Konzeption und Implementation eines Hardwareakzelerators zum RSA-Verfahren

Diplomarbeit
Universität Hamburg - Fachbereich Informatik
Arbeitsbereich Technische Grundlagen der Informatik

Lars H. Hahn
Schlehdornweg 12, 22335 Hamburg
Telefon (040) 59 59 34, Telefax (040) 500 46 22

Juli 1997

Inhaltsverzeichnis

1 Einleitung

Kryptographische Verfahren haben grundlegende Bedeutung, um die Anforderung an die Datensicherheit in informationstechnischen Systemen zu erfüllen. Dabei geht es um Geheimhaltung, Fälschungssicherheit und Authentizität. Immer wichtiger wird in Zeiten rapide wachsender digitaler Datennetze die Verschlüsselung der versandten Daten in Echtzeit. Maßstäbe hierfür setzen die Übertragungsraten von 9,6 kBit/s einer Chipkarte, von 64 kBit/s im ISDN-Netz der Telekom und von 8 MBit/s für das digitale Fernsehen.

Sowohl Anforderungen an die Geschwindigkeit als auch Aspekte der Sicherheit legen es nahe, kryptographische Algorithmen in Hardware zu implementieren. Dabei reichen oft weder Realisierungen von „optimierten“ Algorithmen in standardmäßiger Hardware noch Realisierungen von Standardalgorithmen in „optimierter“ Hardware aus. Der Algorithmenentwurf in der Kryptographie spannt somit den gesamten Bogen von theoretischen Untersuchungen in der Mathematik bis hin zum Full-Custom-Entwurf in VLSI[1].

Ein wichtiges kryptographisches Verfahren ist das RSA-Verfahren. Es ist weltweit sehr stark verbreitet und erfüllt in besonderer Weise die heutigen Sicherheitsstandards. Das RSA-Verfahren, das nach seinen Erfindern Ron Rivest, Adi Shamir und Leonard Adleman benannt ist, überstand jahrelange gründliche Kryptoanalyse. Die Kryptoanalyse konnte die Sicherheit von RSA zwar weder beweisen noch widerlegen, aber sie stärkt das Vertrauen in den Algorithmus [SCH96].

Ziel dieser Diplomarbeit ist es, einen Chip zu entwickeln, der in der Lage ist, Daten mindestens mit einer Rate von 9,6 kBit/s mit Hilfe des RSA-Verfahrens zu verschlüsseln und wieder zu entschlüsseln. Bei einer Verschlüsselungsrate von 64 kBit/s wäre ein ISDN-Telefon eine denkbare Anwendung für diesen Chip, um so das Abhören eines Telefonates oder einer Datenübertragung unmöglich zu machen. Weitere mögliche Einsatzgebiete sind „low-security“-Systeme, in denen auf „elektronische“ Art kleine Beträge bezahlt werden (electronic cash).

Die Aufgabenstellung dieser Diplomarbeit wurde von der Firma Philips Semiconductors Hamburg im Rahmen eines Kooperationsprojektes mit dem Fachbereich Informatik (Arbeitsbereich TECH) der Universität Hamburg gestellt.

1.1 Motivation

Speziell im Bereich der Kryptographie gilt es, große Datenmengen in kurzer Zeit zu verarbeiten. Oftmals wird in der Praxis eine Verarbeitung in Echtzeit gefordert. Deshalb ist es im besonderen Maß notwendig, auf dem Rechner erprobte Verfahren in Hardware zu implementieren. In vielen Fällen wird eine solche Hardware den Rechner nicht ersetzen,

[1] Very Large Scale Integration (VLSI)

sondern unterstützen. Aber auch Einsatzgebiete, in denen kein Rechner zur Verfügung steht, sind nicht nur denkbar, sondern ohne die Realisierung in eigenständiger Hardware überhaupt nicht zu verwirklichen. Ein Beispiel hierfür stellt das bereits erwähnte ISDN-Telefon dar.

Die direkte Umsetzung eines Computerprogramms in Hardware läßt bei serieller Abarbeitung von Operationen nach bisherigen Erfahrungen eine Beschleunigung um etwa den Faktor zehn zu, da die Interpretation von Prozessorbefehlen entfällt. Eine weitere Beschleunigung erreicht man, wenn die Hardware mehrere für einen Mikroprozessor nacheinander auszuführende Aufgaben parallel ausführen kann [DWE95].

Eine Spezialhardware hat nicht nur eine höhere Leistungsfähigkeit als eine Problemlösung in Software, sondern ermöglicht auch einen kompakteren Aufbau und eine geringere Leistungsaufnahme als ein System, das mit einem genügend leistungsfähigen Universalrechner realisiert wurde. Ein gutes Beispiel für Systeme, bei denen Leistungsaufnahme, Größe und Gewicht kritisch sind, stellen autonome Systeme dar. Ein höherer Energieverbrauch verringert die Betriebsdauer oder erhöht das Gewicht der Energieversorgung und damit die Größe des gesamten Systems.

Ein weiteres Argument für die Realisierung in Hardware ist die größere Sicherheit gegenüber Angriffen. Werden die zur Entschlüsselung benötigten Schlüssel direkt auf dem Chip abgelegt, ist es unmöglich, diese zu erhalten, selbst wenn man Zugriff auf den Chip erlangt.

Die genauen Anforderungen an den zu realisierenden Chip sind im Abschnitt „2.3 Vorgaben“ auf Seite 25 zusammengestellt.

1.2 Kryptographie

Die Kryptographie ist die Wissenschaft von den Methoden der Verschlüsselung (Chiffrierung) und Entschlüsselung (Dechiffrierung) von Daten.

Kryptographische Verfahren

- dienen zur Geheimhaltung von übertragenen oder gespeicherten Informationen gegenüber Dritten,
- erzielen die Geheimhaltung durch Transformation der Zeichen einer Nachricht, so daß die transformierte Nachricht für Dritte keine konstruierbare semantische, statistische oder strukturelle Korrelation zum Original mehr aufweist,
- bestehen aus umkehrbaren Kryptofunktionen (zur Chiffrierung/Dechiffrierung) und Mengen von Schlüsseln, die diese Funktionen parametrisieren. Nur die Kenntnis eines Schlüssels erlaubt insbesondere die relativ schnelle Entschlüsselung einer Nachricht [HOR85].

Geheimhaltung bedeutet, daß nur der oder die gewünschten Empfänger den Inhalt einer Nachricht entschlüsseln können. Neben der Geheimhaltung soll die Kryptographie oft noch andere Ansprüche erfüllen:

- Authentifizierung: Es sollte dem Empfänger möglich sein, die Herkunft einer Nachricht zu ermitteln. Ein Eindringling sollte sich nicht als eine andere Person ausgeben können.
- Integrität: Der Empfänger sollte überprüfen können, ob eine Nachricht bei der Übertragung verändert wurde. Ein Eindringling sollte die echte nicht durch eine gefälschte Nachricht ersetzten können.
- Verbindlichkeit: Ein Sender sollte später nicht leugnen können, daß er eine Nachricht gesendet hat [SCH96].

Kryptographische Verfahren sind von jeher zumeist unter dem Begriff „Geheimschrift" zur Übermittlung von vertraulichen Informationen verwendet worden, wobei nach einer in einem Codebuch festgehaltenen Vorschrift Zeichen, Wörter oder ganze Sätze ausgetauscht wurden. Angewendet wurden kryptographische Verfahren zunächst vor allem beim Austausch von Nachrichten zwischen Staaten im militärischen und diplomatischen Bereich. In der heutigen Zeit finden kryptographische Verfahren auch im zivilen und dort insbesondere im kaufmännischen Bereich Anwendung. Wie bei Verträgen zwischen Staaten wurden und werden größere Kaufverträge mit der unnachahmlichen Unterschrift und mit einem Siegel versehen, das die Echtheit des Dokuments durch die besondere Gestaltung des Siegels und damit seine Fälschungssicherheit gewährleistete. Die Forderung nach Unkenntlichmachung von Informationen, die Notwendigkeit, die Authentizität von Kommunikationspartnern und die Integrität von Nachrichten zu gewährleisten und die Besiegelung von Dokumenten und Verträgen zur Verleihung rechtlicher Verbindlichkeit sind von jeher Probleme, die bei der Kommunikation von Staaten, Gruppen und Einzelpersonen auftreten [RYS80].

1.2.1 Geschichte der Kryptographie

Das Bedürfnis der Menschen, Nachrichten so darzustellen, daß sie ausschließlich für den Schreiber und einen bestimmten Leser verständlich sind, ist wohl so alt wie die (schriftliche) Kommunikation überhaupt [HOR85].

Eine der ältesten überlieferten Formen eines kryptographischen Verfahren ist die Skytala der Spartaner aus dem Jahr 500 v. Chr. Es handelt sich dabei um einen Stab, um den erst ein Papierstreifen spiralförmig gewickelt und dann dieser parallel zum Stab beschrieben wurde. Der Text auf dem abgewickelten Papierstreifen konnte erst wieder gelesen werden, nachdem er um einen Stab gleichen Durchmessers gewickelt worden war [SAU93].

Diese einfache Transposition und auch eine einfache Substitution, wie sie Julius Cäsar nachweislich benutzte, waren aus heutiger Sicht keine ernst zu nehmenden kryptographischen Verfahren [HOR85].

Klartext:	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Geheimtext:	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c

Abbildung 1.1: „Chiffrierung nach Cäsar"

Die von Cäsar benutzte Chiffre erhält man, wenn man unter das Klartextalphabet das Geheimtextalphabet schreibt - aber um drei Stellen versetzt. Chiffriert wird, indem ein

Klartextbuchstabe durch den darunter stehenden Geheimtextbuchstaben ersetzt wird [BEU87].

Das erste erhaltene Buch über Kryptographie stammt aus dem Jahre 1466 von dem italienischen Humanisten, Künstler und Gelehrten Leon Battista Alberti (1404 - 1472). Alberti führte die sogenannte polyalphabetische Substitution[2] ein. Die Bemühungen, sowohl sichere Verfahren zur Chiffrierung zu finden als auch diese wieder zu brechen, nahmen in der Folgezeit stark zu [SAU93].

Neue Impulse brachte der Kryptographie erst wieder das neunzehnte Jahrhundert. Charles Wheatstone führte 1854 seine Bigrammverschlüsselung ein, die wegen ihrer Einfachheit „sogar Diplomaten zugemutet werden konnte". Große Erfolge im Brechen von Chiffren erzielten im Jahre 1863 Friedrich Kasiski und im Jahre 1883 Auguste Kerckhoffs [HOR85]. Während zu Beginn des zwanzigsten Jahrhunderts mechanische und elektromagnetische Verschlüsselungsgeräte entwickelt und eingesetzt wurden, gestattete das Aufkommen programmgesteuerter elektronischer Rechenanlagen die Realisierung immer leistungsfähigerer kryptographischer Verfahren [SAU93].

Mit der allgemeinen Verbreitung der elektronischen Datenverarbeitung in den sechziger Jahren erlebte die Entwicklung der Kryptographie sozusagen einen Quantensprung. Die moderne Hard- und Software ermöglichte die Implementierung von komplexen und anspruchsvollen mathematischen Algorithmen, mit denen ein Grad an Sicherheit erreicht wurde, wie er bis dahin ohne Beispiel war. Was noch hinzu kam, war, daß diese neue Technik für jedermann verfügbar wurde, während bis dahin die Kryptographie eine Geheimwissenschaft für das Militär und die Geheimdienste war [RAN95]. In der heutigen Zeit erhalten kryptographische Applikationen immer größere Bedeutung im zivilen Bereich. Die Anwendungen reichen von Aufgaben mit kurzen Schlüssellängen (low security), wie z.B. zur Sicherung von Funksignalen, bis zu sehr sicherheitskritischen Problemen (high security) im Bereich der Geldtransaktionen von Banken.

Die erste Standardisierung eines kryptographischen Verfahrens erfolgte im Jahre 1977 mit der Normierung des Data Encryption Standard (DES) durch das amerikanische National Bureau of Standards (NBS). Das American National Standards Institute (ANSI) erkannte DES 1981 als Standard für den privaten Sektor an.

Die bisher letzte große Neuerung in der Kryptographie ist die Veröffentlichung des Konzepts der asymmetrischen kryptographischen Verfahren (Public Key Kryptographie) durch Diffie und Hellmann im Jahre 1976 [DIF76]. Ihre Verfahren machten es überflüssig, daß potentielle Kommunikationspartner vor der Nachrichtenübermittlung zunächst untereinander ihren speziellen Schlüssel austauschen [HOR85]. Das Neue an diesem Verfahren ist, daß eine sichere Kommunikation gewährleistet wird, obwohl Schlüssel und Schlüsselfunktion des Übermittlers öffentlich bekannt sind. Dieses ist möglich, weil die Schlüssel zur Ver- und Entschlüsselung nicht nur verschieden sind, sondern es ist sogar „unmöglich", den einen aus dem anderen ohne zusätzliche Informationen zu berechnen.

---

[2] Eine Substitution heißt monoalphabetisch, falls jeder Buchstabe des Alphabet stets zu demselben Buchstaben chiffriert wird. Im Gegensatz dazu bezeichnet man eine Chiffrierung als polyalphabetisch, wenn derselbe Klartextbuchstabe nicht immer mit demselben Geheimtextbuchstaben verschlüsselt wird.

Seit diesem Zeitpunkt ist eine geänderte Entwicklungsrichtung der Kryptographie zu verzeichnen. Dienten kryptographische Verfahren früher im wesentlichen zur vertraulichen Übermittlung und Aufbewahrung von militärischen und diplomatischen Geheimnissen, so ist heute die Erforschung, Entwicklung, Implementierung und der Einsatz kryptographischer Verfahren für alle Bereiche der Informationstechnik von großer Bedeutung. Dabei liegt der Schwerpunkt nicht nur im Bereich der Geheimhaltung, sondern auch im Bereich der Integritätssicherung und der Authentifizierung von Daten. Man geht heute nicht mehr nur von einem passiven Angriff aus, bei dem der Angreifer nur die Möglichkeit hat, Daten abzuhören, sondern die Abwehr eines aktiven Angriffs wird immer wichtiger. Man nimmt also an, daß ein Angreifer nicht nur Daten lesen, sondern auch Daten verändern, ergänzen oder zerstören möchte. Beispielsweise wäre es zwar unangenehm, wenn ein bestimmter Kontostand auf einem elektronisch geführten Bankkonto Unbefugten bekannt würde. Sehr viel schlimmer wäre es allerdings, wenn der Kontostand unbefugt modifiziert würde.

1.3 Kryptographische Verfahren

Moderne kryptographische Algorithmen basieren in der Regel auf dem Kerckhoffs-Prinzip. Das nach Auguste Kerckhoffs (1835 - 1903) benannte Prinzip besagt, daß die gesamte Sicherheit eines Algorithmus nur auf der Geheimhaltung des Schlüssels beruhen soll und nicht auf der Geheimhaltung des kryptographischen Verfahrens. Die Auswirkung dieses allgemein anerkannten, aber oft mißachteten, Prinzips ist, daß viele im zivilen Bereich eingesetzte Kryptoalgorithmen veröffentlicht wurden und teilweise auch genormt sind [RAN95].

Kryptoalgorithmen teilt man in symmetrische und asymmetrische Algorithmen ein. Diese Einteilung bezieht sich auf die verwendeten Schlüssel. Symmetrisch bedeutet, daß der Algorithmus für Ver- und Entschlüsselung den gleichen Schlüssel benutzt bzw. daß der eine Schlüssel leicht aus dem anderen zu berechnen ist. Dies wiederum bedeutet, daß jeder, der eine Nachricht lesen kann, ebenso eine Nachricht verschlüsseln kann. Im Gegensatz dazu benötigen die im Jahre 1976 von Whitfield Diffie und Martin E. Hellman [DIF76] postulierten asymmetrischen Kryptoalgorithmen für Ver- und Entschlüsselung je einen unterschiedlichen Schlüssel.

1.3.1 Symmetrische kryptographische Verfahren

Die traditionellen, seit vielen Jahren bekannten Verfahren sind die symmetrischen kryptographischen Verfahren. Hierbei werden Nachrichten durch Anwendung eines geheimen Schlüssels (secret key), der nur dem Sender und dem Empfänger der Nachricht bekannt ist, chiffriert und beim Empfänger mit dem gleichen Schlüssel wieder dechiffriert. Bei symmetrischen Verfahren ist also der Schlüssel s zum Ver- und Entschlüsseln gleich. Dies bringt Probleme für den Schlüsselaustausch mit sich. Bevor Nachrichten vertraulich über einen potentiell unsicheren Nachrichtenkanal ausgetauscht werden können, muß zunächst auf einem sicheren Weg der Schlüssel s ausgetauscht werden [SAU93].

Von symmetrischen Verfahren wird verlangt, daß man

- mit dem korrekten Schlüssel die Nachricht einfach aus dem Chiffrat zurückgewinnen kann,
- aus dem Chiffrat weder Nachricht noch Schlüssel erhalten kann,
- aus der Nachricht und dem zugehörigem Chiffrat nicht den Schlüssel gewinnen kann [GOL94].

Nachteile der symmetrischen Verfahren sind:

1. Es ist nicht möglich, Dritten gegenüber nachzuweisen, daß ein bestimmter Absender eine bestimmte Meldung geschickt hat.
2. Die Schlüssel müssen vorher über einen hinreichend gesicherten Kanal vereinbart werden.
3. Ein weiteres Problem kann in der Anzahl der benötigten Schlüssel liegen. Bei 100 Partnern in einem Netz, wobei jeder mit jedem gesichert Daten austauschen möchte, sind bereits 4950 Schlüssel erforderlich. Bei 1000 Teilnehmern sind es etwa 500.000 [Schlüsselanzahl = TN · (TN - 1) · ½].
4. Es ist aufwendig, neue Teilnehmer in ein bestehendes System einzufügen.

Das wohl bekannteste symmetrische Verfahren ist DES[3]. Der Algorithmus wurde von IBM zusammen mit dem NBS[4] entwickelt [RAN95]. Bei DES unterscheiden sich Chiffrier- und Dechiffrierschritt nur in der Reihenfolge, in der die aus dem Schlüssel generierten Teilchiffrierungen angewandt werden. Solche symmetrischen Verfahren können heute durch sehr effiziente Algorithmen maschinell realisiert werden. Insbesondere durch Hardwarelösungen in Form spezieller Chips sind Durchsatzraten von mittlerweile mehr als 20 MBit/s erreichbar [BAU93].

DES ist eine aus Substitutionen und Permutationen aufgebaute symmetrische Blockchiffre, die zur Klasse der „Feistel"-Chiffren[5] gehört [FUM88]. Der DES operiert in Abhängigkeit von 64-Bit Schlüsseln auf 64-Bit Blöcken. Die Bits 8, 16, 24, ... , 64 des externen Schlüssels können als Paritätsbits dienen, so daß die tatsächliche wirksame Schlüssellänge nur 56 Bit beträgt. Der Schlüsselraum von DES besitzt somit die Mächtigkeit 2^{56} (ca. $7 \cdot 10^{16}$). Eine ausführliche Beschreibung des DES findet sich z. B. in [FUM88] auf Seite 216 ff.

Zwei wichtige Prinzipien für einen guten Algorithmus zur Verschlüsselung gingen in das Design von DES ein: „Konfusion" und „Diffusion" nach C. Shannon. Das Prinzip „Konfusion" sagt aus, daß die Statistik des Schlüsseltextes die Statistik des Klartextes in einer solchen komplexen Weise beeinflussen soll, daß ein Angreifer daraus keine Vorteile erlangen kann. Das zweite Prinzip, die „Diffusion", besagt, daß jedes Bit des Klartextes und des Schlüssels möglichst viele Bits des Schlüsseltextes beeinflussen soll [RAN95].

---

[3] Data Encryption Standard (DES)
[4] US National Bureau of Standards (NBS)
[5] benannt nach Horst Feistel, er war bei IBM einer der Väter von DES

1.3.2 Asymmetrische kryptographische Verfahren

Im Jahr 1976 beschrieben Diffie und Hellman [DIF76] die Möglichkeit, einen Algorithmus zur Verschlüsselung zu entwickeln, der auf zwei unterschiedlichen Schlüsseln basiert. Ein Schlüssel sollte dabei öffentlich sein, der andere geheim. Diese Idee revolutionierte die bis dahin bekannte Vorstellung von Kryptographie.

Bei den asymmetrischen kryptographischen Verfahren werden zum Ver- und Entschlüsseln unterschiedliche Schlüssel verwendet. Bei diesem Verfahren besitzen also nicht je zwei Kommunikationspartner einen gemeinsamen geheimen Schlüssel s, sondern jeder Teilnehmer hat ein Schlüsselpaar, bestehend aus einem Schlüssel zum Verschlüsseln e (encryption key) und einem zum Entschlüsseln d (decryption key). Einer der beiden Schlüssel (zum Beispiel der zum Verschlüsseln e) kann öffentlich bekannt gemacht werden, ohne die Sicherheit des Verfahrens zu beeinträchtigen, und wird daher auch öffentlicher Schlüssel genannt. Der andere Schlüssel (in diesem Fall der zum Entschlüsseln) muß von seinem Besitzer geheim gehalten werden und wird daher privater[6] Schlüssel genannt. Solche Systeme nennt man auch „Public Key“ Kryptoverfahren [SAU93].

Ihre Wurzel haben asymmetrische Verfahren in den sogenannten Einwegfunktionen (one-way ciphers) [GOL94] (siehe hierzu „Einwegfunktionen mit Falltür“, Seite 20).

Möchte der Sender dem Empfänger eine vertrauliche Nachricht n übermitteln, so sucht er sich den zum Empfänger gehörigen Schlüssel e aus einem öffentlichen Verzeichnis und sendet die mit dem öffentlichen Schlüssel verschlüsselte Nachricht an den Empfänger. Nur dieser kennt den passenden privaten Schlüssel d und kann die Nachricht wieder entschlüsseln.

Ein Algorithmus f ordnet unter dem öffentlichen Schlüssel e jedem Klartext n einen Geheimtext

$$c = f_e(n)$$

zu. Umgekehrt ordnet f unter dem privaten Schlüssel d jedem Geheimtext c einen Klartext

$$n' = f_d(c)$$

zu. Wichtig ist, daß bei jeder Entschlüsselung der korrekte Klartext reproduziert werden kann. Es muß gelten

$$n' = f_d(c) = f_d(f_e(n)) = n$$

für alle Klartexte n.

[6] Im Zusammenhang mit asymmetrischen kryptographischen Verfahren spricht man von privaten Schlüsseln, um sie von den geheimen Schlüsseln der symmetrischen Verfahren abzugrenzen.

Abbildung 1.2: „Nachrichtenaustausch mit einem ‘Public Key’“

Die Abbildung 1.2: „Nachrichtenaustausch mit einem ‘Public Key’“ zeigt die zweifachen Einsatzmöglichkeiten eines „Public Key“ Verfahren. Jeder beliebige Teilnehmer *B* kann mit dem öffentlichen Schlüssel *e* Teilnehmer *A* eine persönliche Nachricht schicken, weil nur dieser zur Entschlüsselung mit seinem privaten Schlüssel *d* in der Lage ist. In der Gegenrichtung kann der Teilnehmer *A* mit seinem privaten Schlüssel *d* Nachrichten unterschreiben. Die Entschlüsselung ist nur mit dem passenden öffentlich bekannten Schlüssel *e* möglich (siehe auch „1.3.4 Digitale Signatur (Authentifizierung)“). Dieses funktioniert nur, wenn das Ver- und Entschlüsseln kommutativ zu einander ist, d.h. es muß

$$n = f_d(f_e(n)) = f_e(f_d(n))$$

gelten.

Natürlich wird im streng mathematischen Sinn durch die Veröffentlichung des öffentlichen Schlüssels auch der private Schlüssel preisgegeben, weil es einen mathematischen Zusammenhang zwischen ihnen geben muß. Wenn jedoch der Angreifer selbst mit den schnellsten Rechnern mehrere Jahre benötigt, um den privaten aus dem öffentlichen Schlüssel abzuleiten, dann ist die Veröffentlichung unproblematisch. Die Minimierung des Nutzens in Bezug zum Aufwand der Ermittlung des privaten Schlüssels ist somit eine Grundvoraussetzung für asymmetrische kryptographische Verfahren.

Für ein Netz mit 1000 Partnern benötigt man nur 1000 asymmetrische Schüsselpaare. Im Gegensatz dazu braucht man beim symmetrischen Verfahren etwa 500 000 Schlüssel.

Ein asymmetrisches Verfahren bietet folgende Vorteile: (vergleiche [BEU87])

1. Es ist kein Schlüsselaustausch notwendig. Mit Hilfe eines asymmetrischen Verfahrens ist somit eine spontane Kommunikation möglich.
2. Man braucht sehr wenige Schlüssel. Die Anzahl der benötigten Schlüsselpaare wächst nur linear mit der Anzahl der Teilnehmer. Bei symmetrischen Verfahren wächst die Anzahl der Schlüsselpaare quadratisch.
3. Asymmetrische Verfahren bieten eine ausgezeichnete Möglichkeit für digitale Unterschriften (vergleiche Kapitel „1.3.4 Digitale Signatur (Authentifizierung)“).

4. Das Einfügen neuer Teilnehmer ist unproblematisch. Der neue Teilnehmer erhält seinen privaten Schlüssel. Der passende öffentliche Schlüssel wird in einer Art „Telefonbuch" veröffentlicht, so daß jeder mit ihm sicher kommunizieren kann.

1.3.3 Schlüsselverteilung

Symmetrische kryptographische Systeme sind dadurch charakterisiert, daß sowohl für das Chiffrieren als auch für Dechiffrieren einer Nachricht der gleiche geheime Schlüssel benötigt wird. Dieser Schlüssel muß vor der eigentlichen Kommunikation auf sicherem Wege zwischen den Teilnehmern ausgetauscht und dann geheimgehalten werden. Zur Schlüsselverteilung benötigt man daher einen sicheren Kanal, was in diesem Fall das Hauptproblem des Schlüsselmanagements darstellt.

Auf den ersten Blick stellt die Schlüsselverteilung durch die Verwendung asymmetrischer kryptographischer Systeme kein Problem mehr dar, da man die öffentlichen Schlüssel offensichtlich gefahrlos über einen unsicheren Kanal austauschen kann. Dies ist jedoch nur richtig, solange man – unrealistischerweise – die Möglichkeit des Abhörens als die einzige Gefährdung der Sicherheit der Kommunikation betrachtet. Neben passiven Abhörversuchen muß in den meisten Fällen jedoch auch mit der Möglichkeit aktiver Angriffe gerechnet werden. Wenn sich ein aktiver Gegner in das Protokoll zum Schlüsselaustausch einschaltet, so muß er nur die öffentlichen Schlüssel, die sich die Kommunikationspartner *A* und *B* gegenseitig zusenden möchten, abfangen und beide Schlüssel durch neue eigene öffentliche Schlüssel ersetzen. Zu diesen Schlüsseln kennt der Angreifer natürlich auch den zugehörigen privaten Schlüssel. Während einer verschlüsselten Kommunikation zwischen *A* und *B* kann sich der Gegner dann alle Nachrichten umverschlüsseln und auf diese Weise unbemerkt den gesamten Nachrichtenverkehr abhören oder nach Belieben verändern, zurückhalten oder erweitern [FUM88]. Man nennt diese Art des Angriffs „Man-in-the-middle"-Angriff. Eine erprobte Gegenmaßnahme gegen ihn ist das sogenannte Interlock-Protokoll. Mehr Informationen hierzu findet man in [SCH96] auf Seite 59.

Zusammenfassend ist festzustellen, daß es durch den Übergang von einem symmetrischen zu einem asymmetrischen kryptographischen System bei der Schlüsselverteilung nicht notwendig ist, den öffentlichen Schlüssel geheimzuhalten. Es ist jedoch auch weiterhin unerläßlich, die übertragenen (öffentlichen) Schlüssel vor einer unbefugten Veränderung zu schützen. Passive Attacken können also bei der Verwendung eines „Public Key"-Systems zur Schlüsselverteilung praktisch ausgeschlossen werden. Zur Absicherung der Protokolle zur Schlüsselverteilung müssen aber auch bei asymmetrischen kryptographischen Systemen Maßnahmen zur Absicherung ergriffen werden. So können beispielsweise Zentralen zur Schlüsselverteilung dafür eingesetzt werden, die öffentlichen Schlüssel aller Teilnehmer eines Kommunikationsnetzes zu authentifizieren [FUM88].

1.3.4 Digitale Signatur (Authentifizierung)

Kann eine sichere Übertragung sowohl mit symmetrischen als auch mit asymmetrischen Verfahren erzielt werden, so sind zur Gewährleistung von Authentizität mit Hilfe sogenannter digitaler Signaturen (auch digitale Unterschrift genannt) nur asymmetrische Verfahren geeignet. Digitale Signaturen sind daher ein wichtiges Anwendungsfeld für asymmetrische kryptographische Verfahren. Wenn eine Nachricht zwischen zwei Kommunikationspartnern übermittelt wird, möchte der Empfänger oftmals sichergehen, daß die Nachricht tatsächlich vom behaupteten Sender stammt. Dieses kann durch eine digitale Signatur gewährleistet werden. Sie dient – ähnlich wie eine handschriftliche Unterschrift – dazu, beweisen zu können, daß die Nachricht tatsächlich von dem behaupteten Urheber stammt. Im Gegensatz zu einer herkömmlichen Unterschrift kann mit einer digitalen Unterschrift zusätzlich die Integrität einer Nachricht geprüft werden [SAU93].

Eine handschriftliche Unterschrift hat die Eigenschaft, daß sie nur von einem einzigen Menschen korrekt erzeugt, aber von allen Empfängern der Nachricht überprüft werden kann. Zumindest von denen, die die echte Unterschrift schon einmal gesehen haben, oder denen sie zum Vergleich vorliegt. Dieses sollte auch eine wesentliche Eigenschaft einer digitalen Signatur sein. Nur eine Person kann ein Dokument unterschreiben, aber jedermann kann überprüfen, ob die Unterschrift echt ist [RAN95].

1.3.5 Hash-Funktion (Integrität)

Hash-Funktionen werden in der Informatik seit langem eingesetzt. Eine Hash-Funktion erhält einen Eingabestring variabler Länge und wandelt ihn in einen in der Regel kürzeren Ausgabestring fester Länge, den Hash-Wert, um. Eine einfache Hash-Funktion wäre eine Funktion, die eine Eingabe entgegennimmt und ein Byte zurückgibt, das sich aus der XOR Verknüpfung aller Eingabebytes berechnet.

Hash-Funktionen sind nicht umkehrbar. Man kann also aus einem Hash-Wert nicht wieder das Original herstellen. Hash-Funktionen sind damit typische Einwegfunktionen. Sie sind so beschaffen, daß eine Änderung des Ausgangsdokuments mit hoher Wahrscheinlichkeit Auswirkungen auf den Hash-Wert hat.

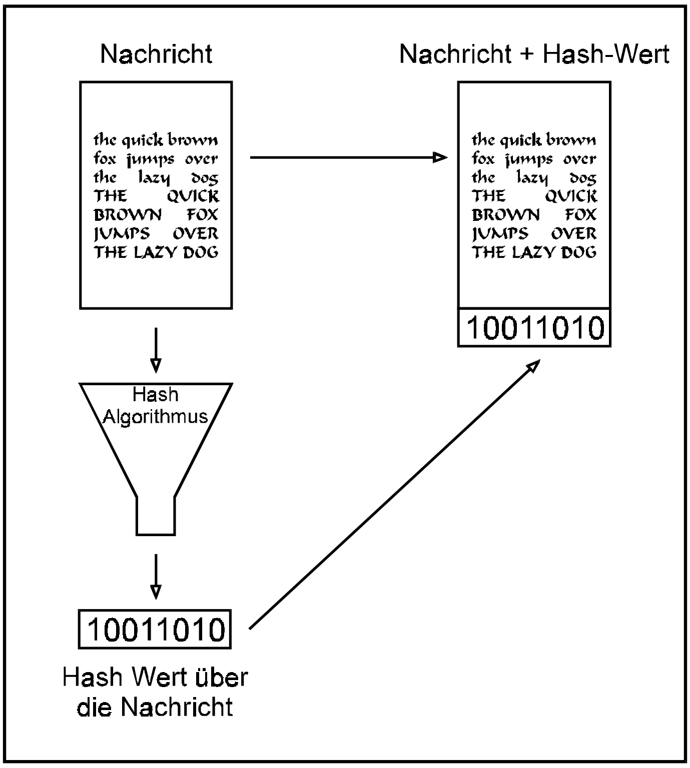

Abbildung 1.3: „Hash-Funktion“

Der Ablauf der Erzeugung eines Hash-Wertes läßt sich recht einfach darstellen (siehe Abbildung 1.3: „Hash-Funktion“). Aus der Nachricht, z.B. einer mit einer beliebigen Textverarbeitung erzeugten Datei, wird mit einem Hash-Algorithmus ein Hash-Wert gebildet. Das Ergebnis dieser Berechnung wird der Nachricht beigefügt. Beides wird verschlüsselt. Der Empfänger entschlüsselt die Botschaft und trennt Nachricht und Hash-Wert wieder voneinander. Die Nachricht wird mit dem gleichen Hash-Algorithmus wie beim Sender verarbeitet und mit dem empfangenden Hash-Wert verglichen. Sind beide Werte gleich, so wurde die Nachricht auf ihrem Übertragungsweg nicht verändert. Im anderen Fall wurde entweder die Nachricht oder die Signatur während der Übertragung verändert oder die Nachricht wurde nur unvollständig übertragen. Die Integrität ist damit nicht mehr gewährleistet.

2 Anforderungen & Problemstellung

2.1 Realisierung von asymmetrischen kryptographischen Verfahren

Wie bereits erläutert wurde, spielen asymmetrische Verfahren für die Verschlüsselung eine wichtige Rolle. Sie sind somit zur Realisierung von Sicherheitsgrundfunktionen in informationstechnischen Systemen von fundamentaler Bedeutung. In der Regel basieren asymmetrische kryptographische Verfahren auf sogenannten Einwegfunktionen mit Falltür (trapdoor one-way functions).

2.1.1 Einwegfunktionen ohne Falltür

Eine Einwegfunktion ohne Falltür ist eine mathematische Abbildung f, bei der $y = f(x)$ leicht zu berechnen ist, während die Berechnung der Umkehrfunktion $x = f^{-1}(y)$ um Größenordnungen schwieriger ist, wie z.B. a^2 gegenüber $\sqrt{a}$. So besitzt die Berechnung von f also eine geringe Komplexität, während die Berechnung von f^{-1} eine sehr hohe Komplexität aufweist.

Wenn eine Einwegfunktion f bijektiv, also eine Permutation ist, so spricht man auch von einer Einwegpermutation. Eine Einwegfunktion heißt kollisionsfrei, falls es praktisch unmöglich ist, zwei verschiedene Werte x und x' in der Urbildmenge X mit $f(x) = f(x')$ zu finden. Es ist klar, daß jede Einwegpermutation kollisionsfrei ist, denn hier gibt es keinen Funktionswert mit zwei verschiedenen Urbildern, da jeder Wert nur genau ein Urbild hat [BEU95].

Ein Beispiel für eine Einwegfunktion ohne Falltür ist die Login-Prozedur eines Rechnersystems. Wenn ein Benutzer zum ersten Mal ein Kennwort benutzt (Zuteilung des Kennwortes) oder wenn er es ändert, wird das Kennwort einem Verschlüsselungsprozeß unterworfen. Die verschlüsselte Form des Kennwortes wird gespeichert. Das heißt, die Login-Prozedur f berechnet zu einem Paßwort PW den Wert $f(PW)$, der in einer Paßwortdatei gespeichert wird. Bei jedem folgenden Login berechnet der Rechner $f(x)$, wobei x das vom Benutzer eingegebene Kennwort ist, und vergleicht diesen Wert mit dem gespeicherten $f(PW)$. Sind die Werte gleich, so wird der Benutzer akzeptiert, d.h. er wird authentifiziert [HOR85].

Verschlüsselungsverfahren, die auf Einwegfunktionen aufgebaut sind, wurden für Problemstellungen entwickelt, bei denen eine Verschlüsselung von Daten, nicht aber deren prinzipielle Entschlüsselung erforderlich ist [HOR85].

2.1.2 Einwegfunktionen mit Falltür

Eine grundlegende Basis für asymmetrische kryptographische Verfahren sind die Einwegfunktionen mit Falltür. Die Berechnung der Umkehrfunktion ist wie bei den Einwegfunktionen ohne Falltür im komplexitätstheoretischen Sinn nicht durchführbar. Besitzt die Einwegfunktion jedoch eine Falltür, ist es möglich die Umkehrfunktion zu berechnen, falls eine zusätzliche Information bekannt ist. Ein „Public Key"-System wird aus solchen Einwegfunktionen mit Falltür aufgebaut. Der private Schlüssel bildet die zusätzliche Information für die Falltür, mit der die Umkehrfunktion berechnet wird.

Eine Falltürfunktion (trapdoor function) besteht aus einer Funktion, die leicht ausgewertet werden kann, und einem Schlüssel. Mit Kenntnis des Schlüssels ist die Funktion einfach zu invertieren, anderenfalls ist dieses ein schwieriges Problem [GOL94].

Das Hauptproblem für die Realisierung von praktisch nutzbaren, asymmetrischen kryptographischen Verfahren ist, Funktionen mit den oben genannten Eigenschaften zu finden. Man kennt bis heute nur sehr wenige davon. Ein wesentliches Problem ist, daß die genaue Komplexität für die Berechnung der Umkehrfunktion meist nicht bekannt ist. Man verwendet zur Realisierung von Einwegfunktionen seit langem untersuchte und bekanntermaßen schwierige mathematische Probleme, zu deren Lösung nur Algorithmen bekannt sind, die exponentiellen Aufwand erfordern. Zu nennen sind hier das Faktorisierungsproblem von großen Zahlen oder die Berechnung des diskreten Logarithmus über bestimmten mathematischen Strukturen.

Sollten in Zukunft jedoch schnelle Algorithmen zur Lösung dieser Probleme gefunden werden, so sind alle kryptographischen Verfahren, die auf ihnen beruhen, nicht mehr einsetzbar. Die Mathematik beschäftigt sich allerdings schon lange Zeit ohne Erfolg damit, solche Algorithmen zu finden. So geht man davon aus, daß auch in Zukunft auf diesen Gebieten kein Durchbruch erzielt werden wird.

2.2 RSA-Verfahren

Eine der ersten praktischen Realisierungen eines asymmetrischen kryptographischen Verfahrens wurde im Jahre 1978 von Rivest, Shamir und Adleman am MIT Laboratory for Computer Science in Cambridge vorgestellt [RIV78]. Das Verfahren ist nach den Autoren RSA-Verfahren genannt. Es ist das bekannteste und bisher am häufigsten eingesetzte und analysierte „Public Key"-Verfahren [BAU93].

Die grundlegende Idee der „Public Key"-Verfahren ist die Aufteilung des Schlüssels in einen Teil, der zum Chiffrieren verwendet wird, und einen Teil, der zum Dechiffrieren verwendet wird. Das RSA-Verfahren beruht auf der Vermutung, daß die Potenzierung modulo k unter bestimmten Voraussetzungen eine Einwegfunktion mit Falltür ist.

2.2.1 Die Eulersche Funktion

Um die Funktionsweise des RSA-Verfahrens verstehen zu können, werden einige mathematische Grundlagen benötigt. Zu diesen gehört die Eulersche Funktion.

Für eine natürliche Zahl n ist $\varphi(n)$ als die Anzahl der natürlichen Zahlen kleiner n, die zu n relativ prim sind, definiert. Es werden also alle natürlichen Zahlen kleiner n, deren größter gemeinsamer Teiler mit n gleich 1 ist, gesucht. Einige Beispiele:

$$\varphi(1) = 1,\ \varphi(2) = 1,\ \varphi(3) = 2,\ \varphi(4) = 2,\ \varphi(6) = 2,\ \varphi(10) = 4,\ \varphi(15) = 8.$$

(Die letze Aussage folgt so: Die zu 15 teilerfremden Zahlen kleiner als 15 sind: 1, 2, 4, 7, 8, 11, 13 und 14. Das sind acht Zahlen, daraus folgt $\varphi(15) = 8$.)

Hieraus folgen die allgemeinen Aussagen:

1. Ist p eine Primzahl, so ist

$$\varphi(p) = p - 1;$$

denn jede der $(p - 1)$ Zahlen $1, 2, 3, ..., (p - 1)$ ist teilerfremd zu p.

2. Sind p und q zwei verschiedene Primzahlen, so ist

$$\varphi(pq) = (p - 1)(q - 1).$$

Dies kann wie folgt begründet werden: Es gibt insgesamt genau $(pq - 1)$ natürliche Zahlen, die kleiner als pq sind. Nun zählt man, wieviele von diesen nicht teilerfremd zu pq sind. Dies sind zum einen die $(q - 1)$ Vielfachen von p, also

$$p, 2p, 3p, ..., (q - 1)p$$

und zum anderen die $(p - 1)$ Vielfachen von q, d.h.:

$$q, 2q, 3q, ..., (p - 1)q.$$

Da dies alles Zahlen zwischen 1 und pq sind, die nicht teilerfremd zu pq sind, ergibt sich

$$\varphi(pq) = (pq - 1) - (q - 1) - (p - 1) = (p - 1)(q - 1).$$

Nun können wir den Satz von Euler formulieren:

Sind m und k zwei natürliche Zahlen, die teilerfremd sind, so gilt

$$m^{\varphi(k)} \equiv 1 \bmod k$$

(vergleiche [BEU87]).

Beispiele:
1. $m = 3$, $k = 5 \Rightarrow \varphi(k) = 4$, $3^4 \bmod 5 = 1$
2. $m = 7$, $k = 13 \Rightarrow \varphi(k) = 12$, $7^{12} \bmod 13 = 13.841.287.201 \bmod 13 = 1$

2.2.2 Der erweiterte Euklidischer Algorithmus

Der Kehrwert von 4 bezüglich der Multiplikation ist ¼, da $4 \cdot ¼ = 1$. In bezug auf die Modulo-Operation ist das Problem komplizierter:

$$4 \cdot x \equiv 1 \text{ (mod 7)}.$$

Diese Gleichung ist äquivalent zur Bestimmung der ganzen Zahlen x und n für die gilt:

$$4 \cdot x = 7 \cdot n + 1.$$

Das allgemeine Problem lautet, ein x zu ermitteln, das

$$1 = (a \cdot x) \bmod k$$

erfüllt. Dieses schreibt man auch

$$a^{-1} \equiv x \text{ (mod } k\text{)}.$$

Allgemein gilt, daß $a^{-1} \equiv x$ (mod k) eindeutig lösbar ist, wenn a und k relativ prim zueinander sind. Ist k eine Primzahl, so ist jede Zahl zwischen 1 und k - 1 relativ prim zu k und besitzt in dieser Menge genau einen Kehrwert modulo k.

Zur Berechnung des Kehrwertes von a mod k kann der erweiterte Euklidische Algorithmus verwendet werden. Ein Sourcecode in C++ findet sich in [SCH96] auf Seite 289 f.

Verschlüsseln: $c = n^e \bmod k$ Entschlüsseln: $n = c^d \bmod k$ $k = p \cdot q$	c: Chiffrat e: öffentlicher Schlüssel n: Nachricht d: privater Schlüssel k: öffentlicher Modulus p, q: geheime Primzahlen

Abbildung 2.1: „Ver- und Entschlüsselung mit RSA“

2.2.3 Funktionsweise des RSA-Verfahren

Das Verschlüsseln mit dem RSA-Verfahren erfolgt folgendermaßen: Jeder Kommunikationspartner bekommt von einer Schlüsselverteilzentrale einen privaten und einen öffentlichen Schlüssel zugeteilt. Der private Schlüssel ist eine Zahl d, die nur seinem Besitzer bekannt ist. Der öffentliche Schlüssel ist eine Zahl e, die öffentlich bekannt gegeben wird. Die zu verschlüsselnde Nachricht wird in eine Folge von ganzen Zahlen zwischen 0 und k-1 zerlegt, wobei k ein öffentlicher Parameter des Verfahrens ist. Die Verschlüsselung einer Teilnachricht n in ein Chiffrat c und die zugehörige Entschlüsselung erfolgt mit

$$c = n^e \bmod k \text{ und } n = c^d \bmod k. \qquad (1)$$

Die Zahlen k, d und e müssen so gewählt werden, daß gilt:

$$k = p \cdot q \tag{2}$$

$$\text{ggT}(d, (p - 1)(q - 1)) = 1, \text{ mit } d > 1 \tag{3}$$

$$e \cdot d \bmod ((p - 1)(q - 1)) = 1 \tag{4}$$

Dabei sind p und q zwei sehr große, unterschiedliche Primzahlen. Je nach den Sicherheitsanforderungen sollte k zwischen 512 Bit und 1024 Bit lang sein. Aus der Gleichung (3) folgt, daß d relativ prim zu $(p - 1)(q - 1)$ ist. Dieses ist eine notwendige Bedingung, damit ein zu d multiplikativ inverses Element e nach der Gleichung (4) existiert.

Mit anderen Worten: Es werden zwei Primzahlen p und q und ein Chiffrierexponent e ausgewählt. Für diesen wird in der Regel eine kleine Zahl gewählt, um das Chiffrieren zu beschleunigen. Dann wird das Produkt $k = p \cdot q$ gebildet und der Dechiffrierexponent d mit Hilfe der Eulerschen φ-Funktion bestimmt, so daß gilt

$$e \cdot d \equiv 1 \bmod \varphi(k).$$

Dabei ist

$$\varphi(k) = (p - 1)(q - 1).$$

Den öffentlichen Schlüssel bilden nun die Zahlen k und e. Die Primzahlen p und q und der Dechiffrierexponent d werden geheimgehalten [GOL94].

Den privaten Schlüssel d erhält man also, indem man eine zu $\varphi(k)$ relativ prime Zahl d bestimmt. Diese Bedingung wird von jeder Primzahl $p' > \max(p, q)$ erfüllt. Es ist allerdings sinnvoll, eine möglichst große Zahl d, $d < \varphi(k) - 1$ zu wählen, damit sie nicht durch direktes Suchen ermittelt werden kann. Direktes Suchen bedeutet hier: Man verschlüsselt zunächst eine beliebige Nachricht n, $n \in [0: k - 1]$ mit dem öffentlichen Schlüssel: $c = n^e \bmod k$. Anschließend sucht man eine Zahl t mit $c^t \bmod k = n$ [HOR85].

Um den öffentlichen Schlüssel e zu ermitteln, berechnet man den größten gemeinsamen Teiler von d und $\varphi(k)$ mit dem erweiterten Euklidischen Algorithmus. Gleichzeitig erhält man $x, y \in Z$ mit $x \cdot d + y \cdot \varphi(k) = \text{ggT}(d, \varphi(k)) = 1$. Somit ist $e = x \bmod \varphi(k)$ die gesuchte Schlüsselkomponente, weil $x \cdot d \equiv 1 \bmod \varphi(k)$ ist [HOR85].

Nachrichten werden als Zahlen n, $0 \leq n < k$, codiert und mit $c = n^e \bmod k$ chiffriert. Dann folgt, im wesentlichen aus dem Satz von Fermat,

$$c^d \equiv n \bmod k .$$

Das Schlüsselpaar (e, d) kann in dualer Weise für digitale Unterschriften herangezogen werden. Dabei dient der private Schlüssel d zum Unterschreiben einer Nachricht und der öffentliche Schlüssel e zum Verifizieren einer Unterschrift.

Als Falltürfunktion wird hier die Abbildung $n \rightarrow n^e$ in Z_n herangezogen. Es ist offensichtlich trivial, diese Operation zu invertieren, wenn *d* oder die Faktoren *p* und *q* bekannt sind. Die Berechnung von *d* (oder von *p* und *q*) nur aus der Kenntnis von *k* und *e* muß schwierig sein. Auf diese Weise ist die Sicherheit von RSA eng mit dem Faktorisierungsproblem verbunden, einem Gebiet, in dem in letzter Zeit wieder überraschende Fortschritte gemacht wurden. Es erscheint daher nicht mehr ratsam, auf die Sicherheit einer Implementierung mit 512-Bit-Zahlen zu vertrauen.

2.2.4 Beispiel zum RSA-Verfahren

Zur Verdeutlichung des RSA-Verfahrens folgt ein Beispiel aus [RIV78]. Zur Demonstration der kryptographischen Stärke sind die benutzten kleinen Werte natürlich nicht geeignet.

