Hardware effiziente Algorithmen zur Beschleunigung der modularen Multiplikation

Jan Schröder

29. September 2006

Hiermit erkläre ich, dass ich diese Arbeit eigenständig verfasst und nur die angegebenen Quellen zur Hilfe genommen habe.

Jan Schröder

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Kryptographie ist eine Wissenschaft, die heute auf das Leben jedes Einzelnen wirkt. Historisch ist sie aus dem Interesse für unabhörbare Kommunikation von Staatsmännern, Diplomaten und vor allem dem Militär entstanden und diesen vorbehalten geblieben. Durch das Internet aber ist eine globale Nachfrage nach Kommunikationssicherheit eines jeden Nutzers entstanden. Die zuvor verwendeten "secret-key" Verschlüsselungsverfahren wie DES und heute AES, die darauf basieren, dass zwei Kommunikationspartner über einen gemeinsamen geheimen Schlüssel verfügen, den niemand sonst kennen darf, waren für die hohe Anzahl an geführten Kommunikationen mit ständig wechselnden Partnern ungeeignet.

Für dieses Problem haben Whitfield Diffie und Martin Hellman 1976 eine Lösung gefunden [1]: die "public-key-cryptography". Diese basiert auf einem Verfahren, das Daten asymmetrisch Verschlüsselt mit unterschiedlichen Schlüsseln zur Ver- und Entschlüsselung. Diese Schlüssel sind teilweise öffentlich, so dass kein geheimer Schlüsselaustausch notwendig ist.

1977 entstand daraus das RSA-Verschlüsselungsverfahren [2], was seitdem das wichtigste Verfahren für den Austausch geheimer Schlüssel für die symmetrischen Verfahren zwischen Kommunikationspartnern und die Erstellung sowie Verifizierung von digitalen Signaturen unter Emails und ähnlichem ist. Neben RSA haben sich wegen geringerer Schlüssellänge die Elliptische-Kurven-Kryptosysteme (ECCs) auf dem Gebiet der public-key-Kryptographie etabliert (siehe [3]).

Beiden Verfahren ist es gemeinsam, das sie als grundlegende und Effizienz bestimmende Rechenoperation die modulare Multiplikation verwenden.

Modularmultiplikation bedeutet nichts weiter als bei der Berechnung eines Produktes zweier ganzer Zahlen X und Y nur den ganzzahligen Rest bezüglich des Modulus' M zu bestimmen: $X \cdot Y \bmod M = X \cdot Y - \lfloor \frac{X \cdot Y}{M} \rfloor \cdot M$.

RSA und ECCs arbeiten auf endlichen Körpern oder Gruppen. Dies hat zur Folge, dass die Ergebnisse von Operationen beschränkt sein müssen - es wird modulare Arithmetik benötigt. Der Vorteil von modularer Arithmetik ist gerade ihre kryptographische Sicherheit. Die Modularmultiplikation auf großen Gruppen beispielsweise ist mit einer Einwegfunktion vergleichbar. Denn es dauert im Vergleich zur Berechnung eines modularen Produktes lange, aus dem Ergebnis die Operanden zu bestimmen.

Dennoch sind sehr große Zahlen für die Verschlüsselung nötig, um heutige Kryptosysteme im Sinne der Kryptoanalyse sicher zu halten. Für RSA z.B. werden Schlüssel der Länge 1024 Bit benötigt. Bald wird die Schlüssellänge schon 2048 Bit betragen, denn auch die Möglichkeiten der Kryptoanalyse - also des Brechens von Kryptosystemen - werden durch neue Technologien immer vielfältiger. Für Elliptische-Kurven-Kryptosysteme sind Schlüssellängen von etwa 160 Bit gebräuchlich und bald schon werden es 190 Bit sein.

Standardmicrocontroller in eingebetteten Systemen wie z.B. Smartcards sind mit so großen Zahlen weitgehend überfordert und schaffen es nicht die zeitkritischen Anforderungen der kryptographischen Anwendung, die sie ausführen, zu erfüllen. Aus diesem Grund wird Spezialhardware entwickelt, die auf kleinster Fläche aufgebracht werden kann, und die modulare Arithmetik mit großen Zahlen beherrscht und so zur Beschleunigung des Kryptosystems dient. Auch große Serverarchitekturen, die viele kryptographische Operationen auf einmal berechnen müssen, profitieren von der Beschleunigung durch spezielle Hardware. Und die zentrale Operation dieser Kryptosysteme ist die modulare Multiplikation, welche also primär im Fokus der Bemühungen steht, modulare Arithmetik zu beschleunigen.

Die Theorie für spezielle Modularmultiplikationsalgorithmen entstand im Wesentlichen in den 80er Jahren des vergangenen Jahrhunderts. Aufgrund der Rasanz, mit der die public-key-Verschlüsselungsverfahren nun in den Fokus rückten, entwickelte sich die Theorie der Algorithmen für Modularmultiplikation entsprechend schnell und vielfältig.

Vor dieser Entwicklung bedeutete Modularmultiplikation von A, B modulo M die Bildung des Produkts $A \cdot B$ und anschließend die Division durch M um so den multiplikativen Rest bei der ganzzahligen Division bestimmen zu können. Mit dieser Technik als Grundlage waren die neuen public-key-Verfahren um mehrere Größenordnungen langsamer als die mit geheimen Schlüsseln. Bereits 1983 entstand aber ein neuer Algorithmus, der die Modularmultiplikation erheblich beschleunigen konnte - dieser ist unter der Bezeichnung "interleaved modular multiplication" bekannt. 1985 entstand der wohl wichtigste Algorithmus zur Modularmultiplikation von Montgomery, der auch dessen Namen trägt. Beide Methoden beruhen auf der Idee, die

Multiplikation der Operanden und die Reduktion durch den Modulus ineinander zu verschränken und so die Größe der Zwischenergebnisse klein zu halten. 1986 kam eine dritte Methode hinzu, die nach Paul Barrett benannt ist. Diese verfolgt den ursprünglichen Ansatz und beschleunigt diesen, indem die Division durch den Modulus nicht genau durchgeführt, sondern zeiteffizient abgeschätzt wird. All diese Arbeiten sind Grundlage vieler weiterer Veröffentlichungen gewesen und sind es noch heute. In diesen weiterführenden Arbeiten sind die ursprünglichen Algorithmen wiederholt optimiert worden, wobei nach dem heutigen Stand der Montgomery-Algorithmus die effizienteste Methode im Bezug auf das Produkt aus Flächen- und Zeitverbrauch der Architektur ist.

Diese Algorithmen werden in den folgenden Kapiteln ausführlich besprochen und auf die bekannten Optimierungen eingegangen werden.

Gegenstand dieser Arbeit ist die Beschleunigung der modularen Multiplikation durch effiziente Hardware, das heißt schnelle aber auch Platz sparende Hardware zur Modularmultiplikation.

Der hier entwickelte Algorithmus setzt auf der Idee des Montgomery-Algorithmus' auf: einer der Eingabewerte wird bitweise durchlaufen, das Zwischenergebnis wird erhöht und anschließend reduziert. Bei Montgomery erfolgt diese Reduktion durch eine Korrektur des Zwischenergebnisses, damit es durch 2 teilbar wird und anschließendes Rechtsschieben. Dies ist eine sehr effiziente Methode, wenn sie sich auch noch verbessern lässt, was ja in zahlreichen Arbeiten geschehen ist. Es entsteht aber grundsätzlich eine kausale Abhängigkeit: das Ergebnis wird erstellt - die Korrektur wird vorgenommen - das Ergebnis wird reduziert. Die erste Abhängigkeit kann aufgelöst und in einem Schritt durchgeführt werden, wie wir später sehen werden. Doch die Nächste - das Warten auf die Reduktion des Ergebnisses - bleibt. Dieser Verzögerung werden wir entgehen, indem wir einen Algorithmus entwickeln, der mit einer Pipeline arbeitet, die in jedem Takt neue Eingabewerte aufnehmen kann. Dies wird dadurch möglich, dass die Reduktion grundsätzlich erfolgt, ob das Ergebnis durch 2 teilbar ist oder nicht. Die eventuell abgetrennten Daten werden gespeichert und zu einem späteren Zeitpunkt erneut in die Pipeline eingespeist.

Wir werden diesen Ansatz sowohl einstufig - d.h. ein Register enthält sowohl die Ergebnisdaten, als auch die abgetrennten Daten - und zweistufig - d.h. in einer Stufe werden die Ergebnisdaten in einer zweiten die abgetrennten Daten gesammelt - verfolgen. Dabei werden einige der bereits bekannten Möglichkeiten zur Effizienzsteigerung zur Anwendung kommen: Einsatz von Lookuptables zur Abspeicherung komplexerer Daten oder Operationen, Verwendung von Carry-Save-Addierern zur Steigerung der zeitlichen Effizienz

sowie der Übergang auf höhere Zahlenbasen.

Der beste Algorithmus wird unter dem in dieser Arbeit angenommenen Komplexitätsmaß eine Flächen-Zeit-Komplexität von $2,75n^2$ erreichen, wobei der beste hier vorgestellte Algorithmus nach Montgomery bei $4,5n^2$ liegt - es wird also eine Beschleunigung der Modularmultiplikation um einen Faktor $1,6$ erzielt

Aufbau der Arbeit:

Kapitel 2 dieser Arbeit wird die Grundbegriffe der behandelten Thematik einführen: einige mathematische Grundlagen, Schreib- und Darstellungsarten, ein Bewertungsmaß für die Effizienz von Hardwarealgorithmen und eine kurze Einführung in eine Technik zur Verifikation von Algorithmen, mit deren Hilfe neu entwickelte Methoden folgender Kapitel als Korrekt bewiesen werden sollen.

In Kapitel 3 werden die oben genannten Algorithmen der modularen Multiplikation eingehend besprochen und deren Effizienz analysiert.

Kapitel 4 beinhaltet den Kern der Arbeit. Es werden mehrere Konzepte und zugehörige Entwürfe entwickelt und auf ihre Flächen-Zeit-Komplexität hin optimiert. Dabei werden, wie oben erwähnt, verschiedene Optimierungstechniken zum Einsatz kommen.

Abschließend werden in Kapitel 5 Details zur Realisierung, die Ergebnisse und mögliche Ansatzpunkte für weitere Bemühungen zur Optimierung besprochen.

Kapitel 2

Grundbegriffe

2.1 Mathematik

Definition (Einwegfunktion):
Eine Einwegfunktion ist eine mathematische Funktion, die im Sinne der Komplexitätstheorie "schwer" umzukehren ist. D.h. eine Funktion f ist dann eine Einwegfunktion, wenn ihr Funktionswert $f(x)$ einer Eingabe x einfach und schnell zu berechnen ist, die Umkehrung $f^{-1}(f(x))$ jedoch im Verhältnis dazu nur mit hohem Aufwand zu berechnen ist. Insbesondere wird eine Funktion, deren Funktionswert in polynomieller Zeit zu bestimmen ist, die Umkehrung aber nicht, als Einwegfunktion bezeichnet.

Definition (modulo/ div):
Die mathematischen Operationen div und mod sind wie folgt definiert:
Sei $a \in \mathbb{N}_0$ und $m \in \mathbb{N}$,

$$a \operatorname{div} m \stackrel{def}{=} \lfloor \tfrac{a}{m} \rfloor \cdot m$$
$$a \bmod m \stackrel{def}{=} a - \lfloor \tfrac{a}{m} \rfloor \cdot m$$

Definition (Kongruenz):
Seien $a, b, m \in \mathbb{N}$. Dann gilt:

$$\begin{aligned} a \equiv b (\mathrm{mod} m) &\Leftrightarrow \exists k \in \mathbb{Z} : a = km + b \\ &\Leftrightarrow a \bmod m = b \bmod m \end{aligned}$$

(Siehe auch [6, Kapitel 2.4.3]).

Satz (Kongruenz):

1. Die Kongruenz $\equiv$ ist eine Äquivalenzrelation.

2. $\equiv$ teilt die Zahlen in Restklassen ein (eine Restklasse besteht aus allen Elementen, die unter $\equiv$ äquivalent sind).

3. Die Restklassen unter der Relation $\equiv$ bilden einen Ring.

(Siehe auch [6, Kapitel 2.4.3]).

Definition (Bitvektor):
Es seien X, n natürliche Zahlen mit $X < 2^n$. Es seien x_0, x_1, ..., $x_{n-1} \in \{0, 1\}$ so gewählt, dass gilt:

$$X = \sum_{i=0}^{n-1} x_i 2^i$$

$x_0, ..., x_{n-1}$ sind eindeutig. Definiere nun $\{0,1\}^n \ni (x_{n-1}...x_0) := \sum_{i=0}^{n-1} x_i 2^i$ den zu X gehörenden Bitvektor.
Weiter sei $() := 0$ - der leere Bitvektor sei 0.

Definition (höhere Zahlenbasen):
Die obige Zahlendarstellung als Vektoren über einer Basis aus 2er-Potenzen lässt sich auch auf andere Zahlenbasen anwenden:

Es seien X, k, $b > 1$ natürliche Zahlen mit $X < b^k$. Es seien x_0, x_1, ..., $x_{k-1} \in \{0, 1, ..., b-1\}$ so gewählt, dass gilt:

$$X = \sum_{i=0}^{k-1} x_i b^i$$

Definiere $(x_{k-1}...x_0) := \sum_{i=0}^{k-1} x_i b^i$ den zu X gehörenden Vektor zur Basis b analog zur Definition zuvor.
Wir werden in dieser Arbeit ausschließlich Zahlenbasen verwenden, die 2er-Potenzen sind.

Satz (endliche Körper):
Es sei $M \in \mathbb{N}$, $G := \{0, 1, ..., M-1\}$, $X \in G$.
Es gilt:

1. G ist kummutativer Ring mit 1 (der Restklassenring) hinsichtlich modularer Arithmetik (siehe Satz über Kongruenz).

2. Ist X prim zu M (teilerfremd), so existiert X^{-1} in G mit $X \cdot_G X^{-1} = 1$

3. Ist M Primzahl, so ist G ein Körper hinsichltich der modularen Arithmetik. (Siehe auch [6, Kapitel 2.5.2, 2.5.3]).

2.2 Algorithmen

Wir werden in den folgenden Kapiteln Algorithmen untersuchen, die wir sowohl hochsprachlich als auch als Hardwareentwurf darstellen werden. Dazu wird in diesem Kapitel eine Übersicht gegeben, wie diese Darstellungen zu interpretieren sind.

2.2.1 Hochsprachliche Algorithmen im Pseudocode

Zur hochsprachlichen Darstellung von Programmen wird in dieser Arbeit Pseudocode verwendet. Das heisst einfache, Programmiersprachen ungebundene Codekonstrukte, die lediglich der Veranschaulichung dienen und keine reale Implementierung darstellen (Gegenstand dieer Arbeit ist ja die Entwicklung von effizienter Hardware). Aus Gründen der Vereinfachung wird jeglicher Code auch auf getypte Variablen verzichten. In vielen Fällen werden die Variablen als Folgen von Werten verwendet werden, um den Datenfluss zu veranschaulichen und Missverständnisse bei der Datenabhängigkeit zu vermeiden. Beispiel: $A_1 := B_0;\ B_1 := A_0;$. Die Werte A und B werden ausgetauscht.
Folgende Auflistung erwähnt sämtliche verwendete syntaktiche Konstrukte.

- **Zuweisung:** x:=e;
  Der Variablen x wird der Wert eines Ausdrucks e zugewiesen.

- **if-Anweisung:** if b then P_1; else P_2;
  In Abhängikeit des boolschen Wertes des Ausdrucks b wird entweder das Programm P_1 oder das Programm P_2 ausgeführt.

- **for-Schleife:** for i:=k to l do P;
  Die in dieser Arbeit verwendeten Schleifenkonstrukte beschränken sich auf for-Schleifen mit der Schrittweite 1. Das Programm P wird l-k+1 mal ausgeführt, wobei der Parameter i mit dem Anfangswert k startet und im letzten Durchlauf den Wert l annimmt. In einem Fall wird auch eine Schleife der Art "for i:=l downto k do P;" verwendet. Hierbei läuft die Variable i von einem Anfangswert l zum Endwert k, wobei i in jedem Schritt um 1 dekrementiert wird.

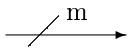
- **Parallele Ausführung:** co begin P_1; // P_2; // ... // P_m; end;
  Anweisungen die im co-Statement durch "//" getrennt werden, können nebenläufig ausgeführt werden. Die Programme P_1 bis P_m laufen also parallel zueinander.

- **Rückgabe:** return e;
  Die Rückgabe eines Ausdrucks e an eine höhere Instanz soll in den Programmen nur verdeutlichen, dass hier ein Ergebnis erzielt wurde. Die Anweisung "return e;" gibt also beispielsweise an, dass e als Ergebnis eines Algorithmus' errechnet wurde.

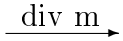
Die verwendeten Konstrukte wirden so gewählt, dass sie selbsterklärend und intuitiv zu verstehen sind.

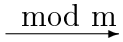
2.2.2 Hardwareentwürfe

Die zweite Darstellungsform für Algorithmen in dieser Arbeit ist die einer möglichen Implementierung in Hardware.
Es folgt ein Überblick über die hierbei verwendeten Hardwarekomponenten und ihrer Funktionsweise.

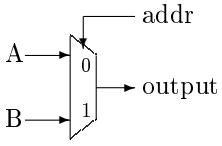


Leitungen:
Alle verwendeten Leitungen werden unabhängig von Ihrer Bitbreite durch einen Strich dargestellt. Die Bitbreite steht gegebenfalls mit einem Schrägstrich an der Leitung vermerkt. Die Leitung links hat eine Breite von m Bit. Der Pfeil am Ende einer Leitung gibt die Datenflussrichtung an.

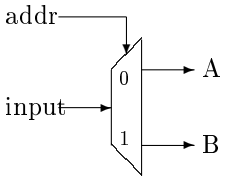




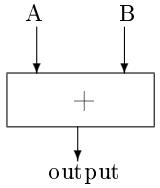
Operationen:
Es sei m eine 2er-Potenz, also $m = 2^k,\ k \in \mathbb{N}$. Da die mathematischen Operationen "div" und "mod" in Hardware einfach durch "Abschneiden" der letzten k Bits bzw. aller vorderen Bits bis auf die letzten k Bits zu realisieren ist, verzichten wir in den Entwürfen auf eine aufwendige Darstellung dieser Operationen.
Die Darstellung links gibt an, dass der auf der Leitung transportierte Wert am nächsten Gatter oder Knotenpunkt durch die Operation div oder mod entsprechend modifiziert wurde.



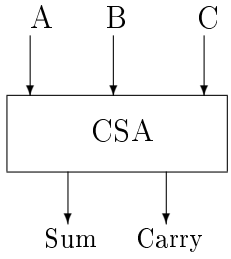
Multiplexer (MUX):
Ein Multiplexer hat zwei Eingänge A und B beliebiger Breite, die in Abhängigkeit von der Adresse addr gewählt werden. Im Beispiel links wird im Falle addr=0 der Eingang A auf den Ausgang geleitet, ist addr=1, wird B gewählt. Die in dieser Arbeit verwendeten Multiplexer sind alle von der Form, dass zwei Eingänge auf einen Ausgang geleitet werden.



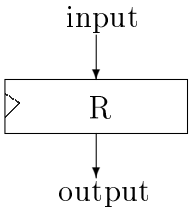
Demultiplexer (DMUX):
Ein Demultiplexer besitzt einen Eingang input und zwei Ausgänge A und B. in Abhängigkeit von addr - dem Adresseingang - wird entweder der Ausgang A oder der Ausgang B mit dem Eingang input getrieben. In dieser Arbeit werden Demultiplexer dazu verwendet, Ergebnisse wahlweise weiterzuverarbeiten oder auf einer zweiten Leitung auszugeben.



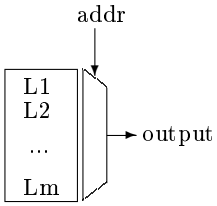
Addierer (ADD):
Ein Addierer hat zwei Eingänge A und B. Der Ausgang output entspricht dem Wert A+B.
Die Eingänge können beliebiger Bitbreite haben, was in den Entwürfen an den zuführenden Leitungen abzulesen ist. Der Ausgang output ist dann den Eingängen entsprechend breit.



Carry-Save-Addierer (CSA):
Ein Carry-Save-Addierer ist ein schneller Addierer, der aus vielen *parallel* geschalteten Volladdieren besteht (Siehe [5, Kapitel 5.5]). Die Eingabewerte A, B, C werden addiert und in Sum und Carry ausgegeben. Dabei gilt: Sum+Carry = A+B+C. ein Carry-Save-Addierer verarbeitet die Eingabe in konstanter Zeit unabhängig von der Länge der Eingänge. Die Eingänge können beliebige Bitbreite haben, was in den Entwürfen an den zuführenden Leitungen abzulesen ist. Der Ausgang output ist dann den Eingängen entsprechend breit.



Register (REG):
Register dienen der Speicherung von Daten über die Dauer eines Taktes hinweg. Mit der steigenden Flanke eines Taktsignals übernimmt das Register R den Wert, der an input anliegt. Dieser ist zu Beginn des nächsten Taktes am Ausgang output verfügbar.
Register können eine beliebige Breite an Bits annehmen - diese ist in den Entwürfen an der Bitbreite der angschlossenen Datenleitungen zu erkennen.
Register verfügen auch über einen zusätzlichen Eingang "Reset", der das Initialisiern des Registers auf '0' steuert.



Lookuptable (LUT):
Eine Lookuptable ist Bestandteil vieler Hardware-Architekuren und wird hier als zentrale Einheit zur Abbildung von Logik verwendet werden.
Die Lookuptable (LUT) links besitzt m Einträge L1 bis Lm. Diese werden über ein Adresswort addr adressiert und stehen dann am Ausgang zur Verfügung. Dabei ist zu beachten, dass addr mindestens $\log m$ Bits benötigt, um die LUT korrekt zu adressieren.

Noch einige Anmerkungen zu den Hardware-Entwürfen der folgenden Kapitel:

Um die Komplexität niedrig zu halten, werden wir darauf verzichten, Taktleitungen in die Schaubilder einzubringen. D.h. getakteten Bausteinen wie den Registern ist es nicht abzulesen, wann sie aktiv sind und wann nicht. Aus diesem Grund steht jedem Hardware Entwurf auch eine hochsprachliche Software variante zur Seite, der man einfach entnehmen kann, welche Schritte im Algorithmus aufeinander folgen und welche gleichzeitig, also im selben Takt ablaufen.

Weiterhin werden wir bei der Angabe der Busbreiten ungenau vorgehen. Die großen Busleitungen über die die Eingabeparamter X,Y und M und die resultierenden Zwischenergebnisse transportiert werden, werden wir grundsätzlich mit der Breite n versehen - der Bitbreite der Eingabewerte. In einer wirklichen Implementierung sind selbstverständlich ein paar mehr Bits nötig, denn die Zwischenergebnisse können ein Vielfaches der Eingabewerte betragen. Da wir aber modulare Multiplikation *großer* Eingabewerte, also Bitbreiten um 1024, realisieren wollen fallen diese wenigen Bit, seien es vier oder fünf, kaum ins Gewicht, so dass wir sie einfach ignorieren werden, um

die Entwürfe übersichtlicher zu halten. Bei Adressleitungen hingegen werden wir genau bleiben, um die Abläufe nachvollziehbar zu gestalten.

Außerdem werden wir die Ansteuerung der Register (das Laden einer '0' - das Laden eines Wertes) über Mutliplexer darstellen. Dies dient jedoch nur der Veranschaulichung - in realen Implementierungen wird das Laden einer '0' über den oben genannten "Reset"-Eingang eines Registers realisiert. Auch die Ausgabe eines Wertes wird nur der Anschauung halber über Demultiplexer dargestellt - auch dies ist in einer realen Implementierung überflüssig. Solche überflüssigen Multi- und Demultiplexer werden in der Berechnung des Flächenbedarfs einer Architektur nicht berücksichtigt.

2.2.3 Gegenüberstellung des Ripple-Carry- und des Carry-Save-Addierers

Eine besondere Beachtung bei den verwendeten Hardwarekomponeten verdient hierbei der Carry-Save-Addierer. Er bringt nämlich neben seiner hohen zeitlichen Effizienz auch einige Problemstellungen mit sich.

Der gewöhnliche Addierer (oder auch "Ripple-Carry-Addierer") besteht aus genausovielen Volladdierern, wie die Eingabewerte lang sind. Die einzelnen Volladdierer sind so verschaltet, dass der Carry-Ausgang des niederwertigsten Bits mit dem Carry-Eingang der folgenden Volladdiererzelle verbunden ist. Deren Carry-Ausgang bestimmt wiederum den Eingang der darauffolgenden Zelle usw. Dies bedeutet für den herkömmlichen Addierer eine Latenz, die der Anzahl an Volladdierern, also der Länge der Eingabewerte entspricht. Denn das Carry-Signal der letzten Addiererzelle wird läuft durch alle vorhergehenden Zellen. Dies ist der entscheidende Nachteil eines solchen Addierers für die Verwendung bei unserer Problemstellung. Denn wenn Eingabewerte sehr großer Bitlänge, wie sie in der Kryptographie vorkommen, verarbeitet werden sollen, bedeutet das eine hohe Durchlaufzeit für jede Addition (siehe [5, Kapitel 5.2]).

Der Carry-Save-Addierer hingegen besteht aus Volladdiererzellen, die parallel geschaltet sind also über keinerlei Verbindung zueinander besitzen. Der Carry-Save-Addierer (CSA) addiert drei Eingabewerte A, B, C zu zwei Ausgabewerten Sum und Carry, wie in Kapitel 2.2.2 dargestellt wird. Dabei entspricht Sum+Carry der Summe aus den drei Eingabewerten. Der Vorteil ist dabei die konstante Laufzeit des Carry-Save-Addierers unabhängig von der Länge der Eingabewerte. Der Nachteil ist in der redundanten Darstellung der Additionsergebnisse zu sehen, weil dies zum einen zusätzliche Register erforderlich macht und zum einen das Vorzeichen einer Zahl im Zweierkomplement nach einer Addition schwer erkennen lässt, weil dessen Propagation an die

höchstwertige Stelle viele Additionen dauern kann. Dies erschwert Vergleiche mit Zahlen in der redundanten Darstellung eines Carry-Save-Addierers (siehe [5, Kapitel 5.5]).

2.3 Ein Bewertungsmaß für Computeralgorithmen

In der Kryptographie ist die modulare Multiplikation eine zeitkritische Operation: ein Sicherheitszertifikat muss schnell verifiziert werden, damit eine Serveranfrage bearbeitet werden kann, Datenströme müssen schnell ver- und entschlüsselt werden usw. Daher ist die zeitliche Effizienz der zu grunde liegenden modularen Multiplikation sehr wichtig. Aber eine Hardware für Kryptosysteme muss auch flächeneffizeit sein, denn häufig ist die zur Verfügung stehende Hardwarefläche begrenzt (Smartcards oder ähnliches), oder man möchte möglichst viele Serveranfragen gleichzeitig verarbeiten und braucht dafür viele unabhängige Einheiten auf einem Kryptosystem nebeneinander. So ist man - und nicht nur für Kryptosysteme - übereingekommen, die Effizienz von Hardwarealgorithmen über das Produkt seiner Laufzeit und der verbrauchten Fläche zu bewerten. Diese Bewertung wird als das AT-Produkt bezeichnet (A=Area, T=Time).

Wir werden nun also einige Parameter festsetzen, aus denen man die Laufzeit bzw. die Fläche einer Hardware bestimmen kann. Dies ist eine heikle Angelegenheit, da beide Größen wesentlich durch die verwendete Hardware-Technologie bestimmt werden: ein Flipflop z.B. benötigt auf einem FPGA eine ganze logische Einheit auf einem ASIC hingegen nimmt es nur den Platz der für ein Flipflop nötigen Transistoren ein. Dennoch wollen wir hier ein allgemeines Bewetungsmaß entwickeln, das es uns ermöglicht, von der verwendeten Hardware unabhängig Algorithmen zu vergleichen. In der folgenden Tabelle sind die hierfür angenommenen Werte bzw. Wertintervalle aufgetragen. Diese rühren aus Beobachtungen verschiedener Architekturen her und werden in [24, Kapitel 3] fundiert. Die Spalten der Tabelle bezeichnen die Komponenten, wobei der Zusatz m bedeutet, dass es sich besipielsweise um einen Addierer handelt, der m-Bit Eingangswerte addiert. Der herkömmliche Addierer wird hier nicht aufgeführt, da er für die Implementierung einer Hardware keine Rolle spielen wird. Hier wird immer der schnelle Carry-Save-Addierer verwendet werden, da er durch seine konstante Laufzeit die Effizienz in hohem Maße steigert.

Die Operationen div und mod kosten weder Zeit noch Fläche, da sie einfach einer Verdrahtung ensprechen, die nur die vordern oder hinteren Bits

berücksichtigt.

Tabelle: AT-Komplexität von Hardwarebausteinen

	ADD_m	LUT_m	REG_m	MUX_m	$DMUX_m$	div /mod
A:	m	$0,125m$	$0,5m$	$0,25m$	$0,25m$	0
T:	1	1	0	0	0	0

Die hier angegebenen Werte sind, wie gesagt, nicht genau, sondern stark von der Implementationsplattform abhängig. Sie dienen im Folgenden eher der Orientierung als der genauen Abschätzung der Effizienz einer Methode. Sie sind aber im Mittel genau genug, um die richtige Tendenz der Komplexität abzuschätzen.

