

Hochschule RheinMain

Fachbereich Design Informatik Medien
Master-Studiengang Informatik
(Smarte Systeme für Mensch und Technik)

Master-Thesis
zur Erlangung des akademischen Grades
Master of Science – M. Sc.

Verifiable delay functions in Hardware

Vorgelegt von Thorsten Knoll
am 20.05.2020

Referent: Prof. Dr. Steffen Reith
Korreferent: M.Sc. Fabio Campos

Für Leo und für Harald.

Danke für die unermüdliche Unterstützung.

Erklärung gem. BBPO 4.1.5.4 (3):

Ich versichere, dass ich die Master-Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum,

Unterschrift Studierende/Studierender

Verbreitungsformen:

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Master-Arbeit:

Verbreitungsform	Ja	Nein
Einstellung der Arbeit in die Hochschulbibliothek mit Datenträger		
Einstellung der Arbeit in die Hochschulbibliothek ohne Datenträger		
Veröffentlichung des Titels der Arbeit im Internet		
Veröffentlichung der Arbeit im Internet		

Ort, Datum,

Unterschrift Studierende/Studierender

Inhalt

Zusammenfassung	1
Abstract (English)	3
1. Einleitung	5
1.1. Ziele	6
1.1.1. VDFs sind neu in der Kryptographie	6
1.1.2. VDF Alliance Wettbewerb	6
1.1.3. Open Source FPGAs und Tools	7
1.1.4. Für Lehre geeignet?	8
1.1.5. Demonstrator	8
1.2. Zitathinweis zu den Kapiteln 2.1 und 2.2	8
2. Zufall und Computer	9
2.1. Was ist Zufall (in der Informatik)?	10
2.1.1. Menschliche Mustererkennung	11
2.1.2. Shannons Entropie	12
2.1.3. Entropiequellen und -pools	13
2.1.4. Anforderungen an kryptographisch sichere Zufallszahlen	17
2.2. Zufallszahlengeneratoren (RNG)	18
2.2.1. Algorithmische Entropie-Erhaltung	19
2.2.2. Pseudo-Zufallszahlengeneratoren (PNRG)	20
2.2.3. Kryptogr. sichere Zufallszahlengeneratoren (CSPRNG)	22
2.2.4. Spezielle Zufallszahlenerzeuger	25
3. Verifiable delay functions	27
3.1. Dezentrale, verteilte Systeme	27
3.2. Proof-of-Work (PoW)	28
3.3. VDF: Allgemeine Idee	30
3.4. Konstruktionsprinzip	31
3.5. Auswahl für die Implementierung	34

4. Hardware	35
4.1. Design	38
4.1.1. Funktionalität	38
4.1.2. Komponenten: Maximal Open Source	39
4.1.3. HDL und Tools	41
4.1.4. Karatsuba Multiplikation	44
4.1.5. Barret Reduktion	46
4.2. Implementierung	48
4.2.1. Installation der Tools	48
4.2.2. Makefile für die Toolchain	49
4.2.3. SpinalHDL	50
4.2.4. Modularität	53
4.2.5. UART	56
4.2.6. Karatsuba: Kombinatorische Lösung	58
4.2.7. Barret Reduktion: Zustandsautomat	59
4.2.8. Ansteuerung mit RISC-V: MikroPython	61
4.3. Testen und debuggen	62
4.4. Demonstrator	64
5. Open Source Code	65
6. Fazit	67
6.1. Zielerreichung?	67
6.1.1. Open Source	68
6.1.2. VDF Wettbewerb	71
6.1.3. In der Lehre	74
6.1.4. Demonstrator	75
A. Anhänge	77
A.1. FPGA Toolchain: Makefile	77
A.2. Kleines SpinalHDL Beispiel: Blinky	78
A.3. UartFsmInOut: SpinalHDL Quellcode	80
A.4. Karatsuba Multiplikation als Klasse (SpinalHDL)	82
A.5. MultMod: SpinalHDL Quellcode inkl. Debugging Leitungen	83
A.6. Ansteuerung RISC-V: Mikropython Implementierung	87
A.7. NextPnR-ECP5 Logging output	89
Abbildungsverzeichnis	91
Tabellenverzeichnis	93

Listingverzeichnis	95
Quellen	97

Zusammenfassung

Diese Arbeit fasst ein aktuelles Thema der Kryptographie auf. Vor zwei Jahren wurde mit der Definition von „Verifiable delay functions“ (VDF) eine neue Kategorie kryptographischer Funktionen eröffnet. Eine Hauptanwendung solcher VDF sind die, auch noch recht neuen, Blockchain-Technologien. In vielen Blockchains (z.Bsp. die kryptographische Währung Bitcoin) werden sehr energieintensive Berechnungen zur Absicherung des Netzwerks verwendet. Am bekanntesten ist hier der „Proof-of-Work“ Algorithmus. Aktuelle Entwicklungen versuchen, diesen Energiebedarf zu verringern. Eine notwendige Funktionalität hierfür ist die Erzeugung von Zufallszahlen in dezentralen, verteilten Netzwerken. Über VDF besteht die Hoffnung, ein Teil der Lösung dieses algorithmischen Problems zu sein. In der Konstruktion von VDF wird die gleiche Berechnung in großer Anzahl wiederholt ausgeführt. Eine der bisher am weitesten untersuchten Konstruktionen berechnet das wiederholte Quadrieren in einer RSA Gruppe. Ist die Ordnung der Gruppe unbekannt, existiert kein effizienter Weg für diese Berechnung außer dem wiederholten Quadrieren. Diese Berechnung sollte in VDF mit der größtmöglichen Geschwindigkeit ausgeführt werden. Daher ist eine Implementierung in Hardware hierfür sinnvoll. In dieser Arbeit wurde ein Hardware-Beschleuniger für VDF entwickelt und für einem Field-Programmable-Gate-Array (FPGA) implementiert. Die erweiterte Zielsetzung beinhaltet die größtmögliche Nutzung von Open Source Komponenten für diese Aufgabe. Weiterhin wurden die verwendeten Open Source Komponenten evaluiert, Vor- und Nachteile herausgearbeitet und die Möglichkeit zur Verwendung in der akademischen Ausbildung untersucht. Ein Demonstrator zeigt die Funktionalität der fertigen Implementierung und ermöglicht Messungen zur Geschwindigkeit.

Abstract (English)

This thesis picks up an actual topic in cryptography. Two years ago a new category of cryptographic functions got defined: Verifiable delay functions(VDF). One main application for VDF are blockchain technologies, which are still quite new, too. Many blockchains (i.e. the peer-to-peer electronic cash system Bitcoin) works with an energy hungry algorithm called Proof-of-Work. It's used to secure the blockchain network. Currently research is done to minimize this electrical consumption. A necessary functionality for this is the generation of random numbers in decentralized, distributed networks. The hopes about VDF are to be a part of the solution for this algorithmic problem. In the construction of VDF a single operation gets repeated many times. One of the most researched and understood constructions calculates squarings in RSA groups. If the order of the group is unknown, there is no shortcut for the calculation. It must be done in an iterative way and it should be done as fast as possible. This is where Field-Programmable-Gate-Arrays (FPGA) come in handy. In this work a hardware accelerator for VDF is developed and implemented in an FPGA. An extended goal is the most possible usage of open source components for this task. Furthermore the open source components got evaluated, advantages and disadvantages were carved out and the possible usage in academic teaching got examined. A demonstrator shows the functionality of the implementation and gives the opportunity to perform measurements.

1. Einleitung

Während des Studiums der Informatik werden viele, notwendige Grundlagen zum allgemeinen Verständnis unterrichtet und vom Studierenden erlernt. Viele dieser Grundlagen sind erst im weiteren Verlauf des Studiums in einen anwendbaren Kontext zu bringen. Die Integration der vielfältigen und teilweise sehr unterschiedlichen, erworbenen Fähigkeiten zu komplexen Zusammenhängen und Anwendungen machen diese erst interessant. Erst in der Anwendung und Kombination der Grundlagen entsteht das Neue und Spannende, die Entdeckung. Im späteren Verlauf des Studiums ergeben sich immer mehr Möglichkeiten, aus den festgefahrenen Wegen der Grundlagen auszubrechen und eigene Entdeckungsreisen im Bereich der Informatik zu starten. Diese Thesis spiegelt das Interesse des Autors an neuen Technologien in Verbindung mit erlernten Grundlagen wieder. Im Verlauf dieser Arbeit werden Grundlagen sowohl aus der theoretischen Informatik und diskreten Mathematik mit dem hoch aktuellen Gebiet der Hardware Programmierung verbunden. Die theoretische Seite bietet mit der Erforschung einer neuen Art kryptographischer Funktionen, den sogenannten Verifiable delay functions (VDF) ein Anwendungsbeispiel für die Programmierung von FPGAs. Innerhalb des Themenbereichs der Hardware Entwicklung mit FPGAs wird die neue und wachsende Verwendung von Open Source Werkzeugen (Tools) untersucht. Auch diese sind erst seit Kurzem verfügbar, befinden sich noch im frühen Entwicklungsstadium und sind noch nicht ausreichend evaluiert. Weiterhin findet die Entwicklung der quelloffenen (Open Source) RISC-V Architektur weltweit wachsende Beachtung und eine steigende Zahl von Anwendungen. In folgenden Kapiteln wird, in der Definition der Ziele dieser Arbeit, das Interesse und die Notwendigkeit der Evaluation dieser neuen Möglichkeiten aufgezeigt. Die gesamte Arbeit lässt sich als die Verbindung von zwei großen Themenbereichen verstehen:

- Die Grundlagen der Erzeugung von Zufallszahlen in der Informatik. Mit dem Ziel, die neue Kategorie von VDFs als Anwendungsbeispiel zu definieren.
- Die Implementierung der definierten Anwendung in einem Field-Programmable-Gate-Array(FPGA) unter Einsatz sovieler Open Source Bestandteile wie möglich.

Die daraus abgeleiteten, kleinteiligeren Ziele sind im Folgenden beschrieben.

1. Einleitung

1.1. Ziele

1.1.1. VDFs sind neu in der Kryptographie

Im Jahr 2009 wurde eine neue, bis dahin nicht entdeckte, Art von dezentralen, verteilten Netzwerken erfunden. Die kryptographische Währung Bitcoin wurde als Whitepaper unter dem Pseudonym „Satoshi Nakamoto“ veröffentlicht[Nak09] und kurze Zeit später folgte die erste Implementierung. Das Bitcoin Netzwerk ist seitdem ununterbrochen in Betrieb und erfreut sich wachsender Beachtung in etlichen Forschungsbereichen. In den letzten zehn Jahren wurden durch diese Entwicklung neue kryptographische Verfahren und Methoden entdeckt. Manche davon greifen alte Ideen auf und entwickeln neue Varianten daraus, andere sind vollständig neu konzipiert. Ein viel diskutiertes Thema ist der weltweite, notwendige Energieeinsatz zum Betrieb des Bitcoin Netzwerks. Die Absicherung des Netzwerks enthält dezentrale, verteilte und wiederholte Hashberechnungen in einer Größenordnung, dass in der weltweiten Summe mehrere Kraftwerke nur dafür benötigt werden. Diese Berechnungen sind weitläufig als „Proof-of-Work(PoW)“ bekannt. Ohne zu tief in die Funktionsweise von PoW einzugehen, kann PoW als eine gewichtete Zufallsfunktion in einem dezentralen, verteilten Wettbewerb betrachtet werden. Die Gewichtung ist die Menge der aufgewandten Energie, also der Einsatz für die Teilnahme am Wettbewerb. Der mögliche Gewinn sind neu generierte Bitcoins. Der Gewinner wird durch die Zufallsfunktion des PoW ermittelt. Das bestehende Problem ist hierbei der enorme Energieeinsatz. Daher wird weltweit dran geforscht, diesen Energieeinsatz zu minimieren. Ein Teilansatz hierfür ist die Ersetzung der dezentralen, verteilten Zufallsfunktion. Hierzu werden, als einer von mehreren möglichen Ansätzen, sogenannte Verifiable delay functions(VDF) entwickelt.

Um die allgemeine Problematik der Erzeugung von Zufallszahlen unter Verwendung von Computern zu verstehen, werden zuerst die Grundlagen der Zufallszahlenerzeugung in Kapitel 2 erläutert. Die Funktionsweise, die bisherige Forschung und der, in dieser Arbeit implementierte Teil von VDFs folgen in Kapitel 3. Diese zwei Kapitel sind der theoretische Teil dieser Arbeit. Die Beschreibung der praktischen Implementierung folgt dann in den weiteren Kapitel.

1.1.2. VDF Alliance Wettbewerb

Anfang 2019 wurde die VDF Alliance zur Erforschung und Entwicklung von VDFs gegründet[All19]. Ein großer Projektpartner dieser Allianz ist die Ethereum Foundation. In Ethereum wird intensiv an der Ersetzung des PoW Algorithmus durch den sogenannten „Proof-of-Stake(PoS)“ Algorithmus geforscht. Ein Bestandteil hierfür sind VDFs. Eine der ersten Aktionen der VDF Alliance war die Auslobung eines weltweiten Wettbewerbs über die Implementierung einer VDF in FPGA Hardware. Der zu imple-

mentierende Teil in diesem Wettbewerb ist die möglichst schnelle Berechnung von Quadraturen in RSA-Gruppen. Die Vorgaben für die zu verwendenden Ressourcen sind die Nutzung von Xilinx FPGAs und der Xilinx Toolchain Vivado. Für Beides wird den akkreditierten Teilnehmern des Wettbewerbs ein gesponsorter Zugang zur sonst kostenpflichtigen Amazon AWS-F1 FPGA Cloud bereitgestellt. Damit steht den Teilnehmern die aktuell leistungsfähigste FPGA-Umgebung des Herstellers Xilinx zur Verfügung. Außerhalb eines solchen, gesponsorten Wettbewerbs bedeutet die Programmierung für Xilinx FPGAs eine Reihe von Einschränkungen:

- Non-disclosure agreements (NDA, Verschwiegenheitserklärung)
- Lizenzgebühren für die Nutzung der Tools (Vivado Suite)
- Eingeschränkte Möglichkeiten zur Portierung auf FPGAs anderer Hersteller
- Nicht für jeden Interessierten selbst zu bauen (ohne die Tools)
- Überladene graphische Benutzeroberflächen
- Sehr große Installationen (Vivado > 30 GB)
- Wenig Unterstützung für konsolenbasierte Buildprozesse (z.Bsp. Makefiles)

Ein Ziel dieser Arbeit ist daher die Implementierung der gleichen Problemstellung (Quadratur in RSA Gruppen), aber unter der Verwendung von möglichst vielen Open Source Bestandteilen. Viele der oben genannten Einschränkungen könnten damit aufgehoben werden. Weiterhin stellt sich die Frage, welche Größe der Implementierung erreicht werden kann. Es existiert bisher kein Open Source FPGA in einer ähnlichen Größenordnung wie die Xilinx FPGAs in der Amazon AWS-F1 Cloud. Als Kenngrößen könnten zum Beispiel die Bitbreite der möglichen Berechnung oder die Zeit für eine einzelne Berechnung verwendet werden.

1.1.3. Open Source FPGAs und Tools

Aus der Verwendung von Open Source Komponenten für die Programmierung von FPGAs ergeben sich weitere Ziele. Es soll in dieser Arbeit evaluiert werden, wie gut sich diese Bestandteile verwenden lassen. Die Fragestellungen hier sind:

- Kann ein vollständiger Open Source Workflow für die FPGA Programmierung erreicht werden? Wie sieht dieser Workflow aus und welche Probleme entstehen?
- Müssen graphische Oberflächen benutzt werden?
- Entstehen Vorteile für die akademische Ausbildung und für akademische Veröffentlichungen durch die Nutzung von Open Source für FPGAs?

1. Einleitung

- Ist eine stark abstrahierte Hardware Beschreibungssprache von Vorteil? Und warum?
- Welche Funktionalitäten sind (noch) nicht mit Open Source Komponenten erreichbar?

Im Verlauf der Arbeit werden mit Sicherheit weitere Fragen oder sinnvolle Änderungen an den genannten Fragen entstehen. Diese werden dann in den jeweiligen Kapiteln und im Fazit (Kapitel 6) erläutert.

1.1.4. Für Lehre geeignet?

Bisher findet die akademische Ausbildung zur Programmierung von FPGAs hauptsächlich auf FPGA-Plattformen der großen Hersteller statt. Dies hat seine eigenen, validen Begründungen. Zum Beispiel erlernen die Studierenden damit gleich eine, in der Industrie weit verbreitete Entwicklungsumgebung kennen und können bei späteren Arbeitgebern ihr Wissen direkt umsetzen. Es gibt hierzu noch eine Reihe weiterer Argumentationen für die Verwendung proprietärer Umgebungen. Aber gibt es auch Argumente für die Lehre über FPGA Programmierung mit Open Source Umgebungen? Wenn ja, welche Argumente sind das?

Ein weiteres Ziel dieser Thesis ist die Erstellung von kleinteiligen Implementierungsprojekten. Es wird in dieser Arbeit darauf geachtet, die einzelnen Schritte der Implementierung möglichst modular und nachvollziehbar in einzelne, lauffähige Projekte zu überführen. Im besten Fall entsteht damit eine nutzbare Projektbasis für eventuelle Lehrveranstaltungen.

1.1.5. Demonstrator

Ein Demonstrator soll die Implementierung in Funktion zeigen. Mindestens soll ein FPGA enthalten sein, der mit Open Source Tools programmiert wurde. Optional ist eine Ansteuerung des FPGA mittels eines RISC-V Mikrocontrollers möglich. Die Funktionalität soll der oben beschriebenen Quadrierung in RSA-Gruppen entsprechen. Messungen über die Geschwindigkeit der Implementierung sollen möglich sein.

1.2. Zitathinweis zu den Kapiteln 2.1 und 2.2

Einige der Beispiele in den Kapiteln 2.1 und 2.2 und der „rote Faden der Erzählung“ sind angelehnt an ein unveröffentlichtes Buch der Autoren Thorsten Knoll (Autor dieser Arbeit) und Harald Heckmann [KH]. Die Beispiele sind entsprechend markiert.

2. Zufall und Computer

„Gott würfelt nicht“ ist ein bekanntes Zitat von Albert Einstein, aus seinen Diskussionen mit Nils Bohr [EBB72]. Während Einstein hier auf seine Theorie der versteckten Variablen im Bereich der Elementarphysik Bezug nimmt, könnte eine moderne, informationstechnische Version dieser Aussage „Computer können nicht würfeln“ lauten. Unsere klassischen Computer sind ausschließlich zu deterministischen Berechnungen fähig. Kein klassischer Algorithmus kann nur durch seine Ausführung selbst echte Zufallszahlen erzeugen, auch wenn manche (z.Bsp. studentische) Implementierungen diesen Eindruck erwecken könnten. Für die Erzeugung echter Zufallszahlen mit klassischen Computern sind immer externe, meistens physikalische Entropiequellen notwendig. Wie sich Zufall in der Informatik kategorisieren und in Kenngrößen (z.Bsp. Ein Maßeinheit für Entropie) verpacken lässt, wird im folgenden Kapitel 2.1 beschrieben. Beispiele der gängigen Methodiken und Konstruktionsprinzipien zur Zufallszahlenerzeugung finden sich dann in Kapitel.

Einsteins Idee der verborgenen Variablen und damit auch seine berühmte Aussage wurde Jahrzehnte nach seinem Tod durch Experimente (1982), basierend auf den Bellschen Ungleichungen (1964), widerlegt [ADR82]. In den Computerwissenschaften zeichnet sich aktuell ein ähnlicher Vorgang ab. Quantencomputer können sehr wohl durch Algorithmik echte Zufallszahlen erzeugen. Die Algorithmik für die Erzeugung eines einzelnen Zufallsbits auf einem Quantencomputer ist ein einfach nachvollziehbares Einführungsbeispiel in den Lernunterlagen für die Programmierung von Quantencomputern [Hom15].

Aktuell sind noch keine leistungsstarken Quantencomputer öffentlich verfügbar. Dies könnte nach aktuellen Prognosen auch noch 10-20 Jahre dauern. In der vorliegenden Arbeit werden daher ausschließlich klassische Algorithmen und Computer betrachtet. Aber mit dem Respekt, dass sich diese Betrachtungen in den nächsten Jahren plötzlich und radikal ändern oder als nicht mehr gültig erweisen könnten. Für die aktuellen Probleme bezüglich der Zufallserzeugung sind aber hier und jetzt praktikable und implementierbare Lösungen erforderlich.

2. Zufall und Computer

In den letzten Jahren sind durch die wachsende Etablierung von dezentralen, verteilten Computernetzwerken (z.Bsp, Peer-to-peer Filesharing, Kryptographische Währungen, Interplanetary Filesystem [IPFS]) neue Anforderungen an die Erzeugung von Zufallszahlen entstanden. Die Grundlagen für diese Anforderungen werden in Kapitel 3.1 erläutert und bilden gleichzeitig die Überleitung zu den sogenannten „Verifiable delay functions (VDF)“, einem Hauptthema dieser Arbeit.

2.1. Was ist Zufall (in der Informatik)?

Der erste Schritt in vielen kryptographischen Verfahren ist das Finden einer sicheren Zufallszahl. Oft wird in der mathematischen und kryptographischen Literatur hierzu genau ein Satz genannt:

Wähle z als eine kryptographisch sichere Zufallszahl.

Menschen haben für die Erzeugung von Zufall und Zufallszahlen vielfältige Methoden entwickelt. Eine der einfachsten ist ein Münzwurf. Es wird davon ausgegangen, dass eine faire Münze die gleichen Wahrscheinlichkeiten für das Ereignis „Landet auf der Kopfseite“ und „Landet auf der Zahlseite“ abbildet, die berühmte 50/50 Chance. Das Ergebnis kann dann auf eine Ja/Nein Entscheidung abgebildet werden und es besteht hinreichend Konsens über die Einordnung als zufällig bestimmte Entscheidung. Für Gesellschaftsspiele mit Zufallselementen werden hierfür Würfel mit mehr als zwei Seiten benutzt. Neben den sehr bekannten 6-seitigen Würfeln sind aber auch Würfel mit sehr viel mehr Seiten verfügbar (z.Bsp. 20-seitige Würfel für Rollenspiele). Hier wird das Ergebnis direkt als zufällige Zahl ausgewertet. Für eine kryptographisch sichere Zufallszahl der Länge 256 Bit könnte also eine Münze genau 256 mal geworfen und jeder Wurf auf ein einzelnes Bit („0“ oder „1“) abgebildet werden. Dies wäre ein aufwändiges, aber gutes und sicheres Verfahren für die Erzeugung von privaten Schlüsseln in der Kryptographie.

Die zu bewältigende Aufgabe in der Informatik besteht nun in der automatischen Erzeugung von Zufallszahlen durch Computer ohne die Einbindung von Menschen. Wie schon weiter oben erwähnt, sind Computer aber durch ihre Eigenschaft der deterministischen Berechnung denkbar ungeeignet zur Erzeugung von Zufallszahlen. Um hierfür sinnvolle Verfahren zu entwickeln, lohnt sich der Blick auf die menschliche Wahrnehmung von zufälligen Ereignissen. Es folgt dann eine Definition der Kenngröße Entropie als Maß von Zufall nach Claude Shannon und die Definition von Anforderungen an kryptographisch sichere Zufallszahlen.

2.1.1. Menschliche Mustererkennung

Eine gut ausgeprägte menschliche Eigenschaft ist die Mustererkennung. Oft reichen für die visuelle Identifikation von Objekten schon einige wenige markante Merkmale aus, um ein Objekt zu kategorisieren. So können alleine die Umrisse von schon zahlreich wiedererkannten Objekten (z.Bsp. Bäume, Autos, Flugzeuge, Stühle) zu einer eindeutigen Erkennung führen. Was aber geht in Menschen vor, wenn eine solche Erkennung nicht stattfindet? Als Beispiel sei hier die folgende Abbildung 2.1 aufgeführt. Die zu lösende Aufgabe ist die Bestimmung der jeweils untersten Reihe. Welche Kästchen sind weiß und welche schwarz zu markieren? Für die Muster links und in der Mitte scheint

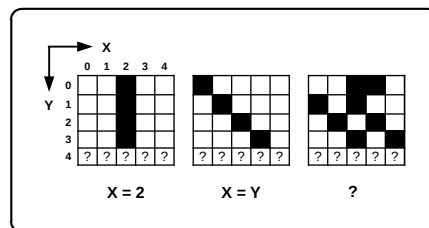


Abb. 2.1.: Menschliche Mustererkennung [KH]

die Bestimmung nicht schwierig und eher eindeutig. Für das rechts abgebildete Muster scheint diese Aufgabe wesentlich schwerer zu sein. Nur durch die gegebene Aufgabenstellung mit der enthaltenen Information über die Existenz eines Musters sucht der Rätsellöser weiter nach einem Muster. Ohne diese Information würde das Muster eventuell direkt als zufälliges Muster beurteilt.

Lösung: Das Muster auf der rechten Seite ist eine Darstellung der Primzahlen bis zur Zahl 20 (2, 3, 5, 7, 11, 13, 17, 19). In einer Formel ausgedrückt wäre das: $x + 5 * y = \text{Primzahl}$. In der zu bestimmenden Zeile würde das vierte Kästchen (entspricht der 23) schwarz markiert werden.

Die Lösung des Rätsels und die Einordnung als nicht zufälliges Muster hängt also stark von der Wissensbasis ab. Ohne das Wissen über Primzahlen ist hier keine Lösung zu ermitteln und es könnte als zufällig bewertet werden. Ähnlich verhält es sich mit einem fairen Münzwurf. Es ist Menschen einfach nicht möglich, alle physikalischen Faktoren zu ermitteln und in eine Berechnung zu integrieren, um das Ergebnis vorherzusagen. Somit ist hier eine entscheidende Aussage zur Definition von Zufall enthalten. Über die Mustererkennung und die Wissensbasis definiert sich hier die fehlende Möglichkeit zur Vorhersage von Ereignissen.

2.1.2. Shannons Entropie

Eine sinnvolle und viel genutzte Möglichkeit der Bemessung von Zufallsereignissen bietet das Modell einer gedächtnislosen Quelle im Kontext der Informationstheorie nach Claude E. Shannon. Hierzu wird ein Zufall erzeugendes System (z.Bsp. Würfel, Münze, physikalische Vorgänge) als Informationsquelle mit der Ausgabe von Ereignissen modelliert. Es folgt die Definition einer diskreten, gedächtnislosen Informationsquelle:

Definition 2.1.1: Gedächtnislose Quelle

Eine diskrete, gedächtnislose Informationsquelle X emittiert in zeitlichen Abständen einzelne Ereignisse e_i mit $1 \leq i \leq n$ aus einem Alphabet $E = \{e_1, e_2, \dots, e_n\}$ mit den zugehörigen Auftrittswahrscheinlichkeiten $P(e_i) = p_i$. Alle emittierten Ereignisse sind statistisch unabhängig voneinander.

In Shannons Arbeit über Informationstheorie von 1948 [Sha48] leitet er hieraus den mittleren Informationsgehalt einer solche Quelle ab. Dieses Maß wird Entropie genannt und hat die Maßeinheit „Shannon“:

Definition 2.1.2: Entropie einer gedächtnislosen Quelle

Sei X eine gedächtnislose Quelle, mit n möglichen Ereignissen e_1, e_2, \dots, e_n und deren Auftrittswahrscheinlichkeiten p_1, p_2, \dots, p_n . Dann ist

$$H(X) = \sum_{i=1}^n (p_i \log_2(\frac{1}{p_i}))$$

die **Entropie von X** (Einheit: Shannon).

Die Interpretation der Entropie nach Shannon wird durch die folgenden zwei Beispiele verständlich.

Beispiel 1: Entropie eines fairen Münzwurfs [KH]

Sei X_1 eine faire Münze mit den Ereignissen $\{Z(ahl), K(opf)\}$. Die Fairness der Münze bedeutet, dass die Auftrittswahrscheinlichkeiten für beide Ereignisse gleich sind: $p_Z = p_K = 0,5$. Die Entropie des Münzwurfs ist dann

$$H(X_1) = 0,5 \cdot \log_2(2) + 0,5 \cdot \log_2(2) = 1 \cdot \log_2(2) = 1 \text{ Shannon}$$

Beispiel 2: Entropie eines fairen 6-seitigen Würfelwurfs [KH]

Sei X_2 ein fairer 6-seitiger Würfel mit den Ereignissen 1, 2, 3, 4, 5, 6. Die Fairness des Würfels bestimmt die Auftrittswahrscheinlichkeiten: $p_i = \frac{1}{6}$ mit $1 \leq i \leq 6$. Die Entropie eines Würfelwurfs ist dann

$$H(X_2) = \sum_{i=1}^6 \frac{1}{6} \cdot \log_2(6) = \log_2(6) \approx 2,58 \text{ Shannon}$$

Die Entropie eines fairen Münzwurfs beträgt genau 1 Shannon. Die Kodierung eines solchen Münzwurfs in Bits entspricht genau 1 Bit („0 “ oder „1 “). Hier wird ersichtlich, warum die Entropie nach Shannon auch als der mittlere Informationsgehalt bezeichnet wird. Ein fairer Münzwurf hat den mittleren Informationsgehalt eines Bits. Analog hierzu hat der Wurf eines fairen 6-seitigen Würfels den mittleren Informationsgehalt $\log_2(6) \approx 2,58$ Bit. Das ist die rechnerische Anzahl an Bits, die benötigt wird um die 6 möglichen Ereignisse des Würfels abzubilden.

