS-Technologie zur ASIC-Fertigung betrachtet. Daneben wird das System auf eine FPGA-basierte Prototyping-Plattform abgebildet. Der Einsatz dieser Rapid-Prototyping-Umgebung bietet viele Vorteile. Zum einen wird durch die FPGA-basierte Emulation eine höhere Ausführungsgeschwindigkeit gegenüber der Software-basierten Simulation erreicht. Diese Beschleunigung ermöglicht die Evaluierung komplexer Testfälle. Zum anderen können reale Daten, wie z. B. Ethernet-Pakete, über entsprechende Schnittstellen des FPGAs in das System eingespeist werden. Beide Faktoren erlauben eine Verifikation des Testsystems unter realen Bedingungen.

Komponente	**Taktfrequenz** [MHz]	**Chipfläche** [mm^2]	**Leistungsaufnahme** [mW]	[mW/MHz]
S-Core	278	0,127	414,98	1,49
S-Core-Subsystem	270	1,014	423,81	1,57
Prozessorfeld	263	4,997	521,47	1,98
Switch-Box	714	0,530	1510,30	2,12
Ethernet-Controller	435	2,181	957,30	2,20

Tabelle 4.2: Ressourcenverbrauch für eine 90-nm-CMOS-Technologie [IFX04]

In Tabelle 4.2 ist der Ressourcenverbrauch für eine 90-nm-CMOS-Technologie dargestellt. Die angegebenen Werte beruhen auf Synthese-Ergebnissen unter typischen Bedingungen (*typical case*), d. h. einem Prozessparameter von 1,0 sowie einer Versorgungsspannung von 1,20 V und einer Betriebstemperatur von 27 °C. Die Synthese wurde mit dem Design Compiler (Version 2003.06) von Synopsys durchgeführt. Die Analyse der Leistungsaufnahme basiert auf einer statistischen Schaltaktivität von 50 % an den Eingangssignalen der Komponenten. Der S-Core entspricht dem Blockschaltbild 4.3 und stellt den reinen Mikroprozessorkern ohne Speicheranbindung dar. Dieser erreicht auf einer Chipfläche von 0,127 mm^2 eine Taktfrequenz von 278 MHz. Das S-Core-Subsystem, entsprechend Abbildung 4.5, beinhaltet 32 KB Dual-Port-SRAM, welcher hauptsächlich für die Vergrößerung der Chipfläche auf 1,014 mm^2 verantwortlich ist. Das Prozessorfeld besteht, wie in Abbildung 4.6 gezeigt, aus vier

S-Core-Subsystemen. Der gemeinsame Speicher wird ebenfalls durch ein Dual-Port-SRAM mit einer Speicherkapazität von 32 KB realisiert. Die Switch-Box besteht zu 88 % aus Register-basierten FIFOs [PNPR07], welche maßgeblich die große Chipfläche und die hohe Verlustleistung verursachen. Das betrachtete On-Chip-Netzwerk verwendet ein Flit-Format, welches sich aus 29 Bit an Protokollinformationen, dem so genannten *Header*, und 64 Bit an Nutzdaten zusammensetzt. Legt man die Taktfrequenz aus Tabelle 4.2 zugrunde, so ergibt sich ein maximaler Nettodatendurchsatz von 42,29 GBit/s pro Switch-Box-Port [PNPR07].

4.5.1 RAPTOR2000 Rapid-Prototyping-Plattform

Am Fachgebiet Schaltungstechnik der Universität Paderborn wurde das Rapid-Prototyping-System *RAPTOR2000* entwickelt [KPR02]. Die FPGA-basierte Plattform ist modular aufgebaut und besteht aus einer Basisplatine, auf welche bis zu sechs Erweiterungsmodule aufgesteckt werden können (siehe Abbildung 4.8). Die RAPTOR2000-Basisplatine besitzt eine PCI-Bus-Schnittstelle zur Anbindung an einen Host-Computer. Es stehen verschiedene Erweiterungsmodule zur Verfügung [Por09]. Diverse FPGA-Erweiterungsmodule ermöglichen die Emulation von digitalen Schaltungsentwürfen mit einer Komplexität von bis zu 200 Millionen Transistoren. Zusätzlich existieren Erweiterungsmodule, die physikalische Schnittstellen unterschiedlicher Kommunikationsstandards zur Verfügung stellen. Auf diese Weise können reale Daten z. B. über Ethernet, USB, Firewire, I^{2}C, CAN oder LON in das System eingespeist werden. Die einzelnen Erweiterungsmodule werden auf der RAPTOR2000-Basisplatine über ein systeminternes Multi-Master-Bussystem (*Lokal-Bus*) als eigenständige Teilnehmer verwaltet. Über diese Schnittstelle findet sowohl die Kommunikation zwischen den Modulen als auch die Kommunikation mit dem Host-Rechner statt. Ein Broadcast-Bus ermöglicht die gleichzeitige Übertragung von Daten an mehrere Module. Zusätzlich bietet jeder Modulsteckplatz frei konfigurierbare Signalverbindungen zu den benachbarten Modulen, über welche die Module untereinander mit einer Bandbreite von mehr als 10 GBit/s kommunizieren können.

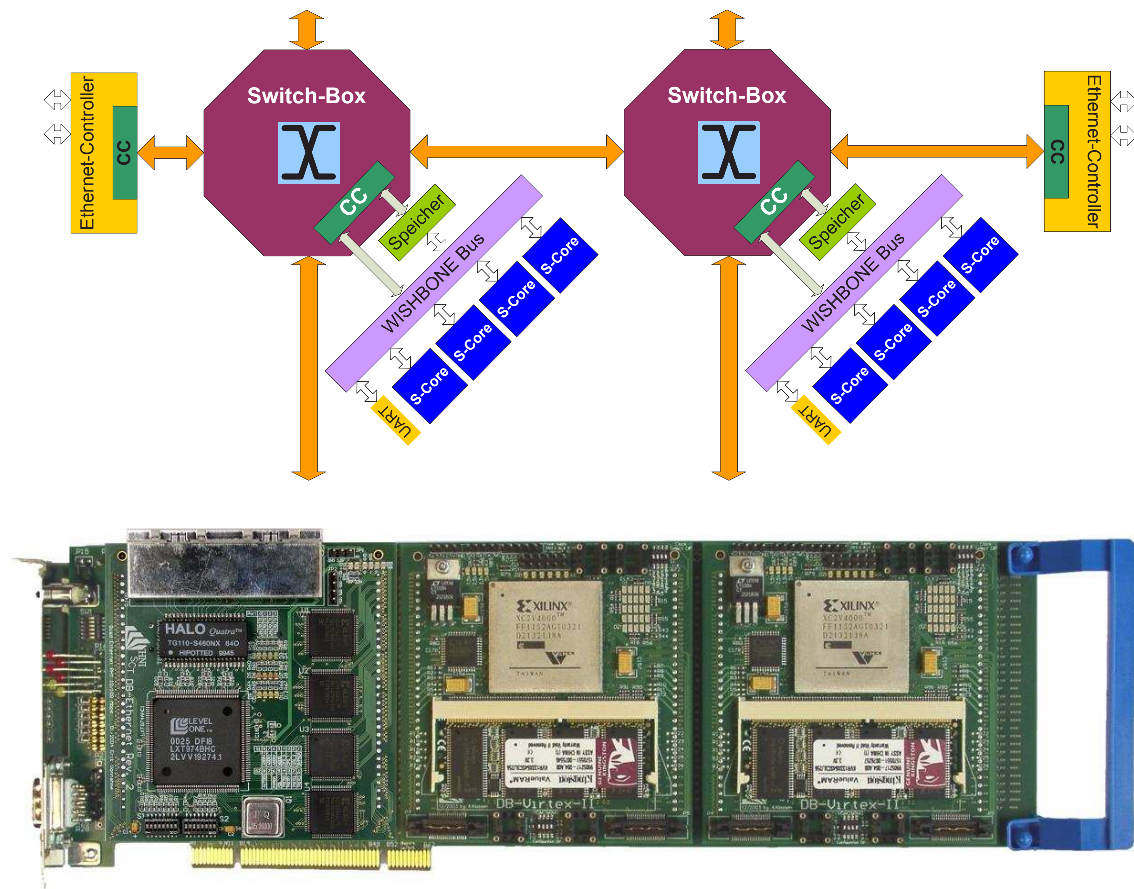


Abbildung 4.8: Testsystem auf Basis der Rapid-Prototyping-Plattform RAPTOR2000

Das RAPTOR2000-System wurde genutzt, um die vorgestellte MPSoC-Architektur (vgl. Kapitel 4.4) prototypisch zu realisieren [Put05][NPPR07]. Abbildung 4.8 zeigt den schematischen Aufbau des Testsystems (oberer Bildteil) und die RAPTOR2000-Basisplatine mit drei aufgesteckten Erweiterungsmodulen (unterer Bildteil). Der implementierte MPSoC-Prototyp besteht aus zwei Prozessorfeldern (vgl. Abbildung 4.6), welche jeweils über eine Switch-Box miteinander verbunden sind. Des Weiteren wird an jede Switch-Box ein Ethernet-Controller angebunden, so dass der Prototyp in einer Testumgebung mit realen Ethernet-Daten stimuliert werden kann. Ein Prozessorfeld samt Switch-Box und angeschlossenem Ethernet-Controller wird jeweils auf ein FPGA-Erweiterungsmodul abgebildet. Zum Einsatz kommen hier die Virtex-II-FPGA-Module DB-V2 [Por09], die zusätzlich zum FPGA-Baustein noch über 8 MB an SRAM verfügen, welcher mit der externen Speicherschnitt-

Komponente	Slices		RAM16s	
S-Core	3206	(6,88 %)	0	–
S-Core-Subsystem	3662	(7,86 %)	16	(9,52 %)
Prozessorfeld	15362	(32,97 %)	80	(47,62 %)
Switch-Box	14133	(30,33 %)	0	–
Ethernet-Controller	5544	(11,90 %)	32	(19,05 %)

Tabelle 4.3: Ressourcenverbrauch für den Xilinx-FPGA Virtex-II-8000

stelle des Prozessorfeldes verbunden ist. Wie in Abbildung 4.8 zu erkennen ist, wird neben den beiden FPGA-Modulen (mittlerer und rechter Modulsteckplatz) ein Kommunikations-Modul verwendet, welches über vier physikalische Ethernet-Schnittstellen verfügt (linker Modulsteckplatz). Jedes der vier S-Core-Subsysteme eines Prozessorfeld besitzt 32 KB an Dual-Port-Speicher zur Ablage von Programmcode und Daten. Der gemeinsam genutzte Paket-Buffer zwischen Switch-Box und Prozessorfeld ist ebenfalls als Dual-Port-Speicher mit 32 KB Speicherkapazität ausgelegt. Weiterhin besitzt jeder Ethernet-Controller einen 16-KB-großen Paketspeicher sowie zusätzlich pro Ethernet-Port 5 KB an Pufferspeicher. Alle Speicherblöcke der MPSoC-Architektur müssen durch FPGA-spezifischen BlockRAM [Xlnx07] realisiert werden. In Tabelle 4.3 ist der Ressourcenverbrauch für einen Xilinx-FPGA vom Typ Virtex-II-8000 dargestellt. Das in Abbildung 4.8 dargestellt Teilsystem, bestehend aus einem Prozessorfeld, einer Switch-Box und einem Ethernet-Controller mit zwei Ethernet-Ports, belegt auf dem Virtex-II-8000 insgesamt 35039 von 46592 verfügbaren Slices (75,20 %) und 112 von 168 verfügbaren RAM16-Blöcken (66,67 %). Der MPSoC-Prototyp kann mit einer Taktfrequenz von 12,50 MHz emuliert werden, was einer durchschnittlichen Beschleunigung um den Faktor 10^5 gegenüber der VHDL-Simulation auf Register-Transfer-Ebene entspricht.

Zur komfortablen Steuerung des FPGA-Prototypen wurde eine grafische Benutzeroberfläche (GUI[12]) in der objektorientierten Programmiersprache Visual-C++ entwickelt [Put05]. Die GUI ist in Abbildung 4.9 dargestellt und baut auf den vom RAPTOR2000-System zur Verfügung gestellten DLL[13]-Funktionen auf. Mittels der grafische Benutzeroberfläche kann der FPGA-Prototyp komfortabel konfiguriert, in-

[12]Graphical User Interface
[13]Dynamic Link Library

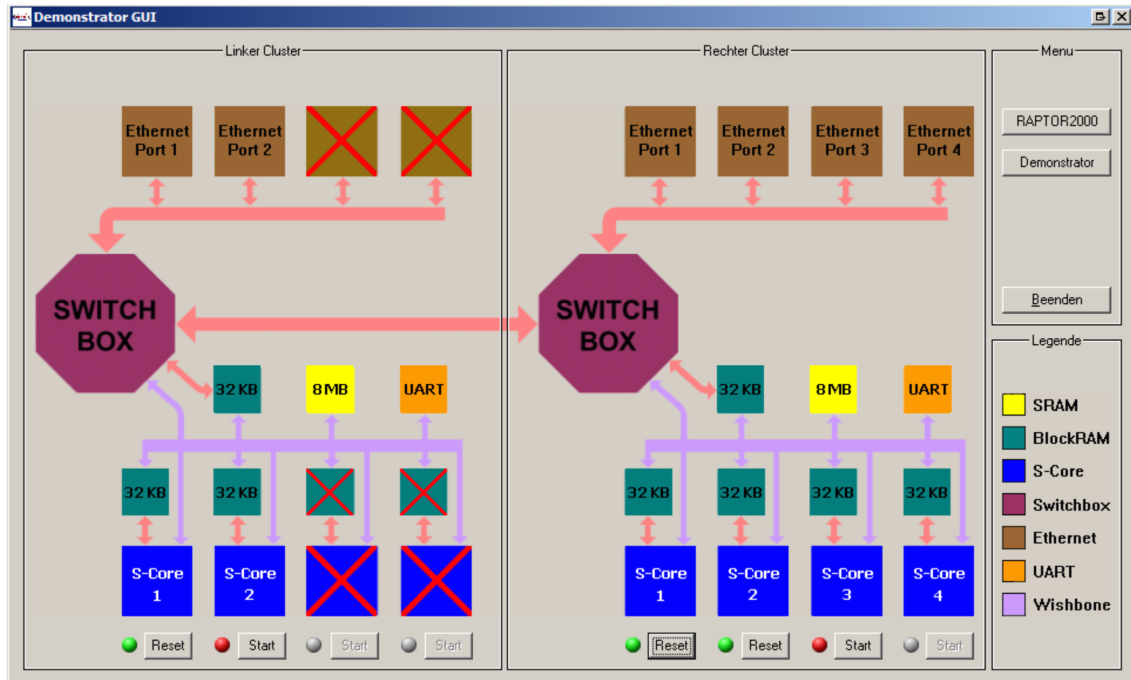


Abbildung 4.9: Grafische Benutzeroberfläche zur Steuerung des FPGA-Prototypen

itialisiert, gesteuert und überwacht werden. Das bedeutet im Einzelnen, dass mit Hilfe der GUI eine entsprechende Konfigurationsdatei (*Bitstream*) auf den jeweiligen FPGA geladen werden kann. Nach der Konfiguration der FPGAs passt sich die grafische Benutzeroberfläche automatisch an die auf den FPGAs abgebildeten Systeme an. Das heißt, es werden die implementierten Speichergrößen sowie die Anzahl der vorhandenen Prozessoren und Ethernet-Ports ermittelt und in der GUI visualisiert (vgl. Abbildung 4.9). Alle dargestellten Symbole der GUI sind kontextsensitiv, d. h. wenn der Benutzer eines der Symbole auswählt erscheint ein symbolspezifisches Dialogfenster. So können beispielsweise die lokalen Speicher der Prozessoren mit Programmcode initialisiert werden, indem der Benutzer einfach das entsprechende Speicher-Symbol in der GUI auswählt. Mit Hilfe der entsprechenden Kontroll-Schaltflächen (*Start*, *Stop*, *Reset*) können die jeweiligen Prozessoren gesteuert werden. Neben der notwendigen Konfiguration, Initialisierung und Steuerung des FPGA-Prototypen bietet die GUI zusätzlich verschiedene Überwachungsmöglichkeiten an. Diese sind besonders für die Fehlersuche sowohl auf Software-Ebene als auch auf Hardware-Ebene geeignet. So ist es beispielsweise möglich, sämtliche Speicherinhalte zur Laufzeit auszulesen. Ferner können Statusinformationen, wie

z. B. FIFO-Füllstände oder Übertragungseigenschaften der Switch-Boxen, Ethernet-Ports und UART-Schnittstellen, angezeigt werden. Das Einblenden dieser Statusinformationen erlaubt die praktische Fehlersuche am laufenden Prototypen. Über die serielle Schnittstelle (UART) kann zusätzlich eine berührungsempfindliche Anzeige (Touch-Display) angeschlossen werden, so dass der Benutzer direkt mit dem FPGA-Prototypen interagieren kann. Die Prozessoren können Textnachrichten an die serielle Schnittstelle schicken, welche dann auf der Anzeige dargestellt werden. Mit Hilfe des Timer-Moduls (vgl. Abbildung 4.5) kann die Ausführungszeit eines Programmabschnitts ermittelt und auf der Anzeige ausgegeben werden. Auf diese Weise können auch komplexe Programme bequem analysiert werden, ohne eine Langzeit-Simulation auf VHDL-Ebene durchführen zu müssen.

4.6 Zusammenfassung

Zunächst wurden in diesem Kapitel zwei Anwendungsszenarien (*Chipkarte*, *Sicherheitsserver*) mit stark konträren Ressourcenanforderungen (vgl. Abbildung 4.1) beschrieben. Während es bei Chipkarten auf eine hohe Energieeffizienz bei kleiner Chipfläche bzw. geringen Herstellungskosten ankommt, steht bei Sicherheitsservern die Rechenleistung des Systems im Vordergrund.

Diese unterschiedlichen Anforderungen erfordern eine skalierbare Systemarchitektur, dessen Ressourcen auf das entsprechende Anwendungsszenario abgestimmt werden können. Die in Abschnitt 4.4 vorgestellte Systemarchitektur ist hierarchisch aufgebaut und ermöglicht vom simplen Einzelkern-Mikroprozessorsystem bis hin zu einem komplexen MPSoC verschiedenste Realisierungsvarianten. Die beschriebene Architektur bietet ferner die Möglichkeit auf verschiedenen Hierarchieebenen lose oder eng gekoppelte Hardwarebeschleuniger einzubinden. Die modulare Struktur ermöglicht eine Skalierung des Systems, um sich zur Entwurfszeit optimal an die Anforderungen verschiedener Anwendungsszenarien anpassen zu können [NPPR06][PPP09].

Um die später betrachteten Implementierungsvarianten miteinander vergleichen zu können, wurde als Metrik die Ressourceneffizienz (*RE*) definiert [PSPR08]. Die Ressourceneffizienz beinhaltet die Entwurfsparameter *Chipfläche*, *Leistungsaufnahme*

und *Rechenleistung* unter Berücksichtigung einer anwendungsspezifischen Gewichtung.

Zur prototypischen Evaluierung von unterschiedlichen Systemvarianten wurde die Rapid-Prototyping-Plattform RAPTOR2000 verwendet. Das RAPTOR2000-System ermöglicht eine schnelle prototypische Realisierung mikroelektronischer Schaltungen, die über verschiedene physikalische Schnittstellen in einer Testumgebung mit realen Daten emuliert und komfortabel am Host-Rechner analysiert werden können. Mit Hilfe der RAPTOR2000-Plattform konnte die Funktionalität des MPSoC-Prototypen erfolgreich verifiziert sowie komplexe Testprogramme mit einer hohen Emulationsgeschwindigkeit evaluiert werden [NPPR07].

5 Evaluierung der Algorithmen

In diesem Kapitel werden die in Abschnitt 3 vorstellten Algorithmen sowohl als Software- wie auch als Hardware-Implementierung umgesetzt und bewertet. Durch diese Evaluierung kann später eine geeignete Hardware-Software-Partitionierung ausgewählt werden. Zunächst wird eine reine Software-Lösung zur Berechnung der Skalarmultiplikation analysiert. Dazu wird eine automatisierte Testumgebung vorgestellt, welche die Evaluierung der Software-Implementierung stark vereinfacht. Von der Berechnung auf Wort-Ebene bis zur kompletten Skalarmultiplikation werden unterschiedliche Algorithmen umgesetzt und miteinander verglichen. Speziell für die kritische Binärkörper-Multiplikation werden verschiedene Hardware-Implementierungen evaluiert. Abschließend wird ein Codegenerator präsentiert, der eine komfortable Erzeugung von parametrisierbaren Multipliziererschaltungen in der Hardware-Beschreibungssprache VHDL erlaubt.

5.1 Software-Evaluierung

Die Berechnung der Skalarmultiplikation dient in diesem Kapitel als Referenzproblem, d. h. als Benchmark, um verschiedene Implementierungen untereinander vergleichen zu können. Ausgangspunkt für die reine Software-Implementierung sind die in Kapitel 3 vorgestellten Algorithmen. Diese werden innerhalb eines Benchmark-Programmes auf der in Abschnitt 4.4 vorgestellten Systemarchitektur ausgeführt. Hierzu wurde eine Skript-basierte Testumgebung entwickelt, welche die Software-Evaluierung durch automatisierte Abläufe vereinfacht.

5.1.1 Automatisierte Testumgebung

Abbildung 5.1 zeigt die automatisierte Testumgebung zur Evaluierung der Software-Implementierungen. Als erstes wird das Benchmark-Programm in ein Speicherabbild übersetzt. Als Übersetzer kommt das M-Core-Derivat der GNU-Compiler-Collection [GCC04] in der Version 4.0.0 zum Einsatz. Dieser wird, solange nicht anders vermerkt, mit dem Optimierungsparameter `-O9` ausgeführt. Das übersetzte Benchmark-Programm wird in den lokalen Speicher des Prozessorsystems geladen und vom S-Core-Mikroprozessor ausgeführt. Das Timer-Modul wird zur Ermittlung der Ausführungszeit herangezogen, indem direkt vor (*Startwert*) und unmittelbar nach (*Stoppwert*) der Ausführung des jeweiligen Algorithmus der Zähler des Timer-Moduls ausgelesen wird. Die Differenz aus Stopp- und Startwert ergibt die Anzahl an Takten, die zur Ausführung des zu untersuchenden Algorithmus benötigt wurden. Zur Ermittlung der Ausführungszeit kann das Testsystem sowohl mit einem entsprechenden Simulator, wie z. B. ModelSim von Mentor Graphics, taktgenau simuliert werden als auch mit der in Abschnitt 4.5 vorgestellten Rapid-Prototyping-Plattform RAPTOR2000 emuliert werden. Insbesondere bei komplexen Algorithmen wie der Skalarmultiplikation kommt der Geschwindigkeitsvorteil einer FPGA-basierten Emulation zum Tragen. Im Vergleich zur Simulation auf Register-Transfer-Ebene kann mit Hilfe der Rapid-Prototyping-Plattform RAPTOR2000 eine durchschnittliche Beschleunigung um den Faktor 10^5 erreicht werden.

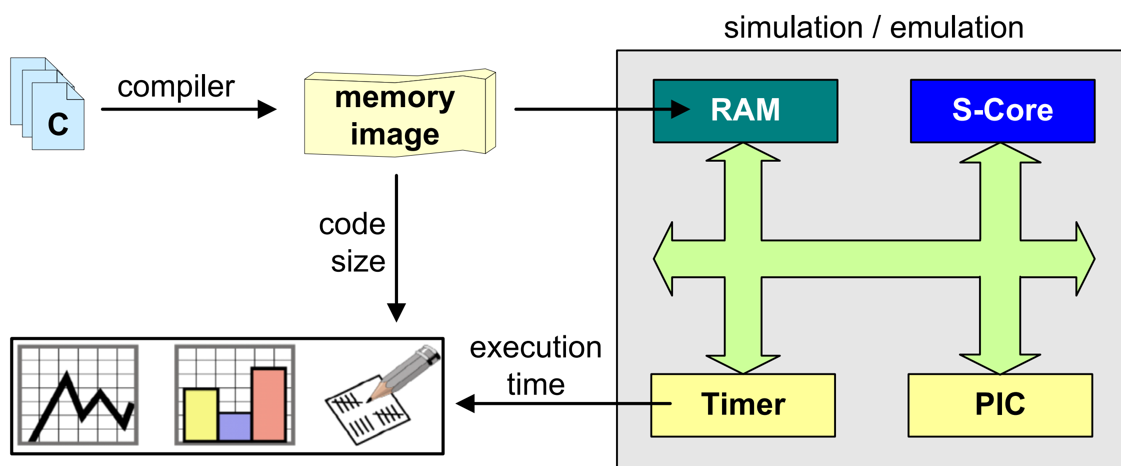


Abbildung 5.1: Automatisierte Testumgebung zur Software-Evaluierung

5.1.2 Evaluierung auf Wort-Ebene

Zunächst werden vier verschiedene Algorithmen zur Binärkörper-Multiplikation auf Wort-Ebene untersucht. Abbildung 5.2 zeigt die Ausführungszeit sowie die Codegröße der analysierten Algorithmen. Die dargestellten Ergebnisse beruhen auf Algorithmus 7, welcher eine w-Bit-Wort-Multiplikation blockweise mit der Blockgröße s berechnet. Die Wortbreite ist entsprechend der S-Core-Architektur auf 32 Bit festgelegt und die Blockgröße s wird zwischen eins und vier variiert. Die Blockgröße $s = 1$ ist ein Sonderfall und entspricht der bitweisen *Schiebe-und-Addier*-Multiplikation, wie sie in Algorithmus 4 beschrieben wird. Da hierbei in jeder Iteration der 64-Bit-breite Ergebnisvektor um ein Bit geschoben werden muss, benötigt die Multiplikationsfunktion `ff_mul_w32_s1` mit 769 Takten die längste Ausführungszeit. Der zugrunde liegende Algorithmus 4 ist sehr simpel, so dass dessen Binärcode nur 184 Byte an Programmspeicher in Anspruch nimmt. Die weiteren drei Multiplikationsfunktionen basieren auf dem komplexeren Algorithmus 7 und benötigen deshalb entsprechend mehr Platz innerhalb des Programmspeichers. Die Ausführungszeit kann jedoch auf Kosten des höheren Speicherplatzverbrauches deutlich gesenkt werden. Den besten Kompromiss zwischen Programmspeicher und Ausführungszeit stellt die Multiplikationsfunktion `ff_mul_w32_s3` dar, welche mit einer Blockgröße von 3 Bit pro Iteration arbeitet (vgl. Anhang B.1). Diese Funktion benötigt zwar 1,63-mal so viel Platz im Programmspeicher, berechnet die Multiplikation jedoch 3,81-mal schneller

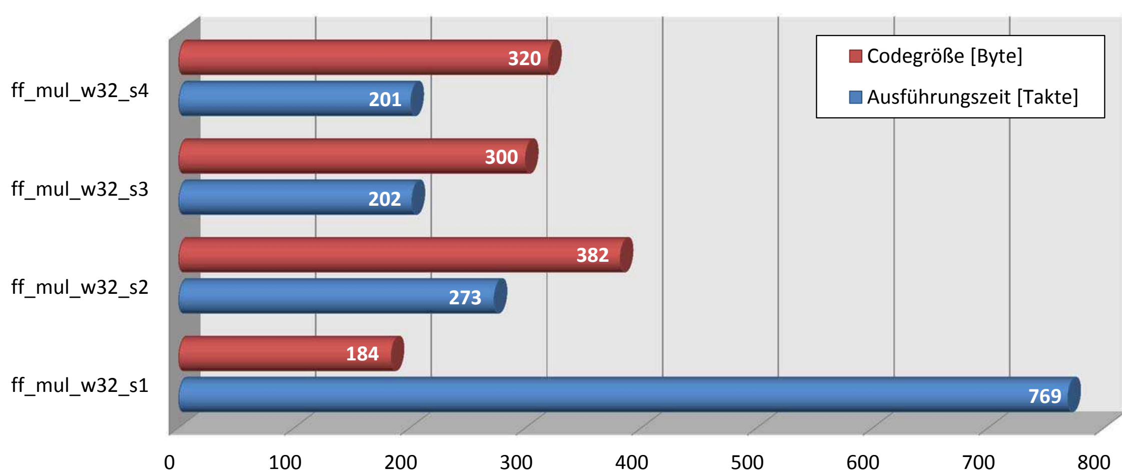


Abbildung 5.2: Funktionen zur Binärkörper-Multiplikation auf 32-Bit-Wort-Ebene

als die Funktion ff_mul_w32_s1. Die Erhöhung der Blockgröße auf $s = 4$ erzielt keine deutliche Verbesserung, da sich die Ausführungszeit nur um einen Takt verkürzt, die Codegröße jedoch um 20 Byte zunimmt.

5.1.3 Evaluierung auf Körper-Ebene

Für die folgenden Analysen wird die Funktion ff_mul_w32_s3 als Standardmethode zur Binärkörper-Multiplikation auf 32-Bit-Wort-Ebene verwendet. Hierauf aufbauend werden nun die in Abschnitt 3 vorgestellten Algorithmen, von der Arithmetik in endlichen Körpern über die Punktoperationen (*Punktaddition* bzw. *Punktverdopplung*) bis hin zur Skalarmultiplikation, untersucht (vgl. Abbildung 3.6) . Die Ergebnisse von verschiedenen Binärkörpergrößen sind in Tabelle 5.1 dargestellt. Die Binärkörper und deren Parameter wurden entsprechend den standardisierten Sicherheitsrichtlinien ausgewählt (siehe Anhang A). Die Codegröße wurde jeweils über alle untersuchten Körpergrößen gemittelt.

Wie bereits zu Beginn dieser Arbeit erwähnt wurde, repräsentiert die Addition im Binärkörper die einfachste Operationen innerhalb der Körperarithmetik. Wie in Tabelle 5.1 zu erkennen ist, benötigt die entsprechende Funktion FF_Add im Durchschnitt lediglich 28 Byte an Programmspeicher und ist in vergleichbar wenigen Takten ausgeführt. Weitaus interessanter gestaltet sich die Auswahl einer geeigneten Funktion zur Multiplikation im Binärkörper $GF(2^n)$. Alle untersuchten Varianten basieren auf der vorgestellten Funktion ff_mul_w32_s3 zur Binärkörper-Multiplikation auf Wort-Ebene. Dennoch lassen sich große Unterschiede in der Ausführungszeit und der Codegröße erkennen. Die Funktion FF_Mul_Karatsuba zur Binärkörper-Multiplikation auf Basis der Karatsuba-Methode (vgl. Algorithmus 6) bietet die kürzeste Ausführungszeit bei gleichzeitig moderater Codegröße. Mit Hilfe von Algorithmus 8 kann die Quadrierung im Binärkörper (FF_Square) deutlich gegenüber der Binärkörper-Multiplikation mit gleichen Operanden beschleunigt werden. Die Kosten hierfür betragen allerdings 238 Byte an zusätzlichem Speicherplatz. Die Funktion FF_Reduce zur Modulo-Reduktion (vgl. Abschnitt 3.3.4) ist notwendiger Bestandteil der Multiplikation und der Quadrierung im Binärkörper. Der entsprechende Algorithmus ist jedoch als eigenständige Funktion implementiert und wird innerhalb der verschiedenen Multiplikations- bzw. Quadrierungsfunktionen

	Funktion	Verweis	Code-größe [Byte]	Ausführungszeit [Takte] für verschiedene Binärkörper $GF(2^n)$ n = 163	n = 233	n = 283	n = 409	n = 571
Körper-arithmetik	FF_Add	*Alg.* 3	28	108	140	156	224	304
	FF_Mul_ShiftXor	*Alg.* 4	162	30 629	58 015	83 105	171 490	324 948
	FF_Mul_Classical	*Alg.* 5	495	17 554	21 706	48 813	79 238	194 835
	FF_Mul_Karatsuba	*Alg.* 6	537	8 887	11 536	18 448	30 691	55 922
	FF_Square	*Alg.* 8	238	850	1 131	1 288	1 852	2 541
	FF_Reduce	*Alg.* 9	169	239	330	389	543	724
	FF_Inversion	*Alg.* 10	621	218 312	377 595	566 440	1 065 461	2 125 639
Punkt-operation	PNT_Add_A	*Gl.* 2.10	137	238 437	403 712	606 816	1 131 820	2 244 755
	PNT_Double_A		125	238 239	403 455	606 836	1 131 859	2 244 850
	PNT_Add_P	*Gl.* 3.6	76	36 780	47 789	75 732	125 747	227 865
	PNT_Double_P	*Gl.* 3.5	68	21 442	27 947	42 529	69 589	123 343
Skalar-multiplikation	EC_Mul_A (BC)	*Alg.* 1	310	718 505	1 216 015	1 827 441	3 404 572	6 746 619
	EC_Mul_A (TC)			38 846 361	94 052 991	171 776 617	462 945 451	1 281 847 508
	EC_Mul_A (WC)			77 450 841	187 698 039	342 936 037	924 753 619	2 561 437 836
	EC_Mul_P (BC)	*Alg.* 2	326 + 352	374 908	581 799	885 391	1 590 976	3 069 512
	EC_Mul_P (TC)			5 045 988	9 310 428	17 487 935	41 330 388	103 106 121
	EC_Mul_P (WC)			9 775 521	18 114 688	34 210 739	81 269 000	203 495 177

Tabelle 5.1: Codegröße und Ausführungszeiten von Funktionen zur Skalarmultiplikation für verschiedene Binärkörpergrößen

aufgerufen. Dieses hat den Vorteil, dass die Codegröße der Modulo-Reduktion nur einfach in die Gesamtbilanz des Speicherbedarfs eingeht, da die Funktion `FF_Reduce` gemeinsam von dem Multiplikationsalgorithmus als auch von dem Quadrierungsalgorithmus benutzt wird. Die Ausführungszeit für den Funktionsaufruf einer Modulo-Reduktion ist allerdings bereits in den Ausführungszeiten der Multiplikations- und Quadrierungsfunktionen enthalten. Die komplexe Funktion `FF_Inversion` zur Bestimmung der Inversen nach Algorithmus 10 benötigt erwartungsgemäß den größten Speicherplatz und die meisten Prozessortakte innerhalb der Körperarithmetik.

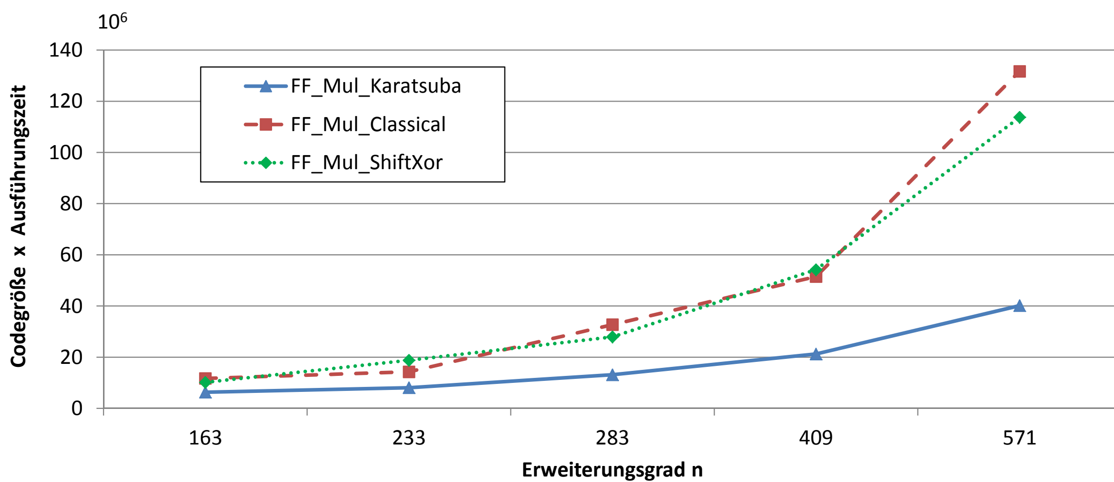


Abbildung 5.3: Bewertung der Multiplikationsfunktionen im Binärkörper $GF(2^n)$

Wie in Kapitel 3.2 erläutert wurde, kann durch eine geeignete Koordinatentransformation die Anzahl der benötigten Invertierungen stark reduziert werden. Die Ausführungszeit einer Multiplikation im entsprechenden Binärkörper ist daher dominierend für die Performanz der Skalarmultiplikation. Aus diesem Grund wird die Funktion `FF_Mul_Karatsuba` im Folgenden als Standardfunktion zur Multiplikation im Binärkörper (`FF_Mul`) verwendet. Der zugrunde liegende Karatsuba-Algorithmus benötigt zwar mehr Speicherplatz als die klassische Variante (`FF_Mul_Classical`) und die naive Variante (`FF_Mul_ShiftXor`), die entscheidende Ausführungszeit ist allerdings zwischen Faktor 1,9 und 5,8 besser. Abbildung 5.3 verdeutlicht diese Auswahl, indem für die verschiedenen Multiplikationsfunktionen im entsprechenden Binärkörper $GF(2^n)$ jeweils das Produkt aus Codegröße und Ausführungszeit aufgetragen ist.

