Entwicklung von neuen Algorithmen der Computerarithmetik in Hinsicht auf ihre Nutzung in der Kryptographie

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Viktor Bunimov

Kiel
2005

1. Gutachter	Prof. Dr. rer. nat. Manfred Schimmler
2. Gutachter	Prof. Dr. Thomas Wilke
3. Gutachter	Prof. Dr.-Ing. Christof Paar
Datum der mündlichen Prüfung	7.12.05

Zusammenfassung:

In dieser Arbeit wird eine Reihe neuer Algorithmen aus dem Bereich der ganzzahligen Langzahlcomputerarithmetik für die Anwendungen vor allem aus dem Bereich der modernen Kryptographie entwickelt. Alle hier behandelten Verfahren wurden weiterhin in Bezug auf eine Realisierung in Hardware optimiert. Es werden drei thematische Schwerpunkte behandelt.

Als erstes werden neue Methoden zur Berechnung der Modularmultiplikation aufgezeigt, die sich durch ein besonders günstiges Flächen-Zeit-Produkt auszeichnen. Dabei wird zunächst eine neue Version der Modularmultiplikation von Montgomery entwickelt, bei der die Anzahl der Additionseinheiten gegenüber den besten bisher bekannten Verfahren halbiert werden kann. Danach wird die Verschränkte Modularmultiplikation behandelt. Bezogen zur derzeit als bestgeltenden Version kann hier die Anzahl der Additionseinheiten um einen Faktor drei reduziert werden. Der dritte Algorithmus ist ein in dieser Arbeit neu entwickeltes Verfahren, bei dem nicht nur die Berechnungen, sondern auch die Eingaben in redundanter Form erfolgen. Dadurch kann die Umrechnung von redundanter in nichtredundante Form bei der mehrfachen Anwendung der Modularmultiplikation, wie z.B. bei der Modularen Exponentiation, eingespart werden, ohne zusätzlichen Zeit- bzw. Flächenaufwand zu benötigen. Alle drei Algorithmen werden in die Modulare Exponentiation eingebettet und in Hinsicht auf ihre Komplexität evaluiert.

Das zweite Thema ist ein zeitoptimaler paralleler Algorithmus für die Modularmultiplikation, der eine Zeitkomplexität von $O(\log n)$ aufweist. Dabei wird die Eigenschaft ausgenutzt, dass bei den meisten Anwendungen, wie z. B. bei RSA oder Elliptischen Kurven, viele aufeinander folgende Modularmultiplikationen mit dem gleichen Modulus erforderlich sind.

Das dritte Thema behandelt ein Verfahren für die zeitoptimale Multiplikation. Die Zeitkomplexität dieses Algorithmus liegt ebenso bei $O(\log n)$. Das Verfahren hat eine bessere asymptotische Flächen-Zeit-Komplexität als der in den meisten Prozessoren benutzte Wallace Tree. Darüber hinaus ist der Flächen-Zeit-Aufwand einer realen Implementierung deutlich niedriger als beim Wallace Tree oder im Falle der Schönhage-Strassen-Multiplikation, welche in ihrer Flächen-Zeit-Komplexität besser ist als alle bisher bekannten Verfahren.

Mein Dank gilt Prof. Dr. Manfred Schimmler, der mir ermöglichte auf diesem Forschungsgebiet zu promovieren und mich während meiner ganzen Arbeit mit viel Leidenschaft betreute. Ich danke Prof. Dr. Thomas Wilke für seine Verbesserungsvorschläge und Prof. Dr. Christof Paar für das Einverständnis bei meiner Promotion Gutachter zu sein. Herzlich bedanken möchte ich mich bei meinem Freund Stefan Vocke und meinem Bruder Stanislav Bunimov, die meine Arbeit mehrfach lasen und mit ihren Ratschlägen zur deren Verbesserung viel beitrugen. Ferner bedanke ich mich bei meinen Kollegen Gerd Pfeiffer, Jürgen Noss und Stefan Baumgart, die beim Lesen der ausgewählten Stellen meiner Arbeit mich auf die stilistischen Ungenauigkeiten aufmerksam machten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

Tabellenverzeichnis

1 Einführung

Die Wissenschaft der Verschlüsselung von Informationen (Kryptographie) spielt seit Jahrtausenden eine wichtige Rolle in der menschlichen Gesellschaft. Schon Julius Cäsar benutzte ein Chiffrierverfahren, um Kriegsnachrichten zu verschlüsseln und somit zu verhindern, dass seine Geheimnisse in fremde Hände geraten. Seitdem nahm die Bedeutung der Kryptographie für militärische Zwecke immer mehr zu: Die Datenmengen, die verschlüsselt werden sollten, wuchsen und gleichzeitig stiegen die Anforderungen an die Sicherheit. Einer der Höhepunkte des Bedeutungszuwachses der Kryptographie war die Entschlüsselung der ENIGMA Chiffre durch die Alliierten im Zweiten Weltkrieg, mit deren Hilfe der Krieg im Atlantik gewonnen wurde. Darüber hinaus kann dies als ein entscheidender Wendepunkt im Verlauf des Zweiten Weltkriegs geltend gemacht werden [69], [37].

Bis Mitte des 20. Jahrhunderts spielte die Kryptographie nur im militärischen Bereich und in der Politik eine wichtige Rolle. Mit der Einführung der EDV wurde sie aber auch für die Wirtschaft unabdingbar, denn die stetig wachsende Speicherkapazität der immer kleiner werdenden Datenträger (elektronische Speichermedien, Filme, etc.) bringt unweigerlich Sicherheitsprobleme für die Unternehmen mit sich. Um zu verhindern, dass unbefugte Personen in der Lage sein könnten, neue Entwicklungen, Kundendaten und andere wichtige Informationen zu stehlen, müssen die Daten heute verschlüsselt werden. Ein weiterer Aspekt zur Steigerung der Sicherheit ist das Problem der Authentifizierung von Personen und Nachrichten. Zudem brachte die Globalisierung zusätzliche Veränderungen im Nutzungsprofil der Kryptographie: Firmen verfügen heute über ein Netz von Filialen in unterschiedlichen Ländern, deren Daten zwischen den verschiedenen Standorten elektronisch übertragen und somit von nicht autorisierten Personen abgefangen werden könnten. Es ist somit unumgänglich, die Daten zu verschlüsseln, um den Zugriff durch Unbefugte zu verhindern.

In der modernen Kryptographie existieren zwei sich vom Wesen her unterscheidende Ansätze: Erstens die so genannten symmetrischen Verschlüsselungsverfahren, bei denen der Klartext und der Chiffretext mit demselben Schlüssel chiffriert bzw. dechiffriert werden. Die symmetrischen Chiffrierverfahren werden auch als „Secret Key“-Ver-

fahren bezeichnet, da der Schlüssel geheim sein muss; denn sollte eine dritte unbefugte Person den Schlüssel kennen, wäre diese in der Lage, den Chiffretext zu dechiffrieren und somit den Klartext zu lesen. Im Allgemeinen gelten die symmetrischen Chiffrierverfahren als sehr schnell und erlauben daher eine Informationsübertragung mit hohen Datenraten [72].

Die zweite Kategorie stellen die asymmetrischen Chiffrierverfahren dar, deren Besonderheit die Verwendung zweier unterschiedlicher Schlüssel ist. Mit dem ersten Schlüssel, der öffentlich bekannt ist, ist jede Person in der Lage, einen Klartext zu chiffrieren, weshalb solche Verfahren auch „Public Key"-Verfahren genannt werden. Der zweite Schlüssel, der nur dem Empfänger bekannt ist und daher als geheimer Schlüssel bezeichnet wird, dient dazu den Chiffretext zu dechiffrieren. Der Vorteil eines solchen Verfahrens ist unter anderem die Möglichkeit der Schlüsselübertragung über einen unsicheren Kanal. Die asymmetrischen Verfahren gelten als sichere Verschlüsselungen [72], allerdings sind die Operationen, die nötig sind, um eine solche Nachricht zu ent- bzw. verschlüsseln, mit einem erhöhten Rechenaufwand verbunden. Demzufolge haben asymmetrische Kryptoverfahren eine niedrige Datenrate. Um diese zu erhöhen, werden solche Verfahren häufig direkt in Hardware implementiert, was eine Beschleunigung um Faktor 100 oder mehr mit sich bringen kann.

Wird ein Problem mit Hilfe von Software (Implementierung der Problemlösung in einer Programmiersprache und Ausführung des so entstehenden Programms auf einer vorgegeben Hardware) gelöst, so muss vor allem die Zeitkomplexität als Bewertungsmaß des zugrundeliegenden Algorithmus betrachtet werden. Diese entspricht dem Produkt aus der Anzahl der Maschinenbefehle und der durchschnittlichen Dauer eines Befehls auf der Hardware. Wird eine Hardwarelösung gewählt, die eigens für einen speziellen Zweck entwickelt wurde und somit keinen Maschinenbefehlssatz besitzt, so entspricht die Zeitkomplexität dem Produkt aus der Anzahl der Takte und der Dauer eines Taktes. Bei einer speziellen Hardwarelösung ist nicht nur die Zeit-, sondern auch die Flächenkomplexität bedeutsam, die ein Maß für die Fläche des Chips, in dem die entsprechenden Funktionen implementiert sind, darstellt. Da die Chipfläche eine technologieabhängige Größe ist, wird ein abstraktes Maß für die in dieser Arbeit benutzten Bauelemente definiert, welches durch eine Evaluierung verschiedener Technologien erarbeitet wurden. Ähnlich wie bei der Flächenkomplexität ist auch die Zeitkomplexität technologieabhängig. Demzufolge wird gleichermaßen auch ein einheitliches Maß für die Zeitkomplexität definiert. Damit dieses Maß eine realistische

Bewertung der zu betrachtenden Algorithmen erlaubt, wird dessen Verifikation in Abschnitt 3.3 ebenfalls durch eine Gegenüberstellung verschiedener Technologien durchgeführt.

Die Zeit- und die Flächenkomplexität sind meistens voneinander abhängig, da durch eine Verringerung der Zeitkomplexität häufig die Flächenkomplexität eines Verfahrens wächst und umgekehrt. Um in der Lage zu sein allgemeingültige Aussagen zu treffen, mit denen unterschiedliche Verfahren miteinander verglichen werden können, ist es daher notwendig ein technologieunabhängiges Maß zu definieren, bei dem sowohl die Flächen- als auch die Zeitkomplexität zum Tragen kommt. Auf Grund der hier aufgeführten Überlegungen empfiehlt sich die Verwendung des Flächen-Zeit-Produkts als ein solches Maß [70]. In Abschnitt 3.3 werden aus diesem Grund die Zeit- und Flächenkomplexität der meisten in dieser Arbeit benutzten Bauelemente definiert und deren Flächen-Zeit-Produkt berechnet.

Viele asymmetrische Verfahren (unter ihnen auch die am häufigsten benutzten Verfahren wie RSA [62] oder Elliptische Kurven [50]) basieren auf der wiederholten Anwendung derselben Rechenoperation: Der Modularmultiplikation. Diese ist eine mathematische Operation, deren Eingabe aus drei ganzen Zahlen besteht, wobei die ersten zwei als Operanden und die dritte als Modulus bezeichnet werden. Das Ergebnis der Modularmultiplikation ist der Rest der Division des Produktes der beiden Operanden durch den Modulus. In modernen kryptographischen Anwendungen werden Operanden mit einer Länge von mehreren Hundert Bits verwendet; so werden z.B. beim RSA-Verfahren oft Operanden ab 1024 Bits und mehr benutzt. Mit steigender Operandengröße erhöht sich die Länge der für die Speicherung erforderlichen Register in der Hardwareschaltung. Ein Register besteht aus einer Menge von Flipflops, die parallel zueinander schalten, wobei jedes Flipflop ein Bit speichern kann. Mit steigender Operandenlänge erhöhen sich sowohl der Bedarf an Logik und somit auch die Flächenkomplexität als auch die Zeitkomplexität der Gesamtschaltung. Demzufolge ist es sinnvoll, die Komplexität der Schaltung durch algorithmische Verbesserungen zu optimieren.

Es gibt mehrere Algorithmen für die Modularmultiplikation (eine ausführliche Auseinandersetzung mit diesen Algorithmen wird im Kapitel 2 durchgeführt). Am häufigsten werden dabei die Modularmultiplikation von Montgomery [53] und die verschränkte (Interleaved) Modularmultiplikation [12] (eine genaue Beschreibung dieser Algorithmen findet in den Abschnitten 4.1 und 4.2 statt) benutzt. Die verschränkte Modularmultiplikation war der erste Algorithmus für die Modularmultiplikation, der es erlaubte die

Multiplikation und die Modulare Reduktion (Berechnung des Divisionsrestes) ineinander zu verschachteln und damit die Bitlänge des Zwischenergebnisses zu reduzieren. Es wurde weiterhin erreicht, dass die verschränkte Modularmultiplikation auf drei schnelle Additionen pro Schleifendurchlauf reduziert wurde [42]. Der Nachteil dabei liegt in der Tatsache, dass die Vergleiche zeitineffizient sind. Der zweite Algorithmus, die Modularmultiplikation von Montgomery, ist heute der am häufigsten benutzte Algorithmus, dessen Vorteil darin besteht, dass er keine komplizierten Vergleiche enthält. Die beste existierende Lösung [40] reduziert den Algorithmus auf zwei redundante Additionen pro Schleifeniteration.

Ziel dieser Arbeit ist die Entwicklung mehrerer Algorithmen für die Modularmultiplikation mit geringerer AT-Komplexität (area-time complexity) als bei allen bisher existierenden Verfahren. Dabei ist die AT-Komplexität das Produkt der Flächen- und der Zeitkomplexität. Dieses wird bei der Hardwareentwicklung oft als das Maß für die Güte der neuen Schaltung verwendet. Um die AT-Komplexität der bereits existierenden Verfahren zu reduzieren werden drei neue Algorithmen entwickelt: Der erste Algorithmus basiert auf der Modularmultiplikation von Montgomery, die auf eine schnelle Addition pro Schleifeniteration reduziert wird, wobei hierdurch eine Verbesserung um das Vierfaches gegenüber dem besten bisher bekannten Verfahren [40] erreicht wird. Der zweite Algorithmus baut auf der verschränkten Modularmultiplikation auf, dabei wird die Anzahl der Additionen auf eine pro Schleifeniteration reduziert, die ineffizienten Vergleiche werden nicht mehr benötigt. Insgesamt wird eine Komplexitätsverbesserung mindestens um den Faktor sechs gegenüber dem gegenwärtig führenden Verfahren [42] erreicht.

Der Nachteil des Algorithmus von Montgomery und der verschränkten Modularmultiplikation ist, dass die Verwendung der schnelleren Addition eine spezielle redundante Zahlendarstellung erfordert. Die Eingaben für die Modularmultiplikation müssen hingegen in der binären Standarddarstellung erfolgen. Da bei den meisten Anwendungen die Modularmultiplikation mehrmals wiederholt wird, wird eine Konvertierung aus der redundanten Zahlendarstellung in die binäre Standarddarstellung notwendig. Das Problem der Konvertierung, die zusätzlichen Flächen- und Zeitaufwand benötigt, wird durch den dritten Algorithmus gelöst. Dieser bietet ein neues mathematisches Modell an, das nicht nur die Nutzung der schnellen Addition, sondern auch Eingaben in redundanter Zahlendarstellung erlaubt. Dadurch ist die mehrmalige Konvertierung bei wiederholter Modularmultiplikation überflüssig.

Die drei neu entwickelten Algorithmen werden danach in die Exponentiation eingebettet.

Darüber hinaus werden zwei weitere neue Algorithmen für die parallele Modularmultiplikation und die parallele Multiplikation vorgestellt. Sowohl bei dem Algorithmus für die parallele Modularmultiplikation, als auch bei dem für die parallele Standard-Multiplikation beträgt die Zeitkomplexität O(log$n$). Der Vorteil des Multiplikationsalgorithmus gegenüber den existierenden Lösungen liegt in der Tatsache, dass er nicht nur eine gute asymptotische Komplexität, sondern auch kleine Koeffizienten besitzt und somit in praktischen Anwendungen implementiert werden kann.

Diese Arbeit ist wie folgt strukturiert: Kapitel 2 bietet einen Vergleich der existierenden Algorithmen für die Modularmultiplikation. In Kapitel 3 werden einige der am häufigsten benutzten Bauelemente (Schaltungen) vorgestellt, ein Komplexitätsmodell definiert und die Zeit- und Flächenkomplexität der oben beschriebenen Bauelemente erläutert. Außerdem wird eine Analyse unterschiedlicher VLSI-Technologien (Very Large Scale Integration) durchgeführt, die die praktische Relevanz des Komplexitätsmodells bestätigen.

In Kapitel 4 werden drei neue Algorithmen für die Modularmultiplikation entwickelt. Zunächst wird in Abschnitt 4.1 die Modularmultiplikation von Montgomery [53] eingeführt, bei der die einzelnen Bits des ersten Operanden schrittweise vom niederwertigsten bis zum höchstwertigsten Bit mit dem zweiten Operand multipliziert werden. Die so entstehenden Teilprodukte werden addiert und produzieren eine Folge von Zwischenergebnissen, wobei nach jeder Addition eine Normierung erfolgt. Insgesamt benötigt die Modularmultiplikation von Montgomery n Schleifendurchläufe. Der Vorteil der Montgomery-Modularmultiplikation liegt in der Tatsache, dass während der Berechnung keine Vergleiche erforderlich sind, die aufwendig in Hinsicht auf die Komplexität sind. Allerdings werden in der Originalfassung langsame Standardadditionen angewendet. Dem Montgomery-Verfahren, welches einen Meilenstein in der Geschichte der modularen Arithmetik darstellt, folgten eine Reihe von Optimierungen und Implementierungen durch andere Wissenschaftler, welche in Kapitel 2 detailliert behandelt werden. Als Basis für die in dieser Arbeit entwickelte Version dient die Arbeit von Kim, Kang und Choi [40], bei der die Standardadditionen durch geschickte schnelle Additionen ersetzt werden, die das Verfahren von Montgomery auf zwei redundante Additionen pro Schleifendurchlauf reduzieren. In Abschnitt 4.1.2 wird zunächst deren Verbesserung

detailliert beschrieben. Danach folgt in Abschnitt 4.1.3 die Entwicklung eines neuen Algorithmus, in dem die Eigenschaft ausgenutzt wird, dass alle Werte, die während der einzelnen Schleifendurchläufe zum Zwischenergebnis aufaddiert werden, konstant sind. Alle möglichen Kombinationen dieser Summanden werden vor der Schleife einmal berechnet und auf eine spezielle Art und Weise gespeichert. Während jedes Schleifendurchlaufs wird vorhergesagt, welcher Wert im nächsten Schleifendurchlauf aufaddiert werden muss. Dieser Korrekturwert wird danach aus dem Speicher gelesen und Hilfe einer schnellen Addition zum Zwischenergebnis mit addiert. Somit kann in einem Schleifendurchlauf eine einzige, anstatt zwei nacheinander folgenden Additionen durchgeführt werden. Demzufolge reduziert sich sowohl die Flächenkomplexität, als auch die Zeitkomplexität des Verfahrens, was wiederum eine Verbesserung der AT-Komplexität um den Faktor vier mit sich bringt. Weiterhin verifiziert die in Abschnitt 4.1.4 durchgeführte Komplexitätsanalyse die Komplexität des neuen Algorithmus. Die Grundidee dieser Version wurde in [17] veröffentlicht; einige Weiterentwicklungen mit dem Vergleich zu anderen Algorithmen wurden in [22] und [23] publiziert.

In Abschnitt 4.2 wird die verschränkte Modularmultiplikation erörtert, die der Idee der Modularmultiplikation von Montgomery ähnlich ist: Der erste Operand wird bitweise mit dem zweiten Operanden multipliziert und die so entstandenen Teilprodukte werden miteinander addiert. Vor jeder Addition einer solchen Art erfolgt eine Verdopplung des Zwischenergebnisses; danach findet eine modulare Reduktion statt. Der Unterschied zur Modularmultiplikation von Montgomery liegt in der Tatsache, dass der erste Operand nicht vom niederwertigsten zum höchstwertigsten Bit, sondern vom höchstwertigsten zum niederwertigsten Bit durchlaufen wird. Der Vorteil dabei ist, dass die verschränkte Modularmultiplikation das Ergebnis einer Modularmultiplikation berechnet, wohingegen der Algorithmus von Montgomery zweimal durchlaufen werden müsste, um das Ergebnis der Modularmultiplikation zu bestimmen. Der Nachteil liegt in der Tatsache, dass die verschränkte Modularmultiplikation mehrere Vergleiche benötigt, die wiederum die Anwendung einer schnellen Addition, wie beim Algorithmus von Montgomery, erschweren.

Die verschränkte Modularmultiplikation benötigt ebenfalls n Schleifendurchläufe, wobei in jedem Schleifendurchlauf das Ergebnis des vorherigen verdoppelt und danach, falls erforderlich, der zweite Operand aufaddiert wird. Das Zwischenergebnis wird einer Modularen Reduktion unterzogen, welche zwei Vergleiche und bis zu zwei Subtraktionen des Modulus enthält. In Abschnitt 4.2.2 werden mehrere Verbesserungen vorge-

schlagen: Zuerst werden die aufwändigen Vergleiche mit dem Modulus durch einfache Vergleiche mit einer Zweierpotenz ersetzt. Bei einem Vergleich zweier Zahlen müssen alle Bits beider Zahlen paarweise miteinander verglichen werden, wohingegen bei einem Vergleich mit der entsprechenden Zweierpotenz bei der zu vergleichenden Zahl nur ein einziges Bit überprüft wird. Ist dieses eine Eins, so ist die Zahl größer oder gleich der zugehörigen Zweierpotenz, ansonsten ist die Zahl kleiner. Danach wird die Anzahl der Subtraktionen des Modulus abgeschätzt. Noch vor den Schleifendurchläufen wurden alle möglichen Subtraktionswerte einmal berechnet und in einer Tabelle (Lookup Table) gespeichert. Die Abschätzung der Subtraktionsanzahl dient als Steuerung (Adressierung) für die Lookup Table. Durch die oben beschriebenen Verbesserungen wird die Anzahl der Operationen in einem Schleifendurchlauf auf zwei Additionen reduziert: Die erste ist für die Addition des Produktes des entsprechenden Bits des ersten Operanden mit dem zweiten Operanden; die zweite dient der Reduktion. Gleichwohl verändert sich der zweite Operand, der während des Schleifendurchlaufs aufaddiert wird, nicht innerhalb der Schleife. Als eine weitere Verbesserung wird auf die Addition mit dem zweiten Operanden verzichtet. Stattdessen wird der zweite Operand in die Lookup Table eingefügt. Die letzte Verbesserung halbiert die Anzahl der Additionen in einem Schleifendurchlauf. In Abschnitt 4.2.3 wird die verbliebene Addition durch eine Carry-Save-Addition ersetzt, was eine weitere Verringerung der Zeitkomplexität des Verfahrens bewirkt. In Abschnitt 4.2.4 wird die Komplexitätsanalyse dieser neuen Verfahren durchgeführt. Einige in diesem Abschnitt vorgestellte Verbesserungen wurden bereits in [18], [21], [22], [23], [64] veröffentlicht.

Alle existierenden Algorithmen – einschließlich der in den Abschnitten 4.1 und 4.2 präsentierten Verfahren – haben einen nicht zu vernachlässigbaren Nachteil: Obwohl während der Modularmultiplikation selbst im redundanten Zahlensystem gerechnet werden kann, erfordern die Modularmultiplikationen die Eingaben in nichtredundanter Form. Die Ausgaben dagegen werden in der redundanten Form berechnet, demzufolge wird nach jeder Modularmultiplikation eine Konvertierung aus der redundanten Form in die nichtredundante Form benötigt, die zusätzliche Hardware und Zeit erfordert. Dementsprechend steigt die Zeit- und die Flächenkomplexität und folglich auch deren Produkt. Der in Abschnitt 4.3 beschriebene Algorithmus bietet eine Möglichkeit, diesen Nachteil zu beheben. Während die neuen Algorithmen aus den Abschnitten 4.1 und 4.2 auf bereits existierenden mathematischen Modellen basieren, wird in Abschnitt 4.3 ein vollkommen neues mathematisches Modell entwickelt. Der Vorteil dieses Modells ge-

genüber anderen existierenden Verfahren für die Modularmultiplikation liegt in der Tatsache, dass sowohl alle Berechnungen als auch alle Eingaben im redundanten Zahlensystem erfolgen. Die Idee des neuen Algorithmus der Modularmultiplikation, basierend auf der Veränderung eines Operanden, ist grundsätzlich einfach: Die einzelnen Bits des ersten Operanden werden mit dem zweiten Operand multipliziert, und diese Teilprodukte werden zum Zwischenergebnis hinzuaddiert, was der Modularmultiplikation von Montgomery und der verschränkten Modularmultiplikation entspricht. Allerdings wird dabei nicht das Zwischenergebnis jedes Mal reduziert und um ein Bit verschoben, sondern der zweite Operand wird in jedem Schleifendurchlauf verdoppelt und einer Modularen Reduktion unterzogen. Als Folge steigt der Wert des Zwischenergebnisses viel langsamer an als bei den anderen Verfahren. In Abschnitt 4.3.2 werden verschiedene Optimierungsvorschläge diskutiert: Der langsame Vergleich mit dem Modulus wird durch den schnelleren Vergleich mit einer Zweierpotenz ersetzt und die Subtraktionen von dem Modulus werden durch die Reduktion mit Hilfe einer Lookup Table mit vorberechneten Werten durchgeführt. Die Verdopplung mit der anschließenden Reduktion des zweiten Operanden und die Summenbildung des Teilproduktes mit dem Zwischenergebnis werden nebenläufig ausgeführt, hierdurch wird die Zeitkomplexität des Verfahrens verbessert. In den Abschnitten 4.3.3 und 4.3.4 werden zwei unterschiedliche Versionen der Modularmultiplikation, basierend auf der Veränderung eines Operanden, vorgestellt. Die nachfolgenden Abschnitte 4.3.5 und 4.3.6 stellen eine Weiterentwicklung der Version aus Abschnitt 4.3.3 dar: Die Addition im redundanten Zahlensystem ersetzt die Addition im nichtredundanten Zahlensystem, das heißt, dass die Addition mit Hilfe eines Carry-Save-Addierers erfolgt, dessen Anwendung wiederum zu einer Verbesserung der Zeitkomplexität des Verfahrens führt. Darüber hinaus werden die Eingänge auf eine solche Art und Weise erweitert, dass die Eingabe im nichtredundanten Zahlensystem möglich wird. Mit dieser Optimierung wird die aufwändige Konvertierung ins nichtredundante Zahlensystem nach jeder Modularmultiplikation überflüssig, was die Anwendung mehrerer Modularmultiplikationen mit dem gleichen Modulus vereinfacht. Die Komplexitätsanalyse und die Zusammenfassung beenden den Abschnitt 4.3. Die meisten Ideen dieses Abschnittes können in [67], [68] und [24] nachgelesen werden.

Kapitel 5 beschreibt die Modulare Exponentiation [34] und [50], die auf der Modularmultiplikation basiert. Zu diesem Zweck wird in dieser Arbeit in Abschnitt 5.1 einer der Algorithmen für die Modulare Exponentiation vorgestellt. In den darauf folgenden

Abschnitten 5.2, 5.3 und 5.4 werden die Algorithmen für die Modularmultiplikation von Montgomery (Abschnitt 4.1), die verschränkte Modularmultiplikation (Abschnitt 4.2) und die Modularmultiplikation basierend auf der Veränderung eines Operanden (Abschnitt 4.3) in die Modulare Exponentiation eingebettet. Darüber hinaus wird in Abschnitt 5.5 die Komplexitätsanalyse der Modularen Exponentiation mit unterschiedlichen Algorithmen für die Modularmultiplikation durchgeführt.

Kapitel 4 und Kapitel 5 konzentrieren sich auf neue Algorithmen für Modulare Arithmetik, die in Hinsicht auf das Flächen-Zeit-Produkt optimiert werden. Solche Optimierungen sind in dem Falle wichtig, wenn z.B. für die Massenfertigung die Kosten minimiert werden müssen. Die Optimierung der Modularmultiplikation in Bezug auf das Produkt von Fläche und Zeit ist auch unter anderen Gesichtspunkten wichtig: Die Modularmultiplikation kann als eingebettete Funktion bei Smartcards, Handys und anderen Kommunikationsgeräten implementiert werden, da solche Geräte eine begrenzte Fläche zur Verfügung stellen. Außerdem ist die Taktrate aus Gründen der Leistungseffizienz häufig niedriger als bei stationären Geräten. Darüber hinaus spielt der Stromverbrauch, welcher neben anderen Faktoren stark von der Fläche des Chips abhängt, eine wichtige Rolle, insbesondere bei mobilen Geräten kommt dies zum Tragen, da deren Leistungsfähigkeit durch die begrenzte Stromkapazität der Batterie eingeschränkt ist.

Manchmal spielen die Kosten und damit das AT-Produkt aber auch eine untergeordnete Rolle. Dann wird versucht die Datenrate zu minimieren und eine zeitoptimale Lösung zu entwickeln. Eine solche Vorgehensweise ist z. B. häufig in technisch-wissenschaftlichen Anwendungen wichtig, da es sich hierbei meist um Einzellösungen, die somit nur einmalige Kosten verursachen, und nicht um Massenprodukte handelt. Ein Beispiel dafür bieten kryptographische Verfahren wie z.B. RSA, für deren Implementierung kostengünstige Lösungen gesucht werden, die es ermöglichen, mehrere Millionen Hardware-Chips zu bauen. Beim Versuch, diese Verfahren zu brechen, wird dagegen oft nur eine einzige Implementierung angestrebt, die es erlaubt, die verschlüsselte Nachricht schnell zu brechen. Diese niedrige Zeitkomplexität wird oft durch hohe Kosten und somit hohen Flächenaufwand erreicht.

Die folgenden Kapitel 6 und Kapitel 7 zeigen zwei zeitoptimale neue Algorithmen für die Standard-Modularmultiplikation und für eine Multiplikation, die mit logarithmischer Zeitkomplexität arbeiten.

In Kapitel 6 wird ein neuer zeitoptimaler Algorithmus für die Modularmultiplikation vorgestellt, der eine logarithmische Zeitkomplexität besitzt. Dies wird erreicht, indem mehrere nur vom Modulus abhängige Werte initial berechnet und dann für mehrere Modularmultiplikationen mit dem gleichen Modulus in eine Lookup Table gespeichert werden. In jeder Modularmultiplikation wird zuerst die Multiplikation, für die ein Wallace Tree [86] mit der Zeitkomplexität von O(log$n$) benutzt wird, durchgeführt. Danach werden die n höchstwertigsten Bits durch die in der Lookup Table gespeicherten Werte parallel zueinander ersetzt. Somit wird eine $2n$-bit-lange Zahl durch eine Summe von n n-bit-langen Zahlen ersetzt, die wiederum mit dem Wallace Tree aufaddiert werden und ein Ergebnis liefern, das durch eine n+log$n$-bit-lange Zahl repräsentiert ist. Diese Berechnungen werden solange wiederholt, bis das Ergebnis länger als n-bit ist. Die Idee für dieses Verfahren wurde bereits in [20] veröffentlicht.

Ein zeitoptimaler Algorithmus für die Standard-Modularmultiplikation wird in Kapitel 7 aufgezeigt. Es gibt viele Algorithmen für die Multiplikation [29], [38], [41], [73], [74], [76], [83], [86], [90], [91] und [92], von denen viele eine AT-Komplexität kleiner als O(n^2) besitzen [38], [41], [72], [83], [90], [91] und [93]. Für eine reale Implementierung in Hardware sind diese Algorithmen jedoch weniger geeignet. Einige von ihnen (z. B. [38] oder [83]) haben eine Zeitkomplexität von mehr als O(n). Andere (z. B. [73] und [90]) haben zwar eine optimale Zeitkomplexität von O(log$n$) und eine hohe AT-Komplexität von O($n$log^2nloglog$n$), benötigen bei der Hardwareimplementierung aber sehr hohe Koeffizienten und sind somit für die praktische Anwendung nur beschränkt geeignet. In modernen Prozessoren wird heute für die Multiplikation ein Wallace Tree mit nachgelagertem Carry-Look-Ahead-Addierer [86] benutzt, der die optimale Zeitkomplexität O(log$n$) hat. Die Flächenkomplexität des Wallace Tree liegt dagegen bei O(n^2). Dementsprechend ist die AT-Komplexität des Wallace Tree O(n^2log$n$). Der Algorithmus, der in Kapitel 7 vorgestellt wird, hat ebenso wie der Wallace Tree die optimale Zeitkomplexität O(log$n$); die Flächenkomplexität liegt dagegen bei O(n^2/log^2n). Demzufolge beträgt die AT-Komplexität des Algorithmus O(n^2/log$n$). Dabei sind die Konstanten, die für die Hardwareimplementierung nötig sind klein. Der Grundgedanke des Algorithmus ist einfach: Es findet eine dreifache Unterteilung des ersten Operanden in mehrere Intervalle statt. Dabei wird die Multiplikation kleinerer Intervalle mit dem zweiten Operanden durch das Einfügen vorberechneter Werter ersetzt, die in einer Lookup Table gespeichert sind. Die Ergebnisse in jedem mittleren Intervall werden mit einem einfachen Addierer aufsummiert. Die so gewonnenen Ergebnisse werden mit

dem Wallace Tree aufaddiert. Die Grundidee dieses Verfahrens wurde in [19] veröffentlicht.

In Kapitel 8 folgt die abschließende Zusammenfassung, der in dieser Arbeit gewonnenen Ergebnisse, und es wird eine Wertung geliefert. Ferner wird ein Ausblick auf mögliche weitere Einsatzgebiete gegeben.

2 Bisherige Lösungen für die Modularmultipli–kation

Modularmultiplikation ist eine mathematische Operation, bei der der ganzzahlige Divisionsrest des Produkts zweier Operanden und einer dritten Zahl, dem so genannten Modulus, berechnet wird. In den meisten Anwendungen sind die Operanden kleiner als der Modulus, der wiederum oft eine ungerade Zahl in der Größenordnung zwischen 2^{150} und 2^{8000} ist. Kryptografische Verfahren stellen das Haupteinsatzgebiet für die Modularmultiplikation dar. So besteht z.B. die Ver- und die Entschlüsselung bei dem als Standard geltendem RSA-Kryptoverfahren [62] jeweils aus einer Modularen Exponentiation [34], [50] und [72]. Die Modularmultiplikation ist wiederum die Grundoperation bei den meisten Algorithmen der letzteren Verfahren. Hierbei wird diese mehrmals nacheinander ausgeführt, was zu einem hohen Rechenaufwand führt. Dies ist der Grund dafür, dass mehrere Algorithmen für die Modularmultiplikation existieren und weitere noch effizientere Methoden entwickelt werden.

Der älteste Algorithmus für die Modularmultiplikation ist die klassische Modularmultiplikation [41]. Hierbei wird zuerst die Multiplikation und dann die modulare Reduktion, bei der der Divisionsrest bestimmt wird durchgeführt. Ein Nachteil der klassischen Modularmultiplikation liegt in der Tatsache, dass das Zwischenergebnis die doppelte Länge der Eingaben besitzt, da das Ergebnis der Multiplikation zweier n-bit Zahlen eine $2n$-bit Zahl ist. Die doppelte Zahlenlänge führt bei einer Implementierung in der Hardware zu einer Erhöhung des Aufwandes der Schaltung, was wiederum eine deutliche Verschlechterung der Gesamtkomplexität der zu betrachtenden Anwendung mit sich bringt. Im letzten Viertel des zwanzigsten Jahrhunderts wurde daher eine Reihe weiterer Algorithmen entwickelt:

In einem – der Modularmultiplikation von Barrett [8] – wurde eine Modularmultiplikation durch mehrere Standardmultiplikationen ersetzt. Der Vorteil dabei ist, dass Barrett eine eventuell schon vorhandene Hardware für Multiplikation wie z.B. einen Wallace Tree [86] anwenden kann. Jedoch zeigen sich dabei auch mehrere Nachteile: Ähnlich wie die Klassische Modularmultiplikation erfordert der Algorithmus von Bar-

rett mindestens ein Register, das doppelt so groß ist wie die Eingabewerte. Darüber hinaus ist die Zeitkomplexität bei Barrett dreimal so hoch wie bei einer normalen Multiplikation. Daraus resultierend ist auch die AT-Komplexität um Faktor drei größer als bei der Verwendung der Multiplikation.

Der Algorithmus von Barrett wurde in mehreren Arbeiten weiterentwickelt. So beschleunigte [33] die Modularmultiplikation von Barrett durch eine Pipeline im Wallace Tree und durch die Anwendung des Algorithmus von Booth [15]. Letzterer erlaubt bei der Berechnung des Produktes zweier Zahlen durch die geschickte Benutzung der Basis vier, die Anzahl der bei der Multiplikation entstehenden Teilprodukte zu halbieren. Trotz dieser Optimierungen ist die AT-Komplexität relativ hoch. Eine weitere Verbesserung, welche auf vergleichbaren Ideen wie in [33] basiert, liefert [56].

Eine andere Vorgehensweise bei der Modularmultiplikation liegt in der Benutzung des RNS (Residue Number System), welches auf folgender Grundidee aufbaut: Durch die Anwendung des Chinesischen Restesatzes wird das Problem der Modularmultiplikation in eine Menge von kleineren Problemen unterteilt. Mehrere Arbeiten setzen diese Methode als Lösungsansatz ein. So wird bei [1], [5] und [6] die Modularmultiplikation in kleinere Probleme unterteilt, die mit Hilfe von gewöhnlichen Standardprozessoren (z.B. 32-bit Prozessoren) gelöst werden können. Eine weitere Möglichkeit ist es, das RNS direkt bei der Montgomery Modularmultiplikation (der dazugehörige Algorithmus wird später in dieser Arbeit detailliert betrachtet) anzuwenden. Somit wird im so genannten MRNS (Montgomery Residue Number System) gearbeitet, wie z.B. in [60] und [61]. Allerdings dienen solche Lösungen nur der Skalierung des Problems, die AT-Komplexität wird hierbei nicht reduziert.

Ein weiterer Algorithmus ist die Interleaved Modularmultiplikation (verschränkte Modularmultiplikation). Die Grundidee dieses Algorithmus ist die Bildung der Teilprodukte einzelner Bits eines Operanden mit dem zweiten Operanden. So wird in jedem Schleifendurchlauf ein solches Produkt zum Zwischenergebnis dazuaddiert, welches nachfolgend der Modularen Reduktion unterzogen wird. Dieser Algorithmus wird in [12] präsentiert und in [75] verbessert. Später folgte eine ganze Reihe von Veröffentlichungen, bei denen die verschränkte Modularmultiplikation überarbeitet wurde. So ersetzt [57] den aufwändigen Vergleich mit dem Modulus durch den schnellen Vergleich mit einer Zweierpotenz. Eine weitere Optimierung [69] basiert darauf, dass die einzelnen Operationen innerhalb eines Schleifendurchlaufs parallelisiert wurden, wodurch sich die Zeitkomplexität verringert. Eine der besten Optimierungen in Bezug auf das

Flächen-Zeit-Produkt wurde in [42] veröffentlicht. Die Interleaved Modularmultiplikation wird mit drei redundanten Addierern durchgeführt. Diese Lösung hat allerdings zwei Nachteile: Zum einen erhöhen drei redundante Addierer (Carry-Save-Addierer) den Flächenaufwand; zum anderen wird während der Berechnung mehrmals ein Vergleich mit Null durchgeführt, wobei der Autor in [42] auch negative Zahlen verwendet, was den oben beschriebenen Vergleich allerdings sehr aufwendig macht, da dieser in einem redundanten Zahlensystem durchgeführt wird. Eine weitere Möglichkeit der Nutzung der redundanten Zahlen bieten [77] und [78], wobei die einzelnen Ziffern aus {-2, -1, 0, 1, 2} bestehen. Die Addition von solchen Zahlen hat im Vergleich zu binären Zahlen eine höhere Zeit- und Hardwarekomplexität. Außerdem benutzen die beiden oben beschriebenen Algorithmen und weitere Algorithmen wie [27], [42] oder [54] höhere Basen. Dies bringt zwar eine Verringerung der Zeitkomplexität, allerdings nur auf Kosten der Flächenkomplexität. Eine weitere Version der verschränkten Modularmultiplikation wurde in [59] veröffentlicht. Hierbei werden ebenso Carry-Save-Addierer, Tabellen mit vorberechneten Werten und höhere Basen benutzt. Allerdings sind sowohl die Anzahl der Carry-Save-Addierer als auch der Umfang der Tabellen sehr groß und die Flächenkomplexität demzufolge relativ hoch. In [39] wurde eine weitere Version der verschränkten Modularmultiplikation aufgezeigt, die ebenfalls den Carry-Save-Addierer benutzt. Einige Werte werden dabei vorberechnet und können später mehrmals angewendet werden. Der Nachteil dabei ist, dass diese Werte in Registern gespeichert werden und sich somit die Flächenkomplexität der Schaltung erhöht. Ferner gibt es Arbeiten wie [11], in denen eine für FPGA (Field Programmable Gate Arrays) optimierte Algorithmen präsentiert werden. Solche Varianten bieten zwar eine sehr kostengünstige individuelle Lösung, sind aber in Bezug auf das Flächen-Zeit-Produkt in der Regel nicht optimal.

