fhwedel
UNIVERSITY OF APPLIED SCIENCES

Reconfigurable Computing

Eliptic Curve Digital Signature Algorithm

FPGA-Implementierung und Vergleich

Eingereicht am:

24. März 2018

Lennart Bublies
inf100434@fh-wedel.de

Leander Schulz
inf102143@fh-wedel.de

Betreut von:
Prof. Dr. S. Sawitzki

Fachhochschule Wedel
Feldstraße 143
22880 Wedel
E-mail: saw@fh-wedel.de

Inhaltsverzeichnis

1

Einleitung

Mit wachsender Digitalisierung wächst auch die Sorge, was mit den eigenen Daten passiert und ob diese nach der Übertragung über ein Medium nicht durch andere manipuliert worden sind. Um die Sicherheit bzw. Authentizität der Daten zu gewährleisten, werden die Daten mit Hilfe eines Passwortes, dem Schlüssel, signiert, sodass ein Empfänger, der ebenfalls den Schlüssel kennt, die Daten verifizieren kann. Der sich dahinter verbergende Algorithmus wird Digitaler Signatur Algorithmus (engl. Digital Signature Algorithm, DSA) genannt.

Will ein Angreifer trotz Signatur eine Nachricht manipulieren, kann dieser über eine Brute-Force-Attacke versuchen, den Schlüssel durch ein Testen jeder erdenklichen Kombination zu erraten. Gelingt dies, kann nach der Manipulation eine neue gültige Signatur erstellt werden. Um diesem Angriffsscenario entgegen zu wirken, werden lange Schlüssel verwendet, sodass eine solche Attacke nicht in akzeptabler Zeit zu bewerkstelligen ist. Dies hat jedoch zur Folge, dass mit steigender Schlüssellänge auch die Zeit steigt, die für das Erstellen der Signatar bzw. die Verifizierung benötigt wird.

Eine mögliche Lösung zur Reduzierung der Schlüssellänge und damit der Laufzeit besteht im Einsatz optimierter Verfahren wie die elliptischen Kurven, die im Gegensatz zu Verfahren wie RSA (engl. Random Se quential Adsorption), deutlich geringe Schlüssellängen benötigen. Diese Arbeit widmet sich den zuvor erwähnten elliptischen Kurven mit dem Ziel, eine effiziente Hardware-Implementierung zu entwickeln, die eine klassische Software-Variante übertrifft.

2

Zielsetzung & Abgrenzung

Das Ziel dieser Arbeit ist die Umsetzung eines kryptografisch sicheren Verfahrens zum digitalen Signieren einer Nachricht und der Verifikation dieser Signatur. Dazu dient der *Elliptic Curve Digital Signature Algorithm (ECDSA)* als Basis, um eine ressourceneffiziente Hardware-Implementierung unter Verwendung eines FPGA-Bausteins[1] zu implementieren. Als Entwicklerboard kommt das Altera DE2 Board zum Einsatz

Die Kommunikation mit dem FPGA soll über eine serielle Schnittstelle erfolgen. Dabei gibt es die zwei Modi Signieren und Verifizieren. Beim Signieren wird die zu signierende Nachricht gesendet und als Antwort eine gültige Signatur empfangen. Beim Verifizieren werden die zu überprüfende Signatur und die Originalnachricht gesendet. Als Antwort wird über einen booleschen Wert signalisiert, ob die Signatur gültig ist oder nicht. Dabei liegt der Fokus auf der VHDL-Implementierung der mathematischen Funktionen und der Lauffähigkeit des Gesamtsystems. Auf Teilkomponenten wie die Generierung einer echten Zufallszahl oder die effiziente Bildung eines Hashes der Nachricht, wie im Algorithmus beschrieben, wird verzichtet. Stattdessen werden die Daten bereits als Hash übertragen und eine Konstante als Zufallszahl verwendet. An dieser Stelle sei erwähnt, dass das Verwenden einer Konstante als Zufallszahl kryptografisch unsicher ist und in produktiven Umgebungen zwingend vermieden werden sollte, um Angriffe wie den berühmten Playstation 3 Hack [2] zu verhinden. Die Vereinfachung ist zum einen der fehlenden Hardware zur Generierung einer echten Zufallszahl geschuldet und um zum anderen den Fokus auf die Implementierung der Kernfunktionen zu lenken.

Abgeschlossen wird diese Arbeit mit einer Gegenüberstellung der Ergebnisse der FPGA-Implementierung mit einer C-Implementierung, die im Rahmen eines anderen Projektes [Kew16] entstanden ist.

[1] Field Programmable Gate Array (FPGA): https://de.wikipedia.org/wiki/Field_Programmable_Gate_Array

[2] Playstation 3 Hack: https://www.edn.com/design/consumer/4368066/The-Sony-PlayStation-3-hack-deciphered-what-consumer-electronics-designers-can-learn-from-the-failure-to-protect-a-billion-dollar-product-ecosystem

3

Grundlagen

3.1 Gruppen und Endliche Körper

Eine abelsche Gruppe im mathematischen Sinn ist ein Tupel $(G, \circ)$ aus einer nicht-leeren Menge G und einer darauf definierten inneren Verknüpfung $\circ$ (vgl. [Put14], S. 11f). Hierbei entspricht die innere Verknüpfung einer Abbildung

$$\circ : X \times X \to X$$

die jedem Paar (x_1, x_2) aus Elementen der Menge X ein Element $x_1 \circ x_2$ zuordnet. Außerdem sind folgende Bedingungen auf der Struktur einer Gruppe definiert:

1. Menge G ist abgeschlossen, d.h. für alle $a, b \in G$ gilt, dass auch jede Verknüpfung $a \circ b$ ein Element von G ist.
2. Assoziativgesetz: $\forall a, b, c \in G : (a \circ b) \circ c = a \circ (b \circ c)$
3. Kommutativgesetz: $\forall a, b \in G : a \circ b = b \circ a$
4. Neutrales Element: $\forall a \in G : a \circ e = e \circ a = a$
5. Inverses Element: $\forall a \in G : a \circ a^{-1} = a^{-1} \circ a = e$

Ein Beispiel für eine so definierte abelsche Gruppe entspricht der Menge der ganzen Zahlen $\mathbb{Z}$ mit der Addition als Verknüpfung $(\mathbb{Z}, +)$. Die Anzahl der Elemente einer Gruppe wird als Ordnung bezeichnet, welche für die ganzen Zahlen unendlich ist. In der Kryptographie kommen jedoch stets Gruppen mit einer endlichen Anzahl von Elementen zum Einsatz, bei denen also die Ordnung der Gruppe beschränkt ist.

Eine besondere Form endlicher Gruppen bilden zyklische Gruppen, die ein Element g besitzen, aus dem alle anderen Elemente der Gruppe durch Verknüpfungen $\circ$ erzeugt werden können.

Das Element g ist das sog. Generatorelement der zyklischen Gruppe $(G', \circ)$.

Ein Körper ist eine Erweiterung von abelschen Gruppen und als Tripel $(K, +, \cdot)$ definiert, welche zu einer nicht-leeren Menge K die inneren Verknüpfungen Addition und Multiplikation beschreibt. Folgende Bedingungen werden dabei erfüllt (vgl. [Put14]):

1. Die Menge K bildet durch die additive Verknüpfung $+$ eine abelsche Gruppe mit 0 als neutralem Element
2. Die Menge $K \setminus \{0\}$, d. h. K ohne das Element 0, bildet durch die multiplikative Verknüpfung $\cdot$ ebenfalls eine abelsche Gruppe mit 1 als neutralem Element
3. Distributivgesetz: $\forall a, b, c \in K : c \cdot (a + b) = c \cdot a + c \cdot b$

Unter einem Endlichen Körper oder Galoiskörper versteht man einen Körper mit endlich vielen Elementen, der oft auch als Restklassenkörper bezeichnet wird. Ein Beispiel dafür sind Primkörper mit Primzahl p und den Verknüpfungen *Addition modulo p* und *Multiplikation modulo p*: $\mathbb{Z}_p = \{0, 1, 2, 3, \cdots, p-1\}$.

Galoiskörper werden mit $GF(p)$ abgekürzt, wobei p eine Primzahl sein muss und für die Anzahl der Elemente im Feld steht. $GF(p)$ repräsentiert damit einen Körper der Restklasse ganzer Zahlen modulo p. Durch die mathematischen Eigenschaften dieser Struktur eignen sie sich besonders für den Einsatz in der Kryptographie.