2.4 Verifikation von sequentiellen Computeralgorithmen

Dieses Kapitel gibt eine Einführung in die Verifikation von sequentiellen Algorithmen, da die entwickelten Algorithmen dieser Arbeit nicht nur plausibel gemacht werden sollen, sondern ihre korrekte Funktionsweise bewiesen werden soll.

Um genau zu sein: die entwickelten Algorithmen sind Hardwareentwürfe, in denen Komponenten durchaus *nebenläufig* arbeiten, aber es werden keine Datenabhängigkeiten auftreten, so dass die Methode auch als sequentiell aufgefasst werden kann.

Wir werden zu jeder Methode - auch wenn es sich in dieser Arbeit ausschließlich um die Entwicklung von Hardware geht - eine hochsprachliche Repräsentation bereitstellen, die zum einen das Verständnis erleichtern soll und anhand der wir den Algorithmus auf seine Korrektheit prüfen.

Die hier verwendete Methode zur Verifikation ist [25] entnommen. Wir werden hier nur eine knappe Einführung in die Methodik geben und darüber hinaus auf [25, Kapitel 9.4.2 - S.506-528] verweisen.

Die Beweismethodik verwendet folgenden Formalismus: wenn vor einem Programmschritt logisch die Aussage A gilt und der Programmschritt P ausgeführt wird, gilt darauf eine Aussage B, die A entspricht und um die geänderten Informationen im Schritt P modifiziert wird. Ein Beispiel: es gelte $\{x = 5\}$. Nach dem Programmschritt "y:=x+1;" muss dann offensichtlich $\{x = 5 \wedge y = 6\}$ gelten, wenn sich die Ausführung von "y:=x+1;" so verhält, wie es zu erwarten ist. Und genau auf solche Basisoperationen - wie beispielweise eine Zuweisung - lassen sich Beweisschritte anwenden, da man genau sagen kann, wie sich die Operation verhält. Dazu benutzt man Hoare-Tripel

in der oben angedeuteten Weise: ein Tripel $\{A\}P;\{B\}$, wobei A, B logische Ausdrücke und P ein Programm ist, ist genau dann gültig, wenn es sich durch eine Beweisregel ableiten lässt.

Wir können uns auf wenige Ableitungsregeln beschränken, da die Algorithmen simpel aufgebaut sind. Eine Regel, die oben bereits ihre Anwendung gefunden hat, ist die Zuweisungsregel, an deren Beispiel wir Ableitungsregeln einführen wollen: betrachte die Zuweisung "x:=e;". An einer Stelle im Programm gelte die Zusicherung Q. Dann besagt die Zuweisungsregel, dass nach Ausführung von "x:=e;" weiterhin Q gilt nur dass logische Aussagen über x abgeändert werden müssen, weil x fortan den Wert e hat. Oder umgekehrt: wenn vor "x:=e" Q gilt, wobei für den Wert von x e verwendet wird, so gilt nach der Zuweisung Q. Wir werden noch weitere Beweisregeln verwenden, die im Folgenden aufgezählt werden.

Der Beweis eines Programms S auf eine bestimmte Eigenschaft, erfolgt also durch die Vorgabe einer Anfangsbedingung $\{P\}$ (zumeist $\{true\}$ - eine allgemeingültige Aussage) und die korrekte Herleitung des Hoare-Tripels $\{P\}\ S;\ \{Q\}$, wobei Q der gewünschten Eigenschaft entspricht.

Für die Beweisregeln verwenden wir die englischen Begriffe, um mit der Literatur nicht in Konflikt zu geraten.

"assignment-rule":

$$\{Q[e/x]\}\ \text{x:=e;}\ \{Q\} \tag{2.1}$$

"consequence-rule":

$$\begin{gathered} P \to P_0, \{P_0\}\ \text{S;}\ \{Q_0\}, Q_0 \to Q \\ \Rightarrow \{P\}\ \text{S;}\ \{Q\} \end{gathered} \tag{2.2}$$

"sequential-composition-rule":

$$\begin{gathered} \{P\}\ S_1;\ \{R\}, \{R\}\ S_2;\ \{Q\} \\ \Rightarrow \{P\}\ S_1;\ S_2;\ \{Q\} \end{gathered} \tag{2.3}$$

"for-statement-rule":

$$\begin{gathered} \{P \wedge i = k\} \to \{Q_0\}, \{Q_0\}\ \text{S; i:=i+1;}\ \{Q_0\}, \{Q_0 \wedge i = l+1\} \to \{Q\} \\ \Rightarrow \{P\}\ \text{for i:= k to l do S;}\ \{Q\} \end{gathered} \tag{2.4}$$

Die "for-statement-rule" ist in [25] nicht aufgeführt. Sie ist direkt von der "Do-loop" abgeleitet und wird hier nicht weiter auf ihre Korrektheit untersucht.

Kapitel 3

Modulare Multiplikation

Dieses Kapitel gibt einen Einblick in das Feld der Algorithmen zur modularen Multiplikation. Die wichtigsten Algorithmen werden vorgestellt und ihre Optimierungsmöglichkeiten diskutiert. Weiterhin werden wir die effizientesten Methoden auf ihre Komplexität untersuchen, um so die Ergebnisse weiterer Kapitel dieser Arbeit vergleichen zu können. Eine umfassende Einführung zum Thema modulare Multiplikation für die Anwendung in public-key-Kryptosystemen geben [5](RSA) und [6].

3.1 Generalvoraussetzung

Für dieses Kapitel sei folgendes vorausgesetzt:

$$\begin{aligned} &\textit{Sei } n \in \mathbb{N}. \textit{ Sei } M \in \mathbb{N} \textit{ ungerade mit } 2^{n-1} < M < 2^n. \\ &\textit{Seien ferner } X, Y \in \mathbb{N} \textit{ mit } X, Y < M \end{aligned} \tag{3.1}$$

3.2 Die klassische Methode

Der klassische Ansatz zur Berechnung eines Produktes $X \cdot Y \bmod M$ orientiert sich einfach an der zu Grunde liegenden Mathematik: wie oben gesehen, gilt

$$X \cdot Y \bmod M = X \cdot Y - \lfloor \frac{X \cdot Y}{M} \rfloor \cdot M.$$

Für die modulare Reduktion genügt es also $\lfloor \frac{X \cdot Y}{M} \rfloor \cdot M$ zu bestimmen und vom Produkt $X \cdot Y$ zu subtrahieren. Dieser Ansatz hat jedoch zwei offensichtliche Nachteile:

1. Der Platzbedarf einer dafür erforderlichen Architektur ist groß, da die Zwischenergebnisse (z.B. $X \cdot Y$) eine Bitlänge von bis zu $2n$ haben.

2. Die zeitliche Komplexität ist hoch, da im Reduktionsschritt eine Division ($\frac{X \cdot Y}{M}$) berechnet werden muss - eine sehr aufwendige Operation.

Diese Methode hat heute keine praktische Relevanz mehr, da vor allem seit den achtziger Jahren des zwanzigsten Jahrhunderts zahlreiche neue Methoden zur modularen Multiplikation entwickelt wurden, die eine geringere Komplexität als haben als die klassische Methode.

3.3 Barrett's modulare Multiplikation

Paul Barrett's Methode zur modularen Multiplikation aus dem Jahre 1987 [12] ist der klassischen wohl am ähnlichsten: zunächst wird das Produkt $X \cdot Y$, darauf der Quotient $q = \lfloor \frac{X \cdot Y}{M} \rfloor$ bestimmt und schliesslich qM von $X \cdot Y$ subtrahiert. Bei der Berechnung von q wird jedoch auf Genauigkeit verzichtet, aber dafür die Division eingespart. Betrachte dazu folgende Überlegung:

$$\begin{aligned} \lfloor \frac{X \cdot Y}{M} \rfloor &= \lfloor \frac{X \cdot Y \cdot 2^{2n}}{M \cdot 2^{2n}} \rfloor \\ &= \lfloor \frac{\frac{X \cdot Y}{2^n} \cdot \frac{2^{2n}}{M}}{2^n} \rfloor \\ &\approx \lfloor \frac{\lfloor \frac{X \cdot Y}{2^n} \rfloor \cdot \lfloor \frac{2^{2n}}{M} \rfloor}{2^n} \rfloor \end{aligned}$$

Durch das Abrunden der Teilprodukte ergibt sich ein wesentlicher Vorteil: die Division durch einer 2er-Potenz ohne Rest entspricht einfach dem abschneiden der letzten niederwertigen Bits (z.B. der letzten n Bits bei einer Division durch 2^n). Der Quotient $\frac{2^{2n}}{M}$ erfordert weiterhin eine Division, er hängt aber nur von M ab und kann so vorausberechnet und solange verwendet werden, wie sich der Modulus nicht ändert. Der Fehler, der bei der Abschätzung des Quotienten q entsteht, ist höchstens zwei (siehe auch [6, Kapitel 14.3.3]). Die modulare Multiplikation nach Barrett lässt sich also durch folgenden Algorithmus ausdrücken:

Barrett - hochsprachlich

```
1  Z := ⌊2^2n / M⌋ ;   (vorausberechnet)
2  S := X · Y ;
3  q := ((S div 2^n) · Z) div 2^n ;
4  S := S − qM ;
5  return S ;
```

Für eine modulare Multiplikation werden nun also drei herkömmliche Multiplikationen mit Parametern der Länge n benötigt. Hinzu kommen die

Schiebeoperationen und eine Addition (gegebenfalls weitere Additionen zur Korrektur des Wertes q).

Barrett's Algorithmus ersetzt also den Schwachpunkt der klassischen Methode - die Division - durch eine Multiplikation und verbessert die Komplexität erheblich.

Der andere Nachteil des klassischen Ansatzes - die hohe Flächenkomplexität - bleibt jedoch auch bei Barrett ein Problem, weshalb diese Methode als *Hardware-Algorithmus* vom heutigen Standpunkt aus nicht konkurrenzfähig ist. Es gibt dennoch diverse Arbeiten, die auf Barrett aufbauen und durch verschiedene Verbesserungen die Komplexität verringern. Ein Vorteil von Barrett ist nämlich im Gegensatz zu komplizierteren Methoden, dass hier Standardmultiplikationen verwendet werden und so Barrett einfach auf Plattformen implementiert werden kann, die bereits über eine Multiplikation verfügen. Die Effizienz von Barrett ist also abhängig von der Effizienz der vorhandenen Hardware. Beispiele für Arbeiten, die sich mit dieser Methode befassen und sie durch Pipelining oder parallele Multiplizierer verbessern sind [13] und [14].

3.4 Verschränkte Modularmultiplikation (Interleaved)

Das zuvor angesprochene Problem der hohen Flächenkomplexität löst eine zweite Klasse von Algorithmen, die die Multiplikation und modulare Reduktion in vielen kleinen Schritten durchführt, und so die Effizenz steigert. Der erste dieser Algorithmen ist die 1983 veröffentlichte verschränkte Modularmultiplikation [7] (in englisch "interleaved modular multiplication").

Hier wird die Multiplikation schrittweise durchgeführt, was vergleichbar mit der schriftlichen Multiplikation aus dem Schulunterricht ist:

1. berechne $x_{n-1} \cdot Y$
2. multipliziere das Zwischenergebnis mit 2
3. addiere $x_{n-2} \cdot Y$ zum Zwischenergebnis
4. multipliziere mit 2
5. usw...

Gleichzeitig dazu also mit der Multiplikation *verschränkt* findet auch die Reduktion statt: nach jedem Additions- und anschließendem Multiplikationsschritt wird das Zwischenergebnis modulo M reduziert. Hierfür muss M

maximal zwei mal vom Zwischenergebnis subtrahiert werden (das Zwischenergebnis ist immer durch $3M$ beschränkt, wenn Y kleiner als M ist).

Interleaved - hochsprachlich

```
1  S:=0;
2  for i:= n-1 downto 0 do begin
3           S:=2*S;
4           S:=S+x_i·Y;
5           if S>M then
6                   S:=S-M;
7           if S>M then
8                   S:=S-M;
9  end;
10 return S;
```

Im obigen Code wird die Reduktion modulo M durchgeführt, indem S zweimal mit M verglichen und gegebenfalls M subtrahiert wird. Betrachte folgende Überlegung:

- Vor Schritt 3 gilt $S < M$. Dann gilt nach der Verdopplung $S < 2M$.
- Nach Schritt 4 gilt also $S < 3M$, da $Y < M$ vorausgesetzt wird (siehe GV 3.1).
- Durch die gegebenfalls durchgeführten Subtraktionen in Schritt 6 und 8 wird also wieder die Gültigkeit von $S < M$ nach einer Schleifeniteration garantiert, so dass die Aussage über die Dauer des Algorithmus' erhalten bleibt.

Dies garantiert also, dass die durchgeführte Reduktion also dem Ergebnis $S := S \bmod M$ entspricht. Man beachte dazu, dass durch den Schritt "S:=S-M;" durch die Struktur des Algorithmus' nicht nur M, sondern im Endeffekt Vielfache von M von der Gesamtsumme S abgezogen werden: wird diese Operation besipielsweise im ersten Schleifendurchlauf ausgeführt, so entspricht diese Operation der Subtraktion von $2^{n-1} \cdot M$ vom Endergebnis, weil im Laufe der Verarbeitung das Zwischenergebnis (und damit $-M$) noch $n-1$ mal verdoppelt wird. Dies stellt aber für die obige Kongruenz keine Beeinträchtigung dar, weil Additionen und Subtraktionen *beliebiger* Vielfacher des Modulus' das Ergebnis modulo M nicht verändern.

Die Flächenkomplexität wird durch diesen Ansatz im Vergleich zu Barrett oder der klasischen Methode also annähernd (ausgehend von einem großen n) halbiert. Ein Nachteil der verschränkten Modularmultipliaktion ist jedoch die hohe Zeitkomplexität: denn in jedem Schleifendurchlauf ist nicht nur der

für die Multiplikation notwendige Additionsschritt notwendig, sondern auch zwei Vergleiche des Zwischenergebnisses mit M. Diese sind aus Sicht einer zu entwickelnden Hardware für die Methode ungünstig, da es sich bei Vergleich um MSB-first-Operationen (most significant bit first) handelt, bei den darauf und davor ausgeführten Additionen aber um LSB-first-Operationen (least significant bit first). Das heisst, ein Vergleich muss immer warten, bis die Addition vollständig - bis zum höchstwertigsten Bit - ausgeführt wurde und so keine Effizienzsteigerung durch Pipelining in der Hardware möglich ist.

3.4.1 Optimierung der verschränkten Modularmultiplikation durch Carry-Save-Addierer, höhere Zahlenbasen und Lookuptabellen

Für diese Algorithmenklasse, die die Multiplikation und Reduktion nicht in einem Schritt durchführen, sondern in viele Teilschritte zerlegen, bieten sich zwei Optimierungsvarianten besonders an.

Die erste besteht aus dem Einsatz von Carry-Save-Addierern, die bereits in Kapitel 2.2.2 und 2.2.3 eingeführt wurden. Diese Addierer haben den großen Vorteil, dass sie Eingabewerte *beliebiger* Länge in einem Takt aufaddieren können. Dies erhöht die zeitliche Effizienz enorm gegenüber herkömmlichen Addierern, die mindestens logarihmischen Zeitaufwand mit der Länge der Eingabewerte benötigen. Der Nachteil des Carry-Save-Addierers ist jedoch die Art seiner Ausgabe: zwei statt einem Register müssen das Ergebnis aufnehmen, welches redundant dargestellt wird. Dies führt zum einen zu einem erhöhten Platzbedarf der Architektur, was aber durch die bessere Zeitkomplexität kompensiert werden sollte (für hohe Bitlängen). Andererseits stellt die redundante Darstellung ein Problem für den Einsatz bei der verschränkten Modularmultiplikation dar: denn das Ergebnis eines Vergleiches (die Bestimmung eines Vorzeichens) kann schwer ermittelt werden, ohne den Operanden in eine nicht-redundante Darstellung umzuwandeln (was den Zeitvorteil wieder aufheben würde). Dennoch ist es geldungen, Carry-Save-Addierer zur Beschleunigung der verschränkten Modularmultiplikation einzusetzen (siehe zum Beispiel [8], [9], [10], [11]).

Die zweite Optimierungsmöglichkeit besteht in der Verwendung von höheren Zahlenbasen bei der Darsellung der Eingabewerte. Zuvor sind wir grundsätzlich von einer Darstellung der Zahlen im Binärsystem ausgegangen, d.h. als Linearkombination der Vektoren $2^0, 2^1, 2^2, 2^3$, usw. mit Koeffizienten 0 oder 1. Wir werden im Folgenden höhere Zahlenbasen betrachten, wobei wir als Basis nur 2er-Potenzen betrachten. Zahlen im 4er-System z.B. sind Vek-

toren zur Basis $4^0, 4^1, 4^2, ...$ mit Koeffizienten $0, 1, 2, 3$ (siehe dazu Kapitel 2.1 Defintion "höhere Zahlenbasen").

Dadurch ändert sich die Darstellung eines Wertes von vielen Koeffizienten der Bitlänge 1, auf wenige Koeffizienten höherer Bitlänge. Für die bei der verschränkten Modularmultiplikation angewendete Vorgehensweise hat das die Konsequenz, dass pro "Schleifendurchlauf", also jeden Teilschritt der Ergebnisberechnung mehr Informationen verarbeitet werden müssen (aufgrund der größeren Koeffizienten), aber dafür weniger Teilschritte vonnönten sind, um das Endergebnis zu bestimmen (weniger Koeffizienten). Bei geschickter Wahl der Basis kann dies zu einer Steigerung der Effizienz führen (wenn der zusätzliche Platz- und Zeitaufwand für die Verarbeitung der Koeffizienten geringer ist als der zugewonnene zeitliche Vorteil). Bei den meisten Archtitekturen ist hier durch einen Übergang von der Basis 2 auf die Basis 4 eine Optimum zu erzielen. [9], [8] und [11] wenden diese Optimierung auf die verschränkte Modularmultiplikation an.

Eine neuere Methode zur Optimierung, die aus der Entwicklung neuer Hardwarekonzepte entstand, ist die Verwendung von Lookuptabellen.

Diese Technik beruht darauf, dass sich in gewissen Hardwareimplementierungen Daten innerhalb der Architektur ablegen lassen und schnell darauf zugegriffen werden kann. Dadurch lassen sich komplexe Operationen einfach durch das Vorausberechnen und Abspeichern sämtlicher Ergebnisse und den schnellen Zugriff darauf realisiieren. Bei FPGAs zum Beispiel wird so fast jede Form komplexerer Logik realisiert. Der Vorteil dieser Technik ist, dass man für relativ wenig Flächen- und Zeitaufwand eine schwierige Operation vorausberechnen kann und auf deren Ergebnisse dann in *konstanter* Zeit zugreifen kann.

Optimierungen der verschränkten Modularmultiplikationen durch Lookuptabllen wurden in [10] und [11] implementiert.

3.5 Montgomery Modularmultiplikation

Die Montgomery-Modularmultiplikation von 1985 [15] ist eine effizientere Implementierung der Modularmultiplikation, die eine Technik verwendet, die als Montgomery-Reduktion bezeichnet wird.

Bei der Montgomery-Multiplikation wird nicht das Produkt $X \cdot Y \bmod M$, sondern $X \cdot Y \cdot R^{-1} \bmod M$ berechnet. In den hier vorgestellten Algorithmen wird $R = 2^n$ gelten, was die Implementierung besonders effizient macht. Der offensichtliche Nachteil bei der Montgomery-Modularmultiplikation ist, dass nicht das gewünschte Endergebnis, sondern ein Wert, der einen zusätz-

lichen Faktor R^{-1} in sich trägt, berechnet wird. Es sind also noch weitere Berechnungen nötig, bevor das Ergebnis der Multiplikation feststeht. Multipliziert man nämlich das Ergebnis $X \cdot Y \cdot R^{-1} \bmod M$ mit dem Faktor R^2 mittels des Montgomery-Algorithmus', erhält man den gewünschten Wert $X \cdot Y \bmod M$, da sich die Vielfachen von R in Zähler und Nenner aufheben (siehe dazu [6, Kapitel 14.3.2]). Außerdem kann die modulare Exponentiation, die bestimmende Operation des RSA-public-key-Kryptosystems, im "Montgomery-Raum" - also in der obigen Darstellung mit dem Faktor R^{-1} - durchgeführt werden: die Eingabeparameter werden mit einem zusätzlichen Faktor $R \pmod M$ versehen. Diese Struktur bleibt dann erhalten:

$$\begin{aligned} &\textit{Seien } A, B \in \mathbb{N} \\ (A \cdot R) \overset{MMM}{\cdot} (B \cdot R) &\equiv A \cdot R \cdot B \cdot R \cdot R^{-1} \bmod M \\ &= A \cdot B \cdot R^2 \cdot R^{-1} \bmod M \\ &= A \cdot B \cdot R \bmod M \end{aligned}$$

Der Faktor R bleibt also in den Zwischenergebnissen der Exponentiation erhalten, ohne dass sich sein Exponent ändert. Es genügt darauf eine einzige Korrektur des Ergebnisses der Exponentiation am Ende der Berechnung - nämlich die Division durch R, bzw. die Montgomery-Modularmultiplikation mit 1. Weitere Informationen zur Montgomery-Exponentiation finden sich in [6, Kapitel 14.6.1].

Der Vorteil des im Folgenden vorgestellten Algorithmus' liegt in seiner Effizienz: genau wie bei der verschränkten Modularmultiplikation wird der Eingabewert X schrittweise durchlaufen. In den Iterationen sind aber keine aufwendigen Vergleiche mit dem Modulus M nötig, sondern lediglich ein Test, ob das Zwischenergebnis gerade oder ungerade ist, also lediglich der Test eines Bits, nämlich des niederwertigsten (LSB). Weiterhin wird das Zwischenergebnis in jeder Iteration reduziert, so dass es eine Schranke eines geringen Vielfachen von M nicht überschreitet. Diese Effizienz macht die Montgomery Modularmultiplikation zu einem der wichtigsten Algorithmen in der Kryptographie.

In diesem Kapitel stellen wir den Montgomery-Algorithmus und eine Implementierungsvaritante vor und untersuchen die beiden Methoden hinsichtlich ihrer AT-Komplexität. Der erste Algorithmus entspricht der klassischen Version von Montgomery: der Paramter X wird vom niederwertigsten bis zum höchstwertigen Bit durchlaufen und so schrittweise mit dem Parameter Y multipliziert. In jedem Schritt wird das Zwischenergebnis S durch 2 dividiert und so in seiner Länge reduziert. Um Datenverluste auszuschliessen, wird dabei auf ein ungerades Zwischenergebnis der Modulus addiert (was

hinsichtlich der Wertigkeit von S mod M irrelevant ist). Da der Modulus als ungerade vorausgesetzt wurde, wird durch diese Addition das Zwischenergebnis zu einer geraden Zahl, so dass die Division durch zwei ohne Rest aufgeht. Die Berechung S mod 2 ist einfach der Test des niederwertigsten Bits von S (s_0).

Montgomery 1 - hochsprachlich

```
1  S:=0;
2  for i:= 0 to n−1 do begin
3           S:=S+x_i·Y;
4           if S mod 2=1 then
5                    S:=S+M;
6           S:=S div 2;
7  end;
8  return S;
```

Da der Montgomery-Algorithmus zu den effizientesten Algorithmen für die modulare Multiplikation gehört, werden wir ihn und ebenso einige Varianten als Maß für die Effizienz neuer, in dieser Arbeit entwickelter Methoden verwenden. Aus diesem Grund ist der Montgomery-Algorithmus auch als mögliche Hardwareimplementierung vorgestellt. In Abbildung 3.1 ist der zugehörige Hardwareentwurf für die Basis 2 mit herkömmlichen Addierern dargestellt.

Die "loop control"-Einheit regelt den Datenfluss in Abhängigkeit des jeweiligen Programmschritts und der Schleifeniteration: im ersten Schritt "S:=0;" (siehe hochsprachliche Darstellung Zeile 1) werden Nullen aus einem externen Register in den Addierer geladen, um das Register S auf den Anfangszustand zu initialisieren (das Laden einer Null bzw. der Rest eines Registers ist von der verwendeten Architektur abhängig und wird bei der Flächenkomplexität im weiteren nicht berücksichtigt). In jeder der n folgenden Iterationen wird zunächst der Wert $x_i \cdot Y$ über den rechten Multiplexer in den Addierer geladen und zum Zwischenergebnis hinzu addiert. Das Resultat wird in S gespeichert und gleichzeitig wird das letzte Bit des Ergebnisses (s_0) verwendet, um gegebenfalls den Korrekturwert M über den Multiplexer zu adressieren. Dieser wird dann über den zweiten Addierer auf das Zwischenergebnis addiert. Darauf ist das Ergebnis kongruent zu 0 mod 2 und kann über die Operation "div 2" reduziert werden. Zuletzt wird das Resultat über den Demultiplexer an die übergeordnete Komponente ausgegeben (return S).

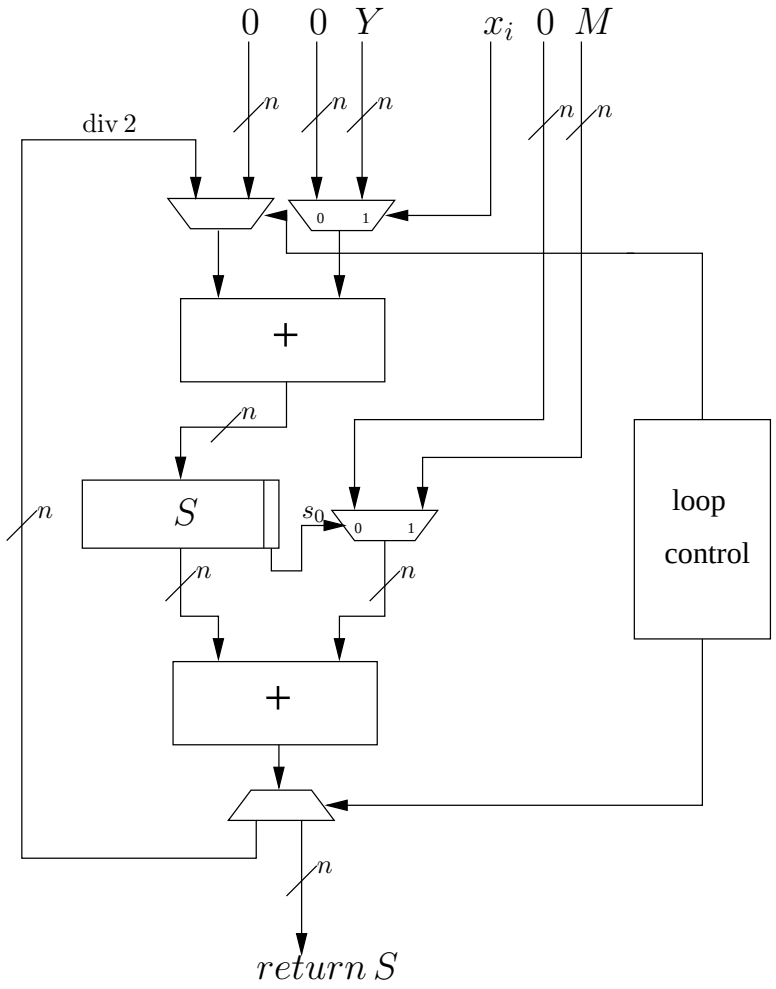


Abbildung 3.1: Montgomery 1

3.5.1 Optimierung der Montgomery Modularmultiplikation

Aufgrund seiner hohen Effizienz ist der Montgomery-Algorithmus am häufigsten unter den Algorithmen zur Modularmultiplikation zitiert, überarbeitet und optimert worden, um diese Effizienz noch zu steigern.

Eine Möglichkeit besteht wiederum in der Verwendung von Carry-Save-Addierern - genau wie bei der verschränkten Modularmultiplikation (siehe Kapitel 3.4.1). Der Montgomery-Algorithmus ist aber wesentlich besser geeignet, durch Carry-Save-Addierer beschleunigt zu werden, da hier keine Vergleiche über die ganze Länge mit dem Zwischenergebnis nötig sind, sondern nur ein Test des letzten (oder der letzten) Bits. Dadurch ist die oben an-

gesprochene Problematik von Carry-Save-Addierern für den Montgomery-Algorithmus irrelevant. Diese Optimierung bringt also direkten Nutzen für die Effizienz und wurde in zahlreichen Arbeiten implementiert ([16], [17], [18], [19], [22]).

Eine weitere Verbesserung, die ebenfalls schon für die verschränkte Modularmultiplikation besprochen wurde, ist die Verwendung höherer Zahlenbasen.

Nach dem gleichen Prinzip wie in Kapitel 3.4.1 angedeutet, lässt sich die Effizienz des Montgomery-Algorithmus' steigern. Die meisten Implementierungen mit höheren Zahlenbasen schlagen die Basis 4 als Optimum für die Flächen-Zeit-Komplexität vor ([16]).

Die nächste Verbesserungsmöglichkeit für den Montgomery-Algorithmus besteht - genau wie bei der verschränkten Modularmultiplikation in der Verwendung von Lookuptabellen.

Komplexere Operationen, wie zum Beispiel die Addition eines Eingabewerts und des Modulus gleichzeitig werden vorausberechnet und so die zeitliche Effizienz erhöht (siehe [17]).

Weniger gebräuchliche Optimerungen verwenden systolische Architekturen zur Implementierung des Montgomery-Algorithmus' ([20], [21]) oder schränken die Eingabeparamter (um genau zu sein: den Modulus) ein, um so die Hardware effizienter gestalten zu können ([23]).