Das am häufigsten benutzte Verfahren ist die Modularmultiplikation von Montgomery [53]. Die Idee dieses Algorithmus ist ähnlich wie bei der verschränkten Modularmultiplikation: Es werden Teilprodukte einzelner Bits eines Operanden mit dem zweiten Operand gebildet und zur Zwischensumme aufaddiert. Im Gegensatz zu Interleaved Modularmultiplikation wird der zweite Operand vom niederwertigsten zum höchstwertigsten Bit bearbeitet. Eine genaue Beschreibung dieses Verfahrens wird in Abschnitt 4.1 durchgeführt. Viele wissenschaftliche Arbeiten beschäftigen sich mit der Modularmultiplikation von Montgomery. So wurden in [80], [81] und [82] skalierbare Algorithmen für die Modularmultiplikation entwickelt, welche hauptsächlich der Flä-

chenminimierung dienen. Dies wiederum geschieht auf Kosten der Zeitkomplexität, so dass die AT-Komplexität dabei nicht verbessert wird. Einige Arbeiten befassen sich mit den Implementierungen auf FPGAs [13], [14], [47], [79], [80], und [82]. Solche Algorithmen sind zwar kostengünstig, aber nicht optimal im Bezug auf die AT-Komplexität, denn die einzelnen Zellen der FPGAs, die jeweils nur eine einfache logische Funktion ausführen, besitzen eine hohe Flächenkomplexität. Eine weitere Vorgehensweise ist die Implementierung systolischer Felder [10], [31] und [89]. Eine solche Implementierung bedeutet, dass die Schaltung in eine Menge der gleichen Funktionen, die als Prozessorelemente bezeichnet werden, unterteilt wird. Eine solche Anordnung hat vor allem zwei Vorteile: An erster Stelle ist zu nennen, dass nicht die ganze Schaltung, sondern nur ein einzelnes Prozessorelement optimiert wird, da die anderen Prozessorelemente nur dessen Kopien sind. Der zweite Vorteil ist die Möglichkeit, eine solche Schaltung in eine regelmäßige Hardwarestruktur (z. B. FPGA) einzubetten. Der Nachteil bei all diesen Algorithmen ist deren Komplexität, sowohl in Bezug auf den Flächen-, als auch auf den Zeitaufwand, da es bei solchen harten Strukturen oft nicht möglich ist Optimalität zu gewährleisten. Darüber hinaus gibt es Arbeiten, bei denen die Zeitkomplexität durch die Nutzung der nicht redundanten Zahlensysteme verbessert wird. Dabei werden die einzelnen Ziffern als {-1, 0, 1} dargestellt [37]. Die Operationen (Additionen) sind bei solchen Lösungen sehr schnell im Vergleich zu nichtredundanten Additionen, allerdings deutlich langsamer und flächenaufwändiger als Carry-Save-Additionen, die in Abschnitt 3.2.1 beschrieben werden. Ein weiterer Modularmultiplizierer mit redundanter Addition wurde in [88] veröffentlicht, wobei der dazugehörige Addierer etwa die doppelte Flächenkomplexität eines Carry-Save-Addierers besitzt. Viele Algorithmen benutzen Carry-Save-Additionen [28], [40], [47], [48], wobei eine der besten Implementierungen die von [40] ist. Die Autoren beschreiben eine Lösung, in der nur zwei nacheinander folgende Carry-Save-Additionen in einer Schleifeniteration benutzt werden. Eine andere Version [48] bietet einen vollständig redundanten Modularmultiplizierer, der eine Weiterentwicklung von [40] und [17] darstellt. Der Beitrag [17] ist ein Teil dieser Doktorarbeit und wird detailliert in Abschnitt 4.1. erläutert. Allerdings wird die vollständige Redundanz durch eine Verdopplung des benötigten Flächen- und Zeitaufwandes erreicht und ist somit nicht AT-optimal.

Ansonsten gibt es Arbeiten, in denen mit höheren Basen gearbeitet wird [13], [36], [79] und [81]. Die höheren Basen verbessern zwar das Zeitverhalten der Algorithmen (die Anzahl der Schleifendurchläufe wird reduziert), dabei steigt die Flächenkomplexi-

tät jedoch entsprechend schnell an. Insofern verbessert sich die AT-Komplexität nicht. Es gibt außerdem noch zeitoptimale Algorithmen wie [87], die mit der Zeitkomplexität von O(log n) arbeiten. Die Flächenkomplexität ist dagegen sehr schlecht, was dazu führt, dass die AT-Komplexität deutlich niedriger als bei den AT-optimalen Lösungen ist.

Darüber hinaus existieren eine Reihe anderer Lösungen, die nicht AT-optimal sind: Bei [16] handelt es sich um den Vorgänger der verschränkten Modularmultiplikation; [26] zeigt eine Modularmultiplikation von 2n-bit Zahlen unter Benutzung der Modularmultiplikation von n-bit Zahlen; bei [7] handelt es sich um einen Algorithmus für modulare Reduktion; [35] beschreibt eine hardwarespezifische Lösung unter Verwendung von DSPs (Digital Signal Processor); [46] ist eine softwarebasierte Optimierung; in [1] werden unterschiedliche Skalierungen der Montgomery Modularmultiplikation miteinander verglichen; [49] beschreibt einen Algorithmus, der nur für bestimmte Moduli sinnvoll ist und [58] vergleicht einige Algorithmen mit nichtredundanten Zahlen. Es gibt eine Reihe weiterer, unbedeutender Algorithmen, eine komplette Auflistung würde den Rahmen dieser Arbeit jedoch sprengen.

3 Ein einfaches Hardwaremodell

3.1 Mathematische Notationen

In dieser Arbeit werden folgende mathematische Notationen benutzt: Großbuchstaben bezeichnen Zahlen, die als binäre Vektoren betrachtet werden. So werden vor allem die Operanden, der Modulus und die Potenz bezeichnet. Kleinbuchstaben werden vor allem bei der Darstellung von Faktoren, Indizes oder der Bitlänge einer Zahl verwendet. Mit x_i wird das i-te Bit von X bezeichnet. „+“, „–“ und „*“ werden jeweils für die klassischen Addition, Subtraktion bzw. Multiplikation angewandt. $\oplus$ stellt eine XOR-Verknüpfung dar. Weitere logische Schaltungen werden entsprechend ihrer englischen Bezeichnung benutzt. Bei Z=RCA(X, Y) und Z=CLA(X, Y) handelt es sich um die Ripple-Carry- bzw. Carry-Look-Ahead-Additionen. (S, C)=CSA(X, Y, Z) beinhaltet eine Carry-Save-Addition mit drei Eingabezahlen X, Y und Z und zwei Ausgabezahlen S und C, wobei S die Konkatenation der Summen und C die Konkatenation der Übertragsbits beinhalten. Die ausführliche Erklärung der Carry-Save-Addition wird in Abschnitt 3.2.1 gegeben. Mit Lookup(…) wird die Lookup Table bezeichnet. Die Funktionen „div“ und „mod“ werden jeweils für die ganzzahlige Division bzw. für die Berechnung des Divisionsrestes verwendet. Dabei ist das Ergebnis der Operation A mod M eine ganze Zahl und es gilt: $0 \le A \bmod M < M$. Das Zeichen „≅“ beinhaltet Kongruenz. Zwei Zahlen A und B heißen kongruent modulo M zueinander, wenn gilt $A \bmod M = B \bmod M$, so bedeutet $X \cong Y \bmod M$, dass X gleich Y mod M plus ein Vielfaches des Modulus M ist.

3.2 Addition

3.2.1 Carry-Save-Addition und redundante Zahlendarstellung

Die zentrale Operation der meisten Algorithmen für die Modularmultiplikation ist die binäre Addition. Es gibt mehrere Additionsverfahren: Ripple-Carry-Addition, Carry-Select-Addition, Carry-Look-Ahead-Addition und andere [24], [43], [44], [63]. Der Nachteil all dieser Methoden ist das Weiterleiten des Übertragsbits, das die Latenzzeit der Addition in die Abhängigkeit zur Operandengröße stellt. Dies hat zwar nur

eine geringe Auswirkung für kleine Operanden (z.B. der Länge 64 Bit), aber eine gravierende Wirkung auf die Addition bei Kryptoverfahren, bei denen die Operandengröße zwischen 1024 und 8192 Bit liegen kann. Die Zeitkomplexität einer solchen Addition bestimmt hierbei die Komplexität des Gesamtverfahrens.

Die Carry-Save-Addition bietet die Möglichkeit, eine Addition auszuführen, ohne den Übertrag weiterleiten zu müssen. Sie ist eine einfache parallele Schaltung von n Volladdierern, die nicht miteinander verbunden sind. Die Funktion dieser Volladdierer ist es, drei n-bit-lange ganze Zahlen X, Y und Z miteinander zu addieren. Das Ergebnis sind zwei ganze n+1-bit lange Zahlen C und S mit der Eigenschaft:

$$C + S = X + Y + Z \tag{3.1}$$

Das i-te Bit der Summe s_i und der i+1-te Bit des Übertrages c_{i+1} werden durch die folgenden booleschen Gleichungen berechnet:

$$\begin{gathered} s_i = x_i \oplus y_i \oplus z_i \\ c_{i+1} = x_i y_i \vee x_i z_i \vee y_i z_i \\ c_0 = 0 \\ s_n = 0 \end{gathered} \tag{3.2}$$

Im Folgenden wird für eine Carry-Save-Addition die Notation

$$(S, C) = \mathrm{CSA}(X, Y, Z) \tag{3.3}$$

verwendet.

Das Zahlenpaar (S, C) bildet eine redundante Darstellung der Zahl S+C, die so auf unterschiedliche Art und Weise dargestellt werden kann. Die Addition der Zahlen in einer solchen redundanten Zahlendarstellung ist einerseits sehr schnell, da bei der Addition kein Weiterleiten des Übertrages nötig ist, bringt anderseits jedoch einen gravierenden Nachteil mit sich: Der Vergleich zwischen zwei Zahlen und ebenso der Vergleich mit Null ist in Bezug auf die Flächen- und Zeitkomplexität sehr aufwändig. Jedoch ist der Vergleich ein wichtiger Bestandteil fast aller Algorithmen der Modularmultiplikation [12], [16], [42].

3.3 Komplexitätsmodell

3.3.1 Definition der Komplexität

Das Ziel dieser Arbeit liegt in der Entwicklung neuer, hardwarenaher Algorithmen für Computerarithmetik. Um zu zeigen, dass die neuen Algorithmen besser als die existierenden sind, werden diese anhand gewisser Maße miteinander verglichen. Die beiden wichtigsten Maße sind die Zeit- und Flächenkomplexität. Die Zeit ist nicht zu vernachlässigen, da die meisten Algorithmen der Computerarithmetik eine sehr hohe Datenrate benötigen und gleichzeitig rechenintensiv sind. Somit ist die Zeit eines der wichtigsten Kriterien für die Güte eines Algorithmus. Die Fläche der Hardware ist ein Maß für die Anzahl der benötigten Gatter und somit die Anzahl der Transistoren auf dem Chip. Die Kosten eines Chips steigen mit der benötigten Fläche. Somit ist die Flächenkomplexität eines der wichtigsten Merkmale bei der Entwicklung von hardwarenahen Algorithmen. Darüber hinaus wird eine Kombination aus diesen beiden Maßen, die AT-Komplexität (Flächen-Zeitkomplexität), definiert. Diese bildet das Produkt aus Flächen- und Zeitkomplexität. Es werden folgende Einheiten definiert: Flächeneinheit, Zeiteinheit und Flächen-Zeit-Produkt-Einheit.

Manchmal werden bei der Analyse der Algorithmen die konstanten Faktoren vernachlässigt. In den praktischen Anwendungen spielen die Konstanten eine sehr wichtige Rolle, da Verfahren mit geringerer asymptotischer Komplexität durch größere Konstanten deutlich höhere Kosten als diejenigen, deren asymptotische Komplexität hoch und deren Konstanten klein sind, verursachen. Als Beispiel dafür kann die ganzzahlige Multiplikation dienen – der Wallace Tree [86] mit einer asymptotischen Komplexität von $O(n^2 \log n)$ ist für praktische Anwendungen besser geeignet als die Schönhage-Strassen Multiplikation [72], deren Komplexität $O(n \log^2 n \log\log n)$ beträgt. Aus diesem Grund werden in dieser Arbeit sowohl die asymptotische Komplexität als auch Konstanten berücksichtigt.

Da die Bewertung sowohl der Fläche als auch des Zeitverhaltens einzelner Hardwareelemente sehr stark technologieabhängig ist, wird ein abstraktes Modell benutzt, das näherungsweise einer realen Hardwareimplementierung in CMOS entspricht.

In den nächsten Abschnitten werden die in dieser Arbeit benutzten Hardwareelemente dargestellt und deren Komplexität definiert.

3.3.2 Volladdierer

Zu den wichtigen Hardwareelementen zählt der Volladdierer. Ein Volladdierer verfügt über drei Eingabebits (z. B. x, y, z), Schaltlogik und zwei Ausgabebits (z. B. s und c), wobei die Ausgabe die Summe der drei Eingabebits produziert. Die Funktionsweise eines Volladdierers kann folgender Wahrheitstabelle entnommen werden:

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Tabelle 3-1: Wahrheitstabelle eines Volladdierers

Ein Volladdierer wird durch das folgende Symbol dargestellt:

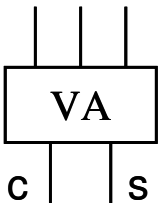

Abbildung 3.3.1: Schaltbild eines Volladdierers

Die Zeitkomplexität des Schaltens eines Volladdierers wird als eine Zeiteinheit definiert.

Die Flächenkomplexität wird als acht Flächeneinheiten definiert. Die Begründung für diese und weitere Flächenkomplexitätsangaben erfolgt in Abschnitt 3.4. Demgemäß liegt die Flächen-Zeitkomplexität eines Volladdierers bei acht Flächen-Zeit-Produkt-Einheiten.

3.3.3 Ripple-Carry-Addierer

Ein *n*-bit Ripple-Carry-Addierer ist ein Schaltkreis, der zwei *n*-bit Zahlen zu einer *n*+1-bit Zahl addiert. Die Addition verläuft von der niederwertigsten zur höchstwertigsten Stelle. Ein *n*-bit Ripple-Carry-Addierer besteht aus *n* Volladdierern, die so miteinander verknüpft sind, dass der Übertrag, der bei der Addition in jedem Volladdierer entsteht, als Eingabe für den nächststehenden Volladdierer dient. Das folgende Bild erläutert die Funktionsweise eines Ripple-Carry-Addierers.

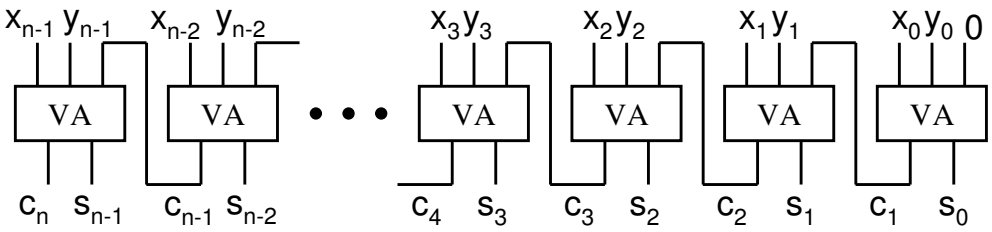

Abbildung 3.3.2: Ripple-Carry-Addierer

Die folgende Abbildung zeigt das Schaltbild eines Ripple-Carry-Addierers:

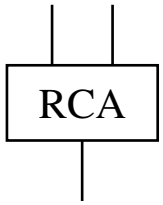

Abbildung 3.3.3: Schaltbild eines Ripple-Carry-Addierers

Die Zeitkomplexität eines Ripple-Carry-Addierers liegt bei *n* Schaltvorgängen eines Volladdierers. Das gilt unter der Voraussetzung, dass sowohl die Berechnung des Übertragsbits als auch die Berechnung des Summenbits die gleiche Zeit benötigen. Somit entspricht die Zeitkomplexität eines Ripple-Carry-Addierers *n* Zeiteinheiten. Die Flächenkomplexität liegt demzufolge bei $8n$ Flächeneinheiten. Die Flächen-Zeitkomplexität eines Ripple-Carry-Addierers beträgt folglich $8n^2$ Flächen-Zeit-Produkt-Einheiten.

3.3.4 Carry-Save-Addierer

Ein *n*-bit Carry-Save-Addierer ist ein Schaltkreis, der drei *n*-bit Zahlen zu zwei *n*+1-bit Zahlen addiert. Eine genauere Beschreibung des Carry-Save-Addierers ist in Abschnitt 3.2.1 zu finden.

Die folgenden Abbildungen zeigen den Aufbau eines Carry-Save-Addierers und dessen Schaltbild:

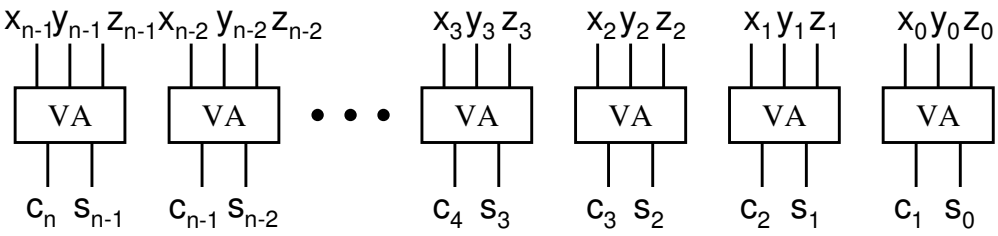

Abbildung 3.3.4: Carry-Save-Addierer

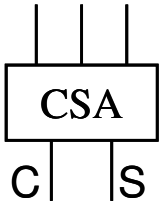

Abbildung 3.3.5: Schaltbild eines Carry-Save-Addierers

Da alle *n* Volladdierer bei einem Carry-Save-Addierer parallel zueinander arbeiten, entspricht die Zeitkomplexität eines Carry-Save-Addierers der Zeitkomplexität eines Volladdierers und beträgt somit eine Zeiteinheit. Die Flächenkomplexität eines *n*-bit Carry-Save-Addierers liegt bei 8*n* Flächeneinheiten. Demzufolge beträgt die Flächen-Zeitkomplexität eines Carry-Save-Addierers 8*n* Flächen-Zeit-Produkt-Einheiten.

3.3.5 Carry-Look-Ahead-Addierer

Der Carry-Look-Ahead-Addierer [9], [25], [43], [63] ist der schnellste bekannte nichtredundante Addierer. Die Hauptidee des Carry-Look-Ahead-Addierers ist das gleichzeitige Generieren aller Überträge. Die so berechneten Werte werden zuerst durch die Struktur eines binären Baumes zur Wurzel und dann an die Blätter zurückgeleitet. Der Carry-Look-Ahead-Addierer hat eine Flächenkomplexität von O(*n*) und eine Zeitkomplexität von O(log*n*) [24], [25]. Ein Carry-Look-Ahead-Addierer ist in Form eines Baumes aufgebaut. Es gibt mehrere Versionen des Carry-Look-Ahead-Addierers [25]. Diese unterscheiden sich durch die Anzahl der Einheiten einer Basiszelle. In dieser Arbeit wird die klassische Version des Carry-Look-Ahead-Addierers verwendet. Die Analyse der Standard Cell Libraries, die in Abschnitt 3.4 durchgeführt wird, ergibt, dass ein Carry-Look-Ahead-Addierer etwa die dreifache Flächenkomplexität eines entsprechenden Carry-Save-Addierers besitzt. Dieser Vergleich wurde ohne die Betrachtung der Fläche der Leitungen durchgeführt, auf die in dieser Arbeit verzichtet wurde. Da die Flächenkomplexität eines Carry-Look-Ahead-Addierers bei 8*n* Flächeneinheiten liegt, wird der Flächenaufwand eines Carry-Look-Ahead-Addierers auf 24*n* Flächeneinheiten geschätzt.

An dieser Stelle wird die Funktionsweise des Carry-Look-Ahead-Addierers erklärt: Um alle Summenbits bei einem Ripple-Carry-Addierer zu bestimmen, müssen im schlechtesten Fall *n* Übertragbits nacheinander berechnet werden, was zu einer höheren Zeitkomplexität führt. Eine der Möglichkeiten, eine solche Addition zu beschleunigen

ist die parallele Berechnung der Überträge. Für diesen Zweck werden zuerst parallel zueinander „carry-propagate-“ und „carry-generate-“ Terme berechnet:

$$p_i=a_i \oplus b_i$$
$$g_i=a_i b_i \tag{3.4}$$

Diese können für weitere „carry-propagate-“ und „carry-generate-“ Terme als Eingabe dienen:

$$p_{i,k}=p_{i,j}p_{j-1,k}$$
$$g_{i,k}=g_{i,j}+p_{i,j}g_{j-1,k} \tag{3.5}$$

Dabei gilt:

$$i \geq j > k,\ g_{i,i}=g_i \text{ und } p_{i,i}=p_i \tag{3.6}$$

Daraus werden die Überträge berechnet:

$$c_j=g_{j-1,k}+p_{j-1,k}c_k \tag{3.7}$$

Die Summenbits werden dann als

$$s_i=a_i \oplus b_i \oplus c_i \tag{3.8}$$

bestimmt.

Ein Carry-Look-Ahead-Addierer wird in der Form eines binären Baumes aufgebaut, dessen Knoten die Schaltungen A oder B sind, wie im folgenden Beispiel für einen 4-bit Carry-Look-Ahead-Addierer gezeigt wird:

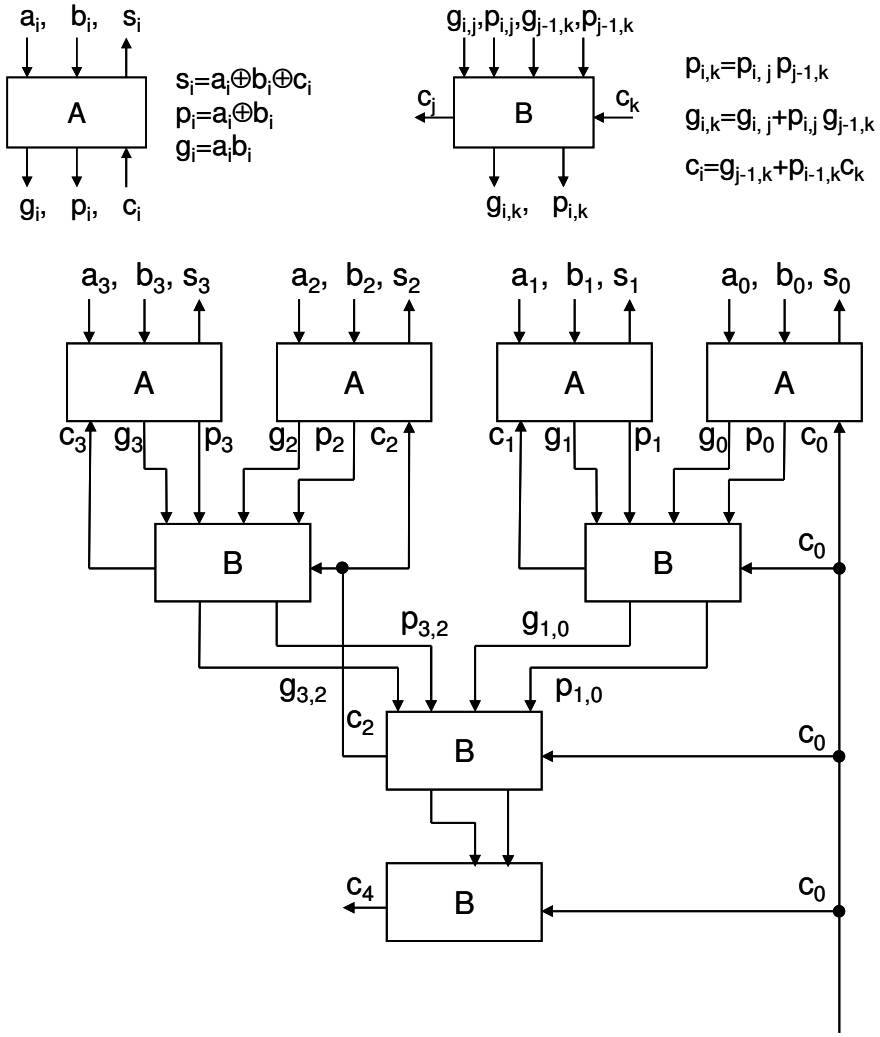

Abbildung 3.3.6: 4-bit Carry-Look-Ahead-Addierer

Der längste Pfad eines Carry-Look-Ahead-Addierers führt durch 2log*n* derartiger Schaltungen. Die Zeitkomplexität einer solchen Schaltung ist vergleichbar mit der Zeitkomplexität eines Volladdierers und entspricht somit einer Zeiteinheit. Demzufolge liegt die Gesamtzeitkomplexität des Carry-Look-Ahead-Addierers bei 2log*n* Zeiteinheiten. Eine genaue Beschreibung eines Carry-Look-Ahead-Addierers ist in [9] und [24] detailliert beschrieben.

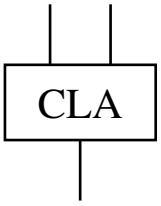

Abbildung 3.3.7: Schaltbild eines Carry-Look-Ahead-Addierers

3.3.6 Kombiaddierer

Im letzten Abschnitt wurde der Carry-Look-Ahead-Addierer bewertet. Der Vorteil dieses Addierers ist seine logarithmische Zeitkomplexität. Der Nachteil dabei ist die Tatsache, dass die Flächenkomplexität eines Carry-Look-Ahead-Addierers dreimal so groß wie die Flächenkomplexität eines Carry-Save-Addierers ist. Um diesen Nachteil zu relativieren, wird ein Addierer verwendet, der eine Kombination aus Carry-Look-Ahead und Ripple-Carry-Addierer ist. Um zwei *n*-bit Zahlen zu addieren, wird eine Addition von *k*-bit Blöcken (k≥2) mehrfach nacheinander durchgeführt. Jede *k*-bit Addition wird mit dem *k*-bit Carry-Look-Ahead-Addierer vom niederwertigsten Bit der Operanden ausgehend gemacht. Bei jeder weiteren Addition wird der Übertrag der nächsten Addition aufaddiert. Folgende Abbildung zeigt die Funktionsweise eines Kombiaddierers:

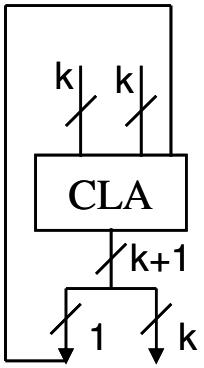

Abbildung 3.3.8: Funktionsweise eines Kombiaddierers

Um zwei *n*-bit Zahlen zu addieren, benötigt ein solcher Kombiaddierer einen *k*-bit Carry-Look-Ahead-Addierer. Die Flächenkomplexität eines solchen Kombiaddierers liegt damit bei 24*k* Flächeneinheiten.

Um eine *n*-bit Addition durchzuführen, sind $\lceil n/k \rceil$ *k*-bit Additionen erforderlich. Ist *k* dabei ein Teiler von *n*, so beträgt die Zeitkomplexität eines solchen Addierers

$$2\frac{n}{k}\cdot\log k \tag{3.9}$$

Zeiteinheiten.

Es wird in dieser Arbeit ein Kombiaddierer verwendet, dessen Flächenkomplexität niedriger als bei einem *n*-bit Carry-Save-Addierer ist. Für diesen Zweck wird *k* auf *n*/4 gesetzt, damit ist die Flächenkomplexität eines Kombiaddierers gleich

$$24k = 24\frac{n}{4} = 6n < 8n \tag{3.10}$$

Die Zeitkomplexität eines solchen Kombiaddierers beträgt

$$2\log k \cdot \frac{n}{k} = 2\log \frac{n}{4} \cdot \frac{n}{n/4} = 8(\log n - \log 4) = 8\log n - 8 \cdot 2 = 8\log n - 16 \qquad (3.11)$$

Zeiteinheiten.

3.3.7 Register

In allen Schaltungen, die Information zwischenspeichern, werden Register benutzt. Ein 1-bit Register besteht aus einem Flip-Flop, ein *n*-bit Register aus *n* parallel geschalteten Flip-Flops.

Die folgende Abbildung zeigt das Schaltbild eines *n*-bit Registers:

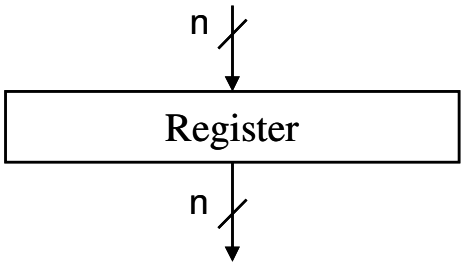

Abbildung 3.3.9: *n*-bit Register

Ein 1-bit Register benötigt nur die Halbe Fläche eines Volladdierers (vgl. 3.4), daher wird die Flächenkomplexität eines 1-bit Registers als 4 Flächeneinheiten definiert. Folglich ist die Flächenkomplexität eines *n*-bit Registers 4*n* Flächeneinheiten. Da die Zeitkomplexität des Schaltens eines Registers bei gegenwärtigen Hardwaretechnologien einerseits niedriger als die Zeitkomplexität eines Volladdierers ist und andererseits in allen in dieser Arbeit betrachteten Algorithmen die gleiche Anzahl der Registerstufen benutzt wird, wird in dem hier verwendeten Komplexitätsmodell auf die Berücksichtigung der Zeit des Registerschaltvorganges (Setup- und Hold-Zeit) verzichtet.

3.3.8 Lookup Table

Im letzten Abschnitt 3.3.7 wurde gezeigt, wie Informationen mit Hilfe von Registern zwischengespeichert werden können. Mehrere Register können gleichzeitig in einem Takt ausgelesen und beschrieben werden. Jedoch muss ihre Verwendung mit relativ hohem Flächenaufwand bezahlt werden. Eine andere Möglichkeit Informationen zwischenzuspeichern bietet die Lookup Table. Eine $k \cdot n$-bit Lookup Table enthält k n-bit-lange Worte, die jeweils entweder aus der Lookup Table gelesen oder in die Lookup

Table geschrieben werden können. Allerdings kann pro Takt nur ein Zugriff erfolgen. Obwohl die Lookup Table im Vergleich zu Registern weniger Funktionalität bietet, kann sie den gewichtigen Vorteil einer niedrigen Flächenkomplexität für sich verbuchen. Eine $k \cdot n$-bit Lookup Table benötigt etwa ein Viertel der Fläche der entsprechenden k n-bit Register. Somit wird die Flächenkomplexität, die für das Speichern eines Bits in der Lookup Table benötigt wird, als eine Flächeneinheit definiert (siehe Abschnitt 3.4). Das Lesen und Schreiben geschieht durch die Adressierung innerhalb der Lookup Table, welche deutlich mehr Zeit als das einfache Lesen und Schreiben in einem Register benötigt. Demgemäß wird die Zeitkomplexität eines Zugriffes auf die Lookup Table als eine Zeiteinheit definiert.

Die folgende Abbildung zeigt das Schaltbild einer $k \cdot n$-bit Lookup Table:

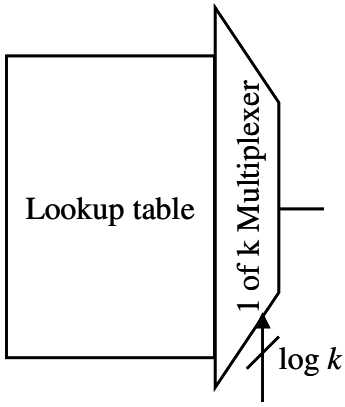

Abbildung 3.3.10: Schaltbild der Lookup Table

3.3.9 Bauelemente für die Datensteuerung

In diesem Abschnitt werden einige Schaltungen eingefügt, deren Aufgabe die Steuerung des Datenflusses ist und deren Schaltbilder von der in der Literatur üblichen Standarddarstellung abweichen. Als erstes wird der Datenwegschalter betrachtet, dessen Funktion darin besteht, einen parallelen Datenstrom (Bitvektor) weiterzuleiten, falls das Steuerungsbit eine Eins ist, oder einen Bitvektor, bestehend aus Nullen, zurückzuliefern, falls das Steuerungsbit eine Null ist. Ein Datenwegschalter besteht aus parallel geschalteten UND-Gattern.

Die folgende Abbildung zeigt das Schaltbild eines n-bit Datenwegschalters:

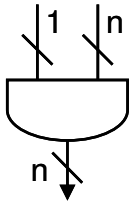

Abbildung 3.3.11: Schaltbild des Datenwegschalters

Eine weitere Funktion, die in dieser Arbeit verwendet wird, ist die Konkatenation. Diese wird durch das folgende Schaltbild dargestellt:

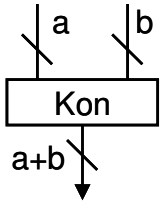

Abbildung 3.3.12: Schaltbild für die Konkatenation

Ebenso wird die Dekonkatenation (Aufspaltung) benutzt. Das dazugehörige Schaltbild ist:

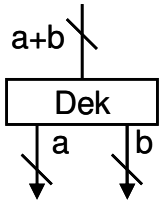

Abbildung 3.3.13: Schaltbild für die Dekonkatenation

Sowohl der Flächen- als auch der Zeitaufwand eines Datenwegschalters ist relativ gering im Vergleich zum Flächenaufwand eines Carry-Save-Addierers. Darüber hinaus benötigen alle Algorithmen der Computerarithmetik zusätzliche Steuerungselemente. Aus diesem Grund wird die Komplexität dieser Elemente nicht berücksichtigt.

Konkatenation und Dekonkatenation werden in Hardware über geeignete Leitungsführung implementiert. Die Komplexität der Leitungen wird in dieser Arbeit vernachlässigt.

3.3.10 Zusammenfassung

In diesem Kapitel wurden die Begriffe der Flächen- und der Zeitkomplexität sowie deren Produkt, die Flächen-Zeitkomplexität, eingeführt. Außerdem wurden die für diese Arbeit wichtigen Hardwareelemente präsentiert und ihre Komplexität definiert. Die folgende Tabelle bietet einen kurzen Überblick:

Bezeichnung:	Flächenkomplexität/ Flächeneinheit	Zeitkomplexität/ Zeiteinheit
Volladdierer	8	1
n-bit Ripple-Carry-Addierer	8*n*	*n*
n-bit Carry-Save-Addierer	8*n*	1
n-bit Carry-Look-Ahead-Addierer	24*n*	2log*n*
n-bit Kombiaddierer mit $n \geq 8$	6*n*	8log*n*–16
n-bit Register	4*n*	0
Lookup Table (*n*-bit Wort)	*n*	1

Tabelle 3-2: Komplexität einzelner Hardwareelemente

3.4 Flächenkomplexitätsanalyse

In diesem Abschnitt wird die Erklärung für die zuvor festgelegten Werte der Flächenkomplexität geliefert. Diese sind das Ergebnis einer Untersuchung mehrerer Technologien, mit denen Hardware implementiert wird. Da die Algorithmen, die in dieser Arbeit präsentiert werden, technologieunabhängig und folglich für keine der Technologien optimiert sind, werden sowohl Standard Cell Libraries für den Application Specific Integrated Circuit (ASIC) Entwurf als auch Full Custom Libraries und FPGAs analysiert.

Als Grundlagen dienten die „standard cell“ Bibliotheken von NEC [55], Mentor Graphics [51], Austria Micro System [4], Toshiba [84], MSU [52], Systola [85] und FPGAs vom derzeitigen marktführenden FPGA-Hersteller, Xilinx [92].

Dabei wurden folgende Bausteine betrachtet: Volladdierer, 1-bit Register und 1-bit Speicher für die Lookup Table. Ein Register besteht aus einem D-Flip-Flop. Die meisten in dieser Arbeit verwendeten Register werden für die Zwischenspeicherung von Werten eingesetzt. Für die hier benötigten Schaltungen reichen dynamische D-Flip-Flops aus, welche ihren Wert nur für einen Takt speichern, da schon im nächsten Takt die zwischengespeicherten Werte ausgelesen und überschrieben werden. Die meisten Zellenbibliotheken bieten nur statische Flip-Flops an. Diese bestehen aus mehr als der doppelten Anzahl von Transistoren, die wiederum nicht so dicht wie die der dynamischen D-Flip-Flops liegen können. Aus diesem Grund wird die Flächenkomplexität der in dieser Arbeit benutzten 1-bit Register mit der Hälfte der Flächenkomplexität der sta-

tischen Flip-Flops aus der Standard Cell Libraries abgeschätzt. Für die Abschätzung der einzelnen 1-bit Zellen der Lookup Table werden S-RAM-Zellen betrachtet. Die Auswahl von S-RAM anstatt des noch flächeneffizienteren D-RAMs basiert auf zwei Überlegungen: Einerseits sind D-RAM-Zellen deutlich langsamer als S-RAM-Zellen, andererseits brauchen D-RAM-Zellen eine zusätzliche Refresh-Schaltung, die bei solch kleinen Informationsmengen, wie sie in dieser Arbeit benötigt werden, einen höheren Flächenaufwand als die eigentlichen D-RAM-Zellen verbrauchen würden. Somit würde der Einsatz von D-RAM sowohl die Zeit- als auch die Flächenaufwand der Gesamtschaltung vergrößern.

Die folgende Tabelle zeigt die Flächenkomplexität einzelner Schaltkreise bei den verschiedenen Technologien an. Dabei werden für die einzelnen Bibliotheken die Flächeneinheiten gewählt, die in deren Beschreibungen verwendet werden. So werden bei NEC „grids“ als Flächeneinheiten, bei Austria Micro System μm^2, bei MSU „units“, bei Toshiba „area units“, bei Systola Einheiten und bei FPGAs Zellen angewandt. Mentor Graphics AMI bietet dagegen nur die Breitenangaben, da alle Gatter die gleiche Länge aufweisen. Da in dieser Arbeit kein Vergleich der Güte unterschiedlicher Bibliotheken, sondern das Verhältnis des Flächenaufwandes unterschiedlicher Bauelemente innerhalb der jeweiligen Bilbliotheken untereinander angestrebt wird, soll auf die Normierung der Flächenangaben verzichtet werden.

Technologie	Voll-addierer	statische D-Flip-Flops (Register)	Abschätzung für die dynamische D-Flip-Flops	Abschätzung für eine S-Ram-Zelle einer Lookup Table
NEC 0,5 µm Zellen-basierte IC (Fläche)	17 grids	12 grids	6 grids	3 grids
Mentor Graphics AMI 0,5 µm (Breite)	4,58 µm	4,79 µm	2,40 µm	1,02 µm
Austria Micro System 0,35 µm (Fläche)	273 μm^2	273 μm^2	137 μm^2	55 μm^2
MSU Standard Cell Library (Breite)	25,5 units	25,5 units	12,8 units	3,8
Toshiba TC300C series low-power primitive cells (Fläche)	14,43 area-units	13,17 area-units	6,59 area units	2,51 area-units
Xilinx FPGAs (Virtex 1, 2 / Zellen mit je einem Flip-Flop und einer Lookup Table)	1 Zellen	1 Zelle	1 Zelle	1/16 Zelle
Systola (Fläche)	8 Einheiten	8 Einheiten	4 Einheiten	1 Einheit

Tabelle 3-3: Standard Schaltkreise bei unterschiedlichen Technologien

Die Analyse der Tabelle zeigt, dass ein Volladdierer im Durchschnitt doppelt so groß wie ein dynamisches D-Flip-Flop ist. Letzteres ist im „full custom“ Entwurf etwa dreimal so groß wie eine 1-bit Zelle für die Lookup Table. Jede FPGA-Zelle enthält eine Lookup Table, deren Nutzung gegenüber der Implementation eines 1-Bit-Registers das 16-fache Speichervolumen ermöglicht. Somit erweist sich die Abschätzung, dass die Flächenkomplexität eines Volladdierers acht Flächeneinheiten, eines 1-bit Register vier Flächeneinheiten und einer 1-bit Zelle für die Lookup Table eine Flächeneinheit groß ist, als geeigneter Kompromiss, der mit allen hier beschriebenen Technologien verträglich ist.

In Kapitel 3 wurden die in dieser Arbeit am häufigsten verwendeten Hardwareelemente sowie deren Flächen- und Zeitkomplexität definiert. In den folgenden Kapiteln werden die neuen Algorithmen der Computerarithmetik präsentiert und auf Basis der mit diesen Mitteln entwickelten Komplexitätsmaße bewertet.

4 Sequenzielle Algorithmen der Modularmultiplikation

4.1 Montgomery Modularmultiplikation und die Verbesserung durch die Nutzung einer Lookup Table

4.1.1 Algorithmus von Montgomery

Wie in der Einleitung schon erwähnt, ist einer der am häufigsten benutzten Algorithmen für die Modularmultiplikation das Verfahren von Montgomery [51]. Bei der Modularmultiplikation von Montgomery wird, im Gegensatz zu den anderen Algorithmen für Modularmultiplikation, nicht $X \cdot Y \bmod M$, sondern $X \cdot Y \cdot 2^{-n} \bmod M$ berechnet. Dabei ist n die Anzahl der Bits von X, M ist eine ungerade Zahl, so dass $X, Y < M$ und $2^{-n} \bmod M$ die Inverse zu 2^n bezüglich des Modulus M ist. Das heißt, dass die folgende Gleichung gilt:

$$(2^n \bmod M)(2^{-n} \bmod M) \bmod M = 1 \tag{4.1}$$

Da der Modulus M eine ungerade Zahl ist, gilt:

$$\mathrm{ggT}(2^n, M)=1 \tag{4.2}$$

Somit gibt es genau eine multiplikative Inverse zu 2^n bezüglich des Modulus M [45].

$X \cdot Y \cdot 2^{-n} \bmod M$ wird berechnet, indem der Operand X vom niederwertigsten zum höchstwertigsten Bit verarbeitet wird. Die Berechnung erfolgt ähnlich zur Standardmultiplikation: In jedem Schritt wird das entsprechende Bit von X mit dem zweiten Operand Y multipliziert und zum Ergebnis des letzten Schleifendurchlaufs addiert. Der Unterschied liegt in der Tatsache, dass bei der Modularmultiplikation von Montgomery nach jeder Addition überprüft wird, ob das Zwischenergebnis gerade oder ungerade ist. Zu diesem Zweck wird das letzte Bit des Zwischenergebnisses untersucht: Ist dieses gleich Eins, so wird der Modulus M zum Zwischenergebnis addiert. Unabhängig davon wird dieses im nächsten Schritt durch zwei geteilt. Nach der letzten Iteration wird das

Ergebnis mit dem Modulus verglichen; sollte es größer als der Modulus sein, so wird dieser einmal abgezogen.

Bei dieser Methode der Modularmultiplikation muss der Modulus *M* eine ungerade Zahl sein.

Der Vorteil der Modularmultiplikation von Montgomery ist, dass keine aufwändigen Vergleiche mit dem Modulus benötigt werden. Außerdem ist es möglich, ohne höheren Aufwand die langsamen nichtredundanten Additionen durch schnelle Carry-Save-Additionen zu ersetzen (dies wird im Abschnitt 4.1.2 beschrieben).

Der folgende Pseudocode zeigt die Funktionsweise der Modularmultiplikation von Montgomery.

Algorithmus 4.1.1: Modularmultiplikation von Montgomery

```
Eingabe: X, Y<M<2^n, wobei 2^(n-1)<M<2^n und M=2t+1, wobei t∈N
Ausgabe: P = X*Y*2^-n mod M
n: Anzahl der Bits in X,
x_i: Das i-te Bit von X
Methode:
(1)  P = 0;
(2)  for (int i=0; i<n; i++) {
(3)    P = P + x_i*Y;
(4)    if (p_0=1) P = P + M;
(5)    P = P div 2;
(6)  }
(7)  if (P≥M) P = P – M;
```

Satz (Montgomery): Der Algorithmus 4.1.1 berechnet $P = X \cdot Y \cdot 2^{-n} \bmod M$.

