LAUFZEITOPTIMIERTE VHDL BIBLIOTHEK ZUR VERIFIKATION UND SIMULATION KRYPTOGRAPHISCHER PROZESSOREN

Mathias Schmalisch, Hagen Ploog und Dirk Timmermann

Institut für Angewandte Mikroelektronik und Datentechnik, Universität Rostock
Richard-Wagner-Str. 31, 18119 Rostock
{mathias.schmalisch, hagen.ploog, dirk.timmermann}@uni-rostock.de

Kurzfassung: In diesem Beitrag wird eine umfassende Bibliothek mit arithmetischen und kryptographischen Funktionen vorgestellt. Diese Bibliothek wurde in VHDL entwickelt und unterstützt sowohl Langzahlarithmetik als auch Multiple Precision Arithmetic (MPA). Dadurch eignet sich die Bibliothek insbesondere zur simulatorunabhängigen Simulation und Verifikation kryptographischer Prozessoren, die auf Langzahlarithmetik beruhen. Des weiteren wurde mit Hilfe der Performance Messung die Bibliothek auf ihre Ausführungszeit hin optimiert.

1 EINLEITUNG

Mit dem ständig steigenden Kommunikationsaufkommen ist die Sicherheit bei der Übertragung von Informationen und Daten ein zunehmend gefragteres Thema. Unter Sicherheit wird in diesem Beitrag nicht die Übertragung als solche, sondern die Unmöglichkeit der Interpretation der mitgehörten Information verstanden. Um diese Sicherheit zu garantieren werden kryptographische Prozessoren benötigt, die durch Verschlüsselung sensible Daten vor den Zugriff durch Unbefugte schützen. Derartige Prozessoren können dann z.B. in Geräten zur mobilen Kommunikation eingesetzt werden [1].
Durch die zunehmende Komplexität digitaler Schaltungen wird immer mehr Zeit bei der Entwicklung in das Simulieren und Testen investiert. Bei den heute üblichen Schaltungen nehmen die Tests bereits 50 – 70% der eigentlichen Entwicklungszeit ein. Aus diesem Grund ist es notwendig, effizientere Methoden für die Simulation zu entwickeln.
So sind die Standard VHDL-Bibliotheken [2] nur unzureichend geeignet, um mit langen Zahlen zu rechnen. Aber gerade diese werden in der Kryptographie z.B. beim RSA-Algorithmus [3] benötigt. Bei diesem Algorithmus beträgt die momentan übliche Schlüssellänge 1024 Bit. Aktuelle Untersuchungen [4] haben gezeigt, dass die Schlüssellänge bis ins Jahr 2050 auf 4096 Bit erweitert werden muss, damit der Algorithmus weiterhin sicher ist.
Um z.B. einen RSA-Prozessor zu simulieren, wird eine Testbench mit Testvektoren benötigt. Üblicherweise werden für die Erzeugung von Testvektoren Programme geschrieben, die diese in eine Textdatei ablegen, um sie dann in den Simulator einlesen zu können. Dadurch wird die Anzahl an möglichen Tests durch die Anzahl der Testvektoren in der erzeugten Datei begrenzt. Besser währe eine automatische Erzeugung von Testvektoren für die Simulation. Für solche Fälle bieten die VHDL-Simulatoren entsprechende C-Schnittstellen an ([5],[6]). Über diese Schnittstellen kann der Simulator mit einem C-Programm gekoppelt werden, welches dann die entsprechenden Testvektoren für die Simulation berechnet. Probleme treten auf, wenn der Entwickler und der Kunde unterschiedliche Simulatoren einsetzen, denn die C-Schnittstellen sind bisher nicht standardisiert. Somit müsste dann jedes Mal das C-Programm an die entsprechende Schnittstelle des

Simulators angepasst werden, was mit hohem Aufwand verbunden ist. Besser währe daher eine VHDL-Bibliothek, welche die benötigte Funktionalität bietet, um damit automatische Tests ablaufen lassen zu können. Solch eine Bibliothek, die simulatorunabhängig entsprechende Testvektoren erzeugen und die Ergebnisse vergleichen kann, wird in den folgenden Kapiteln beschrieben.

