Hardware-Implementation eines Chiffrierverfahrens

Ingo Rohloff

Interdisziplinäres Projekt

Lehrstuhl für Entwurfsautomatisierung
Technische Universität München
Prof. Dr.-Ing. K. Antreich

Ich versichere, daß ich dieses Interdisziplinäre Projekt selbständig verfaßt, und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ingo Rohloff
Gotthelfstr. 101
81677 München

München, 27.02.1999

Inhaltsverzeichnis

1 Einleitung

Das Internet entwickelt sich seit einiger Zeit immer mehr zu einem Massenkommunikationsmittel, das nicht nur mehr von einer kleinen an Computern interessierten Minderheit benutzt wird, sondern auch im alltäglichen Leben immer mehr an Bedeutung gewinnt. Diese Zunahme an Benutzern führt zu neuen Problemen, mit denen bei der Entstehung des Internets niemand gerechnet hatte. Eines davon ist die Frage nach der Datensicherheit und dem Schutz der Privatsphäre, besonders in elektronischer Post (E-Mail), aber auch bei sämtlichen anderen Arten der Datenübertragung.

Als das Internet entstand, interessierte sich zunächst niemand für Datensicherheit, denn der Kreis der Benutzer war sehr begrenzt. Das führte dazu, daß die verschiedenen Kommunikationsprotokolle völlig ohne Abhörschutz entworfen wurden. Als das Internet anfing zu wachsen, verwendete man diese alten Protokolle weiter und aus diesem Grund existiert auch heute noch kein automatischer Abhörschutz und jeder, der das Know-How dazu hat, kann Datenübertragungen anzapfen.

Eine Lösung für dieses Problem ist es private Datenübertragungen zu verschlüsseln. Diese Dokumentation beschäftigt sich mit der Implementierung eines bestimmten Verschlüsselungsalgorithmus als Hardware, dem sogenannten IDEA (International Data Encryption Algorithm).

2 Verschlüsselung im Allgemeinen

Der Sinn einer Verschlüsselung ist es eine Botschaft so zu verändern, daß nur noch ein ausgewählter Kreis von Personen im Stande ist sie zu entziffern. Eine Botschaft auf einem Computer ist einfach ein Strom von Bits. Eine Verschlüsselung auf einem Computer verändert diesen Strom von Bits scheinbar willkürlich und nur jemand der weiß, auf welche Art und Weise die Bits verändert wurden, kann die ursprüngliche Botschaft rekonstruieren, zumindest wenn die Verschlüsselung „gut“ ist.

Moderne Verschlüsselungen benutzen immer einen festen, bekannten Algorithmus zum verändern der Orginalnachricht (dem Klartext). Es gibt lediglich einen Parameter (nämlich den Schlüssel), der verändert wird. Ein Schlüssel ist bei solchen Algorithmen ebenfalls ein Bitmuster mit einer bestimmten Länge.

2.1 Block- und Stromchiffrierung

Man kann die Botschaft für die Verschlüsselung in Blöcke mit einer bestimmten Länge aufteilen und dann die einzelnen Blöcke verschlüsseln. In diesem Fall spricht man von einer Blockchiffrierung; bearbeitet man dagegen die Bits oder Bytes einzeln, spricht man von einer Stromchiffrierung. Zur Zeit werden Blockchiffrierungen bevorzugt.

Der Unterschied verschwimmt, wenn man bedenkt, daß die Bearbeitung von einzelnen Bytes im Prinzip die Bearbeitung eines Blocks von Bits darstellt.

Außerdem werden bei der Benutzung von Blockchiffrierungen die einzelnen Blöcke oft miteinander verknüpft, so daß sich außer dem Anfangsblock kein Block einzeln entschlüsseln läßt. Diese Eigenschaft entspricht eher einer Stromchiffrierung.

2.2 Symmetrische und asymmetrische Verschlüsselung

Man spricht von einer symmetrischen Verschlüsselung, wenn man den gleichen Schlüssel für die Ver- und Entschlüsselung benutzt. Ist dieser Schlüssel bekannt, so kann man genau so schnell Entschlüsseln wie Verschlüsseln. Damit eine Verschlüsselte Botschaft geheim bleibt, muß also vor allem der Schlüssel der benutzt wurde geheim bleiben. Die Schlüsselänge ist deshalb ein wichtiges Kriterium für die Sicherheit einer Verschlüsselung. Ist der Schlüssel zu kurz, dann kann man einfach alle Schlüssel durchprobieren, um eine verschlüsselte Botschaft zu entziffern. Schlüsselängen mit weniger als 64 Bit gelten heutzutage nicht mehr als wirklich sicher, obwohl es bei dieser Länge ungefähr $1,84 * 10^{19}$ verschiedene Schlüssel gibt.

Bei der asymmetrischen Verschlüsselung gibt es zwei Schlüssel: Einen zum Verschlüsseln und einen zum Entschlüsseln. Das Besondere dabei ist, daß man bisher noch keine systematische Methode kennt, den einen Schlüssel aus dem anderen zu berechnen, außer durch ausprobieren.

Asymmetrische Verfahren lösen ein Problem der symmetrischen Verfahren nämlich die Weitergabe des Schlüssels. Bei einem symmetrischen Verfahren muß man den geheimen Schlüssel so weitergeben, daß er auch geheim bleibt. Das ist ohne persönlich den Schlüssel auf einer Diskette zu übergeben kaum sicher möglich. Hinzu kommt, daß man nie weiß ob die anderen „Geheimnisträger“ den Schlüssel auch sorgfältig aufbewahren.

Im Gegensatz dazu wird bei einem asymmetrischen Verfahren einer der beiden Schlüssel öffentlich bekannt gegeben. Jeder kann dann damit eine Nachricht verschlüsseln, aber nur derjenige, der den zweiten Schlüssel des Schlüsselpaares hat, ist imstande sie zu entziffern.

Asymmetrische Verschlüsselungen werden nicht nur zum Sichern von Daten benutzt, sondern mit ihnen werden auch elektronische Unterschriften erzeugt und sie werden zur Identifizierung von Rechnern und Personen benutzt. Elektronisches Geld kann ebenfalls mit bestimmten asymmetrischen Verfahren realisiert werden.

Der Nachteil ist, daß asymmetrische Verschlüsselungen im Vergleich zu Symmetrischen sehr zeitaufwendig sind. Für größere Datenmengen werden deshalb normalerweise die beiden Verfahren gekoppelt und darum ist es wichtig sichere und schnelle symmetrische Verfahren zu haben.

Sehr viel mehr zum Thema Kryptographie findet man in [1].

3 Beschreibung des IDEA

Der IDEA gehört zu der Klasse der Blockchiffrierungen. Er verschlüsselt 64 Bit Blöcke und verwendet dazu einen 128 Bit langen geheimen Schlüssel. Er basiert auf dem Design-Konzept, Operationen aus verschiedenen algebraischen Gruppen zu mischen (siehe [2]). Soweit man weiß, ist der IDEA ein sehr sicherer Algorithmus, d.h. es gibt keine bessere Methode ihn zu brechen als systematisches Ausprobieren, was durch seine Schlüssellänge von 128 Bit ebenfalls nicht sehr erfolgversprechend ist. Der IDEA ist vor allem durch das Programm PGP (Pretty Good Privacy) populär geworden, das ihn zur Verschlüsselung von E-Mail und anderen Daten einsetzt.

3.1 Die Verschlüsselung

Die Verschlüsselung besteht aus acht „Runden“ und einer Ausgabetransformation (vgl. Abbildung 1), die hier als neunte Runde bezeichnet wird. Eine Runde ist ein festes Rechenschema mit einer 64 Bit Ein- und Ausgabe. Die Ausgabe einer Runde wird als Eingabe für die nächste Runde verwendet bis alle neun Runden durchlaufen sind.