Unter Verwendung der Funktion `FF_Mul_Karatsuba` zur Multiplikation im Binärkörper werden die Punktoperationen, d. h. Punktaddition (`PNT_Add`) und Punktverdopplung (`PNT_Double`), sowie die Skalarmultiplikation (`EC_Mul`) jeweils für affine (`_A`) und projektive (`_P`) Koordinatensysteme untersucht. Algorithmen, welche das projektive Koordinatensystem benutzen, basieren, wie in Abschnitt 3.2 dargestellt, auf der Transformation nach López-Dahab [LD99]. Die Codegröße der Funktion `EC_Mul_P` in Tabelle 5.1 setzt sich aus 326 Byte zur Berechnung der Skalarmultiplikation sowie 352 Byte für die Koordinatentransformation zusammen. Da die Ausführungszeit der Skalarmultiplikation $k\mathcal{P}$ im Binärkörper $GF(2^n)$ auf Basis der Montgomery-Leiter (vgl. Abschnitt 3.1) von der Position der führenden Eins hinsichtlich der binären Darstellung von k abhängt, wurden diesbezüglich die folgenden Fallunterscheidungen definiert:

- bester Fall (BC, engl. Best Case): $k = 2^1$
- typischer Fall (TC, engl. Typical Case): $k = 2^{(n-1)/2}$
- schlechtester Fall (WC, engl. Worst Case): $k = 2^{(n-1)}$

Im besten Fall ($k = 2$) entspricht die Skalarmultiplikation $k\mathcal{P}$ einer Punktverdopplung, so dass nur eine Stufe der Montgomery-Leiter berechnet werden muss. Im schlechtesten Fall ($k = 2^{(n-1)}$) entspricht die führende Eins in der binären Darstellung von k dem hochwertigen Bit, so dass die maximale Anzahl an Punktoperationen berechnet werden muss. Konsequenterweise liegt der typische Fall mit $k = 2^{(n-1)/2}$ genau in der Mitte.

Wie in Tabelle 5.1 zu erkennen ist, macht sich der Vorteil der projektiven Koordinaten deutlich in der benötigten Ausführungszeit bemerkbar. Im typischen Fall (`TC`) ergibt sich ein durchschnittlicher Beschleunigungsfaktor von 10,25 bei der Skalarmultiplikation. Addiert man die Codegrößen der Punktoperations- und Skalarmultiplikationsfunktionen jeweils für die affine und projektive Variante, so steht dem Geschwindigkeitsvorteil ein um Faktor 1,44 größerer Speicherplatzbedarf gegenüber.

Aufgrund der weitaus besseren Performanz bei moderater Codevergrößerung werden im Folgenden ausschließlich die Algorithmen aus Tabelle 5.1 verwendet, welche das projektive Koordinatensystem verwenden. Diese lassen sich teilweise, wieder auf Kosten des Speicherplatzbedarf, weiter optimieren. Tabelle 5.2 zeigt die

Funktion	Code-größe [Byte]	Ausführungszeit [Takte] für verschiedene Binärkörper $GF(2^n)$ n = 163	n = 233	n = 283	n = 409	n = 571
FF_Add	83	72	92	102	143	218
FF_Mul	1 373	4 465	6 683	9 097	20 561	27 872
FF_Squ	405	681	937	1 143	1 624	2 425
FF_Mod	345	164	188	303	397	688
FF_Inv	621	218 312	377 595	566 440	1 065 461	2 125 639
PNT_Add	76	18 801	28 062	38 010	84 722	115 209
PNT_Dbl	68	11 855	17 398	23 102	48 216	66 521
EC_Mul$_{BC}$	505 + 352	301 924	501 709	733 230	1 425 698	2 617 702
EC_Mul$_{TC}$		2 762 599	5 748 600	9 327 039	28 539 284	54 481 802
EC_Mul$_{WC}$		5 254 865	11 040 839	17 982 926	55 785 988	106 526 562

Tabelle 5.2: Codegröße und Ausführungszeiten von optimierten Funktionen zur Skalarmultiplikation für verschiedene Binärkörpergrößen

Ergebnisse der zusätzlichen Optimierungsschritte. Als erstes wurde die Compiler-Option `-funroll_loops` benutzt, was den Übersetzer veranlasst, Schleifen in den Algorithmen abzurollen, d. h. sequentiell den Schleifen-Körper mehrfach zu instanziieren. Hierdurch vereinfacht sich der Kontrollfluss, da viele Sprungbefehle sowie die Berechnung von Schleifen-Parametern entfallen, was die Ausführungszeit auf Kosten der Codegröße verbessert [Mar07]. Des Weiteren wurde für die Funktion `FF_Mul` die rekursive Implementierung des Karatsuba-Algorithmus durch eine iterative Variante ersetzt. Anstatt der rekursiven Schleife mit Abbruchkriterium (vgl. Alg. 6) werden separate Funktionen für die verschiedenen Polynomlängen bzw. Datenwortbreiten implementiert (vgl. Anhang B.1). Neben einer Halbierung der Datenwortbreite, wie in Abbildung 3.3 dargestellt, wird nun auch eine Drittelung mit Hilfe der so genannten 3-Segment-Karatsuba-Methode [Bla85][GS06] umgesetzt. Durch iterative Kombination von Karatsuba-Halbierung (K_2) und Karatsuba-Drittelung (K_3) können die vorgegebenen Polynomlängen besser auf ein Vielfaches der Datenwortbreite des Mikroprozessors abgebildet werden. Abbildung 5.4 zeigt die Anwendung des gemischten, iterativen Karatsuba-Verfahrens. Die unterstrichenen Werte entsprechen den verwendeten Vielfachen der Prozessor-Datenwortbreite, um die Binärkörper aus

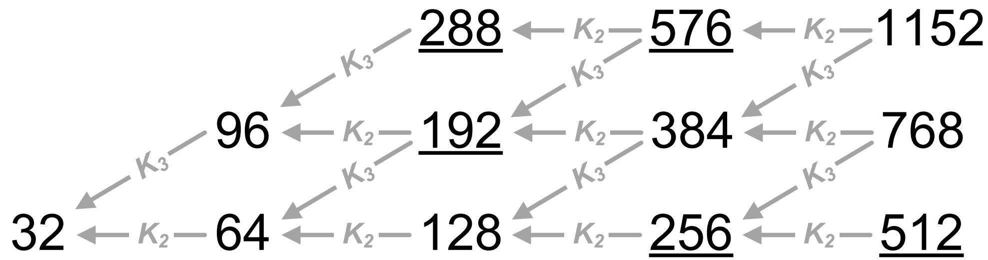


Abbildung 5.4: Iteratives Karatsuba-Verfahren für ausgewählte Binärkörper

Tabelle 5.2 effizient abzubilden. So können beispielsweise Multiplikationen im Binärkörper $GF(2^{283})$ mit Hilfe von zwei iterativen Karatsuba-Drittelungen (K_3) von 288 Bit auf die Prozessor-Datenwortbreite von 32 Bit umgesetzt werden. Im Vergleich dazu wären beim rekursiven Halbierungs-Ansatz (K_2) vier Schleifendurchläufe notwendig, um von 512 Bit auf 32 Bit zu gelangen. Zudem wäre diese Umsetzung sehr ineffizient, da fast die Hälfte ($512-283 = 229$ Bit) der Datenstruktur ungenutzt Speicher belegt.

Wie der Vergleich zwischen Tabelle 5.1 und Tabelle 5.2 zeigt, kann durch die zusätzlichen Optimierungsschritte die Ausführungszeit der einzelnen Funktionen deutlich gesenkt werden. Der typische Fall der Skalarmultiplikation ($\texttt{EC_Mul}_{\texttt{TC}}$) kann so erneut um den Faktor 1,73 beschleunigt werden. Die akkumulierte Codegröße aller Funktionen steigt von 2415 Byte um den Faktor 1,57 auf 3802 Byte an. Die Ausführungszeit der Skalarmultiplikation wird annähernd um den gleichen Faktor beschleunigt, wie der dazu erforderliche Speicherplatzbedarf ansteigt. An diesem Punkt wird deshalb die Optimierung auf Software-Ebene beendet, da eine weitere Reduktion der Ausführungszeit nur noch durch eine unverhältnismäßige Erhöhung der Codegröße erreicht werden kann. Stattdessen wird im folgenden Kapitel die Optimierung von Hardware-Beschleunigern erörtert. Anschließend werden die erarbeiteten Software- und Hardware-Umsetzungen auf unterschiedlichen Ebenen miteinander kombiniert und die Ressourceneffizienz bewertet.

5.2 Hardware-Evaluierung

Wie im vorherigen Abschnitt gezeigt wurde, spielt die Multiplikation von Elementen eines endlichen Körpers eine entscheidende Rolle für die Leistungsfähigkeit eines Kryptosystems. Da die Binärkörper-Multiplikation eine sehr rechenintensive und häufig verwendete Operation in der Kryptographie mit elliptischen Kurven darstellt, wirkt sich die gewählte Multiplikationsmethode direkt auf die Effizienz des gesamten Systems aus. Im folgenden Abschnitt werden daher verschiedene Varianten einer Hardware-Implementierung zur Multiplikation von Polynomen über dem Binärkörper $GF(2^n)$ analysiert und so miteinander kombiniert, dass eine möglichst optimale Grundlage für die aufbauenden Software-Algorithmen zur Verfügung steht. Ferner wird ein parametrisierbarer Codegenerator vorgestellt, mit dem sich effiziente Schaltungen für die Binärkörper-Multiplikation automatisch erzeugen lassen.

5.2.1 Klassische Multiplikation

In Gleichung 2.1 wurde bereits die verwendete Darstellung von Elementen des Binärkörpers $GF(2^n)$ als Polynombasis beschrieben. Die Multiplikation von zwei Polynomen $A, B \in GF(2^n)$ vom Grad $n-1$ mit $A = \sum_{i=0}^{n-1} a_i x^i$ und $B = \sum_{i=0}^{n-1} b_i x^i$ ergibt das Produkt $C = \sum_{i=0}^{2n-2} c_i x^i$ vom Grad $2n-2$. Die Koeffizienten c_i können hierbei mittels der klassischen Multiplikationsmethode wie folgt berechnet werden:

$$\begin{aligned} c_0 &= a_0 b_0 \\ c_1 &= a_0 b_1 + a_1 b_0 \\ &\vdots \\ c_{n-1} &= a_0 b_{n-1} + a_1 b_{n-2} + \ldots + a_{n-2} b_1 + a_{n-1} b_0 \\ &\vdots \\ c_{2n-3} &= a_{n-2} b_{n-1} + a_{n-1} b_{n-2} \\ c_{2n-2} &= a_{n-1} b_{n-1} \end{aligned} \tag{5.1}$$

Die Berechnung der Koeffizienten nach Gleichung 5.1 erfordert n^2 Multiplikationen und $(n-1)^2$ Additionen im Grundkörper $GF(2)$ [GBT+03]. Insgesamt werden bei

der klassischen Multiplikationsmethode also $2(n^2 - n) + 1$ Operationen benötigt, was einer asymptotischen Komplexität von $O(n^2)$ entspricht. Da diese Methode in den meisten Schulen unterrichtet wird, ist die klassische Multiplikation auch als Schulbuch-Multiplikationsmethode bekannt.

Das Produkt zweier Polynome vom Grad 1 ($n = 2$) kann nach der klassischen Methode mit 4 Multiplikationen und einer Addition wie folgt berechnet werden:

$$(a_1x + a_0)(b_1x + b_0) = a_1b_1x^2 + (a_1b_0 + a_0b_1)x + a_0b_0 \tag{5.2}$$

Für den binären Grundkörper $GF(2)$ entspricht die Multiplikation einer logischen `AND`-Verknüpfung und die Addition einer logischen `XOR`-Verknüpfung (vgl. Tabelle 2.1). Aus Sicht der Hardware werden daher bei der klassischen Multiplikationsmethode n^2 `AND`-Gatter und $(n-1)^2$ `XOR`-Gatter mit jeweils zwei Eingangssignalen benötigt. Abbildung 5.5 zeigt eine Schaltung zum Multiplizieren von Polynomen vom Grad 3 ($n = 4$) nach der klassischen Methode. Die `AND`-Gatter zur Multiplikation sind durch das Symbol $\otimes$ und die `XOR`-Gatter zur Addition durch das Symbol $\oplus$ dargestellt. Die vier grau hinterlegten Bereiche stellen jeweils die Schaltung für eine Multiplikation von Polynomen vom Grad 1 dar und entsprechen somit Gleichung 5.2. Wie die Abbildung 5.5 verdeutlicht, wurde die Multiplikation von Polynomen der Länge $n = 4$ in vier Teilmultiplikationen von Polynomen der

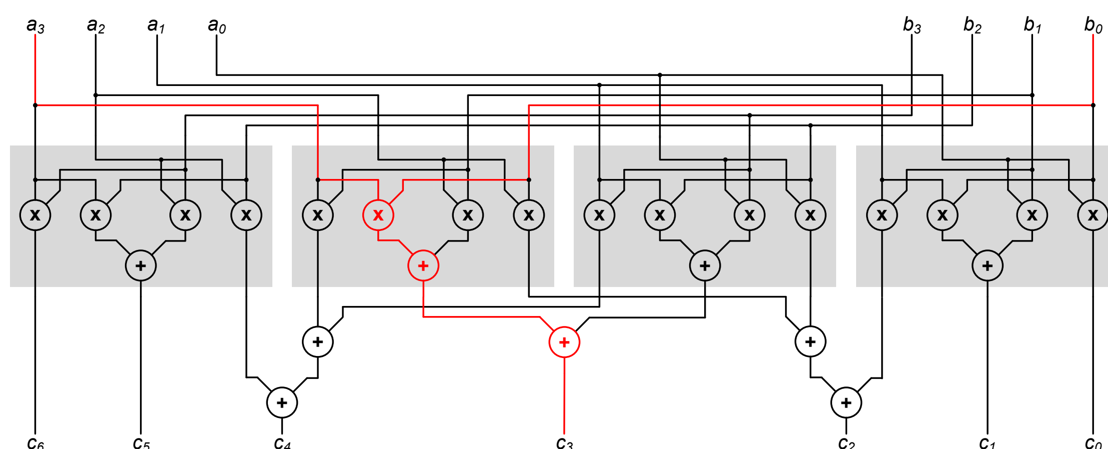


Abbildung 5.5: Kombinatorische Logik zur Multiplikation von Polynomen vom Grad 3 nach der klassischen Methode

Länge $m = 2$ zerlegt. Dieses Verfahren kann rekursive beliebig fortgesetzt werden, um größere Polynome nach der klassischen Methode kombinatorisch zu multiplizieren. Bei jeder Verdopplung der Polynomlänge vervierfacht sich die Anzahl der grau hinterlegten Grundblöcke. Die Multiplikationen können alle parallel ausgeführt werden, so dass die Verzögerungszeit nur von dem Aufsummieren der Teilprodukte für gleiche Koeffizienten abhängt. Diese Addierer-Baumstruktur ist durch die XOR-Gatter außerhalb der grau hinterlegten Grundblöcke dargestellt. Die Baumtiefe für Polynome der Länge n beträgt $\log_2 n$, so dass sich eine Gesamtverzögerung T für eine Multiplikation von Polynomen der Länge n nach der klassischen Methode von $T = T_{\texttt{AND}} + \lceil \log_2 n \rceil T_{\texttt{XOR}}$ ergibt. Die Verzögerungszeit T bildet den so genannten *kritischen Pfad*. Dieser legt, in Abhängigkeit der gewählten Zieltechnologie, die maximal erreichbare Taktfrequenz der Schaltung, d. h. deren Rechenleistung, fest. Der kritische Pfad in Abbildung 5.5 ist rot hervorgehoben.

5.2.2 Karatsuba-Multiplikation

Um die Komplexität der klassischen Multiplikation zu reduzieren, kann die Karatsuba-Methode [KO63] angewandt werden. Die Karatsuba-Methode verringert die Anzahl der benötigten Multiplikationen auf Kosten der Anzahl an Additionen, indem bereits berechnete Teilprodukte wiederverwendet werden. Folgendes Beispiel veranschaulicht die Funktionsweise der Karatsuba-Methode für eine Polynommultiplikation vom Grad 1 ($n = 2$):

$$(a_1x + a_0)(b_1x + b_0) = a_1b_1x^2 + ((a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0)x + a_0b_0 \quad (5.3)$$

Beim Vergleich mit der klassischen Methode (Gleichung 5.2) erkennt man, dass sich die Berechnung des mittleren Koeffizienten c_1 unterscheidet. Anstatt die Koeffizienten der Eingangspolynome (a_1, a_0, b_1, b_0) kreuzweise zu multiplizieren und die Summe der Produkte zu bilden, werden nun die Koeffizienten der Polynome A, B addiert und deren Summen multipliziert. Von dem Produkt werden anschließend die ohnehin für die Berechnung benötigten Teilprodukte a_1b_1 und a_0b_0 subtrahiert. Da für den Binärkörper $GF(2)$ die Subtraktion identisch mit der Addition ist (vgl. Gleichung 2.2), können beide Operationen weiterhin durch ein XOR-Gatter

realisiert werden. Im Gegensatz zur klassischen Methode benötigt die Karatsuba-Methode nur drei anstatt vier Multiplikationen. Die Zahl der Additionen steigt dagegen von einer bei der klassischen Methode auf vier bei der Karatsuba-Methode. Allgemein gilt, dass für die Multiplikation von Polynomen vom Grad $n-1$ (mit $n = 2^i \quad \forall i \in \mathbb{N}$) nach der Karatsuba-Methode $n^{\log_2 3}$ Multiplikationen und $6n^{\log_2 3} - 8n + 2$ Additionen im Grundkörper $GF(2)$ benötigt werden [WP06]. Die Anzahl der Additionen wächst asymptotisch langsamer als bei der klassischen Methode, so dass ab einer Polynomlänge von $n = 16$ die Karatsuba-Methode weniger Bit-Operationen (`AND`- bzw. `XOR`-Gatter) benötigt als die klassische Methode. Ferner kann auch die Karatsuba-Methode rekursiv angewandt werden. Abbildung 5.6 zeigt die kombinatorische Logik zum Multiplizieren von Polynomen vom Grad 3. Ähnlich wie in Abbildung 5.5, werden die Eingangspolynome der Länge $n = 4$ in kleinere

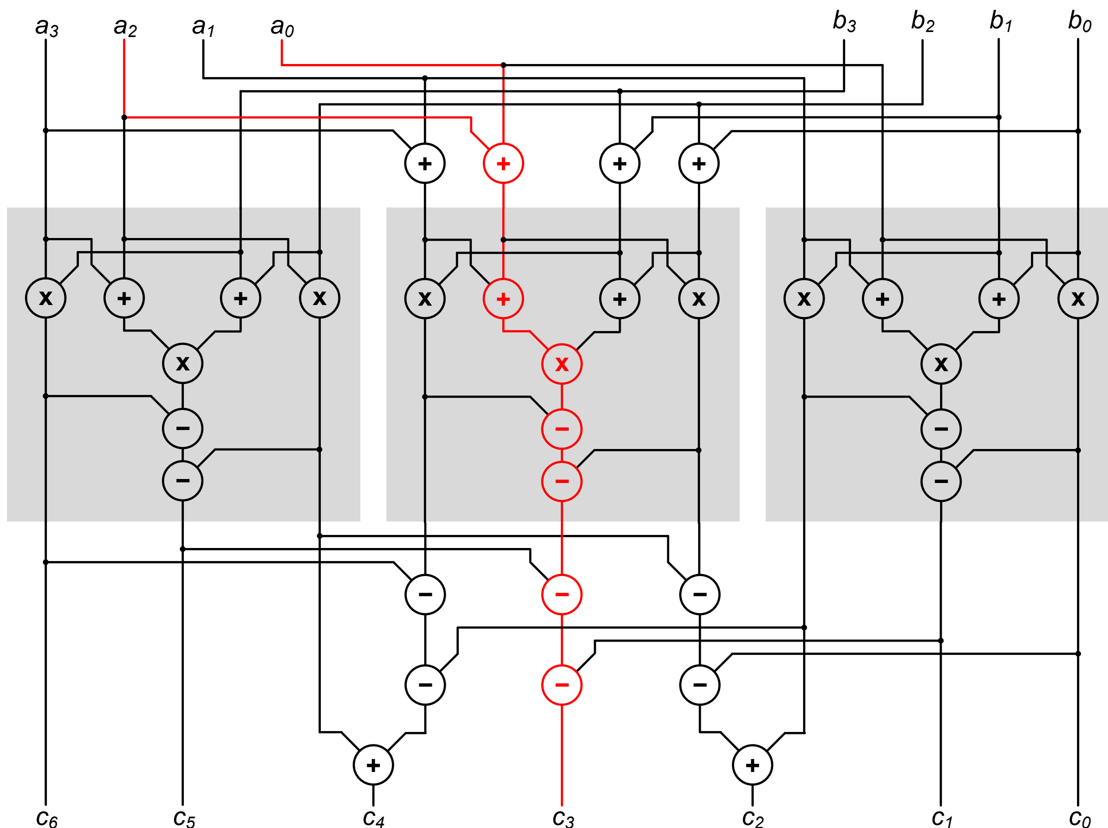


Abbildung 5.6: Kombinatorische Logik zur Multiplikation von Polynomen vom Grad 3 nach der Karatsuba-Methode

Teilpolynome der Länge $m = 2$ zerlegt. Hieraus kann dann nach der Karatsuba-Methode mit drei Teilmultiplikationen und den entsprechenden Addiererstufen das Produkt gebildet werden. Die drei Teilmultiplikationen der Länge $m = 2$ sind in Abbildung 5.6 grau hinterlegt und entsprechen Gleichung 5.3. Genau wie die klassische Multiplikation wird auch die Karatsuba-Methode von der Anzahl an Multiplikationen (AND-Gattern) dominiert. Allerdings beträgt die asymptotische Komplexität der Karatsuba-Multiplikation $O(n^{\log_2 3})$ und ist damit der klassischen Methode überlegen [GG03]. Die Reduzierung der Bit-Operationen entspricht einer Optimierung der Chipfläche, welche auf Kosten der maximalen Verzögerungszeit, also der Rechenleistung, geht. Denn anders als bei der klassischen Methode können nicht alle Koeffizienten zeitgleich multipliziert werden. Wie auch in Abbildung 5.6 deutlich wird, benötigt die Karatsuba-Methode zusätzliche Eingangs- bzw. Ausgangsaddierer zur Berechnung, so dass sich die Gesamtverzögerung auf $T = T_{\texttt{AND}} + 3\lceil \log_2 n \rceil T_{\texttt{XOR}}$ beläuft. Der kritische Pfad in Abbildung 5.6 ist wie zuvor rot markiert.

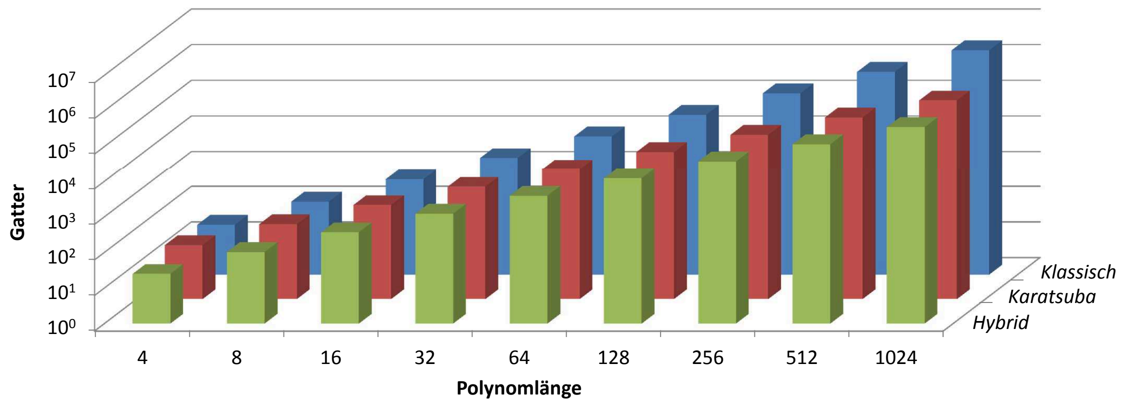
5.2.3 Hybride Multiplikation

Die hybride Multiplikationsmethode kombiniert die Vorteile der klassischen Multiplikation mit denen der Karatsuba-Methode. Für Polynome der Länge $n = 2^i$ mit $i \in \mathbb{N}$ wird zunächst die Karatsuba-Methode rekursiv angewandt. Dabei halbiert sich pro Rekursion die Länge der benötigten Polynommultiplizierer. Ab einer bestimmten Rekursionstiefe $r \in \mathbb{N}$ werden die Polynome mit Länge $m = n/2^r$ dann mit Hilfe der klassischen Methode multipliziert. Dieses Verfahren benötigt die folgende Anzahl an Bit-Operationen [RHK03]:

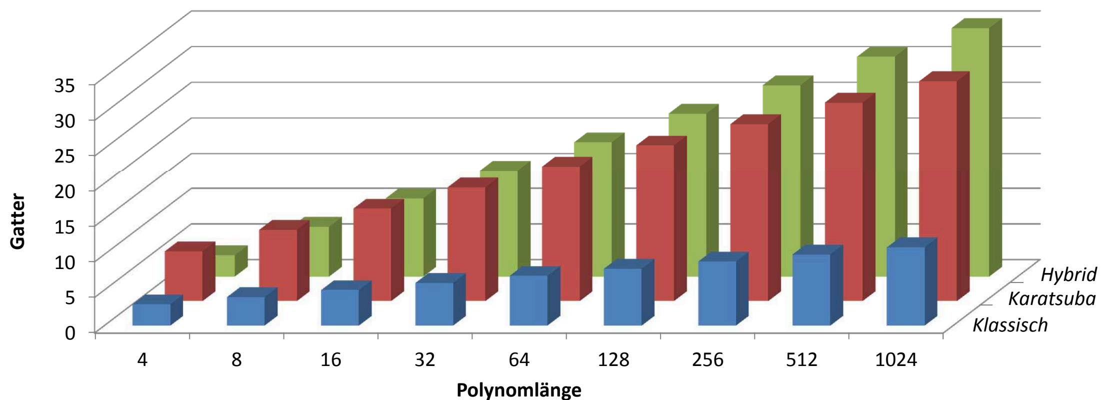
$$\begin{aligned} \texttt{AND}\text{-Gatter}: & \left(\frac{n}{m}\right)^{\log_2 3} m^2 \\ \texttt{XOR}\text{-Gatter}: & \left(\frac{n}{m}\right)^{\log_2 3} (m^2 + 6m - 1) - 8n + 2 \end{aligned} \tag{5.4}$$

Die Gesamtverzögerung lässt sich durch $T = T_{\texttt{AND}} + (4 \log_2 n - 3 \log_2 m)\, T_{\texttt{XOR}}$ berechnen. Weiterhin wurde in [GS06] gezeigt, dass eine Polynomlänge von $m = 4$ ein optimales Abbruchkriterium für die Rekursionstiefe darstellt.

Abbildung 5.7 zeigt einen Vergleich der vorstellten Hardware-Implementierungen zur Multiplikation von Polynomen über dem Binärkörper $GF(2^n)$. Die Polynom-



(a) Anzahl der Bit-Operationen



(b) Tiefe des kritischen Pfades

Abbildung 5.7: Vergleich der unterschiedlichen Multiplikationsmethoden hinsichtlich der Ressourcen (a) Chipfläche und (b) Rechenleistung

länge n ist jeweils über der Abszissenachse aufgetragen. Für die hybride Multiplikationsmethode wurde, wie zuvor beschrieben, $m = 4$ als Abbruchkriterium der Rekursion verwendet. In Abbildung 5.7a ist die Anzahl an Bit-Operationen, d. h. die Summe aller zur Implementierung benötigten `AND`- und `XOR`-Gatter mit jeweils zwei Eingangssignalen, logarithmisch aufgetragen. Hierdurch lässt sich die Chipfläche der verschiedenen Multiplikationsmethoden miteinander vergleichen. Die klassische Methode benötigt aufgrund ihrer quadratischen Komplexität $O(n^2)$ bei wachsender Polynomlänge sehr viele Gatter zur Realisierung des Multiplizierers. Ab einer Polynomlänge von $n = 16$ ist die Karatsuba-Methode der klassischen Methode hinsichtlich der erforderlichen Chipfläche überlegen. Wie aus dem Diagramm deutlich erkennbar ist, benötigt die hybride Multiplikationsmethode für alle darge-

stellten Polynomlängen die wenigsten Ressourcen. Die Optimierung der Chipfläche wird allerdings auf Kosten der maximalen Taktfrequenz erkauft. In Abbildung 5.7b ist die Anzahl der Gatter dargestellt, welche den kritischen Pfad für die jeweilige Polynomlänge bilden. Die klassische Implementierungsvariante erreicht die höchste Taktfrequenz, da alle Teilmultiplikationen parallel ausgeführt werden können. Der kritische Pfad der Karatsuba-Methode ist für alle Polynomlängen mehr als doppelt so lang. Ab einer Polynomlänge von $n = 64$ weist die hybride Multiplikationsmethode den längsten kritischen Pfad auf. Allerdings muss hierbei beachtet werden, dass in Abbildung 5.7b keine logarithmische Skalierung der Ordinatenachse vorliegt. Die absolute Differenz beim kritischen Pfad ist daher viel kleiner als bei der jeweiligen Chipfläche. Selbst bei einer Polynomlänge von $n = 1024$ besitzt der kritische Pfad des hybriden Multiplizierers nur 24 Gatterstufen mehr als die klassische Implementierung, während die Chipfläche um 1742440 Gatter anwächst und sich damit fast versechsfacht. Sollte die Multipliziererschaltung den kritischen Pfad innerhalb eines synchronen Systementwurfes darstellen, kann durch Einfügen entsprechender Pipeline-Stufen die Verzögerungszeit reduziert werden, so dass die maximale Taktfrequenz des Gesamtsystems nicht beeinflusst wird.

Im Folgenden soll die Ressourceneffizienz der verschiedenen Multiplikationsmethoden durch die in Abschnitt 4.3.4 vorstellte Bewertungsmetrik untersucht werden. Gemäß Gleichung 4.3 werden dabei die Ressourcen Chipfläche, Verlustleistung und Ausführungszeit bewertet. Die Chipfläche A ist durch die Anzahl der zur Implementierung benötigten Bit-Operationen (`AND`- bzw. `XOR`-Gatter) festgelegt. Zur Abschätzung der Verlustleistung P wird zu diesem Zeitpunkt angenommen, dass zur Berechnung der Multiplikation die Hälfte aller verwendeten Gatter ihren Zustand ändern, d. h. $P \propto A/2$. Die Rechenleistung bzw. Ausführungszeit T ist durch die Anzahl der Bit-Operationen im kritischen Pfad definiert. Da noch kein konkretes Anwendungsszenario vorliegt, wird keine spezielle Gewichtung der Ressourcen durch die Exponenten c_A, c_P und c_T durchgeführt, d. h. $c_A = c_P = c_T = 1$. Um die Ressourceneffizienz der verschiedenen Hardware-Multiplizierer qualitativ miteinander vergleichen zu können, wurden die Chipfläche und die Ausführungszeit jeweils am Maximum normiert. Abbildung 5.8 zeigt die Ressourceneffizienz der vorgestellten Hardware-Multiplizierer für die verschiedenen Polynomlängen n. Es wird deutlich, dass die hybride Multiplikationsmethode ab einer Polynomlänge von $n = 32$ die be-

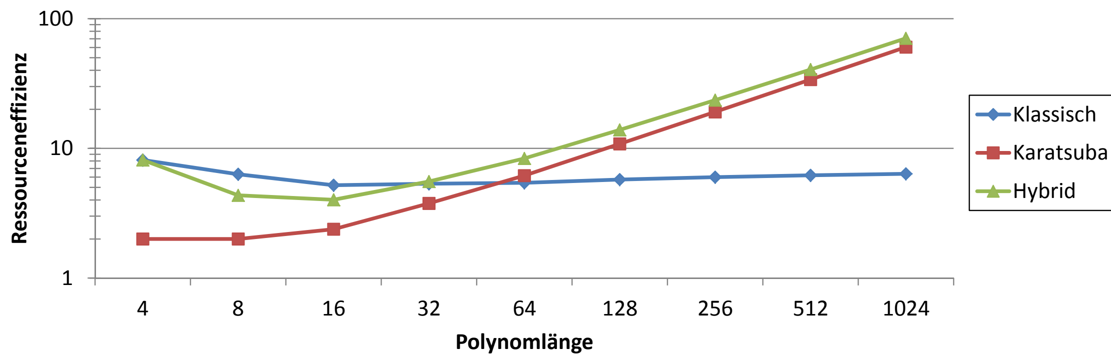


Abbildung 5.8: Ressourceneffizienz der vorgestellten Hardware-Multiplizierer

ste Ressourceneffizienz aufweist. Für kürzere Polynomlängen ist die klassische Implementierungsvariante aufgrund der kurzen Ausführungszeit (vgl. Abbildung 5.7b) im Vorteil. Je nach zugrundeliegendem Anwendungsszenario (siehe Abschnitt 4.1) würde sich der Schnittpunkt zwischen klassischer und hybrider Multiplikationsmethode verschieben. Beim Anwendungsszenario *Chipkarte*, wo die Chipfläche und Verlustleistung im Vordergrund stehen, würde auch bei kleineren Polynomlängen ($n < 32$) die hybride Variante eine bessere Ressourceneffizienz erzielen als die klassische Methode. Umgekehrt würde beim Anwendungsszenario *Sicherheitsserver*, wo es vorrangig um die Rechenleistung geht, die klassische Methode auch bei größeren Polynomlängen ($n \geq 32$) gegenüber der hybriden Multiplikationsmethode eine höhere Ressourceneffizienz erlangen.

5.2.4 Codegenerator

Der manuelle Schaltungsentwurf von Multiplizierern wird mit wachsender Polynomlänge hinreichend komplex. Um Implementierungsfehler auszuschließen und um die Entwurfszeit zu verkürzen, wurde im Rahmen des DFG-Projektes "Krypto-Hardware" [PSRG08] ein Programm entwickelt, das es ermöglicht, Multipliziererschaltungen automatisch zu generieren. Ähnlich wie in Abbildung 5.5 und Abbildung 5.6 dargestellt, erzeugt das Programm als Ausgabe eine Strukturbeschreibung von Multiplizierern auf Register-Transfer-Ebene (RTL) in der Hardware-Beschreibungssprache VHDL. Die graphische Benutzeroberfläche des Codegenerators ist in Abbildung 5.9 dargestellt. Als Eingabeparameter erwartet das Programm die Län-

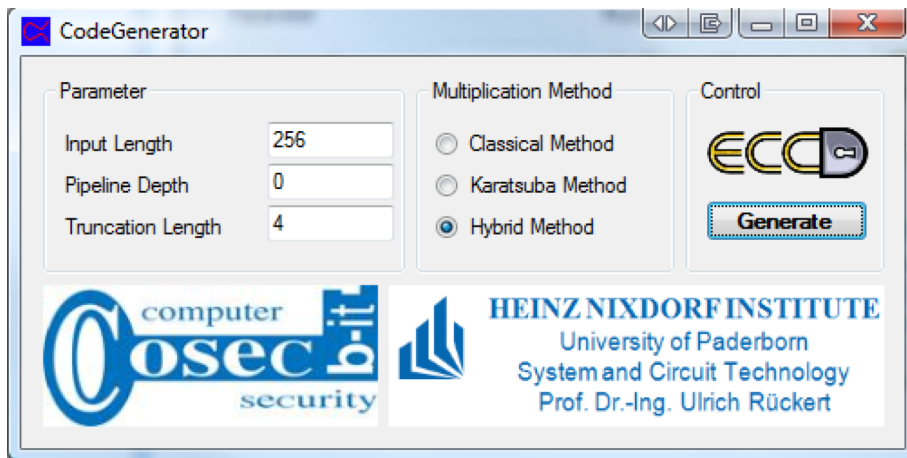
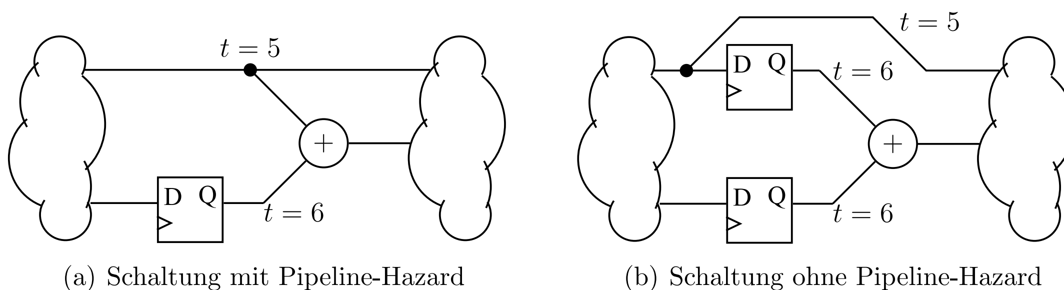


Abbildung 5.9: Graphische Benutzeroberfläche des Codegenerators

ge der Eingangspolynome sowie die Anzahl der Pipeline-Stufen. Weiterhin kann die gewünschte Multiplikationsmethode (*Klassisch*, *Karatsuba* oder *Hybrid*) angegeben werden. Bei Auswahl der hybriden Multiplikationsmethode muss außerdem als Abbruchkriterium der Rekursion die Polynomlänge (engl. Truncation Length) festgelegt werden, ab welcher von der Karatsuba-Methode auf die klassische Methode gewechselt wird.