2 COMPUTER ARITHMETIK

2.1 Langzahlarithmetik

Um mit großen Zahlen zu rechnen wird im allgemeinen die Fließkommaarithmetik verwendet. Dabei wird nur eine gewisse Anzahl an Stellen genau berechnet und der Rest wird gerundet. Diese Art der Arithmetik ist für kryptographische Algorithmen nicht geeignet, denn dort muß auf jede Stelle genau gerechnet werden. Daher kommt für diesen Anwendungsfall auch nur die Langzahlarithmetik in Frage. Denn damit können auch große Zahlen (>1024 Bit) bis auf die letzte Stelle genau berechnet werden.

2.2 Multiple Precision Arithmetic (MPA)

Beim Rechnen mit großen Zahlen entstehen entsprechend große Zwischenergebnisse. Diese Zwischenergebnisse werden oftmals in externen Speicher zwischengespeichert, um so Flip-Flops einzusparen und gleichzeitig ein entsprechend kleineres Design zu erhalten. Diese Technik wird meist bei begrenzter Chipfläche eingesetzt, z.B. Smartcards oder auch FPGAs. Die Busbreite des externen Speichers ist daher eine vorgegebene Eigenschaft, welche bei der Entwicklung einer digitalen Schaltung beachtet werden muss [7].
In der Multiple Precision Arithmetic wird eine große Zahl X, wobei x die Anzahl Bitstellen von X darstellt, in mehrere Teile aufgeteilt. Diese s Teile haben die Bitbreite w, wobei w in diesem Fall die Busbreite des externen Speichers hat (z.B. w = 8, 16 oder 32). So kann dann eine Zahl X folgendermaßen dargestellt werden:

$$X = \sum_{i=0}^{x-1} x_i 2^i = \sum_{i=0}^{s-1} d_i \left(2^w\right)^i = \sum_{i=0}^{s-1} d_i W^i \qquad (1)$$

Wenn $w * s > x$ ist, werden die höchsten Stellen von X mit Nullen aufgefüllt. Die arithmetischen Operationen werden in der MPA rekursiv ausgeführt, wobei der Index von 0 bis (s-1) läuft, mit $s = x / w$. Algorithmus 1 zeigt die Ausführung einer Addition in MPA Sie wird nach dem gleichen Prinzip wie eine Addition zur Basis 2 ausgeführt, wobei allerdings die Basis W verwendet wird [8].

Algorithmus 1: MPA-Addition
EINGABE: x, y in MPA-Darstellung mit der Basis $W = 2^w$
AUSGABE: z = x + y
1. $c \leftarrow 0$
2. Für i von 0 bis (s - 1) mache folgendes:
   2.1. $z_i \leftarrow (x_i + y_i + c) \bmod W$
   2.2. $c \leftarrow 0$ wenn $((x_i + y_i + c) < W)$ sonst $c \leftarrow 1$
3. Gebe (z) zurück

Dieser Algorithmus kann verbessert werden, indem eine Zahl *t* mit der doppelten Ziffernbreite verwendet wird. So können dann die Zeile 2.1. und 2.2. durch folgende ersetzt werden:

2.1. $t \leftarrow x_i + y_i + c$
2.2. $z_i \leftarrow t[0]$, $c \leftarrow t[1]$

2.3 Darstellung vorzeichenbehafteter Zahlen

Für die Darstellung negativer Zahlen gibt es mehrere Möglichkeiten. Für eine möglichst einfache Implementierung der VHDL-Bibliothek eignet sich die Zweierkomplementdarstellung (allgemein r-Komplement, mit Basis $r = 2^w$) am besten.

$$\begin{aligned} A_{ZK} &= 0a_{n-2}a_{n-3}...a_1a_0 && \text{für eine n-stellige positive Zahl} \\ -A_{ZK} &= \left((r-1)\overline{a}_{n-2}\overline{a}_{n-3}...\overline{a}_1\overline{a}_0\right)+1 && \text{für eine n-stellige negative Zahl} \end{aligned} \tag{2}$$

Wobei mit $\overline{a}_i = r-1-a_i$ gemeint ist. Durch diese Darstellung lassen sich Zahlen im Bereich von $-r^{n-1} \le V\left(A_{ZK}\right) < r^{n-1}$ darstellen, wobei der Zahlenwert wie folgt berechnet wird:

$$V\left(A_{ZK}\right) = -a_{n-1}r^{n-1} + \sum_{i=0}^{n-2} a_i r^i \tag{3}$$

Der Vorteil dieser Zahlendarstellung ist, dass positive und negative Zahlen nicht gesondert behandelt werden müssen.