In einer Runde werden drei verschiedene Operationen benutzt:

- Bit-für-Bit XOR von zwei 16 Bit Zahlen.
- Die Addition von zwei vorzeichenlosen 16 Bit Zahlen. (Überträge werden ignoriert.)
- Die Multiplikation von zwei vorzeichenlosen 16 Bit Zahlen modulo $2^{16} + 1$, bei der die 0 auf 2^{16} abgebildet wird. (z.B. $0 \odot 0 = 2^{16} * 2^{16} mod(2^{16} + 1) = 1$)

In einer Runde werden die 64 Eingabebits in vier 16 Bit Zahlen ($X1$ - $X4$) aufgeteilt und mit Hilfe von sechs 16 Bit Teilschlüsseln ($Z1^{(r)}$ - $Z6^{(r)}$, das r steht für die Rundennummer), die aus dem 128 Bit Schlüssel gewonnen werden, verschlüsselt. Die Ausgabe besteht ebenfalls aus vier 16 Bit Zahlen ($Y1$ - $Y4$), die als Eingabe für die nächste Runde benutzt werden. Insgesamt benötigt man 52 Teilschlüssel ($6 \cdot 8 = 48$ für die acht Runden und noch einmal vier für die Ausgabetransformation).

Die 52 Teilschlüssel werden aus dem 128-Bit langen Schlüssel folgendermaßen erzeugt: Die 128 Bit werden in acht 16 Bit Zahlen aufgeteilt, die dann die ersten acht Teilschlüssel darstellen. (D.h. die ersten sechs davon werden für die erste Runde, die übrigen zwei bereits für die nächste Runde benutzt.) Die 128 Bit werden um 25 Bit nach links rotiert und danach wieder in acht 16 Bit Zahlen zerlegt, wodurch man die nächsten acht Teilschlüssel erhält. Mit diesem Schema wird fortgefahren, bis man alle 52 Teilschlüssel erzeugt hat (vgl. Abbildung 2).

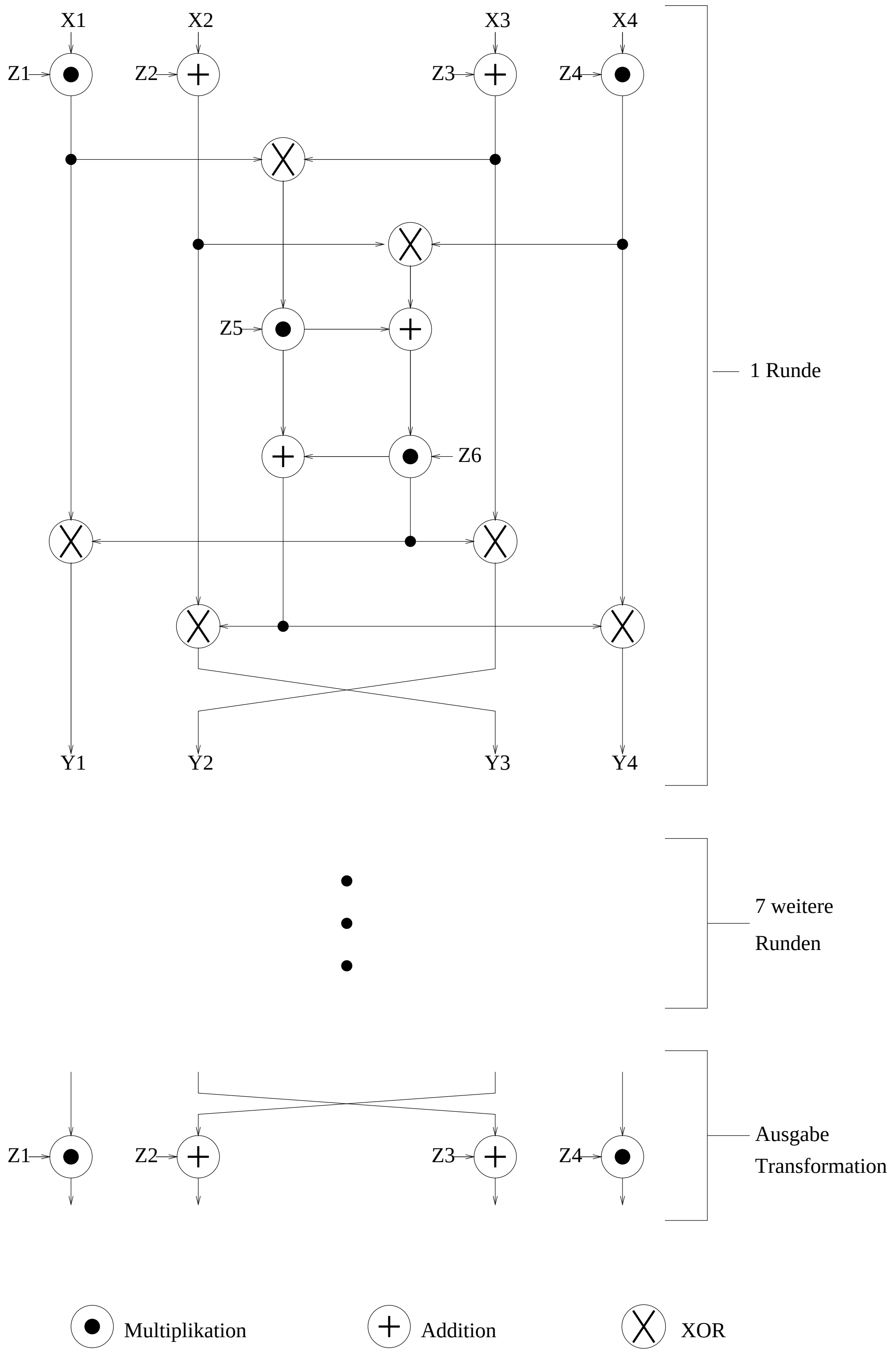


Abbildung 1: Der IDEA als Graph

0001 0002 0003 0004 0005 0006 0007 0008	128-Bit Schlüssel (Hexa-Dezimal)
0400 0600 0800 0A00 0C00 0E00 1000 0200	25 Bits nach links
0010 0014 0018 001C 0020 0004 0008 000C	25 Bits nach links
2800 3000 3800 4000 0800 1000 1800 2000	25 Bits nach links
0070 0080 0010 0020 0030 0040 0050 0060	25 Bits nach links
0000 2000 4000 6000 8000 A000 C000 E001	25 Bits nach links
0080 00C0 0100 0140	nur noch 4 Teilschlüssel

Die erste Zeile stellt die 128-Bit des Schlüssels aufgegliedert in acht Teilschlüssel in hexadezimal Schreibweise dar. In jeder weiteren Zeile wird der Schlüssel um weitere 25 Bit nach links rotiert und das Ergebnis wieder in acht neue Teilschlüssel aufgeteilt. In der letzten Zeile fehlen nur noch vier Teilschlüssel, da bereits 48 Teilschlüssel festgelegt sind.