Durch das Einfügen von Pipeline-Stufen kann der kritische Pfad entsprechend verkürzt werden, um so eine höhere Taktfrequenz zu erreichen. Der Codegenerator platziert dabei die Pipeline-Register so, dass die kritischen Pfade zwischen zwei Pipeline-Stufen möglichst ausgeglichen sind, ohne dabei aber Pipeline-Hazards entstehen zu lassen. Pipeline-Hazards bezeichnen hierbei Konflikte, die entstehen können, wenn die Operanden einer Teilberechnung durch Einfügen von Registern zu



(a) Schaltung mit Pipeline-Hazard

(b) Schaltung ohne Pipeline-Hazard

Abbildung 5.10: Auflösung von Pipeline-Hazards [Sho06]

unterschiedlichen Zeitpunkten vorliegen. Abbildung 5.10 verdeutlicht die Auflösung eines Pipeline-Hazards durch das Einfügen zusätzlicher Register.

Der Codegenerator ist objektorientiert in der Hochsprache C++ geschrieben und verwendet die Multi-Precision-Library NTL [Sho08]. Zur Erzeugung der Schaltungsstruktur verwendet das Programm Binärbäume, welche durch verkettete Listen implementiert sind. Die Knotenpunkte des Baumes repräsentieren entweder eine Multiplikation (AND-Gatter) oder eine Addition (XOR-Gatter). Neben der automatischen Erzeugung des spezifizierten Multiplizierers kann der entwickelte Codegenerator ebenfalls den entsprechenden Modulo-Reduzierer (vgl. Abschnitt 3.3.4) als VHDL-Strukturbeschreibung ausgeben. Die genaue Arbeitsweise des Codegenerators und eine Beschreibung aller Funktionsklassen kann [Sho06] und [PSRG08] entnommen werden.

5.3 Zusammenfassung

Die in Kapitel 3 vorgestellten Algorithmen wurden in der Programmiersprache ANSI-C [ANSI89] als Software-Implementierung umgesetzt und auf dem S-Core-Mikroprozessorsystem (vgl. Abbildung 4.5) ausgeführt. Um die Evaluierung der Software-Implementierungen zu vereinfachen, wurde eine automatisierte Testumgebung entwickelt, mit der die benötigte Ausführungszeit und Codegröße des jeweiligen Algorithmus auf einfache Weise ermittelt werden kann. Mit Hilfe dieser Testumgebung wurden von der Binärkörper-Multiplikation auf 32-Bit-Wort-Ebene bis zur Skalarmultiplikation für verschiedene Binärkörper sämtliche Algorithmen evaluiert. Als Ergebnis entstand eine Sammlung (siehe Tabelle 5.2) an optimierten Funktionen, welche eine effiziente Berechnung der Skalarmultiplikation als Software-Lösung ermöglichen.

Des Weiteren wurden verschiedene Hardware-Implementierungen zur Binärkörper-Multiplikation evaluiert. Hierbei wurden die klassische Methode, die Karatsuba-Methode sowie ein hybrides Verfahren bezüglich ihre Ressourceneffizienz analysiert. Das Ergebnis dieser Evaluierung ist in Abbildung 5.8 zusammengefasst. Als Fazit kann hier festhalten werden, dass die hybride Methode in den meisten Fällen die effizienteste Variante für eine Hardware-Implementierung eines Binärkörper-

Multiplizierers darstellt. Nur in Anwendungsszenarien mit hohen Anforderungen an die maximale Taktfrequenz ist die klassische Implementierungsvariante, aufgrund des kurzen kritischen Pfades, besser geeignet.

Um Fehler bei der zeitaufwendigen, manuellen Erzeugung der Hardware-Beschreibung einer Multipliziererschaltung zu vermeiden, wurde ein Codegenerator entwickelt. Dieses Programm kann komfortabel über eine graphische Benutzeroberfläche parametrisiert werden. So kann auf Knopfdruck eine Strukturbeschreibung der gewünschten Multipliziererschaltungen als synthetisierbarer VHDL-Code generiert werden.

6 Hardware-Software-Kombinationen

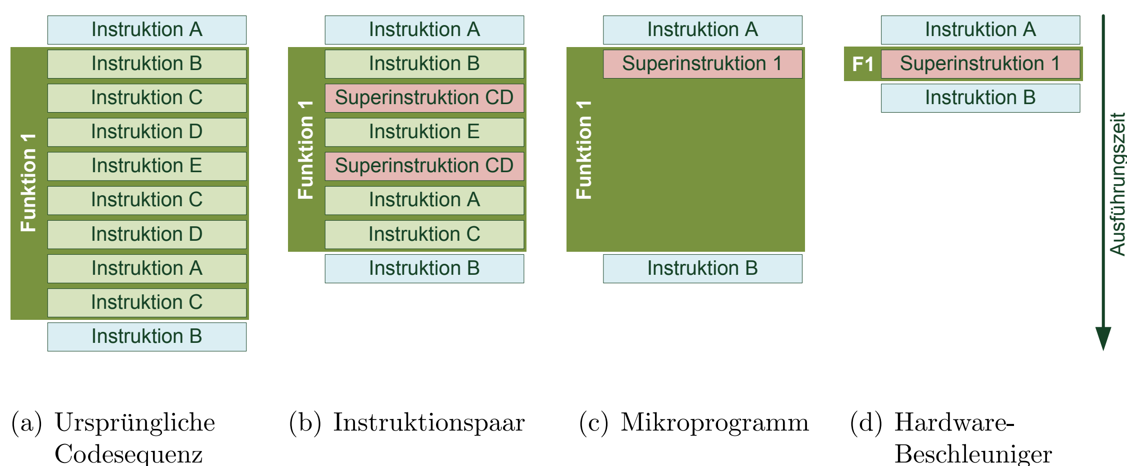
In diesem Kapitel werden mehrere Hardware-Software-Kombination für Kryptographie mit elliptischen Kurven vorgestellt und deren Ressourceneffizienz untersucht. Dabei werden verschiedene Methoden und Werkzeuge zur Optimierung der jeweiligen Hardware-Software-Kombination angewandt. Der Ausgangspunkt für die folgenden Analysen bildet wieder das S-Core-basierte Mikroprozessorsystem nach Abbildung 4.5. Zunächst werden Instruktionssatzerweiterungen als eng gekoppelte Hardware-Beschleuniger untersucht. Hierzu wird ein automatisierter Entwurfsablauf zur Identifikation und Implementierung von geeigneten Instruktionssatzerweiterungen vorgestellt. Es werden drei verschiedene Instruktionssatzerweiterungen realisiert und deren Ressourceneffizienz bewertet. Anschließend wird die Parallelisierbarkeit der eingesetzten Algorithmen untersucht. Dabei wird sowohl ein Prozessorfeld (vgl. Abbildung 4.6) im kombinierten SIMD[1]/MIMD[2]-Betrieb sowie ein Mehrkernprozessor mit VLIW[3]-Architektur untersucht. Schließlich wird ein Coprozessor-Modul entwickelt, welches als dedizierter Hardware-Beschleuniger in einem NoC-basierten Multiprozessorsystem (vgl. Abbildung 4.7) integriert wird. Neben der Ressourceneffizienz des Coprozessor-Moduls wird auch die Performanz des On-Chip-Netzwerkes als Kommunikationskanal betrachtet. In der Zusammenfassung werden die Vor- und Nachteile der vorgestellten Hardware-Software-Kombinationen für unterschiedliche Anwendungsszenarien diskutiert.

[1]Single Instruction Multiple Data
[2]Multiple Instruction Multiple Data
[3]Very Long Instruction Word

6.1 Instruktionssatzerweiterung

Die Instruktionssatzerweiterung (ISE) stellt eine verbreitete Optimierungstechnik für Prozessor-basierte Systeme dar. Viele Prozessoren besitzen neben dem normalen Instruktionssatz (IS) zusätzliche Befehle, die nur für bestimmte Anwendungsgebiete optimiert sind. Bekannte Beispiele sind die Multimedia-Instruktionssatzerweiterungen MMX bei Intel-Prozessoren bzw. die korrespondierende 3DNow!-Technologie bei AMD-Prozessoren [SS02]. Auch eingebettete Prozessoren besitzen häufig Instruktionssatzerweiterungen, wie beispielsweise die NEON-Erweiterung der ARM-Prozessoren [ARM10] zeigt.

Das Prinzip der Instruktionssatzerweiterung besteht darin, häufig vorkommende Befehlssequenzen innerhalb des Programmcodes zu einer neuen *Superinstruktion* zusammenzufassen. Abbildung 6.1 zeigt drei verschiedene Varianten der Instruktionssatzerweiterung, welche in dieser Arbeit näher betrachtet werden. Als Ausgangspunkt dient die in Abbildung 6.1a dargestellte Codesequenz. Diese stellt beispielhaft einen Programmausschnitt dar, welcher 10 Befehle aus dem Instruktionssatz IS = {`A`, `B`, `C`, `D`, `E`} verwendet. Es wird angenommen, dass jede Instruktion $\mathrm{I} \in \mathrm{IS}$ innerhalb eines Taktes von dem zugrundeliegenden Prozessorsystem ausgeführt wird. Die grün hinterlegte Befehlsfolge der Codesequenz stellt dabei die `Funktion 1` dar. Ohne jegliche Instruktionssatzerweiterung werden die Instruktionen in der darge-



(a) Ursprüngliche Codesequenz (b) Instruktionspaar (c) Mikroprogramm (d) Hardware-Beschleuniger

Abbildung 6.1: Varianten der Instruktionssatzerweiterung

stellten Reihenfolge sequentiell vom Prozessor abgearbeitet. Die Ausführung der Codesequenz in Abbildung 6.1a benötigt demnach genau 10 Takte.

Instruktionspaare verkörpern eine Form der Instruktionssatzerweiterung. Wie in Abbildung 6.1b dargestellt, werden bei dieser Variante die beiden aufeinander folgenden Instruktionen `C` und `D` zu einer neuen *Superinstruktion* `CD` zusammengefasst. Unter bestimmten Umständen können auch drei oder vier Befehle zu einem *Instruktions-Trippel* bzw. *Quadrupel* vereint werden. Dieses hängt von den jeweiligen Instruktionen und der Datenpfad-Architektur des ausführenden Prozessors ab (siehe Abschnitt 6.1.2). Die Effizienz neuer Superinstruktionen hängt immer davon ab, wie oft diese im gesamten Programm verwendet werden können. Dabei können Instruktionspaare im Allgemeinen häufiger ersetzt werden als Instruktions-Trippel bzw. Quadrupel. In Beispiel 6.1a kann ein Instruktionspaar zweimal durch die neue Superinstruktion `CD` ersetzt werden. Der Prozessor muss nun jeweils nur eine anstatt zwei Instruktionen ausführen, was die Ausführungszeit insgesamt um 2 Takte verkürzt. Neben der Ausführungszeit verringert sich außerdem der Platzbedarf im Programmspeicher des Prozessorsystems, da in diesem Beispiel dort nun zwei Instruktionen weniger abgelegt werden müssen. Hinsichtlich der Hardware-Implementierung ist die Kombination mehrerer Einzelinstruktionen zu einer Superinstruktion meist ohne großen Aufwand umsetzbar, da lediglich der Kontrollpfad des Prozessors um die neue Instruktion erweitert werden muss. Der Datenpfad muss in der Regel nicht verändert werden, da die Funktionalität der Einzelinstruktionen bereits vorhanden ist.

Mikroprogramme bilden eine weitere Variante der Instruktionssatzerweiterung. Die Funktionsweise ist in Abbildung 6.1c dargestellt. Das Mikroprogramm bildet die Funktionalität der kompletten `Funktion 1` als Hardware-Implementierung im Prozessor ab. Die Superinstruktion `1` ersetzt dabei alle Einzelinstruktionen der ursprünglichen Funktion. Die Codegröße, d. h. der Platzbedarf im Programmspeicher, für die gesamte Funktion wird auf eine Instruktion minimiert. Wie in Abbildung 6.1c erkennbar, verringert sich die Ausführungszeit der Funktion allerdings nicht. Bei einem Mikroprogramm handelt es sich um eine so genannte *multi-cycle*-Instruktion, d. h. der Prozessor benötigt mehrere Takte um den Befehl auszuführen. Prinzipiell

arbeitet der Prozessor weiterhin sequentiell die ursprünglichen Einzelinstruktionen der Funktion ab, allerdings bestimmt die Reihenfolge und Art der Instruktionen nun ein Zustandsautomat (FSM, engl. Finite State Machine) im Kontrollpfad des Prozessors. Der Datenpfad bleibt wie bei den Instruktionspaaren unverändert. Neben der Reduktion der Codegröße kann mit Hilfe des Mikroprogramms Energie gespart werden, da nur einmal auf den Programmspeicher zugegriffen werden muss, um die Superinstruktion `1` zu laden. Ohne das Mikroprogramm müssten alle Einzelinstruktionen der Funktion nacheinander aus dem Speicher gelesen werden. Da die Speicherzugriffe im Allgemeinen mehr Energie verbrauchen als der Zustandsautomat für das Mikroprogramm, wird auf diese Weise bei jedem Aufruf der Funktion Energie gespart.

Hardware-Beschleuniger entsprechen Abbildung 6.1d und stellen die letzte Variante der betrachten Instruktionssatzerweiterungen dar. Genau wie bei dem Mikroprogramm wird die komplette `Funktion 1` durch eine Superinstruktion ersetzt. Diese bildet nun aber keine multi-cycle-Instruktion mehr, sondern kann vom Prozessor innerhalb eines Taktes ausgeführt werden. Neben einer Erweiterung des Kontrollpfades sind hierzu auch Anpassungen am Datenpfad des Prozessors erforderlich. Die Funktionalität der Superinstruktion `1` wird beispielsweise als zusätzliche Komponente, d. h. als Hardware-Beschleuniger, in die ALU integriert. Bezüglich der Reduktion sowohl von Codegröße als auch von Ausführungszeit bildet der Hardware-Beschleuniger die effektivste Variante der Instruktionssatzerweiterung. Dieses kann allerdings nur auf Kosten der Chipfläche erreicht werden, da die Erweiterung des Datenpfades je nach Funktionalität weitaus mehr Hardware benötigt als eine Anpassung des Kontrollpfades, wie sie für ein Mikroprogramm oder Instruktionspaar nötig ist.

Alle vorgestellten Varianten der Instruktionssatzerweiterung erfordern neben der Hardware-Erweiterung auch eine entsprechende Anpassung der Software-Umgebung. Der Übersetzer muss die Funktionalität der neuen Instruktionen genau kennen, um diese bei der Erzeugung des Maschinencodes verwenden zu können. Im Folgenden wird eine zweistufige Entwurfsmethodik vorgestellt, welche die Auswahl geeigneter Instruktionssatzerweiterungen vereinfacht und deren Hardware- als auch Software-Umsetzung vereint.

6.1.1 Entwurfsmethodik

Die große Herausforderung bei der Instruktionssatzerweiterung ist, eine möglichst effiziente Auswahl an neuen Befehlen zu treffen. Dabei stellt sich die Frage, welche Art von Instruktionssatzerweiterung für eine vorgegebene Anwendung die beste Ressourceneffizienz erzielt? Um diese Fragestellung systematisch beantworten zu können, wurde die in Abbildung 6.2 dargestellte Entwurfsmethodik entwickelt [PSP08]. Es handelt sich dabei um einen automatisierten, zweistufigen Ablauf, der sowohl die Software-Anpassung (*compiler-related workflow*) als auch die Hardware-Erweiterung (*hardware-related workflow*) abdeckt. Ein ähnlicher Ansatz wurde bereits in [Nie08] und [Jun11] erfolgreich angewandt.

Software-seitig wurde der Entwurfsablauf in enger Kooperation mit der Fachgruppe Programmiersprachen & Übersetzer der Universität Paderborn erarbeitet. Den Ausgangspunkt für die zweistufige Entwurfsmethodik bildet eine zu untersuchende Anwendung, welche als Quellcode in der Programmiersprache C eingespeist wird. Mit Hilfe des Übersetzers (*Compiler*) wird das Programm in den Prozessor-spezifischen Maschinencode umgewandelt und als Speicherabbild (*memory image*)

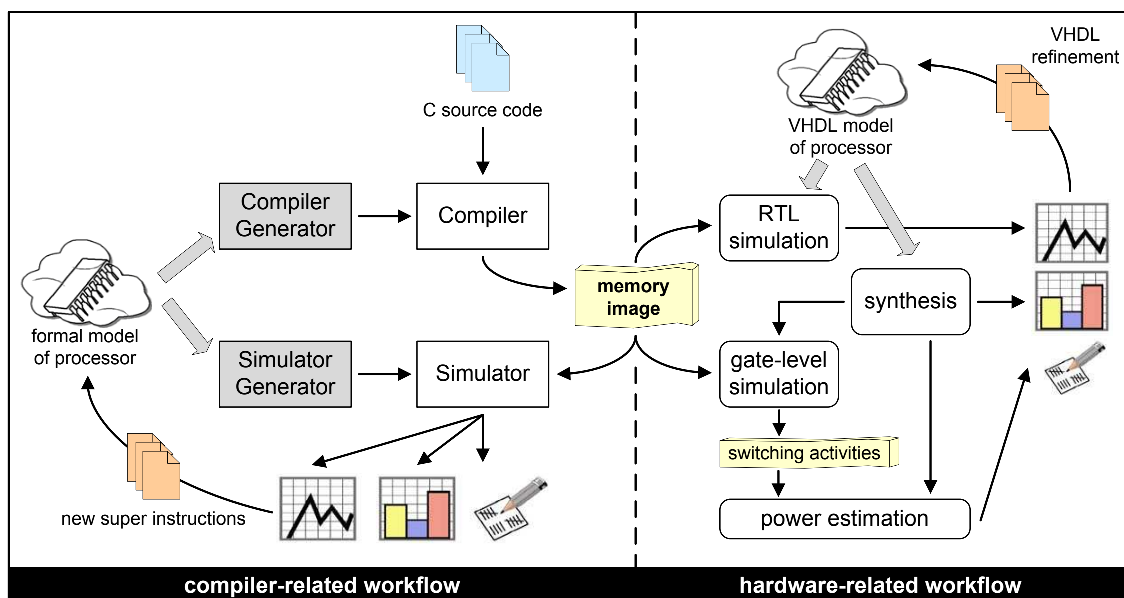


Abbildung 6.2: Zweistufige Entwurfsmethodik zur Instruktionssatzerweiterung

ausgegeben. Die zur Übersetzung benötigte Werkzeugkette (*Compiler*, *Assembler* und *Linker*) kann automatisch durch den *Compiler Generator* erzeugt werden. Dieser benötigt zur Generierung ein formales Prozessormodell, welches unter anderem die Hardware-Architektur sowie den Instruktionssatz beinhaltet.

Zur Modellierung wird die in der Fachgruppe Programmiersprachen & Übersetzer entwickelte Prozessor-Beschreibungssprache UPSLA[4] verwendet [KLST04]. Das UPSLA-basierte Modell des Prozessors enthält beispielsweise genaue Informationen über die Datenwortbreite, Pipeline-Struktur, Anzahl der Register sowie Codierung und Funktionalität aller Instruktionen. Neben dem Übersetzer kann mit Hilfe des *Simulator Generators* aus dem Prozessormodell auch ein Instruktionssatz-Simulator (ISS) erzeugt werden. Der Simulator kann das zuvor erzeugte Speicherabbild verwenden, um die Ausführung der zu untersuchenden Anwendung auf dem Ziel-Prozessor zu simulieren. Während der Simulation werden die ausgeführten Instruktionen protokolliert. Aus den aufgezeichneten Informationen können anschließend automatisch verschiedene Statistiken erzeugt und visualisiert werden. So können beispielsweise Befehlshäufigkeiten, die Anzahl der Speicherzugriffe, die Ausführungsdauer ausgewählter Programmabschnitte und vor allem das Auftreten von Instruktionspaaren komfortabel untersucht werden. Diese Analyse ermöglicht eine schnelle Auswahl der Instruktionspaare, welche besonders häufig während der Ausführung der Anwendung auftreten und somit das größte Optimierungspotential für eine Erweiterung des Instruktionssatzes bieten.

Die Software-seitige Umsetzung der Instruktionssatzerweiterung besteht in der Anpassung des Prozessormodelles. Die neue Instruktion (*super instruction*) muss mit Hilfe von UPSLA modelliert werden. Anschließend kann ein neuer Übersetzer und Simulator generiert werden, welcher die Instruktionssatzerweiterung beinhaltet. Mit dem aktualisierten Übersetzer wird schließlich erneut ein Speicherabbild erzeugt und der Software-seitige Entwicklungskreis ist geschlossen. Durch eine weitere Simulation kann überprüft werden, ob die eingeführte Instruktionssatzerweiterung funktional korrekt modelliert wurde. Um die Ressourceneffizienz der Instruktionssatzerweiterung zu beurteilen wird zum Hardware-seitigen Entwurfsablauf gewechselt.

[4] Unified Processor Specification Language

Hardware-seitig stellt das Speicherabbild der übersetzten Anwendung den Ausgangspunkt der Entwurfsmethodik dar. Das Speicherabbild wird in den Speicher des Prozessorsystems (vgl. Abbildung 5.1) geladen. Auf diese Weise kann die Anwendung im Rahmen einer Hardware-Simulation ausgeführt werden. Zuvor muss die Instruktionssatzerweiterung als Hardware-Beschreibung umgesetzt werden, so dass VHDL- und UPSLA-Modell des Prozessors übereinstimmen. Die RTL-Simulation kann wiederum dazu genutzt werden die funktionale Korrektheit der Hardware-Umsetzung zu überprüfen. Ferner kann die Ausführungszeit der Anwendung taktgenau ermittelt werden. Um die Ressourcen Chipfläche und Verlustleistung zu bestimmen, ist ein Synthese-Schritt notwendig, welcher die Hardware-Beschreibung von der Register-Transfer-Ebene auf die Gatter-Ebene überführt (vgl. Kapitel 4.2). Während der Simulation auf Gatter-Ebene können nun sämtliche Schaltaktivitäten (*switching activities*) protokolliert werden. Mit Hilfe der aufgezeichneten Schaltaktivitäten kann schließlich eine anwendungsspezifische Abschätzung der Verlustleistung durchgeführt werden.

Mit der beschriebenen Entwurfsmethodik können beliebig viele Iterationen vorgenommen werden, bis die gewünschte Performanz des Instruktionssatzes, bezogen auf die untersuchte Anwendung, erreicht ist. Durch die Bewertung der Ressourceneffizienz ist es möglich, verschiedene Arten der Instruktionssatzerweiterung miteinander zu vergleichen. Obwohl die Instruktionssatzerweiterung auf Basis der vorgestellten Entwurfsmethodik eine anwendungsspezifische Optimierung darstellt, können prinzipiell alle Anwendungen von den hinzugefügten Befehlen profitieren, da die Übersetzer-Anpassung im Software-seitigen Entwurfsablauf einbezogen ist.

6.1.2 Implementierungsvarianten

Die soeben vorgestellte Entwurfsmethodik wird im Folgenden herangezogen, um das Optimierungspotential von Instruktionssatzerweiterungen für die Kryptographie mit elliptischen Kurven auszunutzen. Als Referenzproblem wird die bereits als sehr Performanz-kritisch identifizierte Multiplikation im Binärkörper untersucht. Für die Analyse und Implementierung wurde exemplarisch der Binärkörper $GF(2^{233})$ verwendet. Die eingeführten Optimierungen sind jedoch von allgemeiner Bedeutung und für sämtliche Binärkörper der Form $GF(2^n)$ anwendbar. Der Quellcode für die

Multiplikation im Binärkörper $GF(2^{233})$ ist in Anhang B.1 abgebildet. Die Funktion `ff_mul_t8_w32_k2` bekommt als Parameter die drei Speicheradressen (*Pointer*) `*A`, `*B` und `*C` übergeben. An den Speicherpositionen `*A` und `*B` befinden sich jeweils die Koeffizienten a_i und b_i der Faktoren $a, b \in GF(2^{233})$ in der Polynombasis-Darstellung $A = \sum_{i=0}^{232} a_i x^i$ bzw. $B = \sum_{i=0}^{232} b_i x^i$. Die Ergebniskoeffizienten c_i der Binärkörper-Multiplikation $A(x) \cdot B(x) = C(x)$ mit $C = \sum_{i=0}^{464} c_i x^i$ werden entsprechend an die Speicherposition `*C` geschrieben. Aufgrund der gewählten Prozessorarchitektur können die Koeffizientenvektoren nur als Vielfaches der Datenwortbreite $w = 32\,\mathrm{Bit}$ abgelegt werden. Für den Erweiterungsgrad $n = 233$ bedeutet dieses, dass für die Koeffizienten von A und B jeweils $\lceil 233/32 \rceil = 8$ Datenworte bzw. für das Ergebnis C demzufolge 16 Datenworte verwendet werden. Die nicht genutzten Bit werden entsprechend zu Null gesetzt. Wie in Anhang B.1 zu erkennen ist, verwendet die Funktion `ff_mul_t8_w32_k2` die Karatsuba-Methode, um die Polynomlänge iterativ solange zu halbieren, bis die Datenwortbreite des Prozessorsystems erreicht ist. Schließlich wird mit der Funktion `ff_mul_w32_s3` das optimierte Schiebe-und-Addiere-Verfahren gemäß Algorithmus 7 angewandt, um die Binärkörper-Multiplikation auf Wort-Ebene durchzuführen (vgl. Abschnitt 5.1.2).

Um die Binärkörper-Multiplikation auf potentielle Instruktionssatzerweiterungen zu untersuchen, wurde ein Testprogramm geschrieben, welches die zu analysierende Funktion `ff_mul_t8_w32_k2` mit geeigneten Parametern aufruft und das Ergebnis der Berechnung auf Korrektheit überprüft. Dieses Testprogramm wird, wie in Abbildung 6.2 dargestellt, in den Software-seitigen Entwurfsablauf eingespeist. Nach Übersetzung des Programms wird das Speicherabbild mit Hilfe des Instruktionssatz-Simulators ausgeführt. Abbildung 6.3 zeigt die Auswertung der identifizierten Instruktionspaare. Wie man erkennen kann, treten folgende Instruktionspaare besonders häufig während der Ausführung der Funktion `ff_mul_t8_w32_k2` auf:

- `addu,ldw` – add unsigned, load word
- `andi,addu` – `AND` immediate, add unsigned
- `lsli,xor` – logical shift left by immediate, `XOR`
- `lsri,andi` – logical shift right by immediate, `AND` immediate
- `lsri,xor` – logical shift right by immediate, `XOR`

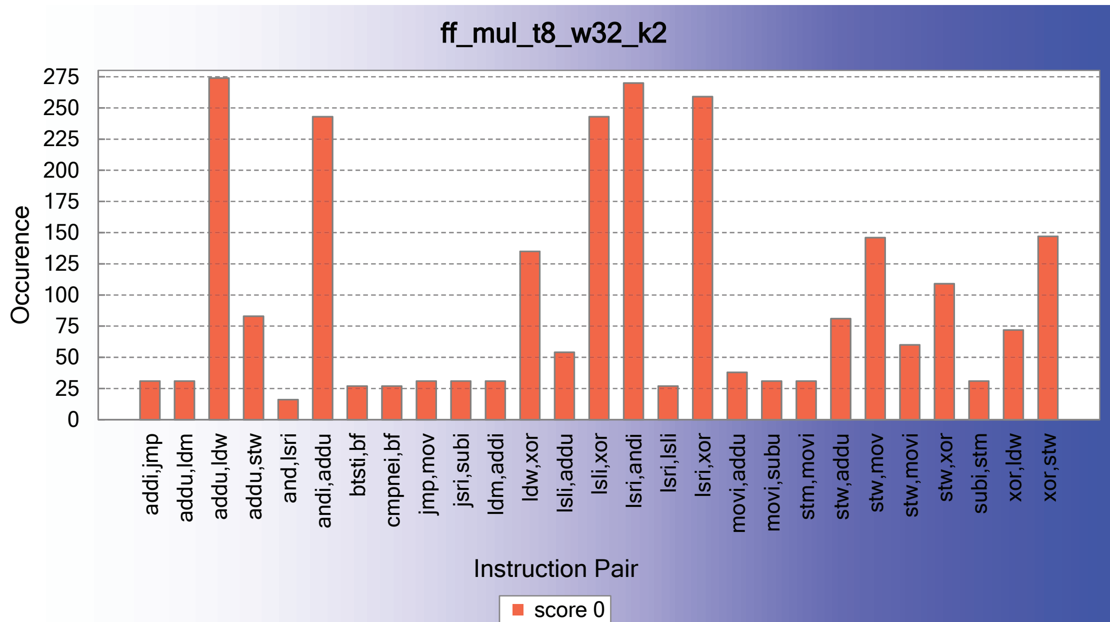


Abbildung 6.3: Instruktionspaare der Multiplikation im Binärkörper $GF(2^{233})$

Bei genauerer Betrachtung des Assembler-Codes, stellt man fest, dass die aufgelisteten Instruktionspaare ihren Ursprung innerhalb der Multiplikationsfunktion `ff_mul_w32_s3` auf Wort-Ebene haben. Wie in Abschnitt 3.3.2 beschrieben, wird diese Funktion insgesamt 27 mal während der Multiplikation im Binärkörper $GF(2^{233})$ durch die Funktion `ff_mul_t8_w32_k2` aufgerufen. Dieses erklärt das häufige Auftreten der genannten Instruktionspaare. Da die Funktion `ff_mul_w32_s3` die Grundfunktion für die Multiplikation in allen betrachten Binärkörpern darstellt, verspricht die Integration von Superinstruktionen hier ein großes, allgemein anwendbares Optimierungspotential. Um dieses Potential optimal auszuschöpfen, wurde eine separate Analyse des Assembler-Codes für die Basisfunktion `ff_mul_w32_s3` durchgeführt. Dabei stellt sich heraus, dass der Übersetzer die Nachschlagetabelle `U` auf dem Stapelspeicher (engl. stack) ablegt. Der Stapelspeicher wird beim S-Core über das Register `R0` adressiert, welches auch *Stackpointer* (`SP`) genannt wird. Nach der Initialisierung (siehe Anhang B.1, Zeilen 8−11) können die acht Werte der Nachschlagetabelle `U` somit relativ zum Stackpointer adressiert werden. Diese Eigenschaft wird anschließend genutzt, um jeweils die Werte für die Hilfsvariable `V` zu berechnen. Alle benötigten Werte von `V` (vgl. Zeilen 16 − 25) werden nacheinander berechnet und in zehn verschiedenen Prozessor-Registern gespeichert. Die zehn Berechnungen

```
1  mov   r5,r2                        mov   r5,r2        // Kopieren von Faktor A nach R5
2  lsri  r5,1                                            // Berechnung des Indexwertes
3  andi  r5,28      ISE    LSRIANDADD       r5,1         // von Nachschlagetabelle U
4  addu  r5,sp                                           // als relative Adresse zum SP
5  ldw   r6,(r5)                      ldw   r6,(r5)      // Lade R6 mit U[(A >> 3) & 7]
```

Abbildung 6.4: Superinstruktion LSRIANDADD

erfolgen dabei immer nach dem gleichen Schema, welches in Abbildung 6.4 dargestellten ist. Das Register `R2` beinhaltet den Multiplikator `A`. Zunächst wird eine Kopie von `A` in Register `R5` erstellt. Entsprechend der Quellcodezeilen 16 – 25 der Funktion `ff_mul_w32_s3` in Anhang B.1 wird der Faktor `A` für jede Berechnung von `V` um ein Vielfaches von drei nach rechts geschoben. Schließlich werden die drei LSB durch eine `UND`-Verknüpfung mit sieben ausmaskiert, um den Index für die Nachschlagetabelle `U` zu erhalten. Diesen Teil des Quellcodes hat der Übersetzer bereits insofern optimiert, dass im Assembler-Code der Faktor `A`, bzw. dessen Kopie in Register `R5`, jeweils um zwei Stellen weniger nach rechts geschoben wird. Diese Optimierung beruht auf der Tatsache, dass der Speicher des S-Cores byteweise adressiert wird und jeder Eintrag von `U` einem 32-Bit-Wort entspricht. Für einen Zugriff auf `U[i]` muss der Stackpointer daher um $4i$ erhöht werden. Der Faktor 4 wird durch das um zwei Stellen reduzierte Rechtsschieben automatisch erreicht. Für eine korrekte Maskierung muss nun konsequenterweise der Wert 7 um zwei Bit nach links geschoben werden, so dass im Assembler-Code eine `UND`-Verknüpfung mit 28 durchgeführt wird. Der Wert des Stackpointers wird nun zum berechneten Adressindex hinzu addiert. Schließlich wird der entsprechende Wert aus der Nachschlagetabelle `U` vom Stack in das Register `R6` geladen. Abbildung 6.4 zeigt den Assembler-Code für die erste Berechnung von `V` entsprechend Zeile 16 im Anhang B.1. Dasselbe Schema wird für die Berechnung der übrigen neun Werte von `V` angewandt. Es wird lediglich die Anzahl der Stellen variiert, um welche der Faktor `A` nach rechts geschoben wird sowie das Ziel-Register, in den der indizierte Wert aus `U` geladen wird.

Die ALU-Architektur des S-Cores ist in Abbildung 6.5 dargestellt und bietet die Möglichkeit, eine Schiebe-Operation, eine Logik-Operation sowie eine Addition bzw. Subtraktion innerhalb eines Prozessortaktes kaskadiert auszuführen. Hierdurch können die Zeilen 2 – 4 des Assembler-Codes in Abbildung 6.4 zu einer neuen Superinstruktion `LSRIANDADD` zusammengefasst werden. Die Realisierung dieser Superin-

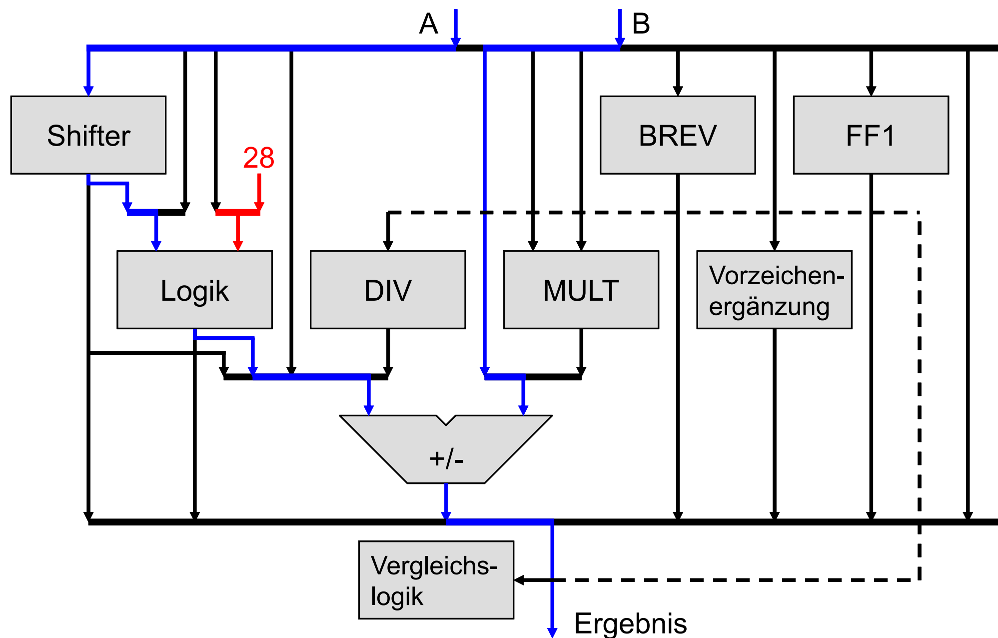


Abbildung 6.5: Erweiterte Recheneinheit (ALU) des S-Cores (vgl. [Has99])

struktion erfordert nur eine minimale Erweiterung der Recheneinheit (ALU), welche in Abbildung 6.5 rot hervorgehoben ist. Mit Hilfe eines zusätzlichen Multiplexers kann der Eingang des Logik-Blockes mit der Konstanten 28 gespeist werden, was für die Maskierung durch die `UND`-Verknüpfung (vgl. Zeile 3 in Abbildung 6.4) notwendig ist. Der Datenpfad für die Superinstruktion `LSRIANDADD` ist in Abbildung 6.5 blau markiert. Für die Superinstruktion `LSRIANDADD r5,1` wird über Eingang A das Register `R5` in den Schiebe-Block (*shifter*) geleitet und dort um den übergebenen Wert, in diesem Beispiel also 1, nach rechts geschoben. Das Ergebnis wird zum Logik-Block geführt und dort mit 28 logisch `UND` verknüpft. Dieses Ergebnis geht wiederum direkt in den Additions- bzw. Subtraktions-Block (+/−) und wird mit dem Wert des Stackpointers, welcher aus Register `R0` über Eingang B in die ALU kommt, addiert. Das Endergebnis wird schließlich wieder in Register `R5` gespeichert. Durch eine geringfügige Erweiterung der ALU ist es also nun möglich, drei sequentielle Befehle in nur einem Takt auszuführen.