3 IMPLEMENTATION

Die neue Bibliothek wurde komplett in Standard VHDL implementiert. Es wurde der VHDL Standard IEEE 1076-1987 [9] gewählt, um auch mit älteren Simulatoren kompatibel zu sein. Der hierarchische Aufbau der Bibliothek ist in Abbildung 1 dargestellt.
In der Bibliothek wird durch die Konstante *digit_size* die Bitbreite pro Ziffer (*w*) festgelegt. Weiterhin ist eine neuer Typ *digit* mit zugehörigem *digit_vector* definiert worden. Die Benutzung der Bibliothek ist einfach, da innerhalb der Bibliothek hauptsächlich überladene Operatoren und Funktionen zum Einsatz kommen.

3.1 mp_logic

Diese unterste Schicht stellt die Schnittstelle zu den VHDL Standard Bibliotheken her. Hier werden die benötigten Konstanten und Typen definiert. Dabei legt die Konstante *digit_size* die Ziffernbreite für den Typ *digit* fest. Mit diesem neuen Typ werden dann die Vektoren *digit_vector* und *udigit_vector* erzeugt. Wobei *udigit_vector* ausschließlich für das Rechnen mit positiven Zahlen geeignet ist, während *digit_vector* auch für das Rechen mit negativen Zahlen geeignet ist. Dazu werden die Zahlen in *digit_vector* in Zweierkomplementdarstellung abgelegt. Somit kann das Vorzeichen der Zahl sehr einfach an der höchstwertigsten Ziffer im höchstwertigsten Bit abgelesen

werden. Weiterhin sind in dieser Schicht die logischen Funktionen *and, nand, or, nor, xor, xnor* und *not* implementiert. Um Zahlen von und in *std_logic_vector* zu konvertieren gibt es die Funktionen *To_DigitVector, To_UdigitVector* und *To_StdLogicVector*. Zusätzlich wurden die Funktionen *To_DigitVector, To_UdigitVector* so überladen, dass auch Hex-Zahlen im Stringformat eingelesen werden können, die Funktion *To_String* ermöglicht das Umwandeln der *digit_vector* bzw. *udigit_vector* Zahlen in einen String. Somit ermöglichen diese Funktionen ein einfaches Einlesen von Testvektoren aus einer Textdatei bzw. das Schreiben der Ergebnisse in eine log-Datei.