3.6 Implementierungsvarianten des Montgomery-Algorithmus'

3.6.1 Implementierung des Montgomery-Algorithmus' mit Carry-Save-Addierern

Als nächstes Betrachten wir den Montgomery-Algorithmus in einer Implementierung mit Carry-Save-Addierern (Abbildung 3.2). Durch die Darstellung der Zwischenergebnisse in redundanter Form, werden nun zwei Register (S,C) benötigt. Der Test des Wertes (S,C) auf die Kongruenz 0 mod 2 lässt sich weiterhin durch das Bit s_0 durchführen, da das letzte Bit des Carry-Registers C grundsätzlich gleich 0 ist (siehe [5, Kapitel 5.5]). Nach der Korrektur werden beide Zwischenergebnisse S und C reduziert.

Analyse der AT-Komplexität des Montgomery-Algorithmus':

Wir untersuchen nun die Effizienz des ersten hier vorgestellten Montgomery-Algorithmus. Eine Schleifeniteration entspricht dem Durchlauf der Daten von oben nach unten durch die in Abbildung 3.2 dargestellte Hardware: durch die Verwendung von Carry-Save-Addierern kostet die erste Addiererstufe (CSA1) nur eine Zeiteinheit. Die Register übernehmen das Zwischenergebnis in der hier angenommenen Zeit 0 und auch der Multiplexer, der den Korrekturwert auswählt, wird hier mit der Zeit 0 veranschlagt. Die zweite Addiererstufe (CSA2) benötigt wieder eine Zeiteinheit, bis das Zwischenergebnis verarbeitet wurde. Die Operation "div 2" benötigt hingegen keine Zeit. Ein Schleifendurchlauf benötigt also zwei - die Schleife also insgesamt $2n$ Zeiteinheiten. Dies ist die Zeitkomplexität dieses Algorithmus'.

Für die Berechnung der Fläche wird die Einheit "loop-control" nicht berücksichtigt. Sie wird als extern angenommen. Weiterhin kann die erste Registerstufe eingespart werden, sofern man Zahlen zur Basis 2, wie hier angegeben, verwendet (wodurch aber die Taktdauer 2 Zeiteinheiten beträgt). Später werden wir auch höhere Zahlenbasen untersuchen, was die Register wieder notwendig machen wird. Weiterhin werden wir auch die Multi- und Demultiplexer zum Initialisieren bzw. zur Ausgabe der Register ignorieren, da sie, wie schon in Kapitel 2.2.2 erwähnt, lediglich der Veranschaulichung dienen und keine notwendige Funktionalität aufweisen. Es bleiben also 2 Multi- bzw. Demultiplexer, 2 Carry-Save-Addierer und 2 Register. Die Hardware benötigt also

$$2 \cdot 0,5n + 2 \cdot 0,25n + 2 \cdot n = 3,5n$$

Flächeneinheiten.

Das AT-Produkt errechnet sich also zu $7n^2$.

3.6.2 Eine Implementierungsvariante des Montgomery-Algorithmus' mit Lookuptable

Mit Hilfe von Lookuptables lässt sich die zeitliche Effizienz der obigen Methode steigern: anstatt auf die Addition von $x_i \cdot Y$ zu warten und anschliessend zu korrigieren, lässt sich dieser Vorgang in einem Schritt durchführen. Die Werte $0, M, Y, Y + M$ werden vorausberechnet und in einer Lookuptable gespeichert. In jeder Schleifeniteration wird nun der benötigte Korrekturwert inklusive Summand $x_i \cdot Y$ für den *nächsten* Schleifendurchlauf berechnet und als Adresse an die Lookuptable gelegt. Parallel dazu wird der aktuelle Wert, der aus der Lookuptable gelesen wurde, auf das Zwischenergebnis addiert.

In der hochsprachlichen Repräsentation dieser Methode stellt Z den kombinierten Wert aus Y und M dar. Dieser berechnet sich über die Parameter k und l, die in dem vorhergehenden Schleifendurchlauf gesetzt werden. k gibt

dabei das zu addierende Vielfache von Y an und entspricht x_i. Die Berechnung von l nimmt das Ergebnis von $(S + x_{i+1} \cdot Y) \bmod 2$ voraus: dazu ist die Berechnung von S selbst (denn das geschieht ja parallel, so dass das Ergebnis noch nicht zur Verfügung steht) und die Addition von $x_{i+1} \cdot Y$ nötig. Da nur das Ergebnis modulo 2 interessiert (also ob das Ergebnis gerade oder ungerade sein wird), benötigt man hier nur die letzten Bits, also $x_{i+1} \cdot y_0$ und $s_0 + z_0$. Wird das Ergebnis ungerade sein, wird l zu 1 berechnet, so dass im nächsten Schleifendurchlauf M zur Korrektur ausgewählt werden wird.

Montgomery 2 - hochsprachlich

```
 1  S:=0;
 2  k:=x_0;
 3  if x_0 · y_0=1 then
 4          l:=1;
 5  else
 6          l:=0;
 7  for i:= 0 to n−1 do begin
 8          Z:=k · Y + l · M;
 9          co begin
10                  S:=S+Z;
11                  S:=S div 2;
12          //
13                  k:=x_{i+1}
14                  l:=(x_{i+1} · y_0 + s_0 + z_0) mod 2;
15          end;
16  end;
17  return S;
```

Im Hardwareentwurf für diese Version des Montgomery-Algorithmus' (siehe Abbildung 3.3) geschieht diese Vorausberechnung in einer Adressierungslogik, die je nach Hardwareplattform als weitere Lookuptable oder eine Verschaltung von Gattern realisiert wird. Diese Logik wird hier also nicht weiter angegeben und ist durch die komponente "addr" dargestellt. Diese arbeitet parallel zu dem Addierer. Zu Beginn wird wieder das Ergebnisregister mit der "loop-control"-Einheit initialisiert, indem eine Null geladen wird. Das Register Z für den zweiten Summanden wird über die Lookuptable mit 0 oder Y (in Abhängigkeit von x_0) geladen. Die Ausgabe erfolgt wie im vorherigen Entwurf über einen Demultiplexer, der ebenfalls von der "loop-control"-Einheit gesteuert wird.

Um die Hardware effizient zu gestalten, werden wir wie zuvor Carry-Save-Addierer verwenden. Es kommen also ein Multiplexer, ein Demultiplexer und

ein weiteres Register hinzu, dafür ist der Zeitverbrauch stark verkleinert. Abbildung 3.4 zeigt einen entsprechenden Entwurf.

Analyse der AT-Komplexität des zweiten Montgomery-Algorithmus':
Der Vorteil dieser Implementierungsvariante ist zunächst nur in seinem geringeren Platzbedarf offensichtlich. Wie wir später sehen werden, ist aber bei höheren Zahlenbasen vor allem ein Geschwindigkeitsgewinn gegenüber des ersten Montgomery-Algorithmus' zu verzeichnen.

Ein Durchlauf des Schleifenrumpfes dauert in der Hardware zwei Zeiteinheiten, da zunächst die Lookuptable adressiert und der entsprechende Wert in das Register Z geladen werden muss. Darauf muss die neue Adresse für die Lookuptable berechnet werden (was wir hier auch mit einer Zeiteinheit annehmen) und parallel dazu das Register Z zu S addiert werden. Da der Algorithmus n Schleifendurchläufe benötigt, beträgt die Zeitkomplexität $2n$.

Für die Berechnung werden wir wieder die Einheit "loop-control" wieder ignorieren, da wir sie als externe Steuersignale annehmen. Die Register für A und die Einheit "addr" werden wir ebenfalls nicht mit einbeziehen, da sie im Vergleich zu dem Addierer oder den großen Registern praktisch nicht ins Gewicht fallen. Zuletzt sollen auch die veranschaulichenden Multi- und Demultiplexer an den Registern den Flächenbedarf nicht beeinflussen und werden ignoriert. Es bleiben also eine Lookuptable mit $4n$ Einträgen, 3 Register der Länge n und ein entsprechender Carry-Save-Addierer. Es folgt ein Flächenbedarf von 3 Flächeneinheiten.

Die AT-Komplexität dieser Montgomery-Variante beträgt also $6n^2$.

3.6.3 Implementierung des Montgomery-Algorithmus' mit höheren Zahlenbasen

Wir wenden nun die oben beschriebene Methode zur Optimierung mit höheren Zahlenbasen auf den Montgomery-Algorithmus an. Wir wählen eine allgemeine Zahlenbasis $b = 2^k$, eine 2er-Potenz. Es ändert sich also die Anzahl der pro Schleifendurchlauf verarbeiteten Bits von 1 auf k und die Schleifendurchläufe insgesamt auf n/k. Der Einfachheit halber nehmen wir grundsätzlich k teilt n an. Ist dies in einer realen Anwendung nicht gegeben, lässt sich die Länge der Eingabewerte einfach auf ein Vielfaches von k durch Hinzufügen von Nullen erweitern.

In jedem Schleifendurchlauf erhält der Algorithmus also k Bits, die einen Koeffizienten des Paramters X kodieren. Im i-ten Schleifendurchlauf wird der Koeffizient zum Basiswert b^i verarbeitet. Dieser wird wie in der vorgestellten Methode auch mit Y multipliziert und zum Zwischenergebnis addiert.

Dadurch wird das Ergebnis aber um bis zu k Bits länger, so dass die Reduktion ebenfalls k Bits erfassen muss. Dafür muss zunächst geprüft werden, ob das Zwischenergebnis kongruent 0 mod b ist, gegebenfalls korrigiert und danach reduziert werden. Zur Korrektur reicht nun nicht mehr M aus, sondern man benötigt Vielfache davon, abhängig vom Zwischenergebnis: ist das Zwischenergebnis beispielsweise kongruent 1 mod b, so müssen wir aM mit $a \in 0, 1, ..., b-1$ und $aM \equiv -1 \bmod b$ zur Korrektur addieren. In Entwurf 3.5 stellt daher eine Lookuptable die Werte $0, -1 \bmod b, -2 \bmod b, ..., -(b-1) \bmod b$ als Vielfache von M zur Verfügung. Da wir aber wie zuvor Carry-Save-Addierer verwenden, wird die LUT über die letzten Bits von C' und von S' adressiert. Würde man die letzten Bits der Register hier zunächst aufaddieren, ginge Zeit verloren, da eine zusätzliche Addiererstufe nötig wäre. So aber ist eine Lookuptable erforderlich, die mehr als nur b Werte enthält, da die Summen von S und C redundant dargestellt werden können: die 2 z.B. kann bei einer 2 im Carry-Register und einer 0 im Summenregister oder umgekehrt auftreten. Andere Summen haben noch mehr Darstellungen, so dass die LUT fast doppelt so groß ist als sie verschiedene Werte enthält. Das letzte Bit des Carry-Registers ist grundsätzlich 0, so dass dieses nicht berücksichtigt werden muss. Die LUT enthält also $2^{k+(k-1)} = 2^{2k-1}$ Werte. Eine weitere Lookuptable stellt Vielfache des Eingabewerts Y bereit, die dann über den Koeffizienten $(x_{(i+1)k-1}...x_{ik})$ adressiert wird.

Die Register sind so getaktet, dass vor einer Schleifenisteration das Register Z_1 den Wert aus der Lookuptable, der über den jeweiligen Koeffizienten von X adressiert wird, übernimmt. Im ersten Takt wird Z_1 auf das Zwischenergebnis (S, C) addiert und in (S', C') übernommen. Im zweiten Takt wird die zweite Lookuptable über die letzten Bits von (S', C') adressiert und der Wert in Z_2 gepseichert. Im letzten Takt werden Z_2 und (S', C') addiert, das Ergebnis geschoben und die LUT vor Z_1 neu adressiert.

AT-Analyse des Montgomery-Algorithmus mit höheren Zahlenbasen:

Der offensichtliche Vorteil des obigen Entwurfs ist, dass die Zahl der Schleifendurchläufe und damit die Zeitkomplexität proportional zum Logarithmus der verwendeten Zahlenbasis reduziert wird. Der Nachteil ist, dass durch die Lookuptables und die zusätzlichen Register der Flächenverbrauch steigt. Das AT-Produkt berechnet sich nun in Abhängigkeit der Basis $b = 2^k$.

Die zeitliche Komplexität beträgt - wie oben beschrieben - 3 Zeiteinheiten je Schleifendurchlauf. Es werden nurnoch n/k Schleifendurchläufe für die Multiplikation benötigt, so dass die Zeitkomplexität des Algorithmus' insgesamt $3\frac{n}{k}$ Zeiteinheiten beträgt.

Für die Fläche werden 6 Register, keine Multi- bzw. Demultiplexer, eine

Lookuptable mit 2^k (b) Einträgen und eine Lookuptable mit 2^{2k-1} Einträgen sowie 2 Carry-Save-Addierer benötigt. Insgesamt ergibt sich also ein Flächenverbrauch von

$$3n + 0 + 0,125 \cdot 2^k n + 0,125 \cdot 2^{2k-1} n + 2n$$

Flächeneinheiten. Die folgende Tabelle stellt die AT-Komplexität des Algorithmus' in Abhängigkeit von der Zahlenbasis (genauer dem Logarithmus k der Zahlenbasis) dar. Der zuvor berechnete Wert für die Basis 2 wird auch mit aufgeführt:

Tabelle: AT-Komplexität des Montgomery-Algorithmus' mit größeren Zahlenbasen

k=1	k=2	k=3	k=4
$7n^2$	$9,75n^2$	$10n^2$	$17,25n^2$

Diese Variante des Montgomery-Algorithmus' erfährt also keine Optimierung durch die Verwendung größerer Zahlenbasen. Das Problem sind nämlich die Lookuptables, die hier zusätzlich geschaffen werden müssen und die im Entwurf für die Basis 2 nicht nötig sind. Dadurch steigt der Flächenbedarf so sehr, dass der Zeitgewinn durch die Verarbeitung mehrerer Bits je Schleifeniteration zunichte gemacht wird.

3.6.4 Implementierung des 2. Montgomery-Algorithmus' mit höheren Zahlenbasen

Das gleiche Verfahren lässt sich auch auf die 2. Montgomery-Variante, die mit einer Lookuptbale arbeitet, anwenden. Wie zuvor wird diese Lookuptable Summen aus dem Eingabewert Y und dem Korrekturwert M enthalten. Durch größere Koeffizienten werden hier aber sowohl Vielfache von Y als auch von M und deren Kombinationen benötigt. Die LUT wird bei der Erweiterung auf höhere Basen nun stark anwachsen, da ihr Speicheraufwand quadratisch mit dem Logarithmus der Basis wächst: sei a ein Koeffizient von X aus der Menge $\{0, 1, ..., b-1\}$. Dann sind für das Ergebnis $(aY + S) \bmod b$, das in der Adressierungseinheit vorausberechnet wird, alle Werte aus $\{0 \bmod b, 1 \bmod b, ..., (b-1) \bmod b\}$ möglich, so dass zu a alle Werte $0, -1 \bmod b, ..., (b-1) \bmod b$ als Vielfache von M nötig sind, um das Ergebnis zu korrigieren. Diese müssen im Voraus auf die Produkte aY addiert werden und in der LUT gespeichert werden. Die LUT enthält also b^2 Einträge.

Der restliche Entwurf bleibt so wie im vorherigen Kapitel, denn breitere Datenbusse, größere Adressregister und geringfügig längere Register spielen im Flächenmodell keine dominierende Rolle. Zur Erinnerung: durch den Wechsel von der Basis 2 auf die Basis 4 ändert aich die Länge der Zwischenergebnisse um 2 Bit. Die Zwischenergebnisse haben aber Längen von mehreren hundert Bit, oder sogar tausend.

Abbildung 3.6 zeigt den Hardwareentwurf für die Basis 4. Höhere Zahlenbasen lohnen sich hier nicht, da die Lookuptable zu groß und die Adressiereiheit zu langsam werden. Aus Gründen der Vergleichbarkeit werden höhere Zahlenbasen in der Berechnung der AT-Komplexität dennoch mit aufgeführt.

AT-Analyse des 2. Montgomery-Algorithmus mit höheren Zahlenbasen: Durch eine größere Basis wird die Lookuptable des Entwurfs 3.6 zur bestimmenden Größe für den Flächenverbrauch. Durch den quadratischen Zuwachs wird sie schnell zu groß, um eine effiziente Funktionsweise zu gestatten. Obwohl der Entwurf konkret für die Basis 4 angegeben ist, werden wir die AT-Analyse allgemein durchführen. Die LUT benötigt dann b^2 Einträge (und entsprechend viele Berechnungnen im Voraus!). Die Anzahl der Register bleibt 3, ebenso benötigen wir weiterhin 1 Addierer und keinerlei Multiplexer. Die Fläche beträgt also

$$b^2 \cdot 0,125n + 3 \cdot 0,5n + 1 \cdot n + 0 = n(2,5 + 0,125b^2)$$

Die Zeiteinheiten pro Schleifendurchlauf beträgt weiterhin 2 Zeiteinheiten, die Anzahl der Druchläufe verringert sich auf n/k, es folgt also eine Komplexität von $2\frac{n}{k}$ Zeiteinheiten.

In Abhängigkeit von der Basis ergeben sich also die folgenden AT-Produkte:

Tabelle: AT-Komplexität des 2. Montgomery-Algorithmus' mit größeren Zahlenbasen

k=1	k=2	k=3	k=4
$6n^2$	$4,5n^2$	$(7n^2)$	$(17,25n^2)$

Hier wird also eine deutliche Verbesserung des AT-Produkts für die Basis 4 gegenüber der Basis 2 erreicht.

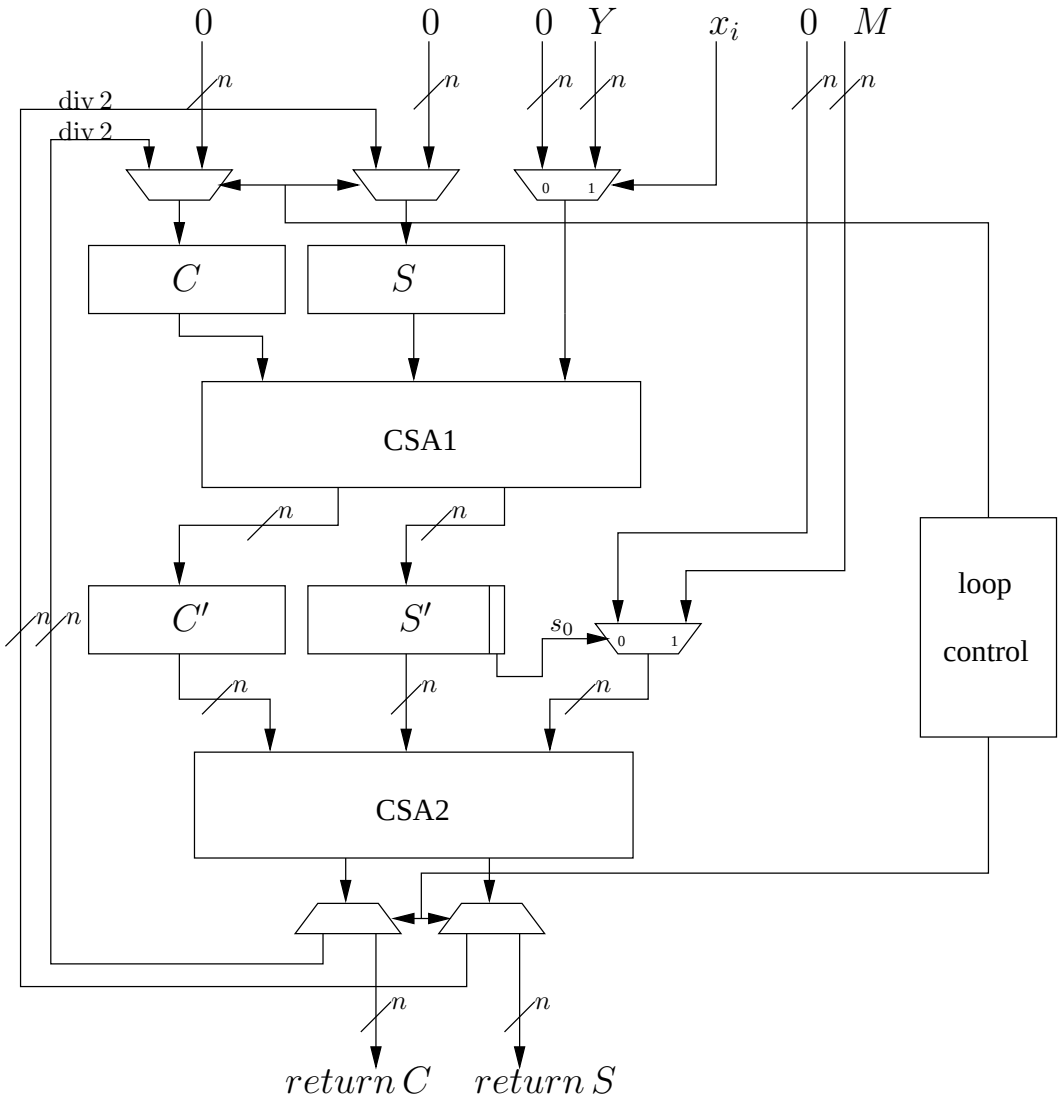


Abbildung 3.2: Montgomery 1 mit Carry-Save-Addierern

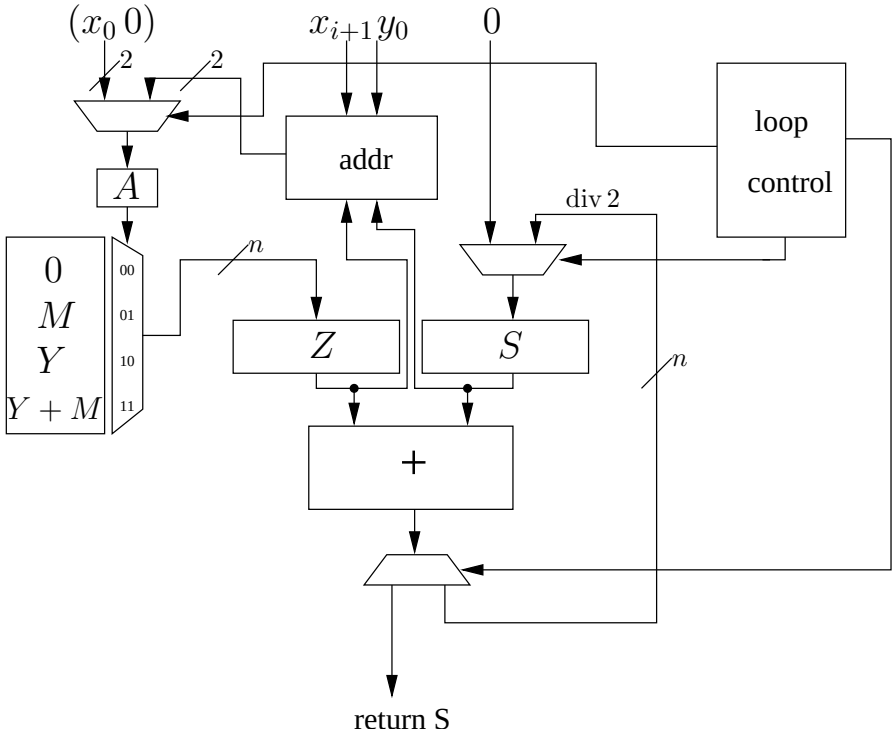


Abbildung 3.3: Montgomery 2

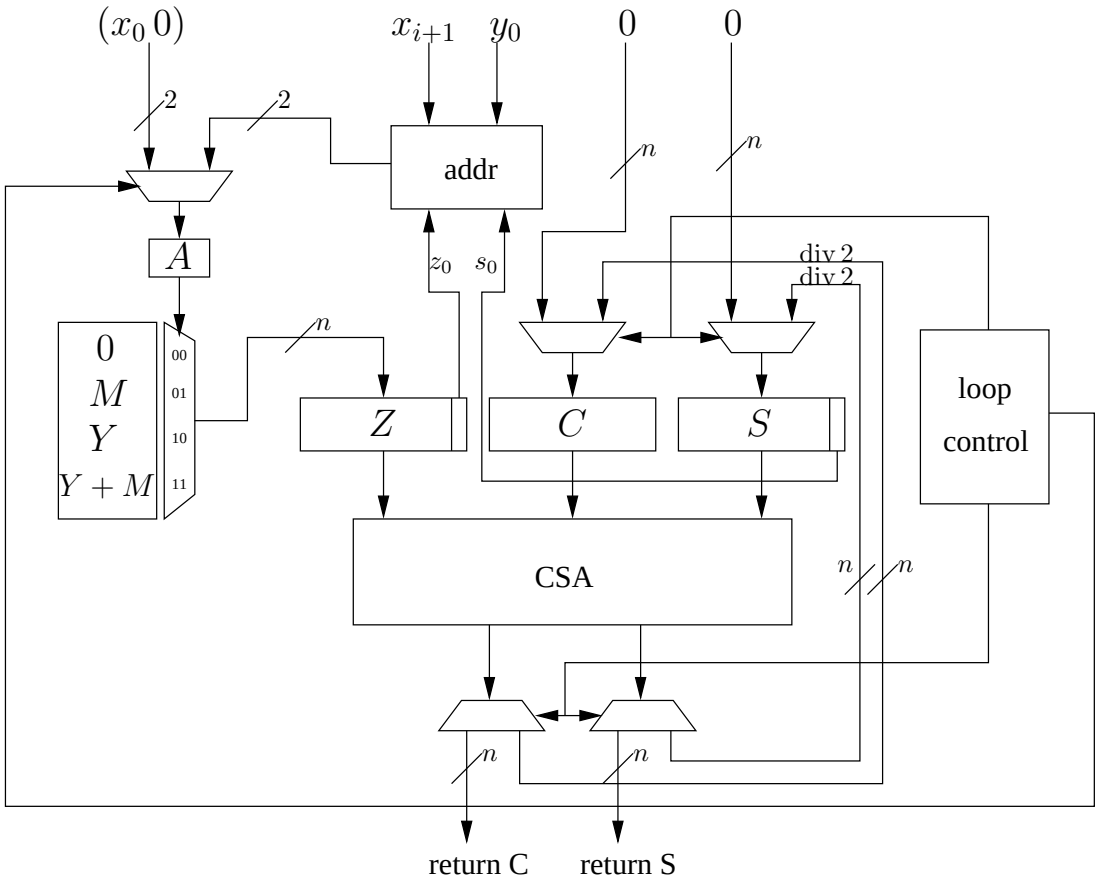


Abbildung 3.4: Montgomery 2 mit Carry-Save-Addierern

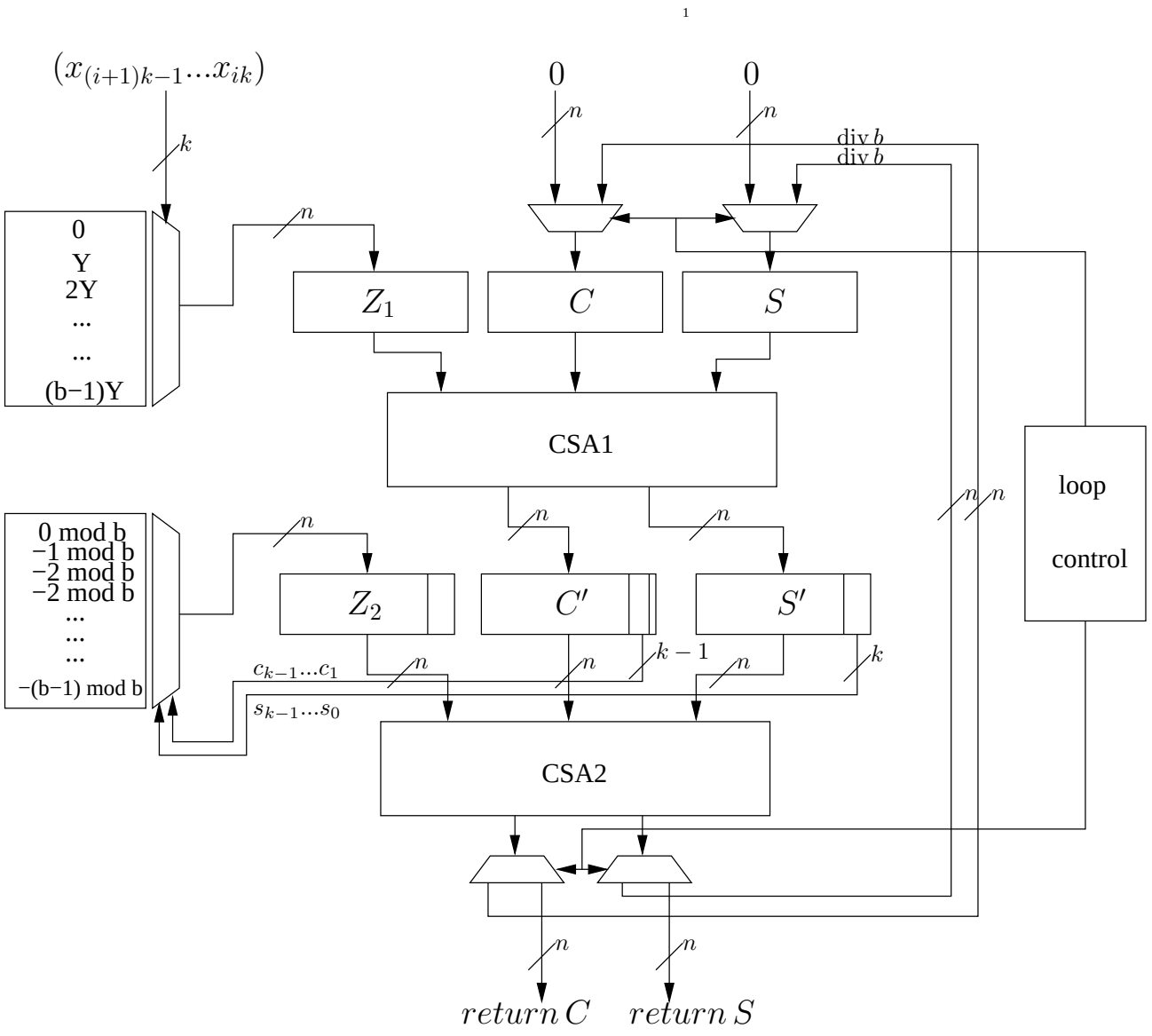


Abbildung 3.5: Montgomery 1 - höhere Zahlenbasen

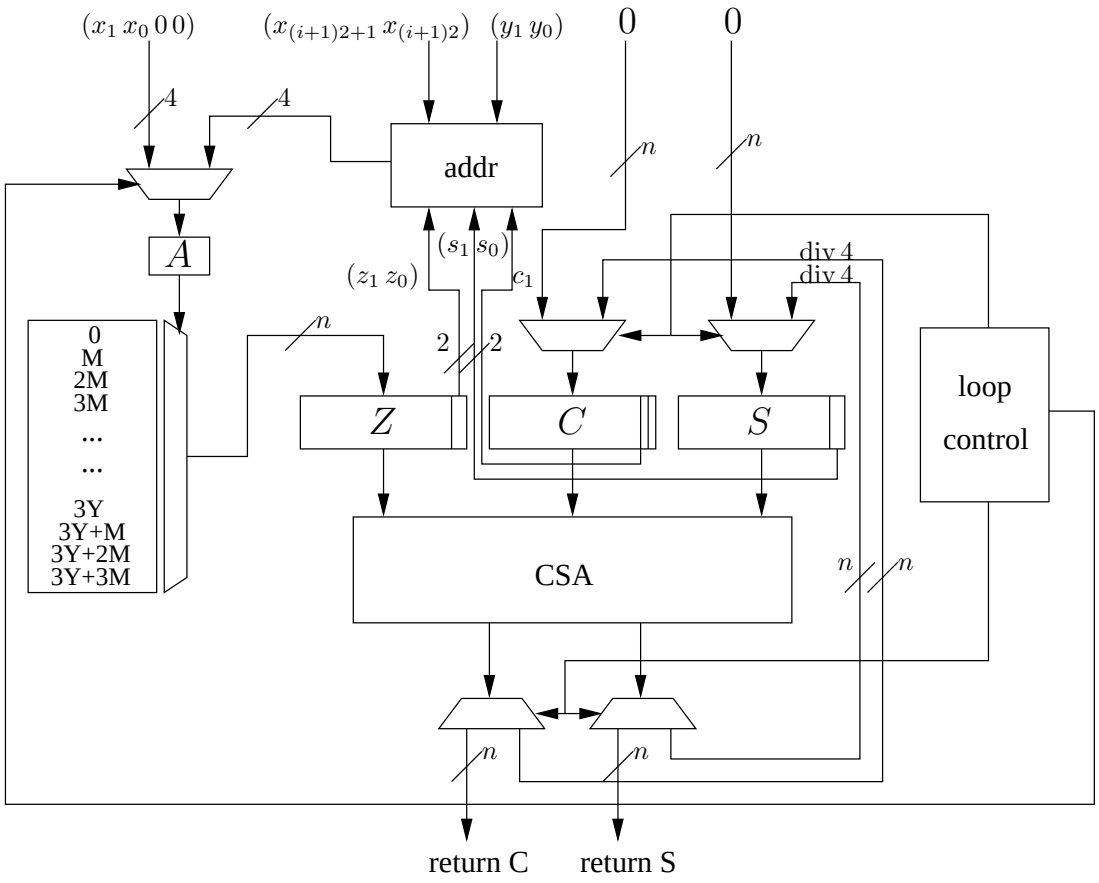


Abbildung 3.6: Montgomery 2 - Basis 4

Kapitel 4

eine neue Methode - Theoretischer Ansatz

4.1 Generalvoraussetzung

Für dieses Kapitel sei folgendes vorausgesetzt:

$$\begin{aligned} &\textit{Sei } n \in \mathbb{N}. \textit{ Sei } M \in \mathbb{N} \textit{ ungerade mit } 2^{n-1} < M < 2^n \\ &\quad \textit{und } M = (m_{n-1}...m_0) \textit{ der zugehörige Bitvektor.} \\ &\textit{Seien ferner } X, Y \in \mathbb{N} \textit{ mit } X, Y < M \\ &\quad \textit{und } X = (x_{n-1}...x_0), Y = (y_{n-1}...y_0) \textit{ die zugeörigen Bitvektoren} \end{aligned} \tag{4.1}$$

4.2 Die Idee

Das Ziel dieser Arbeit ist es, eine effiziente Methode für die modulare Multiplikation $X \cdot Y$ mod M zu entwickeln.
Wie wir in *Kapitel* 3.5 "Der Montgomery Algorithmus" gesehen haben, lohnt es sich für gewisse Anwendungen den Wert $X \cdot Y \cdot 2^{-n}$ mod M zu berechnen und dieser Motivation folgend werden wir einen ähnlichen Ansatz wie den des Montgomery-Algorithmus' wählen.
Genau wie bei Montgomery wird X schrittweise mit Y multipliziert und das erhaltene Ergebnis durch 2 dividiert. Bei Montgomery geschieht dies dadurch, dass das Ergebnis durch eine eventuelle Korrektur mit dem Modulus M durch 2 teilbar gemacht und anschliessend mittels einfachen Rechtsschiebens reduziert wird. An dieser Stelle unterscheidet sich der hier gewählte Ansatz von dem Montgomery-Algorithmus. Betrachten wir dazu folgende Überlegung:

Sei $E \in \mathbb{N}$.

$$\begin{aligned} E &= E \operatorname{div} 2 \cdot 2 + E \bmod 2 \\ \Rightarrow E \bmod M &\equiv (E \operatorname{div} 2 \cdot 2) \bmod M + (E \bmod 2) \bmod M \\ \Rightarrow E \cdot 2^{-1} \bmod M &\equiv (E \operatorname{div} 2 \cdot 2) \cdot 2^{-1} \bmod M + (E \bmod 2) \cdot 2^{-1} \bmod M \\ &= E \operatorname{div} 2 \bmod M + (E \bmod 2) \cdot 2^{-1} \bmod M \\ &\equiv E \operatorname{div} 2 + (E \bmod 2) \cdot 2^{-1} \bmod M \end{aligned} \tag{4.2}$$

Dazu sei gesagt, dass die multiplikativen Inveren von 2 oder jeder 2er Potenz modulo M genau dann existieren, wenn M ungerade ist, was wir vorausgesetzt haben (siehe dazu Kapitel 2.1 - Satz über endliche Zahlenfelder und Generalvoraussetzung 4.1). Die Division durch 2 lässt sich also durch zwei getrennte Operationen berechnen: durch die Bestimmung von $E \operatorname{div} 2$ (simples Rechtsschieben von E) und den Wert $(E \bmod 2) \cdot 2^{-1} \bmod M$ (also die Division des letzten Bits von E mit 2). Letzterer ist also entweder 0 oder $2^{-1} \bmod M$. Die Summe dieser beiden Werte ist kongruent zu $E \cdot 2^{-1} \bmod M$ und - wie wir später noch sehen werden - hinreichend beschränkt. So lässt sich der gesuchte Wert $X \cdot Y \cdot 2^{-n} \bmod M$ also schrittweise durch das Addieren neuer Eingabedaten - etwa $x_i \cdot Y$ - und das schrittweise Dividieren durch 2 berechnen.
Von dieser Tatsache ausgehend, werden wir die Methode entwickeln und feststellen, dass diese Vorgehensweise ein hohes Maß an Effizienz bietet - sowohl zeitlich, als auch in Hinsicht auf den Hardware-Aufwand.

4.3 Entwicklung der einstufigen Methode

Um von diesem Ansatz zu einer funktionierenden und vor allem effizienten Methode für Computerarithmetik zu gelangen, ist es aber noch ein langer Weg. Wir beginnen damit, einen ersten naiven Algorithmus zu formulieren und zu untersuchen:

4.3.1 1. Entwurf

1. Entwurf - hochsprachlich

```
1 E, e := 0;
2 for i := 0 to n−1 do begin
3           E:=E+x_i·Y;
4           e:=E mod 2;
5           E:=E div 2;
6           E:=E + (e·2^−1) mod M;
7 end;
8 return E;
```

Dass dieser Algorithmus die gewünschte Kongruenz $E \equiv X \cdot Y \cdot 2^{-n} \bmod M$ liefert, haben wir zuvor eingesehen. Offensichtlich ist das Endergebnis auch hinreichend klein:

```
1 E, e := 0;                              *E<3·M
2 for i := 0 to n−1 do begin
3           E:=E+x_i·Y;                   *E<·4M
4           e:=E mod 2;
5           E:=E div 2;                   *E<2·M
6           E:=E + (e·2^−1) mod M;        *E<3·M
7 end;
8 return E;
```

Das Ergebnis ist also durch $3M$ beschränkt und lässt sich schnell auf den gesuchten Wert modulo M reduzieren.
Nun betrachten wir diesen Algorithmus in einer Hardwareumsetzung. Siehe dazu Abbildung 4.1. Die Eingabewerte und die Zwischenergebnisse E und e werden in Registern gespeichert und mittels Addierern verknüpft. Die Operationen $x_i \cdot Y$ und $e \cdot 2^{-1} \bmod M$ weichen Vorausberechnungen und das Abspeichern der möglichen Werte 0, Y bzw. 0 und $2^{-1} \bmod M$ in Lookuptables. Das Berechnen von e und das Schieben von E kann nun gleichzeitig erfolgen, indem von dem Datenbus, der das Zwischenergebnis enthält, einfach das letzte Bit abgeschnitten und im Register e gespeichert wird.

4.3.2 1. Entwurf - Analyse

Ein Schleifendurchlauf kostet nun also 3 Taktschritte:

1. berechne $E + x_i \cdot Y$ und damit e und das geschobene Zwischenergebnis E^*

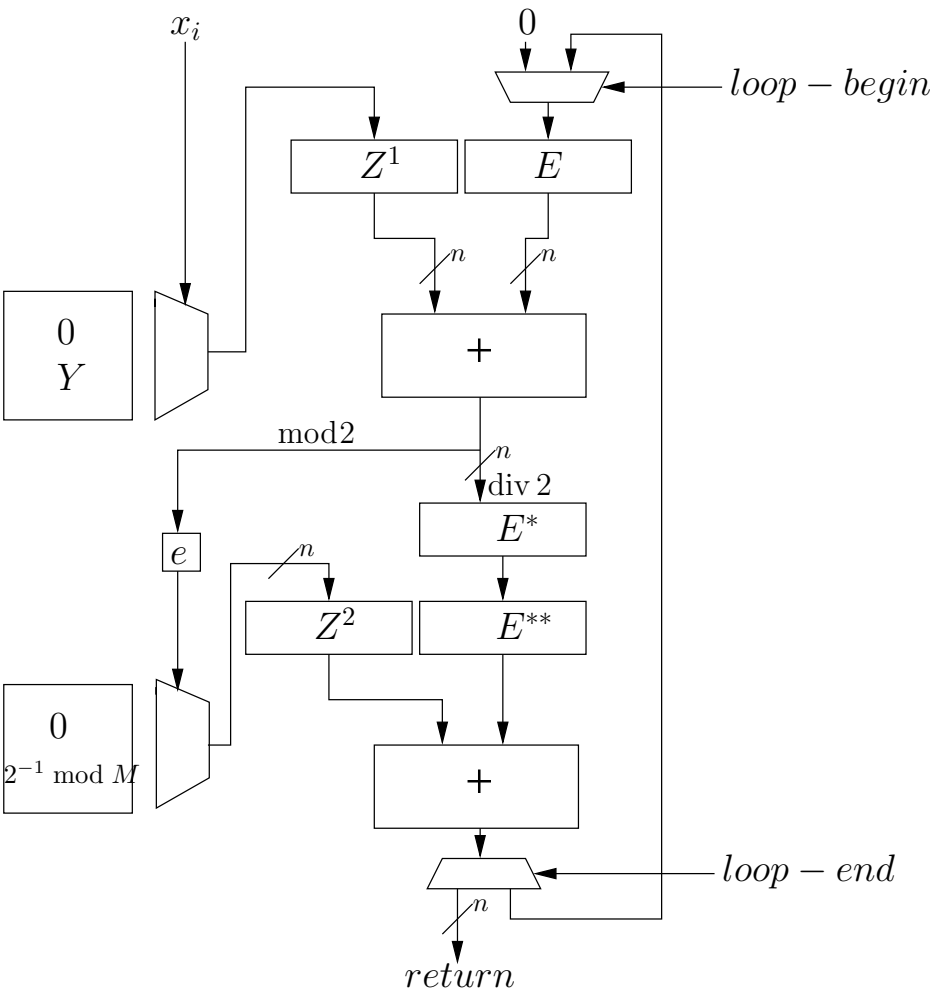


Abbildung 4.1: Entwurf 1

2. wähle in Abhängigkeit von e 0 oder 2^{-1} und speichere das Zwischenergebnis für einen weiteren Takt in E^{**}, solange die Lookuptable arbeitet

3. addiere zuletzt das Zwischenergbenis und den gewählten Korrekturwert, um das Ergebnis des Schleifendurchlaufs zu erhalten. Wähle weiterhin den neuen Eingabewert $x_{i+1} \cdot Y$

Die Fläche der Architektur lässt sich anhand Abbildung 4.1 abschätzen. Dabei gilt es aber zu bedenken, dass seine wirkliche Implementierung nur dann Sinn macht, wenn Carry-Save-Addierer verwendet werden. Diese sind im Entwurf also zu ergänzen, bevor man die Fläche berechnet. Dadurch ändert sich in diesem Entwurf die Anzahl der Register, denn die Zwischenergebnisse des Carry-Save-Addierers werden immer in zwei statt einem Register gehalten. Der Flächenverbrauch setzt sich also aus folgenden Komponenten zusammen:

1. Die zwei Register für Z und weitere sechs für die Zwischenergebnisse E (bzw. C,S) haben eine Fläche von $8 \cdot 0,5n$.

2. Zwei Addierer ergeben eine Fläche in der Größenordung von $2n$.

3. Die Lookuptables haben zusammen eine Fläche von $4 \cdot 0,125n$.

4. Zuletzt sind hier vier Multi- und Demultiplexer für das Laden und die Ausgabe der Register dargestellt, welche aber in einer Implementierung nicht benötigt werden

Die Fläche dieses Entwurfs beträgt also näherungsweise (ohne kleinere Register in Betracht zu ziehen)

$$8 \cdot 0,5n + 2 \cdot n + 4 \cdot 0,125n \approx 6,5n$$

Daraus folgt für das Produkt aus Fläche und Zeit des Entwurfs eine Abschätzung durch

$$6,5n \ \cdot \ 3n \ \approx \ 19,5n^2$$

Ein sehr hoher Wert im Vergleich zu den beiden Montgomeryvarianten.

4.3.3 2. Entwurf

Doch 3 Takte je Schleifendurchlauf sind noch zu viel. Ein zweiter Ansatz beschleunigt das Verfahren: anstatt das Zwischenergebnis zu schieben und auf die Korrektur durch den von e gewählten Wert zu warten, lässt sich dies bereits gleichzeitig mit dem nächsten Bit von X verarbeiten:

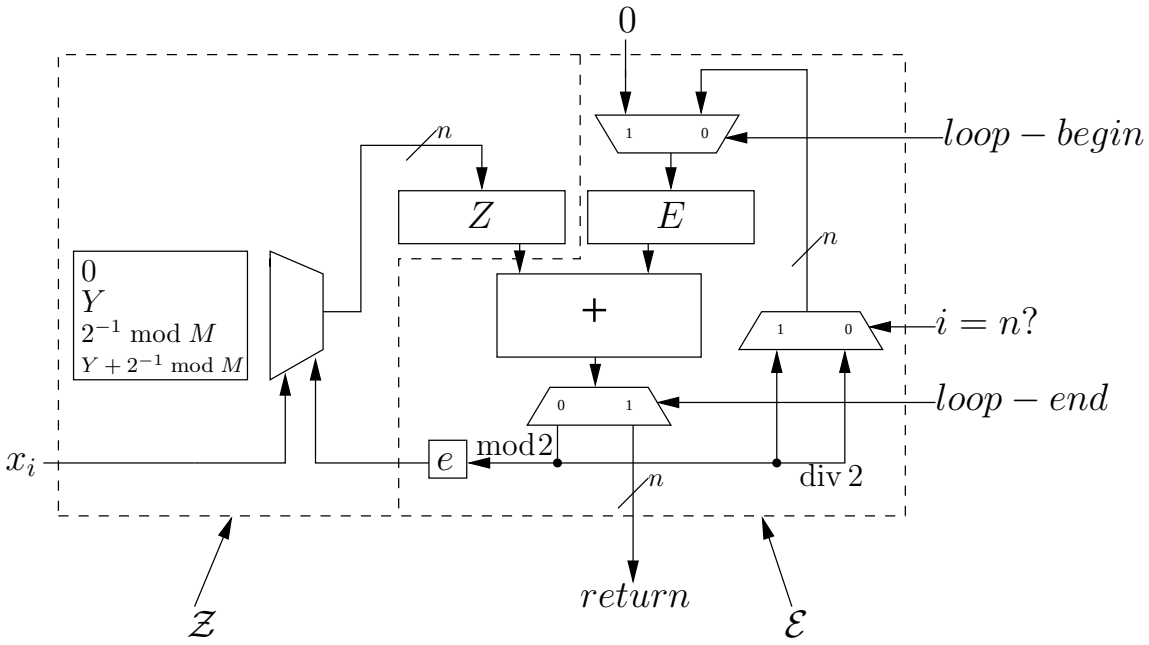


Abbildung 4.2: Entwurf 2

2. Entwurf - hochsprachlich

```
1  E, e := 0;
2  for i := 0 to n−1 do begin
3            E:=E + x_i·Y + (e·2^{-1}) mod M;
4            e:=E mod 2;
5            E:=E div 2;
6  end;
7  return E + e · 2^{-1} mod M;
```

Diese Vorgehensweise birgt eine Schwierigkeit für die Umsetzung in Hardware: die 3. Zeile erfordert die Addition von drei Werten. Die Schaltung eines Addierers mit drei Eingabewerten ist jedoch platzaufwendig und vor allem *langsam*. Und da in einem taktgesteuerten System die zeitaufwendigste Logik die Taktdauer bestimmt und alle weiteren verwendeten Komponenten vergleichsweise schnell sind, würde solch ein Addierer die zeitliche Effizienz verschlechtern.
Eine mögliche Lösung des Problems ist wie zuvor die Vorausberechnung dieser Operation. Denn die Operation $x_i \cdot Y + e \cdot 2^{-1}$ kann ohne großen Aufwand (schliesslich gibt es für jeden Summanden nur zwei mögliche Werte) im Voraus ermittelt werden und in einer Lookuptable abgespeichert werden.

4.3.4 2. Entwurf - Analyse

In Abbildung 4.2 ist ein, dieser Lösung entsprechender, Entwurf dargestellt. Man beachte den Nachbearbeitungsschritt, der die Schleife ein (n+1)-tes mal durchläuft und statt x_i eine 0 annimmt und so nur die Addition von $e \cdot 2^{-1} \bmod M$ durchführt. Die abgebildete Architektur verarbeitet ein Eingabebit in zwei Takten:

1. wähle einen Wert aus der Lookuptable in Abhängigkeit von x_i und e
2. addiere den neuen Eingabewert zu E, schneide e ab und schiebe E

Für die Fläche ergibt sich ein Wert von ungefähr

$$3 \cdot 0,5n + 2 \cdot 0,25n + 1 \cdot n + 4 \cdot 0,125n = 3,5n$$

und damit $7n^2$ für das AT-Produkt (man beachte, dass für diese Berechnung der Entwurf um Carry-Save-Addierer erweitert und dadurch zusätliche Register benötigt werden). Der Entwurf hat nun also von der Größenordnung die gleiche Flächen-Zeit-Komplexität wie der erste, in Kapitel 3.5 vorgestellte, Montgomery Algorithmus.

4.3.5 3. Entwurf

Eine letzte Verbesserung dieses Ansatzes wird die Zeitkomplexität ein weiteres Mal verringern und so das für diese Vorgehensweise erreichbare Optimum erzielen: wir verbessern den Datenfluss durch Pipelining.

Der Hardwareentwurf bleibt der gleiche wie zuvor (siehe Schaubild 4.2), nur dass jetzt die beiden zeitaufwendigsten Komponenten (der Addierer und die Lookuptable) *gleichzeitig* arbeiten und damit die Luafzeit des Algorithmus' verringert wird. Betrachte dazu folgendes Pipelining-Diagramm:

Abbildung 4.3 veranschaulicht den Datenfluss in der Architektur. In der Senkrechten sind die Eingaben in die Architektur (die Koeffizienten von X) aufgetragen. In der Waagerechten ist der Takt also die Zeit dargestellt. Jedes Eingabebit wird zunächst von $\mathcal{Z}$ (also der Lookuptable) und darauf von $\mathcal{E}$ (dem Addierer) verarbeitet. Dabei findet ein Datenfluss von $\mathcal{Z}$ nach $\mathcal{E}$ statt: die LUT übergibt ihren Wert an das Register Z. Es fließen auch Informationen in der umgekehrten Richtung. Das Diagramm stellt dies durch die Pfeile von $\mathcal{E}$ nach $\mathcal{Z}$ dar. Es lässt sich dem Diagramm gut entnehmen, wie "alt" die

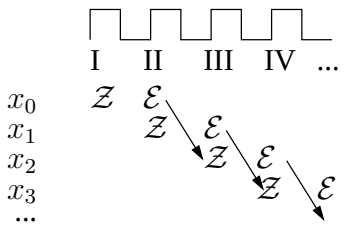


Abbildung 4.3: Pipelining Diagramm - Entwurf 3

Daten sind, wenn sie die Lookuptable erreichen: die Bits gehören noch zur Verarbeitung des Koeffizeiten zwei Takte zuvor.

Hochsprachlig lässt sich dieses Vorgehen wie folgt erfassen:

3. Entwurf - hochsprachlich

```
1  E, e := 0;  Z_{-1} := x_0 · Y;
2  for i := 0 to n−1 do begin
3              co begin
4                       Z_i := x_{i+1} · Y + (e_{i−1} · 2^{−2}) mod M;
5              //
6                       E_i := E_{i−1} + Z_{i−1};
7                       e_i := E_i mod 2;
8                       E_i := E_i div 2;
9              end;
10 end;
11 return  E_n + Z_n + e_n · 2^{−1} mod M;
```

Die Funktionsweise der Methode bleibt also erhalten, aber durch die Parallelisierung ändert sich die zeitliche Abfolge in der Datenverarbeitung:

1. Takt: Auswahl von Z_{-1} als 0 oder Y in Abhängigkeit von x_0

2. Takt: Verarbeitung von Z_{-1} im Addierer und damit Erstellung von E_0 und e_0. Weiterhin die Auswahl von Z_0 in Abhängigkeit von x_1

3. Takt: Verarbeitung von Z_0 ($\Rightarrow E_1, e_1$). Weiterhin die Auswahl von Z_1 nun aber in Abhängigkeit von x_2 und e_0, also den Wert $Z_1 = x_2 \cdot Y + e_0 \cdot 2^{-2} \bmod M$.

usw...

(n+1). Takt: Verarbeitung von Z_{n-2} ($\Rightarrow E_{n-1}, e_{n-1}$) und Auswahl von Z_{n-1} aus x_n ($= 0$) und e_{n-2}.

Durch die Parallelisierung wird also das E enthaltende Register zweimal geschoben ($\approx$ durch zwei geteilt), bevor das abgetrennte Bit e, wieder in die Berechnung einfließt. Deswegen wird e nun auch mit 2^{-2} verrechnet, um nach wie vor ein korrektes Ergebnis zu liefern. Außerdem wird E einmal geschoben, bevor der in der LUT berechnete und in Z gespeicherte Wert die Einheit $\mathcal{E}$ erreicht. Daher wird der Y betreffende Anteil des Wertes in Z einen Takt im voraus bestimmt (x_{i+1}), damit die Daten mit dem richtigen "Alter" aufeinander treffen. Aus diesem Grund muss auch in der Hardware der Eingeabewert X um eine führende Null erweitert werden, damit im letzten Schleifendurchlauf die Lookuptable angesteuert werden kann. Dies ist im Entwurf nicht abgebildet - wir überlassen es dem Verständnis des Lesers.

Der formale Beweis für die Korrektheit dieser Vorgehensweise wird später geliefert werden (siehe Kapitel 4.7.1).

4.3.6 3. Entwurf - Analyse

Um den PLatzbedarf der Architektur abschätzen zu können, benötigen wir zunächst einen Entwurf, der mit den zeiteffizienten Carry-Save-Addierern arbeitet. In Abbildung 4.4 ist eine solche Variante dargestellt.

Der Flächenverbrauch wird von 3 Registern, 1 Addierer und 4xn LUT-Speicherzellen bestimmt (Schleifenkontrolle sowie kleinere Register und Multiplexer werden wir wieder als unwesentliche Verbraucher ignorieren). Es folgt also eine Flächenkomplexität von

$$3 \cdot 0,5n + 1 \cdot n + 4 \cdot 0,125n = 3,5n$$

Flächeneinheiten.

Der Entwurf benötigt jetzt also nur einen Taktschritt zur Verarbeitung eines Eingabebits (zuzüglich der zwei Nachbearbeitungsschritte). Ein zeitlich optimales Ergebnis für den Fall, dass man nur ein Eingabebit pro Takt erhält (Basis 2).

Es folgt eine AT-Komplexität von $3,5n^2$.

4.4 Entwicklung der zweistufigen Methode

4.4.1 Ein neuer Ansatz

Als nächstes werden wir den Flächenverbrauch der Architektur verbessern. Denn der vorherige Entwurf wirft das folgende Problem auf: Die Lookuptable, die die Eingabewerte $x_i \cdot Y$ enthält, muss diese auch mit den Korrekturwerten

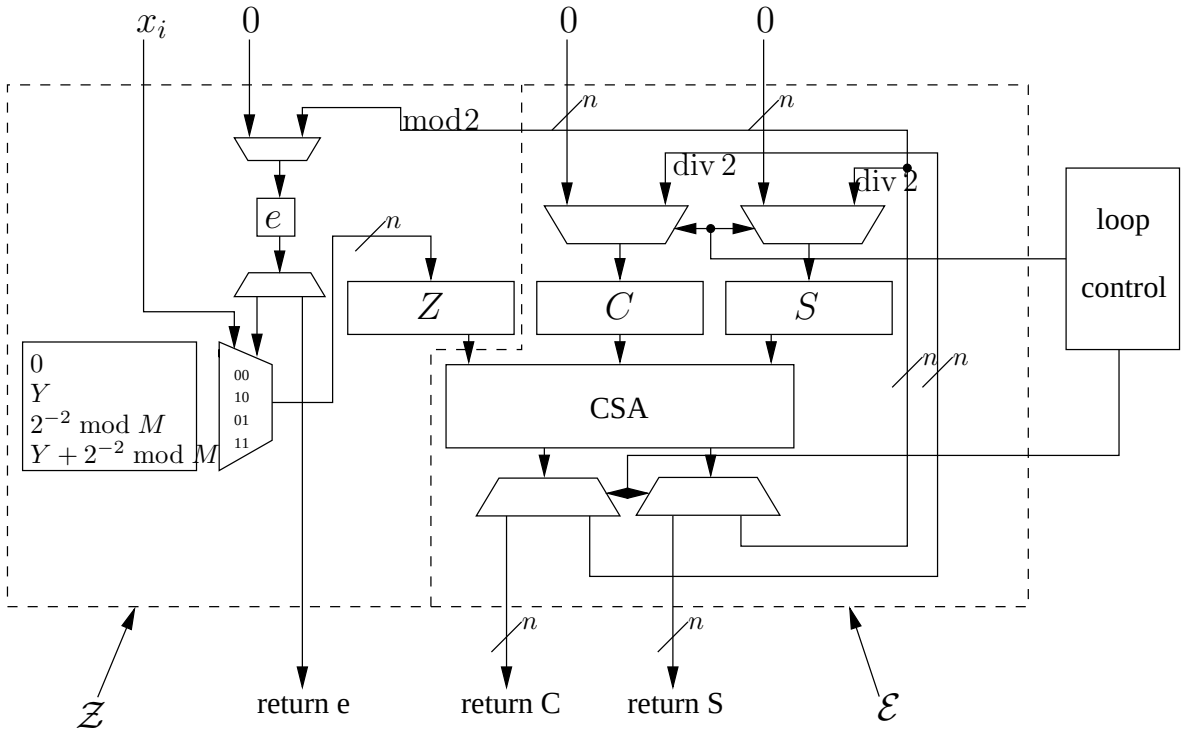


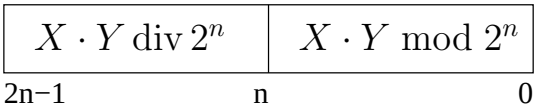
Abbildung 4.4: Entwurf 3 mit Carry-Save-Addierer

$e_{i-1} \cdot 2^{-2}$ kombinieren. Das führt von zwei LUTs mit jeweils zwei Einträgen zu einer LUT mit 4 Einträgen. Will man nun die Basis - also die Anzahl der pro Takt verarbeiteten Bits - erhöhen, wirkt sich diese Kombination dramatischer aus. Bei der Basis 4 wird anstelle zweier LUTs mit 4 Werten eine LUT mit 16 Werten benötigt, bei der Basis 8 64 statt zweimal 8, usw... Der Platzbedarf der zweiten Architektur verhält sich also proportional zum Quadrat des Bedarfes der ersten Architektur. Denn die zu speichernden Werte sind in der Größenordung von M (je nach Implementierung sogar einige Vielfache davon), also n-Bit Zahlen, die also den Flächenbedarf der Architektur bestimmen. Das ist zu viel, da der Zeitgewinn von der ersten zur zweiten Methode nur ein Drittel beträgt und das AT-Produkt durch diese Veränderung also drastisch verschlechtert wird. Betrachten wir nun folgende Überlegung:

$$X \cdot Y = X \cdot Y \operatorname{div} 2^n \cdot 2^n + X \cdot Y \bmod 2^n$$

Siehe dazu Schaubild 4.5:

Abbildung 4.5: Veranschaulichung div/mod



$$
\begin{aligned}
\Rightarrow X \cdot Y \cdot 2^{-n} \bmod M &= ((X \cdot Y \operatorname{div} 2^n \cdot 2^n) \cdot 2^{-n} + (X \cdot Y \bmod 2^n) \cdot 2^{-n}) \bmod M \\
&= ((X \cdot Y \operatorname{div} 2^n) + (X \cdot Y \bmod 2^n) \cdot 2^{-n}) \bmod M \\
&\equiv (X \cdot Y \operatorname{div} 2^n) \bmod M + (X \cdot Y \bmod 2^n) \cdot 2^{-n} \bmod M \\
&\equiv X \cdot Y \operatorname{div} 2^n + (X \cdot Y \bmod 2^n) \cdot 2^{-n} \bmod M
\end{aligned}
$$

Aus $X, Y < 2^n$ folgt $X \cdot Y < (2^n)^2$, also $X \cdot Y \operatorname{div} 2^n < 2^n$. Damit ist die zuletzt geschlossene Kongruenz $(X \cdot Y \operatorname{div} 2^n) \bmod M \equiv X \cdot Y \operatorname{div} 2^n$ entweder eine Gleichheit oder aber ein Unterschied um den Modulus M, denn wir haben $M > 2^{n-1}$ vorausgesetzt (4.1).
Die Berechnung des Ergebnisses lässt sich also in zwei Hälften spalten: in die von $E \equiv X \cdot Y \operatorname{div} 2^n$ und von $K \equiv (X \cdot Y \bmod 2^n) \cdot 2^{-n} \bmod M$.
Der Vorteil dieser Vorgehensweise ist der folgende: die beiden Werte E und K und damit das Endergebnis lassen sich parallel und somit insgesamt effizienter berechnen als in einem gemeinsamen Register, da die Addition der Werte $x_i \cdot Y$ und $e_{i-1} \cdot 2^{-2}$ unabhängig voneinander geschehen kann und somit die Problematik des dreifachen Addierers bzw. der kombinierten Lookuptable umgangen wird.