Auf die gleiche Weise können ebenfalls die bereits identifizierten Instruktionspaare `LSLI,XOR` und `LSRI,XOR` zu den Superinstruktionen `LSLIXOR` bzw. `LSRIXOR` kombiniert werden. Betrachtet man wieder den Quellcode der Funktion `ff_mul_w32_s3`

(Anhang B.1), so wird aus den Zeilen $16-25$ direkt deutlich, dass die Kombination von Schiebe-Operation und `XOR`-Logik sehr vorteilhaft ist. Für die zehn Berechnungsschritte $1 \leq j \leq 10$ der Teilprodukte `C1` und `C0` wird der zuvor ermittelte Wert `V` jeweils um $32-3j$ Stellen nach rechts bzw. um $3j$ Stellen nach links geschoben und das Ergebnis mit `C1` bzw. `C0` logisch `XOR` verknüpft. Für die Superinstruktionen `LSRIXOR` bzw. `LSLIXOR` muss nicht einmal die Struktur der ALU erweitert werden, da die Kombination von Schiebe- und Logik-Operation bereits für vorhandene Befehle, z. B. `BMASKI` oder `XTRB` (vgl. [Has99]), genutzt wird. Allerdings muss während der Dekodierung der Superinstruktionen `LSRIXOR` und `LSLIXOR` etwas mehr Aufwand betrieben werden. Für die Kodierung des Schiebe-Parameters können nur 4 Bit verwendet werden, was nicht ausreicht, um den möglichen Wertebereich zwischen 0 und 32 abzudecken. Daher wird während der Instruktionsdekodierung die in Tabelle 6.1 dargestellte Formel benutzt, um alle benötigten Schiebe-Parameters der Funktion `ff_mul_w32_s3` dekodieren zu können.

Super-instruktion	**Maschinencode**				**Funktionalität**
	[15:12]	[11:8]	[7:4]	[3:0]	
`LSRIANDADD rx,i5`	0010	011	i5	rx	rx = r0 + ((rx >> i5) & 28)
`LSRIXOR rx,ry,i4`	0101	i4	rx	ry	rx = rx ^ (ry >> (3·i4))
`LSLRXOR rx,ry,i4`	0100	i4	rx	ry	rx = rx ^ (ry << (32 − 3·i4))
`MULGF2 rx,ry`	0110	1000	rx	ry	(rx,ry) = rx ⊗ ry

Tabelle 6.1: Maschinencode und Funktionalität eingeführter Superinstruktionen

Tabelle 6.1 zeigt die Kodierung der eingeführten Superinstruktionen im Maschinencode sowie deren Funktionalität. Diese wurde wie zuvor beschrieben sowohl in die Hardware-Beschreibung des S-Cores als auch in das UPSLA-Prozessormodell integriert. Neben den bereits erläuterten Superinstruktionen beinhaltet Tabelle 6.1 zusätzlich den Befehl `MULGF2`. Dieser wird im folgenden Abschnitt für Instruktionssatzerweiterungen in Form von einem Mikroprogramm (vgl. Abbildung 6.1c) und einem Hardware-Beschleuniger (vgl. Abbildung 6.1d) benutzt. Die Auswirkungen der einzelnen Instruktionssatzerweiterungen und deren Einfluss auf die Ressourceneffizienz wird anschließend in Abschnitt 6.1.3 diskutiert.

Mikroprogramm und Hardware-Beschleuniger bilden zwei weitere Varianten der Instruktionssatzerweiterung. Anders als bei den zuvor vorgestellten Instruktionspaaren wird nun eine komplette Funktion durch eine Superinstruktion im Befehlssatz des S-Cores abgedeckt. Wie in Tabelle 6.1 zu erkennen ist, trägt diese Superinstruktion den Namen `MULGF2` und führt eine Binärkörper-Multiplikation auf Wort-Ebene durch. Die 32-Bit-Register `rx` und `ry` beinhalten jeweils die Koeffizienten der Faktoren in Polynombasis-Darstellung, welche im Binärkörper $GF(2^{32})$ multipliziert werden. Das unreduzierte Ergebnis hat eine Länge von 64 Bit und wird wieder in die Register `rx` bzw. `ry` zurückgeschrieben. Die Superinstruktion `MULGF2` entspricht demnach der Funktion `ff_mul_w32_s3`.

Abbildung 6.6 zeigt den Datenpfad der Implementierungsvariante des Mikroprogrammes. Die 32-Bit-Binärkörper-Multiplikation wird iterativ nach der Schiebe-und-Addier-Methode (vgl. Algorithmus 4) berechnet. Zunächst wird das 64-Bit-Ergebnisregister mit Nullen initialisiert. In jeder Iteration wird das Ergebnisregister um ein Bit nach links geschoben und der Multiplikand aufaddiert, falls das MSB des Multiplikators einer logischen Eins entspricht. Anschließend wird der Multiplikator ebenfalls um ein Bit nach links geschoben und die nächste Iteration beginnt. Der Algorithmus benötigt folglich genau 32 Prozessortakte zur Berechnung der Multiplikation im Binärkörper $GF(2^{32})$.

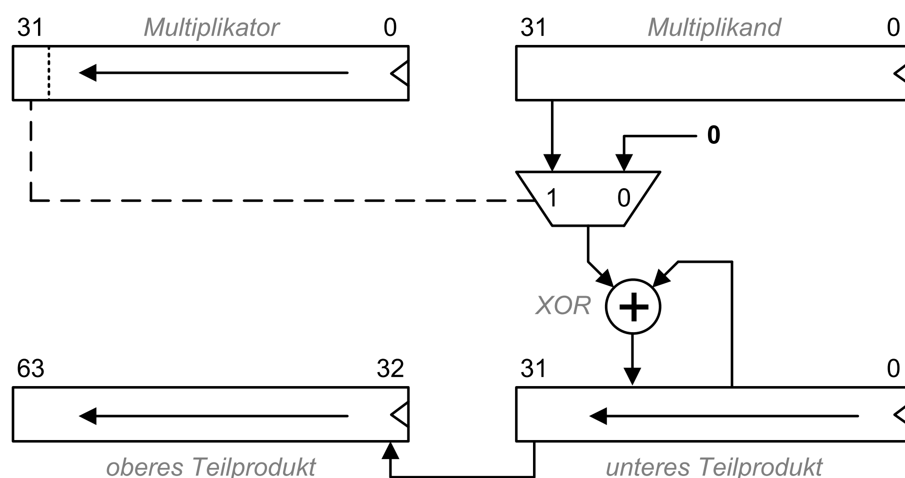


Abbildung 6.6: Datenpfad des Mikroprogrammes MULGF2

Zur Umsetzung des Datenpfades können viele bereits existierende Komponenten der ALU genutzt werden. So wird beispielsweise das Schieben des Multiplikators durch den *Shift*-Block erledigt. Die Addition des Multiplikanden entspricht im Binärkörper einer `XOR`-Operation, welche mit Hilfe des *Logik*-Blockes ausgeführt wird. Lediglich das 64-Bit-Schieberegister zur Zwischenspeicherung des Produktes muss als zusätzliche Hardware-Komponente implementiert werden. Nach Abschluss der Berechnung werden die Teilprodukte wieder in die allgemeinen Register `rx` und `ry` zurückgeschrieben. Mit Hilfe der Superinstruktion `MULGF2` auf Basis eines Mikroprogrammes kann die Binärkörper-Multiplikation auf Wort-Ebene somit in insgesamt 36 Prozessortakten durchgeführt werden (vgl. Abschnitt 6.1.3).

Als letzte Implementierungsvariante wird die Anbindung eines eng gekoppelten Hardware-Beschleunigers zur Instruktionssatzerweiterung betrachtet. Wie bereits in Abbildung 6.1d dargestellt wurde, kann mit dieser Methode die Ausführungszeit am stärksten verkürzt werden. Mit Hilfe des in Abschnitt 5.2.4 vorgestellten Codegenerators wird ein dedizierter Hardware-Block für die Multiplikation im Binärkörper $GF(2^{32})$ erzeugt. Hierzu wird die hybride Methode mit $m = 4$ als Abbruchkriterium für die Polynomlänge verwendet, so dass eine optimale Multiplizierer-Architektur generiert wird (vgl. Abbildung 5.8). Dieser Binärkörper-Multiplizierer wird als zusätzlicher Arithmetik-Block in den S-Core integriert. Der so eingebundene Hardware-Beschleuniger ist sehr eng an die ALU gekoppelt und ermöglicht die Berechnung der ursprünglichen Funktion `ff_mul_w32_s3` in nur einem einzigen Prozessortakt. Dieser Performanzgewinn kann allerdings nur mit hohem Aufwand an zusätzlichen Hardware-Komponenten erreicht werden. Der generierte Hardware-Multiplizierer benötigt 432 `AND`-Gatter, 745 `XOR`-Gatter mit jeweils zwei Eingangssignalen sowie ein 32-Bit-Register zur Zwischenspeicherung des oberen Teilproduktes. Genau wie beim Mikroprogramm wird nach Abschluss der Berechnung das 64-Bit-Ergebnis als oberes und unteres Teilprodukt in die Register `rx` bzw. `ry` zurückgeschrieben.

In diesem Abschnitt wurde die Umsetzung von drei Formen der Instruktionssatzerweiterungen, nämlich verschiedene *Instruktionspaare*, ein *Mikroprogram* sowie ein eng gekoppelter *Hardware-Beschleuniger*, vorgestellt (vgl. [PSP08]). Im folgenden Kapitel wird die Ressourceneffizienz der beschriebenen Implementierungsvarianten untersucht.

6.1.3 Ressourceneffizienz der Instruktionssatzerweiterung

Im vorherigen Abschnitt wurden drei verschiedene Arten von Instruktionssatzerweiterungen (ISE) vorgestellt, welche separat jeweils in einem S-Core integriert wurden. Das Derivat, welches die Instruktionspaare `LSRIANDADD`, `LSRIXOR` und `LSLIXOR` enthält, wird der Einfachheit halber im Folgenden als *S-Core (ISE1)* bezeichnet. Entsprechend lautet die Nomenklatur für die Implementierungsvariante des Mikroprogramms *S-Core (ISE2)* sowie für den Mikroprozessor mit integriertem Hardware-Beschleuniger *S-Core (ISE3)*.

Um die Auswirkungen der verschiedenen Instruktionssatzerweiterungen zu untersuchen, werden die Ressourcen Chipfläche (A), Verlustleistung (P) und Ausführungszeit (T), gemäß der vorgestellten Entwurfsmethodik (vgl. Abbildung 6.2), bestimmt. Auf Basis dieser Daten kann die Ressourceneffizienz (vgl. Abschnitt 4.3) berechnet werden, so dass die unterschiedlichen Implementierungsvarianten direkt miteinander vergleichbar sind.

Chipfläche

Die Chipfläche der verschiedenen Implementierungsvarianten ist in Abbildung 6.7 dargestellt. Bei den Flächenangaben handelt es sich um Synthese-Ergebnisse, welche auf einer Low-Power-Standardzellenbibliothek in 65-nm-CMOS-Technologie [IFX06] basieren. Die Zielfrequenz beträgt bei allen Varianten 200 MHz unter typischen Betriebsbedingungen, d. h. bei nominalen Prozessparametern, einer Temperatur von

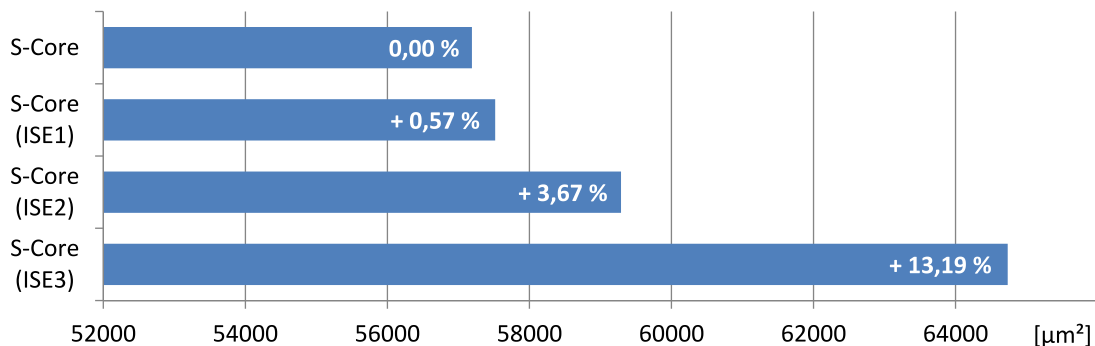


Abbildung 6.7: Chipfläche der verschiedenen Instruktionssatzerweiterungen

27 °C und einer Versorgungsspannung von 1,32 V. Das Hinzufügen der drei Instruktionspaare (ISE1) verursacht erwartungsgemäß nur einen minimalen Flächenzuwachs von 0,57 %. Die Implementierung des Mikroprogrammes (ISE2) benötigt im Vergleich zum ursprünglichen S-Core 3,67 % mehr Chipfläche, was in erster Line an dem zusätzlich erforderlichen 64-Bit-Schieberegister liegt. Der dedizierte Multiplizierer des Hardware-Beschleunigers (ISE3) erzeugt mit 13,19 % den größten Flächenzuwachs aller Implementierungsvarianten.

Verlustleistung

Die Verlustleistung wurde durch Annotierung der Schaltaktivitäten mit Hilfe des Werkzeugs PowerTheater R2006.2 bestimmt. Abbildung 6.8 zeigt den Leistungsverbrauch für eine Multiplikation im Binärkörper $GF(2^{233})$, welche durch die Funktion `ff_mul_t8_w32_k2` berechnet wurde. Neben dem Gesamtverbrauch wurde ebenfalls die Verlustleistung der modifizierten Blöcke, d. h. dem Instruktionsdekoder, der ALU sowie der Registerbank analysiert. Wie in Abbildung 6.8 zu erkennen ist, erzeugt die Implementierungsvariante (ISE3) mit 12,58 mW die meiste Verlustleistung. Dieses ist nicht überraschend, da während der Berechnung durch den Hardware-Beschleuniger hohe Schaltaktivitäten im Multiplizierer und den Registern auftreten. Erstaunlicher ist hingegen, dass mit Hilfe des Mikroprogrammes (ISE2) eine sehr leistungseffiziente Berechnung durchgeführt werden kann. Der implementierte Schiebe-und-Addier-Algorithmus verwendet eine Schleife, in welcher lediglich

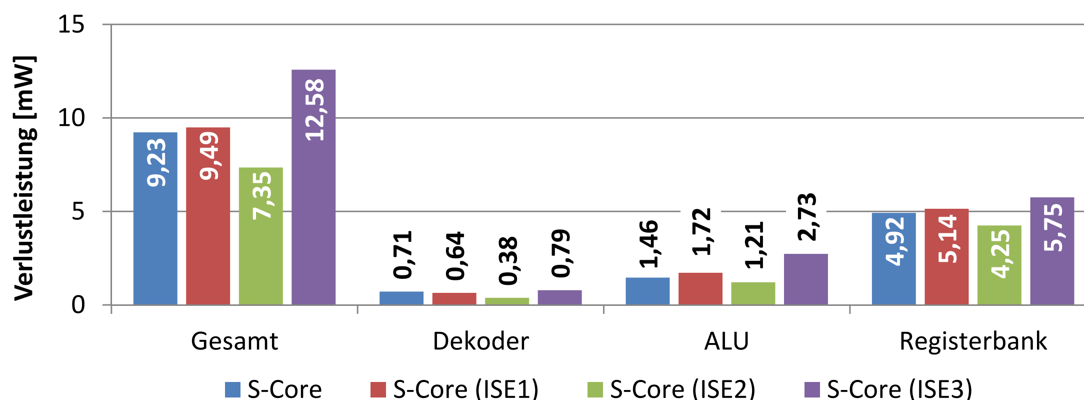


Abbildung 6.8: Verlustleistung der verschiedenen Implementierungsvarianten

die beiden Schieberegister und die `XOR`-Logik Schaltaktivitäten hervorrufen. Während der 32 Schleifendurchläufe werden große Teile des S-Cores, wie z. B. der Buscontroller und die Adressberechnung, nicht verwendet und tragen somit nur geringfügig zum Leistungsverbrauch bei. Der Unterschied zwischen dem ursprünglichen S-Core und der Variante ISE1 ist hinsichtlich der Verlustleistung vernachlässigbar klein. Die hinzugefügten Instruktionspaare ermöglichen zwar eine Berechnung mit weniger Befehlen, was sich in der Verlustleistung des Dekoders bemerkbar macht, allerdings benötigt die ALU zum Ausführen der Superinstruktionen wiederum mehr Leistung als der original S-Core.

Ausführungszeit

Die Beschleunigung der Berechnung sowie die Reduktion der Codegröße als Folge der Instruktionssatzerweiterungen ist in Tabelle 6.2 dargestellt. Der ursprüngliche S-Core benötigt zur Berechnung der Binärkörper-Multiplikation auf Wort-Ebene genau 166 Takte. Die zugrundeliegende Funktion `ff_mul_w32_s3` belegt 268 Byte im Programmspeicher. Mit Hilfe der Instruktionspaare (ISE1) lassen sich insgesamt 36 Takte (21,69 %) an Rechenzeit einsparen. Durch das Zusammenführen mehrerer Instruktionen reduziert sich außerdem der Speicherplatzbedarf um 26,86 %. Die mit 2 Byte geringste Codegröße erreichen das Mikroprogramm (ISE2) und der Hardware-Beschleuniger (ISE3), da diese lediglich die Superinstruktion `MULGF2` sowie einen Rücksprungbefehl zur Umsetzung der Funktion `ff_mul_w32_s3` benötigen. Im Vergleich zum ursprünglichen S-Core reduziert sich die Ausführungszeit der Implementierungsvariante ISE2 um 78,31 % bzw. der Implementierungsvariante ISE3 um 98,80 %. Die Performanzsteigerung relativiert sich allerdings et-

Variante	`ff_mul_w32_s3`		`ff_mul_t8_w32_k2`	
	Takte	Codegröße	Takte	Codegröße
S-Core	166	268 Byte	6 498	956 Byte
S-Core (ISE1)	130	196 Byte	5 545	880 Byte
S-Core (ISE2)	36	2 Byte	2 060	576 Byte
S-Core (ISE3)	2	2 Byte	1 169	576 Byte

Tabelle 6.2: Auswirkung der implementierten Instruktionssatzerweiterungen auf Ausführungszeit und Codegröße

was, wenn man sich die Werte für eine Multiplikation im Binärkörper $GF(2^{233})$ anschaut. Denn neben der Multiplikation auf Wort-Ebene müssen in der Funktion `ff_mul_t8_w32_k2` viele weitere Operationen (vgl. Anhang B.1) ausgeführt werden, welche nicht von den Instruktionssatzerweiterungen profitieren können. Während der original S-Core 6498 Takte und 956 Byte an Speicherplatz zur Multiplikation im Binärkörper $GF(2^{233})$ benötigt, kann durch Variante ISE1 die Ausführungszeit um 14,67 % beschleunigt und die Codegröße um 7,95 % reduziert werden. Mit Hilfe von Implementierungsvariante ISE2 bzw. ISE3 kann 39,75 % des ursprünglichen Speicherplatzbedarfes eingespart werden. Die Ausführungszeit für Variante ISE2 reduziert sich um 68,30 %, für Variante ISE3 um 82,01 %.

Anhand der Einzelergebnisse ist es schwierig eine Implementierungsvariante auszuwählen, welche die Anforderungen für eine vorgegebene Anwendung am besten erfüllt. Mit Hilfe der Ressourceneffizienz als Bewertungsmetrik (vgl. Gleichung 4.3) können die Instruktionssatzerweiterungen leichter miteinander verglichen werden. Wie in Abschnitt 4.3 beschrieben wurde, können die Parameter Chipfläche, Verlustleistung und Ausführungszeit für das betrachtete Anwendungsszenario entsprechend gewichtet werden. Abbildung 6.9 zeigt die normierte Ressourceneffizienz der zuvor beschriebenen Implementierungsvarianten für die vier anwendungsspezifischen Gewichtungen RE_1, RE_2, RE_3 und RE_4. Die für das jeweilige Szenario verwendeten Exponenten sind in Tabelle 6.3 aufgelistet. RE_1 repräsentiert das häufig

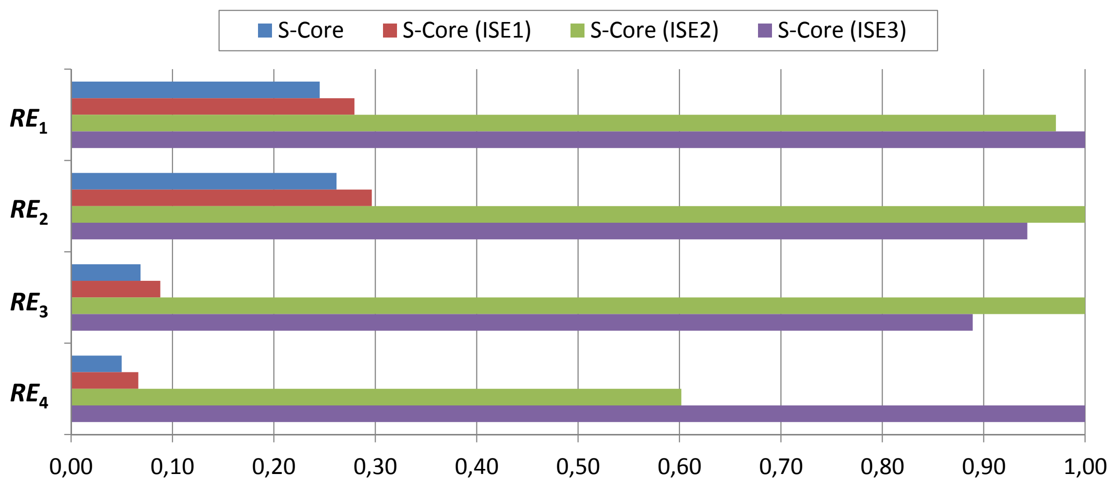


Abbildung 6.9: Normierte Ressourceneffizienz der Implementierungsvarianten

Ressourcen-effizienz	c_A	c_P	c_T	**Bemerkung/Szenario**
RE_1	0	1	1	Power-Delay-Produkt
RE_2	1	1	1	gleiche Gewichtung aller Ressourcen
RE_3	2	2	2	Anwendungsszenario Chipkarte
RE_4	1	1	2	Anwendungsszenario Sicherheitsserver

Tabelle 6.3: Gewählte Exponenten zur Gewichtung der Ressourceneffizienz

verwendete Power-Delay-Produkt [CB95][SS05][GP08]. Bei Betrachtung dieser Bewertungsmetrik erreicht die Variante ISE3 die beste Ressourceneffizienz. Allerdings berücksichtigt das Power-Delay-Produkt nicht die zur Implementierung benötigte Chipfläche ($c_A = 0$). Bezieht man diese mit in die Ressourceneffizienz ein, ergibt sich Bewertungsmetrik RE_2, bei der alle Ressourcen mit normaler Gewichtung ($c_A = c_P = c_T = 1$) einbezogen werden. Unter Einbeziehung der Chipfläche erzielt nun die Implementierungsvariante ISE2 die beste Ressourceneffizienz. Noch deutlicher zeigt sich der Unterschied zwischen ISE2 und ISE3 im Hinblick auf das Anwendungsszenario Chipkarte. Wie in Abschnitt 4.1.1 beschrieben, ist bei diesem Anwendungsszenario eine Minimierung der Chipfläche (A) sowie des Energieverbrauchs ($P{\cdot}T$) von enormer Wichtigkeit. Diese anwendungsspezifischen Anforderungen spiegelt Bewertungsmetrik RE_3 wieder, indem alle Gewichtungsexponenten auf den Wert Zwei gesetzt wurden. Wie in Abbildung 6.9 gut erkennbar ist, besitzt das Mikroprogramm (ISE2) für dieses Anwendungsszenario eine deutlich bessere Ressourceneffizienz als der Hardware-Beschleuniger (ISE3). Genau umgekehrt verhält es sich bei Bewertungsmetrik RE_4, welche das Anwendungsszenario Sicherheitsserver abbildet. Bei dieser Anwendung hat eine möglichst kurze Ausführungsgeschwindigkeit ($c_T = 2$) Priorität gegenüber der Chipfläche und der Leistungsaufnahme. Für dieses Anwendungsszenario hat der Hardware-Beschleuniger (ISE3) eindeutig die beste Ressourceneffizienz.

Als Fazit dieser Untersuchung kann festgehalten werden, dass populäre Bewertungsmetriken, wie z. B. das Power-Delay-Produkt, nur bedingt aussagekräftig sind. Die Bewertung der Ressourceneffizienz unter Berücksichtigung einer anwendungsspezifischen Gewichtung bildet häufig eine bessere Basis zur Auswahl einer geeigneten Implementierungsvariante [PSP08][PSPR08].

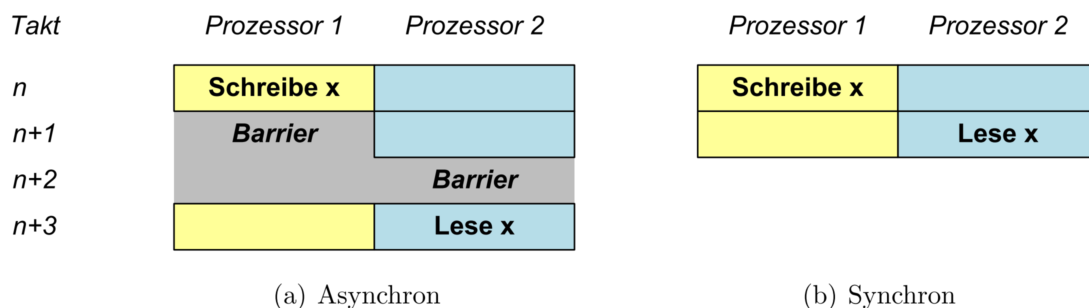
6.2 Parallelisierung

Die bislang vorgestellten Hardware-Software-Kombinationen basieren alle auf einem Einzelkernprozessor bzw. einem Mikroprozessorsystem, welches auf einem einzelnen S-Core basiert (vgl. Abbildung 4.5). Im Folgenden wird die Parallelisierbarkeit von Algorithmen für die Kryptographie mit elliptischen Kurven untersucht. Durch die Parallelisierung der Algorithmen können hauptsächlich zwei Ziele erreicht werden. Zum einen ist es durch die parallele Verarbeitung häufig möglich, ein definiertes Problem in kürzerer Zeit zu lösen. Dieses ermöglicht einen höheren Datendurchsatz, was dem Anwendungsszenario Sicherheitsserver sehr entgegenkommt. Zum anderen kann aber auch die Ausführungszeit als konstant angesetzt werden, so dass die Taktfrequenz der jeweiligen Prozessorelemente (PE) reduziert werden kann. Eine geringere Taktfrequenz wiederum erlaubt ebenfalls die Reduktion der Versorgungsspannung, so dass durch diesen Ansatz der Energieverbrauch (Anwendungsszenario Chipkarte) gesenkt wird.

Eine Berechnung, wie beispielsweise die Multiplikation im Binärkörper $GF(2^{233})$, lässt sich jedoch nicht beliebig in parallel ausführbare Blöcke zerlegen. Wie in jedem anderen Programm auch gibt es einen bestimmten sequentiellen Anteil, welcher nicht mit der Anzahl an verfügbaren Prozessorelementen skaliert [Amd67]. Des Weiteren muss bei der Parallelisierung zusätzlicher Aufwand für Kommunikation und Synchronisation berücksichtigt werden [Roo00][Pur09].

Kommunikation beschreibt den Aufwand für einen gegebenenfalls notwendigen Austausch von Daten zwischen einzelnen Prozessorelementen. Dieser Datenaustausch kann auf unterschiedliche Weise erfolgen. Am verbreitetsten sind die beiden Konzepte *Shared Memory* bzw. *Message Passing* [CSG98][PH05]. Beim Shared-Memory-Prinzip können alle Prozessorelemente auf einen gemeinsamen Speicherbereich zugreifen. Da jeweils immer nur ein Prozessor den Speicher Schreiben bzw. Lesen kann, eignet sich dieses Konzept vorrangig für Aufgaben mit geringem Kommunikationsbedarf und großen Datenmengen. Beim Message-Passing-Verfahren hingegen können die einzelnen Prozessorelemente direkt Nachrichten miteinander austauschen. Bei dieser Punkt-zu-Punkt-Kommunikation können mehrere Datenströme gleichzeitig zwischen verschiedenen Prozessoren ausgetauscht werden.

Synchronisation ist bei paralleler Verarbeitung notwendig, um Datenabhängigkeiten auflösen zu können. Wenn beispielsweise Prozessorelement 1 ein Datum lesen möchte, welches von Prozessorelement 2 berechnet wird, so muss PE1 mit der Leseoperation solange warten, bis PE2 mit der Berechnung fertig ist und das Ergebnis verfügbar ist. Zur Synchronisation von mehreren Prozessorelementen existieren verschiedene Methoden, wie zum Beispiel *Semaphor*- oder *Barrier*-Synchronisation [Tau06][CSG98]. Diese Synchronisationsmethoden kommen bei asynchronen Multiprozessorsystemen zum Einsatz. Asynchrone Parallelrechner zeichnen sich dadurch aus, dass die Prozessorelemente unabhängig voneinander ihre Instruktionen ausführen. Falls eine Datenabhängigkeit besteht, muss die Synchronisation explizit durch die genannten Methoden im Programmcode integriert werden (vgl. Abbildung 6.10a). Bei synchronen Mehrkernprozessoren hingegen folgen alle Recheneinheiten einem globalen Taktschema, so dass Datenabhängigkeiten schon im Vorfeld durch den Übersetzter aufgelöst werden können. Die Synchronisation ist hierbei implizit vorhanden (siehe Abbildung 6.10b), da der Übersetzer die Instruktionsfolgen der einzelnen Prozessorelemente exakt vorausplanen kann. Ein Beispiel für synchrone Parallelrechner bilden VLIW-Prozessoren [Roo00].



(a) Asynchron

(b) Synchron

Abbildung 6.10: Arten der Synchronisation in Multiprozessorsystemen [Pur09]

Ob synchrone oder asynchrone Mehrkernprozessoren besser zur Berechnung eines Algorithmus geeignet sind, hängt stark von der Art des Algorithmus bzw. von dessen inhärenter Parallelisierungsgranularität ab [HP03]. Enthält der Algorithmus Befehlsfolgen, bei denen die einzelnen Instruktionen unabhängig voneinander ausgeführt werden können, so spricht man von Instruktionsparallelität (ILP, engl. Instruction Level Parallelism). Werden bei einem Algorithmus häufig die gleichen Befehle auf verschiedene Datenblöcke angewandt, wird von Datenparallelität (DLP, engl.

Data Level Parallelism) gesprochen. Falls auf einer gröberen Granularitätsebene sogar größere Teilfunktionen eines Algorithmus unabhängig voneinander ausgeführt werden können, handelt es sich um Funktionsparallelität (TLP, engl. Thread Level Parallelism).

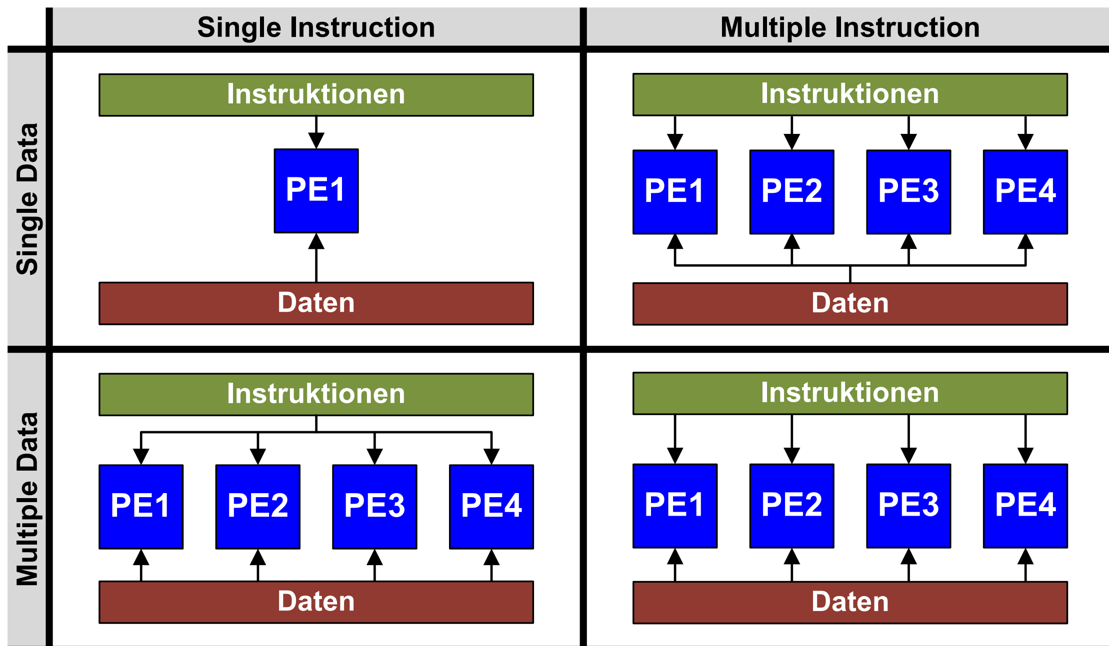


Abbildung 6.11: Klassifizierung von Rechnerarchitekturen nach Flynn [Fly72]

Die verschiedenen Granularitätsebene der Parallelität (ILP, DLP, TLP) erfordern unterschiedliche Rechnerarchitekturen, um den entsprechenden Algorithmus möglichst optimal auf einem Multiprozessorsystem auszuführen. Eine allgemeine Klassifizierung von Rechnerarchitekturen bildet die Flynn'sche Taxonomie [Fly72]. Wie in Abbildung 6.11 dargestellt ist, erfolgt die Einteilung der Architektur nach Anzahl der vorhandenen Instruktions- bzw. Datenströmen. Der in Abschnitt 6.1 betrachtete Einzelkernprozessor ist im linken, oberen Quadranten abgebildet und kann genau einen Instruktions- und Datenström zeitgleich verarbeiten. Bei dem S-Core-basierten Mikroprozessorsystem nach Abbildung 4.5 handelt es sich folglich um eine SISD[5]-Architektur. Die MISD[6]-Architektur im rechten, oberen Quadranten von Abbildung 6.11 kommt in fehlertoleranten Systemen zum Einsatz, wo redundante

[5] Single Instruction Single Data
[6] Multiple Instruction Single Data

Berechnungen auf einem Datenstrom parallel ausgeführt werden. Diese Rechnerarchitektur ist nicht weit verbreitet und wird in dieser Arbeit nicht näher betrachtet. Weitaus größere Bedeutung für die Parallelisierung von Algorithmen besitzen SIMD- bzw. MIMD-Architekturen [CSG98][HP03], wie sie im unteren Bereich von Abbildung 6.11 dargestellt sind. Ein Multiprozessorsystem mit SIMD-Architektur eignet sich optimal für Algorithmen mit großem Anteil an Datenparallelität (DLP). Algorithmen, welche hohe Instruktionsparallelität (ILP) aufweisen, lassen sich sehr gut auf einer synchronen MIMD-Architektur, wie z. B. einem VLIW-Prozessor, abbilden. Eine Parallelisierung auf Funktionsebene (TLP) erreicht häufig auf asynchronen MIMD-Architekturen die beste Performanz [Roo00].

Im Folgenden wird der Algorithmus zur Multiplikation im Binärkörper $GF(2^{233})$ auf zwei unterschiedliche Multiprozessorsysteme, den QuadroCore und den CoreVA, abgebildet [PPP08][JPD+10]. Beim QuadroCore-Mehrkernprozessor handelt es sich um eine zur Laufzeit rekonfigurierbare Parallelrechner-Architektur, welche sowohl eine SIMD-Struktur als auch einen asynchronen oder synchronen MIMD-Betrieb ermöglicht. Das CoreVA-Multiprozessorsystem stellt einen typischen VLIW-Parallelrechner mit synchroner MIMD-Architektur dar. Die Berechnung der Binärkörper-Multiplikation auf den jeweiligen Multiprozessorsystem kann mit Hilfe der Ressourceneffizienz direkt miteinander verglichen werden.