Abbildung 2: Erzeugung von 52 Teilschlüsseln

3.2 Die Entschlüsselung

Zum entschlüsseln wird der gleiche Algorithmus verwendet, allerdings mit anderen Teilschlüsseln ($K1^{(r)}$ - $K6^{(r)}$), die sich aus den Teilschlüsseln zum Verschlüsseln berechnen lassen:
für r=2,...,8 :

$$K1^{(r)}, K2^{(r)}, K3^{(r)}, K4^{(r)} = Z1^{(10-r)^{-1}}, -Z3^{(10-r)}, -Z2^{(10-r)}, Z4^{(10-r)^{-1}}$$

für r=1,9 :

$$K1^{(r)}, K2^{(r)}, K3^{(r)}, K4^{(r)} = Z1^{(10-r)^{-1}}, -Z2^{(10-r)}, -Z3^{(10-r)}, Z4^{(10-r)^{-1}}$$

für r=1,...,8 :

$$K5^{(r)}, K6^{(r)} = Z5^{(9-r)}, Z6^{(9-r)}$$

Z^{-1} bezeichnet das multiplikative Inverse modulo $2^{16}+1$ von Z, d.h. $Z \odot Z^{-1} = 1$
$-Z$ bezeichnet das additive Inverse modulo 2^{16} von Z, d.h. $-Z \oplus Z = 0$

3.3 Die Berechnung von $ab \bmod (2^{16}+1)$

Die „schwierigste" Operation des IDEA ist die Multiplikation modulo $2^{16}+1$. Diese Operation kann vereinfacht werden, wenn man folgende Gleichung benutzt:

$$ab \bmod (2^n+1) = \begin{cases} (ab \bmod 2^n) - (ab \operatorname{div} 2^n) & \text{wenn } (ab \bmod 2^n) \geq (ab \operatorname{div} 2^n) \\ (ab \bmod 2^n) - (ab \operatorname{div} 2^n) + 2^n + 1 & \text{wenn } (ab \bmod 2^n) < (ab \operatorname{div} 2^n) \end{cases}$$

Wenn man diese Gleichung auf 16 Bit anwendet, wird es noch einfacher: $ab \bmod 2^{16}$ entsprechen den 16 niederwertigen Bits und $ab \operatorname{div} 2^{16}$ den 16 höherwertigen Bits von ab. Der Ausdruck $(2^{16} + 1)$ vereinfacht sich zu 1, weil man sowieso nur mit 16 Bit Breite rechnet.

Man muß nur noch den Sonderfall beachten, daß a oder b gleich Null sind und deshalb als 2^{16} interpretiert werden. In diesem Fall gilt $(ab \bmod 2^{16}) = 0 < (ab \operatorname{div} 2^{16})$. Das Ergebnis ist dann $1 - (ab \operatorname{div} 2^{16})$ (zweiter Fall), was wiederum bedeutet daß man die Zahl, die ungleich von Null ist, von Eins abziehen muß. Sind beide Zahlen, a und b, gleich Null, ist das Ergebnis Eins.

4 Hardwarebeschreibungssprachen

Seit ein paar Jahren werden Schaltungen immer häufiger in sogenannten Hardwarebeschreibungssprachen (HDLs[1]) beschrieben und entworfen. Inzwischen ist die Technik weit genug fortgeschritten, um aus der Beschreibung in einer HDL einen Schaltungsentwurf zu generieren, so daß man oft auf den Entwurf am Reißbrett verzichten kann.

Diese Sprachen unterstützen die gleichen Konzepte wie andere imperative Sprachen (z.B. C oder Pascal); zu den Konzepten gehören die bedingte Ausführung von Programmteilen, Schleifen und die Aufteilung des Programmes in einzelne Module. Zusätzlich bieten diese Sprachen Elemente, um Nebenläufigkeiten und Signale beschreiben zu können und tragen damit ihrem Zweck Rechnung, Hardware modellieren zu können.

Der Entwurf einer Schaltung in einer HDL hat gegenüber dem Entwurf auf dem Reißbrett Vorteile:

- Man befindet sich auf einer höheren Abstraktionsstufe und das führt erfahrungsgemäß schneller zu funktionsfähigen Entwürfen.
- Viele Techniken des Softwareengineerings können auch auf Entwürfe in einer HDL angewendet werden.
- Der Entwurf in einer HDL kann als Ausgangsbasis für verschiedene Zielarchitekturen verwendet werden, d.h. er ist portabel.

4.1 Die Entwicklungsumgebung

Für dieses Projekt wurde die Hardwarebeschreibungssprache VHDL benutzt. VHDL[2] ist zur Zeit eine der am häufigsten benutzten Hardwarebeschreibungssprachen. Das hängt nicht zuletzt damit zusammen, daß sie von dem IEEE[3] als Norm verabschiedet wurde und

[1] Hardware Description Language
[2] VHSIC Hardware Description Language; VHSIC = Very High Speed Integrated Circuit
[3] Institute of Electrical and Electronics Engineers

damit eine einheitliche Referenz existiert. Der Standard wurde inzwischen überarbeitet und deshalb unterscheidet man zwischen VHDL-87 (aus dem Jahre 1987) und VHDL-93 (von 1993) um anzugeben, welchen Standard man benutzt. Mittlerweile hat das ANSI[4] VHDL ebenfalls als Standard definiert. In diesem Projekt wurde der VHDL-87 Standard benutzt.

VHDL bietet die gängigen Konzepte von modernen Programmiersprachen und unterstützt zusätzlich die Modellierung von nebenläufigen Prozessen und Signalen. Mehr über VHDL kann man in [3] nachlesen.

Als Entwicklungsumgebung wurden die Softwarewerkzeuge der Firma Synopsys benutzt. Die Umgebung enthält unter anderem einen Simulator zum Testen des VHDL Entwurfes und Werkzeuge zur automatischen Synthese der Schaltung aus der VHDL-Beschreibung.

Als Zielarchitektur wurden FPGAs[5] vom Modeltyp 4028 der Firma Xilinx gewählt. Ebenso wurden Softwarewerkzeuge von Xilinx benutzt, um die Platzierung der Schaltung auf den FPGAs zu realisieren.

5 Die Implementation

Die Implementation des IDEA in VHDL geschah schrittweise. Zuerst wurde eine möglichst einfache und leicht zu verifizierende Version implementiert, ohne auf Hardwarebeschränkungen zu achten.

Mit Hilfe dieser Referenzimplementation lassen sich Testdaten erzeugen, um kompliziertere Implementationen zu überprüfen.

Jede Implementation benötigt die drei Rechenoperationen des IDEA, deshalb begann die Programmierarbeit mit der Realisierung dieser Operationen als eigenständige Submodule.

5.1 Die Rechenmodule

Jedes Rechenmodul besitzt zwei 16 Bit Eingänge und einen 16 Bit Ausgang. Die innere Logik läßt sich in VHDL einfach ausdrücken, weil die vier Grundrechenarten und logische Operationen (wie z.B. XOR) in der Sprache direkt unterstützt werden. Das Interface der Schaltung wird mit Hilfe einer *Entity* festgelegt. Die Rechenmodule sind ungetaktet, d.h. sobald die Eingangsdaten anliegen oder sich verändern, kann man am Ausgang das Ergebnis nach einer Modulabhängigen Verzögerungszeit abgreifen.

[4] American National Standards Institute
[5] Field Programmable Gate Arrays

5.2 Der Schlüsselgenerator

Die Teilschlüssel werden durch einen Schlüsselgenerator aus dem 128 Bit Schlüssel extrahiert. Der Schlüsselgenerator besitzt einen 4 Bit Eingang, mit dem die Runde ausgewählt wird. Die sechs 16 Bit Ausgänge werden in Abhängigkeit von der Rundennummer mit den richtigen Teilbits des Schlüssels beschaltet.

Dieser Schlüsselgenerator generiert nur die Teilschlüssel zur Verschlüsselung. Die Schaltung zum Erzeugen der Teilschlüssel zum Entschlüsseln wäre leider sehr kompliziert, da man zum Berechnen des multiplikativen Inversen den Euklidischen Algorithmus in Hardware implementieren müßte. Eine bessere Lösung wäre es, statt einem Schlüsselgenerator ein RAM zu benutzen, das mit 4 Bit adressiert wird und eine Busbreite von 96 Bit hat (16 Bit für 6 Teilschlüssel). In dieses RAM würde man vor Beginn der Verschlüsselung die Teilschlüssel laden. Die Teilschlüssel müßten vorher extern berechnet werden, z.B. mit Hilfe einer Implementation in C.