$p = 47,\ q = 59$
$k = p \cdot q = 47 \cdot 59 = 2773$
$\varphi(k) = (p - 1) \cdot (q - 1) = 46 \cdot 58 = 2668$
$d = 157 \in P \Rightarrow ggT(d, \varphi(k)) = 1$
e erhält man mit Hilfe des erweiterten Euklidischen Algorithmus: $e = 17$

Wir haben somit folgende Werte erhalten:

$p = 47$	geheim, Trapdoor Information
$q = 59$	geheim, Trapdoor Information
$k = 2773$	öffentlich
$\varphi(k) = 2668$	geheim
$e = 17$	öffentlich
$d = 157$	geheim

Zunächst müssen wir das Alphabet geeignet kodieren. Setzen wir also „_“ = 00, A = 01, B = 02, C = 03, ... , Y = 25 und Z = 26, so können wir mit k = 2773 zwei Buchstaben zu einem Block zusammenfassen und dann, nach der entsprechenden Codierung, verschlüsseln.

„ITS_ALL_GREEK_TO_ME_“ wird durch
0920 1900 0112 1200 0718 0505 1100 2015 0013 0500 codiert.

Da $e = 17$ ist, wird der erste Block ($n = 920$) durch $n^e \bmod k = 920^{17} \bmod 2773 = 948$ verschlüsselt.

Die gesamte Nachricht wird verschlüsselt zu
0948 2342 1084 1444 2663 2390 0778 0774 0219 1655.

Wegen $d = 157$ erhält man $n'^d \bmod k = 948^{157} \bmod 2773 = 920$.

2.2.5 Sicherheit von RSA

Die Sicherheit des Algorithmus basiert auf dem Problem der Faktorisierung großer Zahlen. Es ist sehr einfach, den öffentlichen Modulus aus den beiden Primzahlen durch

Multiplikation zu berechnen, aber sehr schwierig, den Modulus wieder in seine beiden Primfaktoren zu zerlegen, da es zur Zeit keine hinreichend schnell arbeitenden Algorithmen dafür gibt.

Strenggenommen ist das nicht wahr: Man *vermutet*, daß die Sicherheit von RSA auf dem Problem der Faktorisierung großer Zahlen basiert. Es wurde nie mathematisch bewiesen, daß man den Modulus k faktorisieren muß, um den Klartext n aus dem öffentlichen Schlüssel e und dem Geheimtext c zu berechnen. Es ist denkbar, daß eine völlig andere Art, RSA zu brechen, entdeckt wird. Könnte ein Angreifer allerdings auf diese neue Art den Wert des privaten Schlüssels d ermitteln, so hätte man damit auch ein neues Verfahren zur Faktorisierung großer Zahlen [SCH96].

Um eine ausreichende Sicherheit zu gewährleisten, werden hinreichend große Schlüssel gefordert. Zur Zeit kommen noch überwiegend 512 Bit lange Schlüssel zum Einsatz. Allerdings finden auch schon 768 Bit und 1024 Bit lange Schlüssel Verwendung. Mit steigender Schlüssellänge steigt auch der Rechenaufwand zur Ver- und Entschlüsselung. Diese Steigerung ist jedoch nicht linear, sondern annähernd exponentiell [RAN95]. Eine der Stärken des RSA-Verfahren ist es aber gerade, daß es nicht auf eine bestimmte Schlüssellänge fixiert ist.

In [SCH96] beschäftigt sich das Kapitel 11.4 ausschließlich mit der Faktorisierung großer Zahlen. Zur Zeit (1996) stellt eine Zahl mit 129 Dezimalziffern die Grenze dessen dar, was machbar ist. Natürlich kann ein Angreifer jedes mögliche d probieren, bis er über den korrekten Wert stolpert. Dieser Brute-Force-Angriff ist jedoch noch ineffizienter als der Versuch, k zu faktorisieren.

2.3 Vorgaben

Das Ziel dieser Diplomarbeit ist es, einen Hardwareakzelerator für das RSA-Verfahren mit einer Schlüssellänge von 1024 Bit zu entwickeln. Der zu erstellende Chip muß in der Lage sein, die Funktion

$$a = x^y \bmod k$$

möglichst schnell zu berechnen. Hierbei sind die folgenden Vorgaben zu beachten. Die maximale Größe der Zahlen x, y und k beträgt 2^{1024}-1. Die Geschwindigkeit, mit der die Daten verschlüsselt werden sollen, beträgt minimal 9,6 kBit/s. Die Taktfrequenz des ASICs[7] sollte nicht höher als 12 MHz liegen. Die maximale Größe des Chips sollte zwischen 80 und 90 mm^2 betragen. Die Leistungsaufnahme ist unkritisch. In der folgenden Abbildung sind die Vorgaben zusammengefaßt.

[7] Application Specific Integrated Circuit (ASIC)

Operandenbreite	*1024 Bit*
Taktfrequenz	*maximal 12 MHz*
Geschwindigkeit	*mindestens 9,6 kBit/s*
Größe	*maximal 80 - 90 mm*^{2}
Leistungsaufnahme	*unkritisch*

Abbildung 2.2: „Vorgaben RSA-Chip“

Die Aufgabe ist es somit, einen Chip zu entwerfen, der eine möglichst schnelle Exponentiation mit Modulo-Reduktion durchführen kann. Diese läßt sich durch eine Folge von Modulo-Multiplikationen erreichen. Deren effiziente Implementierung ist die zentrale Herausforderung dieser Diplomarbeit.

3 Zugrundeliegende Methoden

3.1 Exponentiation

Das Problem, wie eine Exponentiation möglichst effizient ausgeführt werden kann, ist in der Literatur seit langem untersucht worden. Eine ausführliche Übersicht findet man bei [KNU81] im Kapitel 4.6.3. Ziel eines jeden Verfahrens ist es, eine Exponentiation auf möglichst wenige Multiplikationen zurückzuführen. Beachtet werden muß, daß eine geringe Anzahl von Multiplikationen nicht mit einem inakzeptabel hohen Speicherplatzbedarf erkauft wird. Aus diesem Grunde scheint für den vorliegenden Fall, nur die „Binärmethode" in Frage zu kommen. Alle anderen im Folgenden erläuterten Verfahren (wie zum Beispiel die Methode mit Hilfe von Additionsketten) benötigen sehr viel Platz für zusätzliche Register. Weitere Methoden zur Beschleunigung der Exponentiation sind redundante Zahlendarstellungen und der chinesische Restsatz.

3.1.1 Die Binärmethode (Square-and-Multiply)

Die Binärmethode ist die Standardmethode zur Exponentiation [KNU81]. Hierbei wird die Exponentiation auf Quadrierungen und bedingte Multiplikationen zurückgeführt. Meist wird das Verfahren auch als Square-and-Multiply (Quadrieren und Multiplizieren) bezeichnet.

Prinzipiell könnte man beispielsweise zur Berechnung von x^{16} den Wert x sechzehnmal multiplizieren. Wesentlich effizienter ist es jedoch, mit vier Quadrierungen zum Endergebnis zu gelangen: x, x^2, x^4, x^8, x^{16}. Diese Idee liegt dem Binärverfahren zugrunde. Zur Exponentiation x^y wird y als Dualzahl ($y_{1023}y_{1022}$ ... y_1y_0) dargestellt. Die jetzt vorliegende Binärrepräsentation des Exponenten wird bitweise abgearbeitet. Dabei wird die Basis x sukzessive quadriert und das jeweilige Zwischenergebnis in Abhängigkeit vom Wert des aktuellen Exponentenbits mit x multipliziert.

Es gibt zwei verschiedene Binärmethoden. Bei der einen Methode wird der Exponent beim höchstwertigen Bit beginnend durchlaufen. Bei der anderen Methode beginnt man beim niederwertigsten Bit (LSB). Wenn der Exponent von links nach rechts, also vom höchsten Bit her, abgearbeitet wird, spricht man vom „seriellen" Binärverfahren:

```
/* a = xʸ */
int exp1(int x, int y)
{
  int i,a;
  a = 1;
  for(i=1023;i>=0;i--)
  {
    a = a * a;
    if (yᵢ == 1) a = a * x;
  }
  return(a);
}
```

In jedem Schleifendurchlauf muß der neue Wert von *a* vor der Multiplikation mit *x* berechnet werden. Wenn der Exponent von rechts nach links, also vom niedrigsten Bit her, abgearbeitet wird, erhält man das „parallele“ Binärverfahren:

```
/* a = xʸ */
int exp2(int x, int y)
{
  int i,a,b;
  a = 1;
  b = x;
  for(i=0;i<1024;i++)
  {
    if (yᵢ == 1) a = a * b;
    b = b * b;
  }
  return(a);
}
```

In jedem Schleifendurchlauf können die neuen Werte von *a* und *b* parallel berechnet werden. Dieses bedeutet allerdings einen erhöhten Hardwareaufwand. Es werden ein zweiter Multiplizierer bzw. ein Quadrierer und ein weiteres Register für *b* benötigt. Der Hardwareaufwand verdoppelt sich somit, die Geschwindigkeit nimmt aber im Schnitt nur um 50 Prozent zu, wenn man annimmt, daß der Exponent etwa zur Hälfte aus Nullen und Einsen besteht. Im schlechtesten Fall (der Exponent besteht nur aus Nullen) ergibt sich keine Verbesserung. Im besten Fall (der Exponent besteht nur aus Einsen) verdoppelt sich der Durchsatz.

3.1.2 Additionsketten

Eine Additionskette für eine Zahl *e* ist eine Zahlenfolge, die mit 1 beginnt und mit *e* endet, wobei jedes Element in der Folge die Summe von zwei früheren Folgeelementen ist. Kürzeste Additionsketten sind keineswegs eindeutig. Es ist noch immer ein offenes Problem, effiziente Algorithmen zu finden, die kurze Additionsketten erzeugen [GOL94].

Aufgrund der allgemeinen Regel $c^x \cdot c^y = c^{x+y}$ könnte beispielsweise der Wert c^{15} abweichend von der Binärmethode über die Berechnungsfolge c, c^2, c^3, c^6, c^{12}, c^{15} ermittelt werden. Jedes Teilergebnis entsteht durch Multiplikation zweier bereits berechneter Teilergebnisse oder der Quadrierung eines bereits berechneten Teilergebnisses. Die hierbei entstehende Folge von Exponenten ist die Additionskette. Im obigen Fall lautet die Kette (1, 2, 3, 6, 12, 15). Zur Berechnung von c^{15} über diese Folge werden fünf Multiplikationen benötigt. Die Binärmethode würde für c^{15} zur Kette (1, 2, 3, 6, 7, 14, 15) führen, was sechs Multiplikationen entspricht. Je kürzer die für den Exponenten gefundene Additionskette ist, desto geringer ist die benötigte Anzahl von Multiplikationen zur Exponentiation.

Die Ersparnis von Multiplikationen ergibt sich dadurch, daß ein Weg zum Zielexponenten 15 verfolgt wird, auf dem Teilpotenzen entstehen, die den weiteren Weg zum Ziel möglichst kurz machen. Zum Beispiel erlaubt die Zwischenspeicherung von c^3, vom vorletzten Element c^{12} der Kette mit einer Multiplikation auf das Endergebnis zu kommen. Es gibt also Zwischenergebnisse, die mehrmals zur Berechnung weiterer Potenzen dienen und daher zwischengespeichert werden müssen. Dadurch erkauft man die Verringerung der Anzahl der Multiplikationen mit einem erhöhten Speicherplatzbedarf für Zwischenergebnisse [SAU93].

Der Aufwand zusätzlicher Hardware, um ein auf Additionsketten basierendes Verfahren zu implementieren, ist erheblich, weil Zwischenergebnisse für die weitere Berechnung gespeichert werden müssen. Für jedes Zwischenergebnis ist ein weiteres Register nötig. Das Problem der Ermittlung der minimalen Länge einer Additionskette ist NP-vollständig. Detaillierte Untersuchungen zu Additionsketten bezüglich ihrer Zeit- und Platzkomplexität finden sich in [YAC90].

3.1.3 Redundante Zahlendarstellungen

Eine andere Möglichkeit zur Verringerung der Zahl der Multiplikationen einer Exponentiation bieten redundante Darstellungen des Exponenten. Ein Beispiel hierfür ist die „Signed-Digit"-Darstellung, in der Zahlen bezüglich einer Basis β mit Koeffizienten aus dem Intervall $]-\beta, \beta[$ dargestellt werden. Die „Signed-Digit"-Darstellung bezüglich der Basis 2 wird also mit Koeffizienten aus der Menge {-1, 0, 1} dargestellt. Das serielle Binärverfahren wird wie folgt modifiziert:

```
/* a = x^y */
int exp3(int x, int y)
{
  int i,a;
  a = x;
  for(i=1022;i>=0;i--)
  {
    a = a * a;
    if (y_i == 1) then a = a * x;
    if (y_i == -1) then a = a * x^-1;
  }
  return(a);
}
```

Es wird selten effizient sein, die Multiplikation mit x^{-1} als Division durch x zu realisieren. Vielmehr liegt es nahe, x^{-1} einmal zu berechnen und dann im Speicher abzulegen.

Ein Vorteil dieser Methode ist, daß der Exponent nach der Umcodierung in die redundante Zahlendarstellung mehr Nullen enthält. Auf diese Weise können Multiplikationen eingespart werden. Es gibt verschiedene Algorithmen um die Anzahl der Nullen im Exponent zu maximieren (vergleiche [SAU93], Seite 44 ff.). Ein Nachteil ist der erhöhte Platzbedarf für ein zusätzliches Register.

3.1.4 Chinesischer Restsatz

Es ist möglich die Exponentiation des RSA-Verfahren durch zwei parallel ausführbare Exponentiationen mit jeweils halb so großen Operanden zu realisieren. Dies ist möglich, wenn die Zerlegung des Modulus k in seine beiden Primfaktoren p und q bekannt ist.[8] Für jede dieser Exponentiationen fallen halb so viele Multiplikationen an. Jede Multiplikation kann aufgrund der halbierten Operandengröße doppelt so schnell ausgeführt werden. Hierbei wird davon ausgegangen, daß sich die der Multiplikation zugrunde liegende Addition mit halber Operandenbreite nicht schneller ausführen läßt, sondern nur die Anzahl der Additionen halbiert wird. Insgesamt wird eine Steigerung der Geschwindigkeit der Exponentiation um den Faktor vier erreicht. Der Chinesische Restsatz besagt für den vorliegenden Spezialfall:

1. Sind p und q zwei teilerfremde Zahlen und $x_p \in [0, p - 1]$ und $x_q \in [0, q - 1]$ zwei beliebige Zahlen, dann gibt es ein eindeutiges $x \in [0, p \cdot q - 1]$ für das gilt

$$x \equiv x_p \pmod p \text{ und } x \equiv x_q \pmod q.$$

2. Wenn $u \cdot q \equiv 1 \pmod p$, dann ergibt sich x aus x_p und x_q durch

$$x = (\,(\,(x_p - x_q)\,u\,) \bmod p)\,q + x_q.$$

Für der Beweis wird auf [SAU93] Seite 57 verwiesen. Jede Zahl $x \in [0, p \cdot q - 1]$ kann also durch zwei Zahlen $x_p \in [0, p - 1]$ und $x_q \in [0, q - 1]$ eindeutig repräsentiert werden. Der Chinesische Restsatz kann zur Berechnung von

$$n = x^y \bmod k, \text{ (mit } k = p \cdot q)$$

wie folgt verwendet werden:

Die Zahl n wird repräsentiert durch die beiden Zahlen n_p und n_q,

$$n_p = x^y \bmod p$$

$$n_q = x^y \bmod q.$$

Mit Hilfe des Chinesischen Restsatzes wird dann aus n_p und n_q der Wert n berechnet:

$$n = \begin{cases} (\,(\,(n_p - (n_q \bmod p)\,)\,u\,) \bmod p)\,q + n_q & \text{falls } n_p \geq n_q \bmod p \\ (\,(\,(n_p + p - (n_q \bmod p)\,)\,u\,) \bmod p)\,q + n_q & \text{sonst} \end{cases}$$

[8] Die Primfaktoren p und q sind nur dem Besitzer des privaten Schlüssels bekannt, bzw. sie können von ihm berechnet werden. Somit kann nur er die Vorteile des „Chinesischen Restsatzes“ nutzen.

Die benötigte Zahl u, für die gilt $u \cdot q \equiv 1 \pmod{p}$, kann mit dem erweiterten Euklidischen Algorithmus gefunden werden. Genauere Informationen zum Chinesischen Restsatz und die Herleitung des hier vorgestellten Verfahrens zur Vereinfachung der Exponentiation des RSA-Verfahrens finden sich in [SAU93].

3.2 Multiplikation

Es gibt zwei grundlegende Wege, um die Multiplikation zu beschleunigen:

1. Beschleunigung der zugrunde liegenden Addition
2. Reduzierung der Anzahl der benötigten Additionen

Der Basisalgorithmus für die Multiplikation addiert in Abhängigkeit von den Ziffern y_i des Multiplikators y Teilprodukte zu einem Gesamtprodukt $y \cdot x$. Dabei werden, beginnend mit der höchstwertigen Ziffer von y, sukzessiv die Vielfachen $y_i \cdot x$ aufsummiert. Nach der Teiladdition wird das bisherige Teilprodukt um eine Stelle nach links verschoben. Das Ergebnis ergibt sich nach der folgenden Gleichung:

$$x \cdot y = x \cdot \sum_{i=0}^{l-1} y_i b^i$$

Dabei ist b die Zahlenbasis, in der der Operand y dargestellt ist, und l seine Stellenzahl. Wählt man $b = 2$ als Zahlenbasis (Binärdarstellung) und $l = 1024$ als Stellenzahl, so ergibt sich:

$$x \cdot y = x \cdot \sum_{i=0}^{1023} y_i \cdot 2^i \quad \text{mit } y_i \in \{0, 1\}$$

Die Effizienz der Multiplikation kann durch verschiedene Verfahren verbessert werden. So können mehrere Teiladditionen parallel ausgeführt werden. Dies bedeutet allerdings immer eine erhebliche Erhöhung des Hardwareaufwandes. Ein klassisches Verfahren zur Parallelisierung der Berechnung des Produkts $x \cdot y$ ist der Wallace-Baum. Alle (oder einige) Teilprodukte $x \cdot y_i$ werden parallel berechnet und die Addition

$$x \cdot y = \sum_{i=0}^{1023} x \cdot y_i \cdot 2^i$$

wird mit einem Baum von Addierern vorgenommen.

Vuillemin [VUI83] schlägt einen einfachen rekursiven parallelen Algorithmus vor, um zwei n-Bit Zahlen in O(log n) Zeit zu multiplizieren. Ein sequentieller Algorithmus benötigt O(n) Zeit. Eine 32-Bit Multiplikation arbeitet nach seinem Verfahren dreimal so schnell wie der vergleichbare sequentielle Algorithmus, benötigt aber nur 40 Prozent mehr an Fläche. Die einzelnen Additionen werden nach der „Carry-Save“-Methode durchgeführt.

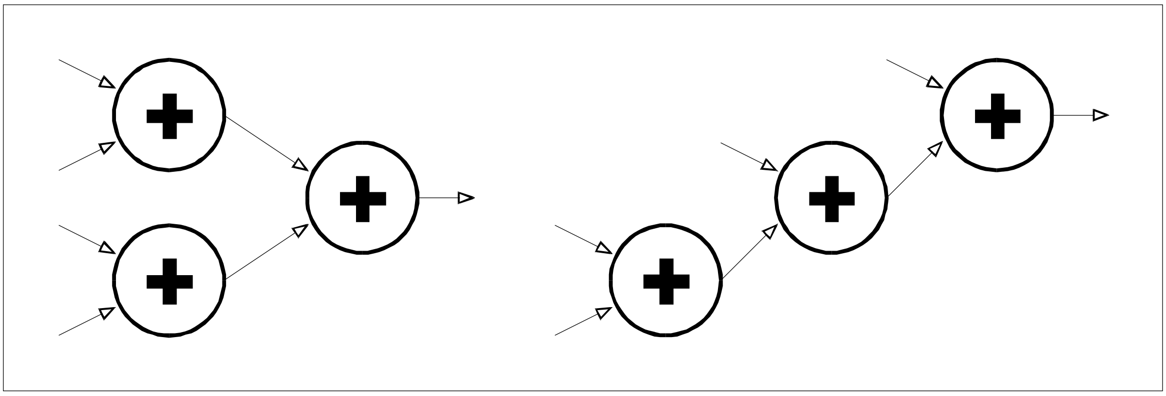

Abbildung 3.1: „paralleler und sequentieller Multiplizierer“

Eine andere Möglichkeit ist die Untersuchung von mehreren Bits pro Zyklus. In Abbildung 3.2: „Multiplizierer mit zwei Bit Verarbeitung pro Takt“ sind die Aktionen für einen Multiplizierer dargestellt, die benötigt werden um zwei Bits pro Takt verarbeiten zu können [OMO84].

$MR_{i+1,i}$	*Aktion*
00	addiere 0 · MD; PP um zwei Bits schieben
01	addiere 1 · MD; PP um zwei Bits schieben
10	addiere 2 · MD; PP um zwei Bits schieben
11	addiere 3 · MD; PP um zwei Bits schieben
	(MR = Multiplikator, MD = Multiplikand, PP = partielles Produkt)

Abbildung 3.2: „Multiplizierer mit zwei Bit Verarbeitung pro Takt“

Eine weitere Möglichkeit zur Beschleunigung der Multiplikation stellt das Verfahren von Sedlak [SED87] dar. Es ist eine Variante des sogenannten „Booth-Recoding“. Der Multiplikator wird in Läufe („runs“) zerlegt, d.h. in Folgen von Nullen oder Einsen. Ein Lauf von Nullen bewirkt eine Verschiebung. Ein Lauf von Einsen erfordert zwei Additionen und eine Verschiebung. Demnach entspricht ein Lauf von i Einsen dem Multiplikator 2^i - 1 (siehe hierzu Abbildung 3.3: „Beispiel zum Verfahren nach Sedlak“). Es wird eine maximale Verschiebung um drei Stellen implementiert. Längere Läufe müssen in mehrere Schritte zerlegt werden.

Es soll

$$5\,(101) \cdot 7\,(111) = 35\,(100011)$$

berechnet werden.

```
„normal“:            „nach Sedlak“:
                     (8 - 1) · 5
   101
+   101                101
+    101             -    101
  ------               ------
= 100011             = 100011
```

Abbildung 3.3: „Beispiel zum Verfahren nach Sedlak“

Einigkeit besteht in der Literatur darüber, daß die Grundlage, um einen effizienten Multiplizierer zu implementieren, eine Addition nach dem „Carry-Save“-Prinzip ist. Um einen „Carry-Save“-Multiplizierer aufzubauen, wird relativ wenig Hardware gebraucht. Es sind pro Bit nur ein Volladdierer und ein Speicherelement („Flip-Flop“) nötig. Außerdem ist eine hohe Taktfrequenz möglich, weil die Signallaufzeit, die zur Weiterleitung des Übertrags erforderlich ist, sehr kurz ist. Ein Problem ist, daß die Zahlendarstellung eines „Carry-Save“-Multiplizierers redundant ist. Unterschiedliche Kombinationen der Register für den Übertrag und für die Partialsumme repräsentieren die gleiche Zahl. Dieses wird deutlich, wenn man sich klar macht, daß eine n Bit große Zahl mit $2n$ Bit dargestellt wird. Deshalb ist es erforderlich, nach der letzten Addition eine „normale“ Darstellung zu erhalten. Dieses ist ohne zusätzliche Hardware möglich, indem man n mal Null addiert, hat allerdings den Nachteil, daß sehr viel Performanz verloren geht. Es werden pro Multiplikation nicht mehr n Takte benötigt, sondern $2n$, d.h. die Geschwindigkeit verlangsamt sich um den Faktor zwei.

3.2.1 Modulo-Multiplikation

Die Modulo-Multiplikation von ganzen Zahlen ($x \cdot y \bmod k$) ist im allgemeinen im Gegensatz zur normalen Multiplikation sehr viel rechenaufwendiger. Für die Modulo-Multiplikation muß das Ergebnis der normalen Multiplikation in den Zahlenbereich $[0, k - 1]$ überführt werden. Dieses kann durch geeignet häufiges Subtrahieren des Modulus bzw. Division durch den Modulus erfolgen. Beides erfordert zunächst einen Größenvergleich zwischen Modulus und Ergebnis.

Um die Modulo-Reduktion durchzuführen, wird das Ergebnis der Multiplikation durch Subtraktion eines geeigneten Vielfachen von k in den Bereich $[0, k\text{-}1]$ gebracht. Das Endergebnis entspricht dem Rest einer Division des Produktes $x \cdot y$ durch den Modulus k.

$$x \cdot y \bmod k = x \cdot y - a \cdot k, \quad a = \left\lfloor \frac{x \cdot y}{k} \right\rfloor.$$

Sowohl eine Subtraktion als auch eine Division erfordert einen Größenvergleich zwischen Modulus und Ergebnis. Wenn dieser Vergleich exakt durchgeführt werden soll, hat dieses, aufgrund der großen Zahlen, die verglichen werden müssen, einen erheblichen Hardwareaufwand zur Folge. Eine Möglichkeit, diesen Aufwand zu verringern, stellt die Quotientenschätzung dar. Es werden hierbei nur die höchstwertigen Stellen verglichen. Das Ergebnis muß allerdings am Ende gegebenenfalls noch korrigiert werden. Elementar ist hier analog zur Multiplikation die effiziente Implementierung der Addition, da sich die Subtraktion leicht auf sie zurückführen läßt.

Zur Verdeutlichung betrachten wir das folgende Beispiel. Es soll die Berechnung

$$f(z) = z \bmod k$$

durchgeführt werden. Wir wählen $z = 217$ und $k = 10$ und berechnen

$$a = \left\lfloor \frac{z}{k} \right\rfloor = \frac{217}{10} = 21$$

und erhalten so schließlich

$$f(z) = z - a \cdot k = 217 - 21 \cdot 10 = 7.$$

Wir können aber auch, um die Division zu beschleunigen, nur die vier höchstwertigen Bits von $z = 217$ („11011001") betrachten und deshalb mit $z' = 208$ („11010000") die Berechnung durchführen. Es ergibt sich so

$$a' = \left\lfloor \frac{z'}{k} \right\rfloor = \frac{208}{10} = 20$$

und

$$f'(z) = z - a' \cdot k = 217 - 20 \cdot 10 = 17.$$

Der so entstandene Fehler muß durch eine weitere Subtraktion korrigiert werden:

$$f(z) = f'(z) - k = 17 - 10 = 17.$$

3.3 Addition

Die Grundlage jeglicher Implementation der Multiplikation stellt die Addition dar. Es erscheint deshalb sinnvoll, mit der Verbesserung der Effizienz hier anzusetzen. Um schnelle Addierer zu konstruieren, muß in die Komplexität der Schaltung oder in die der Zahlendarstellung investiert werden. Diese beiden Optionen schließen einander natürlich nicht aus.

Die sich stellende Aufgabe ist es, die Summe zweier Binärzahlen $a = a_{n-1} \ldots a_0$ und $b = b_{n-1} \ldots b_0$ zu berechnen. Im üblichen Additionsverfahren werden beide Summanden von rechts her abgearbeitet. Man addiert in jeder Komponente modulo zwei, wobei der Übertrag

(Carry) c_i von der vorigen Stelle mit einbezogen wird, und berechnet den nächsten Übertrag. Für die Überträge gilt

$$c_{-1} = 0$$
$$c_i = c_{i-1}a_i \vee c_{i-1}b_i \vee a_ib_i$$

und für die Koeffizienten der Summe $s = a + b$

$$s_i = a_i \oplus c_{i-1} \oplus b_i$$

Ein Volladdierer berechnet s_i und c_i aus a_i, b_i und c_{i-1}. Aus n Volladdierern kann ein Addierwerk für n-stellige Zahlen gebaut werden. Die n Bitadditionen müssen seriell durchgeführt werden, da immer auf den Übertrag von der vorigen Stelle gewartet werden muß. Diese konventionellen „Ripple-Carry"-Addierer benötigen sehr lange Signallaufzeiten. Aus diesem Grunde können sie nur relativ niedrig getaktet werden. Sie sind daher zu langsam für die Anforderungen, die in kryptographischen Anwendungen auftreten [GOL94].

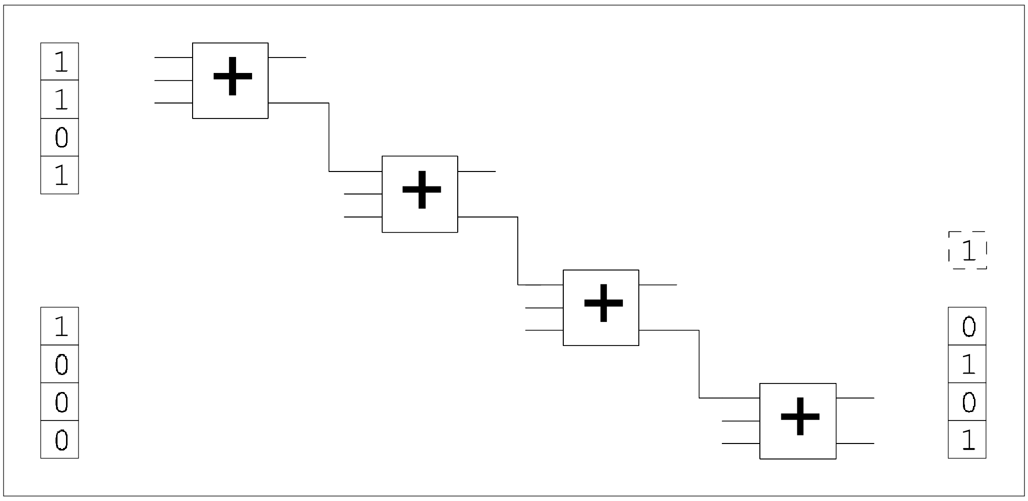

Abbildung 3.4: „Prinzip 'Ripple-Carry'"

Um einen brauchbaren Addierer zu realisieren, gibt es verschiedene Konzepte (siehe hierzu [SPA76]). Zwei wesentliche Ansätze stellen der „Carry-Lock-Ahead"- und der „Carry-Save"-Addierer dar.

In einem „Carry-Lock-Ahead"-Addierer wird der Übertrag c_{i+1} direkt aus den Eingaben a_j, b_j, $0 \le j \le i$ berechnet. Zur Herleitung von c_{i+1} und auch als Vorschlag zur Implementierung in Hardware werden die Hilfsvariablen g_i (generate) und p_i (propagate) wie folgt eingeführt:

$$g_i = a_ib_i$$
$$p_i = a_i \oplus b_i$$
$$s_i = p_i \oplus c_i$$
$$c_{i+1} = g_i \vee p_ic_i$$

Daraus erhält man:

$c_0 \quad = 0$

$c_{i+1} \quad = g_i \vee p_i c_i$

$\quad = g_i \vee p_i(g_{i-1} \vee p_{i-1}c_{i-1})$

$\quad = g_i \vee p_i g_{i-1} \vee p_i p_{i-1} c_{i-1}$

$\quad = g_i \vee p_i g_{i-1} \vee p_i p_{i-1} g_{i-2} \vee \ldots \vee p_i \cdots p_1 g_0 \vee p_i \cdots p_0 c_0$

Die Komplexität der Schaltung steigt quadratisch mit der Weite des „Look-Ahead“. Eine „Carry Lock Ahead“-Stufe über eine Wortbreite von 1024 Bit kommt daher nicht in Frage. Man kommt aber bereits zu günstigen Ergebnissen, wenn man kürzere Stufen kaskadiert. Es ist ebenfalls denkbar, ein „Carry Lock Ahead“-Addierer in mehreren Stufen aufzubauen.

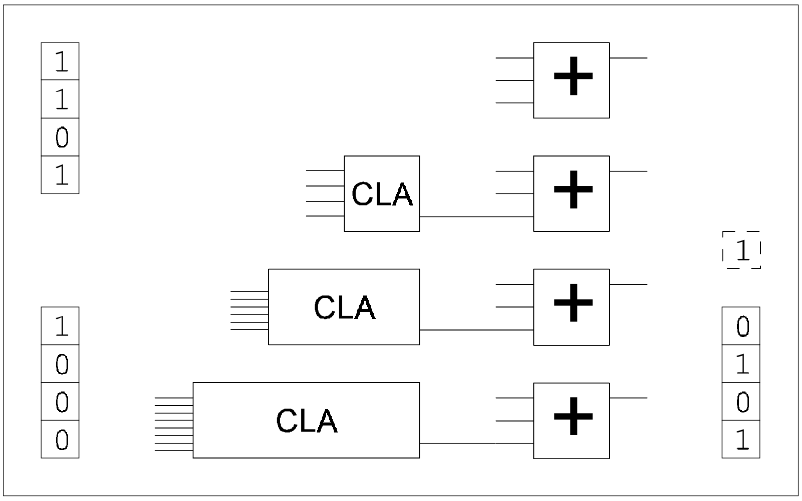

Abbildung 3.5: „Prinzip ‘Carry-Look-Ahead’“

Redundante Zahlendarstellungen sind der zweite Weg zu schnellen Additionsverfahren. In diese Gattung gehört der „Carry-Save“-Addierer. Die „Carry-Save“-Darstellung einer Zahl B der Länge n besteht aus zwei n-stelligen Binärvektoren, einem Summenvektor b und einem Übertragsvektor β. Der Wert von B ist:

$$B = \sum_{i=0}^{n-1} (b_i + \beta_i) 2^i \ .$$

Zu B kann eine Binärzahl A übertragsfrei addiert werden. Das Ergebnis $D = (d, \delta)$ ist wieder in „Carry-Save“-Darstellung gegeben. Man berechnet

$$d_i = a_i \oplus b_i \oplus \beta_i$$
$$\delta_{i+1} = a_i b_i \vee (a_i \oplus b_i)\beta_i$$

Auf diese Weise wird die Summe von drei Binärzahlen als Summe von zwei Binärzahlen ausgedrückt.

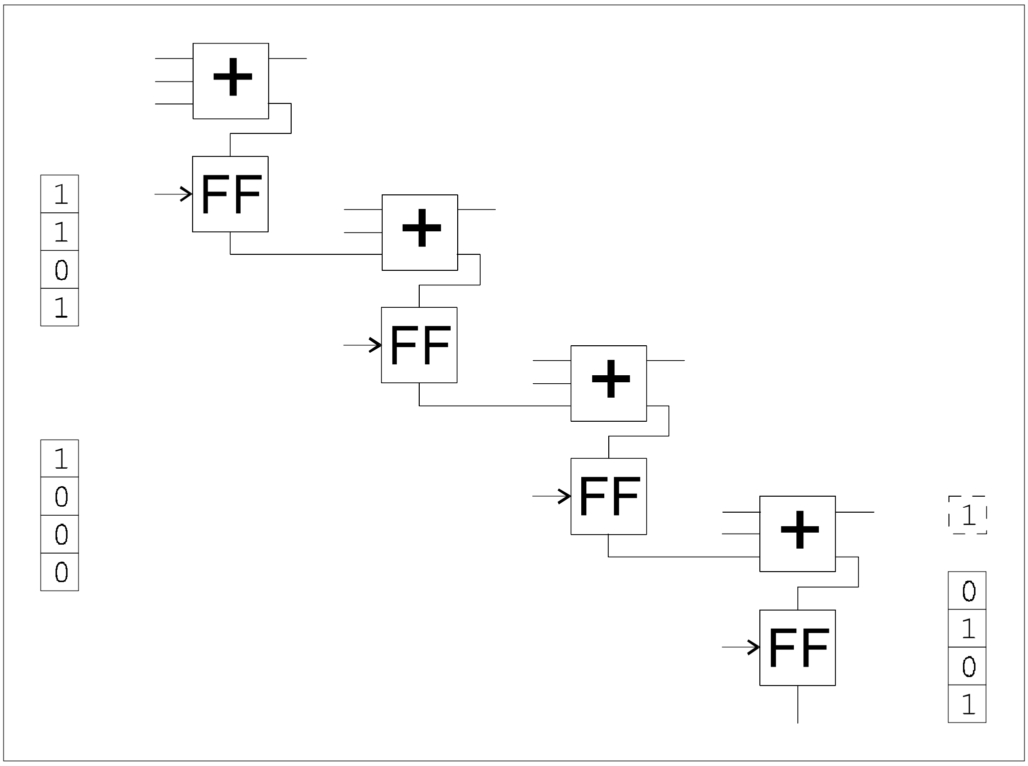

Abbildung 3.6: „Prinzip ‘Carry-Save’“

3.4 Reduktion

Eine Modulo-Multiplikation kann nach der zuvor beschriebenen Methode mit aufeinander folgenden Additionen und Schiebeoperationen ausgeführt werden. Nach jeder Addition kann dann modulo k reduziert werden. In vielen Fällen versucht man, das Zwischenergebnis im Intervall $[0, k\text{-}1]$ zu halten, wobei die Reduktion als Subtraktion implementiert werden kann.

Werden Zwischenergebnisse zugelassen, die länger als der Modulus sind, muß ein günstiger Mittelwert zwischen der Zahl der Reduktionsschritte und der Länge des „Überlaufbereichs“ gefunden werden. Zusätzlich kann der Vergleich von Zwischenresultat und Modulus auf verschiedene Arten vereinfacht werden [GOL94].

Bei der Reduktion nach Brickell [BRI82] wird nicht nach jeder Addition reduziert. Man legt daher einen Überlaufbereich der Länge λ fest. Für ein Zwischenergebnis d gelte nun $d < 2^{n+\lambda}$. Je nachdem, wie die Kosten von Zeit und Speicherplatz bewertet werden, wird λ größer oder kleiner gewählt. Um zu entscheiden, welches Vielfache von k von d abgezogen werden soll, vergleicht man die obersten vier Bit von d und von $K := 2n - k$.

Beim Verfahren von Sedlak [SED87] wird versucht, die Reduktion an das Verfahren zur Multiplikation anzupassen. Es kommt ein modifiziertes serielles „Shift-and-Add“ Verfahren zum Einsatz, bei dem der Darstellungsbereich um einen 20-stelligen Buffer erweitert wird.

Von Walter [WAL91] kommt der Vorschlag, die Modulo-Multiplikation durch Skalierung des Modulus k zu beschleunigen. Das Problematische an der Modulo-Reduktion ist die

Division durch k. Walter schlägt deshalb vor, den Modulus k mit einem Faktor f zu multiplizieren (skalieren), so daß die höchstwertigen Bits k_{fix} des „skalierten" Modulus einen – immer wieder benutzten – festen Wert erhalten. Sinnvoll ist ein Wert $k_{\text{fix}} = 2^n - 1$. Dieses erleichtert die Division erheblich, da sie dann durch eine Schiebeoperation ersetzt werden kann. Es wird nur durch die höchstwertigen Bits geteilt. Der mögliche Fehler wird später korrigiert. Zwar wird der neue „skalierten" Modulus ein paar Bit größer, die Geschwindigkeit bei vergleichbarem Hardwarebedarf soll aber so um 70 Prozent schneller sein. Die Skalierung von k kann einmalig für jeden Modulus k per Software erfolgen.

3.4.1 Reduktion durch Approximation von k^{-1}

Es ist nicht unbedingt notwendig, die Multiplikation in Additionen und Schiebeoperationen aufzuteilen. Man kann mit einer genügend genauen Approximation des Inversen von k multiplizieren und anschließend die überflüssigen Stellen abschneiden. Es gelte

$$a \cdot b = q \cdot k + r, \; 0 \le r < k$$

und

$$a \cdot b \bmod k = r$$

sowie

$$2^n > k \text{ (n ist die Stellenzahl von k).}$$

Im Verfahren von Norris und Simmons [NOR81] berechnet man zuerst

$$k^{-1} = \left\lceil \frac{2^{3n}}{k} \right\rceil \cdot 2^{-3n}$$

$$q = \left\lfloor a \cdot b \cdot k^{-1} \right\rfloor .$$

k^{-1} ist die Gleitkommazahl $1 / k$, aufgerundet an der $3n$-ten Stelle nach dem Komma. Für diese Zahl gilt

$$k^{-1} = \frac{1}{k} + \varepsilon, \quad 0 \le \varepsilon \le \frac{1}{2^{3n}} .$$

Dann folgt aus

$$a \cdot b \cdot k^{-1} = (q \cdot k + r) \cdot \frac{1}{k} + a \cdot b \cdot \varepsilon$$

und

$$a \cdot b \cdot \varepsilon \le \frac{2^n \cdot 2^n}{2^{3n}} = \frac{1}{2^n} < \frac{1}{k} ,$$

schließlich

$$\frac{r}{k} + a \cdot b \cdot \varepsilon < \frac{k-1}{k} + \frac{1}{k} = 1.$$

Daher wird q korrekt berechnet. Den Rest r erhält man aus

$$r = a \cdot b - q \cdot k.$$

In diesem Verfahren wurde $1 / k$ approximiert. Arbeitet man statt dessen mit a / k, wie etwa beim seriellen „Square-and-Multiply“, dann muß die Approximation nur auf $2n$ Stellen genau sein [GOL94].

3.5 Reduktion nach Montgomery

Ein völlig anderer Ansatz zur Verringerung des Aufwandes zur Durchführung von Modulo-Multiplikationen stammt von Montgomery [MON85]. Durch eine geeignete Abbildung der beiden zu multiplizierenden Operanden in die sogenannte Montgomery-Darstellung wird die Division durch den Modulus durch eine Division durch eine relativ frei wählbare Zahl ersetzt. Wenn diese Zahl eine Potenz der Zahlenbasis ist, in der die Operanden dargestellt sind, kann die Division durch eine Rechtsverschiebung durchgeführt werden.

3.5.1 Algorithmus Montgomery

Es sei die Modulo-Multiplikation

$$z = x \cdot y \bmod k$$

zu berechnen. Dazu sind die folgenden Schritte auszuführen:

(1) Es wird eine Zahl r ausgewählt. Um die Montgomery-Methode durchführen zu können, muß r größer als k sein. Außerdem muß r relativ prim zu k sein. Dies ist notwendig, da sonst zu r kein inverses Element r^{-1} existiert und die Abbildung von x nach x' somit nicht umkehrbar ist. Beim RSA-Verfahren ist k in jedem Fall ungerade, so daß r eine beliebige Zweierpotenz ($r = 2^n > k$) sein kann. Wenn für r eine Zweierpotenz gewählt wurde, ist die Division durch r besonders einfach durch Rechtsschieben durchführbar.

(2) Als nächstes muß die Zahl k' berechnet werden. Dieses geschieht mit der folgenden Formel. Der Faktor i ist so zu wählen, daß k' ganzzahlig wird.

$$k' = \frac{-1}{k} \bmod r = \frac{i \cdot r - 1}{k}, \quad k' \in \mathrm{N}$$

Zur Erläuterung folgen zwei Beispiele (Es wurde jeweils der kleinste Wert für i gewählt.):

1. $k = 13$, $r = 100 \Rightarrow k' = 23$, $i = 3$
2. $k = 13$, $r = 128 \Rightarrow k' = 59$, $i = 6$

(3) Der Wert r^{-1} wird mit Hilfe der folgenden Beziehung berechnet:

$$r \cdot r^{-1} - k \cdot k' = 1$$

$$\Rightarrow r^{-1} = \frac{k \cdot k' + 1}{r}.$$

Wieder zwei Beispiele zur Verdeutlichung:

1. $k = 13$, $r = 100$, $k' = 23 \Rightarrow r^{-1} = 3$
2. $k = 13$, $r = 128$, $k' = 59 \Rightarrow r^{-1} = 6$

r^{-1} ist das inverse Element zu r, da

$$(r \cdot r^{-1}) \bmod k = 1.$$

Beweis:

$$(r \cdot r^{-1}) \bmod k = (k \cdot k' + 1) \bmod k = 1$$

(4) Es wird x und y in die Montgomery-Darstellung abgebildet, die durch $x' = xr \bmod k$ und $y' = yr \bmod k$ definiert ist.

(5) Das Montgomery-Produkt $z' = MP(x', y', k, r) = x'y'r^{-1} \bmod k$ wird gebildet.

(6) Es wird das Ergebnis z' nach z, d.h. $z = z'r^{-1} \bmod k$, abgebildet.

Der Vorteil dieser auf den ersten Blick sehr umständlich erscheinenden Methode resultiert aus der einfachen Berechnungsmöglichkeit der Funktion *MP*. Die sehr aufwendige Modulo-Multiplikation wird durch zwei „normale“ Multiplikationen, eine Addition und zwei Schiebeoperationen[9] ersetzt. Die Berechnung der Konstanten r^{-1} und k' muß nur einmal erfolgen, da der Modulus k einen festen Wert hat.

[9] Eine Division durch r (= 2^n) läßt sich effizient durch „schieben nach rechts“ ausführen.

Der grundsätzliche Algorithmus zur Berechnung des Montgomery-Produktes in C++ lautet (k, k' und r sind Konstanten.):

```
int mp1(int x,int y)
{
  int p,v,t;
  p = x * y;
  v = (p * k') % r;
  t = (p + v*k) / r;
  return(t);
}
```

(Anmerkung: Man kann diesen Algorithmus auch benutzen, um eine Zahl a in die Montgomery-Darstellung zu transferieren. Hierzu benötigt man eine weitere Konstante $r^2 = (r \cdot r) \bmod k$. Man setzt dann $x = a$ und $y = r^2$.)

Das Verfahren nach Montgomery wird mit dem folgenden Beispiel verdeutlicht. Als gegeben nehmen wir die Zahlen $x = 5$ und $y = 8$. Der Moduls sei $k = 13$.

(1) Es wird zunächst $r = 2^7 = 128$ (128 > 13) gewählt

(2) und $k' = \frac{-1}{k} \bmod r = \frac{i \cdot r - 1}{k} = \frac{6 \cdot 128 - 1}{13} = 59$

(3) sowie $r^{-1} = \frac{k \cdot k' + 1}{r} = \frac{13 \cdot 59 + 1}{128} = 6$ berechnet.

(4) $x' = (x \cdot r) \bmod k = (5 \cdot 128) \bmod 13 = 3$
$y' = (y \cdot r) \bmod k = (8 \cdot 128) \bmod 13 = 10$

(5) $p = 3 \cdot 10 = 30$
$v = (30 \cdot 59) \bmod 128 = 106$
$z' = t = (30 + 106 \cdot 13) / 128 = 11$

(6) $z = (z' \cdot r^{-1}) \bmod k = (11 \cdot 6) \bmod 13 = 1$

Die Probe ergibt: $z = (x \cdot y) \bmod k = (5 \cdot 8) \bmod 13 = 1$.

3.5.2 Leistungsvergleich

In [BOS93] werden drei Methoden zur Modulo-Reduktion miteinander verglichen: Der „klassische“ Algorithmus sowie die Verfahren nach Barrett [BAR86] und Montgomery [MON85]. Alle drei Algorithmen wurden als C-Programm implementiert. Das Verfahren nach Montgomery erwies sich als das leistungsfähigste bei einer Blocklänge von 1024 Bit (siehe Abbildung 3.7: „Modulo-Reduktion [PC 386 / 33 MHz]“). Die Ausführungszeit läßt sich auf etwa 88 Prozent der Zeit der klassischen Methode reduzieren.

	Ausführungszeit (1024 Bit)
„klassisch“	21,74 s
Barrett	20,30 s
Montgomery	19,14 s

Abbildung 3.7: „Modulo-Reduktion [PC 386 / 33 MHz]“

3.6 Systolische Felder

Das Konzept der systolischen Felder (systolic arrays) wurde erstmals von H. T. Kung und C. E. Leierson [KUN78] Ende der siebziger Jahre an der Carnegie-Mellon-Universität entwickelt und seitdem in den unterschiedlichsten Anwendungsgebieten eingesetzt. In seinen Arbeiten zeigte Kung, daß solche Felder eine sehr effiziente Verwendung von Prozessoren zur Realisierung von kostengünstigen, hochleistungsfähigen Systemen in einem weiten Anwendungsfeld ermöglichen. Die Bezeichnung „systolisches Feld“ resultiert aus der Art des entstehenden Datenflusses, der an die Blutzirkulation vom Herzen durch die Körperzellen erinnert [SAU93].