6.2.1 QuadroCore

Der QuadroCore wurde am Fachgebiet Schaltungstechnik der Universität Paderborn entwickelt [Pur09]. Die Basis für die QuadroCore-Architektur bildet das in Abbildung 4.6 dargestellte S-Core-Prozessorfeld mit vier Prozessorelementen. Das S-Core-basierte Multiprozessorsystem wurde um rekonfigurierbare Verbindungsstrukturen so erweitert, dass zur Laufzeit eine Umschaltung zwischen verschiedenen Arten der Parallelverarbeitung möglich ist. Der so entstandene QuadroCore kann zwischen einem SIMD-Modus sowie einem asynchronen oder synchronen MIMD-Betrieb wechseln. Wie in Abbildung 6.12 dargestellt ist, sind auch heterogene Konfigurationen möglich, so dass z. B. drei der vier Prozessorelemente im synchronen MIMD-Modus arbeiten, während der vierte Prozessor asynchron dazu betrieben wird. Die Rekonfiguration des Betriebsmodus geschieht durch speziell hinzugefügte Instruktionen und

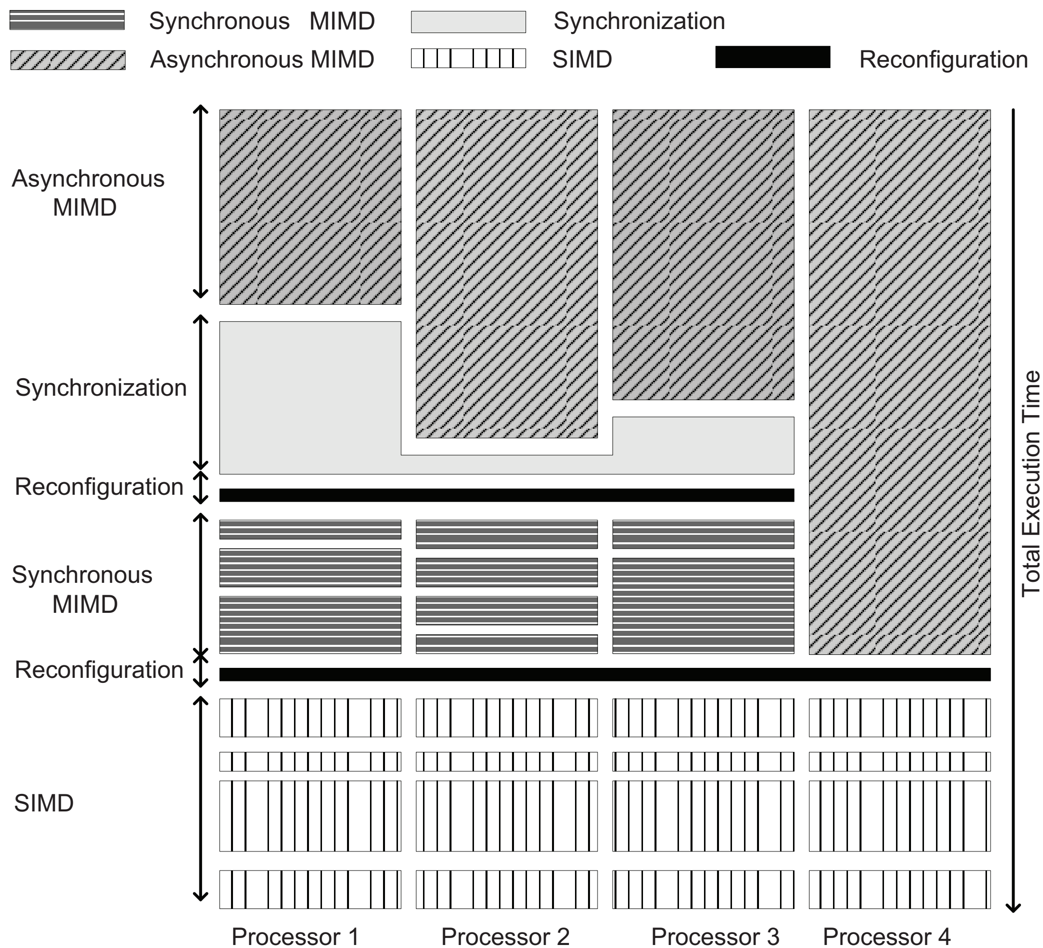
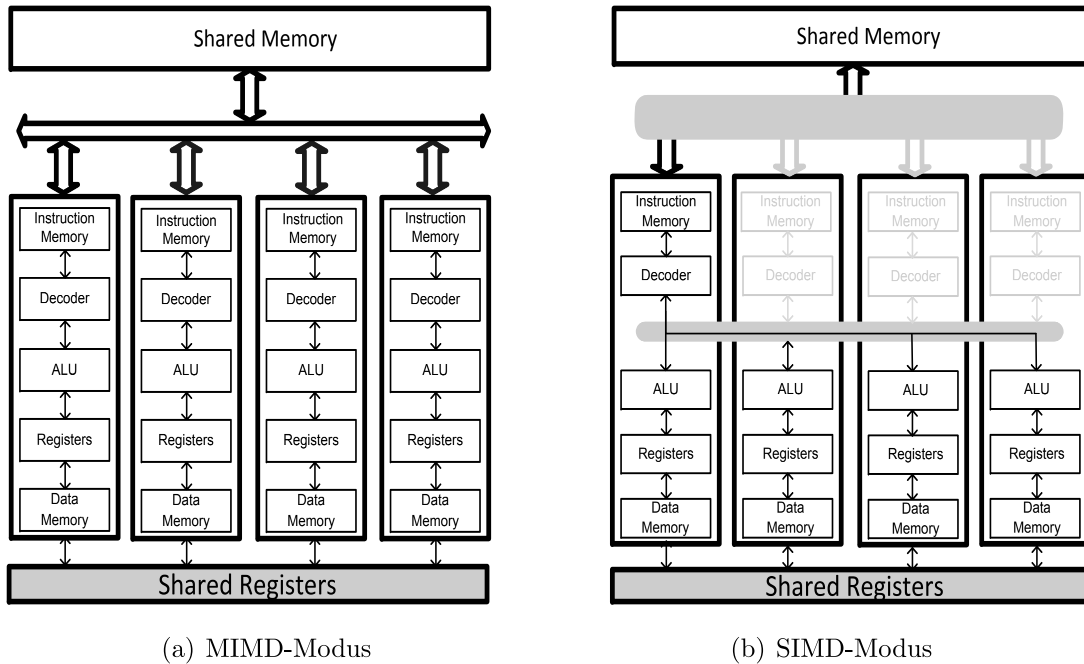


Abbildung 6.12: Betriebsmodi der rekonfigurierbaren QuadroCore-Architektur [Pur09]

kann in einem Taktzyklus durchgeführt werden. Die Synchronisation der Prozessorelemente im asynchronen MIMD-Betrieb wird durch *Barrier*-Instruktionen realisiert. Das schnelle Umschalten zwischen den verschiedenen Betriebsmodi ermöglicht die Kombination aller Granularitätsebenen (ILP, DLP, TLP), um einen Algorithmus möglichst effizient zu parallelisieren. Für die Kommunikation der Prozessorelemente untereinander steht neben einem gemeinsamen Speicher (*shared memory*) auch eine gemeinsam zugreifbare Registerbank (*shared registers*) zur Verfügung (siehe Abbildung 6.13). Die 32 globalen Register ermöglichen einen sehr schnellen Datenaustausch. Wird der QuadroCore, wie in Abbildung 6.13b dargestellt, im SIMD-Modus betrieben, können drei der vier Instruktionsdekoder deaktiviert werden. Hierdurch arbeitet der QuadroCore im SIMD-Modus sehr energieeffizient.



(a) MIMD-Modus (b) SIMD-Modus

Abbildung 6.13: QuadroCore-Architektur [PPP08]

Um die Multiplikation $a(x) \cdot b(x) = c(x)$ im Binärkörper $GF(2^n)$ zu berechnen wird wieder die Polynombasis-Darstellung (vgl. Kapitel 3.3) herangezogen, so dass $a(x) = \sum_{i=0}^{n-1} a_i x^i$ und $b(x) = \sum_{i=0}^{n-1} b_i x^i$ bei der Abbildung auf Wort-Ebene ($w = 32$) genau $A(x) = \sum_{j=0}^{t-1} A_j x^{jw}$ und $B(x) = \sum_{j=0}^{t-1} B_j x^{jw}$ mit $t = \lceil n/w \rceil$ entsprechen. Die Binärkörper-Multiplikation $A(x) \cdot B(x) = C(x)$ wurde mit Hilfe des iterativen Karatsuba-Verfahrens (vgl. Kapitel 3.3.2) in $3^{(\log_2 t)}$ Teilmultiplikationen auf 32-Bit-Wort-Ebene zerlegt. Die einzelnen Teilmultiplikationen können unabhängig voneinander ausgeführt werden. Tabelle 6.4 zeigt die Parallelisierung der Multiplikation im Binärkörper $GF(2^{233})$. Vor jeder partiellen Multiplikation werden die Faktoren durch Summation von A_j bzw. B_j gemäß der zweiten Spalte von Tabelle 6.4 berechnet. Diese Initialisierung ist für jede Teilmultiplikation individuell und wird daher im MIMD-Modus des QuadroCores ausgeführt. Sind die Faktoren berechnet, kann in den SIMD-Modus umgeschaltet werden, da der Programmcode (`ff_mul_w32_s3`) für alle Teilmultiplikationen identisch ist. Das Ergebnis der partiellen Multiplikation auf Wort-Ebene ist ein Polynom der Länge $2w - 1$, welches in einem oberen 32-Bit-Datenwort H und einem unteren 32-Bit-Datenwort L abgelegt wird. Die partiellen

PE	**sum of input words j**	**partial words of the resulting product**															
		C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0
PE 1	0								H	LH	LH	LH	LH	LH	LH	LH	L
	0,1								H	L	H	L	H	L	H	L	
	1							H	LH	LH	LH	LH	LH	LH	LH	L	
	0,2								H	LH	L		H	LH	L		
	0,1,2,3								H	L			H	L			
	1,3							H	LH	L		H	LH	L			
PE 2	2						H	LH	LH	LH	LH	LH	LH	LH	L		
	2,3						H	L	H	L	H	L	H	L			
	3					H	LH	LH	LH	LH	LH	LH	LH	L			
	0,4								H	LH	LH	LH	L				
	0,1,4,5								H	L	H	L					
	1,5							H	LH	LH	LH	L					
	0,2,4,6								H	LH	L						
PE 3	0,1,2,3,4,5,6,7								H	L							
	1,3,5,7							H	LH	L							
	2,6						H	LH	LH	LH	L						
	2,3,6,7						H	L	H	L							
	3,7					H	LH	LH	LH	L							
	4				H	LH	LH	LH	LH	LH	LH	LH	L				
	4,5				H	L	H	L	H	L	H	L					
PE 4	5			H	LH	LH	LH	LH	LH	LH	LH	L					
	4,6				H	LH	L		H	LH	L						
	4,5,6,7				H	L			H	L							
	5,7			H	LH	L		H	LH	L							
	6		H	LH	LH	LH	LH	LH	LH	LH	L						
	6,7		H	L	H	L	H	L	H	L							
	7	H	LH	LH	LH	LH	LH	LH	LH	L							

Tabelle 6.4: Parallelisierte Karatsuba-Multiplikation im Binärkörper $GF(2^{233})$ [PPP08]

Ergebnisse H und L werden, entsprechend Tabelle 6.4, an der jeweiligen Stelle C_v auf das Gesamtergebnis aufaddiert. Der Tabelleneintrag LH bedeutet hierbei, dass beide Teilergebnisse bzw. die Summe von L und H zum Gesamtergebnis addiert wird. Das Aufaddieren der partiellen Ergebnisse wird wieder im MIMD-Modus des QuadroCores ausgeführt.

Für den konkreten Fall einer Multiplikation im Binärkörper $GF(2^{233})$ werden, bei einer Datenwortbreite von $w = 32$, jeweils $t = \lceil n/w \rceil = 8$ Datenworte für die Faktoren $A(x)$ und $B(x)$ sowie $2t = 16$ Datenworte für das unreduzierte Ergebnis in Polynombasis-Darstellung verwendet. Das erste Prozessorelement (PE1) berechnet sechs der insgesamt $3^{(\log_2 t)} = 27$ Teilmultiplikationen auf Wort-Ebene, die anderen Prozessorelemente (PE2, PE3, PE4) führen jeweils sieben Teilmultiplikationen aus. Um das Prinzip der Parallelisierung zu verdeutlichen, wird die letzte Teilmultiplikation von Prozessorelement 1 (sechste Zeile in Tabelle 6.4) näher erläutert. Mit dieser partiellen Multiplikation auf Wort-Ebene wird der Term $(A_1 + A_3) \cdot (B_1 + B_3)$ berechnet. Die oberen 32 Bit (H) des 64-Bit-Teilergebnisses werden zu dem Datenwort C_9 und C_5 des Gesamtergebnisses addiert. Die unteren 32 Bit (L) des Teilergebnisses werden entsprechend der Tabelle zum Datenwort C_7 und C_3 des Gesamtergebnisses addiert. Die Summe $L + H$ wird schließlich zum Datenwort C_8 bzw. C_4 addiert. Zur Erinnerung sei hier nochmals darauf hingewiesen, dass eine Addition im erweiterten Binärkörper $GF(2^n)$ durch eine logische `XOR`-Operation vereinfacht werden kann, d. h. es können keinerlei Überträge beim Aufaddieren der partiellen Ergebnisse auf das Gesamtergebnis auftreten.

Die parallelisierte Karatsuba-Methode zur Multiplikation im Binärkörper $GF(2^{233})$ wurde zunächst komplett im asynchronen MIMD-Modus untersucht. Anschließend wurde die hybride Variante analysiert, bei welcher die 27 partiellen Multiplikationen auf Wort-Ebene jeweils im synchronen SIMD-Modus ausgeführt werden. Der Ressourcenbedarf des QuadroCores im Hinblick auf Chipfläche, Verlustleistung und Ausführungszeit für die verschiedenen Betriebsmodi ist in Tabelle 6.6 von Kapitel 6.2.3 dargestellt. Ferner wird in diesem Abschnitt die Bewertung mit Hilfe der Ressourceneffizienz diskutiert.

6.2.2 CoreVA

Der VLIW-Prozessor CoreVA (*Configurable, resource efficient VLIW Architecture*) wurde ebenfalls am Fachgebiet Schaltungstechnik der Universität Paderborn entwickelt [Jun11]. Im Gegensatz zum QuadroCore basiert der CoreVA-Prozessor nicht auf der S-Core-Architektur, sondern beruht komplett auf einer eigenständigen Entwicklung. Wie in Abbildung 6.14 dargestellt ist, besitzt der CoreVA-Prozessor eine klassische Harvard-Architektur mit getrenntem Instruktionsbus (4×32 Bit) und Datenbus (2×32 Bit). Die sechsstufige Pipeline-Struktur verfügt über vier parallele Recheneinheiten (ALU) in der Ausführungsstufe (*EX, engl. execute stage*). Jeweils zwei Recheneinheiten teilen sich einen Multiplizierer und einen Dividierer. Ferner sind zwei der vier Verarbeitungseinheiten mit einem dedizierten Speichercontroller (*LD/ST, engl. load/store unit*) ausgestattet, um auf den Datenspeicher oder eng gekoppelte Hardware-Beschleuniger zuzugreifen. Der Software stehen einunddreißig 32-Bit-Register zur allgemeinen Ausführung sowie zwei 8-Bit-Register zur bedingten Ausführung (*condition register*) zur Verfügung. Genau wie der S-Core wurde der CoreVA ebenfalls in der Prozessor-Spezifikationssprache UPSLA beschrieben. Mit Hilfe der UPSLA-Modellierung kann somit auch für den CoreVA-Prozessor die

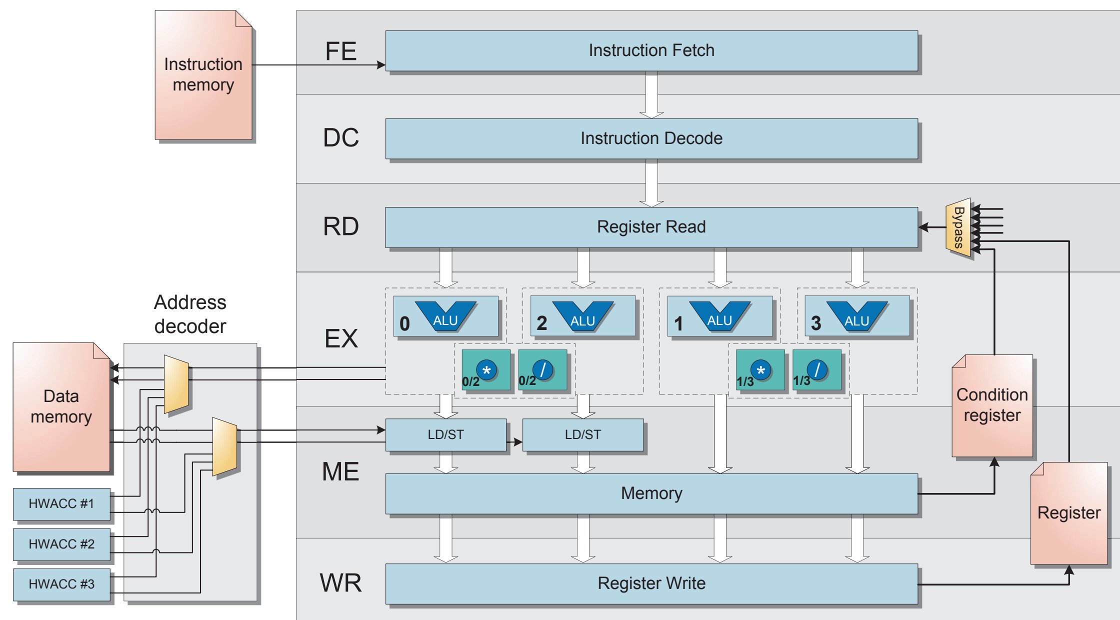


Abbildung 6.14: Pipeline-Struktur des VLIW-Prozessors CoreVA [Jun11]

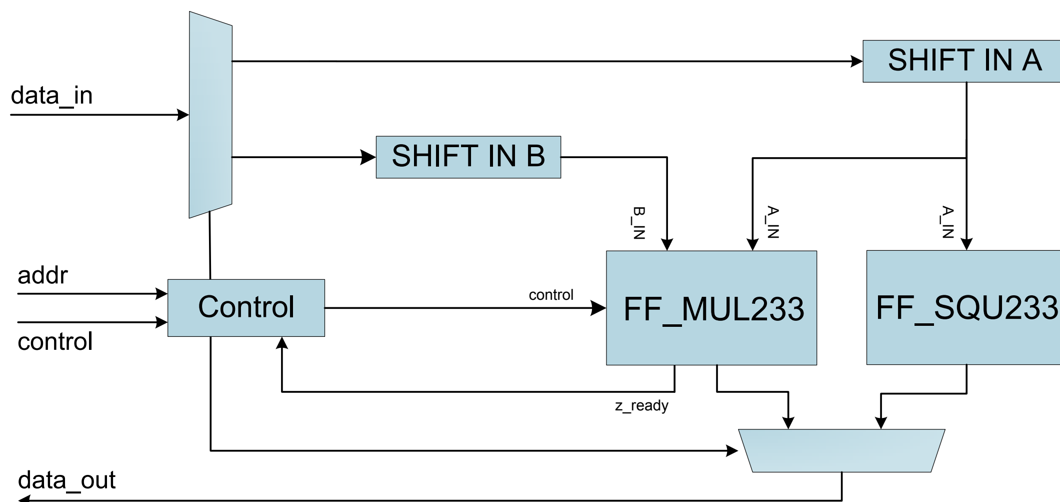


Abbildung 6.15: Struktur des Hardware-Beschleunigers im CoreVA-Prozessor [JPD+10]

komplette Software-Werkzeugkette (*Compiler*, *Assembler*, *Linker*, *Simulator* und *Profiler*) automatisch generiert werden (vgl. Abschnitt 6.1.1).

Der CoreVA-Prozessor unterstützt die Anbindung von eng gekoppelten Hardware-Beschleunigern. Ähnlich wie bei dem S-Core-basierten Mikroprozessorsystem (vgl. Abbildung 4.5) werden die Hardware-Beschleuniger in den globalen Adressraum des Prozessors eingeblendet (*memory mapped I/O*). Hierdurch kann die Software mit äußerst geringer Latenz auf die internen Register der Hardware-Beschleuniger zugreifen. Abbildung 6.15 zeigt die Anbindung eines Hardware-Beschleunigers für Kryptographie mit elliptischen Kurven. Zur Beschleunigung der Skalarmultiplikation wurden die Multiplikation und die Quadrierung im Binärkörper $GF(2^{233})$ als dedizierte Hardware-Module implementiert. Die entsprechenden Blöcke `FF_MUL233` bzw. `FF_SQU233` wurden dabei mit Hilfe des in Abschnitt 5.2.4 vorgestellten Codegenerators erzeugt. Die Operanden `A_IN` bzw. `B_IN` werden vom Prozessor durch jeweils acht 32-Bit-Schreibbefehle in die zugehörigen Schieberegister eingetaktet. Über die Kontrolllogik können die Rechenoperationen gestartet bzw. deren Status abgefragt werden. Nach Abschluss der Berechnung wird das bereits reduzierte Ergebnis wiederum durch acht 32-Bit-Lesebefehle vom CoreVA-Prozessorkern zurückgelesen.

Neben dem eng gekoppelten Hardware-Beschleuniger gemäß Abbildung 6.15 wurden beim CoreVA auch Superinstruktionen sowie der Einfluss von handoptimiertem

Programmcode untersucht. Die Instruktionspaare wurden wie beim S-Core mit Hilfe der in Abschnitt 6.1.1 vorgestellten Entwurfsmethodik identifiziert. Als Ergebnis [JPD+10] wurden die folgenden Superinstruktionen im CoreVA-Prozessorkern implementiert:

- `LSLXOR` – logisches Linksschieben und XOR-Verknüpfung
- `LSRXOR` – logisches Rechtsschieben und XOR-Verknüpfung
- `MVBITS` – Extraktion eines Teilworts aus einem 32-Bit-Datenwort

Des Weiteren wurde der vom Übersetzer generierte Programmcode auf Assembler-Ebene analysiert. Durch fundierte Kenntnisse des Befehlssatzes und der zugrundeliegenden Hardware-Architektur konnten einzelne Programmabschnitte von Hand zusätzlich optimiert werden. Besonders die automatische Parallelisierung von sequentiellen Programmteilen bereitet dem Übersetzer zuweilen Schwierigkeiten. Durch die manuelle Optimierung auf Assembler-Ebene konnte häufig eine effektivere Auslastung aller vier Verarbeitungseinheiten erreicht werden. Ferner konnten durch gezieltes Umsortieren bestimmter Codesequenzen unnötige Wartezyklen vermieden werden.

Die Auswirkungen der verschiedenen Optimierungen sind in Tabelle 6.5 dargestellt. Alle Angaben zur Chipfläche basieren auf einer 65-nm-CMOS-Standardzellentechnologie [STM08]. Es handelt sich um Synthese-Ergebnisse, welche unter *Worst-Case*-Bedingungen (1,1 V bei 125 °C) mit einer Zielfrequenz von 200 MHz erreicht wurden. Die Spalte Variante gibt an, welche Hardware-Software-Kombination untersucht wurde. Als Software-Variante kommt entweder der automatisch übersetzte C-Code

Variante		**Chip-fläche**	**Verlust-leistung**	**Skalarmultiplikation**		**Körpermultiplikation**	
SW	**HW**			**Takte**	**Code***	**Takte**	**Code***
C	STD	0,20 mm²	3,73 mW	3 552 571	10 336	2 111	2 052
ASM				2 941 035	8 752	1 839	2 068
C	ISE1	0,20 mm²	3,78 mW	3 408 875	8 656	2 018	1 940
ASM				2 622 409	8 576	1 636	1 924
C	ISE2	0,26 mm²	4,83 mW	260 615	6 732	73	2 052

* Codegröße in Byte

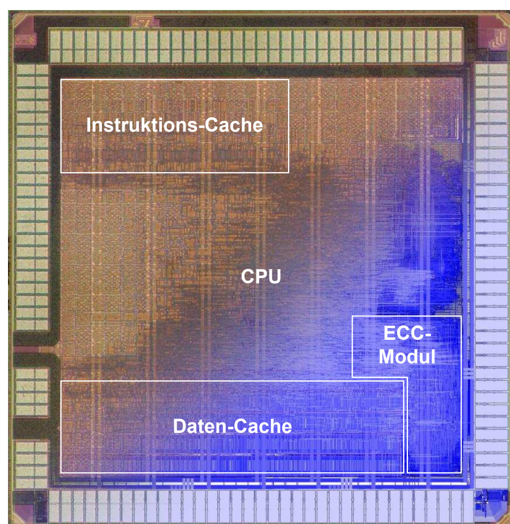
Tabelle 6.5: Ressourcenbedarf der Hardware-Software-Kombinationen [JPD+10]

oder der handoptimierte Assembler-Code zum Einsatz. Die Hardware-Varianten bilden der unmodifizierte CoreVA-Standardprozessor (STD), die Version mit erweitertem Instruktionssatz (ISE1) sowie der Hardware-Beschleuniger (ISE2) nach Abbildung 6.15. Mit Hilfe des automatisierten Entwurfsablaufes (vgl. Abbildung 6.2) wurden die Ausführungszeit und Codegröße für die Multiplikation im Binärkörper $GF(2^{233})$ sowie für die Skalarmultiplikation $k\mathcal{P}$ mit $k = 2^{232}, \mathcal{P} \in GF(2^{233})$ bestimmt.

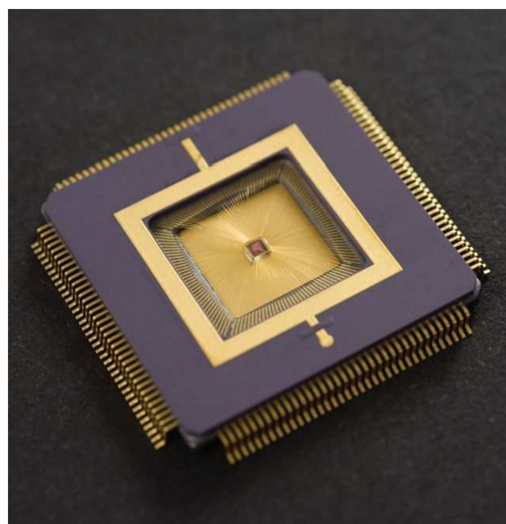
Verglichen mit den Ergebnissen des S-Core-Einzelkernprozessors (vgl. Tabelle 5.2 bzw. 6.2), erreicht der automatisch übersetzte Programmcode auf dem vierfach parallel rechnenden CoreVA-Prozessor eine durchschnittliche Beschleunigung von Faktor 3,1. Aufgrund der VLIW-Architektur besitzen die automatisch übersetzten CoreVA-Programme durchschnittlich das 2,4-fache an Codegröße gegenüber den entsprechenden S-Core-Funktionen. Im Vergleich zum übersetzten C-Code konnte durch die Handoptimierung auf Assembler-Ebene die Ausführungszeit durchschnittlich um 18 % und die Codegröße durchschnittlich um 4 % gesenkt werden. Ferner kann festgehalten werden, dass die Integration von Superinstruktionen (ISE1) beim CoreVA genau wie beim S-Core (vgl. Abbildung 6.7) ohne bemerkenswerte Vergrößerung der Chipfläche möglich ist. Durch eine Vergrößerung der Chipfläche um 30 % (ISE2) kann die Ausführung der Körpermultiplikation um den Faktor 28,9 und die Berechnung der Skalarmultiplikation um den Faktor 13,6 beschleunigt werden.

Die Hardware-Variante CoreVA (ISE2) wurde als ASIC in einer 65-nm-CMOS-Standardzellentechnologie realisiert [Jun11]. Wie in Abbildung 6.16a zu erkennen ist, verfügt der CoreVA-Chip neben dem Hardware-Beschleuniger (ECC-Modul) zusätzlich über einen Instruktions- und Daten-Cache. Hierbei handelt es sich jeweils um eine 16-KB-große, direkt-abbildende Cache-Architektur. Der Daten-Cache verfolgt eine *Write-Back*-Strategie und besitzt zwei unabhängige 32-Bit-Schreib- bzw. Lese-Ports. Der CoreVA-Chip kann unter typischen Betriebsbedingungen (1,2 V bei 25 °C) mit einer Taktfrequenz von 400 MHz betrieben werden. Der ASIC besitzt eine *Die*-Fläche, d. h. Chipfläche inklusive Pad-Ring, von 2,64 mm^2. Die gemessene Leistungsaufnahme bei der Berechnung der Skalarmultiplikation beträgt 197 mW. Abbildung 6.16b zeigt den CoreVA-Prototypen in einem CQFP[7]-Gehäuse mit 144 Ein bzw. Ausgängen und freigelegtem ASIC-Die.

[7]Ceramic Quad Flat Package



(a) Die-Foto mit ASIC-Layout



(b) Prototyp im Keramikgehäuse

Abbildung 6.16: Fabrizierter CoreVA-VLIW-Prozessor [Jun11]

6.2.3 Ressourceneffizienz der Parallelisierung

Genau wie der CoreVA-Prozessor wurden ebenfalls der S-Core sowie der QuadroCore in einer 65-nm-CMOS-Standardzellentechnologie [STM08] abgebildet. Die Synthese wurde jeweils unter *Worst-Case*-Bedingungen (1,1 V bei 125 °C) mit einer Zielfrequenz von 200 MHz durchgeführt. Tabelle 6.6 zeigt den Ressourcenbedarf für

<table>
<tr><th>Architektur</th><th>Variante</th><th>Chipfläche</th><th>Taktzyklen</th><th>Verlustleistung</th></tr>
<tr><td>S-Core</td><td>STD</td><td>0,04 mm²</td><td>6 498</td><td>1,38 mW</td></tr>
<tr><td>S-Core</td><td>ISE1</td><td>0,04 mm²</td><td>5 545</td><td>1,42 mW</td></tr>
<tr><td>QuadroCore</td><td>MIMD</td><td rowspan="2">0,23 mm²</td><td>3 193</td><td>4,23 mW</td></tr>
<tr><td>QuadroCore</td><td>MIMD/SIMD</td><td>3 337</td><td>3,82 mW</td></tr>
<tr><td>CoreVA</td><td>STD</td><td rowspan="2">0,20 mm²</td><td>1 839</td><td>3,73 mW</td></tr>
<tr><td>CoreVA</td><td>ISE1</td><td>1 636</td><td>3,78 mW</td></tr>
</table>

Tabelle 6.6: Ressourcenbedarf verschiedener Architektur-Varianten für eine Multiplikation im erweiterten Binärkörper $GF(2^{233})$ [Pur09][JPD+10][Jun11]

eine Multiplikation im Binärkörper $GF(2^{233})$. Beim QuadroCore wurden die in Abschnitt 6.2.1 vorgestellten Ausführungsmodi (MIMD, SIMD) untersucht. Der S-Core und der CoreVA wurden jeweils als ursprüngliche Standard-Variante (STD) sowie unter Berücksichtigung der hinzugefügten Superinstruktionen (ISE1) ausgewertet.

Bei der Chipfläche fällt auf, dass der QuadroCore-Mehrkernprozessor um Faktor 1,4 größer ist als die Chipfläche, welche vier einzelne S-Core-Prozessoren benötigen würden. Die zusätzliche Chipfläche wird durch die Implementierung der rekonfigurierbaren Verbindungsstrukturen und der gemeinsam genutzten Registerbank verursacht. Auch der vierfach parallel arbeitende VLIW-Prozessor CoreVA ist um 25 % größer als vier S-Core-Einzelkernprozessoren.

Weiterhin kann festgestellt werden, dass der QuadroCore wesentlich mehr Taktzyklen zur Berechnung der Binärkörper-Multiplikation benötigt als der CoreVA-Prozessor. Obwohl beide Mehrkernprozessoren vier parallele Ausführungseinheiten besitzen, erreicht der QuadroCore nur einen Beschleunigungsfaktor von 2,0 gegenüber dem S-Core, während der CoreVA im direkten Vergleich mit dem S-Core einen Beschleunigungsfaktor von 3,5 erzielt. Durch das Hinzufügen der rekonfigurierbaren Verbindungsstrukturen weist der QuadroCore eine größere Speicherlatenz auf. Dieses wird deutlich, wenn man das S-Core-Programm auf einem der vier Prozessorkerne des QuadroCores ausführt und die restlichen drei Prozessorkerne deaktiviert. Die Ausführung auf einem dediziertem QuadroCore-Prozessorkern benötigt anstatt 6498 Takte ganze 9402 Takte [Pur09].

Interessant ist beim QuadroCore die Reduktion der Verlustleistung durch Nutzung des hybriden MIMD/SIMD-Ausführungsmodus. Durch Deaktivierung der nicht benutzten Funktionsblöcke im SIMD-Modus (vgl. Abschnitt 6.2.1) kann die Leistungsaufnahme um 9,7 % gesenkt werden. Allerdings benötigt der hybride Ausführungsmodus 4,5 % mehr Taktzyklen als der reine MIMD-Modus, so dass die Energieeinsparung lediglich 5,6 % beträgt [PPP08].

Um die verschiedenen Hardware-Software-Kombinationen anschaulich miteinander vergleichen zu können, wird die in Kapitel 4.3 vorgestellte Berechnung der Ressourceneffizienz herangezogen. Abbildung 6.17 zeigt die normierte Ressourceneffizienz von S-Core, QuadroCore und CoreVA für die Berechnung der Multiplikation im Binärkörper $GF(2^{233})$. Es kommen hierbei die gleichen Gewichtungen wie bei der

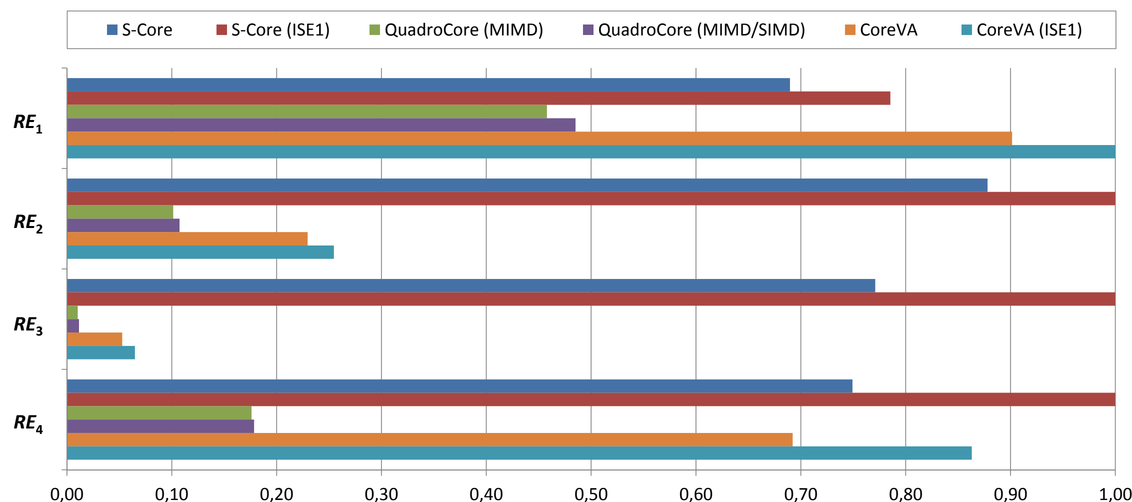


Abbildung 6.17: Ressourceneffizienz von S-Core, QuadroCore und CoreVA im Vergleich

Analyse der Ressourceneffizienz von Instruktionssatzerweiterungen (vgl. Tabelle 6.3) zum Einsatz:

- RE_1: Power-Delay-Produkt
- RE_2: Gleichgewichtung aller Ressourcen
- RE_3: Anwendungsszenario Chipkarte
- RE_4: Anwendungsszenario Sicherheitsserver

Wie Abbildung 6.17 zeigt, besitzt die Variante CoreVA (ISE1) die beste Ressourceneffizienz hinsichtlich des Power-Delay-Produkts. Bei dieser Bewertungsmetrik wird die Chipfläche allerdings komplett vernachlässigt. Berücksichtigt man diese, ändert sich das Ergebnis zugunsten des S-Cores. Bei Bewertungsmetrik RE_2, RE_3 und RE_4 erreicht die Hardware-Software-Kombination S-Core (ISE1) die beste Ressourceneffizienz. Der Grund dafür ist, dass die Parallelisierung nicht ideal skaliert. Wie zuvor erläutert, schlägt sich bei Mehrkernprozessoren der zusätzliche Aufwand an Kommunikation und Synchronisation auf alle Ressourcen negativ nieder. Bei gleichen Betriebsbedingungen, im Sinne von Versorgungsspannung, Taktfrequenz und Standardzellen-Technologie, ist es daher sehr schwierig, durch Parallelisierung eine bessere Ressourceneffizienz zu erzielen als ein vergleichbarer Einzelkernprozessor. Allerdings kann der CoreVA-Mehrkernprozessor die gleiche Performanz auch bei einer geringeren Versorgungsspannung (d. h. geringere Verlustleistung) bzw. eine bessere Performanz bei einer höheren Taktfrequenz (d. h. geringere Ausführungszeit) erreichen, womit die Ressourceneffizienz der des S-Cores überlegen ist [Jun11].