Zusammenhang zwischen den Einheiten „Bit “ und „Shannon “:

Die Herleitung der Entropie wurde von Shannon als die „Verringerung der Ungewissheit“ über die Ereignismöglichkeiten der Informationsquelle bezeichnet. In mehreren Schritten werden hierzu die Ereignismöglichkeiten immer weiter eingegrenzt (verringert), bis nur noch die Einzelereignisse übrig bleiben. Diese Schritte der Verringerung sind als binäre Entscheidungen modelliert und daraus ergibt sich der Logarithmus zur Basis 2 in der Entropieformel. Und genau die Logarithmusbasis 2 stellt die direkte Beziehung der Einheiten „Bit “ und „Shannon “ her. Es wäre auch eine Herleitung über eine andere Logarithmusbasis machbar, dann wäre aber 1 Shannon \neq 1 Bit.

2.1.3. Entropiequellen und -pools

Da Computer sich eher schlecht mit Münz- oder Würfelwurfautomaten bestücken lassen, werden algorithmische Lösungen für die Zufallszahlenerzeugung benötigt. Und genau hier fangen die Herausforderungen an. Computer führen deterministische Berechnungen aus, die gleiche Eingabe führt immer zur gleichen Ausgabe. Das ist erstmal genau das Gegenteil von zufälligem Verhalten. Daher werden Entropiequellen benötigt. Eine Entropiequelle erzeugt Zahlen aus physikalischen Vorgängen, für welche sowohl die Wissenbasis als auch eine vorhersagende Berechnungsmöglichkeit fehlt (siehe Kapitel 2.1.1). Eine Standardmessgröße ist hier die Zeit. Für Entropiequellen werden also physikalische Vorgänge zeitlich gemessen und daraus Zahlen generiert. Die Hoffnung ist hier, dass durch mehrere, voneinander unabhängige Quellen genügend Entropie im Computersystem vorhanden ist um sichere Zufallszahlen erzeugen zu können. Einige Beispiele für solche Quellen sind [KH]:

2. Zufall und Computer

- Timings des Netzwerkverkehrs
- Timings der Benutzereingaben (Maus, Tastatur)
- Festplattenzugriffe und Interrupt Timings
- Physikalische Vorgänge in elektronischen Schaltungen
- Takt Jitter, thermisches Rauschen

Mit mehreren Entropiequellen in einem einzelnen Computersystem stellt sich die Frage, wie diese Quellen zur Erzeugung von Zufallszahlen gemeinsam genutzt werden können. Die Antwort hierzu heißt Entropiepool. Ein Entropiepool sammelt die Entropie aus den verschiedenen Entropiequellen eines Computersystems. Zum Verständnis von Entropiepools wird eine Eigenschaft von Zufallszahlengeneratoren benötigt, die im folgenden erläutert wird: Der festgelegte Wertebereich.

Auch wenn die Anforderungen an kryptographisch sichere Zufallszahlen erst im nächsten Kapitel beschrieben werden, kann hier eine intuitive Annahme schon vorab getroffen werden:

Die zu erzeugenden Zufallszahlen sollen aus einem
festgelegten Wertebereich sein.

Für einen 6-seitigen Würfel ist dieser Wertebereich $\{1, 2, 3, 4, 5, 6\}$. Für eine in Computern verwendbare Zufallszahl wird üblicherweise die Bitbreite angegeben. Bei einer Bitbreite von 256 Bit (z.Bsp. für private Schlüssel mit Elliptischer Kurven Kryptographie) ergibt sich der benötigte Wertebereich als das Intervall $[0, 2^{256}[$. Dieser Wertebereich ist groß, sogar sehr groß! Die Entropiequellen eines Computersystems haben, jede für sich betrachtet, einen nicht annähernd so großen Wertebereich. Dies wird im Folgenden anhand eines konstruierten Beispiels deutlich:

Beispiel: Wertebereich des Timings von Netzwerkpaketen:

Sei die Messung des Eingangs von Netzwerkpaketen mit einer Auflösung in Nanosekunden messbar. Das größte messbare Intervall sei 1 Sekunde. Dann ist das Wertebereichs-Intervall einer einzelnen Messung

$$[0 \text{ ns} , 10^9 \text{ ns}] = [0 \text{ ns} , 1.000.000.000 \text{ ns}] \approx [0 \text{ ns} , 2^{30} \text{ ns}]$$

Folglich hat jede pro Messung erzeugte Zahl in etwa die Bitlänge von 30 Bits. Man spricht hier von einer Entropiequelle mit 30 Bit Ausgabebreite. Das ist sehr viel weniger als die geforderte Bitlänge von 256 Bit für die Zufallszahl. Daher wird Algorithmik benötigt, um diese Entropiequelle auf den benötigten Wertebereich zu expandieren und

in den Entropiepool „einzumischen“. Diese Algorithmik kann im Kontext von Entropie-pools als „Poolmixer“ bezeichnet werden. Für das Expandieren und Mischen können Hashfunktionen verwendet werden. Hashfunktionen sind per Definition Abbildungen der Form

$$\{0, 1\}^* \rightarrow \{0, 1\}^{\text{Bitbreite } n}$$

und werden allgemein eher als Kompressionsfunktionen betrachtet. D.h. Beliebige lange Eingaben bilden auf eine feste Ausgabelänge ab. Sind die Eingaben allerdings kürzer als die Ausgabelänge kann hier auch von einer Längen-Expansion gesprochen werden. Die Expansions-Eigenschaft ist auch direkt in der Definition der Abbildung enthalten, wird allerdings wesentlich seltener im Zusammenhang mit Anwendungen von Hashfunktionen genannt und verwendet. Sind Hashfunktionen auch noch kryptographisch sicher, haben sie zusätzlich die folgenden Eigenschaften ([MVO96]):

- Erste und zweite Urbildresistenz
- Kollisionsresistenz
- Gleichverteilung über den Ausgabe-Wertebereich

Die Expansion und die Gleichverteilung sind die notwendigen Eigenschaften für die Benutzung einer Hashfunktion als Poolmixer. Die Urbildresistenzen geben Sicherheit gegen die Manipulation von Entropiequellen mit dem Entropiepool als Angriffsziel. Unter gewissen Voraussetzungen können anstelle von Hashfunktionen auch Stromchiffren oder CRC-ähnliche (Cyclic redundancy codes) Funktionen als Poolmixer verwendet werden.

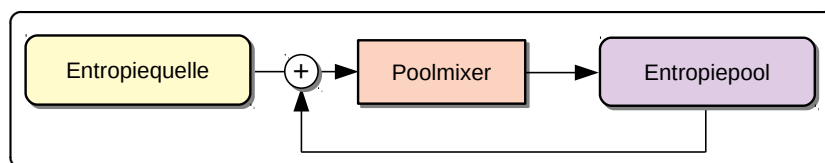


Abb. 2.2.: Entropiequelle, Poolmixer und Entropiepool

In Abbildung 2.2 ist das Zusammenspiel einer einzelnen Entropiequelle, dem Poolmixer und dem Entropiepool schematisch dargestellt. Die Entropiequelle liefert in zeitlich variablen Intervallen eine Zahl an den Poolmixer. Bevor der Poolmixer diese Zahl expandiert wird der vorherige Wert des Entropiepools mit der Zahl der Quelle verknüpft. Für die Verknüpfung existieren verschiedene, mögliche Varianten. So könnte die Verknüpfung zum Beispiel als XOR oder als Konkatination (evtl. mit Padding) ausgeführt sein.

2. Zufall und Computer

Das Wichtige ist hierbei, dass die schon vorhandene Entropie im Pool nicht überschrieben wird, also nicht verloren geht. Auf diese Weise wird er Entropiepool ständig von den Entropiequellen mit „neuer Entropie aufgefüllt“.

Einige Bemerkungen zu Entropiequellen in problematischen Umgebungen:

In manchen Umgebungen sind physikalische Entropiequellen, wie die weiter oben genannten, nicht vorhanden oder nicht algorithmisch zugänglich. So zum Beispiel oft in diesen Umgebungen:

- Mikrocontroller
- Headless server
- Script Container (z.Bsp. PHP)
- Virtuelle Umgebungen (Virtuelle Maschinen)

Weiterhin sind Solid State Drives (SSD) eine wesentlich schlechtere Entropiequelle als die ältere Technologie der Hard Disc Drives (HDD), wegen ihrer geringeren Toleranzen der Zugriffszeiten.

In solchen Umgebungen sind die Versuche, eine gute Entropiequelle zu simulieren, im besten Fall als interessant aber oft einfach als schlecht zu bezeichnen. So gibt es zahlreiche Implementierungen, die Unique Identifier (UID), Seriennummern oder MAC-Adressen als Entropiequelle benutzen. Diese Nummern sind natürlich in keinsten Weise zufällig. Daher sind für die Implementierung von kryptographischen Anwendungen in solchen Umgebungen immer die Implementierungen für die Zufallszahlengeneratoren zu kontrollieren. Es kann nicht oft genug erwähnt werden: Kein Algorithmus kann aus sich heraus Zufallszahlen erzeugen. Die Qualität der Zufallszahlen steigt und fällt mit der Qualität der Entropiequellen.

2.1.4. Anforderungen an kryptographisch sichere Zufallszahlen

In den vorherigen Kapiteln wurde schon mehrfach von kryptographisch sicheren Zufallszahlen gesprochen. Im Folgenden werden mehrere Anforderungen definiert, die als Minimum für kryptographisch sicheren Zufallszahlen erfüllt sein müssen.

Gleichverteilt (uniform):

Für Zufallszahlen wird ein Wertebereich definiert der meistens als Bitbreite der Zufallszahlen angegeben wird. So liegt eine Zufallszahl z der Länge n Bits im Wertebereich $0 \leq z < 2^n$. Die Verteilung der Zufallszahlen soll gleichverteilt über diesen Wertebereich sein. In anderen Worten: Jede Zufallszahl soll mit der gleichen Auftretswahrscheinlichkeit erzeugt werden. Dann ergibt sich die maximale Entropie.

Satz 2.1.1: Entropie-Maximum

Sei z_i eine Zufallszahl der Länge n Bit. Die maximale Entropie ergibt sich, wenn die Auftretswahrscheinlichkeiten $P(z_i) = p_i$ für alle möglichen Zahlen z_i gleichverteilt sind:

$$p_i = \frac{1}{2^n} \text{ mit } 0 \leq i < 2^n$$

Anhand einer beliebig manipulierbaren Münze lässt sich dieser Satz einfach nachvollziehen. Sei X eine Münze mit den Wurfereignissen $\{Z(ahl), K(opf)\}$. Die Auftretswahrscheinlichkeiten der beiden Ereignisse addieren sich zu $p_K + p_Z = 1$, denn Eines der beiden wird auf jeden Fall eintreten. Dann ergibt sich aus der Formel für Shannons Entropie (Definition 2.1.2):

$$H(X, p_K) = -p_K \cdot \log_2(p_K) - (1 - p_K) \cdot \log_2(1 - p_K)$$

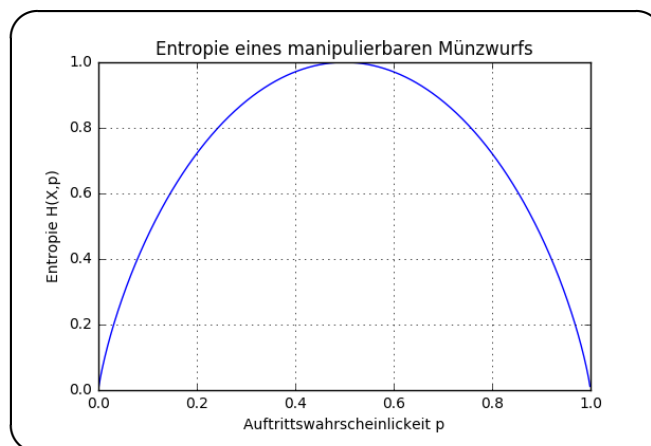


Abb. 2.3.: Entropie eines manipulierbaren Münzwurfs [KH]

2. Zufall und Computer

In Abbildung 2.3 ist diese Funktion graphisch dargestellt. Es ist erkennbar, dass das Maximum der Entropie genau bei $p_K = p_Z = 0,5$ auftritt. Also genau wenn die Ereignisse gleichverteilt sind. Weiterhin nimmt die Entropie zu den Randbereichen hin ab. Wenn nur eine Seite der Münze als Ergebnis auftreten kann, liefert das System (der Münzwurf) gar keine Entropie $H(X, p_K) = 0$.

Unvorhersehbar und nicht zurückrechenbar:

Es darf nicht möglich sein, auf Grund einer bekannten Zufallszahl oder einer Reihe von bekannten Zufallszahlen die vorherigen oder nächsten Zufallszahlen zu ermitteln.

Unbeschränkt:

Nachfolgend auf jede Zufallszahl muss eine weitere Zufallszahl erzeugbar sein. Die Anzahl der Zufallszahlen darf somit nicht beschränkt sein.

Testsuiten:

Es gibt eine Reihe von Testsuiten für Zufallszahlen. Die meisten davon haben mehrere verschiedene statistische Methoden. Diese Tests sind keine notwendige Bedingung an kryptographisch sichere Zufallszahlen, können aber eventuell Aufschluss über systematische Fehler in der Zufallszahlenerzeugung geben. Hier einige der bekanntesten Testsuiten:

- Die hard und Die harder Testsuites [Bro20]
- Maurers Universaltest [Mau92]
- NIST SP800-22 Test [ST10]
- NIST Liste von Testsuites [ST20]

2.2. Zufallszahlengeneratoren (RNG)

Um Folgen von Zufallszahlen mit den in Kapitel 2.1.4 definierten Anforderungen zu erzeugen, werden algorithmische Zufallszahlengeneratoren (Random Number Generator, RNG) verwendet. Diese Algorithmen sind deterministisch und können, wie weiter oben beschrieben, keine Entropie selbst erzeugen. Der Zufall kommt aus dem Entropiepool und wird als sogenannter Seed (Samen) in den RNG eingegeben. Der RNG kann mit einem Seed beliebig viele Zufallszahlen generieren (siehe Anforderung: Unbeschränkt). Es ist ein heiß diskutiertes Thema, wie viele Zufallszahlen ein RNG auf Basis eines Seeds maximal generieren sollte, bevor ein neuer Seed mit „frischer“ Entropie benötigt wird. In den folgenden Kapiteln wird zuerst anhand von zwei sehr einfachen Beispielen erläutert, wie schmal der Grad zwischen „Entropie-Vernichtung“ und

„Entropie-Erhaltung“ ist. Im Anschluss wird ein gewöhnlicher Pseudo-RNG (PRNG) vorgestellt (Kapitel 2.2.2) der in vielen Programmiersprachen als Bestandteil von Bibliotheken implementiert ist, aber nicht den Anforderungen an kryptographisch sichere Zufallszahlen genügt. Danach folgt eine kleine Übersicht über kryptographisch sichere Pseudo-Zufallszahlengeneratoren (CSPRNG, Kapitel 2.2.3) und ein Blick in die Implementierung des viel verwendeten Linux Zufallszahlengenerators.

2.2.1. Algorithmische Entropie-Erhaltung

Die Aufgabenstellung für die zwei folgenden Beispiele ist die algorithmische Verarbeitung von zwei aufeinanderfolgenden Würfeln eines fairen 6-seitigen Würfels. Das Ziel ist hier, die Entropie der zwei Würfelwürfe unter Erhaltung der Entropie in eine einzelne Zufallszahl zu überführen. Für beide Beispiele seien die zwei Würfelwürfe die getrennten, voneinander unabhängigen Zufallsvariablen W_a und W_b und deren Ereignisraum jeweils $\{1, 2, 3, 4, 5, 6\}$. Die Auftretswahrscheinlichkeiten sind gleichverteilt (fairer Würfel). Die Einzelereignisse werden mit w_a und w_b benannt.

Beispiel 1: Entropie-Vernichtung [KH]:

Sei der Algorithmus für die Verarbeitung der zwei Würfelwürfe eine einfache Addition:

$$RNG_{schlecht} = w_a + w_b$$

Die Verteilung der Auftretswahrscheinlichkeiten von $RNG_{schlecht}$ ist dann:

$RNG_{schlecht}$	2	3	4	5	6	7	8	9	10	11	12
$p(RNG_{schlecht})$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

Die Entropie von $RNG_{schlecht}$ ist:

$$\begin{aligned}
H(RNG_{schlecht}) &= 2 \frac{1}{36} \log_2(36) + 2 \frac{2}{36} \log_2(18) \\
&\quad + 2 \frac{3}{36} \log_2(12) + 2 \frac{4}{36} \log_2(9) \\
&\quad + 2 \frac{5}{36} \log_2\left(\frac{36}{5}\right) + 1 \frac{6}{36} \log_2(6) \\
&\approx 3,27 \text{ Bit}
\end{aligned}$$

Die Ausgangsentropie von $RNG_{schlecht} \approx 3,27$ Bit liegt damit nur knapp über der Entropie eines einzelnen Würfelwurfs (2,58 Bit). Der Grund für diese geringe Ausgangsentropie liegt in der Verteilung der Ausgangsereignisse, diese sind nicht gleichverteilt. Die Zahl Sieben tritt in diesem RNG sehr viel wahrscheinlicher auf, als die Zahlen Zwei oder Zwölf.

2. Zufall und Computer

Beispiel 2: Entropie-Erhaltung [KH]:

Im zweiten Beispiel sei der Algorithmus für die zwei Würfelwürfe nun die Konkatenation der Einzelereignisse zu einer Zufallszahl mit der Zahlenbasis 6:

$$W_{gut} = w_a || w_b$$

Der Wertebereich dieses RNG umfasst nun 36 mögliche Zahlen und die Auftretenswahrscheinlichkeiten sind gleichverteilt:

RNG_{gut}	11	12	13	14	15	...	62	63	64	65	66
$p(RNG_{gut})$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$...	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$

Die Entropie von RNG_{gut} ist:

$$\begin{aligned} H(RNG_{gut}) &= \sum_{i=1}^{36} \frac{1}{36} \log_2(36) \\ &\approx 5,17 \text{ bit} \end{aligned}$$

Anmerkungen:

In Beispiel 2 wird die Entropie der zwei unabhängigen Würfelwürfe vollständig erhalten und in einer resultierenden Zufallszahl abgebildet. Ein einzelner Wurf enthält $\approx 2,58$ Bit Entropie und RNG_{gut} gibt die doppelte Entropie $\approx 5,17$ Bit aus. Erreicht wird dies durch die erhaltene Gleichverteilung der Ergebnisse. Es ist also wünschenswert einen möglichst großen, gleichverteilten Wertebereich durch die Algorithmik eines RNG zu erzeugen. Dies entspricht auch den Anforderungen an kryptographisch sichere Zufallszahlen aus Kapitel 2.1.4. In einer umgekehrten Argumentation stellt sich also die Frage, ob ein nicht gleichverteilter Wertebereich (sowohl von Entropiequellen als auch von RNGs) immer zu geringerer Entropie führt. Diese Frage wird in Kapitel 2.2.3 zum Thema „Blockierend oder nicht?“ nochmals aufgegriffen.

2.2.2. Pseudo-Zufallszahlengeneratoren (PNRG)

Nachdem die Beispiele aus dem vorherigen Kapitel die Notwendigkeit für die algorithmische Erhaltung der Entropie und die daraus resultierende Notwendigkeit für die Gleichverteilung über den Wertebereich eines RNG gezeigt haben, wird nun die nächste Kategorie von algorithmischen RNG betrachtet, die sogenannten Pseudo-RNG (PRNG) Algorithmen. Diese können im besten Fall die Anforderungen der Gleichverteilung und der Unbeschränktheit erfüllen, scheitern aber meistens an der Unvorhersehbarkeit (und auch an der Zurückrechenbarkeit).

2.2. Zufallszahlengeneratoren (RNG)

Wiedermal wird hierzu das Beispiel eines PRNG betrachtet, im diesem Fall ein linearer Kongruenzgenerator (Linear congruence generator, LCG). Die formale Definition eines LCG erfolgt induktiv:

$$s_0 = \text{Seed}$$

$$s_{i+1} = a \cdot s_i + b \mod m, \text{ mit } i \in \mathbb{N}$$

Die Parameter a, b, m sind ganzzahlige Konstanten. Der Startwert der Zahlenfolge wird als Seed bezeichnet. Als Seed könnte eine Zahl aus einer, weiter oben beschriebenen, Entropiequelle verwendet werden. Die Ausgabe-Zahlenfolgen von linearen Kongruenzgeneratoren wiederholen sich periodisch. Durch die Kenntnis weniger Zahlen aus einer Folge sind die Parameter und der Seed einfach berechenbar. Dies ist anhand eines einfachen Beispiels schnell zu erkennen. Sei ein linearer Kongruenzgenerator $PRNG_{LCG}$ mit den Parametern $a = 5, b = 7, m = 8, s_0 = 2$ definiert als:

$$s_0 = 2$$

$$s_{i+1} = 5 \cdot s_i + 7 \mod 8$$

Dann ist die Folge der ersten 14 Zahlen dieses $PRNG_{LCG}$:

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
2	1	4	3	6	5	0	7	2	1	4	3	6	5	0	7

Die Periodenlänge dieses LCG beträgt 8 und durch das Modul $m = 8$ ist dies auch die maximale Länge. Die Bestimmung der Parameter für die maximale Länge erfolgt nach dem Hull–Dobell Theorem [HD62]. Damit erfüllt dieser LCG die Anforderung an die Gleichverteilung über den Wertebereich. Trotzdem ist ein solcher PRNG nicht für die Erzeugung kryptographisch sicherer Zufallszahlen geeignet. Die generierten Zahlen sind sowohl einfach vorhersehbar als auch zurückrechnenbar. Direkt erkennbar ist, dass bei Kenntnis einer vollen Periode alle weiteren Perioden offenliegen. Im Allgemeinen reichen durchschnittlich 4 aufeinander folgende Ausgabewerte um alle Parameter eines LCG zu bestimmen [Sch01]. Angriffe und Angriffsmethoden auf Kongruenzgeneratoren sind vielfältig bekannt und reichen bis zu Arbeiten von James Reeds aus dem Jahr 1977 [Ree77] und Joan Plumstead (1982, [Plu82]) zurück.

Außer Kongruenzgeneratoren gibt es noch viele weitere PRNGs, die auch nicht kryptographisch sicher sind. Eine weitere Beispielkategorie hierfür sind die linearen und rückgekoppelten Schieberegister. Auch diese Generatoren sind nicht für kryptographische Anwendungen geeignet.

2.2.3. Kryptogr. sichere Zufallszahlengeneratoren (CSPRNG)

Für die Konstruktion von kryptographisch sicheren Zufallszahlengeneratoren (CSPRNG) fehlt bisher noch die Erfüllung der Unvorhersehbarkeit und der Nicht-Zurückrechenbarkeit. Hier kommen sogenannte Einwegfunktionen zum Einsatz. Deren Kerneigenschaft besteht aus der effizienten und schnellen Berechnung eines Ausgabewertes, ohne die Möglichkeit, auch effizient und schnell den Eingabewert zurückrechnen zu können. Weiterhin sollte eine Gleichverteilung über den Wertebereich gesichert sein. In der Konstruktion von CSPRNGs häufig verwendete Einwegfunktionen sind Stromchiffre, Hashfunktionen und speziell für CSPRNGs entwickelte Einwegfunktionen. Einige bekannte CSPRNG Beispiele sind im Folgenden erläutert.

ChaCha20 [KH]:

ChaCha20 ist ein Stromchiffre Algorithmus (Stream Cipher) von Daniel J. Bernstein [Ber08a]. Es gibt eine ganze Familie von ChaCha Algorithmen, welche alle auf Weiterentwicklungen der Salsa-Algorithmen aufbauen [Ber08b]. Der interne Zustand wird in 16 Wörtern der Länge 32 Bit vorgehalten und rundenweise weiter berechnet. Die Rundenberechnungen basieren auf den Funktionen XOR, Addition in \mathbb{Z}_{32} und Bitshift. Die Vorteile von ChaCha20 sind schnelle Berechenbarkeit und einfache Implementierbarkeit. ChaCha20 wird unter anderem in Linux seit Kernel-Version 4.8 zur Zufallszahlenerzeugung mit `/dev/urandom`, in der Google Implementierung von TLS und in OpenSSH verwendet.

Blum-Blum-Shub-Generator (BBS-Generator) [KH]:

Der Blum-Blum-Shub-Generator (BBS) ist ein Kongruenzgenerator [BBS83]. In vorherigen Kapiteln wurde zwar dringend vor der Verwendung von einfachen Kongruenzgeneratoren für kryptographische Anwendungen im Allgemeinen gewarnt, für BBS wurde aber ein Kongruenzgenerator so modifiziert, dass er als kryptographisch sicher eingestuft werden kann. BBS benutzt das schwer zu lösende Problem der Primzahlfaktorisation, wie es auch im RSA Kryptosystem verwendet wird. Die induktive Definition dieses Generators ist wie folgend:

$$x_0 = \text{Seed}$$

$$x_{i+1} = x_i^2 \mod N$$

N ist das Produkt aus zwei Primzahlen p und q ($N = p \cdot q$). Außerdem muss $N \equiv 3 \mod 4$ gelten und der seed (x_0) muss co-prim zu N sein. Die zwei Primzahlen p und q sind Teil eines Geheimnisses und dürfen einem potentiellen Angreifer nicht bekannt werden. Die hier verwendete Quadrierung in einer RSA Gruppe mit unbekannter Gruppenordnung wird in späteren Kapiteln noch häufiger auftauchen.

AES-CTR [KH]:

Der Advanced Encryption Standard (AES) ist ein, von NIST standardisierter Blockchiffre Algorithmus [IS01]. Für AES gibt es den Betriebsmodus „Counter Mode“ (CTR) mit dem AES auch als Streamchiffre genutzt werden kann. AES CTR wird zwar auch für CSPRNGs verwendet, erfüllt aber nicht alle Anforderungen an die nicht mögliche Zurückrechenbarkeit.

Hintertüren (Backdoors) [KH]:

Der Dual_EC_DRBG (Dual Elliptic Curve Deterministic Random Bit Generator) war ein von der NIST standardisierter Generator für CSPRNGs, der nach heutigem Wissen wahrscheinlich von der National Security Agency (NSA) mit einer Hintertür versehen und von RSA Security über knapp 10 Jahre hinweg als Standard Algorithmus für die Bibliothek BSAFE verwendet wurde. Erst nach den Snowden-Veröffentlichungen zog NIST den Algorithmus aus dem Standard zurück und RSA Security empfiehlt die nicht weitere Verwendung von Dual_EC_DRBG in BSAFE [Mag13].

Linux CSPRNG in /dev/urandom:

In Linux ist ein anerkannt guter und sicherer Zufallszahlengenerator als virtueller Kernel-Gerätetreiber implementiert. Dieser ist unter `/dev/random` und `/dev/urandom` im Linux System abrufbar. Damit können beliebig lange, kryptographisch sichere Zufallszahlen erzeugt werden. In Abbildung 2.4 ist dessen Schema dargestellt.

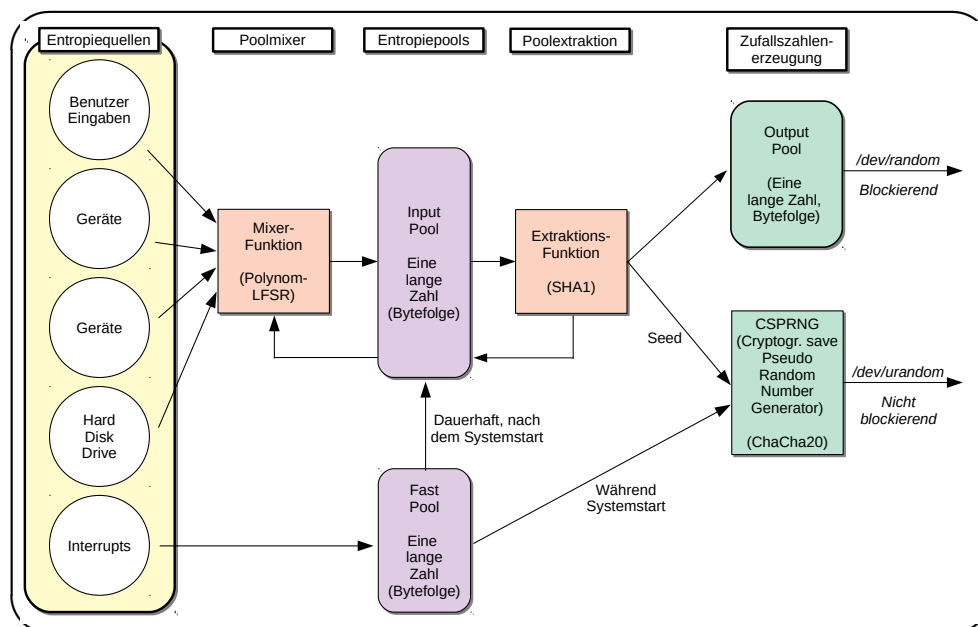


Abb. 2.4.: Schema des Zufallszahlengenerators `/dev/(u)random`

2. Zufall und Computer

In den vorherigen Kapiteln wurden einige Bestandteile dieses Schemas bereits vorgestellt. Hier ist nun erstmals die vollständige Zusammensetzung dieser Bestandteile aufgezeigt. Die Entwicklung dieses Schemas basiert hauptsächlich auf Review des Treiber-Quellcodes¹. Es existiert keine offizielle Dokumentation und die Quellcode Kommentierung ist eher auf die Benutzung ausgerichtet. Das Bundesamt für Sicherheit in der Informationstechnik (BSI) untersucht in einer Langzeitstudie seit 2012 die Sicherheit von `/dev/(u)random` in unregelmäßig aktualisierten Berichten [Sic20].