Der Schlüsselgenerator in dieser Implementation wurde vor allem programmiert, um das Verschlüsselungsmodul auch ohne RAM testen zu können.

5.3 Die Referenzimplementation

Die einfachste Möglichkeit, den IDEA in VHDL zu beschreiben, ist die strukturierte Programmierung: aus den Rechenmodulen wird einfach eine Schaltung zusammengesetzt, die genau dem Graphen des gesamten IDEA entspricht. Für jede Runde wird ein Schlüsselgenerator benutzt, um die Teilschlüssel für diese Runde aus dem 128 Bit Schlüssel zu extrahieren. Die jeweiligen Eingänge der Schlüsselgeneratoren für die Rundennummer werden dabei mit Konstanten beschaltet.

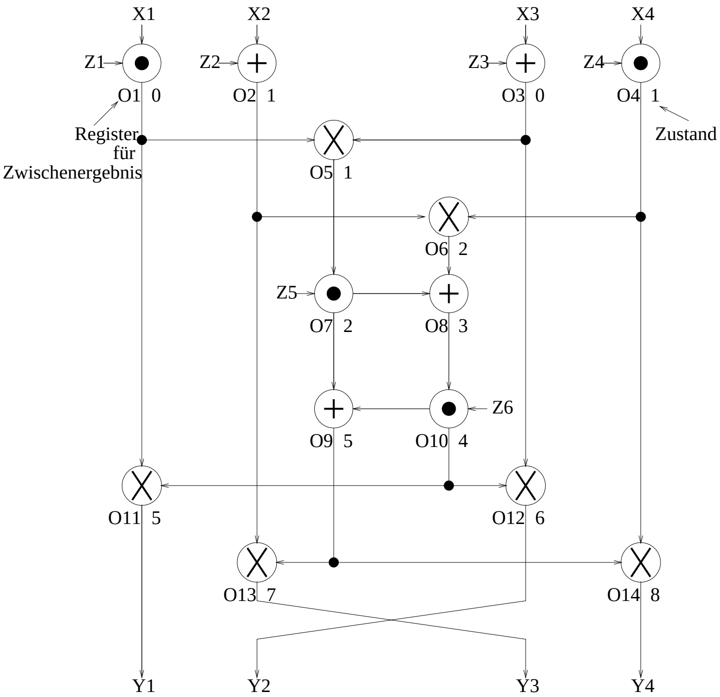
Eine solche Schaltung ist für die gewählte Zielarchitektur nicht geeignet, weil die Schaltung zu groß wäre. Es würden allein 34 Multiplikationsmodule benötigt: vier für jede Runde und noch einmal zwei für die Ausgabetransformation; hinzu kämen noch die gleiche Zahl an Addieren und noch mehr XOR-Module.

Der Vorteil ist, daß durch diesen stark strukturierten Aufbau der Schaltung weniger Implementationsfehler in VHDL gemacht werden. Man erhält deshalb in kurzer Zeit einen simulierbaren Entwurf in VHDL, der die richtigen Ergebnisse liefert und mit dem man sich in die Simulationsumgebung von Synopsys einarbeiten kann.

5.4 Die erste getaktete Runde

Aus der Tatsache, daß ein Multiplizierer in Hardware immer ein vergleichsweise großes Modul ist, ergibt sich die Notwendigkeit möglichst wenige Multiplizierer, am besten nur einen, zu verwenden.

Um dieses Ziel zu erreichen, muß die Runde getaktet arbeiten. In jedem Takt wird ein Zwi-



Zustands-nummer	Berechnung
0	$O1 = X1 \odot Z1$ $O3 = X3 \oplus Z3$
1	$O2 = X2 \oplus Z2$ $O4 = X3 \odot Z4$ $O5 = O1 \otimes O3$
2	$O6 = O2 \otimes O4$ $O7 = O5 \odot Z5$
3	$O8 = O6 \oplus O7$
4	$O10 = O8 \odot Z6$
5	$O9 = O10 \oplus O7$ $O11 = O10 \otimes O1$
6	$O12 = O10 \otimes O3$
7	$O13 = O9 \otimes O2$
8	$O14 = O9 \otimes O4$

Abbildung 3: Die erste Version der getakteten Runde

schenergebnis berechnet und in einem Register abgelegt. Man muß die Runde schrittweise berechnen, wobei in jedem Schritt nur maximal eine Multiplikation berechnet werden darf.

Die erste Implementation des getakteten Runden-Moduls benutzt neben der Clock zwei zusätzliche Steuersignale: Ein Init-Signal, das in dem Runden-Modul die Berechnung in den Anfangszustand versetzt und ein Result-Signal, das nach Außen das Ende der Rundenberechnung mitteilt. Das Init-Signal entspricht einem synchronen Reset, d.h. der Reset wird erst bei der nächsten steigenden Flanke der Clock ausgeführt.

Das Ziel dieser Implementation war, nur eine einzige Instanz jedes Rechenmoduls zu benutzen. In jedem Zustand werden nur unabhängige Operationen berechnet. In VHDL wurde die Logik und der Zustandszähler in einem einzigen *Process* implementiert. Prozesse dienen in VHDL zur Beschreibung von Nebenläufigkeiten, denn Prozesse werden im Simulator als zu einander parallel ablaufende Programmstücke simuliert. Die Anweisungen innerhalb eines *Process* werden sequentiell abgearbeitet, d.h. es gibt in dieser Implementation keine Nebenläufigkeiten, da es nur einen *Process* gibt.

Es gibt 10 Zustände, in denen die jeweils berechneten Zwischenergebnisse in verschiedene Register abgelegt und neue Berechnungen angestoßen werden. Im neunten Zustand wird das Ergebnis an die Ausgabeports angelegt und das Result-Signal auf „1“ gesetzt. Der 10te Zustand ist ein Haltezustand, in dem sich nichts mehr ändert. Abbildung 3 verdeutlicht den Ablauf der Berechnung in dieser Implementation.