4.4.2 4. Entwurf

Forumlieren wir diesen Ansatz also zunächst wieder naiv und hochsprachlich:

4. Entwurf - hochsprachlich

```
 1  E, e, K, k := 0;
 2  for i := 0 to n−1 do begin
 3              E := E + x_i · Y;
 4              e := E mod 2;
 5              E := E div 2;
 6              K := K + e;
 7              k := K mod 2;
 8              K := K div 2;
 9              K := K + k · 2^−1 mod M;
10  end;
```

```
11  return  E + K;
```

Die Berechnung von $X \cdot Y \cdot 2^{-n} \bmod M$ erfolgt nun also in zwei jeweils aufeinanderfolgenen Rechenschritten zur Bestimmung von E und K. Die Lookuptable trennt sich nun wieder in zwei (bei größerer Basis erheblich-) kleinere Teile zur Vorausberechnung von $x_i \cdot Y$ und $k \cdot 2^{-1} \bmod M$. Doch vom Standpunkt der Geschwindigkeit aus ist dieses Verfahren natürlich unbrauchbar: 5 Takte sind nötig, bevor das Eingabebit x_i verarbeitet wurde und der nächste Durchlauf beginnen kann. Wir werden also direkt zum nächsten Entwurf übergehen, der das Verfahren beschleunigen wird.

4.4.3 5. Entwurf

Das Problem des 4. Entwurfes ist vor allem die Rückkopplung des abgeschnittenen Bits k auf das Register K. Wie schon im 2. Entwurf werden wir diese in k enthaltene Information erst im darauffolgenden Schleifendurchlauf wieder in die Berechnung mit einfliessen lassen:

5. Entwurf - hochsprachlich

```
1  E,e,Z,K,k:=0;
2  for  i:=  0  to  n-1  do  begin
3              co  begin
4                          E_i := E_{i-1} + x_i · Y;
5                          e_i := E_i mod 2;
6                          E_i := E_i div 2;
7              //
8                          Z_i := k_{i-1} · 2^{-1} mod M;
9              end;
10             K_i := K_{i-1} + e_i + Z_i;
11             k_i := K_i mod 2;
12             K_i := K_i div 2;
13 end;
14 return  E_{n-1} + K_{n-1} + k_{n-1} · 2^{-1} mod M;
```

4.4.4 5. Entwurf Analyse

Die Division von K durch 2 wurde also wie zuvor in den nächsten Durchlauf mit einbezogen, was wie zuvor einen Geschwindigkeitsgewinn von 2 Takten bringt. Wie zuvor führt dies aber auch zum Problem des dreifachen Addierers bzw. der großen Lookuptable (siehe 2. Entwurf). Man beachte jedoch, dass mit e im Basis 2-Fall nur ein einzelnes Bit addiert werden muss. Je nach

Addiererarchitektur kann dies auch ohne eine große LUT, sondern mittels eines ungenutzten Carry-Eingangs erfolgen (z.B. bei einem Riple-Carry- oder auch beim Carry-Save-Adder). Dennoch ist dieser Entwurf zeitlich viel zu langsam, um ernsthaft in Betracht gezogen zu werden.

4.4.5 6. Entwurf

Wenden wir nun auf den eben entworfenen Algorithmus die gleiche Prozedur noch einmal an: genauso wie die Berechnung von $K \cdot 2^{-1} \bmod M$ auf zwei nachfolgende Schleifendurchläufe gestreckt werden kann, hilft Pipelining auch, die Berechnung von E und K zu entkoppeln. Anstatt auf die Verarbeitung des abgeschnittenen Bits e zu warten, verfährt der E berechnende Teil der Archtitektur gleich mit dem nächsten Eingabebit. Bei der Verarbeitung von e muss aber wie zuvor das "Alter" der Information beachtet werden.

6. Entwurf - hochsprachlich

```
1  E,e,Z,K,k:=0;
2  for i:= 0 to n-1 do begin
3              co begin
4                       E_i := E_{i-1} + x_i · Y;
5                       e_i := E_i mod 2;
6                       E_i := E_i div 2;
7              //
8                       Z_i := e_{i-1} · 2^{-2} mod M + k_{i-1} · 2^{-2} mod M;
9              //
10                      K_i := K_{i-1} + Z_{i-1};
11                      k_i := K_i mod 2;
12                      K_i := K_i div 2;
13             end;
14 end;
15 return  E_{n-1} + e_{n-1} · 2^{-1} mod M + Z_{n-1} + K_{n-1} + k_{n-1} · 2^{-1} mod M;
```

Da der Schleifendurchlauf nun also ein weiteres mal entkoppelt wurde sieht der Ablauf der Datenverarbeitung nun wie folgt aus:

1. Takt: Verarbeitung von x_0 ($\Rightarrow E_0, e_0$).

2. Takt: Verarbeitung von x_1 ($\Rightarrow E_1, e_1$) / Verarbeitung von e_0 ($\Rightarrow Z_1$).

3. Takt: Verarbeitung von x_2 ($\Rightarrow E_2, e_2$) / Verarbeitung von e_1 ($\Rightarrow Z_2$) / Verarbeitung von Z_1 ($\Rightarrow K_2, k_2$).

4. Takt: Verarbeitung von x_3 ($\Rightarrow E_3, e_3$) / Verarbeitung von e_2 *und* k_2 ($\Rightarrow Z_3$) / Verarbeitung von Z_2 ($\Rightarrow K_3, k_3$).

usw...

Die komplizierteste Operation dieses 6. Entwurfs ist die 8. Zeile. Diese birgt einige Probleme für die Hardwareumsetzung. Man beachte, dass beide Bits e und k mit 2^{-2} mod M multipliziert werden. Mathematisch gesehen kann die Operation also auch $Z_i := (e_{i-1} + k_{i-1}) \cdot 2^{-2}$ mod M lauten. Dies könnte man durch eine einfache Logik vor zwei Registern , die 0 und 2^{-2} mod M enthalten realisieren und dann durch einen Addierer laufen lassen. Abhängig von der Zielarchitektur und von den dafür gewählten Addierern, kann dieser Ansatz jedoch unpraktikabel sein. Denn ein Carry-Save-Addierer z.B. kann nur eine redundant dargestellte Zahl mit einer normal dargestellten verknüpfen, also muss entweder das Register Z oder K in normaler, nicht redundanter Darstellung auftreten. Weiterhin werden bei größeren Basen auch viele Register bzw. Einträge in Lookuptabellen benötigt, um die möglichen Vielfachen von 2^{-k} mod M zu speichern (und man bedenke, dass diese Werte die gleiche Bitlänge wie M haben) und eine entsprechend aufwendige Auswahllogik (k sie hier der Logarithmus der zu Grunde gelegten Zahlenbasis). Aus diesem Grund werden wir hier später einen anderen Ansatz wählen. Doch zunächst lösen wir diese Operation wenig effizient durch eine kombinierte LUT, die die möglichen Werte enthält und mit den Bits e und k adressiert wird. Den dazu gehörigen Hardwareentwurf zeigt Abbildung 4.6.

4.4.6 6. Entwurf - Analyse

Die Verarbeitung eines Eingabeits benötigt also wieder nur einen Takt - genau wie im 3. Entwurf. Der Platzbedarf der Architektur beläuft sich auf

$$6 \cdot 0,5n + 2 \cdot n + 6 \cdot 0,125n = 5,75n$$

(man beachte, dass die Mutliplexer wieder aus der Flächenberechnung herausgenommen wurden) und das AT-Produkt damit auf $5,75n^2$ für die Basis 2. Für höhere Basen lassen sich bessere Werte erzielen, zunächst steigern wir die Effizienz durch eine weitere Veränderung.

4.4.7 7. Entwurf

Der 7. Entwurf wird eine letzte Optimierung bringen und die theoretische Entwicklung der Methode abschließen.
Wenden wir uns nun der im 6. Entwurf aufgekommenen Problematik zu. Die Möglichkeit tatsächlich eine Addition durchzuführen, kann - wie gesagt - unpraktikabel sein. Die im 6. Entwurf entwickelte Lösung jedoch ist zu

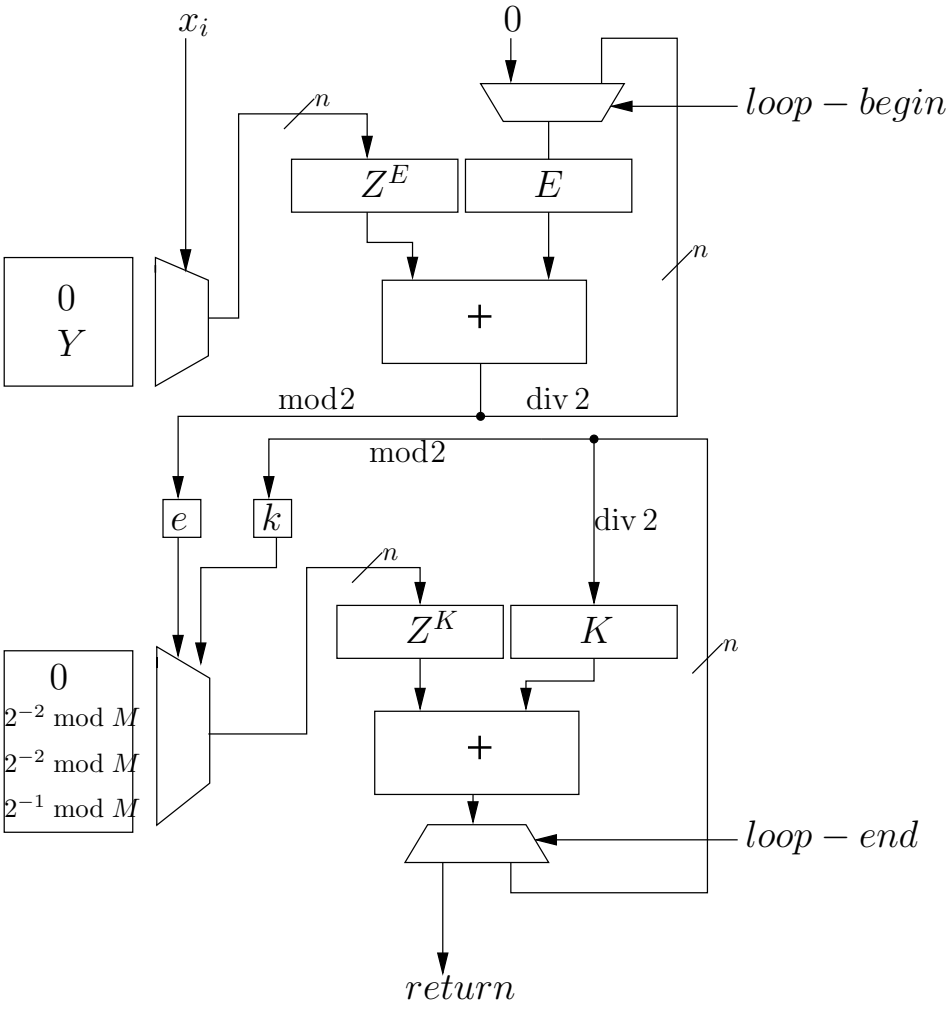


Abbildung 4.6: Entwurf 6

speicheraufwendig, da eine große LUT zum Berechnen der Operation $(e_{i-1} + k_{i-1}) \cdot 2^{-2} \bmod M$ benötigt wird.

Es sei b die verwendete Zahlenbasis. Dann gilt für die Anzahl an Einträgen in der Lookuptable ($\#LUT$):

$$\#LUT = 2^{\log_2(b)+\log_2(b)} = 2^{\log_2(b)} \cdot 2^{\log_2(b)} = b \cdot b = b^2 \tag{4.3}$$

Gleichung 4.3 erklärt den Zusammenhang zwischen der Zahlenbasis und der Größe der Lookuptable. Mit b^2 Einträgen ist diese LUT genauso groß, wie die vorausberechneten Operationen des 3. Entwurfs, also unbrauchbar.
Trennt man den Ausdruck $(e_{i-1} + k_{i-1}) \cdot 2^{-2} \bmod M$ jedoch in zwei Berechnungen - die von $S := e_{i-1} + k_{i-1}$ und $S \cdot 2^{-2} \bmod M$ - lässt sich der Speicher effizienter nutzen: die Summe $e_{i-1} + k_{i-1}$ ist offensichtlich durch $2 \cdot b$ beschränkt. Die LUT zur Berechnung von $S \cdot 2^{-2} \bmod M$ kann sich also auf $2 \cdot b$ Werte beschränken, was für größere Basen einen deutlich kleineren Platzbedarf als bei der vorherigen Lösung bedeutet. Es bleibt die Operation $S := e_{i-1} + k_{i-1}$ zu realisieren. In diesem Entwurf werden wir auch diese als LUT vorausberechnen. Diese hat dann zwar wieder b^2 Einträge (denn alle möglichen Kombinationen von e und k müssen berücksichtigt werden), aber jeder Eintrag enthält nur eine Adresse für die zweite Operation - ist um genau zu sein nur $log_2(2 \cdot b) = 1 + log_2(b)$ Bits lang. In Sachen Platzbedarf ist sie also zu vernachlässigen. Auf die Geschwindigkeit ist der Einfluss dieser Operation größer. Mit wachsender Basis wird der für die Ansteuerung dieser Lookuptable benötigte Multiplexer die taktbestimmende Komponente im Entwurf sein.

Abbildung 4.7 zeigt den zugehörigen Hardwareentwurf. Man beachte die zusätzliche Überalterung der Daten durch die erneute Teilung einer Operation und deren Berücksichtigung in der Operation $S \cdot 2^{-3} \bmod M$:

7. Entwurf - hochsprachlich

```
1  E, e, S, Z, K, k := 0;
2  for i := 0 to n−1 do begin
3              co begin
4                        E_i := E_{i−1} + x_i · Y;
5                        e_i := E_i mod 2;
6                        E_i := E_i div 2;
7              //
8                        S_i := e_{i−1} + k_{i−1};
9              //
10                       Z_i := S_{i−1} · 2^{−3} mod M;
11             //
```

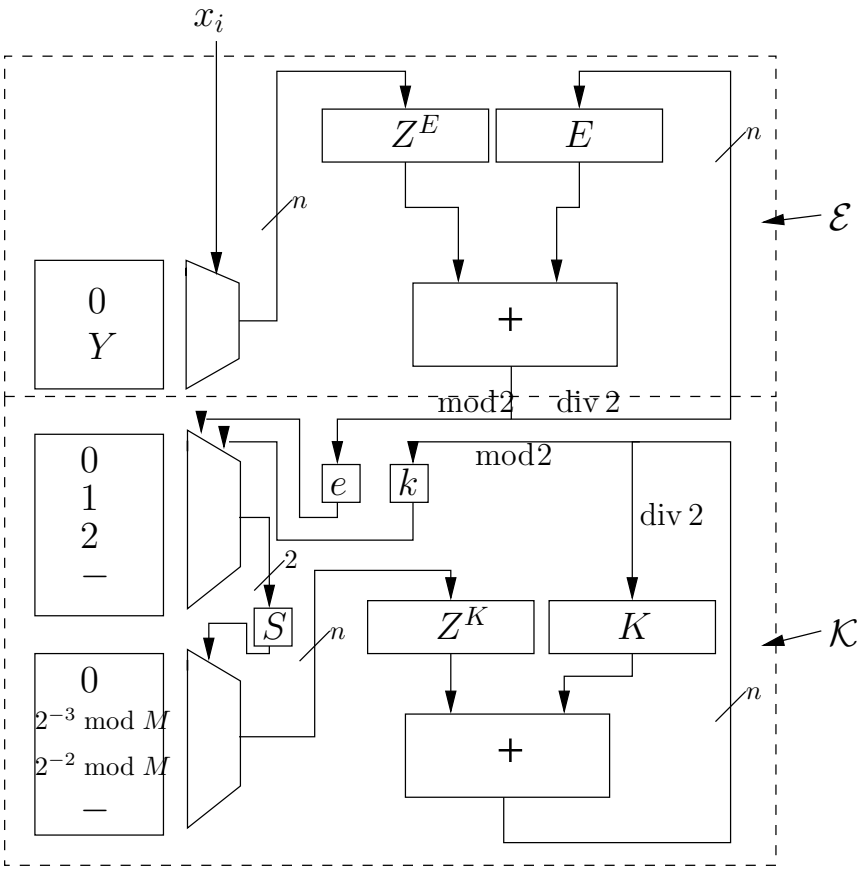


Abbildung 4.7: Entwurf 7

```
12                          K_i := K_{i-1} + Z_{i-1};
13                          k_i := K_i mod 2;
14                          K_i := K_i div 2;
15              end;
16  end;
17  return  E_{n-1} + e_{n-1} · 2^{-1} mod M + S_{n-1} · 2^{-2} mod M
18              +Z_{n-1} + K_{n-1} + k_{n-1} · 2^{-1} mod M;
```

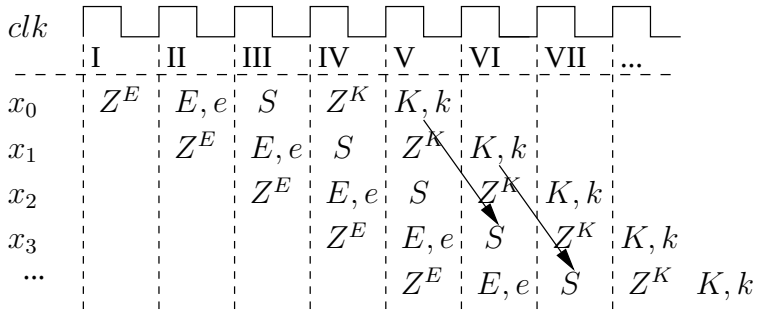


Abbildung 4.8: Pipelining - Entwurf 7

4.4.8 7. Entwurf - Analyse

Wie im 6. Entwurf, erfolgt die Datenverarbeitung in einem Takt je Eingabebit zuzüglich der Nachbearbeitungsschritte, um das Endergebnis festzustellen. Der Platzbedarf der Architektur ist nun aber durch die kleineren Lookuptables auf $5,625n$ gesunken. Ein kaum merklicher Unterschied im Vergleich zum 6. Entwurf. Diese Architektur wird jedoch für höhere Zahlenbasen Vorteile aufweisen.

4.5 Die zweistufige Methode für die Basis 2

Der finale Entwurf des letzten Kapitels stellt nun die endgültige Version auf *theoretischem* Niveau dar. Um diesen Ansatz für die Praxis effizient zu gestalten, sind noch kleinere Änderungen und konkrete Implementierungen von Vor- und Nachberechnungen nötig. Dies wird aber Gegenstand weiterer Kapitel sein. In diesem Kapitel werden wir den theoretischen Entwurf auf seine Korrektheit hinsichtlich der Arbeitsweise und des Ergebnisses überprüfen. Dazu werden wir uns einer Beweismethode für sequentielle Computerprogramme - und in diese lässt sich der Entwurf logisch überführen - nach

Hoare (siehe Kapitel 2.4) bedienen. Diese sieht vor, den Algorithmus mit einem Zusicherungsnetzwerk zu versehen, und die einzelnen Zusicherungen Schritt für Schritt ineinander zu überführen. Siehe dazu Kapitel 4.5.1. Der Beweis der Korrektheit der Methode wird beinhalten, dass das Ergebnis nach Verarbeitung aller Eingabewerte dem gesuchten Produkt

$$(X \cdot Y) \cdot 2^{-n} \bmod M$$

entspricht. Genauer: wir wollen folgende Kongruenz nach Termination der Methode beweisen:

$$\begin{aligned} & E + K + (e + k) \cdot 2^{-1} \cdot M + Z^K + S \cdot 2^{-2} \bmod M \\ & \equiv (X \cdot Y) \cdot 2^{-n} \bmod M \end{aligned} \tag{4.4}$$

Dies wird in drei Beweisteilen erfolgen. Teil 1 und 2 werden Teile der Methode auf ihre korrekte Funktionsweise überprüfen: die Einheit $\mathcal{E}$ und die Einheit $\mathcal{K}$ (vgl. Abbildung 4.7). Teil 3 wird dann diese Ergebnisse zu dem gesuchten Endergebnis verbinden. Abschliessend wird die Methode als ganzes bewertet werden.

4.5.1 Beweis der Korrektheit der Methode für die Basis 2 - Teil 1

Zunächst werden wir uns der Einheit $\mathcal{E}$ zu und dem dort berechneten Wert E. Ziehen wir nur diese Einheit aus dem Entwurf heraus, ergibt sich hochsprachlich folgender Code, den wir im folgenden mit P bezeichnen werden:

Codefragment P - Berechnung von E

```
1  E, e := 0; Z_{-1}^E := x_0 · Y;
2  for i := 0 to n−1 do begin
3            E_i := E_{i−1} + Z_{i−1}^E;
4            e_i := E_i mod 2;
5            E_i := E_i div 2;
6            Z_i^E := x_{i+1} · Y;
7  end;
```

Man beachte die sequenzielle Abfolge bei der Berechnung von E und Z, die hier aus Gründen der Einfachheit eingeführt wurde. Schließlich ist für es für die mathematische Betrachtung unerheblich, ob diese Schritte gleichzeiitg oder aufeinanderfolgend ausgeführt werden, da durch die Datenunabhängigkeit beider Operationen keine Interferenz auftritt.

Wir beweisen nun induktiv die Kongruenz 4.5, die nach Ausführung des Programms P gilt:

$$E \equiv X \cdot Y \operatorname{div} 2^n. \tag{4.5}$$

Dazu verwenden wir das Zusicherungsnetzwerk 4.6:

$\{true\}$
E,e:=0;$Z_{-1}^E := x_0 \cdot Y$;
$\{E_{i-1} = 0 \wedge Z_{-1}^E = x_0 \cdot Y\}$
for i:= 0 to n-1 do begin
$\{Q_0 := E_{i-1} = (x_{i-1}...x_0) \cdot Y \operatorname{div} 2^i \wedge Z_{i-1}^E = x_i \cdot Y\}$
E_i:=E_{i-1} + Z_{i-1}^E;
$\{Q_1 := E_i \equiv (x_i...x_0) \cdot Y \operatorname{div} 2^i\}$
e_i:=E_i mod 2; (4.6)
$\{Q_2 := E_i \equiv (x_i...x_0) \cdot Y \operatorname{div} 2^i \wedge e_i = ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \operatorname{div} 2^i\}$
E_i:=$E_i \operatorname{div} 2$;
$\{Q_3 := E_i \equiv (x_i...x_0) \cdot Y \operatorname{div} 2^{i+1}\}$
Z_i^E:=$x_{i+1} \cdot Y$;
$\{Q_4 := E_i \equiv (x_i...x_0) \cdot Y \operatorname{div} 2^{i+1} \wedge Z_i^E = x_{i+1} \cdot Y\}$
end;
$\{E \equiv X \cdot Y \operatorname{div} 2^n\}$

Wir werden nun den Algorithmus als korrekt hinsichtlich des Zusicherungsnetzwerks 4.6 beweisen. Die anfängliche Zusicherung $\{E = 0 \wedge Z_{-1}^E = x_0 \cdot Y\}$ folgt offensichtlich aus der "assignment-rule". Wir fahren mit dem Schleifeninneren fort. Alle Herleitungen erfolgen nach der "assignment-" oder der "consequence-rule" und werden hier rein mathematisch plausibel gemacht. Man beachte, dass die Zusicherung Q_2 (genauer: die Aussage über e_i) keinerlei Bedeutung für den Beweis dieses Methodenausschnitts hat; doch wir werden sie im späteren Verlauf nutzen können.

1. $\models \{Q_0\}E_i := E_{i-1} + Z^E_{i-1}; \{Q_1\}$:

   Es gilt: $E_{i-1} = (x_{i-1}...x_0) \cdot Y \operatorname{div} 2^i \wedge Z^E_{i-1} = x_i \cdot Y$

   $$\begin{aligned}
   \Rightarrow E_i &= (x_{i-1}...x_0) \cdot Y \operatorname{div} 2^i + x_i \cdot Y \\
   &= ((x_{i-1}...x_0) \cdot Y + x_i \cdot Y \cdot 2^i) \operatorname{div} 2^i \\
   &= (\sum_{j=0}^{i-1} x_j \cdot Y \cdot 2^j + x_i \cdot Y \cdot 2^i) \operatorname{div} 2^i \\
   &= (\sum_{j=0}^{i} x_j \cdot Y \cdot 2^j) \operatorname{div} 2^i \\
   &= (x_i...x_0) \cdot Y \operatorname{div} 2^i \\
   &\stackrel{def}{=} \{Q_1\}
   \end{aligned}$$

   Es folgt aus der "assigment-" und "consequence-rule": $\{Q_0\}E_i := E_{i-1} + Z^E_{i-1}; \{Q_1\}$

2. $\models \{Q_1\}e_i := E_i \bmod 2; \{Q_2\}$:

   Es sei $\{0,1\}^{2n} \ni (p_{i+n}...p_0) := (x_i...x_0) \cdot Y$.

   Dann gilt: $((x_i...x_0) \cdot Y \operatorname{div} 2^i) \bmod 2$

   $$\begin{aligned}
   &= ((p_{i+n}...p_0) \operatorname{div} 2^i) \bmod 2 \\
   &= (p_{i+n}...p_i) \bmod 2 \\
   &= p_i \\
   &= (p_i...p_0) \operatorname{div} 2^i \\
   &= ((p_{i+n}...p_0) \bmod 2^{i+1}) \operatorname{div} 2^i \\
   &= ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \operatorname{div} 2^i
   \end{aligned}$$

   Es folgt aus der "assigment-" und "consequence-rule": $\{Q_1\}e_i := E_i \bmod 2; \{Q_2\}$

3. $\models \{Q_2\}E_i := E_i \operatorname{div} 2; \{Q_3\}$: folgt offensichtlich aus der "assigment-" und "consequence-rule".

4. $\models \{Q_3\}Z^E_i := x_{i+1} \cdot Y; \{Q_4\}$: folgt offensichtlich aus der "assigment-" und "consequence-rule".

