Seminar Datentechnik

Modularmultiplikation

Till Merker

18. Dezember 2003

Inhaltsverzeichnis

1 Einleitung

In diesem Seminarvortrag sollten drei Optimierungen von Hardware-Implementierungen der Modularmultiplikation nach dem Montgomery-Verfahren dargestellt und verglichen werden.

Zuerst wurde eine kurze Einführung in die Thematik gegeben. Dann wurden verschiedene Algorithmen zur Berechnung der Modularmultiplikation genannt, und im Anschluss der Algorithmus von Montgomery näher beschrieben. Danach wurden die drei verschiedenen Optimierungen vorgestellt.

1.1 Definition der Modularmultiplikation

Definition 1 *Die Modularmultiplikation zweier Operanden X und Y wird berechnet, indem nach der normalen Multiplikation von X und Y der Rest bei Division durch den Modulus M gebildet wird:*

$$P = (X \cdot Y) \; mod \; M$$

1.2 Anwendungsgebiet

Die Modularmultiplikation wird hauptsächlich in der Kryptographie benötigt. Public-Key-Verfahren wie z.B. das Pohlig-Hellmann- oder das RSA-Verfahren verwenden die *Modulare Exponentiation* zur Ver-/Entschlüsselung von Nachrichten:

Definition 2 *Die modulare Exponentiation wird berechnet, indem nach der normalen Exponentiation noch eine Restbildung mod M vorgenommen wird:*

$$P = A^x \; mod \; M$$

Ein möglicher Algorithmus zur Berechnung der modularen Exponentiation sieht folgendermaßen aus:

Algorithmus 1

```
1 if x_{n-1} = 1 then P := A else P := 1 fi
2 for i := n - 2 to 0 do
3     P := P · P mod M
4     if x_i = 1 then P := P · A mod M fi
5 end
```

Man sieht, dass innerhalb der Schleife immer wieder eine bzw. zwei Modularmultiplikationen ausgeführt werden. Die Geschwindigkeit, mit der die Exponentiation durchgeführt werden kann, und damit auch die Performance von Ver- und Entschlüsselung bei Exponentiationschiffren, wird also maßgeblich durch die Geschwindigkeit der Modularmultiplikation bestimmt.

Aus diesem Grund implementiert man die Modularmultiplikation in Hardware, und versucht natürlich, diese Implementierungen zu optimieren.

2 Algorithmen

Zur Berechnung der Modularmultiplikation gibt es eine Vielzahl von Algorithmen. Einige davon sind hier aufgeführt:

- „Klassischer“Algorithmus:
  Multiplikation mit anschließender Restbildung
- Verschachtelte („interleaved“) Multiplikation und Reduktion:
  Reduktion des Zwischenergebnisses nach jedem Schritt der Multiplikation
- Barrett's Algorithmus
- Brickell's Algorithmus
- Der Algorithmus von Montgomery

Der Algorithmus von Montgomery soll im Folgenden näher beschrieben werden.

3 Montgomery's Algorithmus

Das Besondere an der Montgomery-Multiplikation ist, dass nicht, wie eigentlich gefordert, $P = X \cdot Y \; mod \; M$ berechnet wird, sondern ein zusätzlicher Faktor r^{-1} eingeführt wird:

Definition 3 $MonPro(X, Y) := (X \cdot Y \cdot r^{-1}) \; mod \; M$

Damit das Verfahren funktioniert, müssen der Faktor r und der Modulus M teilerfremd sein, d.h. $ggt(r, M) = 1$. Üblicherweise wird $r = 2^n$ gewählt; der Modulus M muss dann ungerade sein.

Der Faktor r^{-1} ist natürlich nicht erwünscht und muss nach der Multiplikation entfernt werden. Das Montgomery-Verfahren lässt sich aber sehr effizient in Hardware implementieren, weshalb dieser zusätzliche Aufwand in Kauf genommen wird.

3.1 Kompensation des Faktors

Zum Entfernen des unerwünschten Faktors r^{-1} gibt es zwei mögliche Vorgehensweisen.