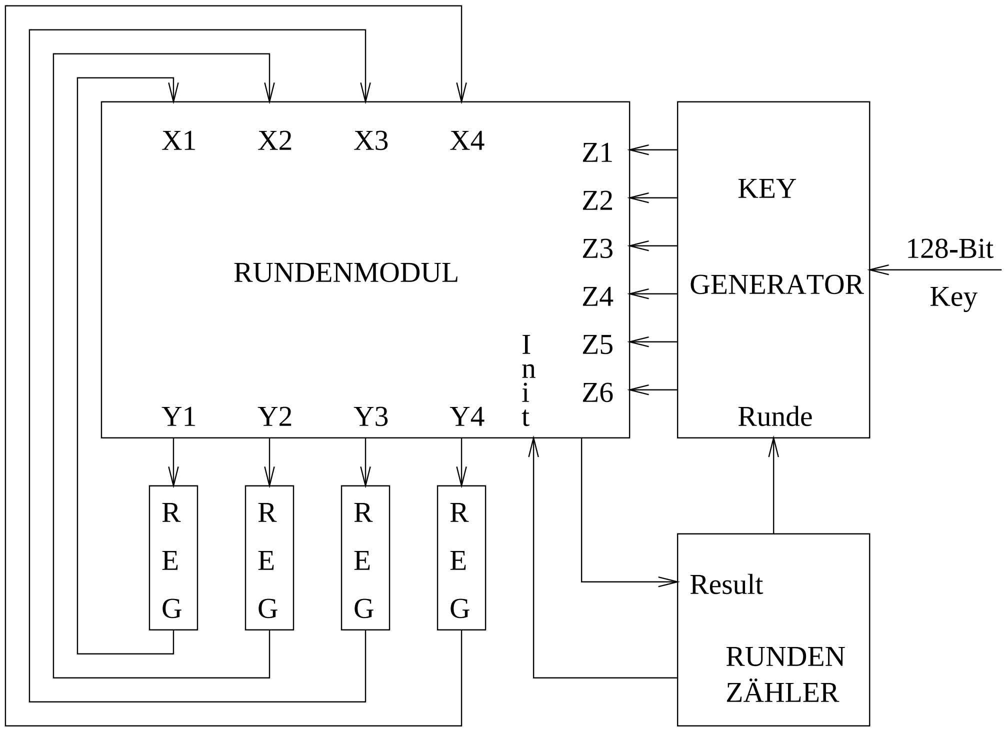


Abbildung 4: Schema der gesamten Schaltung

Der Wert der Register wird während eines Taktes berechnet. Das Ergebnis wird erst beim Übergang in den nächsten Takt in die Register gespeichert. (Zum Beispiel erhalten $O1$ und $O3$ ihren Wert erst dann, wenn von den 0ten in den 1ten Zustand umgeschaltet wird.)

5.5 Die äußere Logik

Wenn man nur ein Rundenmodul verwendet, braucht man eine äußere Logik, die das Ergebnis des Moduls zwischenspeichert und wieder an seinen Eingang legt, bis acht Runden durchlaufen sind.

Nach Außen besitzt diese Logik einen 128-Bit Eingang, an dem der Schlüssel anliegt, vier 16-Bit Eingänge für die zu verschlüsselnden Daten, vier 16 Bit Ausgänge für das Ergebnis, einen Eingang für den Takt und einen Steuereingang zum Initialisieren der Berechnung.

Im Inneren befinden sich vier Register, die als Zwischenspeicher für die Rundenergebnisse verwendet werden und ein Rundenzähler. Die Logik bindet die getaktete Runde und den Schlüsselgenerator als Submodule ein (vgl. Abbildung 4).

Die Runde wird durch eine „1“ am Init-Signal gestartet. Die äußere Logik wartet danach, bis das Result-Signal der Runde das Ende der Rundenberechnung signalisiert. Der Rundenzähler der äußeren Logik wird erhöht und die Runde erhält erneut ein Init-Signal und rechnet mit den neuen Zwischenergebnissen weiter. Sobald die achte Runde beendet ist, wird das Ergebnis nach Außen weitergegeben.

Die Äußere Logik und das Rundenmodul befinden sich so immer in einem Wechselspiel, daß mit den Taktflanken synchronisiert ist: Erst wartet das Rundenmodul auf eine „1“ der äußeren Logik, dann wartet die Logik auf eine „1“ vom Rundenmodul.

Diese Schaltung berechnet noch keine Ausgabetransformation, da es zunächst darum ging die Implementierung des Rundenmoduls zu testen und zu verbessern.

5.6 Die zweite optimierte Version

Das Modul das bei weitem am meisten Ressourcen verbraucht ist der Multiplizierer. Im Gegensatz dazu läßt sich ein Exklusiv-Oder Modul mit wenigen Gattern aufbauen. Aus diesem Grund ist es sinnvoll, mehr als nur ein einziges Exklusiv-Oder Modul zu verwenden, um die Anzahl der Zustände zu verringern und damit die Geschwindigkeit der Berechnung zu erhöhen. Außerdem ist es möglich, mehrere Module zu einer einzigen großen Berechnung zusammen zu schließen, was die Anzahl der benötigten Zwischenregister senkt.

Führt man diese Optimierungen durch, erhält man eine simulierbare Runde, die mit nur fünf Zuständen auskommt und damit nur halb so viele Zustände benötigt wie die vorhergehende Version. In Abbildung 5 wird der Ablauf der Berechnung erklärt.

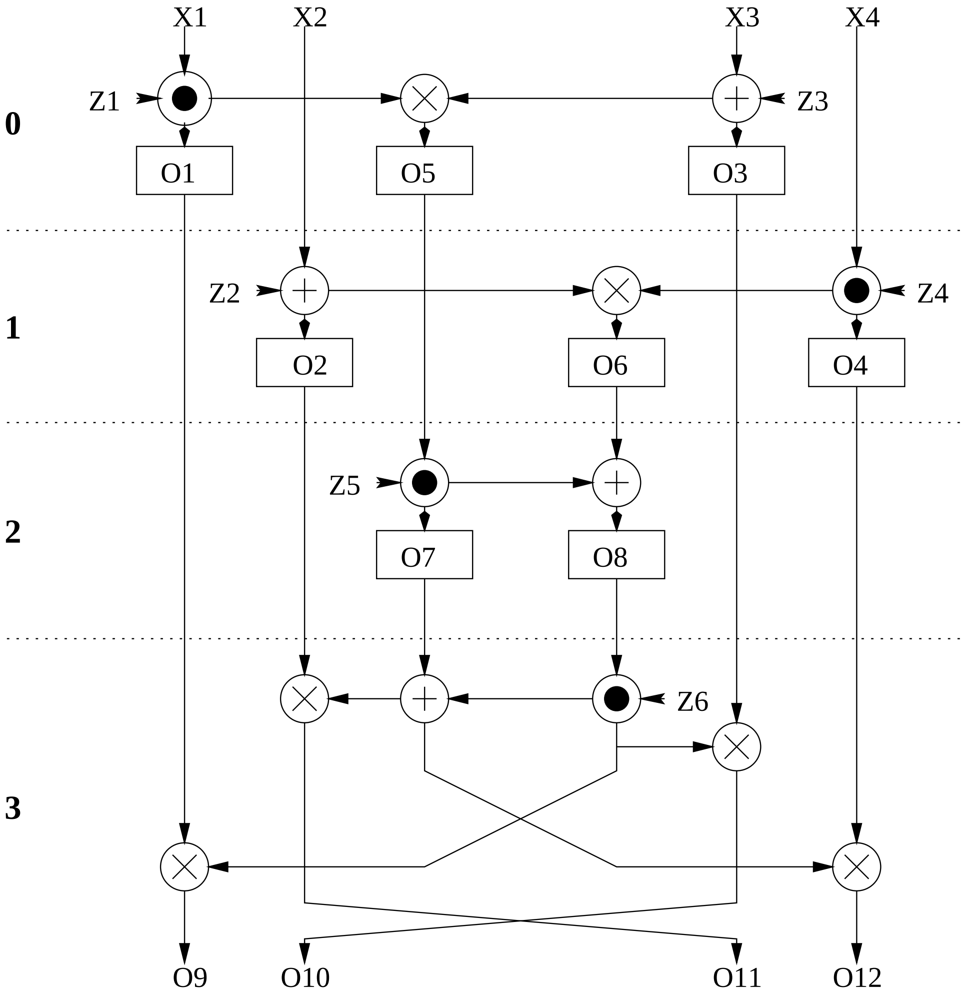
Es hat den Anschein, als ob zum Beispiel im nullten Zustand mehr als ein Multiplizierer und Addierer gebraucht wird. Tatsächlich wird aber jeweils nur *eine* Addition und *eine* Multiplikation durchgeführt. Die Ergebnisse der beiden Module werden eben nicht nur in $O1$ bzw. $O3$ gespeichert, sondern auch an die Eingänge eines XOR-Moduls geschaltet, um so $O6$ zu berechnen. Tatsächlich wird in jedem aufgeführten Zustand nur eine Addition und Multiplikation durchgeführt. Der letzte Zustand,der in der Tabelle nicht aufgeführt ist, ist wieder ein Haltezustand in dem nichts mehr berechnet wird.