Durch wiederholtes Anwenden der "sequential-composition-rule" erhalten wir nun:

$$\begin{aligned} \models &\{Q_0\} \\ & \quad E_i := E_{i-1} + Z^E_{i-1}; \\ P_{for} := & \quad e_i := E_i \bmod 2; \\ & \quad E_i := E_i \operatorname{div} 2; \\ & \quad Z^E_i := x_{i+1} \cdot Y; \\ &\{Q_4\} \end{aligned}$$

Weiter folgt $\models \{Q_4\} i := i+1; \{Q_0\} \Leftrightarrow \models \{E_i \equiv (x_i ... x_0) \cdot Y \operatorname{div} 2^{i+1} \wedge Z^E_i = x_{i+1} \cdot Y\} i := i+1; \{E_{i-1} = (x_{i-1} ... x_0) \cdot Y \operatorname{div} 2^i \wedge Z^E_{i-1} = x_i \cdot Y\}$ direkt aus der "assignment-rule". Aus der "sequential-composition-rule" folgt also $\models \{Q_0\} P_{for}; i := i+1; \{Q_0\}$. Offensichtlich folgt $\models \{E = 0 \wedge Z^E_{-1} = x_0 \cdot Y \wedge i = 0\} \rightarrow \{Q_0\}$ und damit aus der "for-statement-rule":

$$\models \{Q_0\} \text{ for i:=0 to n-1 do } P_{for}; \{Q_0 \wedge i = n\}$$

. Eine letzte Anwendung der "consequence-" und "sequential-composition-rule" ergibt also:

$$\begin{aligned} & \models \{true\} \text{ P; } \{Q_0 \wedge i = n\} \\ \Rightarrow & \models \{true\} \text{ P; } \{E_{n-1} = ((x_{n-1} ... x_0) \cdot Y) \operatorname{div} 2^n\} \\ \Leftrightarrow & \models \{true\} \text{ P; } \{E_{n-1} = (X \cdot Y) \operatorname{div} 2^n\} \end{aligned} \tag{4.7}$$

Die erste Eigenschaft der Methode ist also bewiesen: E wird korrekt berechnet.

4.5.2 Beweis der Korrekt der Methode für die Basis 2 - Teil 2

Den zweiten Teil der Archtitektur - die Komponente $\mathcal{K}$ - als korrekt zu beweisen wird sich als etwas aufwendiger gestalten als zuvor. Zunächst fassen wir die Komponente $\mathcal{K}$ wieder hochsprachlich auf, wobei wir jegliche Nebenläufigkeit aus Gründen der Simplizität entfernen; auch hier treten aufgrund der Taktung keinerlei Interferenzen beim Schreiben und Lesen der Daten auf.

Codefragment P - Berechnung von K

```
1  S , Z^K ,K, k := 0;
2  for  i := 0  to  n−1  do  begin
3              S_i := e_{i−1} + k_{i−1} ;
```

4 $\quad Z_i^K := S_{i-1} \cdot 2^{-3} \bmod M\,;$
5 $\quad K_i := K_{i-1} + Z_{i-1}^K\,;$
6 $\quad k_i := K_i \bmod 2\,;$
7 $\quad K_i := K_i \operatorname{div} 2\,;$
8 **end**;

Wir werden nun folgende Eigenschaft dieses Methodenabschnitts beweisen:

$$K+(k+e)\cdot 2^{-1} \bmod M + Z^K + S\cdot 2^{-2} \bmod M \equiv ((X\cdot Y) \bmod 2^n)\cdot 2^{-n} \bmod M \tag{4.8}$$

Zunächst einmal: was bedeutet die Kongruenz 4.8? Sie gibt an, dass in der Einheit $\mathcal{K}$ das gewünschte Ergebnis $((X \cdot Y) \bmod 2^n) \cdot 2^{-n} \bmod M$ berechnet wird. Durch das Pipelining der Daten ist das Ergebnis jedoch über die gesamte Einheit - dh. über alle verwendeten Register - verteilt, wobei jedes Register Daten unterschiedlichen Alters enthält. Durch die Rückkopplung der Architektur des Ausgangs von $\mathcal{K}$ - also K - auf den Eingang von $\mathcal{K}$ - also S - ist ein "Leerlaufen" der Pipeline nicht möglich. Daher müssen alle nach dem n-ten Takt nachbearbeitet und ihrem "Alter" entsprechend im Ergebnis berücksichtigt werden, was zu dieser komplizierten Kongruenz führt.
Wie zuvor geben wir nun ein Zusicherungsnetzwerk an, was die gewünschte Eigenschaft an der Methode nachweisen wird:

$\{true\}$
S,Z^K,K,k:=0;
$\{S_{i-1} = Z^K_{i-1} = K_{i-1} = k_{i-1} = 0\}$
for i:= 0 to n-1 do begin
$\{Q_0 := K_{i-1} + (e_{i-1} + k_{i-1}) \cdot 2^{-1} \bmod M + Z^K_{i-1} + S_{i-1} \cdot 2^{-2} \bmod M$
$\equiv ((x_{i-1}...x_0) \cdot Y \bmod 2^i) \cdot 2^{-i} \bmod M$
$\wedge\, e_i = ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \operatorname{div} 2^i\}$
$S_i := e_{i-1} + k_{i-1}$;
$\{Q_1 := K_{i-1} + e_i + S_i \cdot 2^{-1} \bmod M + Z^K_{i-1} + S_{i-1} \cdot 2^{-2} \bmod M$
$\equiv ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-i} \bmod M\}$
$Z^K_i := S_{i-1} \cdot 2^{-3} \bmod M$;
$\{Q_2 := (K_{i-1} + Z^K_{i-1} + e_i) \cdot 2^{-1} \bmod M + S_i \cdot 2^{-2} \bmod M + Z^K_i$
$\equiv ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-(i+1)} \bmod M\}$
$K_i := K_{i-1} + Z^K_{i-1}$;
$\{Q_3 := K_i \cdot 2^{-1} \bmod M + e_i \cdot 2^{-1} \bmod M + S_i \cdot 2^{-2} \bmod M + Z^K_i$
$\equiv ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-(i+1)} \bmod M\}$
$k_i := K_i \bmod 2$;
$K_i := K_i \operatorname{div} 2$;
$\{Q_4 := K_i + (e_i + k_i) \cdot 2^{-1} \bmod M + S_i \cdot 2^{-2} \bmod M + Z^K_i$
$\equiv ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-(i+1)} \bmod M\}$
end;
$\{K + (k + e) \cdot 2^{-1} \bmod M + Z^K + S \cdot 2^{-2} \bmod M \equiv ((X \cdot Y) \bmod 2^n) \cdot 2^{-n} \bmod M\}$ (4.9)

Beginnen wir nun mit dem Beweis.

1. $\models \{true\}$
   S,Z^K,K,k:=0;
   $\{S_{i-1} = Z^K_{i-1} = K_{i-1} = k_{i-1} = 0\}$:

   Dieser Programmschritt erschließt sich trivialerweise aus der ”assignment-rule”.

2. $\models \{Q_0\}\ S_i := e_{i-1} + k_{i-1};\ \{Q_1\}$:

Obwohl dieser Schritt nur eine simple Zuweisung enthält, ist hier mathematisch bei Folgerung der Nachbedingung einiges zu beachten. Trivial folgt aus "consequence-" und "assignment-rule":

$$\begin{aligned}
&\models\{Q_0\} = \{K_{i-1} + (e_{i-1} + k_{i-1}) \cdot 2^{-1} \bmod M + Z_{i-1}^K + S_{i-1} \cdot 2^{-2} \bmod M \\
&\qquad\qquad \equiv ((x_{i-1}...x_0) \cdot Y \bmod 2^i) \cdot 2^{-i} \bmod M \\
&\qquad\qquad \wedge e_i = ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \operatorname{div} 2^i\} \\
&S_i := e_{i-1} + k_{i-1}; \\
&\{K_{i-1} + S_i \cdot 2^{-1} \bmod M + Z_{i-1}^K + S_{i-1} \cdot 2^{-2} \bmod M \\
&\quad \equiv ((x_{i-1}...x_0) \cdot Y \bmod 2^i) \cdot 2^{-i} \bmod M
\end{aligned}$$

Doch Q_1 enthält auch eine Aussage über e_i. In Kapitel 4.5.1 haben wir bereits bewiesen, dass der e im $i-ten$ Takt zugewiesene Wert $((x_i...x_0) \cdot Y \bmod 2^{i+1}) \operatorname{div} 2^i$ entspricht. Zur Erinnerung ist dies in Q_0 noch einmal vermerkt, aber nicht weiter hergeleitet. Wir möchten nun folgendes zeigen:

$$\begin{aligned}
&e_i \ + \ ((x_{i-1}...x_0) \cdot Y \bmod 2^i) \cdot 2^{-i} \bmod M \\
&\equiv ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-i} \bmod M
\end{aligned}$$

Es gilt:

$$\begin{aligned}
&(x_i...x_0) \cdot Y \bmod 2^{i+1} \\
&= ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \operatorname{div} 2^i \cdot 2^i + ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \bmod 2^i \\
&= ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \operatorname{div} 2^i \cdot 2^i + (x_i...x_0) \cdot Y \bmod 2^i \\
&= ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \operatorname{div} 2^i \cdot 2^i + (x_{i-1}...x_0) \cdot Y \bmod 2^i \quad (*)
\end{aligned}$$

Die letzte Umformung setzt die Gleichheit von $(x_{i-1}...x_0) \cdot Y \bmod 2^i$ und $(x_i...x_0) \cdot Y \bmod 2^i$ voraus. Diese gilt, da die Operation $\bmod 2^i$ nur die letzten $i-1$ Bits betrifft, das Bit x_i aber mit dem Faktor 2^i in das Produkt eingeht und so das Ergebnis der Operation $\bmod 2^i$ nicht betrifft.

Weiter gilt:

$$\begin{aligned}
&e_i \ + \ ((x_{i-1}...x_0) \cdot Y \bmod 2^i) \cdot 2^{-i} \bmod M \\
&\equiv (((x_i...x_0) \cdot Y \bmod 2^{i+1}) \operatorname{div} 2^i \ + \ ((x_{i-1}...x_0) \cdot Y \bmod 2^i) \cdot 2^{-i}) \bmod M \\
&= (((x_i...x_0) \cdot Y \bmod 2^{i+1}) \operatorname{div} 2^i \cdot 2^i \ + \ (x_{i-1}...x_0) \cdot Y \bmod 2^i) \cdot 2^{-i} \bmod M \\
&\overset{(*)}{=} ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-i} \bmod M.
\end{aligned}$$

Es folgt:

$$K_{i-1} + e_i + S_i \cdot 2^{-1} \bmod M + Z_{i-1}^K + S_{i-1} \cdot 2^{-2} \bmod M$$

$\equiv e_i \ + \ ((x_{i-1}...x_0) \cdot Y \bmod 2^i) \cdot 2^{-i} \bmod M$
$\equiv ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-i} \bmod M$
also die Zusicherung Q_1.

3. $\models \{Q_1\}\ Z_i^K := S_{i-1} \cdot 2^{-3} \bmod M;\ \{Q_2\}$:

Auch dieser Programmschritt beinhaltet nur eine simple Zuweisung. Wieder ist die logische Folgerung von $\{Q_1[S_{i-1} \cdot 2^{-3} \bmod M / Z_i^K]\} \rightarrow \{Q_2\}$ dennoch etwas anspruchsvoller.

Es gilt:
$K_{i-1} + e_i + S_i \cdot 2^{-1} \bmod M + Z_{i-1}^K + S_{i-1} \cdot 2^{-2} \bmod M$
$\equiv ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-i} \bmod M$
$\Rightarrow (K_{i-1} + e_i + S_i \cdot 2^{-1} \bmod M + Z_{i-1}^K + S_{i-1} \cdot 2^{-2} \bmod M) \cdot 2^{-1} \bmod M$
$\equiv (((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-i} \bmod M) \cdot 2^{-1} \bmod M$
$\Rightarrow (K_{i-1} + Z_{i-1}^K + e_i) \cdot 2^{-1} \bmod M + S_i \cdot 2^{-2} \bmod M + S_{i-1} \cdot 2^{-3} \bmod M$
$\equiv ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-(i+1)} \bmod M$
Es folgt also zusammen mit $Z_i^K = S_{i-1} \cdot 2^{-3} \bmod M$:
$\{(K_{i-1} + Z_{i-1}^K + e_i) \cdot 2^{-1} \bmod M + S_i \cdot 2^{-2} \bmod M + Z_i^K$
$\equiv ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-(i+1)} \bmod M\}$
$= \{Q_2\}$

Die Aussage folgt also direkt aus der "assignment-" und mit etwas Aufwand aus der "consequence-rule".

4. $\models \{Q_2\}\ K_i := K_{i-1} + Z_{i-1}^K;\ \{Q_3\}$:

Diese Aussage folgt direkt aus der "assignment-" und der "consequence-rule". Zur Erinnerung:

$$\begin{aligned}\{Q_2\} = \{&(K_{i-1} + Z_{i-1}^K + e_i) \cdot 2^{-1} \bmod M + S_i \cdot 2^{-2} \bmod M + Z_i^K \\ &\equiv ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-(i+1)} \bmod M\}, \\ \{Q_3\} = \{&K_i \cdot 2^{-1} \bmod M + e_i \cdot 2^{-1} \bmod M + S_i \cdot 2^{-2} \bmod M + Z_i^K \\ &\equiv ((x_i...x_0) \cdot Y \bmod 2^{i+1}) \cdot 2^{-(i+1)} \bmod M\}\end{aligned}$$

5. $\models \{Q_3\}\ k_i := K_i \bmod 2;\ K_i := K_i \operatorname{div} 2;\ \{Q_4\}$:

Wir werden hier gleich zwei Schritte gleichzeitig mit einem impliziten Gebrauch der "sequential-composition-rule" abhandeln, da so der logische Zusammenhang besser zu verstehen ist.

Es gilt:
$K_i = K_i \operatorname{div} 2 \cdot 2 + K_i \bmod 2$
$\Rightarrow K_i \cdot 2^{-1} \bmod M = (K_i \operatorname{div} 2 \cdot 2 + K_i \bmod 2) \cdot 2^{-1} \bmod M$
$\equiv (K_i \operatorname{div} 2 \cdot 2) \cdot 2^{-1} \bmod M \ + \ (K_i \bmod 2) \cdot 2^{-1} \bmod M$
$\equiv K_i \operatorname{div} 2 \ + \ (K_i \bmod 2) \cdot 2^{-1} \bmod M$
Es folgt also mit der "assignment-rule":
$\{K_i \cdot 2^{-1} \bmod M + U \equiv V\}$, (U,V beliebig)
$k_i := K_i \bmod 2$; $K_i := K_i \operatorname{div} 2$;
$\{K_i + k_i \cdot 2^{-1} \bmod M + U \equiv V\}$

Es folgt also insgesamt die Gültigkeit von
$\models \{Q_3\}\ k_i := K_i \bmod 2;\ K_i := K_i \operatorname{div} 2;\ \{Q_4\}$

6. Durch wiederholtes Anwenden der "sequential-composition-rule" erhalten wir also:

$$\begin{aligned} \models & \{Q_0\} \\ & P_{for} := \begin{array}{l} E_i := E_{i-1} + Z^E_{i-1}; \\ e_i := E_i \bmod 2; \\ E_i := E_i \operatorname{div} 2; \\ Z^E_i := x_{i+1} \cdot Y; \end{array} \\ & \{Q_4\} \end{aligned}$$

7. $\models \{Q_4\}\ i := i + 1\ \{Q_0\}$:
Eine direkte Folge aus der "assignment-" und der "consequence-rule" - wiederum mit dem Verweis auf Kapitel 4.5.1, um die Aussage über e_i treffen zu können.

8. $\models \{S_{i-1} = Z^K_{i-1} = K_{i-1} = k_{i-1} = 0\}$
for i:= 0 to n-1 do P_{for};
$\{Q_0 \wedge i = n\}$:

Diese Gültigkeit lässt sich aus der "for-statement-rule" ableiten. Denn die Gültigkeit von
$\{S_{i-1} = Z^K_{i-1} = K_{i-1} = k_{i-1} = 0 \wedge i = 0\} \rightarrow Q_0$
ist offensichtlich: beide Seiten der Kongruenz ergeben 0. Und mit (6.) und (7.) folgt aus der "sequential-composition-rule"
$\{Q_0\}\ P_{for};\ i := i + 1;\ \{Q_0\}$

9. $\models \{true\}$ P; $\{Q_0 \wedge i = n\}$
$= K_{i-1} + (e_{i-1} + k_{i-1}) \cdot 2^{-1} \bmod M + Z^K_{i-1} + S_{i-1} \cdot 2^{-2} \bmod M$

$$
\begin{aligned}
&\equiv ((x_{i-1}...x_0) \cdot Y \bmod 2^i) \cdot 2^{-i} \bmod M \\
&\wedge i = n \\
&= K + (e+k) \cdot 2^{-1} \bmod M + Z^K + S \cdot 2^{-2} \bmod M \\
&\equiv (X \cdot Y \bmod 2^n) \cdot 2^{-n} \bmod M
\end{aligned}
$$

folgt also aus einer abschließenden Anwendung der "sequntial-composition-rule" auf (1.) und (8.). Es gilt also die Aussage 4.8 und der zweite Teil des Beweises sit abgeschlossen.

4.5.3 Beweis der Korrekt der Methode für die Basis 2 - Teil 3

Nun fügen wir die in den Kapiteln 4.5.1 und 4.5.2 gewonnen Erkenntnisse zusammen und beweisen insgesamt die Korrektheit der Methode durch die gesuchte Kongruenz 4.4:

$$
\begin{aligned}
&E + K + (e+k) \cdot 2^{-1} \cdot M + Z^K + S \cdot 2^{-2} \bmod M \\
&\equiv (X \cdot Y) \cdot 2^{-n} \bmod M
\end{aligned}
$$

Kombinieren wir die Ergebnisse der beiden Beweise erhalten wir

$$
\begin{aligned}
&E + K + (e+k) \cdot 2^{-1} \cdot M + Z^K + S \cdot 2^{-2} \bmod M \\
\equiv &X \cdot Y \operatorname{div} 2^n \\
&+(X \cdot Y \bmod 2^n) \cdot 2^{-n} \bmod M \\
\equiv &((X \cdot Y \operatorname{div} 2^n) \cdot 2^n) \cdot 2^{-n} \bmod M \\
&+(X \cdot Y \bmod 2^n) \cdot 2^{-n} \bmod M \\
\equiv &((X \cdot Y \operatorname{div} 2^n) \cdot 2^n + X \cdot Y \bmod 2^n) \cdot 2^{-n} \bmod M \\
= &(X \cdot Y) \cdot 2^{-n} \bmod M
\end{aligned}
$$

Das Ergebnis ist also kongruent zu dem gesuchten Wert und damit ist der Beweis abgeschlossen.

Doch wie nah dran ist das Ergebnis zu dem eigentlichen Wert - dh. um wieviele Vielfache von M größer? Für die Summanden des Ergebnisses gilt:

- $E < M$: aus $Z^E < M$ $(Y < M)$ folgt $E + Z^E < 2 \cdot M \Rightarrow E \operatorname{div} 2 < M$
- $K < M$: denn $Z^K < M$, also $K + Z^K < 2 \cdot M \Rightarrow K \operatorname{div} 2 < M$
- $Z^K < M$: siehe oben
- $(e+k) \cdot 2^{-1} \bmod M < M$: offensichtlich, da bereits modular.
- $S \cdot 2^{-2} \bmod M < M$: offensichtlich.

Es folgt also dass das Ergbnis insgesamt kleiner $5 \cdot M$ ist und damit in höchstens 5 Schritten korrigiert werden kann.

4.6 Die zweistufige Methode für höhere Zahlenbasen (8. Entwurf)

Die in den vorherigen Kapiteln entwickelte Methode lässt sich auch für höhere zugrundeliegende Zahlenbasen nutzbar machen. Der offensichtliche Vorteil einer höheren Zahlenbasis ist die pro Takt verarbeitete höhere Anzahl von Eingabebits und damit ein besseres Zeitverhalten. Der offensichtliche Nachteil ist, dass ein höherer Speicher- und Rechenaufwand nötig ist, um die Lookuptables zu füllen und die Eingabebits zu verarbeiten.

Wir betrachten nun einen allgemeinen, skalierbaren Entwurf nach dem Beispiel der Methode für die Basis 2. Aus Gründen des praktischen Nutzens und der allgemeinen Architektur, werden wir die möglichen Zahlenbasen auf Zweierpotenzen beschränken. Wir betrachten also die Basis $b = 2^k$. Zudem werden wir die Allgemeinheit weiter dadurch einschränken, dass wir $k|n$ fordern. Dies vereinfacht die Beweisführung und die theoretischen Betrachtungen der Methode. Wir setzen $r := \frac{n}{k}$. Bei einer Implementierung lassen sich die Eingabewerte hinreichend Verlängern (indem man Nullen hinzufügt), wenn die Bedingung $k|n$ nicht zutrifft.

Bei der Erweiterung des Entwurfs für die Basis 2 auf die Basis $b = 2^k$ ist folgendes zu beachten:

1. Die Eingabe von X erfolgt in Teilen zu je k Bits. Daher muss die LUT, die Z^E ausgibt, $(x_{ik+k-1}...x_{ik}) \cdot Y$ in allen Eingabekombinationen enthalten. Das entspricht genau b Werten.

2. Von den Zwischenergebnissen E und K werden nach der Verarbeitung je k Bits abgeschnitten - und da wir Carry-Save-Addierer verwenden wollen, fallen diese Bits doppelt an. Die Register e und k werden also je $k + (k-1)$ Bits lang (von den Carry-Registern müssen nur $(k-1)$ Bit gespeichert werden, da das letzte Bit grundsätzlich '0' ist).

3. Die Zwischensumme S erhält nun

$$2 \cdot (k + (k-1)) = 2 \cdot (2k-1) = 4k-2$$

   Bits zur Eingabe, und muss *k+1* Bits Ausgabe an die LUT vor Z^K liefern. Denn die Summe S ist durch $2b$ beschränkt - genauer: durch $2(b-1)$. Die Summe kann also Werte zwischen 0 und $2(2^k-1) = 2^{k+1}-2$ annehmen.

4. Die LUT vor Z^K benötigt wie wir zuvor eingesehen haben $2^{k+1}-1$ (ca. zweimal die Basis) Einträge.

Hochsprachlich lässt sich die verallgemeinerte Version der Methode wie folgt erfassen:

Entwurf zur Basis b - hochsprachlich

```
1  E, e, S, Z, K, k := 0;
2  for i := 0 to r−1 do begin
3              co begin
4                          Z_i^E := (x_{(i+2)k−1} ... x_{(i+1)k}) · Y;
5              //
6                          E_i := E_{i−1} + Z_{i−1}^E;
7                          e_i := E_i mod 2^k;
8                          E_i := E_i div 2^k;
9              //
10                         S_i := e_{i−1} + k_{i−1};
11             //
12                         Z_i^K := S_{i−1} · 2^{−3k} mod M;
13             //
14                         K_i := K_{i−1} + Z_{i−1}^K;
15                         k_i := K_i mod 2^k;
16                         K_i := K_i div 2^k;
17             end;
18 end;
19 return  E_{n−1} + (e_{n−1} + k_{n−1}) · 2^{−k} mod M + S_{n−1} · 2^{−2k} mod M + Z_{n−1} + K_{n−1};
```

Man beachte die Faktoren mit denen das "Alter" der Informationen berücksichtigt wird: "$Z_i^K := S_{i-1} \cdot 2^{-3k} \bmod M$;" in Zeile 12, bzw. "return $E_{n-1} + (e_{n-1} + k_{n-1}) \cdot 2^{-k} \bmod M + S_{n-1} \cdot 2^{-2k} \bmod M + Z_{n-1} + K_{n-1}$;" in Zeile 19. Da in jedem Takt die Register E bzw. K um k Bits geschoben werden (also durch 2^k "geteilt" werden), müssen die abgetrennten Bits e und k mehrfach durch 2^k geteilt werden, damit die die richtige Wertigkeit haben, sobald sie erneut in die Berechnung einfliessen (Zeile 12), bzw. ihr Alter muss bei der Nachberechnung berücksichtigt werden (Zeile 19). Abbildung 4.9 zeigt den zugehörigen Hardware-Entwurf.

4.6.1 8. Entwurf - Analyse

Die Lookuptabellen beginnen also mit höheren Basen schnell zu wachsen: die obere LUT benötigt 2^k Einträge, um die Vielfachen von Y aufzunehmen - die untere $2^{k+1} - 1$ für die Korrekturwerte. Dazu kommen wie zuvor 6 Register und 2 Carry-Save-Addierer. Der Flächenbedarf berechnet sich also zu

$$6 \cdot 0,5n + 2 \cdot n + (2^k + 2^{k+1} - 1) \cdot 0,125n = 5n + (3 \cdot 2^k - 1) \cdot 0,125n$$

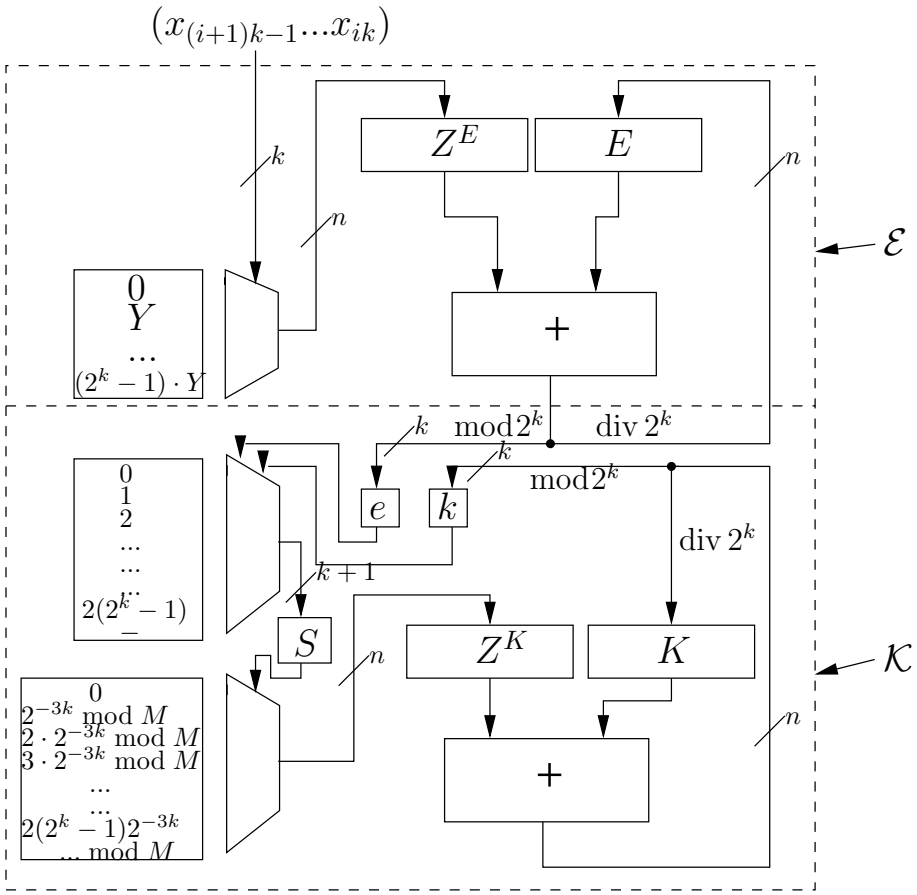


Abbildung 4.9: Entwurf 8

Flächeneinheiten - abhängig von der gewählten Zahlenbasis.

Der Zeitaufwand beträgt n/k Zeiteinheiten.

In Abhängigkeit von der Basis ergeben sich also die folgenden AT-Produkte:

Tabelle: AT-Komplexität der zweistufige Methode mit größeren Zahlenbasen

k=1	k=2	k=3	k=4
$5,625n^2$	$3,125n^2$	$(2,583n^2)$	$(2,688n^2)$

Hier ist aber folgendes anzumerken: wie oben angesprochen, werden bei disem Entwurf für höhere Zahlenbasen $4k-2$ Bits für die Zwischensumme aufaddiert, um die LUT mit den Korrekturwerten zu adressieren. Das bedeutet, dass bereits bei der Basis 8 zehn Bits addiert werden müssen, bzw. dass die LUT für die Zwischensumme 2^{10} Einträge fassen muss. Dies kann nicht mehr unter den Annahmen unseres Komplexitätsmodells (dem Zeitverbrauch 1 für den Zugriff auf eine Lookuptable) realisiert werden. Und eine Erhöhung der Taktdauer würde sich dramatisch auf die Architektur auswirken, da die Struktur eine Pipeline ist und so jede Stufe verzögert werden würde. Es kann also das Optimum für die AT-Komplexität in der obigen Tabelle mit dieser Architektur praktisch *nicht* realisiert werden.