1. **Multiplikation in zwei Schritten**
   Zuerst wird eine normale Montgomery-Multiplikation ausgeführt:

$$P' = MonPro(X, Y) = (X \cdot Y \cdot r^{-1}) \; mod \; M$$

Dann wird das Zwischenergebnis mit $r^2 \; mod \; M$ multipliziert:

$$\begin{aligned} P &= MonPro(P', r^2 \; mod \; M) \\ &= X \cdot Y \cdot r^{-1} \cdot r^2 \cdot r^{-1} \; mod \; M \\ &= X \cdot Y \; mod \; M \end{aligned}$$

2. **Umwandeln der Operanden**
   Vor der Multiplikation werden die Operanden mit einem Faktor r multipliziert:
   $$X' = X \cdot r \; mod \; M, \; Y' = Y \cdot r \; mod \; M$$
   Dann wird das Montgomery-Produkt P' berechnet:
   $$\begin{aligned} P' &= MonPro(X', Y') \\ &= (X \cdot r \cdot Y \cdot r \cdot r^{-1}) \; mod \; M \\ &= (X \cdot Y \cdot r) \; mod \; M \end{aligned}$$
   Das Ergebnis P' hat jetzt, genau wie die Operanden, einen zusätzlichen Faktor r. Dieser kann durch Multiplikation mit 1 entfernt werden:
   $$\begin{aligned} P &= MonPro(P', 1) \\ &= (X \cdot Y \cdot r \cdot 1 \cdot r^{-1}) \; mod \; M \\ &= (X \cdot Y) \; mod \; M \end{aligned}$$
   Dieses Verfahren ist sinnvoll, wenn viele Modularmultiplikationen hintereinander ausgeführt werden müssen, wie z.B. bei der modularen Exponentiation. Dann müssen die Operanden nur einmal ganz am Anfang umgewandelt werden. Nach der Durchführung der Multiplikationen wird das Endergebnis durch $MonPro(P', 1)$ ermittelt.

3.2 Berechnung von $MonPro(X, Y)$

Zuerst soll ein Verfahren zur Berechnung von $X \cdot Y$ gezeigt werden, aus dem dann der Algorithmus für $MonPro(X, Y)$ abgeleitet wird.

3.2.1 Berechnung von $X \cdot Y$

Der Operand X wird zunächst in seine Bit-Darstellung umgewandelt.

$$X = (x_{n-1}, ..., x_2, x_1) = \sum_{i=0}^{n-1} x_i 2^i$$

Das Produkt $X \cdot Y$ lässt sich dann folgendermaßen schreiben:

$$\begin{aligned} X \cdot Y &= \sum_{i=0}^{n-1} x_i 2^i \cdot Y \\ &= (x_0 + x_1 2^1 + ... + x_{n-1} 2^{n-1}) \cdot Y \\ &= ((((0 + x_{n-1} Y) \cdot 2) + x_{n-2} Y) \cdot 2 + ... + x_1 Y) \cdot 2 + x_0 Y \end{aligned}$$

Aus der unteren Darstellung lässt sich der Algorithmus für die Berechnung von $X \cdot Y$ ableiten:

Algorithmus 2

```
1 P := 0
2 for i := n − 1 to 0 do
3     P := 2 · P
4     P := P + x_i · Y
5 end
```

3.2.2 Berechnung von $X \cdot Y \cdot r^{-1}$

Für die Montgomery-Multiplikation muss noch der zusätzliche Faktor $r^{-1} = 2^{-n}$ berücksichtigt werden. Berechnet wird

$$\begin{aligned} 2^{-n} \cdot X \cdot Y &= 2^{-n} \cdot (x_0 + x_1 2^1 + ... + x_{n-1} 2^{n-1}) \cdot Y \\ &= (x_0 2^{-n} + x_1 2^{-n+1} + ... + x_{n-1} 2^{-1}) \cdot Y \\ &= (((0 + x_0 Y) \cdot 2^{-1} + x_1 Y) \cdot 2^{-1} + ... + x_{n-1} Y) \cdot 2^{-1} \end{aligned}$$

Wieder lässt sich aus der unteren Darstellung der Algorithmus zur Berechnung ableiten:

Algorithmus 3

```
1 P := 0
2 for i := 0 to n − 1 do
3     P := P + x_i · Y
4     P := P div 2
5 end
```

Vergleicht man diesen Algorithmus mit dem für die Berechnung von $X \cdot Y$, so fällt auf, dass sie sich nur in zwei Punkten unterscheiden: Die Reihenfolge der Summation ist umgekehrt (hier von 0 bis n - 1), und die Multiplikation mit zwei wurde durch eine Division durch zwei ersetzt.