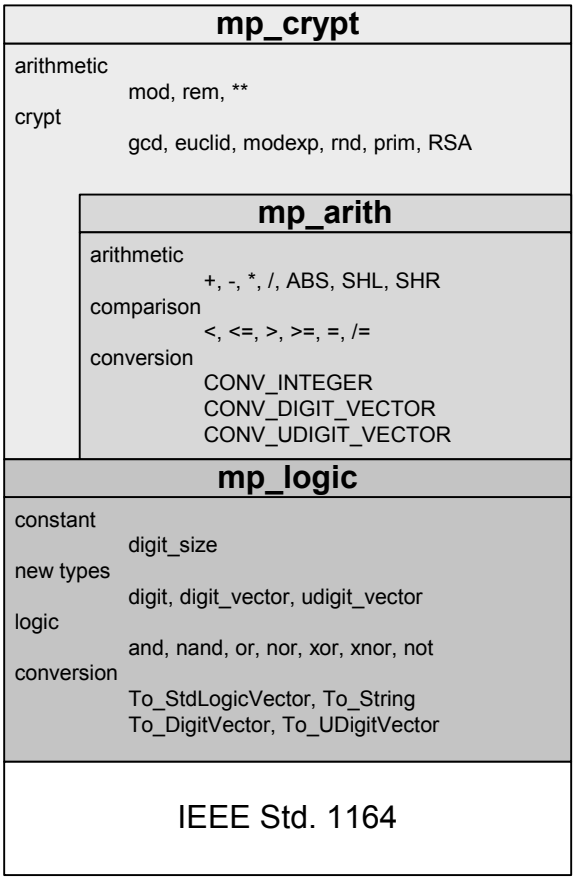

Abbildung 1: Aufbau der VHDL-Bibliothek

3.2 mp_arith

Die zweite Schicht wird durch *mp_arith* dargestellt. In dieser Ebene sind die mathematischen Grundfunktionen Addition, Subtraktion, Multiplikation und Division implementiert. Dabei stellt die Division eine Besonderheit dar, denn diese Funktion ist in den Standard VHLD Bibliotheken für std_logic_vector Zahlen nicht vorhanden. Zusätzlich zu den mathematischen Grundfunktionen sind in dieser Schicht noch die folgenden Funktionen zu finden: Absolutwert, Linksschieben und Rechtsschieben. Um zwei Vektoren miteinander vergleichen zu können, sind die Vergleichsfunktionen kleiner (<), kleiner gleich (<=), größer (>), größer gleich (>=), gleich (=) und ungleich (/=) implementiert worden. Damit alle Funktionen in dieser Schicht auch mit unterschiedlich großen Vektoren arbeiten können, wird zuerst eine Längen- und Typenanpassung der beiden Zahlen mit den Funktionen *CONV_INTEGER, CONV_DIGIT_VECTOR, CONV_UDIGIT_VECTOR* vorgenommen. Somit ist die Länge des Ergebnisvektors automatisch

genauso lang wie der größte Eingangsvektor, mit der die Funktion aufgerufen wird. Dieses Schema ist in Abbildung 2 dargestellt.

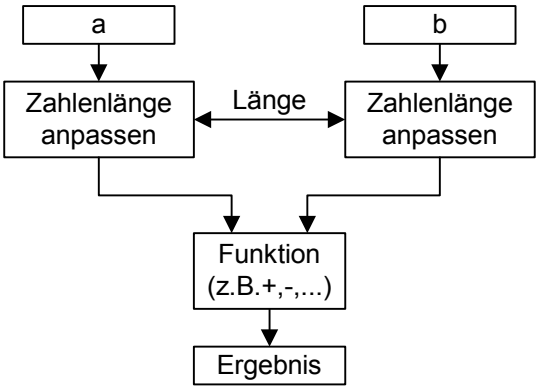

Abbildung 2: Längenanpassung für die Funktionen

3.3 mp_crypt

Dies ist die oberste Schicht in der Hierarchie der Krypto-VHDL-Bibliothek. Hier sind einige erweiterte mathematische Funktionen wie rem, mod und Exponentiation enthalten, sowie die eigentlichen kryptographischen Funktionen, die im folgendem genauer beschrieben werden:

rem: Diese Funktion liefert den Rest (remainder) *r* bei einer Division von zwei Zahlen *a* und *b*. Dabei wird das Vorzeichen von *r* durch das Vorzeichen von *a* bestimmt.

$$r = a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b \qquad \text{mit } 0 \le r < b \tag{4}$$

mod: Mit dieser Funktion wird der Modulus berechnet. Wobei sie das selbe Ergebnis liefert wie *rem*, solange die beiden Zahle *a* und *b* positiv sind.

$$a \bmod b = \begin{cases} a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b & \text{Wenn Vorzeichen(a) = Vorzeichen(b)} \\ a + \left\lceil \left| \frac{a}{b} \right| \right\rceil \cdot b & \text{ansonsten} \end{cases} \tag{5}$$

gcd: Hiermit wird der größte gemeinsame Teiler (**g**reates **c**ommon **d**evisior) von zwei Zahlen *a* und *b* berechnet. Für die Berechnung wird der Euklidische Algorithmus genutzt:

Algorithmus 2: Berechnung des größten gemeinsamen Teilers (gcd)
EINGABE: zwei positive natürliche Zahle a und b mit a ≥ b
AUSGABE: der größte gemeinsame Teiler von a und b
1. Solange b ≠ 0 ist, mache folgendes:
   1.1. Setze r ← a mod b, a ← b, b ← r
2. Gebe (a) zurück

Der größte gemeinsame Teiler wird z.B. dafür benötigt, um festzustellen, ob zwei Zahlen relativ prim zueinander sind.

euclid: Der erweiterte Euklidische Algorithmus wird genutzt, um das Modular Inverse *a* einer Zahl *x* und des Modulus *n* zu berechnen, $a^{-1} \equiv x \bmod n$.