3.2 Elliptische Kurven

Eine elliptische Kurve E über einem Körper K ist eine nicht-singuläre[1] Kurve (vgl. Abb. 3.1), definiert als Menge aller Punkte (x, y) mit $x, y \in K$, für die gilt $F(x, y) = 0$ zusammen mit einem "Punkt in der Unendlichkeit" O (vgl. [Gre05], S.25).

Eine elliptische Kurve über ein endliches Feld F_p kann mit Hilfe der Weierstraß-Gleichung in Normalform wie folgt definiert werden:

$$E(K) : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

[1] Nicht-singulär: keine Knoten, keine Spitzen, keine "Einsiedler" (isolierte Punkte)

Die Parameter $a_1 \dots a_6$ legen fest, welche Form die elliptische Kurve annimmt und sind Elemente des Feldes F_p. Sie werden durch die Domain-Parameter festgelegt.

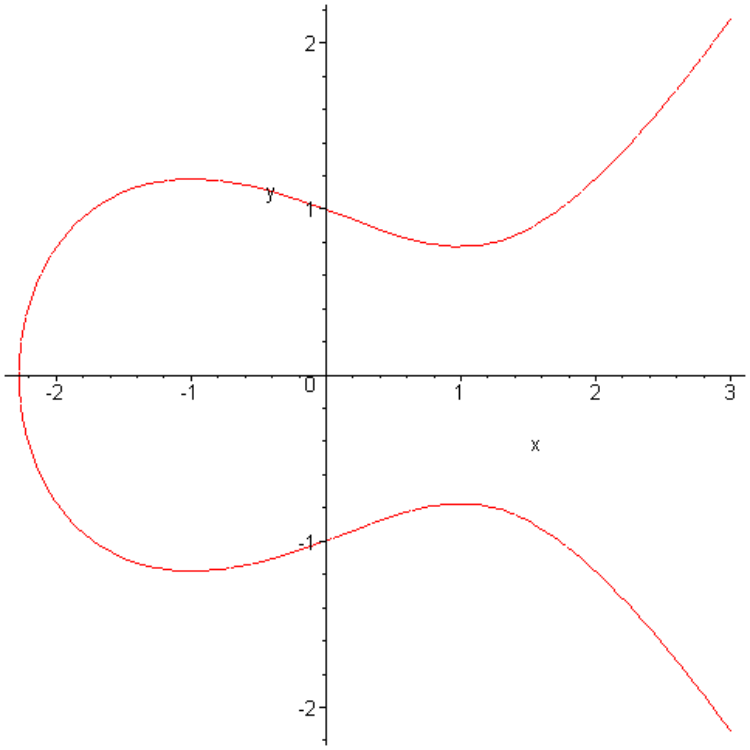


Abbildung 3.1: Graph einer Elliptischen Kurve

Über eine Reihe von Vereinfachungen lässt sich die Weierstraß-Gleichung auf die folgende Formel reduzieren:

$$y^2 \bmod p = x^3 + ax + b \bmod p$$

Damit eine elliptische Formel in der Kryptographie eingesetzt werden kann, muss zusätzlich die nachfolgende Bedingung gelten. Sie folgt aus der nicht-singulären Eigenschaft von E, also Diskriminante $\neq 0$ (Herleitung und Beweis siehe [Gre05], S. 28ff):

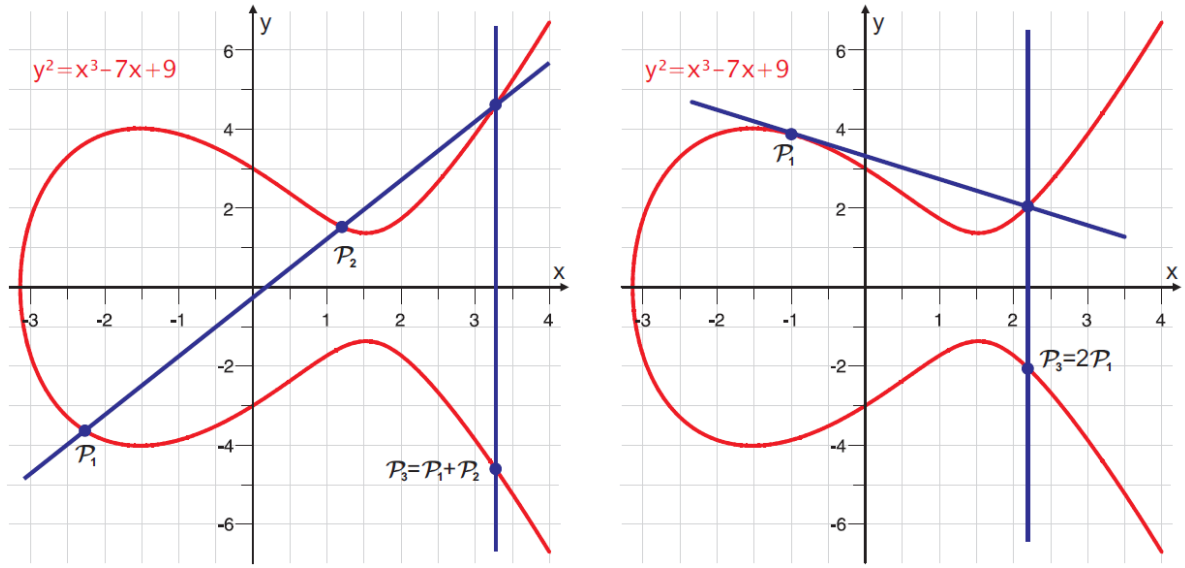
$$4a^3 + 27b^2 \bmod p \neq 0, \text{ mit } a, b \in K$$

Entsprechend wie bei endlichen Körpern ist die Ordnung auf einer zyklischen Untergruppe U_P von $E(F_P)$ definiert (vgl. [Gre05], S. 35):

$$U_P := \{kP | k \in \mathbb{Z}\}\ k_0 P = P + P + \dots + P = O, \text{ mit } k_0 > 0$$

3.2.1 ECC-Operationen

Auf elliptischen Kurven sind eine Reihe von Operationen definiert. Dazu zählen die Punktaddition, Punktmultiplikation und die Punktdopplung als Spezialfall der Punktaddition.



(a) Punktaddition (b) Punktverdopplung

Abbildung 3.2: Arithmetische Operationen auf einer elliptischen Kurve

Die als Punktaddition bezeichnete arithmetische Operation verknüpft zwei verschiedene Punkte ($P \neq Q$) auf der Kurve miteinander.

$$
\begin{aligned}
R &= P + Q \\
s &= \tfrac{p_y - q_y}{p_x - q_x} \mod p \\
r_x &= s^2 - p_x - q_x \mod p \\
r_y &= s(p_x - r_x) - p_y \mod p
\end{aligned}
$$

Wenn die beiden Punkte gleich sind (also $R = P_1 + P_2 = 2P_1$), wird diese Operation Punktverdopplung genannt (vgl. [Put14], S. 16f).

$$
\begin{aligned}
R &= P + P \\
s &= \tfrac{3{p_x}^2 + a}{2p_y} \mod p \\
r_x &= s^2 - 2p_x \mod p \\
r_y &= s(p_x - r_x) - p_y \mod p
\end{aligned}
$$

Beide Operationen lassen sich mittels Sekanten- bzw. Tangentenmethode auf einer elliptischen Kurve darstellen (s. Abb. 3.2).

Die Punkt-Multiplikation führt eine k-fache Punktaddition durch.

$R = kA \mod p$

3.2.2 Koblitz-Kurven

Eine besondere Form der elliptischen Kurven sind `Koblitz`-Kurven (engl. Anomalous Binary Curves) über das endliche Feld F_{2^m}, deren Kurvengleichung ausschließlich die Koeffizienten "0" und "1" enthalten. In kryptographischen Protokollen werden häufig Koblitz-Kurven der folgenden Formen verwendet (vgl. [HMV04], Kap. 3.4):

$$E_0 : y^2 + xy = x^3 + 1$$
$$E_1 : y^2 + xy = x^3 + x^2 + 1$$

Der große Vorteil dieser Kurven ist, dass Algorithmen zur Punktmultiplikation prinzipiell ohne Punktverdopplung auskommen können (s. [HMV04], S. 114). Darüberhinaus eignen sich Koblitz-Kurven besonders für den Einsatz in Hardwaresystemen, da für sie allgemein hocheffiziente Algorithmen für Punktarithmetik existieren, wohingegen sich klassische Kurven F_P eher für Softwaresysteme eignen. In der Praxis wurden bisher keine Sicherheitsunterschiede festgestellt.