3.2.3 Berechnung von $X \cdot Y \cdot r^{-1} \, mod \, M$

Im letzten Abschnitt wurde gezeigt, wie $X \cdot Y \cdot r^{-1}$ berechnet werden kann. Nun soll aber $X \cdot Y \cdot r^{-1} \, mod \, M$ berechnet werden. Eine Möglichkeit wäre, nach jeder Iteration zu Überprüfen, ob das Zwischenergebnis größer als der Modulus M ist, und wenn ja, M zu subtrahieren.

Beim Montgomery-Verfahren wird aber ein Ansatz gewählt. Es wird nach jedem Schritt überprüft, ob das Zwischenergebnis gerade oder ungerade ist. Das kann einfach durch Test eines Bits erreicht werden. Ist es ungerade, wird einmal der Modulus M addiert. Da der Modulus immer gerade ist (s. Kapitel 3 auf

Seite 4), ist das Ergebnis nach diesem Schritt immer eine gerade Zahl. Das hat zur Folge, dass die anschließende Division durch zwei ohne Informationsverlust einfach durch einen Rechtsshift gemacht werden kann, da das letzte Bit immer 0 ist. Das Ergebnis bleibt trotz der Addition richtig, weil natürlich $P \, mod \, M = (P + M) \, mod \, M$ gilt.

Hier der komplette Algorithmus für die Berechnung von $MonPro(X, Y)$:

Algorithmus 4

```
1 P := 0
2 for i := 0 to n − 1 do
3     P := P + x_i · Y
4     if p_0 = 1 then P := P + M fi
5     P := P div 2
6 end
```

3.3 Komplexität des Verfahrens

3.3.1 Zeitkomplexität

Nun wollen wir uns die Zeitkomplexität einer einfachen Hardware-Implementierung der Montgomery-Multiplikation ansehen. Die Dauer eines Schleifendurchlaufs wird im wesentlichen bestimmt durch die zwei Additionen (Division durch zwei und Test auf gerade/ungerade benötigen keine bzw. vernachlässigbare Zeit).

Der schnellste Standard-Addierer, der Carry-Lookahead-Adder, benötigt die Zeit $O(\log n)$, wobei n die Anzahl der Bits der beteiligten Operanden ist.

Die Anzahl der Schleifendurchläufe ist n. Es müssen allerdings zwei Multiplikationen ausgeführt werden, da der Faktor r^{-1} noch entfernt werden muss, so dass die Gesamtanzahl der Schleifendurchläufe gleich $2n$ ist. Die Zeitkomplexität ist also insgesamt $O(n \log n)$.

3.3.2 Flächenkomplexität

Da die Modularmultiplikation in Hardware implementiert werden soll, interessiert uns aber nicht nur die Zeit, sondern auch die benötigte Chipfläche.

Wir brauchen vier n-bit-Register (zwei zum Speichern der Operanden X und Y, eines für den Modulus M und eines für das Zwischenergebnis P). Des weiteren werden zwei n-bit-Addierer benötigt.

Hier ist aber nur die asymptotische Komplexität interessant - diese ist offensichtlich $O(n)$.

3.3.3 AT-Komplexität

Als Zusammenfassung von Zeit- und Flächenkomplexität ist die Area-Time (AT) Komplexität definiert:

Definition 4 *AT-Komplexität* $AT(n) = A(n) \cdot T(n)$

In diesem Fall ist sie $AT(n) = O(n) \cdot O(n \log n) = O(n^2 \log n)$.

4 Optimierungen

Es gibt verschiedene Ansätze, das Verfahren zu optimieren. Man kann versuchen, die benötigte Zeit so weit wie möglich zu senken, ohne Rücksicht auf die Chipfläche. Das ist immer durch Parallelisierung möglich - allerdings steigt die Chipfläche stärker an als die Zeit sinkt.

Hier sollen sowohl Zeit- als auch Flächenkomplexität optimiert werden (AT-Komplexität).

In den folgenden drei Optimierungen ist die asymptotische AT-Komplexität immer $O(n^2)$. Deshalb muss auch der Faktor vor dem n^2 berücksichtigt werden. Es wird also eine Gewichtung der verwendeten Bauteile bezüglich benötigter Zeit und Fläche notwendig.