Der Kern der Implementierung besteht aus mehreren Entropiepools (Inputpool, Outputpool, Fastpool) für unterschiedliche Stufen des Generators. Der Inputpool erhält seine Entropie durch einen Poolmixer aus diversen Entropiequellen. Der Poolmixer ist hier ein lineares Rückkopplungs-Schieberegister. Am Ausgang des Inputpools werden Zahlen durch eine SHA1 Hashfunktion extrahiert. Die Extraktionsfunktion mischt einen Teil ihres Ergebnisses wieder in den Pool ein.

Neben dem Inputpool existiert ein Fastpool, der die Entropie aus Interrupttimings sammelt und viel schneller betriebsbereit ist. Dieser Fastpool gibt in der Startphase eines Systems Entropie direkt an den CSPRNG und nach mehreren Iterationen danach dann nur noch an den Inputpool. Weiterhin wird aber auch beim Herunterfahren des Systems der letzte Wert des Inputpools gespeichert und beim Systemstart wieder geladen. Diese Mechanismen zusammengenommen ermöglichen die sichere Benutzung auch während des Systemstarts.

Für die Ausgabe der Zufallszahlen gibt es zwei verschiedene Varianten. Die eine Variante (`/dev/random`) blockiert die Ausgabe von Zufallszahlen, wenn nicht mehr genügend Entropie im Pool verfügbar ist. Und die andere Variante (`/dev/urandom`) gibt kontinuierlich weiter Zufallszahlen aus, ohne zu blockieren.

`/dev/random:`

Bis zum letzten Verarbeitungsschritt in Abbildung 2.4 ist die Zahlenverarbeitung bei der Varianten algorithmisch gleich. In der Variante `/dev/random` wird eine Maßzahl für die Menge der im Inputpool vorhandenen Entropie berechnet. Diese Maßzahl ist eine einfache Integer-Variable, also eine Art Entropiezähler für den Inputpool. Wenn neue Entropie aus einer Quelle eingemischt wird, erhöht sich der Wert dieser Variable. Wenn Entropie entnommen wird, verringert sich ihr Wert. Soll über den Gerätetreiber eine Zufallszahl erzeugt werden und der Entropiezähler ist zu klein, wird die Ausführung blockiert, bis wieder ausreichend Entropie verfügbar ist. Die Ausgabe hat noch einen kleinen Outputpool vorgeschaltet. Dieser ist aber nur eine Art Puffer.

¹<https://github.com/torvalds/linux/blob/master/drivers/char/random.c>

/dev/urandom:

Die Ausgabe mit `/dev/urandom` ist nicht blockierend. Zwischen dem Inputpool und der Ausgabe durch den Gerätetreiber steckt ein CSPRNG, der kontinuierliche, unbeschränkte Zufallszahlenfolgen generiert. Der Inputpool dient hierbei zur Erzeugung der Seeds. Aktuell wird der CSPRNG ca. alle 5 Minuten mit einem neuen Seed initialisiert. Als CSPRNG wird hier der ChaCha-20 Algorithmus verwendet.

Blockierend oder nicht blockierend?

Um die Frage, welche der Beiden Implementierungen für die Erzeugung kryptographisch sicherer Zufallszahlen verwendet werden sollte gibt es viele Diskussionen [Hüh20] [Ber14] [PP14] und einen empfehlenswerten Konferenzvortrag von Filippo Valsorda [Val15]. Im Folgenden werden einige der Argumentationen aufgezählt. Der Autor dieser Thesis ist Befürworter der ausschließlichen Nutzung des nicht blockierenden `/dev/urandom`.

Das Messen der eingemischten Entropie ist ungenau. Die Inkrementierung des Entropiezählers erfolgt durch Schätzungen. Satz 2.1.4 zeigt, dass die maximale Entropie nach Shannon aus der Gleichverteilung der Ereignisse entsteht. Die Entropiequellen sind aber nicht gleichverteilt. Die Schätzungen über die enthaltene Entropie sind daher nichtmal annähernd korrekt. Bei der Entnahme von Zufallszahlen aus einem Entropiepool „verschwindet“ keine Entropie. Die Entropie wird nicht durch die Entnahme verringert. Die Rückkopplung des SHA1-Hashwerts erhält die Entropie im Pool.

ChaCha-20 und Hashfunktionen sind kryptographisch sicher und werden ohne ständiges re-seeden in vielfältigen kryptographischen Implementierungen eingesetzt. Es gibt keine vernünftige Begründung, hier eine Ausnahme zu machen. Wenn diese Ausnahme notwendig wäre, müsste ein Großteil der aktuellen kryptographischen Verfahren als unsicher eingestuft werden.

Blockaden leiten Entwickler zu unsicheren „Workarounds“ an. So ist eine der am meisten beachteten Reddit-Antworten für die Nutzung von `/dev/random` der Vorschlag, den Seed für `/dev/random` einfach mit `/dev/urandom` zu erzeugen, was offensichtlich nicht mehr Entropie erzeugt. Dafür kann dann auch gleich die nicht blockierende Variante verwendet werden. In anderen Implementierungen werden bei einer Blockade dann doch wieder Seriennummern, UUIDs und ähnlich schlechte Entropiequellen verwendet.

2.2.4. Spezielle Zufallszahlenerzeuger

Eine eigene Kategorie von Zufallszahlenerzeugern sind Hardware-RNG (HW-RNG), auch gerne als True-RNG (TRNG) bezeichnet. In solchen Hardware-Modulen wird immer eine physikalische Entropiequelle verwendet. Ein geläufiges Beispiel ist `RDRAND` von Intel. Oft liegt die Sicherheit des HW-RNG aber in der Geheimhaltung der Konstruktions-

2. Zufall und Computer

/Funktionsweise. Ist ein solcher HW-RNG in einem Linuxsystem verfügbar, wird damit der initiale Vektor für SHA1 in `/dev/(u)random` erzeugt.

Eine weitere, faszinierende Möglichkeit zur Zufallszahlenerzeugung sind Quanteneffekte und Quantencomputer. Es gibt schon seit Jahren kommerziell verfügbare Quanten-RNG (QRNG) als USB Gerät, Erweiterungs-Chip oder PC-Steckkarte [Qua20]. Diese basieren auf optischen Quanteneffekten. Seit IBM [IBM20] und mittlerweile auch D'WAVE [DWA20] einige ihrer Quantencomputer zur allgemeinen Benutzung geöffnet haben, lassen sich echte Zufallszahlen auch für Endanwender mit Quantencomputern erzeugen. Ein Kommilitone an der Hochschule RheinMain arbeitet gerade an einer Abstraktions-Bibliothek, um diese Quantencomputer in Python nutzbar zu machen [Hec20].

Webseiten bieten vielfältige Entropiequellen auch unter der Nutzung von Application Programming Interfaces (API) an. Eine bekannte Webseite hierfür ist `random.org` ². Dort werden die Zufallszahlen aus atmosphärischem Rauschen erzeugt.

²<https://www.random.org/>

3. Verifiable delay functions

In den vorherigen Kapiteln wurde ausführlich die lokale Erzeugung Zufallszahlen besprochen. In diesem Kapitel wird diese Aufgabenstellung erweitert. Ohne eine einzelne, lokale Stelle, an der Zufallszahlen erzeugt werden können, wird diese Aufgabe schwieriger. Zuerst werden hierfür dezentrale, verteilte Netzwerke und deren derzeit prominenteste Entsprechungen kurz vorgestellt (Kapitel 3.1). Die Rede ist hier von Blockchain-Technologien und deren Anwendungen in Form von kryptographischen Währungen (z.Bsp. Bitcoin und Ethereum). Die Problematik von „Proof-of-Work (PoW)“ und die resultierende Notwendigkeit für eine neue Zufallsfunktion wird in Kapitel 3.2 erläutert. Nachfolgend wird das Konstruktionsprinzip von „Verifiable delay functions (VDF)“ anhand von Beispielen in den Kapiteln 3.3 und 3.4 hergeleitet. Abschließend erfolgt die Auswahl der VDF Bestandteile für die Implementierung (Kapitel 3.5).

3.1. Dezentrale, verteilte Systeme

In verteilten und dezentralen Netzwerken gibt es keine zentrale, bestimmende Instanz. Alle Teilnehmer des Netzwerks sind gleichberechtigt. In Abbildung 3.1 ist der Vergleich zwischen einem zentralisierten und einem dezentralen Netzwerk abgebildet.

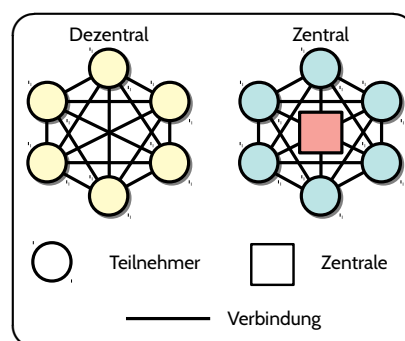


Abb. 3.1.: Dezentrales und zentrales Netzwerk

Die abgebildeten Netzwerke sind vollständig verbunden, also hat jeder Teilnehmer eine Verbindung zu jedem Anderen. Das ist nicht zwingend notwendig, es können auch Verbindungen fehlen, solange das Netzwerk dadurch nicht aufgeteilt wird. Um zu zei-

3. Verifiable delay functions

gen, dass ein solches Netzwerk nicht unbedingt immer ein Computernetzwerk sein muss, sei folgendes Beispiel gegeben:

Beispiel: Lotto in Deutschland:

Sei das Netzwerk das staatliche Lotto in Deutschland. Dann sind die Teilnehmer die Personen aus der Bevölkerung, die einen ausgefüllten und bezahlten Lottoschein abgegeben haben. Die zentrale Stelle ist die Lottogesellschaft und diese ist zuständig für die Ermittlung der Zufallszahlen, die den gewinnenden Lottoschein bestimmen. Die Lottogesellschaft ist hierfür staatlich zertifiziert, um die Korrektheit des Verfahrens zu gewährleisten. Offensichtlich ist dieses Netzwerk zentralisiert, also wie auf der rechten Seite der Abbildung dargestellt.

An diesem Beispiel wird auch schon das Problem mit Zufall in dezentralen Netzwerken deutlich. Angenommen, es gäbe die zentrale Lottogesellschaft nicht, wie sollten die Teilnehmer dann die gewinnenden Zahlen ermitteln? Per Definition sind in einem dezentralen, verteilten Netzwerk alle Teilnehmer gleichberechtigt. Die Aufgabe könnte also kein einzelner Teilnehmer übernehmen, da er damit eine zentrale Rolle innehalten und ihm die anderen Teilnehmern nicht vertrauen könnten. Eine mögliche und in der Informatik übliche Lösung hierfür ist ein festgelegtes Protokoll im Netzwerk, dass durch seine Protokolldefinition die Lösung integriert hat. In einer sehr abstrakten Betrachtungsweise sind Blockchain-Technologien und deren Hauptanwendung, die kryptographischen Währungen, solche Protokolle.

3.2. Proof-of-Work (PoW)

Seit der Veröffentlichung und der ersten Implementierung von Bitcoin[Nak09] ist dieses verteilte und dezentrale Netzwerk stabil in Betrieb. Entgegen der öffentlichen Wahrnehmung als elektronische Währung ist Bitcoin in erster Linie die Definition eines Netzwerkprotokolls für ein dezentrales, verteiltes Computernetzwerk. Der Bitcoin Client ist die implementierte Entsprechung dieses Protokolls. Ähnlich verhält es sich mit anderen kryptographischen Währungen wie Ethereum, Stellar, Monero oder Dash.

In vielen dieser Netzwerkprotokolle ist die ursprüngliche Idee der Netzwerkabsicherung durch den „Proof-of-Work (PoW)“ Algorithmus aus Bitcoin noch enthalten. Daher wird im Folgenden nur Bezug auf die Bitcoin Definition von PoW genommen. Eine detaillierte Beschreibung von PoW auf Protokollebene ist nicht Bestandteil dieser Arbeit. In einer abstrakten Betrachtung von PoW ist dieser Teil des Bitcoin Protokolls eine gewichtete, dezentrale und verteilte Zufallsfunktion. Die Teilnehmer des Netzwerks befinden sich in einem Wettbewerb um das Finden des nächsten Blocks für die Block-

chain. Dieser Wettbewerb steht allen Teilnehmern offen. Der Einsatz für die Teilnahme ist die aufgewendete, elektrische Energie jedes Teilnehmers zur wiederholten, iterativen Hashberechnung. Die eingesetzte Energiemenge entspricht der Gewichtung der Zufallsfunktion. Der Gewinner wird durch den Ausgang der Zufallsfunktion bestimmt und der Gewinn besteht im Erhalt von neu generierten Bitcoins. Hier ist eine bemerkenswerte Eigenschaft von PoW enthalten, die auf den ersten Blick nicht unbedingt auffällt. Der Einsatz für den Wettbewerb wird in der „Währung Energie“ geleistet und der Gewinn in der „Währung Bitcoin“ ausgeschüttet. Da es attraktiv ist, bei der langfristigen Preisentwicklung von Bitcoin, an diesem Wettbewerb teilzunehmen, steigt der weltweite Energieeinsatz für diesen Wettbewerb beständig weiter.

Genau an diesem Problem wird aktuell viel geforscht. Die grundlegende Idee, wieder in einer hohen Abstraktion beschrieben, ist die Ersetzung dieser Zufallsfunktion PoW durch eine andere Funktion mit weniger Energieaufwand. Wenn der Energieaufwand nicht mehr als Einsatz für den Wettbewerb, sondern nur noch für die Berechnung der Zufallszahlen selbst anfallen würde, wäre aus ökologischer Sicht schon viel erreicht. Somit kann eine erste Problembeschreibung getroffen werden:

Sei ein Computernetzwerk dezentral und verteilt, wie oben beschrieben. Es wird eine Zufallsfunktion gesucht, die ohne den Energieaufwand von Proof-of-Work die Erzeugung von Zufallszahlen in diesem Netzwerk ermöglicht. Die Zufallszahlen sollten die Anforderungen an kryptographisch sichere Zufallszahlen erfüllen (siehe Kapitel 2.1.4).

Die Idee der Erzeugung von Zufallszahlen unter sich nicht vertrauenden, gleichberechtigten Teilnehmern ist natürlich nicht erst mit Bitcoin und PoW entstanden. Schon im Jahr 1983 hat Manuel Blum ein Protokoll für zwei Teilnehmer entwickelt, die über eine Telefonleitung gemeinsam einen Münzwurf durchführen[Blu83]. Schon die Einleitung seiner Veröffentlichung bietet ein amüsantes Anwendungsbeispiel für sein Protokoll: Alice und Bob befinden sich in Scheidung und leben getrennt. Ein Münzwurf soll entscheiden, wer das gemeinsam angeschaffte Auto behalten darf. Durch die räumliche Trennung kommt als Kommunikationsmittel nur das Telefon in Betracht.

Blum setzt in seinem Protokoll die Verfügbarkeit von sicheren Einweg-Funktionen voraus und wählt hierfür ein RSA Kryptosystem. Wie später in diesem Kapitel noch zu lesen, zieht sich die Idee der Verwendung von RSA Gruppen mit geheimer (für die Teilnehmer unbekannter) Gruppenordnung bis heute weiter. Aktuelle VDF benutzen auch das RSA System als Einweg-Funktion.

Im gleichen Jahr hat Michael O. Rabin den Begriff „Random Beacon“ definiert[Rab83]. In der Übersetzung wäre dies ein Zufalls-Leuchfeuer. Dieser Begriff hält sich bis Heute für die algorithmische, verteilte Zufallszahlenerzeugung.

Vor zwei Jahren, in 2018, haben Boneh et al. die Idee von Verifiable delay functions (VDF) veröffentlicht[Bon+18]. Wenige Monate später, noch im gleichen Jahr gab es zwei konkrete Vorschläge zur Verbesserung dieser VDF[Pie18][Wes18].

3. Verifiable delay functions

Seitdem hat sich in der Forschung über VDF noch einiges mehr getan. In vorherigen Kapiteln wurde schon der Wettbewerb der VDF Alliance erwähnt[All19]. Hier hat sich die Ethereum Foundation mit anderen Kooperationspartnern zusammen zum Ziel gesetzt die Entwicklung von VDF voran zu bringen. Insgesamt wurde ein Preisgeld von 100.000\$ für die schnellste Implementierung in einem FPGA ausgelobt.

Weitere Projekte mit Bezug zu VDF und Blockchains sind die Proof-of-Stake Blockchain Algorand¹, die „League of Entropy“² und das Drand Projekt³.

3.3. VDF: Allgemeine Idee

Zur Verdeutlichung der Idee von VDF sei wieder eine Lotterie angenommen. Aber diesmal ohne eine zentrale Instanz für die Zufallserzeugung. Die Lotterie läuft in mehreren Schritten ab:

1. Schritt: Teilnahme (Setup)

Die Teilnehmer können in einem festgelegten Zeitintervall an der Lotterie teilnehmen. Jeder Teilnehmer muss hierzu innerhalb dieses Intervalls seinen Einsatz und einen möglichst zufällig gewählten String einbringen. Die Strings werden unter den Teilnehmern über das Netzwerk verteilt und zur Berechnung im nächsten Schritt aneinander konkateniert.

2. Schritt: Berechnung (Evaluation)

Eine Zufallsfunktion berechnet aus dem gemeinsamen String eine, für alle Teilnehmer zufällig wirkende Zahl. Diese Zahl bestimmt den Gewinner der Lotterie.

Hier entsteht das erste Problem. Das Zeitintervall für die Teilnahme macht ein Betrugsszenario möglich. Angenommen, alle Teilnehmer bis auf einen haben ihre Strings über das Netzwerk verteilt und es ist noch Zeit für die Teilnahme übrig. Der eine (betrügerische) Teilnehmer könnte die vorhandenen Strings dazu benutzen, möglichst viele Berechnungen mit der Zufallsfunktion alleine durchzuführen und seinen eigenen String bis zum Ablauf der Teilnahmefrist so zu manipulieren, dass seine Gewinnchancen steigen. Hat er seinen optimalen String gefunden, verteilt er diesen dann auch (noch innerhalb der Frist) an die anderen.

Die Lösung hierfür ist eine Zufallsfunktion, die eine parametrisierbare zeitliche Verzögerung (delay) in der Berechnung integriert hat. Dieses delay wird so eingestellt, dass die Berechnung länger dauert als das Teilnahmeintervall lang ist. Somit können keine

¹<https://www.algorand.com/>

²<https://www.cloudflare.com/leagueofentropy/>

³<https://developers.cloudflare.com/randomness-beacon/about/drand/>

Eingabestrings vorausberechnet werden und es darf für die Berechnung keinen schnelleren Weg geben. Die Berechnung muss also immer minimal so lange rechnen wie es der delay Parameter vorgibt. Daher kommt das Wort „delay“ in VDF.

Die Entropiequelle für die Erzeugung der Zufallszahlen ist hier der gemeinsame String, von dem jeder Teilnehmer nur seinen eigenen Anteil selbst wählen kann. Der gesamte String ist dann wie ein Seed für die Zufallsfunktion zu verstehen.

3. Schritt: Überprüfung (Verifikation)

Steht der Gewinner durch die Erzeugung der Zufallszahl fest, sollen alle Teilnehmer überprüfen können (verifizieren), ob der richtige Gewinner ermittelt wurde. Es muss also die Möglichkeit für alle Teilnehmer existieren, das Ergebnis der Zufallsfunktion eigenständig zu verifizieren. Eine denkbare Methode wäre, dass die Teilnehmer die Berechnung aus Schritt 2 nochmals durchführen. Das würde aber wieder solange dauern, wie durch das delay angegeben wurde. Wenn dieses delay aber ein sehr langer Zeitraum ist, zum Beispiel Tage oder Wochen, macht diese Verifikationsmethode keinen Sinn. Es wird daher eine Zufallsfunktion benötigt, die effizient zu verifizieren ist. Daher kommt das Wort „verifiable“ in VDF.

Alle Schritte zusammen betrachtet wird also zur Durchführung dieser dezentralen, verteilten Lotterie eine „Verifiable delay function,“ benötigt.

3.4. Konstruktionsprinzip

Für das folgende Konstruktionsprinzip einer VDF wird die Idee aus den oben genannten Veröffentlichungen von Boneh, Pietrzak und Wesolowski verwendet[Bon+18][Pie18][Wes18]. Hierfür wird in einer vertrauenswürdigen und geheimen Vorberechnung eine RSA Gruppe definiert:

Geheime Vorberechnung:

Sei $n = p \cdot q$ ein RSA Integer mit geheimer Gruppenordnung. Das Modul n ist öffentlich, die Primzahlen p und q sind geheim. Daher ist auch die Gruppenordnung $\Phi(N) = (p - 1)(q - 1)$ nicht öffentlich.

Setup:

Während der Setup Phase werden die folgenden Aktionen durchgeführt:

- Sei m der konkatenierte shared Input (String).
- Sei $x \in \mathbb{Z}_n^*$ ein Element aus der Gruppe mit $H(m) = x$ (in die Gruppe hashen).
- Sei $t \in \mathbb{N}$ ein Zeitparameter.

3. Verifiable delay functions

Evaluation:

Die Phase der Evaluation besteht aus der t -fach wiederholten Berechnung des Quadrats von x in der RSA Gruppe, also

$$y = x^{2^T} \mod N$$

Ohne Kenntnis der Gruppenordnung ist diese Berechnung nur iterativ lösbar:

$$x \rightarrow x^2 \rightarrow x^{2^2} \rightarrow x^{2^3} \rightarrow \dots \rightarrow x^{2^T} \mod n$$

Wenn die Gruppenordnung $\Phi(N) = (p-1)(q-1)$ bekannt wäre, dann könnte die Berechnung wie folgend abgekürzt werden:

$$e = 2^T \mod \Phi(n)$$

$$y = x^e \mod n$$

Mit dem Parameter t kann die Anzahl der iterativen Berechnungen (Quadrierung und Modulo) eingestellt werden. Damit ist t der **delay** Parameter der VDF. Die reale Zeit für die Berechnung hängt auch noch von der Geschwindigkeit der Implementierung ab. Da durch das **delay** der Betrug in der Setup Phase verhindert werden soll, sind möglichst schnelle Implementierungen für die Berechnung notwendig.

Verifikation:

Bis zum letzten Schritt (Evaluation) könnte auch argumentiert werden, dass eine Hashfunktion auch funktionieren würde. So zum Beispiel die t -fache Ausführung einer Hashfunktion H auf den String aus dem Setup:

$$y = \underbrace{H(H(H(\dots H(m))))}_{t\text{-faches Hashen}}$$

Die Bedingungen an eine VDF enthalten aber auch die Möglichkeit zur effizienten Verifikation des Ergebnisses. Und für die Verifikation des letzten Hashwerts gibt es keine Möglichkeit, außer alle Hashberechnungen erneut durchzuführen.

Eine weitere Idee wäre, die Gruppenordnung der RSA Gruppe aufzudecken und die Berechnung wie oben gezeigt abzukürzen. Dann müsste aber für jede Verwendung der VDF eine neue Gruppe erzeugt werden. Die VDF soll aber mehrfach benutzbar sein, ohne jedes mal geheime Vorausberechnungen machen zu müssen.

Die umsetzbare Idee ist letztendlich das Anlegen eines Beweises, parallel zur Evaluation. Für diesen Beweis wird das Teile-und-herrsche Prinzip angewendet. Diese Beweisart stammt aus der Veröffentlichung von Pietrzak. Die Beschreibung dieses Beweises erfolgt in mehreren Schritten:

Definition von Beweiser, Verifizierer und der zu beweisenden Berechnung:

- Sei P der Beweiser und V der Verifizierer.
- P will beweisen, dass sein Tupel (x, y) die Gleichung $y = x^{2^T}$ erfüllt.

Erster Schritt:

- P sendet ein Zwischenergebnis nach der Hälfte der Berechnungen: $\mu = x^{2^{T/2}}$
- V verifiziert: $y = \mu^{2^{T/2}}$

Nach diesem Schritt muss der Verifizierer V nur noch die Hälfte der Berechnungen ausführen, also nur noch $T/2$ Berechnungen um sich von der Korrektheit der gesamten Berechnung zu überzeugen. Die nächsten Schritte werden durch das Teile-und-herrsche Prinzip und eine geschickte Randomisierung der Berechnung der Zwischenergebnisse erreicht:

- Der Verifizierer V randomisiert die Berechnung durch die Zufallszahl r .
- $(x', y') = (x^r \cdot \mu, \mu^r \cdot y)$
- Dann ist $y' = x'^{2^{T/2}}$ nur wahr, wenn das korrekte μ gesendet wurde und $x = x^{2^T}$ erfüllt ist.
- Die Halbierung wird $\log(T)$ mal fortgesetzt, bis $T = 1$.
- Bei $T = 1$ kann V dann effizient selbst nachrechnen.

Es gibt durch die Randomisierung mit r eine sehr kleine Wahrscheinlichkeit, dass die Zwischenberechnungen auch mit falschen Zwischenwerten als korrekt verifiziert werden würden. Diese Wahrscheinlichkeit sinkt mit jeder Teilung und einem neuen r . Weiterhin ist der Beweis hier als interaktive Variante zwischen V und P aufgeführt. Mittels Fiat-Shamir-Transformation kann dieser Beweis auch Nicht-interaktiv geführt werden. Die Beschreibung dieser Varianten ist nicht Bestandteil dieser Arbeit.

3.5. Auswahl für die Implementierung

Für die FPGA Implementierung des Beschleunigers wird das oben genannte Quadrieren und Modulberechnen als Aufgabenstellung aus den VDF übernommen, analog zum Wettbewerb der VDF Alliance. Dies ist auch genau der Teil von VDF, der so schnell wie möglich ausgeführt werden muss.

4. Hardware

Im weitesten Sinne ist Hardware nur ein materielles, anfassbares Gut. In der Informatik ist die Entsprechung von Hardware in der klassischen Betrachtung die Maschine auf der die Software ausgeführt wird. Umgangssprachlicher formuliert ist Hardware (in der Informatik) all das, was einen Stromstecker hat und Berechnungen durchführen kann. Natürlich ist dieses Bild sehr einfach gezeichnet und manch einer mag argumentieren, dass Aktoren, Sensoren, Bildschirme, usw. nicht selbstständig berechnen und trotzdem in die Kategorie Hardware gezählt werden können. Aber letztendlich steckt irgendwo doch eine Central Processing Unit (CPU), eine Micro Controller Unit (MCU), ein Digital Signal Processor (DSP) oder ein anderes Rechenwerk mit in der Hardware. Die praktische Aufgabe der Informatik war und ist seit Jahrzehnten die Entwicklung von Programmcode für diese Hardware, nämlich die Software.

Die Entwicklung der Hardware selbst war vornehmlich und historisch betrachtet im Fachbereich der Elektrotechnik und der Physik zu finden. Dies hat sich innerhalb des letzten Jahrzehnts verändert. Durch Programmierung konfigurierbare Logik-Schaltkreise sind mittlerweile sowohl günstig als auch leistungsfähig verfügbar und haben in der akademischen Ausbildung und Forschung auf breiter Basis Einzug gefunden. Die aktuelle Technologie sind Field Programmable Gate Arrays (FPGA) und das Einstiegssegment an FPGAs ist schon für zweistellige Euro-Beträge erhältlich¹. Immer häufiger werden Kursangebote zur Programmierung dieser FPGAs in der akademischen Informatikausbildung angeboten. Der zu verfolgende Trend ist, dass sich die Informatik mit Mitteln und Methoden der Softwaretechnik die Hardware selbst erstellt:

Die Erstellung von Hardware ist zu einer
Software-Aufgabe in der Informatik geworden!

An der Hochschule RheinMain wird die Programmierung von FPGAs seit 2013 als Wahlfach in den Informatikstudiengängen angeboten und ist seit 2016 fester Bestandteil des Lehrplans im Studiengang „Informatik - Technische Systeme“ [Rei20][Rhe20].