Beweis: Für die Korrektheit des Satzes werden zwei Aussagen bewiesen, aus denen die Gesamtaussage unmittelbar folgt. Innerhalb des Beweises werden folgende Notationen benutzt: Das Ergebnis des Schrittes (1) wird als P_0 bezeichnet. In der Schleife werden die Werte in Abhängigkeit von der Schleifendurchlaufzahl indiziert, so ist $P_k^{'}$ das Ergebnis des Schrittes (3), $P_k^{''}$ das Resultat des Schrittes (4) und P_k das Ergebnis des Schrittes (5) des *k*-ten Schleifendurchlaufs. Dabei ist *k* eine beliebige Zahl zwischen 1 und *n*.

Aussage 1: Nach jedem Durchlauf der Schleife liegt P im Intervall zwischen 0 und $2M$.

Beweis durch vollständige Induktion:

Vor der Schleife gilt:

$$P_0 = 0 < 2M \tag{4.3}$$

Annahme: Nach dem k-ten Schleifendurchlauf gilt:

$$P_k < 2M \tag{4.4}$$

Zu zeigen: Nach dem k+1-ten Schleifendurchlauf gilt:

$$P_{k+1} < 2M \tag{4.5}$$

Beweis:

Nach Schritt (3) gilt:

$$P_{k+1}^{'} = P_k + x_k \cdot Y < 2M + M = 3M \tag{4.6}$$

Nach Schritt (4) gilt:

$$P_{k+1}^{''} \le P_{k+1}^{'} + M < 3M + M = 4M \tag{4.7}$$

Und nach Schritt (5) gilt:

$$P_{k+1} = P_{k+1}^{''} / 2 < 4M / 2 = 2M \tag{4.8}$$

Somit gilt die Aussage 1.

Aussage 2: $\forall t \in \mathrm{N}, 0 < t \le n$: Nach dem t-ten Schleifendurchlauf gilt:

$$P_t \cong (\sum_{i=0}^{t-1} x_i \cdot 2^i \cdot Y) / 2^t \bmod M \tag{4.9}$$

Beweis durch vollständige Induktion:

Vor der Schleife gilt:

$$P_0 = 0 \cong 0 \bmod M \tag{4.10}$$

Annahme: Nach dem *k*-ten Schleifendurchlauf gilt:

$$P_k \cong (\sum_{i=0}^{k-1} x_i \cdot 2^i \cdot Y) / 2^k \bmod M \tag{4.11}$$

Zu zeigen: Nach dem *k*+1-ten Schleifendurchlauf gilt:

$$P_{k+1} \cong (\sum_{i=0}^{k} x_i \cdot 2^i \cdot Y) / 2^{k+1} \bmod M \tag{4.12}$$

Beweis:

Nach Schritt (3) gilt:

$$\begin{aligned} P'_{k+1} &= P_k + x_k \cdot Y \cong ((\sum_{i=0}^{k-1} x_i \cdot 2^i \cdot Y) / 2^k \bmod M + x_k \cdot Y) \bmod M \\ &\cong ((\sum_{i=0}^{k-1} x_i \cdot 2^i \cdot Y) / 2^k + x_k \cdot Y) \bmod M \\ &\cong (2(\sum_{i=0}^{k-1} x_i \cdot 2^i \cdot Y) / 2^{k+1} + 2^{k+1} \cdot x_k \cdot Y / 2^{k+1}) \bmod M \\ &\cong (2(\sum_{i=0}^{k-1} x_i \cdot 2^i \cdot Y) / 2^{k+1} + 2(2^k \cdot x_k \cdot Y) / 2^{k+1}) \bmod M \\ &\cong (2\sum_{i=0}^{k} x_i \cdot 2^i \cdot Y / 2^{k+1}) \bmod M \end{aligned} \tag{4.13}$$

In Schritt (4) wird eine Fallunterscheidung durchgeführt: Ist *P* eine gerade Zahl, so wird *P* nicht verändert, andernfalls wird der Modulus *M* addiert. In jedem Fall gilt:

$$P''_{k+1} \cong (P'_{k+1} + M) \bmod M \cong P'_{k+1} \bmod M \cong (2\sum_{i=0}^{k} x_i \cdot 2^i \cdot Y / 2^{k+1}) \bmod M \tag{4.14}$$

und P''_{k+1} ist eine gerade Zahl. Somit gilt nach Schritt (5):

$$\begin{aligned} P_{k+1} &= P''_{k+1} / 2 \cong ((2\sum_{i=0}^{k} x_i \cdot 2^i \cdot Y / 2^{k+1}) / 2) \bmod M \\ &\cong (\sum_{i=0}^{k} x_i \cdot 2^i \cdot Y) / 2^{k+1} \bmod M \end{aligned} \tag{4.15}$$

Folglich gilt die Aussage 2.

Im nächsten Abschnitt wird gezeigt, wie die Modularmultiplikation von Montgomery effizient in Hardware implementiert werden kann.

4.1.2 Redundante Variante der Implementierung des Algorithmus von Montgomery in Hardware.

In dieser Arbeit wird die Implementierung von [40] detailliert beschrieben, da diese Optimierung einerseits eine der besten in der Literatur vorhandenen Lösungen darstellt und anderseits die Plattform für den neuen Algorithmus bietet.

Der Grundgedanke dabei ist einfach: Da eine nichtredundante Addition langsam ist, werden alle nichtredundanten Addierer durch Carry-Save-Addierer ersetzt, wobei das Zwischenergebnis in der redundanten Form als ein Paar (*S*, *C*) dargestellt wird, welches zwischen den Schleifendurchläufen in zwei Registern gespeichert werden muss.

Der folgende Algorithmus zeigt die Funktionsweise der redundanten Modularmultiplikation von Montgomery auf.

Algorithmus 4.1.2: Redundante Modularmultiplikation von Montgomery

```
Eingabe: X,Y,M mit 0 ≤ X,Y < M
Ausgabe: P = (X*Y*2^-n) mod M
n: Anzahl der Bits in X,
x_i: i-tes Bit von X
s_0: LSB von S
Methode:
(1)  S = 0; C = 0;
(2)  for (int i=0; i<n; i++) {
(3)     (S,C) = CSA(S, C, x_i*Y);
(4)     (S,C) = CSA(S, C, s_0*M);
(5)     S = S div 2; C = C div 2;
(6)  }
(7)  P = S + C
(8)  if (P≥M) P = P - M;
```

Im Algorithmus 4.1.1 wurde in jedem Schleifendurchlauf nach der ersten Addition (3) das niederwertigste Bit des Zwischenergebnisses mit Eins verglichen (4). Da aber bei der Carry-Save-Addition das niederwertigste Bit von *C* immer eine Null ist, ist es nicht nötig, die Summe von *S* und *C* zu bilden, sondern es wird nur das niederwertigste Bit von *S* überprüft. Ist s_0=0, so ist die Summe von (*S*+*C*) eine gerade Zahl; ist s_0=1, dann ist die Summe (*S*+*C*) eine ungerade Zahl. Dementsprechend kann im Algorithmus 4.1.1 gleich nach der ersten Addition (3), sofort die zweite Addition (4) ohne Überprüfung ausgeführt werden.

Der Vorteil dieser Lösung liegt in der Tatsache, dass zwei langsame Additionen durch zwei schnellere Additionen ersetzt werden können.

Die folgende Abbildung zeigt die Funktionsweise des Algorithmus 4.1.2

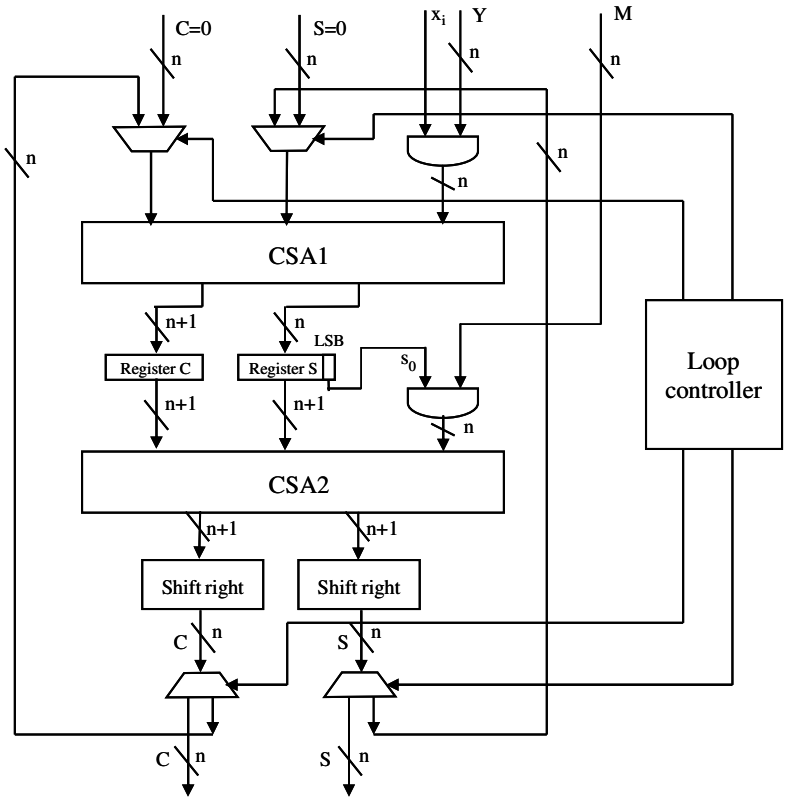

Abbildung 4.1.1: Modularmultiplikation von Montgomery mit Carry-Save-Addierern

Dabei stehen x_i für das i-te Bit von X und s_0 bzw. LSB(S) für das niederwertigste Bit von S.

In diesem Abschnitt wurde die Implementierung der Modularmultiplikation von Montgomery aus [40] vorgestellt. Diese benutzt Carry-Save-Addierer anstelle von nichtredundanten Addierern, wodurch die Modularmultiplikation signifikant beschleunigt werden kann. In jedem Schleifendurchlauf sind hierbei zwei Additionen erforderlich. Im nächsten Abschnitt wird eine Möglichkeit gezeigt, wie die Anzahl der Additionen in jeder Schleifeniteration sogar auf nur eine Addition reduziert werden kann.

4.1.3 Optimierung des Algorithmus von Montgomery unter Benutzung vorberechneter Werte

Im letzten Abschnitt wurde eine nichtredundante Version der Modularmultiplikation von Montgomery vorgestellt. Der Nachteil dabei ist, dass die Schaltung nach wie vor immer noch zwei Addierer benötigt.

Wie aus dem Algorithmus 4.1.1 und aus der Abbildung 3.3.1 entnommen werden kann, werden in jedem Schleifendurchlauf zwei Zahlen zum Zwischenergebnis (S, C)

addiert – Schritte (3) und (4). Diese Zahlen werden jeweils durch eine Bit-Vektor-Multiplikation gebildet. Die Ergebnisse dieser Multiplikationen können jeweils nur zwei Werte annehmen: $x_i \cdot Y$ kann entweder 0 oder Y sein (0 wenn x_i=0 ist und Y wenn x_i=1 ist) und $s_0 \cdot M$ nimmt den Wert von 0 an, wenn s_0=0 ist und M wenn s_0=1 ist. Werden beide Additionen zusammen betrachtet, so ergeben sich folgende vier Fälle:

$$x_i \cdot Y + s_0 \cdot M = \begin{cases} 0, & x_i = 0, s_0 = 0 \\ Y, & x_i = 1, s_0 = 0 \\ M, & x_i = 0, s_0 = 1 \\ Y + M, & x_i = 1, s_0 = 1 \end{cases} \tag{4.16}$$

Das heißt, dass

(i) wenn die Summe der alten Werte von S und C eine gerade Zahl ist und wenn das aktuelle Bit x_i von X eine Null ist, dann müssen S und C nicht erhöht werden. Das bedeutet, dass eine Null dazuaddiert werden muss. Danach kann die Reduktion von S und C durch eine Division durch Zwei erfolgen.

(ii) wenn die Summe der alten Werte von S und C eine ungerade Zahl ist und wenn das aktuelle Bit x_i von X eine Null ist, dann muss M addiert werden, um aus dem Zwischenergebnis eine gerade Zahl zu bilden. Nachfolgend wird die Reduktion durch eine Division durch Zwei durchgeführt.

(iii) wenn die Summe der alten Werte von S und C eine gerade Zahl ist, das aktuelle Bit x_i von X eine Eins ist und das Produkt von $x_i \cdot Y$ eine gerade Zahl ist, dann ist es nicht nötig M dazu zu addieren, um aus dem Zwischenergebnis eine gerade Zahl zu bilden. Stattdessen muss Y dazuaddiert und die Reduktion von S und C durch eine Division durch Zwei erfolgen. Dasselbe passiert, wenn die Summe von S und C ungerade, das aktuelle Bit x_i von X eine Eins und Y eine ungerade Zahl ist. In diesem Fall ist $S+C+Y$ ebenso eine gerade Zahl.

(iv) wenn die Summe der alten Werte von S und C eine ungerade Zahl ist, das aktuelle Bit x_i von X eine Eins ist und wenn das Produkt $x_i \cdot Y$ eine gerade Zahl ist, dann müssen sowohl Y als auch M hinzuaddiert werden, damit das Zwischenergebnis gerade wird. Somit muss die Summe $Y+M$ dazuaddiert werden, danach kann die Reduktion durch eine Division durch Zwei erfolgen. Dasselbe ist notwendig, wenn $S+C$ gerade, das aktuelle Bit x_i von X eine Eins und Y ungerade sind. In diesem Fall ist $S+C+Y+M$ ebenfalls gerade.

Y und *M* sind vor jeder Modularmultiplikation bekannt und ändern sich während der Schleife nicht. Somit ist es sinnvoll, *Y*+*M* vor der eigentlichen Schleife einmal zu berechnen und alle vier Werte 0, *Y*, *M* und *Y*+*M* zu speichern, um sie bei Bedarf abzurufen. Hierfür eignet sich eine Lookup Table aus RAM-Zellen, wie in Abschnitt 3.3.8 beschrieben, sehr gut. Während der Schleife muss jeweils genau ein Wert aus der Lookup Table gelesen und zu S und C addiert werden.

Die Funktionsweise der oben beschriebenen Version der Modularmultiplikation von Montgomery wird durch den folgenden Pseudocode beschrieben:

Algorithmus 4.1.3: Schnelle Modularmultiplikation von Montgomery

```
Eingabe: X,Y,M mit 0 ≤ X,Y < M
Ausgabe: P = (X*Y*2^-n) mod M
n: Anzahl der Bits in X,
x_i: i-tes Bit von X
s_0: LSB von S
R: Vorberechneter Wert von Y+M
Methode:
(1)  S = 0; C = 0;
(2)  R = Y + M;
(3)  for (int i=0; i<n; i++) {
(4)     if (s_0==c_0 and x_0==0) I = 0;
(5)     if (s_0≠c_0 and x_0==0) I = M;
(6)     if ((s_0 ⊕ c_0 ⊕ y_0)==0 and x_0==1) I = Y;
(7)     if (s_0 ⊕ c_0 ⊕ y_0)==1 and x_0=1) I = R;
(8)     (S,C) = CSA(S, C, I);
(9)     S = S div 2; C = C div 2;}
(10) P = S + C;
(11) if (P≥M) P = P - M;
```

Dabei ist Operation ⊕ die Funktion für das „Exklusive ODER“ (XOR). Die Vorteile des Algorithmus 4.1.3 im Vergleich zum Algorithmus 4.1.2 liegen in der günstigen Implementierung der „for“-Schleife.

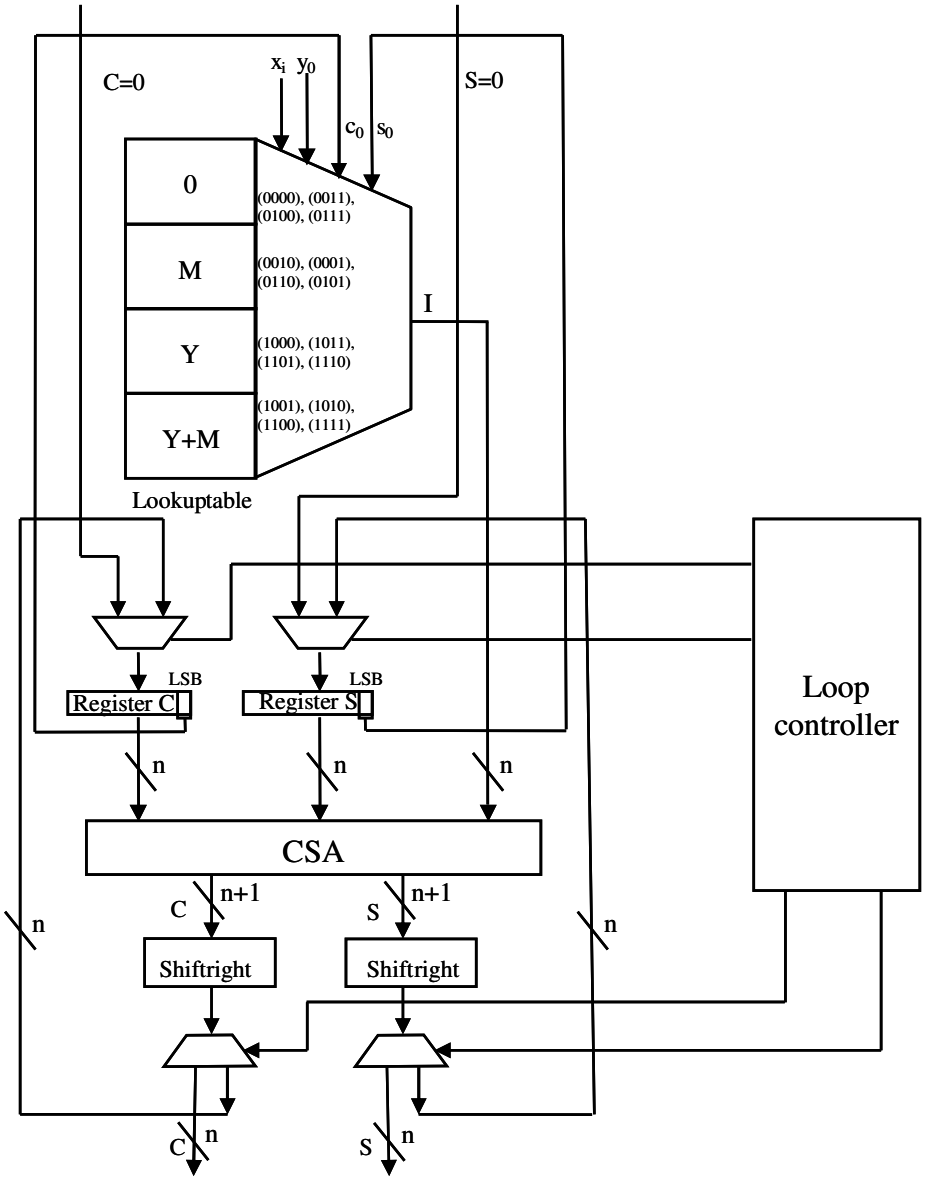

Abbildung 4.1.2: Schleifendurchlauf der schnellen Modularmultiplikation von Montgomery

Alle möglichen Werte von *I* werden in der Lookup Table gespeichert. Diese wird durch die aktuellen Bits x_i, y_0, s_0 und c_0 adressiert. Die Anzahl der Operationen in der Schleife wird damit auf einen Zugriff auf die Lookup Table und eine Carry-Save-Addition reduziert. Die Schiebeoperation, durch die die Division durch Zwei realisiert wird, kann durch einfaches Routing erreicht werden. So wird keine zusätzliche Logik gebraucht.

Korrektheit des Algorithmus 4.1.3:

Algorithmus 4.1.3 berechnet $P = X \cdot Y \cdot 2^{-n} \bmod M$.

Beweis: Es ist ausreichend zu zeigen, dass der i-te Schritt des Algorithmus 4.1.3 dieselben Werte von S und C berechnet, wie der entsprechende Schritt des Algorithmus 4.1.2.

Wenn $x_i = 0$ ist und S und C gerade Zahlen sind, dann werden S und C in dem Algorithmus 4.1.2 nicht verändert. Dasselbe gilt, wenn S und C ungerade sind, da der erste Carry-Save-Addierer s_0 auf 0 setzt. Deckungsgleich wird I im Algorithmus 4.1.3 in Schritt (4) auf 0 gesetzt.

Wenn x_i=0 ist und genau eine Variable von S und C gerade und die andere ungerade ist, dann wird im Algorithmus 4.1.2 M addiert, um aus dem Zwischenergebnis eine gerade Zahl zu bilden. Das Gleiche wird erreicht, indem im Algorithmus 4.1.3 in Schritt (5) I=M gesetzt wird.

Sei x_i =1: In diesem Fall addiert Algorithmus 4.1.2 in Schritt (3) Y zu S und C. Das Ergebnis ist eine gerade Zahl, genau wenn y_0, s_0 und c_0 vor der Addition eine gerade Parität aufweisen. Folglich verändert der Algorithmus 4.1.2 in Schritt (4) das Ergebnis nicht und der neue Wert von s_0 ist 0. Eine ähnliche Operation wird im Algorithmus 4.1.3 durchgeführt: I wird auf Y gesetzt und somit S, C und Y in Schritt (8) addiert. So werden die gleichen Werte von S und C wie im Algorithmus 4.1.2 berechnet.

Wenn das Ergebnis nach Schritt (3) in Algorithmus 4.1.2 ungerade ist, so müssen y_0, s_0 und c_0 eine ungerade Parität vor der Addition besitzen. In diesem Fall ist der Wert von $s_0 = 1$ und M wird in Schritt (4) zu S und C addiert. S und C werden um Y und M erhöht. Analoges geschieht im Algorithmus 4.1.3 in Schritt (7), indem $I = R = Y+M$ gesetzt wird. Schritt (8) erzeugt die Gesamtsumme von S, C, Y und M.

Die Grundideen dieses Abschnittes wurden in [17] veröffentlicht.

4.1.4 Komplexitätsanalyse

In den letzten Abschnitten wurde die Modularmultiplikation von Montgomery und ihre Optimierung in Bezug auf die Zeitreduktion durch das Nutzen von redundanten Carry-Save-Addierern dargestellt. In Abschnitt 4.1.3 wurde eine Möglichkeit aufgezeigt, wie die Anzahl der Carry-Save-Additionen durch das Nutzen einer Lookup Table mit vorberechneten Werten halbiert werden kann.

In diesem Abschnitt wird die Komplexität des Verfahrens mit Hilfe des in Abschnitt 3.3 definierten Komplexitätsmodells untersucht. Dafür wird Algorithmus 4.1.3 in drei

Teile unterteilt: Vorberechnungsphase, Schleife und Abschlussphase. Bei der Untersuchung des ersten Teils werden die für die Lookup Table nötigen Vorberechnungen analysiert. Danach wird die Komplexität der Schleife aus Teil Zwei berechnet und darauf folgend die nach der Schleife nötigen Operationen aus dem dritten Teil erörtert und analysiert.

Um die Analyse einfacher zu gestalten wird Algorithmus 4.1.3 auf folgende Weise ausführlich dargestellt:

Algorithmus 4.1.4: Ausführliche Darstellung der Modularmultiplikation von Montgomery mit Lookup Table

```
Eingabe: X,Y,M mit 0 ≤ X,Y < M
Ausgabe: P = (X*Y*2^-n) mod M
n: Anzahl der Bits in X,
x_i: i-tes Bit von X
s_0: LSB von S
R: Vorberechneter Wert von Y+M
Methode:
(1)  S = Y; C = M;
(2)  P = S + C;
(3)  S = 0; C = 0;
(4)  LookUp(0)=0;
(5)  LookUp(1)=M;
(6)  LookUp(2)=Y;
(7)  LookUp(3)=S;
(8)  for (int i=0; i<n; i++) {
(9)    I = LookUp(2*x_0 + not(x_0)*(s_0⊕c_0) or x_0*(s_0⊕c_0⊕y_0));
(10)   (S,C) = CSA(S, C, I);
(11)   S = S div 2; C = C div 2;
(12) }
(13) P = S + C;
(14) P'=P-M;
(15) if (P'≥0) P = P';
```

In der oben beschriebenen Schleife entspricht das Multiplikationszeichen „*“ der logischen „UND-Verknüpfung“, da es sich dabei um eine Operation mit 1-bit Zahlen handelt.

Vorberechnungsphase (Schritte (1) bis (7)):

Für die Lookup Table sind folgende Werte erforderlich: 0, *Y*, *M* und *Y*+*M*. Da 0, *Y* und *M* schon vorhanden sind, ist es nur nötig *Y*+*M* zu berechnen. Dafür wird eine nichtredundante Addition benötigt. Diese kann in 8log*n*–16 Zeiteinheiten mit einem Kombiaddierer aus Abschnitt 3.3.6 durchgeführt werden. Der Kombiaddierer bildet eine Hardwareeinheit, die vom „Loop controller“ gesteuert wird. Der Kombiaddierer liest

wahlweise aus den Registern (*S*, *C*) oder (*P*, *M*) und schreibt in die Register *P* bzw. *P*'. Aus diesem Grund wird sowohl für die Vorberechnungsphase, als auch für die Abschlussphase ein einzelner Kombiaddierer benutzt. Bei den anderen Verfahren (Abschnitte 4.2 und 4.3) wird der Kombiaddierer auf ähnliche Art verwendet.

Darüber hinaus müssen 0, *Y*, *M* und *Y*+*M* in die Lookup Table gespeichert werden. Dafür wird jeweils eine Zeiteinheit gebraucht. Demzufolge liegt die Zeitkomplexität der Vorberechnungsphase bei

$$T_{\text{Vorberechnung}} = 8\log n - 16 + 4 = 8\log n - 12 \tag{4.17}$$

Zeiteinheiten.

Jede Schleifeniteration benötigt eine Carry-Save-Addition, das Berechnen der Adresse für die Lookup Table und die Adressierung der Lookup Table. Dabei kann die Berechnung der Adresse für die Lookup Table parallel zur Carry-Save-Addition stattfinden. Somit reduziert sich die Zeitkomplexität eines Schleifendurchlaufs auf die Zeitkomplexität der Carry-Save-Addition und der Adressierung der Lookup Table, demgemäß auf zwei Zeiteinheiten. Da die Schleife *n* Mal durchlaufen wird, ist die Gesamtzeitkomplexität der Schleife

$$T_{\text{Schleife}} = 2n \tag{4.18}$$

Zeiteinheiten.

Bei der Abschlussphase müssen zwei vollständige Additionen durchgeführt werden (Schritte (13) und (14)). Dementsprechend beträgt die Zeitkomplexität der Abschlussphase

$$T_{\text{Abschschlussphase}} = 2 \cdot (8\log n - 16) = 16\log n - 32 \tag{4.19}$$

Zeiteinheiten und die Gesamtzeitkomplexität liegt bei

$$T_{\text{Montgomery}} = 8\log n - 12 + 2n + 16\log n - 32 = 2n + 24\log n - 44 \tag{4.20}$$

Zeiteinheiten.

Nun folgt die Untersuchung der Flächenkomplexität:

Für die Vorberechnungsphase werden ein Kombiaddierer und drei Register *Y*, *M* und *S* benötigt. Die Schleife erfordert einen Carry-Save-Addierer, eine Lookup Table mit vier *n*-bit Zahlen und drei Register *S*, *C* und *X*. Für die Abschlussphase werden ein Kombiaddierer und drei Register *P*, *P*' und *M* gebraucht. Allerdings können anstelle von *P* und *P*' die Register *S* und *C* benutzt werden. Somit werden insgesamt ein Carry-

Save-Addierer, ein Kombiaddierer, eine 4*n*-bit Lookup Table und fünf Register *X*, *Y*, *M*, *S* und *C* verwendet.

Die Flächenkomplexität einer Montgomery Modularmultiplikation beträgt damit

$$A_{Montgomery} = 8n + 6n + 4n + 5 \cdot 4n = 38n \tag{4.21}$$

Flächeneinheiten.

Dementsprechend liegt die Flächen-Zeit-Komplexität der Modularmultiplikation von Montgomery bei

$$AT_{Montgomery} = 38n \cdot (2n + 24\log n - 44) = 76n^2 + 912n\log n - 1672n \tag{4.22}$$

Allerdings liefert eine Modularmultiplikation von Montgomery nicht das gewünschte Ergebnis, sondern $(X \cdot Y \cdot 2^{-n}) \bmod M$. Eine vollständige Modularmultiplikation besteht dagegen aus zwei Modularmultiplikationen von Montgomery

$$\begin{aligned} X \cdot Y \bmod M &= Montgomery(Montgomery((X, Y, M), (2^{2n} \bmod M), M)) \\ &= Montgomery((X \cdot Y * 2^{-n} \bmod M), (2^{2n} \bmod M), M) \\ &= ((X \cdot Y \cdot 2^{-n} \bmod M) \cdot (2^{2n} \bmod M) \cdot 2^{-n}) \bmod M \\ &= (X \cdot Y \cdot 2^{-n} \cdot 2^{2n} \cdot 2^{-n}) \bmod M \\ &= X \cdot Y \bmod M \end{aligned} \tag{4.23}$$

Dementsprechend ist die Zeitkomplexität einer vollständigen Modularmultiplikation

$$T_{Modularmultiplikation} = 2T_{Montgomery} = 2(2n + 24\log n - 44) = 4n + 48\log n - 88 \tag{4.24}$$

Die Flächen-Zeit-Komplexität einer vollständigen Modularmultiplikation beträgt damit

$$\begin{aligned} AT_{Modularmultiplikation} &= 2 \cdot AT_{Montgomery} \\ &= 2 \cdot (76n^2 + 912n\log n - 1672n) = 152n^2 + 1824n\log n - 3344 \end{aligned} \tag{4.25}$$

Flächen-Zeit-Einheiten.

Die folgende Tabelle präsentiert die Komplexität der vollständigen Modularmultiplikation von Montgomery für unterschiedliche Wortlängen

n	Flächenkomplexität/ Flächeneinheit	Zeitkomplexität/ Zeiteinheit	AT-Komplexität/ Flächen-Zeit-Einheit
512	19.456	2.392	46.538.752
1024	38.912	4.488	174.637.056
2048	77.824	8.632	671.776.768
4096	155.648	16.872	2.626.093.056
8192	311.296	33.304	10.367.401.984

Tabelle 4-1: Komplexität der vollständigen Modularmultiplikation von Montgomery

Wenn allerdings eine mehrfache Anwendung der Modularmultiplikation benötigt wird, ist es nicht notwendig das Ergebnis nach jeder Modularmultiplikation von Montgomery zurück zu konvertieren, sondern es wird im so genannten Montgomery-Raum gearbeitet. Lediglich am Ende der Gesamtberechnung wird das Ergebnis durch eine weitere Modularmultiplikation von Montgomery zurückkonvertiert. Solche Berechnungen werden in Abschnitt 5.2 näher betrachtet.

4.1.5 Zusammenfassung

In diesem Kapitel wurde der Algorithmus von Montgomery für die Modularmultiplikation, sowie die Erweiterung auf die Carry-Save-Addition vorgestellt. Die Innovation in dieser Arbeit ist es, die Anzahl der nötigen Additionen durch die Nutzung einer Lookup Table mit vorberechneten Werten zu halbieren. Ferner wurde die Komplexität des Verfahrens nach dem in Abschnitt 3.3 vorgestellten Komplexitätsmodell analysiert.

4.2 Interleaved Modularmultiplikation.

4.2.1 Interleaved Modularmultiplikation

Der wichtigste Algorithmus für Modularmultiplikation nach dem von Montgomery ist die Interleaved oder verschränkte Modularmultiplikation. Die Idee der Interleaved Modularmultiplikation besteht darin, die Multiplikation und die Division ineinander zu verschränken und somit das Zwischenergebnis so klein wie möglich zu halten. Das geschieht, indem die einzelnen Schritte der Multiplikation und der Division im Zeitmultiplex durchgeführt werden. Das Problem besteht darin, dass bei n-bit-zahligen Operanden das Produkt eine $2n$-bit Zahl ist.

Beispiel: $P = X{\cdot}Y \bmod M = 12{\cdot}13 \bmod 15 = 1100_2 \cdot 1101_2 \bmod 1111_2$.
Zuerst wird das Produkt $P_1 = 1100 \cdot 1101 = 10011100$, danach wird $P = 10011100 \bmod 1111 = 0110 = 6_{10}$ berechnet. Dabei ist das Zwischenprodukt P_1 eine $2n$-bit Zahl.

Die Idee der verschränkten Modularmultiplikation wird auf die folgende Art und Weise realisiert: In jedem Schritt wird ein Bit x_i des ersten Operanden X mit dem zweiten Operanden Y multipliziert. Das Ergebnis dieser Multiplikation wird zum Zwischenergebnis addiert. Von diesem Zwischenergebnis wird der Modulus M wiederholt subtrahiert, bis das Zwischenergebnis kleiner als M ist. Die mehrfachen Additionen liefern das Produkt $X{\cdot}Y$. Von diesem Produkt wird M so oft abgezogen, dass es im Intervall zwischen 0 und M liegt. Der folgende Algorithmus zeigt die Funktionsweise der verschränkten Modularmultiplikation:

Algorithmus 4.2.1: Verschränkte Modularmultiplikation

```
Eingabe: X, Y, M mit 0≤ X,Y ≤ M
Ausgabe: P = X*Y mod M
n: Anzahl der Bits in X,
x_i: i-tes Bit von X
Methode:
(1)  P=0;
(2)  for (int i=n-1; i≥0; i--){
(3)    P=2*P;
(4)    I=x_i*Y;
(5)    P=P+I;
(6)    if (P≥M) P=P-M;
(7)    if (P≥M) P=P-M;
(8)  }
```

Satz (Schleifeninvariante): Nach jedem Durchlauf der Schleife liegt P im Intervall zwischen 0 und M.

Beweis durch vollständige Induktion: Es wird ähnlich wie bei dem Beweis des Algorithmus 4.1.1 folgende Notation verwendet: Das Resultat des Schrittes (1) wird als P_0 bezeichnet. Die Indizierung der Variablen in der Schleife hängt von der Iterationsnummer ab, so repräsentieren P_k', P_k'' und P_k die Ergebnisse der Schritten (3), (5) und (7) des k-ten Schleifendurchlaufs, wobei k eine beliebige Zahl zwischen 1 und n ist.

Vor der Schleife gilt:

$$P_0 = 0 < M \tag{4.26}$$

Annahme: Nach dem *k*-ten Schleifendurchlauf gilt:

$$P_k < M \tag{4.27}$$

Zu zeigen: Nach dem *k*+1-ten Schleifendurchlauf gilt:

$$P_{k+1} < M \tag{4.28}$$

Beweis:

Nach Schritt (3) gilt:

$$P'_{k+1} = 2P_k < 2M \tag{4.29}$$

Nach Schritt (5) gilt:

$$P''_{k+1} = P'_{k+1} + M < 2M + M = 3M \tag{4.30}$$

Und nach den Schritten (6) und (7) gilt:

$$P_{k+1} \leq P''_{k+1} - M - M < 3M - M - M = M \tag{4.31}$$

Somit ist der Satz bewiesen.

Die folgende Abbildung zeigt die Funktionsweise der verschränkten Modularmultiplikation:

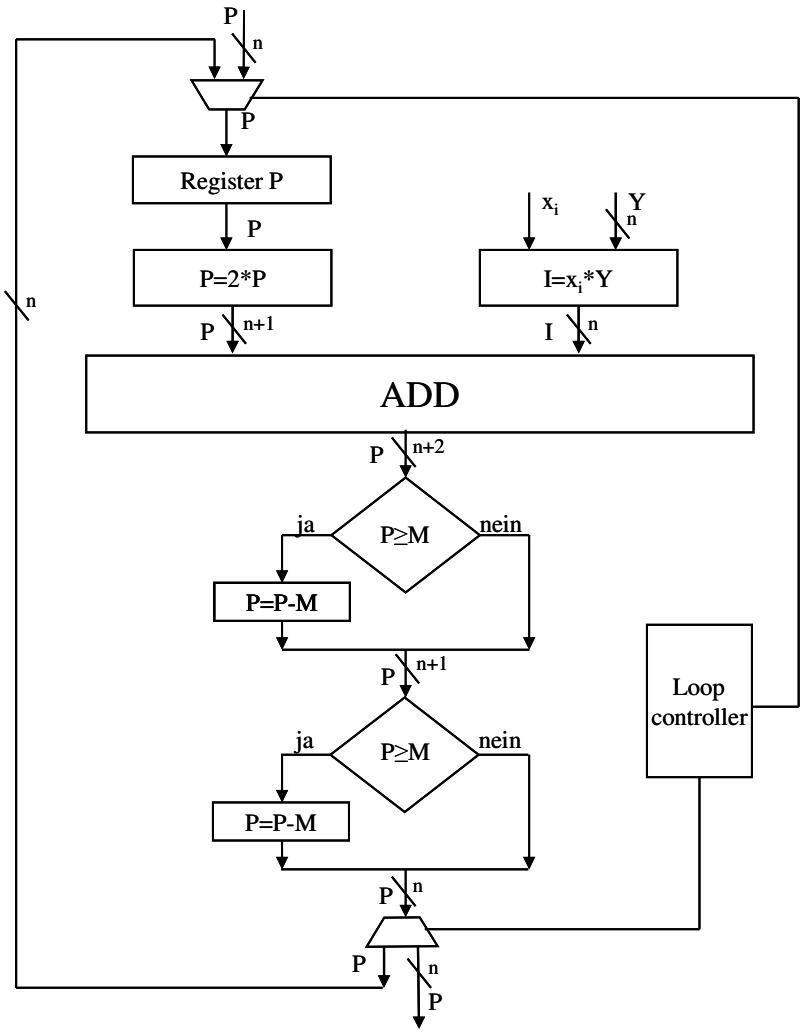

Abbildung 4.2.1: Ein Schleifendurchlauf des Algorithmus 4.2.1

Der Algorithmus 4.2.1 weist Vorteile gegenüber der Multiplikation mit der nachfolgenden Division auf:

- Es wird nur eine Schleife für alle Operationen benötigt.
- Das Zwischenergebnis ist nie länger als eine (n+2)-bit Zahl (dadurch wird die Addierer- und Registerfläche reduziert).

Allerdings gibt es auch Nachteile:

- Der Algorithmus erfordert drei nichtredundante Additionen in den Schritten (5), (6) und (7) (die Verdopplung von P ist nur eine Schiebeoperation).
- Um die Vergleiche in den Schritten (6) und (7) durchzuführen, werden die Ergebnisse der vorhergehenden Additionen benötigt. Dies ist für das Zeitverhalten sehr

wichtig, da bei der Addition der großen Zahlen die Zeit, die für die Generierung der Übertragungsbits benötigt wird, eine sehr große Rolle spielt.

– Die Vergleiche in den Schritten (6) und (7) müssen im schlechtesten Fall die Untersuchung der Operanden auf voller Bitlänge durchführen. Im Unterschied zur Addition werden die Operanden bei den Vergleichen in der Reihenfolge vom höchstwertigsten bis zum niederwertigsten Bit untersucht. Aus diesem Grund können eine Addition und ein Vergleich nicht ohne Zeitverlust in einer Pipeline ausgeführt werden.

In diesem Abschnitt wurde das Verfahren der verschränkten Modularmultiplikation, sowie dessen Vor- und Nachteile beschrieben. Im nächsten Abschnitt werden die Nachteile der verschränkten Modularmultiplikation durch algorithmische Verbesserungen beseitigt.

4.2.2 Optimierung des Algorithmus im Bezug auf Fläche und Zeit

Wie schon oben beschrieben, tritt beim Vergleich des Zwischenergebnisses mit dem Modulus eine gewisse Zeitverzögerung auf. Diese entsteht, da bei einem Vergleich im schlechtesten Falle alle Bits der Operanden berücksichtigt werden müssen. Um dieses Problem zu vermeiden, wird auf den vollständigen Vergleich verzichtet und an dessen Stelle ein approximativer Vergleich durchgeführt: Das Zwischenergebnis P wird mit einer Konstanten 2^n verglichen, wobei n die Anzahl der Bits von M ist. Dabei gilt folgende Ungleichung:

$$M < 2^n < 2M \tag{4.32}$$

Sollte beim Vergleich $P \geq 2^n$ sein, wird P durch $P–2^n+(2^n \bmod M)$ ersetzt. Da

$$P - 2^n + (2^n \bmod M) = P - 2^n + 2^n - kM = P - kM \cong P \bmod M \tag{4.33}$$

ist, bleibt die Kongruenz modulo M erhalten.

Der Vorteil bei diesem Vergleich liegt in der Tatsache, dass nicht die vollständigen Operanden, sondern nur wenige Bits (unter Umständen nur ein Bit) vom Zwischenergebnis P zum Vergleich herangezogen werden müssen.

2^n mod M ist dabei eine Konstante, die sich während der ganzen Modularmultiplikation nicht ändert. Aus diesem Grund kann diese Konstante vor der Modularmultiplikation berechnet und gespeichert werden. Diese Konstante A ist ein so genannter Korrekturwert. Jedes Mal, wenn 2^n mod M benötigt wird, wird sie gelesen und verwendet. Der Pseudocode des veränderten Programms sieht folgendermaßen aus:

Algorithmus 4.2.2: Verschränkte Modularmultiplikation mit approximativem Vergleich

```
Eingabe: X, Y, M mit 0≤ X,Y ≤ M
Ausgabe: P = X*Y mod M
n: Anzahl der Bits in X,
x_i: i-te Bit von X
Methode:
(1)  A=2^n mod M;
(2)  P=0;
(3)  for (int i=n-1; i≥0; i--){
(4)    P=2*P;
(5)    I=x_i*Y;
(6)    P=P+I;
(7)    if (P≥2^n) P=P-2^n+A;
(8)    if (P≥2^n) P=P-2^n+A;
(9)    if (P≥2^n) P=P-2^n+A;
(10)   if (P≥2^n) P=P-2^n+A;
(11) }
(12) if (P≥M) P=P-M;
```

Das Zwischenergebnis P ist jetzt kleiner als 2^n und somit kleiner als $2M$. Also muss in Schritt (12) nach der Schleife P mit M verglichen und danach falls nötig M abgezogen werden.

Nach Ausführung von Schritt (4) ist P kleiner als $2 \cdot 2^n < 4M$. Nach Schritt (6) ist P daher kleiner als $5M$. In den Schritten (7) bis (10) wird das Zwischenergebnis P mit 2^n verglichen und im Fall $P \geq 2^n$, wird

$$P=P-2^n+A=P-2^n+(2^n \bmod M)=P-2^n+2^n-kM=P-kM \leq P-M \text{ mit } k>0 \qquad (4.34)$$

berechnet. Dies bedeutet, dass bei jeder Subtraktion mindestens einmal M abgezogen wird. Somit werden bis zu vier Subtraktionen und vier Additionen benötigt, damit P wieder kleiner als 2^n ist. Die Subtraktion von 2^n ist nur das Setzen eines Bits von 1 auf 0. Die nachfolgende Addition ist leider eine komplette Addition.