4.1 Gewichtungen

Die folgenden Gewichtungen sind mehr oder weniger willkürlich und in hohem Masse von der verwendeten Hardware-Technologie abhängig. Die damit ermittelten Werte können somit nur als Schätzwerte angesehen werden.

Definition 5 *benötigte Zeit und Fläche der verwendeten Bauteile:*

Bauteil	***Zeit***	***Fläche***
Volladdierer	*1*	*8*
Lookup-Table	*1*	*1 (pro bit)*
Register	*0*	*4 (pro bit)*

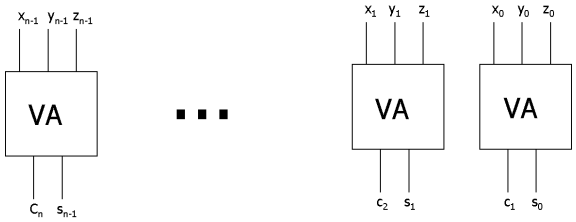
5 Optimierung 1 (Y. Sae Kim, W. Seok Kang, J. Rim Choi)

Die erste Optimierung wurde beschrieben in Young Sae Kim, Woo Seok Kang, Jun Rim Choi: „Implementation Of 1024-Bit Modular Processor For RSA Cryptosystem“. Durch Verwendung von Carry-Save-Addierern wird eine Verringerung der Zeitkomplexität von $O(n \log n)$ auf $O(n)$ erreicht.

5.1 Carry-Save-Addierer

Ein Carry-Save-Addierer addiert, anders als ein normaler Addierer, *drei* Summanden zu *zwei* Ergebnissen. Er besteht aus n parallel geschalteten Volladdierern, die keinerlei Verbindung untereinander haben (s. Abbildung 1). Deshalb ist die Zeit, die für eine Addition benötigt wird, gleich der Zeit, die ein Volladdierer benötigt, also $O(1)$.

Abbildung 1: Carry-Save-Addierer



5.2 Modifizierter Algorithmus

Weil der Carry-Save-Addierer zwei Ergebnisse produziert, muss das Zwischenergebnis P durch ein Paar (C, S) ersetzt werden. Hier der modifizierte Algorithmus:

Algorithmus 5

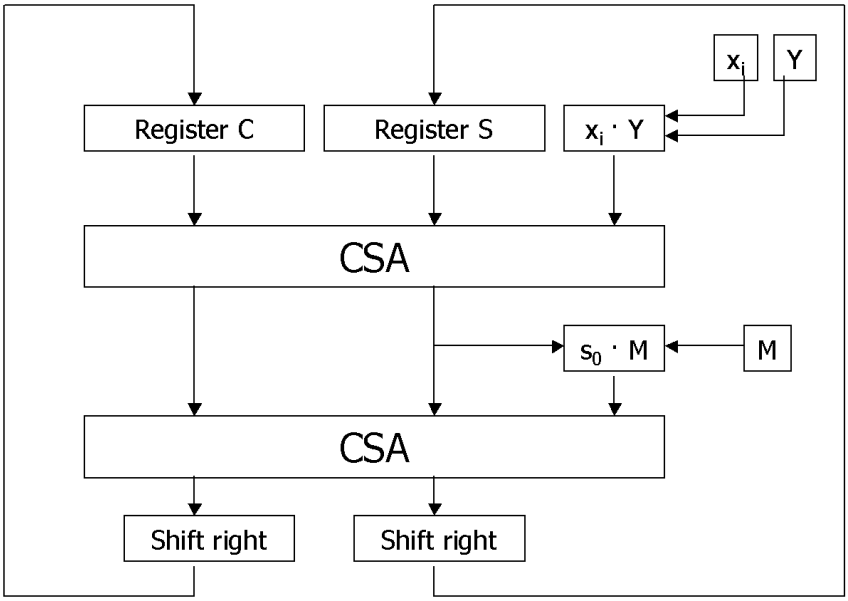
```
1 (C, S) := 0
2 for i := 0 to n − 1 do
3     (C, S) := C + S + x_i · Y
4     if s_0 = 1 then (C, S) := C + S + M fi
5     (C, S) := (C, S) div 2
6 end
```

Die beiden Additionen in Zeile 3 und 5 addieren jeweils drei Summanden zu zwei Ergebnissen und können durch Carry-Save-Addierer realisiert werden. Ein Blockschaltbild des Schleifenrumpfs ist in Abbildung 2 zu sehen.