3.3 Das Diskrete Logarithmus-Problem

Das Problem des diskreten Logarithmus in endlichen Körpern besteht darin, dass die Lösung eines solchen Logarithmus viel schwieriger ist, als die Umkehrfunktion, also die diskrete Exponentialfunktion. Die Sicherheit von auf elliptischen Kurven basierenden kryptographischen Verfahren beruht auf diesem mathematischen Problem.

Definition nach [Bau09] (S. 6):

- **Voraussetzungen:**
  Sei $(G, \cdot)$ eine multiplikative Gruppe, $x \in G$ ein Element der Ordnung n und $y \in \langle x \rangle$.

- **Problem:**
  Man berechnet a mit $(0 \leq a \leq n-1)$, sodass $x^a = y$.
  "a" wird diskreter Logarithmus von y zur Basis x genannt.

Umgangssprachlich wird eine solche Funktion *Einwegfunktion* genannt. Dabei entspricht der diskrete Logarithmus einer Funktion $f : M \implies N$, wenn für "fast alle" Bilder $y \in N$ kein Urbild $x \in M$ mit $f(x) = y$ effizient bestimmbar ist (vgl. [Pom08], S. 54).

Bezogen auf eine elliptische Kurve (engl. Elliptic Curve Discrete Logarithm Problem (ECDLP)) $E(GF(2^n))$ bedeutet das, dass die Skalarmultiplikation $Q = kP$ für eine natürliche Zahl k und einen Punkt P nach der Kurvengleichung sehr einfach zu berechnen ist (Q ist die $(k-1)$-fache Addition mit P), es aber *schwierig* ist, aus zwei vorgegebenen Punkten $Q, P \in E(GF(2^n))$ wieder die Zahl k zu ermitteln (vgl. [Put14], S. 17f).

3.4 Digitale Signaturen

Eine digitale Signatur nutzt ein asymmetrisches Verschlüsselungsverfahren, um eine schlüsselabhängige Prüfsumme eines Dokuments zu erzeugen (vgl. [Wol09]). Der Schlüssel zum Überprüfen einer Signatur ist öffentlich, während der Schlüssel zum Signieren geheim gehalten werden muss. Somit kann jeder ein digital "unterschriebenes" Dokument auf seine Echtheit überprüfen bzw. verifizieren.

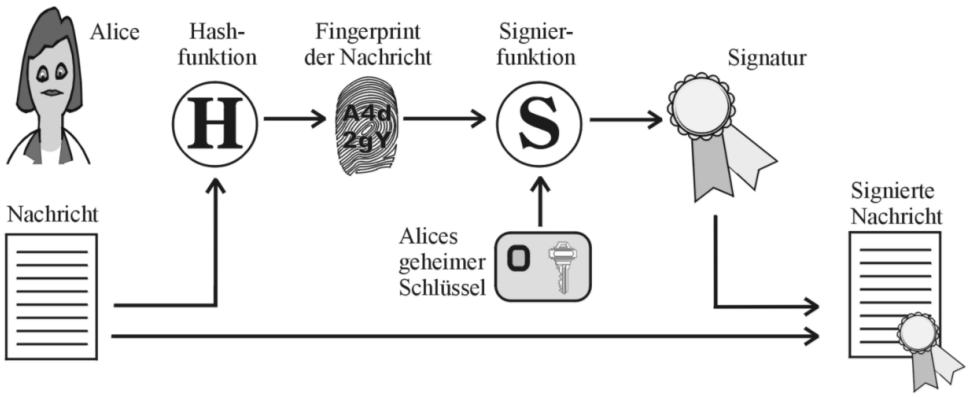


Abbildung 3.3: Erzeugung einer Digitalen Signatur

Abbildung 3.3 zeigt den Vorgang der Erzeugung einer Signatur. Die im Bild zu sehende "Signierfunktion" muss eine asymmetrische Verschlüsselung sein und entspricht dem in dieser Arbeit implementierten ECDSA-Algorithmus.

3.5 Der ECDSA Algorithmus

3.5.1 Schlüsselgenerierung & Domain Parameter

Beim *Elliptic Curve Digital Signature Algorithm* (ECDSA) wird von einer Partei A ein Schlüssel generiert und damit die Signatur zu einer Nachricht erstellt. Eine andere Partei B nutzt den öffentlichen Schlüssel von A und verifiziert damit die Echtheit der Nachricht von A anhand der Signatur. Folgende Definition (vgl. [AS12]) berechnet die Schlüssel von A:

1. Wähle ein zufälliges d aus $[1, n-1]$.
2. Berechne $Q = dP$.
3. Der öffentliche Schlüssel von A entspricht Q; der private Schlüssel ist d.

Der private Schlüssel d ist hierbei eine positive ganze Zahl. Der Parameter n entspricht der Ordnung des Basis-Punkts P. Die Domain Parameter werden allgemein angegeben als $D = (q, FR, S, a, b, P, n, h)$ und werden definiert als (vgl. [HMV04], S. 172):

- q: Ordnung der Kurve.
- FR: Repräsentation der Elemente in F_q.
- S: *Seed*, falls die Kurve zufällig generiert ist.
- $a, b \in F_q$: Koeffizienten, die die elliptische Kurve beschreiben.
- $P = (x_P, y_P) \in E(F_q)$: Basis-Punkt der elliptischen Kurve.
- n: Ordnung von P.
- h: *Ko-Faktor* $h = \#E(F_q)/n$.

3.5.2 Signatur

Erstellen der Signatur (vgl. [AS12]) : (A)

1. Zufallszahl k im Bereich $[1, n-1]$ bestimmen.
2. Berechne $kG = (x_1, y_1)$ mit $r = x_1 \pmod{n}$.
3. Wenn $r = 0$ dann zurück zu 1.

4. Berechne $k^{-1} \pmod{n}$.
5. Berechne $s = k^{-1}(SHA-1(m) + dr) \pmod{n}$.
6. Wenn $s = 0$ dann zurück zu 1.
7. Sende m und (r, s) zu B

3.5.3 Verifikation

Verifizierung der Signatur (vgl. [AS12]) : (B)

1. Überprüfe das r und s Zahlen in $[1, n-1]$ sind.
2. Berechne $e = SHA-1(m)$.
3. Berechne $w = s^{-1} \pmod{n}$.
4. Berechne $u_1 = e * w \pmod{n}$ und $u_2 = r * w \pmod{n}$.
5. Berechne $u_1 P + u_2 Q = (x1, y1)$ und $v = x_1 \pmod{n}$.
6. Wenn $s = 0$ dann zurück zu 1.
7. Akzeptiere die Signatur $v = r$ gilt.

3.6 UART-Schnittstelle

Bei der UART-Kommunikation (engl. Universal Asynchronous Receiver Transmitter, UART) werden Daten über zwei Pins ausgetauscht. Dabei dient ein Pin dem Empfangen (RX-Pin) und der andere dem Senden (TX-Pin). Zur Übertragung der Daten zwischen zwei UART-Schnittstellen werden die Pins über Kreuz verbunden (vgl. [Bas16]). Üblicherweise übernimmt ein UART-Hardwaremodul auch die Transformation paralleler Daten in einen sequentiellen Datenstrom und umgekehrt.