Vier Additionen in jedem Schleifendurchlauf benötigen einen hohen Zeitaufwand. Allerdings ist es möglich, die Anzahl der erforderlichen Additionen abzuschätzen. Wie oben gezeigt, ist das Zwischenergebnis P nach Schritt (6) kleiner als $5M$ und somit klei-

ner als $5 \cdot 2^n$. Dies wiederum bedeutet, dass P eine n+3-bit-lange Zahl ist und die drei höchstwertigsten Bits folgende Werte annehmen können: 000, 001, 010, 011, 100. Der Korrekturwert A kann daher wie folgt berechnet werden:

$$p_{n+2}p_{n+1}p_n = \begin{cases} 000 & A = 0 \\ 001 & A = 2^n \bmod M \\ 010 & A = 2 \cdot 2^n \bmod M \\ 011 & A = 3 \cdot 2^n \bmod M \\ 100 & A = 4 \cdot 2^n \bmod M \end{cases} \qquad 4.35$$

Das Zwischenergebnis P wird modulo M reduziert. Der Korrekturwert A wird am Anfang des nächsten Schleifendurchlaufs zum Zwischenergebnis P addiert. Die folgende Abbildung zeigt die Funktionsweise dieser Verbesserung.

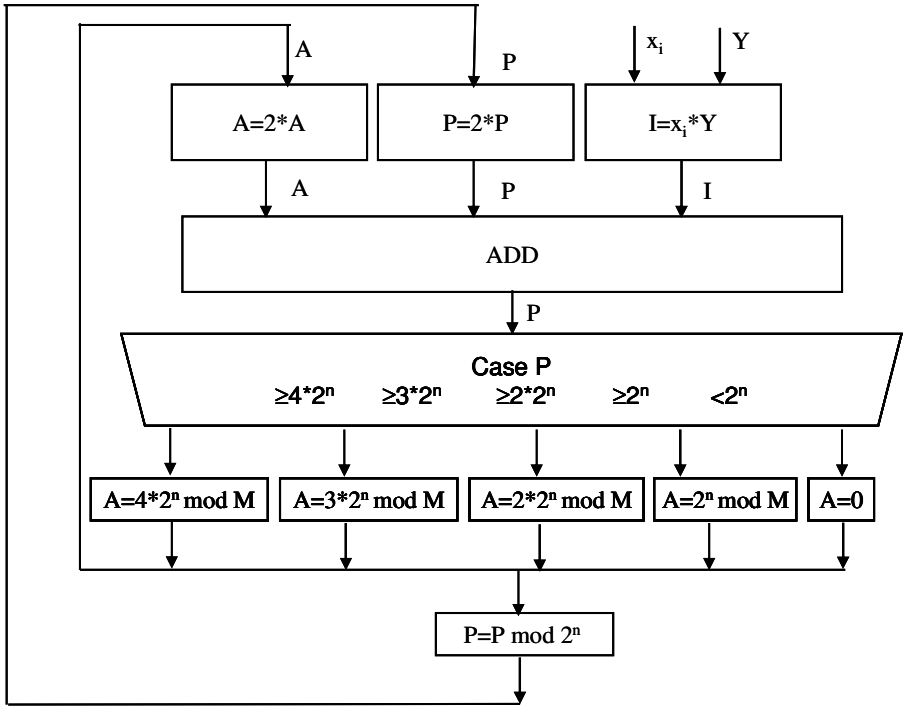

Abbildung 4.2.2: Ein Schleifendurchlauf der verbesserten verschränkten Modularmultiplikation

Bei der Abbildung 4.2.2 handelt es sich lediglich um die interne Schleife. Weitere Abbildungen werden den reellen Datenfluss der Algorithmen präsentieren.

Alle Korrekturwerte sind weder von X noch von Y abhängig. Das heißt, dass es ausreicht, diese fünf Werte vor der eigentlichen Schleife zu berechnen und für die weitere Verwendung zu speichern. Eine Lookup Table bietet hierbei die effizienteste Methode. Die Abbildung 4.2.3 zeigt den Datenfluss der verschränkten Modularmultiplikation in dieser Variante.

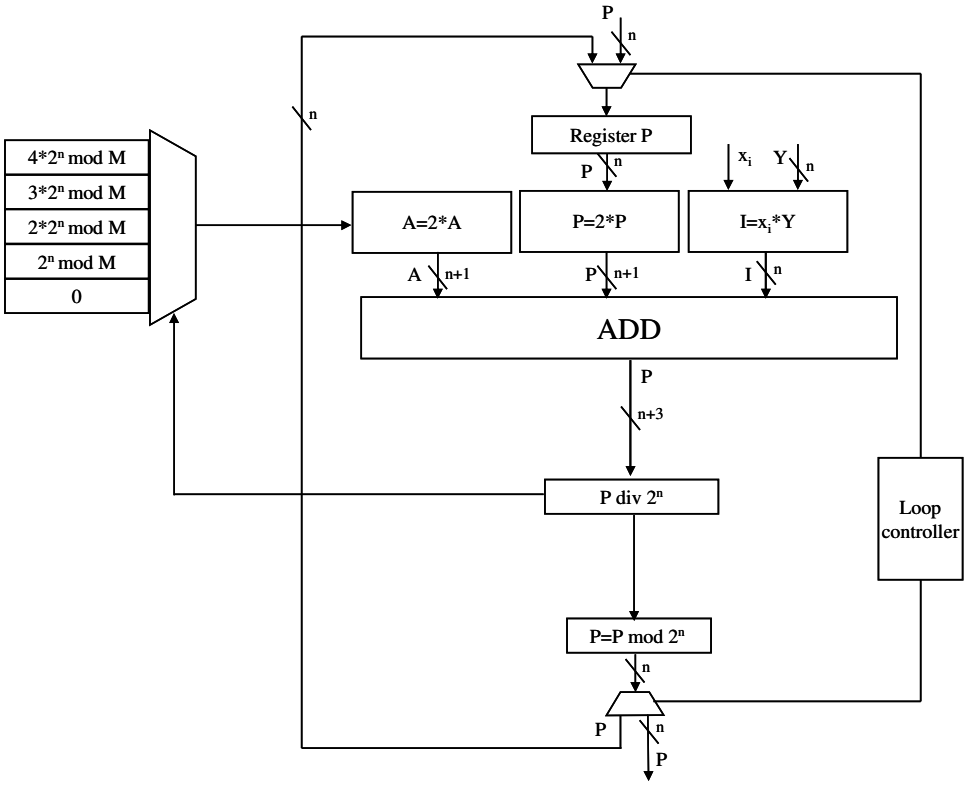

Abbildung 4.2.3: Verschränkte Modularmultiplikation unter Benutzung einer Lookup Table

Der folgende Pseudocode beschreibt die oben beschriebenen Verbesserung:

Algorithmus 4.2.3: Verschränkte Modularmultiplikation unter Benutzung einer Lookup Table

```
Eingabe: X, Y, M mit 0≤ X,Y ≤ M
Ausgabe: P = X*Y mod M
n: Anzahl der Bits in X,
x_i: i-te Bit von X
LookUp(4) = 4*2^n mod M;
LookUp(3) = 3*2^n mod M;
LookUp(2) = 2*2^n mod M;
LookUp(1) =   2^n mod M;
LookUp(0) = 0;
Methode:
(1)  P=0; A=0;
(2)  for (int i=n-1; i≥0; i--){
(3)    P = P mod 2^n;
(4)    A = 2*A;
(5)    P = 2*P;
(6)    I = x_i*Y;
(7)    P = P+A;
(8)    P = P+I;
(9)    A = LookUp(p_{n+2}p_{n+1}p_n);
(10) }
(11) if (P≥M) P=P-M;
```

Alle Werte in der Lookup Table sind kleiner als 2^n, somit ist der Korrekturwert $A < 2^n$. Deshalb gilt:

Nach Schritt (4) $A<2\cdot 2^n$. Da P am Anfang der Schleifeniteration kleiner als 2^n ist, ist P nach Schritt (5) kleiner als $2\cdot 2^n$. Außerdem gilt: $x_i \cdot Y \le Y < M < 2^n$ nach Schritt (6) und demgemäß nach Schritt (8):

$$P < 2\cdot 2^n + 2\cdot 2^n + 2^n = 5\cdot 2^n \tag{4.36}$$

In Schritt (4) wird der Korrekturwert A verdoppelt. Da A ein vorberechneter Wert ist und niemals ohne Verdopplung benutzt wird, ist es sinnvoll anstelle von A die Verdopplung $2A \bmod M$ in die Lookup Table zu speichern (siehe nachfolgende Abbildung).

Der Korrekturwert A ist jetzt kleiner als 2^n. Somit gilt

$$P < 2^n + 2\cdot 2^n + 2^n = 4\cdot 2^n \tag{4.37}$$

Somit werden nur vier Korrekturwerte benötigt und die Lookup Table kann um einen Wert reduziert werden. Um die Adresse für die Lookup Table zu berechnen, werden nur die beiden höchstwertigsten Bits gebraucht.

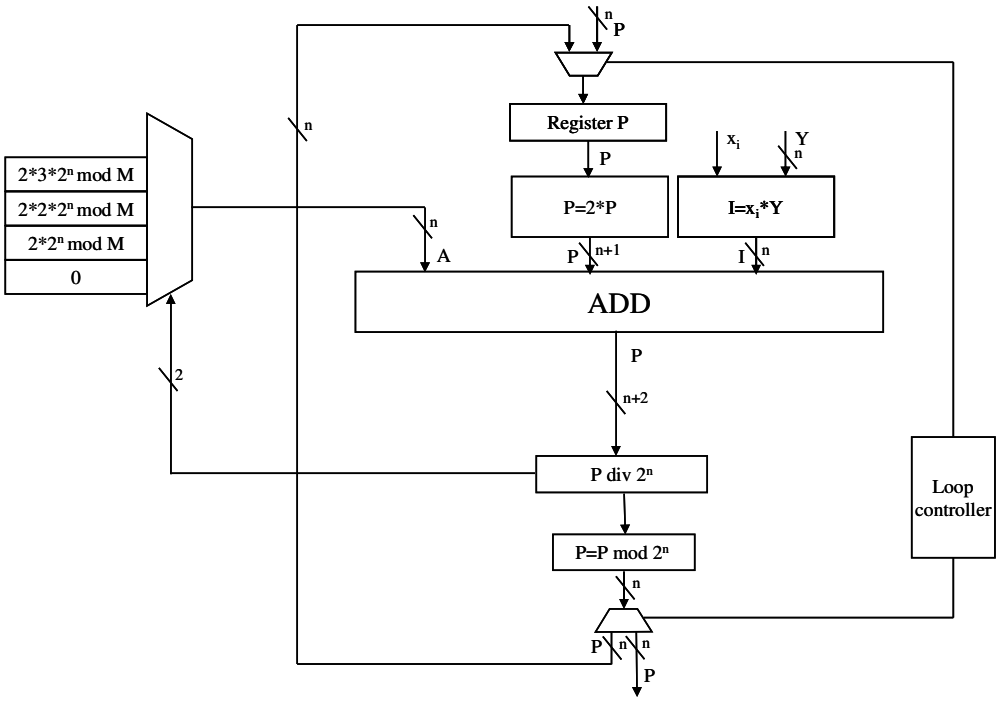

Abbildung 4.2.4: Ein Schleifendurchlauf der verschränkten Modularmultiplikation ohne Verdopplung des Korrekturwertes

Der folgende Pseudocode beschreibt diese Modifikation:

Algorithmus 4.2.4: Verschränkte Modularmultiplikation ohne Verdopplung des Korrekturwertes

```
Eingabe: X, Y, M mit 0≤ X,Y ≤ M
Ausgabe: P = X*Y mod M
n: Anzahl der Bits in X,
x_i: i-te Bit von X
LookUp(0) = 0;
LookUp(1) =   2*2^n mod M;
LookUp(2) = 2*2*2^n mod M;
LookUp(3) = 2*3*2^n mod M;
Methode:
(1)  P=0;
(2)  for (int i=n-1; i≥0; i--){
(3)    P = P mod 2^n;
(4)    P = 2*P;
(5)    I = x_i*Y;
(6)    P = P+A;
(7)    P = P+I;
(8)    A = LookUp(p_{n+1}p_n);
(9)  }
(10) P = P mod M;
```

In jedem Schleifendurchlauf werden zwei Additionen ausgeführt: Das Zwischenergebnis P wird um A und das Ergebnis dieser Addition noch einmal um $I=x_i \cdot Y$ erhöht. Der Operand I kann nur zwei Werte annehmen, da das Bit x_i entweder 0 oder 1 sein kann:

$$x_i = \begin{cases} 0 & I = 0 \\ 1 & I = Y \end{cases} \tag{4.38}$$

Die zwei möglichen Werte von I können zusammen mit den Korrekturwerten vorberechnet und in die Lookup Table gespeichert werden. Der folgende Datenfluss beschreibt einen solchen Schleifendurchlauf:

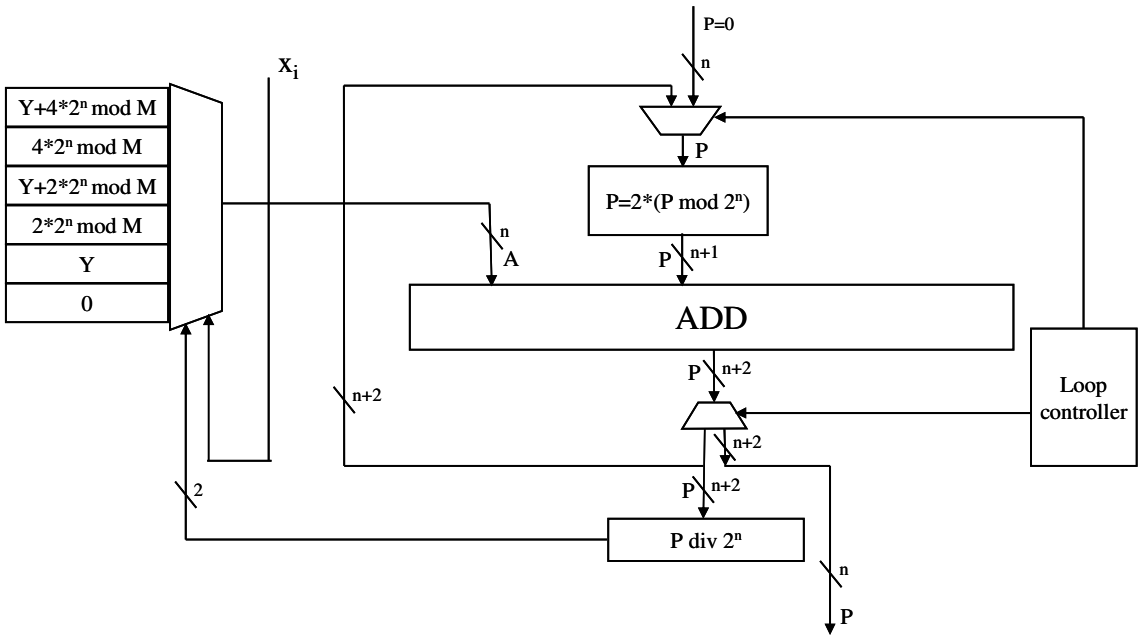

Abbildung 4.2.5: Ein Schleifendurchlauf der verschränkten Modularmultiplikation mit einem Addierer

An dieser Stelle wird im Gegensatz zu den bisherigen Algorithmen nur ein einziger Addierer benötigt. Die Lookup Table wird jedoch etwas größer, da *Y* berücksichtigt und somit sechs Werte gespeichert werden müssen.

Das komplette Verfahren wird im Algorithmus 4.2.5 präsentiert:

Algorithmus 4.2.5: Verschränkte Modularmultiplikation mit einem Addierer

```
Eingabe: X, Y, M mit 0≤ X,Y < M
Ausgabe: P = X*Y mod M
n: Anzahl der Bits in X,
x_i: i-te Bit von X
x_-1=0;
LookUp(0) = 0;
LookUp(1) = Y;
LookUp(2) = 2*2^n mod M;
LookUp(3) = (Y+2*2^n) mod M;
LookUp(4) = (4*2^n) mod M;
LookUp(5) = (Y+4*2^n) mod M;
Methode:
(1)  P=0;
(2)  A=LookUp(x_n-1);
(3)  for (int i=n-1; i≥0; i--){
(4)    P = P mod 2^n;
(5)    P = 2*P;
(6)    P = P+A;
(7)    A = Lookup(2*p_n+1+p_n+x_i-1);
(8)  }
(9)  P = P mod M;
```

In Schritt (4) ist $P < 2^n$. Nach der Verdopplung in Schritt (5) ist $P < 2 \cdot 2^n$. Der Korrekturwert beträgt $A < M < 2^n$. Außerdem gilt $M < 2^n < 2M$. Demzufolge gelten nach Schritt (6) folgende Ungleichungen:

$$P < 2 \cdot 2^n + M < 2 \cdot 2^n + 2^n = 3 \cdot 2^n \tag{4.39}$$

und

$$P < 2 \cdot 2^n + M < 4M + M = 5M \tag{4.40}$$

Das heißt, dass nach der Schleife in Schritt (9) der Modulus M noch bis zu viermal vom Ergebnis P abgezogen werden muss.

In diesem Abschnitt wurden mehrere Verbesserungen der verschränkten Modularmultiplikation dargestellt. Alle Vergleiche wurden durch approximative Vergleiche ersetzt, die durch die Analyse der zwei bis drei höchstwertigsten Bits in konstanter Zeit durchgeführt werden können. Außerdem wurde die Anzahl der Additionen in jedem Schleifendurchlauf von drei (Algorithmus 4.2.1) auf nur eine (Algorithmus 4.2.5) reduziert. Ferner wurde die Lookup Table als eine effiziente Möglichkeit, bestimmte mehrfach verwendete Werte zu speichern, genutzt. Alle diese Verbesserungen ziehen gewisse Nachteile mit sich: Es müssen sechs Werte vor der Schleife berechnet und in die Lookup Table gespeichert werden. Darüber hinaus müssen nach der Schleife bis zu vier Subtraktionen durchgeführt werden, um das Ergebnis in den Bereich zwischen 0 und M zurückzuführen. Zudem existiert noch ein weiterer gravierender Nachteil: Die Addition, die in jedem Schleifendurchlauf durchgeführt wird, ist eine nichtredundante Addition und somit von der Länge der Operanden abhängig.

4.2.3 Verschränkte Modularmultiplikation mit einem Carry-Save-Addierer

Im letzten Abschnitt wurde die verschränkte Modularmultiplikation so verbessert, dass sie nur eine einzige nicht redundante Addition pro Schleifendurchlauf benötigt. In diesem Abschnitt wird eine Möglichkeit vorgestellt, wie die herkömmliche Addition durch eine Carry-Save-Addition ohne Flächenverlust ersetzt werden kann, um jeden Schleifendurchlauf in konstanter Zeit durchführen zu können. Die Carry-Save-Addition wurde in Abschnitt 3.2.1 eingeführt.

An der Stelle des konventionellen Addierers (Algorithmus 4.2.5, Schritt (6)) wird ein Carry-Save-Addierer eingefügt. Dessen Ausgaben *S* und *C* werden ähnlich wie *P* in Algorithmus 4.2.5 behandelt: *S* und *C* aus dem *i*-ten Durchlauf werden als Eingaben für den *i*+1-ten Schleifendurchlauf benutzt und durch die Analyse der zwei höchstwertigsten Bits von *S* und *C* wird der Korrekturwert ermittelt. Die genaue Funktionsweise eines Schleifendurchlaufs ist in der folgenden Abbildung dargestellt:

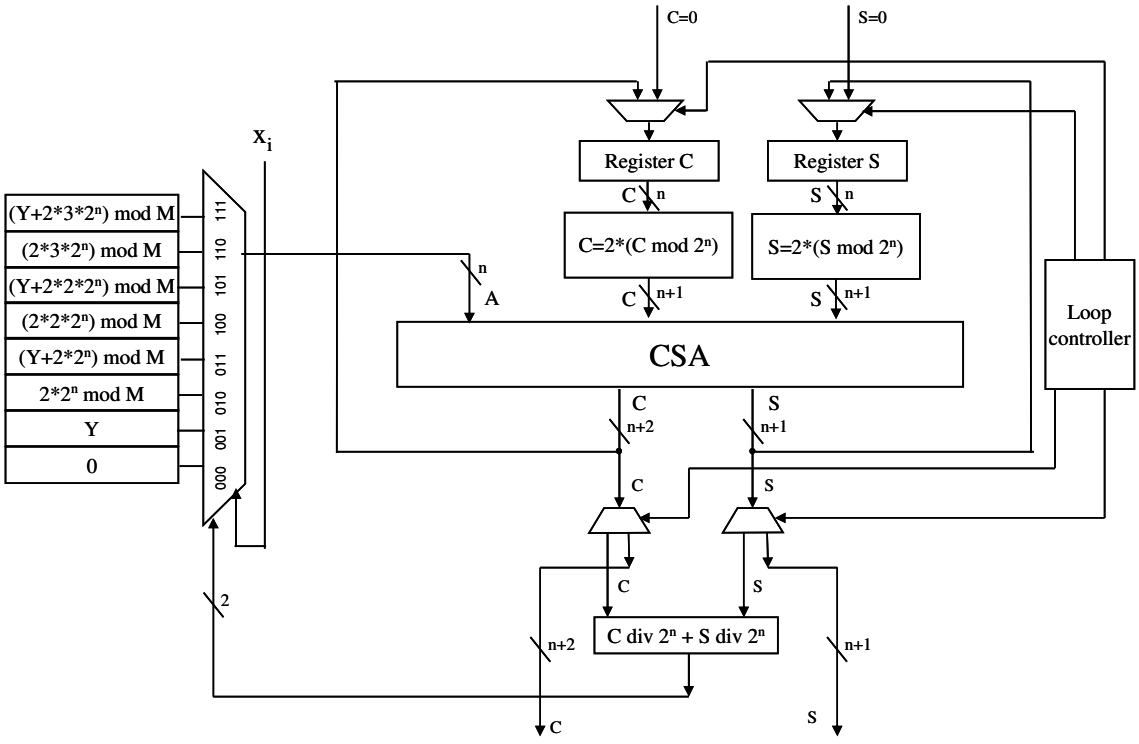

Abbildung 4.2.6: Ein Schleifendurchlauf der verschränkten Modularmultiplikation mit einem Carry-Save-Addierer

Vor der Addition sind *S* und *C* zwei *n*+1-bit-lange Zahlen. Der Korrekturwert *A* ist eine *n*-bit-lange Zahl, demzufolge ist *S* nach der Carry-Save-Addition eine *n*+1-bit und *C* eine *n*+2-bit-lange Zahl. Dabei gilt:

$$S + C < 2 \cdot 2^n + 2 \cdot 2^n + M < 4 \cdot 2^n + 2^n = 5 \cdot 2^n \tag{4.41}$$

und

$$S + C < 2 \cdot 2^n + 2 \cdot 2^n + M < 8M + M = 9M \tag{4.42}$$

Da aber der Korrekturwert $A<M$ und daher eine *n*-bit Zahl ist, ist a_n=0. Demgemäß ist die Summe

$$2c_{n+1} + s_n = c_n^{alt} + s_n^{alt} + a_n = c_n^{alt} + s_n^{alt} + 0 \le 2 \tag{4.43}$$

und

$$c_n \leq 1 \tag{4.44}$$

demzufolge ist

$$2c_{n+1} + s_n + c_n \leq 2 + 1 = 3 \tag{4.45}$$

Das heißt, dass der Korrekturwert acht mögliche Werte annehmen kann:

$$A = \begin{cases} 0 & 2c_{n+1} + s_n + c_n = 0 \text{ und } x_i = 0 \\ Y & 2c_{n+1} + s_n + c_n = 0 \text{ und } x_i = 1 \\ 2 \cdot 2^n \bmod M & 2c_{n+1} + s_n + c_n = 1 \text{ und } x_i = 0 \\ (Y + 2 \cdot 2^n) \bmod M & 2c_{n+1} + s_n + c_n = 1 \text{ und } x_i = 1 \\ 4 \cdot 2^n \bmod M & 2c_{n+1} + s_n + c_n = 2 \text{ und } x_i = 0 \\ (Y + 4 \cdot 2^n) \bmod M & 2c_{n+1} + s_n + c_n = 2 \text{ und } x_i = 1 \\ 6 \cdot 2^n \bmod M & 2c_{n+1} + s_n + c_n = 3 \text{ und } x_i = 0 \\ (Y + 6 \cdot 2^n) \bmod M & 2c_{n+1} + s_n + c_n = 3 \text{ und } x_i = 1 \end{cases} \tag{4.46}$$

Der folgende Pseudocode zeigt das Gesamtverfahren:

Algorithmus 4.2.6: Verschränkte Modularmultiplikation mit redundanter Carry-Save-Addition

```
Eingabe: X, Y, M (X, Y < M);
Ausgabe: P = X * Y mod M;
n: Anzahl der Bits in X,
x_i: i-tes Bit von X
x_-1=0;
LookUp(7) = (2*3*2^n + Y) mod M; LookUp(6) = 2*3*2^n mod M;
LookUp(5) = (2*2*2^n + Y) mod M; LookUp(4) = 2*2*2^n mod M;
LookUp(3) = (2*1*2^n + Y) mod M; LookUp(2) = 2*1*2^n mod M;
LookUp(1) = Y; LookUp(0) = 0;
Methode:
(1)  S = 0;
(2)  C = 0;
(3)  A = LookUp(x_n-1);
(4)  for (int i=n-1; i≥0; i--) {
(5)    S = S mod 2^n;
(6)    C = C mod 2^n;
(7)    S = 2*S;
(8)    C = 2*C;
(9)    (S, C) = CSA (S, C, A);
(10)   A=LookUp(2*(s_n+2*c_n+1+c_n)+x_i-1);
(11) }
(12) P = (S + C) mod M;
```

Nun benötigt die Schleife nur noch eine einzige Carry-Save-Addition pro Durchlauf. Die Carry-Save-Addition hängt nicht von der Operandenlänge ab und kann in konstanter Zeit ausgeführt werden.

Das Ergebnis der Schleife ist $S+C<9M$. Demgemäß sind eine Addition und bis zu acht Subtraktionen erforderlich, um $P=(S+C) \bmod M$ zu erhalten:

```
Reduktion:
(1)  P = S +C;
(2)  for (int i=0; i<8; i++) {
(3)    if (P≥M) P=P-M;
(4)  }
```

Allerdings ist es möglich die Anzahl der Subtraktionen zu reduzieren: In Schritt (12) des Algorithmus 4.2.6 wird $P=(S+C)<9M$, anstatt von $P=(S+C) \bmod M$ berechnet. Der Algorithmus 4.2.6 wird auf folgende Weise verändert:

Algorithmus 4.2.7: Verschränkte Modularmultiplikation mit verkürzter Abschlussphase

```
Eingabe: X, Y, M (X, Y < M);
Ausgabe: P = X * Y mod M;
n: Anzahl der Bits in X,
x_i: i-te Bit von X
x_-1=0;
LookUp(7) = (2*3*2^n + Y) mod M; LookUp(6) = 2*3*2^n mod M;
LookUp(5) = (2*2*2^n + Y) mod M; LookUp(4) = 2*2*2^n mod M;
LookUp(3) = (2*1*2^n + Y) mod M; LookUp(2) = 2*1*2^n mod M;
LookUp(1) = Y; LookUp(0) = 0;
Methode:
(1)  S = 0;
(2)  C = 0;
(3)  A = LookUp(x_n-1);
(4)  for (int i=n-1; i≥0; i--) {
(5)    S = S mod 2^n;
(6)    C = C mod 2^n;
(7)    S = 2*S;
(8)    C = 2*C;
(9)    (S, C) = CSA (S, C, A);
(10)   A=LookUp(2*(s_n+2*c_n+1+c_n)+x_i-1);
(11) }
(12) P = S + C;
(13) P' = P-4M;
(14) if (P'>0) P = P';
(15) P' = P-2M;
(16) if (P'>0) P = P';
(17) P' = P-M;
(18) if (P'>0) P = P';
(19) P' = P-M;
(20) if (P'>0) P = P';
```

Satz: Der Algorithmus 4.2.7 berechnet $P = X \cdot Y \bmod M$

Beweis: Der Beweis besteht aus drei Teilen, aus denen die Aussage unmittelbar folgt. Für die Durchführung des Beweises werden folgende Notationen benutzt: In der Initialisierungsphase werden die Ergebnisse der Schritte (1) und (2) durch S_0 und C_0 repräsentiert. Innerhalb der Schleife werden die einzelnen Werte in Abhängigkeit von der Schleifendurchlaufsnummer bezeichnet. Somit werden im k-ten Schleifendurchlauf die Resultate der Schritte (5), (6), (7), (8) als $S_k^{'}$, $C_k^{'}$, $S_k^{''}$, $C_k^{''}$, das Ergebnis des Schrittes (9) als S_k und C_k und das Resultat des Schrittes (10) als A_k bezeichnet.

Teil 1: Nach jedem Durchlauf der Schleife ist $S+C<9M$.

Beweis durch vollständige Induktion:

Vor der Schleife gilt:

$$S_0 + C_0 = 0 < 2M \tag{4.47}$$

Annahme: Nach dem k-ten Schleifendurchlauf gilt:

$$S_k + C_k < 9M \tag{4.48}$$

Zu zeigen: Nach dem k+1-ten Schleifendurchlauf gilt:

$$S_{k+1} + C_{k+1} < 9M \tag{4.49}$$

Beweis:

Nach den Schritten (5) und (6) gilt:

$$S_{k+1}^{'} < 2^n < 2M \quad \text{und} \quad C_{k+1}^{'} < 2^n < 2M \tag{4.50}$$

Nach den Schritten (7) und (8) sind

$$S_{k+1}^{''} = 2S_{k+1}^{'} < 4M \text{ und } C_{k+1}^{''} = 2C_{k+1}^{'} < 4M \tag{4.51}$$

Und nach Schritt (9) gilt:

$$S_{k+1} + C_{k+1} = S_{k+1}^{''} + C_{k+1}^{''} + A < 4M + 4M + M = 9M \tag{4.52}$$

Teil 2: Nach dem t-ten Durchlauf der Schleife gilt:

$$S_t + C_t \cong \sum_{i=n-t}^{n-1} x_i Y \cdot 2^{i-(n-t)} \bmod M \tag{4.53}$$

wobei t eine beliebige Zahl zwischen 1 und n ist.

Beweis durch vollständige Induktion:

Vor der Schleife gilt:

$$S_0 + C_0 = 0 \cong 0 \bmod M \tag{4.54}$$

Annahme: Nach dem *k*-ten Schleifendurchlauf gilt:

$$S_k + C_k \cong \sum_{i=n-k}^{n-1} x_i Y \cdot 2^{i-(n-k)} \bmod M \tag{4.55}$$

Zu zeigen: Nach dem *k*+1-ten Schleifendurchlauf gilt:

$$S_{k+1} + C_{k+1} \cong \sum_{i=n-k-1}^{n-1} x_i Y \cdot 2^{i-(n-(k+1))} \bmod M \tag{4.56}$$

Beweis:

In Schritt (10) des *k*-ten Schleifendurchlaufs wird der Wert A_k berechnet, der in Schritt (9) des *k*+1-ten Schleifendurchlauf bei der Carry-Save-Addition berücksichtigt wird:

$$A_k = LookUp(2 \cdot (s_n + 2c_{n+1} + c_n) + x_{n-(k+1)}) \tag{4.57}$$

Dabei ist S_k eine *n*+1-bit- und C_k eine *n*+2-bit-lange Zahl. Das heißt, dass das *n*-te Bit von S_k durch S_k div 2^n und die zwei höchstwertigsten Bits von C_k durch C_k div 2^n berechnet werden können. Somit gilt:

$$A_k = LookUp(2 \cdot (S_k \; div \, 2^n + C_k \; div \, 2^n) + x_{n-(k+1)}) \tag{4.58}$$

In der Gleichung (4.45) wurde bewiesen, dass

$$S_k \; div \, 2^n + C_k \; div \, 2^n = s_n + 2c_{n+1} + c_n < 4 \tag{4.59}$$

ist.

Die Lookup Table wurde so konstruiert (Gleichung (4.46)), dass die folgende Gleichung stimmt:

$$\begin{aligned} &LookUp(S_k \; div \, 2^n + C_k \; div \, 2^n) = LookUp(2(s_n + 2c_{n+1} + c_n) + x_{n-(k+1)}) \\ &= (2 \cdot (S_k \; div \, 2^n + C_k \; div \, 2^n) \cdot 2^n + x_{n-k-1} Y) \bmod M \end{aligned} \tag{4.60}$$

Nach Schritt (9) gilt:

$$
\begin{aligned}
S_{k+1} + C_{k+1} &\cong ((S_k \bmod 2^n + C_k \bmod 2^n) \cdot 2 \\
&+ ((2 \cdot (S_k \; div \, 2^n + C_k \; div \, 2^n) \cdot 2^n + x_{n-k-1}Y) \bmod M) \bmod M \\
&\cong (2 \cdot ((S_k \bmod 2^n + 2^n S_k \; div \, 2^n) + (C_k \bmod 2^n + 2^n C_k \; div \, 2^n)) \\
&+ x_{n-(k+1)}Y) \bmod M \cong (2 \cdot (S_k + C_k) + x_{n-(k+1)}Y) \bmod M \\
&\cong (2 \cdot \sum_{i=n-k}^{n-1} (x_i Y \cdot 2^{i-(n-k)}) \bmod M + x_{n-(k+1)}Y) \bmod M \\
&\cong (\sum_{i=n-k}^{n-1} (x_i Y \cdot 2^{i-(n-k-1)}) \bmod M + x_{n-(k+1)}Y) \bmod M \\
&\cong (\sum_{i=n-(k+1)}^{n-1} x_i Y \cdot 2^{i-(n-(k+1))}) \bmod M
\end{aligned}
\tag{4.61}
$$

Teil 3: Nach Schritt (20) ist $P = X \cdot Y \bmod M$:

In Schritt (12) wird $P=S+C<9M$ berechnet. P ist dabei kongruent zu $X \cdot Y \bmod M$. Es wird ein neues Register P' eingefügt. In Schritt (13) wird $P'=P-4M$ berechnet, wobei $4M$ durch das Schieben von M um zwei Bits nach links berechnet wird. In Schritt (14) wird P' mit 0 verglichen. Dies geschieht durch die Überprüfung des Vorzeichenbits. Ist P' positiv, so wird $P = P$' gesetzt und somit ist

$$P < 9M - 4M = 5M \tag{4.62}$$

Ist P' negativ, so ist

$$P < 4M < 5M \tag{4.63}$$

Mehrfache Subtraktion von M verletzt die Eigenschaft $P \cong X \cdot Y \bmod M$ nicht.

Die oben beschriebene Berechnung wird auf die gleiche Art und Weise fortgeführt: In Schritt (15) wird $P'=P-2M$ berechnet und danach in Schritt (16) P' wiederum mit 0 verglichen. Ist P' positiv, so wird $P = P$' gesetzt und es gilt:

$$P < 5M - 2M = 3M \tag{4.64}$$

Ist P negativ, so ist

$$P < 2M < 3M \tag{4.65}$$

In Schritt (17) wird $P' = P-M$ berechnet. Ist P' in Schritt (18) positiv, so wird $P = P$' gesetzt und damit trifft

$$P < 3M - M = 2M \tag{4.66}$$

zu. Ist dies nicht der Fall, so ist

$$P < M < 2M \tag{4.67}$$

Die Schritte (19) und (20) wiederholen die Schritte (17) und (18). Demzufolge gilt nach Schritt (20) folgende Aussage:

$$0 \leq P < M \tag{4.68}$$

Da P in den Schritten (13) bis (20) nur insofern verändert wurde, dass der Modulus M mehrmals abgezogen wurde, stimmt die Aussage

$$P \cong X \cdot Y \bmod M \tag{4.69}$$

weiterhin.

Demgemäß gilt nach der Schleife:

$$P = X \cdot Y \bmod M \tag{4.70}$$

Der Beweis ist erbracht.

Die Grundideen der letzten Abschnitte wurden in [18], [22] und [23] veröffentlicht.

4.2.4 Komplexitätsanalyse

In den vorherigen Abschnitten wurde sowohl die verschränkte Modularmultiplikation, als auch deren Optimierungen dargestellt. Die Anzahl der für einen Schleifendurchlauf benötigten Operationen wurde zuerst auf nur eine konventionelle Addition minimiert und diese wiederum durch eine redundante Carry-Save-Addition ersetzt. Demzufolge bietet der Algorithmus 4.2.7 eine effiziente Lösung für die Modularmultiplikation.

In diesem Abschnitt wird eine Komplexitätsanalyse dieser Lösung durchgeführt.

Zuerst soll hier die Funktionsweise des Algorithmus 4.2.7 dargestellt werden. Der Algorithmus besteht aus drei Teilen: Vorberechnungsphase (die Berechnung der Werte für die Lookup Table und Schritte (1) bis (3)), Schleife (Schritte (4) bis (11)) und Abschlussphase (Schritte (12) bis (20)). Als erstes wird die Schleife analysiert.

Die Schleife besteht aus n Durchläufen. In jedem Durchlauf werden S und C modulo 2^n reduziert und danach verdoppelt (Schritte (5), (6) und (7), (8) entsprechend). Diese Operationen sind in einer Hardwarerealisierung nichts anderes als das Routing der entsprechenden Leitungen und benötigen damit weder zusätzlich Zeit noch Fläche. Schritt

(9) dagegen ist eine Carry-Save-Addition, die die Zeitkomplexität eines Volladdierers und somit, wie in Abschnitt 3.3.2 definiert wurde, eine Zeiteinheit benötigt. Schritt (10) ist das Berechnen der Adresse und die Adressierung der Lookup Table. Sowohl die Berechnung der Adresse, als auch die Adressierung benötigen Zeit. Allerdings kann erstere parallel zur Carry-Save-Addition stattfinden. Dies wird möglich, da die Adresse eine Summe der zwei höchstwertigsten Bits von C c_{n+1}, c_n und dem höchstwertigsten Bit von S s_n ist. Diese werden als Carry-Save-Summe zweier n-1-bit Zahlen $S^{\text{vor CSA}}$, $C^{\text{vor CSA}}$ und einer n-bit Zahl A gebildet. Da bei der Carry-Save-Addition gilt

$$\begin{aligned} 2c_{n+1} + s_n &= s_n^{vorCSA} + c_n^{vorCSA} \\ c_n &= (s_{n-1}^{vorCSA} + c_{n-1}^{vorCSA})\ div\ 2 \end{aligned} \tag{4.71}$$

Somit ist die Adresse gleich

$$\begin{aligned} Adresse &= C\ div\ 2^n + S\ div\ 2^n = 2c_{n+1} + s_n + c_n \\ &= s_n^{vorCSA} + c_n^{vorCSA} + (s_{n-1}^{vorCSA} + c_{n-1}^{vorCSA})\ div\ 2 \end{aligned} \tag{4.72}$$

Also kann die Adresse nur aus den Werten, die vor der Carry-Save-Addition vorhanden sind parallel zur Carry-Save-Addition berechnet werden. Demgemäß verbleibt nur noch die Zeit des Zugriffs auf die Lookup Table, welche per Definition eine Zeiteinheit beträgt. Demzufolge liegt die Zeitkomplexität eines Schleifendurchlaufs bei zwei Zeiteinheiten und dementsprechend benötigt die Gesamtschleife

$$T_{\text{Schleife}} = 2n \tag{4.73}$$

Zeiteinheiten.

In der Vorberechnungsphase werden acht Werte berechnet:

```
LookUp(7) = (2*3*2^n + Y) mod M; LookUp(6) = 2*3*2^n mod M;
LookUp(5) = (2*2*2^n + Y) mod M; LookUp(4) = 2*2*2^n mod M;
LookUp(3) = (2*1*2^n + Y) mod M; LookUp(2) = 2*1*2^n mod M;
LookUp(1) = Y; LookUp(0) = 0;
```

Zwei Werte: *LookUp*(1) = *Y* und *LookUp*(0) = 0 sind bereits vorhanden. Die anderen können wie folgt berechnet werden:

```
(1)  LookUp(0) = 0
(2)  LookUp(1) = Y
(3)  S = 2^n mod M;
(4)  S = 2*S;
(5)  C = S – M;
(6)  if (C>0) S = C;
(7)  LookUp(2) = S;
(8)  S = S + Y;
(9)  C = S – M;
(10) if (C>0) S = C;
(11) LookUp(3) = S;
(12) S = LookUp(2);
(13) S = 2*S;
(14) C = S – M;
(15) if (C>0) S = C;
(16) LookUp(4) = S;
(17) S = S + Y;
(18) C = S – M;
(19) if (C>0) S = C;
(20) LookUp(5) = S;
(21) S = LookUp(4);
(22) C = Lookup(2);
(23) S = S + C;
(24) C = S – M;
(25) if (C>0) S = C;
(26) LookUp(6) = S;
(27) S = S + Y;
(28) C = S – M;
(29) if (C>0) S = C;
(30) LookUp(7) = S;
```

Die Operation in (3) ist das einfache Invertieren aller Bits und das Aufaddieren einer Eins auf das Ergebnis. Wäre das niederwertigste Bit von *M* eine Null, so würde das Aufaddieren einer Eins auf die Inverse eine Übertragsbildung mit sich ziehen, da aber *M* immer eine ungerade Zahl ist, kann kein Übertrag entstehen, somit braucht die Zeit, die für diese Operation benötigt wird, nicht berücksichtigt werden. Die Schritte (4) und (13) stellen nur einfache Schiebeoperationen dar. Bei den Schritten (6), (10), (15), (19), (25) und (29) handelt es sich um die Untersuchung eines Bits mit nachfolgender Zuweisung eines Wertes. Die Schritte (1), (2), (7), (11), (12), (16), (20), (21), (22), (26) und (30) sind das Lesen oder das Schreiben in bzw. aus der Lookup Table. Solche Schritte erfordern eine Zeiteinheit pro Schritt. Insgesamt sind es elf Zeiteinheiten. Die restlichen zehn Schritte: (5), (8), (9), (14), (17), (18), (23), (24), (27), (28) sind nichtredundante Additionen. Alle nichtredundanten Additionen werden mit dem in Abschnitt 3.3.6 vorgestellten Kombiaddierer durchgeführt. Dabei bildet der Kombiaddierer eine eigene Hardwareschaltung, die die schon vorhandenen Hardwareelemente, wie z.B. einzelne

Volladdierer des Carry-Save-Addierers nicht verwendet, denn eine solche Nutzung würde die Steuerung der Gesamtschaltung signifikant verkomplizieren, was wiederum zu einem höheren Zeitaufwand führen würde. Demzufolge liegt die Zeitkomplexität der Vorberechnungsphase bei

$$T_{\text{Vorberechnungsphase}} = 10 \cdot (8 \log n - 16) + 11 = 80 \log n - 149 \tag{4.74}$$

Zeiteinheiten.