4.6.2 Beweis zur Korrektheit des 8. Entwurfs

Auch für diese verallgemeinerte Methode zur Basis b wollen wir die korrekte Funktionsweise beweisen. Wir werden dazu analog zur Vorgehensweise aus Kapitel 4.5 verfahren: zunächst teilen wir die Methode in zwei Komponenten $\mathcal{E}$ und $\mathcal{K}$ und beweisen deren Korrektheit, anschliessend fügen wir diese Ergebnisse zu einer Gesamtaussage zusammen. Sämtliche Zusicherungen und Beweisschritte sind hierbei analog zu denen in Kapitel 4.5. Daher sollte es keine Schwierigkeit darstellen, den Beweis zu verstehen und daher sind die Ausführungen der Beweisschritte hier etwas knapper gehalten. Die zu beweisende Eigenschaft der Methode ist analog zur Eigenschaft 4.4 (siehe Seite 58)

$$\begin{aligned} & E + K + (e+k) \cdot 2^{-k} \cdot M + Z^K + S \cdot 2^{-2k} \bmod M \\ & \equiv (X \cdot Y) \cdot 2^{-n} \bmod M \end{aligned} \tag{4.10}$$

4.6.3 Beweis der Korrektheit der Methode für die Basis b - Teil 1

Analog zu Kapitel 4.5.1 werden wir hier die Korrektheit der Komponente $\mathcal{E}$ zeigen. Dazu weisen wir folgende Eigenschaft nach:

$$E \equiv X \cdot Y \operatorname{div} 2^n. \tag{4.11}$$

Das hochsprachliche Äquivalent zur Komponente $\mathcal{E}$ lautet wie folgt:

Codefragment P - Berechnung von E

```
1  E, e := 0; Z^E_{-1} := (x_{k-1}...x_0) · Y;
2  for i := 0 to r−1 do begin
3              E_i := E_{i-1} + Z^E_{i-1};
4              e_i := E_i mod 2^k;
5              E_i := E_i div 2^k;
6              Z^E_i := (x_{(i+2)k-1}...x_{(i+1)k}) · Y;
7  end;
```

Wie zuvor haben wir hier auf eine nebenläufige Darstellung der Berechnungen von E und Z^E verzichtet, da es den Sachverhalt nur unötig verkompliziert. Zusicherungsnetzwerk 4.12 soll nun genügen, die gewünschte Eigenschaft zu beweisen.

$\{true\}$
E,e:=0;$Z^E_{-1} := (x_{k-1}...x_0) \cdot Y$;
$\{E_{-1} = e_{-1} = 0 \wedge Z^E_{-1} = (x_{k-1}...x_0) \cdot Y\}$
for i:= 0 to r-1 do begin
$\{Q_0 := E_{i-1} = (x_{ik-1}...x_0) \cdot Y \operatorname{div} 2^{ik} \wedge Z^E_{i-1} = (x_{(i+1)k-1}...x_{ik}) \cdot Y\}$
E_i:=E_{i-1} + Z^E_{i-1};
$\{Q_1 := E_i \equiv (x_{(i+1)k-1}...x_0) \cdot Y \operatorname{div} 2^{ik}\}$
e_i:=$E_i \bmod 2^k$;
$\{Q_2 := E_i \equiv (x_{(i+1)k-1}...x_0) \cdot Y \operatorname{div} 2^{ik} \wedge e_i = ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \operatorname{div} 2^{ik}\}$
E_i:=$E_i \operatorname{div} 2^k$;
$\{Q_3 := E_i \equiv (x_{(i+1)k-1}...x_0) \cdot Y \operatorname{div} 2^{(i+1)k}\}$
Z^E_i:=$(x_{(i+2)k-1}...x_{(i+1)k}) \cdot Y$;
$\{Q_4 := E_i \equiv (x_{(i+1)k-1}...x_0) \cdot Y \operatorname{div} 2^{(i+1)k} \wedge Z^E_i = (x_{(i+2)k-1}...x_{(i+1)k}) \cdot Y\}$
end;
$\{E \equiv X \cdot Y \operatorname{div} 2^n\}$

(4.12)

Wir werden nun die Gültigkeit des Zusicherungsnetzwerka induktiv herleiten.

1. $\models\{true\}$
   E,e:=0;$Z^E_{-1} := (x_{k-1}...x_0) \cdot Y$;
   $\{E_{-1} = e_{-1} = 0 \wedge Z^E_{-1} = (x_{k-1}...x_0) \cdot Y\}$:

   folgt offensichtlich.

2. $\{E_{i-1} = 0 \wedge Z^E_{-1} = (x_{k-1}...x_0) \cdot Y \wedge i = 0\} \rightarrow Q_0$:

   folgt offensichtlich, da Z^E_{-1} gesetzt ist und alle anderen Werte hier gleich 0 sind.

3. $\models \{Q_0\}$ E_i:=E_{i-1} + Z^E_{i-1}; $\{Q_1\}$:

   Es gilt: $\{Q_0\} \stackrel{def}{=} \{E_{i-1} = (x_{ik-1}...x_0) \cdot Y \operatorname{div} 2^{ik} \wedge Z^E_{i-1} = (x_{(i+1)k-1}...x_{ik}) \cdot Y\}$
   $\Rightarrow E_{i-1} + Z^E_{i-1} = (x_{ik-1}...x_0) \cdot Y \operatorname{div} 2^{ik} + (x_{(i+1)k-1}...x_{ik}) \cdot Y$

$$
\begin{aligned}
&= ((x_{ik-1}...x_0)\cdot Y + (x_{(i+1)k-1}...x_{ik})\cdot Y\cdot 2^{ik}) \operatorname{div} 2^{ik} \\
&= (\textstyle\sum_{j=0}^{ik-1} x_j\cdot Y\cdot 2^j + \underbrace{(\sum_{l=ik}^{(i+1)k-1} x_l\cdot Y\cdot 2^{l-ik})\cdot 2^{ik}}_{\begin{array}{l}=\sum_{l=ik}^{(i+1)k-1} x_l\cdot Y\cdot 2^{l-ik}\cdot 2^{ik}\\ =\sum_{l=ik}^{(i+1)k-1} x_l\cdot Y\cdot 2^{l-ik+ik}\\ =\sum_{l=ik}^{(i+1)k-1} x_l\cdot Y\cdot 2^{l}\end{array}}) \operatorname{div} 2^{ik} \\
&= (\textstyle\sum_{j=0}^{(i+1)k-1} x_j\cdot Y\cdot 2^j) \operatorname{div} 2^{ik} \\
&= (x_{(i+1)k-1}...x_0)\cdot Y \operatorname{div} 2^{ik}
\end{aligned}
$$

$\Rightarrow Q_1$

4. $\models \{Q_1\}\ e_i{:=}E_i \bmod 2^k;\ \{Q_2\}$:
   Wie schon im ersten Beweis, hat auch hier die Aussage über e_i keine relevanz für die Funktionalität von $\mathcal{E}$. Sie wird jedoch im zweiten Abschnitt dieses Beweises benötigt.

   Es sei $\{0,1\}^{2n} \ni (p_{(i+1)k-1+n}...p_0) := (x_{(i+1)k-1}...x_0)\cdot Y$.
   Dann gilt: $(x_{(i+1)k-1}...x_0)\cdot Y \operatorname{div} 2^{ik} \bmod 2^k$
   $= (p_{(i+1)k-1+n}...p_0) \operatorname{div} 2^{ik} \bmod 2^k$
   $= (p_{(i+1)k-1+n}...p_{ik}) \bmod 2^k$
   $= (p_{(i+1)k-1}...p_{ik})$
   $= (p_{(i+1)k-1+n}...p_{ik}) \bmod 2^{(i+1)k}$
   $= ((p_{(i+1)k-1+n}...p_0) \bmod 2^{(i+1)k}) \operatorname{div} 2^{ik}$

   Es folgt $\{Q_2\}$ also aus "assignment-" und "consequence-rule".

5. $\models \{Q_2\}\ E_i{:=}E_i \operatorname{div} 2^k;\ \{Q_3\}$:

   Es gilt: $Q_2 \rightarrow E_i \equiv (x_{(i+1)k-1}...x_0)\cdot Y \operatorname{div} 2^{ik}$
   $\Rightarrow E_i \operatorname{div} 2^k \equiv (x_{(i+1)k-1}...x_0)\cdot Y \operatorname{div} 2^{ik} \operatorname{div} 2^k$
   $= (x_{(i+1)k-1}...x_0)\cdot Y \operatorname{div} 2^{ik+k}$
   $= (x_{(i+1)k-1}...x_0)\cdot Y \operatorname{div} 2^{(i+1)k}$
   $\Rightarrow \{Q_3\} \stackrel{def}{=} \{E_i \equiv (x_{(i+1)k-1}...x_0)\cdot Y \operatorname{div} 2^{(i+1)k}\}$

6. $\models \{Q_3\}\ Z_i^E{:=}(x_{(i+2)k-1}...x_{(i+1)k})\cdot Y;\ \{Q_4\}$:
   folgt direkt aus der "assignment-" und "consequence-rule".

7. $\models \{Q_0\}\ P_{for};\ \{Q_4\}$:
   durch wiederholte Anwendung der "sequential-composition-rule" erhalten wir

$$\begin{aligned} \models\{Q_0\}& \\ & E_i := E_{i-1} + Z^E_{i-1}; \\ P_{for} := & \ e_i := E_i \bmod 2^k; \\ & E_i := E_i \operatorname{div} 2^k; \\ & Z^E_i := (x_{(i+2)k-1}...x_{(i+1)k}) \cdot Y; \\ \{Q_4\}& \end{aligned}$$

8. $\models \{Q_4\}$ i:=i+1; $\{Q_0\}$:
   folgt offensichtlich.

9. $\models \{E_{i-1} = 0 \wedge Z^E_{-1} = (x_{k-1}...x_0) \cdot Y\}$
   for i:= 0 to r-1 do P_{for};
   $\{Q_0 \wedge i = r\}$:

   folgt mit (2.), (7.) und (8.) aus der "for-statement-rule".

10. $\models \{true\}$ P; $\{E \equiv X \cdot Y \operatorname{div} 2^n\}$:

    aus der "sequential-composition-rule" und (1.) und (9.) folgt
    $\models \{true\}$ P; $\{Q_0 \wedge i = r\}$.
    Weiter gilt:
    $\{Q_0 \wedge i = r\}$
    $\rightarrow\{E_r = (x_{rk-1}...x_0) \cdot Y \operatorname{div} 2^{rk}\}$
    $\rightarrow\{E_r = (x_{n-1}...x_0) \cdot Y \operatorname{div} 2^n\}$
    $\rightarrow\{E_r = X \cdot Y \operatorname{div} 2^n\}$

Die letzte Zusicherung entspricht der geforderten Eigenschaft 4.11 und die Komponente $\mathcal{E}$ funktioniert folglich korrekt.

4.6.4 Beweis der Korrekt der Methode für die Basis b - Teil 2

Der zweite Teil des Beweises erfolgt analog zu Kapitel 4.5.2. Wir werden die Korrektheit der Komponente $\mathcal{K}$ beweisen, indem wir folgende Eigenschaft

nach Terminierung des Algorithmus überprüfen:

$$K+(k+e)\cdot 2^{-k} \bmod M + Z^K + S\cdot 2^{-2k} \bmod M \equiv ((X\cdot Y) \bmod 2^n)\cdot 2^{-n} \bmod M \tag{4.13}$$

Zunächst erfassen wir die Komponente hochsprachlich:

Codefragment P - Berechnung von K

```
1  S, Z^K, K, k := 0;
2  for i := 0 to r-1 do begin
3            S_i := e_{i-1} + k_{i-1};
4            Z_i^K := S_{i-1} · 2^{-3k} mod M;
5            K_i := K_{i-1} + Z_{i-1}^K;
6            k_i := K_i mod 2^k;
7            K_i := K_i div 2^k;
8  end;
```

Betrachte nun folgendes Zusicherungsnetzwerk:

$\{true\}$
S,Z^K,K,k:=0;
$\{S_{i-1} = Z^K_{i-1} = K_{i-1} = k_{i-1} = 0\}$
for i:= 0 to r-1 do begin
$\{Q_0 := K_{i-1} + (e_{i-1} + k_{i-1}) \cdot 2^{-k} \bmod M + Z^K_{i-1} + S_{i-1} \cdot 2^{-2k} \bmod M$
$\equiv ((x_{ik-1}...x_0) \cdot Y \bmod 2^{ik}) \cdot 2^{-ik} \bmod M$
$\wedge\, e_i = ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \operatorname{div} 2^{ik}\}$
$S_i := e_{i-1} + k_{i-1}$;
$\{Q_1 := K_{i-1} + e_i + S_i \cdot 2^{-k} \bmod M + Z^K_{i-1} + S_{i-1} \cdot 2^{-2k} \bmod M$
$\equiv ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-ik} \bmod M\}$
$Z^K_i := S_{i-1} \cdot 2^{-3k} \bmod M$;
$\{Q_2 := (K_{i-1} + Z^K_{i-1} + e_i) \cdot 2^{-k} \bmod M + S_i \cdot 2^{-2k} \bmod M + Z^K_i$
$\equiv ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-(i+1)k} \bmod M\}$
$K_i := K_{i-1} + Z^K_{i-1}$;
$\{Q_3 := K_i \cdot 2^{-k} \bmod M + e_i \cdot 2^{-k} \bmod M + S_i \cdot 2^{-2k} \bmod M + Z^K_i$
$\equiv ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-(i+1)k} \bmod M\}$
$k_i := K_i \bmod 2^k$;
$K_i := K_i \operatorname{div} 2^k$;
$\{Q_4 := K_i + (e_i + k_i) \cdot 2^{-k} \bmod M + S_i \cdot 2^{-2k} \bmod M + Z^K_i$
$\equiv ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-(i+1)k} \bmod M\}$
end;
$\{K + (k + e) \cdot 2^{-k} \bmod M + Z^K + S \cdot 2^{-2k} \bmod M \equiv ((X \cdot Y) \bmod 2^n) \cdot 2^{-n} \bmod M\}$ (4.14)

Es folgt die induktive Herleitung der Korrektheit des Zusicherungsnetzwerks 4.14.

1. $\models \{true\}$ S,Z^K,K,k:=0; $\{S_{i-1} = Z^K_{i-1} = K_{i-1} = k_{i-1} = 0\}$:
   folgt offensichtlich.

2. $\{S_{i-1} = Z^K_{i-1} = K_{i-1} = k_{i-1} = 0 \wedge i = 0\} \rightarrow Q_0$:
   folgt offensichtlich mit der in Kapitel 4.6.3 Beweisschritt 1. getroffenen Aussage über e_{-1}.

3. $\models \{Q_0\}\ S_i := e_{i-1} + k_{i-1};\ \{Q_1\}$:

Es gilt:

$(x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}$

$= ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \operatorname{div} 2^{ik} \cdot 2^{ik}$

$\quad + ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \bmod 2^{ik}$

$= ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \operatorname{div} 2^{ik} \cdot 2^{ik}$

$\quad + (x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{ik}$

$= ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \operatorname{div} 2^{ik} \cdot 2^{ik}$

$\quad + (x_{ik-1}...x_0) \cdot Y \bmod 2^{ik}$ (*)

letztere Gleichung gilt,da die Operation mod2^{ik} die obersten k Bits von $(x_{(i+1)k-1}...x_0) \cdot Y$ nicht betrifft (siehe Kapitel 4.5.2: Schritt (2.)).

Weiter gilt:

$Q_0 \overset{def}{=} K_{i-1} + (e_{i-1} + k_{i-1}) \cdot 2^{-k} \bmod M + Z_{i-1}^K + S_{i-1} \cdot 2^{-2k} \bmod M$

$\equiv ((x_{ik-1}...x_0) \cdot Y \bmod 2^{ik}) \cdot 2^{-ik} \bmod M$

$\wedge e_i = ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \operatorname{div} 2^{ik}$

$\Rightarrow e_i\ +\ ((x_{ik-1}...x_0) \cdot Y \bmod 2^{ik}) \cdot 2^{-ik} \bmod M$

$\equiv [((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \operatorname{div} 2^{ik}\ +\ ((x_{ik-1}...x_0) \cdot Y \bmod 2^{ik}) \cdot 2^{-ik}] \bmod M$

$=[((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \operatorname{div} 2^{ik} \cdot 2^{ik}$

$\quad + ((x_{ik-1}...x_0) \cdot Y \bmod 2^{ik})] \cdot 2^{-ik} \bmod M$

$\overset{(*)}{\equiv} ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-ik} \bmod M$

Es folgt:

$\{Q_0 \wedge S_i = e_{i-1} + k_{i-1}\} \rightarrow Q_1$

4. $\models \{Q_1\}\ Z_i^K := S_{i-1} \cdot 2^{-3k} \bmod M;\ \{Q_2\}$:

Es gilt:

$Q_1 \overset{def}{=} K_{i-1} + e_i + S_i \cdot 2^{-k} \bmod M + Z_{i-1}^K + S_{i-1} \cdot 2^{-2k} \bmod M$

$\equiv ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-ik} \bmod M$

$\Rightarrow [K_{i-1} + e_i + S_i \cdot 2^{-k} \bmod M + Z_{i-1}^K + S_{i-1} \cdot 2^{-2k} \bmod M] \cdot 2^{-k} \bmod M$

$\equiv [((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-ik} \bmod M] \cdot 2^{-k} \bmod M$

$\Rightarrow (K_{i-1} + e_i + Z_{i-1}^K) \cdot 2^{-k} \bmod M + S_i \cdot 2^{-2k} \bmod M + S_{i-1} \cdot 2^{-3k} \bmod M$

$\equiv ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-(i+1)k} \bmod M$

Es folgt:

$Q_1 \rightarrow \{(K_{i-1} + e_i + Z_{i-1}^K) \cdot 2^{-k} \bmod M + S_i \cdot 2^{-2k} \bmod M + S_{i-1} \cdot 2^{-3k} \bmod M$

$\equiv ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-(i+1)k} \bmod M\}$

Also folgt:

$\{Q_1 \wedge Z_i^K = S_{i-1} \cdot 2^{-3k} \bmod M\}$
$\rightarrow \{(K_{i-1} + e_i + Z_{i-1}^K) \cdot 2^{-k} \bmod M + S_i \cdot 2^{-2k} \bmod M + S_{i-1} \cdot 2^{-3k} \bmod M$
$\equiv ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-(i+1)k} \bmod M$
$\wedge Z_i^K = S_{i-1} \cdot 2^{-3k} \bmod M\}$
$\rightarrow \{(K_{i-1} + e_i + Z_{i-1}^K) \cdot 2^{-k} \bmod M + S_i \cdot 2^{-2k} \bmod M + Z_i^K$
$\equiv ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-(i+1)k} \bmod M\}$
$\stackrel{def}{=} Q_2$

5. $\models \{Q_2\}\ K_i := K_{i-1} + Z_{i-1}^K;\ \{Q_3\}$:
folgt direkt aus "assigment-" und "consequence-rule".

6. $\models \{Q_3\}\ k_i := K_i \bmod 2^k;\ K_i := K_i \operatorname{div} 2^k;\ \{Q_4\}$:

Es gilt:
$K_i \cdot 2^{-k} = (K_i \operatorname{div} 2^k \cdot 2^k\ +\ K_i \bmod 2^k) \cdot 2^{-k} \bmod M$
$\equiv (K_i \operatorname{div} 2^k \cdot 2^k) \cdot 2^{-k} \bmod M\ +\ (K_i \bmod 2^k) \cdot 2^{-k} \bmod M$
$\equiv K_i \operatorname{div} 2^k\ +\ (K_i \bmod 2^k) \cdot 2^{-k} \bmod M\ (*)$
Es folgt:
$\{Q_3[K_i \operatorname{div} 2^k / K_i, K_i \bmod 2^k / k_i]\}$
$\stackrel{(*)}{\rightarrow} \{K_i + k_i \cdot 2^{-k} \bmod M + e_i \cdot 2^{-k} \bmod M + S_i \cdot 2^{-2k} \bmod M + Z_i^K$
$\equiv ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \cdot 2^{-(i+1)k} \bmod M\}$
$\stackrel{def}{=} Q_4$

7. $\models \{Q_0\}$
$$P_{for} := \begin{cases} S_i := e_{i-1} + k_{i-1}; \\ Z_i^K := S_{i-1} \cdot 2^{-3k} \bmod M; \\ K_i := K_{i-1} + Z_{i-1}^K; \\ k_i := K_i \bmod 2^k; \\ K_i := K_i \operatorname{div} 2^k; \end{cases}$$
$\{Q_4\}$:

folgt unmittelbar aus der "sequential-composition-rule" und (3.) - (6.).

8. $\models\ \{Q_0\}\ P_{for};\ i := i + 1;\ \{Q_0\}$:
Es folgt $\models \{Q_4\}\ i := i + 1;\ \{Q_0\}$ aus der "assignment-rule" und der Aussage aus Beweisteil 1 (siehe Zusicherungsnetzwerk 4.12, Zusicherung Q_2).

Die "sequential-composition-rule" liefert dann zusammen mit (7.) die gewünschte Gültigkeit.

9. $\models \{S_{i-1} = Z^K_{i-1} = K_{i-1} = k_{i-1} = 0\}$
   for i:= 0 to r-1 do P_{for};
   $\{Q_0 \wedge i = r\}$:

   folgt aus der "for-statement-rule" mit (2.) und (8.).

10. $\models \{true\}\ P;\ \{K + (k+e) \cdot 2^{-k} \bmod M + Z^K + S \cdot 2^{-2k} \bmod M \equiv ((X \cdot Y) \bmod 2^n) \cdot 2^{-n} \bmod M\}$:
    zunächst folgt aus (1.) und (9.) und der "sequential-composition-rule"

$$\left.\begin{array}{l} \models \{true\} \\ \quad S, Z^K, K, k := 0; \\ \quad P_{for}; \\ \{Q_0 \wedge i = r\}. \end{array}\right\} = P$$

Weiter folgt

$$\begin{aligned} \{Q_0 \wedge i = r\} \overset{def}{=} & \{K_{i-1} + (e_{i-1} + k_{i-1}) \cdot 2^{-k} \bmod M + Z^K_{i-1} + S_{i-1} \cdot 2^{-2k} \bmod M \\ & \equiv ((x_{ik-1}...x_0) \cdot Y \bmod 2^{ik}) \cdot 2^{-ik} \bmod M \\ & \wedge e_i = ((x_{(i+1)k-1}...x_0) \cdot Y \bmod 2^{(i+1)k}) \operatorname{div} 2^{ik} \\ & \wedge i = r\} \\ \rightarrow\ & K + (e+k) \cdot 2^{-k} \bmod M + Z^K_{i-1} + S_{i-1} \cdot 2^{-2k} \bmod M \\ & \equiv \underbrace{((x_{rk-1}...x_0) \cdot Y \bmod 2^{rk}) \cdot 2^{-rk} \bmod M} \\ & \quad = ((x_{n-1}...x_0) \cdot Y \bmod 2^n) \cdot 2^{-n} \bmod M \\ & \quad = (X \cdot Y \bmod 2^n) \cdot 2^{-n} \bmod M \end{aligned}$$

Es gilt also 4.13.

4.6.5 Beweis der Korrekt der Methode für die Basis b - Teil 3

Analog zum Beweis für die Basis 2 fassen wir nun die Ergebnisse aus Kapitel 4.6.3 und Kapitel 4.6.4 zusammen, um die Eigenschaft 4.10 zu erhalten. Nach Ausführung der Komponenten gilt:

1. $E \equiv X \cdot Y \operatorname{div} 2^n$

2. $K + (k + e) \cdot 2^{-k} \bmod M + Z^K + S \cdot 2^{-2k} \bmod M$
   $\equiv ((X \cdot Y) \bmod 2^n) \cdot 2^{-n} \bmod M$

Es folgt:

4.7 Die einstufige Methode für die Basis 2

In diesem Kapitel wollen wir die in Kapitel 4.3.5 entwickelte Methode noch einmal genauer betrachten und auf ihre korrekte Funktionsweise hin prüfen.

Wir verfahren dazu analog zu den Beweisen zuvor. Zunächst werden wir die Methode für die Basis 2 untersuchen und im folgenden Kapitel für die Basis 4. Höhere Zahlenbasen werden wir nicht betrachten aus Gründen, die im Folgenden noch erläutert werden.

4.7.1 Beweis der Korrektheit für die Basis 2

Aufgrund ihrer Einfachheit, werden wir diese Methode in nur einem Beweisschritt verifizieren können. Die gesuchte Eigenschaft, die es nachzuweisen gilt, lautet ähnlich wie zuvor:

$$E + Z + e \cdot 2^{-1} \bmod M \equiv X \cdot Y \cdot 2^{-n} \bmod M \tag{4.15}$$

Zur Erinnerung ist hier der hochsprachliche Entwurf der einstufigen Methode noch einmal aufgeführt:

3. Entwurf - hochsprachlich

```
1  E, e := 0;  Z_{-1} := x_0 · Y;
2  for  i := 0 to n−1 do begin
3              E_i := E_{i−1} + Z_{i−1};
4              e_i := E_i mod 2;
5              E_i := E_i div 2;
6              Z_i := x_{i+1} · Y + (e_{i−1} · 2^{−2}) mod M;
7  end;
8  return  E_n + Z_n + e_n · 2^{−1} mod M;
```

Obige Darstellung des Algortihmus' sieht wieder über die in Kapitel 4.3.5 angedeutete Parallelität hinweg da ja die Daten voneinander unabhängig (Interferenzfreiheit) sind. Zur Erinnerung: die Entwürfe in Pseudocode stellen keine Implementierungen, sondern lediglich Veranschaulichungen der Funktionsweise dar. Durch das Weglassen der Parallelität ändert sich nichts in der Berechnung. Auch die Ausführungsreihenfolge wurde in dieser Darstellung geändert, was ebenfalls nichts an der Berechnung ändert, da alle gelesenen

Daten fest stehen und zwar den Ergebnissen der letzten Schleifeniteration entsprechen.

Anhand von Zusicherungsnetzwerk 4.16 werden wir die gesuchte Eigenschaft der Methode nachweisen.

$\{true\}$
E_{-1},e_{-1}:=0; Z_{-1}:=$x_0 \cdot Y$;
$\{E_{-1} = e_{-1} = 0 \wedge Z_{-1} = x_0 \cdot Y\}$
for i:= 0 to n do begin
$\{Q_0 := E_{i-1} + Z_{i-1} + e_{i-1} \cdot 2^{-1} \bmod M \equiv (x_i ... x_0) \cdot Y \cdot 2^{-i} \bmod M\}$
E_i:=E_{i-1}+Z_{i-1};
$\{Q_1 := E_i + e_{i-1} \cdot 2^{-1} \bmod M + 2 \cdot x_{i+1} \cdot Y \equiv (x_{i+1} ... x_0) \cdot Y \cdot 2^{-i} \bmod M\}$
e_i:=$E_i \bmod 2$;
E_i:=$E_i \operatorname{div} 2$;
$\{Q := 2 \cdot E_i + e_i + e_{i-1} \cdot 2^{-1} \bmod M + 2 \cdot x_{i+1} \cdot Y \equiv (x_{i+1} ... x_0) \cdot Y \cdot 2^{-i} \bmod M\}$
$\{Q_3 := E_i + e_i \cdot 2^{-1} \bmod M + e_{i-1} \cdot 2^{-2} \bmod M + x_{i+1} \cdot Y$
$\equiv (x_{i+1} ... x_0) \cdot Y \cdot 2^{-(i+1)} \bmod M\}$
Z_i:=$x_{i+1} \cdot Y + e_{i-1} \cdot 2^{-2} \bmod M$;
$\{Q_4 := E_i + Z_i + e_i \cdot 2^{-1} \bmod M \equiv (x_{i+1} ... x_0) \cdot Y \cdot 2^{-(i+1)} \bmod M\}$
end;
$\{Q_5 := E_n + Z_n + e_n \cdot 2^{-1} \bmod M \equiv X \cdot Y \cdot 2^{-n} \bmod M\}$ (4.16)

Es folgt der Beweis obigen Netzwerks in einer Abfolge von Beweisschritten.

1. $\models \{true\} E_{-1}, e_{-1} := 0; Z_{-1} := x_0 \cdot Y; \{E_{-1} = e_{-1} = 0 \wedge Z_{-1} = x_0 \cdot Y\}$: Dieser Schritt folgt unmittelbar aus der "assignment-" und der "sequential-composition-rule".

2. $\models \{E_{-1} = e_{-1} = 0 \wedge Z_{-1} = x_0 \cdot Y \wedge i = 0\}$
   $\rightarrow \{E_{i-1} + Z_{i-1} + e_{i-1} \cdot 2^{-1} \bmod M \equiv (x_i ... x_0) \cdot Y \cdot 2^{-i} \bmod M\}$:
   Es gelte $i = 0$. Es folgt:
   $$(x_i ... x_0) \cdot Y \cdot 2^{-i} \bmod M = (x_0) \cdot Y \cdot 2^0 \bmod M$$
   $$= x_0 \cdot Y \quad \text{(unter der Annahme Y<M)}$$
   Weiter gilt: $E_{-1} + Z_{-1} + e_{-1} \cdot 2^{-1} \bmod M = 0 + x_0 \cdot Y + 0 = x_0 \cdot Y$. Es folgt also die Aussage aus der "consequence-rule".