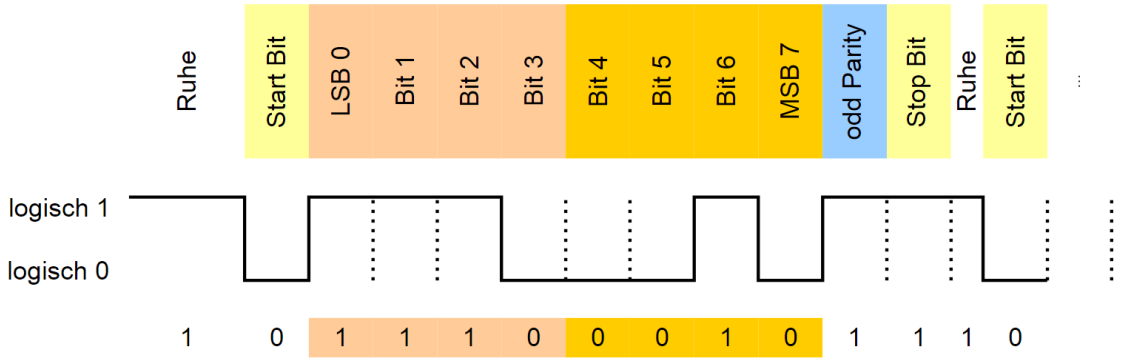


Abbildung 3.4: Signalübertragung über das UART-Protokoll

Die Daten werden zu Paketen von meist 8 Bit übertragen, wobei die Sende-Reihenfolge in der Regel vom geringwertigstem Bit (engl. Least Significant Bit, LSB) zum höchstwertigstem Bit (engl. Most Significant Bit, MSB) versendet wird (vgl. Abb. 3.4). Wenn bei einer UART-Verbindung keine Daten übertragen werden, liegt ein “High”-Pegel an (logisch “1”). Die Übertragung beginnt mit einem Start-Bit (logisch “0”) für einen UART-Taktzyklus, wonach die Daten-Bits folgen. Je nach Implementierung kann eine Parität ergänzt werden, die kennzeichnet, ob das Datenpaket eine gerade oder ungerade Anzahl “1” enthält. Ein Stop-Bit beendet die Transmission eines Datenpaketes. Ein *UART-Taktzyklus* beschreibt, wie lange die Spannung eines einzelnen Symbols anliegt. Die entsprechende Einheit wird in *Baud* angegeben und entspricht $1 * s^{-1}$. Bei einer Übertragungsrate von 9600 Baud liegt ein Bit also für $1/9600s = 104,167\mu s$ am Ausgang Tx an.

4

Implementierung

4.1 Architektur & Moduldesign

Die Architektur der VHDL-Implementierung verwendet einen modularen Ansatz. Die zentrale Komponente wird durch den Controller repräsentiert, der den Zustandsautomat, die ECC-, sowie die arithmetischen Operationen beinhaltet und damit die ECDSA-Funktionen (Signieren, Verifizieren) implementiert. Abbildung 4.1 zeigt den grundlegenden Aufbau des Systems.

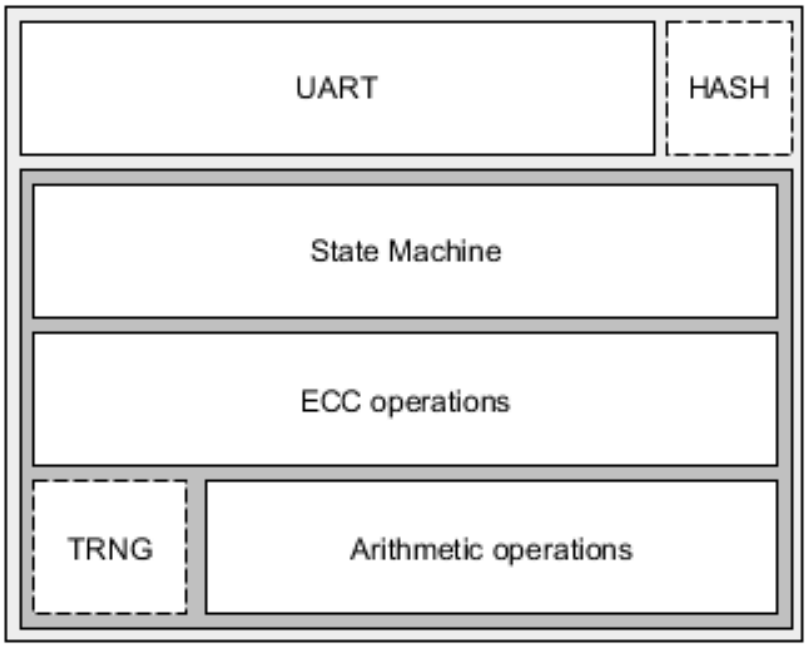


Abbildung 4.1: Modulübersicht der FPGA-Hardware. Hauptbestandteil ist der Controller, der den Zustandsautomat, die ECC-, sowie die arithmetischen Operationen beinhaltet. Die Datenübertragung findet über die UART-Schnittstelle statt. Module zum Hashen bzw. zur Zufallszahlengenerierung sind optional und lediglich zur Vollständigkeit erwähnt

Die Umschaltung zwischen den verschiedenen Algorithmen findet über einen Zustandsautomat (engl. State Machine) statt. Die Hardware-Implementierung kann zwischen den Modi Signieren

und Verifizieren unterschieden. Die Umschaltung findet über die UART-Schnittstelle statt, indem der Datenstrom analysiert wird. Eine detaillierte Beschreibung ist in Abschnitt 4.5 zu finden. Je nach gewünschter Funktion, werden andere Parameter über UART empfangen bzw. zurückgeschickt. Eine Übersicht der Parameter beinhaltet Tabelle 4.1.

Funktion	**Typ**	**Parameter**
Signieren	Input	Nachricht m (163 Bit)
Signieren	Ouput	Signatur (r, s) (326 Bit)
Verifizieren	Input	Nachricht m (163 Bit)
Verifizieren	Input	Signatur (r, s) (326 Bit)
Verifizieren	Output	Ergebnis der Verifizierung (1 Bit)

Tabelle 4.1: Eingabe- und Ausgabedaten der VHDL-Implementierung

Wie bereits in Kapitel **??** erwähnt, wurde auf eine sicherheitsorientierte Implementierung des Algorithmus verzichtet, sodass anstatt eines "echten" Zufallszahlengenerators (engl. True Random Number Generator, TRNG), lediglich eine festgelegte Konstante zum Einsatz kommt. Hauptgrund für diese Entscheidung begründet sich mit der fehlenden Hardware zum Generieren einer sicheren Zufallszahl. Als eine weitere Vereinfachung wurde auf eine Einheit zum Generieren eines Hashes verzichtet, um den Fokus auf die Implementierung der ECC-Operationen zu lenken, ohne die Messergebnisse durch die HASH-Generierung zu verfälschen. Aus Gründen der Vollständigkeit sind beide Komponenten dennoch in Abbildung 4.1 zu finden.

4.2 Parameter

Die gesamte VHDL-Implementierung setzt ausschließlich auf VHDL-typische `Generics`. Diese werden über das globale Paket `tld_ecdsa_package` verwaltet. Neben den Parametern enthält das Paket globale Hilfsfunktionen, die zum Beispiel eine Matrix-Reduktion durchführen. Tabelle 4.2 zeigt eine Übersicht der wichtigsten Parameter.

Parameter	**Beschreibung**
M	Bitbreite des Schlüssels bzw. der Daten (z.B. 163)
N	Modolo-Parameter der Kurve (Bestandteil des ECDSA-Algorithmus)
P	Anzahl der Elemente im GF(2^M)
BAUD_RATE	Eingestellte Baud-Rate der UART-Schnittstelle

Tabelle 4.2: Parameter der VHDL-Implementierung

Durch den Einsatz der generischen Implementierung ist es möglich, Parameter nachträglich zu ändern, um beispielsweise eine andere Schlüssellänge oder Kurvenparameter zu verwenden.

Ein Nachteil dagegen ist, dass durch die generische Implementierung weniger Spielraum für Performanz-Optimierungen besteht. Da mit wachsender Leistungsfähigkeit heutiger Systeme aber auch die Anforderung an kryptografische steigt, wurde die nachträgliche Anpassbarkeit der Parameter als wichtiger angesehen.

4.3 Top-Level-Entität

Wie bereits erwähnt kommuniziert die Top-Level-Entität über eine serielle RS232-Schnittstelle mit einem Computer, um Daten auszutauschen. Dazu werden lediglich zwei Pins zur UART-Kommunikation, sowie ein globales 50Mhz-Takt- und Resetsignal benötigt. Abbildung 4.2 zeigt die genaue Zuordnung der Pins auf dem FPGA.

Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved	Current Strength
clk_i	Input	PIN_N2	2	B2_N1	3.3-V LVTTL (default)		24mA (default)
rst_i	Input	PIN_G26	5	B5_N0	3.3-V LVTTL (default)		24mA (default)
rst_led	Output	PIN_Y18	7	B7_N0	3.3-V LVTTL (default)		24mA (default)
uart_rx_i	Input	PIN_C25	5	B5_N0	3.3-V LVTTL (default)		24mA (default)
uart_wx_i	Output	PIN_B25	5	B5_N0	3.3-V LVTTL (default)		24mA (default)

Abbildung 4.2: Pin-Zuordnung des FPGA

Nach vollständigem Erhalt der Daten, die sich je nach Algorithmus unterscheiden (vgl. Tabelle 4.1), werden diese dem zentralen Modul `e_ecdsa` bereitgestellt. Über binäre Eingänge wird festgelegt, welcher Modus (Signieren vs. Verifizieren, Pin `mode_i`) verwendet und wann die Berechnung gestartet werden soll (Pin `enable_i`).