¹<https://www.latticesemi.com/icestick>

4. Hardware

In der reinen Softwareentwicklung sind die Werkzeuge mittlerweile in der Breite öffentlich verfügbar und in vielen Betriebssystemen fest integriert. So zum Beispiel:

- Editoren und Entwicklungsumgebungen (IDE)
- Compiler, Interpreter, Debugger und Testumgebungen
- Container, Virtuelle Umgebungen
- Betriebssysteme

Diese Werkzeuge (Tools) und Werkzeugketten (Toolchains) sind vermehrt unter Open Source Lizenz veröffentlicht. Eine große und wachsende Gemeinschaft von Softwareentwicklern kümmert sich hier um die Verbesserung, Wartung, Fehlerbehebung, Dokumentation und Veröffentlichung dieser Toolchains. Endanwender profitieren hier von der größtenteils kostenfreien Nutzung und der Verfügbarkeit von Informationen. Einige bekannte Beispiele sind das Betriebssystem Linux, die Officesuite LibreOffice, Programmiersprachen wie C, Java, Python und die Compiler gcc, clang. Diese Liste ließe sich fast beliebig lang erweitern.

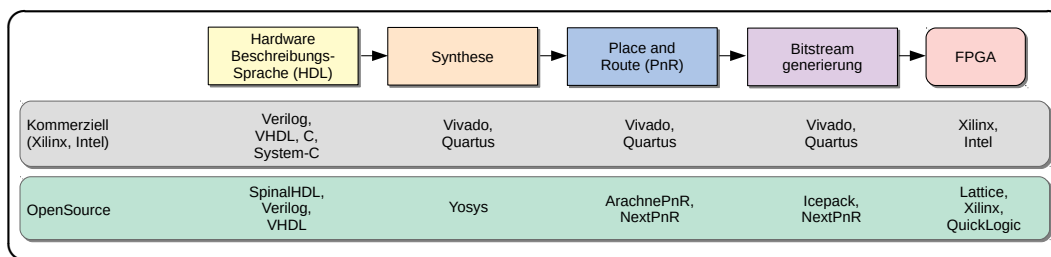


Abb. 4.1.: Vereinfachte Darstellung vom Workflow und den Toolchains für FPGA

In Abbildung 4.1 sind die Arbeitsschritte (Workflow) der FPGA-Programmierung in einer vereinfachten Darstellung aufgezeigt und mit Beispielen aus den kommerziellen und Open Source Umgebungen versehen. Der Quellcode wird in einer Hardwarebeschreibungssprache (HDL) geschrieben. Dieser Code durchläuft die Synthese, das Platzieren, das Routen und die Generierung des Bitstreams. Der Bitstream wird dann auf den FPGA-Chip geladen und konfiguriert den FPGA zur gewünschten Funktionalität.

In der Hardwareentwicklung für FPGAs ist die Welt der als Open Source verfügbaren Tools und Toolchains noch längst nicht so weit fortgeschritten wie in der Softwareentwicklung. Im Jahr 2015 wurde die weltweit erste Open Source Toolchain für die Programmierung von FPGAs veröffentlicht (Projekt IceStorm[WL]). Initial war nur die Programmierung einiger, weniger FPGA-Chips des Herstellers Lattice Semiconductors damit möglich. Die FPGA-Chips wurden hierfür „reverse engineered“ und damit das Bitstream Format herausgefunden und veröffentlicht. Üblicherweise und nach wie vor sind bei fast allen bekannten FPGA-Herstellern diese Bitstream-Dokumentationen

unter Verschluss und können nur mit proprietären, lizenz- und kostenpflichtigen Toolchains programmiert werden (z.Bsp. Xilinx Vivado und Intel Quartus). Das Innenleben der FPGA-Chips und damit auch des Bitstream Formats sind eine Art Betriebsgeheimnis, mit dem die Endanwender an die Hersteller Toolchains, die damit verbundenen Verschwiegenheitserklärungen (NDA), die Lizenzmodelle und -kosten gebunden sind. Daher hat das Projekt IceStorm in den letzten Jahren weltweit wachsende Beachtung und eine solide Entwicklergemeinschaft gefunden. Aus der ursprünglichen Gruppe des Projekts hat sich das Unternehmen Symbiotic EDA gegründet² und die Entwicklung der Toolchain wird durch die kollaborative Projektgruppe Symbiflow fortgeführt³. Die Liste der unterstützten FPGA-Plattformen wächst beständig und beinhaltet mittlerweile nicht mehr nur Lattice FPGAs. Bemerkenswert sind hier die Arbeiten an der Unterstützung der Xilinx 7-Serie FPGAs. Weiterhin hat die Firma QuickLogic als erster FPGA-Hersteller das Bitstream Format seiner FPGA-Chips offengelegt und damit die Nutzung von Open Source Tools ermöglicht⁴. Es besteht die Hoffnung, dass dieses Beispiel weitere Hersteller dazu anregt, ihre Formate auch zu öffnen.

Die Programmierung von FPGAs erfolgt in Hardwarebeschreibungssprachen (HDL). Die zwei Sprachen Verilog und VHDL (Very High Speed Integrated Circuit Hardware Description Language) sind standardisiert und die wahrscheinlich meistbenutzten HDL. Es existieren viele Projekte, um diese zwei Sprachen abzulösen, zu ergänzen oder zu erweitern. Einige Beispiele hierfür sind C, Chisel(Scala), MyHDL(Python) und SpinalHDL(Scala). Vom Autor dieser Masterthesis liegt eine Bachelorthesis aus 2016 vor, in welcher die FPGA Programmierung in der Sprache C untersucht und evaluiert wurde[Kno16]. Die Programmierung in C erfolgte mit der proprietären Vivado Toolchain und ist daher ausschließlich auf Xilinx FPGAs lauffähig. Für die Implementierung in dieser Arbeit wurde SpinalHDL als Sprache ausgewählt. Einen Einblick in die Begründung dieser Auswahl wird in Kapitel 4.1 zusammen mit dem allgemeinen Design gegeben. Die Details der Implementierung sind in Kapitel 4.2 beschrieben. Testen und debuggen gestaltet sich in der Hardwareentwicklung für FPGAs unterschiedlich zu reiner Softwareentwicklung. Kapitel 4.3 erläutert die Methoden und Lösungen hierzu. Aus der Implementierung wurde ein Demonstrator gebaut, dessen Beschreibung in Kapitel 4.4 zu finden ist.

²<https://www.symbioticeda.com/>

³<https://symbiflow.github.io/>

⁴<https://www.quicklogic.com/products/eos-s3/>

4.1. Design

4.1.1. Funktionalität

Hardwarebeschleuniger:

Es wird ein Hardwarebeschleuniger in einem FPGA implementiert. Die Funktionalität der Implementierung ist die in Kapitel 3.5 ausgewählte, wiederholte Quadrierung in einer RSA Gruppe. Sei G_{RSA} eine RSA Gruppe mit dem Modul n und geheimer Gruppenordnung. D.h. Die Primfaktoren p und q mit $n = p \cdot q$ wurden in einem geheimen, sicheren Setup bestimmt und sind, genauso wie die Gruppenordnung, nicht bekannt. Als Eingabeparameter für die Berechnung werden definiert:

- Gruppenelement $x \in G_{RSA}$
- Modul n
- Zeitparameter t

Die zu lösende Berechnung ist dann $y = (x^2)^t \bmod n$ und kann nur iterativ in t Schritten berechnet werden. Das berechnete Gruppenelement y ist der Rückgabewert der Funktion. Der Zeitparameter t muss im Intervall $[0, 2^{32}[$ liegen, was einer Zahl der Länge vier Bytes entspricht. Die Funktion soll als Hardwarebeschleuniger ausgeführt werden. Der verwendete FPGA ist hierzu durch ein noch zu definierendes Übertragungsprotokoll ansteuerbar. Der Beschleuniger soll die Berechnung so schnell wie möglich ausführen.

Ansteuerung des Beschleunigers:

Die Ansteuerung des Hardwarebeschleunigers führt das vertrauenswürdige Setup aus und bestimmt dabei die Primzahlen p und q . Daraus wird das Modul n berechnet. Der Zeitparameter t wird festgelegt und ein zufälliges Gruppenelement x gewählt. Die Ansteuerung implementiert das Übertragungsprotokoll. In Abbildung 4.2 ist ein Sequenzdiagramm für eine vollständige Ausführung der Kommunikation zwischen Ansteuerungsgerät und Hardwarebeschleuniger dargestellt.

Die Ansteuerung kennt die geheimen Setup-Parameter p , q und kann eine schnelle Berechnung von y wie folgend durchführen:

$$\Phi(n) = (p - 1)(q - 1)$$

$$e = 2^t \bmod \Phi(n), y = x^e \bmod n$$

Weiterhin kann die Ansteuerung die iterative Berechnung selbst durchführen und die Laufzeiten sowohl des Beschleunigers als auch ihrer eigenen, iterativen Berechnung

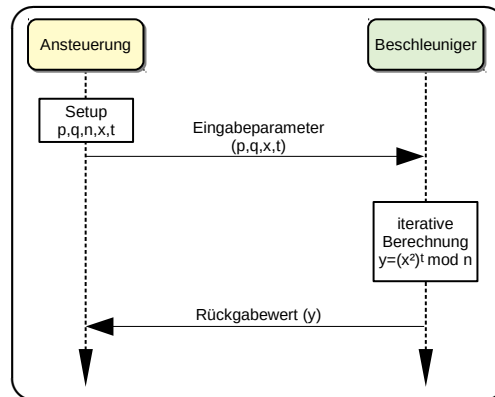


Abb. 4.2.: Sequenzdiagramm für Ansteuerung und Beschleuniger

messen und ausgeben. Nach Erhalt des berechneten Rückgabewerts y vom FPGA wird ein Vergleich mit den selbst berechneten Werten (schnell und iterativ) durchgeführt und eine Angabe über die Korrektheit ausgegeben. Die gesamte Sequenz soll wiederholt ausführbar sein.


4.1.2. Komponenten: Maximal Open Source

Eines der Ziele dieser Arbeit ist die maximale Verwendung von Open Source Bestandteilen (siehe Kapitel 1.1). Wie schon beschrieben, hat sich diese Möglichkeit in der Programmierung von FPGA Plattformen erst in den letzten Jahren eröffnet. Die Einschränkungen bestehen hier hauptsächlich noch in der Leistungsfähigkeit der verwendbaren FPGA Plattformen. Daher eignen sich momentan noch nicht alle FPGA Projekte für die Erstellung mit offenen Tools. Die Hardware Implementierung dieser Arbeit ist jedoch gut für dieses Ziel geeignet. Die hauptsächliche Fragestellung aus Kapitel 1.1.2 ist:

Wie weit kann das Ziel des VDF Wettbewerbs
mit Open Source Lösungen erreicht werden?

Dieser Ansatz erlaubt und erfordert die Skalierung der Implementierung in einer generischen und parametrisierbaren Form. Ein Teil des Ergebnisses ist dann die erreichte Größenordnung der Implementierung im Vergleich zu der Größenordnung der nicht mit Open Source erstellten Implementierungen aus dem Wettbewerb. Daher ist die folgende Auswahl der FPGA Komponenten auf die maximale Größe des FPGA ausgerichtet. In Abbildung 4.3 ist der aktuelle Stand der Funktionalität in den Symbiflow Tools abgebildet. Vollständige Funktionalität ist in „Project Icestorm“ und „Project Trellis“ verfügbar. Im zweiten Projekt (Trellis) wird der Lattice FPGA-Chip ECP5-85k unterstützt. Dies ist der leistungsstärkste FPGA, der aktuell mit Open Source Tools und vollständiger Funktionalität programmiert werden kann.

4. Hardware



	Project Icestorm	Project Trellis	Project X-Ray	QuickLogic Database
Basic Tiles:	✓	✓	✓	✓
- Logic	✓	✓	✓	✓
- Block RAM	✓	✓	✗	✓
Advanced Tiles:	✓	✓	✗	✗
- DSP	✓	✓	✗	✓
- Hard Blocks	✓	✓	✗	✓
- Clock Tiles	✓	✓	✓	✗
- IO Tiles	✓	✓	✓	✓
Routing:	✓	✓	✓	✗
- Logic	✓	✓	✓	✓
- Clock	✓	✓	✓	✗

Bildquelle: <https://symbiflow.github.io/>, Abgerufen am 12.05.2020

Abb. 4.3.: Aktueller Stand der Symbiflow Tools

Die FPGA Auswahl fällt daher auf den Lattice FPGA-Chip ECP-85k. Der FPGA-Chip benötigt Schnittstellen und Peripheriebausteine. Daher wird nicht alleine der Chip, sondern ein Entwicklungsboard mit diesem Chip benötigt. Der kroatische Hackerspace „Radiona“⁵ hat ein solches Entwicklungsboard als Open Source Projekt erstellt, genannt ULX3S. Im Rahmen einer Crowdfunding Kampagne werden diese Boards demnächst produziert⁶. Die Kampagne ging am 14. April 2020 zu Ende und die Produktion hat zum Zeitpunkt der Erstellung dieser Arbeit gerade erst begonnen. Aber es war möglich, aus einer Vorab-Serie ein ULX3S Board für diese Thesis aus der FPGA Gemeinschaft zu erhalten. Für diese Arbeit wird also eine sehr neue FPGA Toolchain (Symbiflow, Project Trellis) und ein noch nicht auf dem Markt verfügbares FPGA Board (Radiona ULX3S) verwendet.

Eine ähnliche Situation stellt sich für die Hardwarekomponente der Ansteuerung dar. Diese Aufgabe könnte jeder Laptop, ein RaspberryPi oder ein ARM-Mikrocontroller übernehmen. In all diesen Geräten arbeiten CPUs oder MCUs die unter ClosedSource erstellt und mit kommerziellen Lizenzmodellen vertrieben werden. Aber seit wenigen Jahren gibt es die offene ISA-Spezifikation RISC-V[Fou19]. Bisher sind die meisten Implementierungen für RISC-V als sogenannte Softcores implementiert und laufen dann auf FPGA-Plattformen. Aber es gibt schon einige, wenige fertige RISC-V Mikrocontroller als Chips am Markt zu kaufen. So zum Beispiel der RISC-V Dualcore Mikrocontroller

⁵<https://radiona.org/>

⁶<https://www.crowdsupply.com/radiona/ulx3s>

Kendryte K210, verbaut auf dem Entwicklungsboard Maixduino vom Hersteller Sipeed. Diese sind bisher nur in China erhältlich, aber nach Deutschland importierbar. Auch hier ist ein solches Maixduino Board für die Erstellung dieser Thesis verfügbar. Für die Datenausgabe an der Ansteuerung (Maixduino Board) ist ein einfaches LCD Display vorgesehen. Die verwendeten Komponenten für diese Arbeit als Zusammenfassung in Tabelle 4.1 und mit den wichtigsten Eigenschaften (Tabellen 4.2 und 4.3):

Komponente	Hersteller	Bezeichnung	Board
FPGA	Lattice	ECP5-85k Chip	Radiona ULX3S
MCU	Kendryte	K210 RISC-V	Sipeed Maixduino
Display	Sipeed	2,4 Zoll Farb-TFT	Für Maixduino

Tab. 4.1.: Komponenten: Übersicht

85.000 Look-Up Tables (LUTs)
156 DSP (18x18 Multiplizierer)
4 PLL (Phase Locked Loops)
669Kb verteilter RAM
3744Kb embedded Block RAM

Tab. 4.2.: Lattice ECP5-85k FPGA: Eigenschaften [Sem]

Dualcore RISC-V 64bit mit IMAFDC Extensions, 400MHz
8MB On-Chip SRAM
KPU (Neuraler Netzwerk Prozessor), 64 KPU mit 576bit
APU (Audio Prozessor), für bis zu 8 Mikrofone, 192KHz Samplerate, FFT Einheit
DVP Kamera und MCU Display Schnittstellen (24 Pins)
Hardware Beschleuniger für AES, SHA256 und FFT

Tab. 4.3.: Kendryte K210 RISC-V: Eigenschaften [CAN]

Die Komponenten werden nach der schematischen Abbildung 4.4 miteinander verbunden. Sowohl das FPGA- und das RISC-V-Board führen Chipinterne UART Schnittstellen auf die Input-/Output-Anschlüsse. Diese werden für die Datenübertragung genutzt. Das RISC-V Board hat eine 24-polige Displayschnittstelle zum einfachen Anschluss des verwendeten LCD/TFT Displays.

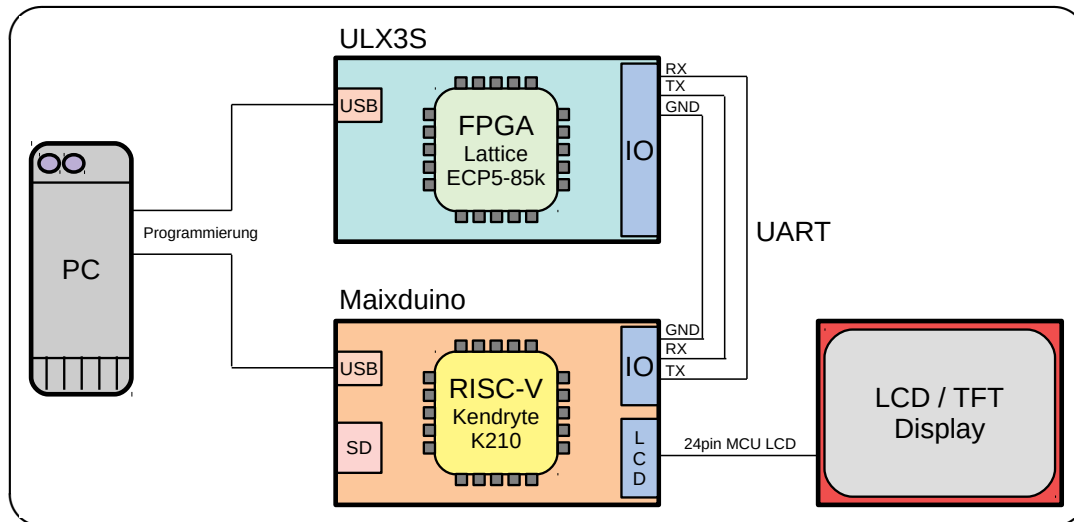


Abb. 4.4.: Verbindungsschema der Komponenten

4.1.3. HDL und Tools

Ein Ziel dieser Thesis ist die Evaluierung der neuen Hardwarebeschreibungssprache (HDL) SpinalHDL (siehe Kapitel 1.1.3). SpinalHDL ist im klassischen Sinn keine eigenständige HDL. Vielmehr ist SpinalHDL eine Scala-Bibliothek mit der die HDLs Verilog und VHDL generiert werden können. Die Entwicklung von SpinalHDL wird hauptsächlich durch Charles Papon vorangetrieben. Erst in den letzten zwei Jahren entstand langsam eine Entwickler- und Anwender-Gemeinschaft rund um SpinalHDL. Hilfreich für diese Beachtung und Anerkennung war wahrscheinlich die Implementierung des RISC-V Projekts „VexRisc“ durch C. Papon und der Gewinn des ersten Platzes im SoftCPU Wettbewerb der RISC-V Foundation im Jahr 2018[Fou]. Hier konnte sich SpinalHDL mit der VexRisc gegen andere Implementierungen und HDL-Sprachen durchsetzen. SpinalHDL verschiebt die Entwicklung von Hardware wesentlich weiter in die Richtung von Softwareentwicklung, als die meisten anderen HDLs. Es wird eine hohe Abstraktionsebene über die FPGA Programmierung angeboten, die zwar eine steile Lernkurve und das Wissen um die tatsächliche Struktur von FPGAs voraussetzt, sich dann aber fast wie eine Mischung aus Objekt-orientierter und Funktionaler Softwareentwicklung anfühlt. Der generierte Verilog und VHDL Sourcecode ist vollständig kompatibel mit deren Standards und hat den weiteren Vorteil, immer noch gut menschlich lesbar, wartbar und weiterverwendbar zu sein. Der SpinalHDL Sourcecode wird in Scala unter Einbindung der Bibliothek SpinalHDL geschrieben und bei Ausführung des Codes entstehen Verilog und/oder VHDL Projekte. Diese werden dann durch die FPGA Toolchains bis zum Bitstream weiter verarbeitet.

Wie weiter oben beschrieben, wird durch den Fokus auf Open Source Tools und die Verwendung des größtmöglichen FPGA die folgende Toolchain (und der Workflow) zur Verarbeitung der HDL bis zum Bitstream vorgegeben:

- SpinalHDL: Scala zu Verilog
- Yosys: Synthese von Verilog zu Netzliste
- NextPnr: Place and Route der Netzliste für den ECP5-85k FPGA
- NextPnr: Erzeugung des Bitstreams für den ECP5-85k FPGA
- Ujprog: Upload Tool Bitstream zu FPGA Speicher

Die Buildsysteme der gesamten Toolchain basieren unter Linux entweder auf dem Scala-Buildsystem SBT (SpinalHDL) oder Makefiles (für den ganzen Rest). D.h. Es müssen keine aufwändigen, grafiklastigen GUI-Tools verwendet werden. Der Bau eines vollständigen Projekts ist, bis zum letzten Schritt auf den FPGA, einfach und schnell automatisierbar.

RISC-V:

Auf der Seite der Ansteuerung durch den RISC-V Kendryte K210 gestaltet sich die Auswahl der Tools wesentlich übersichtlicher. Auf dem K210 ist bei Auslieferung ein MikroPython System installiert⁷. Die Standard Bibliotheken sind vollständig implementiert und verwendbar. MikroPython ist in C99 geschrieben, wurde speziell für Mikrokontroller entwickelt und gilt als ähnlich performant wie C selbst. Das hochladen und ausführen von Sourcecode wird über ein spezielles Tool vom Hersteller des Entwicklungsboards ermöglicht. Die ausführbare Datei wird hierbei auf den permanenten Speicher des K210 hochgeladen und kann als „automatische Ausführung nach Booten“ konfiguriert werden. Weiterhin hat der Hersteller (Kendryte) eigene Zusatzbibliotheken für die Peripherie des K210 geschrieben. Diese sind gut dokumentiert und erlauben die einfache Anbindung des LCD/TFT Displays. Die Liste der Tools ist daher recht kurz:

- Texteditor zur Erstellung von Mikropython Sourcecode
- Uploadtool uPyLoader⁸

⁷<https://micropython.org/>

⁸<https://github.com/BetaRavener/uPyLoader>

4.1.4. Karatsuba Multiplikation

Für die Quadrierung in der RSA Gruppe wird ein schneller Multiplikationsalgorithmus benötigt, welcher sich auch für die Implementierung in Hardware (FPGA) eignet. Ein Standard Verfahren hierzu ist die Karatsuba Multiplikation [KO63]. Die Idee des Algorithmus ist ein Teile-und-herrsche (Divide and Conquer) Ansatz. Die zu multiplizierenden Zahlen werden aufgeteilt, in kleinere Multiplikationen überführt und die Ergebnisse wieder zusammengesetzt. Für eine Hardwareimplementierung ist der Rekursionsabbruch erreicht, wenn die zu multiplizierenden Zahlen mit den schon vorhandenen Hardware Multiplizierern berechnet werden können. Im Fall des verwendeten ECP5-85k FPGA sind das die DSPs mit 18x18 Bit Multiplizierern, wovon aber nur 16x16 Bit benutzt werden, um die Zahlenbasis der Berechnung passend zu einer 2-er Potenz zu halten. Für den Algorithmus werden einige Annahmen getroffen:

- Es werden zwei natürliche, gleichlange Zahlen X und Y multipliziert. Wenn diese nicht gleichlang sind, wird am Anfang mit Nullen aufgefüllt.
- X und Y werden als Bitfolgen der Länge n interpretiert.
- Die Bitbreite des vorhandenen Hardware-Multiplizierers definiert die Basis b . Für diese Implementierung also $b = 2^{16}$.
- Die Länge n ist durch die Basis b teilbar.

Der Berechnung der Karatsuba Multiplikation ergibt sich wie folgend:

1. Aufteilen zur Basis b^m mit $m = n/2$:

Die zwei, zu multiplizierenden Zahlen X und Y werden jeweils in zwei Hälften der Länge $m = n/2$ geteilt.

$$X = x_1 b^m + x_0$$

$$Y = y_1 b^m + y_0$$

2. Umformen der Multiplikation:

$$\begin{aligned} XY &= (x_1 b^m + x_0)(y_1 b^m + y_0) \\ &= x_1 y_1 b^{2m} + (x_1 y_0 + x_0 y_1) b^m + x_0 y_0 \\ &= z_2 b^{2m} + z_1 b^m + z_0 \end{aligned}$$

$$z_2 = x_1 y_1$$

$$z_1 = (x_1 y_0 + x_0 y_1)$$

$$z_0 = x_0 y_0$$

Soweit zeigt sich, dass die Aufteilung der einer Multiplikation der zwei Zahlen X und Y in vier Multiplikationen der halben Bitbreite ($n/2$) überführt wurde.

3. Der Karatsuba Trick:

Karatsuba hat gezeigt, dass die zwei Multiplikationen in z_1 in eine einzelne Multiplikation mit zwei Additionen umgeformt werden können:

$$\begin{aligned} z_1 &= (x_1 y_0)(x_0 y_1) \\ &= (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0 \\ &= (x_1 + x_0)(y_1 + y_0) - z_2 - z_0 \end{aligned}$$

Die verbliebene Multiplikation könnte einen Überlauf erzeugen, da das Ergebnis der Multiplikation im Intervall $[b^m, 2b^m[$ liegt. Dies würde ein weiteres Bit für den Überlauf benötigen. Die folgende, äquivalente Gleichung erzeugt ein Multiplikations-Ergebnis im Intervall $] - b^m, b^m[$, kann daher aber negative Ergebnisse haben:

$$z_1 = (x_0 - x_1)(y_0 - y_1) + z_2 + z_0$$

Um keine Multiplikation mit Vorzeichen durchführen zu müssen, werden die Vorzeichen der Faktoren gespeichert und die Multiplikation dann mit den Beträgen berechnet. Das Vorzeichen des Ergebnisses wird dann korrigiert, je nachdem ob die Vorzeichen der Faktoren gleich oder ungleich waren. Dies ist eine „hardwarefreundliche“ Variante, da die Multiplikation mit nur natürlichen Zahlen auskommt.

4. Rekursion:

Die drei verbliebenen Multiplikationen in z_2, z_1 und z_0 können rekursiv wieder bei Schritt 1 beginnen. Der Rekursionsabbruch tritt ein, wenn die zu multiplizierenden zwei Zahlen nur noch die Bitbreite des Hardwaremultiplizierers haben (hier 16 Bit).

Es gibt Optimierungen für Karatsuba, wenn zwei gleiche Zahlen multipliziert werden, also eine Quadrierung erfolgt. Im folgenden Kapitel wird die Modulo Berechnung mittels der Barrett Reduktion erläutert. Und dafür wird ein voller Multiplizierer benötigt. Aus Platzgründen auf dem FPGA werden nicht zwei verschiedene Karatsuba Algorithmen (Quadrieren und volles Multiplizieren) implementiert, sondern nur die hier vorgestellte Karatsuba Variante.

In Listing 4.1 ist der Pseudocode für die Karatsuba Multiplikation wie oben beschrieben angegeben. Im Pseudocode ist zu erkennen, dass in jedem Rekursionsschritt nur noch drei Multiplikationen aufgerufen werden.

4. Hardware

```
1  b = 16
2  function karatsuba(x,y)
3      if (bitlength(x) <= b)    // Rekursionsende
4          return x * y

6      m = bitlength(x) / 2
7      x1, x0 = split_in_middle(x)
8      y1, y0 = split_in_middle(y)

10     // Rekursionen
11     z2 = karatsuba(x1, y1)
12     z1 = karatsuba((x0-x1),(y0-y1))
13     z0 = karatsuba(x0, y0)

15     return (z2 * 22m + (z1 + z2 + z0)*2m + z0)
```

Listing 4.1: Karatsuba Pseudocode

4.1.5. Barret Reduktion

Nach der Karatsuba Multiplikation bleibt noch die Frage nach einem gut in FPGAs zu implementierenden Algorithmus für die Modulo Berechnung. Divisionen in Hardware haben großen Platzbedarf und sind meistens nicht so schnell wie Multiplikationen. Da für die Implementierung immer wieder mit dem gleichen Modul n gerechnet wird, bietet sich die Barret Reduktion als Algorithmus an. In der Barret Reduktion wird ein vorausberechneter Wert μ verwendet, um dann in der eigentlichen Modulo Berechnung nur noch mit Multiplikationen zu arbeiten. Die Grundidee ist die Vorberechnung des Inversen als möglichst genaue rationale Zahl (\mathbb{Q}_+).