Die Abschlussphase erfordert fünf vollständige Additionen (Algorithmus 4.2.7 Schritte (12), (13), (15), (17) und (19)). Die Zeitkomplexität der Abschlussphase liegt damit bei

$$T_{\text{Abschlussphase}} = 5 \cdot (8 \log n - 16) = 40 \log n - 80 \tag{4.75}$$

Zeiteinheiten.

Demgemäß beträgt die Zeitkomplexität des Algorithmus 4.2.7 insgesamt

$$\begin{aligned} & T_{\text{verschränkte Mod. Mult.}} \\ & = 2n + 80 \log n - 149 + 40 \log n - 80 = 2n + 120 \log n - 229 \end{aligned} \tag{4.76}$$

Zeiteinheiten.

Nachdem die Zeitkomplexität ermittelt wurde, wird die Flächenkomplexität berechnet:

Die Schleife erfordert einen Carry-Save-Addierer, eine Lookup Table mit acht *n*-bit Werten und drei Register *S*, *C* und *X*. Es werden weder Register für *Y* noch für *M* gebraucht, da die entsprechenden Werte in der Lookup Table gespeichert sind. Die Vorberechnungsphase benötigt einen Kombiaddierer und vier Register *Y*, *M*, *S* und *C*, wobei es sich bei *S* und *C* um dieselben Register handeln kann, die auch in der Schleife benutzt werden. Die Abschlussphase erfordert einen Kombiaddierer, dabei kann der Kombiaddierer aus der Vorberechnungsphase verwendet werden, sowie die drei Register *P*, *P*' und *M*. Die Register *P* und *P*' können durch die schon in der Schleife eingesetzten Register *S* und *C* ersetzt werden.

Somit sind für den Algorithmus 4.2.7 insgesamt ein Carry-Save-Addierer, ein Kombiaddierer, eine Lookup Table für acht *n*-bit Worte und fünf Register *X*, *Y*, *M*, *S* und *C* erforderlich.

Folglich ist die Flächenkomplexität des Algorithmus 4.2.7 gleich

$$A_{\text{verschränkte Mod. Mult.}} = 8n + 6n + 8n + 5 \cdot 4n = 42n \tag{4.77}$$

Flächeneinheiten.

Die Flächen-Zeit-Komplexität liegt damit bei

$$\begin{aligned} &AT_{\text{verschränkte Mod. Mult.}} \\ &= 42n\cdot(2n + 120\log n - 229) = 84n^2 + 5040n\log n - 9618n \end{aligned} \tag{4.78}$$

Flächen-Zeit-Einheiten.

Die folgende Tabelle präsentiert die Komplexität der verschränkten Modularmultiplikation für unterschiedliche Wortlängen

n	Flächenkomplexität/ Flächeneinheit	Zeitkomplexität/ Zeiteinheit	AT-Komplexität/ Flächen-Zeit-Einheiten
512	21.504	1.875	40.320.000
1024	43.008	3.019	129.841.152
2048	86.016	5.187	446.164.992
4096	172.032	9.403	1.617.616.896
8192	344.064	17.715	6.095.093.760

Tabelle 4-2: Komplexität der verschränkten Modularmultiplikation

4.2.5 Zusammenfassung

In diesem Abschnitt wurde einer der am meisten benutzten Algorithmen für die Modularmultiplikation, die verschränkte Modularmultiplikation, dargestellt. Es wurden mehrere Verbesserungsvorschläge präsentiert. Die Kulmination der Verbesserungen ist der Algorithmus 4.2.7. Die vollständigen Vergleiche wurden durch approximative Vergleiche ersetzt, die unabhängig von der Wortlänge in konstanter Zeit durchgeführt werden können. Durch die Benutzung einer Lookup Table mit vorberechneten Werten wurde die Anzahl der Additionen bis auf eine Addition pro Schleifendurchlauf reduziert. Und die so verbliebene Addition wurde durch eine redundante Carry-Save-Addition ersetzt. Alle diese Verbesserungen brachten eine Performancesteigerung, sowie mehrfache Flächenreduzierungen. Durch das im Abschnitt 3.3 definierte Komplexitätsmodell wurde eine Analyse des neuen Algorithmus durchgeführt und dessen Ergebnisse dargelegt.

4.3 Modularmultiplikation basierend auf der Veränderung eines Operanden

In den letzten zwei Abschnitten wurden die zwei am häufigsten benutzten Algorithmen für die Modularmultiplikation und deren Verbesserungen dargestellt. Bei beiden Algorithmen wird ein Operand bitweise vom höchstwertigsten zum niederwertigsten Bit (Kapitel 4.2) oder vom niederwertigsten zum höchstwertigsten Bit (Kapitel 4.1) bearbeitet. Der zweite Operand bleibt dabei unverändert.

In diesem Kapitel wird ein neuer Algorithmus für die Modularmultiplikation vorgestellt. Dieser Algorithmus greift auf die in den vorherigen Abschnitten beschriebenen Techniken, wie Vorberechnung und Lookup Tables zurück. Der Vorteil gegenüber den oben beschriebenen Algorithmen liegt in der Tatsache, dass alle vorberechneten Werte nur vom Modulus abhängig sind. Somit muss die Vorberechnung nur einmal für mehrere Modularmultiplikationen mit dem gleichen Modulus stattfinden. Dies kann vor allem bei der Modularen Exponentiation genutzt werden, da diese die Hintereinanderausführung mehrerer Modularmultiplikationen mit dem gleichen Modulus erfordert [50]. Die Besonderheit des neuen Algorithmus ist die Idee der schrittweisen Veränderung eines Operanden während der Modularmultiplikation. Der andere Operand wird bitweise vom niederwertigsten zum höchstwertigsten Bit (wie bei Montgomery 4.1) bearbeitet.

In den folgenden Abschnitten wird der neue Algorithmus dargestellt und verschiedene Optimierungen erarbeitet.

4.3.1 Darstellung des neuen Algorithmus

In diesem Algorithmus wird der erste Operand *X* bitweise bearbeitet. Der zweite Operand *Y* wird in jedem Schritt verändert. Diese Idee basiert auf der folgenden mathematischen Gleichung:

$$\begin{aligned} X \cdot Y \bmod M &= \sum_{i=0}^{n-1} (2^i \cdot \mathrm{x_i} \cdot Y) \bmod M = \sum_{i=0}^{n-1} (x_i \cdot (2^i \cdot Y) \bmod M) \bmod M \\ &= \sum_{i=0}^{n-1} (x_i \cdot \mathrm{Y_i}) \bmod M \end{aligned} \tag{4.79}$$

wobei

$$Y_i = 2^i \cdot Y \bmod M = [2 \cdot (2^{i-1} \cdot Y \bmod M)] \bmod M = [2 \cdot Y_{i-1}] \bmod M \tag{4.80}$$

Daraus kann das folgende rekursive Gleichungssystem abgeleitet werden

$$\begin{aligned} X \cdot Y \bmod M &= \sum_{i=0}^{n-1} (x_i \cdot Y_i) \bmod M, \\ Y_i &= 2 * Y_{i-1} \bmod M, \\ Y_o &= Y. \end{aligned} \tag{4.81}$$

Es wird festgelegt, dass

$$I_i = x_i \cdot Y_i \tag{4.82}$$

und

$$P = X \cdot Y \bmod M = \sum_{i=0}^{n-1} I_i \bmod M \tag{4.83}$$

Der Algorithmus berechnet $Y_i=2 \cdot Y_{i-1}$ mod M und $I_i=x_i \cdot Y_i$ in jedem Durchlauf und addiert I_i zu der Zwischensumme P. Anschließend wird die Zwischensumme entsprechend zum Modulus M reduziert. Somit bleibt P immer kleiner als M und folglich $P=X \cdot Y$ mod M.

Der veränderbare Operand Y_i kann durch

$Y_i=2 \cdot Y_{i-1}$;

`if` $(Y_i \geq M)$ $Y_i=Y_i - M$;

berechnet werden.

Beweis: $Y_0 < M$

Sei $Y_k < M$. Nach der Verdopplung gilt: $2Y_k < 2M$. Es werden zwei Fälle unterschieden: ist $2Y_k < M$, so ist $Y_{k+1} = 2Y_k$; ist $M \leq 2Y_k < 2M$, so wird $Y_{k+1} = 2Y_k - M < 2M - M = M$ berechnet. In beiden Fällen ist $Y_{k+1} < M$.

Die Zwischensumme P_i kann folgendermaßen berechnet werden:

$P_i=P_{i-1}+x_i \cdot Y_i$;

`if` $(P_i \geq M)$ $P_i = P_i - M$;

Der folgende Pseudocode zeigt die einfache Implementierung des neuen Algorithmus.

Algorithmus 4.3.1: Grundvariante des neuen Verfahrens

```
Eingabe: X, Y<M<2^n, wobei 2^(n-1)<M<2^n.
Ausgabe: P = X*Y mod M.
Methode:
(1)  P=0;
(2)  for (int i=0; i<n; i++) {
(3)    P=P+x_i*Y;
(4)    if (P≥M) P=P-M;
(5)    Y=2*Y;
(6)    if (Y≥M) Y=Y-M;
(7)  }
```

Algorithmus 4.3.1 ist eine einfache und effiziente Methode für die Modularmultiplikation. Im folgenden Abschnitt werden verschiedene Optimierungen des neuen Algorithmus in Bezug auf die Zeit- und Flächenkomplexität erörtert.

4.3.2 Optimierungsvorschläge

In Schritt (3) des Algorithmus 4.3.1 wird x_i mit Y multipliziert. Die Zeit- sowie die Flächenkomplexität dieser Berechnungen ist niedrig, da es sich nur um eine Parallelschaltung von „AND-Gattern“ handelt. In Schritt (5) wird der Wert von Y verdoppelt. Da diese Operation nur aus der Verschiebung um ein Bit besteht, sind sowohl die Zeit- als auch die Flächenkomplexität, die für diese Operation benötigt werden, unbedeutend gering. Die verbliebene Operation in Schritt (3) ist eine Addition. In den Schritten (4) und (6) wird ein Vergleich mit anschließender Subtraktion durchgeführt. Eine Addition, eine Subtraktion und ein Vergleich haben ähnliche Zeit- und Flächenkomplexität. Die Zeitkomplexität einer jeden solchen Operation wird als Zeitkomplexität einer Addition festgesetzt. Die Zeitkomplexität eines Schleifendurchlaufs des Algorithmus 4.3.1 liegt entsprechend bei fünf Additionen. Im Folgenden soll der Algorithmus 4.3.1 beschleunigt und gleichzeitig die dazugehörige Fläche minimiert werden.

1. Optimierungsansatz: Die Berechnung von P und Y in den Schritten (3), (4) und (5), (6) kann gepipelined werden. Dadurch verringert sich die Zeitkomplexität auf bis zu drei Additionen pro Schleifendurchlauf, allerdings erhöht sich die Flächenkomplexität entsprechend.

2. Optimierungsansatz: Der Vergleich mit M ist langsam, da im schlechtesten Fall alle Bits von M untersucht werden müssen. Um dieses Problem zu umgehen, wird der Algorithmus so modifiziert, dass P und Y mit 2^n und nicht mit M verglichen werden.

Somit wird nur ein Bit untersucht: `if` $P \geq 2^n$ `then` $P=P-M$; ähnlich für Y: `if` $Y \geq 2^n$ `then` $Y=Y-M$.

3. Optimierungsansatz: Die Subtraktion verbraucht zusätzliche Hardware. Es gibt aber eine Möglichkeit, die Subtraktion durch eine Addition zu ersetzen. Um dies zu erreichen, wird $K=2^n-M$ berechnet und gespeichert. P und Y werden dann auf folgende Art und Weise berechnet: $P=P-2^n+(2^n-M)$ und $Y=Y-2^n+(2^n-M)$.

Der folgende Pseudocode zeigt die Implementierung der oben beschriebenen Optimierungen.

Algorithmus 4.3.2: Erste Optimierung des neuen Algorithmus

```
Eingabe: X, Y<M<2^n, wobei 2^(n-1)<M<2^n.
Ausgabe: P = X*Y mod M.
Methode:
(1)  P=0;
(2)  K=2^n-M;
(3)  for (int i=0; i<n; i++) {
(4)    P=P+x_i*Y;
(5)    if (P≥2^n) P=P-2^n+K;
(6)    if (P≥2^n) P=P-2^n+K;
(7)    Y=2*Y;
(8)    if (Y≥2^n) Y=Y-2^n+K;
(9)    if (Y≥2^n) Y=Y-2^n+K; }
(10) if (P≥M) P=P-M;
```

Nach der Verdopplung von Y in Schritt (7) gilt $Y<2\cdot 2^n=4\cdot 2^{n-1}<4M$. Um Y wieder in das Intervall zwischen 0 und 2^n zurück zu führen, müssen bis zu $2M$ subtrahiert werden ((8), (9)).

Ein Schleifendurchlauf erfordert nun fünf Additionen in den Schritten (4), (5), (6), (8) und (9). Der Block mit den Schritten (4), (5), (6) und der Block mit den Schritten (8), (9) können gepipelined werden. Demzufolge entspricht die Zeitkomplexität eines Schleifendurchlaufs dem Zeitaufwand von drei Additionen, und die der gesamten Schleife $3n$ Additionen.

Schleifeninvariante: Das Zwischenergebnis ist $P<2^n$.

Beweis: Am Anfang der Schleife ist $P=0<2^n$. In jedem Schleifendurchlauf wird $x_i \cdot Y<2^n$ addiert und dann bis zu $2M>2^n$ abgezogen. Somit bleibt $P<2^n+2^n-2M<2^n$.

Demzufolge ist das Ergebnis der Schleife $P<2^n$.

Da $2^{n-1}<M<2^n$, benötigt der Algorithmus 4.3.2 einen zusätzlichen Vergleich mit nachfolgenden Subtraktionen (Schritt (10)), um das Ergebnis kleiner als M zu halten.

Die Abbildung 4.3.1 zeigt den Datenfluss eines Schleifendurchlaufs des Algorithmus 4.3.2:

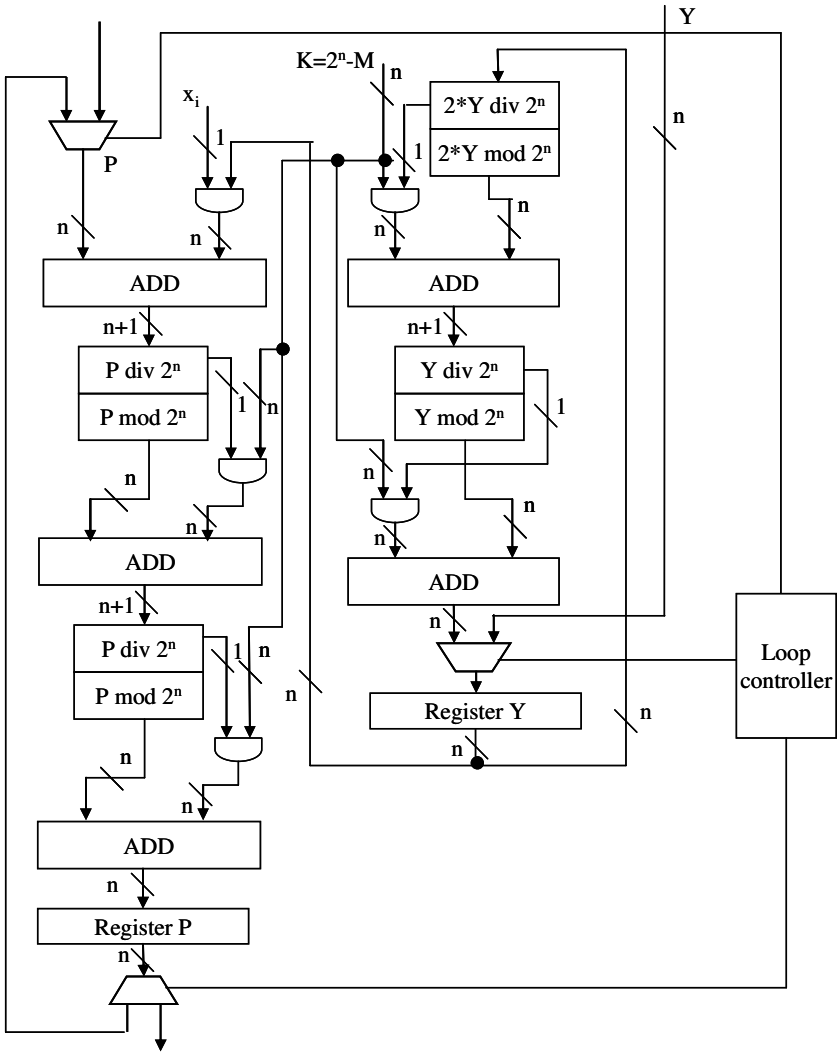

Abbildung 4.3.1: Ein Schleifendurchlauf des Algorithmus 4.3.2

Die Abbildung 4.3.1 zeigt einen Schleifendurchlauf des Algorithmus 4.3.2. Der Block aus den Schritten (4), (5), (6) kann mit dem Block aus den Schritten (7), (8), (9) eine Pipeline bilden: Der linke Teil und der rechte Teil der Abbildung können gleichzeitig schalten und somit die Berechnungszeit verkürzen.

4. Optimierungsansatz: In jedem Schleifendurchlauf werden fünf Additionen durchgeführt. An dieser Stelle wird die Anzahl der Additionen auf drei reduziert. Nach der Verdopplung wird Y so reduziert, dass $Y<2^n+M$, anstelle von $Y<2^n$ ist. Daraus folgt, dass $2Y<2^{n+1}+2M<2^{n+2}$. Diese Tatsache bedeutet, dass es vier unterschiedliche Möglichkeiten

für die zwei höchstwertigsten Bits von Y gibt: 00, 01, 10 und 11. Die Werte K_0=0, K_1=2^n mod M, K_2=$2 \cdot 2^n$ mod M und K_3=$3 \cdot 2^n$ mod M können vorberechnet und gespeichert werden. Nach der Verdoppelung wird Y durch Y mod 2^n + K_{TMSB} ersetzt, wobei TMSB für die zwei höchstwertigsten Bits von Y steht. Der Pseudocode dieses Teils lautet:

```
(6) Y=2*Y;
(7) case y_{n+1}y:
(8)   00: Y=Y;
(9)   01: Y=Y-2^n+K_1;
(10)  10: Y=Y-2*2^n+K_2;
(11)  11: Y=Y-3*2^n+K_3;
(12) end case;
```

Das Ergebnis P muss, analog zu Y, kleiner als 2^{n+2} sein. $x_i \cdot Y < 2^{n+1}$, damit ist $P + x_i \cdot Y < 2^{n+1} + 2^{n+1} = 2^{n+2}$. Daher können die zwei höchstwertigsten Bits von P ähnlich wie bei Y vier unterschiedliche Werte besitzen: 00, 01, 10, 11. Demzufolge kann P durch die vorberechneten Werte K_0=0, K_1=2^n mod M, K_2=$2 \cdot 2^n$ mod M und K_3=$3 \cdot 2^n$ mod M reduziert werden. Das Ergebnis dieser Reduktion ist kleiner als $2^n + K_j$, wobei der dazugehörige vorberechnete Wert ist. Diese sind allerdings kleiner als M. Somit ist das Ergebnis $P < 2^n + M$.

Da $2^{n-1} < M < 2^n$, werden zwei zusätzliche Vergleiche mit nachfolgenden Subtraktionen benötigt, um das Ergebnis kleiner als M zu halten. Da $M > 2^{n-1}$ ist, gilt: $2M > 2^n$; der erste Vergleich `if (`$P \geq M$`) P=P-M` wird durch `if (`$P \geq 2^n$`) P=P-`2^n`+K` ersetzt. Dabei wird genau diese Operation in der Schleife bei der Reduktion von P durchgeführt. Dies kann ausgenutzt werden, indem der Operand X um ein zusätzliches Bit x_n=0 und die Schleife um einen zusätzlichen Durchlauf erweitert werden. Dieses Vorgehen führt dazu, dass in einem zusätzlichen Schleifendurchlauf $x_n Y$=0 ist. Da P nach dem n-ten Schleifendurchlauf kleiner als $2^n + M$ ist und in dem weiteren Durchlauf nichts hinzuaddiert wird und sogar eine zusätzliche Reduktion von P stattfindet, ist P nach der Schleife kleiner als 2^n und ein Vergleich mit M mit abschließender Reduktion reicht dafür aus, dass das Endergebnis kleiner als M wird. Der folgende Pseudocode veranschaulicht die oben beschriebenen Überlegungen:

Algorithmus 4.3.3: Optimierte Version der Modularmultiplikation basierend auf der Veränderung eines Operanden

```
Eingabe: X, Y<M<2^n, wobei 2^(n-1)<M<2^n and x_n=0;
Ausgabe: P = X*Y mod M.
Methode:
(1)  P=0;
(2)  K_1=2^n mod M; K_2=2*2^n mod M; K_3=3*2^n mod M;
(3)  for (int i=0; i<n+1; i++) {
(4)    P=P+x_i*Y;
(5)    case p_(n+1)p_n:
(6)      00: P=P;
(7)      01: P=P-2^n+K_1;
(8)      10: P=P-2^(n+1)+K_2;
(9)      11: P=P-3*2^n+K_3;
(10)   end case;
(11)   Y=2*Y;
(12)   case y_(n+1)y_n:
(13)     00: Y=Y;
(14)     01: Y=Y-2^n+K_1;
(15)     10: Y=Y-2^(n+1)+K_2;
(16)     11: Y=Y-3*2^n+K_3;
(17)   end case;
(18) }
(19) if (P≥M) P=P-M;
```

Der Block aus den Schritten (4) bis (10) kann mit den Schritten (11) bis (17) eine Pipeline bilden. Nun werden n+1 Schleifendurchläufe mit jeweils drei Additionen benötigt. Die Zeitkomplexität des Algorithmus 4.3.4 ist hauptsächlich durch den Zeitaufwand der Additionen bestimmt.

4.3.3 Unvollständige Modularmultiplikation

Ein großer Teil des Aufwandes wird für die Reduktion der Zwischenergebnisse getrieben. In diesem Abschnitt soll diskutiert werden, ob und wie der Gesamtaufwand durch Verzicht auf diese Operation verbessert werden kann. Das Zwischenergebnis P wird nicht mehr wie in früheren Algorithmen reduziert. Daher ist P am Ende der Berechnung die Summe aller Teilprodukte $x_i \cdot Y$. Dabei ist jedes Teilprodukt $x_i \cdot Y < 2 \cdot 2^n$. Demzufolge gilt für das Endergebnis folgende Aussage: $P < n \cdot 2 \cdot 2^n = 2n \cdot 2^n$. Folglich ist P eine Zahl, die maximal aus $n+\log(2n)=n+\log n+1$ Bits besteht. P kann als $P=(P \bmod 2^n) + 2^n \cdot (P \operatorname{div} 2^n)$ dargestellt werden. Es wird $P \bmod M = [(P \operatorname{div} 2^n) \cdot 2^n + (P \bmod 2^n)] \bmod M$ berechnet. Dieses Ergebnis kann also durch eine neue Modularmultiplikation berechnet werden, für die gilt: $X' = (P \operatorname{div} 2^n) \cdot 2^n$ und $Y' = 2^n \bmod M$, wobei X' eine $\log n+1$-bit-lange Zahl und nicht mehr eine n-bit-lange Zahl ist. $[(P \operatorname{div} 2^n) \cdot 2^n + (P \bmod 2^n)] \bmod M$ kann wiederum mit dem neuen Algorithmus berechnet werden, wobei $P_{\text{Anfang}} = P \bmod 2^n$. Da-

bei werden log$n$+1 Additionen benötigt, und das Ergebnis ist kleiner als (log$n$+2)·2^n. Demzufolge ist das Ergebnis eine log[(log$n$+2)·2^n]=n + log(log$n$+2)-bit-lange Zahl. Eine solche Berechnung kann mehrmals wiederholt werden, bis das Ergebnis kleiner als 2^n ist.

Der nachfolgende Pseudocode zeigt die Funktionsweise der unvollständigen Modularmultiplikation:

Algorithmus 4.3.4: Unvollständige Modularmultiplikation

```
Eingabe: X, Y<M<2^n, wobei 2^(n-1)<M<2^n and x_n=0;
Ausgabe: P = X*Y mod M.
Methode:
(1) P=0;
(2)  K_1=2^n mod M; K_2=2*2^n mod M; K_3=3*2^n mod M;
(3)  while (X≠0) {
(4)   for (int i=0; i<n+1; i++) {
(5)    P=P+x_i*Y;
(6)    Y=2*Y;
(7)    case y_(n+1)y_n:
(8)      00: Y=Y;
(9)      01: Y=Y-2^n+K_1;
(10)     10: Y=Y-2^(n+1)+K_2;
(11)     11: Y=Y-3*2^n+K_3;
(12)   end case;
(13)  }
(14)  X=P div 2^n;
(15)  Y=K_1;
(16)  n=log(n)+2;
(17)  P=P mod 2^n;
(18) }
(19) if (P≥M) P=P-M;
```

Die Abbildung 4.3.2 zeigt das Datenflussdiagram eines Schleifendurchlaufs der inneren Schleife des Algorithmus 4.3.4:

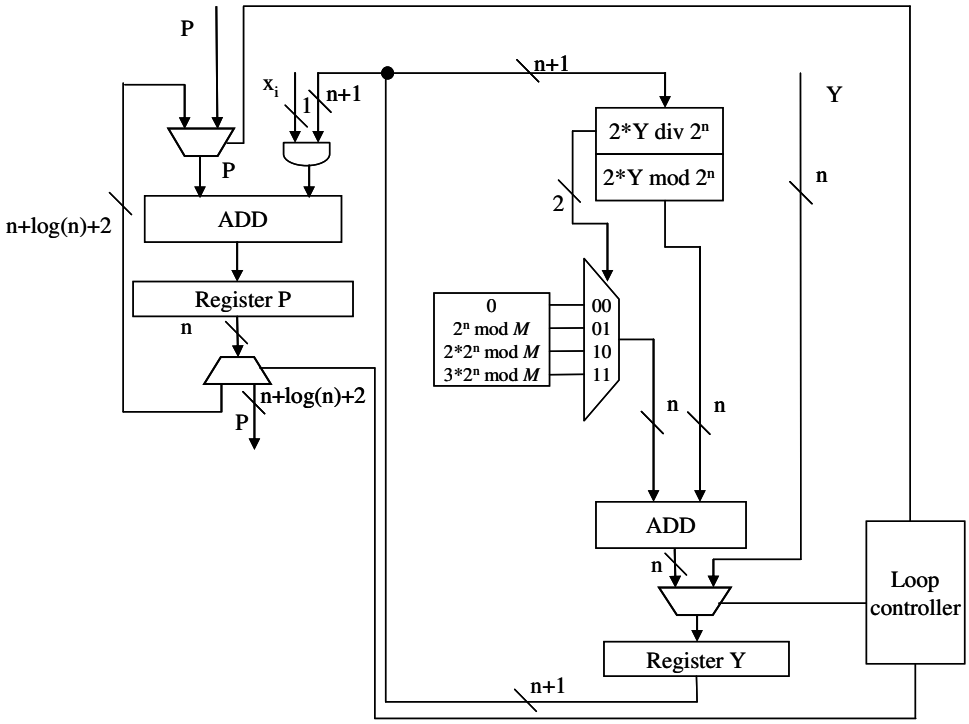

Abbildung 4.3.2: Ein Schleifendurchlauf des Algorithmus 4.3.4

Algorithmus 4.3.4 erfordert n + log$n$ + 1 + log(log$n$ + 2) + log(log(log$n$+2)+1) + log(log(log(log$n$+2)+1)+1) ... < n +2(log$n$ + 2) = n +2log$n$ + 4 Schleifendurchläufe. Da für die oben beschriebene Formel kein exakter Lösungsweg existiert, wurde eine obere Schranke als Abschätzung gewählt. Die Rekursion terminiert, da immer wenn x_i>0 ist, das Zwischenergebnis S mindestens um M reduziert wird. Schritt (5) und der Block aus den Schritten (6) bis (12) können eine Pipeline bilden. Die Zeitkomplexität des neuen Algorithmus ist kleiner als der Zeitaufwand von n + 2log$n$ + 5 Additionen.

4.3.4 Vollständige Modularmultiplikation

Im letzten Abschnitt wurde ein Weg aufgezeigt, wie die Schaltung bis auf zwei Addierer mit einer Zeitkomplexität von n+log$n$ reduziert werden kann. In diesem Abschnitt wird der neue Algorithmus auf eine Zeitkomplexität von n verbessert. Der Grundgedanke ist hierbei einfach: Im Abschnitt 4.3.3 wurde P als eine Summe von positiven Y berechnet. In diesem Abschnitt werden Y und Y-3M parallel berechnet, denn Y-3M<0. Ist die Zwischensumme P positiv, so wird Y-3M zu P addiert. Demgemäß wird P verkleinert. Falls P negativ ist, wird Y aufaddiert und dadurch P vergrößert.

In jedem Schritt wird die Summe P mit 0 verglichen: `if` $P < 0$ `then` $P = P + Y$; `else` $P = P + Y - 3M$. Dafür wird Y nach der Reduktion in das Register $Y1$ gespeichert, darüber hinaus wird Y-3M berechnet und in das Register $Y2$ gespeichert. Dies geschieht parallel zueinander.

Pseudocode dieser Variante:

Algorithmus 4.3.5: Vollständige Modularmultiplikation

```
Eingabe: X, Y<M<2^n, wobei 2^(n-1)<M<2^n and x_n=0;
Ausgabe: P = X*Y mod M.
Methode:
(1)  P=0; Y=0; Y1=0; Y2=0;
(2)  K_0=0; K_1=2^n mod M; K_2=2*2^n mod M; K_3=3*2^n mod M;
(3)  for (int i=0; i<n+1; i++) {
(4)    P=P+Y2;
(5)    if (P<0) Y2=Y*x_i;
(6)    else Y2=(Y-3M)*x_i;
(7)    Y=2*Y;
(8)    case y_(n+1)y_n:
(9)      00: Y=Y+K_0;
(10)     01: Y=Y-2^n+K_1;
(11)     10: Y=Y-2*2^n+K_2;
(12)     11: Y=Y-3*2^n+K_3;
(13)   end case;
(14) }
(15) if (P<0) P=P+2M;
(16) if (P<0) P=P+2M;
(17) if (P>M) P=P-M;
```

Schritt (4) des Algorithmus 4.3.5, der Block aus den Schritten (5) bis (6) und der Block aus den Schritten (7) bis (13) können eine Pipeline bilden. Der längste Datenpfad der Schleife besteht aus einer Addition und einem Zugriff auf die Lookup Table. Da es sich allerdings bei der Addition um eine nichtredundante Addition handelt, deren Zeitkomplexität relativ hoch ist, ist die Zeitkomplexität des Zugriffs auf die Lookup Table relativ gering in Relation zur Addition. Aus diesem Grund wird der Zeitaufwand eines Schleifendurchlaufs mit der Zeitkomplexität einer Addition abgeschätzt. Die Zeitkomplexität des Algorithmus liegt bei dem Zeitaufwand von n Additionen. Da drei Additionen parallel zueinander ausgeführt werden, entspricht die Flächenkomplexität dem Flächenaufwand von drei Addierern.

Die Abbildung 4.3.3 zeigt das Datenflussdiagramm eines Schleifendurchlaufs des Algorithmus 4.3.5:

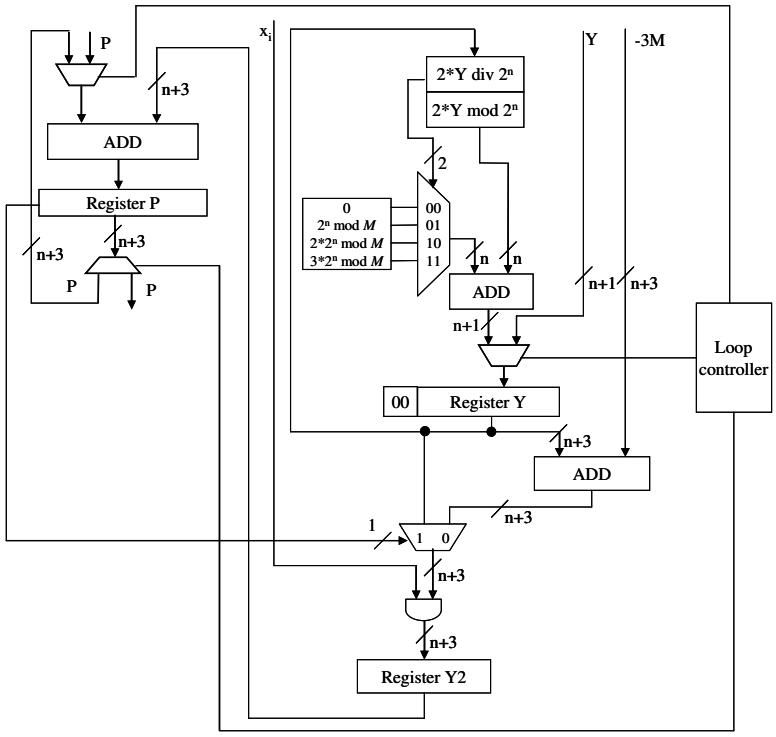

Abbildung 4.3.3: Ein Schleifendurchlauf des Algorithmus 4.3.5

Ein Vorteil des neuen Algorithmus ist die Verbesserung der Zeitkomplexität von n+log$n$ auf n Additionen. Leider wird dies auf Kosten des Flächenaufwandes erreicht, der durch den Einsatz eines zusätzlichen Addierers und eines Registers ansteigt.

4.3.5 Modularmultiplikation mit dem redundanten Addierer

In diesem Kapitel wurden drei Verfahren für die Modularmultiplikation dargestellt. Die ersten beiden wurden vor mehr als zwanzig Jahren erfunden und in dieser Arbeit lediglich optimiert. Ihre Performance wurde um ein Vielfaches gesteigert und die Flächenkomplexität mehrfach reduziert. Beim dritten Verfahren, handelt es sich um ein neues Verfahren, dessen Idee so offensichtlich ist, dass viele Wissenschaftler, die im Bereich der Modularmultiplikation arbeiteten, vermutlich bereits über ein ähnliches Verfahren nachgedacht haben. Allerdings wurden die Vorteile, die es mit sich bringt, nicht gesehen. Das Verfahren ist vor allem theoretisch interessant. Aus diesem Grund wurden bei der Optimierung mehrere Verbesserungen vorgeschlagen, die teilweise in unterschiedliche Richtungen gehen. Denn diese Optimierungen sollen nicht als eine endgültige Lösung, sondern als Ansätze für weitere Verbesserungen dienen. Um das

Verfahren mit dem oben beschriebenen Algorithmus 4.1.3 und dem Algorithmus 4.2.7 vergleichen zu können, wird eine Optimierung der Modularmultiplikation basierend auf der Veränderung eines Operanden unter der Benutzung zuvor entwickelter Optimierungstechniken vorgestellt. Dabei wird der nichtredundante Addierer durch einen redundanten Carry-Save-Addierer ersetzt. Der Algorithmus für die vollständige Modularmultiplikation (Algorithmus 4.3.5) benutzt den Vergleich mit Null, welcher aber bei der Carry Save Zahlenrepräsentation nur mit hohem Zeitaufwand durchführbar ist. Ein approximativer Vergleich wie z.B. bei [42] ist jedoch nicht anstrebenswert. Aus diesen Gründen wird die unvollständige Modularmultiplikation (Algorithmus 4.3.4) als Basismodell für eine neue Verbesserung ausgewählt.

Die nichtredundanten Addierer werden nun durch Carry-Save-Addierer ersetzt. Folgende Abbildung zeigt die Funktionsweise:

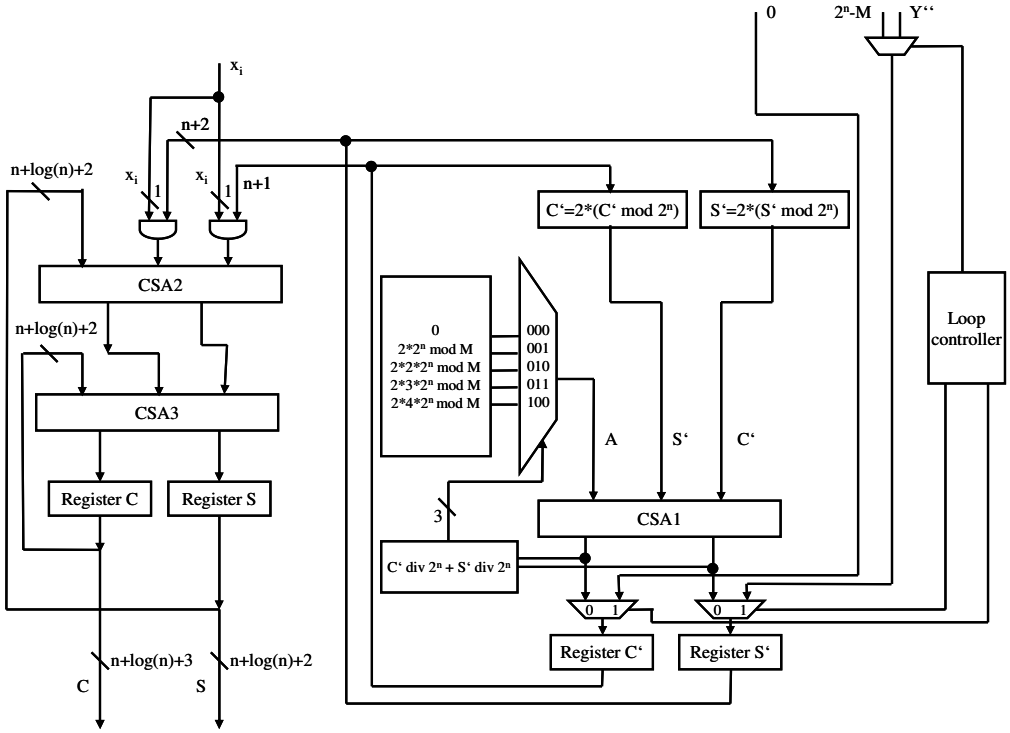

Abbildung 4.3.4: Modularmultiplikation basierend auf der Veränderung eines Operanden unter der Benutzung von redundanten Carry-Save-Addierern

Der erste Carry-Save-Addierer (CSA1) besitzt zwei Ausgänge, die rückgeschaltet sind. *S*' und *C*' sind zwischen der Verdopplung und der Addition jeweils kleiner als $2 \cdot 2^n$. Der Korrekturwert *A* ist kleiner als *M* und folglich kleiner als 2^n. Die Summe von *S*' und *C*' ist nach der Addition somit kleiner als $5 \cdot 2^n$. Dementsprechend gibt es nur fünf Möglichkeiten für die Reduktion modulo *M*

$$A = \begin{cases} 0 & (S'+C')div2^n = 0 \\ 2*2^n \bmod M & (S'+C')div2^n = 1 \\ 2*2*2^n \bmod M & (S'+C')div2^n = 2 \\ 2*3*2^n \bmod M & (S'+C')div2^n = 3 \\ 2*4*2^n \bmod M & (S'+C')div2^n = 4 \end{cases} \quad (4.84)$$

Der Addierer in der linken Hälfte wird durch zwei nacheinander geschaltete Carry-Save-Addierer ersetzt, da die „linke Hälfte“ vier Eingabewerte besitzt: Der Wert von *Y* in redundanter Form (*S*’ und *C*’) und beide Ausgänge müssen rückgekoppelt werden. Der folgende Pseudocode beschreibt das neue Verfahren:

Algorithmus 4.3.6: Modularmultiplikation basierend auf der Veränderung eines Operanden unter der Benutzung von redundanten Carry-Save-Addierern

```
Eingabe: X, Y<M<2^n, wobei 2^(n-1)<M<2^n and x_n=0;
Ausgabe: S = X*Y mod M.
LookUp(0)=0; LookUp(1)=2*2^n mod M;
LookUp(2)=2*2*2^n mod M; LookUp(3)=2*3*2^n mod M;
LookUp(4)=2*4*2^n mod M;
Methode:
(1)  S'=Y; C'=0; S=0; C=0;
(2)  while (X≠0) {
(3)    for (int i=0; i<n+1; i++) {
(4)      (S, C)=CSA(S, x_i*S', x_i*C');
(5)      (S, C)=CSA(S, C, C^alt);
(6)      S'=2*S' mod M;
(7)      C'=2*C' mod M;
(8)      (S', C')=CSA(S', C', A);
(9)      A = LookUp(S div 2^n + C div 2^n);
(10)   }
(11)   X=S div 2^n; X'=C div 2^n;
(12)   X=X+X';
(13)   S'=2^n-M; C'=0;
(14)   n=log(n)+2;
(15)   S=S mod 2^n;
(16)   C=C mod 2^n;
(17) }
(18) S=S+C;
(19) S=S mod M;
```

(C^{alt} ist der Wert von *C* aus dem vorherigen Schleifendurchlauf. Im ersten Schleifendurchlauf gilt C^{alt}=0.)

Während des ersten Durchlaufs der inneren Schleife verhält sich der Algorithmus 4.3.6 ebenso wie der Algorithmus 4.3.4 (nichtredundante Version). Danach zeigen sich Unterschiede, denn der Operand *X* wird jetzt aus einer redundanten Carry Save Dar-

stellung durch eine Addition gebildet. Allerdings wird in jedem Schleifendurchlauf nur ein Bit von X benötigt – das niederwertigste Bit. Das bedeutet, dass X parallel zu den weiteren Additionen berechnet werden kann. Die Berechnung von X kann bit-seriell mit einem einzigen Volladdierer erfolgen: In jedem Durchlauf werden das Summenbit x_i, sowie das Übertragsbit car_{i+1}, das für die Ermittlung von x_{i+1} erforderlich ist, berechnet. Folgende Abbildung zeigt die Berechnung von X:

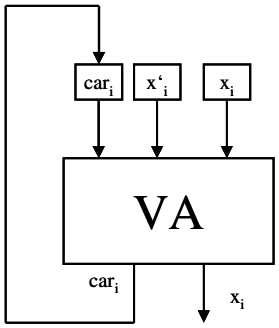

Abbildung 4.3.5: Bit-serielle Berechnung von *X*

In Algorithmus 4.3.6 werden weiterhin die Schritte (4), (5) und (6) bis (9) in einer Pipeline ausgeführt. Damit verläuft der längste Pfad des Algorithmus 4.3.6 durch zwei Carry-Save-Addierer.

Die Grundidee dieses Kapitels wurde in [24], [67] und [68] veröffentlicht.

4.3.6 Redundante Modularmultiplikation

Im letzten Abschnitt wurde die Modularmultiplikation basierend auf der Veränderung eines Operanden um redundante Addierer erweitert. Das ermöglicht es, die einzelnen Durchläufe der inneren Schleife des Algorithmus 4.3.6 in konstanter Zeit durchführen zu können. Allerdings bleibt das Problem der Konvertierung des Ergebnisses (Schritt (18)) in eine nichtredundante Zahl und der anschließenden Reduktion modulo M (Schritt (19)). In diesem Abschnitt wird aufgezeigt, wie die Modularmultiplikation vollständig in redundanter Form verwirklicht werden kann.