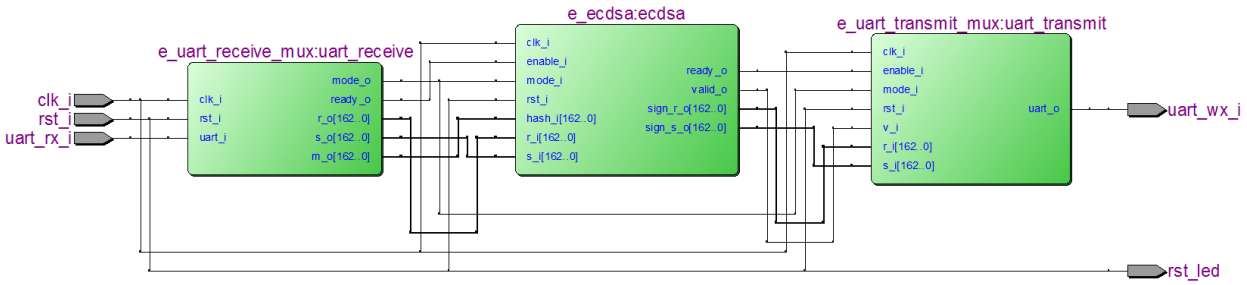


Abbildung 4.3: Top-Level-Entity der VHDL-Implementierung

Sobald die ECDSA-Entität die Berechnung abgeschlossen hat, wird ein binäres Flag gesetzt, sodass die Daten über das Modul `e_uart_transmit_mux` zurückschickt werden können.

Parameter	Beschreibung
clk_i	Globales Taktsignal
rst_i	Resetsignal
uart_rx_i	Pin zum Lesen der UART-Kommunikation
uart_tx_i	Pin zum Schreiben der UART-Kommunikation
rst_led	LED zur Kennzeichnung ob das Reset-Signal aktiv ist

Tabelle 4.3: Eingabe- und Ausgabeparameter der Haupt-Entität

4.4 ECDSA-Implementierung

4.4.1 Allgemeiner Aufbau der VHDL-Entitäten

Jede Entität, sei es eine mathematische oder eine zusammengesetzte Komponente wie die ECC-Operation, verwendet den selben Aufbau. So besitzen die Entitäten neben einem globalen Takt- und Resetsignal ein Aktivierungs- und ein Status-Flag, sowie eine beliebige Anzahl anwendungsabhängiger Eingabe- bzw. Ausgabeparameter. Eine Übersicht der Parameter ist in Tabelle 4.4 zu finden.

Parameter	Beschreibung
clk_i	Globales Taktsignal
rst_i	Resetsignal
enable_i	Flag zum Aktivieren der Berechnung
x_i	Eingabeparameter 1-n
z_o	Ausgabeparameter 1-m
p_i	Zusatzparameter 1-p
ready_o	Flag welches anzeigt, ob die Berechnung abgeschlossen ist

Tabelle 4.4: Allgemeine Eingabe- und Ausgabeparameter der Entitäten

Das Aktivierungs-Flag startet unmittelbar bei einem “High”-Pegel die Berechnung. Bleibt das Signal aktiviert, nachdem die Berechnung abgeschlossen ist, wird eine erneute Berechnung gestartet. Das Status-Flag führt standardmäßig einen “High”-Pegel, der auf “Low” wechselt, sobald das Aktivierungs-Flag gesetzt wurde. Ist die Berechnung abgeschlossen ist, wird das Status-Flag wieder auf “High” gesetzt.

Die Ablaufsteuerung findet über einen Zustandsautomaten statt. Die grundlegende Struktur der Automaten zeigt Listing 4.1

Listing 4.1: Grundstruktur der Zustandsautomaten

```
CASE current_state IS
```

```
  WHEN 0 TO 1 => enable_fct_i <= '0'; ready_o <= '1';;
  WHEN 2 => enablefct_i <= '1'; ready_o <= '0';
   ...
  WHEN N => enable_fct_i <= '0'; ready_o <= '0';
END CASE;

IF rst_i = '1' THEN
   current_state <= 0;
ELSIF clk_i'event and clk_i = '1' THEN
  CASE current_state IS
    WHEN 0 =>
      IF enable_i = '0' THEN
        current_state <= 1;
      END IF;
    WHEN 1 =>
      IF enable_i = '1' THEN
        current_state <= 2;
      END IF;
    WHEN 2 => current_state <= 3;
     ...
    WHEN N => current_state <= 0;
  END CASE;
END IF;
```

4.4.2 ECDSA-Algorithmus

Die Implementierung des ECDSA-Algorithmus findet über die Entität `e_ecdsa` statt. Für die Umschaltung zwischen Signieren und Verfifizieren wird ein Zustandsautomat verwendet, der im wesentlichen im Abhängigkeit des Parameters `mode_i` zwischen den Algorithmen wechselt. Damit kann pro Ablauf immer nur ein Algorithmus zur Zeit gestartet werden.

Die Implementierungen der Algorithmen folgt dem Ablauf aus Kapitel 3.5. Der Ablauf ist streng sequentiell, wobei versucht wird, einen maximalen Parallelisierungsgrad anzustreben. So werden bei der Verifikation beispielsweise die Variablen u_1 und u_2 parallel berechnet. Die Algorithmen sind so umgesetzt, dass jede Operation über eine eigene Entität abgebildet wird. So besteht das Signieren aus 3 Entitäten (1x Punktmultiplikation, 1x Multiplizierer, 1x Dividierer), während die Verifikation aus 6 Entitäten (1x Punktaddition, 2x Punktmultiplika-

tion, 2x Multiplizierer, 1xInverter) besteht. Grund für die Verdopplung liegt in erster Linie an der deutlich einfacheren und übersichtlicheren Implementierung. Der Nachteil dagegen ist, dass der Ressourcenverbrauch deutlich höher ist. Da die eingesetzte Hardware aber genügend Ressourcen zur Verfügung stellt, wurde auf eine Optimierung verzichtet. Ein weiterer Vorteil dieser Implementierung ist, dass theoretisch parallel signiert und verifiziert werden kann, was bei einer gemeinsamen Nutzung von Entitäten aufgrund der Umschaltlogik nicht möglich wäre. Für die zeitliche Aktivierung der Entitäten wird wieder auf den Zustandsautomat zurückgegriffen. Dazu werden die in Abschnitt 4.4.1 erwähnten Aktivierungs- bzw. Status-Flags verwendet.

Parameter	**Beschreibung**
clk_i	Globales Taktsignal
rst_i	Resetsignal
enable_i	Flag zum Aktivieren der Berechnung
mode_i	Flag zum Umschalten zwischen Signieren und Verifizieren
hash_i	Eingabe Text
r_i	R-Komponente der zu verifizierenden Signatur
s_i	S-Komponente der zu verifizierenden Signatur
ready_o	Status-Flag ob die Berechnung abgeschlossen ist
valid_o	Flag ob die zu verifizierenden Signatur valide ist
r_o	R-Komponente der erstellten Signatur
s_s	S-Komponente der erstellten Signatur

Tabelle 4.5: Eingabe- und Ausgabeparameter der ECDSA-Entität

4.4.3 Punktaddition und -dopplung

Die Punktaddition und Punktdopplung basieren auf den Formeln aus Abschnitt 3.2.

Parameter	**Beschreibung**
clk_i	Globales Taktsignal
rst_i	Resetsignal
enable_i	Flag zum Aktivieren der Berechnung
x1_i	x Komponente des Basis-Punktes
y1_i	y Komponente des Basis-Punktes
x2_o	x Komponenten der Punktdopplung
y2_o	y Komponenten der Punktdopplung
ready_o	Status-Flag ob die Berechnung abgeschlossen ist

Tabelle 4.6: Eingabe- und Ausgabeparameter der Punktdopplungs-Entität

Der zugrundeliegende Ablauf ist identisch mit der Implementierung des ECDSA-Algorithmus, wobei sich lediglich die Formeln unterscheiden. So wird für jede mathematische Operation eine

separate VHDL Entität verwendet, die sequentiell über die Aktivierungs- bzw. Status-Flags abgearbeitet werden. Die Umschaltung findet wieder über einen Zustandsautomat statt.