Sei $s = \frac{1}{n} \in \mathbb{Q}_+$ das Inverse des Moduls n . Dann gilt

$$x \bmod n = x - \left\lfloor x \frac{1}{n} \right\rfloor n = x - \lfloor xs \rfloor n$$

Bisher ist noch viel gewonnen, da die Berechnung von s in Computern mit beliebig präzisen Kommazahlen stattfinden müsste. Daher wird eine Zahlenbasis für die Berechnung festgelegt (z.Bsp. $b = 2^{16}$), die eine einfache Shiftoperation anstelle der Division ermöglicht. Hierfür wird ein Wert μ vorbereitet. Sei $k = \lfloor \log_b(n) \rfloor + 1$ die Länge des Moduls n zur Basis b . Dann ist b^{2k} die doppelte Länge des Moduls in Bits und

$$\frac{\mu}{b^{2k}} = \frac{1}{n}$$

dann folgt

$$\mu = \frac{b^{2k}}{n}$$

Um eine natürliche Zahl zu erhalten, wird das Ergebnis abgerundet:

$$\mu = \left\lfloor \frac{b^{2k}}{n} \right\rfloor \in \mathbb{N}$$

Die Berechnung von μ kann interpretiert werden als das Inverse von n , welches durch die Multiplikation mit b^{2k} und der Abrundung in den Raum der natürlichen Zahlen geschoben (Linkshift) wird. Natürlich ist dies nur eine Annäherung, die aber nach der vollständigen Berechnung einfach zu korrigieren ist. Dieses μ bleibt für alle Quadrierungen gleich, wird daher mit dem Setup in der Ansteuerungseinheit vorausberechnet und in das Eingabeparameterset aus Kapitel 4.1.1 mit aufgenommen. Gleiches gilt für den Parameter k .

Auf dem Beschleuniger werden dann die folgenden algorithmischen Schritte zur Modulo Berechnung ausgeführt:

1. Berechnung von q :

$$q = \left\lfloor \frac{\left\lfloor \frac{x}{b^k} \mu \right\rfloor}{b^k} \right\rfloor$$

2. Berechnung der Annäherung des Rests r :

$$r = (x \bmod b^{k+1}) - (qn \bmod b^{k+1})$$

3. Korrektur von r , falls negativ:

Wenn $r < 0$, dann $r = r + b^{k+1}$.

4. Korrektur von r , falls zu groß:

Solange $r \geq n$, ausführen von $r = r - n$.

Die Berechnungsschritte sind alle sehr gut geeignet, um in Hardware implementiert zu werden. In Schritt 1 sind zwei Divisionen durch 2-er Potenzen enthalten. Diese sind als Bitshifts (Rechtsshift) auszuführen. In Schritt 2 sind zwei Moduloberechnungen mit dem Modul b^{k+1} enthalten. Diese können in Hardware durch AND-Maskierungen erfolgen. Damit bleiben anstelle von Divisionen nur zwei Multiplikationen im Algorithmus übrig. Und hier zeigt sich die Notwendigkeit für einen vollen Multiplizierer, wie weiter oben erläutert.

Weiterhin ist für die Implementierung der Barret Reduktion für den FPGA die Bitbreite der einzelnen Zwischenvariablen (später dann Register) zu beachten. Das vorausberechnete μ kann um einige Bits länger sein, als die Parameter x und n . Die daraus resultierenden Implementierungsdetails folgen in Kapitel 4.2.

4.2. Implementierung

4.2.1. Installation der Tools

Die gesamte Programmierung des Projekts wurde auf einem Linux Hostsystem unter Mint 19.3 erstellt. Die Tools wurden alle aus den entsprechenden GitHub Repositories gebaut. Die Reihenfolge der Installationen sollte wie folgend eingehalten werden:

1. Yosys:

Projektverzeichnis:

<https://github.com/YosysHQ/yosys> (abgerufen am 14.05.2020)

Installation:

In der Projekt README Datei sind die Projektabhängigkeiten für Ubuntu Systeme angegeben. Diese funktionieren auch unter Mint 19.3 aus den Paketquellen der Distribution. Danach ist die Installation Makefile basiert.

```
1 $ make
2 $ sudo make install
```

2. Projekt Trellis:

Projektverzeichnis:

<https://github.com/SymbiFlow/prjtrellis> (abgerufen am 14.05.2020)

Installation:

Für die Installation von Projekt Trellis sind die Abhängigkeiten auch in der README Datei angegeben und sind in den Mint 19.3 Quellen verfügbar. Wichtig für das Klonen des Projektverzeichnisses ist die Option `--recursive`, damit die Chip-Datenbanken mit heruntergeladen werden. Die Buildtools sind `cmake` und `make`.

```
1 git clone --recursive https://github.com/SymbiFlow/prjtrellis
2 cd prjtrellis/libtrellis
3 cmake -DCMAKE_INSTALL_PREFIX=/usr .
4 make
5 sudo make install
```

Der Installationspfad (hier `/usr`) wird im nächsten Schritt wieder benötigt.

3. NextPnR-ECP5:

Projektverzeichnis:

<https://github.com/YosysHQ/nextpnr> (abgerufen am 14.05.2020)

Installation:

Soll ausschließliche für den Lattice ECP5 FPGA entwickelt werden, reicht die Installation eines Teils des Projekts NextPnR aus, nämlich die ECP5 Variante. Auch hier sind die Abhängigkeiten in der README angegeben und funktionieren aus den Mint 19.3 Quellen.

Die Installation erfordert die Angabe des Pfads zum Projekt Trellis aus dem vorherigen Schritt. Im Folgenden ist dieser Pfad (`/usr`) gleich, wie weiter oben verwendet.

```
1  cmake -DARCH=ecp5 -DTRELLIS_INSTALL_PREFIX=/usr .
2  make -j$(nproc)
3  sudo make install
```

4. Ujprog für ULX3S:

Projektverzeichnis:

<https://github.com/emard/ulx3s-bin/blob/master/usb-jtag/linux-amd64/ujprog>
(abgerufen am 14.05.2020)

Installation:

Es ist keine Installation notwendig. Die Datei `ujprog` muss nur ausführbar gemacht werden. Die Ausführung erfordert Rechte, um auf die USB Schnittstelle des Hostsystems zugreifen zu dürfen. Dafür kann die Ausführung als Superuser erfolgen oder der ausführende Benutzer in die `dialout` Gruppe des Linuxsystems eingetragen werden.

4.2.2. Makefile für die Toolchain

Für die Benutzung der FPGA-Toolchain wurde ein Makefile benutzt, mit dem die folgenden Aufgaben erledigt werden können:

- `make clean`
- Alle Tools durchlaufen und Bitstream generieren: `make <Verilog_topmodul>.bit`
- Programmierung des Bitstreams auf das ULX3S Board: `make prog`

Eine leicht gekürzte Version des Makefiles ist in Anhang A.1 aufgelistet. Die Ausführung des Makefiles greift auf zwei weitere Dateien zu. Zuerst auf die Datei `<TOPMOD>.ys`, in welcher Ausführungsanweisungen für die Synthese mit Yosys enthalten sind:

```
1  read_verilog Ulx3s.v
2  synth_ecp5 -json Ulx3s.json
```

Listing 4.2: Beispieldatei: `Ulx3s.ys` für Verilog Topmodul `Ulx3s.v`

Für die Ausführung von NextPnR-ECP5 wird eine Constraint Datei benötigt, welche die Zuordnung von In- und Output Variablen zu den tatsächlichen Pins des FPGA-Chips enthält. Eine Vorlage mit allen Anschlüssen des ULX3S Boards ist die Datei `ulx3s_v20.lpf`⁹. Die Constraint Datei wird im Makefile in Zeile 28 (Anhang A.1) angegeben.

Weiterhin kann im Makefile die Zielfrequenz für die Timinganalyse von NextPnR-ECP5 angegeben werden. Diese Angabe ist ein Parameter von NextPnR-ECP5 (Zeile 26, Makefile). Soll der Bitstream mittels des Makefiles auf den FPGA programmiert werden, ist auch der Pfad zu `ujprog` einzutragen (Zeile 11, Makefile).

⁹https://github.com/emard/ulx3s/blob/master/doc/constraints/ulx3s_v20.lpf

4.2.3. SpinalHDL

SpinalHDL ist im wesentlichen eine HDL als Scala Bibliothek. Die Programmierung findet in Scala statt und durch die Ausführung des Scala Codes wird Verilog oder VHDL generiert. Durch die Sprache Scala wird die Programmierung von Hardware (hier FPGAs) stark abstrahiert und durch die Paradigmen der objektorientierten und der funktionalen Programmierung ergänzt. Viele Strukturen und Befehle sind durch die SpinalHDL Bibliothek in zwei Varianten verfügbar. Scala selbst wird hierbei zu einer Art Metasprache für die Hardware Entwicklung und SpinalHDL gibt an, welche Hardware erzeugt werden soll. Ohne hier ausführlich auf die Programmierung mit SpinalHDL einzugehen, wird diese Unterscheidung an dem folgenden, kurzen SpinalHDL Beispiel verdeutlicht:

```
1 class Adder(width: Int) extends Component{
2     val io = new Bundle{
3         val a = in Bits(width bits)
4         val b = in Bits(width bits)
5         val result = out Bits(2*width bits)
6     }
7     result := (a.asUInt * b.asUInt).asBits
8 }
```

Listing 4.3: SpinalHDL Beispiel 1

Die Klasse `Adder` definiert einen Hardware Addierer für die zwei Bitfolgen `a` und `b`. Das Resultat ist die Bitfolge `result`. In Hardware haben diese Bitfolgen ihre Entsprechung dann als Ein- und Ausgangsleitungen des Addierers. Die Schlüsselwörter `val`, `in` sind aus der SpinalHDL Bibliothek und erzeugen Hardware. `Bits` ist eine Klasse aus SpinalHDL und erzeugt einen Vektor von Bitleitungen. Der Parameter `width` für `Bits` gibt die Bitbreite des Vektors an und ist ein Standarddatentypen aus Scala (`Int`). Somit kann durch Instanziierung dieser Klasse ein Hardwareaddierer erzeugt werden, dessen Bitbreite aber für jede Instanz eigens angegeben werden kann. Die Bitbreite wird durch eine Scalavariablen (`width: Int`) angegeben und die zu erzeugende Hardware durch die Verwendung der SpinalHDL Bibliothek. Die Erzeugung einer Instanz dieses Addierers ist, wie in der Objektorientierung oft verwendet, mit dem Schlüsselwort `new` verbunden:

```
1 val adder128 = new Adder(128)
```

Listing 4.4: SpinalHDL Instanziierung der Klasse `Adder`

Dieses Konzept führt zu einer steilen Einsteiger-Lernkurve, bietet aber nach der Einarbeitung einen schnellen und übersichtlichen Arbeitsfluss. Der SpinalHDL Quellcode ist daher sehr gut menschlich les- und verstehbar, was nicht auf jede HDL zutrifft. Weiterhin ist an dem Beispiel zu erkennen, wie stark der Quellcode an die klassische Objektorientierung angelehnt ist. Zu SpinalHDL gibt es eine ausführliche und beständig wachsende Dokumentation mit vielen Beispielen ¹⁰.

¹⁰<https://spinalhdl.github.io/SpinalDoc-RTD/>

Die Ausführung des in SpinalHDL geschriebenen Quellcodes kann mittels des Scala-Build-Tools (SBT) erfolgen. SBT setzt eine Projektdatei mit dem Namen `build.sbt` im Projektverzeichnis voraus. Hier ein Beispiel dieser Datei:

```

1  name := "UartMult"
2  version := "1.0"
3  scalaVersion := "2.11.12"

5  libraryDependencies ++= Seq(
6    "com.github.spinalhdl" % "spinalhdl-core_2.11" % "1.3.5",
7    "com.github.spinalhdl" % "spinalhdl-lib_2.11" % "1.3.5"
8  )

10 fork := true

```

Listing 4.5: Scala-Build-Tool: Beispieldatei `build.sbt`

Neben dem Namen des Scala-Toplevel Moduls (hier `UartMod`), welches eine Main-Funktion enthalten muss, sind die Abhängigkeiten zu den SpinalHDL Bibliotheken mit Pfad im Internet angegeben. Durch den Befehl `sbt run` werden die Abhängigkeiten aufgelöst (z.Bsp. automatisch heruntergeladen), die Main-Funktion ausgeführt und dadurch der Verilog oder VHDL Sourcecode generiert. Der Vorteil an SBT ist die automatische Auflösung dieser Abhängigkeiten. Ansonsten ist dieser Prozess dem von Makefiles sehr ähnlich. Der erzeugte Verilog Quellcode wird dann mit den FPGA Tools, wie in den Kapiteln 4.2.1 und 4.2.2 beschrieben, zu einem Bitstream weiter verarbeitet.

Eine weitere Funktionalität von SpinalHDL ist die Möglichkeit zur Erzeugung von Simulationen. Basierend auf dem generierten Verilog Code kann SpinalHDL unter Verwendung des Tools Verilator eine taktgenaue Simulation erzeugen. Diese Simulation wird in einer Datei gespeichert und kann mit Anzeigetools (z.Bsp. GTKWave) ausgewertet werden. Verilator und GTKWave auch als Open Source veröffentlicht und können daher für Anwender ohne weitere Kosten verwendet werden. Die erzeugten Simulationen sind für die Fehlersuche sehr hilfreich und wurden in diesem Projekt ausgiebig verwendet (siehe Kapitel 4.3).

In Anhang A.2 ist kleines, vollständiges Beispiel in SpinalHDL aufgelistet. Die Funktion des Beispiels ist das Blinken einer LED im Sekundentakt. Die Klasse `Blinky` hat nur eine Ausgangsleitung `led` mit der Breite eines Bits. Diese Leitung wird abwechselnd auf High(1) oder Low(0) geschaltet. Hierfür wird ein Zustandsautomat aus der SpinalHDL Bibliothek verwendet. Auch dieser Zustandsautomat ist eine Klasse und wird in Zeile 14 instanziiert. Innerhalb des Automaten werden zwei Zähler (Counter) aus der SpinalHDL Bibliothek angelegt.

4. Hardware

Es folgen drei definierte Zustände:

- StateA: Startzustand des Automaten. Direkter Übergang in StateB.
- StateB: LED an, Wartezustand bis der erste Zähler (counter1) überläuft. Danach Übergang in StateC.
- StateC: LED aus, Wartezustand bis der zweite Zähler (counter2) überläuft. Danach Übergang in StateA.

Dieser Zustandsautomat läuft nach Inbetriebnahme (Programmierung des Bitstreams auf den FPGA) endlos.

Am Ende des Quellcodes befindet sich die Main-Funktion (ab Zeile 54). In dieser wird zuerst die Verilog Ausgabe erzeugt (Zeilen 55-58). Hier ist das Paradigma der funktionalen Programmierung in Scala zu erkennen: Es wird eine SpinalConfig definiert und auf diese Konfiguration wird die Funktion `generate(new Blinky)` ausgeführt. Im weiteren Teil der Main-Funktion wird die Simulationsdatei erzeugt (Zeilen 62-73). Hier wird die SpinalConfig als eigenes Objekt definiert und erzeugt. Diese wird dann als Parameter and die SimConfig gegeben. Im Bereich `doSim` wird dann nur die Systemclock für 3 Mio. Taktzyklen stimuliert. Die erzeugten Ergebnisse sind zum einen die Verilogdatei und zum anderen die Simulationsdatei (anzeigbar mit GTKWave).

Dieses Beispiel wäre auch wesentlich kürzer möglich, zum Beispiel mit nur einem Zähler und dem Togglen der Ausgangsleitung. Es wurde bewusst etwas ausführlicher gestaltet, um in dem Zustandsautomaten mehrere Zustände zu zeigen.

Die einfachste Struktur für den Start eines neuen SpinalHDL Projekts ist in Abbildung 4.5 dargestellt. Die Datei `MeinProjekt.scala` enthält den SpinalHDL Quellcode und muss in der verschachtelten Verzeichnisstruktur gespeichert werden. Die Datei `build.sbt` (siehe weiter oben) ist im Hauptverzeichnis des Projekts abzulegen, auf gleicher Höhe mit dem `src` Verzeichnis. Die Ausführung von `sbt run` im Projektverzeichnis erstellt dann alle weiteren Verzeichnisse und Dateien, inklusive des Verilog oder VHDL Codes und der Simulationsdatei (wenn in der Main-Funktion implementiert).

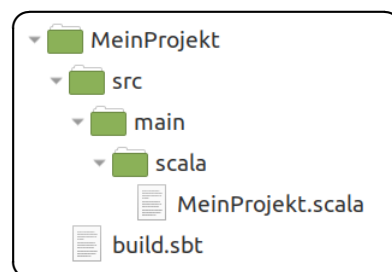


Abb. 4.5.: SBT Projektstruktur als einfachstes Beispiel

4.2.4. Modularität

Für die SpinalHDL Implementierung wurden zwei Einzelprojekte angelegt. Das Hauptprojekt mit dem Namen `UartVdf` erfüllt die t -fach wiederholte Berechnung $x^2 \bmod n$, wie in Kapitel 4.1.1 beschrieben. Das zweite, kleinere Projekt ist ein Taktteiler (Clock Divider), um niedrigere Taktraten für die Implementierung erreichen zu können.

UartVdf:

In Abbildung 4.6 sind die SpinalHDL Einzelmodule des Projekts mit deren Funktionalität als Übersicht dargestellt. Das Toplevel Modul heißt wie das Projekt `UartVdf.scala`, enthält die Main-Funktion und die Verbindungen nach außen (UART und Clock). Untergeordnet sind die zwei Module `UartFsmInOut` und `MultMod`.

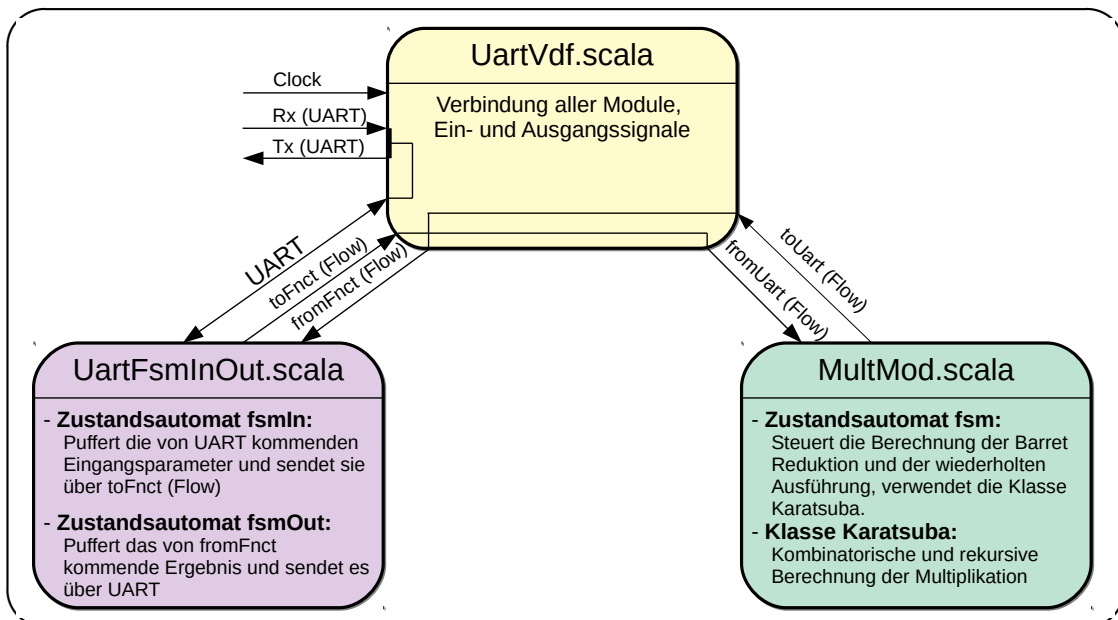


Abb. 4.6.: Projekt UartVdf: Modulübersicht

Die Implementierung der Untermodule wird in den folgenden Kapiteln beschrieben. An dieser Stelle sei aber eine weitere Eigenschaft von SpinalHDL hervorgehoben, die Art der Verbindung der Module untereinander. Hierfür gibt es die Klassen `Stream` und `Flow`. Dies sind leichtgewichtige Kommunikationsprotokolle und ersetzen das teilweise sehr mühsame Verbinden einzelner Leitungen in den Sprachen Verilog oder VHDL.

4. Hardware

Im folgenden Quellcode (Listing 4.6) ist der Verbindungsteil des Moduls `UartVdf.scala` aufgelistet.

```
1 class UartVdf(width: Int) extends Component{
2     val io = new Bundle{
3         val uart = master(Uart())
4     }

6     val uartfsm = new UartFsmInOut(width)
7     val multmod = new MultMod(width)

9     uartfsm.io.uart <> io.uart
10    uartfsm.io.toFnct >> multmod.io.fromUart
11    multmod.io.toUart >> uartfsm.io.fromFnct
12 }
```

Listing 4.6: UartVdf.scala: Verbindungen

Als Ein- und Ausgang des Moduls wird nur eine Instanz der Klasse `Uart()` angegeben. Danach werden zwei Objekte von den Untermodulen `UartFsmInOut` und `MultMod` erzeugt (Zeilen 6-7). Die Verbindungen zwischen den Modulen sind dann die drei Zeilen 9-11. Die Verbindungen zwischen den Untermodulen sind als `Flow` in den Untermodulen selbst definiert. In Listing 4.7 ist der Auszug aus dem Modul `UartFsmInOut` für diese Definitionen dargestellt und in Listing 4.8 der Auszug aus `MultMod`.

```
1 val io = new Bundle{
2     val uart = master(Uart())
3     val toFnct = master Flow(Bits(((3*width)+48) bits))
4     val fromFnct = slave Flow(Bits(width bits))
5 }
```

Listing 4.7: UartFsmInOut.scala: Uart und Flow Definitionen

```
1 val io = new Bundle{
2     val fromUart = slave Flow(Bits(((3*width)+48) bits))
3     val toUart = master Flow(Bits(width bits))
4 }
```

Listing 4.8: MultMod.scala: Flow Definitionen

Diese drei Quellcode Auszüge stellen die vollständige Definition der Verbindungen unter den Modulen dar. In anderen HDL wären dies sehr viele Zeilen mehr. Die Verbindungen vom Typ `Flow` enthalten eine Payload (Typangabe ist der Parameter der Klasse) und ein Valid Signal. Diese sind aber in der Klasse gekapselt und müssen nicht einzeln verbunden werden. Zusätzlich wird in der Definition angegeben, welche Seite der Verbindung als `master` und welche Seite als `slave` agiert. Insgesamt betrachtet ist dies einer der großen Vorteile von SpinalHDL. Nicht nur solche schlanken Verbindungsprotokolle sind auf diese elegante Weise in SpinalHDL verfügbar, sondern auch komplette Bussysteme (z.Bsp. AXI) können auf diese Art erzeugt und verbunden werden.

Generische Bitbreite:

Weiterhin ist in den obigen Listings zu sehen, dass das Toplevel Modul `UartVdf.scala` den Parameter `width: Int` erhält und bei Erstellung der Untermodule diesen Parameter weitergibt. Diese Parametrisierung wurde durch die komplette Implementierung durchgezogen. Der Parameter wird in der Main-Funktion bei Erstellung des `UartVdf`-Objekts angegeben und bestimmt damit an einer einzelnen Stelle die Bitbreite der gesamten FPGA Implementierung.

Clock Divider:

Der FPGA-Chip ECP5-85k wird auf dem ULX3S Board mit einer externen Taktquelle bei 25 MHz betrieben. Die Implementierung des Karatsuba Algorithmus ist als kombinatorischer Schaltkreis ausgelegt. Somit sinkt die mögliche Taktfrequenz bei steigender Größe der Implementierung schnell ab. Um den Eingangstakt des Toplevel Moduls `UartVdf` nach unten anpassen zu können, wurde ein einfacher Taktteiler (Clock Divider) in SpinalHDL implementiert. Dieser Taktteiler ergibt eine eigenständige Verilog Datei und wird mittels eines weiteren Verilog Wrappers „angedockt“. In Listing 4.9 ist der funktionale Teil des Clock Dividers gelistet. Im Beispiel wird der Takt um den Faktor Fünf verlangsamt (25MHz zu 5MHz). Hierdurch ergibt sich die Möglichkeit, auch größere Bitbreiten der Implementierung mit niedrigeren Taktfrequenzen stabil zu betreiben.

```

1  class ClkDiv extends Component{

3      val io = new Bundle{
4          val clk_out = out Bool
5      }

7      io.clk_out := False

9      val counter = Counter(1 to 5)
10     counter.increment()
11     when(counter.willOverflow){
12         io.clk_out := True
13     }
14 }

```

Listing 4.9: ClkDiv.scala: Taktteiler in SpinalHDL

4.2.5. UART

Der Beschleuniger erhält die Parameter über die UART Schnittstelle. Die in Kapitel 4.1.1 definierten Parameter x , n und t wurden in Kapitel 4.1.5 um die zwei weiteren Parameter μ und k erweitert. Die komplette Liste der zu übertragenden Parameter ist in Tabelle 4.4 aufgelistet. Das Protokoll ist so definiert, dass die Parameter in der Reihenfolge ihrer Nummern (0 bis 4) als Little Endian übertragen werden.

Nummer	Parameter	Bitbreite
0	x	width
1	n	width
2	μ	width + 8
3	k	8
4	t	32
	Gesamtlänge	3*width+48

Tab. 4.4.: UART Payload der Eingangsparameter

Als Beispiel für die Übertragung in Little Endian sei der Parameter $t = 4$ und damit als Bytefolge (Länge: 4 Bytes) in Big Endian $t = 0x00\ 00\ 00\ 04$. Für die Übertragung in Little Endian muss diese Bytefolge zu $0x04\ 00\ 00\ 00$ gedreht werden, wobei $0x04$ zuerst gesendet wird.

Das Berechnungsergebnis wird vom Beschleuniger über UART zurückgesendet. Hier ist die Payload nur das Ergebnis mit der Bitbreite **width** (Tabelle 4.5) und wird auch in Little Endian übertragen.

Nummer	Parameter	Bitbreite
0	Ergebnis	width
	Gesamtlänge	width

Tab. 4.5.: UART Payload des Ausgabeergebnisses

Die SpinalHDL Implementierung der Klasse `Uart()` sieht die Übertragung Byteweise vor. Daher ist das Empfangen und Senden von längeren Bytefolgen (Payload) eigens zu implementieren. In dem Scala Modul `UartFsmInOut.scala` sind hierfür zwei Zustandsautomaten zuständig. Das Listing des Moduls ist im Anhang A.3 enthalten. Als Erläuterung sind im Folgenden die zwei Zustandsautomaten graphisch dargestellt und können mit dem Quellcode im Anhang verglichen werden.

Beim Empfangen der Eingabeparameter wird ein Register (**buffer**) der Länge der Payload byteweise befüllt und nach dem vollständigen Empfang über den Flow **toFnc**

weiter gesendet. In Abbildung 4.7 ist der implementierte Zustandsautomat `FsmIn` dargestellt. Der Empfang eines Bytes wird durch `uart.read.valid=True` signalisiert. Wenn die komplette Payload in den `buffer` übertragen wurde, wird dies über das Flowsignal `toFunct.valid=True` signalisiert. Das SpinalHDL Modul `MultMod.scala` erkennt dann, auf der anderen Seite des Flows, dass die vorliegenden Daten vollständig sind und startet die Berechnung.

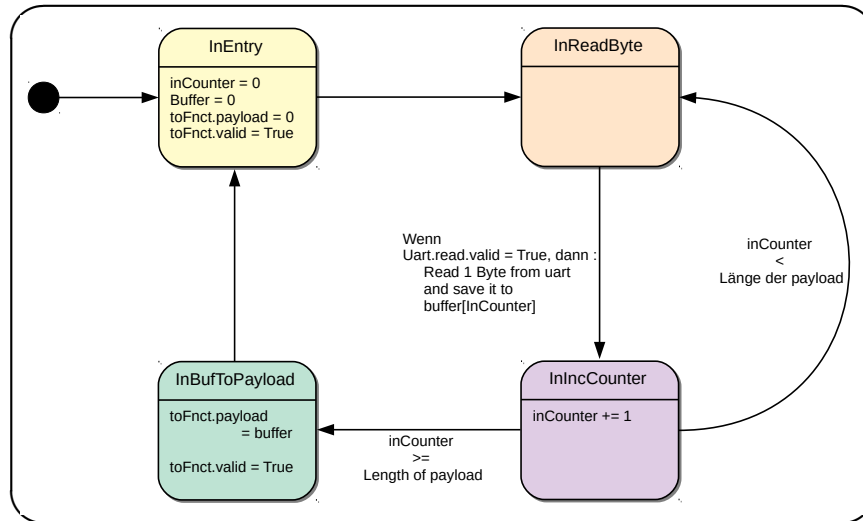


Abb. 4.7.: UART Zustandsautomat für die eingehenden Daten (`FsmIn`)

Beim Senden kommt das Ergebnis über den Flow `fromFunct` an, wird in ein Register (`buffer`) zwischengespeichert und dann byteweise versandt. Die graphische Entsprechung des Zustandsautomaten ist in Abbildung 4.8 dargestellt. Auch hier werden die Zustandswechsel von den entsprechenden `valid` Signalen ausgelöst.