Ein systolisches Feld ist eine matrixförmige Anordnung von Prozessoren (Zellen), die gemeinsam und synchron eine Aufgabe bearbeiten. Jede Zelle kommuniziert dabei nur mit ihren direkten Nachbarzellen. Bei jedem Schritt liest jede Zelle Daten von ihren Nachbarzellen ein, führt eine einfache Operation mit diesen aus und übermittelt das Ergebnis an ihre Nachbarzellen. Die Eingangsdaten werden am Rand des Feldes angelegt, die Ausgabedaten werden ebenfalls von dort gelesen. Die Reihenfolge der Bereitstellung von Eingabedaten und die Art der Zusammenschaltung der Zellen ist für die Funktion des Feldes von entscheidender Bedeutung. Je nach Art des zu lösenden Problems kommen unterschiedliche Architekturen zur Anwendung. Die Zellen können beispielsweise linear, orthogonal, hexagonal oder baumartig angeordnet werden. Dabei wird angestrebt, daß jeder Prozessor in jedem Schritt eine Operation ausführt und sich nicht in einem Zustand des „Abwartens“ befindet. Ein Algorithmus mit quadratischer Ordnung kann dann unter Benutzung einer linear wachsenden Anzahl von Prozessoren mit linearer Ordnung ausgeführt werden.

Das Konzept eignet sich aus folgenden Gründen besonders für die VLSI-Implementierung:

- Skalierbarkeit und Erweiterbarkeit: Durch die Verwendung identischer Zellen, die nur mit ihren nächsten Nachbarzellen verbunden sind, entsteht ein einfacher und regulärer Daten- und Kontrollfluß. Durch diese Regularität und die lokalen Verbindungsstrukturen ist eine leichte und modulare Erweiterbarkeit des Systems gegeben. Dieses ist besonders für den Full-Custom-Entwurf wichtig.
- Hoher Berechnungsdurchsatz bei geringer Ein- und Ausgabebandbreite: Es wird ein hoher Berechnungsdurchsatz erzielt, weil jedes Datenelement fließbandartig in mehreren aufeinanderfolgenden Zellen bearbeitet wird (pipelining). Durch die gleichzeitige Nutzung der vielen einfachen Zellen (concurrency) entsteht bei gleichem Ressourceneinsatz ein besserer Durchsatz als bei einer rein sequentiellen Nutzung einer leistungsfähigerer Zelle. Durch die mehrfache Verwendung jedes Eingabedatenelements entsteht auch ein geringerer Bedarf an Ein- und Ausgabebandbreite von und zu dem systolischen Feld.

- Niedrige Entwurfs- und Implementierungskosten: Um die Komplexität von Systemen mit mehreren hunderttausend Transistoren zu beherrschen, ist es notwendig, einfache, reguläre und modulare Entwürfe zu bevorzugen. Dabei sollten einzelne Teilsysteme bzw. Blöcke möglichst oft eingesetzt werden. Genau diese Anforderungen werden von systolischen Feldern hervorragend erfüllt, weil sie aus vielen gleichen, einfachen Zellen bestehen. Sie haben somit eine reguläre Struktur. Eine Zelle kann optimiert und dann beliebig vervielfältigt werden. Damit werden Entwurfs- und Implementierungskosten niedrig gehalten. Dies ist insbesondere für den Full-Custom-Entwurf eine notwendige Bedingung.
- Geringer Flächenbedarf und hohe Taktrate: Die Verbindungs- und Kommunikations-Strukturen in massiv-parallelen Systemen haben einen starken Einfluß auf den resultierenden Flächen- und Leistungsverbrauch und die Arbeitsgeschwindigkeit des Systems. In systolischen Feldern ist der Daten- und Kontrollfluß einfach und regulär. Die einzige globale Verbindung der Zellen untereinander stellt der Takt dar. Eine sonstige globale Verteilung von Signalen ist nicht notwendig, was zu einer geringen Belastung von Eingangs- und Ausgangsleitungen und zu kurzen Signallaufzeiten führt. Dadurch kann das System schnell getaktet werden. Ein möglicher Taktversatz ist bei eindimensionalen systolischen Feldern unkritisch, auch wenn bidirektionaler Datenfluß stattfindet. Die oben genannten Eigenschaften ermöglichen außerdem eine einfache und platzsparende Plazierung und Verdrahtung des Systems.

Das Ziel muß es deshalb sein, die Algorithmen für die Multiplikation und die Modulo-Reduktion auf systolische Strukturen abzubilden.

4 Lösungsansätze

4.1 Allgemeine Abschätzungen

Um die Probleme und Schwierigkeiten einer Implementation erfassen zu können, wurde zunächst eine allgemeine Abschätzung erstellt. Als erstes wurden die Anforderungen an die Geschwindigkeit abgeschätzt. Hierbei wurde davon ausgegangen, daß die Funktion

$$f(x) = x^y \bmod k$$

mit der Vorgabe $x, y, k < 2^{1024}$ zu berechnen ist, d.h. alle Zahlen sind maximal 1024 Bit groß. Weiterhin wurde angenommen, daß im schlechtesten Fall für die Exponentiation x^y mit Hilfe des „seriellen Binärverfahren“ (siehe hierzu Kapitel 3.1.1 Die Binärmethode (Square-and-Multiply) auf Seite 27) $2 \cdot \lceil \mathrm{ld}(y) \rceil = 2048$ Modulo-Multiplikationen nötig sind. Dieser schlechteste Fall tritt ein, wenn $y = 2^{1024} - 1$ (y besteht in der Binärdarstellung nur aus Einsen). In jedem Schleifendurchlauf des „Binärverfahren“ sind in diesem Fall zwei Multiplikationen[10] nötig. Im besten Fall besteht der Exponent y nur aus Nullen[11] und es ist nur eine Multiplikation pro Schleifendurchlauf nötig, insgesamt also 1024 (Die Schleifenvariable i durchläuft die Werte von 1023 bis 0.). Im Mittel müssen 1024 + 512 = 1536 Multiplikationen durchgeführt werden, wenn man annimmt, daß der Exponent y im Durchschnitt je zur Hälfte aus Nullen und Einsen besteht. Hintergrund dieser Überlegung ist es, daß in jedem Durchgang der Schleife immer eine Multiplikation für das Quadrieren anfällt, also immer 1024. Eine zweite Multiplikation fällt nur dann an, wenn im Exponent y eine Eins steht.

Für jede Modulo-Multiplikation werden maximal 1024 Additionen und 1024 Subtraktionen für die Reduktion benötigt. Es wird hier zur Vereinfachung angenommen, daß eine Subtraktion nicht aufwendiger als eine Addition ist. Somit besteht eine Modulo-Multiplikation maximal aus 2048 Additionen. Für die Exponentiation werden somit für den schlechtesten Fall $2048 \cdot 2048 \approx 4{,}2 \cdot 10^6$ Additionen benötigt und im Mittel $2048 \cdot 1536 \approx 3{,}1 \cdot 10^6$ Additionen (Anzahl Multiplikationen x Anzahl Additionen).

Mit jeder Exponentiation werden 1024 Bit = 1 KBit verschlüsselt. Bei einer Taktfrequenz von 12 MHz werden $12 \cdot 10^6$ Operationen pro Sekunde ausgeführt. Unter der Voraussetzung, daß es gelingt, eine Addition pro Takt auszuführen, ergibt sich für den schlechtesten Fall die folgende Datenrate:

[10] Eigentlich sind es eine Multiplikation und eine Quadrierung. Hier und im folgenden werden für diesen Fall immer zwei Multiplikationen angenommen.

[11] Dies ist selbstverständlich in der Praxis kein sinnvoller Exponent. Er dient nur der theoretischen Abschätzung.

$$DR = \frac{1\,\text{KBit} \cdot 12 \cdot 10^6\,\frac{1}{s}}{4{,}2 \cdot 10^6} = 2{,}9\,{}^{\text{KBit}}\!/_{s}\,.$$

Im Mittel ergibt sich eine Datenrate von 3,9 KBit/s. Will man beispielsweise eine Datenrate von 64 KBit/s erreichen, darf jede Exponentiation nur etwa $0{,}2 \cdot 10^6$ Takte benötigen. Es müßten somit 21 Additionen parallel ausgeführt werden.

Höhere Geschwindigkeit bedeutet immer auch einen größeren Platzbedarf. Es gibt viele Methoden die Exponentiation schnell auszuführen, wie z.B. die Exponentiation mit Hilfe von Additionsketten oder das parallele Ausführen möglichst vieler Additionen. Alle diese Verfahren benötigen aber besonders viel Chipfläche, da entweder weitere Register oder zusätzliche Addierer benötigt werden. Insbesondere beim „Standardzellen“ Entwurf ist die zur Verfügung stehende Fläche kostbar, wie im folgenden noch erläutert wird.

Mit Hilfe der zuvor ermittelten Werte wird die Schwierigkeit der Aufgabenstellung deutlich. Schon die Addition zweier Zahlen in der Größenordnung von 1024 Bit in einem Takt ist nur unter idealen Bedingungen zu erreichen. 21 Addition, selbst wenn sie datenunabhängig von einander sind, parallel auf einem Chip auszuführen, erscheint demzufolge utopisch. Anzustreben ist deshalb, die Anzahl der benötigten Additionen zu reduzieren.

Die Verschlüsselung erfolgt wesentlich schneller, wenn man den Wert des öffentlichen Schlüssels *e* geschickt wählt. Die gebräuchlichsten Werte sind 3, 17 und 65537 ($= 2^{16} + 1$). Da die Binärdarstellung von 65537 nur zwei Einsen enthält, sind für die Potenzierung nur 17 Multiplikationen nötig. Die Wahl eines dieser drei Werte für *e* stellt kein Sicherheitsproblem dar, wenn man die Nachricht mit zufälligen Werten auffüllt. Das gilt selbst dann, wenn eine Gruppe von Benutzern den gleichen Wert für *e* verwendet [SCH96].

4.2 Software vs. Hardware

4.2.1 Softwarelösung

Es wurde zunächst ein C++ Programm erstellt, um den grundsätzlichen Algorithmus zur Verschlüsselung nach dem RSA-Verfahren zu testen. C++ ermöglicht die Verwendung von beliebig großen Zahlen mit Hilfe der Klasse „Integer“. Die Grenze setzt nur der zur Verfügung stehende Hauptspeicher. Weiterhin dient das Programm später zur Überprüfung der in VHDL realisierten Algorithmen. Mit Hilfe des C++ Programms können sowohl die Ausgaben der Simulation als auch die der zu entwickelnden Hardware überprüft werden.

```
/* a = x^y mod k */

a = 1;
while (y != 0)
{
  if (y%2 == 1)
  {
    y--;
    a = (a * x) % k;
  }
  y /= 2;
  x = (x * x) % k;
}
```

Das Programm wurde mit dem GNU-C++-Compiler unter Linux übersetzt. Es wurde eine Nachricht mit einem Schlüssel der Länge 1024 Bit verschlüsselt. Die Laufzeit auf einem Rechner mit einem Pentium-Prozessor (100 MHz) betrug im Mittel etwas weniger als 6 Sekunden. Dies entspricht einer Verschlüsselungsrate von 171 Bit/s. Die Verschlüsselungsrate läßt sich durch eine effiziente Implementierung in Assembler steigern. Die nach [SCH96] schnellste Implementierung einer Softwarelösung schafft eine 1024-Bit Verschlüsselung in 0,93 Sekunden auf einer SPARC II. Dieses kommt einer Datenrate von 1,1 KBit/s gleich.

4.2.2 Hardwarelösung

Eine Implementierung in Hardware ist aus unterschiedlichen Gründen erforderlich. Der wesentliche Grund liegt in der höheren Geschwindigkeit einer Hardwarelösung. Selbst im schlechtesten Fall (gemäß Abschätzung in „4.1 Allgemeine Abschätzungen“ auf Seite 45) erreicht sie eine Verschlüsselungsrate von 2,9 KBit/s. Dieses ist etwa 17 mal schneller als die zuvor beschriebene Softwarelösung. Weitere Vorteile einer Realisierung in Hardware sind:

- Der Einsatz in Bereichen, in denen kein Rechner (PC) zur Verfügung steht.
- Die Leistungsaufnahme gegenüber einem Rechner ist wesentlich geringer.
- Angriffe auf die Sicherheit des Systems sind sehr viel schwerer.

4.3 Praktische Randbedingungen

4.3.1 VLSI-Entwurf

In seiner eigentlichen Bedeutung trifft VLSI, d.h. „Very large scale integration“, auf Schaltungen mit mehr als 10^5 Gattern zu. Heute werden Entwürfe mit dem Namen „VLSI“ belegt, die weit außerhalb des Bereichs liegen, für den dieser Name ursprünglich definiert war. Solche Entwürfe wurden erst durch den Einsatz entsprechender CAD-Werkzeuge möglich. Daher hat sich der Begriff VLSI ganz allgemein als Bezeichnung für integrierte Schaltungen eingebürgert, die mit CAD-Werkzeugen entworfen werden. Durch solche Werkzeuge ist neben dem Entwurf von Massenprodukten wie Speichern und Mikroprozessoren auch der Entwurf von solchen Schaltungen wirtschaftlich geworden, die nur in kleiner Auflage hergestellt werden. Diese anwendungsspezifischen integrierten Schaltungen (ASICs) sind heute, insbesondere im Kommunikationsbereich, ein beliebtes Medium für die

Implementierung von Algorithmen. In diesem Sinn ist im folgenden unter VLSI-Entwurf meistens ASIC-Entwurf zu verstehen [GOL94].

Auf den Algorithmenentwurf ist es nicht ohne Einfluß, welche physikalischen Strukturen in der gegebenen Technologie überhaupt möglich sind. Die heute gebräuchlichste Technologie im ASIC-Entwurf ist CMOS[12]. Die wesentlichen Schaltelemente sind dort Feldeffekt-Transistoren. Zur Verdrahtung und zur Stromversorgung stehen Polysiliziumschichten (meist eine) und Metallschichten (mindestens zwei) zur Verfügung.

Heute (1997) sind im ASIC-Entwurf 0,35µ bis 2µ CMOS-Prozesse üblich.

4.3.2 Standardzell- oder Full-Custom-Entwurf?

Zwei unterschiedliche Entwurfsmethoden bieten sich für die Entwicklung des hier zu entwickelnden Hardwareakzelerators an: Der Entwurf mit Standardzellen oder Full-Custom-Entwurf.

Beim Entwurf mit Hilfe von Standardzellen arbeitet der Entwickler mit einer Bibliothek von Standardzellen und kann diese Zellen nach eigener Wahl auf dem Chip plazieren, das Layout der Zellen selbst bleibt unverändert. Meist werden die Zellen in Zeilen angeordnet, die durch Verbindungskanäle getrennt sind. Das Konzept der Standardzellen wurde auf parametrisierbare Zellen (Makros), z.B. RAMs, ROMs, PLAs erweitert. Es existieren auch Generatoren für komplexere Schaltungen. Entwurfsstrategien, die auf Zellen basieren, werden auch beim Gate-Array-Entwurf eingesetzt. Der wesentliche Unterschied liegt im Ausmaß der Optimierung der einzelnen Zellen. Es können dabei Optimalitätskriterien für Fläche und Verbindungseigenschaften berücksichtigt werden. Die Transparenz von Zellen fällt etwa unter den zweiten Gesichtspunkt.

Beim Full-Custom-Entwurf kann der Entwickler die Zellen selbst auf die jeweilige Anwendung zuschneiden. Auch dabei gibt es noch verschiedene Freiheitsgrade. In manchen Systemen ist man auf eine vorgegebene Menge elementarer Artefakte eingeschränkt (z.B. Transistoren, Kontakte), andere gestatten es, mit beliebigen Polygonen auf den einzelnen physischen Ebenen zu arbeiten. Dementsprechend werden manche dieser Entwurfsverfahren vorzugsweise Semi-Full-Custom-Methoden genannt. Darüber hinaus ist es selten der Fall, daß eine Schaltung vollständig von Hand entworfen wird. Eher werden gewisse essentielle Teile als Full-Custom Layout entworfen, während der Rest mit Hilfe von Generatoren oder Standardzellen implementiert wird. Ein kompletter Full-Custom-Entwurf bietet sich erst für echte Massenprodukte an, die durch ihre hohen Stückzahlen die höheren Entwurfskosten rechtfertigen.

[12] „complementary metal oxide semiconductor“ oder auch „complementary metal oxide silicon“ (CMOS)

Folgende Vor- und Nachteile haben Standardzellen und Full-Custom:

	Vorteile	**Nachteile**
Standardzellen	1. Der Entwurf wird gut durch automatische CAD-Werkzeuge unterstützt. 2. Wegen des Gebrauchs bewährter Strukturen ist es wahrscheinlich, daß die Ergebnisse der Simulation das tatsächliche Verhalten gut vorhersagen.	1. Fläche des Chips wird verschwendet, weil der Entwurf durch die gegebene Bibliothek und starre Plazierungsverfahren beschränkt ist. 2. Keine oder unzureichende Unterstützung bzw. Ausnutzung regulärer Strukturen.
Full-Custom	Beim Entwurf kann man Zellen und Geometrien wählen, die für die Aufgabe am besten geeignet sind. Optimale Geschwindigkeit und Flächennutzung werden so ermöglicht.	Langer Entwurfszyklus, weil mehr Entwurfsschritte durchlaufen werden. Dadurch muß die Schaltung auf mehreren Beschreibungsebenen verifiziert werden.

Abbildung 4.1: „Vor- und Nachteile von Standardzellen und Full-Custom“

Bei der Realisierung des RSA-Verfahrens bietet der Full-Custom-Entwurf die Chance, den Ansprüchen in bezug auf Chipfläche und hoher Geschwindigkeit zu genügen. Die Nachteile verhindern aber einen Full-Custom-Entwurf im Rahmen dieser Diplomarbeit, da dieser nicht von einer Person innerhalb eines Jahres zu leisten ist. Es wurde deshalb in dieser Arbeit versucht, die Vorgaben mit Hilfe des Entwurfs mit Standardzellen zu erreichen. Gleichzeitig werden die Einsparungen in bezug auf die Chipfläche, die ein Full-Custom-Entwurf ermöglichen könnte, aufgezeigt (siehe Kapitel „6.2.2 Fläche“ auf Seite 99). Eine genaue Vorhersage der möglichen Einsparungen an Chipfläche läßt sich allerdings nicht machen, da Verbesserungen im bezug auf die Fläche im wesentlichen vom jeweiligen Problem abhängen.

4.3.3 Vorgehen VHDL-Entwurf

Es ist unmöglich, eine integrierte Schaltung, die von einer Problemstellung ausgeht, direkt in einem Schritt zu entwickeln. Deshalb erscheint es sinnvoll, nach der sogenannten „Top-Down-Methode“ vorzugehen. Hierbei geschieht der Entwurf in mehreren Stufen, die systematisch von einer sehr abstrakten Beschreibung zu einer konkreten Struktur führen. Da die Realisierung aber auch die verfügbaren technischen Möglichkeiten berücksichtigen muß, spielen auch sogenannte „Bottom-Up-Einflüsse“ eine wesentliche Rolle. Im Bereich der Halbleitertechnologie und bei der Realisierung darauf basierender Grundschaltungen wird daher „Bottom-Up“ entwickelt. Der RSA-Chip wird zunächst mit einem Standardzell-Entwurfssystem als ASIC realisiert. Der Halbleiterhersteller hat beim Entwurf seiner Standardzellen bereits die „Bottom-Up-Einflüsse“ berücksichtigt, so daß der Anwender des Entwurfssystems seine Schaltung hierarchisch von einer hohen Abstraktionsebene her anfangend nach der „Top-Down-Methode“ entwickeln kann.

Der eigentliche Algorithmus ist in einer funktionalen Beschreibung in einer Hardwarebeschreibungssprache gegeben. Damit sind das Verhalten und die funktionale Grobstruktur festgelegt. Im Syntheseschritt soll die funktionale Struktur soweit verfeinert werden, daß ein einfacher Übergang zur geometrischen Beschreibung erfolgen kann. In der Regel baut man dabei auf einer Bibliothek von Grundmodulen auf, deren Layout bereits vorhanden ist (z.B. Standardzellen). Damit ist auch die Menge der möglichen Hardwarekomponenten bestimmt, während die Auswahl und Zahl der Komponenten noch variabel ist [GOL94].

Zur Entwicklung des RSA-Chips wurde die Hardwarebeschreibungssprache VHDL[13] ausgewählt. VHDL wurde 1983 vom amerikanischen „Department of Defence" initiiert und ist seit Ende 1987 als IEEE Standard 1076 genormt. Inzwischen ist VHDL quasi der Standard der Hardwarebeschreibungssprachen. Die Sprache VHDL dient der Beschreibung und Simulation digitaler Systeme (ASICs) und deren Umgebung. Alle während des Entwurfsvorgangs anfallenden Beschreibungen der Schaltung werden vom Sprachumfang abgedeckt.

VHDL hat eine Syntax, die mit der einer prozeduralen Programmiersprache vergleichbar ist. VHDL unterstützende Simulations- und Synthese-Werkzeuge ermöglichen es, einen Großteil der Schritte von der Spezifikation bis zum Layout eines Chips nahezu nahtlos aneinanderzufügen. In VHDL kann sowohl losgelöst von einer konkreten Struktur Verhalten beschrieben werden als auch eine Struktur detailliert beschrieben werden.

Der eigentliche „Design-Prozeß" bei der Verwendung von VHDL entspricht einem „Top-Down" Vorgehen. Beim Entwurf der Algorithmen ist der Ausgangspunkt die Verhaltensbeschreibung der Schaltung, deren Funktionalität durch Simulation geprüft wird. So können verschiedene Algorithmen implementiert und miteinander verglichen werden. Auf oberster Ebene ist dies eine Beschreibung des zu entwerfenden ICs oder Systems sowie eine Testumgebung, die das Interface zur Schaltung darstellt.

Die „Top-Down" Strukturierung erhält man, indem man im weiteren Vorgehen die Schaltung in Funktionsblöcke gliedert, bis man letztlich eine Strukturbeschreibung erhält. Dieser Prozeß – algorithmischer Entwurf von Funktionseinheiten, hierarchische Verfeinerung und Umsetzung in Strukturbeschreibungen – wird rekursiv ausgeführt, bis man schließlich bei Elementen einer Zellbibliothek angekommen ist, und die Schaltung praktisch realisiert werden kann. Dadurch, daß die Schnittstelle der Schaltung und deren eigentliche Realisierung voneinander getrennt sind, werden Alternativen im Entwurf unterstützt (exploring the design-space). Durch den Einsatz von Synthesewerkzeugen wird die Entwurfsaufgabe auf den unteren Abstraktionsebenen dabei zunehmend vereinfacht. Ausgehend von Verhaltensbeschreibungen werden Netzlisten für Zielbibliotheken generiert. Der derzeitige Stand der Technik ist, daß die Synthese für Logik (Schaltnetze) und für endliche Automaten problemlos beherrscht wird. Für die Synthese komplexerer Algorithmen gibt es viele gute Ansätze, die zumindest bei Einschränkungen auf bestimmte Anwendungsfelder (Einschränkung des Suchraums) mit den Entwürfen guter Designer konkurrieren können [MAE93].

4.3.4 Hochsprachliche VHDL-Beschreibung

Um sowohl die Geschwindigkeit als auch den Platzbedarf abschätzen zu können, wurde der zuvor in C++ realisierte Algorithmus praktisch „Eins-zu-Eins" in VHDL umgesetzt. Aus dem VHDL-Code wurde mit dem „SYNOPSYS" Programmpaket der Universität direkt eine

[13] Very High Speed Integrated Circuits Hardware Description Language (VHDL)

„Netzliste“ synthetisiert. Das Ergebnis hängt somit wesentlich von der Leistungsfähigkeit der Synthesewerkzeuge ab. Mit anderen Worten: Es wurde vom Entwickler kein Einfluß auf die Wahl des Addiererprinzips genommen.

Diese Art der Implementierung wurde gewählt, um einen Vergleichswert in bezug auf Fläche und Geschwindigkeit zu erhalten. Als Ergebnis erhielt man so die in jedem Fall erreichbare Verschlüsselungsrate sowie die maximale Chipfläche. Die Chipfläche ist hier die Summe der Fläche für die verwendeten Standardzellen plus einem Schätzwert für den Flächenbedarf der Verdrahtung. Die tatsächliche Verdrahtung wird von einem weiteren Programmpaket („CADENCE DFW II“) vorgenommen. Auf den Einsatz wurde an dieser Stelle verzichtet, weil nur eine grobe Abschätzung nötig ist.

Die hochsprachliche VHDL-Beschreibung wurde mit einer RSA Blocklänge von 64 Bit vom Programmpaket „SYNOPSYS“ synthetisiert. Die Ergebnisse sind in der Abbildung 4.2: „Ergebnisse ‘hochsprachliche Beschreibung’ (64 Bit)“ zusammengefaßt. Es fällt die große Chipfläche und die niedrige Taktfrequenz auf. Dieses ist auf die von „SYNOPSYS“ standardmäßig verwendeten Komponenten für die Multiplikation und Modulo-Berechnung zurückzuführen. Gemeinsam belegten diese Komponenten mit 111 mm^2 den größten Teil der Chipfläche (64x64 Multiplikation: 39 mm^2, 128 auf 64-Bit Modulo-Berechnung: 72 mm^2).

Chipfläche	122,5 mm^2
Signalverzögerung „Längster Pfad“	9500 ns
maximale Taktfrequenz	100 kHz
Takte pro Modulo-Multiplikation	1
Takte pro Exponentiation (Mittelwert)	1536
Verschlüsselungsrate	8,3 KBit/s

Abbildung 4.2: „Ergebnisse ‘hochsprachliche Beschreibung’ (64 Bit)“

4.4 Untersuchte Verfahren

4.4.1 „Shift-and-Add“-Multiplizierer

Als erstes wurde ein einfacher Multiplizierer nach dem „Shift-and-Add“-Prinzip implementiert. Dieser ist in der Lage, zwei 64-Bit Zahlen x und y zu multiplizieren. Er berechnet das Ergebnis a einer Multiplikation nach der folgenden Formel:

$$a = x \cdot y = \sum_{i=0}^{63} x \cdot y_i \cdot 2^i \ .$$

Es wird x in Abhängigkeit von y_i aufsummiert. Die Fläche und Geschwindigkeit dieser Implementation diente als Anhaltspunkt für den Vergleich der folgenden Realisierungen.

4.4.2 „Carry-Save“-Multiplizierer

Die Grundlage für eine erfolgreiche Realisierung des RSA-Verfahrens ist die effiziente Implementierung der Addition. Der erfolgversprechendste Ansatz ist ein Addierer nach dem „Carry-Save"-Prinzip, wie bereits in Abschnitt „3.3 Addition" auf Seite 34 erläutert. Der größte Vorteil ist die hohe Geschwindigkeit der Verarbeitung, die sich durch die geringen Signallaufzeiten und die daraus resultierende hohe Taktfrequenz ergibt.

Ein weiterer Vorteil ist die Möglichkeit, praktisch eine beliebig große Addition in einem Takt auszuführen, da die Überträge der aktuellen Addition immer mit der folgenden Addition verarbeitet werden. Es soll das Produkt a von x und y berechnet werden. In c ist der Übertrag der vorangegangenen Addition zwischengespeichert.

$$a = y \cdot x = \sum_{i=0}^{l-1} (y_i \cdot x + c_{i-1}) \cdot 2^i$$

Aufgrund des zusätzlich benötigten Registers für den Übertrag c erhöht sich der Platzbedarf. Die Signallaufzeiten werden aber deutlich reduziert. Dieses liegt an der wesentlich einfacheren Logik (im Vergleich zum „Carry Lock Ahead"-Prinzip), die zur Berechnung der Zwischenergebnisse und des Übertrags nötig ist:

$$c_{-1} = 0$$

$$a_i = a_i \oplus c_{i-1} \oplus x_i$$

$$c_i = c_{i-1}a_i \vee c_{i-1}x_i \vee a_ix_i$$

Hierbei steht a_i für ein Bit des (Zwischen-)Ergebnisses a, c_i für ein Bit des Übertrages c und x_i für ein Bit des Multiplikators x.

Die Modulo-Reduktion wird durch die Subtraktion des Modulus k nach jeder Addition erreicht. Die Reduktion wird nur dann durchgeführt, wenn die Größe des Zwischenergebnisses eine zuvor festgelegte Grenze überschritten hat. Der Größenvergleich des Zwischenergebnisses mit dieser Grenze erfolgt nicht exakt. Es ist ausreichend, nur die höchstwertigsten Bits zu betrachten. In den seltenen Fällen, wo diese Schätzung zu einem Irrtum führt, kann nachträglich eine Korrektur durch eine Addition von k erfolgen.

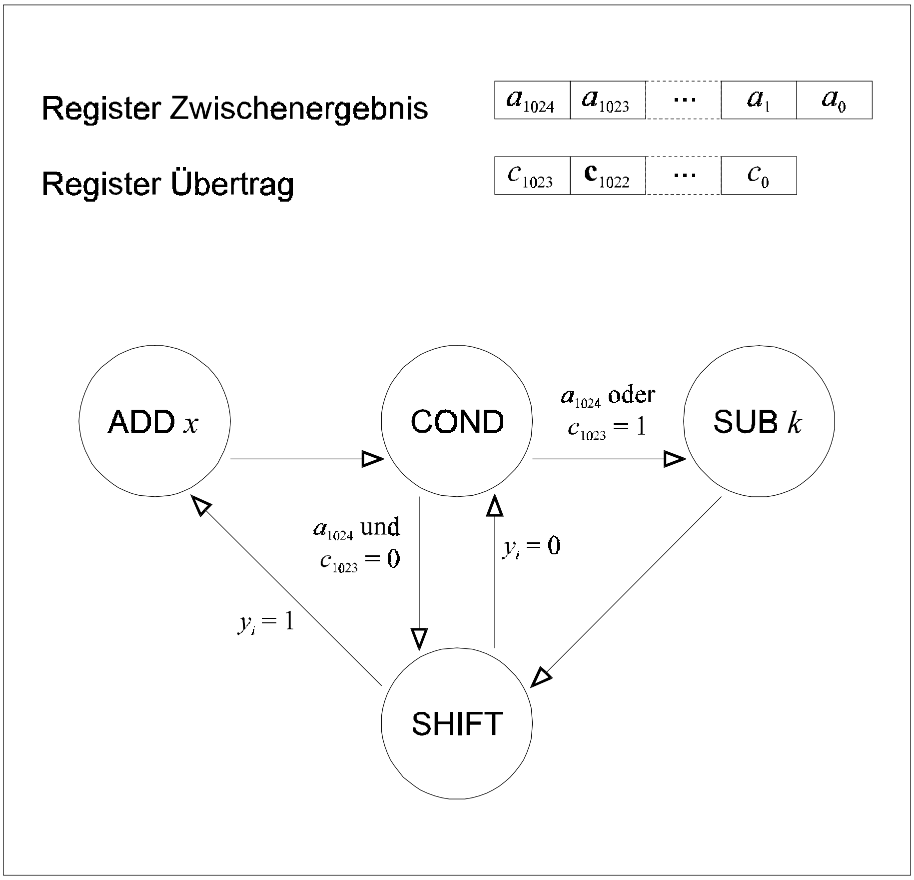

Abbildung 4.3: „Zustandsautomat ‘Carry-Save’“

Die Abbildung 4.3: „Zustandsautomat ‘Carry-Save’“ zeigt vereinfacht das Vorgehen beim Verfahren „Carry-Save“. Es wird im Prinzip

$$a + c <= x \cdot y \bmod k$$

berechnet. Nur wenn das jeweils höchstwertigste Bits von a und c Null ist, werden beide Register nach links geschoben, um die nächste Addition von x zu ermöglichen. Diese wird nur dann ausgeführt, wenn $y_i = 1$ ist. Sind die höchstwertigen Bits von a oder c nicht Null wird zunächst k abgezogen. Im folgenden Sourcecode wird der Algorithmus sehr vereinfacht dargestellt:

```
for(i=1023;i>=0;i--)
{
  if (a >= 1024) a = a - k;
  a = a << 1;
  if (yi == 1) a = a + x;
}
if (a >= k) a = a - k;
```

Es folgt ein Beispiel mit den Werten $x = 7$ (`0111`), $y = 5$ (`0101`), $k = 15$ (`1111`). Als Ergebnis erhält man:

$$a = x \cdot y \bmod k = 7 \cdot 5 \bmod 15 = 4.$$

Da es sich um 4-Bit Zahlen handelt, braucht die Schleife nur von i = 3 bis 0 laufen.

i =	a =	
3	0 0 0	(0)0000 (0)0000 (0)0000
2	0 0 7	(0)0000 (0)0000 (0)0111
1	7 14 21	(0)0111 (0)1110 (1)0101
0	6 12 19	(0)0110 (0)1100 (1)0011
	4	(0)0100

4.4.3 Aufteilung in kleine Multiplizierer

In einer weiteren, zunächst realisierten Methode wird die Multiplikation nicht ausschließlich auf die Addition zurückgeführt. Es werden statt dessen mehrere „kleine“ Multiplikationen durchgeführt. In einem Beispiel, das zu Testzwecken implementiert wurde, werden zwei 64-Bit Zahlen x und y multipliziert:

$$z = x \cdot y = x_{63}x_{62} \ldots x_0 \cdot y_{63}y_{62} \ldots y_0$$

Beide 64-Bit Zahlen werden jeweils zu acht Blöcken mit je acht Bit aufgeteilt. Somit wird die Multiplikation zweier 64-Bit Zahlen mit Hilfe von 64 8-Bit Multiplikationen organisiert.

$$a_7 = x_{64}x_{63} \ldots x_{58}x_{57},\ a_6 = x_{56}x_{55} \ldots x_{50}x_{49},\ \ldots\ ,\ a_0 = x_7x_6 \ldots x_1x_0$$

$$b_7 = y_{64}y_{63}y_{62}y_{61},\ b_6 = y_{60}y_{59}y_{58}y_{57},\ \ldots\ ,\ b_0 = y_3y_2y_1y_0$$

Es kommen vier Multiplizierer parallel zum Einsatz. Die Resultate der Multiplikationen werden in einer vorgegebenen Reihenfolge von Additionen aufsummiert (siehe Abbildung 4.4: „Algorithmus ‘kleine Multiplizierer’“). Begonnen wird mit den Teilprodukten a_7b_7, a_6b_6, a_5b_5 und a_4b_4 in der ersten Zeile (dunkel hinterlegt). Dieses wird bis in die neunte Zeile fortgesetzt. In den Zeilen zehn bis 15 werden die hell unterlegten Teilprodukte zusammen mit den restlichen der ersten Zeilen aufsummiert.

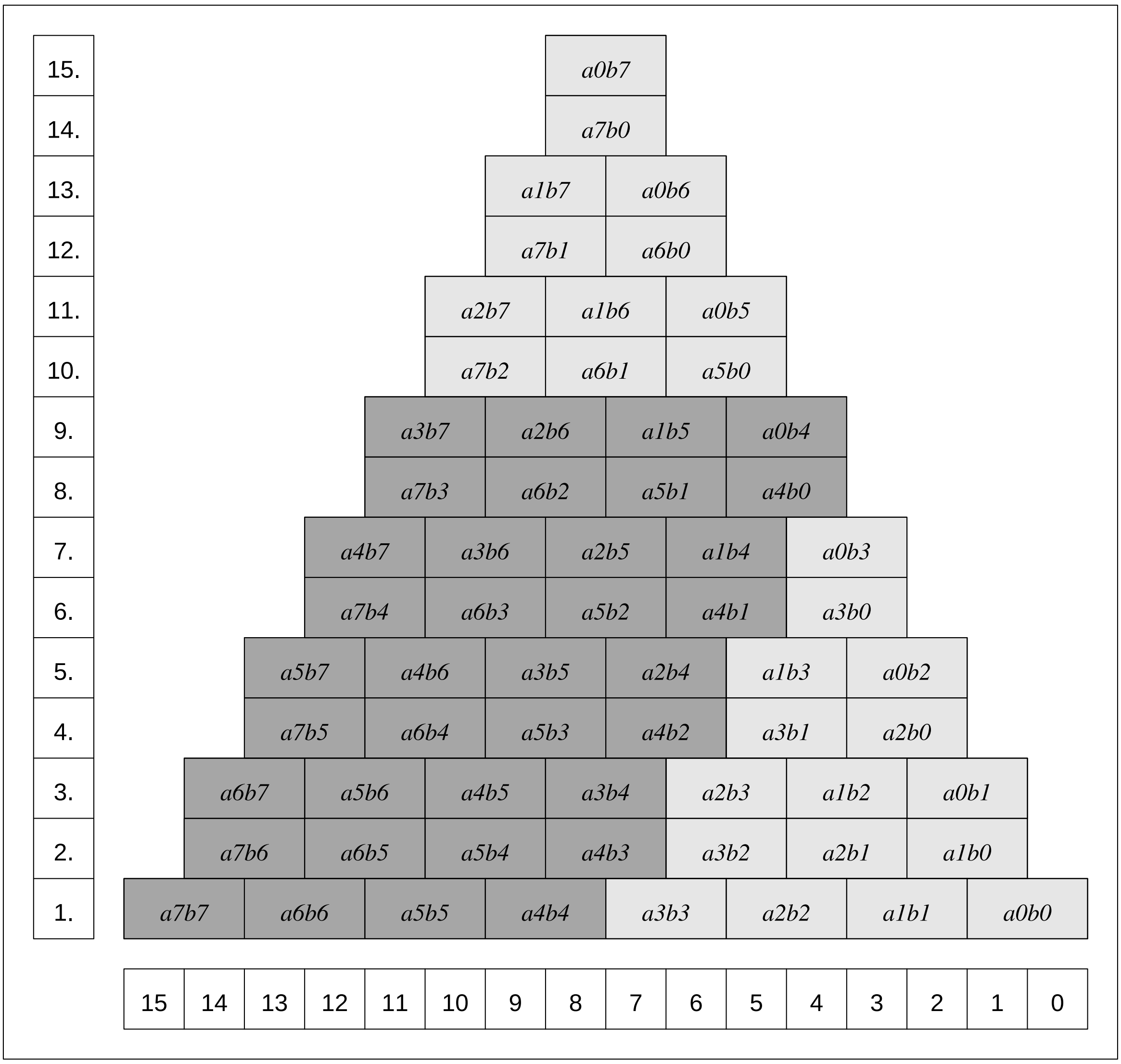

Abbildung 4.4: „Algorithmus ‘kleine Multiplizierer’“

Die Reihenfolge der Multiplikationen wurde im voraus festgelegt und in einem ROM abgelegt. In dem folgenden Auszug aus dem VHDL-Code für den Entwurf „kleine Multiplizierer“ steht jede Spalte für die im jeweiligen Takt zu verarbeitenden Teilprodukte:

1. Spalte: 7 - 7 - 6 - 6 - 5 - 5 - 4 - 4 $\Rightarrow$ 1. Takt: $s = s + a_7b_7 + a_6b_6 + a_5b_5 + a_4b_4$
2. Spalte: 6 - 7 - 5 - 6 - 4 - 5 - 3 - 4 $\Rightarrow$ 2. Takt: $s = s + a_6b_7 + a_5b_6 + a_4b_5 + a_3b_4$
...
16. Spalte: 3 - 3 - 2 - 2 - 1 - 1 - 0 - 0 $\Rightarrow$ 16. Takt: $s = s + a_3b_3 + a_2b_2 + a_1b_1 + a_0b_0$

```
...
type rom is array (0 to 15) of integer range 0 to 7;
constant rom3a: rom := (7,6,7,5,7,4,7,3,7,2,7,1,7,0,7,3);
constant rom3b: rom := (7,7,6,7,5,7,4,7,3,7,2,7,1,7,0,3);
constant rom2a: rom := (6,5,6,4,6,3,6,2,6,1,6,0,6,2,3,2);
constant rom2b: rom := (6,6,5,6,4,6,3,6,2,6,1,6,0,3,2,2);
constant rom1a: rom := (5,4,5,3,5,2,5,1,5,0,5,1,3,1,2,1);
constant rom1b: rom := (5,5,4,5,3,5,2,5,1,5,0,3,1,2,1,1);
constant rom0a: rom := (4,3,4,2,4,1,4,0,4,0,3,0,2,0,1,0);
constant rom0b: rom := (4,4,3,4,2,4,1,4,0,3,0,2,0,1,0,0);

FSM: process(current_state)
begin
    case current_state is

  ...

      when MULT_ADD =>
        reg <= reg + (oflow&
               (xx(rom3a(count)) * yy(rom3b(count)))&
               (xx(rom2a(count)) * yy(rom2b(count)))&
               (xx(rom1a(count)) * yy(rom1b(count)))&
               (xx(rom0a(count)) * yy(rom0b(count))));
        count <= count + 1;

  ...

  end case;
end process;
```

Es zeigte sich, daß die Signallaufzeiten länger wurden als bei der „Shift-and-Add“ Methode. Die gesamte Multiplikation läßt sich nun aber in weniger Takten durchführen, weil die 8-Bit Multiplikation in einem Takt ausgeführt wird. Die Methode „Kleine Multiplizierer“ war deshalb insgesamt 80 Prozent schneller als die „Shift-and-Add“ Methode.

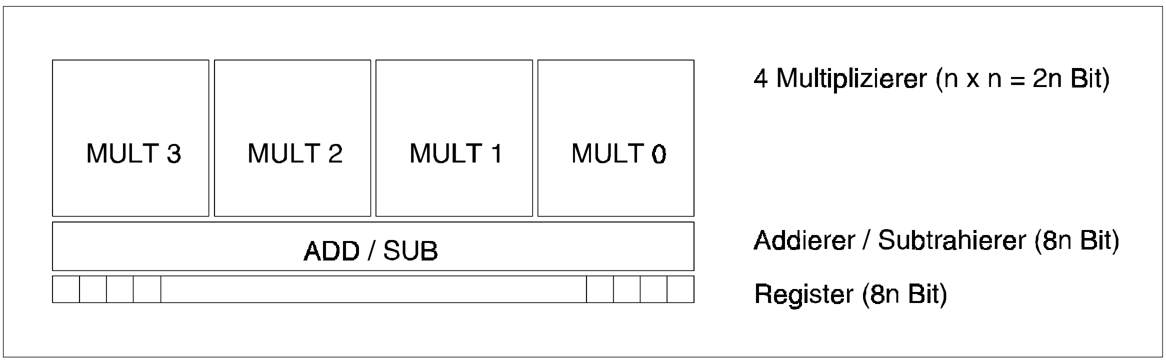

Abbildung 4.5: „Prinzip ‘kleine Multiplizierer’“

Auch bei diesem Entwurf stellte die Realisierung der Modulo-Reduktion die entscheidende Schwierigkeit dar. Die Reduktion wurde mit Hilfe einer kontinuierlichen Subtraktion des Modulus *k* erreicht. Die Abbildung 4.6: „Zustandsautomat ‘kleine Multiplizierer’“ zeigt den zentralen Steuerungsautomaten dieser Methode. Es werden immer vier Multiplikationen parallel ausgeführt und das Resultat zum Zwischenergebnis addiert (`MULT`). Im Folgezustand (`COND`) wird entschieden, ob eine Reduktion (`SUB`) nötig ist oder ob das Zwischenergebnis für die nächste Addition der Resultate der Multiplikation vorbereitet werden kann (`SHIFT`).

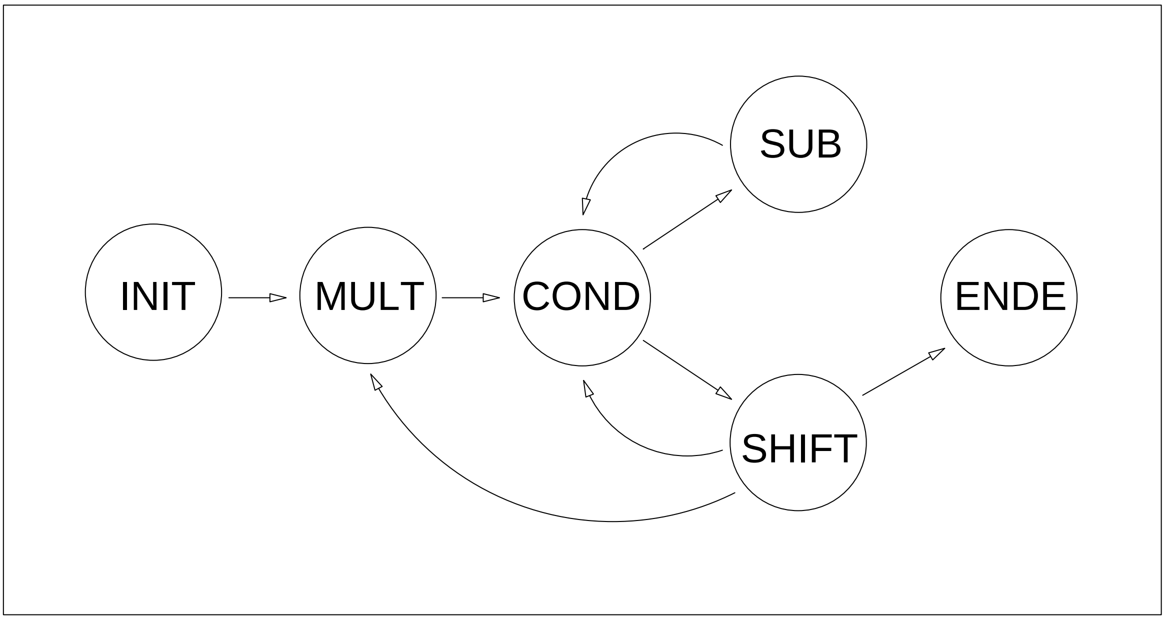

Abbildung 4.6: „Zustandsautomat 'kleine Multiplizierer'"

4.4.4 Montgomery

Die Konzeption und Implementation des RSA-Verfahren nach der Methode von Montgomery wurde von Sauerbrey [SAU93] im Rahmen seiner Dissertation ausführlich untersucht. Zur effizienten Implementierung schlägt er unterschiedliche systolische Arrays vor. Diese unterscheiden sich in der Größe der verwendeten Zellen. Von ihm werden Zellen vorgeschlagen, die ein, zwei oder vier Bit parallel verarbeiten können.

Um die Komplexität und die Durchlaufverzögerung bei unterschiedlicher Zellengröße vergleichen zu können, müssen sie auf die Anzahl der pro Zelle und Takt verarbeiteten Operandenbits bezogen werden. Sauerbrey gibt eine Zelle, die zwei Bit pro Takt verarbeitet, als optimal in bezug auf die Fläche an. Ein optimales Zeitverhalten soll die Zelle mit drei Bit liefern (siehe Abbildung 4.7: „Flächen- und Zeitbedarf nach Sauerbrey"). Er stützt sich hierbei auf Angaben eines Herstellers von Standardzellen und addiert die Werte für Fläche und Durchlaufverzögerung der von ihm benötigten Standardzellen von Hand.

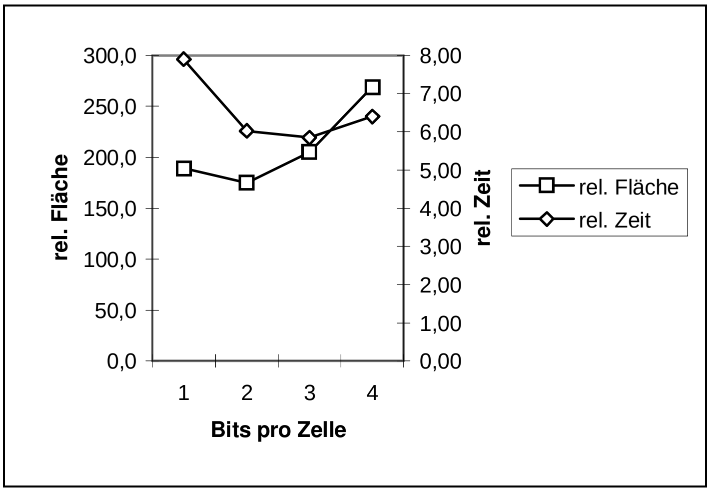

Abbildung 4.7: „Flächen- und Zeitbedarf nach Sauerbrey“

Untersuchungen mit dem Entwurfsystem von „SYNOPSYS“ konnten diese Werte nicht bestätigen. Optimal in bezug auf die Fläche war hier die Zelle, die nur ein Bit pro Takt verarbeitet. Das beste Zeitverhalten zeigte die Zelle mit 4 Bit (siehe Abbildung 4.8: „Flächen- und Zeitbedarf (Synopsys)“).