5.3 AT-Komplexität

Jetzt soll die AT-Komplexität dieser Optimierung ermittelt werden.

Abbildung 2: Blockschaltbild Optimierung 1



5.3.1 Zeitkomplexität

Die Dauer einer Iteration wird bestimmt durch die zwei hintereinander geschalteten Carry-Save-Addierer, die wie in 5.1 gezeigt genau die Zeit eines Volladdierers benötigen.

$$T_1 = 2 \cdot T_{VA} = 2 \cdot 1 = 2$$

Die Anzahl der Schleifendurchläufe ist $2n$ (n für eine Multiplikation, 2 Multiplikationen zum Entfernen des Faktors r^{-1}). Insgesamt ergibt sich für die Zeitkomplexität

$$T = 2 \cdot 2n = 4n$$

5.3.2 Flächenkomplexität

Folgende Bauteile werden benötigt:

- 5 n-bit Register (X, Y, M, C, S): $A_1 = 5 \cdot n \cdot A_{Reg} = 5 \cdot n \cdot 4 = 20n$
- 2 n-bit Carry-Save-Addierer: $A_2 = 2 \cdot n \cdot A_{VA} = 2 \cdot n \cdot 8 = 16n$

Die Flächenkomplexität ist also $A = A_1 + A_2 = 36n$.

Für Die AT-Komplexität ergibt sich $AT = A \cdot T = 4n \cdot 36n = 144n^2$.

6 Optimierung 2 (V. Bunimov, M. Schimmler, B. Tolg)

Die nächste Optimierung, beschrieben in V.Bunimov, M.Schimmler, B.Tolg: „A Complexity-Effective Version Of Montgomery's Algorithm“, baut auf der vorherigen auf. Fasst man die beiden Additionen in der Schleife zu einer einzigen zusammen, gibt es vier Möglichkeiten: Entweder es wird 0, Y, M, oder Y + M addiert. x_i bestimmt ob Y addiert wird, und s_0, ob M addiert wird.

Diese vier verschiedenen Werte kann man nun im voraus berechnen und in einer Lookup-Table speichern. In der Schleife muss dann nur noch eine einzige Addition ausgeführt werden, nämlich das Zwischenergebnis plus den richtigen Eintrag aus der Lookup-Table.

Den richtigen Wert ermittelt man aus x_i, c_0, s_0, y_0:

- Wenn $x_i = 1$, dann addiere Y.
- Wenn $c_0 \oplus s_0 \oplus x_i y_0$ (das neue s_0!) $= 1$, dann addiere M.

6.1 Modifizierter Algorithmus

Im Algorithmus wird eine neue Variable I eingeführt, die den richtigen Wert aus der Lookup-Table temporär speichert:

Algorithmus 6

```
1  (C, S) := 0
2  for i := 0 to n − 1 do
3      if x_i = 1
4        then
5              if c_0 ⊕ s_0 ⊕ y_0 = 1 then I := Y + M else I := Y fi
6         else
7              if c_0 ⊕ s_0 = 1 then I := M else I := 0 fi
8      fi
9      (C, S) := C + S + I
10     (C, S) := (C, S) div 2
11 end
```

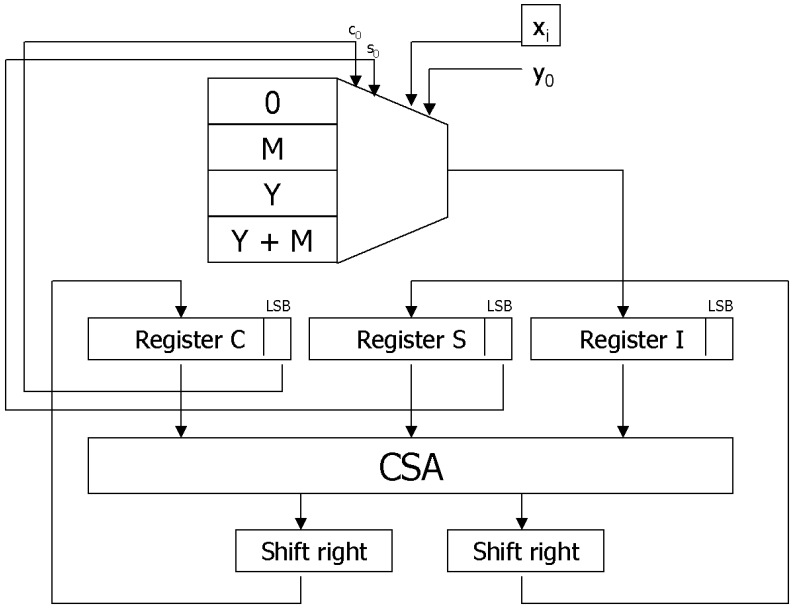
Ein Blockschaltbild dieser Optimierung zeigt Abbildung 3.