Die Eingabe eines redundanten Addierers besteht aus zwei nichtredundanten Zahlen (X, X') und (Y, Y'), sowie aus dem nichtredundanten Modulus M. Der Algorithmus 4.3.6 wird um ein n-bit Register X' erweitert. Die Zahlen X und X' werden in den Registern X und X' und die Zahlen Y und Y' in den Registern S' und C' gespeichert. Danach wird die Modularmultiplikation ausgeführt. Die folgende Abbildung zeigt den Datenfluss eines Schleifendurchlaufs der neuen Verbesserung:

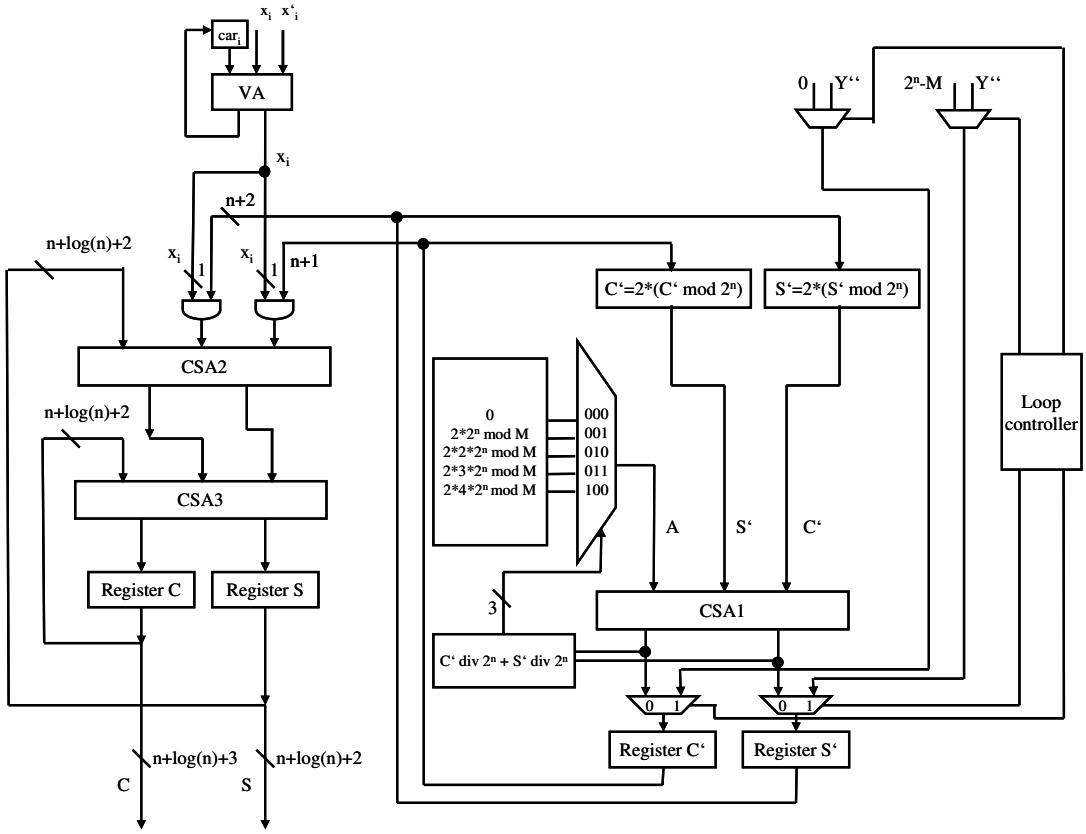

Abbildung 4.3.6: Ein Schleifendurchlauf des redundanten Modularmultiplizierers

Algorithmus 4.3.7: Redundante Modularmultiplikation basierend auf der Veränderung eines Operanden

```
Eingabe: X,X',Y<M<2^n, wobei 2^(n-1)<M<2^n;
Ausgabe: S = X*Y mod M.
LookUp(0)=0; LookUp(1)=2*2^n mod M;
LookUp(2)=2*2*2^n mod M; LookUp(3)=2*3*2^n mod M;
LookUp(4)=2*4*2^n mod M;
Methode:
(1)  S'=Y; C'=Y'; S=0; C=0; car=0;
(2)  while (X≠0 and X'≠0) {
(3)  (x_i, car)=VA(x_i, x_i', car);
(4)    for (int i=0; i<n+1; i++) {
(5)      (S, C)=CSA(S, x_i*S', x_i*C');
(6)      (S, C)=CSA(S, C, C^alt);
(7)      S'=2*S' mod M;
(8)      C'=2*C' mod M;
(9)      (S', C')=CSA(S', C', A);
(10)     A = LookUp(S div 2^n + C div 2^n);
(11)     (x_i, car)=VA(x_i, x_i', car);
(12)   }
(13)   X=S div 2^n; X'=C div 2^n;
(14)   S'=2^n-M; C'=0;
(15)   n=log(n)+2;
(16)   S=S mod 2^n;
(17)   C=C mod 2^n; }
```

Dabei werden die Schritte (7) bis (10) parallel zu Schritt (11) und in einer Pipeline mit den Schritten (5) und (6) ausgeführt.

Die folgende Abbildung zeigt das Schaltungsbild des Algorithmus 4.3.7

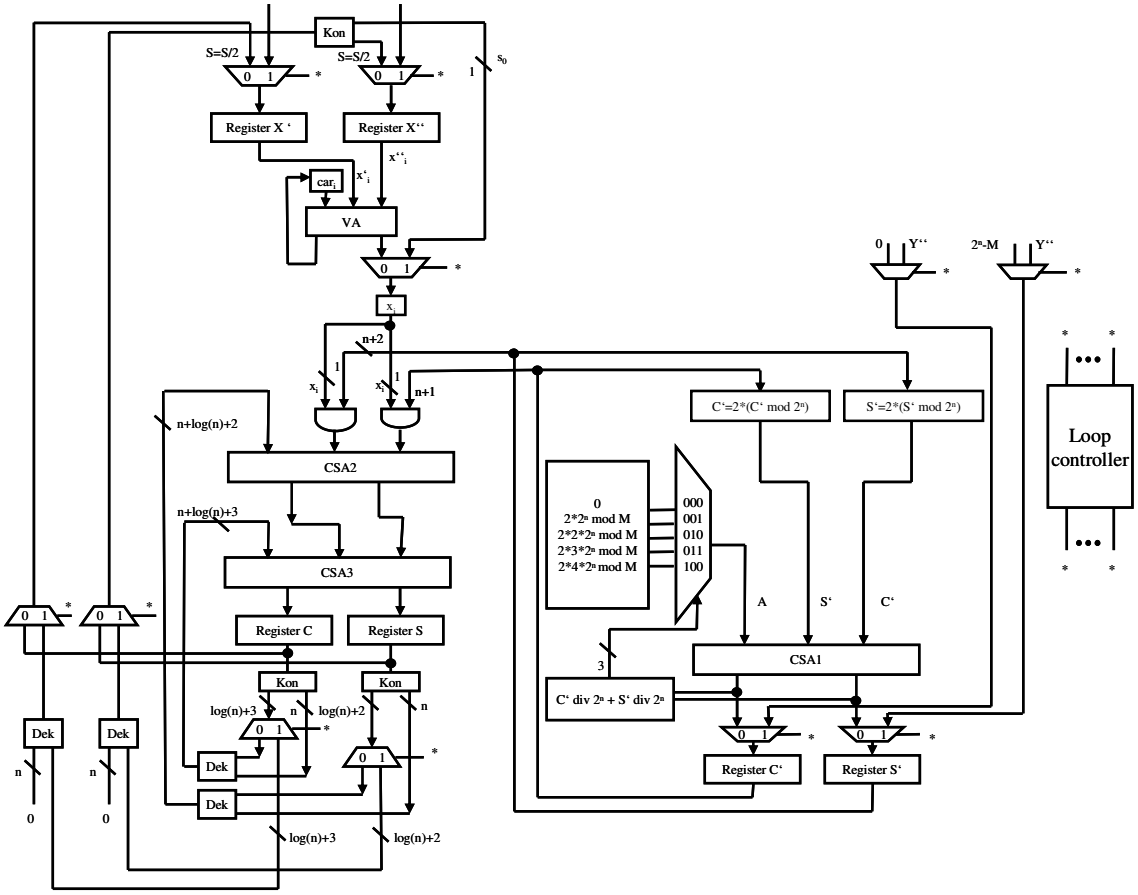

Abbildung 4.3.7: Redundanter Modularmultiplizierer

In diesem Abschnitt wurde gezeigt, wie die Modularmultiplikation basierend auf der Veränderung eines Operanden fast ohne zusätzliche Hardware zu der vollständig redundanten Modularmultiplikation erweitert werden kann.

4.3.7 Komplexitätsanalyse

Im letzten Abschnitt wurde der Algorithmus 4.3.7 für die vollständig redundante Modularmultiplikation dargestellt. An dieser Stelle wird die Komplexität dieses Algorithmus berechnet.

In Abschnitt 4.3.3 wurde bewiesen, dass die unvollständige Modularmultiplikation höchstens n+2log$n$+4 Schleifendurchläufe erfordert. Die Zeitkomplexität eines Schleifendurchlaufs ist das Maximum der Zeitkomplexität zweier Carry-Save-Additionen (Schritte (5) und (6)), einer Carry-Save-Addition und eines Zugriffs auf die Lookup

Table (Schritte (7) bis (10)) sowie der Schaltzeit eines Volladdierers (Schritt (11)). Daraus folgt, dass die Zeitkomplexität eines Schleifendurchlaufs bei t=2 Zeiteinheiten liegt. Die Zeitkomplexität der Gesamtschleife entspricht

$$T_{\text{Schleife}} = 2 \cdot (n + 2\log n + 4) = 2n + 4\log n + 8 \tag{4.85}$$

Zeiteinheiten.

Da das Verfahren weder Vor- noch Abschlussberechnung erfordert, gilt:

$$T_{\text{Modularmultiplikation}} = T_{\text{Schleife}} = 2n + 4\log n + 8 \tag{4.86}$$

Zeiteinheiten.

Der Algorithmus 4.3.7 benötigt einen n-bit Carry-Save-Addierer, zwei (n+log$n$)-bit Carry-Save-Addierer, eine Lookup Table für fünf n-bit Zahlen, vier n-bit Register X, X', C', S' und zwei (n+log$n$)-bit Register C und S. Demgemäß liegt die Flächenkomplexität bei

$$\begin{aligned} A_{\text{Modularmultiplikation}} &= 8n + 2 \cdot 8 \cdot (n + \log n) + 5n + 4 \cdot 4n + 2 \cdot 4 \cdot (n + \log n) \\ &= 53n + 24\log n \end{aligned} \tag{4.87}$$

Flächeneinheiten.

Somit liegt die Flächen-Zeit-Komplexität des Algorithmus bei

$$\begin{aligned} AT_{\text{Modularmultiplikation}} &= (53n + 24\log n) \cdot (2n + 4\log n + 8) \\ &= 106n^2 + 260n\log n + 424n + 96\log^2 n + 192\log n \end{aligned} \tag{4.88}$$

Flächen-Zeit-Einheiten.

Die folgende Tabelle präsentiert die Komplexität der Modularmultiplikation basierend auf der Veränderung eines Operanden für unterschiedliche Wortlängen. Dabei handelt es sich um Werte, die mit den oben beschriebenen Formeln berechnet wurden. Diese dienen der Veranschaulichung der Komplexität.

n	Flächenkomplexität/ Flächeneinheit	Zeitkomplexität/ Zeiteinheit	AT-Komplexität/ Flächen-Zeit-Einheiten
512	27.352	1.068	29.211.936
1024	54.512	2.096	114.257.152
2048	108.808	4.148	451.335.584
4096	217.376	8.248	1.792.917.248
8192	434.488	16.444	7.144.720.672

Tabelle 4-3: Komplexität der Modularmultiplikation basierend auf der Veränderung eines Operanden.

4.3.8 Zusammenfassung

In Abschnitt 4.3 wurde ein neues Verfahren für die Modularmultiplikation dargestellt. Im Unterschied zu allen bereits bekannten Algorithmen wird darin während der Berechnung ein Operand ständig verändert.

Es wurden unterschiedliche Optimierungen für den neuen Algorithmus dargestellt, die unter anderem auch Grundlage für zusätzliche Arbeiten bieten. Ferner wurde ein Optimierungsvorschlag weiterentwickelt: Die vollständigen Additionen wurden durch redundante Carry-Save-Additionen ersetzt, die Einzelteile des Schaltwerkes wurden parallelisiert und die Berechnung von Korrekturwerten wurde durch eine Lookup Table mit vorberechneten Werten ersetzt. Darüber hinaus wurde ein vollständig redundanter Modularmultiplizierer entwickelt, bei dem weder eine Vorberechnung für die einzelnen Modularmultiplikationen noch eine Konvertierung der redundanten Zahlen in nichtredundante Form erforderlich ist. Ferner wurde die Komplexität des Verfahrens analysiert und für unterschiedliche Wortlängen berechnet.

4.4 Algorithmenvergleich und Zusammenfassung

In diesem Kapitel wurden drei unterschiedliche Typen von Algorithmen für die Modularmultiplikation sowie deren Optimierungen dargestellt. Bei allen Optimierungen wurde die Anzahl der vollständigen Additionen, die für die Modularmultiplikation erforderlich sind, minimiert. Die von der Operandenlänge abhängigen Vergleiche wurden durch in konstanter Zeit durchführbare Vergleiche ersetzt. Durch die Nutzung von Lookup Tables mit vorberechneten Werten konnten mehrfach zu wiederholende, ähnliche Berechnungen eingespart werden. Zuletzt wurden die nichtredundanten, von der Operandengröße abhängigen Additionen durch Carry-Save-Additionen ersetzt. Bei der Modularmultiplikation basierend auf der Veränderung eines Operanden wurde nicht nur ein neues Verfahren für die Modularmultiplikation erarbeitet, sondern auch eine vollständig redundante Modularmultiplikation erreicht.

Im folgenden wird die Komplexität der drei besten, in diesem Kapitel präsentierten Algorithmen (Algorithmus 4.1.3: Schnelle Modularmultiplikation von Montgomery, Algorithmus 4.2.7: Verschränkte Modularmultiplikation mit verkürzter Abschlussphase und Algorithmus 4.3.7: Redundante Modularmultiplikation basierend auf der Veränderung eines Operanden) und den zwei besten in der Literatur vorgestellten Verfahren (verschränkte Modularmultiplikation von Koc[42] und Modularmultiplika-

tion von Montgomery von Kim[40]) miteinander verglichen. Dabei werden sowohl die Rückkonvertierung in die nichtredundante Form als auch die Normierung (Darstellung des Ergebnisses im Bereich zwischen 0 und *M*-1) mit dem in dieser Arbeit präsentierten Kombiaddierer stattfinden.

Folgende Diagramme zeigen die Zeit-, Flächen- und die Flächen-Zeit-Komplexität unterschiedlicher Algorithmen:

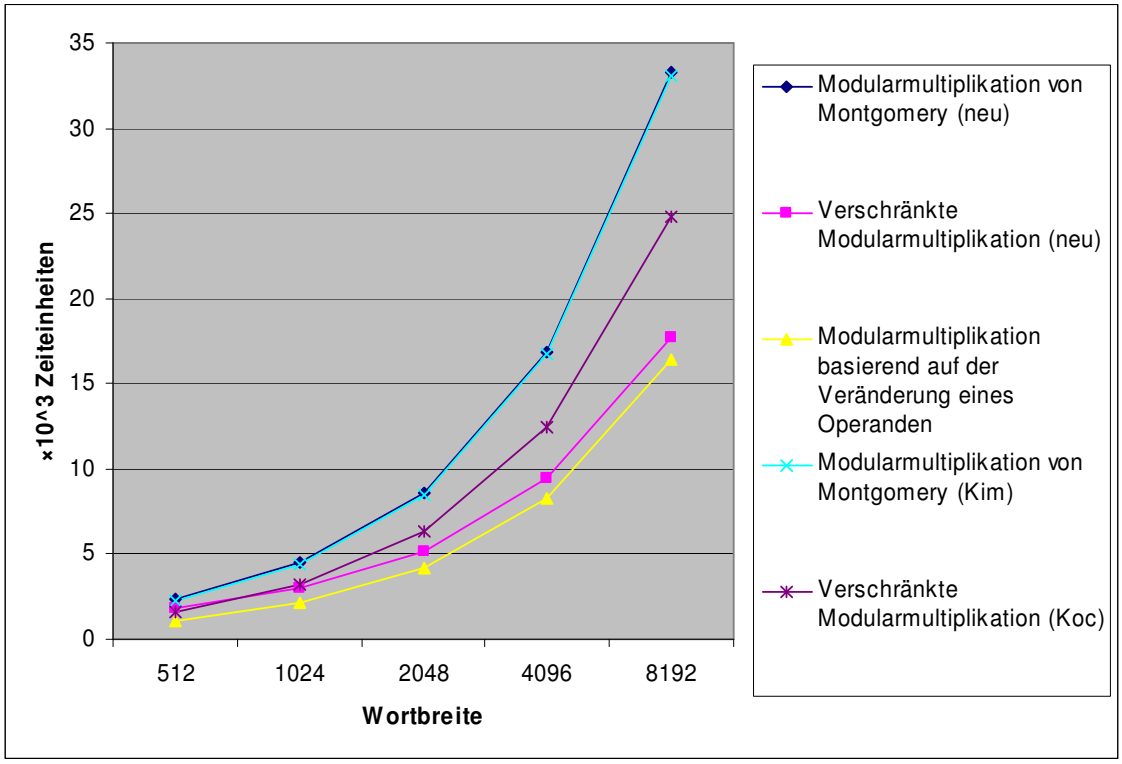

Diagramm 4-1 Vergleich der Zeitkomplexität der unterschiedlichen Algorithmen für Modularmultiplikation

Die Modularmultiplikation basierend auf der Veränderung eines Operanden erweist sich in Bezug auf die Zeitkomplexität als das mit Abstand beste Verfahren. Wie bereits beschrieben verdankt sie dies der Tatsache, dass der entsprechende Algorithmus in der redundanten Zahlendarstellung arbeitet und somit weder Vorberechnung noch Rückkonvertierung in das nichtredundante Zahlensystem erfordert. Jedoch schrumpft dieser Vorsprung mit steigender Operandengröße – der Anteil der Zeitkomplexität der Schleife in allen drei Algorithmen erhöht sich mit wachsenden Operanden in Relation zur Zeitkomplexität der Vorberechnungs- und Abschlussphase.

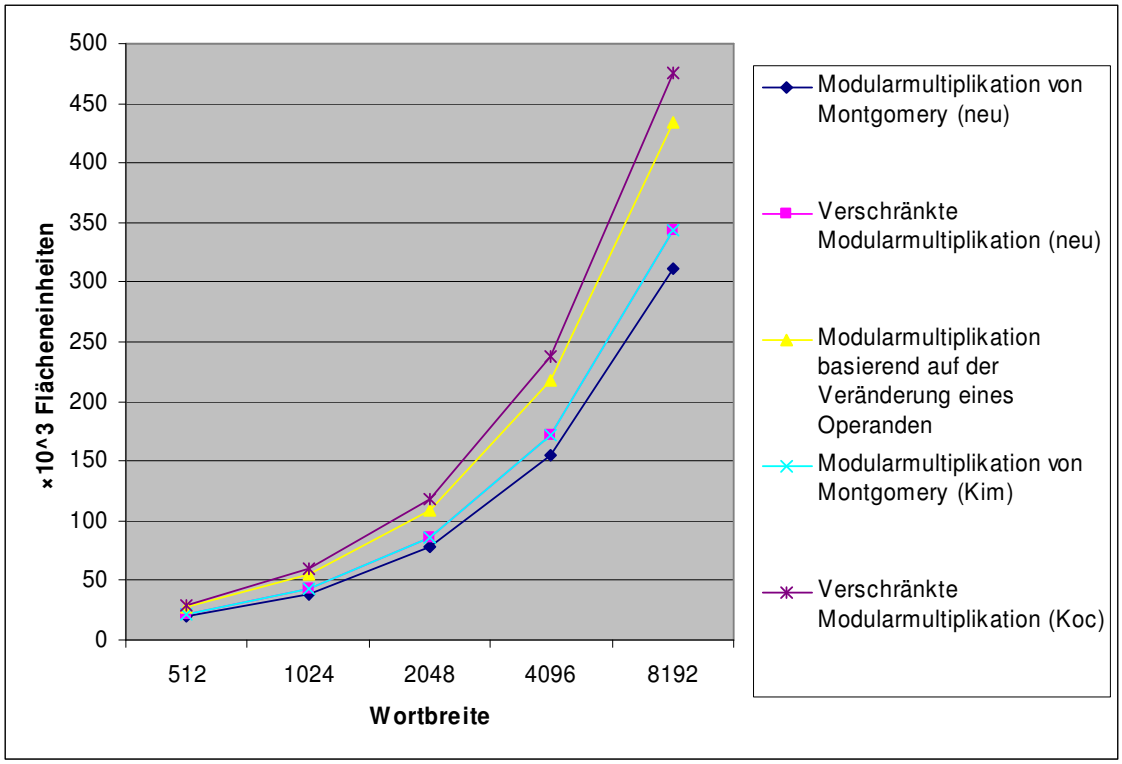

Diagramm 4-2 Vergleich der Flächenkomplexität der unterschiedlichen Algorithmen für Modularmultiplikation

Diagramm 4-2 zeigt auf, dass die in dieser Arbeit entwickelte Variante der Modularmultiplikation von Montgomery die beste Flächenkomplexität besitzt. Den zweitbesten Flächenaufwand teilen sich die ebenso in dieser Arbeit entwickelte Version der verschränkten Modularmultiplikation, die eine größere Lookup Table verwendet und die Implementierung der Modularmultiplikation von Montgomery von Kim. Das schlechteste Ergebnis liefert die verschränkte Modularmultiplikation von Koc, da sie drei Carry-Save-Addierer anstelle von einem benötigt.

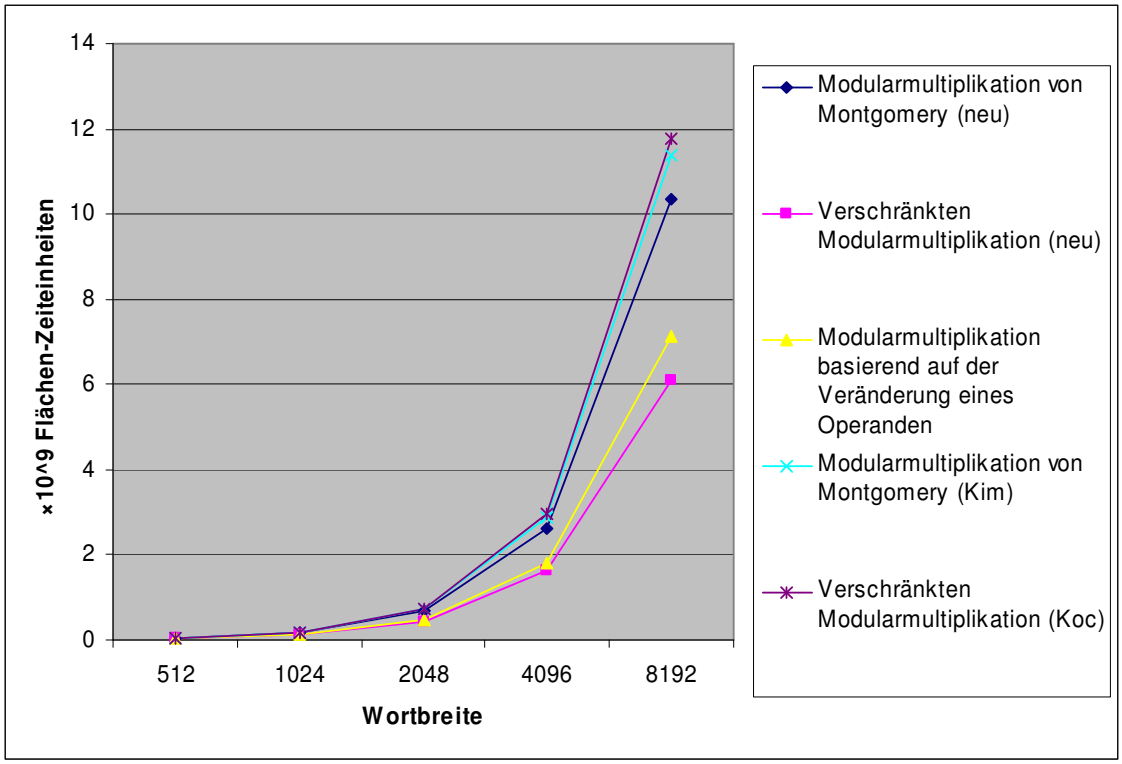

Diagramm 4-3 Vergleich der Flächen-Zeit-Komplexität der unterschiedlichen Algorithmen für Modularmultiplikation

Bei einer Operandengröße von 512 bzw. 1024 Bits bietet die Modularmultiplikation basierend auf der Veränderung eines Operanden die beste Flächen-Zeit-Komplexität trotz relativ höherer Flächenkomplexität. Bei steigender Operandengröße liefert die verschränkte Modularmultiplikation bessere Werte.

Alle drei neuentwickelten Algorithmen bieten eine hohe Leistung bei einer geringen Flächenkomplexität. Es ist jedoch anzuraten, je nach Anwendungsfall den passenden Algorithmus auszuwählen.

Eine Implementierung mit anschließendem Vergleich der in dieser Arbeit entwickelten Algorithmen für Modularmultiplikation von Montgomery, verschränkten Modularmultiplikation und der Version von Kim [40] wurden in [2] durchgeführt. Die ausführlichen Ergebnisse dieser Arbeit wurden in [3] veröffentlicht.

5 Redundante Modulare Exponentiation

5.1 Algorithmus für Modulare Exponentiation

Eine der wichtigsten Anwendungen der Modularmultiplikation ist die Modulare Exponentiation. Diese ist eine der wichtigsten Operationen der „Public Key"- Kryptographie. Fast alle Algorithmen der Modularen Exponentiation [34], [50] bestehen aus wiederholter Durchführung der Modularmultiplikation. Im voangegangenen Kapitel wurden drei Algorithmen für die Modularmultiplikation sowie deren Optimierung in Bezug auf Fläche und Zeit dargestellt. Diese drei Algorithmen können in jedem Algorithmus für Modulare Exponentiation benutzt werden. Aus diesem Grund wird der Standardalgorithmus gewählt, da dieser in Hardware sehr einfach implementierbar ist.

Algorithmus 5.1.1: Binäre Modulare Exponentiation

Eingabe: A, M, wobei $2^{n-1}<M<2^n$ und $Z<2^n$.
Ausgabe: A^Z mod M.
Methode:
(1) V=1;
(2) for (int i=n-1; i>=0; i--) {
(3) V=V*V mod M;
(4) if (z_i=1) V=V*A;
(5) }

Obwohl der Algorithmus 5.1.1 einer der ersten Algorithmen für die Modulare Exponentiation ist, gibt es in der Literatur keine anderen, die signifikant besser wären.

Für die Anwendung der Algorithmen für Modularmultiplikation werden folgende Notationen benutzt:

a) Redundante Modularmultiplikation basierend auf der Veränderung eines Operanden (Algorithmus 4.3.7): $P=MM_{Red}(X', X'', Y', Y'', M)$,

b) Verschränkte Modularmultiplikation (Algorithmus 4.2.7): $P=MM_{\text{Verschr}}(X, Y, M)$, wobei in beiden Fällen X und Y für die Eingaberegister der Operanden und M für das Eingaberegister des Modulus stehen.

c) Montgomery Modularmultiplikation (Algorithmus 4.1.4): $P=MM_{\text{Mont}}(X, Y, M)$, dabei sind X', X'', Y', Y'' Register für die redundante Darstellung von X und Y.

In den folgenden drei Abschnitten wird gezeigt, wie der entsprechende Algorithmus für die Modularmultiplikation in die Modulare Exponentiation eingebettet werden kann.

5.2 Modulare Exponentiation basierend auf Redundanter Modularmultiplikation

5.2.1 Beschreibung des Algorithmus

In diesem Abschnitt wird gezeigt, wie der Algorithmus für die Redundante Modularmultiplikation bei der Modularen Exponentiation angewendet werden kann.

Algorithmus 5.2.1: Binäre Modulare Exponentiation basierend auf Redundanter Modularmultiplikation

```
Eingabe: A, M, wobei 2^(n-1)<M<2^n und Z<2^n.
Ausgabe: A^Z mod M.
Procedure: Reduktion(V, W, M);
(1)  W=V-M;
(2)  if (W>=0) V=W;
Methode:
(1)  Lookup(0)=0;
(2)  V=2^n-M;
(3)  V=2*V;
(4)  Reduktion(V, W, M);
(5)  Lookup(1)=V;
(6)  V=2*V;
(7)  Reduktion(V, W, M);
(8)  Lookup(2)=V;
(9)  V=2*V;
(10) Reduktion(V, W, M);
(11) Lookup(4)=V;
(12) V=2^n-M;
(13) V=2*V;
(14) Reduktion(V, W, M);
(15) W=2*V;
(16) V=V+W;
(17) Reduktion(V, W, M);
(18) Reduktion(V, W, M);
(19) Lookup(3)=V;
(20) V=1; W=0;
(21) for (int i=n-1; i>=0; i--) {
(22)    (V,W)=MM_Red(V,W,V,W,M);
(23)    if (z_i=1) (V, W)=MM_Red(V,W,A,0,M); }
(24) V=V+W;
(25) Reduktion(V, W, M);
(26) Reduktion(V, W, M);
(27) Reduktion(V, W, M);
```

Zuerst wird die Lookup Table für die Modularmultiplikation berechnet, was sieben Additionen bzw. Subtraktionen benötigt. Danach kann die Schleife des Algorithmus 5.1.1 ausgeführt werden. Diese besteht aus *n* Schleifendurchläufen, wovon jeder entweder eine oder zwei Modularmultiplikationen enthalten kann. Da die Wahrscheinlichkeit

für beide Fälle bei zufällig gewählten Operanden gleich ist, besteht die Schleife aus durchschnittlich 1,5n Modularmultiplikationen. Das Ergebnis der Schleife ist eine redundante Zahl, die in den Registern V und W gespeichert wird. Dabei gilt für die Zahlen in diesen Registern $V,W<2^n$. Demgemäß gilt $V+W<2\cdot 2^n<4M$. Um das Ergebnis zu reduzieren, sind eine Addition (um $V+W$ zu berechnen) und noch drei weitere Additionen bzw. Subtraktionen erforderlich.

Der neue Algorithmus für die Modularmultiplikation besteht aus durchschnittlich 1,5n Modularmultiplikationen und elf Additionen.

5.2.2 Komplexitätsanalyse

In diesem Abschnitt wird die Komplexitätsanalyse des soeben beschriebenen Algorithmus 5.2.1 durchgeführt.

Die Modulare Exponentiation besteht im besten Fall aus n Modularmultiplikationen, sowie elf Additionen und im schlechtesten Fall aus 2n Modularmultiplikationen, sowie elf Additionen. Die Additionen können mit einem einfachen bit-seriellen Addierer, der aus einem rückgekoppelten Volladdierer besteht, durchgeführt werden. Die Zeitkomplexität liegt dabei im besten Fall bei

$$T_{\min}=n\cdot(2n+4\log n+8)+11n=2n^2+4n\log n+19n \tag{5.1}$$

Zeiteinheiten, und im schlechtesten Fall bei

$$T_{\max}=2n\cdot(2n+4\log n+8)+11n=4n^2+8n\log n+27n \tag{5.2}$$

Zeiteinheiten.

Die Modulare Exponentiation benötigt einen Modularmultiplizierer, einen Volladdierer und fünf n-bit Register A, Z, M, V and W. In Abschnitt 5.2.1 wurde ein universeller Modularmultiplizierer mit redundanten Eingaben präsentiert, dabei kommen in Algorithmus 5.2.1 nur zwei Spezialfälle zum Tragen: Die Modulare Quadrierung $(V,W)=MM_{\text{Red}}(V,W,V,W,M)$ und die Modularmultiplikation mit einer redundanten Eingabe (V,W). Das heißt, dass die Register V und W anstelle der Register X' und X'' eingesetzt werden können. Für den zweiten Operanden können das Register A und 0 für die Modularmultiplikation mit A, und die Register V und W für die Modulare Quadrierung verwendet werden. Somit können die Register Y' und Y'' eingespart werden. Dementsprechend werden bei der Modularen Exponentiation ein (n+1)- und zwei (n+log$n$+2)-bit Carry-Save-Addierer, sieben (n+1)-bit Register S', C', M, A, Z, V, und W, zwei (n+log$n$+2)-bit Register S und C und eine Lookup Table für fünf n-bit Zahlen benutzt.

Die folgenden zwei Tabellen präsentieren die Ergebnisse der Komplexitätsanalyse der Modularen Exponentiation basierend auf der Veränderung eines Operanden.

Logische Schaltung	Anzahl der Voll-addierer	Anzahl der 1-bit Register	Anzahl der Elemente in der Lookup Table	Flächen-komplexität	Anzahl der Zeiteinheiten im besten Fall	Anzahl der Zeiteinheiten im schlechtesten Fall
Modular-multi-plikation	$3n+2\log n+5$	$9n+2\log n+11$	$5n$	$65n+10\log n+70$	$2n+4\log n+8$	$2n+4\log n+8$
Modulare Expo-nentiation	$3n+2\log n+6$	$9n+2\log n+11$	$5n$	$65n+10\log n+78$	$2n^2+4n\log n+19n$	$4n^2+8n\log n+27n$

Tabelle 5-1: Komplexitätsanalyse der Redundanten Modularen Exponentiation basierend auf der Veränderung eines Operanden

	Flächenkomplexität	Zeitkomplexität	AT-Komplexität
Bester Fall	$65n+10\log n+78$	$2n^2+4n\log n+19n$	$130n^3+280n^2\log n+1391n^2$ $+40n\log^2 n+502n\log n+1482n$
Schlechtester Fall	$65n+10\log n+78$	$4n^2+8n\log n+27n$	$260n^3+560n^2\log n+2067n^2$ $+80n\log^2 n+894n\log n+2106n$

Tabelle 5-2: Flächen-Zeit-Komplexität der Modularen Exponentiation basierend auf der Veränderung eines Operanden

5.3 Modulare Exponentiation basierend auf der verschränkten Modularmultiplikation

5.3.1 Beschreibung des Algorithmus

Analog zu Abschnitt 5.2 wird gezeigt, wie die verschränkte Modularmultiplikation bei der Modularen Exponentiation effizient eingesetzt werden kann.

Zuerst werden die vom Operanden Y unabhängigen Werte in der Lookup Table berechnet. Dafür wird die erste Modularmultiplikation vollständig durchgeführt. Bei weiteren Modularmultiplikationen werden in der Vorberechnungsphase die Werte $(2\cdot 2^n \bmod M)$, $(2\cdot 2\cdot 2^n \bmod M)$ und $(2\cdot 3\cdot 2^n \bmod M)$ nicht mehr neu berechnet. Dies bedeutet, dass die Zeitkomplexität einer solchen Modularmultiplikation

$$\begin{aligned} T_{\text{verschr. Reduziert MM}} &= T_{\text{verschränkte Mod. Mult}} - 4T_{\text{Addition}} = \\ &= 2n + 120\log n - 229 - 4\cdot(8\log n - 16) = 2n + 88\log n - 165 \end{aligned} \tag{5.3}$$

Zeiteinheiten ist.

Die Exponentiation sieht damit folgendermaßen aus:

Algorithmus 5.3.1: Binäre Modulare Exponentiation basierend auf der verschränkten Modularmultiplikation

```
Eingabe: A, M, wobei 2^(n-1)<M<2^n und Z<2^n.
Ausgabe: A^Z mod M.
Methode:
(1)  V=1;
(2)  for (int i=n-1; i>=0; i--) {
(3)    if (i=0) V=MM_Verschr.(V,V,M);
(4)    else V=MM_Verschr. Reduziert(V, V, M);
(5)    if (z_i=1) V= MM_Verschr. Reduziert(V, A, M);
(6)  }
```

Der Algorithmus 5.3.1 besteht aus einer verschränkten Modularmultiplikation und durchschnittlich 1,5*n* reduzierten verschränkten Modularmultiplikationen.

5.3.2 Komplexitätsanalyse

In diesem Abschnitt wird die Komplexitätsanalyse des Algorithmus 5.3.1 durchgeführt.

Im besten Fall erfordert der Algorithmus eine verschränkte Modularmultiplikation und *n*-1 reduzierte verschränkte Modularmultiplikationen. Im schlechtesten Fall werden eine verschränkte Modularmultiplikation und 2*n*-1 reduzierte verschränkte Modularmultiplikationen angewandt. Das heißt, dass die Zeitkomplexität im besten Fall

$$\begin{aligned} T_{\min} &= 2n + 120\log n - 229 + (n-1)\cdot(2n + 88\log n - 165) = \\ &= 2n^2 + 88n\log n - 165n + 32\log n - 64 \end{aligned} \tag{5.4}$$

Zeiteinheiten, und im schlechtesten Fall

$$\begin{aligned} T_{\max} &= 2n + 120\log n - 229 + (2n-1)\cdot(2n + 88\log n - 165) = \\ &= 4n^2 + 176n\log n - 330n + 32\log n - 64 \end{aligned} \tag{5.5}$$

Zeiteinheiten beträgt.

Die Modulare Exponentiation benötigt einen Modularmultiplizierer und drei zusätzliche Register *A*, *Z* und *V*. Somit liegt die Flächenkomplexität der Gesamtschaltung bei

$$\begin{aligned} A &= A_{\text{verschränkte Mod. Mult.}} + 3\cdot A_{\text{n-bit Register}} \\ &= 8n + 6n + 8n + 5\cdot 4n + 3\cdot 4n = 42n + 3\cdot 4n = 54n \end{aligned} \tag{5.6}$$

Flächeneinheiten.

Die folgende Tabelle beschreibt die Flächen-Zeit-Komplexität der Modularen Exponentiation basierend auf der verschränkten Modularmultiplikation:

	Flächenkomplexität	Zeitkomplexität	AT-Komplexität
Bester Fall	$54n$	$2n^2+88n\log n-165n+32\log n-64$	$108n^3+4752n^2\log n-8910\cdot n^2+1728\cdot n\cdot\log(n)-3456\cdot n$
Schlechtester Fall	$54n$	$4n^2+176n\log n-330n+32\log n-64$	$216n^3+9504n^2\log n+17820n^2+1728n\log n-3456n$

Tabelle 5-3: Flächen-Zeit-Komplexität der Modularen Exponentiation basierend auf der verschränkten Modularmultiplikation

5.4 Modulare Exponentiation basierend auf der Modularmultiplikation von Montgomery

5.4.1 Beschreibung des Algorithmus

In diesem Abschnitt wird wiederum analog zu Abschnitt 5.2 gezeigt, wie die Modularmultiplikation von Montgomery in die Modulare Exponentiation eingebettet werden kann. Hierbei muss auf die Ergebnisse des Abschnitts 4.1 zurückgegriffen werden, in dem gezeigt wurde, dass eine Modularmultiplikation durch zwei Modularmultiplikationen von Montgomery ersetzt werden kann. Da bei der Modularen Exponentiation mehrere Modularmultiplikationen verwendet werden, würde eine solche Verdopplung die Verdopplung der Gesamtzeitkomplexität bedeuten. Um dies zu vermeiden, wird die Modulare Exponentiation im so genannten Montgomery Raum durchgeführt. Bei einer Modularmultiplikation von Montgomery wird $X\cdot Y\cdot 2^{-n} \bmod M$, anstatt $X\cdot Y \bmod M$ berechnet. Es gibt allerdings eine Möglichkeit, die Zahlen X und Y in den Montgomery Raum umzuwandeln, indem aus X und Y die Zahlen $X\cdot 2^n \bmod M$ und $Y\cdot 2^n \bmod M$ berechnet werden. Werden diese der Modularmultiplikation von Montgomery unterzogen, entsteht die Zahl $X\cdot 2^n\cdot Y\cdot 2^n\cdot 2^{-n} \bmod M = X\cdot Y\cdot 2^n \bmod M$ und somit verbleibt das Produkt im Montgomery Raum. Diese Methode wird auch bei der Modularen Potenzierung verwendet. Nachdem die Zahl A in den Montgomery Raum umgewandelt wurde, wird $(A\cdot 2^n \bmod M)^Z \bmod M$ berechnet, wobei die wiederholte Anwendung der Modularmultiplikationen durch die wiederholte Anwendung der Modularmultiplikation von Montgomery ersetzt wird. Das Ergebnis einer solchen Modularen Exponentiation ist die

Zahl $(A^Z \cdot 2^n)$ mod M [50]. Eine weitere Modularmultiplikation von Montgomery mit Eins liefert das gewünschte Ergebnis.

$$MM_{Mont}((A^Z \cdot 2^n) \bmod M, 1) = A^Z \cdot 2^n \cdot 2^{-n} \bmod M = A^Z \bmod M \tag{5.7}$$

Der Algorithmus für die Montgomery-Modular-Exponentiation lässt sich somit folgendermaßen darstellen:

Algorithmus 5.4.1: Binäre Modulare Exponentiation basierend auf der Modularmultiplikation von Montgomery

```
Eingabe: A, M, wobei 2^(n-1)<M<2^n, Z<2^n, L1=2^n mod M, L2=2^2n mod M.
Ausgabe: A^z mod M.
Methode:
(1)  A=MMMont(A, L2, M);
(2)  V=L1;
(3)  for (int i=n-1; i>=0; i--) {
(4)    V=MMMont(V, V, M);
(5)    if (zi=1) V=MMMont(V, A, M);
(6)  }
(7)  V=MMMont(V, 1, M);
```

Dabei ist die Berechnung von L_1 trivial. Die Berechnung von L_2 ist dagegen relativ schwierig, denn jede Modularmultiplikation von Montgomery benötigt L_2 als Eingabe. Demzufolge kann L_2 nicht durch eine Montgomery Modularmultiplikation berechnet werden. Eine gründliche Literaturrecherche brachte keine Problemlösung. Somit sind drei mögliche Lösungen für die Berechnung von 2^{2n} mod M denkbar:

- Die Berechnung von 2^{2n} mod M auf einem Rechner und die Übertragung des Ergebnisses auf die hier betrachtete Hardware. Diese Lösung ist nicht akzeptabel, da die Modulare Exponentiation vollständig in Hardware durchgeführt werden soll.
- Einbau von zusätzlicher Hardware. Dies würde die Flächenkomplexität stark erhöhen und wird deswegen nicht weiter betrachtet.
- Berechnung von 2^{2n} mod M mit der Hardware, die für die Modularmultiplikation von Montgomery benutzt wird. Da diese Hardware für die oben beschriebene Aufgabe nicht optimiert ist, entsteht ein relativ hoher Zeitaufwand, was jedoch zu vernachlässigen ist, weil es sich hierbei um eine einmalige Berechnung innerhalb der Modularen Exponentiation handelt. Somit ist diese Lösung zu favorisieren.