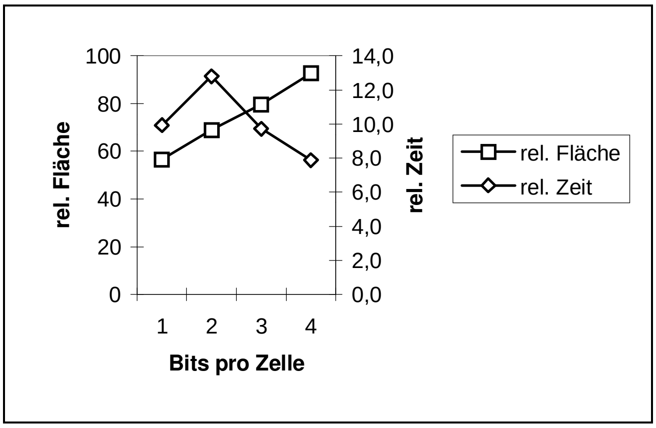

Abbildung 4.8: „Flächen- und Zeitbedarf (SYNOPSYS)“

4.5 Zusammenfassung und Bewertung

Die im vorherigen Kapitel beschriebenen Ansätze wurden in VHDL beschrieben und mit Hilfe des Programmpaketes „SYNOPSYS“ synthetisiert. Sie wurden in Hinsicht auf Ausführungszeit und Komplexität verglichen. In allen Implementierungen wurde zunächst die Multiplikation mit Zahlen der Größe 64 Bit durchgeführt. Die Exponentiation wurde bei allen Verfahren mit Hilfe des „Binärverfahrens“ implementiert.

	Fläche
„Shift-and-Add"-Multiplizierer	1.00
„Carry-Save"-Multiplizierer	1.35
Partielle Multiplikation	0.72

Abbildung 4.9: „Abschätzungen Fläche (Multiplikation)"

	Takte pro Multiplikation	*Signallaufzeit*	*Summe Zeit*
„Shift-and-Add"-Multiplizierer	1.00	1.00	1.00
„Carry-Save"-Multiplizierer	1.00	0.12	0.12
Partielle Multiplikation	0.75	1.09	0.82

Abbildung 4.10: „Abschätzungen Zeit (Multiplikation)"

Die Abbildung 4.9: „Abschätzungen Fläche (Multiplikation)" und die Abbildung 4.10: „Abschätzungen Zeit (Multiplikation)" zeigen den Platz- bzw. den Zeitbedarf der verwendeten Verfahren zur Durchführung der Multiplikation. Die „Shift-and-Add" Methode wurde zur Indizierung auf den Wert 1.00 gesetzt. Als Flächenabschätzung dienten die Werte von „SYNOPSYS". Sie beziehen sich auf die Fläche der Standardzellen plus einem Schätzwert für die Fläche der Verdrahtung. Die Signallaufzeit für den „längsten Pfad" wurde ebenfalls mit „SYNOPSYS" ermittelt. Diese wurde mit der Anzahl der für eine Multiplikation benötigen Taktzyklen multipliziert, um so einen Wert für den Zeitbedarf des Entwurfs zu erhalten.

Zur Implementation der Modulo-Reduktion wurden ebenfalls Abschätzungen vorgenommen. Die Ergebnisse sind in der Abbildung 4.11: „Abschätzung Modulo-Reduktion" zusammengefaßt. Während die Methode ① einen exakten Größenvergleich durchführt, bedienen sich die Methoden ② und ③ der Quotientenschätzung. Hierbei werden nur die höchstwertigsten Stellen verglichen. Das Ergebnis muß allerdings am Ende gegebenenfalls noch korrigiert werden (siehe „3.2.1 Modulo-Multiplikation" auf Seite 33). Dem Konzept ③ liegt außerdem ein Subtrahierer nach dem „Carry-Save"-Prinzip zugrunde. Die Methode ① wurde wieder zur Indizierung auf den Wert 1.00 gesetzt.

	Fläche	*Takte pro Reduktion*	*Signal-laufzeit*	*Summe Zeit*
①	1.00	1.00	1.00	1.00
②	0.47	1.33	0.51	0.68
③	0.84	1.33	0.11	0.15

Abbildung 4.11: „Abschätzung Modulo-Reduktion"

Aufgrund der vorliegenden Ergebnisse wurde für die endgültige Realisierung ein Verfahren gewählt, das sowohl die Multiplikation als auch die Reduktion nach dem „Carry-Save"-Prinzip durchführt. Die Subtraktion sollte aufgrund der Ergebnisse nach einer Quotientenschätzung erfolgen und nicht nach einem exakten Größenvergleich. Die Methode „kleine Multiplizierer" scheint wegen der hohen Signallaufzeiten keine effiziente Lösung zu sein. Auch wenn ein Teil der Zeit aufgrund der weniger benötigten Taktzyklen wieder eingespart wird. Der relativ hohe Platzbedarf der „Carry-Save"-Methode ist darin begründet, daß zur Speicherung des Übertrags ein zusätzliches Register benötigt wird. Dieses wurde zunächst mit D-Flip-Flops realisiert. Durch die Verwendung von RAM kann die Fläche auf sechs Prozent (!) der Fläche der D-Flip-Flop Lösung reduziert werden (siehe hierzu Abschnitt „6.2.2 Fläche“ auf Seite 99).

Für die vollständige Implementierung wurde neben der Methode „Carry-Save“ das Verfahren nach Montgomery ausgewählt. Dieses wurde – wie bereits erwähnt – von Sauerbrey [SAU93] ausführlich untersucht. Nach seinen Ergebnissen erscheinen die Vorgaben bezüglich Geschwindigkeit erreichbar zu sein. Zum Flächenbedarf wurden von Sauerbrey keine präzisen Werte ermittelt. Eine Untersuchung zum Flächenbedarf erschien deshalb im Rahmen dieser Arbeit als sinnvoll. Aufgrund der in Abschnitt „3.5.2 Leistungsvergleich“ auf Seite 41 beschriebenen Untersuchung und dem hohen erwarteten Flächenbedarf wurde die Entscheidung zugunsten einer Realisierung mit einer Zelle, die nur ein Bit verarbeiten kann, getroffen.

5 Realisierung und Implementation

5.1 Montgomery-Verfahren zur Modulo-Multiplikation

Ein interessanter Ansatz zur Verringerung des Aufwandes bei der Durchführung von Modulo-Multiplikationen stammt von Montgomery [MON85]. Durch die geeignete Abbildung der beiden zu multiplizierenden Operanden wird die Division durch den Modulus k durch eine Division durch eine relativ frei wählbare Zahl ersetzt. Wenn diese Zahl eine Potenz der Zahlenbasis ist, in der die Operanden dargestellt sind, kann die Division durch eine Rechtsverschiebung durchgeführt werden.

Es wird also eine Transformation der Operanden in den „Montgomery-Bereich" vorgenommen, dort eine der Modulo-Multiplikation entsprechende, aber einfachere Operation vorgenommen und dann das Ergebnis wieder zurücktransformiert.

5.1.1 Algorithmen Montgomery-Produkt

Die grundsätzliche Funktionsweise der Montgomery-Methode wurde bereits in Abschnitt '3.5 Reduktion nach Montgomery' auf Seite 39 ausführlich behandelt. Der Algorithmus zur Berechnung des Montgomery-Produktes in C++ wird an dieser Stelle noch einmal wiederholt (k, k' und r sind Konstanten.):

```
int mp1(int x,int y)
{
  int p,v,t;
  p = x * y;
  v = (p * k') % r;
  t = (p + v*k) / r;
  return(t);
}
```

Abbildung 5.1: „Algorithmus Montgomery-Produkt (1)"

Um den Algorithmus effizient implementieren zu können, müssen die Multiplikationen, die Division und die Modulo-Berechnung auf Additionen und Schiebeoperationen zurückgeführt werden. Es wird jetzt nicht ein Faktor ($v \cdot k$) gesucht, um das Ergebnis t ganzzahlig werden zu lassen, sondern durch fortgesetztes bedingtes Addieren von k werden die niederwertigen Bits von t zu Null[14]. Das Ergebnis t wird so nach der Division durch r

[14] Das niederwertigste Bit des Modulus k ist immer gleich Eins ($k_0 = 1$), weil der Modulus k immer eine ungerade Zahl sein muß. Durch Addition von k wird das niederwertigste Bit des (Zwischen-)Ergebnisses t zu Null, wenn es bisher gleich Eins war.

ganzzahlig. Die zweite Version des Algorithmus zur Berechnung des Montgomery-Produktes (*r* und der Modulus *k* sind Konstanten.):

```
int mp2(int x, int y)
{
  int t,i;
  t = 0;
  for(i=0;i<1024;i++)
  {
    t = t + xi * y * 2^i;
    t = t + ti * k * 2^i;
  }
  return(t/r);
}
```

Abbildung 5.2: „Algorithmus Montgomery-Produkt (2)“

Es folgt ein Beispiel für den obigen Algorithmus zur Berechnung des Montgomery-Produktes mit den Werten $x = 3$ (`0011`), $y = 10$ (`1010`) und $k = 13$ (`1101`). Es wurde $r = 2^7 = 128$ gewählt. Daraus folgt $r^{-1} = 6$ (siehe „3.5.1 Algorithmus Montgomery“ auf Seite 39). Als Ergebnis erwartet man:

$$z = MP(x, y, k, r) = (x \cdot y \cdot r^{-1}) \bmod k = 180 \bmod 13 = 11\ (\texttt{1011})$$

In jedem Durchlauf der Schleife wird geprüft, ob das *i*-te Bit von *x* bzw. *t* gesetzt ist. Ist dieses der Fall, wird *y* bzw. *k* zum Zwischenergebnis addiert. Die Multiplikation mit der entsprechenden Zweierpotenz ersetzt das „Schieben-nach-links“. So entspricht „$\cdot\ 2^i$“ einem Schieben um *i* Stellen nach links. Die Schleife läuft in diesem Beispiel nur von 0 bis 6, da hier $r = 2^7$.

i	*t*	*Bemerkung*	
0	1010	$+ x_0 \cdot y \cdot 2^0$	$\sim + y$
	....	$+ t_0 \cdot k \cdot 2^0$	$\sim + 0$
1	11110	$+ x_1 \cdot y \cdot 2^1$	$\sim + y \cdot 2$
	111000	$+ t_1 \cdot k \cdot 2^1$	$\sim + k \cdot 2$
2	......	$+ x_2 \cdot y \cdot 2^2$	$\sim + 0$
	......	$+ t_2 \cdot k \cdot 2^2$	$\sim + 0$
3	......	$+ x_3 \cdot y \cdot 2^3$	$\sim + 0$
	10100000	$+ t_3 \cdot k \cdot 2^3$	$\sim + k \cdot 8$
4	....	$+ x_4 \cdot y \cdot 2^4$	$\sim + 0$
	....	$+ t_4 \cdot k \cdot 2^4$	$\sim + 0$
5	....	$+ x_5 \cdot y \cdot 2^5$	$\sim + 0$
	1001000000	$+ t_5 \cdot k \cdot 2^5$	$\sim + k \cdot 32$
6	....	$+ x_6 \cdot y \cdot 2^6$	$\sim + 0$
	10110000000	$+ t_6 \cdot k \cdot 2^6$	$\sim + k \cdot 64$
7	1011	$\div r$	$\sim \div 128$

Abbildung 5.3: „Beispiel Algorithmus Montgomery (2)“

Ein Problem ist jetzt noch, daß das Ergebnis im Laufe der Berechnung zu groß wird. Die Lösung hierfür ist die Verlagerung der Division durch *r* in die Schleife. Das Ergebnis *t* wird jetzt nicht einmal durch *r* geteilt, sondern *i*-mal durch Zwei ($r = 2^i$)[15]. Auf diese Weise wird das Ergebnis *t* klein gehalten. Außerdem spart man so zudem zwei weitere Schiebeoperationen ein. Es folgt die endgültige Form des Algorithmus zur Berechnung des Montgomery-Produktes:

```
int mp3(int x, int y)
{
  int t,i;
  t = 0;
  for(i=0;i<1024;i++)
  {
    t = t + xi * y;
    t = t + t0 * k;
    t = t >> 1;
  }
return(t);
}
```

Abbildung 5.4: „Algorithmus Montgomery-Produkt (3)“

Es folgt wieder das gleiche Beispiel zur Verdeutlichung des Algorithmus. In Abhängigkeit von x_i und t_0 wird *y* bzw. *k* zum Zwischenergebnis addiert:

[15] Dieses ist möglich, weil in jedem Schleifendurchlauf das niederwertigste Bit t_0 des Ergebnisses *t* zu Null wird.

i	*t*	*Bemerkung*

0	`1010`
	`....`
	`101`
		+ y
		+ 0
		÷ 2

1	`1111`
	`11100`
	`1110`
		+ y
		+ k
		÷ 2

2	`....`
	`....`
	`111`
		+ 0
		+ 0
		÷ 2

3	`...`
	`10100`
	`1010`
		+ 0
		+ k
		÷ 2

4	`....`
	`....`
	`101`
		+ 0
		+ 0
		÷ 2

5	`.....`
	`10010`
	`1001`
		+ 0
		+ k
		÷ 2

6	`....`
	`10110`
	`1011`
		+ 0
		+ k
		÷ 2

Abbildung 5.5: „Beispiel Algorithmus Montgomery (3)“

5.1.2 Systolisches Feld zur Multiplikation

Bereits im Jahre 1965 hat J. Atrubin [ATR65] ein eindimensionales systolisches Feld zur Multiplikation zweier positiver *n*-stelliger Dualzahlen *x* und *y* vorgeschlagen.

$$p = x \cdot y$$

Das Feld besteht aus einer linearen Anordnung von $\lceil n / 2 \rceil + 1$ Zellen, die zentral getaktet werden. Die Abbildung 5.6: „Prinzip des systolischen Arrays“ zeigt den prinzipiellen Aufbau des Feldes. Die Anordnung der Operandenbits x_i, y_i und der Produktbits p_i spiegeln die Reihenfolge wieder, wie die Operanden in das Feld geschoben werden und wie das Ergebnis am Ausgang erscheint.

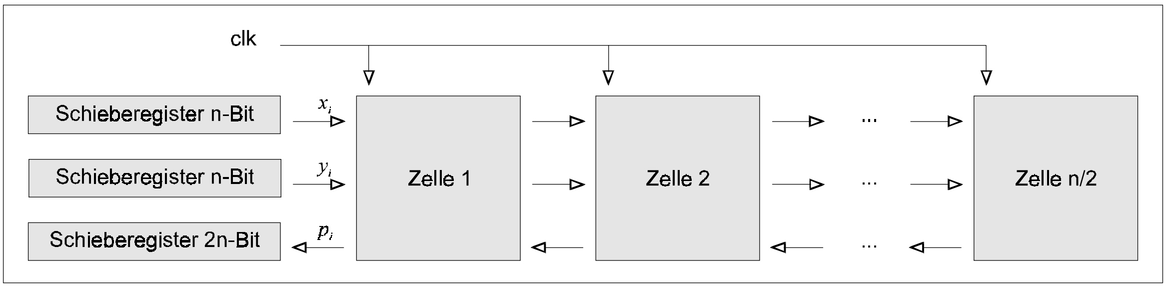

Abbildung 5.6: „Prinzip des systolischen Arrays“

Die Operanden x und y werden mit dem LSB[16] voran in n Takten seriell in das Array geschoben. In weiteren n Takten werden Nullen an die beiden Eingänge angelegt. Das LSB p_0 des Produktes erscheint sofort (bzw. nach einem Takt) am Ausgang p, die weiteren Produktbits p_i folgen in den nächsten Takten, so daß die Multiplikation nach $2n$ Takten abgeschlossen ist. Die Zeit, um das Produkt zu berechnen, wächst linear mit der Stellenzahl der Operanden. Der Datenfluß erfolgt in zwei Richtungen. Die Operanden wandern von links nach rechts, während das Ergebnis von rechts nach links fließt.

Im folgenden (siehe „5.1.6 Systolische Arrays zur “ auf Seite 77) wird der Entwurf von Atrubin so modifiziert, daß er als Baustein für einen Modulo-Multiplizierer nach Montgomery eingesetzt werden kann. Der Wert der zu multiplizierenden n-stelligen Operanden x und y ergibt sich aus den einzelnen Ziffern x_i und y_i in der üblichen Radixschreibweise zu

$$x = \sum_{i=0}^{n-1} x_i \cdot 2^i \quad \text{und} \quad y = \sum_{i=0}^{n-1} y_i \cdot 2^i \quad \text{mit } x_i, y_i \in \{0, 1\}.$$

Das Produkt $p = x \cdot y$ wird dargestellt als

$$p = \sum_{i=0}^{2n-1} p_i \cdot 2^i .$$

Das Ziel ist es, aus den seriell eingelesenen Ziffern x_i und y_i die Ziffern p_i des Produktes zu berechnen. Die Abbildung 5.7: „Addition von Teilprodukten“ zeigt, wie die Berechnung der Ziffern p_i des Produktes bei der Multiplikation von vierstelligen Zahlen durch die übliche „Schulmethode“ des verschobenen Addierens von Teilprodukten realisiert werden kann. Die Ziffern p_i ergeben sich aus der Addition modulo 2 der Einträge in Spalte i, wobei Überträge aus der Addition weiter rechts stehender Spalten berücksichtigt werden müssen. Werden die Ziffern x_i und y_i seriell (LSB zuerst) bereitgestellt, so ist im Takt i die komplette Information vorhanden, um Ziffer p_i berechnen zu können.

Zeile \ Spalte	7	6	5	4	3	2	1	0
0					$x_0 \cdot y_3$	$x_0 \cdot y_2$	$x_0 \cdot y_1$	$x_0 \cdot y_0$
1				$x_1 \cdot y_3$	$x_1 \cdot y_2$	$x_1 \cdot y_1$	$x_1 \cdot y_0$	
2			$x_2 \cdot y_3$	$x_2 \cdot y_2$	$x_2 \cdot y_1$	$x_2 \cdot y_0$		
3		$x_3 \cdot y_3$	$x_3 \cdot y_2$	$x_3 \cdot y_1$	$x_3 \cdot y_0$			
p	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Abbildung 5.7: „Addition von Teilprodukten“

[16] Least Significant Bit (LSB)

5.1.3 Aufbau des systolische Arrays zur Multiplikation

Die Abbildung 5.8: „Aufbau des systolischen Arrays zur Multiplikation“ zeigt den groben Aufbau des systolischen Arrays für die Multiplikation. Es ist in der Lage die Berechnung

$$p = x \cdot y + q$$

durchzuführen. Die Zahlen x ($x_{1023}x_{1022}...x_1x_0$), y ($y_{1023}y_{1022}...y_1y_0$) und q ($q_{1023}q_{1022}...q_1q_0$) werden bitweise an das systolische Array angelegt[17]. Es wird mit den niederwertigsten Bits (x_0, y_0 und q_0) begonnen. Bereits nach einem Takt steht das niederwertigste Bit (p_0) des Ergebnisses p am Ausgang an, weil die Berechnung

$$p_0 = x_0 \cdot y_0 + q_0$$

schon im ersten Takt in der ersten Zelle durchgeführt werden kann. Die genaue Funktionsweise der Zellen und die Berechnung der weiteren Bits des Ergebnisses wird im folgenden Abschnitt „5.1.4 Aufbau einer Zelle des systolischen Multiplizierers“ beschrieben.

Das Feld besteht insgesamt aus 513 identischen Zellen. Jede Zelle des Feldes besteht aus einem Zustandsteil, einem Operandenteil und einem Summenteil. Jeder Teil besitzt seine eigenen Register, verarbeitet seine Eingangswerte und erzeugt Ausgangswerte. Es gibt aber auch Interaktionen zwischen den Teilen. Dieses wird ebenfalls im folgenden erläutert.

[17] Da der Eingang q eine 2-Bit Eingabe erfordert, wird sein hochwertiges Bit permanent auf Null gesetzt.

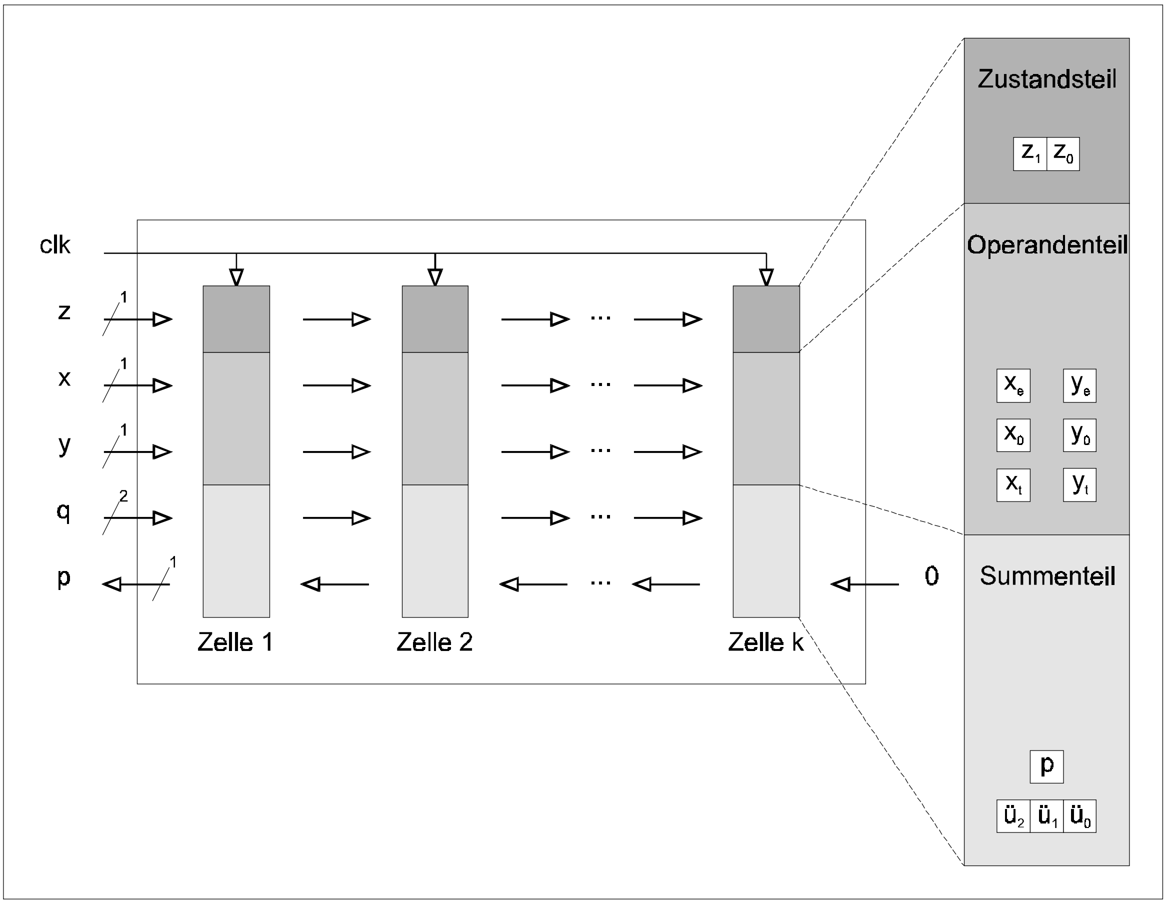

Abbildung 5.8: „Aufbau des systolischen Arrays zur Multiplikation“

5.1.4 Aufbau einer Zelle des systolischen Multiplizierers

Als erstes wird zunächst die Funktionsweise des Operandenteils beschrieben. Die Abbildung 5.9: „Operandenteil einer Zelle“ zeigt den Operandenteil einer Zelle.

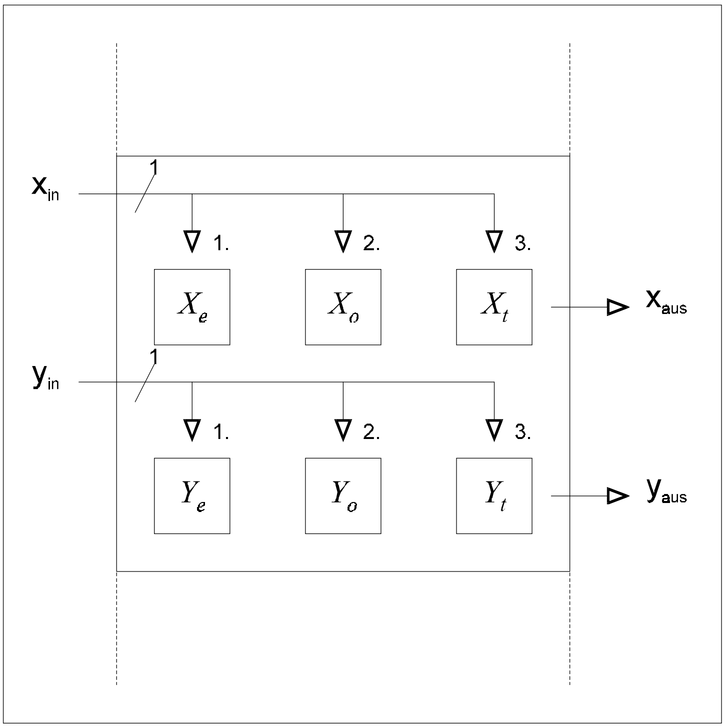

Abbildung 5.9: „Operandenteil einer Zelle“

Jede Zelle des Arrays enthält sechs Register zum dauerhaften Aufnehmen der Bits x_i und y_i. Die Register X_e und Y_e speichern je ein x_i bzw. y_i mit geradem i (e für ‘even’), während die Register X_o und Y_o jeweils ein x_i bzw. y_i mit ungeradem i speichern (o für ‘odd’). Sind die Register X_e, Y_e, X_o und Y_o einmal belegt, wird ihr Wert während der Multiplikation nicht mehr verändert. Zum Durchschieben durch das Array werden x_i und y_i in X_t und Y_t zwischengespeichert. Der Wert dieser Register ändert sich somit ständig. Sie dienen nur der temporären Aufnahme von x_i und y_i.

Die Eingangswerte x_{in} und y_{in} werden folgendermaßen auf die 1-Bit Register verteilt. In jedem Takt werden die folgenden Schritte ausgeführt:

1. Falls X_e und Y_e noch nicht belegt sind, werden sie mit den Eingangswerten x_{in} und y_{in} der Zelle belegt.
2. Falls 1. nicht zutrifft und X_o und Y_o noch nicht belegt sind, werden sie mit den Eingangswerten x_{in} und y_{in} der Zelle belegt.
3. Falls 1. und 2. nicht zutrifft, werden X_t und Y_t mit den Eingangswerten x_{in} und y_{in} der Zelle belegt.

Mit anderen Worten: Als erstes werden die Register X_e und Y_e belegt. Es folgen die Register X_o und Y_o. Diese beiden Registerpaare erhalten ihren Wert einmal zugewiesen und behalten ihn während der ganzen Multiplikation. In X_t und Y_t werden die folgenden Werte temporär für einen Takt gespeichert und dann an die rechte Nachbarzelle weitergereicht. Die Ausgangswerte x_{aus} und y_{aus} sind somit die aktuellen Werte in X_t und Y_t. Diese sind dann wiederum die Eingangswerte der rechten Nachbarzelle. Die 1. Zelle speichert somit die Werte x_0, y_0, x_1 und y_1, die 2. Zelle speichert x_2, y_2, x_3 und y_3. Die n-te Zelle schließlich speichert die Werte x_{2n-2}, y_{2n-2}, x_{2n-1} und y_{2n-1}. Es folgt die Beschreibung des Operandenteils in VHDL:

```
OP: process(reset,clk,current_state,zin)
  begin
    if (reset = '0') then
      p <= '0';
      ue <= "000";
      xe <= '0'; ye <= '0';
      xo <= '0'; yo <= '0';
      xt <= '0'; yt <= '0';
    elsif (clk'event and clk='1') then -- positive Flanke
      p <= pin;
      ue <= s(3 downto 1);
      if ((current_state=Z0) and (zin='1')) then
        xe <= xin;
        ye <= yin;
      elsif ((current_state=Z1) and (zin='0')) then
        xo <= xin;
        yo <= yin;
      else
        xt <= xin;
        yt <= yin;
      end if;
    end if;
  end process;
```

Durch diese Vorgehensweise werden die Operanden in den Zellen mit jedem Takt so verteilt, daß jede Zelle zum frühestmöglichen Zeitpunkt einen Teil der Spaltensumme nach Abbildung 5.7: „Addition von Teilprodukten“ berechnen kann. Die Spaltensumme wird im Summenteil der Zelle durchgeführt. Jede Zelle ist in der Lage, die folgenden Berechnungen durchzuführen:

$$q_{aus} = x_{in} \cdot y_{in}$$

$$q_{aus} = x_{in} \cdot Y_o + y_{in} \cdot X_o$$

$$zs = x_{in} \cdot y_{in}$$

$$zs = x_{in} \cdot Y_e + y_{in} \cdot X_e$$

Das folgende Beispiel zeigt, wie mit Hilfe dieser Teilergebnisse unter Verwendung von zwei Zellen die Multiplikation p $(p_2p_1p_0) = x$ $(x_2x_1x_0) \cdot y$ $(y_2y_1y_0) = 101 \cdot 110 = 11110$ zu berechnen ist. Der Eingang q_{in} der ersten Zelle wurde zur Vereinfachung auf den Wert Null gesetzt. Es sonst grundsätzlich möglich, zum Ergebnis der Multiplikation noch einen Wert q zu addieren.

Takt	Zelle #1	Zelle #2
1	$zs = x_0 \cdot y_0 = 1{\cdot}0 = 0$ $\leftarrow p_{aus} = zs + q_{in} + p_{in} = 0{+}0{+}0 = 0$ $\rightarrow q_{aus} = 0$	$zs = 0$ $\leftarrow p_{aus} = zs + q_{in} + p_{in} = 0{+}0{+}0 = 0$ $\rightarrow q_{aus} = 0$
2	$zs = x_0 \cdot y_1 + x_1 \cdot y_0 = 1{\cdot}1{+}0{\cdot}0 = 1$ $\leftarrow p_{aus} = zs + q_{in} + p_{in} = 1{+}0{+}0 = 1$ $\rightarrow q_{aus} = x_1 \cdot y_1 = 0{\cdot}1 = 0$	$zs = 0$ $\leftarrow p_{aus} = zs + q_{in} + p_{in} = 0{+}0{+}0 = 0$ $\rightarrow q_{aus} = 0$
3	$zs = x_0 \cdot y_2 + x_2 \cdot y_0 = 1{\cdot}1{+}1{\cdot}0 = 1$ $\leftarrow p_{aus} = zs + q_{in} + p_{in} = 1{+}0{+}0 = 1$ $\rightarrow q_{aus} = x_1 \cdot y_2 + x_2 \cdot y_1 = 0{\cdot}1{+}1{\cdot}1 = 1$	$zs = 0$ $\leftarrow p_{aus} = zs + q_{in} + p_{in} = 0{+}1{+}0 = 1$ $\rightarrow q_{aus} = 0$
4	$zs = 0$ $\leftarrow p_{aus} = zs + q_{in} + p_{in} = 0{+}0{+}1 = 1$ $\rightarrow q_{aus} = 0$	$zs = x_2 \cdot y_2 = 1{\cdot}1 = 1$ $\leftarrow p_{aus} = zs + q_{in} + p_{in} = 1{+}0{+}0 = 1$ $\rightarrow q_{aus} = 0$
5	$zs = 0$ $\leftarrow p_{aus} = zs + q_{in} + p_{in} = 0{+}0{+}1 = 1$ $\rightarrow q_{aus} = 0$	$zs = 0$ $\leftarrow p_{aus} = zs + q_{in} + p_{in} = 0{+}0{+}0 = 0$ $\rightarrow q_{aus} = 0$

Abbildung 5.10.: „Beispiel systolische Multiplikation“

Die Abbildung 5.11: „Funktionsweise der Zellen (systolischer Multiplizierer)“ zeigt ein Beispiel mit drei Zellen. Im Vergleich zum vorhergehenden Beispiel, wird zusätzlich die sukzessive Belegung der Register verdeutlicht. Jede Zelle benutzt zur Berechnung nur die Belegung der eigenen Register sowie einen Teil der Registerbelegung der linken Nachbarzelle. Das Ergebnis der Berechnung wird an die linke Nachbarzelle weitergegeben, die im nächsten Takt weitere Summanden der entsprechenden Spaltensumme hinzuaddiert. Wenn die Spaltensumme der Spalte i im Takt i+1 in der linken Zelle angekommen ist, wurde die Ziffer p_i komplett berechnet. Entstehende Überträge werden in der Zelle für einen Takt gespeichert, so daß sie im nächsten Takt zur nächsthöherwertigen Spalte hinzuaddiert werden können. Die folgende Abbildung zeigt die sukzessive Belegung der Register und die Berechnung der Variablen p_{aus} und q_{aus}. Der Wert von p_{aus} wird mit dem nächsten Takt an die linke Nachbarzelle weitergereicht und dort weiterverarbeitet. Die Variable p_{in} bezeichnet somit das weitergereichte Ergebnis der rechten Nachbarzelle aus dem vorhergehenden Takt. So wird zum Beispiel im 3. Takt in der Zelle 1 das Ergebnis der Zelle 2 aus dem 2. Takt ($p = x_1\, y_1$) weiterverarbeitet.

Takt	*Zelle 1*	*Zelle 2*	*Zelle 3*
1	x_0 – – y_0 – – $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = x_0 y_0$	– – – – – – $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 0$	– – – – – – $q = 0$ $\leftarrow p = 0$
2	x_0 x_1 – y_0 y_1 – $q_{aus} = x_1 y_1 \rightarrow$ $\leftarrow p_{aus} = x_0 y_1 + x_1 y_0$	– – – – – – $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = q_{in}$	– – – – – – $q_{aus} = 0$ $\leftarrow p_{aus} = 0$
3	x_0 x_1 x_2 y_0 y_1 y_2 $q_{aus} = x_1 y_2 + x_2 y_1 \rightarrow$ $\leftarrow p_{aus} = p_{in} + x_0 y_2 + x_2 y_0$	– – – – – – $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = q_{in}$	– – – – – – $q_{aus} = 0$ $\leftarrow p_{aus} = 0$
4	x_0 x_1 x_3 y_0 y_1 y_3 $q_{aus} = x_1 y_3 + x_3 y_1 \rightarrow$ $\leftarrow p_{aus} = p_{in} + x_0 y_3 + x_3 y_0$	x_2 – – y_2 – – $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = x_2 y_2 + q_{in}$	– – – – – – $q_{aus} = 0$ $\leftarrow p_{aus} = 0$
5	x_0 x_1 0 y_0 y_1 0 $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = p_{in}$	x_2 x_3 – y_2 y_3 – $q_{aus} = x_3 y_3 \rightarrow$ $\leftarrow p_{aus} = p_{in} + x_2 y_3 + x_3 y_2$	– – – – – – $q_{aus} = 0$ $\leftarrow p_{aus} = q_{in}$
6	x_0 x_1 0 y_0 y_1 0 $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = p_{in}$	x_2 x_3 0 y_2 y_3 0 $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = p_{in}$	– – – – – – $q_{aus} = 0$ $\leftarrow p_{aus} = 0$
7	x_0 x_1 0 y_0 y_1 0 $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = p_{in}$	x_2 x_3 0 y_2 y_3 0 $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = p_{in}$	0 – – 0 – – $q_{aus} = 0$ $\leftarrow p_{aus} = 0$
8	x_0 x_1 0 y_0 y_1 0 $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = p_{in}$	x_2 x_3 0 y_2 y_3 0 $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = p_{in}$	0 0 – 0 0 – $q_{aus} = 0$ $\leftarrow p_{aus} = 0$

Abbildung 5.11: „Funktionsweise der Zellen (systolischer Multiplizierer)“

Jedes Paar x_i, y_i hat also einen Speicherplatz in einer bestimmten Zelle des Feldes. Das im Takt i an Zelle 1 angelegte Paar x_i, y_i wandert in den folgenden Takten im Feld weiter nach rechts. Beim Passieren anderer Paare x_j, y_j ($0 \leq j < i$), die bereits ihren endgültigen Speicherplatz erreicht haben, werden gleichzeitig zwei Teilprodukte $x_{j'} \cdot y_i + x_i \cdot y_{j'}$ (j' gerade) und $x_{j''} \cdot y_i + x_i \cdot y_{j''}$ (j'' ungerade) gebildet. Dieses ist notwendig, weil in jedem Takt zwei Paare x_i, y_i passiert werden. Das Teilprodukt $x_{j'} \cdot y_i + x_i \cdot y_{j'}$ (j' gerade) wird zur aktuellen Spaltensumme addiert, die von dieser Zelle augenblicklich berechnet wird. Das Teilprodukt $x_{j''} \cdot y_i + x_i \cdot y_{j''}$ (j'' ungerade) wird an die rechte Nachbarzelle weitergegeben, damit es dort zur passenden Spaltensumme addiert wird. Wenn das Paar x_i, y_i seinen Speicherplatz (den ersten unbelegten) erreicht hat, wird das Teilprodukt $x_i \cdot y_i$ gebildet. Jedes dieser Teilprodukte wird zur entsprechenden Spaltensumme addiert, die dann zurück zu Zelle 1 wandert.

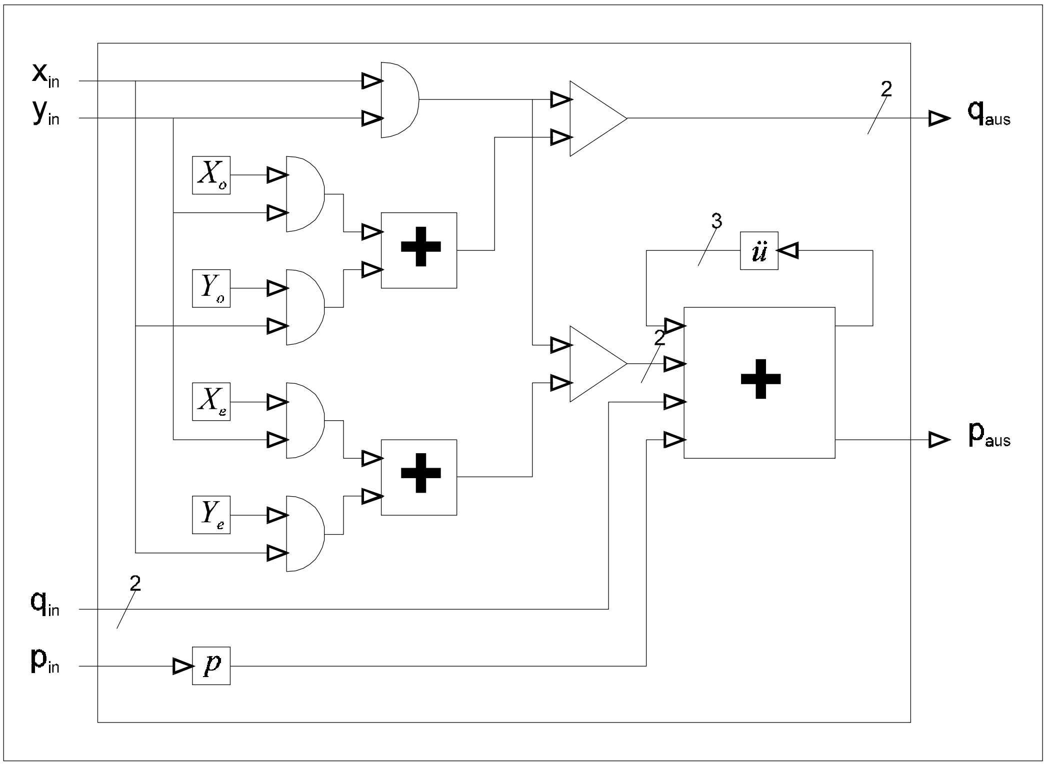

Abbildung 5.12: „Logik Summenteil“

Um mit jedem Takt eine Spaltensumme erzeugen zu können, müssen je zwei Paare x_i, y_i in einer Zelle gespeichert werden (Zur Erinnerung: Dies geschieht für gerade i in den Registern X_o und Y_o, sowie für ungerade i in den Registern X_e und Y_e). So wird beispielsweise x_j, y_j in einem Abstand von $j/2$ Zellen rechts von Zelle 1 gespeichert. Wird nun x_i, y_i im Takt i ($i > j$) an Zelle 1 angelegt, so hat x_i, y_i die Zelle, in der x_j, y_j gespeichert sind, nach $j/2$ Takten erreicht. Das jetzt entstehende, für p_{i+j} notwendige Teilprodukt hat nach weiteren $j/2$ Takten (also in Takt $j + i$) den Ausgang erreicht. Das Teilprodukt $x_2 \cdot y_2$ ist beispielsweise Bestandteil der Spaltensumme p_4 (siehe Abbildung 5.7: „Addition von Teilprodukten“) und muß also nach dem vierten Takt am Ausgang anliegen. Deshalb wird das Paar x_2, y_2 in Zelle 2 gespeichert.

Der folgende VHDL-Code zeigt die Verhaltensbeschreibung des Summenteils der Zelle.

```
-- In mit jeder Takt-Flanke erfolgt:
-- p  <= pin
-- ue <= s(3 downto 1)

xaus <= xt;
yaus <= yt;
paus <= s(0);
s <= ("00"&zs) + ("000"&p) + ('0'&ue) + ("00"&qin);

SUM: process(current_state,zin,xin,yin,xe,ye,x0,y0)
  begin
    if ((current_state=Z0) and (zin='1')) then
      qaus    <= "00";
      zs(0)   <= xin and yin;
      zs(1)   <= '0';
    elsif ((current_state=Z1) and (zin='0')) then
      qaus(0) <= xin and yin;
      qaus(1) <= '0';
      zs      <= (xin and ye) + (xe and yin);
    else
      qaus    <= (xin and yo) + (xo and yin);
      zs      <= (xin and ye) + (xe and yin);
    end if;
  end process;
```

Den groben Aufbau einer kompletten Zelle zeigt die Abbildung 5.13: „Aufbau einer Zelle des systolischer Multiplizierer“. An x_{in} und y_{in} werden die durch die Zellen von links nach rechts wandernden Bitpaare x_i, y_i angelegt. Diese werden entweder in X_e, Y_e bzw. X_o, Y_o abgespeichert oder im nächsten Takt über x_{aus} und y_{aus} an die rechte Nachbarzelle weitergegeben. Eines der beiden von der Zelle berechneten Teilprodukte wird über q_{aus} an die rechte Nachbarzelle weitergegeben. Das andere wird zur aktuellen Spaltensumme addiert und über p_{aus} an die linke Nachbarzelle weitergereicht.

Da die Auswahl des als nächstes zu belegenden Registers davon abhängt, welche Register bereits belegt wurden, benötigt jede Zelle einen Zustandsspeicher z, dessen Wert angibt, welches Register als nächstes zu belegen ist und welche Registerwerte zur Berechnung des Teilprodukts herangezogen werden sollen. Weiterhin benötigt jede Zelle zwei Register p und $ü$ zur Speicherung des Zwischenergebnisses der Spaltensumme, die von der rechten Nachbarzelle berechnet wurde und zur Speicherung des Übertrags. Das Register hat eine Breite von einem Bit, das Registers $ü$ drei Bit.

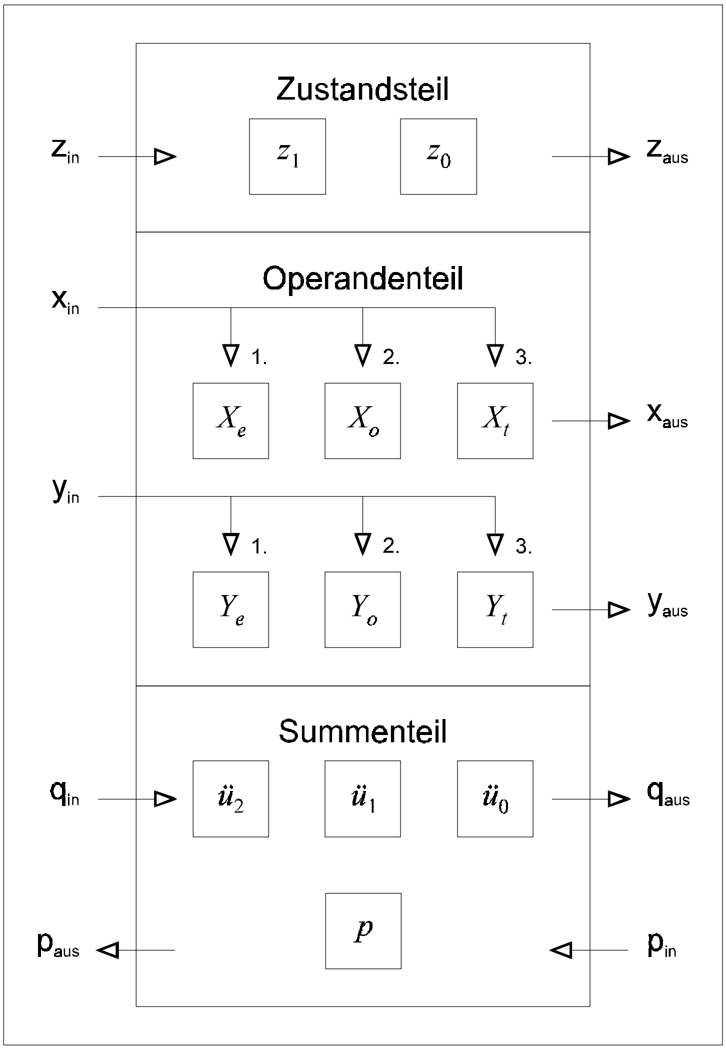

Abbildung 5.13: „Aufbau einer Zelle des systolischer Multiplizierer“

Der Zustandsteil ist sehr einfach aufgebaut. Er besteht aus einem Automaten mit vier Zuständen (siehe Abbildung 5.14: „Zustandsautomat (systolischer Multiplizierers)“). Der Startzustand ist Z_0. Als Signal für den Beginn einer neuen Multiplikation wird einen Takt lang eine ‘1’ an z_{in} angelegt. Die Register X_e, Y_e und X_o, Y_o werden dann als nicht belegt angenommen. Nachdem die Belegung der Register von X_e, Y_e und X_o, Y_o in den nächsten zwei Takten erfolgt ist, wird diese ‘1’ im Zustand Z_3 über z_{aus} an die rechte Nachbarzelle weitergegeben, die dann wiederum ihre Register X_e, Y_e und X_o, Y_o belegt.

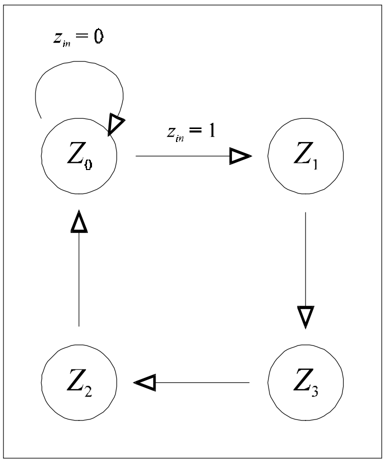

Abbildung 5.14: „Zustandsautomat (systolischer Multiplizierers)“

5.1.5 Aufbau des systolischen Multiplizierers aus Zellen

Der Multiplizierer besteht aus einer linearen Anordnung von Zellen, die zentral getaktet werden und ein systolisches Feld bilden. Die Anzahl der benötigten Zellen hängt von der maximalen Operandengröße ab. Wenn n die maximale Stellenzahl der im Dualsystem dargestellten Operanden ist, ergibt sich die Anzahl m, der für das Feld benötigten Zellen, zu

$$m = \left\lceil \frac{n}{2} \right\rceil + 1 .$$

Ein solches Feld kann alle Operanden mit einer Stellenzahl kleiner oder gleich n verarbeiten. Die benötigte Zahl von Taktimpulsen ist nicht von der Anzahl der Zellen im Feld, sondern von der aktuellen Operandengröße n_{akt} abhängig. Haben die Operanden eine ungleiche Stellenzahl, so müssen die kürzeren Operanden durch führende Nullen auf die Länge des längsten Operanden gebracht werden. Die benötigte Taktzahl t_{akt} beträgt

$$t_{akt} = 2 \cdot \left\lceil \frac{n_{akt}}{2} \right\rceil - 1 .$$

Das niederwertige Bit des Eingangs q_{in} von Zelle 1 kann genutzt werden, um zum Produkt $x \cdot y$ einen Wert q addieren zu können. Die benötigte Taktzahl kann dabei um eins erhöht sein, weil die Stellenzahl des Ergebnisses unter Umständen um eins höher ist.

Zusammenfassend folgt noch einmal die Funktionsweise des systolischen Feldes zur Berechnung von $p = x \cdot y + q$. Die Operanden x und y werden mit dem LSB voran in n_{akt} Takten seriell in das Array geschoben. Dabei wird pro Takt ein Bit nachgezogen. In den nächsten Takten folgen Nullen. Gleichzeitig zu x und y kann q seriell angelegt werden. Das LSB des Ergebnisses erscheint bereits nach einem Takt am Ausgang p, die weiteren Bits des Ergebnisses folgen in den nächsten Takten, so daß die Multiplikation nach $2 \cdot n_{akt}$ Takten abgeschlossen ist. Es folgt ein Beispiel mit den Werten $x = 5$ (0101) und $y = 14$ (1110). Zur Vereinfachung wurde $q = 0$ gesetzt. Als Ergebnis erwartet man $p = 5 \cdot 14 = 70$ (1000110).