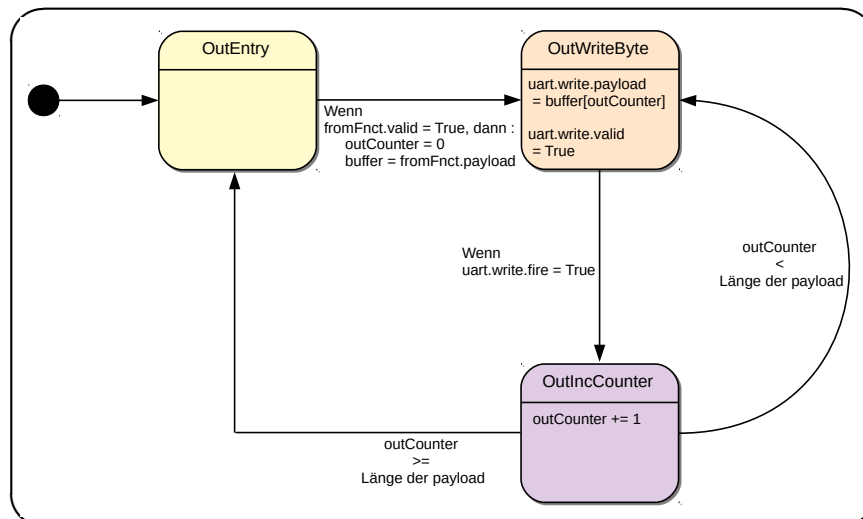


Abb. 4.8.: UART Zustandsautomat für die ausgehenden Daten (`FsmOut`)

4.2.6. Karatsuba: Kombinatorische Lösung

Die Implementierung des Karatsuba Algorithmus startete mit der Fragestellung, ob sich die Rekursion aus dem Pseudocode (Listing 4.1) möglichst exakt auf die Programmierung in SpinalHDL übertragen lässt. SpinalHDL ist in der Grundlage die Sprache Scala und damit sind Klassen und Funktionen aus der Objektorientierung enthalten. Wird rekursive Programmierung auf einer CPU ausgeführt, werden die Zwischenergebnisse auf dem Stack abgelegt, beim Aufstieg der Rekursion von dort entnommen und zu Zwischenergebnissen verarbeitet. Übertragen auf Hardwareprogrammierung wären die im Stack gespeicherten Zwischenergebnisse durch Register auszuführen. Ist eine rein kombinatorische Lösung ohne Register mit SpinalHDL möglich und ist eine solche Lösung auch synthetisierbar und kann geroutet werden? Wenn dies alles mit Ja beantwortet werden kann, wie schnell ist diese Implementierung dann noch? Kombinatorische Schaltkreise werden mit ihrer steigenden Größe auch langsamer in ihrer Taktfrequenz. Die Antworten auf diese Fragen können hier schon vorweg gegeben werden: Ja, es ist möglich und gar nicht mal so langsam. Die Ergebnisse sind im Fazit in Kapitel 6 festgehalten.

Die erste Karatsuba Implementierung war eine Funktion in SpinalHDL und hat die gewünschte Funktionalität erreicht. Das Problem mit der Funktion ist die mehrfache Verwendung. Für jeden Funktionsaufruf wird eine eigene Hardwareinstanz erstellt. Wenn die Funktion also an drei verschiedenen Stellen aufgerufen wird, werden auch drei voneinander getrennte Schaltkreise dafür erzeugt. Sinnvoller ist allerdings die einmalige Instanziierung. Dies ist möglich, wenn der Karatsuba Algorithmus als Klasse angelegt und dann im entsprechenden Modul eine Instanz davon erzeugt wird. Diese Instanz kann an verschiedenen Stellen im Modul verwendet werden ohne dass neue Hardware daraus entsteht. Wie weiter oben schon beschrieben, wird die Multiplikation an mehreren Stellen für das Quadrieren und die Modulorechnung benötigt. Hier kommt wieder Mächtigkeit der SpinalHDL Bibliothek zum tragen. In anderen HDL müsste ein synchronisierter Multiplexer für die verschiedenen Eingangssignale der Wiederverwendung per Hand implementiert werden. In SpinalHDL passiert dies „unter der Haube“ und ist für die Programmierung nicht zu beachten. In anderen Worten: Es funktioniert einfach.

Im Anhang A.4 ist die vollständige Klasse Karatsuba (`KaraMult()`) gelistet. Im Vergleich mit dem Pseudocode 4.1 ist erkennbar, wie nahe diese SpinalHDL Implementierung an der entsprechenden Softwarevariante liegt. Der Quellcode sieht fast aus, wie in einer objektorientierten Sprache (z.Bsp. Java, Python) und ist sehr gut les- und nachvollziehbar.

Diese Karatsuba Klasse ist im SpinalHDL Modul `MultMod.scala` enthalten, wird dort als Objekt erstellt und an mehreren Stellen verwendet. Welche Stellen das sind, wird im folgenden Kapitel 4.2.7 über die Barret Reduktion ersichtlich.

4.2.7. Barret Reduktion: Zustandsautomat

Das SpinalHDL Modul `MultMod` besteht aus einem großen Zustandsautomat (`fsm`) in dem sowohl das Quadrieren (Karatsuba) als auch die Modulo Berechnung (Barret Reduktion) enthalten sind. Der komplette Quellcode ist in Anhang A.5 zu finden. In diesem Quellcode sind zusätzlich noch viele Signale mit dem Präfix `temp_` und dem Postfix `_out` enthalten. Diese wurden bewusst nicht aus dem Quellcode entfernt, da im späteren Kapitel 4.3 noch auf diese Signale Bezug genommen wird.

Der Zustandsautomat `fsm` enthält 9 Zustände und diese werden, bis auf eine Ausnahme, nur streng sequentiell nacheinander überführt. D.h. Es gibt bis auf die eine Ausnahme keine Verzweigungen im Automaten. Daher wurde auf eine graphische Darstellung, wie die Zustandsautomaten weiter oben, verzichtet. Statt dessen folgt eine Textbeschreibung der Zustände in ihrer Ausführungsreihenfolge, so wie sie auch im Quellcode aufeinander folgen.

Innerhalb des Zustandsautomaten werden zuerst Register für die Eingabeparameter, das Ergebnis, eine Bitmaske, einen Zähler und mehrere Zwischenergebnisse definiert. Hier ist zu bemerken, dass diese Register alle nur mit der Angabe `width` als Bitbreite erstellt werden. Die Implementierung bleibt auch hier mit generischer Bitbreite erstellbar. Weiterhin wird hier auch eine Instanz (`mult`) der Karatsuba Klasse erstellt.

StateA:

Wenn `fromUart.valid` die Bereitstellung der Eingangsparameter signalisiert, wird die Uart Payload (`fromUart.payload`) aufgeteilt in die einzelnen Register für a, n, μ, k, t kopiert. Danach folgt der Übergang zu StateB.

StateB:

In diesem Zustand wird die Quadrierung von a mittels der Karatsuba Instanz `mult` berechnet und das Ergebnis nach einer Verzögerung von einem Takt im Register `result_mult` gespeichert. Mit dem Verlassen dieses Zustands ist die Quadrierung abgeschlossen. Die weiteren Zustände sind ausschließlich für die Barret Reduktion und die t -fache Wiederholung implementiert. Es folgt der Übergang zu StateC.

StateC:

Ab diesem Zustand werden die Berechnungen aus Kapitel 4.1.5 durchgeführt. Dort wurde erwähnt, dass der Parameter μ mehrere Bits länger sein kann, als die Bitbreite `width`. Da der Karatsuba Multiplizierer aber nur die Bitbreite `width` verarbeiten kann, wird die Multiplikation mit μ in zwei einzelne Multiplikationen aufgeteilt. Vor der Multiplikation erfolgt noch die Division durch b^k , ausgeführt als Bitshift. Danach wird mit den unteren `width` Bits von μ multipliziert, danach dann mit den restlichen,

4. Hardware

oberen Bits von μ . Die beiden Ergebnisse werden dann durch shiften und addieren zum Gesamtergebnis der Multiplikation zusammengefügt. Es geht hierbei um die folgende Berechnung:

$$q = \left\lfloor \frac{\left\lfloor \frac{x}{b^k} \mu \right\rfloor}{b^k} \right\rfloor$$

Hier in Zustand StateC wird die Berechnung mit den unteren Bits von μ durchgeführt und in Register `temp_2` gespeichert. Danach folgt der Übergang zu StateD.

StateD:

In StateD wird der zweite Teil der Multiplikation mit μ , also mit dessen oberen Bits, berechnet. Danach wird das Ergebnis mittels Bitshift durch b^k geteilt und in Register `temp_3` gespeichert. In dieser Berechnung sind einige „Hilfsleitungen“ (`temp_2a`, `temp_2b`, `temp_2c`) enthalten, die aber keine Register sind, sondern nur die kombinatorische Logik für die Zusammenführung der einzelnen Multiplikationsergebnisse verbinden. In `temp_3` steht bei Verlassen dieses Zustands der berechnete Wert für q aus der obigen Gleichung. Es folgt die Überführung in StateE.

StateE:

In StateE ist der erste Schritt der Berechnung des Rests r enthalten. Die Gleichung für r (siehe Kapitel 4.1.5) ist

$$r = (x \bmod b^{k+1}) - (qn \bmod b^{k+1})$$

Für die Moduloberechnungen mit dem Modul b^{k+1} wird eine Bitmaske erstellt und im Register `mask` gespeichert. Danach folgt die Multiplikation von `q=temp_3` und dem Modul n . Mit weiteren Hilfsleitungen (`temp_4`, `temp_5`, `temp_6`) wird das Ergebnis für r berechnet und in Register `temp_7` gespeichert. Die Moduloberechnungen sind als AND-Operationen mit der erstellten Bitmaske `mask` angelegt. Somit bleiben nach diesem Zustand nur noch die Korrekturen für Unter- und Überläufe in r übrig. Es folgt der Übergang in StateF.

StateF:

Dieser Zustand korrigiert ein eventuell negatives Ergebnis in `temp_7`, speichert das neue Ergebnis in Register `temp_8` und überführt danach in StateG.

StateG:

In StateG wird die Korrektur eines eventuell zu großen Ergebnisses in `temp_8` vorgenommen und in Register `result_barret` gespeichert. Die Quadrierung und Moduloberechnung ist mit diesem Zustand vollständig und in `result_barret` gespeichert.

StateH:

Das Quadrieren und Modulorechnen soll t mal wiederholt werden. Hierfür wird der Zähler `repeatCounter` jedes mal bei Erreichen dieses Zustands inkrementiert. Ist der Zähler bei der, durch den Parameter t vorgegebenen Anzahl an Wiederholungen angekommen, wird danach in den Zustand StateI gewechselt. Stehen noch weitere Wiederholungen aus, wird `barret_result` in das Register a kopiert und die Berechnung erneut bei Zustand StateB begonnen.

StateI:

StateI markiert das Ende der wiederholten Quadrierungen und Moduloberechnungen. Das finale Ergebnis steht in Register `result_barret`, wird von dort in die Payload `toUart.payload` kopiert und das entsprechende `toUart.valid` Signal auf `True` gesetzt. Danach erkennt das Modul `UartFsmInOut` die Bereitschaft der Daten und sendet diese. Der Zustand StateA wird angenommen und dort auf neue Berechnungen gewartet.

4.2.8. Ansteuerung mit RISC-V: MikroPython

Im Vergleich zur FPGA Implementierung ist die Programmierung für den RISC-V eine klassische Softwareentwicklung. Der Kendryte K210 bringt eine performante Mikropython Umgebung ab Werksauslieferung mit. Der Quellcode ist in Anhang A.6 enthalten. Es wird die Klasse `Maixduino` definiert. Diese hat die Funktionalität der Initialisierung des Boards, der Ansteuerung des Displays und handhabt die UART Schnittstelle. Die UART Parameter sind fest im Konstruktor der Klasse eingetragen. Die Bitbreite der FPGA Implementierung ist als globale Variable `WIDTH` angelegt. Die Zufallszahlen werden aus der Mikropython Implementierung von `os.urandom()` entnommen und sind erwartungsgemäß nicht wirklich zufällig. Bei jedem Neustart wird die gleiche Abfolge erzeugt.

Die Implementierung erstellt die Eingabeparameter a , n , μ , k und t . Als erstes werden diese Parameter über UART zum FPGA gesendet und auf das Ergebnis gewartet. Danach wird die gleiche Berechnung auf der RISC-V MCU ausgeführt. Das Display zeigt derweilen Ausgaben über den aktuellen Zustand. Für beide Berechnungen (FPGA und RISC-V) wird sowohl die jeweilige Zeit in μs gemessen und ausgegeben, als auch die beiden Ergebnisse miteinander verglichen und eine Ausgabe über die Gleichheit auf dem Display ausgegeben. Mit Hilfe der Taktfrequenzen der beiden Systeme kann hier eine Analyse über die Geschwindigkeit beider Systeme erfolgen. Diese ist im Kapitel 6 enthalten.

4.3. Testen und debuggen

Testen und Debuggen ist in der FPGA Programmierung unterschiedlich zur Fehlersuche in Software. Eine Standard Methode ist die Simulation der Hardware in verschiedenen Stufen der FPGA Toolchain. In diesem Projekt wurden Simulationen ausschließlich auf Codebasis verwendet. In diesen Simulationen wird die FPGA Hardware nicht berücksichtigt, sondern als ideal angenommen. Es können zum Beispiel keine Aussagen über das reale Timingverhalten oder Leitungslängen getroffen werden. SpinalHDL bietet eine einfache Möglichkeit, solche Simulationen mit Open Source Tools zu erstellen. Wie weiter oben besprochen wird hierfür in SpinalHDL ein eigener Teil in der Main Funktion geschrieben, der die Simulation mit Verilator im Hintergrund erstellt. Das Ergebnis ist eine Simulationsdatei, welche mit GTKWave analysiert werden kann. In Abbildung 4.9 ist eine solche Simulation mit GTKWave als Screenshot dargestellt.

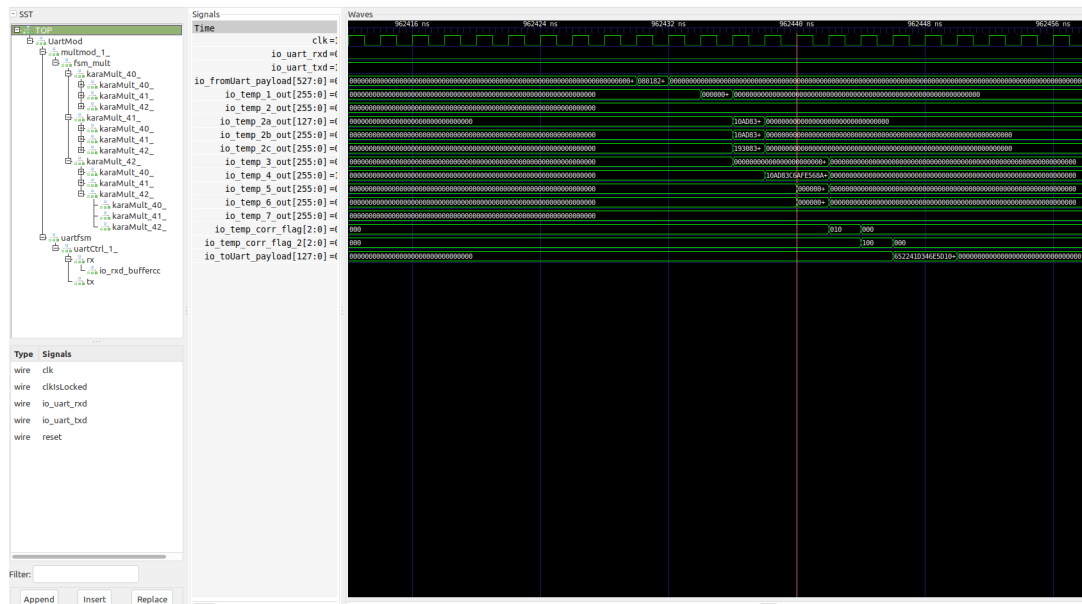


Abb. 4.9.: GTKWave: Beispiel Simulation mit Zustandsautomat aus MultMod

Im linken Teil der Abbildung sind die Module (Verilog), deren Ein- und Ausgabesignale und deren Register zu sehen. Diese können als taktgenaue Simulation in der Mitte der Abbildung angezeigt werden. Im Bild ist die Verarbeitung des Zustandsautomaten für die Quadrierung und die Barret Reduktion zu sehen. Die treppenartige Befüllung der Register in der Mitte sind die Ergebnisse jedes einzelnen Zustands, da diese zeitlich nacheinander ausgeführt werden. Die Werte der einzelnen Register können zu jedem Zeitpunkt der Simulation einzeln betrachtet werden. Um hier Berechnungsfehler zu finden, müssen allerdings (korrekte) Vergleichswerte vorliegen. Mit solchen Vergleichswerten kann dann der einzelne Zustand herausgefunden werden, in dem der Fehler passiert.

Um solche korrekten Vergleichswerte zu erhalten, wurden Python Implementierungen erstellt. Diese simulieren jeden Berechnungsschritt (Zustand) einzeln und geben die gleichen Werte wie die FPGA Register aus. Beide Implementierungen (FPGA und Python) wurden dann mit den gleichen Testvektoren gestartet. Die Zwischenergebnisse aus der Python Implementierung wurden dann mit den Registerwerten aus der GTKWave Simulation verglichen. So konnten viele der Fehler ermittelt und beseitigt werden. Dieses Vorgehen erfordert Sorgfalt, Konzentration und ist zeitintensiv. Es müssen viele lange Zahlen visuell miteinander verglichen werden.

Wo es möglich ist, wären weitere Methoden zum Finden und Beseitigen von Fehlern wünschenswert. Die geforderte Funktionalität sieht UART als Kommunikations-Schnittstelle des FPGA vor. Daraus entstand die Idee, zuerst die UART Kommunikation zu implementieren und stabil zu bekommen. Diese Idee hat letztendlich zu insgesamt 15 einzelnen Projekten in SpinalHDL geführt. Diese Projekte bauen aufeinander auf, sind aber jedes für sich einzeln lauffähig. Die ersten Projekte bringen LEDs zum blinken, danach wird die UART Schnittstelle implementiert. Erst mit einer stabilen UART Kommunikation sind dann die weiteren Funktionen (Multiplikation, Modulo, Wiederholungen) entstanden. Dieses Vorgehen gleicht der modularen Entwicklung in der Softwaretechnik. Die Module werden einzeln entwickelt und getestet. Erst wenn mehrere Module einzeln fehlerfrei sind, werden diese zu einer Gesamtanwendung integriert und dann erneut getestet. Mehr über diese einzelnen Projekte ist in Kapitel 5 zu lesen.

Ein weiteres, sehr spezielles Fehlerszenario ist auch erwähnenswert. Die Quadrierung und Moduloberechnung soll t fach wiederholt werden und t kann hierbei millionenfache Wiederholung bedeuten. Es ergab sich der Fall, dass die Implementierung nach mehreren Tausend korrekten Berechnungen plötzlich eine fehlerhafte Berechnung durchführt und der FPGA ein falsches Ergebnis zurück gibt. Es stellte sich die Frage, wie dieser zufällig erscheinende Fehler zu finden ist. Eine Simulation mit mehreren Millionen Ausführungen des Zustandsautomaten ist nicht manuell vergleichbar. Das wäre das Bild aus der obigen Abbildung, nur millionenfach hintereinander auf der Zeitachse. Der helfende Trick war hier ein Teile-und-herrsche Prinzip. Es wurden Testvektoren immer in der Mitte der Anzahl der Berechnungen ermittelt. Somit konnte eingegrenzt werden, ob der Fehler in der ersten oder zweiten Hälfte der wiederholten Berechnung stattfindet. Diese Teilung wurde wiederholt, bis ein einzelner Testvektor mit der fehlerhaften Berechnung vorlag. Es hat sich herausgestellt, dass die Umsetzung der Barret Reduktion einen Fehler enthielt, der nur sehr selten auftritt. Der Algorithmus in Kapitel 4.1.5 sieht vor, dass die Korrekturen von r nacheinander Unter- und Überläufe korrigieren. Die Implementierung hatte aber nur eine Entweder-Oder Korrektur. Es konnte nur entweder ein Unterlauf oder ein Überlauf korrigiert werden. In sehr seltenen Fällen sind aber beide Korrekturen nacheinander notwendig. Dies war ein schwer zu findender Fehler und es hat einige Zeit gekostet, die oben genannte Methode zu erarbeiten.

4.4. Demonstrator

Der Demonstrator wurde direkt nach dem Schema aus Abbildung 4.4 gebaut. Für die beiden Developerboards ULX3S und Maixduino wurden Gehäuse im 3D-Druck angefertigt. Die Vorlagen hierfür sind von der Internetplattform Thingiverse¹¹ und als Open Source veröffentlicht. Beide Geräte werden über die USB Schnittstelle mit den Implementierungen aus den vorherigen Kapiteln programmiert. Die UART Schnittstellen wurden mit drei einzelnen Steckleitungen verbunden. Das Display wurde mit dem Maixduino geliefert und ist über einen 24-poligen MCU Port direkt mit dem Board verbunden. In Abbildung 4.10 ist ein Foto des fertigen Demonstrators abgebildet. Im Vordergrund ist der Maixduino mit dem Display zu sehen. Es wurde gerade eine erfolgreiche Berechnung durchgeführt. Im Hintergrund ist das ULX3S Board abgebildet. Die Messergebnisse aus dem Fazit (Kapitel 6) wurden zum Teil mit dem Demonstrator erstellt.



Abb. 4.10.: Demonstrator: Foto nach erfolgreicher Berechnung

¹¹<https://www.thingiverse.com/>

5. Open Source Code

In der Entwicklung dieser Arbeit sind über 15 einzelne, separat lauffähige SpinalHDL Projekte entstanden. Eine Übersicht dieser Projekte ist in Abbildung 5.1 dargestellt. Es stehen mehrere Begründungen hinter der Idee, eine solche Menge an einzelnen und teilweise redundanten Projekten anzulegen. Eine der Begründungen wurde im letzten Kapitel schon erwähnt, die modulare Entwicklung im Sinne der Softwaretechnik. Diese half bei der Fehlersuche und der Integration der Bestandteile. Eine weitere Begründung ergibt sich aus den Zielen dieser Arbeit. Es sollte evaluiert werden, in wie weit die maximale Verwendung von Open Source Hardware, Tools und Sprachen in der FPGA Programmierung für eine Verwendung in der Lehre geeignet ist. Wenn möglich, sollten die erstellten Implementierungen so aufbereitet werden, dass sie als Einstiegsbeispiele für SpinalHDL unter Verwendung des ULX3S FPGA Boards dienen. Nicht zuletzt ist eine Vorgabe aus den Zielen, alle Bestandteile dieser Arbeit der Öffentlichkeit unter Open Source Lizenz zur Verfügung zu stellen. Die Projekte sind daher unter einer Open Source Lizenz im GitHub Verzeichnis des Autors veröffentlicht:

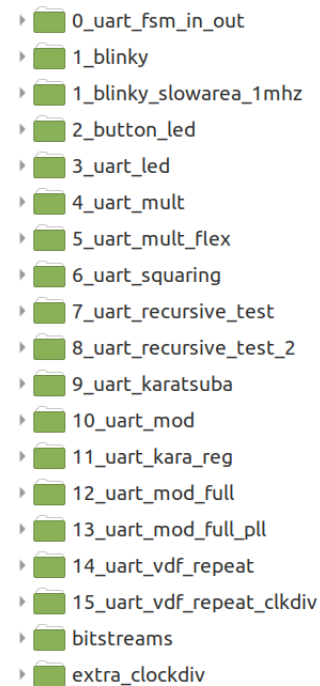


Abb. 5.1.: SpinalHDL Projekte

<https://github.com/ThorKn/SpinalHDL-Projects>, abgerufen am 18.05.2020.

Eine weitere Verwendung für die Lehre, zum Beispiel für Übungsblätter als praktischer Teil eines Kursmoduls, ist damit ausdrücklich erlaubt und erwünscht. Eventuell wird der Autor dieser Arbeit im Rahmen der angestrebten Tätigkeit als wissenschaftlicher Mitarbeiter an der Hochschule RheinMain diese Aufbereitung als Kursmaterial in naher Zukunft selbst vornehmen.

6. Fazit

6.1. Zielerreichung?

Die Erstellung dieses Projekts enthält einige, erst seit kurzem verfügbare Bestandteile:

- SpinalHDL ist eine neue HDL Sprache, die noch stark in der Entwicklung ist.
- Projekt Icestorm mit dem Synthesetool Yosys ist in der ersten Fassung seit 2016 verfügbar.
- Die Tools für den Lattice ECP5 FPGA mit dem Projekt Trellis und NextPnR sind erst seit 2019 voll funktional.
- Die verwendete RISC-V MCU (Kendryte K210) ist seit 2018 erhältlich.

Zu Beginn dieser Arbeit war daher noch nicht sicher, ob das Implementierungsziel mit den gewählten Mitteln erreicht werden kann. Hätten sich auf dem Weg der Implementierung größere Probleme aufgetan, die entweder gar nicht oder nicht im festen Zeitrahmen der Arbeit lösbar gewesen wären, hätte der einzige Ausweg im Wechsel zu proprietären, geschlossenen Tools und Hardware Komponenten gelegen (z.Bsp. Xilinx FPGA und die Vivado Suite). Dieses Szenario ist nicht eingetreten. Auf dem Weg waren einige Adjustierungen notwendig. So war die Thesis in der grundlegenden Idee sehr viel theoretischer, mit dem Fokus auf die theoretische Analyse von Verifiable delay functions geplant. Letztendlich wurde aber ein großer Teil des verfügbaren Zeitrahmens auf die Anwendung der neuen Tools und Plattformen verwendet. In der Ausführlichkeit der Kapitel in dieser Arbeit ist diese Entwicklung eindeutig erkennbar. Die Hauptbestandteile sind durch die umfangreiche Implementierung und deren Evaluation gegeben. Der erzeugte Mehrwert der Ergebnisse dieser Arbeit ist daher auch eher im Bereich der Implementierung und den daraus abgeleiteten Erkenntnissen zu finden. In den folgenden Kapiteln wird analysiert, in wie weit die gesetzten Ziele dieser Arbeit umgesetzt und erreicht wurden.

6.1.1. Open Source

In der Zieldefinition in Kapitel 1.1.3 sind einige zu klärende Fragen zur Verwendung von Open Source Komponenten enthalten. Ein Teil der Antworten ist in den einzelnen Kapiteln dieser Arbeit schon beantwortet. Im Folgenden werden die Fragen nochmals genannt und die Antworten aus den Kapiteln zusammen dargestellt.

Kann ein vollständiger Open Source Workflow für die FPGA Programmierung erreicht werden? Wie sieht dieser Workflow aus und welche Probleme entstehen?

In der folgenden Tabelle 6.1 sind alle verwendeten Komponenten und die Angabe ob sie als Open Source verfügbar sind abgegeben.

Bestandteil	Beschreibung	Lizenz
Entwicklungssystem	Laptop (Intel i5)	Geschlossen
	Betriebssystem Mint 19.3	Open Source
	Atom IDE	Open Source
FPGA Plattform	Lattice ECP5 Chip	Reverse engineered
	ULX3S Board	Open Source
RISC-V Plattform	RISC-V ISA	Open Source
	Kendryte K210 RISC-V Chip	Geschlossen
	Maixduino Board	Geschlossen
FPGA Tools	Yosys	Open Source
	Projekt Trellis	Open Source
	NextPnR	Open Source
	ujprog	Open Source
	SpinalHDL	Open Source
RISC-V Tools	uPyLoader	Open Source
	uPython	Open Source
Thesis	L ^A T _E X	Open Source
	TexMaker	Open Source

Tab. 6.1.: Projektbestandteile: Open Source

Wie aus der Tabelle erkennbar, sind fast alle verwendeten Komponenten als Open Source verfügbar. Es fällt auf, dass die geschlossenen Bestandteile ausschließlich die Chips der verwendeten Hardware sind. Hier zeigt sich ein allgemeines Problem, welches nicht in der Entwicklung von FPGAs, sondern eine Ebene tiefer liegt. Die Produktion von Elektronik-Chips, sogenannte „Anwendungsspezifische integrierte Schaltungen (ASIC)“, ist bisher noch nicht mit Open Source Werkzeugen machbar. In der Produktion von ASICs sind nicht nur Software Tools im Einsatz. Viele der Prozesse in der Fertigung sind physikalischer Natur. So zum Beispiel die Erstellung der Belichtungsmasken und die Verarbeitung der Silizium Wafer. Erst wenn auch diese Prozesse als

Open Source erhältlich sind, bestehen Chancen auf wirkliche Open Source Hardware in Form von ASICs.

Der Workflow mit diesen Tools ist in den Kapiteln 4.2.2 und 4.2.3 beschrieben. Für die FPGA Tools wurde ein Makefile geschrieben, welches aus den Verilog Dateien bis zum Bitstream alle Zwischenprodukte automatisiert erzeugt. Es sind zwei Konfigurationsdateien zusätzlich zu erstellen. Die eine Datei (`<Projekt>.ys`) enthält Anweisungen für das Synthesetool Yosys. Die andere Datei (`<FPGA Board>.lpf`) enthält die Zuweisungen der Leitungen im Quellcode zu den physikalischen Anschlüssen des FPGA Boards. Dieser Aufwand ist im Vergleich zu anderen FPGA Toolchains sehr gering und ermöglicht einen schnellen und übersichtlichen Workflow. Die Benutzung von SpinalHDL ist ähnlich einfach. Hier ist das Scala-Build-Tool (SBT) zu benutzen. Hierfür ist nur eine Datei (`build.sbt`) zu erstellen und der zu verarbeitende Quellcode in einer speziellen Verzeichnisstruktur anzulegen. Danach wird das Projekt mit der Konsoleneingabe `sbt run` gestartet. Es entstehen die fertigen Verilog (oder auch VHDL) Dateien und optional eine Simulationsdatei.