Algorithmus 3: Erweiterter Euklidischer Algorithmus
EINGABE: Zahl x, Modulus n
AUSGABE: $a^{-1} \equiv x \bmod n$
1. $s \leftarrow n, v \leftarrow 0, r \leftarrow x, u \leftarrow 1$
2. Wiederhole
   2.1. $q \leftarrow \lfloor s / r \rfloor$
   2.2. $tmp \leftarrow s - q * r, s \leftarrow r, r \leftarrow tmp$
   2.3. $tmp \leftarrow v - q * r, v \leftarrow u, u \leftarrow tmp$
3. solange, bis $(r = 0)$
4. Gebe $(a^{-1} = v)$ zurück

modexp: Mit dieser Funktion wird die modulare Exponentiation berechnet, $x = a^e \bmod n$. Dazu wurde der Algorithmus von Knuth genutzt [10].

Algorithmus 4: Berechnung der modularen Exponentiation
EINGABE: a, der Exponent e und der Modulus n
AUSGABE: $x = a^e \bmod n$
1. $x \leftarrow 1$, l ← Anzahl Bitstellen von e
2. Für i von (l-1) bis 0 mache folgendes
   2.1. $x \leftarrow x^2 \bmod n$
   2.2. Wenn $(e_i = 1)$ dann $x \leftarrow x * a \bmod n$
3. Gebe (x) zurück

Damit kann die Ver- bzw. Entschlüsselung für den RSA-Algorithmus berechnet werden. Aber es sind auch andere Algorithmen, die auf der modularen Exponentiation beruhen, wie z.B. ElGamal, möglich.

rnd: Diese Funktion liefert eine Zufallszahl zurück. Dazu wird die Zahl mit Hilfe eines linearen Kongruenzgenerators nach dem Schema $x_n = a \cdot x_{n-1} + b \bmod n$ erzeugt [11].

prim: Hiermit kann eine Zahl *a* darauf getestet werden, ob sie prim ist. In der Kryptographie sind diese Zahlen typischerweise mehrere hundert Bit lang. Um so lange Primzahlen zu erhalten wird eine ungerade Zufallszahl mit einem Primzahltest, wie z.B. dem Miller-Rabin Test [8], getestet.

RSA: Mit dieser Funktion werden die Zahlen *p, q, n, e* und *d* erzeugt, welche für die Ver- und Entschlüsselung mit dem RSA-Algorithmus benötigt werden. Dabei ist *n* der Systemmodulus der aus dem Produkt der beiden Primzahlen p und q berechnet wird, $n = p * q$. Die Zahl *e* stellt den öffentlichen Schlüssel dar und ist eine Zufallszahl, die relativ prim zur Eulerschen Phi Funktion $\varphi(n)$ ist, $\gcd(e, \varphi(n)) = 1$. Der private Schlüssel *d* wird mit Hilfe des Euklidischen Algorithmus berechnet, so dass $e * d \equiv 1 \bmod \varphi(n)$ ist.

4 PERFORMANCE MESSUNG

Für die einzelnen Funktionen, die in der VHDL Bibliothek enthalten sind, gibt es oftmals mehrere Möglichkeiten diese zu implementieren. Dabei haben die unterschiedlichen Implementationen unterschiedlich lange Simulationszeiten. Um die schnellste Implementation herauszufinden ist eine Performance Messung notwendig.
Für dieser Messung wurde eine SUN ULTRA 1 mit Solaris 2.5 und dem Synopsys VSS Simulator 1998.08 verwendet. Der VSS Simulator ist ein ereignisgesteuerter VHDL Simulator. Da es in VHDL keine Möglichkeit gibt, die Systemzeit zu messen, wurde auf die C-Schnittstelle des Simulators (C-Language Interface – CLI [5]) zurückgegriffen. Für die eigentliche Zeitmessung wurde dann in einem C-Programm auf die im Betriebssystem vorhandenen Zeit-Funktionen zurückgegriffen. Um die Zeitmessung relativ unabhängig von den Zeiten für die Task Switches zu machen, wurde jeder Test mit 10000 Wiederholungen durchgeführt.
Am Beispiel der Addition wird im Anschluss eine derartige Optimierung aufgezeigt werden. Nach einigen Test hat sich ergeben, dass die Implementation, wie sie in Algorithmus 1 dargestellt ist, selbst mit der Optimierung der Zeilen 2.1. und 2.2., nicht die schnellste Möglichkeit ist. Als schnellste Implementierung stellte sich die Addition mit Benutzung der IEEE Bibliothek heraus:

Algorithmus 5: Addition mit IEEE Funktion
EINGABE: x, y in digit_vector Darstellung
AUSGABE: z = x + y
1. Umwandlung von x und y in std_logic_vector Darstellung
2. IEEE-Addition z = x + y
3. Umwandlung von z in digit_vector Darstellung
4. Gebe (z) zurück

So konnte eine Performancegewinn von 30% erreicht werden. Um die Abhängigkeit zwischen der Bitlänge *n* und der Ziffernbreite *w* von Zahlen zu ermitteln, wurden zwei Tests durchgeführt. Zu erst wurde ein Test mit konstantem *n* und variablem *w* durchgeführt und dann ein Test mit konstantem *w* und variablem *n*.

4.1 Konstantes *n*, variables *w*

Bei diesem Test wird *n* konstant auf 1024 Bit gesetzt und die Ziffernbreite *w* wird variiert. Die Ergebnisse dieses Tests ist in Abbildung 3 zu sehen. Dabei lässt sich erkennen, dass ab dem Erreichen der Registerbreite des ausführenden Prozessors kaum noch eine signifikante Performancesteigerung zu verzeichnen ist.

4.2 Konstantes *w*, variables *n*

Im zweiten Test wird die Ziffernbreite *w* konstant gehalten (16 bzw. 32 Bit) und die Bitlänge *n* der Zahlen variiert. Dabei konnte festgestellt werden, dass die Rechenzeit linear mit der Zahlenlänge steigt. Die Ergebnisse für die Addition sind in Tabelle 1 dargestellt. Es ist gut zu erkennen, dass bei einer Verdoppelung der Zahlenlänge die Rechenzeit ebenfalls auf etwa das Doppelte steigt.