Takt		*Zelle 1*	*Zelle 2*	*Zelle 3*
0	x=0101 y=1110	Z_0 – – – – – – – –	– – – – – – – – –	– – – – – – – – –
1	x=010 y=111	Z_1 1 – – 0 – – $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 0$	– – – – – – – – –	– – – – – – – – –
2	x=01 y=11 $p = 0$	Z_2 1 0 – 0 1 – $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 1$	– – – – – – – – –	– – – – – – – – –
3	x=0 y=1 $p = 10$	Z_3 1 0 1 0 1 1 $q_{aus} = 1 \rightarrow$ $\leftarrow p_{aus} = 1$	Z_0 – – – – – – $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 1$	– – – – – – – – –
4	$p = 110$	Z_0 1 0 0 0 1 1 $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 2$	Z_1 1 – – 1 – – $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 1$	– – – – – – – – –
5	$p = 0110$	Z_0 1 0 0 0 1 0 $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 2$	Z_2 1 0 – 1 1 – $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 1$	– – – – – – – – –
6	$p = 00110$	Z_0 1 0 0 0 1 0 $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 2$	Z_3 1 0 0 1 1 0 $q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 0$	Z_0 – – – – – – $q_{aus} = 0$ $\leftarrow p_{aus} = 0$

7		Z_0	Z_0	Z_1
		1 0 0 0 1 0	1 0 0 1 1 0	0 – – 0 – –
	$p = 000110$	$q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 1$	$q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 0$	$q_{aus} = 0$ $\leftarrow p_{aus} = 0$
8		Z_0	Z_0	Z_2
		1 0 0 0 1 0	1 0 0 1 1 0	0 0 – 0 0 –
	$p = 1000110$	$q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 0$	$q_{aus} = 0 \rightarrow$ $\leftarrow p_{aus} = 0$	$q_{aus} = 0$ $\leftarrow p_{aus} = 0$

Abbildung 5.15: „Beispiel systolischer Multiplizierer“

5.1.6 Systolische Arrays zur Berechnung des Montgomery-Produktes

Das beschriebene systolische Feld kann laut Sauerbrey [SAU93] auf einfache Weise zur Modulo-Multiplikation nach Montgomery verwendet werden. Sowohl die Multiplikation als auch die Reduktion sollen mit Hilfe eines der zuvor beschriebenen systolischen Arrays durchführbar sein. Diese Aussage von Sauerbrey ist nicht korrekt. Wie im folgenden gezeigt wird, sind zur Reduktion zwei der erwähnten Arrays nötig. Dieses hat eine Erhöhung des Flächenbedarfs um 50 Prozent zur Folge.

Das Kernprinzip des Algorithmus zur Berechnung des Montgomery-Produktes ist es, ein Vielfaches v so zu bestimmen, daß

$$t = \frac{x \cdot y + v \cdot k}{r}$$

ganzzahlig wird (siehe hierzu Abbildung 5.1: „Algorithmus Montgomery-Produkt (1)“ auf Seite 61). Um dieses zu erreichen, ist v wie folgt zu bestimmen:

$$v = (x \cdot y \cdot k\,') \bmod r$$

Beim der Realisierung nach Sauerbrey wird hingegen v mit

$$v = (x \cdot y) \bmod r$$

falsch berechnet. Sauerbrey führt aus, daß beginnend mit der niederwertigsten Stelle des Produktes, jeweils das v_i-fache von k gefunden werden muß, das nach der Addition zum Produkt $p = x \cdot y$ die jeweilige niederwertigste Stelle zu Null werden läßt. Ist das Produktbit p_i bekannt, kann das entsprechende Bit v_i durch Multiplikation von p_i mit der Konstanten k_0' (im vorliegenden Fall ist $k_0' = 1$) gebildet werden. Dieses ist nicht korrekt. Es muß vielmehr $v = p \cdot k\,'$ berechnet werden. Es ist nicht ausreichend, die einzelnen Bits v_i von v mit eins zu multiplizieren. Die Abbildung 5.16: „Blockschaltbild Montgomery-Produkt gemäß Sauerbrey“ zeigt die unkorrekte Verschaltung zweier systolischer Multiplizierer wie sie Sauerbrey beschreibt.

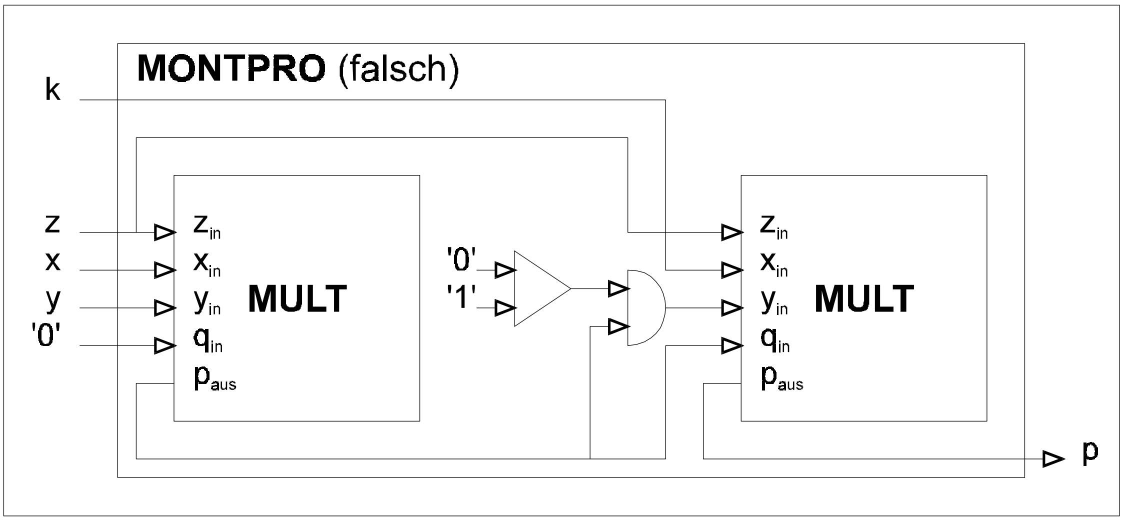

Abbildung 5.16: „Blockschaltbild Montgomery-Produkt gemäß Sauerbrey“

In Abbildung 5.17: „Korrigierte Komponente Montgomery-Produkt“ ist der verbesserte Aufbau zur Berechnung des Montgomery-Produktes dargestellt. Die Abbildung zeigt die Verschaltung von drei systolischen Multiplizierern. Der linke Multiplizierer führt die Multiplikation

$$p = x \cdot y$$

aus, während der mittlere und der rechte Multiplizierer im Fließbandverfahren (pipelining) die Modulo-Reduktion durchführen. Sie berechnen

$$v = p \cdot k' \text{ und } t = v \cdot k.$$

Die Division t durch r wird berücksichtigt, indem nur die 1024 höchstwertigen Stellen des Ergebnisses betrachtet werden.

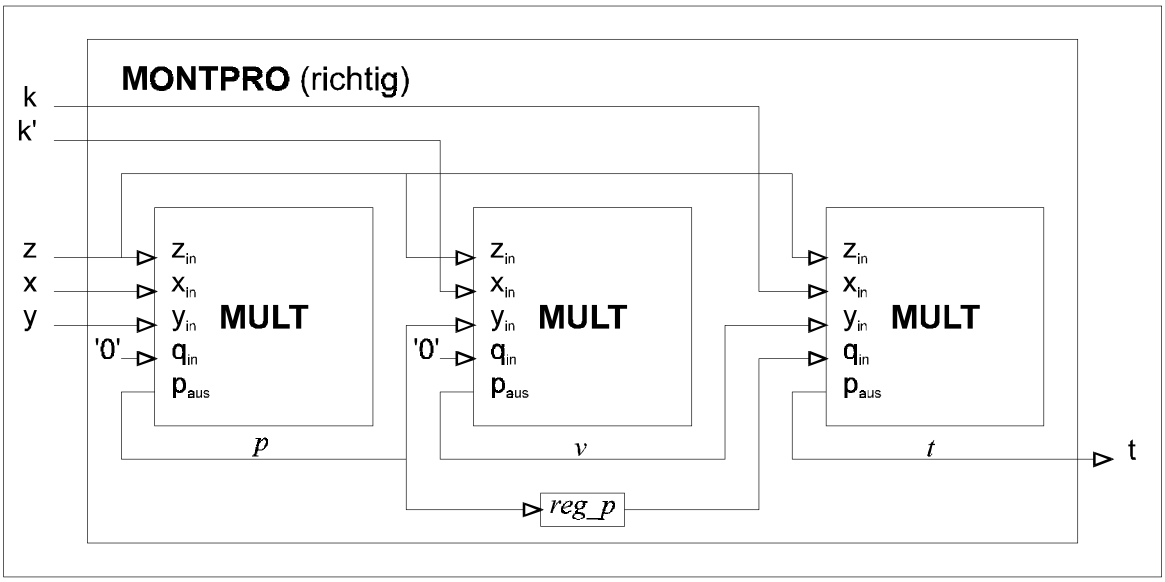

Abbildung 5.17: „Korrigierte Komponente Montgomery-Produkt“

Da die systolischen Multiplizierer ohne Latenzzeit auskommen (das niederwertigste Bit des Ergebnis steht bereits nach einen Takt zur Verfügung), können die drei Multiplizierer im Pipelinebetrieb arbeiten. Das niederwertigste Bit des Montgomery-Produktes ist somit nach drei Takten berechnet. Da in jedem weiterem Takt ein Bit der 1024-stelligen Operanden verarbeitet wird, benötigt der Modulo-Multiplizierer 2048 Takte für eine komplette Modulo-Multiplikation nach Montgomery. Die ersten 1024 Bit des Ergebnisses sind wie beabsichtigt alle Null. Das LSB des eigentlichen Ergebnisses liegt nach 1024 + 3 = 1027 Takten vor, so daß ein weiterer Modulo-Multiplizierer im Fließbandverfahren mit diesem Bit eine weitere Modulo-Multiplikation beginnen könnte. Mit zwei Modulo-Multiplizierern werden also im Dauerbetrieb 1027 Takte pro Modulo-Multiplikation benötigt.

5.1.7 Aufbau der Modulo-Exponentiationseinheit (Montgomery)

Unter Verwendung eines oder mehrerer Modulo-Multiplizierer kann eine universelle Exponentiationseinheit aufgebaut werden. Diese führt nacheinander oder parallel eine Folge von Modulo-Multiplikationen mit den entsprechenden Operanden aus, so daß eine Modulo-Exponentiation durchgeführt wird. Operanden und Zwischenergebnisse können in einem oder mehreren Schieberegistern abgelegt werden. Die Abbildung 5.18 zeigt das Blockschaltbild einer solchen Modulo-Exponentiationseinheit.

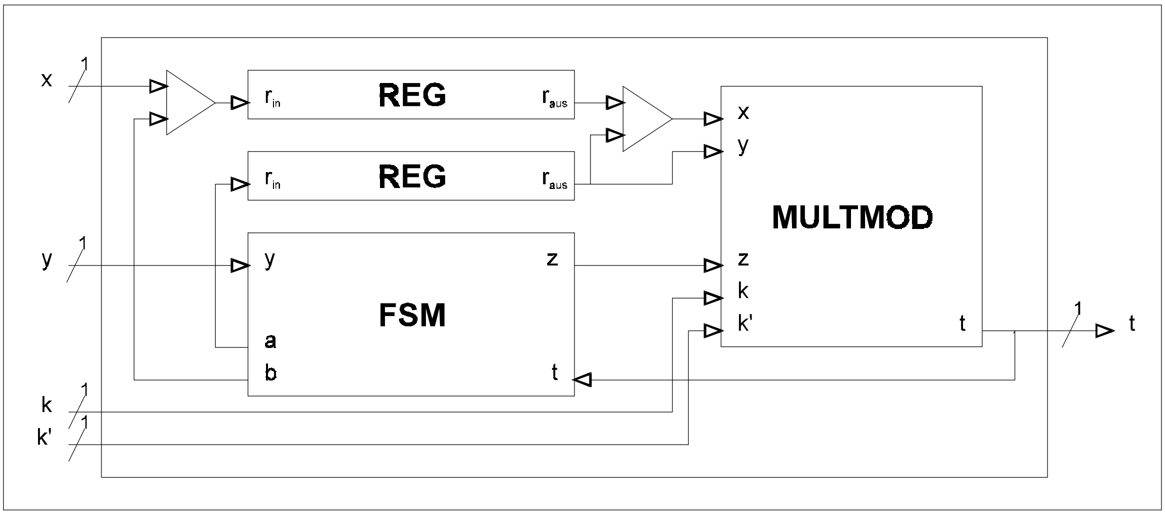

Abbildung 5.18: „Blockschaltbild Montgomery“

5.2 „Carry-Save“-Addierer zur Modulo-Multiplikation

Eine mögliche Methode zur Multiplikation ganzer Zahlen ist ein Multiplizierer, dem ein Addierer nach dem „Carry-Save“-Prinzip zugrunde liegt, da dieses für die Multiplikation besonders gut geeignet ist. In einem Takt können der neue Summand und die Überträge der vorherigen Addition zum aktuellen Zwischenergebnis addiert werden. Von Vorteil sind insbesondere die geringen Signallaufzeiten. Nachteilig wirkt sich der hohe Hardwareaufwand aus, der entsteht, weil ein zusätzliches Register zur Speicherung der Überträge notwendig ist.

Die deutlich reduziert Signallaufzeiten ergeben sich durch die wesentlich einfachere Logik (im Vergleich zum „Carry-Lock-Ahead“-Prinzip), die zur Berechnung der Zwischenergebnisse und des Überträge notwendig ist (siehe 4.4.2 „Carry-Save“-Multiplizierer auf Seite 51). Ein reiner Multiplizierer nach dem „Carry-Save“-Prinzip ist relativ einfach zu realisieren. Als schwierig erweist sich die Integration der Modulo-Reduktion unter der Veraussetzung, daß die Lösung in bezug auf ihr Platz- und Zeitverhalten möglichst effizient verhält.

Im folgenden wird zunächst die allgemeine Funktionsweise eines „Carry-Save“-Multiplizierers erläutert, danach die Erweiterung für die Modulo-Reduktion. Schließlich wird die Funktionsweise der einzelnen Zelle und deren Zusammenspiel erläutert.

$$a = x \cdot y = x_3x_2x_1x_0 \cdot y_3y_2y_1y_0$$

$$a_i = a_i \text{ XOR } c_{i-1} \text{ XOR } x_i$$

$$c_i = c_{i-1}a_i \text{ OR } c_{i-1}x_i \text{ OR } a_ix_i$$

Register (Zwischen-)Ergebnis	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
Register Übertrag	c_6	c_5	c_4	c_3	c_2	c_1	c_0	

Abbildung 5.19: „Register ‘Carry-Save’“

5.2.1 Funktionsweise des „Carry-Save“-Multiplizierer

Ein „Carry-Save“-Multiplizierer multipliziert zwei n Bit große Zahlen x und y zu einem $2n$ Bit großen Ergebnis a. Benötigt werden zwei Register: Eines mit einer Breite von $2n$ Bit für das (Zwischen-)Ergebnis und eines mit einer Breite von $2n$ - 1 für die Überträge. Es wird zunächst in n Takten zum Zwischenergebnis a der Operand x in Abhängigkeit vom zweiten Operanden y hinzuaddiert. Vor jeder weiteren Addition wird das Zwischenergebnis a um ein Bit nach links geschoben. Mit jeder neuen Addition werden die Überträge der vorhergehenden Addition verarbeitet. Nach den n Additionen sind maximal $2n$ weitere Takte nötig, um alle Überträge zu verarbeiten, da sich im schlechtesten Fall ein Übertrag aus der niederwertigsten Stelle bis in die höchstwertigste Stelle des Ergebnisses auswirkt. Im Prinzip wird eine Multiplikation mit verzögerter Verarbeitung des Übertrages angewandt.

Es folgt ein Beispiel mit zwei Operanden der Größe vier Bit. Das Register für das Zwischenergebnis benötigt somit acht Bit, das Register für die Überträge sieben Bit. Es wird abwechselnd eine Addition und eine Shift-Operation ausgeführt.

```
x · y = 12 · 13 = 156
x = 1100
y = 1101

a = 00000000   1. Takt: init
c = 0000000

a = 00001100   2. Takt: add c, add x (da y3 = 1)
c = 0000000

a = 00011000   3. Takt: shift
c = 0000000

a = 00010100   4. Takt: add c, add x (da y2 = 1)
c = 0001000

a = 00101000   5. Takt: shift
c = 0010000

a = 00001000   6. Takt: add c, add 0 (da y1 = 0)
c = 0100000

a = 00010000   7. Takt: shift
c = 1000000

a = 10011100   8. Takt: add c, add x (da y0 = 1)
c = 0000000
```

In jedem zweiten Takt wird *x* zum Zwischenergebnis addiert. Nur im sechsten Takt wird eine Null addiert, weil das Bit 1 von *y* den Wert '0' besitzt. In diesem Takt wird somit nur der Übertrag *c* zum Zwischenergebnis *a* addiert.

5.2.2 Erweiterung zum Modulo-Multiplizierer („Carry-Save")

Bei der Ausführung der Modulo-Multiplikation

$$a = x \cdot y \bmod k$$

muß das Zwischenergebnis immer wieder einer Modulo-Reduktion unterzogen werden, damit das Ergebnis nicht zu groß wird.

Eine wichtige Entscheidung war, wie häufig diese Reduktion durchzuführen ist. Das Endergebnis muß schließlich kleiner als *k* sein. Die Zwischenergebnisse können aber beliebig viel größer sein. Der Nachteil von großen Zwischenergebnissen ist, daß entsprechend größere Register benötigt werden. Der Vorteil ist, daß eine Reduktion seltener erfolgen muß. Die Multiplikation wird somit schneller ausgeführt. Es ist somit – wie so oft – ein Kompromiß zwischen Kosten und Geschwindigkeit zu treffen. Da schon nach den ersten Abschätzungen feststand, daß die Größe des Chips an der äußeren Grenze oder sogar über ihr liegen wird, fiel die Entscheidung für eine häufige Reduktion des Ergebnisses.

Die Reduktion wurde mit Hilfe einer Subtraktion durchgeführt. In Abhängigkeit von der Größe des Zwischenergebnisses wurde entweder k oder $2k$ abgezogen. Die Größe des Zwischenergebnisses wurde mit Hilfe der höchstwertigen drei Bits abgeschätzt, weil ein Größenvergleich über mehrere Stelle zu weiterer Hardware geführt hätte. Stellt sich in einem der folgenden Takte heraus, daß das Zwischenergebnis kleiner als Null geworden ist, wird zur Korrektur einfach wieder der Modulus k hinzuaddiert.

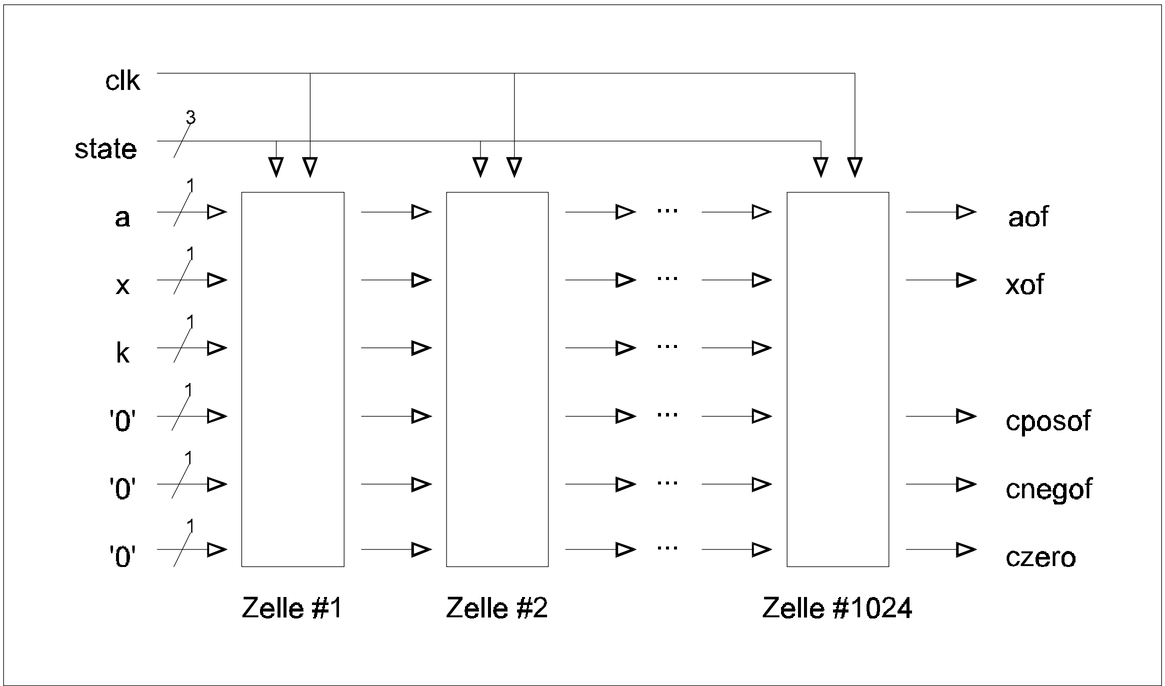

Abbildung 5.20: „Aufbau Multiplikationseinheit 'Carry-Save'"

5.2.3 Aufbau des „Carry-Save"-Multiplizierers

Die Implementierung des Multiplizierers nach dem „Carry-Save"-Prinzip besteht ebenso wie die Realisierung nach Montgomery aus einer linearen Anordnung von Zellen, die zentral getaktet werden. Jede Zelle ist nur mit den beiden direkten Nachbarzellen verbunden. Weiterhin ist ein drei Bit breites zentrales Steuersignal nötig. Dieses erhöht den Verdrahtungsaufwand gegenüber der Lösung gemäß Montgomery erheblich. Die Anzahl der Zellen hängt wiederum von der Operandengröße ab. Jede Stelle der im Dualsystem dargestellten Zahlen wird von einer Zelle bearbeitet. Für eine maximale Operandenlänge von 1024 Bit benötigt man also 1024 Zellen. Die benötigte Zahl von Taktimpulsen ist von der Anzahl der Zellen und von der Beschaffenheit der Operanden abhängig.

Jede Zelle enthält sechs Flip-Flops. Fünf dieser sechs Flip-Flops dienen zur Aufnahme jeweils eines Bits eines imaginären großen Registers mit einer Breite von 1024 Bit. Das Register a dient der Speicherung des (Zwischen-)Ergebnisses. In k wird der jeweilige Modulus abgelegt. In x steht der Multiplikator. In *cpos* und *cneg* werden ein möglicher positiver bzw. negativer Übertrag gespeichert. Das sechste Flip-Flop *czero* enthält den Wert Null, wenn die Überträge dieser und aller rechts von ihr stehenden Zellen Null sind. Es wird benötigt, um das Ende einer Multiplikation zu erkennen.

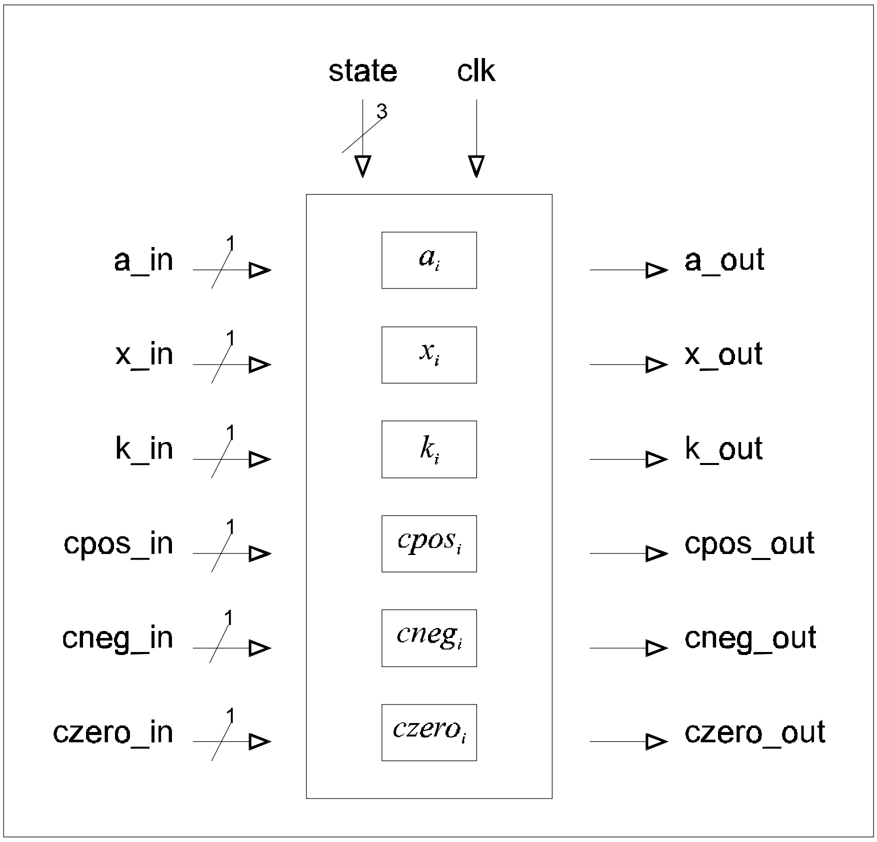

Abbildung 5.21: „Aufbau einer Zelle ‘Carry-Save’“

Jede Zelle kann acht verschiedene Befehle ausführen (siehe Abbildung 5.22: „Befehle Zelle ‘Carry-Save’“). Grundsätzlich kann jede Zelle addieren und subtrahieren. Es kann der Modulus *k* sowie das Doppelte des Modulus *k* addiert und subtrahiert werden. Weiterhin kann der Multiplikator *x* addiert werden. Jede Zelle kann das Ergebnis *a* um eine Stelle nach links schieben, um die nächste Addition des Multiplikators *x* vorzubereiten. Schließlich gibt es noch einen Zustand zum Laden von *k* und *x*, sowie einen Zustand in dem nur die Überträge *cpos* und *cneg* verarbeitet werden, ohne daß eine weitere Addition oder Subtraktion durchgeführt wird.

000	add k1	Addition von k
001	add k2	Addition von 2k
010	add x	Addition von x
011	sub k1	Subtraktion von k
100	sub k2	Subtraktion von 2k
101	shift	schieben des Ergebnisses nach links
110	laden	laden von k und x
111	warten	warten bzw. Verarbeitung der Überträge

Abbildung 5.22: „Befehle Zelle ‘Carry-Save’“

5.2.4 Aufbau der Modulo-Exponentiationseinheit („Carry-Save“)

Um eine komplette Modulo-Exponentiationseinheit nach dem „Carry-Save“-Prinzip aufbauen zu können, werden zunächst zwei Modulo-Multiplizierer benötigt. Diese sind nötig, um beide Multiplikationen des Algorithmus „Square-and-Multiply“ parallel ausführen zu können. Selbstverständlich ist theoretisch auch die sequentielle Ausführung der Multiplikationen möglich. Dieses hat den Vorteil, daß sich die Chipfläche verringert. Gleichzeitig bedeutet dieses im schlechtesten Fall eine Einbuße an Geschwindigkeit um den Faktor zwei.

Weiterhin werden zwei Schieberegister mit einer Breite von 1024 Bit für die Zwischenergebnisse der Multiplikation benötigt. Schließlich benötigt man noch einen zentralen Steuerungsautomaten, der für die korrekte Durchführung des „Binärverfahrens“ sorgt. Er übernimmt ebenso das Laden der Eingangswerte und die Ausgabe der Ergebnisse.

5.2.5 Verbesserungen des „Carry-Save“-Multiplizierers

In einem weiterem Entwicklungsschritt wurden weitere Verbesserungen an dem „Carry-Save“-Entwurf vorgenommen. Die Verbesserungen zielten auf eine Verringerung des Platzbedarfs bei unverändert hoher Leistung.

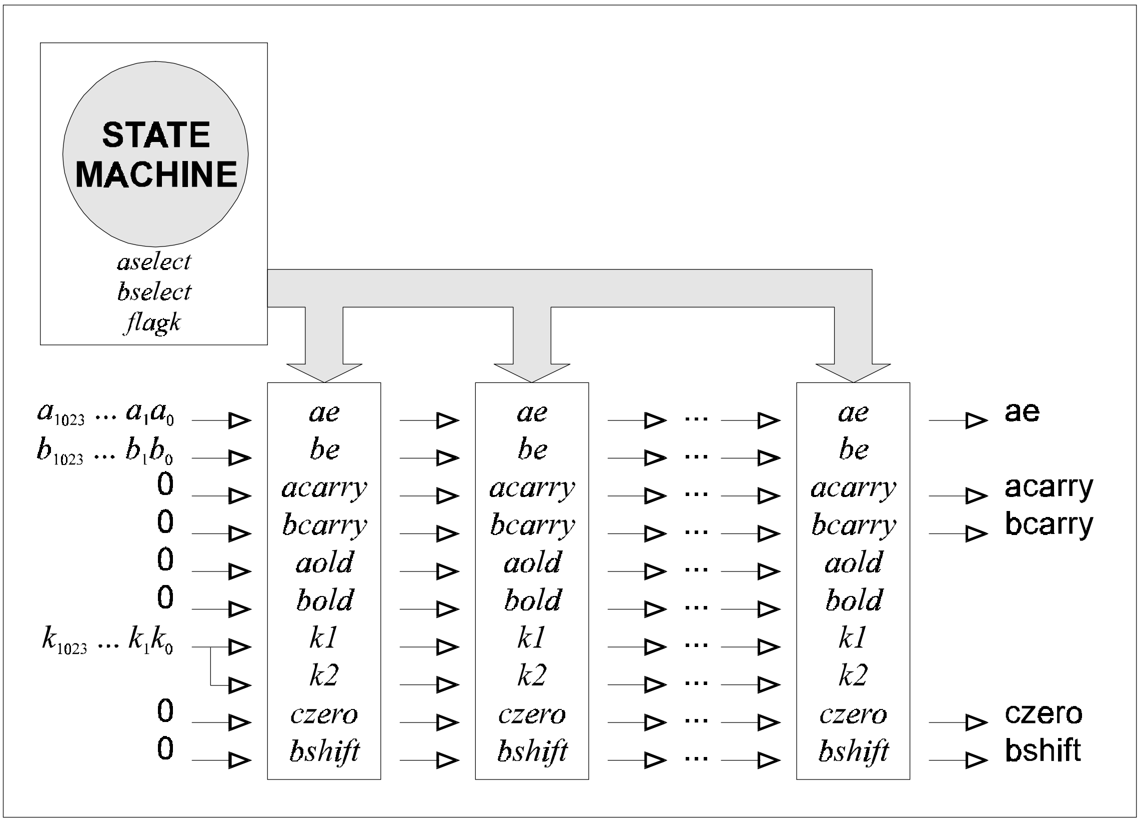

Abbildung 5.23: „Verbessertes systolisches Array (‘Carry-Save’)“

Bisher wurde die Modulo-Reduktion mit Hilfe von Subtraktionen des Modulus *k* durchgeführt. Um dieses zu bewerkstelligen, wurde eine quasi „doppelt redundante“ Zahlendarstellung eingeführt. Jedes Bit des Zwischenergebnisses wurde durch ein „Drei-Bit-Register“ repräsentiert: Ein Bit für das eigentliche Zwischenergebnis und jeweils eines für den positiven und den negativen Übertrag. Der neue Entwurf führt die Modulo-Reduktion durch die Addition des Zweier-Komplements des Modulus *k* durch. Auf diese Weise konnte das Register *cneg* für den negativen Übertrag eingespart werden.

Weiterhin wurde ein „Überlauf-Puffer“ eingeführt. Dieser ermöglicht es, die Modulo-Reduktion deutlich seltener als bisher durchzuführen. Überträge, die bei den Additionen auftreten, werden „gesammelt“ und später gemeinsam reduziert. Durch den „Überlauf-Puffer“ erhöht sich zwar der Platzbedarf, die Anzahl der benötigten Taktzyklen sinkt zugleich aber deutlich. Die Größe des Puffers wurde mit 16 Bit als ausreichend angenommen. Im Zuge einer weiteren Verbesserung der Methode wäre es sinnvoll, Untersuchungen anzustellen, bei welcher Puffergröße eine optimales Verhältnis zwischen Platz und Geschwindigkeit erreicht ist.

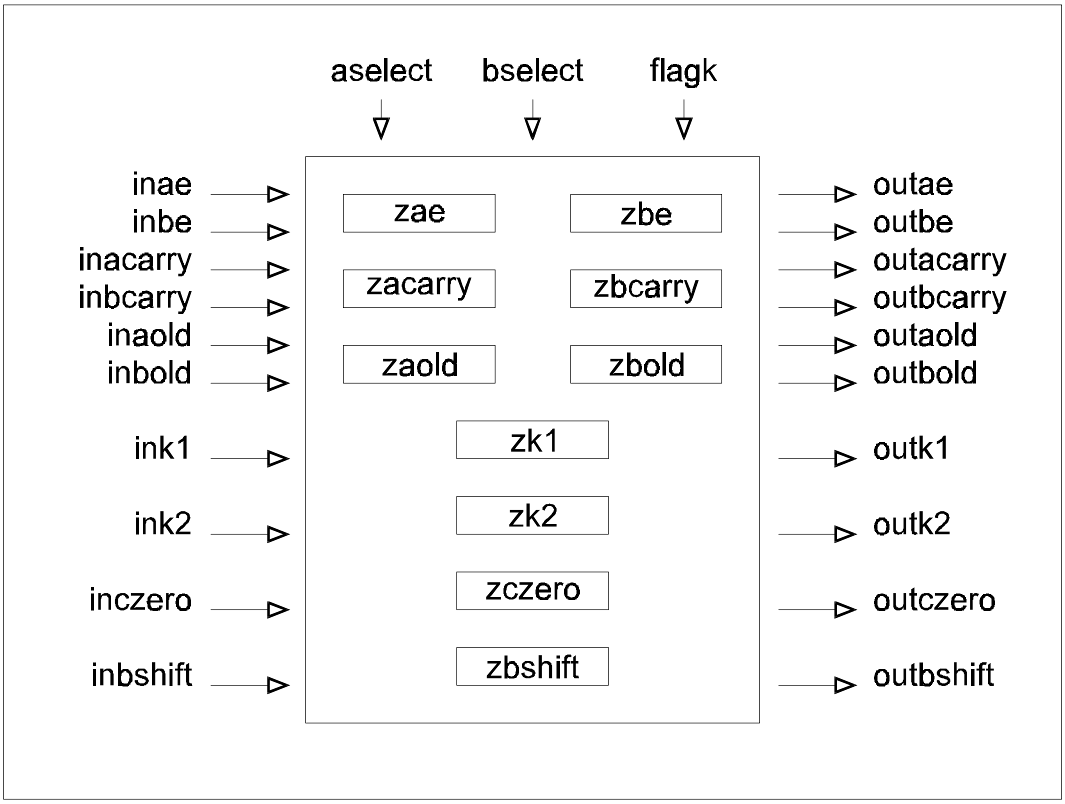

Abbildung 5.24: „doppelte Zelle 'Carry-Save'"

Die größte Einsparung an Chipfläche konnte durch die Einführung der „doppelten Zelle" erzielt werden (siehe Abbildung 5.24: „doppelte Zelle 'Carry-Save'"). In Zuge des verwendeten „Square-and-Multiply"-Verfahrens werden zwei Multiplikationen parallel ausgeführt. In dem ersten Entwurf wurden deshalb zwei völlig unabhängig voneinander arbeitende Modulo-Multiplizierer eingesetzt. Dieses führte dazu, daß das Register für den Modulus k und das Register *czero* zum Feststellen, ob die Multiplikationen bereits terminiert ist, doppelt auf dem Chip vorhanden waren. Viel sinnvoller ist es aber, diese Register von beiden Multiplizierern gemeinsam nutzen zu lassen. Aus diesem Grund wurde eine neue Zelle entwickelt, die zwei Modulo-Multiplikationen parallel ausführen kann. Beide Multiplikationen arbeiten nicht nur mit dem gleichen Modulus k, sondern benutzen auch den gleichen Multiplikator b_{shift}:

$$a = (a_{old} \cdot b_{shift}) \bmod k$$
$$b = (b_{old} \cdot b_{shift}) \bmod k$$

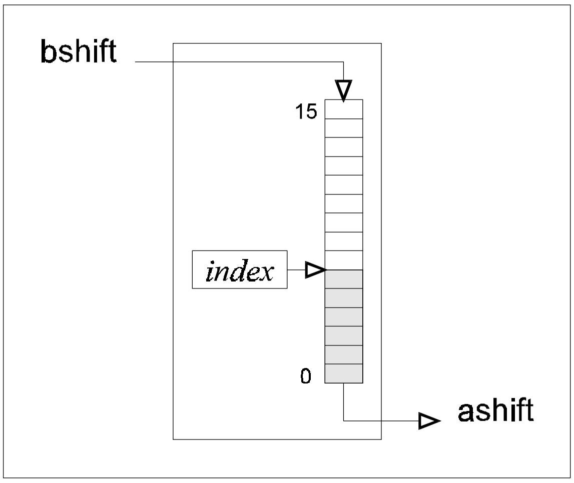

Abbildung 5.25: „Shift-Buffer“

Dieser Umstand machte eine weitere Erweiterung notwendig. Der Multiplikator b_{shift} wird sequentiell bitweise abgearbeitet. Jedes Bit geht nach seiner Verwendung verloren. Dieses erforderte einen weiteren Puffer. Der „Shift-Buffer“ speichert die von dem einen Multiplizierer bereits abgearbeiteten Bits von b_{shift} für den anderen Multiplizierer zwischen (siehe Abbildung 5.25: „Shift-Buffer“). Der Puffer arbeitet nach dem FIFO-Prinzip (first in - first out). Dieses bedeutet, daß ein Bit, das zuerst im Puffer abgelegt wurde, auch zuerst wieder ausgegeben wird.

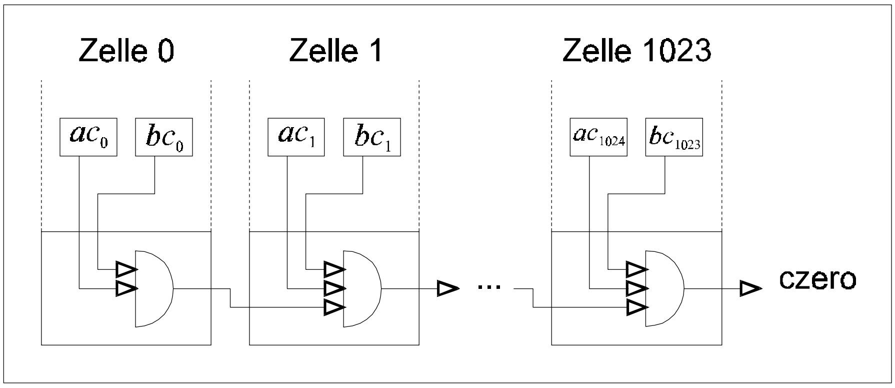

Abbildung 5.26: „Signal *czero*“

Als letztes wurden die Flip-Flops für das Signal *czero* eingespart. Das Signal *czero* hat den Wert Null, wenn alle Überträge verarbeitet wurden. In der ersten Version der „Carry-Save“-Implementation würde dieses Signal in jeder Zelle in einem Flip-Flop zwischen gespeichert. Nachteil war, daß das Signal 1024 Takte benötigte, bis es von der Zelle 0 bis zur Zelle 1023 gewandert war. Nach Entfernen der Flip-Flops entstand ein sehr langer Signalpfad. In jeder Zelle wird zur Berechnung des Signals *czero* ein dreifaches OR-Gatter benötigt. Das entsprechende Bauteil (`LIBOR3`) der ES2 Standardzell-Bibliothek hat eine Zellenverzögerungszeit von 1,28 ns. Dieses bedeutet bei 1024 aufeinander folgenden Bauteilen eine Verzögerungszeit von insgesamt 1310,72 ns. Die Folge wäre eine Taktrate von weniger als 1 MHz für den Chip gewesen. Alle anderen Signallaufzeiten gestatten eine maximale Taktfrequenz von weit über 100 MHz. Eine mögliche Lösung ist, den Chip mit 100 MHz zu takten. Wenn nun überprüft werden soll, ob alle Überträge verarbeitet wurden, wird einfach einige Takte auf das Signal *czero* gewartet. Bei einer Taktrate von 100 MHz dauert jeder Taktzyklus 10 ns an. Nach 132 Takten (bzw. 1320 ns) ist das Signal in Zelle 1023 angekommen. Fall es den Wert Eins hat, müssen weitere Taktzyklen abgewartet werden, bis es den Wert Null annimmt. Dieser Fall tritt nur dann ein, wenn sich ein Übertrag über mehr als 132 Stellen fortpflanzt. Die Wahrscheinlichkeit hierfür beträgt etwa $3{,}27 \cdot 10^{-37}$. Der Fall wird in Praxis somit niemals eintreten.

Eine alternative Implementierung wäre die Berechnung des Signal *czero* durch ein mehrstufiges Schaltnetz (siehe Abbildung 5.27: „Signal *czero* (2)“). Bei 1024 Zellen wären ld(1024) = 10 Stufen nötig. Der Wert für *czero* wäre auf diese Weise nach 12,8 ns berechnet. Die Wahl fiel nicht auf diese Möglichkeit der Implementierung, weil sie nicht im Rahmen des Konzeptes des systolischen Arrays zu realisieren ist.

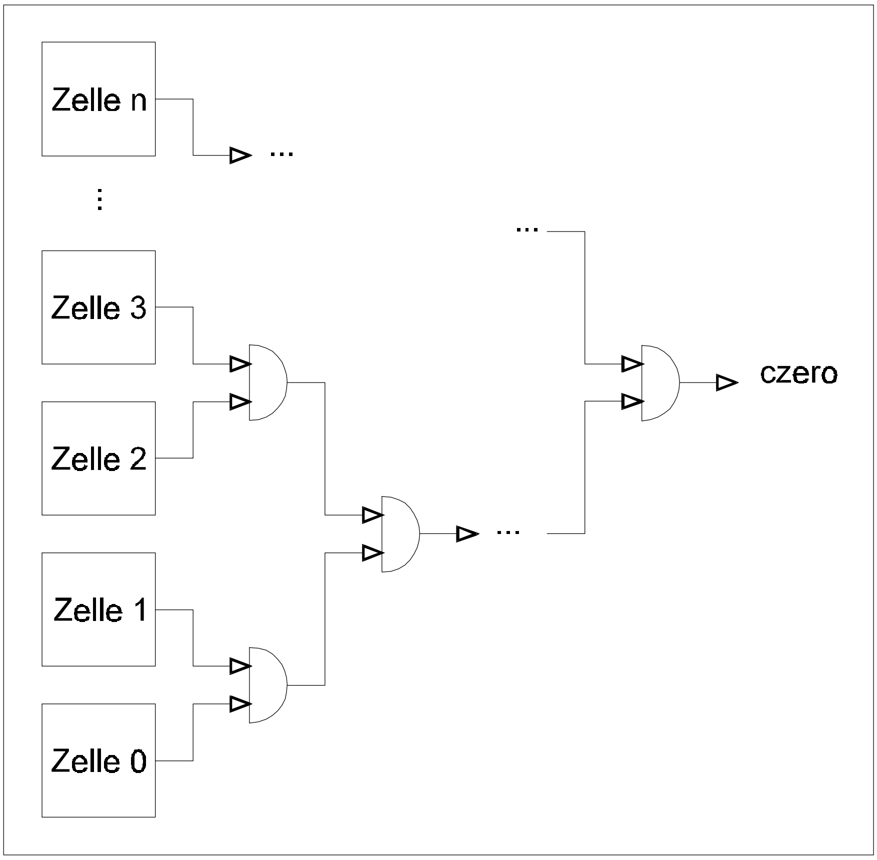

Abbildung 5.27: „Signal *czero* (2)“

Für jede Modulo-Multiplikation werden im Schnitt 1732 Operationen benötigt (siehe Abbildung 5.28: „Anzahl Operationen (‘Carry-Save’)“). Da jede Operation in einem Taktzyklus ausgeführt wird, werden somit 1732 Takte pro Modulo-Multiplikationen bei der Methode „Carry-Save“ benötigt.

	Anzahl
Additionen (Mittelwert)	512
Subtraktionen (Mittelwert)	64
Schiebeoperationen	1024
Korrekturzyklen (bei 100 MHz)	132
Summe Operationen	**1732**

Abbildung 5.28: „Anzahl Operationen (‘Carry-Save’)“

Es folgt ein Auszug aus der Verhaltensbeschreibung der „doppelten Zelle“ in VHDL. Die Werte für *zae*, *zacarry* und *zaold* werden mit exakt der gleichen Logik berechnet wie die Werte *zbe*, *zbcarry* und *zbold*. Aus diesem Grund wurde auf eine (doppelte) Auflistung des VHDL-Codes verzichtet. Die Ausgänge aller Register werden ohne weitere Logik direkt an die entsprechenden Ausgänge weitergeleitet (z.B. Register *zae* an Ausgang *zaout*). Der komplette VHDL-Code findet sich im Anhang.

```
B: process(clk,bselect,flagk)
  begin
    if (clk'event and clk='1') then -- positive Flanke
      case bselect is
        when "00" => zbe <= zbe XOR inbcarry XOR xb0;
                     zbcarry <= (zbe AND inbcarry) OR
                                (xb0 AND inbcarry) OR
                                (xb0 AND zbe);
        when "01" => zbe <= zbe XOR inbcarry XOR x1;
                     zbcarry <= (zbe AND inbcarry) OR
                                (x1 AND inbcarry) OR
                                (x1 AND zbe);
        when "10" => zbe <= inbe;
                     zbcarry <= inbcarry;
                     zbshift <= inbshift;
                     if (flagk='1') then
                       zk1 <= ink1;
                       zk2 <= ink2;
                     end if;
        when others => if (flagk='1') then
                       zbcarry <= '0';
                       zbold <= zbe;
                       zbshift <= zbe;
                       if (aselect="11") then
                         zbe <= '0';
                       end if;
                     end if;
      end case;
    end if;
  end process;
```

Die Abbildung 5.29: „Blockschaltbild (‘Carry-Save')“ zeigt das Blockschaltbild für die Implementation „Carry-Save“. Um das Bild übersichtlich zu gestalten, wurde für die weniger wichtigen Signale keine Verbindungen eingezeichnet. Es stehen bei jeder Komponente links die Eingänge und rechts die Ausgänge. Signale mit einem gleichen Namen sind immer verbunden.