Die entstandenen Probleme in der Verwendung von den oben genannten Open Source Tools sind eher gering. Es seien hier zwei Vorfälle genannt, die relativ viel Zeit zur Fehlersuche in Anspruch genommen haben:

Build-Versionen: Bei der Entwicklung mit solch neuen Tools und FPGA Plattformen sollten die Tools immer mit der neusten, stabilen Version selbst gebaut werden. Manchmal sind aber auch in einer stabilen Version noch Fehler enthalten, die nur bisher nicht entdeckt wurden. Die Entwicklung dieses Projekts fand auf drei unterschiedlichen Computern mit gleichem Betriebssystem (Mint 19.3) statt. Die FPGA Tools wurden auf diesen Computern zu unterschiedlichen Zeitpunkten jeweils aktuell gebaut. An einem Punkt der Implementierung hat nur einer dieser Computer einen funktional korrekten Bitstream erzeugt. Und zwar der, mit der ältesten Version der Tools. Das Problem konnte nur behoben werden, indem die anderen zwei Computer mit der gleichen, älteren Version der Tools versehen wurden. Hierfür wurden die GitHub Commit Hashwerte verglichen und die älteren Versionen auf den zwei betroffenen Computern neu gebaut.

Ewiges Place and Route: Mit der größtmöglichen Bitbreite (128 Bit) ergaben sich sehr lange Berechnungszeiten für den Bearbeitungsschritt Place and Route. Mehrere Tage waren hier nicht selten. Es fehlt gänzlich eine Angabe in den Tools, ob der Vorgang noch terminieren könnte oder schon hoffnungslos „ins Leere läuft“. Es scheint, dass manche Arbeitsschritte in den FPGA Tools (noch) nicht parallelisiert sind. D.h. die Rechenzeit hängt stark von der Leistungsfähigkeit eines einzelnen CPU Kerns ab.

Müssen graphische Oberflächen benutzt werden?

Nein, die komplette Entwicklung kann mit Texteditoren und Konsolen Buildtools erreicht werden. Die Simulation erfordert ein Anzeigewerkzeug. Hier wurde GTKWave benutzt, was eine schlanke, intuitive graphische Oberfläche bietet.

Entstehen Vorteile für die akademische Ausbildung und für akademische Veröffentlichungen durch die Nutzung von Open Source für FPGAs?

Eindeutig: Ja. Alle mit diesen Tools erstellten Projekte können ohne Kosten selbst gebaut und nachvollzogen werden. Natürlich steht da immer noch die Anschaffung des FPGA als Kostenfaktor, aber der Rest ist mit einem Betriebssystem und einem Internetanschluss erledigt. Die gewonnen Freiheiten in der Nachvollziehbarkeit der Projekte könnten zu besseren und zahlreicheren Codereviews von Veröffentlichungen führen.

Ist eine stark abstrahierte Hardware Beschreibungssprache von Vorteil?

Hier ist die Antwort dialektischer Natur. Auf der einen Seite hilft die Abstraktion von SpinalHDL stark zum schnelleren und gezielteren Arbeiten. Als Beispiele wurden im Kapitel über die Implementierung die Verbindungsklassen `Flow` und `Stream` erwähnt. Ein weiteres Beispiel aus der SpinalHDL Bibliothek waren die Zustandsautomaten. Im Vergleich zu Verilog oder VHDL wird in SpinalHDL einiges leichter und übersichtlicher. Der Quellcode der Karatsuba Multiplikation ist ein schönes Beispiel für die Verständlichkeit und Lesbarkeit von SpinalHDL Code. Auf der anderen Seite nützt dem Programmierer diese Abstraktion nichts, wenn nicht das grundlegende Verständnis für die Funktionsweise der entstehenden Schaltungen vorhanden ist. Aus Sicht des Autors erleichtert SpinalHDL die FPGA Programmierung sehr stark, wenn vorher eine andere, weniger abstrahierende HDL (Verilog oder VHDL) erlernt wurde. Ein Studium der Elektrotechnik würde aber auch helfen.

Welche Funktionalitäten sind (noch) nicht mit Open Source Komponenten erreichbar?

Alle, für die Implementierung geforderten, Funktionalitäten konnten mit den verwendeten Tools erreicht werden. Die Befürchtung, mitten in der Entwicklung auf andere FPGA Plattformen mit proprietären Tools umsteigen zu müssen ist nicht eingetreten. Sicher wären größere FPGAs wünschenswert, aber es scheint nur eine Frage der Zeit zu sein bis das möglich ist. Die Integration von Xilinx Serie-7 FPGAs in die SymbiFlow Tools schreitet schnell voran (siehe die SymbiFlow Übersicht in Abbildung 4.3).

Während der Erstellung der Implementierung wurde der Autor von Kollegen und Kommilitonen wiederholt mit einer speziellen Frage bedacht. Obwohl sie nicht allzu wissenschaftlich klingt, steckt aber mehr Ernst dahinter als offensichtlich ist. Daher sei auch diese Frage (und die sehr subjektive Antwort) hier genannt:

Hat die Entwicklung mit den Open Source Tools Spaß gemacht?

Auch hier ein eindeutiges Ja. Nach Jahren der FPGA Entwicklung mittels Hersteller-Tools war die Einfachheit des Buildprozesses eine Erfrischung. Die Abstraktion von SpinalHDL setzt eine steile Einstiegs-Lernkurve voraus, ist dann aber sehr produktiv. Die Hoffnung ist, dass es noch sehr viel mehr Open Source Entwicklungen für FPGAs geben wird und diese Thesis ein kleiner Baustein dafür ist. Nicht zuletzt kann wissenschaftliches Arbeiten ziemlich trocken und mühsam sein. Diese Tools erleichtern diesen Prozess enorm.

6.1.2. VDF Wettbewerb

Für den Wettbewerb der VDF Alliance steht den Teilnehmern eine Instanz der Amazon AWS FPGA Cloud zur Verfügung. In dieser Cloud sind die Xilinx Virtex Ultra Scale+ mit der Typbezeichnung XCVU9P verbaut. In Tabelle 6.2 ist ein Vergleich des Xilinx FPGA und des in dieser Arbeit verwendeten Lattice FPGA enthalten. Beachtenswert ist hier die Zeile mit der Anzahl der Digitalen Signal Prozessoren (DSP). In den DSP sind die Hardware Multiplizierer integriert. Der Xilinx FPGA enthält die 30-fache Anzahl an Hardware Multiplizierern.

	Xilinx XCVU9P	Lattice ECP5-85k
Logik Zellen	2,586 Mio.	85 k
DSP	6,840	156
Speicher	345,9 MB	3,7 MB
SysClock	300MHz	25MHz
Preis (Dev-Board)	7.000 \$	200 \$

Tab. 6.2.: FPGA Vergleich: Amazon AWS F1 und Lattice ECP5-85k

Die Vorgaben für den VDF Wettbewerb sind 1024 Bit Bitbreite der Implementierungen (D.h. das Modul n ist eine Zahl mit 1024 Bit Länge) und die Referenzzeit von 50ns für eine Berechnung von $x^2 \bmod n$. Für jede Nanosekunde schneller als die Referenz wird ein Preisgeld von 3000\$ ausgegeben.

6. Fazit

An diese Bitbreite und Geschwindigkeit kommt die Implementierung aus dieser Arbeit nicht heran. Als maximale Bitbreite wurden 128 Bit erreicht. Eine Übersicht der Messwerte auf den verwendeten Plattformen sind in den folgenden zwei Tabellen aufgelistet. In Tabelle 6.3 sind die Werte für die Bitbreite 64 Bit enthalten und in Tabelle 6.4 für 128 Bit Bitbreite.

Plattform	Taktfrequenz	μs (1 Mio)	ns (Einzel)	Taktzyklen (Einzel)
FPGA	25 MHz	323.402	323	8
RISC-V	400 MHz	16.143.835	16.143	5381
Intel i5	2,5 GHz	369.355	369	922

Tab. 6.3.: Geschwindigkeitsvergleich 64 Bit: FPGA, RISC-V und Intel i5

Plattform	Taktfrequenz	μs (1 Mio)	ns (Einzel)	Taktzyklen (Einzel)
FPGA	5 MHz	2.088.120	2.088	10
FPGA	1 MHz	10.087.858	10.087	10
RISC-V	400 MHz	24.969.218	24.969	8323
Intel i5	2,5 GHz	439.161	439	1097

Tab. 6.4.: Geschwindigkeitsvergleich 128 Bit: FPGA, RISC-V und Intel i5

Es folgen einige Erklärungen und Analysen zu den Zahlen aus den beiden Tabellen:

Maximale Bitbreite:

Es war zu erwarten, dass die Bitbreite der Implementierung nicht die des Wettbewerbs erreichen wird. Es war zu Beginn der Arbeit noch nicht mal sicher, ob mit den neuen Tools und dem FPGA überhaupt eine stabile Implementierung erreicht wird. Im Ergebnis liegen zwei stabile, gut getestete Varianten vor: 64 Bit und 128 Bit. Damit ist das erste Ziel einer Implementierung mit Open Source Tools erreicht. Ein Vergleich der FPGA Größen in Tabelle 6.2 zeigt, dass die erreichte Bitbreite im Verhältnis zu den jeweiligen Ressourcen (insbesondere DSP) durchaus gleichauf liegt. Mit knapp 7000 DSP wurde eine Bitbreite von 1024 Bit erreicht und mit den 156 DSP eine Bitbreite von 128 Bit. Die wirklich genutzten Ressourcen im ECP5-85k sind in den Anhängen A.7 und A.9 zu sehen. Für die 64 Bit Implementierung wurden nur 13 DSP und für die 128 Bit Implementierung nur 35 DSP genutzt. Trotzdem waren keine höheren Bitbreiten möglich. Dies liegt offensichtlich nicht an der Anzahl der DSP. Das Routing des kompletten Schaltkreises ist hier die Begrenzung. Während die Routingzeit für 64 Bit bei 2 Minuten liegt (Anhang A.8), hat das Routing für 128 Bit mehr als 6 Tage gedauert (Anhang A.9). Bei dieser Steigerung der Routingzeiten erscheint es unwahrscheinlich, dass eine höhere Bitbreite als 128 Bit jemals mit dem Routing fertig geworden wäre. Es wurde ein

Test mit der Xilinx Vivado Suite gestartet, weil auch ein größeres Xilinx FPGA Board verfügbar war. Vivado hat sich aber vollständig geweigert, den rein kombinatorischen Multiplizierer zu synthetisieren. Es wurden ca. 300 kritische Fehler über das Fehlen der Ein- und Ausgangsregister an den DSP gemeldet und die Synthese abgebrochen. Daher liegen hier keine Vergleichswerte vor.

Maximale Taktfrequenz:

Der ECP5-85k FPGA wird auf dem ULX3S Board mit einem Systemtakt von 25MHz betrieben. Die 64 Bit Implementierung erreicht als Maximum ziemlich genau diese 25 MHz. Die 128 Bit Implementierung musste mit langsameren Takten betrieben werden. Hier liegt das Maximum bei 5 MHz, also um dem Faktor 5 langsamer. Dies ist durch die rein kombinatorische Logik der Karatsuba Multiplikation begründet. Je größer die Kombinatorik wird, um so länger werden die Leitungen und die mögliche Taktfrequenz wird langsamer. Ob daher eine Karatsuba Multiplikation mit Registern in den Rekursionsstufen sinnvoller wäre (staged), wird weiter unten diskutiert.

Taktzyklen pro Ausführung:

Die Anzahl der benötigten Taktzyklen einer vollständigen Quadrierung und Modulo-rechnung wurde mit $t = 1$ Mio. Ausführungen gemessen und dann auf die einzelne Berechnung geteilt. Somit kann der Kommunikationsaufwand über die UART Schnittstelle vernachlässigt werden. Bei 64 Bit benötigt der FPGA 8 Takte für die Berechnung. In der 128 Bit Implementierung sind es 2 Takte mehr, also 10 Takte. Diese zwei zusätzlichen Taktzyklen werden in der Überlaukorrektur der Barret Reduktion benötigt. Es ist nicht ganz klar warum, aber die 128 Bit Implementierung lief erst fehlerfrei mit dieser Modifikation. Die Anzahl der Taktzyklen entspricht genau der Erwartung aus dem Zustandsautomaten im Quellcode (Anhang A.5). In den Zustandsautomaten können die Zustände und deren angegebene Wartezeit abgezählt werden. Und es sind dann genau die 8 (bzw. 10) Takte aus der Messung. Die Messung stimmt also mit der Erwartung aus dem Quellcode überein. Die Anzahl der Taktzyklen auf den anderen Plattformen (RISC-V und i5) sind nicht als „schnellstmögliche Implementierungen“ anzusehen. Die Software Implementierungen sind in Mikropython (RISC-V) und Python3.6 (Intel i5) geschrieben. Obwohl Python mit Integerberechnungen recht schnell ist, würde C oder Assembler wahrscheinlich um größere Faktoren schneller werden. Die geringste Anzahl an Taktzyklen in sequentieller Ausführung (Software) hat die i5 CPU mit 922 Zyklen für die 64 Bit Berechnung erreicht. Das Maximum liegt in der Berechnung von 128 Bit auf dem RISC-V bei über 8000 Zyklen. In der Gesamtbewertung ist eine Multiplikation (64 Bit und 128 Bit) in nur einem Takt aus dem FPGA als Beschleuniger für kleine Controller (z.Bsp. RISC-V) sehr attraktiv. Und ein Maximum von 10 Takten für die gesamte Berechnung ist auch sehr wenig.

6. Fazit

Gesamtgeschwindigkeit:

In der 64 Bit Variante schlägt der FPGA bei 25 MHz sowohl die RISC-V und die i5 Implementierung. Wie schon erwähnt, würde dies mit Software Implementierung in hardwarenäheren Sprachen wahrscheinlich anders aussehen. Aber trotzdem zeigen die Werte, dass auch mit Open Source FPGAs und Tools schon beachtliche Resultate erreicht werden können. Als Beschleuniger für einen RISC-V macht auch diese Implementierung in der jetzigen Variante schon Sinn. Bei 64 Bit ist der FPGA um den Faktor 50 schneller als der RISC-V. Mit dem reduzierten Takt (5 MHz) in der 128 Bit Implementierung ist der Speedup Faktor zum RISC-V immer noch 10. Das der FPGA mit 64 Bit Bitbreite sogar schneller als der Python Code auf dem Intel i5 ist, war unerwartet und wird mit der Taktreduzierung bei 128 Bit nicht mehr erreicht.

Kombinatorisch versus Staged:

Es bleibt noch die Frage zu beantworten, ob eine nicht kombinatorische Implementierung für die Karatsuba Multiplikation besser wäre. Aus den Messwerten zeigt sich, dass für 64 Bit der volle Systemtakt des ULX3S Boards verwendet werden kann. Die Anzahl der benötigten Registerstufen für eine staged Implementierung ist ungefähr die Rekursionstiefe mal zwei, da sowohl am Ab- als auch am Aufstieg aus der Rekursion jeweils eine Registerstufe benötigt würde. Eine grobe Abschätzung für 64 Bit wäre daher: Rekursionstiefe ist zwei. Registerstufen werden dann vier benötigt. Es würde sich eine geschätzte Erhöhung der benötigten Taktzyklen um 4 Takte ergeben. Für 128 Bit wäre diese Abschätzung 6 Takte mehr. Es ist wahrscheinlich, dass die staged Implementierung auch mit 128 Bit Breite auf dem vollen FPGA Takt von 25 MHz funktionieren würde. Rein kombinatorisch sind nur 5 MHz möglich, also eine Reduktion um den Faktor 5. Der Vergleich der „Mehr-Takte“ zur „Frequenzreduktion“ ergibt eine Schere, in der die 64 Bit Implementierung als kombinatorische Lösung noch ein Gesamtvorteil hat. Aber schon bei 128 Bit könnte eine staged Version in der Gesamtgeschwindigkeit auf dem ULX3S Board schneller werden.

6.1.3. In der Lehre

Die Verwendung von Open Source FPGAs und Tools für den Einsatz in der Lehre ist möglich. Das verwendete FPGA Board ULX3S wurde sogar mit der Vorgabe dieses Verwendungsszenarios entwickelt. Es enthält eine reichhaltige und durchdachte Auswahl an Schnittstellen und der Lattice ECP5-85k hat genug Ressourcen für mittelgroße Projekte. Insgesamt ist der Buildprozess mit Yosys und NextPnR so einfach, dass sich dadurch die Lehre mehr auf die Vermittlung der Programmierung konzentrieren könnte, anstatt viel Zeit im Tooling verbringen zu müssen. Die Installation der Toolchain sollte hier kein Hinderungsgrund sein.

Für den Einstieg reichen meist auch die fertig kompilierten Versionen aus den Repositories der Distributionen. Aber auch das Selbstbauen der Tools ist gut dokumentiert und funktioniert ohne Komplikationen.

Die Verwendung von SpinalHDL erfordert (nach Ansicht des Autors) nicht nur eine steile Einstiegslernkurve, sondern auch ein fundamentiertes Wissen um die Funktionsweise der zu entwickelnden Schaltkreise. Es ist eventuell nicht ratsam, sondern eher verwirrend, mit SpinalHDL als erste HDL Sprache in die Programmierung von FPGAs einzusteigen. Auch die bisher noch fehlende Literatur könnte hier ein hinderlicher Faktor sein. Die Dokumentation von SpinalHDL wird beständig weiter verbessert, aber es gibt nicht ein einziges Lehrbuch zu dieser Sprache. Das ist mit Verilog oder VHDL anders. Als Aufbau-Kurs für fortgeschrittene FPGA-Entwickler ist SpinalHDL aber mit Sicherheit eine sinnvolle Bereicherung des Lehrangebots.

Wie in Kapitel 5 beschrieben, sind im Rahmen dieser Arbeit mehr als 15 Einzelprojekte in SpinalHDL entstanden. Diese sind in dem erwähnten. öffentlichen GitHub des Autors unter Open Source Lizenz verfügbar. Es ist ausdrücklich erwünscht und erlaubt, diese als Vorlage für eventuelles Kursmaterial zu verwenden.

6.1.4. Demonstrator

Ein funktionaler Demonstrator, wie in Kapitel 6 beschrieben, wurde fertiggestellt. Einige der Messergebnisse, die weiter oben besprochen wurden, wurden mit dem Demonstrator erstellt. Die Ausgabe auf dem Display des Demonstrators zeigen die Eingabeparameter und die berechneten Ergebnisse vom FPGA und vom RISC-V an. Eine Angabe, ob die Ergebnisse gleich sind, wird auch ausgegeben. Zusätzlich werden noch die Berechnungszeiten angezeigt. Das Ziel, einen funktionsfähigen Demonstrator zu entwickeln wurde erreicht.

A. Anhänge

A.1. FPGA Toolchain: Makefile

```
1  .PHONY: all
2  .DELETE_ON_ERROR:
3  TOPMOD    := <Top Verilog module here>
4  VLOGFIL   := $(TOPMOD).v
5  VCDFILE   := $(TOPMOD).vcd
6  SIMPROG   := $(TOPMOD)_tb
7  RPTFILE   := $(TOPMOD).rpt
8  BINFILE   := $(TOPMOD).bin
9  SIMFILE   := $(SIMPROG).cpp
10 VDIRFB    := ./obj_dir
11 PROGPATH   := ~/bin/ujprog/ujprog

13 all: $(VCDFILE)

15 .PHONY: clean
16 clean:
17     rm -rf $(VDIRFB)/ $(SIMPROG) $(VCDFILE) $(TOPMOD)/ $(BINFILE) $(RPTFILE)
18     rm -rf $(TOPMOD).json $(TOPMOD).config $(TOPMOD).bit

21 $(TOPMOD).bit: $(TOPMOD).config
22     ecppack $(TOPMOD).config $(TOPMOD).bit

24 $(TOPMOD).config: $(TOPMOD).json
25     nextpnr-ecp5 --85k --package CABGA381 \
26         --freq 1 \
27         --json $(TOPMOD).json \
28         --lpf ulx3s_v20_constraints.lpf \
29         --textcfg $(TOPMOD).config

31 $(TOPMOD).json: $(TOPMOD).ys $(TOPMOD).v
32     yosys $(TOPMOD).ys

34 prog: $(TOPMOD).bit
35     sudo $(PROGPATH) $(TOPMOD).bit
```

Listing A.1: Makefile der FPGA Toolchain

A.2. Kleines SpinalHDL Beispiel: Blinky

```
1 import spinal.core._
2 import spinal.core.sim._
3 import spinal.lib._
4 import spinal.lib.fsm._
5 import spinal.lib.com.uart._

7 import scala.util.Random

9 class Blinky extends Component{
10     val io = new Bundle{
11         val led = out Bool
12     }

14     val fsm = new StateMachine{
15         val counter1 = Counter(0 to 12500000)
16         val counter2 = Counter(0 to 12500000)
17         io.led := False

19         val stateA : State = new State with EntryPoint{
20             whenIsActive {
21                 goto(stateB)
22             }
23         }

25         val stateB : State = new State{
26             onEntry {
27                 counter1.clear()
28             }
29             whenIsActive {
30                 io.led := True
31                 counter1.increment()
32                 when(counter1.willOverflow){
33                     goto(stateC)
34                 }
35             }
36         }

38         val stateC : State = new State{
39             onEntry {
40                 counter2.clear()
41             }
42             whenIsActive {
43                 io.led := False
44                 counter2.increment()
45                 when(counter2.willOverflow){
46                     goto(stateA)
47                 }
48             }
49         }
50     }
51 }
```

A.2. Kleines SpinalHDL Beispiel: Blinky

```
56 object UartFsmInOutMain{
57   def main(args: Array[String]) {
58     SpinalConfig(
59       mode = Verilog,
60       defaultClockDomainFrequency = FixedFrequency(25 MHz)
61     ).generate(new Blinky)

63     val spinalConfig = SpinalConfig(defaultClockDomainFrequency = FixedFrequency(25 MHz))

65     SimConfig
66       .withConfig(spinalConfig)
67       .withWave
68       .compile(new Blinky)
69       .doSim{ blinky =>
70         blinky.clockDomain.forkStimulus(2)
71       }
72       var idx = 0
73       while(idx < 3000000){
74         blinky.clockDomain.waitSampling()
75         idx += 1
76       }
77     }
78   }
79 }
```

Listing A.2: SpinalHDL Beispiel: Blinky.scala

A.3. UartFsmInOut: SpinalHDL Quellcode

```

1  import spinal.core._
2  import spinal.core.sim._
3  import spinal.lib._
4  import spinal.lib.fsm._
5  import spinal.lib.com.uart._
6  import scala.math

8  class UartFsmInOut(width: Int) extends Component{
9      val io = new Bundle{
10         val uart = master(Uart())
11         val toFnct = master Flow(Bits(((3*width)+48) bits))
12         val fromFnct = slave Flow(Bits(width bits))
13     }

15     // Setup UartCtrl
16     val uartCtrl = new UartCtrl()
17     uartCtrl.io.config.setClockDivider(HertzNumber(9600))
18     uartCtrl.io.config.frame.dataLength := 7 //8 bits
19     uartCtrl.io.config.frame.parity := UartParityType.NONE
20     uartCtrl.io.config.frame.stop := UartStopType.ONE
21     uartCtrl.io.uart <> io.uart

23     val write = Stream(Bits(8 bits))
24     write >-> uartCtrl.io.write

26     // Log2 Helper
27     var log2 = (x: Double) => math.log(x)/math.log(2.0)

29     // Statemachine for receiving the payload, storing
30     // it into a buffer and then sending the buffer
31     // onto the toFnct Flow
32     val fsmIn = new StateMachine{
33         val inCounter = Reg(UInt(math.ceil(log2(((3*width)+48)/8)).toInt bits)) init (0)
34         val buffer = Reg(Bits((3*width)+48 bits)) init(0)
35         io.toFnct.payload := 0
36         io.toFnct.valid := False

38         val inEntry : State = new State with EntryPoint{
39             whenIsActive{
40                 goto(inReadByte)
41             }
42             onExit{
43                 inCounter := 0
44                 buffer := 0
45             }
46         }

47         val inReadByte : State = new State{
48             whenIsActive{
49                 when(uartCtrl.io.read.valid){
50                     buffer(inCounter*8, 8 bits) := uartCtrl.io.read.payload.asBits
51                     goto(inIncCounter)
52                 }
53             }
54         }

```



```

56     val inIncCounter : State = new State{
57         whenIsActive{
58             inCounter := inCounter + 1
59             when(inCounter === (((3*width)+48)/8-1)){
60                 goto(inBufToPayload)
61             }.otherwise{
62                 goto(inReadByte)
63             }
64         }
65     }
66     val inBufToPayload : State = new State{
67         whenIsActive{
68             io.toFnct.payload := buffer.asBits
69             io.toFnct.valid := True
70             goto(inEntry)
71         }
72     }
73 }

75 // Statemachine for sending the payload over Uart
76 val fsmOut = new StateMachine{
77     val outCounter = Reg(UInt(math.ceil(log2(width/8)).toInt bits)) init (0)
78     val outBuffer = Reg(Bits(width bits)) init(0)
79     write.valid := False
80     write.payload := 0

82     val outEntry : State = new State with EntryPoint{
83         whenIsActive{
84             when(io.fromFnct.valid){
85                 outCounter := 0
86                 outBuffer := io.fromFnct.payload
87                 goto(outWriteByte)
88             }
89         }
90     }
91     val outWriteByte : State = new State {
92         whenIsActive{
93             write.payload := outBuffer(outCounter*8, 8 bits)
94             write.valid := True
95             when(write.fire) {
96                 goto(outIncCounter)
97             }
98         }
99     }
100    val outIncCounter : State = new State{
101        whenIsActive{
102            outCounter := outCounter + 1
103            when(outCounter === ((width/8)-1)){
104                goto(outEntry)
105            }.otherwise{
106                goto(outWriteByte)
107            }
108        }
109    }
110 }
111 }

```

Listing A.3: UartFsmInOut.scala

A.4. Karatsuba Multiplikation als Klasse (SpinalHDL)

```

1  class KaraMult(width: Int) extends Component{
2      val io = new Bundle{
3          val a = in Bits(width bits)
4          val b = in Bits(width bits)
5          val result = out Bits(2*width bits)
6      }

8      if (width <= 16){
9          io.result := (io.a.asUInt*io.b.asUInt).asBits.resize(width*2)
10     } else {
11         // Splitting a and b into two parts with length width/2 each (high and low)
12         val al = io.a(0 to ((width/2)-1))
13         val ah = io.a((width/2) to (width-1))
14         val bl = io.b(0 to ((width/2)-1))
15         val bh = io.b((width/2) to (width-1))

17         // Calc ahl and bhl as signed integers
18         val alh_int = (al.resize(width/2+1).asSInt - ah.resize(width/2+1).asSInt)
19         val bhl_int = (bh.resize(width/2+1).asSInt - bl.resize(width/2+1).asSInt)

21         // Recursions
22         val p1 = new KaraMult(width/2)
23         p1.io.a := ah
24         p1.io.b := bh
25         val rh = p1.io.result

27         val p2 = new KaraMult(width/2)
28         p2.io.a := al
29         p2.io.b := bl
30         val rl = p2.io.result

32         val p3 = new KaraMult(width/2)
33         p3.io.a := alh_int.abs.asBits.resize(width/2)
34         p3.io.b := bhl_int.abs.asBits.resize(width/2)
35         val rm = p3.io.result

37         // Calc the results from the recursions and return them
38         val result_m = Bits ((width + (width/2) + 2) bits)
39         when((alh_int < 0) ^ (bhl_int < 0)){
40             result_m := (((rh.asUInt +^ rl.asUInt - rm.asUInt).asBits) << width/2).resized
41         }.otherwise {
42             result_m := (((rh.asUInt +^ rl.asUInt +^ rm.asUInt).asBits) << width/2).resized
43         }
44         val result_h = rh << width
45         val result_l = rl

47         io.result := (result_h.asUInt +^ result_m.asUInt +^ result_l.asUInt).asBits.resize(width*2)
48     }
49 }