6.3 Coprozessor-Modul

In den vorherigen Kapiteln wurden Prozessor-basierte Hardware-Software-Kombinationen für Kryptographie mit elliptischen Kurven untersucht. Im folgenden Abschnitt wird eine dedizierte Hardware-Lösung zur Berechnung der Skalarmultiplikation vorgestellt. Im Gegensatz zu den bislang präsentierten Instruktionssatzerweiterungen kann dieses Coprozessor-Modul vollkommen eigenständig agieren und benötigt keine direkte Verbindung zum Hauptprozessor. Es wird daher auch von einem lose gekoppelten Hardware-Beschleuniger besprochen.

Im allgemeinen sind Coprozessor-Module für eine bestimmte Aufgabe optimiert. In dem hier betrachtetem Beispiel kann das Coprozessor-Modul die Skalarmultiplikation um ein Vielfaches schneller berechnen als alle bislang betrachteten Einzel- oder Mehrkernprozessoren. Um die enorme Performanz effektiv zu nutzen, ist es daher sinnvoll, das Coprozessor-Modul in ein Prozessorfeld (vgl. Abbildung 4.6) bzw. in ein Multiprozessorsystem (vgl. Abbildung 4.7) zu integrieren. Auf diese Weise können mehrere Prozessorkerne von der Performanz des Coprozessor-Moduls profitieren.

Abbildung 6.18 zeigt die Architektur des Coprozessor-Moduls zur Berechnung der Skalarmultiplikation [PS07]. Neben der Registerbank zur Speicherung der Daten verfügt der Coprozessor über einen Block zum Multiplizieren, Quadrieren und Addieren von Elementen eines Binärkörpers $GF(2^n)$. Die modulare Struktur des Coprozessor-Moduls ermöglicht eine generische Unterstützung verschiedener Binärkörper. Die folgenden Untersuchungen basieren auf dem bereits zuvor beschriebenen Binärkörper $GF(2^{233})$ gemäß Anhang A. Die Hardware-Beschreibung des Multiplizierers und des Quadrierers wurde mit dem in Abschnitt 5.2.4 vorgestellten Codegenerator

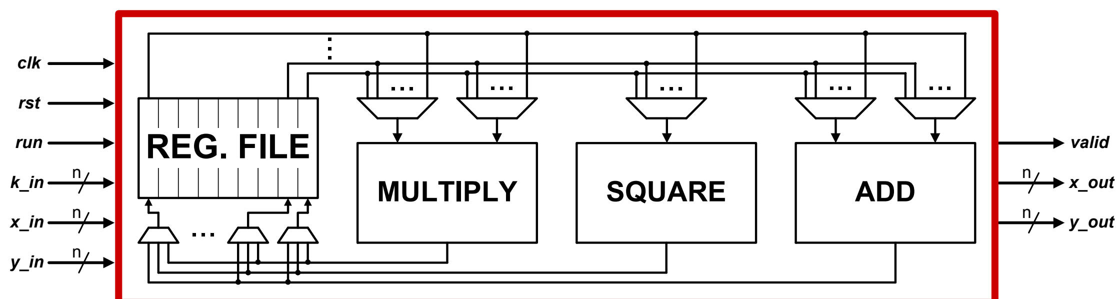


Abbildung 6.18: Struktur des Coprozessors für Kryptographie mit elliptischen Kurven

erzeugt. Der Addierer lässt sich durch eine einfache `XOR`-Operation realisieren. Zur Berechnung der Skalarmultiplikation $k\mathcal{P} = (x_{out}, y_{out})$ werden die Eingangsdaten k und $\mathcal{P} = (x_{in}, y_{in})$ an das Coprozessor-Modul angelegt. Die Skalarmultiplikation wird auf Basis von Algorithmus 2 mit Hilfe von mehreren, verschachtelten Zustandsautomaten (FSM) berechnet. Die Berechnung gliedert sich im Wesentlichen in drei Schritte. Zunächst wird der affine Eingabepunkt $\mathcal{P}$ in das projektive Koordinatensystem nach López-Dahab konvertiert (vgl. Kapitel 3.2.2). Anschließend werden für jedes Bit des Eingabevektors k die Punktverdopplung (Gleichung 3.5) und Punktaddition (Gleichung 3.6) mittels des in Abbildung 6.19 dargestellten Datenflussgraphen berechnet. Zum Schluss wird das Ergebnis wieder in das affine Koordinatensystem zurück transformiert.

Wie in Abbildung 6.19 dargestellt ist, wird die Punktaddition und Punktverdopplung quasi parallel berechnet. Während sowohl die Addition $\textcircled{+}$ als auch die Quadrierung $\textcircled{\hat{}2}$ jeweils nur einen Takt benötigen, weist die Multiplikation $\textcircled{*}$ eine Latenz von drei Takten auf. Da zur Punktverdopplung zwei Multiplikationen und zur Punktaddition vier Multiplikationen notwendig sind, beträgt die Ausführungszeit für jede Iteration genau 18 Takte. Inklusive der Koordinatentransformationen kann das Coprozessor-Modul eine Skalarmultiplikation im Binärkörper $GF(2^{233})$ in-

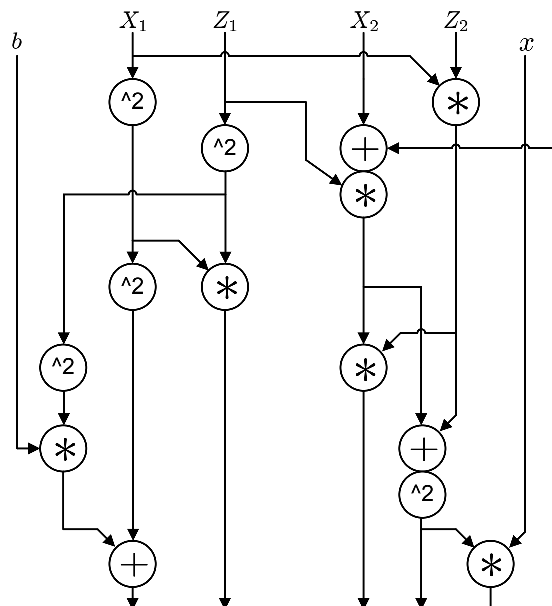


Abbildung 6.19: Datenflussgraph zur Berechnung der Skalarmultiplikation [AH08]

nerhalb von 4498 Takten berechnen [PS07][PSPR08]. Das Coprozessor-Modul wurde sowohl auf eine 65-nm-CMOS-Standardzellentechnologie [IFX06] als auch auf einen Xilinx-FPGA vom Typ XC2V8000-4 [Xlnx07] abgebildet. Tabelle 6.7 zeigt den Ressourcenbedarf für die jeweilige Zieltechnologie.

Zieltechnologie	**Takt-frequenz**	**Flächen-ressourcen**	**Ausführungs-dauer**	**Verlust-leistung**
ASIC 65 nm	625 MHz	0,279 mm^2	7,20 μs	75,79 mW
FPGA XC2V8000-4	50 MHz	15 365 Slices	89,96 μs	—

Tabelle 6.7: Ressourcenbedarf des Coprozessor-Moduls für zwei Zieltechnologien

Vergleicht man die Performanz des Coprozessor-Moduls mit jener der Hardware-Software-Kombination CoreVA (ISE2), so stellt sich heraus, dass bei annähernd gleichem Flächenbedarf das Coprozessor-Modul für die Berechnung der Skalarmultiplikation 58-mal weniger Taktzyklen benötigt. Obwohl das Coprozessor-Modul auf den ersten Blick eine hohe Verlustleistung aufweist, ist auch der Energiebedarf zur Berechnung der Skalarmultiplikation mit 0,546 μJ um Faktor 11,5 geringer als der von Hardware-Software-Kombination CoreVA (ISE2).

Das Coprozessor-Modul kann mit Hilfe von generischen Schnittstellen-Adaptern, so genannten *I/O-Wrapper* [NPPR07], an verschiedenste Bussysteme angeschlossen werden [PSPR08]. Dadurch ist es möglich, das Coprozessor-Modul z.B. direkt an den Prozessorsystembus eines S-Core-Mikroprozessorsystems (vgl. Abbildung 4.5) anzuschließen. Die Kopplung an den WISHBONE-Bus des S-Core-Prozessorfeldes (vgl. Abbildung 4.6) ist ebenfalls mit Hilfe der I/O-Wrapper sehr leicht realisierbar. Um das Coprozessor-Modul von möglichst vielen Prozessoren auszulasten, wird im Folgenden die Integration in ein Multiprozessor-System-on-Chip (MPSoC) untersucht. Hierzu wird das Coprozessor-Modul über den I/O-Wrapper an ein On-Chip-Netzwerk (vgl. Abbildung 4.7) gekoppelt. Jeder Prozessor innerhalb dieser MPSoC-Architektur kann somit von dem Coprozessor-Modul profitieren. Da sämtliche Kommunikation zwischen den Prozessorelementen und dem Coprozessor-Modul über das On-Chip-Netzwerk abgewickelt wird, haben dessen Eigenschaften einen wesentlichen Einfluss auf die Performanz des Gesamtsystems. Aus diesem Grund wird im folgenden Abschnitt eine Performanzanalyse des zugrundeliegenden On-Chip-Netzwerk vorgestellt.

6.3.1 Performanzanalyse des On-Chip-Netzwerks

Eng gekoppelte Hardware-Beschleuniger, wie beispielsweise die in Abschnitt 6.1 vorgestellten Instruktionssatzerweiterungen, sind direkt im jeweiligen Prozessorkern integriert, so dass der Kommunikationsaufwand praktisch keinen Einfluss auf die Performanz aufweist. Bei lose gekoppelten Hardware-Beschleunigern, wie dem hier betrachteten Coprozessor-Modul, können die Eigenschaften der Kommunikationsinfrastruktur die Performanz des Gesamtsystems jedoch erheblich beeinflussen. Wenn das Coprozessor-Modul, wie in Abbildung 4.7 dargestellt, an das On-Chip-Netzwerk einer MPSoC-Architektur angeschlossen wird, haben primär die Datenübertragungsrate und die Latenz der zugrundeliegenden Kommunikationsinfrastruktur einen Einfluss auf die Performanz des Gesamtsystems.

Die Datenübertragungsrate des eingesetzten On-Chip-Netzwerkes wird bereits in [Nie08] ausführlich untersucht. Wie sich auch in Kapitel 4.5 dieser Arbeit herausstellt, kann die hier betrachtete Konfiguration in einer 90-nm-CMOS-Technologie eine maximale Nettodatenübertragungsrate von 42,29 GBit/s pro Switch-Box-Port erreichen [PNPR07]. Im Folgenden steht daher die Analyse der Latenz des On-Chip-Netzwerks im Fokus.

Zur Bestimmung der Latenz wurden spezielle Analyse-Module an bestimmte Stellen des On-Chip-Netzwerks angeschlossen. Diese Analyse-Blöcke, so genannte *Listener*, überwachen die Kommunikation im On-Chip-Netzwerk. Um auf eine bestimmte Datensequenz reagieren zu können, müssen die Listener das Kommunikationsprotokoll des On-Chip-Netzwerkes beherrschen. Zur Generierung der Listener wurde die explizit für diesen Aufgabenbereich entwickelte Programmiersprache SiLLis[8] [GS09] verwendet. SiLLis wurde am Institut für Elektronik und Informationstechnik der Politecnico di Milano entworfen und ist an die Hochsprache C angelehnt. Durch die hohe Abstraktionsebene lassen sich mit Hilfe von SiLLis komplexe Kommunikationsprotokolle einfach spezifizieren (vgl. Anhang B). Der SiLLis-Übersetzer verwandelt den Programmcode in synthetisierbaren VHDL-Code, welcher dann sowohl in einer Simulationsumgebung integriert als auch auf einem FPGA-basierten Rapid-Prototyping-System abgebildet werden kann. Mittels dieser Methodik können statistische Daten auch in großen Systemen komfortabel erhoben werden [PPG+10].

[8] Simplified Language for Listeners

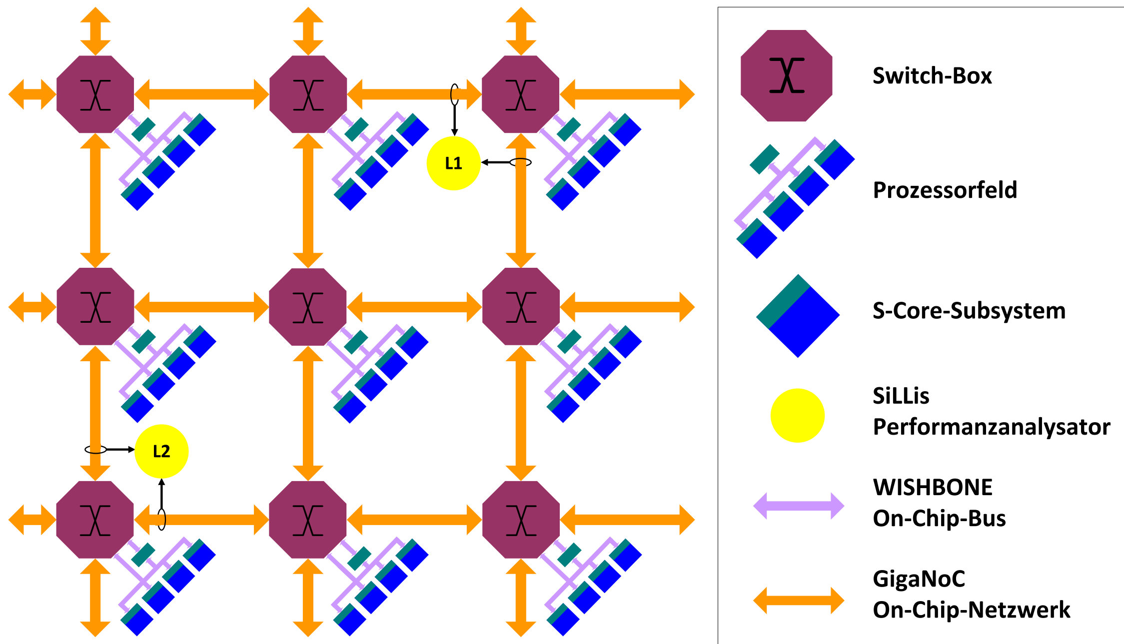


Abbildung 6.20: Einsatz von SiLLis zur Performanzanalyse des On-Chip-Netzwerks

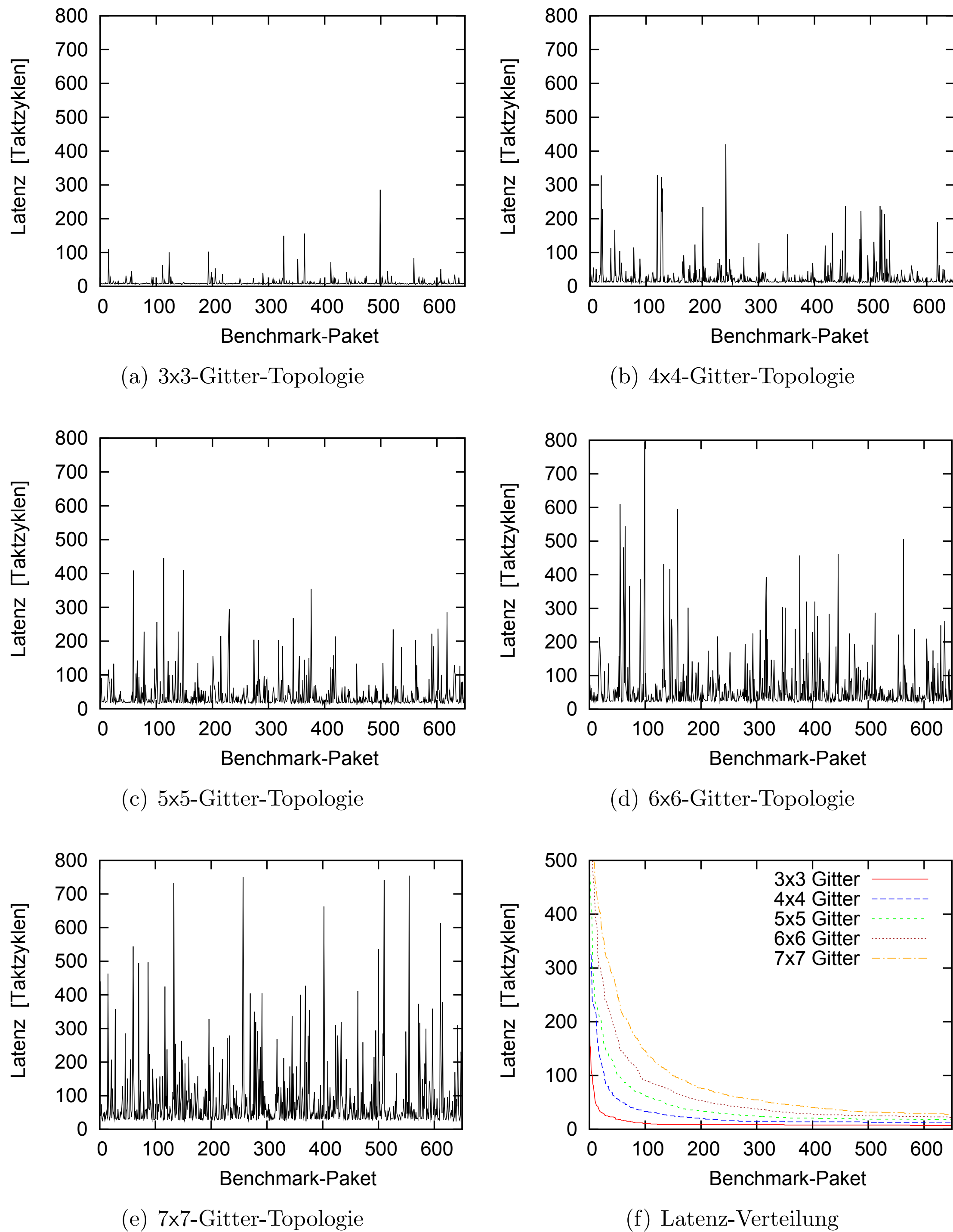
Abbildung 6.20 zeigt das zu untersuchende On-Chip-Netzwerk und die daran angeschlossenen Instanzen des SiLLis-basierten Performanzanalysators [GSP+09]. Die Latenzanalyse wird in verschiedenen Netzwerkgrößen mit einer NxN-Gitter-Topologie durchgeführt. Wie in Abbildung 6.20 dargestellt ist, überwacht jeweils ein SiLLis-Performanzanalysator die On-Chip-Netzwerkverbindungen der oberen rechten sowie der unteren linken Switch-Box. Da alle Switch-Boxen ein statisches XY-Routing verwenden, stellt die diagonale Kommunikation die höchsten Anforderung bezüglich der Latenz. Ferner ist an jeder Switch-Box ein S-Core-Prozessorfeld angeschlossen. Während der Latenzanalyse sendet jedes Prozessorelement periodisch ein Datenpaket bestimmter Größe an ein zufällig ausgewähltes Empfänger-Prozessorfeld. Die Größe der gesendeten Datenpakete variiert ebenfalls zufällig innerhalb der so genannten *Simple-Imix*-Verteilung [AT07]. Diese stellt eine repräsentative Verteilung von Datenpaketgrößen auf Basis einer typischen Internet-Kommunikation dar (siehe Tabelle 6.8). Mit Hilfe der Simple-Imix-Verteilung wird eine realistische Grundauslastung des On-Chip-Netzwerks simuliert. Parallel dazu sendet ein Prozessorelement des oberen rechten Prozessorfeldes periodisch ein so genanntes Benchmark-Paket zu einem Prozessorelement des linken unteren Prozessorfeldes. Die Listener-Module *L1*

Bezeichnung	Paketgröße	Häufigkeit
kleine Pakete	40 Byte	58,3 %
mittlere Pakete	576 Byte	33,3 %
große Pakete	1500 Byte	8,3 %

Tabelle 6.8: Datenpaketgrößen nach der Simple-Imix-Verteilung [AT07]

und *L2* können das Benchmark-Paket von allen anderen Datenpaketen unterscheiden und protokollieren jeweils einen Zeitstempel, wenn ein Benchmark-Paket das On-Chip-Netzwerk des entsprechenden Listener-Moduls passiert. Aus der Differenz der Zeitstempel lässt sich schließlich die Latenz des Benchmark-Pakets eindeutig bestimmen.

Abbildung 6.21 zeigt das Ergebnis der Performanzanalyse für verschiedene Gittergrößen des On-Chip-Netzwerks [PPG+10]. Es wurde für jeweils 650 Benchmark-Pakete die Latenz im Sinne der zur Übertragung benötigten Taktzyklen ausgewertet. Abbildung 6.21a veranschaulicht die Latenzen innerhalb einer 3x3-Gitter-Topologie, wie sie in Abbildung 6.20 dargestellt ist. Bis auf wenige Ausnahmen können alle Benchmark-Pakete des oberen rechten Prozessorfeldes mit einer durchschnittlichen Latenz von 12 Takten zum unteren linken Prozessorfeld übertragen werden. Wenn viele Pakete zeitgleich über denselben Ausgangs-Port einer Switch-Box transferiert werden müssen, führt dieses zu Verzögerungen bei der Übertragung. Vergleicht man die Abbildungen 6.21a - 6.21e, so ist klar zu erkennen, dass mit steigender Gittergröße des On-Chip-Netzwerks die Häufigkeit einer erhöhten Latenz zunimmt. Dieses ist nicht weiter verwunderlich, da die Wahrscheinlichkeit eines blockierten Switch-Box-Ports mit der Anzahl an sendenden Prozessorelementen skaliert. Abbildung 6.21f stellt die Latenz-Verteilung über allen untersuchten Gittergrößen dar. Hierbei zeigt sich, dass selbst in dem größten betrachteten On-Chip-Netzwerk 77 % aller Benchmark-Pakete in weniger als 100 Takten innerhalb der 7x7-Gitter-Topologie vom Sender zum Empfänger transferiert werden können. Aufgrund der nicht-blockierenden Round-Robin-Arbitrierung innerhalb der Switch-Boxen, weist das eingesetzte On-Chip-Netzwerk auch mit wachsender Gittergröße sehr gute Übertragungseigenschaften bezüglich der Latenz auf. Diese Aussage wird ebenfalls durch die statistische Auswertung der Latenzmessung in Abbildung 6.22 bestätigt. Für jede untersuchte Gittergröße des On-Chip-Netzwerks sind in Abbil-



(a) 3x3-Gitter-Topologie

(b) 4x4-Gitter-Topologie

(c) 5x5-Gitter-Topologie

(d) 6x6-Gitter-Topologie

(e) 7x7-Gitter-Topologie

(f) Latenz-Verteilung

Abbildung 6.21: Performanzanalyse des On-Chip-Netzwerks [PPG+10]

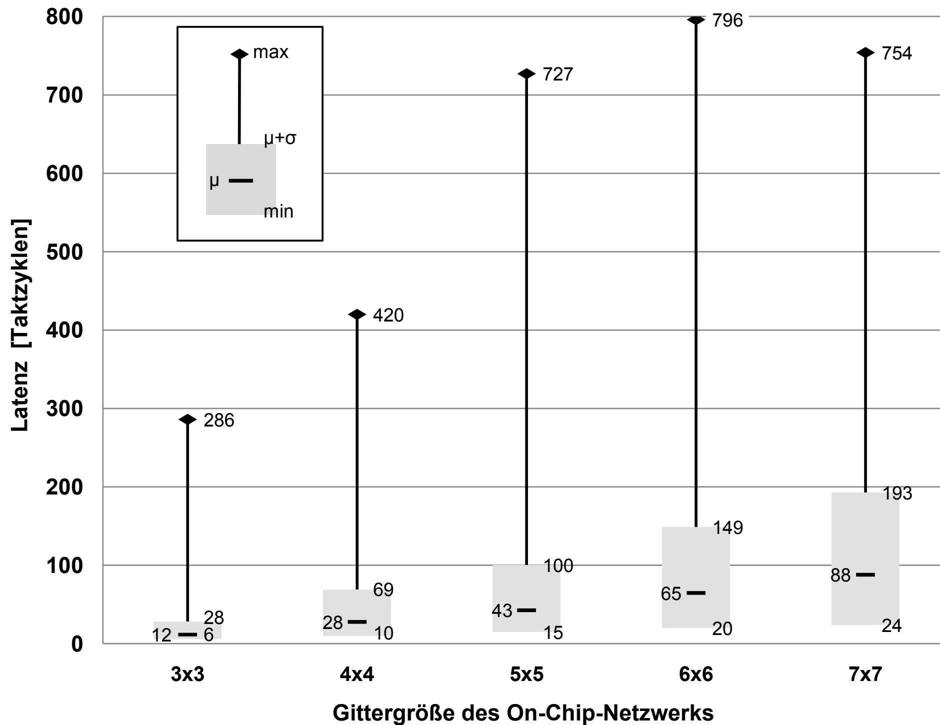


Abbildung 6.22: Statistische Auswertung der Latenzmessung [PPG+10]

dung 6.22 die minimale und maximale Latenz sowie den Mittelwert (μ) und die Summe von Mittelwert und Standardabweichung ($\mu + \sigma$). Wie man sieht, erhöht sich durch die Vergrößerung der Gitter-Topologie der Mittelwert und die Varianz der Übertragungslatenz nur geringfügig im Vergleich zu den maximalen Verzögerungswerten. Ferner ist zu beobachten, dass die maximale Latenz ab einer Gittergröße von 5x5 saturiert. Die minimale Latenz wiederum entspricht bei allen betrachteten NxN-Gittern annähernd dem theoretisch erreichbaren Optimum von $T_{\min} = 4N - 6$.

Als Fazit dieser Performanzanalyse kann festgehalten werden, dass sowohl die Datenübertragungsrate [PNPR07] als auch die Latenzeigenschaften [PPG+10] des verwendeten On-Chip-Netzwerks für die Anbindung des Coprozessor-Moduls zur Berechnung der Skalarmultiplikation geeignet ist [PS07]. Selbst im größten betrachteten 7x7-Gitter, beträgt die durchschnittliche Latenz zur Datenübertragung nur 1,7 % der Berechnungsdauer des Coprozessor-Moduls. Der Mehraufwand an Kommunikation über das On-Chip-Netzwerk ist für lose gekoppelte Hardware-Beschleuniger demnach vernachlässigbar.

6.4 Zusammenfassung

In diesem Kapitel wurden mehrere Hardware-Software-Kombinationen für die Kryptographie mit elliptischen Kurven vorgestellt. Es wurde die Anbindung von eng und lose gekoppelten Hardware-Beschleunigern auf verschiedenen Ebenen einer skalierbaren, hierarchischen MPSoC-Architektur präsentiert [PSPR08]. Wie in Abbildung 6.23 dargestellt ist, können Hardware-Beschleuniger durch den Einsatz generischer Schnittstellen-Adapter wahlweise auf Mikroprozessor-Ebene, auf Prozessorfeld-Ebene oder auf NoC-Ebene in die hierarchische MPSoC-Struktur integriert werden [NPPR06].

Mit der in Abschnitt 6.1 beschriebenen Entwurfsmethodik lassen sich Instruktionssatzerweiterungen auf einfache Weise identifizieren und in den Prozessorkern integrieren [PSP08]. Mit Hilfe der Ressourceneffizienz als Bewertungsmetrik konnte gezeigt werden, dass von den drei untersuchten Implementierungsvarianten (Instruktionspaare, Mikroprogramm und dedizierter Multiplizierer) das Mikroprogramm am besten für das Anwendungsszenario Chipkarte geeignet ist (vgl. Abschnitt 6.1.3).

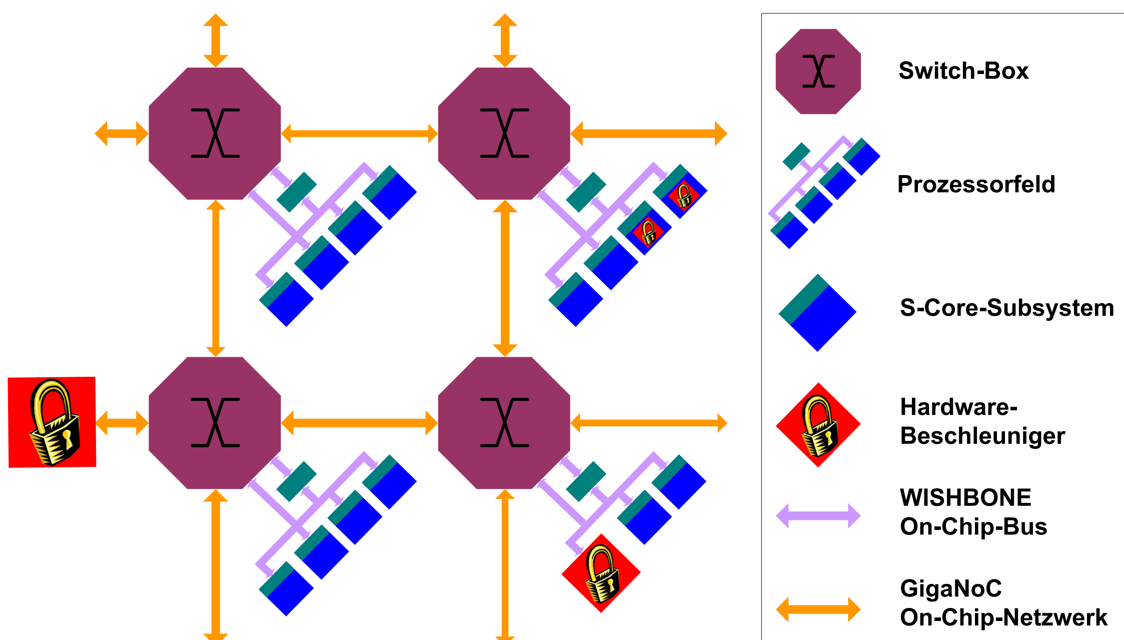


Abbildung 6.23: Integration von Hardware-Beschleunigern auf unterschiedlichen Hierarchieebenen einer Multiprozessor-System-on-Chip-Architektur

Auf Prozessorfeld-Ebene wurde neben der reinen Anbindung von Hardware-Beschleunigern auch die Parallelisierung der Binärkörper-Multiplikation für Mehrkernprozessoren im SIMD-, MIMD- und VLIW-Modus untersucht [PPP08][JPD+10]. Wie sich in Abschnitt 6.2.3 herausstellt, erreicht die VLIW-Architektur des CoreVA-Multiprozessors die beste Ressourceneffizienz für das Anwendungsszenario Sicherheitsserver.

Das in Abschnitt 6.3 vorgestellte Coprozessor-Modul wurde auf höchster Hierarchieebene direkt an das On-Chip-Netzwerk angeschlossen [PS07]. Auf diese Weise können alle Prozessorelemente der gesamten MPSoC-Architektur von der Performanz dieses lose gekoppelten Hardware-Beschleunigers profitieren. Um den Einfluss der Kommunikationsinfrastruktur auf die Performanz der Coprozessor-Moduls zu bewerten, wurde in Abschnitt 6.3.1 ein Verfahren präsentiert, mit der die Latenz des On-Chip-Netzwerks unter realen Bedingungen ermittelt werden kann [PPG+10].

7 Zusammenfassung und Ausblick

Motiviert durch die rasant zunehmende Vernetzung von technischen Geräten aller Art besteht vermehrt der Bedarf an sicherem Datenaustausch. Nicht nur die Anzahl der Internet-Nutzer steigt von Jahr zu Jahr stark an (vgl. Abbildung 1.1), sondern es sind auch immer mehr Alltagsgegenstände mit einer Kommunikationsschnittstelle ausgerüstet. Insbesondere im Hinblick auf das Thema *Internet der Dinge* (engl. Internet of Things, IoT) wird das Kommunikationsaufkommen in den kommenden Jahren weiter steigen [SWZL12]. Die Sicherheit bei der elektronischen Übermittlung von sensiblen Daten spielt hierbei stets eine entscheidende Rolle.

Anders als in der Vergangenheit werden heutzutage mehr und mehr mobile Geräte zum Datenaustausch verwendet, welche entweder mit einer Batterie betrieben werden oder ihre benötigte Energie selber produzieren können (*Energy Harvesting*). Diese Geräte verfügen meist über nur sehr beschränkte Ressourcen im Sinne der Rechenleistung, des Speicherplatzes sowie der Kommunikationsbandbreite. Genau für solch ein Anwendungsszenario eignet sich die Kryptographie mit elliptischen Kurven hervorragend, um Daten sicher zu übertragen sowie deren Integrität und Authentizität zu gewährleisten.

Im Rahmen dieser Arbeit wurden, am Beispiel von Algorithmen für die Kryptographie mit elliptischen Kurven, verschiedene Methoden vorgestellt, um ressourceneffiziente Hardware-Software-Kombinationen zu entwickeln. Im Vergleich zu vielen anderen Bewertungsmetriken umfasst die eingeführte *Ressourceneffizienz* die drei wesentlichen Parameter (*Chipfläche*, *Verlustleistung*, *Ausführungsdauer*) des Entwurfsraumes einer ASIC-Entwicklung.

Basierend auf einer hierarchisch entwickelten, skalierbaren Systemarchitektur (vgl. Abschnitt 4.4) wurde eine Entwurfsraumexploration für die beiden Anwendungsszenarien *Chipkarte* und *Sicherheitsserver* durchgeführt. Mit den angewandten Kon-

zepten der Instruktionssatzerweiterung, der Parallelisierung sowie eines Coprozessor-Ansatzes konnte die Ressourceneffizienz auf allen Hierarchieebenen der zugrundeliegenden Systemarchitektur anwendungsspezifisch optimiert werden.

Ein Vergleich von den zwei Prozessor-basierten Implementierungsvarianten S-Core und CoreVA mit kommerziell vertriebenen Produkten ist in Abbildung 7.1 dargestellt. Die Kreisgröße ist proportional zur Chipfläche der jeweiligen Prozessoren [Hal07][Hal08a][Hal08b]. Der Ausführungsdauer entspricht die Erzeugung eines Schlüsselpaares mit Hilfe des `ecdonaldb233`-Algorithmus aus dem SUPERCOP[1]-Benchmark [BL12], welchem eine OpenSSL-Implementierung der Algorithmen für digitale Signaturen basierend auf Kryptographie mit elliptischen Kurven (ECDSA) zugrunde liegt. Die zentrale Berechnung stellt hierbei wiederum die Skalarmultiplikation im Binärkörper $GF(2^{233})$ dar.

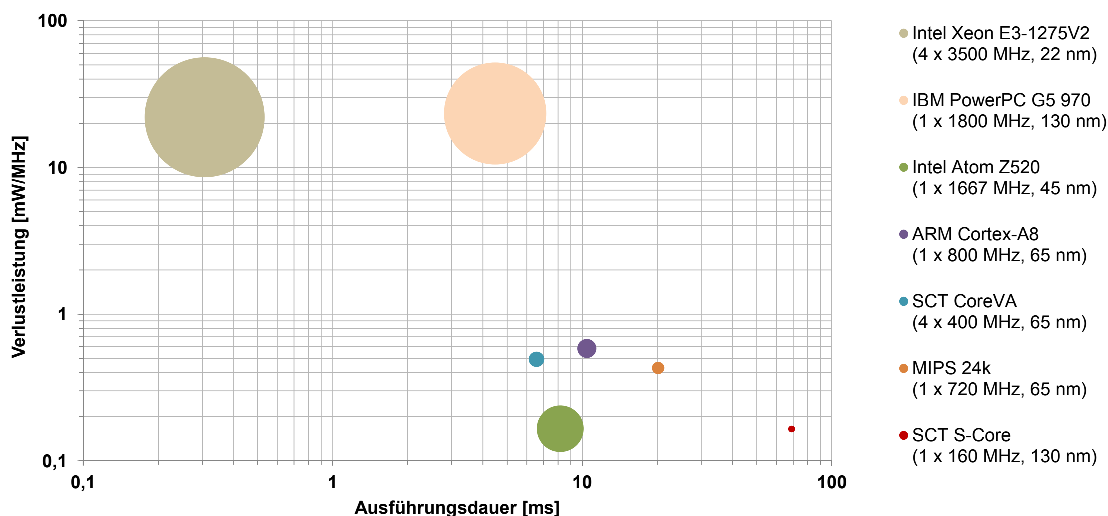


Abbildung 7.1: Ressourcenverbrauch verschiedener Prozessorsysteme zur Erzeugung eines Schlüsselpaares im Binärkörper $GF(2^{233})$

Wie aus Abbildung 7.1 ersichtlich ist, erreicht der Xeon-Prozessor von Intel die schnellste Ausführungsdauer. Die vier Prozessorkerne, welche mit 3500 MHz getaktet werden, berechnen den `ecdonaldb233`-Algorithmus [BL12] in nur 1074396 Takten. Neben der Mehrkernarchitektur und der hohen Taktfrequenz gibt es noch einen weiteren Grund für die hohe Performanz. Mit Einführung der *Sandy Bridge*-Mikroarchitektur von Intel im Jahr 2011 besitzen die Prozessoren eine Sammlung an neuen

[1]System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives

Instruktionssatzerweiterungen namens AVX[2] [Int09]. Darunter befindet sich die Instruktionssatzerweiterung `PCLMULQDQ`, welche eine übertragsfreie Multiplikation von zwei 64-Bit-Operanden durchführt. Diese Instruktion basiert auf dem gleichen Prinzip wie die in Kapitel 6.1 vorgestellten Instruktionssatzerweiterungen. Um weiterhin binärkompatibel zu Intel-Prozessoren zu sein, unterstützt AMD ab der *Bulldozer*-Mikroarchitektur ebenfalls diese Instruktionssatzerweiterung [Moo09]. Der Preis für die extrem schnelle Ausführung schlägt sich beim Xeon-Prozessor in der erforderlichen Chipfläche sowie der benötigten Verlustleistung nieder.