6.2 AT-Komplexität

6.2.1 Zeitkomplexität

Der Zugriff auf die Lookup-Table und die Addition können parallel ausgeführt werden, so dass die Dauer eines Schleifendurchlaufs nur durch den Carry-Save-

Abbildung 3: Blockschaltbild Optimierung 2



Addierer bestimmt wird:

$$T_1 = 1 \cdot T_{VA} = 1$$

Die Anzahl der Schleifendurchläufe ist wie vorher $2n$, so dass die Zeitkomplexität insgesamt

$$T = 1 \cdot 2n = 2n$$

ergibt.

6.2.2 Flächenkomplexität

Die Register für Y und M entfallen, da die Werte jetzt in der Lookup-Table gespeichert sind. Folgende Bauteile werden noch benötigt:

- 4 n-bit Register (X, C, S, I): $A_1 = 4 \cdot n \cdot A_{Reg} = 4 \cdot n \cdot 4 = 16n$
- 4 n-bit Einträge in der Lookup-Table: $A_2 = 4 \cdot n \cdot A_{Lookup} = 4 \cdot n \cdot 1 = 4n$
- 1 n-bit Carry-Save-Addierer: $A_2 = 1 \cdot n \cdot A_{VA} = 1 \cdot n \cdot 8 = 8n$

Die Flächenkomplexität ist also $A = A_1 + A_2 + A_3 = 28n$.

Für Die AT-Komplexität ergibt sich $AT = A \cdot T = 2n \cdot 28n = 56n^2$.

7 Optimierung 3 (Stephen E. Eldridge, Colin D. Walter)

In der dritten Optimierung, aus Stephen E. Eldridge, Colin D. Walter: „Hardware Implementation Of Montgomery's Modular Multiplication Algorithm“wird durch eine andere redundante Zahlendarstellung ebenfalls eine Addition in konstanter Zeit erreicht.

7.1 Redundante Darstellung der Zahlen

Eine Stelle kann nicht, wie in der binären Darstellung üblich, nur die Werte 0 und 1 annehmen, sondern auch den Wert 2. Dargestellt wird sie durch zwei Bits:

Bit 1	Bit 2	Bedeutung
0	0	0
0	1	1
1	0	2
1	1	kommt nicht vor

Die Variablen Y und P haben diese redundante Darstellung, X und M werden normal gespeichert.

7.2 Modifizierter Algorithmus

Die beiden Additionen innerhalb der Schleife werden hier zu einer einzigen Addition zusammengefasst, wobei vorher ein temporäres Bit q_i berechnet wird, das angibt, ob M addiert werden muss oder nicht (Zeile 3).

Algorithmus 7

1 $P := 0$
2 **for** $i := 0$ **to** $n - 1$ **do**
3 $\quad q_i := p_0 \oplus x_i y_0$
4 $\quad P := (P + x_i Y + q_i M) \text{ div } 2$
5 **end**

Dieser Algorithmus wird nun optimiert, indem nicht mehr q_i für *diese*, sondern q_{i+1} für die *nächste* Iteration berechnet wird:

Algorithmus 8

1 $P := 0$
2 **for** $i := 0$ **to** $n - 1$ **do**
3 $\quad P := (P + x_i Y + q_i M) \text{ div } 2$
4 $\quad q_{i+1} := p_0 \oplus x_i y_0$
5 **end**

Wenn man jetzt den Operanden Y um eine Stelle nach links schiebt, so dass das 0-te Bit von Y immer 0 ist, wird der Term in Zeile 4 unabhängig von $x_i y_0$. Die Schleife muss dann allerdings einen Durchlauf mehr machen, da der Rechtsshift durch eine zusätzliche Division durch 2 kompensiert werden muss.