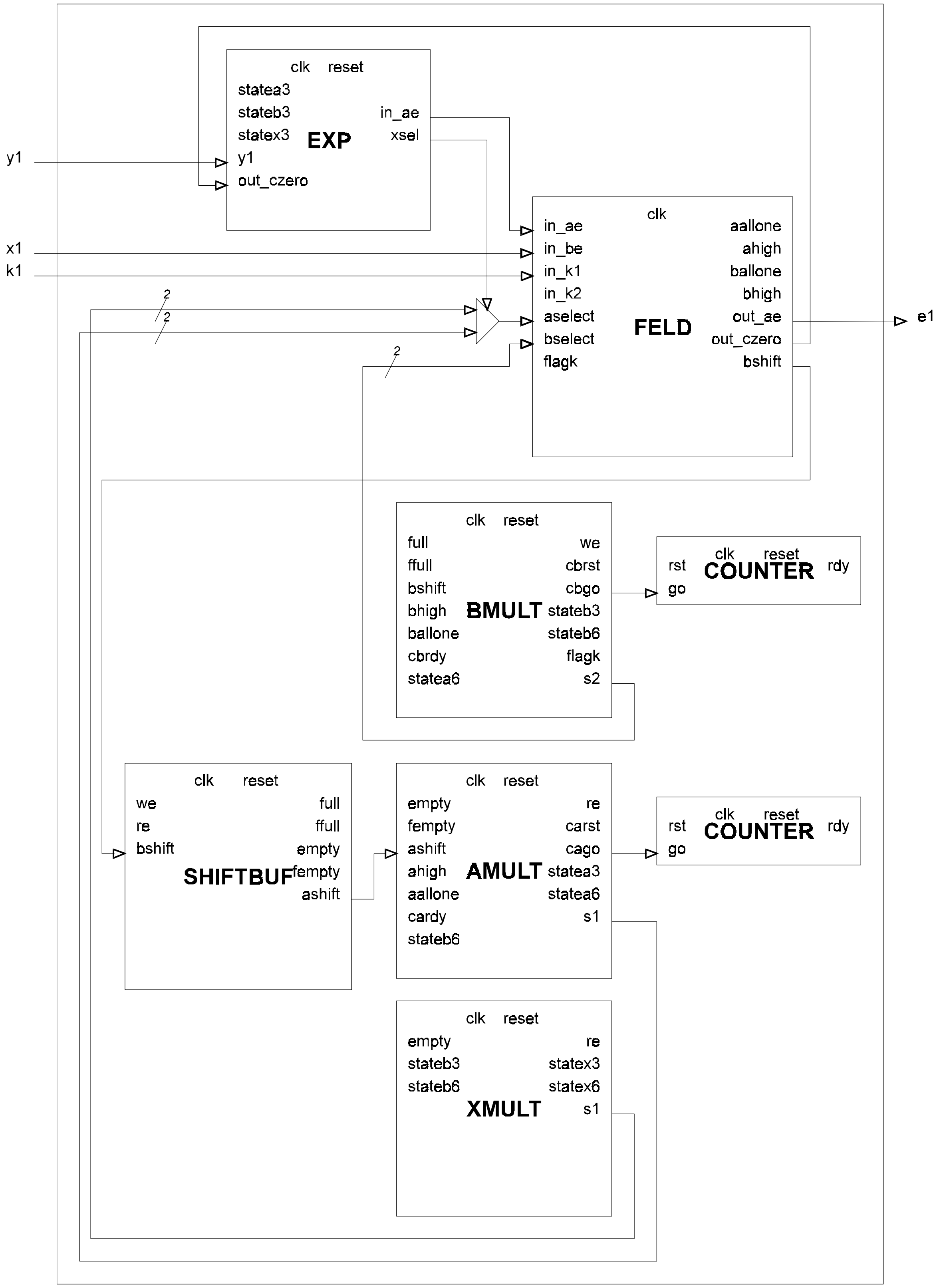

Abbildung 5.29: „Blockschaltbild (‘Carry-Save')“

In der Komponente **EXP** befindet sich der Automat zur Durchführung der Exponentiation nach der „Square-and-Multiply“-Methode. Die Komponenten **AMULT**, **BMULT** und

XMULT enthalten die Automaten zur Steuerung der Modulo-Multiplikation. Von ihnen werden die Steuersignale *aselect*, *bselect* und *flagk* für das systolische Feld (**FELD**) erzeugt.

5.3 SOSCARD[18]

Von der Universität Graz (Österreich) kommen weitere vielversprechende Ansätze zur Implementation von effizienten RSA-Chips. Die Forschungsgruppe um Reinhard und Karl Posch hat in der letzten Zeit eine Reihe von Arbeiten zu diesem Thema veröffentlicht, u.a. [LIP91], [POS90], [POS95] und [POS96]. Die in Graz gefertigten Prototypen liegen in bezug auf die Effizienz (Datenrate und Chipfläche) deutlich vor den Entwürfen nach Montgomery und der „Carry-Save"-Methode. Aus diesem Grunde wurde einer der Chips aus Graz mit Hilfe des Standardzell-Entwurfssystems des Fachbereichs Informatik der Universität Hamburg „nachgebaut". Das Ziel war es, einen der Chips aus Graz mit den anderen, in dieser Diplomarbeit verfolgten, Ansätzen vergleichen zu können.

5.3.1 Funktionsweise SOSCARD

Zum Nachbau wurde eine Implementation, die eigentlich als Co-Prozessor für die Chipkarte SOSCARD dient, ausgewählt (vergleiche [POS96]). Dieser Entwurf wurde ausgewählt, da seine Fläche aufgrund seines Einsatzgebietes auf einer Chipkarte besonders klein ist[19]. Dies ist wichtig, damit der Chip auch mit einem Standardzellentwurf noch hergestellt werden kann.

Die Realisierung der SOSCARD unterscheidet sich in einem wesentlichen Punkt von den beiden anderen Ansätzen (Montgomery und „Carry-Save"): Es kommt kein systolisches Array zum Einsatz. Die Modulo-Reduktion erfolgt nicht während der Multiplikation permanent, sondern erst nach dem Ende der Multiplikation mit Hilfe des Verfahrens des „Relaxed-Residuums".

5.3.2 Methode „Relaxed-Residuum"

Die Methode „Relaxed-Residuum" ist eine Modifikation des von Norris und Simmons [NOR81] beschriebenen Verfahrens (vergleiche „3.4.1 Reduktion durch Approximation von *k-1*" auf Seite 38) [POS90]. Sie berechnet eine Modulo-Multiplikation

$$x = a \cdot b \bmod k$$

mit Hilfe von drei Multiplikationen und einer nachfolgenden Addition. Der Modulus k wird dabei in Form zweier Konstanten, $-k$ und einer Approximation von k^{-1}, benötigt. Beide Konstanten können effizient einmalig im voraus ermittelt werden. Die Approximation von k^{-1} wird durch

[18] Secure Operating System Smartcard (SOSCARD)

[19] Die maximale Chipfläche einer Chipkarte beträgt 24 mm^2, um die Bruchfestigkeit zu gewährleisten [RAN95].

$$k^{-1} = \left\lfloor \frac{2^{2n+4}}{k} \right\rfloor$$

berechnet. Hier und im folgenden ist n die Stellenzahl der Binärdarstellung von k ($2^n > k$). Die Konstante $-k$ ist das Zweier-Komplement von k mit n+2 Stellen.

Zur Modulo-Multiplikation wird zunächst

$$x = a \cdot b$$

und

$$q = \left\lfloor \frac{x}{2^{n-1}} \right\rfloor \cdot k^{-1}$$

berechnet. Schließlich wird

$$y = \left\lfloor \frac{q}{2^{n+5}} \right\rfloor \cdot (-k)$$

und

$$z = (x + y) \bmod 2^n$$

ermittelt. Die Korrektheit des Algorithmus wird in [POS90] bewiesen.

Es soll die Modulo-Multiplikation

$$z = a \cdot b \bmod k = 9 \cdot 7 \bmod 13 = 11$$

mit Hilfe des „Relaxed-Residuums" berechnet werden. Also ist der Moduls k = 13 und seine Stellenzahl n = 4. Das Zweier-Komplement von k ist $-k$ = 51. Für die Approximation von k^{-1} ergibt sich:

$$k^{-1} = \left\lfloor \frac{2^{2n+4}}{k} \right\rfloor = \left\lfloor \frac{2^{12}}{13} \right\rfloor = 315$$

Die eigentliche Berechnung wird mit drei Multiplikationen und einer Addition wie folgt durchgeführt:

1.	$x = a \cdot b = 9 \cdot 7 = 63$
2.	$q = \left\lfloor \frac{x}{2^{n-1}} \right\rfloor \cdot k^{-1} = \left\lfloor \frac{63}{2^3} \right\rfloor \cdot 315 = 2205$
3.	$y = \left\lfloor \frac{q}{2^{n+5}} \right\rfloor \cdot (-k) = \left\lfloor \frac{2205}{2^9} \right\rfloor \cdot 51 = 204$
4.	$z = (x + y) \bmod 2^n = (63 + 204) \bmod 2^4 = 11$

Die Division durch die Zweierpotenzen wird effizient durch „Schieben-nach-rechts“ ausgeführt.

5.3.3 Aufbau der ALU (SOSCARD)

Der Kern des RSA-Chips stellt eine ALU (Arithmetic Logic Unit) dar. Sie ist in der Lage, eine 1024 x 1024 Bit Multiplikation durchzuführen, indem diese in mehrere 16 x 16 Bit Multiplikationen aufgeteilt wird. Komponenten der ALU sind ein 16 x 16 Bit Multiplizierer, ein 16-Bit Addierer, zwei 16-Bit Multiplexer und zwei 1024-Bit Register (siehe Abbildung 5.30: „Blockschaltbild ALU“). Weiterhin werden in der Komponente **MCOUNT** diverse Signale generiert, die zur korrekten Bereitstellung der Ein- und Ausgaben dienen. Am Eingang c kann ein weiterer Wert angelegt werden. Dieser wird zum Ergebnis der Multiplikation hinzuaddiert. Die ALU ist somit in der Lage

$$z = a \cdot b + c$$

zu berechnen. Wobei a, b und c 1024-Bit große Zahlen sind. Die Größe des Ergebnisses z beträgt 2048 Bit[20].

[20] Die ALU führt keine Modulo-Reduktion durch. Die Reduktion wird erst nach der eigentlichen Multiplikation durchgeführt.

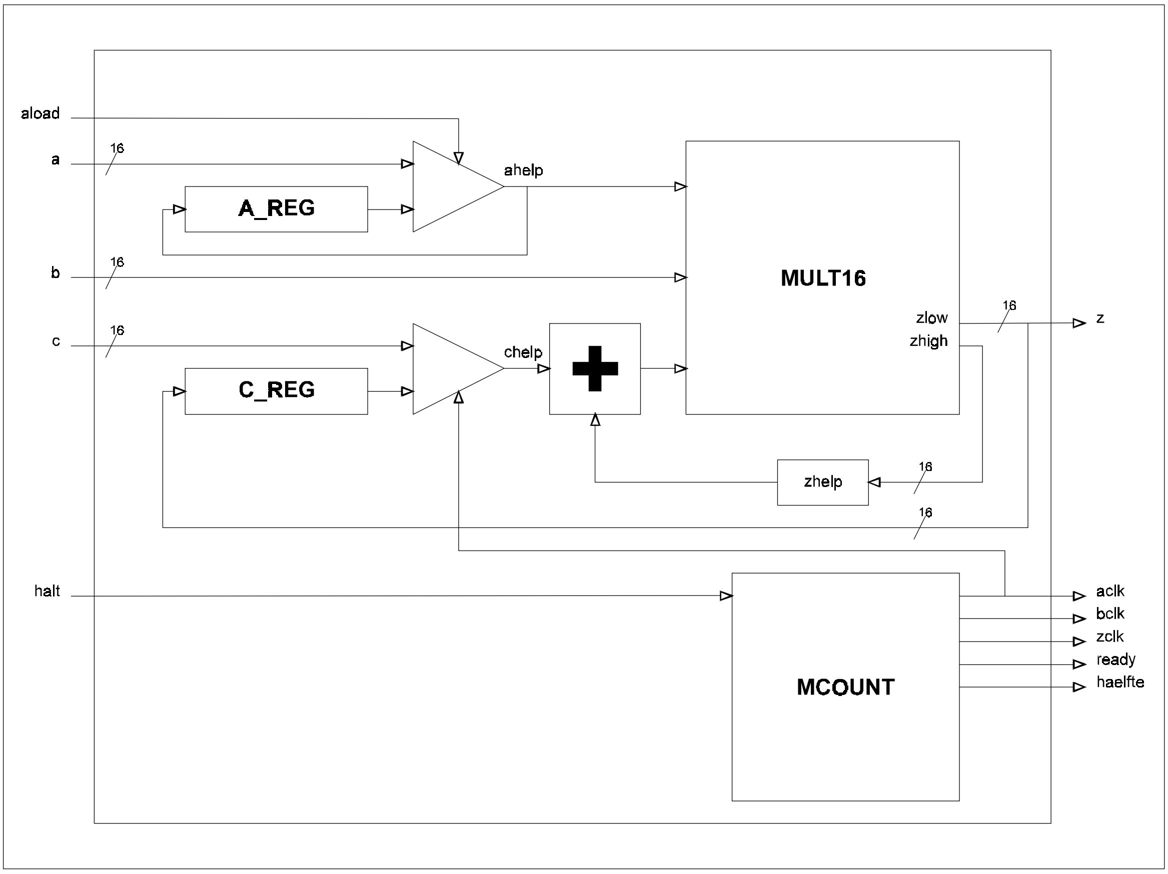

Abbildung 5.30: „Blockschaltbild ALU“

Die 1024-Bit breiten Eingangswerte *a*, *b* und *c* müssen in 64 Abschnitten zu je 16 Bit angelegt werden. Das Ergebnis *z* wird ebenso in 128 Blöcken zu 16 Bit ausgegeben. Die beiden Register dienen zur Zwischenspeicherung des Ergebnisses *z* und des Eingangswertes *a*, der mehrfach benötigt wird.

5.3.4 Erweiterung zum Modulo-Multiplizierer (SOSCARD)

Um den 1024-Bit Multiplizierer zum Modulo-Multiplizierer zu erweitern, ist ein Steuerungsautomat (State machine) notwendig (siehe Abbildung 5.31: „Statemachine ‘MultMod’ (Soscard)“) zur korrekten Ansteuerung nötig. Auf diese Weise werden die drei Multiplikationen der „Relaxed-Residuum“ Methode mit Hilfe der ALU nacheinander berechnet.

In „State 1“ und „State 2“ findet keine zusätzliche Addition statt. Am Eingang *c* der ALU wird deshalb permanent eine Null angelegt. Zur Speicherung der Zwischenergebnisse *x* und *q* werden zwei weitere 1024-Bit Register benötigt.

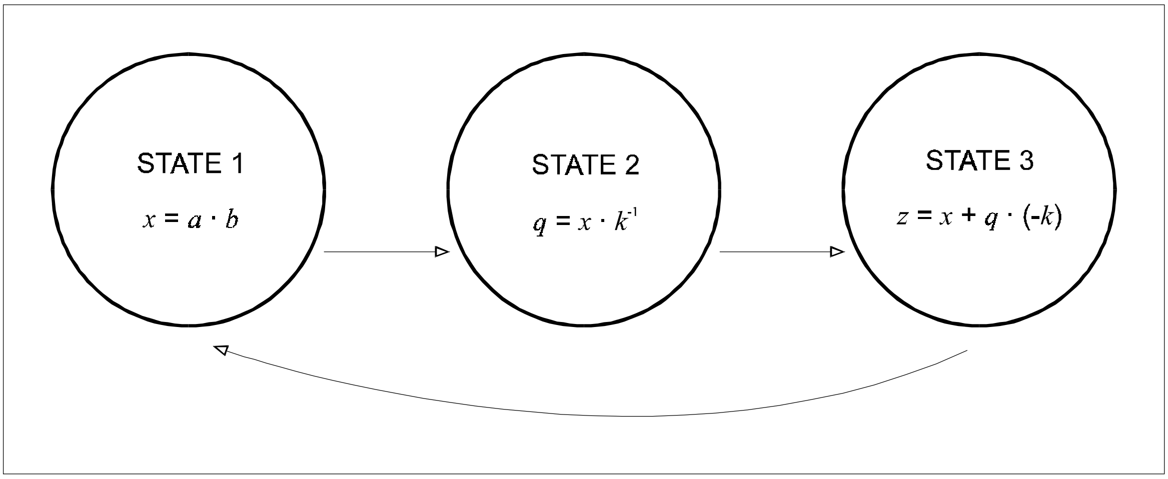

Abbildung 5.31: „Statemachine 'MultMod' (SOSCARD)“

Da eine „16x16“ Multiplikation in einem Taktzyklus ausgeführt wird, werden für jede „1024x1024“ Multiplikation $n_{\text{mult16}} = 64 \cdot 64 = 4096$ „16x16“ Multiplikationen benötigt. Jede Modulo-Multiplikation erfordert $n_{\text{mult1024}} = 3$ „1024x1024“ Multiplikationen. Damit benötigt die Methode „SOSCARD“

$$Anz_{\text{Takte}} = n_{\text{mult1024}} \cdot n_{\text{mult16}} = 3 \cdot 4096 = 12288$$

Takte für eine Modulo-Multiplikation.

5.3.5 Aufbau der Modulo-Exponentiationseinheit (SOSCARD)

Im letzten Schritt wird der Modulo-Multiplizierer zur Exponentiationseinheit erweitert. Hierzu ist ein weiterer Zustandsautomat nötig (siehe Abbildung 5.32: „Square-and-Multiply (Soscard)“). Diese arbeitet nach dem „Square-and-Multiply“-Algorithmus. Es werden zwei weitere 1024-Bit Register zur Speicherung der Zwischenergebnisse benötigt.

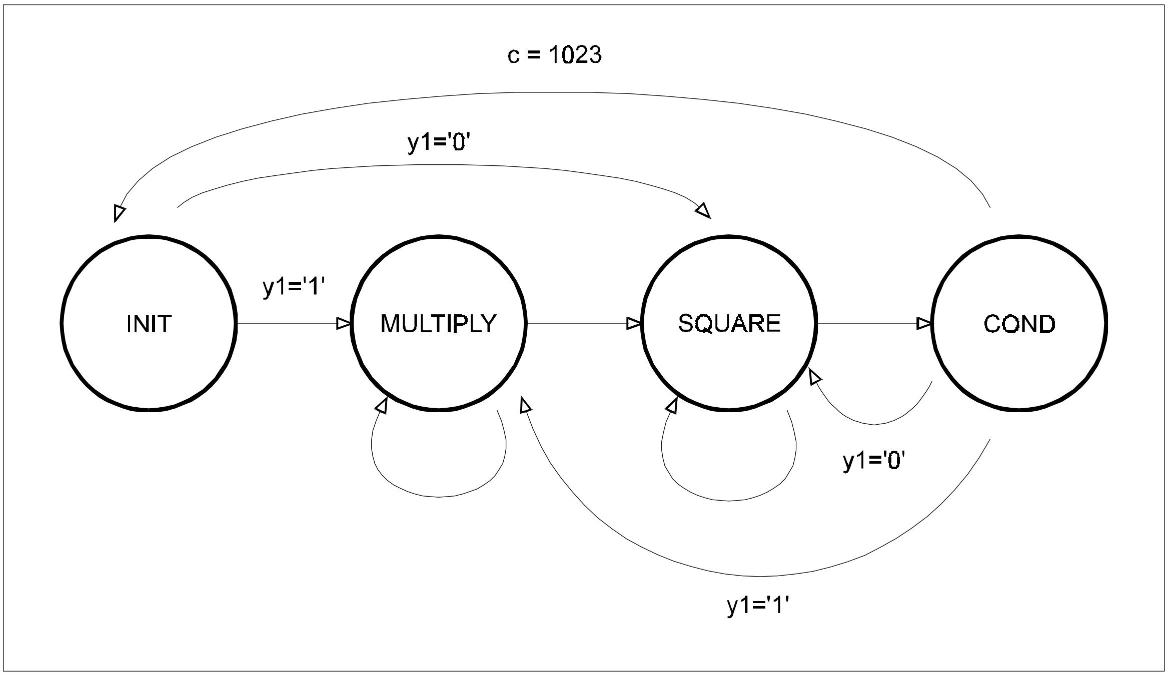

Abbildung 5.32: „Square-and-Multiply (SOSCARD)“

Die Abbildung 5.33: „Blockschaltbild SOSCARD“ zeigt das Blockschaltbild des Modulo-Multiplizierers der SOSCARD. Es zeigt den Modulo-Multiplizierer und zwei 1024-Bit Register sowie symbolisch den zentralen Steuerungsautomaten (State machine).

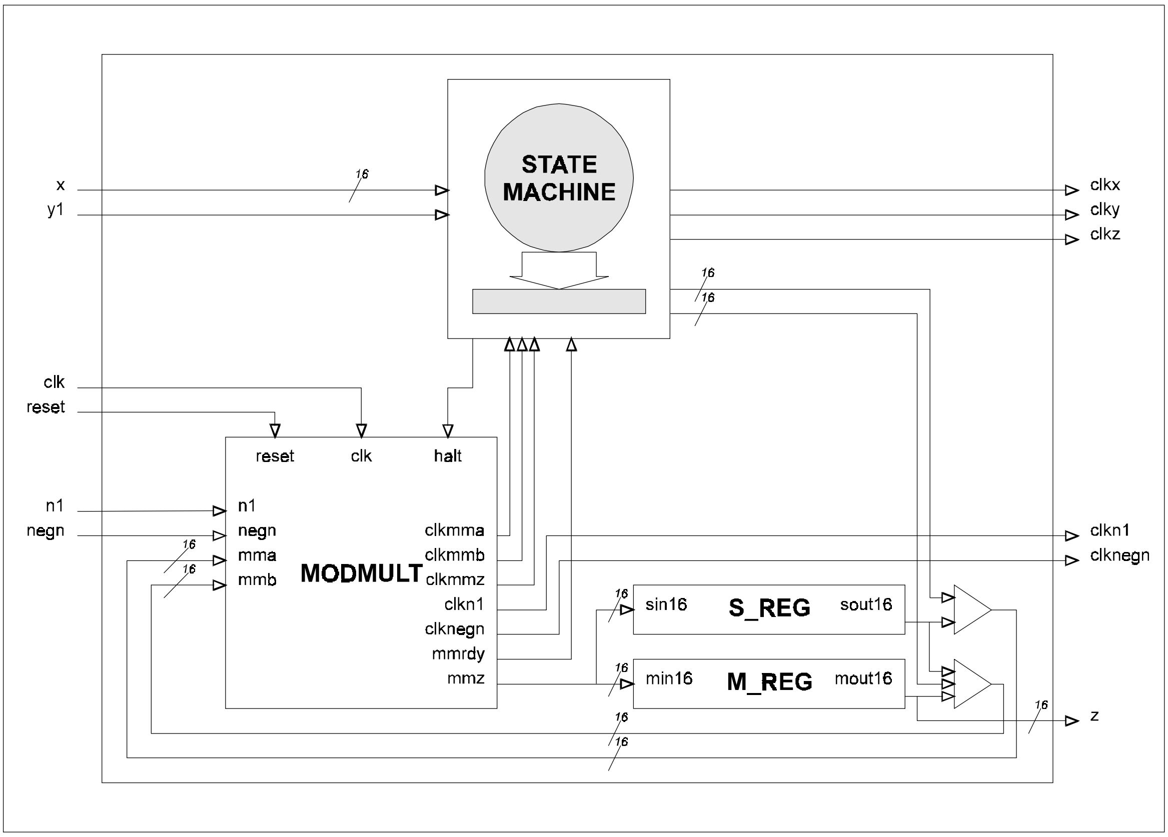

Abbildung 5.33: „Blockschaltbild SOSCARD“

6 Bewertung und Ausblick

6.1 Probleme

Im Rahmen der Diplomarbeit gab es in einem Punkt essentielle Schwierigkeiten: Die Tools zum Chipentwurf. Sie sorgten für erhebliche Wartezeiten während der Arbeit. Sie weisen drei wesentliche Schwächen auf:

1. Die Ausführungsgeschwindigkeit ist selbst auf den leistungsstärksten Workstations des Arbeitsbereich TECH des Fachbereich Informatik viel zu langsam. Die sehr umfangreichen Entwürfe stellten ausgesprochen hohe Anforderungen an die Rechenleistung und den Speicherplatz. Nur zum Teil konnte Rechenzeit durch einen stärker hierarchischen Aufbau der Entwürfe eingespart werden.
2. Die Tools arbeiten ausgesprochen „unselbständig". Es ist keineswegs so, daß nach dem Entwurf in VHDL, das entsprechende Werkzeug zum Chipentwurf gestartet wird und nach einiger Zeit der Chip „fertig" ist. Vielmehr sind ständig Eingaben des Benutzers nötig. Diese sind in Abständen von etwa zwei Stunden notwendig. Es war also nicht möglich, den Chipentwurf zu starten und sich nach ein oder zwei Wochen Rechenzeit das Ergebnis anzuschauen.
3. Die Simulation des Entwurfs war nur mit „kleinen" Schlüssellängen (64 Bit) möglich, da das Programm zur Simulation besonders hohe Ansprüche an die Größe des Arbeitsspeichers stellt.

Nur die Abbildung der Verhaltensbeschreibung auf die Standardzellen (Synthese) konnte mit Entwürfen mit den angestrebten 1024-Bit Zahlen erfolgen. Ansonsten wurde die Größe der verwendeten Zahlen von 1024 Bit auf 128 Bit (Verdrahtung) bzw. 64 Bit (Simulation) reduziert, um die Laufzeiten der Programme zu reduzieren bzw. deren Einsatz überhaupt erst zu ermöglichen.

6.2 Ergebnisse

Alle drei zuvor beschriebenen Realisierungen des RSA-Verfahrens wurden mit der Hardwarebeschreibungssprache VHDL beschrieben. Die VHDL-Beschreibung wurde dann mit dem „SYNOPSYS Design Analyzer" synthetisiert und auf die ES2-Zellbibliotheken abgebildet. Die „SYNOPSYS" Datenbasis wurde schließlich in eine EDIF-Datei für den Datentransfer zu „CADENCE DFW II" umgewandelt. Letztlich wurde das physikalische Layout (Placement und Routing) mit „CADENCE DFW II" durchgeführt. Es kam ein 0,7 µm CMOS-Prozeß zum Einsatz.

6.2.1 Geschwindigkeit

Um die Verschlüsselungsrate eines Chips (in KBit/s) und damit seine Geschwindigkeit berechnen zu können, benötigt man zwei Größen:

1. Die für eine Exponentiation benötigten Taktzyklen n.
2. Die Taktfrequenz[21] f (in MHz) mit der der Chip betrieben werden soll bzw. kann.

Die Datenrate *DR* ergibt sich dann nach der folgenden Formel:

$$DR = \frac{1024 Bit \cdot f \cdot 10^6 \, \frac{1}{s}}{n}$$

Wie bereits in Abschnitt „4.1 Allgemeine Abschätzungen“ auf Seite 45 gezeigt wurde, werden für eine Exponentiation nach dem seriellen „Square-and-Multiply“ Verfahren im Mittel 1536 Modulo-Multiplikationen benötigt. Die parallele Version braucht 1024 Modulo-Multiplikationen. Alle drei realisierten Entwürfe benutzen das „Square-and-Multiply“-Verfahren zur Exponentiation. Um die insgesamt nötigen Taktzyklen zu berechnen, muß man also zunächst die für eine Modulo-Multiplikation benötigen Zyklen ermitteln. Diese wurden bereits im Kapitel 5 „Realisierung und Implementation“ im Rahmen der Funktionsbeschreibung der einzelnen Implementationen berechnet. Die Durchlaufverzögerung für den längsten Pfad wurde mit Hilfe des Programmpaketes „SYNOPSYS“ bestimmt. In Abbildung 6.1: „Zeiten Modulo-Multiplikation (1024 Bit)“ findet sich eine Zusammenfassung der Werte.

1024 Bit	*Takte*	*Durchlauf-verzögerung*	*max. Takt-frequenz*
Montgomery	2051	10,5 ns	95 MHz
Carry-Save	1732	7,0 ns	143 MHz
SOSCARD	12288	50,6 ns	20 MHz

Abbildung 6.1: „Zeiten Modulo-Multiplikation (1024 Bit)“

Die Anzahl der Taktzyklen pro Exponentiation (mit „Square-and-Multiply“) erhält man durch Multiplikation der für eine Modulo-Multiplikation durchschnittlich benötigten Takte mit 1536 („seriell“) bzw. 1024 („parallel“). Um die verschiedenen Entwürfe miteinander vergleichen zu können, wurde eine gemeinsame Taktfrequenz von 100 MHz festgelegt. Der Entwurf „Carry-Save“ erreicht diese Taktfrequenz ohne Probleme. Die Implementation der Montgomery-Methode verfehlt die 100 MHz nur knapp. Die hohe Durchlaufverzögerung des Chips „SOSCARD“ ist in dem verwendeten „16x16“ Multiplizierer begründet. Es wurde die „Standardlösung“ des „SYNOPSYS Design Analyzer“ verwendet. Diese zeichnet sich durch eine geringe Performanz aus. Wird der Multiplizierer im Full-Custom-Entwurf speziell erstellt, sollte eine Taktfrequenz von 100 MHz durchaus erreichbar sein. In der Abbildung 6.2: „1024 Bit, 100 MHz“ und Abbildung 6.3: „Durchsatz (1024 Bit)“ sind die Ergebnisse zusammengefaßt.

[21] Die maximale Taktfrequenz f kann man aus der Signallaufzeit l des „längsten Pfades“ eines Chips berechnen. Wenn $l = 10$ ns, so ist $f = 10^3 / l = 100$ MHz $[10^6 / s]$.

1024 Bit	*seriell*	*parallel*
Montgomery	31,7 KBit/s	47,6 KBit/s
Carry-Save	37,6 KBit/s	56,4 KBit/s
SOSCARD	5,3 KBit/s	7,9 KBit/s

Abbildung 6.2: „1024 Bit, 100 MHz“

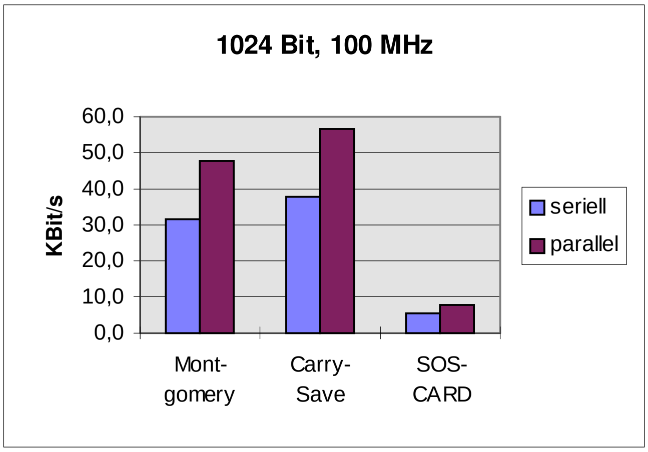

Abbildung 6.3: „Durchsatz (1024 Bit)“

6.2.2 Fläche

Die in VHDL abgefaßte Verhaltensbeschreibung der unterschiedlichen Implementationen wurde zunächst mit dem Programmpaket „SYNOPSYS“ analysiert und synthetisiert. Nach der Synthese berechnet das Programm aus der Anzahl der für den Entwurf benötigten Standardzellen eine Abschätzung der Chipgröße. Nicht enthalten ist hierbei die Fläche, die für die Verdrahtung benötigt wird, sondern nur die reine Fläche für die Standardzellen. Die Abbildung 6.4: „Chipfläche (Synopsys)“ zeigt die Zusammenfassung der Abschätzung.

1024 Bit	*Chipfläche*
Montgomery seriell	92,4 mm^2
Carry-Save parallel	44,8 mm^2
SOSCARD seriell	19,5 mm^2

Abbildung 6.4: „Chipfläche (SYNOPSYS)“

Um eine Vorstellung vom zusätzlichen Platzbedarf für die Verdrahtung zu erhalten, wurde zunächst ein Entwurf (128 Bit) nach der Montgomery-Methode dem kompletten Entwurfsvorgang unterzogen. Die insgesamt benötigte Fläche vergrößerte sich im Vergleich zu dem von „SYNOPSYS“ erhaltenen Wert für die Fläche der Standardzellen etwa um den Faktor zwei. Dieser Wert deckt sich mit den Erfahrungen, die bisher am Arbeitsbereich TECH des Fachbereichs Informatik gemacht wurden. Der komplette Entwurfsvorgang wurde für die drei Entwürfe mit eine Blockgröße von 1024 Bit nicht vorgenommen, weil die Abschätzung der Fläche als ausreichend genau angesehen wurde.

1024 Bit	*Fläche Zelle*	*benötigte Zellen*	*Abschätzung Full-Custom*
Montgomery parallel	79.713 µm²	3072	245 mm²
Montgomery seriell	79.713 µm²	1536	122 mm²
Carry-Save parallel	72.588 µm²	1024	74 mm²

Abbildung 6.5: „Fläche (Full-Custom; 1024 Bit)“

Weiterhin wurde versucht abzuschätzen, wie groß die Chipfläche maximal wird, wenn statt des Entwurfs mit Standardzellen ein Full-Custom-Entwurf verwendet würde. Für beide Methoden (Montgomery und „Carry-Save“) wurde eine einzelne Zelle des systolischen Arrays dem Entwurfsvorgang unterzogen. Dabei wurde die Zelle jeweils aus Standardzellen aufgebaut. Diese wurden mit Hilfe von „CADENCE DFW II“ plaziert und verdrahtet (siehe Abbildung 6.6: „Layout Zelle (Montgomery)“). Unter der Annahme, daß man mit Hilfe eines Full-Custom-Entwurfs in der Lage ist, die einzelnen Zellen so zu konstruieren, daß keine weitere Verdrahtung nötig ist, wurde die ermittelte Fläche einer Zelle mit der benötigten Anzahl von Zellen multipliziert. Die Abschätzung ist in der Abbildung 6.5: „Fläche (Full-Custom; 1024 Bit)“ zusammengestellt. Bei dieser Abschätzung handelt es sich um die obere Grenze. Ein erfahrener Entwickler würde die einzelnen Zellen natürlich noch optimieren. Eine weitere Reduzierung der Chipfläche wäre sicher, weil

1. jede Standardzelle Treiber enthält, die bei einem Full-Custom-Entwurf zum Teil entfallen können,
2. jede Standardzelle die gleiche Höhe haben muß, um ihre Plazierung zu ermöglichen. Bei einem Full-Custom-Entwurf ist dies nicht der Fall.

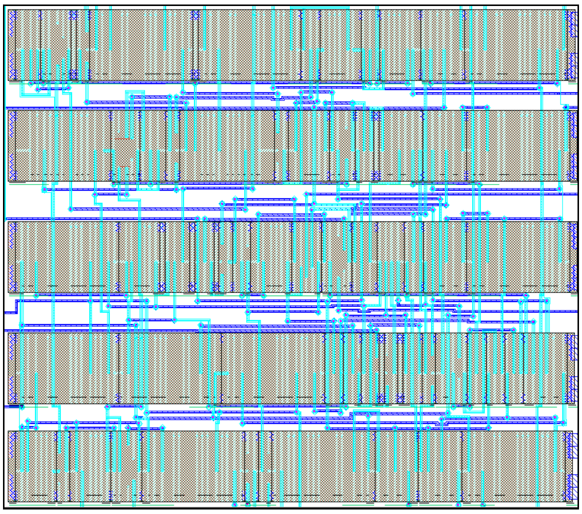

Abbildung 6.6: „Layout Zelle (Montgomery)“

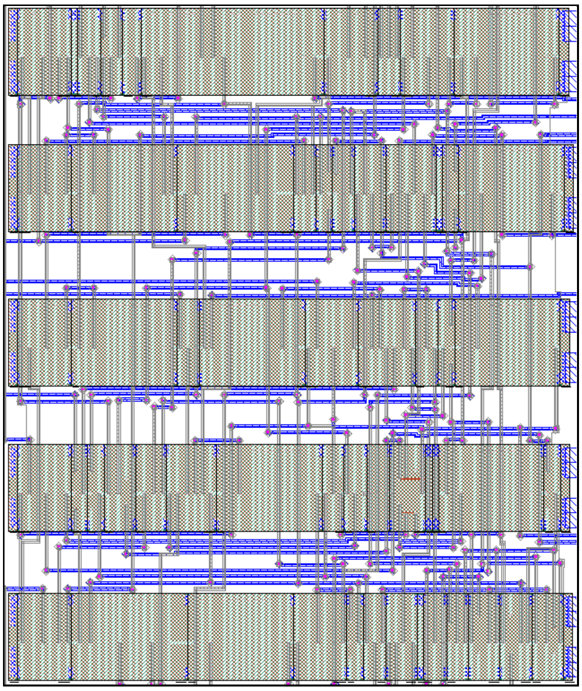

Abbildung 6.7: „Layout Zelle (‘Carry-Save’)“

Ein wesentlicher Grund für die enorme Größe der Standardzell-Entwürfe liegt in der Art der verwendeten Register. Sie wurden aus einzelnen D-Flip-Flops aufgebaut. Ein einziges 1024-Bit Register benötigt so eine Chipfläche – ohne Verdrahtung – von 2,7 mm^2. Der Entwurf „SOSCARD“ enthält zum Beispiel insgesamt sechs Stück, so daß allein für die Register eine Chipfläche von 16,2 mm^2 verbraucht würde. Werden statt der D-Flip-Flops RAMs verwendet, so benötigt man für ein 1024-Bit Register lediglich 0,45 mm^2. Für sechs solcher Register sind nur noch 2,7 mm^2 Chipfläche nötig. Dies sind nur 17 Prozent der ursprünglichen Fläche. Nimmt man statt des „normalen“ RAMs ein sogenanntes „Dual Ported RAM“, um ein gleichzeitiges Lesen und Schreiben zu ermöglichen, vergrößert sich die Chipfläche für ein 1024-Bit Register auf 0,87 mm^2. In der Abbildung 6.8: „Chipgröße mit RAM-Registern“ sind die Ergebnisse zusammengefaßt. Es wurde die Flächenabschätzung für die Standardzellen von „SYNOPSYS“ zugrunde gelegt. Es fehlt die Fläche für die Verdrahtung.

	Anzahl Register	*Fläche D-FF*	*Fläche RAM*	*Gesamt-Chipfläche mit RAM*
Montgomery seriell	8	21,6 mm^2	3,6 mm^2	74,4 mm^2
Carry-Save parallel	10	27,0 mm^2	4,5 mm^2	22,3 mm^2
SOSCARD seriell	6	16,2 mm^2	2,7 mm^2	6,0 mm^2

Abbildung 6.8: „Chipgröße mit RAM-Registern“

Die Entwürfe wurden dennoch mit Registern aus D-Flip-Flops erstellt,
1. um eine einfache Simulation zu ermöglichen,
2. da der Speicherzugriff auf Flip-Flops leichter ist als auf RAM. Beim Zugriff auf Flip-Flops ist keine Adressierung notwendig.

Die folgenden Abbildungen zeigen die Fläche der Standardzellen der einzelnen Entwürfe und deren Verteilung auf die einzelnen Komponenten. Die Werte wurden mit „SYNOPSYS“ ermittelt. Eine etwaige Fläche für Verdrahtung wurde nicht berücksichtigt.

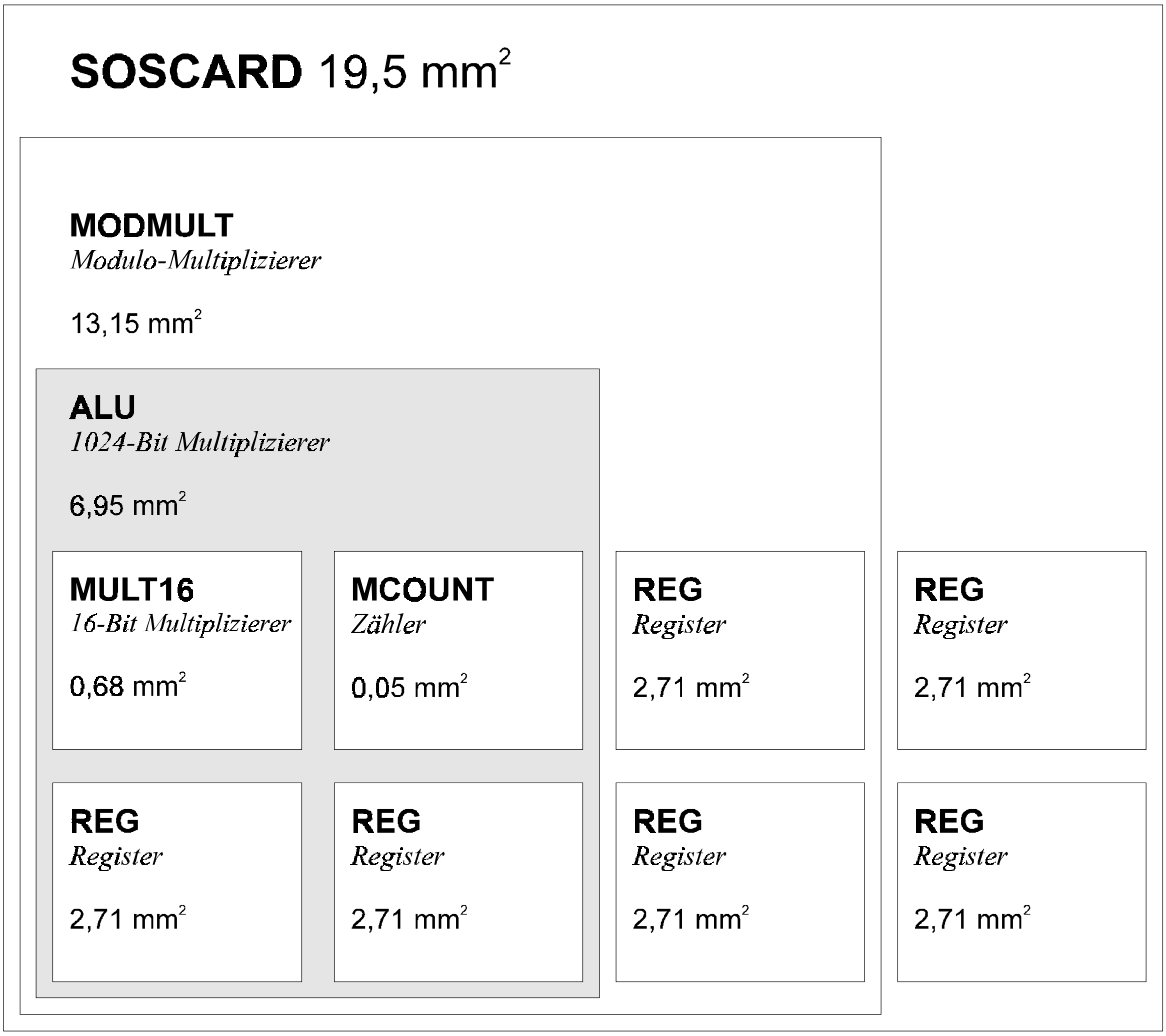

Abbildung 6.9: „Fläche ‘SOSCARD seriell’“

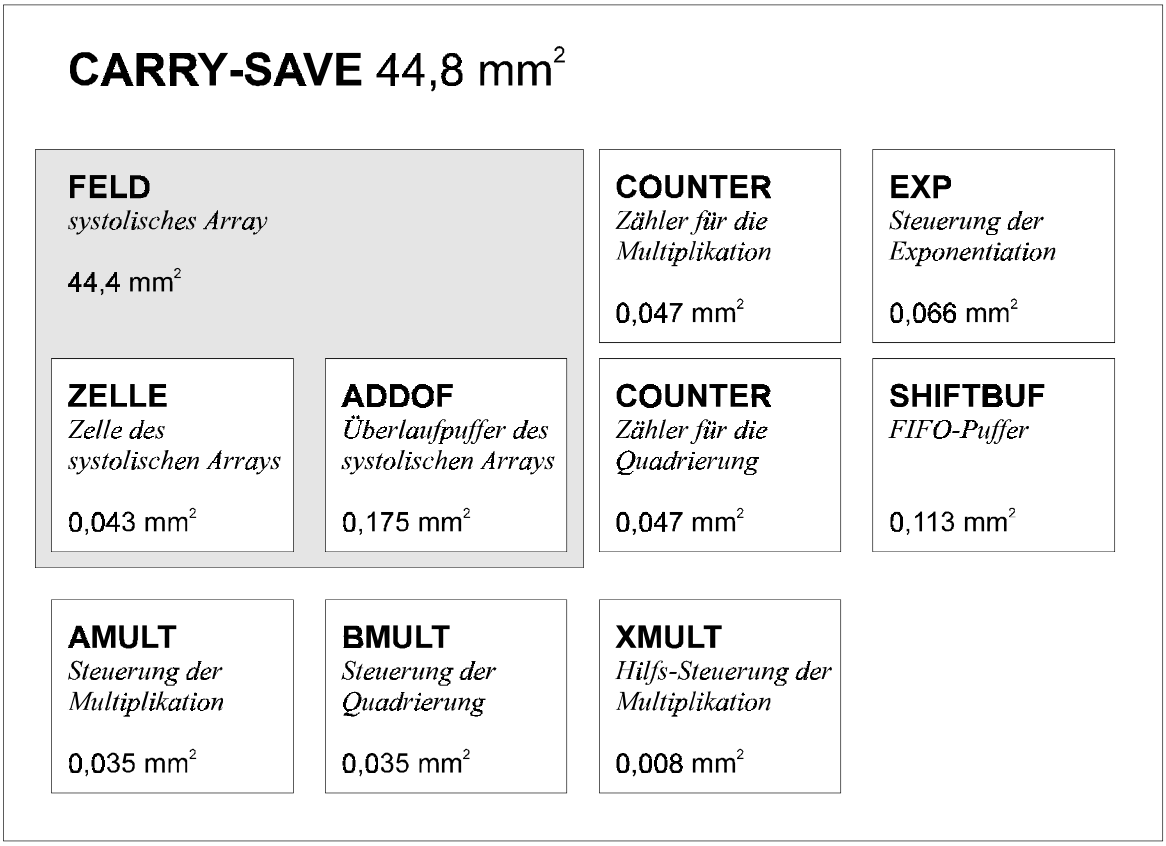

Abbildung 6.10: „Fläche ‘Carry-Save parallel’“

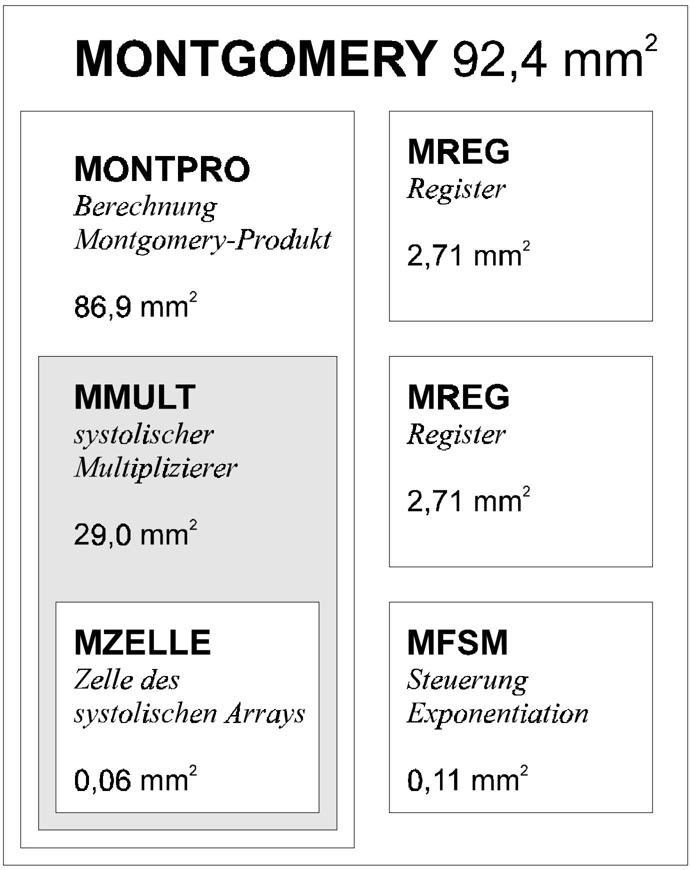

Abbildung 6.11 „Fläche ‘Montgomery seriell’“

Eine Realisierung mit 128 Bit nach Montgomery wurde mit Hilfe des Standardzellentwurfs komplett gefertigt. Die Abbildung 6.12: „Layout (Montgomery; 128 Bit)“ zeigt das Layout des Chips. Man kann erkennen, daß viel Fläche für die Verdrahtung verschenkt wurde. Ebenfalls sieht man, daß der Standardzellentwurf die regulären Strukturen des systolischen Arrays überhaupt nicht unterstützt. Standardzellen werden immer zeilenweise plaziert, jede Zelle hat deshalb die gleiche Höhe. Nur ein Full-Custom-Entwurf kann die Vorteile systolischer Architekturen nutzen.