Während sowohl der Xeon-Prozessor als auch der PowerPC von IBM für den Server- bzw. Desktop-Bereich entwickelt wurden, kommen die restlichen Prozessoren aus Abbildung 7.1 auch in eingebetteten Systemen zum Einsatz. Unter diesen Prozessoren erzielt der CoreVA die beste Ausführungszeit. Die VLIW-Architektur und der integrierte Hardware-Beschleuniger für Kryptographie mit elliptischen Kurven (vgl. Abbildung 6.15) ermöglichen diese Performanz bei einer Taktfrequenz von nur 400 MHz. Die Chipfläche und Verlustleistung ist dabei vergleichbar mit dem Cortex-A8 von ARM und dem MIPS-Prozessor. Der S-Core und der Atom-Prozessor von Intel benötigen die geringste Verlustleistung zur Berechnung des `ecdonaldb233`-Algorithmus. Die Ausführungsdauer der beiden Prozessoren skaliert im Verhältnis ihrer Taktfrequenz. Allerdings benötigt der Atom-Prozessor, obwohl dieser in einer weitaus kleineren Technologie gefertigt wird, wesentlich mehr Chipfläche als der S-Core.

Im Gegensatz zu den Prozessor-basierten Software-Realisierungen in Abbildung 7.1 zeigt Tabelle 7.1 einen repräsentativen Überblick von Hardware-Implementierungen zur Berechnung der Skalarmultiplikation im Binärkörper $GF(2^{233})$. Das in Abschnitt 6.3 vorgestellte Coprozessor-Modul wird in Tabelle 7.1 mit Implementierungen anderer Forschergruppen [EGCS03][GBG+03][SGEG05][AH08][RRM12] sowie mit einem kommerziell angebotenen IP-Core [GR06] verglichen. Neben den in Tabelle 7.1 aufgeführten Arbeiten gibt es eine Vielzahl an weiteren Hardware-Realisierungen zur Berechnung der Skalarmultiplikation auf elliptischen Kurven. [DQ07] und [SNS12] geben einen ausführlichen Überblick über die Forschungsergebnisse der letzten Jahre. Um einen direkten Vergleich mit dem Coprozessor-Modul anzustellen, wurden in Tabelle 7.1 die Arbeiten berücksichtigt, welche ebenfalls den

[2] Advanced Vector Extensions

Implementierung		FPGA	Takt-frequenz	Größe [LUT/FF]	Latenz
Eberle et al.	[EGCS03]	XCV2000E	66 MHz	20068/6321	144 μs
Grabbe et al.	[GBG+03]	XC2V6000	100 MHz	19440/16970	130 μs
Shu et al.	[SGEG05]	XCV2000E	68 MHz	35800/10632	89 μs
Gaisler Research	[GR06]	XC2V6000	93 MHz	12850/ k. A.	180 μs
Ansari et al.	[AH08]	XC4VLX200	190 MHz	13396/2219	31 μs
Rebeiro et al.	[RRM12]	XC4VLX100	154 MHz	23147/ k. A.	13 μs
Coprozessor-Modul		XC2V8000	50 MHz	28571/4842	90 μs

Tabelle 7.1: Vergleich von Hardware-Implementierungen zur Berechnung einer Skalarmultiplikation im Binärkörper $GF(2^{233})$ [DQ07][SNS12]

Binärkörper $GF(2^{233})$ sowie die genormten NIST-Kurvenparameter (vgl. Anhang A) verwenden und auf Xilinx-FPGAs abgebildet wurden. Hierdurch kann anstelle der Chipfläche die Anzahl der benötigten *Look-Up Table* (LUT) und Register (engl. Flip-Flop, FF) als Größenmaß verwendet werden. Über die Verlustleistung ist in den entsprechenden Veröffentlichungen häufig keine Angabe gemacht worden.

Die in Tabelle 7.1 aufgeführten Implementierungen verfolgen alle ein ähnliches Konzept zur Berechnung der Skalarmultiplikation. Wie in Abbildung 6.18 für das Coprozessor-Modul dargestellt, verfügen alle Implementierungen über Schaltungsblöcke zur Addition, Multiplikation bzw. Quadrierung im zugrundeliegenden Binärkörper $GF(2^{233})$. Eine entsprechende Kontrolllogik verwendet diese Blöcke, um auf Basis der Punktaddition und Punktverdopplung die Skalarmultiplikation (vgl. Abbildung 2.6) durchzuführen. Die Struktur des Kontrollpfades ist jedoch zwischen den Implementierungen stark unterschiedlich. Während das Coprozessor-Modul, [SGEG05], [AH08] und [RRM12] schnelle Zustandsautomaten (FSM) zur Ablaufsteuerung einsetzen, wurde in [GBG+03] und [EGCS03] eine flexible Mikroprozessorarchitektur mit eigenem Instruktionssatz realisiert. Alle Implementierungen verwenden die Polynombasis-Darstellung mit projektivem Koordinatensystem. Die Implementierungen [EGCS03], [SGEG05], [AH08] und [RRM12] benutzen, genau wie das Coprozessor-Modul, das Koordinatensystem nach López-Dahab, wohingegen [GBG+03] auf Jacobi-Koordinaten setzt (vgl. Tabelle 3.1).

Neben der Struktur des Kontrollpfades ist ebenfalls die Umsetzung des Datenpfades entscheidend für den unterschiedlichen Ressourcenverbrauch hinsichtlich Größe und Latenz. Insbesondere die Architektur des Binärkörper-Multiplizierers hat hier großen Einfluss. Die Arbeiten [EGCS03], [SGEG05] und [AH08] implementieren einen Block-Seriellen-Multiplizierer (engl. Digit Serial Multiplier, DSM) nach dem MSDF[3]-Prinzip [SP98]. Das Coprozessor-Modul, [GBG+03] und [RRM12] verwenden dagegen eine Schaltung auf Basis der in Abschnitt 5.2.3 erläuterten hybriden Multiplikationsmethode.

Wie aus Tabelle 7.1 ersichtlich ist, erreicht [RRM12] die kürzeste Latenz, d. h. die schnellste Ausführungsdauer. Die Implementierung benötigt zur Berechnung der Skalarmultiplikation im Binärkörper $GF(2^{233})$ weniger als 2000 Taktzyklen. Leider wurde bei dieser Arbeit keine Angabe über die Anzahl benötigter Register veröffentlicht. Aufgrund vieler Pipeline-Stufen in dem hochgradig optimierten Datenpfad ist tendenziell von einer großen Anzahl an Registern auszugehen. Bei [AH08] handelt es sich um eine sehr flächeneffiziente Implementierung, welche relativ wenige LUT bzw. Register benötigt und trotzdem eine schnelle Ausführungsdauer aufweist. Mit ca. 5900 Taktzyklen liegt diese Implementierung ungefähr in der gleichen Größenordnung wie das Coprozessor-Modul ($\sim$ 4500 Taktzyklen) und [SGEG05] mit $\sim$ 6000 Taktzyklen. Alle anderen Implementierungen benötigen sowohl vergleichsweise viele LUT bzw. Register als auch Taktzyklen ([EGCS03] $\sim$ 9500 Taktzyklen, [GBG+03] $\sim$ 13000 Taktzyklen und [GR06] $\sim$ 16700 Taktzyklen) zur Berechnung der Skalarmultiplikation im Binärkörper $GF(2^{233})$.

Zusammenfassend kann festgehalten werden, dass sich je nach Anwendungsszenario sowohl das entwickelte Coprozessor-Modul als auch die mit Hardware-Beschleunigern erweiterten Mikroprozessorarchitekturen hervorragend als ressourceneffiziente Hardware-Software-Kombination für Kryptographie mit elliptischen Kurven eignet. Als Alternative zum traditionell verwendeten RSA-Verfahren wird sich die Elliptische-Kurven-Kryptographie, insbesondere im Bereich der eingebetteten Systeme, in Zukunft weiter etablieren.

[3] Most Significant Digit First

Abkürzungsverzeichnis

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
ATP	Area Time Product
AVX	Advanced Vector Extensions
BC	Best Case
BFFL	Binary Finite Field Library
BSI	Bundesamt für Sicherheit in der Informationstechnik
CAN	Controller Area Network
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CQFP	Ceramic Quad Flat Package
DES	Data Encryption Standard
DFG	Deutsche Forschungsgemeinschaft
DLL	Dynamic Link Library
DLP	Data Level Parallelism
DLP	Discrete Logarithm Problem
DPA	Differential Power Analysis
DRC	Design Rule Check
DSM	Digit Serial Multiplier
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
ECMQV	Elliptic Curve Menezes-Qu-Vanstone
ECNR	Elliptic Curve Nyberg-Rueppel
ECPVS	Elliptic Curve Pintsov-Vanstone Signatures
EDAP	Energy Delay Area Product
EDP	Energy Delay Product

FF	Flip-Flop
FIFO	First In First Out
FIPS	Federal Information Processing Standard
FPGA	Field Programmable Gate Array
FSM	Final State Machine
GCC	GNU Compiler Collection
GCR	Global Control Register
GF	Galois Field
GSR	Global Status Register
GUI	Graphical User Interface
I^2C	Inter-Integrated Circuit
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ILP	Instruction Level Parallelism
IoT	Internet of Things
IP	Intellectual Property
ISE	Instruction Set Extension
ISO	International Organization for Standardization
ISS	Instruction Set Simulator
LON	Local Operating Network
LUT	Look-Up Table
LVS	Layout Versus Schematic
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MMIO	Memory Mapped Input/Output
MPSoC	Multiprocessor System-on-Chip
MSDF	Most Significant Digit First
NIST	National Institute of Standards and Technology
NoC	Network-on-Chip
NTL	Number Theory Library
OCV	On-Chip Variation
PDAP	Power Delay Area Product
PDP	Power Delay Product
PE	Prozessorelement
PGP	Pretty Good Privacy
PIC	Programmable Interrupt Controller
PKI	Public-Key Infrastruktur
RAM	Random Access Memory
RE	Ressourceneffizienz

RFID	Radio Frequency Identification
RISC	Reduced Instruction Set Computer
RSA	Rivest-Shamir-Adleman
RTL	Register Transfer Level
SB	Switch-Box
SEC	Standards for Efficient Cryptography
SI	Signal Integrity
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SoC	System-on-Chip
SP	Stackpointer
SPA	Simple Power Analysis
SRAM	Static Random Access Memory
TA	Timing Analysis
TC	Typical Case
TLP	Thread Level Parallelism
TPM	Trusted Platform Module
UART	Universal Asynchronous Receiver/Transmitter
UPSLA	Unified Processor Specification Language
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
WC	Worst Case

Symbolverzeichnis

S_A	Geheimer Schlüssel von Person A (Private-Key)
P_A	Öffentlicher Schlüssel von Person A (Public-Key)
c	Verschlüsselte Nachricht (Chiffretext)
m	Unverschlüsselte Nachricht (Klartext)
K	Kryptographischer Schlüssel
$E(K, m)$	Verschlüsselungsfunktion
$D(K, c)$	Entschlüsselungsfunktion
T_L	Lebenszeit von verschlüsselten Daten
$\circ$	Verknüpfung von Elementen einer Gruppe
$GF(2^n)$	endlicher Körper (Galois Field) vom Grad n zur Basis 2
n	Erweiterungsgrad eines Binärkörpers
$\mathcal{O}$	Punkt im Unendlichen
E	Elliptische Kurve
a, b	Parameter zur Definition der elliptischen Kurve $E(GF(2^n))$
$G(x_G, y_G)$	Basispunkt der elliptischen Kurve $E(GF(2^n))$
$\#E$	Ordnung der elliptischen Kurve $E(GF(2^n))$
$\mathcal{P}$	Punkt auf einer elliptischen Kurve $E(GF(2^n))$
$f(t)$	Irreduzierbares Polynom
w	Datenwortbreite in Bit
t	Anzahl von w-Bit-Datenwortelementen eines Arrays
$K(m)$	Additionskette zur Zahl m
$l(K)$	Länge der Additionskette $K(m)$
$S(K)$	Sequenz zur Erzeugung der Additionskette $K(m)$

A	Chipfläche
P	Leistungsaufnahme
T	Ausführungszeit
C	Rechenleistung
O	Asymptotische Laufzeitkomplexität
RE	Ressourceneffizienz
c_A	Gewichtung der Chipfläche bzgl. der Ressourceneffizienz
c_P	Gewichtung der Leistungsaufnahme bzgl. der Ressourceneffizienz
c_T	Gewichtung der Ausführungszeit bzgl. der Ressourceneffizienz

C_L	Lastkapazität
E	Elektrisches Feld
f	Taktfrequenz
s	Skalierungsfaktor
α	Schaltaktivität
P_{last}	Verlustleistung durch Umladen der Lastkapazität
P_{quer}	Verlustleistung durch Querströme
P_{leck}	Verlustleistung durch Leckströme

Algorithmenverzeichnis

1 Montgomery-Leiter zur Berechnung der Skalarmultiplikation 28
2 Montgomery-Leiter nach López-Dahab 34
3 Addition von zwei Elementen des Binärkörpers $GF(2^n)$ 36
4 Schiebe-und-Addiere-Methode zur Multiplikation in $GF(2^n)$ 37
5 Rekursive Multiplikation nach der klassischen Methode 38
6 Rekursive Multiplikation nach der Karatsuba-Methode 39
7 w-Bit-Wort-Multiplikation mit Blockgröße s 40
8 Quadrierung von zwei Elementen des Binärkörpers $GF(2^n)$ 42
9 Modulo-Reduktion 43
10 Invertierung in $GF(2^n)$ mit Hilfe von Additionsketten 45

Abbildungsverzeichnis

1.1 Internetnutzung in Deutschland 1
1.2 Beispiele von Chipkarten mit kryptographischen Funktionen 2

2.1 Verschlüsselte Kommunikation nach dem Private-Key-Verfahren . . 8
2.2 Verschlüsselte Kommunikation nach dem Public-Key-Verfahren . . . 9
2.3 Additive Verknüpfung von Punkten auf einer elliptischen Kurve . . 17
2.4 Schlüssellängen äquivalenter Sicherheitsniveaus 19
2.5 Elektronischer Reisepass mit integriertem Mikroprozessor für Kryptographie mit elliptischen Kurven 22
2.6 Arithmetik der Elliptischen-Kurven-Kryptographie über Binärkörpern 24

3.1 Beispiel einer Skalarmultiplikation auf Basis der Montgomery-Leiter 29
3.2 Datenstruktur zur Speicherung der Koeffizienten eines Binärkörper-Elementes in Polynombasis-Darstellung 35
3.3 Reduktion der Polynomlänge nach dem Teile-und-Herrsche-Prinzip . 37
3.4 Quadrierung im Binärkörper $GF(2^n)$ 41
3.5 Additionskette $K(m)$ für $m = 232$ mit minimaler Länge $l(K) = 10$. 44
3.6 Algorithmische Optimierung auf verschiedenen Arithmetik-Ebenen . 47

4.1 Anforderungen an die Ressourcen zweier Anwendungsszenarien . . . 50
4.2 Vereinfachter Ablauf des digitalen Schaltungsentwurfs 52
4.3 Schematischer Aufbau eines S-Core-Einzelprozessorkerns 61
4.4 Verteilung der Instruktionen des S-Core-Mikroprozessors 62
4.5 Prozessorsystem auf Basis des S-Core-Mikroprozessors 63
4.6 Prozessorfeld auf Basis des S-Core-Mikroprozessorsystems 65
4.7 Multiprozessor-System-on-Chip auf Basis des S-Core-Prozessorfeldes 66
4.8 Testsystem auf Basis der Rapid-Prototyping-Plattform RAPTOR2000 70

4.9 Grafische Benutzeroberfläche zur Steuerung des FPGA-Prototypen . 72

5.1 Automatisierte Testumgebung zur Software-Evaluierung 76
5.2 Funktionen zur Binärkörper-Multiplikation auf 32-Bit-Wort-Ebene . 77
5.3 Bewertung der Multiplikationsfunktionen im Binärkörper $GF(2^n)$. 80
5.4 Iteratives Karatsuba-Verfahren für ausgewählte Binärkörper 83
5.5 Kombinatorische Logik zur Multiplikation von Polynomen vom Grad 3 nach der klassischen Methode 85
5.6 Kombinatorische Logik zur Multiplikation von Polynomen vom Grad 3 nach der Karatsuba-Methode 87
5.7 Vergleich der unterschiedlichen Multiplikationsmethoden 89
5.8 Ressourceneffizienz der vorgestellten Hardware-Multiplizierer 91
5.9 Graphische Benutzeroberfläche des Codegenerators 92
5.10 Auflösung von Pipeline-Hazards 92

6.1 Varianten der Instruktionssatzerweiterung 96
6.2 Zweistufige Entwurfsmethodik zur Instruktionssatzerweiterung . . . 99
6.3 Instruktionspaare der Multiplikation im Binärkörper $GF(2^{233})$ . . . 103
6.4 Superinstruktion LSRIANDADD 104
6.5 Erweiterte Recheneinheit (ALU) des S-Cores 105
6.6 Datenpfad des Mikroprogrammes MULGF2 107
6.7 Chipfläche der verschiedenen Instruktionssatzerweiterungen 109
6.8 Verlustleistung der verschiedenen Implementierungsvarianten 110
6.9 Normierte Ressourceneffizienz der Implementierungsvarianten . . . 112
6.10 Arten der Synchronisation in Multiprozessorsystemen 115
6.11 Klassifizierung von Rechnerarchitekturen nach Flynn 116
6.12 Betriebsmodi der rekonfigurierbaren QuadroCore-Architektur 118
6.13 QuadroCore-Architektur 119
6.14 Pipeline-Struktur des VLIW-Prozessors CoreVA 122
6.15 Struktur des Hardware-Beschleunigers im CoreVA-Prozessor 123
6.16 Fabrizierter CoreVA-VLIW-Prozessor 126
6.17 Ressourceneffizienz von S-Core, QuadroCore und CoreVA im Vergleich 128
6.18 Struktur des Coprozessors für Kryptographie mit elliptischen Kurven 129
6.19 Datenflussgraph zur Berechnung der Skalarmultiplikation 130

6.20 Einsatz von SiLLis zur Performanzanalyse des On-Chip-Netzwerks . 133
6.21 Performanzanalyse des On-Chip-Netzwerks 135
6.22 Statistische Auswertung der Latenzmessung 136
6.23 Integration von Hardware-Beschleunigern auf unterschiedlichen Hierarchieebenen einer Multiprozessor-System-on-Chip-Architektur . . . 137

7.1 Ressourcenverbrauch verschiedener Prozessorsysteme zur Erzeugung eines Schlüsselpaares im Binärkörper $GF(2^{233})$ 140

Tabellenverzeichnis

2.1 Additive und multiplikative Verknüpfung im Binärkörper 13
2.2 Lebenszeit und Schlüssellänge äquivalenter Sicherheitsniveaus . . . 18
2.3 Überblick an Standards zur Kryptographie mit elliptischen Kurven 21
2.4 Überblick an Protokollen zur Kryptographie mit elliptischen Kurven 22

3.1 Kosten der Punkt-Operationen in verschiedenen Koordinatensystemen 32
3.2 Iterationsschritte gemäß Algorithmus 10 am Beispiel $GF(2^{233})$ . . . 46

4.1 Skalierungsfaktoren für die Ressourcen Chipfläche, Leistungsaufnahme und Rechenleistung 59
4.2 Ressourcenverbrauch für eine 90-nm-CMOS-Technologie 68
4.3 Ressourcenverbrauch für den Xilinx-FPGA Virtex-II-8000 71

5.1 Codegröße und Ausführungszeiten von Funktionen zur Skalarmultiplikation für verschiedene Binärkörpergrößen 79
5.2 Codegröße und Ausführungszeiten von optimierten Funktionen zur Skalarmultiplikation für verschiedene Binärkörpergrößen 82

6.1 Maschinencode und Funktionalität eingeführter Superinstruktionen 106
6.2 Auswirkung der implementierten Instruktionssatzerweiterungen auf Ausführungszeit und Codegröße 111
6.3 Gewählte Exponenten zur Gewichtung der Ressourceneffizienz . . . 113
6.4 Parallelisierte Karatsuba-Multiplikation im Binärkörper $GF(2^{233})$. 120
6.5 Ressourcenbedarf der Hardware-Software-Kombinationen 124
6.6 Ressourcenbedarf verschiedener Architektur-Varianten für eine Multiplikation im erweiterten Binärkörper $GF(2^{233})$ 126
6.7 Ressourcenbedarf des Coprozessor-Moduls für zwei Zieltechnologien 131

6.8 Datenpaketgrößen nach der Simple-Imix-Verteilung 134

7.1 Vergleich von Hardware-Implementierungen zur Berechnung einer Skalarmultiplikation im Binärkörper $GF(2^{233})$ 142

Literaturverzeichnis

[AH08] B. Ansari, M. A. Hasan: “High-Performance Architecture of Elliptic Curve Scalar Multiplication”. In: *IEEE Transcations on Computers*, Bd. 57, Nr. 11, IEEE Computer Society, S. 1443–1453. November 2008.

[Amd67] G. M. Amdahl: “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the 1967 Spring Joint Computer Conference*, American Federation of Information Processing Societies (AFIPS), ACM, S. 483–485. Atlantic City, NJ, USA, 18.-20. April 1967.

[AMRK02] E. Al-Daoud, R. Mahmod, M. Rushdan, A. Kilicman: “A New Addition Formula for Elliptic Curves over GF(2^n)”. In: *IEEE Transactions on Computers*, Bd. 51, Nr. 8, IEEE Computer Society, S. 972–975. August 2002.

[ANSI89] American National Standard for Information Systems: “Programming Language C, Nr. X3.159-1989, Dezember 1989.

[ARM10] ARM Limited: “Cortex-A8 Technical Reference Manual”. Revision: r3p2. Juni 2010.

[AT07] Agilent Technologies, Inc.: “Mixed Packet Size Throughput”. In: *Journal of Internet Test Methodologies*, S. 29–31. September 2007. Edition 3.1

[BDB11] Bundesverband Deutscher Banken: “Online Banking – Ergebnisse einer repräsentativen Umfrage”. April 2011. URL: https://www.bankenverband.de/downloads/042011 (Stand: 03.02.2012)

[BDM02] L. Benini, G. De Micheli: “Networks on Chips: A New SoC Paradigm”. In: *IEEE Computer Magazine*, Bd. 35, Nr. 1, S. 70–78. 2002.

[Bel01] Bellezza, Antonio: “Binary Finite Field Library (BFFL)”. C Library under GNU General Public License. Version 0.0.2. URL: http://www.beautylabs.net (Stand: 08.01.2010)

[BGTZ08] R. P. Brent, P. Gaudry, E. Thomé, P. Zimmermann: “Faster Multiplication in GF(2)[x]”. In: *Proceedings of the 8th International Symposium on Algorithmic Number Theory (ANTS)*, Lecture Notes in Computer Science, Nr. 5011, Springer-Verlag, S. 153–166. Banff, AB, Kanada, 17.-22. Mai 2008.

[BIT12] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien (BITKOM): “Presseinfo Mediennutzung”. April 2012. URL: http://www.bitkom.org/de/presse/8477_71745.aspx (Stand: 26.04.2012)

[BKK+12] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, P. L. Montgomery: “Solving a 112-bit Prime Elliptic Curve Discrete Logarithm Problem on Game Consoles using Sloppy Reduction”. In: *International Journal of Applied Cryptography (IJACT)*, Bd. 2, Nr. 3, Inderscience Enterprises Ltd., S. 212–228. Januar 2012.

[BL12] Bernstein, Daniel J. and Lange, Tanja: “eBACS: ECRYPT Benchmarking of Cryptographic Systems”. 2012. URL: http://bench.cr.yp.to (Stand: 29.11.2012)

[Bla85] R. E. Blahut: “Fast Algorithms for Digital Signal Processing”. Addison-Wesley. September 1985.

[BMG11] Bundesministerium für Gesundheit: “Elektronische Gesundheitskarte”. Oktober 2011. URL: http://bmg.bund.de (Stand: 09.10.2011)

[BP01] D. V. Bailey, C. Paar: “Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography”. In: *Journal of Cryptology*, Bd. 14, Nr. 3, Springer-Verlag, S. 153–176. Dezember 2001.

[BSI11] Bundesamt für Sicherheit in der Informationstechnik: “Elektronische Ausweise”. Oktober 2011. URL: https://www.bsi.bund.de (Stand: 09.10.2011)

[BSI12a] Bundesamt für Sicherheit in der Informationstechnik: “eCard-Projekte der Bundesregierung”. Technische Richtline TR-03116. Version 3.16. August 2012.

[BSI12b] Bundesamt für Sicherheit in der Informationstechnik: “Elliptic Curve Cryptography”. Technische Richtline TR-03111. Version 2.0. Juni 2012.

[Buc08] J. Buchmann: “Einführung in die Kryptographie”. 4. erweiterte Auflage, Springer-Verlag. April 2008.

[BVH11] Bundesverband des Deutschen Versandhandels: “Entwicklung des E-Commerce in Deutschland”. November 2011. URL: http://www.versandhandel.org/studien/ (Stand: 10.12.2011)

[CB95] A. P. Chandrakasan, R. W. Brodersen: “Low Power Digital CMOS Design”. Kluwer Academic Publishers. 1995.

[CFA+05] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, F. Vercauteren: “Handbook of Elliptic and Hyperelliptic Curve Cryptography”. Discrete Mathematics and Its Applications, Chapman & Hall/CRC. Juli 2005.

[CSG98] D. Culler, J. Singh, A. Gupta: “Parallel Computer Architecture: A Hardware/Software Approach”. Morgan Kaufmann. 1998.

[DH76] W. Diffie, M. Hellman: “New directions in cryptography”. In: *IEEE Transactions on Information Theory*, Bd. 22, Nr. 6, S. 644–654. November 1976.

[DP12] B. Driessen, C. Paar: “Solving Binary Linear Equation Systems over the Rationals and Binaries”. In: *Proceedings of the 4th International Workshop on Arithmetic of Finite Fields (WAIFI)*, Springer-Verlag, S. 187–195. Bochum, Deutschland, 16.-19. Juli 2012.

[DQ07] G. M. de Dormale, J.-J. Quisquater: "High-speed hardware implementations of Elliptic Curve Cryptography: A survey". In: *Journal of Systems Architecture (JSA)*, Bd. 53, Nr. 2-3, S. 72–84. Februar 2007.

[DT01] W. J. Dally, B. Towles: "Route Packets, Not Wires: On-Chip Interconnection Networks". In: *Proceedings of the Design Automation Conference (DAC)*, S. 684–689. Las Vegas, Nevada, USA, 18.-22. Juni 2001.

[EGCS03] H. Eberle, N. Gura, S. Chang-Shantz: "A Cryptographic Processor for Arbitrary Elliptic Curves over GF(2m)". In: *Proceedings of the 14th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, S. 444–454. Den Haag, Niederlande, 24.-26. Juni 2003.

[EYK06] S. S. Erdem, T. Yanik, Ç. K. Koç: "Polynomial Basis Multiplication over GF(2^m)". In: *Acta Applicandae Mathematicae*, Bd. 93, Nr. 1-3, Springer-Verlag, S. 33–55. September 2006.

[FBB+10] J. Fan, D. V. Bailey, L. Batina, T. Güneysu, C. Paar, I. Verbauwhede: "Breaking Elliptic Curve Cryptosystems Using Reconfigurable Hardware". In: *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE Computer Society, S. 133–138. Mailand, Italien, 31. August - 2. September 2010.

[FDN+01] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, H.-S. P. Wong: "Device Scaling Limits of Si MOSFETs and Their Application Dependencies". In: *Proceedings of the IEEE, Special Issue on Limits of Semiconductor Technology*, Bd. 89, Nr. 3, S. 259–288. März 2001.

[Fly72] M. J. Flynn: "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers*, Bd. 21, Nr. 9, S. 948–960. September 1972.

[GBG+03] C. Grabbe, M. Bednara, J. von zur Gathen, J. Shokrollahi, J. Teich: "A High Performance VLIW Processor for Finite Field Arithmetic". In: *Proceedings of the 10th Reconfigurable Architectures Workshop (RAW)*, S. 189–194. Nizza, Frankreich, 22. April 2003.

[GBT+03] C. Grabbe, M. Bednara, J. Teich, J. von zur Gathen, J. Shokrollahi: "FPGA Designs of Parallel High Performance $GF(2^{233})$ Multipliers". In: *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, Bd. 2, S. 268–271. Bangkok, Thailand, 25.-28. Mai 2003.

[GCC04] GNU-Projekt: "GNU Compiler Collection (GCC)". Version 3.3.3. Februar 2004. URL: http://gcc.gnu.org/gcc-3.3 (Stand: 02.10.2009)

[GE04] J. Großschädl, S. Erkay: "Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m)". In: *Proceedings of the 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Lecture Notes in Computer Science, Nr. 3156, Springer-Verlag, S. 161–169. Cambridge, MA, USA, 11.-13. August 2004.

[GG03] J. von zur Gathen, J. Gerhard: "Modern Computer Algebra". 2. Auflage, Cambridge University Press. September 2003.

[GGK+06] J. Guajardo, T. Güneysu, S. S. Kumar, C. Paar, J. Pelzl: "Efficient Hardware Implementation of Finite Fields with Applications to Cryptography". In: *Acta Applicandae Mathematica*, Bd. 93, Nr. 1-3, Springer-Verlag, S. 75–118. September 2006.

[GGK08] V. Gopal, S. Grover, M. E. Kounavis: "Fast Multiplication Techniques for Public Key Cryptography". In: *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*, S. 316–325. Marrakesch, Marokko, 6.-9. Juli 2008.

[GIP+06] J. Großschädl, P. Ienne, L. Pozzi, S. Tillich, A. K. Verma: "Combining Algorithm Exploration with Instruction Set Design: A Case Study in Elliptic Curve Cryptography". In: *Proceedings of the 9th International Conference on Design, Automation and Test in Europe (DATE)*, European Design and Automation Association, S. 218–223. München, Deutschland, 6.-10. März 2006.

[GK10] S. Gueron, M. E. Kounavis: "Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode". White Paper 323640-001, Revision 2.0. Intel Corporation, Mai 2010.

[GN04] J. von zur Gathen, M. Nöcker: "Computing Special Powers in Finite Fields". In: *Mathematics of Computation*, Bd. 73, Nr. 247, American Mathematical Society, S. 1499–1523. Juli 2004.

[GP08] S. Govindarajulu, T. J. Prasad: "Considerations of Performance Factors in CMOS Designs". In: *Proceedings of the International Conference on Electronic Design (ICED)*, Penang, Malaysia, 1.-3. Dezember 2008.

[GPT04] J. Großschädl, K.-C. Posch, S. Tillich: "Architectural Enhancements to Support Digital Signal Processing and Public-Key Cryptography". In: *Proceedings of the 2nd Workshop on Intelligent Solutions in Embedded Systems (WISES)*, Graz University of Technology, S. 129–143. Graz, Österreich, 25. Juni 2004.

[GPW+04] N. Gura, A. Patel, A. Wander, H. Eberle, S. C. Shantz: "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs". In: *Proceedings of the 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Lecture Notes in Computer Science, Nr. 3156, Springer-Verlag, S. 119–132. Cambridge, MA, USA, 11.-13. August 2004.

[GR06] S. Habinc: "GRECC - Elliptic Curve Cryptography (ECC) IP Core User's Manual". Based on GRLIB-1.0.7. Gaisler Research, 2006.

[GS06] J. von zur Gathen, J. Shokrollahi: "Fast arithmetic for polynomials over $\mathbb{F}_2$ in hardware". In: *Proceedings of the IEEE Information Theory Workshop (ITW)*, S. 107–111. Punta del Este, Uruguay, 13.-17. März 2006.

[GS09] P. R. Grassi, M. D. Santambrogio: "SiLLis: Simplified Language for Listeners". Tech. Rep. 2009.36. Dipartimento di Elettronica ed Informazione, Politecnico di Milano, Via Ponzio 34/5, 20013 Milano (Italy), Oktober 2009. URL: http://www.dresd.org/doc/polimi-tr/dei-tr-2009.36.pdf (Stand: 02.11.2012)

[GWZ+05] V. Gupta, M. Wurm, Y. Zhu, M. Millard, S. Fung, N. Gura, H. Eberle, S. C. Shantz: "Sizzle: A standards-based end-to-end security architecture for the embedded Internet". In: *Pervasive and Mobile Computing Journal*, Bd. 1, Nr. 4, Elsevier B.V., S. 425–445. Dezember 2005.

[Hal07] T. R. Halfhill: “Cortex-R4X: Extreme Makeover”. In: *Microprocessor Report*, November 2007.

[Hal08a] T. R. Halfhill: “Freescale's Designer SoCs”. In: *Microprocessor Report*, November 2008.

[Hal08b] T. R. Halfhill: “Intel's Tiny Atom”. In: *Microprocessor Report*, April 2008.

[Has99] D. Hasse: “Entwurf eines Mikroprozessorkerns unter Berücksichtigung der Energieeffizienz und der Wiederverwertbarkeit”. Diplomarbeit, Bd. 52. Fachgebiet Schaltungstechnik, Heinz Nixdorf Institut, Universität Paderborn, April 1999.

[HH07] D. Harris, S. Harris: “Digital Design and Computer Architecture”. Morgan Kaufmann. 2007.

[HHM00] D. Hankerson, J. L. Hernandez, A. Menezes: “Software Implementation of Elliptic Curve Cryptography over Binary Fields”. In: *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Lecture Notes in Computer Science, Nr. 1965, Springer-Verlag, S. 1–24. Worcester, MA, USA, 17.-18. August 2000.

[HMV04] D. Hankerson, A. J. Menezes, S. Vanstone: “Guide to Elliptic Curve Cryptography”. Springer-Verlag. Januar 2004.

[HP03] J. Hennessy, D. Patterson: “Computer Architecture - A Quantitative Approach”. 2. Auflage, Morgan Kaufmann. November 2003.

[IFX04] Infineon Technologies AG: “90nm CMOS Data Book”. Edition 3.2004. 2004.

[IFX06] Infineon Technologies AG: “65nm CMOS Data Book”. Edition 3.2006. 2006.

[Int09] Intel Corporation: “Intel® Advanced Vector Extensions”. Programming Reference 319433-006. Juli 2009. URL: http://software.intel.com/file/21558 (Stand: 03.12.2009)

[IT88] T. Itoh, S. Tsujii: "A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ using Normal Bases". In: *Journal of Information and Computation*, Bd. 78, Nr. 3, S. 171–177. September 1988.

[ITRS07] Semiconductor Industry Association: "International Technology Roadmap for Semiconductors". 2007 Edition. 2007.

[JT03] A. Jantsch, H. Tenhunen: "Networks on Chip". Kluwer Academic Publishers. Januar 2003.

[Jun11] T. Jungeblut: "Entwurfsraumexploration ressourceneffizienter VLIW-Prozessoren". Dissertation. Technische Fakultät, Universität Bielefeld, Oktober 2011.

[Kar95] A. A. Karatsuba: "The Complexity of Computations". In: *Proceedings of the Steklov Institute of Mathematics*, Bd. 211, S. 169–183. Januar 1995.

[KLST04] U. Kastens, D. K. Le, A. Slowik, M. Thies: "Feedback Driven Instruction-Set Extension". In: *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Bd. 39, Nr. 7, ACM, S. 126–135. Washington, DC, USA, 11.-13. Juni 2004.

[Knu97] D. E. Knuth: "Art of Computer Programming". 3. Auflage, Bd. 2: Seminumerical Algorithms. Addison-Wesley Professional. November 1997.

[KO63] A. Karatsuba, Y. Ofman: "Multiplication of Multidigit Numbers on Automata". In: *Soviet Physics Doklady*, Bd. 7, S. 595–596. 1963.