Zusätzlich wurden in der Implementierung Zustandszähler und Schaltungslogik getrennt und in jeweils einen eigenen *Process* untergebracht, um den Sourcecode übersichtlicher zu machen.

5.7 Die hardwarenahe Version

Durch Betrachtung der optimierten Version kann man erkennen, daß sich eine Runde auch mit Hilfe von Multiplexern beschreiben läßt, was zu einer sehr hardwarenahen Beschreibung führt.

Im Prinzip steuert der Zustandszähler nur, welche Signale an die Eingänge des Multiplizierers und Addierers geschaltet werden. Die folgende Tabelle verdeutlicht das. MO bezeichnet in dieser Tabelle den Ausgang des Multiplizierers; „Add 1“ und „Add 2“ die beiden Eingänge des Addierers und „Mul 1“ und „Mul 2“ die beiden Eingänge des Multiplizierers.



Zustand	Berechnung
0	$O1 = (X1 \odot Z1)$
	$O3 = (X3 \oplus Z3)$
	$O5 = (X1 \odot Z1) \otimes (X3 \oplus Z3)$
1	$O2 = (X2 \oplus Z2)$
	$O4 = (X4 \odot Z4)$
	$O6 = (X2 \oplus Z2) \otimes (X4 \odot Z4)$
2	$O7 = (O5 \odot Z5)$
	$O8 = (O5 \odot Z5) \oplus O6$
3	$O9 = (O8 \odot Z6) \otimes O1$
	$O10 = (O8 \odot Z6) \otimes O3$
	$O11 = ((O8 \odot Z6) \oplus O7) \otimes O2$
	$O12 = ((O8 \odot Z6) \oplus O7) \otimes O4$

Abbildung 5: Die optimierte Runde

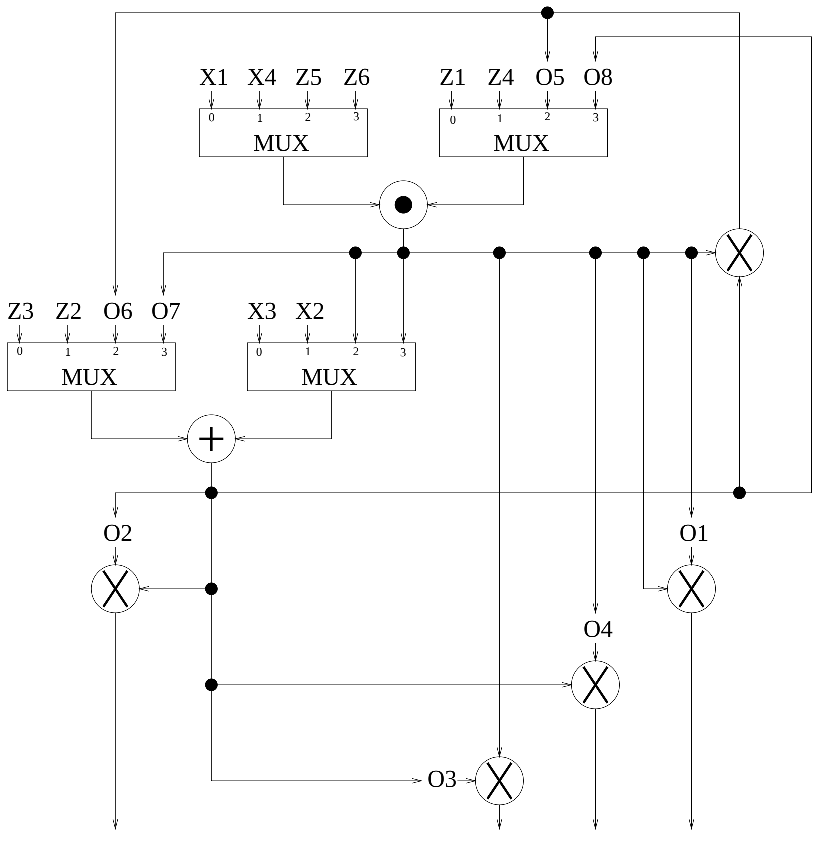


Abbildung 6: Eine Runde mit Multiplexern

Zustand	Add 1	Add 2	Mul 1	Mul 2	Ergebnis-register
0	X3	Z3	X1	Z1	O1 O3 O5
1	X2	Z2	X4	Z4	O2 O4 O6
2	MO	O5	O6	Z5	O7 O8
3	MO	O8	O7	Z6	

Zusätzlich müssen in bestimmten Zuständen die Ausgänge der beiden Module in Register zwischengespeichert werden.

Es ist klar, daß man nur einen Multiplexer vor jeden Eingang des Addierers und Multiplizierers zu schalten braucht, um die Berechnungslogik zu verwirklichen. Die Multiplexer werden dabei direkt durch das Zustandsregister gesteuert.

Die Zwischenspeicher lassen sich als Latches realisieren, d.h. als Register die den Wert an ihrem Eingang nur speichern, wenn eine logische „1“ an ihrem Steuereingang liegt und an sonsten ihren Wert behalten.

Die Beschreibung in VHDL enthält in dieser Version vor allem die Strukturbeschreibung der Schaltung und die Logik beschränkt sich nur noch auf den Zustandszähler. Die eigentliche Berechnungslogik steckt in den Multiplexern und ist nicht mehr direkt zu erkennen.

Für diese Implementierung wurden noch zwei weitere Module, die die Latches und die Multiplexer beschreiben, programmiert.

Diese Version benötigt nur noch vier Zustände und acht Latches, um die Zwischenergebnisse zu beschreiben. Eine graphische Darstellung der Schaltung ist in Abbildung 6 zu sehen. Die Latches sind mit $O1$ bis $O8$ gekennzeichnet; die Steuerleitungen wurden der Übersicht wegen weggelassen.

5.8 Die Ausgabe Transformation

Die vorhergehenden Versionen realisieren alle eine Runde des IDEA, aber nicht die Ausgabetransformation am Schluß der Berechnung.

Es wäre möglich für diese Transformation ein eigenes Modul zu programmieren, aber es stellte sich heraus, daß man ein Rundenmodul so erweitern kann, daß es auch die Ausgabetransformation berechnet.

Dazu muß man erkennen, daß die Ausgabetransformation große Ähnlichkeit mit dem Anfang einer Runde hat. Im Prinzip werden wieder zwei Multiplikationen und Additionen ausgeführt, allerdings mit anderen Schlüssel- und Eingabepaaren. Man muß eine Runde nur so konstruieren, daß sie die Berechnung im richtigen Moment abbricht und die richtigen Zwischenergebnisse an die Ausgabe schaltet, um die Ausgabetransformation zu realisieren.

Als erstes braucht eine Runde dafür ein zusätzliches Steuersignal, das anzeigt wann die Ausgabetransformation berechnet werden muß. Benutzt man die Runde mit Multiplexern,

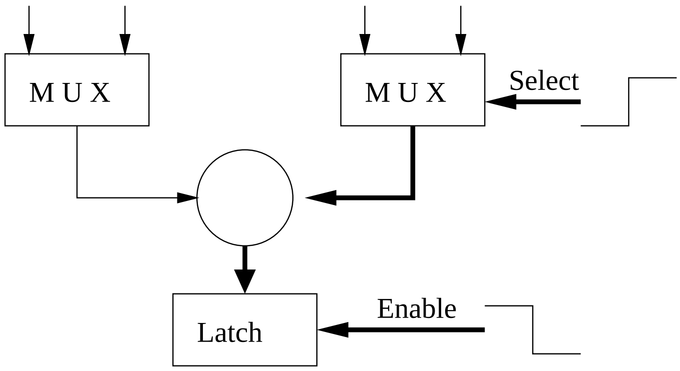


Abbildung 7: Ein Race zwischen *Enable* und *Select* Signal

dann läßt sich das Berechnen der Ausgabetransformation dadurch erreichen, daß man die Exklusiv-Oder Module vor den Ausgängen zu steuerbaren Modulen umwandelt, die entweder eine Exklusiv-Oder Operation durchführen oder einfach nur einen der beiden Eingänge weiterleiten.