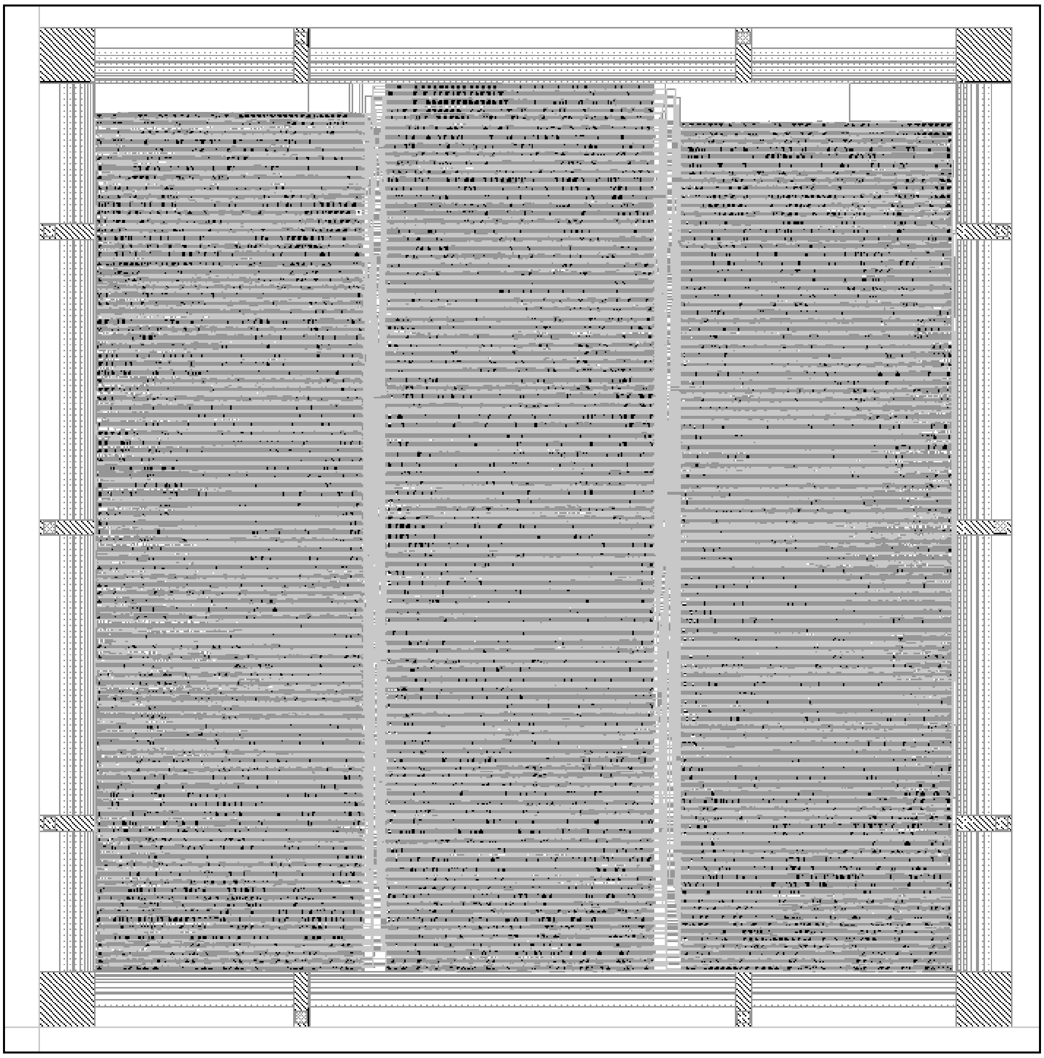

Abbildung 6.12: „Layout (Montgomery; 128 Bit)“

6.2.3 Vergleich Original und Nachbau (SOSCARD)

Um den Nachbau (1024 Bit) mit dem Original (640 Bit) vergleichen zu können, wurden auch für den Nachbau eine Schlüssellänge von 640 Bit zugrunde gelegt. In der Abbildung 6.13: „Vergleich SOSCARD“ sind die Daten der Chips gegenüber gestellt. Die Werte für die original SOSCARD sind [POS96] entnommen. Es zeigen sich deutlich die Vorteile des Full-Custom-Entwurfs des Originals: An Chipfläche wird im Vergleich zum Nachbau nur etwa 62 Prozent benötigt und die Zeit (Durchlaufverzögerung) läßt sich um ein Drittel reduzieren.

640 Bit	*Original*	*Nachbau*
Fläche	< 24 mm^2	~ 39 mm^2
Durchlaufverzögerung	< 16,7 ns	50,6 ns
Taktfrequenz	60 MHz	20 MHz
Takte pro Multiplikation	5841	4800
Datenrate	4,2 KBit/s	1,7 KBit/s

Abbildung 6.13: „Vergleich SOSCARD“

6.2.4 Zusammenfassung und Bewertung

In der Abbildung 6.14: „Zusammenfassung (1024 Bit)“ sind die Ergebnisse aller drei Entwürfe den Vorgaben für diese Diplomarbeit gegenübergestellt. Bei der Chipfläche der Entwürfe Montgomery und „Carry-Save“ ist die jeweils obere Abschätzung für den Full-Custom-Entwurf zugrunde gelegt. Für den Entwurf „SOSCARD“ sind die Ergebnisse von „SYNOPSYS“ für die Fläche der Standardzellen plus einem Schätzwert für die Verdrahtung angegeben. Eine sinnvolle Abschätzung für den Full-Custom-Entwurf konnte im Fall der „SOSCARD“ nicht gemacht werden, weil diese Implementation nicht dem Prinzip des systolischen Arrays folgt. Die Taktfrequenz wurde rechnerisch auf die vorgegebenen 12 MHz reduziert. Entsprechend verringert sich die erwartete Datenrate erheblich. Bei der für RSA-Chips weit verbreiteten Taktfrequenz von 25 MHz erfüllt die Implementierung „Carry-Save (parallel)“ die Vorgaben in bezug auf Chipfläche und Datenrate.

	Datenrate (12 MHz)	*Datenrate (25 MHz)*	*Chipfläche*
Vorgaben	≥ 9,6 KBit/s	–	≤ 90 mm^2
Montgomery seriell	3,8 KBit/s	7,9 KBit/s	122 mm^2
Carry-Save parallel	6,8 KBit/s	14,1 KBit/s	74 mm^2
SOSCARD seriell	0,6 KBit/s	1,3 KBit/s	39 mm^2

Abbildung 6.14: „Zusammenfassung (1024 Bit)“

Zusammenfassend läßt sich sagen, daß die Vorgaben mit den gegebenen Randbedingungen nicht erreicht wurden. Das Erfüllen der Vorgaben erscheint möglich wenn,

1. der Chip komplett als Full-Custom-Entwurf erstellt wird,
2. ein kleinerer als der 0,7 µm CMOS-Prozeß und
3. eine höhere Taktfrequenz als 12 MHz verwendet wird.

Auch die Montgomery-Methode wird, wenn mehr Chipfläche zur Verfügung steht, die Randbedingungen erreichen können. Ein „doppelt-paralleler“ Entwurf – mit vier Modulo-Multiplizierern pro Chip – erreicht eine Datenrate von 11,8 KBit/s bei einer Taktfrequenz von 12 MHz. Er benötigt zur Zeit aber selbst nach der Abschätzung für den Full-Custom-Entwurf (0,7 µm CMOS Prozeß) eine beachtliche Chipfläche von 490 mm^2.

Als Endergebnis gibt es drei funktionstüchtige Entwürfe eines Hardwareakzelerators für das RSA-Verfahren. Alle Entwürfe liegen als Verhaltensbeschreibung in VHDL vor. Die Beschreibungen lassen sich einfach an unterschiedliche Schlüssellänge anpassen.

6.3 Vergleich mit anderen Arbeiten

In den letzten Jahren wurden in unterschiedlichen Veröffentlichungen Prototypen von RSA-Chips vorgestellt. Im folgenden werden die Ergebnisse dieser Diplomarbeit mit denen anderen Arbeiten verglichen.

In [VAN89] wird ein 1024 Bit RSA-Chip vorgestellt, der bei einer Taktrate von 25 MHz eine Verschlüsselungsrate von acht KBit/s erreicht. Der zugrunde liegende Algorithmus arbeitet mit einem Multiplizierer nach dem „Carry-Save"-Prinzip und einer „übertragsfreien" Division. Diese wird durch spezielle Zahlendarstellung („Redundant Signed Digit" - RSD) ermöglicht. Hierbei wird eine Zahl X als Differenz zweier positiver Zahlen X' und X'' dargestellt:

$$X = \sum_{i=0}^{n} x_i \cdot 2^i = \sum_{i=0}^{n} (x_i' - x_i'') \cdot 2^i$$

Für jede Ziffer von X gilt: $x_i \in \{-1, 0, 1\}$. Die relativ geringe Größe des Chips (80 mm^2 bei einem 2 µm Prozeß) ist zum einen durch einen Full-Custom-Entwurf erreicht worden, zum anderen wird pro Bit nur ein Volladdierer verwendet.

In [ORU90] wird eine andere Implementation vorgestellt (512 Bit, 100 mm^2, 2 µm Prozeß, 20 MHz, 78 KBit/s). Diese benutzt ebenfalls einen Multiplizierer nach dem „Carry-Save"-Prinzip. Das Besondere ist, daß immer zwei Multiplikationen parallel ausgeführt werden. Weiterhin wird bei der Multiplikation der Multiplikator nicht bitweise abgearbeitet, sondern es werden immer n Bits auf einmal betrachtet. Als sinnvolle Werte haben sich $n = 4$ und $n = 5$ herausgestellt.

Beim Verfahren von Sedlak [SED87] wird versucht, die Reduktion an das Verfahren zur Multiplikation anzupassen. Die Reduktion wird nicht nach der Multiplikation durchgeführt, sondern während der Multiplikation. Es kommt ein modifiziertes serielles „Shift-and-Add"-Verfahren zum Einsatz, bei dem der Darstellungsbereich um einen 20-stelligen Buffer erweitert wird. Es wird ein spezieller „Look-Ahead" Algorithmus verwendet. Hierbei wird nicht das ganze Zwischenergebnis mit dem Modulus verglichen, um zu entscheiden, ob eine Subtraktion zur Reduktion stattfinden muß oder nicht. Vielmehr werden nur die höchstwertigen Bits verglichen. Im Gegensatz zu vielen anderen Verfahren kommt kein „Carry-Save"-Addierer zum Einsatz, sondern ein Addierer nach dem „Carry-Lock-Ahead"-Prinzip. Er ist so konstruiert, daß je nach Länge der Berechnung für den Übertrag eine unterschiedliche Anzahl von Taktzyklen benötigt wird.

Der von Sedlak gefertigte Prototyp soll eine Verschlüsselungsrate von 195 KBit/s bei einer Schlüssellänge von 780 Bit erreichen. Der Prototyp ist 24 mm^2 groß (1,5µ CMOS Prozeß). Diese Werte erscheinen auf den ersten Blick sehr beachtlich. Berücksichtigen muß man, daß in der Regel eine Verdoppelung der Schlüssellänge zu einer Geschwindigkeit von einem Viertel des ursprünglichen Wertes führt. Dieses läßt bei einer Schlüssellänge von 1024 Bit eine Rate von 113 KBit/s erwarten. Außerdem wurde bei dieser Implementation der chinesische Restsatz verwendet (vergleiche „3.1.4 Chinesischer Restsatz"). Kommt dieser nicht zum Einsatz, verringert sich die Verschlüsselungsrate auf ein Viertel. Dies würde für diesen Fall eine Rate von 28 KBit/s bedeuten.

Von Hoornaert u.a. [HOO88] wird ein RSA-Verfahren vorgestellt, das mit dem „Square-and-Multiply“-Algorithmus arbeitet. Die Multiplikationen werden nach dem „Shift-and-Add“-Prinzip mit „Carry-Save“-Addierern durchgeführt. Die Reduktion erfolgt nach jeder Addition. Eine höhere Geschwindigkeit wird durch einige Modifikationen erreicht. So wird die Anzahl der möglichen Quotienten auf die Werte 0, 1, 2 und 3 limitiert, so daß die Multiplikation durch eine Tabelle von vorausberechneten Werten ersetzt werden kann. Es kommt ein zweistufiger Addierer zum Einsatz. Die erste Stufe übernimmt die bedingte Addition des Multiplikators. Die zweite Stufe führt die Modulo-Reduktion aus.

Um eine Verschlüsselungsrate von 13 KBit/s bei einer Schlüssellänge von 672 Bits zu erreichen werden sechs Chips (!) benötigt. Die Chips werden mit einer Taktfrequenz von 14 MHz betrieben. Sie wurden mit einem 1,5µ CMOS Prozeß gefertigt.

Hersteller	*Quelle*	*Jahr*	*Technologie*	*Takt*	*Rate*	*Größe*
Cylink	[BRI89]	1987	1,5 µm	16 MHz	6,8 KBit/s	-
CNET	[BRI89]	1988	1 µm	25 MHz	5,3 KBit/s	-
UCL	[VAN89]	1988	2 µm	25 MHz	16 KBit/s	80 mm^2
Vandemeulebroecke	[VAN89]	1989	2 µm	25 MHz	8,0 KBit/s	80 mm^2
Alpha Tech.	[SAU93]	1990	2 µm	25 MHz	13 KBit/s	-
Pijenburg	[SAU93]	1992	1 µm	25 MHz	43 KBit/s	

Abbildung 6.15: „Übersicht über RSA-Chips (1024 Bit)“

In der Abbildung 6.15: „Übersicht über RSA-Chips (1024 Bit)“ sind Implementationen der letzten Jahre zusammengestellt. Es wurden nur Chips aufgeführt, die mit einer Schlüsselbreite von 1024 Bit arbeiten, um einen direkten Vergleich mit den Entwürfen dieser Diplomarbeit führen zu können. Die erstellten Entwürfe Montgomery und „Carry-Save“ können sich durchaus mit den aufgeführten Chips messen, denn bei einer Taktfrequenz von 25 MHz erreicht der Entwurf „Carry-Save (parallel)“ immerhin eine Datenrate von 14,1 KBit/s.

6.4 Ausblick

Die Skalierbarkeit der Entwürfe ermöglicht nicht nur in Zukunft, die Schlüssellängen weiter zu vergrößern, sondern selbstverständlich auch, sie zu verkleinern. Auf diesem Wege könnten Produkte für den Massenmarkt mit niedrigeren Sicherheitsanforderungen („low security“) entstehen. In der Abbildung 6.16: „Abschätzung (128 Bit)“ und der Abbildung 6.17: „Durchsatz (128 Bit)“ sind die erwarteten Daten für eine Schlüssellänge von 128 Bit zusammengefaßt. Die gewählte Taktfrequenz von 10 MHz wurde bewußt niedrig angesetzt. Die Fläche für die Entwürfe Montgomery und „Carry-Save“ wurden wieder entsprechend Kapitel „6.2.2 Fläche“ für den Full-Custom-Entwurf abgeschätzt. Da nur ein Achtel der Zellen des 1024-Bit Entwurfs benötigt wurden, reduzierte sich die Fläche um genau diesen Faktor. Die Chipfläche der „SOSCARD“ konnte im Vergleich zur 1024-Bit Variante nur um den Faktor vier verringert werden, weil nur im Bereich der Registergrößen Einsparungen möglich waren.

128 Bit, 10 MHz	*Rate (seriell)*	*Rate (parallel)*	*Fläche (seriell)*	*Fläche (parallel)*
Montgomery	25,4 KBit/s	38,1 KBit/s	15,3 mm²	30,6 mm²
Carry-Save	24,1 KBit/s	36,2 KBit/s	–	9,3 mm²
SOSCARD	33,9 KBit/s	50,9 KBit/s	10,5 mm²	21,1 mm²

Abbildung 6.16: „Abschätzung (128 Bit)“

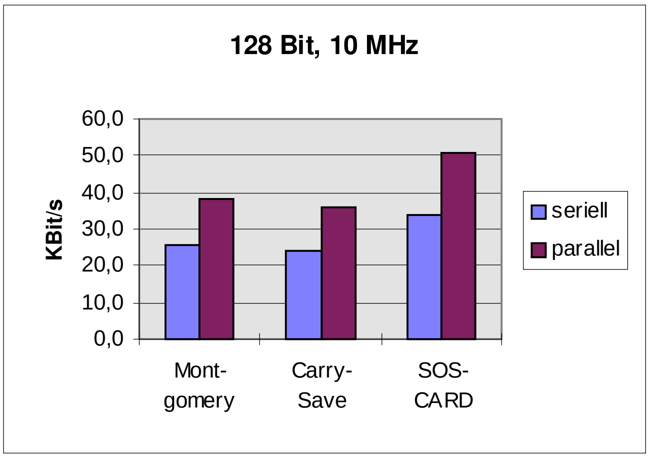

Abbildung 6.17: „Durchsatz (128 Bit)“

6.4.1 Alternative Montgomery Implementation

Von Iwamura, Matsumoto und Imai kommt ein weiterer Vorschlag für eine Implementation nach dem Montgomery-Verfahren [IWA92]. Der Aufbau erfolgt analog zu der bereits vorgestellten Methode nach Sauerbrey mit Hilfe eines systolischen Arrays (siehe Abbildung 6.18: „Systolisches Array (Iwamura)“). Zur Verarbeitung von 1024-Bit Zahlen ist ein Feld mit 1025 Zellen nötig. Im Gegensatz zur Realisierung nach Sauerbrey [SAU93] sind die Zellen speziell für die Berechnung des Montgomery-Produktes entwickelt und optimiert worden.

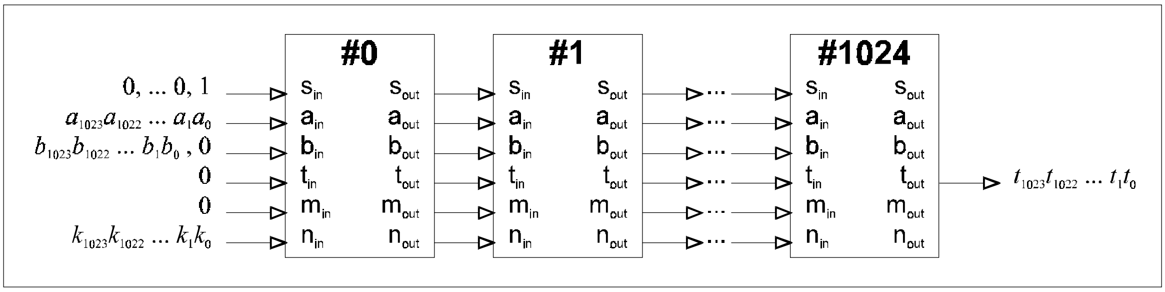

Abbildung 6.18: „Systolisches Array (Iwamura)“

Das systolische Feld ist in der Lage, die Berechnung des Montgomery-Produktes

$$t = MP(a,b,k,2^{1024}) = \frac{a \cdot b}{2^{1024}} \bmod k$$

durchzuführen. Der zugrundeliegende Algorithmus wurde bereits im Kapitel „5.1.1 Algorithmen Montgomery-Produkt“ auf Seite 61 erläutert. Die Berechnung erfolgt gemäß Abbildung 5.2: „Algorithmus Montgomery-Produkt (2)“. Es werden a ($a_{1023}a_{1022} \ldots a_1a_0$), b ($b_{1023}b_{1022} \ldots b_1b_0$) und k ($k_{1023}k_{1022} \ldots k_1k_0$) seriell, beginnend mit dem LSB, links an das Feld (Zelle #0) angelegt. Wobei das LSB von b erst im zweiten Takt angelegt wird. Im ersten Takt wird eine Null benötigt. Das Ergebnis t ($t_{1023}t_{1022} \ldots t_1t_0$) wird seriell von der Zelle #1024 ausgegeben. Das erste Bit des Ergebnisses steht nach einer Latenzzeit von 1026 Takten zur Verfügung. Das Ergebnis hat insgesamt eine Länge von 2048 Bit. Die niederwertigen 1024 Bit des Ergebnisses sind gemäß des Algorithmus zur Berechnung des Montgomery-Produktes Nullen (siehe „5.1.1 Algorithmen Montgomery-Produkt“ auf Seite 61). Die hochwertigen 1024 Bit sind das eigentliche Ergebnis der Berechnung. Für eine Berechnung sind somit insgesamt 3074 Takte notwendig. Wird das Feld im Pipelinebetrieb betrieben, reduzieren sich die durchschnittlich benötigten Taktzyklen auf 2048.

Die folgende Abbildung 6.19: „Zelle Iwamura“ zeigt den Aufbau einer Zelle gemäß der Realisierung nach Iwamura. Es werden lediglich 13 Flip-Flops, zwei UND-Gatter und ein Addierer benötigt.

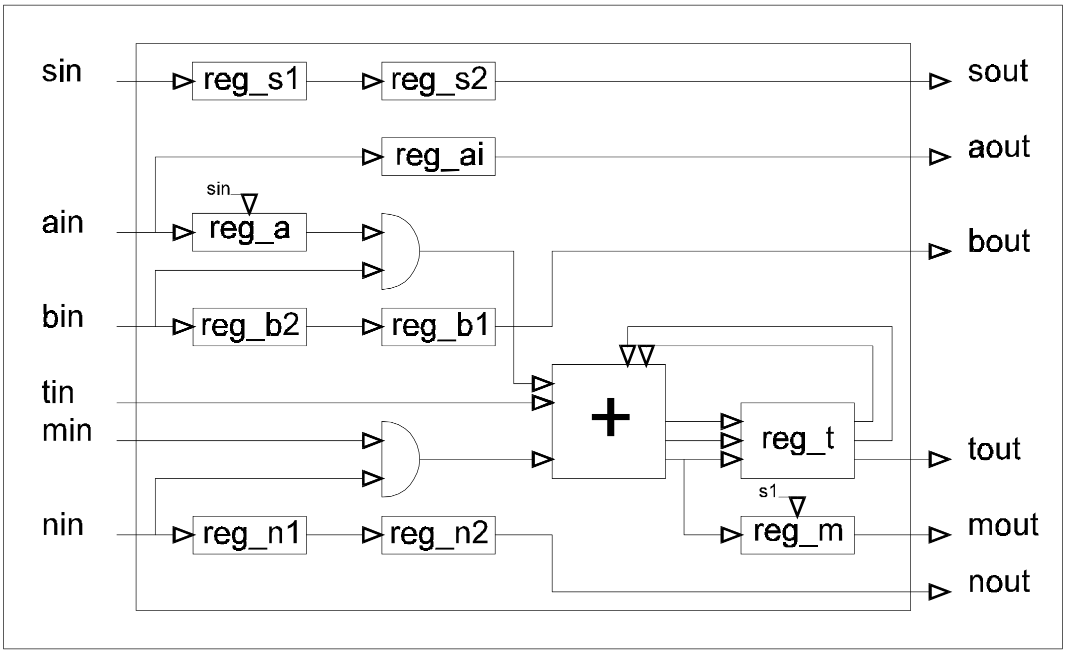

Abbildung 6.19: „Zelle Iwamura“

Die Abbildung 6.20: „Vergleich der Zellen (Montgomery)“ stellt die Eigenschaften der Implementierungen nach Sauerbrey und Iwamura gegenüber. Beide Entwürfe benötigen etwa die gleiche Anzahl von Taktzyklen für eine Modulo-Multiplikation. Die Zelle nach Iwamura ist nur etwa halb so groß wie die nach Sauerbrey. Die maximale Signalverzögerung ist geringer.

	Sauerbrey	*Iwamura*	
Anzahl Standardzellen	66	28	42 %
Fläche (SYNOPSYS)	118.219	65.514	55 %
benötigte Zellen	1536	1024	67 %
Längster Pfad	10,5 ns	8,3 ns	79 %
Takte pro Modulo-Multiplikation	2051	2048	~ 100 %

Abbildung 6.20: „Vergleich der Zellen (Montgomery)“

Für einen Full-Custom-Entwurf mit zwei parallelen Modulo-Multiplizierern ergibt sich für eine Realisierung nach Iwamura eine geschätzte Fläche von knapp 90 mm^2. Die Verschlüsselungsrate beträgt 47,6 KBit/s bei einer Taktfrequenz von 100 MHz. Leider wurde die Veröffentlichung [IWA92] erst am Ende der Zeit für die Diplomarbeit entdeckt, so daß weitergehende Untersuchungen nicht mehr möglich waren.

7 Danksagung

Meinen besonderen Dank möchte ich meinem Diplomarbeitsbetreuer Herrn Dr. Reinhard Rauscher aussprechen. Er stellte den Kontakt zur Firma Philips Semiconductors Hamburg her und begleitete das Entstehen der Arbeit über viele Monate als stets ansprechbarer Ratgeber. Ohne seine Anregungen und Unterstützung wäre das Ergebnis in dieser Form nicht möglich gewesen.

Ebenso danke ich meinem Zweitbetreuer Herrn Dr. Martin Lehmann. Er gab mir zahlreiche konstruktive Hinweise zum theoretischen Teil der Arbeit. Danken möchte ich auch der Firma Philips Semiconductors Hamburg, insbesondere den Herren Dr. Steffen Drews und Helmut Hörner, für die Anregungen aus der Praxis und die zahlreichen Diskussionen.

Für die Hilfestellung bei der Benutzung der Design-Tools möchte ich Herrn Andreas Maeder danken. Besonderen Dank gebührt Bernhard Dwersteg, Manfred Grove, Carsten Kelling, Kerstin Schumacher und Alke Voigt für ihre persönliche Unterstützung.

8 Literaturverzeichnis

[ATR65]
Atrubin, A. J.: A One-Dimensional Real-Time Interative Multiplier, IEEE Transactions on Computers, Vol 14., Seite 394-399, 1965

[BAR86]
Barrett, Paul D.: Implementing the RSA Public key encryption algorithm on a standard digital signal processor, CRYPRO ’86 [ODL87], Seite 311-323, 1986

[BAU93]
Bauer, Friedrich L.: Kryptologie: Methoden und Maximen - Berlin; Heidelberg, 1993

[BER68]
Berlekamp, Elwyn R.: Algebraic Coding Theory - New York: MacGraw-Hill, 1968

[BET83]
Beth, Thomas: Kryptographie: Eine Einführung in die Methoden und Verfahren der geheimen Nachrichtenübermittlung - Stuttgart: Teubner, 1983

[BET92]
Beth, Thomas; Frisch, Markus; Simmons, Gustavus J. [Hrsg.]: Public-Key Cryptography: State of the Art and Future Directions - Berlin; Heidelberg: Springer-Verlag, 1992

[BEU87]
Beutelspacher, Albrecht: Kryptologie - Braunschweig: Vieweg, 1987

[BEU95]
Beutelspacher, Albrecht; Schwenk, Jörg; Wolfenstetter, Klaus-Dieter: Moderne Verfahren der Kryptographie - Braunschweig: Vieweg, 1995

[BOS93]
Bosselares, Antoon; Govaerts, René; Vandewalle, Joos: Comparison of three modular reduction functions, CRYPTO ’93 [STI94], Seite 175 - 186, 1993

[BRA90]
Brassard, G. [Hrsg.]: Advances in cryptology - CRYPTO ’89: Annual conference, proceedings: Theory and applications of cryptology; August 20 - 24, 1989 - Berlin; Heidelberg: Springer-Verlag, 1990

[BRI82]
Brickell, Ernest. F.: A fast modular multiplication algorithm with applications to two key cryptography, CRYPTO ’82, Seite 51 - 60, 1982

[BRI89]
Brickell, Ernest. F.: A Survey of Hardware Implementations of RSA, CRYPTO ’89 [BRA90], Seite 368 - 370, 1989

[CHA88]
Chaum, D.; Price, W. [Hrsg.]: Eurocrypt ‘87: Advances in cryptology; Theory and application of cryptographic techniques; European annual conference, proceedings; 13 - 15 April 1987 - Berlin: Springer, 1988

[DAM91]
Damgård, I. B. (Ed.): Advances in Cryptology - EUROCRYPT ’90: Workshop on the Theory and Application of Cryptographic Techniques, Aarhus, Denmark, May 1990, Proceedings - Berlin; Heidelberg: Springer-Verlag, 1991

[DEN82]
Denning, Dorothy E.: Cryptography and data security - Reading, Massachusetts: Addison-Wesley, 1982

[DIF76]
Diffie, Whitfield; Hellman, Martin E.: New Directions in Cryptography - IEEE Transactions on Information Theory, IT - 22, Seite 644 - 654, 1976

[DWE95]
Dwersteg, Bernhard: Konzeption und Entwicklung eines Histogrammprozessors als ASIC, Diplomarbeit - Rellingen, April 1995

[FEI92]
Feigenbaum, Joan (Ed.): Advances in Cryptology - CRYPTO ‘91: Proceedings, Santa Barbara, California, USA, August 11 - 15, 1991 - Berlin; Heidelberg: Springer-Verlag, 1992

[FUM88]
Fumy, Walter; Rieß, Hans Peter: Kryptographie: Entwurf und Analyse symmetrischer Kryptosysteme - München; Wien: Oldenbourg, 1988

[GRU94]
Gruska, Jozef: Theoretische Grundlagen der Programmierung, Prüfungsunterlagen zur Diplomprüfung Theoretische Informatik - Universität Hamburg, Fachbereich Informatik, 1994

[GOL94]
Gollmann, Dieter: Algorithmenentwurf in der Kryptographie (Aspekte komplexer Systeme) - Mannheim; Leipzig; Wien; Zürich: BI Wissenschaftsverlag, 1994

[GUE88]
Günther, C. [Hrsg.]: Eurocrypt '88: Advances in cryptology; Theory and application of cryptographic techniques; European annual conference, proceedings; 25 - 27 may 1988 - Berlin: Springer, 1988

[HON96]
Hong, Seong-Min u.a.: New Modular Multiplication Algorithm for Fast Modular Exponentiation, Eurocrypt '96 [MAU96], Seite 166 - 177, 1996

[HOO88]
Hoornaert, Frank u.a.: Fast RSA Hardware: Dream or Reality?, Eurocrypt '88 [GUE88], Seite 257 - 262, 1988

[HOR85]
Horster, Patrick: Kryptologie - Mannheim; Wien; Zürich: Bibliographisches Institut, 1985

[IWA92]
Iwamura, Keiichi u.a.: Systolic-arrays for Modular Exponentiation using Montgomery Method, Eurocrypt '92 [RUE93], Seite 477 - 481, 1992

[KNU81]
Knuth, Donald E.: The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Second Edition - Reading, Massachusetts: Addison-Wesley, 1981

[KOB87]
Koblitz, Neal: A Course in Number Theory and Cryptography - New York; Berlin; Heidelberg: Springer-Verlag, 1987

[KOB96]
Koblitz, Neal (Ed.): Advances in Cryptology: Proceedings / CRYPTO '96: 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18 - 22, 1996

[KUN78]
Kung, H. T.; Leierson, Charles E.: Systolic Arrays for VLSI, in Sparse Matrix Proceedings, Seite 256 - 282, SIAM, Philadelphia, 1978

[LIP91]
Lippitsch, P.; Posch, Karl C.; Posch, Reinhard: Multiplication As Parallel As Possible, aus Zima, H. P.: Parallel Computation, Seite 350 - 358, 1991

[MAE93]
Maeder, Andreas: VHDL Kurzbeschreibung zum Projekt „VLSI-Entwurf", 1993

[MAS69]
Massey, J. L.: Shift-Register Synthesis and BCH-Decoding, IEEE Trans. Inf. Theory, IT-15, Seite 122 - 127, 1969

[MAU96]
Maurer, U. [Hrsg.]: Eurocrypt '96: Advances in cryptology; Theory and application of cryptographic techniques; European annual conference, proceedings; 12 - 16 may 1996 - Berlin: Springer, 1996

[MON85]
Montgomery, P. L.: Modular Multiplication Without Trial Division, Mathematics of Computation, Vol. 44, pp. 519 - 521, 1985

[MUL63]
Muller, D. E.: Asynchronous Logic and Application to information processing, 1963

[NOR81]
Norris, M. J.; Simmons, G. J.: Algorithms for high-speed modular arithmetics, Congressus Numerantium, Vol. 31, pp. 151 - 163, 1981

[ODL87]
Odlyzko, Andrew M. (Ed.): Advances in Cryptology: Proceedings / CRYPTO '86: International Cryptology Conference, Santa Barbara, California, USA, August 11 - 15, 1986 - Berlin; Heidelberg: Springer-Verlag, 1987

[OMO94]
Omondi, Amos R.: Computer Arithmetic Systems - Algorithms, Architecture and Implementations - Prentice Hall International, 1994

[ORT86]
Orton, G. A. u.a.: VLSI implementation of Public-Key encryption algorithms, CRYPTO '86 [ODL87], Seite 277 - 301, 1986

[ORU90]
Orup, Holger u.a.: VICTOR an efficient RSA hardware implemenation, EUROCRYPT '90 [DAM91], Seite 245-252, 1990

[POS90]
Posch, Karl C.; Posch, Reinhard: Approaching encryption at ISDN speed using partial parallel modulus multiplication, Microprocessing and Microprogramming, Vol. 29, Seite 177 - 184 - Amsterdam, 1990

[POS95]
Posch, Karl C.; Posch, Reinhard: Modulo Reduction in Residue Number Systems - IEEE Transactions on parallel and distributed systems, May 1995

[POS96]
Posch, Karl C.; Posch, Reinhard: Kryptografische Co-Prozessoren: Möglichkeiten neuer Technologien; aus Telematik Nr. 2/96, 1996

[QUI90]
Quisquater, Jean-Jacques; Vandewalle, Joos (Eds.): Advances in Cryptology - EUROCRYPT '89: Workshop on the Theory and Application of Cryptographic Techniques, Houthalen, Belgium, April 1989, Proceedings - Berlin; Heidelberg: Springer-Verlag, 1990

[RAN95]
Rankl, Wolfgang; Wolfgang Effing: Handbuch der Chipkarten: Aufbau, Funktionsweise, Einsatz - München; Wien: Hanser, 1995

[RIE85]
Riesel, Hans: Prime numbers and computer methods for factorization - Stuttgart: Birkhäuser, 1985

[RIV78]
Rivest, Ronald L.; Shamir, Adi; Adleman, Len: A Method for Obtaining Digital Signatures and Public key Cryptosystems, Communications of the ACM, Vol. 21, No. 2, Seite 120 - 126, Februar 1978

[RUE93]
Rueppel, R. [Hrsg.]: Eurocrypt '92: Advances in cryptology; Theory and application of cryptographic techniques; European annual conference, proceedings; 24 - 28 may 1992 - Berlin: Springer, 1993

[RYS80]
Ryska, Norbert; Herda, Siegfried: Kryptographische Verfahren in der Datenverarbeitung - Berlin; Heidelberg; New York: Springer, 1980

[SAL90]
Salomaa, Arto: Public-Key Cryptography - Berlin; Heidelberg: Springer-Verlag, 1990

[SAU93]
Sauerbrey, Jörg: Langzahl-Modulo-Arithmetik für kryptographische Verfahren: Konzepte zur effizienten Implementierung - Wiesbaden: DUV Deutscher Universitäts-Verlag, 1993

[SCH95]
Schmeck, Hartmut: Analyse von VLSI-Algorithmen - Heidelberg; Berlin; Oxford: Akademischer Verlag, 1995

[SCH96]
Schneier, Bruce: Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C - Bonn: Addison-Wesley, 1996

[SED87]
Sedlak, Holger: The RSA cryptography processor, Eurocrypt '87 [CHA88], Seite 95 - 105, 1987

[SIM92]
Simmons, Gustavus J. [Ed.]: Contemporary cryptology: The science of information integrity - New York: IEEE Press, 1992

[SPA76]
Spaniol, Otto: Arithmetik in Rechenanlagen: Logik und Entwurf - Stuttgart: Teubner, 1976

[STI94]
Stinson, Douglas R. (Ed.): Advances in Cryptology - CRYPTO '93: 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 1993, Proceedings - Berlin; Heidelberg: Springer-Verlag, 1994

[VAN89]
Vandemeulebroecke, André u.a.: A Single Chip 1024 Bits RSA Processor, EUROCRYPT '89 [QUI90], Seite 219 - 236, 1989

[VUI83]
Vuillemin, Jean: A Very Fast Multiplication Algorithm For VLSI Multiplication, Institut National de Recherche en Informatique et en Automatique, Domaine de Voluceau, 1983

[WAL91]
Walter, Colin D.: Faster Modular Multiplication by Operand Scaling, CRYPTO '91 [FEI92], Seite 313 - 323, 1991

[YAC90]
Yacobi, Y.: Exponentiating faster with addition chains, EUROCRYPT '90 [DAM91], Seite 222 - 229, 1990

9 Anhang

9.1 Abbildungsverzeichnis

9.2 Quellcode

Im folgenden und auf der beiliegenden Diskette findet sich der gesammelte Sourcecode dieser Arbeit. Die mit einem Stern (*) gekennzeichneten Dateien sind nur auf der Diskette vorhanden. Bei ihnen handelt es sich um Hilfsdateien für die Simulation. Die folgende Übersicht entspricht der Anordnung der Textdateien:

Modulo-Exponentiation.cc

hochsprachlich

FIRSTTIME.VHD
FTMULT.VHD

montgomery

M1ARRAY.VHD *
M1STI.CMS *
M1TESTIT.VHD *
MFSM.VHD
MMULT.VHD
MONTPRO.VHD
MREG.VHD
MRSA.VHD
MZELLE.VHD
MZ_DREI.VHD
MZ_VIER.VHD
MZ_ZWEI.VHD

carrysave

ADDOF.VHD
AMULT.VHD
BMULT.VHD
COUNTER.VHD
EXP.VHD
FCS.VHD
FELD.VHD
SHIFTBUF.VHD
XMULT.VHD
ZELLE.VHD

soscard

ALU.VHD
MCOUNT.VHD
MODMULT.VHD
MULT16.VHD
REG.VHD
RSA.VHD

iwamura

ARRA.CMS *
ARRAY.VHD
BEST.VHD
TESTIT.VHD *

C-PROGRAMM

```
/****************************************
 *                                      *
 *   Konzeption und Implementation eines  *
 * Hardwareakzelerators zum RSA-Verfahren *
 *                                      *
 *   Diplomarbeit - Lars H. Hahn - 1997   *
 *                                      *
 ****************************************

 * g++ -o mm Modulo-Exponentiation.cc -lm */

#include <sys/time.h>
#include <sys/resource.h>
#include <iostream.h>
#include <Integer.h>

float stopwatch();

main()
{
  Integer i, x, y, k, a = 1;

  y = pow(2,1023)-361;
  k = pow(2,1024)-105;
  cout << "Eine Zahl: ";
  cin >> x;

  cout << "X: " << x << "\n";
  cout << "Y: " << y << "\n";
  cout << "K: " << k << "\n";

  stopwatch();
  while(y != 0)
  {
    if ((y & 1) == 1)
    {
      y--;
      a = (a * x) % k;
    }
    y = y >> 1;
    x = (x * x) % k;
  }
  cout << "x^y mod k: " << a << "\n";
  stopwatch();
}

float stopwatch()
{
  static long last_ts=0L,last_tus=0L,ts=0L,tus=0L;
  static int  firstcall = 1;
  float       s = 0.0;

  struct rusage rusage;

  if (firstcall==1)
  {
    firstcall = 0;
    if ((getrusage(RUSAGE_SELF, &rusage))==0)
    {
      last_ts = rusage.ru_utime.tv_sec;
      last_tus = rusage.ru_utime.tv_usec;
    }
```

```
  }
  else
  {
    if ((getrusage(RUSAGE_SELF, &rusage))==0)
    {
      ts = rusage.ru_utime.tv_sec;
      tus = rusage.ru_utime.tv_usec;
      s = ((float)(ts-last_ts)) +((float)(tus-last_tus))/1.0E6;
      printf("   CPU TIME = %.2f seconds.\n",s);
      last_ts = ts; last_tus=tus;
    }
  }
  return(s);
}
```

HOCHSPRACHLICH

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: HOCHSPRACHLICH                --
--                                        --
-- firsttime.vhd                          --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity firsttime is
  port(
    reset,clk:  in std_logic;
    y        :  in std_logic;
    x,k      :  in UNSIGNED(63 downto 0);
    z        : out UNSIGNED(63 downto 0)
    );
end firsttime;

architecture behaviour of firsttime is

component ftmult
  port(
    a,b,k    :  in UNSIGNED(63 downto 0);
    c        : out UNSIGNED(63 downto 0)
    );
end component;

type state_type is (init,square,multiply,cond,ready);
signal current_state, next_state: state_type;
signal count: integer range 0 to 63;
signal a,b,c,r1,r2: UNSIGNED(63 downto 0);

begin

  my_ftmult: ftmult port map (a,b,k,c);
  z <= c;

  FSM: process(current_state, y, count)
  begin
    case current_state is
    when init     => if (y = '1') then next_state <= multiply;
                     else next_state <= square; end if;
    when multiply => next_state <= square;
    when square   => next_state <= cond;
    when cond     => if (count = 63) then next_state <= ready;
                     elsif (y = '1') then next_state <= multiply;
                     else next_state <= square; end if;
    when ready    => next_state <= init;
    end case;
  end process;
```

```
  COUNTIT: process(clk, reset, current_state, count)
  begin
    if (clk'event and clk = '1') then
      if ((current_state = init) OR (count = 63)) then
        count <= 0;
      elsif (current_state = cond) then
        count <= count + 1;
      end if;
    end if;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= init;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

  a <= r2 when (current_state = square) else r1;
  b <= r2;

  MULT: process(clk,reset,current_state,x,c)
  begin
    if (reset = '0') then
      r1 <=
"0000000000000000000000000000000000000000000000000000000000000000";
      r2 <=
"0000000000000000000000000000000000000000000000000000000000000000";
    elsif (clk'event and clk = '1') then
      if (current_state = init) then
        r1 <=
"0000000000000000000000000000000000000000000000000000000000000001";
        r2 <= x;
      elsif (current_state = square) then
        r2 <= c;
      elsif (current_state = multiply) then
        r1 <= c;
      end if;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: HOCHSPRACHLICH                --
--                                        --
-- ftmult.vhd                             --
--                                        --
--------------------------------------------

library IEEE, DW02;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use DW02.DW02_components.all;

entity ftmult is
  port(
    a,b,k    :  in UNSIGNED(63 downto 0);
    c        : out UNSIGNED(63 downto 0)
    );
end ftmult;

architecture behaviour of ftmult is

signal chelp: UNSIGNED(127 downto 0);

begin

  chelp <= a * b;
  c <= chelp rem k;