Für diese Berechnung wird der Standardalgorithmus für die verschränkte Modularreduktion ausgewählt. Für die Addition wird der Kombiaddierer aus Abschnitt 3.3 verwendet. Demzufolge stellt sich der Algorithmus für die modulare Exponentiation mit der Vorberechnung von 2^{2n} mod M folgendermaßen dar:

Algorithmus 5.4.2: Binäre Modulare Exponentiation basierend auf der Modularmultiplikation von Montgomery

```
Eingabe: A, M, wobei 2^(n-1)<M<2^n, Z<2^n.
Ausgabe: A^Z mod M.
Methode:
(1)  V=2^n mod M;
(2)  for (int i=n-1; i>=0; i--) {
(3)    V = 2*V;
(4)    W = V – M;
(5)  if (W>=0) V=W;
(6)  }
(7)  W = V;
(8)  A=MM_Mont(A, W, M);
(9)  V=2^n mod M;
(10) for (int i=n-1; i>=0; i--) {
(11)   V=MM_Mont.(V,V,M);
(12)   if (z_i=1) V=MM_Mont(V, A, M);
(13) }
(14) V=MM_Mont(V, 1, M);
```

Der Algorithmus besteht aus durchschnittlich $1{,}5n + 2$ Modularmultiplikationen von Montgomery und n Additionen.

5.4.2 Komplexitätsanalyse

In diesem Abschnitt wird die Komplexitätsanalyse des im Abschnitt 5.4.1 beschriebenen Algorithmus 5.4.2 durchgeführt.

Im besten Fall benötigt der Algorithmus 5.4.2 n+2 Modularmultiplikationen von Montgomery und n Additionen, im schlechtesten Fall entsprechend 2n+2 Modularmultiplikationen von Montgomery und n Additionen. Dabei können die Additionen mit dem in Abschnitt 3.3.6 präsentierten Kombiaddierer durchgeführt werden. Die Zeitkomplexität beträgt dabei im besten Fall

$$\begin{aligned} T_{\min} &= n\cdot(8\log n - 16) + (n+2)\cdot(2n + 24\log n - 44) = \\ &= 8n\log n - 16n + 2n^2 + 24n\log n - 44n + 4n + 48\log n - 88 = \\ &= 2n^2 + 32n\log n - 56n + 48\log n - 88 \end{aligned} \tag{5.8}$$

Zeiteinheiten, und im schlechtesten Fall

$$\begin{aligned} T_{\max} &= n\cdot(8\log n - 16) + (2n+2)\cdot(2n + 24\log n - 44) = \\ &= 8n\log n - 16n + 4n^2 + 48n\log n - 88n + 4n + 48\log n - 88 = \\ &= 4n^2 + 56n\log n - 100n + 48\log n - 88 \end{aligned} \tag{5.9}$$

Zeiteinheiten.

Die Modulare Exponentiation erfordert einen Modularmultiplizierer, einen Kombiaddierer und vier *n*-bit Register *A*, *Z*, *V*, und *W*. Infolgedessen liegt die Flächenkomplexität der Schaltung bei

$$\begin{aligned} A &= A_{Mont.Modularmultiplikation} + A_{Kombiaddierer} + 4\cdot A_{register} = \\ &= 38n + 6n + 4\cdot 4n = 60n \end{aligned} \tag{5.10}$$

Flächeneinheiten.

Folgende Tabelle beschreibt die Flächen-Zeit-Komplexität der Modularen Exponentiation basierend auf der Modularmultiplikation von Montgomery

	Flächenkomplexität	Zeitkomplexität	AT-Komplexität
Bester Fall	$60n$	$2n^2+32n\log n-56n+48\log n-88$	$120n^3+1920n^2\log n-3360n^2+2880n\log n-5280n$
Schlechtester Fall	$60n$	$4n^2+56n\log n-100n+48\log n-88$	$240n^3+3360n^2\log n-6000n^2+2880n\log n-5280n$

Tabelle 5-4: Flächen-Zeit-Komplexität der Modularen Exponentiation basierend auf der Modularmultiplikation von Montgomery

5.5 Algorithmenvergleich und Zusammenfassung

In Kapitel 5 wurde der Algorithmus für Modulare Exponentiation vorgestellt. Darüber hinaus wurden drei in dieser Arbeit entwickelte Algorithmen für die Modularmultiplikation in die Modulare Exponentiation eingebettet und die Komplexitätsanalyse der einzelnen Verfahren durchgeführt. In diesem Abschnitt wird ein übergreifender Komplexitätsvergleich von drei neuentwickelten Algorithmen und zwei der besten in der Literatur bekannten Verfahren der Modularmultiplikation von Montgomery von Kim [40] und der verschränkten Modularmultiplikation von Koc [42] geboten. Die zwei letzteren Algorithmen werden in den im Abschnitt 5.1 beschriebenen Verfahren für die Modulare Exponentiation eingebettet.

Folgende Diagramme zeigen die Flächen-, Zeit- und die Flächen-Zeit-Komplexität unterschiedlicher Algorithmen.

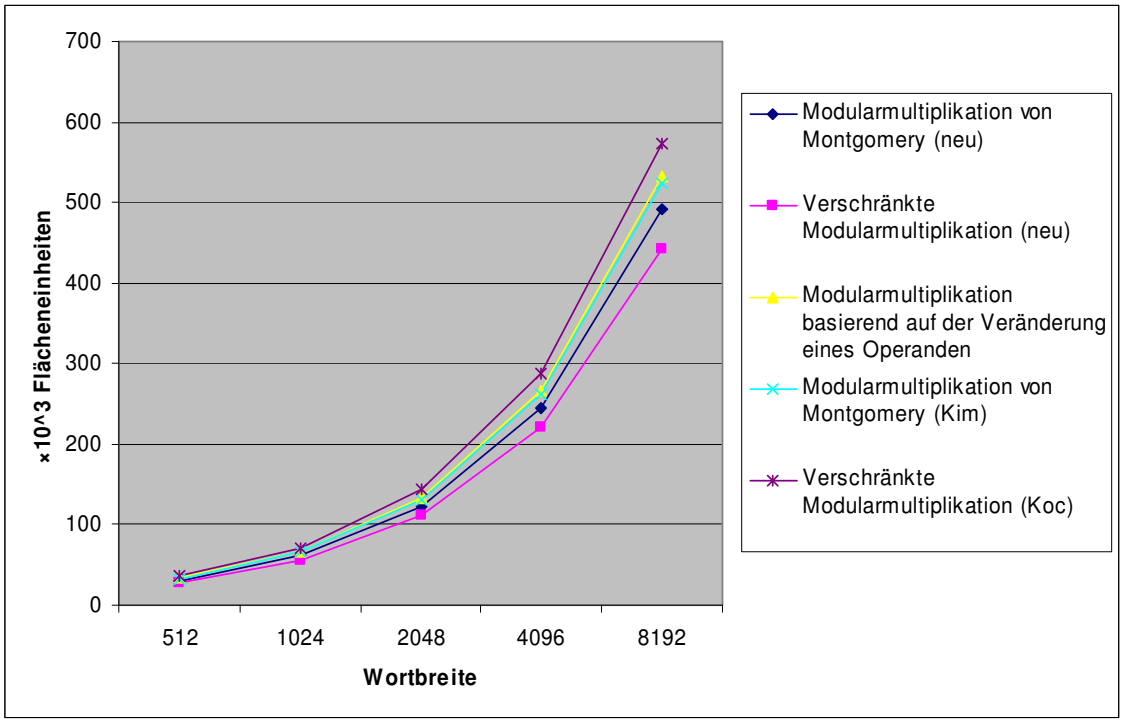

Diagramm 5-1 Vergleich der Flächenkomplexität der unterschiedlichen Algorithmen für Modulare Exponentiation

Der Flächenkomplexitätsvergleich zeigt, dass die Exponentiation mit der in dieser Arbeit entwickelten Version der verschränkten Modularmultiplikation die kleinste Fläche benötigt. Die schlechteste Flächenkomplexität liefert die Modulare Exponentiation mit der verschränkten Modularmultiplikation von Koc. Dies resultiert aus der Tatsache, dass diese Modularmultiplikation drei Carry-Save-Addierer verwendet.

Nachfolgend wird die Zeitkomplexität der verschiedenen Algorithmen verglichen. Da bei der Modularen Exponentiation die Anzahl der Modularmultiplikationen variabel ist, wird jeweils der Durchschnitt des besten und schlechtesten Falls betrachtet. Dabei ist zu beachten, dass die Anzahl der Modularmultiplikationen von der Operandenauswahl und nicht von der Algorithmusauswahl abhängt.

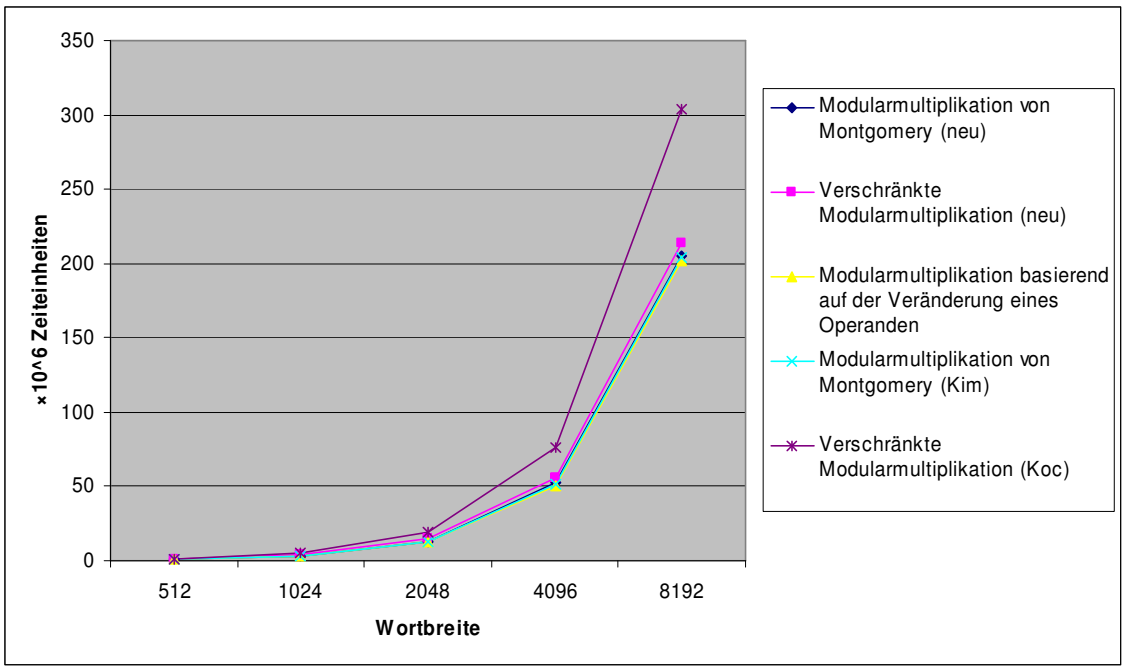

Diagramm 5-2 Vergleich der Zeitkomplexität der unterschiedlichen Algorithmen für Modulare Exponentiation

Bei der Zeitkomplexität hingegen ist die Modulare Exponentiation mit der Modularmultiplikation basierend auf der Veränderung eines Operanden führend. Der Grund dafür ist, dass die Modularmultiplikation basierend auf der Veränderung eines Operanden im redundanten Zahlensystem arbeitet und folglich weder Vor- noch Nachberechnung benötigt.

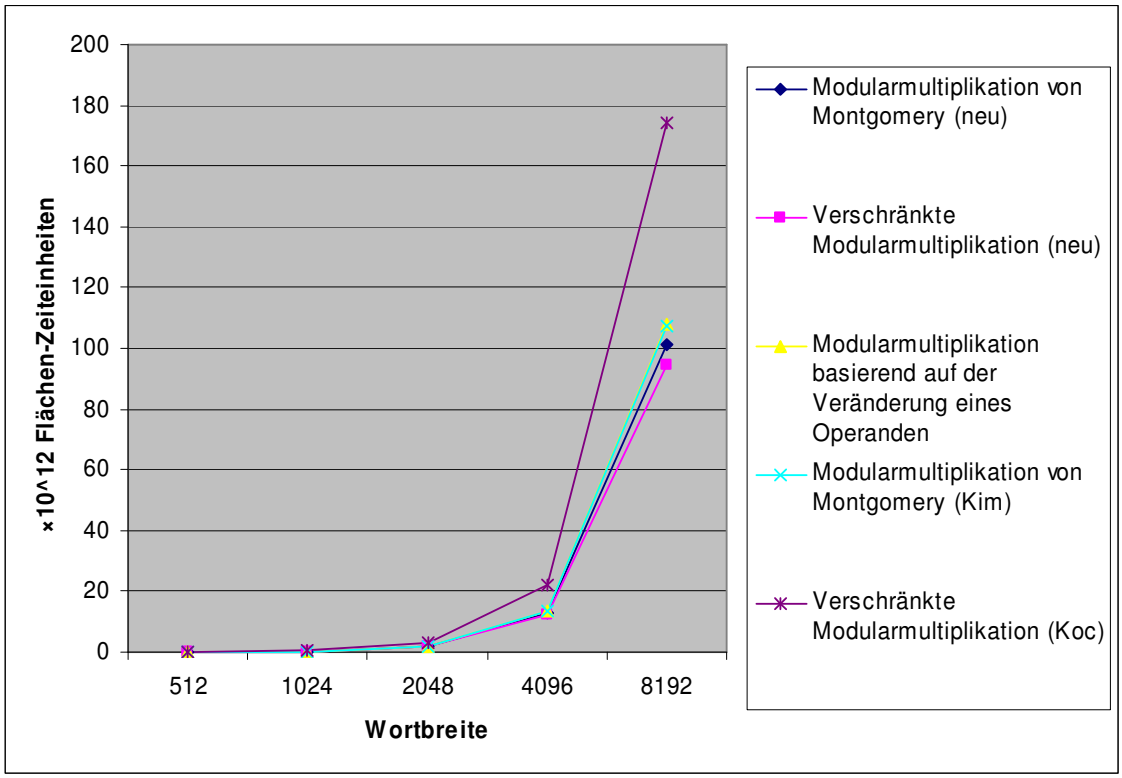

Diagramm 5-3 Vergleich der Flächen-Zeit-Komplexität der unterschiedlichen Algorithmen für Modulare Exponentiation

Beim Flächen-Zeit-Produkt liefert die Modulare Exponentiation mit der Modularmultiplikation basierend auf der Veränderung eines Operanden bei einer Operandengröße von bis zu 2048 Bits das beste Ergebnis. Danach folgt die in dieser Arbeit entwickelte Modulare Exponentiation mit verschränkter Modularmultiplikation. Der Grund hierfür liegt in der Tatsache, dass bei steigenden Operanden die Vor- und Nachberechnungszeiten eine kleinere Rolle spielen und das Zeitverhalten durch die Hauptschleife bestimmt wird.

6 Parallele Algorithmen der Modularmultiplikation

6.1 Parallele Modularmultiplikation basierend auf der Benutzung von Lookup Tables

6.1.1 Idee der schnelleren Modularen Potenzierung

Die Modulare Potenzierung ist eine der wichtigsten Operationen der „Public Key"-Kryptographie. Sie wird durch wiederholte Anwendung der Modularmultiplikation realisiert. Eine modulare Potenzierung $E=A^B \bmod M$, wobei A, B, $M < 2^n$ und $A < M$ sind, erfordert mindestens n nacheinander ausgeführte Modularmultiplikationen. Dabei verwenden alle Modularmultiplikationen innerhalb einer modularen Potenzierung denselben Modulus M.

Der hierfür entwickelte Algorithmus für Modularmultiplikation besteht aus einer gewöhnlichen Multiplikation mit nachfolgender Reduktion. Um die Zeit der Reduktion einer $2n$-bit-langen Zahl zu minimieren, wird diese durch eine parallele Addition von n Zahlen der Länge n ersetzt und das Ergebnis anschließend reduziert. Diese n-bit-langen Zahlen werden aus einer Lookup Table entnommen, deren Werte schon vorher berechnet wurden. Da die Werte der Lookup Table ausschließlich vom Modulus M abhängen, werden sie einmal für die gesamte modulare Potenzierung berechnet. Jeder Schritt der Modularmultiplikation wird in der Zeit $O(\log n)$ berechnet. Der so entstehende Algorithmus für die Modularmultiplikation wird im nächsten Abschnitt erläutert.

6.1.2 Algorithmus für Modularmultiplikation

Der in dieser Arbeit entwickelte Algorithmus besteht aus einer Vorberechnungsphase und zwei weiteren Phasen für einzelne Multiplikationen:

In der Vorberechnungsphase werden n Werte der Form $2^i \bmod M$ mit $n \le i < 2n$ berechnet und in eine Lookup Table gespeichert. Diese Berechnung wird nur einmal für alle Modularmultiplikationen innerhalb einer modularen Potenzierung durchgeführt.

In der darauf folgenden ersten Phase wird eine gewöhnliche Multiplikation unter Benutzung von parallelen Additionen durchgeführt.

In der zweiten Phase wird das Ergebnis der Multiplikation zu einer n-bit-langen Zahl, die kongruent zu $A \cdot B \bmod M$ ist, reduziert.

Vorberechnung der Lookup Table:

Sei M der Modulus, mit $2^{n-1} < M < 2^n$. Es werden die Werte von $2^0 \bmod M$, $2^1 \bmod M$, ..., $2^{n-1} \bmod M$, $2^n \bmod M$, ..., $2^{2n-1} \bmod M$ berechnet. Die Zahlen 2^0, ..., 2^{n-1} sind kleiner als M. Daher gilt: für alle $0 \le i < n$: $2^i \bmod M = 2^i$ – diese Werte müssen nicht extra berechnet werden; für alle $n \le i < 2n$: $2^i \bmod M < 2^i$ – diese Werte werden berechnet und in einer Lookup Table gespeichert.

t[2n-1]	$2^{2n-1} \bmod M$
t[2n-2]	$2^{2n-2} \bmod M$
...	...
t[n]	$2^n \bmod M$

Tabelle 6-1: Lookup Table mit *n* unterschiedlichen Zweierpotenzen modulo 2

Erste Phase: Multiplikation

In dieser Phase wird das Produkt $P'=X \cdot Y$ ($X, Y < M < 2^n$) gebildet. P' ist eine $2n$-bit Zahl und $P' = \sum_{i=0}^{n-1} (2^i x_i) \cdot Y$. Alle Teilprodukte werden mit dem Zeitaufwand von O(1) parallel zueinander berechnet und unter der Nutzung des Prinzips des binären Baumes aufaddiert: Die Addition der Teilprodukte verläuft dabei paarweise. Das Ergebnis besteht aus $n/2$ neuen Teilprodukten. Diese $n/2$ Teilprodukte werden wiederum parallel aufaddiert, usw. Nach $\log n$ Additionen entsteht das Gesamtprodukt P'.

Für eine praktische Implementierung ist es nicht sinnvoll, einen binären Baum mit nicht redundanten Additionen einzusetzen, da die Dauer dieser Addition von der Länge der Operanden abhängt [24]. Stattdessen wird der Wallace Tree [86] verwendet. Eine schnellere Implementierung wird in Abschnitt 6.1.4 dargelegt.

Zweite Phase: Reduktion

Das Produkt $P = X \cdot Y \bmod M = P' \bmod M$ wird berechnet. Dabei ist P' eine $2n$-bit Zahl. P' wird dabei dargestellt als:

$$P' = \sum_{i=0}^{2n-1} p'_i \cdot 2^i = P' \bmod 2^n + \sum_{i=n}^{2n-1} p'_i \cdot 2^i \tag{6.1}$$

Das modulare Produkt ist $P = P' \bmod M \cong (P' \bmod M + k \cdot M) \bmod M$, wobei $k \in N$. Demzufolge gilt:

$$P \cong (P' \bmod 2^n + \sum_{i=n}^{2n-1} (p'_i \cdot 2^i \bmod M)) \bmod M \tag{6.2}$$

Alle Werte von $p'_i \cdot 2^i \bmod M$ sind in der Lookup Table gespeichert. Es gibt $n+1$ partielle Produkte. Das $(n+1)$-te partielle Produkt ist $P' \bmod 2^n$, wobei $p'_i \cdot 2^i \bmod M$ entweder Null oder der i-te Wert aus der Lookup Table ist. Alle $n+1$ Teilprodukte werden, unter der Nutzung desselben binären Baumes wie der binäre Baum aus Phase 1, addiert. Dabei werden $\log(n+1)$ Schritte benötigt.

Alle Teilprodukte sind kleiner als 2^n. Demgemäß ist die Summe S kleiner als $(n+1) \cdot 2^n$. Das heißt, dass S eine Zahl der Länge $\log((n+1) \cdot 2^n) = n + \log(n+1)$ ist. Darüber hinaus ist die Summe S kongruent zu P' modulo M.

Beispiel: $P = 1101 \cdot 1011 \bmod 1111 \Rightarrow$ Lookup Table:

t[7]	$2^7 \bmod 1111$	1000
t[6]	$2^6 \bmod 1111$	0100
t[5]	$2^5 \bmod 1111$	0010
t[4]	$2^4 \bmod 1111$	0001

Tabelle 6-2: Lookup Table für eine parallele 4-bit Modularmultiplikation

$P' = 10001111 \Rightarrow S = 1111 + 0 \cdot 0001 + 0 \cdot 0010 + 0 \cdot 0100 + 1 \cdot 1000 = 10111$

Die Reduktion wird solange wiederholt, bis die Summe S eine n-bit-lange Zahl ist. Das bedeutet, dass bei der ersten Reduktion die Eingabe eine $2n$-bit-lange Zahl und die Ausgabe eine $n+\log(n+1)$-bit-lange Zahl ist. In der zweiten Iteration ist die Ausgabe eine $n+\log(\log(n+1)+1)$-bit-lange Zahl, usw. Mit jeder weiteren Reduktion wird das Ergebnis kleiner, da, wenn die Zahl $S > 2^n$ ist, mindestens einmal ein Teilprodukt, das größer oder gleich 2^n ist, durch eine Zahl, die kleiner als 2^n ist, ersetzt wird. Somit ist das Ergebnis der mehrfachen Reduktion schließlich eine n-bit-lange Zahl. Dabei gilt die folgende Ungleichung: $S < 2^n < 2M$. Diese Vorgehensweise ist korrekt, da die Werte in der Lookup Table kleiner als $M < 2^n$ sind, wobei das Gesamtergebnis aber im Intervall zwischen 0 und M-1 liegen muss. Um dies zu gewährleisten, wird am Ende des Algo-

rithmus ein Standardvergleich mit M mit anschließender Subtraktion (falls nötig) durchgeführt:

```
if S ≥ M then P = S – M; else P = S;
```

Das Ergebnis P ist kleiner als M und kongruent zu $X \cdot Y$ mod M. Demzufolge ist $P=X \cdot Y$ mod M. Der Algorithmus 4.3.1 liefert eine vollständige Beschreibung der Multiplikation mit anschließender Reduktion. Das i-te Bit von S wird definiert als s_i:

Algorithmus 6.1.1: Parallele Modularmultiplikation

```
Eingabe: X, Y, LookUp (M); (X, Y < M);
Ausgabe: P=X*Y mod M;
Methode:
(1)  l=length(X)
(2)  P'=parallel_binary_addition(x_{l-1}*2^{l-1}*Y,  x_{l-2}*2^{l-2}*Y,  ...,
     x_0*2^0*Y);
(3)  S = P';
(4)  while (S≥2^n) {
(5)    l=length(S);
(6)    S=parallel_binary_addition(s_{l-1}*LookUp[l-1],
       s_{l-2}*LookUp[l-2],...,s_n* LookUp[n],(s_{n-1}s_{n-2}... s_0));
(7)  }
(8)  if S ≥ M then P = S – M;
(9)  else P = S;
```

Der Algorithmus für die parallele binäre Addition (2), (6) wird im nächsten Kapitel erläutert.

In diesem Abschnitt wurde ein neuer Algorithmus für die parallele Modularmultiplikation unter Nutzung einer Lookup Table mit vorberechneten Werten dargestellt.

6.1.3 Komplexität der Parallelen Modularmultiplikation

An dieser Stelle wird die Zeitkomplexität des neuen Algorithmus untersucht. Die Vorberechnungsphase wird einmal für alle Modularmultiplikationen innerhalb einer modularen Exponentiation durchgeführt. Die Zeitkomplexität der Vorberechnung wird am Ende des Abschnitts erörtert. Die erste Phase (Multiplikation) besteht aus log$n$ Schritten, wobei die Zeitkomplexität jedes Schrittes gleich der Zeitkomplexität einer Addition ist. Vereinfachend wird in diesem Abschnitt 6.1.3 die Dauer einer Addition als eine Zeiteinheit betrachtet.

Die zweite Phase besteht aus zwei Teilen: Der Reduktion auf die Länge von n Bits und nachfolgendem Vergleich mit dem Modulus mit anschließender Subtraktion. Die Zeitkomplexität eines Vergleiches entspricht der Zeitkomplexität einer Subtraktion oder Addition. Demgemäß liegt die Zeitkomplexität einer Subtraktion und eines Vergleiches bei $T_{\text{Endergebnis}}$=2 Zeiteinheiten. Die Zeitkomplexität für die Reduktion ist

$$\begin{gathered} T_{\text{Reduktion}} = \log(n+1) + \log(\log(n+1)+1) + \log(\log(\log(n+1)+1)+1) + \ldots < \\ \log n + 1 + 2\cdot\log(\log(n+1)+1) < \log n + 1 + 2\log(\log n + 2) < \\ \log n + 1 + 2\log\log n + 4 = \\ \log n + 2\log\log n + 5 \end{gathered} \tag{6.3}$$

Dementsprechend ist die Zeitkomplexität der Phase 2

$$T_2 = T_{\text{Reduktion}} + T_{\text{Endergebnis}} = \log n + 2\log\log n + 5 + 2 = \log n + 2\log\log n + 7 \tag{6.4}$$

Die Zeitkomplexität der gesamten Modularmultiplikation ist gleich

$$T_{\text{Multiplikation}} = T_1 + T_2 = \log n + \log n + 2\log\log n + 7 = 2\log n + 2\log\log n + 7 \tag{6.5}$$

Demzufolge ist die asymptotische Zeitkomplexität

$$O(\log n) \tag{6.6}$$

Nachdem nun die Zeitkomplexität der Modularmultiplikation analysiert worden ist, wird die Zeitkomplexität der modularen Potenzierung berechnet. Es gibt mehrere Algorithmen für die modulare Potenzierung. Da sie alle eine ähnliche Zeitkomplexität besitzen [34], [50], wird als Beispiel der „left-to-right“ binäre Algorithmus verwendet (Algorithmus 5.1.1) [34], [50]. Nach der Vorberechnungsphase werden mehrere (mindestens n) Modularmultiplikationen durchgeführt:

Algorithmus 6.1.2: Modulare Exponentiation

```
Eingabe: A, B, M (A, B, M<2^n);
Ausgabe: P = A^B mod M;
Vorberechnung:
(1)  D = 2^(n-1);
(2)  for (int i=n; i≤2*n-1; i++) {
(3)    D = 2*D;
(4)    if (D>=M) {
(5)      D = D – M;
(6)    }
(7)    LookUp[i] = D;
(8)  }
Exponentiation:
(9)  S=1;
(10) for (int i=n-1; i≥0; i--) {
(11)   S=Parallele Modularmultiplikation(S,S, LookUp);
(12)   if (b_i=1) {
(13)     Parallele Modularmultiplikation(S, A, LookUp);
(14)   }
(15) }
```

Die Zeitkomplexität der Vorberechnung liegt bei O(n) Additionen. Die Modulare Potenzierung erfordert zwischen n und $2n$ Modularmultiplikationen. Die Zeitkomplexität jeder einzelnen Modularmultiplikation beträgt wiederum O(log n) Additionen. Dementsprechend liegt die asymptotische Zeitkomplexität einer Modularen Potenzierung bei

$$O(n + n \cdot \log n) = O(n \cdot \log n) \tag{6.7}$$

6.1.4 Optimierung

Im letzten Abschnitt wurde ein neuer paralleler Algorithmus für die Modularmultiplikation beschrieben und die Analyse seiner Zeitkomplexität durchgeführt. Dabei wurde die Zeitkomplexität des Grundschrittes (der Addition) als eine Zeiteinheit angenommen. Diese Annahme entspricht aber nicht der Realität, da das Zeitverhalten einer Addition von der Länge der Operanden abhängig ist. Selbst mit einem Carry-Look-Ahead-Addierer ist die Zeitkomplexität der Vorberechnung O($n \cdot \log n$) anstelle von O(n). Dies bringt jedoch keine Probleme mit sich, da die Zeitkomplexität des Algorithmus ohnehin bei O($n \cdot \log n$) liegt. Da aber jede Modularmultiplikation eine Zeitkomplexität von O($\log^2 n$) und nicht O($\log n$) besitzt, steigt die Komplexität des neuen Algorithmus auf O($n \log^2 n$). Um dies zu verhindern, wird keine normale Addition durchgeführt, sondern der so genannte Wallace Tree [86] benutzt. Der Wallace Tree verwendet Carry-Save-Addierer [43], [63], [42] anstelle von Standard- oder 2-zu-1-Addierern. Die Zeit-

komplexität eines Carry-Save-Addierers beträgt O(1). Somit ist die Zeitkomplexität einer Carry-Save-Addition von der Länge der Operanden unabhängig. Das Ergebnis einer Multiplikation mit dem Wallace Tree besteht aus zwei Zahlen. Diese Zahlen werden danach mit einem Carry-Look-Ahead-Addierer mit einer Zeitkomplexität von O(log$n$) summiert. Die asymptotische Zeitkomplexität eines Wallace Trees liegt bei O(log$n$). Demgemäß beträgt die Zeitkomplexität des Algorithmus 6.1.2 O($n$log$n$).

Eine solche parallele Addition kann in Algorithmus 6.1.1 verwendet werden.

6.1.5 Abschlussfolgerung

In diesem Kapitel wurde ein neuer paralleler Algorithmus für die Modularmultiplikation innerhalb der Modularen Potenzierung dargestellt. Der Algorithmus wurde für eine zeiteffiziente Implementierung konzipiert. Die Zeitkomplexität der Vorberechnungsphase liegt bei O(n·log$n$) und jeder Modularmultiplikation bei O(log$n$). Die Zeitkomplexität des Algorithmus beträgt damit O($n$log$n$). Diese Technik kann bei unterschiedlichen Anwendungen, wie „Public Key“-Kryptographie, Computerarithmetik und anderen, verwendet werden. Eine einführende Darstellung des in diesem Abschnitt erörterten parallelen Algorithmus für Modularmultiplikation ist in [20] gegeben.

7 Andere Algorithmen der Computerarithmetik

7.1 Parallele Multiplikation

7.1.1 Problembeschreibung

Eine der wichtigsten Operationen der Mathematik ist die Multiplikation. Diese wurde von Menschen vor Tausenden von Jahren erfunden. Sie ist eine der Grundoperationen und wird als solche in vielen Berechnungen durchgeführt. Es gibt viele unterschiedliche Algorithmen für die Multiplikation. Bei der klassischen Methode für die ganzzahlige Multiplikation werden die Bits eines Operanden mit der geschobenen Kopie des zweiten Operanden multipliziert und die Ergebnisse addiert [41], [90]. Wenn die Operanden aber sehr groß werden (z. B. die Anzahl n der Bits größer als 128 ist), dann benötigt der Algorithmus zu viele Additionen (n). Die beste, bekannte Zeitkomplexität für eine Addition liegt bei O(log$n$) [24], [90]. Dabei hat die beste Implementierung für einen Addierer eine AT-Komplexität von O($n$log$n$). Folglich beträgt die Zeitkomplexität dieses Algorithmus für die Multiplikation O($n$log$n$) und die AT-Komplexität (Fläche-Zeit) liegt bei O(n^2log$n$). Es gibt viele weitere Algorithmen für die ganzzahlige Multiplikation: Die Karatsuba Multiplikation [38], Toom Multiplikation [83], FFT Multiplikation, Schönhage-Strassen Multiplikation [72] und andere. Es handelt sich dabei um sequentielle Algorithmen, die zwar eine gute AT-Komplexität besitzen, jedoch ist die Zeitkomplexität der oben genanten sequentiellen Lösungen im Vergleich zu parallelen Methoden, die in Hardware implementiert werden können, relativ schlecht: Die Zeitkomplexität der Karatsuba Multiplikation liegt bei O($n^{\log 3}$); die Toom Multiplikation besitzt eine Zeitkomplexität von O(n·log$^{1/2}$n); Winograd [91], Cook [41], Zuras [92] und Knuth [41] erreichten weitere Verbesserungen der Multiplikation von Toom. Ein anderer interessanter Ansatz ist die Multiplikation unter Verwendung der Schnellen Fourier Transformation (FFT) [30], [32]. Einer der besten Algorithmen ist die Schönhage-Strassen Multiplikation mit einer Zeitkomplexität von ($n$log$n$loglog$n$).

Die Zeitkomplexität aller sequentiellen Algorithmen ist asymptotisch größer als O(log$n$). Die schnellsten parallelen Algorithmen besitzen eine Zeitkomplexität von O(log$n$) [74], [86], [90].

Eine der Möglichkeiten die Multiplikation zu beschleunigen, ist die Parallelisierung der Teiloperationen. Es existieren mehrere Algorithmen für die ganzzahlige Multiplikation: Die Wallace Tree Multiplikation [29], [76], [86] verwendet die Parallelisierung des klassischen Algorithmus für die Multiplikation unter Benutzung der redundanten Zahlendarstellung. Damit wird die optimale Zeitkomplexität von O(log$n$) erreicht, die Flächenkomplexität liegt aber bei O(n^2) – als Folge ist die AT-Komplexität O(n^2log$n$). Der parallele Algorithmus für die ganzzahlige Multiplikation in [74] hat die optimale Zeitkomplexität von O(log$n$) bei einer AT-Komplexität von O(n^2). Der beste parallele Algorithmus für die Multiplikation ist die Parallelisierung der Schönhage-Strassen Multiplikation [72], [90] mit der Zeitkomplexität von O(log$n$) und der Flächenkomplexität von O($n$log$n$loglog$n$). Die AT-Komplexität des Algorithmus beträgt O($n$log^2nloglog$n$). Dieser Algorithmus findet jedoch in praktischen Anwendungen keine Verwendung, da die Konstanten, die bei der O-Notation nicht betrachtet werden, zu groß sind.

Im nächsten Unterkapitel wird ein paralleler Algorithmus für die Multiplikation mit optimaler Zeitkomplexität von O(log$n$), sowie auch einer recht guten Flächenkomplexität von O(n^2/log^{2}n) vorgestellt [19]. Im Unterschied zu den oben genannten Algorithmen ist diese Methode aufgrund der kleineren Konstanten mit heutigen Schaltungstechnologien real implementierbar.

7.1.2 Darstellung des Algorithmus

Das Ziel des in dieser Arbeit entwickelten Algorithmus ist die Berechnung von $P=X\cdot Y$, wobei X und Y n-bit-lange Zahlen sind ($n\geq 1024$). Der Algorithmus soll zeitoptimal, also mit einer Zeitkomplexität von T=O(log$n$), und gleichzeitig flächeneffizient, mit einer Flächenkomplexität von O(n^2/log^2n), arbeiten. Die Konstanten sollen dabei klein bleiben, damit sich der Algorithmus für reale Anwendungen eignet.

Es werden zwei Variablen definiert, die für den gesamten Algorithmus von grundlegender Bedeutung sind: k=log^2n und t=log$n^{1/2}$.

Zu Anfang wird X in k-bit-große Intervalle unterteilt. Insgesamt existieren n/k Intervalle, die k-bit-lang sind. Jedes k-bit Intervall wird wiederum in t-bit-lange Intervalle unterteilt:

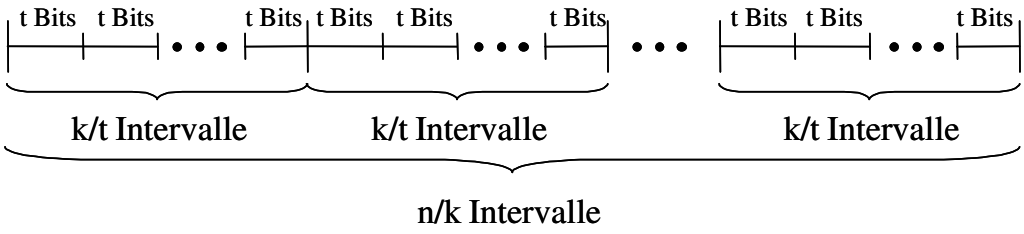

Abbildung 7.1.1: Unterteilung von *n* Bits in *t*-bit-lange Intervalle

Die neue Methode besteht aus drei Phasen:

Phase 1: Die Produkte von *Y* mit allen möglichen *t*-bit Kombinationen werden berechnet. Die Ergebnisse werden in Registern gespeichert.

Phase 2: Die Produkte von *Y* mit den *n*/*k* *k*-bit-langen Teilsequenzen von *X*, die der Aufteilung in *n*/*k* Intervalle entsprechen, werden berechnet.

Phase 3: Die Teilprodukte aus Phase 2 werden addiert.

Es wird gezeigt, dass bei der Implementierung jeder Phase höchstens $O(n^{1/2} + n/\log^2 n)$ Addierer (und Register) benutzt und höchstens $O(\log n)$ Schritt durchgeführt werden, was die Flächen- und Zeitkomplexität bestimmt.

Phase 1: 2^t Teilprodukte müssen berechnet werden. Dies geschieht unter Verwendung von 2^t Addierern vollständig parallel. Jeder Addierer berechnet das Produkt einer *t*-bit-langen Zahl mit *Y*. Diese Multiplikation wird mit der klassischen Multiplikation durchgeführt: log*t* geschobene Kopien von *Y* oder 0 werden entsprechend addiert.

Diese Operation erfordert $\log n^{1/2}$ Additionen der Länge $n+t < 2n$. Die Ergebnisse werden in Registern für die Verwendung in Phase 3 gespeichert.

In Phase 1 werden 2^t Addierer und 2^t Register eingesetzt.

$$A_1 = O(n \cdot 2^t) = O(n^{3/2}) \tag{7.1}$$

Die Zeitkomplexität ist durch die Anzahl der Additionen bestimmt und beträgt folglich

$$T_1 = O[(\log n)] \cdot T_{add} \tag{7.2}$$

wobei T_{add} die Zeitkomplexität einer nicht redundanten Addition ist.

Phase 2: In dieser Phase werden *n*/*k* Teilprodukte (Produkte von *Y* und *k*-bit Teilstrings von *X*) parallel berechnet. *n*/*k* Addierer berechnen jeweils ein Partialprodukt.

Jede einzelne dieser Berechnungen besteht aus k/t Additionen. Dabei werden die Operanden, deren Werte in Phase 1 berechnet wurden, aus den Registern ausgelesen. Die Prozesse für jede k-bit Sequenz sind rein sequentiell. Die Addierer starten mit dem Initialwert 0 des Zwischenergebnisses. Im ersten Schritt liest jeder Addierer den Wert eines Registers entsprechend der jeweils niederwertigsten t Bits eines k-bit-langen Intervalls von X aus. Dieser Wert wird zum Zwischenergebnis addiert. Danach werden die nächsten t Bits des k-bit Teiles betrachtet. Die entsprechenden Werte werden aus einem bestimmten Register entnommen, um t Bits geschoben und zu der Zwischensumme addiert. Auf demselben Weg werden vollständige k-bit-lange Teilsequenzen von X durch das k/t-fache Aufaddieren der entsprechenden t-bit Sequenzen berechnet. Das Ergebnis ist ein Produkt von Y und der k-bit-langen Teilsequenz von X.

Für die oben beschriebene Vorgehensweise werden n/k Addierer der Länge $2n$ benötigt. Also entspricht die Flächenkomplexität:

$$A_2 = O(n^2/k)=O(n^2/\log^2 n) \tag{7.3}$$

und die Zeitkomplexität verhält sich entsprechend gleich:

$$T_2 = O(k/t)T_{add} = O[(\log^2 n/\log n^{1/2})]\cdot T_{add} = O[(\log n)]\cdot T_{add} \tag{7.4}$$

Bei der Flächenkomplexitätsanalyse handelt es sich um eine vereinfachte Analyse, denn alle n/k Addierer lesen aus den gleichen Registern. Dies erfordert mehrere zusätzliche Multiplexer mit deren Hilfe die Eingangswerte für die Addierer ausgewählt werden und kann auf den Registerausgängen zu hohem Fan-out führen. Allerdings bietet das im Abschnitt 3.3 eingeführte Komplexitätsmodell nicht die Möglichkeit die Komplexität der Multiplexer oder Fan-out zu berücksichtigen.

Phase 3: Alle n/k Teilergebnisse werden zu einem Ergebnis summiert. Diese Addition wird in Form eines binären Baumes durchgeführt, welcher durch $n/2k$ Addierer implementiert wird. Wie in Phase 2 wird jedes Teilprodukt entsprechend der dazugehörigen k-bit Teilsequenz von X geschoben.

Die Flächenkomplexität von Phase 3 wird durch die Anzahl der Additionen bestimmt. Jede Addition hat eine Länge von höchstens $2n$ Bits.

$$A_3 = O(n^2/2k) = O(n^2/\log^2 n) \tag{7.5}$$

Die Zeitkomplexität ist durch die Zahl der Additionen angegeben, die jeder Addierer durchführen muss. Sie ist identisch mit der Tiefe des binären Baumes.