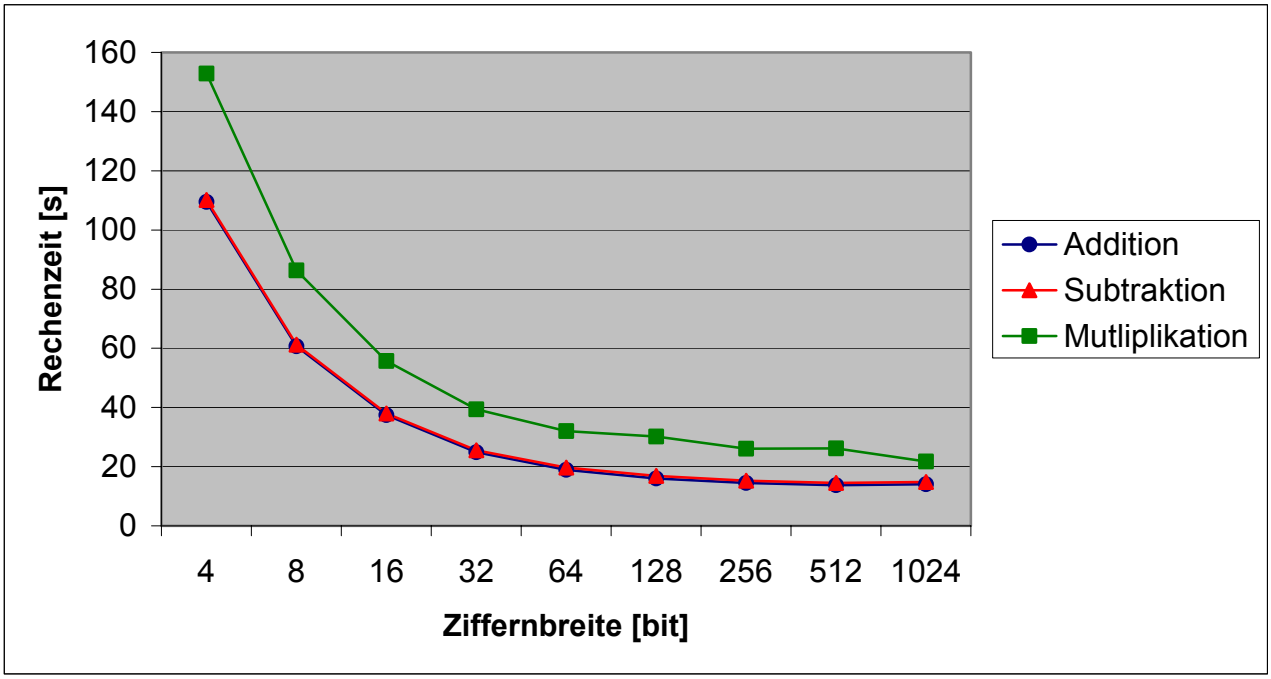

Abbildung 3: Rechenzeit bei konstantem *n* und variablem *w*

N [Bit]	256	512	1024	2048	4096
W=16	11,06	19,29	38,8	75,95	166,16
W=32	7,73	13,02	26,38	49,95	104,49

Tabelle 1 : Rechenzeit bei konstantem *w* und variablem *n* Sekunden

5 VALIDIERUNG DER BIBLIOTHEK

Die VHDL Bibliothek wurde auf ihre korrekte Funktion hin gegen die FreeLIP-C-Bibliothek [12] getestet. Dazu wurden diese Bibliothek so erweitert, dass sie zufällige Eingangsvektoren über die C-Schnittstelle des Simulators an die VHDL-Bibliothek weiterreicht und diese dann mit den zugehörigen Ergebnissen innerhalb der Testbench verglichen werden. So konnte ein Test über einen längeren Zeitraum durchgeführt werden, bei dem keine Fehler aufgetreten sind.

6 ZUSAMMENFASSUNG

In diesem Beitrag wurde eine neue VHDL-Bibliothek für die Simulation und Verifikation kryptographischer Prozessoren vorgestellt. Dazu wurde der Aufbau der Bibliothek und die Performance Messung mit der Optimierung einzelner Funktionen beschrieben. Der große Vorteil dieser Bibliothek besteht darin, dass sie in Standard VHDL implementiert wurde und so auf jeden VHDL Simulator zu Einsatz kommen kann.
Die Bibliothek wurde bei der Entwicklung eines ISDN Verschlüsselungsgerätes erfolgreich für den Test des kryptographischen Koprozessor eingesetzt [13].

LITERATUR

[1] Siemens AG, "TopSec GSM – das Crypto-Handy für vertrauliche Telefongespräche," 2001.

[2] IEEE Std. 1164-1993, "IEEE standard multivalue logic system for VHDL model interoperability (Stdlogic1164)," 1993.

[3] R. L. Rivest, A. Shamir, L. Adleman, "A Method of obtaining digital signature and public key cryptosystems," Comm. of ACM, Vol.21, No.2, pp.120-146, Februar 1978.

[4] A. K. Lenstra, E. R. Verheul, "Selecting Cryptographic Key Sizes," 3rd workshop on Elliptic Curve Cryptography (ECC '99), 1999.

[5] Synopsys C-Language-Interface, Version 1998.08

[6] Accolade Design Automation, http://www.acc-eda.com/

[7] H. Ploog, D. Timmermann, "FPGA based architecture evaluation of cryptographic coprocessors for smart cards," Symposium on Field-Programmable Custom Computing Machines FCCM'98, USA, April 1998.

[8] A.J. Menezes, P.C. von Oorschot, S.A Vanstone, "Handbook of Applied Cryptography," CRC Press, 1997.

[9] IEEE Std. 1076-1987, "VHDL Language Reference Manual," 1987

[10] D. Knuth, "The Art of Computer Programming: Volume 2, Seminumerical Algorithms," 2nd edition, Addison-Wesley, 1981.

[11] S. K. Park, K. W. Miller, "Random Number Generators: Good Ones Are Hard to Find," Communications of the ACM, V 31, N 10, Oct. 1988.

[12] A. Lenstra, ftp://ftp.ox.ac.uk/pub/math/freelip/freelip_1.0.tar.gz

[13] H. Ploog, M. Schmalisch, D. Timmermann, "Security Upgrade of Existing ISDN Devices by Using Reconfigurable Logic," Field-Programmable Logic and Applications - FPL 2000, LNCS 1896, Springer Verlag, Heidelberg, Villach, August 2000.