Das funktioniert, weil an diesen Modulen genau die richtigen Zwischenergebnisse anliegen. Man muß diese Ergebnisse einfach nur an die Ausgabe weiterleiten, ohne sie zu verändern, um die Ausgabetransformation zu berechnen.

Ein steuerbares Exklusiv-Oder Modul erhält man, wenn man jedes Bit des zweiten Eingangs des Moduls mit einem logischen „Und“ mit der Steuerleitung verknüpft. Solange die Steuerleitung auf „1“ liegt, wird die Exklusiv-Oder Operation ausgeführt; liegt die Steuerleitung auf „0“, wird der erste Eingang einfach durchgeschaltet, da kein Bit mehr invertiert wird.

5.9 Die Runde ohne „Races“

Ein „Race“ entsteht, wenn das Ergebnis einer Berechnung von der Reihenfolge in der Signaländerungen eintreffen abhängt, obwohl diese Reihenfolge nicht vorhersagbar ist. Abbildung 7 zeigt so ein „Race“.

Leider enthält die hardwarenahe Version solche „Races“. Ändert sich der Zustandszähler des Rundenmoduls, so verändern sich die Steuerleitungen der Latches und die Ausgaben der Rechenmodule. Das Ergebnis der Rundenberechnung hängt davon ab, welche Änderung zu erst an einem Latch ankommt.

Die endgültige Version der Runde löst diese Problem, indem die Anzahl der Zustände verdoppelt wird und der Zustandszähler auf die fallende Flanke der Clock synchronisiert wird. Die Multiplexer werden nur von den beiden höherwertigen Bits des Zustandszählers gesteuert, d.h. in Zustand 0 und 1 schalten die Multiplexer ihren ersten Eingang durch, in Zustand 2 und 3 ihren Zweiten und so weiter. Abbildung 8 verdeutlicht wie sich die unterschiedli-

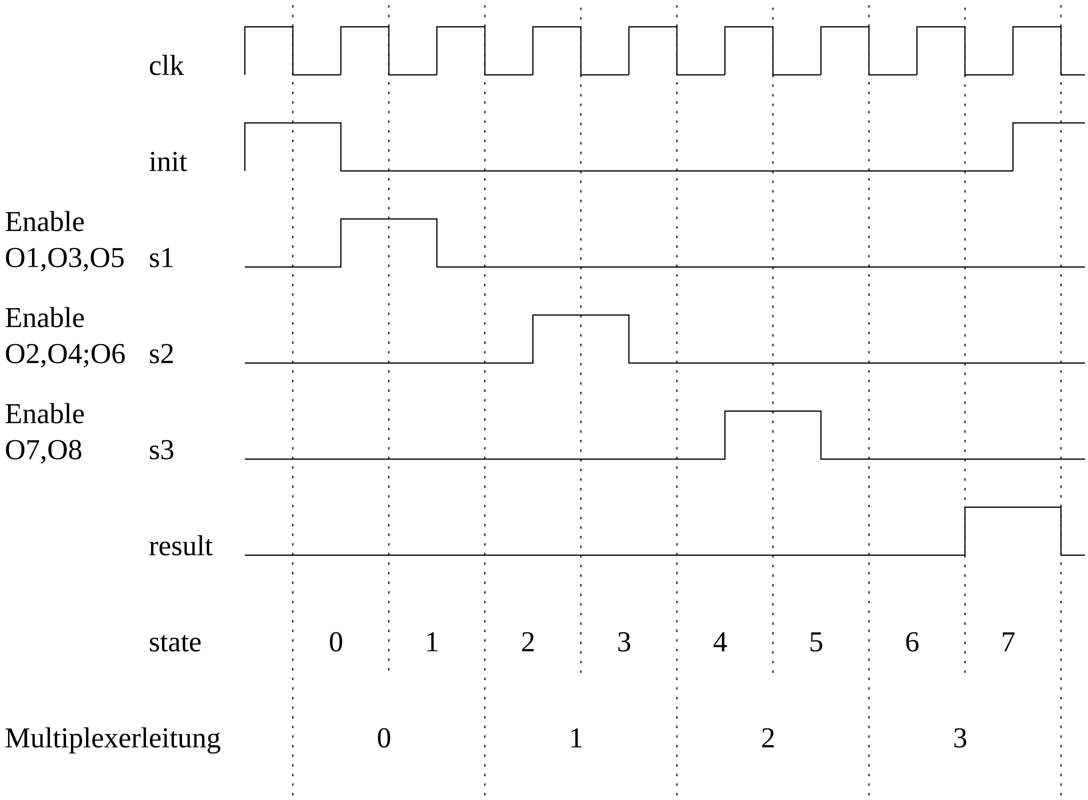


Abbildung 8: Die Signale in einer racefreien Runde

chen Signale während einer Runde verhalten. s1,s2 und s3 sind dabei die Steuerleitungen für die acht Latches ($O1$ - $O8$).

6 Simulation und Synthese

Wie schon erwähnt wurde zunächst die Referenzimplementation simuliert, um Erfahrungen mit dem VHDL Simulator von Synopsys zu sammeln. Mit Hilfe eines Satzes von bekannten Eingaben und den dazugehörigen Ergebnissen, wurde die korrekte Arbeitsweise des VHDL Codes überprüft.

Dieses Vorgehen wurde bei jeder Veränderung der Module wiederholt, wobei die Referenzimplementation dazu diente, weitere Testdaten zu liefern. Mit Hilfe des Simulators wurden die genauen Abläufe innerhalb einer Runde überprüft und analysiert. Diese Analyse führte zu der Entwicklung der hardwarenahen Beschreibung, die nur noch mit Multiplexern arbeitet. Durch weitere Testläufe wurde das Problem der „Races“ deutlich und die Runde wurde dementsprechend modifiziert. Abbildung 8 wurde ebenfalls mit Hilfe des Synopsyssimulators erzeugt.

Die Synthesewerkzeuge von Synopsys wurden benutzt, um eine grobe Abschätzung der Signallaufzeiten innerhalb der Rechenmodule durchzuführen. Mit Hilfe dieser Daten wurde die Signallaufzeit des längsten Signalpfades abgeschätzt. Dieser Pfad entsteht, wenn die Multiplexer ihren vierten Eingang durchschalten und geht durch folgende Module: Durch den Multiplexer vor dem Multiplizierer, durch den Multiplizierer, durch den Multiplexer vor dem Addierer, durch den Addierer und zu letzt durch verschiedene XOR-Gatter. Diese Signallaufzeit bestimmt die Taktrate, bei der die Schaltung noch korrekt arbeiten kann; das Synthesetool von Synopsys bestimmte dafür eine Zeit von ungefähr 230 Nanosekunden. Die Berechnung muß innerhalb von 1,5 Takten stattfinden (vgl. Abbildung 8), d.h. die maximale Taktrate beträgt mit diesen Daten ca 6.5 MHz. Die Abschätzung enthält noch keine Leitungsverzögerungen, deshalb sollte der reale Wert ein Stück darunter liegen.

Die *reale* Funktionsfähigkeit wurde mit Hilfe eines Prototyping Boards überprüft. Dazu wurden zwei Schaltungen realisiert:

- Eine einzelne Runde mit festen Teilschlüsseln
- Ein komplettes IDEA Modul mit einem festen Schlüssel

Als Test wurden verschiedene Eingabevektoren benutzt und die Ausgabe mit Hilfe eines Logikanalysers überprüft.