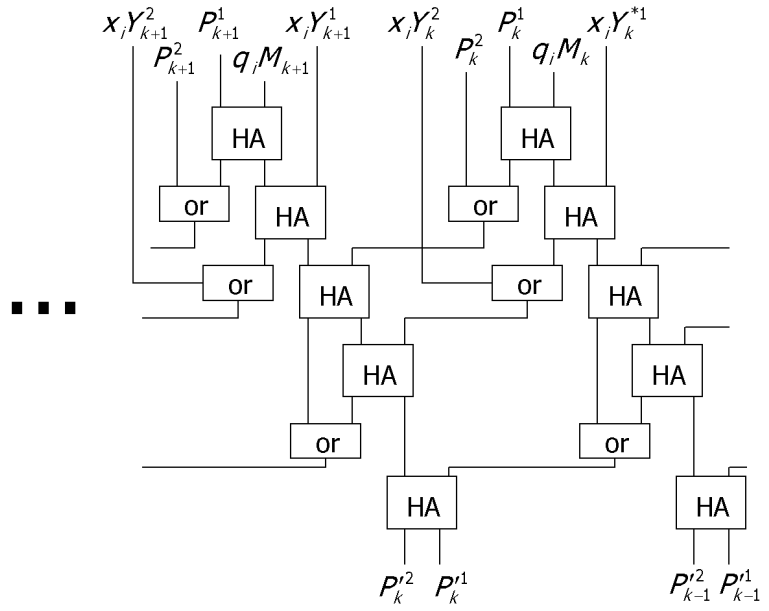
Algorithmus 9

```
1 P := 0
2 for i := 0 to n do
3     P := (P + x_i Y + q_i M) div 2
4     q_{i+1} := p_0
5 end
```

Tatsächlich wird Y sogar um zwei Stellen nach oben geschoben, da q_{i+1} dann noch schneller berechnet werden kann, und so einige Gatter zur Signalverstärkung nicht mehr im kritischen Pfad liegen. Darauf soll hier aber nicht näher eingegangen werden.

Das Schaltbild für den Addierer, der die beiden Additionen ausführt, ist in Abbildung 4 zu sehen.

Abbildung 4: Modifizierter Addierer aus Optimierung 3



7.3 AT-Komplexität

7.3.1 Zeitkomplexität

In dieser Optimierung werden keine Volladdierer verwendet. Deshalb müssen die Logik-Gatter in Volladdierer ($T_{VA} = 1$) umgerechnet werden. Die Tiefe der Logik beträgt sechs Gatter (siehe Abbildung 4), was in etwa zwei Volladdierern entspricht.

$$T_1 = 2 \cdot T_{VA} = 2 \cdot 1 = 2$$

Die Anzahl der Iterationen beträgt wie bisher 2n, so dass sich für die Zeitkomplexität

$$T = 2 \cdot 2n = 4n$$

ergibt.

7.3.2 Flächenkomplexität

Auch hier müssen die Logik-Gatter in Volladdierer umgerechnet werden. Pro bit sind 13 Gatter notwendig, das ist ungefähr die Anzahl Gatter für 2,5 Volladdierer.

$$A_1 = 2,5 \cdot n \cdot A_{VA} = 2,5 \cdot n \cdot 8 = 20n$$

Des weiteren werden 6 n-bit Register benötigt (X, Y^1, Y^2, M, R^1, R^2).

$$A_2 = 6 \cdot n \cdot A_{Reg} = 6 \cdot n \cdot 4 = 24n$$

Insgesamt ist

$$A = A_1 + A_2 = 44n$$

Als AT-Komplexität ergibt sich $AT = A \cdot T = 4n \cdot 44n = 176n^2$.

8 Zusammenfassung, Vergleich

Eine einfache Implementierung des Montgomery-Verfahrens zur Modularmultiplikation hat die AT-Komplexität $O(n^2 \log n)$. Es wurden drei verschiedene Optimierungen vorgestellt, die alle die (asymptotische) AT-Komplexität $O(n^2)$ haben. Hier eine Übersicht:

	T(n)	**A(n)**	**AT(n)**
Optimierung 1 (Y. Sae Kim, W. Seok Kang, J. Rim Choi)	$4n$	$36n$	$144n^2$
Optimierung 2 (V. Bunimov, M. Schimmler, B. Tolg)	$2n$	$28n$	$56n^2$
Optimierung 3 (Stephen E. Eldridge, Colin D. Walter)	$4n$	$44n$	$176n^2$