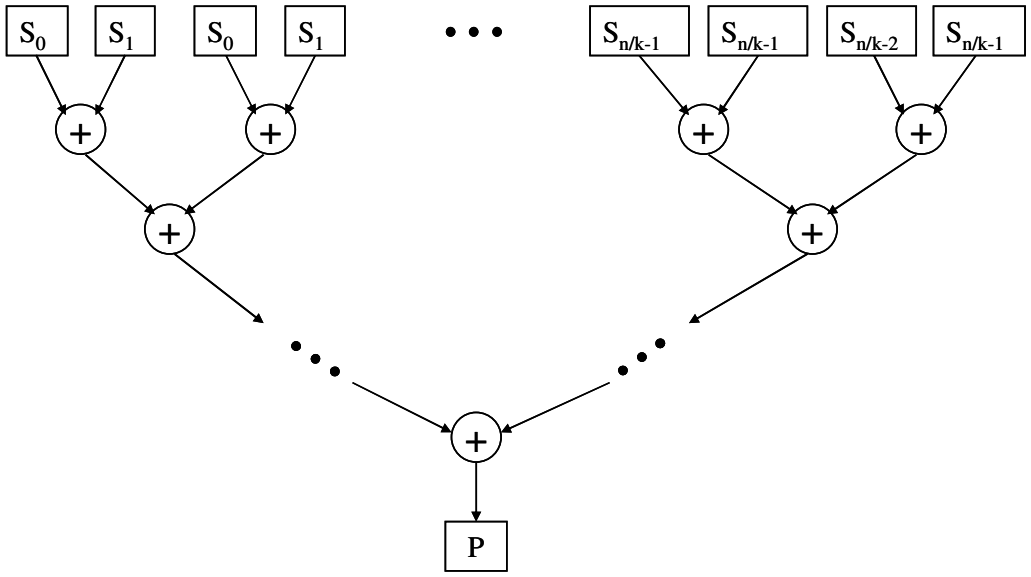

Abbildung 7.1.2: Binärer Baum der Addition der Teilergebnisse

$$T_3 = O[(\log n/k)T_{add}] \subseteq O[(\log n)T_{add}] \tag{7.6}$$

Das Ergebnis ist $P=X\cdot Y$.

Die Zeitkomplexität des ganzen Algorithmus ist offensichtlich

$$T_1+T_2+T_3 = O(\log n)\ T_{add} \tag{7.7}$$

Die Flächenkomplexität liegt bei

$$A_1+A_2+A_3 = O(n^2/\log^2 n+ n^2/\log^2 n+n^{3/2}) = O(n^2/\log^2 n) \tag{7.8}$$

Demzufolge beträgt die AT-Komplexität

$$AT = O[(\log n\cdot n^2/\log^2 n)T_{Add}] = O[(n^2/\log n)T_{add}] \tag{7.9}$$

Die Komplexität des Verfahrens hängt von der Addition ab. Wird eine Addition mit nichtredundanten Zahlen ausgeführt, beträgt die Zeitkomplexität einer Addition O(log(*n*)) (siehe Abschnitt 3.3). Somit liegt die AT-Komplexität bei

$$AT = O[(n^2/\log n)T_{add}] = O[(n^2/\log n)\log n] = O(n^2) \tag{7.10}$$

Im nächsten Abschnitt wird eine Möglichkeit dargestellt, wie die Addition und demzufolge das Verfahren durch die Benutzung von Carry-Save-Addierern um Faktor log*n* beschleunigt werden kann.

7.1.3 Optimierte Version des Algorithmus unter Verwendung von Carry-Save-Addierern

In diesem Abschnitt wird die Erweiterung des Algorithmus für die parallele Multiplikation durch redundante Zahlen dargestellt.

Die AT-Komplexität des neuen Algorithmus liegt bei

$$O(n^2/\log n) \tag{7.11}$$

Die Standardaddition erfordert eine Zeitkomplexität von

$$O(\log n) \tag{7.12}$$

Es wird ein Carry-Save-Addierer anstelle eines nicht redundanten Addierers eingesetzt. Das Produkt

$$P'=X\cdot Y \tag{7.13}$$

wird berechnet, wobei P' eine redundante Zahl ist:

$$P'=(S, C) \tag{7.14}$$

Sowohl die Zeit- als auch die Flächenkomplexität sind mit den Zeit- und Flächenkomplexitäten des Hauptalgorithmus in Abschnitt 7.1.2 identisch. Das Ergebnis besteht aus zwei Zahlen S und C. Zum Schluss werden S und C mit einem nicht redundanten Addierer mit einer Flächenkomplexität von $O(n)$ sowie einer Zeitkomplexität von $O(\log n)$ addiert.

Die Zeitkomplexität des hier entwickelten Algorithmus beträgt nun

$$O(\log n+\log n) = O(\log n) \tag{7.15}$$

Die Flächenkomplexität liegt bei

$$O(n^2/\log^2 n+n) = O(n^2/\log^2 n) \tag{7.16}$$

Entsprechend beträgt die AT-Komplexität

$$O(n^2/\log^2 n\cdot\log n) = O(n^2/\log n) \tag{7.17}$$

Die redundante Zahlendarstellung erhöht die Anzahl der Register, die für die Zwischenergebnisse benötigt werden, um den Faktor Zwei, denn jede redundante n-bit-lange Zahl wird als Summe von zwei nichtredundanten n-bit-langen Zahlen dargestellt. Eine Verdoppelung der Komplexität wird bei der O-Notation nicht berücksichtigt.

Allerdings ist in dieser Arbeit nicht nur die asymptotische Komplexität, sondern vor allem die praktische Realisierung wichtig. Hierbei ist es erforderlich, dass die Konstanten so klein wie möglich bleiben. Im nächsten Abschnitt wird eine Beispielrealisierung präsentiert.

7.1.4 Beispielrealisierung

In diesem Abschnitt wird ein *n*-bit Multiplizierer konstruiert. Wie in den Abschnitten 7.1.2 und 7.1.3 festgelegt, werden $k=\log^2 n$ und $t=\log n^{1/2}$ gewählt.

Im ersten Schritt werden 2^t *n*-bit-lange Carry-Save-Addierer eingesetzt. Jeder Carry-Save-Addierer berechnet eines der Produkte einer *t*-bit-langen Zahl mit *Y*. Das Ergebnis sind jeweils zwei *n*+*t*-bit-lange Zahlen. Diese werden in jeweils zwei *n*+*t*-bit-lange Register gespeichert. Für diese Berechnung sind 2^t *n*+*t*-bit-lange Carry-Save-Addierer und $2\cdot 2^t$ *n*+*t*-bit-lange Register erforderlich.

Bei Nutzung des Komplexitätsmodells aus Kapitel 2 liegt die Flächenkomplexität der ersten Phase bei

$$\begin{aligned} A_1 &= 2^t (n+t)\cdot 8 + 2\cdot 2^t (n+t)\cdot 4 \\ &= 16\cdot 2^{\log n^{1/2}} (n+\log n^{1/2}) \\ &= 16\cdot n^{1/2}(n+0{,}5\log n) \\ &= 16(n^{1{,}5}+0{,}5n^{0{,}5}\log n) \end{aligned} \tag{7.18}$$

Flächeneinheiten.

Die Zeitkomplexität beträgt

$$T_1 = t = \log n^{1/2} = 0{,}5\log n \tag{7.19}$$

In Phase 2 werden die Produkte von *Y* mit entsprechenden *k*-bit-langen Teilsequenzen von *X* berechnet. Dafür werden *n*/*k* (*n*+*t*)-bit-lange Carry-Save-Addierer verwendet. Jeder Carry-Save-Addierer addiert 2*k*/*t* Zahlen. Die Ergebnisse werden in jeweils zwei (*n*+*t*)-bit-langen Registern gespeichert. Die Flächenkomplexität liegt bei

$$\begin{aligned} A_2 &= n/k\cdot 8(n+t) + 4\cdot 2(n+t)\cdot n/k \\ &= 16n/\log^2 n(n+\log n^{1/2}) \\ &= 16n^2/\log^2 n + 16n\cdot 0{,}5\log n/\log^2 n \\ &= 16n^2/\log^2 n + 8n/\log n \end{aligned} \tag{7.20}$$

Flächeneinheiten.

Die Zeitkomplexität entspricht

$$T_2 = 2k / t = 2\log^2 n / \log n^{1/2} = 4\log n \tag{7.21}$$

Zeiteinheiten.

In Phase 3 werden die Ergebnisse aus Phase 2 mit einem Wallace Tree [86] addiert. Das Ergebnis besteht aus zwei 2*n*-bit-langen Zahlen. Diese werden mit einem Carry-Look-Ahead-Addierer addiert. Es existieren nach dem zweiten Schritt insgesamt 2*n*/*k* (*n*+*k*)-bit-lange Zahlen. Der Wallace Tree besteht aus 2*n*/*k* Carry-Save-Addierern, die jeweils bis zu 2*n*-bit-lang sind. Somit liegt die Flächenkomplexität

$$\mathrm{A}_3 = 2n / k \cdot 2n + 2 \cdot 24n = 4n^2 / \log^2 n + 48n \tag{7.22}$$

Flächeneinheiten.

Die Zeitkomplexität beträgt

$$\begin{aligned} T_3 &= \log_{1,5}(2n/k) = 2\log_{2,25}(2n/\log^2 n) < 2\log(2n/\log^2 n) \\ &= 2(\log 2 + \log n - \log\log^2 n) = 2\log n - 2\log\log^2 n + 2 \end{aligned} \tag{7.23}$$

Zeiteinheiten.

Somit entspricht die Gesamtflächenkomplexität

$$\begin{aligned} \mathrm{A} &= \mathrm{A}_1 + \mathrm{A}_2 + \mathrm{A}_3 \\ &= 16(n^{1,5} + 0{,}5n^{0,5}\log n) + [16n^2/\log^2 n + 8n/\log n] + 4n^2/\log^2 n + 48n \\ &= 20n^2/\log^2 n + 16n^{1,5} + 8n^{0,5}\log n + 48n + 8n/\log n \end{aligned} \tag{7.24}$$

Flächeneinheiten.

Die Zeitkomplexität der Multiplikation liegt bei

$$\begin{aligned} T &= T_1 + T_2 + T_3 \\ &= 0{,}5\log n + 4\log n + 2\log n - 2\log\log^2 n + 2 \\ &= 6{,}5\log n - 2\log\log^2 n + 2 \end{aligned} \tag{7.25}$$

Zeiteinheiten.

Dementsprechend beträgt die AT-Komplexität des Verfahrens

$$\begin{aligned} AT &= (20n^2/\log^2 n + 16n^{1,5} + 8n^{0,5}\log n + 48n + 8n/\log n) \\ &\cdot 6{,}5\log n - 2\log\log^2 n + 2 \\ &< (20n^2/\log^2 n + 16n^{1,5} + 64n)\cdot 6{,}5\log n \\ &= 130n^2/\log n + 104n^{1,5}\log n + 416n\log n \end{aligned} \tag{7.26}$$

Flächen-Zeit-Einheiten.

An dieser Stelle wird ein Vergleich zu zwei anderen parallelen Algorithmen für die Multiplikation durchgeführt. Einer davon ist die Multiplikation mit einem Wallace Tree [86], die eine asymptotische AT-Komplexität von $O(n^2\log n)$ besitzt und als Standardmultiplikation in Hardware gilt. Die Zweite, die Multiplikationsmethode von Schönhage und Strassen [73], basiert auf der schnellen Fouriertransformation und hat die beste bekannte AT-Komplexität für die parallele Implementierung von $O(n\log^2 n\log\log n)$. Da einerseits die Hardwareimplementierung des Algorithmus von Schönhage und Strassen sehr kompliziert und mit dem in dieser Arbeit eingeführten vereinfachten Komplexitätsmodell nicht analysierbar ist, und es andererseits nicht angestrebt wird, das Verfahren von Schönhage und Strassen zu untersuchen, wird gezeigt, dass der in dieser Arbeit entwickelte Algorithmus für die gängigen Operandengrößen von bis zu 8192 Bits eine deutlich niedrigere AT-Komplexität besitzt, indem eine untere Schranke für die AT-Komplexität der Multiplikationsmethode von Schönhage und Strassen berechnet und dessen AT-Komplexität dem hier präsentierten Verfahren gegenübergestellt wird. Um die Komplexität dieser unteren Schranke zu berechnen, wird der Aufwand der Transformation der beiden Operanden in den Fourierraum und die Rücktransformation des Ergebnisses der im Fourierraum stattgefundenen Berechnungen ermittelt. Die AT-Komplexität dieses wichtigen Teils der Multiplikation [90] wird als untere Schranke betrachtet.

Das folgende Diagramm zeigt den Vergleich der Flächen-Zeit-Komplexität der in dieser Arbeit entwickelten Verfahren mit der Multiplikation mit einem Wallace Tree und dem Verfahren von Schönhage und Strassen.

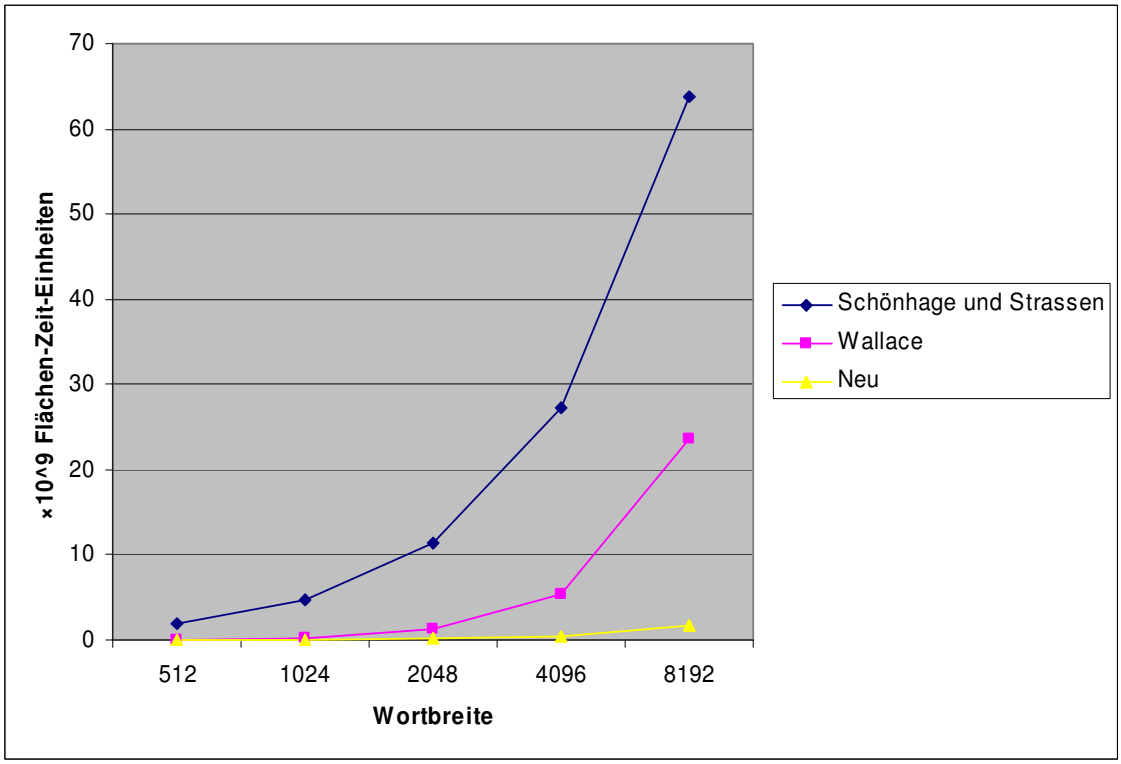

Diagramm 7-1 Vergleich der AT-Komplexität des neuen Verfahrens für Modularmultiplikation mit Wallace Tree und der Schönhage und Strassen Multiplikationsmethode.

7.1.5 Zusammenfassung

In diesem Kapitel wurde ein neues Verfahren zur zeitoptimalen Berechnung von $P=X\cdot Y$ für große, ganze Zahlen dargestellt. Das Verfahren hat eine Zeitkomplexität von $O(\log n)$ und eine Flächenkomplexität von $O(n^2/\log^2 n)$. Um eine zeitoptimale Ausführung zu erreichen, werden mehrere Operationen parallel ausgeführt. Im Unterschied zu anderen Algorithmen, die zwar teilweise eine bessere Flächenkomplexität besitzen, aber durch größere Konstanten und die komplexere Struktur schwer implementierbar sind, ist der neue Algorithmus sowohl theoretisch, als auch praktisch anwendbar. Die Hauptideen dieses Verfahrens wurden in [19] veröffentlicht.

8 Zusammenfassung und Ausblick

Mit der Entwicklung schnellerer Algorithmen für Computerarithmetik steigt die Performance kryptographischer Verfahren, wie z.B. bei RSA oder einem Verfahren mit Elliptischen Kurven. Damit erhöht sich der Durchsatz bei der Verwendung von Public-Key-Chiffrierverfahren. Diese Tatsache führt dazu, dass asymmetrische Chiffrierverfahren in Bereichen eingesetzt werden, in denen früher andere, schnellere aber nicht so sichere Kryptoalgorithmen verwendet wurden. Dies bedeutet mehr Sicherheit bei der Datenübertragung. Die Hauptoperation bei den meisten Chiffrierverfahren, die heute verwendet werden, ist die Modularmultiplikation, die innerhalb eines solchen Chiffrierverfahrens mehrmals mit dem gleichen Modulus ausgeführt wird. Die Hauptaufgabe dieser Arbeit war es, eine Reihe von neuen Algorithmen zu entwickeln, mit denen sich die Modularmultiplikation schneller und mit geringerem Flächenaufwand ausführen lässt.

In den ersten beiden Kapiteln wurde eine Einführung in die in dieser Arbeit zu behandelnden Probleme geboten. Dies geschah in mehreren Analyseschritten: Die Literaturrecherche zu Beginn der Arbeit sollte einen Einblick in die existierenden Algorithmen der Modularmultiplikation geben und ihre Vor- und Nachteile erläutern. Dabei wurden die Algorithmen nach ihrer Funktionsweise eingeordnet. Danach wurden die Grundideen der in dieser Arbeit neu entwickelten Algorithmen skizziert.

In Kapitel 3 wurde die Funktionsweise der einzelnen, in dieser Arbeit benutzten Funktionen erklärt, ein einfaches Komplexitätsmodell, mit dessen Hilfe die Zeit- und Flächenkomplexität der einzelnen Funktionen evaluiert werden kann, eingeführt und ferner durch den Vergleich verschiedener Technologien die praktische Relevanz dieses Komplexitätsmodells bestätigt.

Kapitel 4 stellt den Hauptteil dieser Arbeit dar. Hier wurden drei neue Algorithmen entwickelt:

Der Erste (Abschnitt 4.1) basiert auf der Modularmultiplikation von Montgomery. Im Gegensatz zu allen anderen Versionen des Verfahrens von Montgomery, die in jedem Schleifendurchlauf mehrere Additionen durchführen, wurde die Anzahl der Additionen pro Schleifendurchlauf im hier vorgestellten neuen Verfahren bis auf eine redu-

ziert. Dabei wurde die Carry-Save-Addition angewandt, was eine deutliche Verbesserung sowohl der Flächen- als auch der Zeitkomplexität des Verfahrens mit sich brachte. Die Anzahl der verwendeten Volladdierer des neuen Algorithmus ist mindestens um den Faktor Zwei besser als die der bisher bekannten Versionen des Montgomery Algorithmus.

Der zweite in dieser Arbeit vorgestellte Algorithmus (Abschnitt 4.2) behandelt die verschränkte Modularmultiplikation. Ebenso wie bei Montgomery wurde die Anzahl der Additionen per Schleifendurchlauf bis auf eine reduziert. Dabei wurde wiederum die Carry-Save-Addition benutzt, wodurch eine Verbesserung mindestens um den Faktor drei gegenüber den schon existierenden Verfahren erreicht wurde.

Der dritte Algorithmus für die Modularmultiplikation, der in Abschnitt 4.3 präsentiert wurde, macht die vollständig redundante Modularmultiplikation ohne Zeit- bzw. Flächenverlust möglich. Das Verfahren bietet neue Ansätze für die Entwicklung der Modularmultiplikation und der darauf basierenden Algorithmen und Protokolle.

Im nachfolgenden Abschnitt 4.4 wurden die neu entwickelten Algorithmen miteinander verglichen. Alle drei Algorithmen zeigen eine hohe Leistung bei geringer Flächenkomplexität. Als Fazit ist zu konstatieren, dass die Modularmultiplikation basierend auf der Veränderung eines Operanden bei Operandengrößen von 512 bzw. 1024 Bits trotz der höheren Flächenkomplexität im Vergleich zu den beiden anderen in dieser Arbeit entwickelten Algorithmen die beste Flächen-Zeit-Komplexität bietet. Bei steigender Operandengröße liefert hingegen die verschränkte Modularmultiplikation bessere Werte.

In Kapitel 5 wurde die redundante Modulare Exponentiation eingeführt. Diese ist eine der wichtigsten Anwendungen der Modularmultiplikation. Dabei wurde in Abschnitt 5.2 die in dieser Arbeit entwickelte Version der Modularmultiplikation von Montgomery in die Modulare Exponentiation eingebettet, in Abschnitt 5.3 die Modulare Exponentiation mit der neuen Version der verschränkten Modularmultiplikation ausgeführt und in Abschnitt 5.4 gezeigt, wie die Modularmultiplikation basierend auf der Veränderung eines Operanden für die Modulare Exponentiation benutzt werden kann. Jeder der drei oben genannten Abschnitte endet mit der Komplexitätsanalyse des jeweiligen Verfahrens. Danach folgt der Algorithmenvergleich: Die beste Flächenkomplexität bietet die Modulare Exponentiation mit der verschränkten Modularmultiplikation – die niedrigste Zeitkomplexität zeigt die Modulare Exponentiation mit der Modularmultiplikation basierend auf der Veränderung eines Operanden. Bei der AT-Komplexität

führt die Modulare Exponentiation mit der Modularmultiplikation basierend auf der Veränderung eines Operanden bei Operandengrößen bis 2048 Bits. Für längere Operanden ist der Einsatz der Modularen Exponentiation mit verschränkter Modularmultiplikation von Vorteil.

In Kapitel 6 wurde ein Algorithmus für die parallele Modularmultiplikation aufgezeigt. Dieser basiert auch auf der Benutzung einer Lookup Table mit vorberechneten Werten. Die Zeitkomplexität des Algorithmus liegt bei $O(\log n)$ und ist somit zeitoptimal.

Das folgende Kapitel 7 beschreibt ein Verfahren für die zeitoptimale Multiplikation (die Zeitkomplexität beträgt $O(\log n)$). Dabei liegt die Flächenkomplexität des Algorithmus bei $O(n^2/\log^2 n)$ und demzufolge die AT-Komplexität bei $O(n^2/\log n)$. Im Gegensatz zu den bisherigen Verfahren, die eine bessere asymptotische Komplexität aufweisen, aber durch höhere Konstanten in den praktischen Anwendungen kaum verwendbar sind, sind die Konstanten des neuen Algorithmus relativ klein. Sie sind vergleichbar mit denen eines anderen Algorithmus für zeitoptimale Multiplikation, dem Wallace Tree. Der Nachteil des Wallace Tree ist, dass dessen AT-Komplexität bei $O(n^2 \log n)$ liegt, was um den Faktor $\log^2 n$ schlechter, als das in dieser Arbeit präsentierte Verfahren ist.

Das Ziel dieser Arbeit war die Entwicklung neuer, besserer Algorithmen für Computerarithmetik und vor allem für die Modularmultiplikation. In Kapitel 4 wurden drei sequentielle Algorithmen für die Modularmultiplikation vorgestellt. Wie bei jeder wissenschaftlichen Arbeit entstehen durch die neuen Algorithmen auch neue Fragen, die als Grundlage für weitere Arbeiten dienen können: In dieser Arbeit wurde mit der Basis zwei gearbeitet. Es wäre denkbar, die neu entwickelten Algorithmen mit höheren Basen zu implementieren, so dass eine höhere Datenrate ohne Flächenverlust erreicht wird, was die AT-Komplexität der Verfahren verbessern würde. Die oben beschriebenen Algorithmen sind für das Produkt von Fläche und Zeit optimiert. Dabei ist die Flächenkomplexität variabel (sie hängt von der Operandengröße ab). Eine weitere Entwicklung könnte die Suche nach skalierbaren Implementierungen für eine vorgegebene Fläche (z. B. für einen FPGA) sein. Darüber hinaus wäre es interessant, herauszuarbeiten, in wieweit die Algorithmen zu parallelisieren sind. Besonders viele Möglichkeiten bietet die Modularmultiplikation mit der Veränderung eines Operanden. Die Erweiterung des Algorithmus um weitere Pipelinestufen könnte hier zu einer signifikanten Beschleunigung führen. Außerdem ist es sinnvoll zu analysieren, ob eventuell die Veränderung des

zweiten Operanden und des Modulus weitere Optimierungen ermöglichen. Ferner könnte es auch von Interesse sein zu untersuchen, ob eine Möglichkeit besteht, die Operanden zu splitten und somit die Zeitkomplexität zu verringern.

Referenzen

[1]. Acar, T., Kaliski, Jr., BS, and Koc, C., *Analyzing and Comparing Montgomery Multiplication Algorithms*, IEEE Micro, 16(3), pp. 26-33, June 1996.

[2]. Amanor, D. N.: *Efficient Hardware Architectures for Modular Multiplication*, Master Thesis, Uni Bochum, 2005.

[3]. Amanor, D. N., Bunimov, V., Paar, C., Pelzl. J., Schimmler, M.: *Efficient Hardware Architectures for Modular Multiplication on FPGAs*, The 15th International Conference on Field Programmable Logic and Applications, 2005.

[4]. AUSTRIAMICROSYSTEMS: *Digital Standard Cells Library*, http://asic.austriamicrosystems.com/databooks/.

[5]. Bajard, J., Didier, L., Kornerup, P.: *An RNS Montgomery modular multiplication algorithm*, IEEE Transactions on Computers, Vol. 47, pp. 766-776, July 1998.

[6]. Bajard, J.-C., Didier, L.-S., Kornerup, P.: *Modular Multiplication and Base Extensions in Residue Number Systems*, 15th IEEE Symposium on Computer Arithmetic, pp. 59-65, June 11 - 13, 2001 Vail, Colorado.

[7]. Bajard, J.-C., Imbert, L., Plantard, T.: *Improving Euclidean Division and Modular Reduction for some Classes of Divisors*, 37th IEEE Asilomar Conference on Signals, Systems, and Computers, 2003.

[8]. Barrett, P.: *Implementating the Rivest, Shamir and Aldham public-key encryption algorithm on standard digital signal processor*, Proc. of CRYPTO'86, Lecture Notes in Computer Science 263, Springer-Verlag, pp. 311-323, 1986.

[9]. Becker, B., Drechsler, R., Molitor, P.: *Technische Informatik. Eine Einführung*, Pearson Studium 2005, ISBN 3-8273-7092-2.

[10]. Berna, S., Batina, O. L., Preneel, B., Vandewalle, J.: *Hardware Implementation of a Montgomery Modular Multiplier in a Systolic Array*, Proceedings of the 10th Reconfigurable Architectures Workshop (RAW 2003), 2003.

[11]. Beuchat, J.-L., Muller, J.-M. : *Modulo M multiplication-addition: algorithms and FPGA implementation*, Electronics Letters, Volume: 40, Issue: 11, pp: 654- 655, 27 May 2004.

[12]. Blakley, G. R.: *A Computer Algorithm for Calculating the Product A·B modulo M*, IEEE on Computers, Vol. C-32, No. 5, pp. 497-500, May 1983.

[13]. Blum, T, Paar, C.: *High Radix Montgomery Modular Exponentiation on Reconfigurable Hardware*, IEEE Transactions on Computers, Vol. 50, No. 7, pp. 759-764, July 2001.

[14]. Blum, T., Paar, C.: *Montgomery Modular Exponentiation on Reconfigurable Hardware*, 14th IEEE Symposium on Computer Arithmetic (ARITH-14), pp. 70-77, 1999.

[15]. Booth, A.: *A signed binary multiplication technique*, Journal of Mechanics and Applied Mathematics, pp. 236–240, 1951.

[16]. Brickell, E.F., *A Fast Modular Multiplication Algorithm with Application to Two Key Cryptography*, Proc. of Crypto'82, pp. 51-60, Plenum Press 1983.

[17]. Bunimov, V., Schimmler, M., Tolg, B.: *A Complexity-Effective Version of Montgomery's Algorithm*, ISCA'02, Workshop on Complexity Effective Designs, May 2002, http://www.ee.rochester.edu:8080/~albonesi/wced02/.

[18]. Bunimov, V., Schimmler, M.: *Area and Time Efficient Modular Multiplication of Large Integers*, IEEE 14th International Conference on Application-specific Systems, Architectures and Processors, pp. 400-411, June 2003.

[19]. Bunimov, V., Schimmler, M.: *Efficient Parallel Multiplication Algorithm for Large Integers*. Euro-Par. Springer-Verlag, pp. 923-928, August 2003.

[20]. Bunimov, V., Schimmler, M.: *A Simple Algorithm For Time Optimal Modular Multiplication and Exponentiation*. The Proceedings of the 15th IASTED International Conference Parallel And Distributed Computing And Systems, November 2003.

[21]. Bunimov, V., Schimmler, M.: *High Radix Modular Multiplication of Large Integers Optimised with Respect to Area and Time*, The 2004 International Conference on VLSI, pp. 427-433, June 2004.

[22]. Bunimov, V., Schimmler, M.: *Two New Algorithms For Modular Multiplication*, Embedded Cryptographic Hardware: Methodologie & Architectures, pp. 39-56, 2004, ISBN: 1-59454-012-8.

[23]. Bunimov, V., Schimmler, M.: *Area-Time Optimal Modular Multiplication*, akzeptiert für International Journal of Computer Research.

[24]. Bunimov, V., Schimmler, M.: „*Completely Redundant Modular Exponentiation by Operand Changing*“, The 2005 International Conference on Computer Design, Las Vegas, June 2005.

[25]. Cheng, F-C., Unger, S. H., Theobald, S. H.: *Self-Timed Carry-Lookahead Adders*, IEEE Transactions on Computers, Vol. 49, NO. 7, pp. 659-672 Juli 2000.

[26]. Chevallier-Mames, B., Joye, M., Paillier, P.: *Faster Double-Size Modular Multiplication From Euclidean Multipliers*, Hardware and Embedded Systems – CHES 2003, vol. 2779 of Lecture Notes in Computer Science, Springer-Verlag, pp. 214-227, 2003.

[27]. Cho, K.-S., Ryu, Je-H., Cho J-D: *High-Speed Modular Multiplication Algorithm for RSA Cryptosystem*, Industrial Electronics Society, IECON '01. The 27th Annual Conference of the IEEE, vol.1. pp. 479-483, 2001.

[28]. Cilardo, A., Mazzeo, A., Romano, L., Saggese, G. P.: *Carry-Save Montgomery Modular Exponentiation on Reconfigurable Hardware in Design*, Automation and Test in Europe Conference and Exhibition Designers' Forum (DATE'04), pp. 206-211, February 16 - 20, 2004, Paris, France.

[29]. Ciminiera, L., Montuschi, P.: *Carry-Save Multiplication Schemes Without Final Addition*, IEEE Transactions on Computers VOL. 45, NO. 9, pp. 1050-1055, 1996.

[30]. Cooley, J. W., Tukey, J. W.: *An Algorithm for the machine calculation of complex Fourier series*, Mathematics of Computation 19, pp. 297-301, 1965.

[31]. Even, S.: *Systolic Modular Multiplication*, Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology table of contents pp. 619 – 624, 1990 ISBN:3-540-54508-5.

[32]. Gentleman, W. M., Sande, G.: *Fast Fourier transforms–for fun and profit*, AFIPS 1966 Fall Joint Computer Computer Conference, Spartan Books, Washington, pp. 563-578, 1966.

[33]. Großschädl, J.: *High-Speed RSA Hardware Based on Barret's Modular Reduction Method*, Workshop on Cryptographic Hardware and Embedded Systems, LNCS 1965, pp. 191-203, Worchester, USA, Springer-Verlag Berlin Heidelberg 2000.

[34]. Hamann, A.: „*Konzeption einer virtuellen Maschine für Algorithmen und Aufgabenstellungen der Kryptographie*“, Diplomarbeit, Technische Universität Braunschweig, 2003.

[35]. Hars, L.: *Long Modular Multiplication for Cryptographic Applications*, Workshop of Cryptographic Hardware and Embedded Systems, pp. 45-61, Cambridge, MA, USA, August 11-13, 2004.

[36]. Hong, J.-H., Wu, C.-W.: *Radix-4 Modular Multiplication and Exponentiation Algorithms for the RSA Public-Key Cryptosystem*, Asia and South Pacific Design Automation Conference 2000, pp. 565-570, Yokohama, Japan, January 25 - 28, 2000.

[37]. Joyner, D.: „*Coding Theory and Cryptography From Enigma and Geheimschreiber to Quantum Theory* ", Springer-Verlag Berlin Heidelberg New York, 1991, ISBN: 3-540-66336-3.

[38]. Karatsuba, A. A., Ofman, Y.: *Multiplication of multidigit numbers on automata*, Sovjet Physics Doklady 7 pp. 595-596, 1963.

[39]. Kim, S., Sobelman, G. E.: *Digit-Serial Modular Multiplication Using Skew-Tolerant Domino CMOS,* IEEE International Conference on Acoustics, Speech and Signal Processing, Vol. 2, pp. 1173-1176, 2001.

[40]. Kim, Y. S. Kang, W. S. Choi, J. R.: *Implementation of 1024-bit modular processor for RSA cryptosystem*, School of Electronic and Electrical Engineering, Kyungpook National University, 1370 Sankyok-Dong, Book-Gu, Taegu, Korea, http://www.ap-asic.org/2000/proceedings/10-4.pdf

[41]. Knuth, D. E.: *The Art of Computer Programming*, Vol. 2, Seminumerical Algorithms. Reading, MA: Addison-Wesley, 2nd printing, Nov. 1971.

[42]. Koc, C. K.: *RSA Hardware Implementation*, RSA Laboratories, RSA Data Security, Inc. August 1995, http://security.ece.orst.edu/koc/papers/reports.html.

[43]. Lang, H. W.: „*Algorithmen in Java*", Oldenbourg Wissenschaftsverlag GmbH, München 2003, ISBN: 3-486-25900-8.

[44]. Lee, S. C.: *Digital Circuits and Logic Design*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1976.

[45]. Leutbecher, A.: *Zahlentheorie. Eine Einführung in die Algebra*, Springer Verlag, 1991, ISBN: 3-540-58791.

[46]. Lim, C. H., Hwang, H. S., Lee, P. J.: *Fast Modular Reduction With Precomputation*, Proc. of JW-ISC'97, pp.65-79, Oct. 1997.

[47]. Marnane, W., Daly, A.: *Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic*, Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays, pp. 40-49, Monterey, California, USA, 2002.

[48]. McIvor, C., McLoone, M., McCanny, J. V.: *Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures*, 37th Annual Asilomar Conference on Signals, Systems and Computers, pp 379-384, November 9-12, 2003.

[49]. Meister, G.: *On an implementation of the Mohan-Adiga algorithm*, Advences in Cryptographie – EuroCrypt'90, Springer-Verlag, pp. 496-500, 1990.

[50]. Menezes et. al: *Handbook of Applied Cryptography*, CRC Press, 1997, ISBN: 0-8493-8523-7.

[51]. Mentor Graphics: "*ASIC Design Kit*", http://www.mentor.com/company/higher_ed/asic/

[52]. Mississippi State University: "*MSU Standard Cell Library*", http://www.erc.msstate.edu/mpl/distributions/scmos/

[53]. Montgomery, P. L.: M*odular Multiplication without Trial Division*, Mathematics of Computation, 44, pp. 519-521, 1985.

[54]. Morita, H.: *A Fast Modular-multiplication Algorithm based on a Higher Radix*, 9. CRYPTO 1989, Santa Barbara, California, USA, 371-386 Electronic Edition Springer LINK, pp. 387 – 399, 1989.

[55]. NEC User's Manual: "*CB-C8 Family VX/VM Type, 0,5µm Cell-based IC*", NEC Electronics (Europe), http://www.ee.nec.de/_pdf/A10245XJ5V1UM00.PDF

[56]. Nedjah, N., Luiza de Macedo Mourelle: „*Fast Hardware of Booth-Barrett's Modular Multiplication for Efficient Cryptosystems*, 18th International Symposium, Antalya, Turkey, November 3-5, 2003, Proceedings. LNCS 2869, Springer Verlag, pp. 27-34, 2003.

[57]. Omura, J. K.: *A public key cell design for smart card chips*. In International Symposium on Information Theory and its Applications, Hawaii, USA, November 27-30, 1990.

[58]. Orton, G.A., Roy, M.P., Scott, P.A.: *VLSI implementation of public-key encryption algorithms*, Advances in Cryptology - Crypto '86, Santa Barbara, California, United States, pp. 277-301, 1987.

[59]. Orup, H., Svendsen, E. and Andreasen, E.: *VICTOR - an efficient RSA hardware implementation*", Advances in Cryptology – EuroCrypt '90, Aarhus, Denmark, pp. 245 – 252, 1991.

[60]. Phillips, B.: *Modular multiplication in the Montgomery residue number system*, in Proc. 35th Asilomar Conference on Signals, Systems and Computers, pp. 1637–1640, Pacific Grove, CA, USA, 2001.

[61]. Phillips, B. J. and Burgess, N., *Implementing 1,024-bit RSA exponentiation on a 32-bit processor core*, in Proc. 2000 International Conference on Application Specific Systems, Architectures, and Processors, pp. 127-137, Boston, MA, USA, 2000.

[62]. Rivest, R., Shamir, A. and Adleman, L., *A method for obtaining digital signature and public-key cryptosystems*, Communications of the ACM 21, pp. 120–126, 1978.

[63]. Schiffmann, W, Schmitz, R.: "*Technische Informatik 1*", Springer-Verlag, Berlin Heidelberg New York 1992 and 1993. ISBN 3-540-56815-8 2. Auflage.

[64]. Schimmler, M., Bunimov, V.: *Verfahren und integrierte Schaltung zur Durchführung einer Multiplikation modulo M*. Deutsches Patent Nr. 10223853, 29.05.2002.

[65]. Schimmler, M., Bunimov, V.: *Method and Integrated circuit for carrying out a multiplication modulo M*, US-Patentanmeldung Nr. 10515810, 26.05.2003.

[66]. Schimmler, M., Bunimov, V.: *Verfahren und integrierte Schaltung zur Durchführung einer Multiplikation modulo M*. EU-Patentanmeldung Nr. EP1508087, 26.05.2003.

[67]. Schimmler, M., Bunimov, V.: *Fast Modular Multiplication by Operand Changing*. The International Conference on Information Technology ITCC 2004, pp. 518-524, April 5-7, 2004.

[68]. Schimmler, M., Bunimov, V.: *Reducing the Complexity of Modular Multiplication by Modification of One Operand*, Embedded Cryptographic Hardware: Design and Security – 2004, ISBN:1-59454-145-0.

[69]. Schmeh, K.: „*Die Welt der geheimen Zeichen*", W3L-Verlag, Herdecke, Dortmund 2004, ISBN: 3-937137-90-4.

[70]. Schmeck, H.: „*Analyse von VLSI-Algorithmen*", Spektrum Akademischer Verlag, Heidelberg-Berlin-Oxford, 1995.

[71]. Schmidt, B., Schimmler, M., Adi, W.: *Area Efficient Modular Arithmetic for Mobile Security*, The 2002 International Conference on Wireless Networks, Las Vegas, USA 2002.
[72]. Schneier, B.: „*Angewandte Kryptogrphie*“, Addison-Wesley, Bonn 1996, ISBN 3-89319-854-7.
[73]. Schönhage, A., Strassen, V.: *Schnelle Multiplikation großer Zahlen*, Computing 7, Springer Verlag, pp. 281-292, 1971.
[74]. Singer, B., Saon, G.: *An efficient algorithm for parallel integer multiplication*, Journal of Network and Computer Applications 19, pp. 415-418, 1996.
[75]. Sloan, K. R.: *Comments on A Computer Algorithm for Calculating the Product A·B modulo M*, IEEE Transactions on Computers, Vol. C-34, No. 3, pp. 290-292, March 1985.
[76]. Stelling, P. F., Martel, C. U., Oklobdzija, V. G., and Ravi, R.: *Optimal Circuits for Parallel Multipliers*, IEEE Transactions on Computers 47, pp. 273-285, 1998.
[77]. Takagi, Naomi: *A Radix-4 Modular Multiplication Hardware Algorithm for Modular Exponentiation*, IEEE Transactions on Computers, Vol. 41, No. 8, pp. 949-956, August 1992.
[78]. Takagi, N., Yajima S.: *Modular Multiplication Hardware Algorithms with a Redundant Representation and their Application to RSA Cryptosystem*, IEEE Transactions on Computers, Volume 4, Issue 7, pp. 887-891, July 1992.
[79]. Tang, S.H., Tsui, K.S. Leong, P.H.W.: *Modular Exponentiation using Parallel Multipliers*, Proceedings of the 2003 IEEE International Conference on Field Programmable Technology (FPT), Tokyo, pp. 52-59, 2003.
[80]. Tenca, A. F., Koc, C. K.: *A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm*, IEEE Transactions on Computers, Vol. 52, No. 9, pp. 1215-1221, September 2003.
[81]. Tenca, A. F., Todorov, G., Koc, C. K.: *High-Radix Design of a Scalable Modular Multiplier*, Workshop Cryptographic Hardware and Embedded Systems, LNCS 2162, 2001, Springer-Verlag Berlin Heidelberg, pp. 94-108, 2001.
[82]. Tenca, A.F., Koc,C. K.: *A Scalable Architecture for Montgomery Multiplication*, Workshop Cryptographic Hardware and Embedded Systems, eds., pp. 94-108, Aug. 1999.
[83]. Toom, A. L.: *The complexity of a scheme of functional elements realizing the multiplication of integers*, Sovjet Physics Doklady 3, pp. 714-716, 1963.
[84]. Toshiba: “*ASIC DATA BOOK TC300*” Toshiba Asic Databook.
[85]. Treuer, E.: “*Entwurf und Full-Custom-Implementierung eines RAM-Registerfilters für den Feldprozessor eines Befehlssystolischen Prozessorfeldes*”, Diplomarbeit, FH Flensburg, 1997.
[86]. Wallace, C. S.: *A Suggestion for a Fast Multipliers*, IEEE Transactions Electronic Computing, 13, pp. 14-17, 1964.
[87]. Walter, C. D.: *Logarithmic Speed Modular Multiplication*, Electronics Letters 30, No 17, pp. 1397-1398, 1994.
[88]. Walter, C. D.: *Still Faster Modular Multiplication*, Electronics Letters vol. 31, no. 4, pp. 263-264, 16th Feb. 1995
[89]. Walter C. D.: *Systolic modular multiplication*, IEEE Transactions on Computers, Volume 42, Issue 3, pp. 376 – 378, March 1993.
[90]. Wegener, I.: *Effiziente Algorithmen für grundlegende Funktionen*, B. G. Teubner, 1989.

[91]. Winograd, S.: *Arithmetic complexity of computations*, CBMS-NSF Regional conference Series in Applied Mathematics, Philadelphia, 1980. ISBN 0-89871-163-0. MR 81k:68039.
[92]. XILINX: "*Virtex-II Platform FPGAs: Complete Data Sheet*", http://direct.xilinx.com/bvdocs/publications/ds031.pdf.
[93]. Zuras, D.: *More on squaring and multiplying large integers*, IEEE Transactions on Computers Volume 43, Issue 8, pp. 899 – 908, 1994.