[Kob87] N. Koblitz: "Elliptic Curve Cryptosystems". In: *Mathematics of Computation*, Bd. 48, Nr. 177, American Mathematical Society, S. 203–209. Januar 1987.

[Koç09] Ç. K. Koç: "Cryptographic Engineering". Springer-Verlag. 2009.

[KPR02] H. Kalte, M. Porrmann, U. Rückert: "A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs". In: *Proceedings of the IEEE Workshop on Heterogeneous Reconfigurable Systems on Chip*, Hamburg, Germany, 2002.

[KSFV08] M. Knežević, K. Sakiyama, J. Fan, I. Verbauwhede: “Modular Reduction in $GF(2n)$ without Pre-computational Phase”. In: *Proceedings of the 2nd International Workshop on Arithmetic of Finite Fields (WAIFI)*, Lecture Notes in Computer Science, Nr. 5130, Springer-Verlag, S. 77–87. Siena, Italien, 6.-9. Juli 2008.

[Lan05] D. Langen: “Abschätzung des Ressourcenbedarfs von hochintegrierten mikroelektronischen Systemen”. Dissertation, Bd. 161, HNI-Verlagsschriftenreihe. Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, April 2005.

[LD98] J. López, R. Dahab: “Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n)”. In: *Proceedings of the 5th Annual International Workshop on Selected Areas in Cryptography (SAC)*, Lecture Notes in Computer Science, Nr. 1556, Springer-Verlag, S. 201–212. Kingston, Ontario, Kanada, 17.-18. August 1998.

[LD99] J. López, R. Dahab: “Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation”. In: *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Lecture Notes in Computer Science, Nr. 1717, Springer-Verlag, S. 316–327. Worcester, MA, USA, 12.-13. August 1999.

[Len00] A. K. Lenstra: “Integer Factoring”. In: *Designs, Codes and Cryptography*, Bd. 19, Nr. 2-3, Kluwer Academic Publishers, S. 101–128. März 2000.

[LS08] K. Lauter, K. Stange: “The Elliptic Curve Discrete Logarithm Problem and Equivalent Hard Problems for Elliptic Divisibility Sequences”. In: *Proceedings of the 15th Annual Workshop on Selected Areas in Cryptography (SAC)*, Lecture Notes in Computer Science, Bd. 5381, Springer-Verlag, S. 309–327. Sackville, New Brunswick, Kanada, 14.-15. August 2008.

[LV01] A. K. Lenstra, E. R. Verheul: “Selecting cryptographic key sizes”. In: *Journal of Cryptology*, Bd. 14, Nr. 4, Springer-Verlag, S. 255–293. Dezember 2001.

[Mar07] P. Marwedel: “Embedded Systems”. 2. Auflage, Springer-Verlag. Februar 2007.

[MHH12] N. Mohamed, M. Hashim, M. Hutter: “Improved Fixed-Base Comb Method for Fast Scalar Multiplication”. In: *Proceedings of the 5th International Conference on Progress in Cryptology*, Lecture Notes in Computer Science, Bd. 7374, Springer-Verlag, S. 342–359. Ifrane, Marokko, Afrika, 10.-12. Juli 2012.

[Mil85] V. S. Miller: “Uses of Elliptic Curves in Cryptography”. In: *Proceedings of the 5th International Cryptology Conference (CRYPTO)*, Lecture Notes in Computer Science, Bd. 218, Springer-Verlag, S. 417–426. Santa Barbara, CA, USA, 18.-22. August 1985.

[Mon87] P. L. Montgomery: “Speeding the Pollard and Elliptic Curve Methods of Factorization”. In: *Mathematics of Computation*, Bd. 48, Nr. 177, American Mathematical Society, S. 243–264. Januar 1987.

[Mon05] P. L. Montgomery: “Five, Six, and Seven-Term Karatsuba-Like Formulae”. In: *IEEE Transactions on Computers*, Bd. 54, Nr. 3, IEEE Computer Society, S. 362–369. März 2005.

[Moo65] G. E. Moore: “Cramming more components onto integrated circuits”. In: *Electronics*, Bd. 38, Nr. 8, S. 114–117. 19 April 1965.

[Moo09] C. Moore: “Accelerated Processing Unit Strategy”. AMD Financial Analyst Day. 11. November 2009. URL: http://phx.corporate-ir.net/phoenix.zhtml?c=74093&p=irol-analystday (Stand: 03.12.2009)

[Mot98] Motorola, Inc.: “M-Core™ microRISC Engine”. Application Note. 1998. URL: http://www.freescale.com/files/32bit/doc/app_note/MCORE.pdf (Stand: 02.10.2009)

[MOV96] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone: “Handbook of Applied Cryptography”. Discrete Mathematics and Its Applications, CRC Press. 1996.

[Mül98] V. Müller: "Fast Multiplication on Elliptic Curves over Small Fields of Characteristic Two". In: *Journal of Cryptology*, Bd. 11, Nr. 4, Springer-Verlag, S. 219–234. November 1998.

[Nie08] J.-C. Niemann: "Ressourceneffiziente Schaltungstechnik eingebetteter Parallelrechner". Dissertation, Bd. 247, HNI-Verlagsschriftenreihe. Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, Dezember 2008.

[NIST77] National Institute of Standards and Technology: "Data Encryption Standard". In: *Federal Information Processing Standards Publication (FIPS PUB)*, Nr. 46, United States Department of Commerce. Januar 1977

[NIST94] National Institute of Standards and Technology: "Digital Signature Standard". In: *Federal Information Processing Standards Publication (FIPS PUB)*, Nr. 186, United States Department of Commerce. Mai 1994

[NIST01] National Institute of Standards and Technology: "Advanced Encryption Standard". In: *Federal Information Processing Standards Publications (FIPS PUBS)*, Nr. 197, United States Department of Commerce. November 2001

[NIST07] National Institute of Standards and Technology: "Recommendation for Key Management". In: *NIST Special Publications*, Nr. 800-57, United States Department of Commerce. März 2007

[OCO02] OpenCores Organization: "WISHBONE System-on-Chip (SoC) Interconnect Architecture for Portable IP Cores". Revision B.3. September 2002. URL: http://opencores.org/downloads/wbspec_b3.pdf (Stand: 05.11.2009)

[PH05] D. A. Patterson, J. L. Hennessy: "Computer Organization and Design: The Hardware/Software Interface". Morgan Kaufmann. 2005.

[Por09] Porrmann, Mario: "RAPTOR – Modular Rapid Prototyping". Heinz Nixdorf Institut, Universität Paderborn. URL: http://www.raptor2000.de (Stand: 06.12.2009)

[PP10] C. Paar, J. Pelzl: “Understanding Cryptography”. Springer-Verlag. 2010.

[PPH11] C. Pendl, M. Pelnar, M. Hutter: “Elliptic Curve Cryptography on the WISP UHF RFID Tag”. In: *Proceedings of the 7th Workshop on RFID Security and Privacy (RFIDSec)*, Lecture Notes in Computer Science, Bd. 7055, Springer-Verlag, S. 32–47. Amherst, Massachusetts, USA, 26.-28. Juni 2011.

[PSNB11] G. C. C. F. Pereira, M. A. Simplício, M. Naehrig, P. S. L. M. Barreto: “A Family of Implementation-Friendly BN Elliptic Curves”. In: *Journal of Systems and Software*, Bd. 84, Nr. 8, Elsevier Science Inc., S. 1319–1326. August 2011.

[PSRG08] C. Puttmann, J. Shokrollahi, U. Rückert, J. von zur Gathen: “Abschlussbericht zum DFG-Projekt Krypto-Hardware”. Fördernummer RU 477/8 und GA 528/5. Deutsche Forschungsgemeinschaft (DFG), Dezember 2008.

[Pur09] M. Purnaprajna: “Run-time Reconfigurable Multiprocessors”. Dissertation, Bd. 261, HNI-Verlagsschriftenreihe. Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, Dezember 2009.

[Put04] C. Puttmann: “Leistungsbewertung eingebetteter Prozessoren”. Studienarbeit, Bd. 80. Fachgebiet Schaltungstechnik, Heinz Nixdorf Institut, Universität Paderborn, Juli 2004.

[Put05] C. Puttmann: “Portierung von Netzwerkanwendungen auf eingebettete Parallel-Prozessoren”. Diplomarbeit, Bd. 112. Fachgebiet Schaltungstechnik, Heinz Nixdorf Institut, Universität Paderborn, Januar 2005.

[PY05] D. Pan, Y. Yang: “FIFO-Based Multicast Scheduling Algorithm for Virtual Output Queued Packet Switches”. In: *IEEE Transactions on Computers*, Bd. 54, Nr. 10, S. 1283–1297. Oktober 2005.

[Rab09] J. Rabaey: “Low Power Design Essentials”. Integrated Circuits and Systems, Springer-Verlag. Mai 2009.

[RHK03] F. Rodríguez-Henríquez, Ç. K. Koç: “On fully parallel Karatsuba Multipliers for $GF(2^m)$”. In: *Proceedings of the International Conference on Computer Science and Technology (CST)*, ACTA Press, S. 405–410. Cancun, Mexiko, 19.-21. Mai 2003.

[Roo00] S. H. Roosta: “Parallel Processing and Parallel Algorithms: Theory and Computation”. Springer-Verlag. 2000.

[RRM12] C. Rebeiro, S. Roy, D. Mukhopadhyay: “Pushing the Limits of High-Speed $GF(2^m)$ Elliptic Curve Scalar Multiplication on FPGAs”. In: *Proceedings of the 14th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Lecture Notes in Computer Science, Nr. 7428, Springer-Verlag, S. 494–511. Leuven, Belgien, 9.-12. September 2012.

[RSA78] R. L. Rivest, A. Shamir, L. Adleman: “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM*, Bd. 21, S. 120–126. Februar 1978.

[SGEG05] C. Shu, K. Gaj, T. El-Ghazawi: “Low Latency Elliptic Curve Cryptography Accelerators for NIST Curves over Binary Fields”. In: *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, S. 309–310. Singapur, 11.-14. Dezember 2005.

[Sho08] Shoup, Victor: “Number Theory Library (NTL)”. Portable C++ Library under GNU General Public License. Version 5.4.2. URL: http://www.shoup.net/ntl (Stand: 05.03.2008)

[Sho06] J. Shokrollahi: “Efficient Implementation of Elliptic Curve Cryptography on FPGAs”. Dissertation. Mathematisch-Naturwissenschaftliche Fakultät, Rheinische Friedrich-Wilhelms-Universität Bonn, Dezember 2006.

[Sil09] J. H. Silverman: “The Arithmetic of Elliptic Curves”. 2. Auflage, Graduate Texts in Mathematics, Springer-Verlag. Mai 2009.

[SJD04] K. Sun-Ho, L. Joo-Ae, K. Daejeong: “Design methodology adopting normalized power-delay-and-area product (N-PDAP) for digital-circuit op-

timization". In: *Current Applied Physics*, Bd. 4, Nr. 1, Elsevier B.V., S. 87–90. Amsterdam, 2004.

[SK10] E. Savaş, Ç. K. Koç: "Finite Field Arithmetic for Cryptography". In: *IEEE Circuits and Systems Magazine*, Bd. 10, Nr. 2, S. 40–56. Juli 2010.

[SNS12] N. Shylashree, B. Nagarjun, V. Sridhar: "FPGA Implementations of High Speed Elliptic Curve Cryptography: A Survey". In: *International Journal of Scientific & Engineering Research (IJSER)*, Bd. 3, Nr. 3, März 2012.

[SP98] L. Song, K. K. Parhi: "Low-Energy Digit-Serial/Parallel Finite Field Multipliers". In: *Journal of VLSI Signal Processing*, Bd. 19, Nr. 2, S. 149–166. Juli 1998.

[SPG02] D. Soudris, C. Piguet, C. Goutis: "Designing CMOS Circuits for Low Power". Kluwer Academic Publishers. 2002.

[SS02] N. Slingerland, A. J. Smith: "Measuring the Performance of Multimedia Instruction Sets". In: *IEEE Transactions on Computers*, Bd. 51, Nr. 11, S. 1317–1332. November 2002.

[SS05] D. Sengupta, R. Saleh: "Power-Delay Metrics Revisited for 90nm CMOS Technology". In: *Proceedings of the 6th International Symposium on Quality of Electronic Design (ISQED)*, IEEE Computer Society, S. 291–296. San Jose, CA, USA, 21.-23. März 2005.

[STM08] STMicroelectronics: "CMOS065_LP Technology". User Manual & Databook 5.1. April 2008.

[SWZL12] H. Suo, J. Wan, C. Zou, J. Liu: "Security in the Internet of Things: A Review". In: *Proceedings of the International Conference on Computer Science and Electronics Engineering (ICCSEE)*, Bd. 3, S. 648–651. Hangzhou, China, 23.-25. Mai 2012.

[Tan07] A. S. Tanenbaum: "Modern Operating Systems". 3. Auflage, Prentice Hall. Dezember 2007.

[Tau06] G. Taubenfeld: "Synchronization Algorithms and Concurrent Programming". Prentice Hall. 2006.

[TK03] A. F. Tenca, Ç. K. Koç: "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm". In: *IEEE Transactions on Computers*, Bd. 52, Nr. 9, IEEE Computer Society, S. 1215–1221. 2003.

[TNS11] TNS Infratest: "(N)ONLINER Atlas 2011". Initiative D21. Juli 2011. URL: http://www.initiatived21.de/publikationen (Stand: 08.01.2012)

[TOSK12] V. Trujillo-Olaya, T. Sherwood, Ç. K. Koç: "Analysis of performance versus security in hardware realizations of small elliptic curves for lightweight applications". In: *Journal of Cryptographic Engineering*, Bd. 2, Nr. 3, Springer-Verlag, S. 179–188. 2012.

[Vee08] H. Veendrick: "Nanometer CMOS ICs: From basics to ASICs". Springer-Verlag. 2008.

[Was03] L. C. Washington: "Elliptic Curves: Number Theory and Cryptography". Discrete Mathematics and Its Applications, Chapman & Hall/CRC. Mai 2003.

[WP06] A. Weimerskirch, C. Paar: "Generalizations of the Karatsuba Algorithm for Efficient Implementations". Technischer Bericht 2006/224. Ruhr-Universität Bochum, Deutschland, 2006.

[Xlnx07] Xilinx, Inc.: "Virtex-II Platform FPGAs: Complete Data Sheet". Product Specification DS031 (v3.5). November 2007.

[Yan13] S. Y. Yan: "Computational Number Theory and Modern Cryptography". John Wiley & Sons. Februar 2013.

[Zim95] P. Zimmermann: "The official PGP user's guide". MIT Press. Juni 1995.

[ZJ04] L. Zhonghai, A. Jantsch: "Flit Admission in On-chip Wormhole-switched Networks with Virtual Channels". In: *Proceedings of the International Symposium on Systems-on-Chip (SoC)*, S. 21–24. Tampere, Finnland, 16.-18. November 2004.

Eigene Veröffentlichungen

[GPS07] E. Gorla, C. Puttmann, J. Shokrollahi: "Explicit Formulas for Efficient Multiplication in $GF(3^{6m})$". In: *Proceedings of the 14th Annual Workshop on Selected Areas in Cryptography (SAC)*, Lecture Notes in Computer Science, Bd. 4876, Springer-Verlag, S. 173–183. Ottawa, Ontario, Kanada, 16.-17. August 2007.

[GSP+09] P. R. Grassi, M. D. Santambrogio, C. Puttmann, C. Pohl, M. Porrmann: "A High Level Methodology for Monitoring Network-on-Chips". In: *Digest of the DATE Workshop on Diagnostic Services in Network-on-Chips (DSNOC)*, S. 79–97. Nizza, Frankreich, 24. April 2009. URL: http://www.dsnoc.org/files/dsnoc09-program.pdf (Stand: 10.07.2009)

[JPD+10] T. Jungeblut, C. Puttmann, R. Dreesen, M. Porrmann, M. Thies, U. Rückert, U. Kastens: "Resource Efficiency of Hardware Extensions of a 4-issue VLIW Processor for Elliptic Curve Cryptography". In: *Advances in Radio Science (ARS)*, Bd. 8, S. 295–305. Dezember 2010. URL: http://www.adv-radio-sci.net/8/295/2010 (Stand: 11.09.2011)

[NPPR06] J.-C. Niemann, C. Puttmann, M. Porrmann, U. Rückert: "GigaNetIC – A Scalable Embedded On-Chip Multiprocessor Architecture for Network Applications". In: *Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS)*, Lecture Notes in Computer Science, Bd. 3894, Springer-Verlag, S. 268–282. Frankfurt am Main, Deutschland, 13.-16. März 2006.

[NPPR07] J.-C. Niemann, C. Puttmann, M. Porrmann, U. Rückert: "Resource Efficiency of the GigaNetIC Chip Multiprocessor Architecture". In: *Journal of Systems Architecture (JSA), Special Issue on Architectural Premises for Pervasive Computing*, Bd. 53, Nr. 5-6, Elsevier B.V., S. 285–299. Amsterdam, Mai 2007.

[PNPR07] C. Puttmann, J.-C. Niemann, M. Porrmann, U. Rückert: "GigaNoC – A Hierarchical Network-on-Chip for Scalable Chip-Multiprocessors". In:

Proceedings of the 10th EUROMICRO Conference on Digital System Design (DSD), IEEE Computer Society, S. 495–502. Lübeck, Deutschland, 29.-31. August 2007.

[PPG+10] C. Puttmann, M. Porrmann, P. R. Grassi, M. D. Santambrogio, U. Rückert: "High Level Specification of Embedded Listeners for Monitoring of Network-on-Chips". In: *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, S. 3333–3336. Paris, Frankreich, 30. Mai - 2. Juni 2010.

[PPP08] M. Purnaprajna, C. Puttmann, M. Porrmann: "Power Aware Reconfigurable Multiprocessor for Elliptic Curve Cryptography". In: *Proceedings of the 11th International Conference on Design, Automation and Test in Europe (DATE)*, European Design and Automation Association, S. 1462–1467. München, Deutschland, 10.-14. März 2008.

[PPP09] M. Porrmann, M. Purnaprajna, C. Puttmann: "Self-optimization of MPSoCs Targeting Resource Efficiency and Fault Tolerance". In: *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, IEEE Computer Society, S. 467–473. San Francisco, Kalifornien, USA, 29. Juli - 1. August 2009.

[PPR10] C. Puttmann, M. Porrmann, U. Rückert: "Extending GigaNoC towards a Dependable Network-on-Chip". In: *Digest of the DAC Workshop on Diagnostic Services in Network-on-Chips (DSNOC)*, S. 13–14. Anaheim, Kalifornien, USA, 13.-18. Juni 2010. URL: http://www.dsnoc.org/files/dsnoc10-program.pdf (Stand: 14.07.2010)

[PS07] C. Puttmann, J. Shokrollahi: "MPSoC-coupled Hardware Accelerator for Elliptic Curve Cryptography". In: *Conference Records of the Western European Workshop on Research in Cryptology (WEWoRC)*, S. 16–20. Bochum, Deutschland, 4.-6. Juli 2007. URL: http://2007.weworc.org/index.html (Stand: 03.02.2008)

[PSP08] C. Puttmann, J. Shokrollahi, M. Porrmann: "Resource Efficiency of Instruction Set Extension for Elliptic Curve Cryptography". In: *Proceedings of the 5th International Conference on Information Technology:*

New Generation (ITNG), IEEE Computer Society, S. 131–136. Las Vegas, Nevada, USA, 7.-9. April 2008.

[PSPR08] C. Puttmann, J. Shokrollahi, M. Porrmann, U. Rückert: "Hardware Accelerators for Elliptic Curve Cryptography". In: *Advances in Radio Science (ARS)*, Bd. 6, S. 259–264. Mai 2008. URL: http://www.adv-radio-sci.net/6/259/2008/ (Stand: 21.08.2008)

[SGP07] J. Shokrollahi, E. Gorla, C. Puttmann: "Efficient FPGA-based Multipliers for $\mathbb{F}_{3^{97}}$ and $\mathbb{F}_{3^{6\cdot 97}}$". In: *Proceedings of the 17th International Conference on Field Programmable Logic and its Applications (FPL)*, S. 339–344. Amsterdam, Niederlande, 27.-29. August 2007.

Anhang A: Kurvenparameter

Im Folgenden sind die in dieser Arbeit verwendeten Kurvenparameter aufgelistet. Die Parameter wurden als Standard (FIPS 186-3) vom *National Institute of Standards and Technology* (NIST) veröffentlicht [NIST94]. Die hier angegebenen Parameter $n, f(t), b, x_G, y_G$ beziehen sich auf eine elliptische Kurve E über dem Binärkörper $GF(2^n)$ in der Form:

$$E : y^2 + xy = x^3 + x^2 + b$$

Der Parameter n gibt dabei den Erweiterungsgrad des Binärkörpers an. Durch $f(t)$ ist das irreduzierbare Polynom für die Modulo-Reduktion festgelegt. Die elliptische Kurve E ist durch den Parameter b eindeutig definiert (vgl. Gleichung 2.8). Schließlich bilden die Koordinaten (x_G, y_G) den Basispunkt auf der elliptischen Kurve E. Die Parameter b, x_G, y_G sind als hexadezimale Werte in Polynombasis-Form dargestellt (vgl. Gleichung 2.1).

$n = 163, \quad f(t) = t^{163} + t^7 + t^6 + t^3 + 1$

$b =$ 0x 00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD

$x_G =$ 0x 00000003 F0EBA162 86A2D57E A0991168 D4994637 E8343E36

$y_G =$ 0x 00000000 D51FBC6C 71A0094F A2CDD545 B11C5C0C 797324F1

$n = 233, \quad f(t) = t^{233} + t^{74} + 1$

$b =$ 0x 00000066 647EDE6C 332C7F8C 0923BB58 213B333B 20E9CE42 81FE115F 7D8F90AD

$x_G =$ 0x 000000FA C9DFCBAC 8313BB21 39F1BB75 5FEF65BC 391F8B36 F8F8EB73 71FD558B

$y_G =$ 0x 00000100 6A08A419 03350678 E58528BE BF8A0BEF F867A7CA 36716F7E 01F81052

$n = 283, \quad f(t) = t^{283} + t^{12} + t^7 + t^5 + 1$

$b =$ 0x 027B680A C8B8596D A5A4AF8A 19A0303F CA97FD76 45309FA2 A581485A
F6263E31 3B79A2F5

$x_G =$ 0x 05F93925 8DB7DD90 E1934F8C 70B0DFEC 2EED25B8 557EAC9C 80E2E198
F8CDBECD 86B12053

$y_G =$ 0x 03676854 FE24141C B98FE6D4 B20D02B4 516FF702 350EDDB0 826779C8
13F0DF45 BE8112F4

$n = 409, \quad f(t) = t^{409} + t^{87} + 1$

$b =$ 0x 0021A5C2 C8EE9FEB 5C4B9A75 3B7B476B 7FD6422E F1F3DD67 4761FA99
D6AC27C8 A9A197B2 72822F6C D57A55AA 4F50AE31 7B13545F

$x_G =$ 0x 015D4860 D088DDB3 496B0C60 64756260 441CDE4A F1771D4D B01FFE5B
34E59703 DC255A86 8A118051 5603AEAB 60794E54 BB7996A7

$y_G =$ 0x 0061B1CF AB6BE5F3 2BBFA783 24ED106A 7636B9C5 A7BD198D 0158AA4F
5488D08F 38514F1F DF4B4F40 D2181B36 81C364BA 0273C706

$n = 571, \quad f(t) = t^{571} + t^{10} + t^5 + t^2 + 1$

$b =$ 0x 02F40E7E 2221F295 DE297117 B7F3D62F 5C6A97FF CB8CEFF1 CD6BA8CE
4A9A18AD 84FFABBD 8EFA5933 2BE7AD67 56A66E29 4AFD185A 78FF12AA
520E4DE7 39BACA0C 7FFEFF7F 2955727A

$x_G =$ 0x 0303001D 34B85629 6C16C0D4 0D3CD775 0A93D1D2 955FA80A A5F40FC8
DB7B2ABD BDE53950 F4C0D293 CDD711A3 5B67FB14 99AE6003 8614F139
4ABFA3B4 C850D927 E1E7769C 8EEC2D19

$y_G =$ 0x 037BF273 42DA639B 6DCCFFFE B73D69D7 8C6C27A6 009CBBCA 1980F853
3921E8A6 84423E43 BAB08A57 6291AF8F 461BB2A8 B3531D2F 0485C19B
16E2F151 6E23DD3C 1A4827AF 1B8AC15B

Anhang B: Quellcode

Der Anhang B.1 beinhaltet den Quellcode für eine Binärkörper-Multiplikation in der Programmiersprache ANSI-C. Als Erstes ist die Funktion `ff_mul_w32_s3` abgebildet, welche die Binärkörper-Multiplikation auf 32-Bit-Wort-Ebene umsetzt (vgl. Abschnitt 5.1.2). Diese Funktion entspricht Algorithmus 7 mit den Parametern $w = 32$ (Datenwortbreite) und $s = 3$ (Blockgröße). Im Anschluss sind folgende Funktionen, exemplarisch für alle in dieser Arbeit betrachteten Binärkörper, dargestellt:

- `ff_mul_t2_w32_k2` ($t = 2 \rightarrow$ max. Polynomgrad: 63)
- `ff_mul_t4_w32_k2` ($t = 4 \rightarrow$ max. Polynomgrad: 127)
- `ff_mul_t6_w32_k3` ($t = 6 \rightarrow$ max. Polynomgrad: 191)
- `ff_mul_t8_w32_k2` ($t = 8 \rightarrow$ max. Polynomgrad: 255)

Diese Funktionen realisieren jeweils eine Multiplikation im Binärkörper $GF(2^n)$ mit dem Erweiterungsgrad $n < (t \cdot w)$. Ausgehend von der Datenwortbreite $w = 32$, wird die maximale Polynomlänge n jeweils mit Hilfe der Karatsuba-Methode K_2 verdoppelt bzw. K_3 verdreifacht (vgl. Abbildung 5.4). Zum Beispiel verwendet die Funktion `ff_mul_t2_w32_k2` gemäß dem Karatsuba-Verfahren K_2 (Gleichung 3.10) drei Aufrufe der Multiplikationsfunktion auf 32-Bit-Wort-Ebene (`ff_mul_w32_s3`), um Polynome mit einer maximalen Länge von 64 Bit multiplizieren zu können. Entsprechend können mit der Funktion `ff_mul_t8_w32_k2` Polynome bis zur Länge von $8 \cdot 32 = 256$ multipliziert werden. Diese Funktion wird für alle Multiplikationen im Binärkörper $GF(2^{233})$ verwendet (vgl. Kapitel 5 und 6). Der dargestellte Quellcode basiert zum Teil auf der *Number Theory Library* [Sho08] sowie der *Binary Finite Field Library* [Bel01].

Im Anhang B.2 ist der SiLLis-Quellcode für die Performanzanalyse eines On-Chip-Netzwerkes abgebildet (vgl. Abschnitt 6.3.1).

B.1 Binärkörper-Multiplikation

```
void ff_mul_w32_s3 (unsigned int  A, unsigned int  B,
                    unsigned int *C)
{
  unsigned int U[8], V;
  unsigned int C1, C0;

  /* Initialisierung der Nachschlagetabelle U */
  U[0] = 0;            U[1] = B;
  U[2] = U[1] << 1;  U[3] = U[2] ^ U[1];
  U[4] = U[2] << 1;  U[5] = U[4] ^ U[1];
  U[6] = U[3] << 1;  U[7] = U[6] ^ U[1];

  /* Abrollen der Schleife zur Berechnung der 32-Bit-
     Binärkörper-Multiplikation mit Blockgröße s = 3  */
  C0 = U[A & 7];
   V = U[(A >> 3)  & 7]; C1  = V >> 29; C0 ^= V << 3;
   V = U[(A >> 6)  & 7]; C1 ^= V >> 26; C0 ^= V << 6;
   V = U[(A >> 9)  & 7]; C1 ^= V >> 23; C0 ^= V << 9;
   V = U[(A >> 12) & 7]; C1 ^= V >> 20; C0 ^= V << 12;
   V = U[(A >> 15) & 7]; C1 ^= V >> 17; C0 ^= V << 15;
   V = U[(A >> 18) & 7]; C1 ^= V >> 14; C0 ^= V << 18;
   V = U[(A >> 21) & 7]; C1 ^= V >> 11; C0 ^= V << 21;
   V = U[(A >> 24) & 7]; C1 ^= V >> 8;  C0 ^= V << 24;
   V = U[(A >> 27) & 7]; C1 ^= V >> 5;  C0 ^= V << 27;
   V = U[(A >> 30)];     C1 ^= V >> 2;  C0 ^= V << 30;

  /* Korrektur des möglichen Überlaufs von U */
  if (B >> 31) C1 ^= ((A & 0xb6db6db6UL) >> 1);
  if ((B >> 30) & 1) C1 ^= ((A & 0x24924924UL) >> 2);

  /* Zurückschreiben des Ergebnisses */
  C[0] = C0; C[1] = C1;
}
```

```
1  void ff_mul_t2_w32_k2 (unsigned int *A, unsigned int *B,
2                         unsigned int *C) {
3    unsigned int hs0, hs1, hl2[2];
4
5    /* Addition der Teilkoeffizienten */
6    hs0 = A[0] ^ A[1];   hs1 = B[0] ^ B[1];
7
8    ff_mul_w32_s3(A[0], B[0], C);     // 3 Multiplikationen
9    ff_mul_w32_s3(A[1], B[1], C+2);   // halber Polynomlänge
10   ff_mul_w32_s3(hs0,  hs1,  hl2);   // (Karatsuba-Methode K2)
11
12   /* Subtraktion vom mittleren Teilprodukt */
13   hl2[0] ^= C[0] ^ C[2];  hl2[1] ^= C[1] ^ C[3];
14
15   /* Addition der überlappenden Teilprodukte */
16   C[1] ^= hl2[0];  C[2] ^= hl2[1];
17 }
```

```
1  void ff_mul_t4_w32_k2 (unsigned int *A, unsigned int *B,
2                         unsigned int *C) {
3    unsigned int hs0[2], hs1[2], hl2[4];
4
5    /* Addition der Teilkoeffizienten */
6    hs0[0] = A[0] ^ A[2];  hs0[1] = A[1] ^ A[3];
7    hs1[0] = B[0] ^ B[2];  hs1[1] = B[1] ^ B[3];
8
9    ff_mul_t2_w32_k2(A,   B,   C);     // 3 Multiplikationen
10   ff_mul_t2_w32_k2(A+2, B+2, C+4);   // halber Polynomlänge
11   ff_mul_t2_w32_k2(hs0, hs1, hl2);   // (Karatsuba-Methode K2)
12
13   /* Subtraktion vom mittleren Teilprodukt */
14   hl2[0] ^= C[0] ^ C[4];   hl2[1] ^= C[1] ^ C[5];
15   hl2[2] ^= C[2] ^ C[6];   hl2[3] ^= C[3] ^ C[7];
16
17   C[2] ^= hl2[0];  C[3] ^= hl2[1];  // Addition der über-
18   C[4] ^= hl2[2];  C[5] ^= hl2[3];  // lappenden Teilprodukte
19 }
```

```
void ff_mul_t6_w32_k3 (unsigned int *A, unsigned int *B,
                       unsigned int *C)
{
  unsigned int m20a[2], m20b[2], m20[4], c2[4];
  unsigned int m21a[2], m21b[2], m21[4], c4[4];
  unsigned int m30a[2], m30b[2], m30[4], c6[4];

  /* Addition der Teilkoeffizienten */
  m20a[0] =     A[0] ^ A[2];  m20a[1] =     A[1] ^ A[3];
  m20b[0] =     B[0] ^ B[2];  m20b[1] =     B[1] ^ B[3];
  m21a[0] =     A[2] ^ A[4];  m21a[1] =     A[3] ^ A[5];
  m21b[0] =     B[2] ^ B[4];  m21b[1] =     B[3] ^ B[5];
  m30a[0] = m20a[0] ^ A[4];  m30a[1] = m20a[1] ^ A[5];
  m30b[0] = m20b[0] ^ B[4];  m30b[1] = m20b[1] ^ B[5];

  ff_mul_t2_w32_k2(A,   B,   C);      // ------------------
  ff_mul_t2_w32_k2(A+2, B+2, C+4);    // 6 Multiplikationen
  ff_mul_t2_w32_k2(A+4, B+4, C+8);    // mit einem Drittel
  ff_mul_t2_w32_k2(m20, m20a, m20b);  // der Polynomlänge
  ff_mul_t2_w32_k2(m21, m21a, m21b);  // (K3-Karatsuba)
  ff_mul_t2_w32_k2(m30, m30a, m30b);  // ------------------

  /* Subtraktion von den mittleren Teilprodukten */
  c2[0] =   C[0]^  C[4]^m20[0];  c2[1] =   C[1]^  C[5]^m20[1];
  c2[2] =   C[2]^  C[6]^m20[2];  c2[3] =   C[3]^  C[7]^m20[3];
  c4[0] = m20[0]^m21[0]^m30[0];  c4[1] = m20[1]^m21[1]^m30[1];
  c4[2] = m20[2]^m21[2]^m30[2];  c4[3] = m20[3]^m21[3]^m30[3];
  c6[0] =   C[4]^  C[8]^m21[0];  c6[1] =   C[5]^  C[9]^m21[1];
  c6[2] =   C[6]^ C[10]^m21[2];  c6[3] =   C[7]^ C[11]^m21[3];

  /* Addition der überlappenden Teilprodukte */
  C[2]  ^= c2[0];            C[3]  ^= c2[1];
  C[4]   = c2[2] ^ c4[0];    C[5]   = c2[3] ^ c4[1];
  C[6]   = c4[2] ^ c6[0];    C[7]   = c4[3] ^ c6[1];
  C[8]  ^= c6[2];            C[9]  ^= c6[3];
}
```

```
1  void ff_mul_t8_w32_k2 (unsigned int *A, unsigned int *B,
2                         unsigned int *C)
3  {
4    unsigned int hs0[4], hs1[4], hl2[8];
5
6    /* Addition der Teilkoeffizienten */
7    hs0[0] = A[0] ^ A[4];    hs0[1] = A[1] ^ A[5];
8    hs0[2] = A[2] ^ A[6];    hs0[3] = A[3] ^ A[7];
9    hs1[0] = B[0] ^ B[4];    hs1[1] = B[1] ^ B[5];
10   hs1[2] = B[2] ^ B[6];    hs1[3] = B[3] ^ B[7];
11
12   ff_mul_t4_w32_k2(A,   B,   C);    // 3 Multiplikationen
13   ff_mul_t4_w32_k2(A+4, B+4, C+8);  // halber Polynomlänge
14   ff_mul_t4_w32_k2(hs0, hs1, hl2);  // (Karatsuba-Methode K2)
15
16   /* Subtraktion vom mittleren Teilprodukt */
17   hl2[0] = hl2[0] ^ C[0] ^ C[8];
18   hl2[1] = hl2[1] ^ C[1] ^ C[9];
19   hl2[2] = hl2[2] ^ C[2] ^ C[10];
20   hl2[3] = hl2[3] ^ C[3] ^ C[11];
21   hl2[4] = hl2[4] ^ C[4] ^ C[12];
22   hl2[5] = hl2[5] ^ C[5] ^ C[13];
23   hl2[6] = hl2[6] ^ C[6] ^ C[14];
24   hl2[7] = hl2[7] ^ C[7] ^ C[15];
25
26   /* Addition der überlappenden Teilprodukte */
27   C[4]  = C[4]  ^ hl2[0];
28   C[5]  = C[5]  ^ hl2[1];
29   C[6]  = C[6]  ^ hl2[2];
30   C[7]  = C[7]  ^ hl2[3];
31   C[8]  = C[8]  ^ hl2[4];
32   C[9]  = C[9]  ^ hl2[5];
33   C[10] = C[10] ^ hl2[6];
34   C[11] = C[11] ^ hl2[7];
35 }
```

B.2 Performanz-Analysator für On-Chip-Netzwerke

```
listener performance_listener
  parameter
    SENDER_ID = 4;
    DATA_ID   = 15;
  end parameter;

  interface
    flow_id_0 : in std_logic_vector(4 downto 0);
    type_0    : in std_logic_vector(1 downto 0);
    data_0    : in std_logic_vector(3 downto 0);
    flow_id_1 : in std_logic_vector(4 downto 0);
    type_1    : in std_logic_vector(1 downto 0);
    data_1    : in std_logic_vector(3 downto 0);
    request   : in std_logic_vector(3 downto 0);
  end interface;

  internal_variable
    packet_counter : std_logic_vector(31 downto 0) = 0;
  end internal_variable;

  path (request > 0)  is
    if flow_id_0 == SENDER_ID and type_0 == 2 and
          data_0 == DATA_ID then
      packet_counter = packet_counter + 1;
      flush(packet_counter);
      wait(type_0 == 0);
    elsif flow_id_1 == SENDER_ID and type_1 == 2 and
             data_1 == DATA_ID then
      packet_counter = packet_counter + 1;
      flush(packet_counter);
      wait(type_1 == 0);
    end if;
  end path;
end listener;
```