Es stellte sich heraus, daß mit dem benutzten Schlüssel selbst bei einer Taktrate von 10MHz noch die richtigen Ergebnisse berechnet werden. Diese hohe Taktrate ist möglich, weil die Signallaufzeiten durch den Multiplizierer datenabhängig sind und damit auch durch den Schlüssel beeinflußt werden. Der Schlüssel enthielt in der getesteten Schaltung sehr viele „0“ Bits und beschleunigt dadurch die Berechnung der Multiplikation enorm. Um einen realeren Wert zu erhalten, müßte man versuchen, einen möglichst ungünstigen Schlüssel zu verwenden, aber für diesen Versuch fehlte leider die Zeit.

7 Mögliche Optimierungen und Performancesteigerung

7.1 Pipelining

Beim Pipelining teilt man eine Berechnung in mehrere Phasen auf. Jede Phase berechnet Zwischenergebnisse, die dann von der nächsten Phase weiterverarbeitet werden können. Mit dieser Technik kann man den Durchsatz einer Schaltung erhöhen, da die Phasen parallel die Zwischenergebnisse für mehrere Eingabeblöcke berechnen (Abbildung 9). Eine Runde des IDEA läßt sich nur schlecht als Pipeline verwirklichen, da zum Beispiel das Zwischenergebnis der ersten Multiplikation am Schluß für die XOR-Operation erneut benötigt wird. Man müßte das Zwischenergebnis durch eine Register Pipeline laufen lassen, um es dann in einer späteren Phase benutzen zu können. Dazu kommt, daß man die Runde über mehrere FPGAs verteilen müßte, da auf ein FPGA nur ein einziger Multiplizierer paßt; diese Verteilung verkompliziert die äußere Logik.

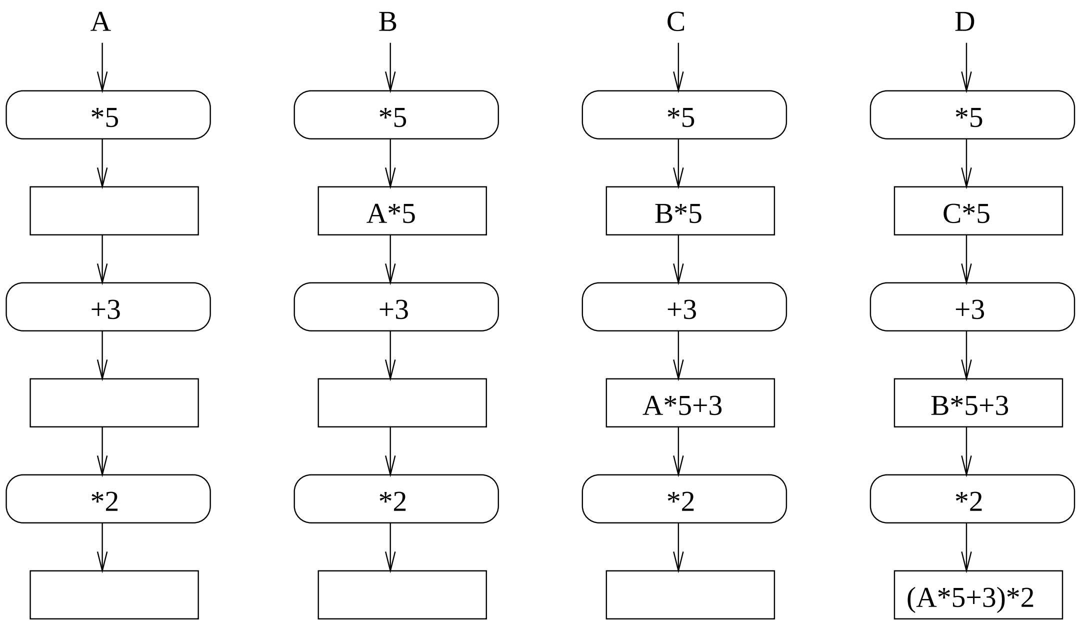


Abbildung 9: Der Term $(X * 5 + 3) * 2$ als Pipeline

Erfolgversprechender ist es, den gesamten IDEA zu pipelinen und dabei in einer Phase eine komplette Runde zu berechnen. Der Durchsatz könnte sich mit Hilfe einer solchen Pipeline maximal um das 8.5fache erhöhen, wenn man davon ausgeht, daß die Ausgabetransformation halb so viel Zeit wie eine komplette Runde benötigt. Außerdem könnte man in diesem Fall acht gleiche FPGAs verwenden, die jeweils eine Runde enthalten würden.

7.2 Performancebeurteilung

Der IDEA ist in der vorgestellten Implementation leider sehr langsam. Werden keine Pipelinestufen benutzt, so liegt die Performance wenn man die oben angegebene Abschätzung benutzt, deutlich unter der einer heutigen PC-CPU (vgl. Abbildung 10). Der Grund dafür ist, daß beim Entwurf des IDEA extra darauf geachtet wurde, daß er sich auf einem gängigen Workstation Prozessor gut realisieren läßt. Der IDEA benutzt deshalb vor allem arithmetische Operationen, die auf jedem modernen Prozessor in optimierter Form zur Verfügung stehen.

Leider läßt sich besonders die Multiplikation zweier 16 Bit Zahlen mit der benutzten Hardware nicht optimal realisieren. Die Größe eines Multiplikators ist enorm, so daß es nicht möglich ist, mehr als ein Multiplikationsmodul auf dem FPGA unterzubringen. Zusätzlich ist dieser Multiplikator auch noch sehr langsam, da die Signallaufzeiten lang sind.

Meiner Meinung nach dürfte es ohne Pipelinestufen kaum möglich sein, eine Hardware zu bauen, die von der Performance her besser ist, als die Implementation auf einem moder-

Prozessor	Algorithmus	MByte/s	Takte/Block
AMD K6-200MHz	IDEA	2.4	680
AMD K6-200MHz	DES	3.0	535
Intel PII-300MHz	IDEA	5.2	462
Intel PII-300MHz	DES	4.2	574

Abbildung 10: Performance von DES und IDEA als C Implementation

nen PC. Für mehrere Pipelinestufen braucht man vor allem wegen der Komplexität der Multiplikation viel Platz, FPGAs sind deshalb dafür nicht besonders gut geeignet.

Im Gegensatz dazu lassen sich Algorithmen, die vor allem auf der Manipulation von einzelnen Bits arbeiten, sehr gut als Hardware und nur schlecht als Software implementieren. Ein Beispiel für einen solchen Algorithmus ist der DES[6]. Auf der gleichen Zielarchitektur (Xilinx 4028 FPGAs) lassen sich hier über 400MBit/s als Durchsatzrate erreichen (siehe [4])! Man kann daher zusammenfassend sagen, daß es sich nicht besonders anbietet, den IDEA als Spezialhardware zu implementieren, zumindest nicht auf FPGAs.

Literatur

[1] Bruce Schneier. *Angewandte Kryptographie*, Addison-Wesley Verlag, Bonn, 1996;

[2] Xuejia Lai. *On the Design and Security of Block Ciphers*, Hartung-Gorre Verlag, Wissenschaftliche Publikationen, 1992;

[3] Lehmann, Wunder, Selz. *Schaltungsdesign mit VHDL*, Franzis-Verlag, Poing, 1994;

[4] Jens-Peter Kaps, Christof Paar. *Fast DES Implementations for FPGAs and its application to a Universal Key-Search Machine*, SAC' 98 (http://ece.wpi.edu/Research/crypt)

[6] Data Encryption Standard