3. $\models \{Q_0\}E_i := E_{i-1} + Z_{i-1}; \{E_i + e_{i-1} \cdot 2^{-1} \bmod M + 2 \cdot x_{i+1} \cdot Y \equiv (x_{i+1}...x_0) \cdot Y \cdot 2^{-i} \bmod M\}$:

   Zunächst folgt offenbar

   $\models \{Q_0\}E_i := E_{i-1}+Z_{i-1}; \{E_i+e_{i-1}\cdot 2^{-1} \bmod M \equiv (x_i...x_0)\cdot Y\cdot 2^{-i} \bmod M\}$

   aus der "assignment-rule".
   Es folgt weiterhin aus der "consequence-rule":

$$
\begin{aligned}
&\{E_i + e_{i-1} \cdot 2^{-1} \bmod M \equiv (x_i...x_0) \cdot Y \cdot 2^{-i} \bmod M\} \\
&\rightarrow \{E_i + e_{i-1} \cdot 2^{-1} \bmod M + 2 \cdot x_{i+1} \cdot Y \equiv \underbrace{(x_i...x_0) \cdot Y \cdot 2^{-i} \bmod M + 2 \cdot x_{i+1} \cdot Y}\} \\
&\qquad \equiv ((x_i...x_0) \cdot Y \cdot 2^{-i} + 2 \cdot x_{i+1} \cdot Y) \bmod M \\
&\qquad = ((x_i...x_0) \cdot Y + 2 \cdot x_{i+1} \cdot Y \cdot 2^{i}) \cdot 2^{-i} \bmod M \\
&\qquad \stackrel{def}{=} (\sum_{j=0}^{i} x_j \cdot 2^j \cdot Y + x_{i+1} \cdot 2^{i+1} \cdot Y) \cdot 2^{-i} \bmod M \\
&\qquad = (\sum_{j=0}^{i+1} x_j \cdot 2^j \cdot Y) \cdot 2^{-i} \bmod M \\
&\qquad \stackrel{def}{=} (x_{i+1}...x_0) \cdot Y \cdot 2^{-i} \bmod M \\
&\stackrel{def}{=} Q_1
\end{aligned}
$$

4. $\models \{Q_1\}e_i := E_i \bmod 2; E_i := E_i \operatorname{div} 2; \{Q_2\}$: Q_2 folgt unmittlebar aus der "assignment-" und der "sequential-composition-rule", denn es gilt:

$$E_i = 2 \cdot (E_i \operatorname{div} 2) + E_i \bmod 2$$

5. $\models \{Q_2\} \rightarrow \{Q_3\}$: Q_3 ist eine direkte Implikation von Q_2 und folgt also aus der "consequence-rule". Es gilt:

$$
\begin{aligned}
&Q_2 \stackrel{def}{=} \{2 \cdot E_i + e_i + e_{i-1} \cdot 2^{-1} \bmod M + 2 \cdot x_{i+1} \cdot Y \equiv (x_{i+1}...x_0) \cdot Y \cdot 2^{-i} \bmod M\} \\
&\rightarrow \{(2 \cdot E_i + e_i + e_{i-1} \cdot 2^{-1} \bmod M + 2 \cdot x_{i+1} \cdot Y) \cdot 2^{-1} \bmod M \\
&\qquad \equiv ((x_i...x_0) \cdot Y \cdot 2^{-i} \bmod M) \cdot 2^{-1} \bmod M\} \\
&\rightarrow \{2 \cdot E_i \cdot 2^{-1} \bmod M + e_i \cdot 2^{-1} \bmod M + e_{i-1} \cdot 2^{-2} \bmod M + 2 \cdot x_{i+1} \cdot Y \cdot 2^{-1} \bmod M \\
&\qquad \equiv (x_{i+1}...x_0) \cdot Y \cdot 2^{-i} \cdot 2^{-1} \bmod M\} \\
&\rightarrow \{E_i + e_i \cdot 2^{-1} \bmod M + e_{i-1} \cdot 2^{-2} \bmod M + x_{i+1} \cdot Y \\
&\qquad \equiv (x_{i+1}...x_0) \cdot Y \cdot 2^{-(i+1)} \bmod M\} \\
&\stackrel{def}{=} Q_3
\end{aligned}
$$

6. $\models \{Q_3\}Z_i := x_{i+1} \cdot Y + (e_{i-1} \cdot 2^{-2}) \bmod M\{Q_4\}$: Diese Zusicherung erfolgt trivialerweise aus der "assignment-rule".

7. $\models \{true\}P;\{Q_5\}$: Die Korrektheit des ganzen Programms folgt nun aus 4 Teilschritten:

   (a) Zunächst sei P_{for} das Programm, das dem Schleifeninnern entspricht:

   $$P_{for} := \begin{array}{l} E_i := E_{i-1} + Z_{i-1}; \\ e_i := E_i \bmod 2; \\ E_i := E_i \operatorname{div} 2; \\ Z_i := x_{i+1} \cdot Y + (e_{i-1} \cdot 2^{-2} \bmod M; \end{array}$$

   Aus den Gültigkeiten 3. bis 6. folgt mit der "sequantial-composition-rule":

   $$\models \{Q_0\}P_{for};\{Q_4\}$$

   (b) $\models \{Q_4\}i := i + 1;\{Q_0\}$: folgt direkt aus der "assignment-" und der "consequence-rule".

   (c) $\models \{Q_0 \wedge i = n\} \rightarrow \{Q_5\}$: Diese Gültigkeit folgt wiederum direkt aus der "consequence-rule" und $(x_n...x_0) = (0\ x_{n-1}...x_0) = (x_{n-1}...x_0) \stackrel{def}{=} X$.

   (d) $\models \{E_{-1} = e_{-1} = 0 \wedge Z_{-1} = x_0 \cdot Y\}P_{for};\{Q_5\}$: folgt also aus 2., a), b) und c) mit der "for-statement-rule".

   Die Gesamtaussage folgt also nach einer letzten Anwendung der "sequential-composition-rule" auf 1. und d).

Es folgt Aussage 4.15 und damit die Korrektheit der Methode.

4.8 Die einstufige Methode für die Basis 4

Wie schon bei den grundlegenden Methoden der Modularmultiplikation gesehen (siehe Kapitel 3), lässt sich die Effizienz von Algorithmen mitunter steigern, wenn man höhere Zahlenbasen zugrunde legt. Auch diese Methode kann davon profitieren, wenngleich auch nur die Erweiterung auf die Basis 4 möglich ist, wie wir später sehen werden.

Wenn wir die Methode nun auf die Basis 4 erweitern, ist Folgendes zu beachten:

- Pro Schleifendurchlauf werden zwei Bits in der Architektur verarbeitet - die Anzahl der Schleifendurchläufe halbiert sich also.
- In jedem Durchlauf muss das Ergebnis durch 4 geteilt werden, um mit den schneller wachsenden Zwischenergebnissen schrittzuhalten - Es werden also zwei Bits des Summenregisters S und ein Bit des Registers C (letzes Bit von C ist immer 0) abgetrennt und deren Summe muss in die Berechnung aufgenommen werden.
- Die Lookuptable enthält nun vier mögliche Eingaben für den Koeffizienten von X und muss diese mit allen fünf möglichen Summen der abgetrennten Bits von S und C korrigieren.
- Die Korrekturwerte für die abgetrennten Bits werden mit $2^{-4} \bmod M$ verrechnet, da die Stufe $\mathcal{E}$ die Zwischenergebnisse zweimal durch vier teilt, bevor die Korrekturwerte wieder einfließen.

An dieser Liste von Veränderungen ist einzusehen, warum hohe Zahlenbasen für diese Methode ein Problem sind: durch die größeren Koeffizienten, die jeden Schleifendurchlauf verarbeitet werden, und durch die größere Zahl an abgetrennten Bits wächst die Lookuptable quadratisch mit dem Logarithmus der Basis. Gerade durch die Verwendung von Carry-Save-Addierern muss ja fast die doppelte Anzahl an Bits abgeschnitten und verarbeitet werden als bei herkömmlichen Addierern. Die Lookuptable wird also sehr schnell sehr groß und vor allem auch langsam. Bei der Basis 8 z.B. wird schon eine LUT mit mehreren hundert Einträgen der Länge n benötigt, was die Annahme unseres Komplexitätsmodells, der Zugriff auf die Lookuptable brauche ungefähr die gleiche Zeit wie ein Volladdierer, unhaltbar macht.

Die hochsprachliche Darstellung ändert aich den obigen Angaben entsprechend:

3. Entwurf - hochsprachlich

```
1  E_{-1}, e_{-1} := 0;  Z_{-1} := (x_1, x_0) · Y;
2  for i := 0 to n/2 − 1 do begin
3              co begin
4                       Z_i := (x_{2(i+1)+1}, x_{2(i+1)}) · Y + (e_{i−1} · 2^{−4}) mod M;
5              //
6                       E_i := E_{i−1} + Z_{i−1};
7                       e_i := E_i mod 4;
8                       E_i := E_i div 4;
9              end;
10 end;
11 return  E_n + Z_n + e_n · 2^{−2} mod M;
```

Abbildung 4.10 zeigt den Hardwareentwurf für die Basis 4. Hier gilt es zu beachten, dass die Lookuptable mit 24 Werten (4 Eingangs- x 6 Korrekturwerte) gefüllt ist. Redundante Bitkombinationen für die abgetrennten Bits von S und C sind hier übereinander eingetragen. Bei manchen Hardwarearchitekturen lassen sich solche Lookutables nicht realisieren und es müssten die vollen 32xn Zellen mit einem Multiplexer mit 5 Adressbits verwendet werden.

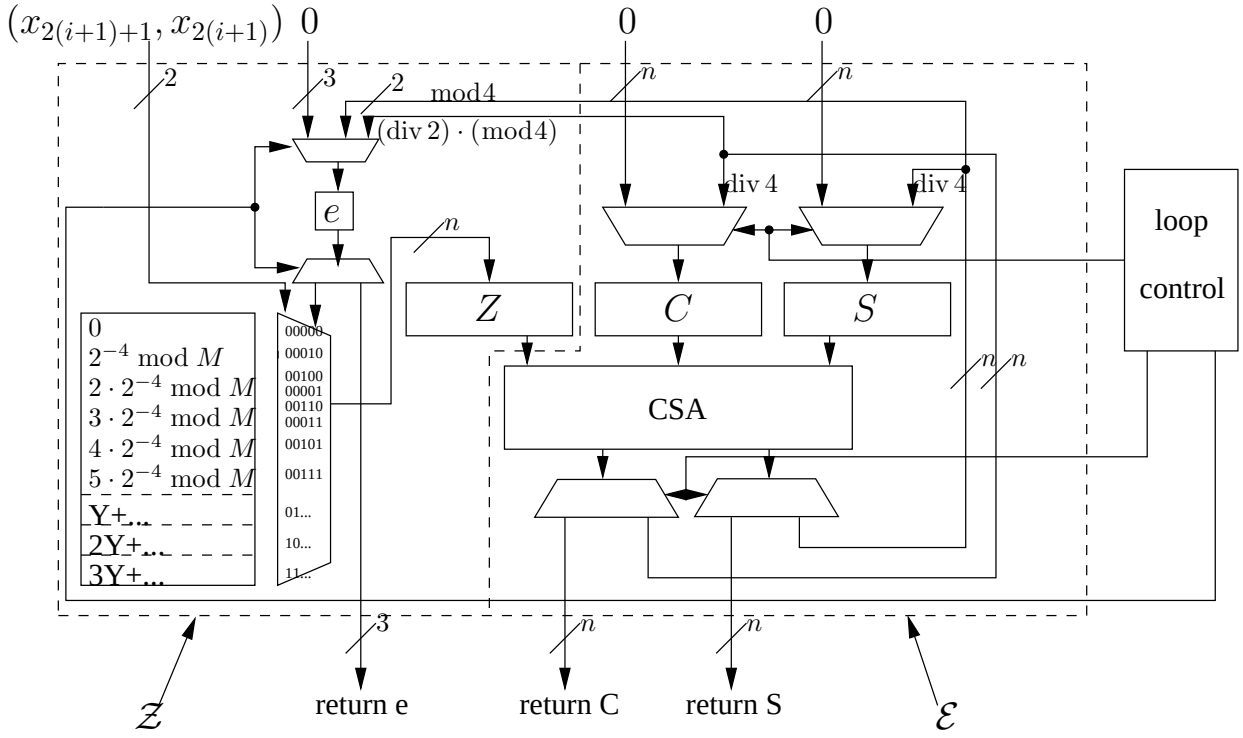


Abbildung 4.10: Entwurf 3 mit Carry-Save-Addierer und der Basis 4

Komplexitätsanalyse für die Basis 4:

Der Flächenverbrauch der Architektur ist also bei der Erweiterung auf die Basis 4 gewachsen: die Lookuptable benötigt nun 24xn (anstatt zuvor 4xn) Speicherzellen. Ansonsten bleibt es bei drei Registern und einem Addierer. Es folgt ein Bedarf an $5,5n$ Flächeneinheiten.

Die Zeitkomplexität sinkt dafür auf die Hälfte und beträgt nurnoch $1/2n$ Zeiteinheiten.

Es folgt eine AT-Komplexität von $2,75n^2$.

4.8.1 Beweisskizze zur Korrektheit der zweistufigen Methode für die Basis 4

Wir werden davon absehen, ein weiteres mal die Korrektheit des Algorithmus' nachzuweisen. Dies wäre lediglich eine Analogie zum vorherigen Beweis bzw. zum Beweis der zweistufigen Methode und ihrer Erweiterung auf allgemeine Zahlenbasen in den vorangegangenen Kapiteln.

Eine kurze Skizze werden wir dennoch betrachten:

- Es gibt nurnoch $n/2$ Schleifendurchläufe.
- Jeden Schleifendurchlauf werden die Register um 2 Bits verkürzt - im i-ten Durchlauf entspricht das einem Faktor von 2^{-2i}.
- Die Koeffizienten von X bestehen aus Paaren von Bits, die mit Y multipliziert werden und zu dem Korrekturwert addiert werden. Der Korrekturwert wird nun mit dem Faktor 2^{-4} versehen, da die Register um 4 Bits geschoben werden, bevor abgetrennte Bits wieder einfließen.

Kapitel 5

Abschließende Betrachtungen

5.1 Spezielle Betrachtungen in Hinsicht auf die Realisierung

Auf einige Fragestellungen hinsichtlich der Implementierung wurde in den vorangegenagenen Kapiteln noch gar nicht eingegangen. Dies soll in diesem Kapitel umfassend für alle Entwürfe und Algorithmen geschehen.

5.2 Füllen der Lookuptables

Zuvor haben wir die Lookuptables in sämtlichen Entwürfen so angegebeben, wie es die entsprechnende Methode erforderte. Wir haben auch festgestellt, dass das Füllen der LUTs nicht in die Zeitkompexität mit einzurechnen ist, da es nur einmal vorgenommen werden muss und dann lange vorhält, da sich der Modulus in einer Anwendung meist für viele Modularmultiplikationen nicht ändert, sondern nur die Eingabewerte X und X. Dennoch muss dies ja geschehen und dazu sollen hier einige Anmerkungen gemacht werden.

Es ist üblich, spezielle Hardware, die nur für einen Zweck - etwa die Modularmultiplikation - entwickelt ist, in ein System einzubinden, das es mit Eingabewerten versorgt und entsprechende Nachbearbeitungen der Ergebnisse vornimmt. Dies können wieder spezielle Controller oder aber ein Standardrechner sein. Deren Aufgabe ist es, auch die Lookuptables zu füllen. Dabei gilt es folgendes zu beachten:

1. Für die Basis 2 sind alle Einträge, die vom Eingabewert Y abhängen, simple Summen, die also nur den Eingabewert selbst oder eine Addition eines Korrekturwertes beihalten.

2. Für höhere Zahlenbasen sind auch Vielfache von Y und Summen, die solche beinhalten, nötig. Dies benötigt eine gernige Zahl an einfachen Rechenoperationen, bevor die Lookuptable beschrieben werden kann.

3. Die Korrekturwerte entsprechen entweder Vielfachen des Modulus' M oder multiplikativen Inversen von 2er-Potenzen modulo M. Die Vielfachen von M können mit wenigen Operationen bestimmt werden und im Speicher der Architektur oder eines Host-Systems abgelegt werden, da sie erneut verwendet werden können. Die inversen 2er-Potenzen können ebenfalls schnell bestimmt und gespeichert werden: Die Bildung des multiplikativen Inversen von $2^k, k \in \mathbb{N}$ entspricht dem "Montgomery-Produkt"

   $$\underbrace{(00...01)}_{k} \overset{MMM}{\cdot} 1 = 1 \cdot 1 \cdot 2^{-k} \bmod M = 2^{-k} \bmod M.$$

   Dies kann durch eine übergeordnete Einheit (etwa einen Standardrechner) schnell bestimmt werden, da die Höhe der Potenzen gering ausfällt.

Sämtliche Einträge, die nur vom Modulus abhängen, können also im Vorfeld berechnet und gespeichert werden. Einträge, die von den häufiger veränderten Eingabewerten abhängen, benötigen nur einfache Additionen. Diese können auch auf kleinen Systemen wie einer Smartcard durchgeführt werden, während die kompliziertere Bestimmung von multiplikativen Inversen auf externen Systemen durchgeführt werden und auf der Hardware gespeichert werden kann.

5.3 Nachbearbeitung

Ein weiterer, in vielen Entwürfen ungeklärter Vorgang ist die Nachbearbeitung der Daten. Denn häufig ist das gesuchte Ergebnis $X \cdot Y \cdot 2^{-n} \bmod M$ nicht als Endergbenis in einem Register zu finden, sondern ist auf diverse Register verteilt und es benötigt noch einiger Operationen, bis es wirklich weiterverarbeitet werden kann.

Hierfür gibt es zwei Lösungsmöglichkeiten: entweder implementiert man die Nachbearbeitung außerhalb der eigentlichen Architektur oder bezieht sie in den Algorithmus mit ein. Durch den ersten Ansatz werden einige zusätzliche Operationen nötig, die aber in allen relevanten Algorithmen aus wenigen Additionen (Aufaddieren der verschiedenen Ergebnisregister) und Subtraktionen (Reduktion modulo M) bestehen. Manche Methoden benötigen auch

die Multiplikation eines Paramters mit dem multiplikativen Inversen einer Zweierpotenz (modulo M). Beispielsweise der Entwurf aus Kapitel 4.8 (die einstufige Methode für die Basis 4 - der effizienteste Entwurf dieser Arbeit) muss das Register e mit 2^{-2} mod M verrechnen, wobei e selbst aus 3 Bits besteht (zwei Bits des Summen- und ein Bit des Carry-Registers). Die Summe der in e gespeicherten Bits kann dabei Werte zwischen 0 und 5 annehmen. Die gesuchte Operation besteht also lediglich in der Multiplikation von Werten der Menge $\{0, 1, 2, 3, 4, 5\}$ mit 2^{-2} mod M und hängt dabei nur vom Modulus M ab. Damit kann sie genau wie die Einträge in den Lookuptabellen im Voraus berechnet und für die Dauer vieler Modularmultiplikationen verwendet werden.

Der zweite Ansatz kann je nach Hardware-Architektur effizienter ausfallen. Wenn wir beim Beispiel der einstufigen Methode bleiben, so erkennt man, dass $2^{-2}(= 4 \cdot 2^{-4})$ bereits in der Lookuptable gespeichert ist (siehe Abbildung 4.10). Es lassen sich auch die Vielfachen dieses Wertes dort abspeichern. Wenn die Architektur ohnehin nur Lookuptabellen zuließe, deren Größe 2er-Potenzen entspricht, so könnte dies auch ohne zusätzlichen Platzverbrauch geschehen, da die Tabelle lediglich 24 Einträge hat und so noch freien Platz böte. Dann benötigt man aber in jedem Fall zusätzliche Multiplexer, die den Datenfluss abhängig von der Schleifeniteration steuern, so dass zum Beispiel keine Divisionen durch 4 mehr nach dem n-ten Schleifendurchlauf durchgeführt werden. Durch diese Implementierung lässt sich dann das Endergebnis bis auf die, durch die Verwendung von Carry-Save-Addierern entstehende, Redundanz genau bestimmen.

Die gewählte Art der Nachbearbeitung ist also von der Hardware-Architektur abhängig zu machen, um ein Flächen-Zeit-effizientes Ergebnis zu erzielen. In jedem Fall spielt die Nachbearbeitung im Vergleich zur eigentlichen Modularmultiplikation eine subdominante Rolle (wobei in den meisten Fällen eine Implementierung außerhalb der Architektur vorzuziehen sein wird) und wid deshalb nicht in die Flächen-Zeit-Analyse der Architekturen mit aufgeführt.

5.4 Bewertung

Dieses Kapitel wird alle gewonnen Ergebnisse der vorangegangenen Entwicklungen und Untersuchungen zusammenfassen und gegenüberstellen.

In Kapitel 3 haben wir die bestehenden Methoden kennengelernt und die effizientesten auf ihre Komplexität hin untersucht.

In Kapitel 4 haben wir eine neue Methode in zwei Ansätzen entwickelt, und beide Ansätze auf ihre Effizienz hin optimiert.

Folgende Tabelle fasst die daraus erzielten Ergebnisse zusammen:

Tabelle: Ergebnisse für die AT-Komplexitäten der in Kapitel 3 und 4 betrachteten Methoden

	k=1	k=2	k=3	k=4
Montgomery 1	$7n^2$	$9,75n^2$	$10n^2$	$17,25n^2$
Montgomery 2	$6n^2$	$4,5n^2$	$(7n^2)$	$(17,25n^2)$
Zweistufiger Ansatz	$5,625n^2$	$3,125n^2$	$(2,583n^2)$	$(2,688n^2)$
Einstufiger Ansatz	$3,5n^2$	$2,75n^2$	-	-

Der Algorithmus "Montgomery 1" bezieht sich dabei auf die in Kapitel 3.5 vergstellten Methode, die dem klassischen Verfahren von Montgomery entspricht. Diese wurde durch die Verwendung von Carry-Save-Addierern und die Verwendung höherer Zahlenbasen optimiert und erzielt die größte Effizienz für die Zahlenbasis 2 mit $7n^2$ Flächen-Zeit-Komplexität.

"Montgomery 2" bezieht sich auf den ebenfalls in Kapitel 3.5 besprochenen Algorithmus, der die ursprüngliche Methode um die Verwendung einer Lookuptable erweitert und so die zeitliche Effizienz erhöht. Dieser Algorithmus wurde ebenfalls mit Carry-Save-Addierern und höheren Zahlenbasen präsentiert und erzielt das beste Ergebnis für die Basis 4 mit $4,5n^2$. Diese Methode ist also als Maßstab für neu entwickelte Algorithmen zu sehen.

Die Ergebnisse für den in Kapitel 4.4 entwickelten Ansatz sind der Tabelle in der Zeile "Zweistufiger Ansatz" zu entnehmen. Die entwickelten Methoden wurden mittels Carry-Save-Addierern, Lookuptabellen und höhere Zahlenbasen optimiert und haben für die Basis 4 einen Wert von $3,125n^2$ erzielt. Der eigentlich beste Wert in obiger Tabelle, wird jedoch von der zweistufigen Mehtode für die Basis 8 erzielt. Dieser Entwurf ist aus Gründen, die in Kapitel 4.6.1 bereits erläutert wurden, aber nicht realisierbar.

Der in Kapitel 4.3 verfolgte Ansatz hat sich als am effizientesten herausgestellt. Die Ergebnisse sind der Zeile "Einstufiger Ansatz" zu entnehmen. Auch dieser Ansatz wurd durch die Verwendung von Carry-Save-Addierern, Lookuptabellen und höheren Zahlenbasen optimiert. Dieser Entwurf besticht durch seine Flächeneffizienz (da er nur eine Addiererstufe benötigt) und durch die zeitliche Effizienz (da der Entwurf vollständig gepipelined ist). Das beste Ergebnis für die einstufige Methode wird für die Basis 4 erzielt und beträgt $2,75n^2$ Flächen- mal Zeiteinheiten.

5.5 Zusammenfassung

In dieser Arbeit wurde eine neue Methode für die modulare Multiplikation zweier Eingabewerte $X \cdot Y$ mod M präsentiert. Sie orientiert sich in der Vorgehensweise am Montgomery-Algorithmus zur Modularmultiplikation, der das

derzeit effizienteste Verfahren auf diesem Gebiet ist. Durch eine von diesem Algorithmus verschiedenen Idee für die modulare Reduktion ist eine effizientere Implementierung entstanden, die besonders für die Verwendung höherer Zahlenbasen geeignet ist: durch die Verwendung von Lookuptabellen wird die Division einzelner Bits duch 2 (oder 4, 8 bei höheren Zahlenbasen) vorausberechnet und die Reduktion in konstanter Zeit durchgeführt. Aufgrund dieser Vorgehensweise lassen sich die Entwürfe vollständig pipelinen, was zu einer sehr guten, zeitlichen Effizienz führt. Der neue Ansatz wurde in zwei Varianten verfolgt und durch viele Verbesserungen in seiner Effizienz optimert. Der erste Ansatz führt die Mutliplikation und Reduktion in einem Schritt durch und akkumuliert das Ergebnis in einem Registerpaar - der zweite verwendet hierfür zwei Stufen, die parallel zueinander Multiplikation und Reduktion durchführen. Der erste Ansatz hat sich dabei, aufgrund von Implementierungsschwierigkeiten des zweiten, als effizienter erwiesen. Die Effizienz dieser neuen Methode ist um einen Faktor von $1,6$ in der Flächen-Zeit-Komplexität besser als die der besten Implementierung des Montgomery-Algorithmus'.

5.6 Gegenstand weiterer Forschung

Als Ansatzpunkt für weiterführende Arbeiten sei hier die Optimierung der in Kapitel 4.4 entwickelten, zweistufigen Methode genannt. Denn wie wir in Kapitel 4.6.1 festgestellt haben, eignet sich diese Methode besonders für die Implementierung der Modularmultiplikation auf höheren Zahlenbasen. Der theoretisch erzielte Wert dieses Algorithmus' übertrifft das Ergebnis der in Kapitel 5.4 optimalen Methode an Effizienz. Hier gilt es also die kritische Operation des Entwurfs - die Addition der abgetrennten Bits - zu beschleunigen, damit die Basis 8 realisierbar wird. Dies könnte durch spezielle Logik oder weitere Pipelinestufen geschehen. Auch grundlegende Änderungen des Konzepts sind denkbar, wie etwa die Änderung der Zahlenrepräsentation an sich: durch die Darstellung der Eingabewerte in einer anderen Repräsentation als des Zweier-, Vierer- oder Achtersytems, sondern zum Besipiel der Fibonacci-Zahlen könnte sich eine Steigerung der Effizienz für diverse modulare Multiplikationsalgorithmen erzielen.

Literaturverzeichnis

[1] W. Diffie and M. E. Hellman, "New directions in cryptography", IEEE Trans. Inform. Theory, IT-22: 644-654, November 1976

[2] R. L. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Didital Signatures and Public Key Cryptosystems", Communications of the Association for Compunting Machinery, 21(2): 120-126, February 1978

[3] A. J. Menezes, "Elliptic Kurve Public Key Cryptosystems", Boston: Kluwer Academic Publishers, 1993

[4] Johannes Buchmann, "Einführung in die Kryptographie", 2. Auflage, Springer-Verlag, 2003, ISBN 3-540-41283-2, S. 129-133

[5] Çetin Kaya Koç, "RSA Hardware Implementation", RSA Labratories, August 1995

[6] A. J. Menezes, P. C. Oorschot and S. A. Vanstone, "Handbook of Applied Cryptography", CRC Press, ISBN 0-8493-8523-7, 1996

[7] G. R. Blakley, "A Computer Algorithm for Calculating the Product AB Modulo M", IEEE Transactions on Computers, 32(5): 497-500, May 1983

[8] N. Takagi and S. Yajima, "Modular Multiplication Hardware Algorithms with a Redundant Representation and Their Application to RSA Cryptosystem", IEEE Transactions on Computers, 41(7): 887-891, July 1992

[9] K. Cho, J. Ryu and J. Cho, "High-Speed Modular Multiplication Algorithm for RSA Cryptosystem", KOSEF 2000-2-30300-004-3, 2000

[10] V. Bunimov and M. Schimmler, "Area and Time Efficient Modular Multiplication of Large Integers", IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'03), June 2003

[11] V. Bunimov, M. Schimmler, "High Radix Modular Multiplication of Large Integers Optimised with Respect to Area and Time", 2004 International Conference on VLSI, June 2004

[12] P. Barrett, "Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor", CRYPTO '86 Proceeding, vol. 263 of Lecture Notes in Computer Science, 311-323, Springer-Verlag, 1987

[13] Johann Großschädl, "High-Speed RSA Hardware Based on Barrett's Modular Reduction Method", CHES2000, LNCS 1965: 191-203, 2000

[14] N. Nedjah and L. M. Mourelle, "Fast Hardware of Booth-Barrett's Modular Muliplication for Efficient Cryptosystems", ISCIS 2003, LNCS 2896: 27-34, 2003

[15] P. L. Montgomery, "Modular Multiplication without Trial Division", Mathematics of Computation, 44(170): 519-521, April 1985

[16] T. Blum, C. Paar, "High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware", IEEE Transactions on Computers, 50(7): 759-764. 2001

[17] V. Bunimov, M. Schimmler, and B. Tolg, "A Complexity-Effective Version of Montgomery's Algorithm", Workshop on Complexity Effective Designs (WCED02), May 2002

[18] A. F. Temca and Ç. K. Koç, "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm", IEEE Transactions on Computers, 52(9):1215-1221, September 2003

[19] Y. S. Kim, W. S. Kang and J. R. Choi, "Implementation of 1024-Bit Moular Processor for RSA Cryptosystem", School of Electronic an electrical Engineering, Kyungpook National University

[20] S. B. Örs, L. Batina, B. Preneel and J. Vandewalle, "Hardware Implementation of a Montgomery Modular Multiplier in a Systolic Array", Katholieke Universteit Keuven, ESAT/SCD-COSIC

[21] S. Even, "Systolic Modular Multiplication", Computer Science Dept., Technion, Haifa, Irael

[22] A. Cilardo, A. Mazzeo, L. Romano and G. P. Saggese, "Carry-Save Montgomery Modular Exponentiation on Reconfigurable Hardware, Università degli Studi di Napoli Federico II, Italy

[23] C. D. Walter, "Still Faster Modular Multiplication", Computation Dept., UMIST, Manchester, U.K.

[24] Victor Bunimov "Entwicklung von neuen Algorithmen der Computerarithmetik in Hinsicht auf ihre Nutzung in der Kryptographie", Technische Fakultät der Christian-Albrechts-Universität zu Kiel, Dissertation, 2005

[25] W. P. de Roever et al., "Concurrency Verification", Cambridge Tracts in Theoretical Computer Science 54, ISBN 0-521-80608-9, 2001