Parameter	**Beschreibung**
clk_i	Globales Taktsignal
rst_i	Resetsignal
enable_i	Flag zum Aktivieren der Berechnung
x1_i	x Komponente des ersten Punktes
y1_i	y Komponente des ersten Punktes
x2_i	x Komponente des zweiten Punktes
y2_i	y Komponente des zweiten Punktes
x3_o	x Komponenten des Additionsergebnisses
y3_o	y Komponenten des Additionsergebnisses
ready_o	Status-Flag ob die Berechnung abgeschlossen ist

Tabelle 4.7: Eingabe- und Ausgabeparameter der Punktadditions-Entität

4.4.4 Punktmultiplikation

Die Punktmultiplikation basiert auf dem Double-And-Add-Algorithmus. Der Pseudocode ist in Listing 4.2 zu finden.

Parameter	**Beschreibung**
clk_i	Globales Taktsignal
rst_i	Resetsignal
enable_i	Flag zum Aktivieren der Berechnung
xp_i	x Komponente des Basis-Punktes
yp_i	y Komponente des Basis-Punktes
k_i	Multiplikations-Faktor
xq_o	x Komponenten des Ergebnis-Punktes
yq_o	y Komponenten des Ergebnis-Punktes
ready_o	Status-Flag ob die Berechnung abgeschlossen ist

Tabelle 4.8: Eingabe- und Ausgabeparameter der Punktmultiplikations-Entität

Listing 4.2: Pseudocode des Double-And-Add Algorithmus

```
N = P
Q = 0

for i from 0 to m do
  if di = 1 then
```

```
    Q = point_add(Q, N)
  N = point_double(N)

return Q
```

4.4.5 Multiplizierer und Dividierer

Die Multiplizierer-Entität basiert auf einer klassischen Polynom-Multiplikation, die eine Reihe von Bitverschiebungen, XOR-Verknüpfungen und Polynomreduktionen verwendet.

Parameter	**Beschreibung**
clk_i	Globales Taktsignal
rst_i	Resetsignal
enable_i	Flag zum Aktivieren der Berechnung
a_i	Eingabewert 1
b_i	Eingabewert 2
z_o	$z = a * b \mod p$ in $GF(2^M)$
ready_o	Status-Flag ob die Berechnung abgeschlossen ist

Tabelle 4.9: Eingabe- und Ausgabeparameter der Multiplizierer-Entität

Für den Dividierer wurden zwei verschiedenen Implementierungen verwendet. Die erste Variante basiert auf einer klassischen Polynomdivision, während die zweite Implementierung eine Multiplikation mit dem Inversen durchführt.

Parameter	**Beschreibung**
clk_i	Globales Taktsignal
rst_i	Resetsignal
enable_i	Flag zum Aktivieren der Berechnung
g_i	Eingabewert 1
h_i	Eingabewert 2
z_o	$z = g/h \mod p$ in $GF(2^M)$
ready_o	Status-Flag ob die Berechnung abgeschlossen ist

Tabelle 4.10: Eingabe- und Ausgabeparameter der Dividierer-Entität

4.4.6 Quadrier

Der Quadrierer verwendet eine optimierte Implementierung für Polynome, sodass keine klassische Multiplikation verwendet werden muss. Nach der Bestimmung des Qadradats findet anschließend eine Polynomreduktion statt.

Parameter	**Beschreibung**
clk_i	Globales Taktsignal
rst_i	Resetsignal
enable_i	Flag zum Aktivieren der Berechnung
d_i	Eingabewert, von welchem das Quadrat gebildet wird
c_o	$c = d^2 \mod p$ in $GF(2^M)$
ready_o	Status-Flag ob die Berechnung abgeschlossen ist

Tabelle 4.11: Eingabe- und Ausgabeparameter der Quadrierer-Entität

4.4.7 Inverter

Zum Bestimmen des Inversen wird der erweiterte euklidische Algorithmus eingesetzt. Der Algorithmus bedient sich der Tatsache, dass wenn der Algorithmus das Tripel $d = ggt(a, b), s, t$ ermittelt, ist entweder $d = 1$ und damit $1 = t * b \mod a$, sodass t das multiplikative Inverse von $b \mod a$ ist, oder es gilt $d \neq 1$, was bedeutet, dass $b \mod a$ kein Inverses hat.

Parameter	**Beschreibung**
clk_i	Globales Taktsignal
rst_i	Resetsignal
enable_i	Flag zum Aktivieren der Berechnung
a_i	Eingabewert, von welchem das Inverse gebildet wird
z_o	$z = 1/x \mod p$ in $GF(2^M)$
ready_o	Status-Flag ob die Berechnung abgeschlossen ist

Tabelle 4.12: Eingabe- und Ausgabeparameter der Inverter-Entität

4.4.8 Addierer / Subtrahierer

Die Addition bzw. Subtraktion ist im $GF(2^M)$ trivial, da $-1 = 1$ gilt. Somit muss lediglich eine XOR-Verknüpfung der Operanden durchgeführt werden . Aufgrund der Einfachheit wurde auf eine separate Entität verzichtet.

4.4.9 Test- und Verifizierung

Für den Test- und die Verifizierung der Entities werden zwei verschiedene Arten von Testbenches verwendet. Die triviale Variante dient lediglich dem kurzen Funktionstest der Entity. Dazu werden lediglich statische Werte abgefragt, um die Entwicklung der Entities zu unterstützen. Die "echten" Testbenches dagegen versuchen anhand dynamisch generierter Testdaten die Kombination verschiedener Entities zu verifizieren.

Der Tabelle 4.13 ist zu entnehmen, dass beispielsweise die Punktmultiplikation neben der eigentlichen Multiplikation auch die Punktaddition und damit auch die dafür notwendigen mathematischen Grundoperatoren verifiziert. Da es die Testbench erfordert, dass verschiedene Entitäten separat entwickelt werden, was im Fehlerfall die Indentifizierung deutlich erschwert, wurden weitere einfachere Testbenches geschrieben um zunächst die Basisfunktionalität der Entitäten zu verifizieren.

Die Überprüfung der Bedingungen erfolgt unmittelbar nachdem das Status-Flag der zu testenden Entities ein "High"-Pegel liefert. Neben dem reinen Überprüfen der Bedingungen wird zusätzlich überprüft, ob die Berechnung in einer definierten Anzahl von Taktzyklen abgeschlossen wurde.

Beschreibung	**Typ**	**Bedingung**
tb_ecdsa	dynamisch	$(r, s) = sign(m)$ $verify((r, s), m) = True$
tb_gf2m_multiplier	statisch	
tb_gf2m_squarer	statisch	
tb_gf2m_eea_inversion	dynamisch	$x * x^{-}1 = 1$
tb_gf2m_divider	dynamisch	$c = a/b$ $c * b = a$
tb_gf2m_point_addition	statisch	
tb_gf2m_point_doubling	statisch	
tb_gf2m_point_multiplication	dynamisch	$k.P = (k-1).P + P$ $k.P = (n-1).P = -P = (xP, xP + yP)$

Tabelle 4.13: Beschreibung der Testbenches

4.5 Kommunikation der UART-Verbindung

Die UART-Komponenten übernehmen wie in Kap. 3.6 beschrieben neben der externen Kommunikation auch die Transformation der Daten. Dabei wird der Rx-Input[1] nach dem

[1]Das Rx-Signal wird erst nach Einsynchronisation mit zwei aufeinanderfolgenden Flip-Flops verarbeitet.

Empfangen über Register mit einem Parallel-Ausgang zur Verfügung gestellt. Abbildung 4.4 zeigt die Verschaltung des **Receivers** mit den Schieberegistern im RTL Viewer der Entwicklungsumgebung.