end;
```

MONTGOMERY

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: MONTGOMERY                    --
--                                        --
-- mfsm.vhd - 08.06.97                    --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mfsm is
  port(
    reset,clk,y                 :  in std_logic;
    nullgo,newy,mcountnull,
    nowsquare,nowmultiply,nowinit: out std_logic
    );
end mfsm;

architecture behaviour of mfsm is

type state_type is (init,square,multiply,cond,ready);
signal current_state, next_state: state_type;
signal count: integer range 0 to 1023;
signal mcount: integer range 0 to 2047;

begin

  nullgo <= '1' when (mcount >= 1024) else '0';
  newy <= '1' when ((current_state = init) or (current_state = cond))
else '0';
  mcountnull <= '1' when (mcount = 0) else '0';
  nowsquare <= '1' when (current_state = square) else '0';
  nowmultiply <= '1' when (current_state = multiply) else '0';
  nowinit <= '1' when (current_state = init) else '0';

  FSM: process(current_state, y, count, mcount)
  begin
    case current_state is
    when init     => if (mcount = 1023) then next_state <= multiply;
                     else next_state <= init; end if;
    when multiply => if (mcount = 2047) then next_state <= square;
                     else next_state <= multiply; end if;
    when square   => if (mcount = 2047) then next_state <= cond;
                     else next_state <= square; end if;
    when cond     => if (count = 1023) then next_state <= ready;
                     elsif (y = '1') then next_state <= multiply;
                     else next_state <= square; end if;
    when ready    => next_state <= init;
    end case;
  end process;

  COUNTIT: process(clk, reset, current_state, count)
```

```
  begin
    if (clk'event and clk = '1') then
      if ((current_state = init) OR (count = 1023)) then
        count <= 0;
      elsif (current_state = cond) then
        count <= count + 1;
      end if;
    end if;
  end process;

  MCOUNTIT: process(clk, reset, current_state, mcount)
  begin
    if (clk'event and clk = '1') then
      if ((current_state = init) OR (mcount = 2047)) then
        mcount <= 0;
      elsif ((current_state = square) OR (current_state = multiply))
then
        mcount <= mcount + 1;
      end if;
    end if;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= init;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: MONTGOMERY                    --
--                                        --
-- mmult.vhd                              --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mmult is
  port(
    clk,reset: in std_logic;
    mx,my,mz : in std_logic;
    mq       : in std_logic_vector(1 downto 0);
    mp       : out std_logic
    );
end mmult;

architecture behaviour of mmult is

component mzelle
  port(
    clk,reset     : in std_logic;
    zin           : in std_logic;
    xin, yin, pin : in std_logic;
    qin           : in std_logic_vector(1 downto 0);
    zaus          : out std_logic;
    xaus,yaus,paus: out std_logic;
    qaus          : out std_logic_vector(1 downto 0)
    );
end component;

type AR1 is array (0 to 512) of std_logic;
type AR2 is array (0 to 512) of std_logic_vector(1 downto 0);
signal z, x, y, p: AR1;
signal q: AR2;
signal c1,zp,xp,yp: std_logic;
signal qp: std_logic_vector(1 downto 0);
begin
  GF: for i in 0 to 511 generate
      my_mult: mzelle port
map(clk,reset,z(i),x(i),y(i),p(i+1),q(i),z(i+1),x(i+1),y(i+1),p(i),q(i+1
));
  end generate;

  p(512) <= '0';
  z(0)   <= mz;
  q(0)   <= mq;
  x(0)   <= mx;
  y(0)   <= my;
  mp     <= p(0);

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: MONTGOMERY                    --
--                                        --
-- montpro.vhd                            --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity montpro is
  port(
    clk       : in std_logic;
    reset     : in std_logic;
    x,y,k,k1,z: in std_logic;
    t         : out std_logic
    );
end montpro;

architecture behaviour of montpro is

component mmult
  port(
    clk,reset: in std_logic;
    mx,my,mz : in std_logic;
    mq       : in std_logic_vector(1 downto 0);
    mp       : out std_logic
    );
end component;

signal p,reg_p,v: std_logic;
signal zero,phlp: std_logic_vector(1 downto 0);

begin
  mult_a: mmult port map(clk,reset,x ,y,z,zero,p);
  mult_b: mmult port map(clk,reset,k1,p,z,zero,v);
  mult_c: mmult port map(clk,reset,k,v,z,phlp,t);

  zero <= "00";
  phlp <= '0'&reg_p;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      reg_p <= '0';
    elsif (clk'event and clk = '1') then
      reg_p <= p;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: MONTGOMERY                    --
--                                        --
-- mreg.vhd                               --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mreg is
  port(
    clk    : in std_logic;
    rein,go: in std_logic;
    raus   : out std_logic
    );
end mreg;

architecture behaviour of mreg is

signal reg: std_logic_vector(1023 downto 0);

begin

  raus <= reg(0);

  process(clk,reg,rein,go)
  begin
    if (clk'event and clk='1') then
      if (go = '1') then
        reg <= rein&reg(1023 downto 1);
      end if;
    end if;
  end process;
end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: MONTGOMERY                    --
--                                        --
-- mrsa.vhd                               --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mrsa is
  port(
    clk,reset: in std_logic;
    x,y,k,k1: in std_logic;
    a,newy   : out std_logic
    );
end mrsa;

architecture behaviour of mrsa is

component montpro
  port(
    clk,reset  : in std_logic;
    x,y,k,k1,z: in std_logic;
    t          : out std_logic
    );
end component;

component mreg
  port(
    clk    : in std_logic;
    rein,go: in std_logic;
    raus   : out std_logic
    );
end component;

component mfsm
  port(
    reset,clk,y                  :  in std_logic;
    nullgo,newy,mcountnull,
    nowsquare,nowmultiply,nowinit: out std_logic
    );
end component;

signal modx,mody,modz,moda,sin,sgo,sout,min,mgo,mout: std_logic;
signal nullgo,mcountnull,nowsquare,nowmultiply,nowinit: std_logic;
signal xhelp, yhelp,shelp,mhelp: std_logic;

begin
  my_montpro: montpro port map(clk,reset,modx,mody,k,k1,modz,moda);
  reg_s: mreg port map(clk,sin,sgo,sout);
  reg_m: mreg port map(clk,min,mgo,mout);
  my_mfsm: mfsm port
map(reset,clk,y,nullgo,newy,mcountnull,nowsquare,nowmultiply,nowinit);
```

```
  a <= mout;
  modx <= '0' when (nullgo = '1') else xhelp; -- Nullen nachziehen
  mody <= '0' when (nullgo = '1') else yhelp;
  xhelp <= sout when (nowsquare = '1') else mout;
  yhelp <= sout;
  modz <= mcountnull;
  shelp <= moda when (nowsquare = '1') else sout;
  sin <= x when (nowinit = '1') else shelp;
  sgo <= (nowsquare AND nowmultiply AND nowinit);
  mhelp <= moda when (nowmultiply = '1') else mout;
  min <= mcountnull when (nowinit = '1') else moda;
  mgo <= (nowmultiply AND nowinit);

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: MONTGOMERY                    --
--                                        --
-- mz_drei.vhd                            --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mz_drei is
  port(
    reset,clk     : in std_logic;
    zin           : in std_logic;
    xin, yin, pin : in std_logic_vector(2 downto 0);
    qin           : in std_logic_vector(6 downto 0);
    zaus          : out std_logic;
    xaus,yaus,paus: out std_logic_vector(2 downto 0);
    qaus: out std_logic_vector(6 downto 0)
    );
end mz_drei;

architecture behaviour of mz_drei is
type state_type is (z0, Z1, Z2, Z3);
signal current_state, next_state: state_type;
signal xe,ye,xt,yt,x0,y0,p: std_logic_vector(2 downto 0);
signal zs: std_logic_vector(6 downto 0);
signal ue: std_logic_vector(4 downto 0);
signal s: std_logic_vector(7 downto 0);

begin
  zaus <= '1' when (current_state=Z3) else '0';
  xaus <= xt;
  yaus <= yt;
  paus <= s(2 downto 0);
  s <= ('0'&zs) + ("00000"&p) + ("000"&ue) + ('0'&qin);

  FSM: process(current_state,zin)
  begin
    case current_state is
      when Z0 =>
        if (zin='1') then next_state <= Z1;
        else next_state <= Z0; end if;
      when Z1 =>
        next_state <= Z2;
      when Z2 =>
        next_state <= Z3;
      when Z3 =>
        next_state <= Z0;
    end case;
  end process;

  OP: process(reset,clk,current_state,zin)
  begin
```

```
    if (reset = '0') then
      p <= "000";
      ue <= "00000";
      xe <= "000"; ye <= "000";
      x0 <= "000"; y0 <= "000";
      xt <= "000"; yt <= "000";
    elsif (clk='1' and clk'event) then
      p <= pin;
      ue <= s(7 downto 3);
      if ((current_state=Z0) and (zin='1')) then
        xe <= xin;
        ye <= yin;
      elsif ((current_state=Z1) and (zin='0')) then
        x0 <= xin;
        y0 <= yin;
      else
        xt <= xin;
        yt <= yin;
      end if;
    end if;
  end process;

  SUM: process(current_state,zin,xin,yin,xe,ye,x0,y0)
  begin
    if ((current_state=Z0) and (zin='1')) then
      qaus <= "0000000";
      zs <= ('0'&(xin * yin));
    elsif ((current_state=Z1) and (zin='0')) then
      qaus <= ('0'&(xin * yin));
      zs <= ('0'&(xin * ye)) + ('0'&(xe * yin));
    else
      qaus <= ('0'&(xin * y0)) + ('0'&(x0 * yin));
      zs <= ('0'&(xin * ye)) + ('0'&(xe * yin));
    end if;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= Z0;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: MONTGOMERY                    --
--                                        --
-- mz_vier.vhd                            --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mz_vier is
  port(
    reset,clk     : in std_logic;
    zin           : in std_logic;
    xin, yin, pin : in std_logic_vector(3 downto 0);
    qin           : in std_logic_vector(8 downto 0);
    zaus          : out std_logic;
    xaus,yaus,paus: out std_logic_vector(3 downto 0);
    qaus          : out std_logic_vector(8 downto 0)
    );
end mz_vier;

architecture behaviour of mz_vier is
type state_type is (z0, Z1, Z2, Z3);
signal current_state, next_state: state_type;
signal xe,ye,xt,yt,x0,y0,p: std_logic_vector(3 downto 0);
signal zs: std_logic_vector(8 downto 0);
signal ue: std_logic_vector(5 downto 0);
signal s: std_logic_vector(9 downto 0);

begin
  zaus <= '1' when (current_state=Z3) else '0';
  xaus <= xt;
  yaus <= yt;
  paus <= s(3 downto 0);
  s <= ('0'&zs) + ("000000"&p) + ("0000"&ue) + ('0'&qin);

  FSM: process(current_state,zin)
  begin
    case current_state is
      when Z0 =>
        if (zin='1') then next_state <= Z1;
        else next_state <= Z0; end if;
      when Z1 =>
        next_state <= Z2;
      when Z2 =>
        next_state <= Z3;
      when Z3 =>
        next_state <= Z0;
    end case;
  end process;

  OP: process(reset,clk,current_state,zin)
  begin
```

```
    if (reset = '0') then
      p <= "0000";
      ue <= "000000";
      xe <= "0000"; ye <= "0000";
      x0 <= "0000"; y0 <= "0000";
      xt <= "0000"; yt <= "0000";
    elsif (clk='1' and clk'event) then
      p <= pin;
      ue <= s(9 downto 4);
      if ((current_state=Z0) and (zin='1')) then
        xe <= xin;
        ye <= yin;
      elsif ((current_state=Z1) and (zin='0')) then
        x0 <= xin;
        y0 <= yin;
      else
        xt <= xin;
        yt <= yin;
      end if;
    end if;
  end process;

  SUM: process(current_state,zin,xin,yin,xe,ye,x0,y0)
  begin
    if ((current_state=Z0) and (zin='1')) then
      qaus <= "000000000";
      zs <= ('0'&(xin * yin));
    elsif ((current_state=Z1) and (zin='0')) then
      qaus <= ('0'&(xin * yin));
          zs <= ('0'&(xin * ye)) + ('0'&(xe * yin));
    else
      qaus <= ('0'&(xin * y0)) + ('0'&(x0 * yin));
          zs <= ('0'&(xin * ye)) + ('0'&(xe * yin));
    end if;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= Z0;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: MONTGOMERY                    --
--                                        --
-- mz_zwei.vhd                            --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mz_zwei is
  port(
    clk,reset     : in std_logic;
    zin           : in std_logic;
    xin, yin, pin : in std_logic_vector(1 downto 0);
    qin           : in std_logic_vector(4 downto 0);
    zaus          : out std_logic;
    xaus,yaus,paus: out std_logic_vector(1 downto 0);
    qaus          : out std_logic_vector(4 downto 0)
    );
end mz_zwei;

architecture behaviour of mz_zwei is
type state_type is (z0, Z1, Z2, Z3);
signal current_state, next_state: state_type;
signal xe,ye,xt,yt,x0,y0,p: std_logic_vector(1 downto 0);
signal zs: std_logic_vector(4 downto 0);
signal ue: std_logic_vector(3 downto 0);
signal s: std_logic_vector(5 downto 0);

begin
  zaus <= '1' when (current_state=Z3) else '0';
  xaus <= xt;
  yaus <= yt;
  paus <= s(1 downto 0);
  s <= ("0"&zs) + ("0000"&p) + ("00"&ue) + ("0"&qin);

  FSM: process(current_state,zin)
  begin
    case current_state is
      when Z0 =>
        if (zin='1') then next_state <= Z1;
        else next_state <= Z0; end if;
      when Z1 =>
        next_state <= Z2;
      when Z2 =>
        next_state <= Z3;
      when Z3 =>
        next_state <= Z0;
    end case;
  end process;

  OP: process(reset,clk,current_state,zin)
  begin
```

```
    if (reset = '0') then
      p <= "00";
      ue <= "0000";
      xe <= "00"; ye <= "00";
      x0 <= "00"; y0 <= "00";
      xt <= "00"; yt <= "00";
    elsif (clk'event and clk='1') then
      p <= pin;
      ue <= s(5 downto 2);
      if ((current_state=Z0) and (zin='1')) then
        xe <= xin;
        ye <= yin;
      elsif ((current_state=Z1) and (zin='0')) then
        x0 <= xin;
        y0 <= yin;
      else
        xt <= xin;
        yt <= yin;
      end if;
    end if;
  end process;

  SUM: process(current_state,zin,xin,yin,xe,ye,x0,y0)
  begin
    if ((current_state=Z0) and (zin='1')) then
      qaus <= "00000";
      zs <= ('0'&(xin * yin));
    elsif ((current_state=Z1) and (zin='0')) then
      qaus <= ('0'&(xin * yin));
      zs <= ('0'&(xin * ye)) + ('0'&(xe * yin));
    else
      qaus <= ('0'&(xin * y0)) + ('0'&(x0 * yin));
      zs <= ('0'&(xin * ye)) + ('0'&(xe * yin));
    end if;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= Z0;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: MONTGOMERY                    --
--                                        --
-- mzelle.vhd                             --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mzelle is
  port(
    clk,reset          : in std_logic;
    zin, xin, yin, pin : in std_logic;
    qin                : in std_logic_vector(1 downto 0);
    zaus,xaus,yaus,paus: out std_logic;
    qaus               : out std_logic_vector(1 downto 0)
    );
end mzelle;

architecture behaviour of mzelle is
type state_type is (z0, Z1, Z2, Z3);
signal current_state, next_state: state_type;
signal xe,ye,xt,yt,x0,y0,p: std_logic;
signal zs: std_logic_vector(1 downto 0);
signal ue: std_logic_vector(2 downto 0);
signal s: std_logic_vector(3 downto 0);

begin
  zaus <= '1' when (current_state=Z3) else '0';
  xaus <= xt;
  yaus <= yt;
  paus <= s(0);
  s <= ("00"&zs) + ("000"&p) + ('0'&ue) + ("00"&qin);

  FSM: process(current_state,zin)
  begin
    case current_state is
      when Z0 =>
        if (zin='1') then next_state <= Z1;
        else next_state <= Z0; end if;
      when Z1 =>
        next_state <= Z2;
      when Z2 =>
        next_state <= Z3;
      when Z3 =>
        next_state <= Z0;
    end case;
  end process;

  OP: process(reset,clk,current_state,zin)
  begin
    if (reset = '0') then
      p <= '0';
```

```
      ue <= "000";
      xe <= '0'; ye <= '0';
      x0 <= '0'; y0 <= '0';
      xt <= '0'; yt <= '0';
    elsif (clk'event and clk='1') then
      p <= pin;
      ue <= s(3 downto 1);
      if ((current_state=Z0) and (zin='1')) then
        xe <= xin;
        ye <= yin;
      elsif ((current_state=Z1) and (zin='0')) then
        x0 <= xin;
        y0 <= yin;
      else
        xt <= xin;
        yt <= yin;
      end if;
    end if;
  end process;

  SUM: process(current_state,zin,xin,yin,xe,ye,x0,y0)
  begin
    if ((current_state=Z0) and (zin='1')) then
      qaus <= "00";
      zs(0) <= xin and yin;
      zs(1) <= '0';
    elsif ((current_state=Z1) and (zin='0')) then
      qaus(0) <= xin and yin;
      qaus(1) <= '0';
      if ((xin='1' and ye='1') xor (xe='1' and yin='1')) then
zs(0)<='1'; else zs(0)<='0'; end if;
      if (xin='1' and ye='1' and xe='1' and yin='1') then zs(1)<='1';
else zs(1)<='0'; end if;
    else
      if ((xin='1' and y0='1') xor (x0='1' and yin='1')) then
qaus(0)<='1'; else qaus(0)<='0'; end if;
      if (xin='1' and y0='1' and x0='1' and yin='1') then qaus(1)<='1';
else qaus(1)<='0'; end if;
      if ((xin='1' and ye='1') xor (xe='1' and yin='1')) then
zs(0)<='1'; else zs(0)<='0'; end if;
      if (xin='1' and ye='1' and xe='1' and yin='1') then zs(1)<='1';
else zs(1)<='0'; end if;
    end if;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= Z0;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

end behaviour;
```

CARRY-SAVE

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: CARRY-SAVE                    --
--                                        --
-- addof - 29.04.97                       --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity addof is
  port(
    clk,inof,flagk   : in std_logic;
    sel              : in std_logic_vector(1 downto 0);
    allone,high: out std_logic
    );
end addof;

architecture behaviour of addof is
  signal reg,k: std_logic_vector(15 downto 0);

begin
  allone <= '1' when (reg="1111111111111111") else '0';
  high   <= reg(15);

  process(clk,sel,inof,flagk)

  begin
    if (clk'event and clk='1') then
      case sel is
        when "00" => if (inof='1') then
                       reg <= reg + 1;
                     end if;
        when "01" => if (flagk='1') then
                       if (inof='1') then
                         reg <= reg + 1;
                       end if;
                     else
                       if (inof='1') then
                         reg <= reg + k + 1;
                       else
                         reg <= reg + k;
                       end if;
                     end if;
        when "10" => if (flagk='0') then
                       reg <= (reg(14 downto 0)&inof);
                     else
                       k <= (k(14 downto 0)&inof);
                     end if;
        when others => reg <= reg;
                      k <= k; -- nothing
      end case;
```

```
    end if;

  end process;
end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: CARRY-SAVE                    --
--                                        --
-- amult - 08.05.97                       --
--                                        --
--------------------------------------------

library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity amult is
  port(
    clk, reset                 : in std_logic;
    empty,fempty,ashift,ahigh,
    aallone,cardy,stateb6      : in std_logic;
    s1                         : out std_logic_vector(1 downto 0);
    re,carst,cago,statea3,statea6: out std_logic
    );
end amult;

architecture behaviour of amult is

type state_type is
(add,suba,subs,shift,warten,ready1,ready2,correct1,correct2,correct3);
signal current_state, next_state: state_type;

begin

  re      <= '1' when (current_state=shift) else '0';
  cago    <= '1' when ((current_state=shift) or
(current_state=correct2)) else '0';
  carst   <= '1' when ((current_state=ready2) or
(current_state=correct1) or
                      (current_state=correct3)) else '0';
  statea3 <= '1' when (current_state=ready2) else '0';
  statea6 <= '1' when (current_state=ready1) else '0';

  FSM:
process(clk,empty,fempty,ashift,ahigh,aallone,cardy,stateb6,current_stat
e)
  begin
    case current_state is
        when add =>      s1 <= "00";
                         if (cardy='1') then next_state <= ready1;
                         elsif (ahigh='1') then next_state <= add;
                         elsif (empty='1') then next_state <= warten;
                         else next_state <= shift; end if;
        when suba =>     s1 <= "01";
                         next_state <= shift;
        when shift =>    s1 <= "10";
                         if (ahigh='1') then next_state <= subs;
                         elsif (ashift='1') then next_state <= add;
                         elsif (cardy='1') then next_state <= ready1;
                         elsif (fempty='1') then next_state <= warten;
```

```
                         else next_state <= shift; end if;
        when ready2 =>   s1 <= "11";
                         if (empty='1') then next_state <= warten;
                         elsif (ashift='1') then next_state <= add;
                         else next_state <= shift; end if;
        when warten =>   s1 <= "11";
                         if (empty='1') then next_state <= warten;
                         elsif (ashift='1') then next_state <= add;
                         elsif (cardy='1') then next_state <= ready1;
                         else next_state <= shift; end if;
        when subs =>     s1 <= "01";
                         if (empty='1') then next_state <= warten;
                         elsif (ashift='1') then next_state <= add;
                         elsif (cardy='1') then next_state <= ready1;
                         else next_state <= shift; end if;
        when ready1 =>   s1 <= "11";
                         if (stateb6='1') then next_state <= correct1;
                         else next_state <= ready1; end if;
        when correct1 => s1 <= "01";
                         if (aallone='1') then next_state <= correct2;
                         else next_state <= correct1; end if;
        when correct2 => s1 <= "00";
                         if (cardy='0') then next_state <= correct2;
                         elsif (aallone='1') then next_state <=
correct3;
                         else next_state <= ready2; end if;
        when correct3 => s1 <= "01";
                         next_state <= correct2;
    end case;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= warten;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: CARRY-SAVE                    --
--                                        --
-- bmult - 09.05.97                       --
--                                        --
--------------------------------------------

library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity bmult is
  port(
    clk,reset                         : in std_logic;
    full,ffull,bshift,bhigh,
    ballone,cbrdy,statea6             : in std_logic;
    s2                                : out std_logic_vector(1 downto
0);
    we,cbrst,cbgo,stateb3,stateb6,flagk: out std_logic
    );
end bmult;

architecture behaviour of bmult is

type state_type is
(add,suba,subs,shift,warten,ready1,ready2,correct1,correct2,correct3);
signal current_state, next_state: state_type;

begin

  we      <= '1' when (current_state=shift) else '0';
  flagk   <= '1' when ((current_state=ready2) or
(current_state=correct1) or
                      (current_state=correct2) or
(current_state=correct3)) else '0';
  cbgo    <= '1' when ((current_state=shift) or
(current_state=correct2)) else '0';
  cbrst   <= '1' when ((current_state=ready2) or
(current_state=correct1) or
                      (current_state=correct3)) else '0';
  stateb6 <= '1' when (current_state=ready1) else '0';
  stateb3 <= '1' when (current_state=ready2) else '0';

  FSM:
process(clk,full,ffull,bshift,bhigh,ballone,cbrdy,statea6,current_state)
  begin
    case current_state is
        when add =>       s2 <= "00";
                          if (cbrdy='1') then next_state <= ready1;
                          elsif (bhigh='1') then next_state <= add;
                          elsif (full='1') then next_state <= warten;
                          else next_state <= shift; end if;
        when suba =>      s2 <= "01";
                          next_state <= shift;
        when shift =>     s2 <= "10";
```

```
                         if (bhigh='1') then next_state <= subs;
                         elsif (bshift='1') then next_state <= add;
                         elsif (cbrdy='1') then next_state <= ready1;
                         elsif (ffull='1') then next_state <= warten;
                         else next_state <= shift; end if;
        when ready2 =>   s2 <= "11";
                         if (full='1') then next_state <= warten;
                         elsif (bshift='1') then next_state <= add;
                         else next_state <= shift; end if;
        when warten =>     s2 <= "11";
                         if (full='1') then next_state <= warten;
                         elsif (bshift='1') then next_state <= add;
                         elsif (cbrdy='1') then next_state <= ready1;
                         else next_state <= shift; end if;
        when subs =>     s2 <= "01";
                         if (full='1') then next_state <= warten;
                         elsif (bshift='1') then next_state <= add;
                         elsif (cbrdy='1') then next_state <= ready1;
                         else next_state <= shift; end if;
        when ready1 =>   s2 <= "11";
                         if (statea6='1') then next_state <= correct1;
                         else next_state <= ready1; end if;
        when correct1 => s2 <= "01";
                         if (ballone='1') then next_state <= correct2;
                         else next_state <= correct1; end if;
        when correct2 => s2 <= "00";
                         if (cbrdy='0') then next_state <= correct2;
                         elsif (ballone='1') then next_state <=
correct3;
                         else next_state <= ready2; end if;
        when correct3 => s2 <= "01";
                         next_state <= correct2;
    end case;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= warten;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: CARRY-SAVE                    --
--                                        --
-- counter - 09.05.97                     --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity counter is
  port(
    reset,clk: in std_logic;
    rst,go   : in std_logic;
    rdy      : out std_logic
    );
end counter;

architecture behaviour of counter is

signal c: integer range 0 to 1023;

begin

  rdy <= '1' when (c = 1023) else '0';

  SYNCH: process(clk, reset, rst, go)
  begin
    if (reset = '0') then
      c <= 0;
    elsif (clk'event and clk = '1') then
      if (rst = '1') then
        c <= 0;
      elsif (go = '1') then
        if (c = 1023) then
          c <= 0;
        else
          c <= c + 1;
        end if;
      end if;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: CARRY-SAVE                    --
--                                        --
-- exp - 09.05.97                         --
--                                        --
--------------------------------------------

library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity exp is
  port(
    clk,reset           : in std_logic;
    statea3,stateb3,
    statex3,y1,out_czero: in std_logic;
    in_ae,xsel          : out std_logic
    );
end exp;

architecture behaviour of exp is

type state_type is (init1,init2,cond,mult_ab,mult_xb,ausg1,ausg2);
signal current_state, next_state: state_type;
signal c: integer range 0 to 1023;
signal cgo, crst: std_logic;

begin

  cgo  <= '1' when ((current_state = init1) or (current_state = cond) or
(current_state = ausg1)) else '0';
  crst <= '1' when ((current_state = init2) or (current_state = ausg2))
else '0';
  in_ae <= '1' when ((current_state = init1) and (c = 1023)) else '0';
  xsel  <= '1' when (current_state = mult_xb) else '0';

  FSM: process(clk,current_state,c,y1,statea3,stateb3,statex3,out_czero)
  begin
    case current_state is
        when init1   => if (c = 1023) then next_state <= init2;
                        else next_state <= init1; end if;
        when init2   => next_state <= cond;
        when cond    => if (c = 1023) then next_state <= ausg1;
                        elsif (y1='1') then next_state <= mult_ab;
                        else next_state <= mult_xb; end if;
        when mult_ab => if ((statea3='0') and (stateb3='0')) then
next_state <= mult_ab;
                        else next_state <= cond; end if;
        when mult_xb => if ((statex3='0') and (stateb3='0')) then
next_state <= mult_xb;
                        else next_state <= cond; end if;
        when ausg1   => if ((c >= 132) and (out_czero = '0')) then
next_state <= ausg2;
                        -- ACHTUNG! Der Wert 132 ist nur bei 100MHz
korrekt!!!
```

```
                        else next_state <= ausg2; end if;
        when ausg2   => next_state <= init1;
        when others  => next_state <= init1;
    end case;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= init1;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

  COUNT: process(clk, reset, crst, cgo)
  begin
    if (reset = '0') then
      c <= 0;
    elsif (clk'event and clk = '1') then
      if (crst = '1') then
        c <= 0;
      elsif (cgo = '1') then
        if (c = 1023) then
          c <= 0;
        else
          c <= c + 1;
        end if;
      end if;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: CARRY-SAVE                    --
--                                        --
-- fcs.vhd - 20.05.97                     --
--                                        --
--------------------------------------------

library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity fcs is
  port(
    clk,reset   : in std_logic;
    x1,y1,k1,k2: in std_logic;
    e1         : out std_logic
    );
end fcs;

architecture behaviour of fcs is

component xmult
  port(
    clk,reset                  : in std_logic;
    empty,stateb3,stateb6      : in std_logic;
    s1                         : out std_logic_vector(1 downto 0);
    re,statex3,statex6         : out std_logic
    );
end component;

component amult
  port(
    clk,reset                  : in std_logic;
    empty,fempty,ashift,ahigh,
    aallone,cardy,stateb6      : in std_logic;
    s1                         : out std_logic_vector(1 downto 0);
    re,carst,cago,statea3,statea6: out std_logic
    );
end component;

component bmult
  port(
    clk,reset                  : in std_logic;
    full,ffull,bshift,bhigh,
    ballone,cbrdy,statea6      : in std_logic;
    s2                         : out std_logic_vector(1 downto 0);
    we,cbrst,cbgo,
    stateb3,stateb6,flagk      : out std_logic
    );
end component;

component feld
  port(
    clk                        : in std_logic;
    in_ae,in_be,in_k1,in_k2    : in std_logic;
```

```
    aselect,bselect              : in std_logic_vector(1 downto 0);
    flagk                        : in std_logic;
    aallone,ahigh,ballone,bhigh,
    out_ae,out_czero,out_bshift  : out std_logic
    );
end component;

component shiftbuf
  port(
    clk,reset                    : in std_logic;
    we,re,bshift                 : in std_logic;
    ashift,full,ffull,
    empty,fempty                 : out std_logic
    );
end component;

component exp
  port(
    clk,reset                    : in std_logic;
    statea3,stateb3,
    statex3,y1,out_czero         : in std_logic;
    in_ae,xsel                   : out std_logic
    );
end component;

component counter
  port(
    reset,clk                    : in std_logic;
    rst,go                       : in std_logic;
    rdy                          : out std_logic
    );
end component;

signal s1a,s1x,s2: std_logic_vector(1 downto 0);
signal ahigh,aallone,bhigh,ballone: std_logic;
signal flagk: std_logic;
signal in_ae,in_be: std_logic;
signal in_k1,in_k2: std_logic;
signal aselect,bselect: std_logic_vector(1 downto 0);
signal xsel: std_logic;
signal out_ae: std_logic;
signal out_czero: std_logic;
signal we,re,rea,rex,bshift,ashift: std_logic;
signal full,ffull,empty,fempty: std_logic;
signal statea3,stateb3,statex3: std_logic;
signal state6,statea6,stateb6,statex6: std_logic;
signal carst,cago,cardy: std_logic;
signal cbrst,cbgo,cbrdy: std_logic;

begin

my_xmult: xmult port
map(clk,reset,empty,stateb3,stateb6,s1x,rex,statex3,statex6);
my_amult: amult port
map(clk,reset,empty,fempty,ashift,ahigh,aallone,cardy,stateb6,s1a,rea,ca
rst,cago,statea3,statea6);
my_bmult: bmult port
map(clk,reset,full,ffull,bshift,bhigh,ballone,cbrdy,state6,s2,we,cbrst,c
bgo,stateb3,stateb6,flagk);
my_feld: feld port
map(clk,in_ae,in_be,in_k1,in_k2,aselect,bselect,flagk,
    aallone,ahigh,ballone,bhigh,out_ae,out_czero,bshift);
my_shiftbuf: shiftbuf port
map(clk,reset,we,re,bshift,ashift,full,ffull,empty,fempty);
```

```
my_exp: exp port
map(clk,reset,statea3,stateb3,statex3,y1,out_czero,in_ae,xsel);
my_acount: counter port map(reset,clk,carst,cago,cardy);
my_bcount: counter port map(reset,clk,cbrst,cbgo,cbrdy);

e1 <= out_ae;
in_be <= x1;
in_k1 <= k1;
in_k2 <= k2;
aselect <= s1x when (xsel='1') else s1a;
re <= rex when (xsel='1') else rea;
state6 <= statex6 when (xsel='1') else statea6;
bselect <= s2;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: CARRY-SAVE                    --
--                                        --
-- feld - 07.05.97                        --
--                                        --
--------------------------------------------

library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity feld is
  port(
    clk                 : in std_logic;
    in_ae,in_be,
    in_k1,in_k2         : in std_logic;
    aselect,bselect     : in std_logic_vector(1 downto 0);
    flagk               : in std_logic;
    aallone,ahigh,
    ballone,bhigh,
    out_ae,
    out_czero,out_bshift: out std_logic
    );
end feld;

architecture behaviour of feld is

component zelle
  port(
    clk                           : in std_logic;
    inae,inacarry,
    inbe,inbcarry,
    ink1,ink2,inczero,inbshift    : in std_logic;
    aselect,bselect               : in std_logic_vector(1 downto 0);
    flagk                         : in std_logic;
    outae,outacarry,
    outbe,outbcarry,
    outk1,outk2,outczero,outbshift: out std_logic
    );
end component;

component addof
  port(
    clk,inof,flagk   : in std_logic;
    sel              : in std_logic_vector(1 downto 0);
    allone,high: out std_logic
    );
end component;

signal ae,acarry,be,bcarry: std_logic_vector(1024 downto 0);
signal k1,k2,czero,bshift: std_logic_vector(1024 downto 0);
signal ainof,binof: std_logic;

begin
```

```
  GF: for i in 0 to 1023 generate
      my_zelle: zelle port map

(clk,ae(i),acarry(i),be(i),bcarry(i),k1(i),k2(i),czero(i),bshift(i),
         aselect,bselect,flagk,

ae(i+1),acarry(i+1),be(i+1),bcarry(i+1),k1(i+1),k2(i+1),czero(i+1),bshif
t(i+1));
  end generate;

  a_addof: addof port map(clk,ainof,flagk,aselect,aallone,ahigh);
  b_addof: addof port map(clk,binof,flagk,bselect,ballone,bhigh);

  ae(0)       <= in_ae;
  acarry(0)   <= '0';
  be(0)       <= in_be;
  bcarry(0)   <= '0';
  k1(0)       <= in_k1;
  k2(0)       <= in_k2;
  czero(0)    <= '0';
  bshift(0)   <= '0';

  out_ae      <= ae(1024);
  out_czero   <= czero(1024);
  out_bshift  <= bshift(1024);

  ainof       <= acarry(1024);
  binof       <= bcarry(1024);

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: CARRY-SAVE                    --
--                                        --
-- shiftbuf 09.05.97                      --
--                                        --
--------------------------------------------

library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity shiftbuf is
  port(
    clk,reset         : in std_logic;
    we,re,bshift      : in std_logic;
    ashift,full,ffull,
    empty,fempty      : out std_logic
    );
end shiftbuf;

architecture behaviour of shiftbuf is

signal puffer: std_logic_vector(15 downto 0);
signal idx: integer range 0 to 15;
signal asout: std_logic;

begin
  ashift <= asout;
  full   <= '1' when (idx=15) else '0';
  ffull  <= '1' when (idx=14) else '0';
  empty  <= '1' when (idx=0) else '0';
  fempty <= '1' when (idx=1) else '0';

  SB: process(clk, reset)
  begin
    if (reset = '0') then
      puffer <= "0000000000000000";
      idx <= 0;
      asout <= '0';
    elsif (clk'event and clk = '1') then
      if ((we='1') and (re='0')) then
        puffer(idx) <= bshift;
        idx <= idx + 1;
      elsif ((we='0') and (re='1')) then
        puffer <= ('0'&puffer(15 downto 1));
        asout <= puffer(0);
        idx <= idx - 1;
      elsif ((we='1') and (re='1')) then
        puffer <= ('0'&puffer(15 downto 1));
        puffer(idx) <= bshift;
        asout <= puffer(0);
      end if;
    end if;
  end process;
end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: CARRY-SAVE                    --
--                                        --
-- xmult - 09.05.97                       --
--                                        --
--------------------------------------------

library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity xmult is
  port(
    clk,reset            : in std_logic;
    empty,stateb3,stateb6: in std_logic;
    s1                   : out std_logic_vector(1 downto 0);
    re,statex3,statex6   : out std_logic
    );
end xmult;

architecture behaviour of xmult is

type state_type is (nix,ready1,ready2);
signal current_state, next_state: state_type;

begin

  re <= '1' when ((current_state=nix) and (empty='0')) else '0';
  statex3 <= '1' when (current_state=ready2) else '0';
  statex6 <= '1' when (current_state=ready1) else '0';

  FSM: process(clk,stateb3,stateb6,current_state)
  begin
    case current_state is
        when nix =>      s1 <= "10";
                         if (stateb6='0') then next_state <= nix;
                         else next_state <= ready1; end if;
        when ready1 =>   s1 <= "00";
                         next_state <= ready2;
        when ready2 =>   s1 <= "11";
                         if (stateb3='1') then next_state <= nix;
                         else next_state <= ready2; end if;
    end case;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= nix;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: CARRY-SAVE                    --
--                                        --
-- zelle 29.04.97                         --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity zelle is
  port(
    clk                      : in std_logic;
    inae,inacarry,
    inbe,inbcarry,
    ink1,ink2,inczero,inbshift: in std_logic;
    aselect,bselect          : in std_logic_vector(1 downto 0);
    flagk                    : in std_logic;
    outae,outacarry,
    outbe,outbcarry,
    outk1,outk2,
    outczero,outbshift       : out std_logic
    );
end zelle;

architecture behaviour of zelle is
signal zae,zacarry,zaold: std_logic;
signal zbe,zbcarry,zbold: std_logic;
signal zk1,zk2,zczero,zbshift: std_logic;
signal xa0,xb0,x1: std_logic;

begin

  outae     <= zae;
  outacarry <= zacarry;
  outbe     <= zbe;
  outbcarry <= zbcarry;
  outk1     <= zk1;
  outk2     <= zk2;
  outczero  <= zczero;
  outbshift <= zbshift;

  xa0       <= zaold when (flagk='1') else '0';
  xb0       <= zbold when (flagk='1') else '0';
  x1        <= zk1 when (flagk='1') else zk2;

  zczero    <= inczero OR zacarry OR zbcarry;

  A: process(clk,aselect,flagk)
  begin
    if (clk'event and clk='1') then
      case aselect is
        when "00" =>   zae <= zae XOR inacarry XOR xa0;
```

```
                       zacarry <= (zae AND inacarry) OR (xa0 AND
inacarry) OR (xa0 OR zae);
        when "01" =>   zae <= zae XOR inacarry XOR x1;
                       zacarry <= (zae AND inacarry) OR (x1 AND
inacarry) OR (x1 OR zae);
        when "10" =>   zae <= inae;
                       zacarry <= inacarry;
        when others => if (flagk='1') then
                         zacarry <= '0'; -- sollte schon NULL sein
                         zaold <= zae;
                         if (bselect="11") then
                           zae <= '0';
                         end if;
                       end if;
      end case;
    end if;
  end process;

  B: process(clk,bselect,flagk)
  begin
    if (clk'event and clk='1') then
      case bselect is
        when "00" =>   zbe <= zbe XOR inbcarry XOR xb0;
                       zbcarry <= (zbe AND inbcarry) OR (xb0 AND
inbcarry) OR (xb0 AND zbe);
        when "01" =>   zbe <= zbe XOR inbcarry XOR x1;
                       zbcarry <= (zbe AND inbcarry) OR (x1 AND
inbcarry) OR (x1 AND zbe);
        when "10" =>   zbe <= inbe;
                       zbcarry <= inbcarry;
                       zbshift <= inbshift;
                       if (flagk='1') then
                         zk1 <= ink1;
                         zk2 <= ink2;
                       end if;
        when others => if (flagk='1') then
                         zbcarry <= '0'; -- sollte schon NULL sein
                         zbold <= zbe;
                         zbshift <= zbe;
                         if (aselect="11") then
                           zbe <= '0';
                         end if;
                       end if;
      end case;
    end if;
  end process;

end behaviour;
```

SOSCARD

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: SOSCARD                       --
--                                        --
-- alu.vhd - 15.04.97                     --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity alu is
  port(
    reset, clk      : in std_logic;
    halt, aload     : in std_logic;
    a, b, c         : in std_logic_vector(15 downto 0);
    aclk, bclk, zclk: buffer std_logic;
    z               : out std_logic_vector(15 downto 0);
    ready, haelfte  : out std_logic
    );
end alu;

architecture behaviour of alu is

component mult16
  port(
    a,b: in std_logic_vector(15 downto 0);
    c  : in std_logic_vector(17 downto 0);
    z  : out std_logic_vector(32 downto 0)
    );
end component;

component mcount
  port(
    reset           : in std_logic;
    clk,halt        : in std_logic;
    aclk, bclk, zclk: buffer std_logic;
    ready, haelfte  : out std_logic
    );
end component;

component reg
  port(
    clk, reset, go: in std_logic;
    in16          : in std_logic_vector(15 downto 0);
    out16         : out std_logic_vector(15 downto 0)
    );
end component;

signal ago, cgo: std_logic;
signal ain16, aout16, cin16, cout16: std_logic_vector(15 downto 0);
signal ahelp, a16, b16, chelp: std_logic_vector(15 downto 0);
signal c18: std_logic_vector(17 downto 0);
```

```
signal z33: std_logic_vector(32 downto 0);
signal zhelp17: std_logic_vector(16 downto 0);

begin
  my_mult16: mult16 port map(a16,b16,c18,z33);
  my_mcount: mcount port
map(reset,clk,halt,aclk,bclk,zclk,ready,haelfte);
  a_reg: reg port map(clk,reset,ago,ain16,aout16);
  c_reg: reg port map(clk,reset,cgo,cin16,cout16);

  z <= z33(15 downto 0);
  ahelp <= a when ((aclk = '1') and (aload = '1')) else aout16;
  chelp <= c when (aclk = '1') else cout16;

  a16 <= ahelp;
  b16 <= b;
  c18 <= ("00"&chelp) + ("0"&zhelp17);

  ago <= (not halt);
  ain16 <= ahelp;
  cgo <= ((not zclk) and (not halt));
  cin16 <= z33(15 downto 0);

  ZHELP: process(reset,clk)
  begin
    if (reset = '0') then
      zhelp17 <= "00000000000000000";
    elsif (clk'event and clk='1') then
      zhelp17 <= z33(32 downto 16);
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: SOSCARD                       --
--                                        --
-- mcount.vhd - 15.04.97                  --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mcount is
  port(
    reset         : in std_logic;
    clk,halt      : in std_logic;
    aclk,bclk,zclk: buffer std_logic;
    ready,haelfte : out std_logic
    );
end mcount;

architecture behaviour of mcount is

signal count1, count2: integer range 0 to 63;

begin

  aclk <= '1' when (count2 = 0) else '0';
  bclk <= '1' when (count1 = 0) else '0';
  zclk <= '1' when ((count1 = 0) or (count2 = 63)) else '0';
  ready <= '1' when ((count1 = 63) and (count2 = 63)) else '0';
  haelfte <= '1' when (count2 >= 32) else '0';

  SYNCH: process(clk, reset, count1)
  begin
    if (reset = '0') then
      count1 <= 0;
      count2 <= 0;
    elsif (clk'event and clk = '1') then
      if (halt = '0') then
        if (count1 = 63) then
          count1 <= 0;
          if (count2 = 63) then
            count2 <= 0;
          else
            count2 <= count2 + 1;
          end if;
        else
          count1 <= count1 + 1;
        end if;
      end if;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: SOSCARD                       --
--                                        --
-- multmod.vhd - 19.04.97                 --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity modmult is
  port(
    reset, clk, halt      : in std_logic;
    mma, mmb, n1, negn    : in std_logic_vector(15 downto 0);
    mmz                   : out std_logic_vector(15 downto 0);
    clkmma, clkmmb, clkmmz: out std_logic;
    clkn1, clknegn, mmrdy : out std_logic
    );
end modmult;

architecture behaviour of modmult is

component alu
  port(
    reset, clk      : in std_logic;
    halt, aload     : in std_logic;
    a, b, c         : in std_logic_vector(15 downto 0);
    aclk, bclk, zclk: buffer std_logic;
    z               : out std_logic_vector(15 downto 0);
    ready, haelfte  : out std_logic
    );
end component;

component reg
  port(
    clk, reset, go: in std_logic;
    in16          : in std_logic_vector(15 downto 0);
    out16         : out std_logic_vector(15 downto 0)
    );
end component;

type state_type is (state1,state2,state3);
signal current_state, next_state: state_type;
signal a16, b16, c16, z16: std_logic_vector(15 downto 0);
signal xin16, xout16, qin16, qout16: std_logic_vector(15 downto 0);
signal aload, aclk, bclk, zclk, xgo, qgo, ready, haelfte: std_logic;

begin
  my_alu: alu port map(reset,clk,halt,aload,a16,b16,c16,
                       aclk,bclk,zclk,z16,ready,haelfte);
  x_reg: reg port map(clk,reset,xgo,xin16,xout16);
  q_reg: reg port map(clk,reset,qgo,qin16,qout16);

  aload <= '1' when (current_state = state1) else '0';
```

```
  xgo <= '1' when (((zclk = '1') and (current_state = state1) and
(haelfte = '1')) or
                   ((bclk = '1') and (current_state = state2)) or
                   ((aclk = '1') and (current_state = state3))) else
'0';
  xin16 <= z16 when (current_state = state1) else xout16;
  qgo <= '1' when (((zclk = '1') and (current_state = state1) and
(haelfte = '0')) or
                   ((zclk = '1') and (current_state = state2)) or
                   ((bclk = '1') and (current_state = state3))) else
'0';
  qin16 <= z16;
  mmz <= z16;
  clkmma <= '1' when ((aclk = '1') and (current_state = state1)) else
'0';
  clkmmb <= '1' when ((bclk = '1') and (current_state = state1)) else
'0';
  clkmmz <= '1' when ((zclk = '1') and (current_state = state3)) else
'0';
  clkn1 <= '1' when ((aclk = '1') and (current_state = state2)) else
'0';
  clknegn <= '1' when ((aclk = '1') and (current_state = state3)) else
'0';
  mmrdy <= '1' when ((ready = '1') and (current_state = state3)) else
'0';

  FSM: process(current_state,ready,mma,mmb,n1,xout16,negn,qout16)
  begin
    case current_state is
    when state1 => if (ready = '1') then next_state <= state2;
                   else next_state <= state1; end if;
                   a16 <= mma;
                   b16 <= mmb;
                   c16 <= "0000000000000000";
    when state2 => if (ready = '1') then next_state <= state3;
                   else next_state <= state2; end if;
                   a16 <= n1;
                   b16 <= xout16;
                   c16 <= "0000000000000000";
    when state3 => if (ready = '1') then next_state <= state1;
                   else next_state <= state3; end if;
                   a16 <= negn;
                   b16 <= qout16;
                   c16 <= xout16;
    end case;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= state1;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: SOSCARD                       --
--                                        --
-- mult16.vhd - 15.04.97                  --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mult16 is
  port(
    a,b : in std_logic_vector(15 downto 0);
    c   : in std_logic_vector(17 downto 0);
    z   : out std_logic_vector(32 downto 0)
    );
end mult16;

architecture behaviour of mult16 is

begin

  z <= ('0'&(a * b)) + ("000000000000000"&c);

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: SOSCARD                       --
--                                        --
-- reg.vhd - 19.04.97                     --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity reg is
  port(
    clk, reset, go: in std_logic;
    in16          : in std_logic_vector(15 downto 0);
    out16         : out std_logic_vector(15 downto 0)
    );
end reg;

architecture behaviour of reg is

signal x: std_logic_vector(1023 downto 0);
constant zero: std_logic_vector(1023 downto 0) := (others => '0');

begin

  out16 <= x(15 downto 0);

  SYNCH: process(reset,clk)
  begin
    if (reset='0') then
      x <= zero;
    elsif (clk'event and clk = '1') then
      if (go = '1') then
        x <= in16&x(1023 downto 16);
      end if;
    end if;
  end process;

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: SOSCARD                       --
--                                        --
-- rsa.vhd - 20.04.97                     --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity rsa is
  port(
    reset,clk,y1                : in std_logic;
    x,n1,negn                   : in std_logic_vector(15 downto 0);
    z                           : out std_logic_vector(15 downto 0);
    clkx,clky,clkn1,clknegn,clkz: out std_logic
    );
end rsa;

architecture behaviour of rsa is

component modmult
  port(
    reset, clk, halt      : in std_logic;
    mma, mmb, n1, negn    : in std_logic_vector(15 downto 0);
    mmz                   : out std_logic_vector(15 downto 0);
    clkmma, clkmmb, clkmmz: out std_logic;
    clkn1, clknegn, mmrdy : out std_logic
    );
end component;

component reg
  port(
    clk, reset, go: in std_logic;
    in16          : in std_logic_vector(15 downto 0);
    out16         : out std_logic_vector(15 downto 0)
    );
end component;

type state_type is (init,square,multiply,cond,ready);
signal current_state, next_state: state_type;
signal c: integer range 0 to 1023;
signal sc: integer range 0 to 1024;
signal mma,mmb,mmz: std_logic_vector(15 downto 0);
signal clkmma,clkmmb,clkmmz,mmrdy: std_logic;
signal halt,sgo,mgo: std_logic;
signal sin16,sout16,min16,mout16,s_puffer,eins16,mmbhlp:
std_logic_vector(15 downto 0);

begin
  my_modmult: modmult port
map(reset,clk,halt,mma,mmb,n1,negn,mmz,clkmma,
                               clkmmb,clkmmz,clkn1,clknegn,mmrdy);
  s_reg: reg port map(clk,reset,sgo,sin16,sout16);
```

```
  m_reg: reg port map(clk,reset,mgo,min16,mout16);

  mma <= x when (c = 0) else sout16;
  mmb <= mmbhlp when (current_state = multiply) else s_puffer;
  eins16 <= "0000000000000001" when (sc = 64) else "0000000000000000";
  mmbhlp <= eins16 when (c = 0) else mout16;
  sin16 <=  mmz;
  min16 <=  mmz;
  halt <= '1' when ((current_state = init) or (current_state = cond))
else '0';
  sgo <= '1' when (((current_state = multiply) and (clkmma = '1')) or
                  ((current_state = square) and ((clkmma = '1') or
(clkmmz = '1')))) else '0';
  mgo <= '1' when (((current_state = multiply) and (clkmmb = '1' or
clkmmz = '1')) or
                  (current_state = ready)) else '0';
  clkx <= '1' when ((c = 0) and (clkmma = '1') and (current_state =
multiply)) else '0';
  clky <= '1' when (current_state = cond) else '0';
  clkz <= '1' when (current_state = ready) else '0';
  z <= mout16;

  FSM: process(current_state, y1, mmrdy, c)
  begin
    case current_state is
    when init     => if (y1 = '1') then next_state <= multiply;
                     else next_state <= square; end if;
    when multiply  => if (mmrdy = '1') then next_state <= square;
                     else next_state <= multiply; end if;
    when square   => if (mmrdy = '1') then next_state <= cond;
                     else next_state <= square; end if;
    when cond     => if (c = 1023) then next_state <= ready;
                     elsif (y1 = '1') then next_state <= multiply;
                     else next_state <= square; end if;
    when ready    => if (c = 63) then next_state <= init;
                     else next_state <= ready; end if;
    end case;
  end process;

  COUNT: process(clk, reset, current_state, c)
  begin
    if (clk'event and clk = '1') then
      if (current_state = init) then
        c <= 0;
      elsif ((current_state = cond) OR (current_state = ready)) then
        c <= c + 1;
      end if;
    end if;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= init;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
  end process;

  SBUF: process(clk, reset, halt)
  begin
    if (reset = '0') then
      s_puffer <= "0000000000000000";
      sc <= 0;
```

```
    elsif (clk'event and clk = '1') then
      if (halt = '0') then
        if (sc = 1024) then
          s_puffer <= sout16;
          sc <= 0;
        else
          sc <= sc + 1;
        end if;
      end if;
    end if;
  end process;

end behaviour;
```

IWAMURA

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: IWAMURA                       --
--                                        --
-- array.vhd                              --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity arra is
  port(
    reset,clk      : in std_logic;
    sin,ain,bin,nin: in std_logic;
    tr             : out std_logic
    );
end arra;

architecture behaviour of arra is

component best
 port(
    reset,clk: in std_logic;
    sin,   ain,  bin,  tin,  min,  nin:   in std_logic;
    sout, aout, bout, tout, mout, nout: out std_logic
    );
end component;

type AR1 is array (0 to 9) of std_logic;
signal s,a,b,t,m,n: AR1;

begin
  GF: for i in 0 to 8 generate
      my_best: best port
map(reset,clk,s(i),a(i),b(i),t(i),m(i),n(i),s(i+1),a(i+1),b(i+1),t(i+1),
m(i+1),n(i+1));
  end generate;

  s(0) <= sin;
  a(0) <= ain;
  b(0) <= bin;
  t(0) <= '0';
  m(0) <= '0';
  n(0) <= nin;
  tr <= t(8);

end behaviour;
```

```
--------------------------------------------
--                                        --
--   Konzeption und Implementation eines  --
-- Hardwareakzelerators zum RSA-Verfahren --
--                                        --
--   Diplomarbeit - Lars H. Hahn - 1997   --
--                                        --
--------------------------------------------
--                                        --
-- Projekt: IWAMURA                       --
--                                        --
-- best.vhd                               --
--                                        --
--------------------------------------------

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity best is
  port(
    reset,clk: in std_logic;
    sin,  ain,  bin,  tin,  min,  nin:   in std_logic;
    sout, aout, bout, tout, mout, nout: out std_logic
    );
end best;

architecture behaviour of best is

signal reg_s1,reg_s2,reg_a,reg_ai,reg_m: std_logic;
signal reg_b1,reg_b2,reg_n1,reg_n2: std_logic;
signal reg_t: std_logic_vector(2 downto 0);
signal mult1,mult2: std_logic;
signal add: std_logic_vector(2 downto 0);

begin

  sout  <= reg_s2;
  aout  <= reg_ai;
  bout  <= reg_b2;
  tout  <= reg_t(0);
  mout  <= reg_m;
  nout  <= reg_n2;

  mult1 <= bin AND reg_a;
  mult2 <= min AND nin;
  add   <= ('0'&reg_t(2 downto 1)) + ("00"&tin) + ("00"&mult1) +
("00"&mult2);

  SYNCH: process(reset,clk)
  begin
    if (reset = '0') then
      reg_s1 <= '0';
      reg_s2 <= '0';
      reg_a  <= '0';
      reg_ai <= '0';
      reg_b1 <= '0';
      reg_b2 <= '0';
      reg_n1 <= '0';
      reg_n2 <= '0';
      reg_m  <= '0';
      reg_t  <= "000";
    elsif (clk'event and clk = '1') then
```

```
      reg_s1 <= sin;
      reg_s2 <= reg_s1;
      reg_ai <= ain;
      reg_b1 <= bin;
      reg_b2 <= reg_b1;
      reg_n1 <= nin;
      reg_n2 <= reg_n1;
      reg_t  <= add;
      if (sin = '1') then
        reg_a <= ain;
      end if;
      if (reg_s1 = '1') then
        reg_m <= add(0);
      end if;
    end if;
  end process;

end behaviour;
```

Erklärung

Hiermit erkläre ich an Eides statt, die vorliegende Arbeit selbständig durchgeführt zu haben und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Hamburg, den 28. Juli 1997

Lars Harald Hahn