```

Listing A.4: Karatsuba Multiplikation als SpinalHDL Klasse

A.5. MultMod: SpinalHDL Quellcode inkl. Debugging Leitungen

```

1  import spinal.core._
2  import spinal.core.sim._
3  import spinal.lib._
4  import spinal.lib.fsm._
5  import spinal.sim._

7  import scala.util.Random

9  class MultMod(width: Int) extends Component{
10     val io = new Bundle{
11         val fromUart      = slave Flow(Bits(((3*width)+48) bits))
12         val toUart        = master Flow(Bits(width bits))
13         val temp_1_out    = out Bits(width*2 bits)
14         val temp_2_out    = out Bits(width*2 bits)
15         val temp_2a_out   = out Bits(width bits)
16         val temp_2b_out   = out Bits(width*2 bits)
17         val temp_2c_out   = out Bits(width*2 bits)
18         val temp_3_out    = out Bits(width*2 bits)
19         val temp_4_out    = out Bits(width*2 bits)
20         val temp_5_out    = out Bits(width*2 bits)
21         val temp_6_out    = out Bits(width*2 bits)
22         val temp_7_out    = out Bits(width*2 bits)
23         val temp_corr_flag = out UInt(3 bits)
24         val temp_corr_flag_2 = out UInt(3 bits)
25     }

27     val fsm = new StateMachine{
28         val a      = Reg(Bits(width bits)) init(0)
29         val mod_n   = Reg(Bits(width bits)) init(0)
30         val mue     = Reg(Bits(width + 8 bits)) init(0)
31         val k       = Reg(Bits(8 bits)) init(0)
32         val t       = Reg(Bits(32 bits)) init(0)
33         val result_mult = Reg(Bits(width*2 bits)) init(0)
34         val result_barret = Reg(Bits(width*2 bits)) init(0)
35         val mult     = new KaraMult(width)
36         val temp_2    = Reg(Bits(width*2 bits)) init(0)
37         val temp_3    = Reg(Bits(width*2 bits)) init(0)
38         val temp_7    = Reg(SInt(width*2 bits)) init(0)
39         val temp_8    = Reg(UInt(width*2 bits)) init(0)
40         val mask      = Reg(Bits(width*2 bits)) init(0)
41         val repeatCounter = Reg(UInt(32 bits)) init (0)

43         io.toUart.payload := 0
44         io.toUart.valid := False
45         mult.io.a        := 0
46         mult.io.b        := 0
47         io.temp_1_out    := 0
48         io.temp_2_out    := 0
49         io.temp_2a_out   := 0
50         io.temp_2b_out   := 0
51         io.temp_2c_out   := 0
52         io.temp_3_out    := 0
53         io.temp_4_out    := 0

```

A. Anhänge

```
54      io.temp_5_out      := 0
55      io.temp_6_out      := 0
56      io.temp_7_out      := 0
57      io.temp_corr_flag  := 0
58      io.temp_corr_flag_2 := 0

61      val stateA : State = new State with EntryPoint{
62          whenIsActive {
63              when(io.fromUart.valid) {
64                  a      := io.fromUart.payload.asBits(width-1 downto 0)
65                  mod_n   := io.fromUart.payload.asBits((2*width)-1 downto (width))
66                  mue     := io.fromUart.payload.asBits((3*width)+8-1 downto (2*width))
67                  k       := io.fromUart.payload.asBits((3*width)+16-1 downto (3*width)+8)
68                  t       := io.fromUart.payload.asBits((3*width)+48-1 downto (3*width)+16)
69                  repeatCounter := 0
70                  goto(stateB)
71              }
72          }
73      }

75      val stateB : State = new StateDelay(cyclesCount=1){
76          whenIsActive {
77              mult.io.a := a
78              mult.io.b := a
79          }
80          whenCompleted {
81              result_mult := mult.io.result
82              goto(stateC)
83          }
84      }

86      val stateC : State = new StateDelay(cyclesCount=1){
87          whenIsActive {
88              val temp_1 = result_mult >> (16*k.asUInt)
89              mult.io.a := mue.resize(width)
90              mult.io.b := temp_1.resize(width)
91              temp_2 := mult.io.result
92              io.temp_1_out := temp_1
93              io.temp_2_out := temp_2
94          }
95          whenCompleted {
96              goto(stateD)
97          }
98      }

100     val stateD : State = new State{
101         whenIsActive {
102             val temp_2a = (result_mult >> (16*k.asUInt)).resize(width)
103             mult.io.a := temp_2a
104             mult.io.b := mue(width+7 downto width).resize(width)
105             val temp_2b = (mult.io.result << width).resize(width*2)
106             val temp_2c = (temp_2.asUInt + temp_2b.asUInt).asBits
107             io.temp_2a_out := temp_2a
108             io.temp_2b_out := temp_2b
109             io.temp_2c_out := temp_2c
110             temp_3 := temp_2c >> (16*k.asUInt)
111             io.temp_3_out := temp_2c >> (16*k.asUInt)
```

A.5. MultMod: SpinalHDL Quellcode inkl. Debugging Leitungen

```
112         goto(stateE)
113     }
114 }

116 val stateE : State = new StateDelay(cyclesCount=2){
117     whenIsActive {
118         mask := B(width*2 bits, default -> true) >> (2*width - (16*(k.asUInt+1)))
119         io.temp_3_out := temp_3
120         mult.io.a := temp_3.resize(width)
121         mult.io.b := mod_n
122         val temp_4 = mult.io.result
123         val temp_5 = (result_mult & mask).asSInt
124         val temp_6 = (temp_4 & mask).asSInt
125         temp_7 := temp_5 - temp_6
126         /**
127         //result_barret := temp_7.asBits
128         //io.temp_corr_flag := 6
129         ***/

131         io.temp_4_out := temp_4
132         io.temp_5_out := temp_5.asBits
133         io.temp_6_out := temp_6.asBits
134         io.temp_7_out := temp_7.asBits
135     }
136     whenCompleted {
137         goto(stateF)
138     }
139 }

141 val stateF : State = new StateDelay(cyclesCount=1){
142     whenIsActive {
143         when(temp_7 < 0){
144             temp_8 := (temp_7 + mask.asSInt + 1).asUInt
145             io.temp_corr_flag := 1
146         }.otherwise{
147             temp_8 := temp_7.asUInt
148             io.temp_corr_flag := 2
149         }
150     }
151     whenCompleted{
152         goto(stateG)
153     }
154 }

156 val stateG : State = new StateDelay(cyclesCount=3){
157     whenIsActive {
158         when(temp_8 > 4*mod_n.asUInt){
159             result_barret := (temp_8 - 4*mod_n.asUInt).asBits
160             io.temp_corr_flag_2 := 1
161         }.otherwise{
162             when(temp_8 > 3*mod_n.asUInt){
163                 result_barret := (temp_8 - 3*mod_n.asUInt).asBits
164                 io.temp_corr_flag_2 := 2
165             }.otherwise{
166                 when(temp_8 > 2*mod_n.asUInt){
167                     result_barret := (temp_8 - 2*mod_n.asUInt).asBits
168                     io.temp_corr_flag_2 := 3
169                 }.otherwise{
```

A. Anhänge

```
170             when(temp_8 > mod_n.asUInt) {
171                 result_barret := (temp_8 - mod_n.asUInt).asBits
172                 io.temp_corr_flag_2 := 4
173             }.otherwise{
174                 result_barret := temp_8.asBits
175                 io.temp_corr_flag_2 := 5
176             }
177         }
178     }
179 }
180 }
181 whenCompleted{
182     goto(stateH)
183 }
184 }

186 val stateH : State = new StateDelay(cyclesCount=1){
187     whenIsActive {
188         repeatCounter := repeatCounter + 1
189         a := result_barret.resize(width)
190     }
191     whenCompleted{
192         when(repeatCounter === t.asUInt) {
193             goto(stateI)
194         }.otherwise{
195             goto(stateB)
196         }
197     }
198 }

200 val stateI : State = new StateDelay(cyclesCount=2){
201     whenIsActive {
202         io.toUart.payload := result_barret.resize(width)
203         io.toUart.valid := True
204     }
205     whenCompleted{
206         goto(stateA)
207     }
208 }
209 }

211 Hier folgt der Quellcode der Karatsuba Klasse aus Anhang A.4.:

213 class KaraMult(width: Int) extends Component{
214     ... (siehe weiter oben)
215 }
216 }
```

Listing A.5: MultMod.scala

A.6. Ansteuerung RISC-V: Mikropython Implementierung

```

1  import lcd
2  import os
3  import utime
4  from board import board_info
5  from fpioa_manager import fm
6  from machine import UART

8  WIDTH = 8

10 # A class for handling the display and UART on the Maixduino
11 class Maixduino():
12     def __init__(self):
13         # Init LCD with start message
14         lcd.init(color=(0,0,0))
15         lcd.draw_string(0,0, "Square and modulo with HW-Accelerator", lcd.YELLOW, lcd.BLACK)

17         # Register UART1 pins 10 + 11
18         fm.register (board_info.PIN10, fm.fpioa.UART2_TX)
19         fm.register (board_info.PIN11, fm.fpioa.UART2_RX)

21         # Open UART1
22         self.uart = UART (UART.UART2, 115200, 8, None, 1, timeout = 1000, read_buf_len = 4096)

24     def text_to_display(self, x, y, text):
25         lcd.draw_string(x, y, text, lcd.YELLOW, lcd.BLACK)

27     # Write bytes to uart
28     def write_to_uart(self, msg):
29         for e in reversed(msg):
30             self.uart.write(e.to_bytes(1, 'big'))

32     # Read bytes from uart
33     def read_from_uart(self):
34         msg = self.uart.read(WIDTH)
35         while (msg == None):
36             msg = self.uart.read(WIDTH)
37         return msg

39 # Setup of the parameters for the calculation
40 a = os.urandom(WIDTH)
41 a_int = int.from_bytes(a, 'big')
42 n = os.urandom(WIDTH)
43 n_int = int.from_bytes(n, 'big')
44 a_int = a_int % n_int
45 a = a_int.to_bytes(WIDTH, 'big')
46 b_int = pow(2,16)
47 k_int = 4
48 k = k_int.to_bytes(1, 'big')
49 t_int = 1000000
50 t = t_int.to_bytes(4, 'big')
51 mue_int = pow(b_int, 2*k_int) // n_int
52 mue = mue_int.to_bytes(WIDTH+1, 'big')

```

A. Anhänge

```
54 md = Maixduino()
55 utime.sleep(2)

57 # Write setup parameters to display
58 md.text_to_display(0, 20, "a: " + hex(a_int))
59 md.text_to_display(0, 40, "n: " + hex(n_int))
60 md.text_to_display(0, 60, "mue: " + hex(mue_int))
61 md.text_to_display(0, 80, "k: " + hex(k_int))
62 md.text_to_display(0, 100, "t: " + hex(t_int))

64 # Write setup parameters to UART
65 fpgastart = utime.ticks_us()
66 md.write_to_uart(a)
67 md.write_to_uart(n)
68 md.write_to_uart(mue)
69 md.write_to_uart(k)
70 md.write_to_uart(t)
71 md.text_to_display(250, 100, "Sent!")

73 # Read result from UART
74 result = md.read_from_uart()
75 fpgastop = utime.ticks_us()
76 result_int = int.from_bytes(result, 'little')

78 # Calc FPGA time
79 fpgatime = utime.ticks_diff(fpgastop, fpgastart)

81 # Display FPGA result
82 md.text_to_display(0, 140, "FPGA: " + str(hex(result_int)))
83 md.text_to_display(0, 160, "FPGA Time: " + str(fpgatime))

85 # Do the RISC-V MCU calc
86 b_int = a_int
87 riscstart = utime.ticks_us()
88 for _ in range(t_int+1):
89     z_int = (b_int * b_int) % n_int
90     b_int = z_int
91 riscstop = utime.ticks_us()

93 # Calc RISC-V time
94 risctime = utime.ticks_diff(riscstop, riscstart)

96 # Display RISC-V result
97 md.text_to_display(0,180, "RISCV: " + str(hex(z_int)))
98 md.text_to_display(0,200, "RISCV Time: " + str(risctime))

100 # Correct?
101 if (result_int == z_int):
102     md.text_to_display(250,220, "Correct!")
103 else:
104     md.text_to_display(250,220, "Failure!")
```

Listing A.6: Mikropython auf dem RISC-V

A.7. NextPnR-ECP5 Logging output

```

1 Info: Device utilisation:
2 Info:          TRELLIS_SLICE: 3837/41820      9%
3 Info:          TRELLIS_IO:    4/   365      1%
4 Info:          DCCA:         1/    56      1%
5 Info:          DP16KD:        0/   208      0%
6 Info:          MULT18X18D:    13/   156      8%
7 Info:          ALU54B:        0/    78      0%
8 Info:          EHXPPLL:       1/     4     25%
9 Info:          EXTREFB:       0/     2      0%
10 Info:          DCUA:         0/     2      0%
11 Info:          PCSCLKDIV:     0/     2      0%
12 Info:          IOLOGIC:      0/   224      0%
13 Info:          SIOLOGIC:     0/   141      0%
14 Info:          GSR:          0/     1      0%
15 Info:          JTAGG:        0/     1      0%
16 Info:          OSCG:         0/     1      0%
17 Info:          SEDGA:        0/     1      0%
18 Info:          DTR:          0/     1      0%
19 Info:          USRMCLK:       0/     1      0%
20 Info:          CLKDIVF:       0/     4      0%
21 Info:          ECLKSYNCB:     0/    10      0%
22 Info:          DLLDELD:      0/     8      0%
23 Info:          DDRDLL:       0/     4      0%
24 Info:          DQSBUFM:       0/    14      0%
25 Info:          TRELLIS_ECLKBUF: 0/     8      0%
26 Info:          ECLKBRIDGECS: 0/     2      0%

```

Listing A.7: NextPnr-ECP5 Logging: 64 Bit Ressourcen

```

1 Info: Routing..
2 Info: Setting up routing queue.
3 Info: Routing 19218 arcs.
4 Info:          | (re-)routed arcs | delta | remaining| time spent |
5 Info:  IterCnt | w/ripup  wo/ripup | w/r  wo/r | arcs| batch(sec) total(sec)|
6 Info:    1000 |    266    733 | 266  733 | 18549|    0.82    0.82|
7 Info:    2000 |    473   1526 | 207  793 | 17830|    0.70    1.52|
8 .
9 .
10 .

12 Info:    57000 |   21957   35042 | 488  512 |   253|    3.22   115.21|
13 Info:    57620 |   22156   35464 | 199  422 |     0|    1.69   116.90|
14 Info: Routing complete.
15 Info: Router1 time 116.90s

```

Listing A.8: NextPnr-ECP5 Logging: 64 Bit Routingzeit

A. Anhänge

```

1 Info: Device utilisation:
2 Info:          TRELLIS_SLICE: 10036/41820    23%
3 Info:          TRELLIS_I0:      4/   365    1%
4 Info:          DCCA:           1/    56    1%
5 Info:          DP16KD:          0/   208    0%
6 Info:          MULT18X18D:      35/   156   22%
7 Info:          ALU54B:           0/    78    0%
8 Info:          EHXPLL:           0/     4    0%
9 Info:          EXTREFB:          0/     2    0%
10 Info:          DCUA:            0/     2    0%
11 Info:          PCSCLKDIV:        0/     2    0%
12 Info:          IOLOGIC:         0/   224    0%
13 Info:          SIOLOGIC:         0/   141    0%
14 Info:          GSR:              0/     1    0%
15 Info:          JTAGG:            0/     1    0%
16 Info:          OSG:              0/     1    0%
17 Info:          SEDGA:            0/     1    0%
18 Info:          DTR:              0/     1    0%
19 Info:          USRMCLK:          0/     1    0%
20 Info:          CLKDIVF:          0/     4    0%
21 Info:          ECLKSYNCB:        0/    10    0%
22 Info:          DLLDELD:          0/     8    0%
23 Info:          DDRDLL:           0/     4    0%
24 Info:          DQSBUFM:          0/    14    0%
25 Info:          TRELLIS_ECLKBUF:  0/     8    0%
26 Info:          ECLKBRIDGECS:    0/     2    0%

```

Listing A.9: NextPnr-ECP5 Logging: 128 Bit Ressourcen

```

1 Info: Routing..
2 Info: Setting up routing queue.
3 Info: Routing 51819 arcs.
4 Info:          | (re-)routed arcs |   delta   | remaining|      time spent      |
5 Info:  IterCnt | w/ripup  wo/ripup | w/r  wo/r |   arcs| batch(sec) total(sec)|
6 Info:    1000 |    284    715 | 284  715 |  51384|    0.79    0.79|
7 Info:    2000 |    524   1475 | 240  760 |  50907|    0.62    1.41|
8 .
9 .
10 .
11 Info:    760000 |  344481   415518 |  440   560 |    256|   459.28  562916.56|
12 Info:    760498 |  344627   415871 |  146   353 |     0|   13.23  562929.75|
13 Info: Routing complete.
14 Info: Router1 time 562929.75s

```

Listing A.10: NextPnr-ECP5 Logging: 128 Bit Routingzeit

Abbildungsverzeichnis

2.1. Menschliche Mustererkennung [KH]	11
2.2. Entropiequelle, Poolmixer und Entropiepool	15
2.3. Entropie eines manipulierbaren Münzwurfs [KH]	17
2.4. Schema des Zufallszahlengenerators <code>/dev/(u)random</code>	23
3.1. Dezentrales und zentrales Netzwerk	27
4.1. Vereinfachte Darstellung vom Workflow und den Toolchains für FPGA	36
4.2. Sequenzdiagramm für Ansteuerung und Beschleuniger	39
4.3. Aktueller Stand der Symbiflow Tools	40
4.4. Verbindungsschema der Komponenten	42
4.5. SBT Projektstruktur als einfachstes Beispiel	52
4.6. Projekt UartVdf: Modulübersicht	53
4.7. UART Zustandsautomat für die eingehenden Daten (FsmIn)	57
4.8. UART Zustandsautomat für die ausgehenden Daten (FsmOut)	57
4.9. GTKWave: Beispiel Simulation mit Zustandsautomat aus MultMod	62
4.10. Demonstrator: Foto nach erfolgreicher Berechnung	64
5.1. SpinalHDL Projekte	65

Tabellenverzeichnis

4.1. Komponenten: Übersicht	41
4.2. Lattice ECP5-85k FPGA: Eigenschaften [Sem]	41
4.3. Kendryte K210 RISC-V: Eigenschaften [CAN]	41
4.4. UART Payload der Eingangsparameter	56
4.5. UART Payload des Ausgabeergebnisses	56
6.1. Projektbestandteile: Open Source	68
6.2. FPGA Vergleich: Amazon AWS F1 und Lattice ECP5-85k	71
6.3. Geschwindigkeitsvergleich 64 Bit: FPGA, RISC-V und Intel i5	72
6.4. Geschwindigkeitsvergleich 128 Bit: FPGA, RISC-V und Intel i5	72

Listings

4.1. Karatsuba Pseudocode	46
4.2. Beispieldatei: Ulx3s.js für Verilog Topmodul Ulx3s.v	49
4.3. SpinalHDL Beispiel 1	50
4.4. SpinalHDL Instanziierung der Klasse Adder	50
4.5. Scala-Build-Tool: Beispieldatei build.sbt	51
4.6. UartVdf.scala: Verbindungen	54
4.7. UartFsmInOut.scala: Uart und Flow Definitionen	54
4.8. MultMod.scala: Flow Definitionen	54
4.9. ClkDiv.scala: Taktteiler in SpinalHDL	55
A.1. Makefile der FPGA Toolchain	77
A.2. SpinalHDL Beispiel: Blinky.scala	78
A.3. UartFsmInOut.scala	80
A.4. Karatsuba Multiplikation als SpinalHDL Klasse	82
A.5. MultMod.scala	83
A.6. Mikropython auf dem RISC-V	87
A.7. NextPnr-ECP5 Logging: 64 Bit Ressourcen	89
A.8. NextPnr-ECP5 Logging: 64 Bit Routingzeit	89
A.9. NextPnr-ECP5 Logging: 128 Bit Ressourcen	90
A.10. NextPnr-ECP5 Logging: 128 Bit Routingzeit	90

Quellen

- [ADR82] Alain Aspect, Jean Dalibard, and Gérard Roger. “Experimental Test of Bell’s Inequalities Using Time-Varying Analyzers”. In: *Phys. Rev. Lett.* 49 (25 Dec. 1982), pp. 1804–1807. DOI: 10.1103/PhysRevLett.49.1804.
- [All19] VDF Alliance. *Building open source hardware to improve the security and scalability of blockchain protocols*. Website. <https://www.vdfalliance.org/>; abgerufen am 02. Mai 2020. 2019.
- [BBS83] Lenore Blum, Manuel Blum, and Michael Shub. “Comparison of Two Pseudo-Random Number Generators”. In: *Advances in Cryptology*. Ed. by David Chaum, Ronald L. Rivest, and Alan T. Sherman. Boston, MA: Springer US, 1983, pp. 61–78. ISBN: 978-1-4757-0602-4.
- [Ber08a] Daniel J. Bernstein. “ChaCha, a variant of Salsa20”. In: (Jan. 2008).
- [Ber08b] Daniel J. Bernstein. “New Stream Cipher Designs”. In: ed. by Matthew Robshaw and Olivier Billet. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. The Salsa20 Family of Stream Ciphers, pp. 84–97. ISBN: 978-3-540-68350-6. DOI: 10.1007/978-3-540-68351-3_8.
- [Ber14] Daniel J. Bernstein. *Entropy Attacks!* Website. <http://blog.cr.yp.to/20140205-entropy.html>; abgerufen am 10. Mai 2020. 2014.
- [Blu83] Manuel Blum. “Coin Flipping by Telephone a Protocol for Solving Impossible Problems”. In: *SIGACT News* 15.1 (Jan. 1983), pp. 23–27. DOI: 10.1145/1008908.1008911. URL: <https://doi.org/10.1145/1008908.1008911>.
- [Bon+18] Dan Boneh et al. *Verifiable Delay Functions*. Cryptology ePrint Archive, Report 2018/601. <https://eprint.iacr.org/2018/601>; abgerufen am 18.05.2020. 2018.
- [Bro20] Robert G. Brown. *Dieharder: A Random Number Test Suite*. Website. <https://webhome.phy.duke.edu/~rgb/General/dieharder.php>; abgerufen am 04. Mai 2020. 2020.

QUELLEN

- [CAN] LTD CANAAN CREATIVE CO. *Kendryte K210 Developer Area*. <https://kendryte.com/downloads/>. Abgerufen am 13.05.2020.
- [DWA20] D’WAVE. *D’WAVE Leap*. Website. <https://www.dwavesys.com/take-leap>; abgerufen am 10. Mai 2020. 2020.
- [EBB72] A. Einstein, M. Born, and Hedwig Born. *Briefwechsel 1916-1955*. rororo Taschenbücher. Rowohlt, 1972. ISBN: 9783499114786.
- [Fou] RISC-V Foundation. *RISC-V SoftCPU Contest Highlights*. <https://riscv.org/2018/12/risc-v-softcpu-contest-highlights/>. Abgerufen am 13.05.2020.
- [Fou19] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. Ed. by Andrew Waterman and Krste Asanovi. <https://riscv.org/specifications/isa-spec-pdf/>; abgerufen am 12. Mai 2020. Dec. 2019.
- [HD62] T. E. Hull and A. R. Dobell. “Random Number Generators”. In: *SIAM Review* 4.3 (1962), pp. 230–254. DOI: 10.1137/1004061.
- [Hec20] Harald Heckmann. *QPU - A library for quantum processing units (IBM, D-Wave)*. Website. <https://github.com/sea212/qpu>; abgerufen am 10. Mai 2020. 2020.
- [Hom15] M. Homeister. *Quantum Computing verstehen: Grundlagen - Anwendungen - Perspektiven*. Computational Intelligence. Springer Fachmedien Wiesbaden, 2015. ISBN: 9783658104559.
- [Hüh20] Thomas Hühn. *Myths about /dev/urandom*. Website. <https://www.2uo.de/myths-about-urandom/>; abgerufen am 10. Mai 2020. 2020.
- [IBM20] IBM. *IBM Quantum Experience*. Website. <https://quantum-computing.ibm.com/>; abgerufen am 10. Mai 2020. 2020.
- [IS01] National Institute and Technology of Standards. “Advanced Encryption Standard”. In: *NIST FIPS PUB 197* (2001).
- [KH] Thorsten Knoll and Harald Heckmann. “Blockchain-Technologien für Informatiker”. Nicht veröffentlicht.
- [Kno16] Thorsten Knoll. “Bachelorthesis: Entwurf und Entwicklung einer Vektorgrafikeinheit”. <https://github.com/ThorKn/Vectorgraphicsunit/blob/master/thesis/thesis.pdf>; abgerufen am 12. Mai 2020. Hochschule RheinMain, Sept. 2016.
- [KO63] A. Karatsuba and Yuri Petrovich Ofman. “Multiplication of Many-Digital Numbers by Automatic Computers”. In: 1963.

- [Mag13] Wired Magazine. *RSA Tells Its Developer Customers: Stop Using NSA-Linked Algorithm*. Website. <https://www.wired.com/2013/09/rsa-advisory-nsa-algorithm/>; abgerufen am 07. Mai 2020. 2013.
- [Mau92] Ueli M. Maurer. “A Universal Statistical Test for Random Bit Generators”. In: *J. Cryptol.* 5.2 (Mar. 1992), pp. 89–105. ISSN: 0933-2790.
- [MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. CRC Press, Inc., 1996. ISBN: 0849385237.
- [Nak09] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. <http://www.bitcoin.org/bitcoin.pdf>; abgerufen am 18. Mai 2020. 2009.
- [Pie18] Krzysztof Pietrzak. *Simple Verifiable Delay Functions*. Cryptology ePrint Archive, Report 2018/627. <https://eprint.iacr.org/2018/627/>; abgerufen am 18.05.2020. 2018.
- [Plu82] J. B. Plumstead. “Inferring a sequence generated by a linear congruence”. In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)(FOCS)*. Vol. 00. Nov. 1982, pp. 153–159. DOI: 10.1109/SFCS.1982.73.
- [PP14] Thomas Ptacek and Erin Ptacek. *How To Safely Generate A Random Number*. Website. <https://sockpuppet.org/blog/2014/02/25/safely-generate-random-numbers/>; abgerufen am 10. Mai 2020. 2014.
- [Qua20] ID Quantique. *Quantis Random Number Generator*. Website. <https://www.idquantique.com/random-number-generation/products/quantis-random-number-generator/>; abgerufen am 10. Mai 2020. 2020.
- [Rab83] Michael O. Rabin. “Transaction protection by beacons”. In: *Journal of Computer and System Sciences* 27.2 (1983), pp. 256–267. DOI: [https://doi.org/10.1016/0022-0000\(83\)90042-9](https://doi.org/10.1016/0022-0000(83)90042-9). URL: <http://www.sciencedirect.com/science/article/pii/0022000083900429>.
- [Ree77] James Reeds. “Cracking a random number generator”. In: *Cryptologia* 1.1 (1977), pp. 20–26. DOI: 10.1080/0161-117791832760.
- [Rei20] Steffen Reith. *Lehrveranstaltungen in der Theoretischen Informatik an der Hochschule RheinMain*. Website. <https://www.cs.hs-rm.de/~reith/lehre/>; abgerufen am 12. Mai 2020. 2020.
- [Rhe20] Hochschule RheinMain. *Studienverlauf Informatik - Technische Systeme*. Website. <https://www.hs-rm.de/de/fachbereiche/design-informatik-medien/studiengaenge/informatik-technische-systeme-bsc/studienverlauf-po-2016/>; abgerufen am 12. Mai 2020. 2020.

QUELLEN

- [Sch+18] Philipp Schindler et al. “HydRand: Practical Continuous Distributed Randomness”. In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 319.
- [Sch01] C. Schiestl. *Pseudozufallszahlen in der Kryptographie*. Diplom.de, 2001. ISBN: 9783838641492.
- [Sem] Lattice Semiconductors. *FPGA ECP5 Documentation*. <https://www.latticesemi.com/Products/FPGAandCPLD/ECP5>. Abgerufen am 13.05.2020.
- [Sha48] C. E. Shannon. “A mathematical theory of communication”. In: *The Bell System Technical Journal* 27.3 (July 1948), pp. 379–423. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb01338.x.
- [Sic20] Bundesamt für Sicherheit in der Informationstechnik. *Documentation and Analysis of the Linux Random Number Generator*. Website. https://www.bsi.bund.de/DE/Publikationen/Studien/LinuxRNG/index_hm.html; abgerufen am 10. Mai 2020. 2020.
- [Ski19] Alex Skidanov. *near Protocol Randomness Beacon*. Website. <https://near.org/downloads/NearRandomnessBeacon.pdf>; abgerufen am 30. April 2020. 2019.
- [ST10] NIST - National Institute of Standards and Technology. *SP800-22 A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Website. <https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final>; abgerufen am 04. Mai 2020. 2010.
- [ST20] NIST - National Institute of Standards and Technology. *Random Bit Generation*. Website. <https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software/guide-to-the-statistical-tests>; abgerufen am 04. Mai 2020. 2020.
- [Tea19] Randao Project Team. *RANDAO: A DAO working as RNG of Ethereum*. Website. <https://github.com/randao/randao>; abgerufen am 02. Mai 2020. 2019.
- [Val15] Filippo Valsordas. *The plain simple reality of entropy*. Chaos Communication Congress 2015 - 32C3. https://media.ccc.de/v/32c3-7441-the_plain_simple_reality_of_entropy; abgerufen am 10. Mai 2020. 2015.
- [Wes18] Benjamin Wesolowski. *Efficient verifiable delay functions*. Cryptology ePrint Archive, Report 2018/623. <https://eprint.iacr.org/2018/623>; abgerufen am 18.05.2020. 2018.
- [WL] Clifford Wolf and Mathias Lasser. *Project IceStorm*. <http://www.clifford.at/icestorm/>. Abgerufen am 13.05.2020.