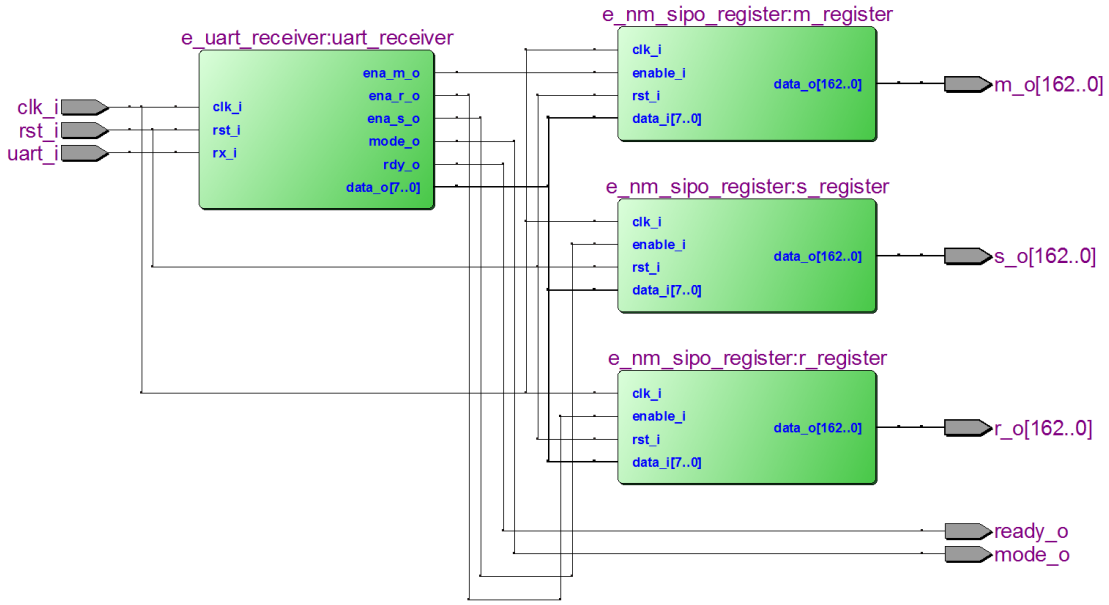


Abbildung 4.4: Ansicht des UART-Receivers im RTL Viewer

Der Receiver selbst enthält einen Zustandsautomaten, der den Eingabedatenstrom in die zwei Modi `Signieren` und `Verifizieren` klassifiziert. Anhand der Zustände werden die Steuerungssignale am Modulausgang (*enable*-Signale für die Punkte r und s auf der elliptischen Kurve sowie die Nachricht) so geschaltet, sodass jeweils die entsprechenden Eingabewörter in den dafür vorgesehenen Registern landen[2] (vgl. Abb. 4.5. Hierfür gibt es zwei vorangegangene Phase, in denen der Modus durch das erste empfangene Byte bestimmt wird (*dmode*) und anschließend über die Dauer eines Taktzyklus umgeschaltet wird (*smode*).

Die Phasen des gezeigten Automaten bilden eine Abstraktionsebene über der eigentlichen Erkennung der sequentiellen Datenbits am Rx-Eingang. Ein weiterer Zustandsautomat (s. Anhang 7.1) agiert innerhalb eines UART-Datenpaketes (Start-Bit, 8 Bit Daten, Stop-Bit) und speichert ein einzelnes Byte zwischen zur Weiterverarbeitung. Der erste Byte für den Modus muss dabei entweder “00000000b” für das Signieren oder “11111111b” für das Verifizieren sein. Der Modus wird als binäres Signal für nachfolgende Module nach außen geführt.

Der **UART-Transmitter** beinhaltet zwei Schieberegister, die einen parallelen Eingang mit einem Byte-weise seriellen Ausgang besitzen (vgl. Abb. 4.6). Das `e_uart_transmit`-Modul

[2] phase1 = r, phase2 = s, phase3 = *message*

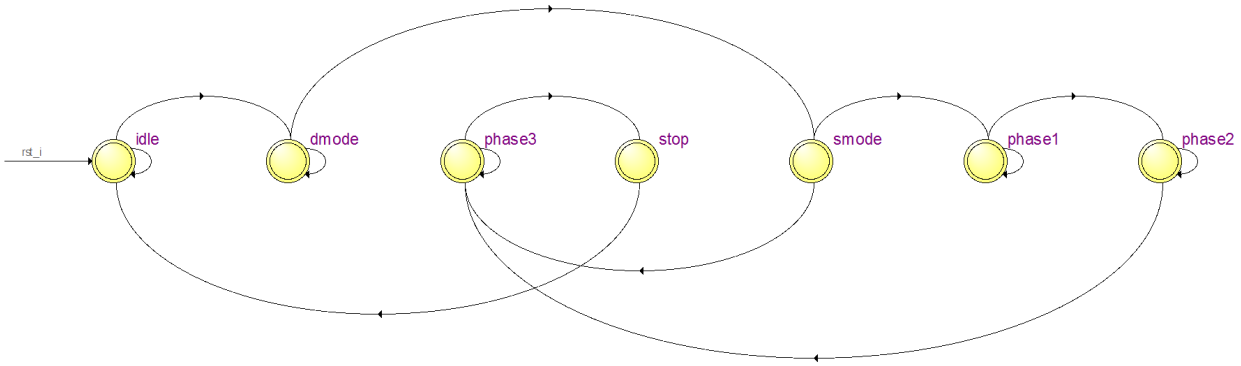


Abbildung 4.5: Zustandsautomat des Receivers zu Phasen-Bestimmung beim Empfang

steuert diese Register über einen internen Zustandsautomaten gibt die entsprechenden Steuerungssignale nach außen an die beiden Multiplexer, welche die *enable*-Eingänge anspricht.

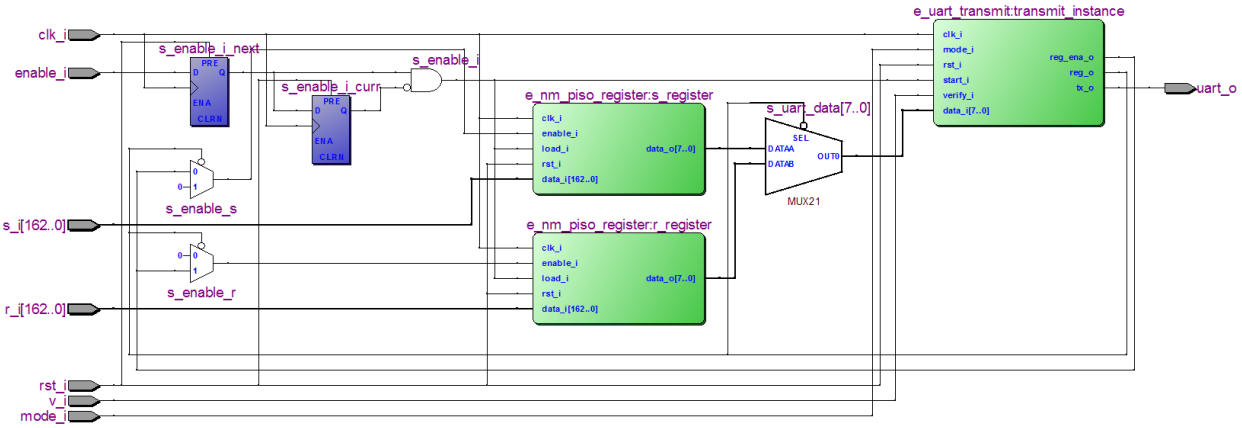


Abbildung 4.6: Ansicht des UART-Transmitter im RTL Viewer

Im Modus Signieren sendet das Modul den in den Registern gespeicherten Punkt (r, s). Beim Verifizieren wird entweder ein Byte Nullen (Signatur passt *nicht* zum Dokument) oder ein Byte Einsen (Signatur gehört zum Dokument) versendet. Der Transmitter instantiiert intern einen Timer, der das Senden am Ausgang Tx in der vor der Synthetisierung eingestellten Baud-Rate taktet.

4.6 Synthetisierungsergebnisse

Der verwendete FPGA ist ein Baustein der Altera Cyclone II Familie (EP2C35F672C6). Für die Synthetisierung wird das in der Entwicklungsumgebung Quartus II vom selben Hersteller

enthaltene Werkzeug verwendet.

Beschreibung	**Wert**
Ressourcennutzung	21%
Anzahl Logikelemente	23097
Anzahl Register	15508
Anzahl Pins	5

Tabelle 4.14: Ergebnisse der Synthetisierung

5

Messung & Vergleich

5.1 FPGA-Implementierung

Zur Messungen der Laufzeit werden Daten über die UART-Schnittstelle gesendet und die Zeit gemessen, bis das Ergebnis empfangen wurde. Da das Signieren bzw. Verifizieren, wie in Abschnitt 4.5 beschrieben, durch separate Kommandos möglich ist, kann so die Laufzeit pro Funktion ermittelt werden.

Die Kommunikation mit der FPGA-Implementierung wird über ein selbstgeschriebenes Python-Skript abgewickelt, welches einen zufälligen 163-Bit Hash erzeugt und via UART versendet. Nach dem Empfangen der erzeugten Signatur wird diese erneut zum FPGA geschickt und verifiziert. Dieser Ablauf wird für 1000 Messungen wiederholt und anschließend der Mittelwert gebildet. Die Ergebnisse sind in Tabelle 5.1 zu finden.

Funktion	**Zeit**
Signieren	77,0 ms
Verifizieren	130,1 ms

Tabelle 5.1: Messergebnisse der VHDL-Implementierung

Bei den Messergebnissen ist zu beachten, dass diese die Zeit zur Übertragung der Daten beinhalten, die je nach verwendeter Funktion stark variieren können. Für eine exakte Bestimmung der Zeiten bedarf es noch einer Bereinigung der Messergebnisse. Dazu wird rechnerisch ermittelt, wie viel Zeit für die Übetragung benötigt wird und dieser Wert von den gemessenen Zeiten abgezogen. Die serielle Übertragung findet mit einer Geschwindigkeit von 9600 Baud statt. Ein Symbol liegt dadurch für die Signaldauer von 1/9600 Sekunden, also etwa 104μs, auf der Leitung an. Zusätzlich zu den 8 Daten-Bits werden pro Übertragung noch das Start-Bit und das Stopp-Bit übermittelt. Somit werden Für die 10 Symbole ca. 1,04 ms benötigt. Da die internen Verzögerungen zwischen der UART-Schnittstelle und der ECDSA-Implementierung zu vernachlässigen ist, wird lediglich die UART-Kommunkation berücksichtigt. Dies führt zu

den folgenden Messergebnissen:

- **Signieren:**
  Senden: 1 Byte Modus, 21 Byte Message
  ⇒ 22 Byte * 10 Symbole * 104μs = 22,92ms
  Empfangen: 2x21 Byte Punkte der ECC-Funktion
  ⇒ 42 Byte * 10 Symbole * 104μs = 43,75ms
  Nettozeit Signieren = 77,0ms - 22,92ms - 43,75ms = **10,3ms**
- **Verifizieren:**
  Senden:
  1 Byte Modus, 2x21 Byte ECC-Punkte, 21 Byte Message
  ⇒ 64 Byte * 10 Symbole * 104 μs = 66,67 ms
  Empfangen:
  1 Byte für True/False ⇒ 1 Byte * 10 Symbole * 104 μs = 1,04 ms
  Nettozeit Verifizieren = 130,1 ms - 66,67 ms - 1,04 ms = **62,4 ms**

Die korrigierten Ergebnisse der Messung werden abschließend dem Simulator-Ergebnis gegenübergestellt: Das Signieren eines 163 Bit Hashes benötigt nur **23,6 ms** im Simulator, während das Verifizieren **32,8 ms** benötigt. Beide berechnete Simulationen sind damit mehr als Faktor 2 schneller. Der Grund dafür kann in der Pufferung der verwendeten "PySerial"-Bibliothek für serielle Schnittstellen in Python sowie an der Puffer-Mechanismen des Betriebssystems liegen.

Funktion	**Zeit**
Signieren	10,3 ms
Verifizieren	62,4 ms

Tabelle 5.2: Korrigierte Messergebnisse der VHDL-Implementierung

5.2 C-Implementierung

Für die Messung des C-Codes wurde die vorhandene C-Implementierung um Code zum Messen der Funktionen ergänzt und auf einem Rechner mit dem Betriebssystem Linux kompiliert und gestartet. Eine Übersicht der verwendeten Hardware ist in Tabelle 5.3 zu finden.

Wie bereits bei der VHDL-Implementierung wurden wieder 1000 Messungen durchgeführt, bei denen im Mittel die Ergebnisse aus Tabelle 5.4 erzielt wurden.

Bezeichnung	Beschreibung
CPU	Intel i5-5200U DualCore (2.20 bis 2.70 GHz, 3MB Cache)
RAM	8GB DDR3L-1600
HDD	256GB SSD
Grafik	Intel® HD 5500 Grafik
OS	Linux

Tabelle 5.3: Konfiguration der verwendeten Hardware

Funktion	**Zeit**
Signieren	169.1 ms
Verifizieren	340.5 ms

Tabelle 5.4: Messergebnisse der C-Implementierung

5.3 Gegenüberstellung & Bewertung

Bei den Messergebnissen aus Tabelle 5.5 ist klar zu erkennen, dass die VHDL-Implementierung je nach Funktion eine Performance-Steigerung zwischen 82 % - 95 % aufweist.

	VHDL	**C**	**Gewinn**
Signieren	10,3 ms	169,1 ms	94 %
Verifizieren	62,4 ms	340,5 ms	82 %

Tabelle 5.5: Gegenüberstelung der Messergebnisse von der C- und VHDL-Implementierung

Bei den Messungen gilt zu berücksichtigen, dass der Performance-Gewinn bereits bei einer geringen Schlüssellänge von 163 Bit erzielt wurde. Da bei einem Erhöhen der Schlüssellänge die Vorteile der Hardware-Implementierung mehr zum Tragen kommen, ist mit weiteren Leistungssteigerungen zu rechnen. Außerdem wurde, wie in Kapitel 4 beschrieben, lediglich eine triviale VHDL-Implementierung umgesetzt. Durch ein Austauschen der generischen Implementierung durch eine feste Verdrahtung der Bitlängen oder anderen Optimierungen sollen sich weitere Performance-Verbesserungen erzielen lassen.

Abschließend sei noch einmal erwähnt, dass die C-Implementierung eine auf Hardware optimierte Implementierung (Bit-Felder) verwendet. Es ist daher davon auszugehen, dass sich auch die C-Ergebnisse verbessern lassen, indem beispielsweise auf numerische Verfahren zurückgegriffen wird!

6

Zusammenfassung & Ausblick

7 Anhang

7.1 Zustandsautomat im UART-Receiver-Modul

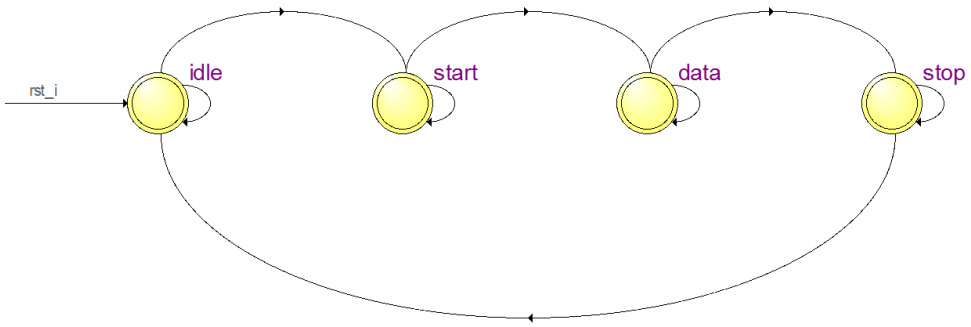


Abbildung 7.1: Zustandsautomat des Receivers zu Erkennung eines UART-Datenpakets

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcode

Literaturverzeichnis

[AS12] Moncef Amara and Amar Siad. Hardware implementation of elliptic curve point multiplication over gf(2m) for ecc protocols. International Journal for Information Security Research (IJISR), Volume 2, Issue 1, 2012.

[Bas16] Circuit Basics. Basics of uart communication. http://www.circuitbasics.com/basics-uart-communication/, 2016. Zugriff: 17.02.2018.

[Bau09] Carsten Baum. Public-key-verschlüsselung und diskrete logarithmen. Institut für Informatik, 2009.

[Gre05] Ingo Grebe. Elliptische kurven in der kryptographie. Institut für Informatik, 2005.

[HMV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[Kew16] Philipp Kewish. *Reconfigurable Computing am Beispiel ECDSA*. 2016.

[Pom08] Klaus Pommerening. Der diskrete logarithmus. Vorlesungsskript Kryptologie, 2008.

[Put14] Christoph Puttmann. Ressourceneffiziente hardware-software-kombinationen für kryptographie mit elliptischen kurven. Technische Fakultät, 2014.

[Wol09] Ruben Wolf. Verifikation digitaler signaturen. https://www.informatik.tu-darmstadt.de/BS/Lehre/Sem98_99/T11/index.html, 2009. Zugriff: 15.02.2018.