Fachhochschule Dortmund
University of Applied Sciences and Arts

Cologne Chip

FPGA-Implementierung einer Ringoszillator basierten Physically Unclonable Function

Bachelorarbeit

im Studiengang „Elektrotechnik Automation & Antriebe“

vorgelegt von

Mehdi Mneja

am 10.10.2022

an der Fachhochschule Dortmund

Erstprüfer: Herr Prof. Dr. Karagounis

Zweitprüfer: Herr Felix Schneider

Thema der Arbeit

FPGA-Implementierung einer Ringoszillator basierten Physically Unclonable Function

Abstract

Physical Unclonable Functions (PUFs) sind Schaltkreisprimitive, die abhängig von den unkontrollierbaren Schwankungen im Herstellungsprozess chip-spezifische und einzigartige Ausgaben erzeugen. Diese kostengünstigen und hocheffizienten Strukturen haben eine breite Palette von Anwendungsbereichen einschließlich Authentifizierung, Schlüsselgenerierung und IP-Schutz. In dieser Arbeit geht es um die FPGA-Implementierung einer Ringoszillator basierten Physically Unclonable Function, die mit dem Yosys-Framework auf einem Gatemate FPGA der Firma Cologne Chip implementiert werden soll.

Title of the paper

FPGA implementaion of ring-oscillator based Physically Unclonable Function

Abstract

Physical Unclonable Functions (PUFs) are circuit primitives that generate chip-specific and unique outputs depending on the uncontrollable variations in the manufacturing process. These low-cost and highly efficient structures have a wide range of applications including authentication, key generation, and IP protection. This work focuses on the FPGA implementation of a ring oscillator based Physically Unclonable Function, to be implemented with the Yosys framework in an Gatemate FPGA from the company Cologne Chip.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AC: alternating current

CPE: combining programmable element

CRP: Challenge-Response Pair

DC: direct current

EDA: Electronic Design Automation

FH: Fachhochschule Dortmund

FPGA: Field Programmable Gate Array

FTDI: Future Technology Devices International

GPIO: General Purpose Input/Output

HD: Hamming-Distanz

PCB: printed circuit board

PMOD: Peripheral Modul

PUF: Physically Unclonable Function

RO: Ringoszillator

SLP: Super Low Power

SPI: Serial Peripheral Interfac

USB: Universal Serial Bus

UART: Universal Asynchronous Receiver Transmitter

VHDL: Very High Hardware Description Language

1. Einleitung

Diese Arbeit konzentriert sich auf die FPGA-Implementierung einer Ringoszillator basierten Physically Unclonable Function. Der Ringoszillator ist ein elektronischer Oszillator, der aus einer ungeraden Anzahl von NOT-Gattern in einer Ring Anordnung besteht, deren Ausgang zwischen zwei Spannungspegeln schwingt, die wahr und falsch repräsentieren. Die NOT-Gatter bzw. Inverter sind zu einer Kette verbunden, wobei der Ausgang des letzten Inverters in der Kette den Eingang des ersten speist.

In dieser Arbeit wird zunächst die Definition einer PUF geklärt und das Funktionsprinzip der PUF auf Basis des Ringoszillators vorgestellt. Anschließend wird die verwendete Hardware beschrieben, insbesondere der FPGA und die erforderlichen Zusatzmodule. In diesem Projekt wird die Open-Source-Synthese-Software Yosys in Kombination mit dem proprietären Place & Route Software des Anbieters Cologne Chip verwendet, um digitale Schaltungen auf dem GateMate FPGA von Cologne-Chip zu implementieren. Anschließend wird das entworfene VHDL-Schaltungsdesign vorgestellt, das darauf abzielt, Responses zu generieren und über eine UART-Schnittstelle zu übertragen. Schließlich werden die Synthese, die Implementation und die Simulation der Ringoszillator PUF vorgestellt.

2. Aufgabenbeschreibung

Das Ziel ist die Implementierung einer auf Ringoszillatoren basierenden Physically Unclonable-Function in einem FPGA. Hierbei soll der GateMate FPGA von Chip Cologne verwendet und auf einem Evaluierungsboard in Betrieb genommen werden. Durch einen Vergleich der Schwingfrequenz der Ringoszillatoren wird eine Response erstellt und diese Response Bit für Bit über eine UART-Schnittstelle ausgegeben. Für die Kopplung an den PC wird ein UART zu USB-Baustein von FTDI verwendet, der auf einer kleinen "USBUART"-Platine integriert ist und über einen PMOD-Anschluss mit dem FPGA verbunden werden kann. Ein zweiter PMOD-Anschluss wird verwendet, um eine Platine mit Drucktasten anzuschließen und die Bedienung des im FPGA implementierten Schaltungsentwurfs zu ermöglichen.

Die Hardwaremodule werden in VHDL entworfen, simuliert und auf dem FPGA mithilfe von Programmen wie Yosys, Ghdl und openFPGALoader unter Ubuntu (Linux) implementiert.

3. PUF

Die Sicherheit von integrierten Schaltkreisen (ICs) ist aufgrund von hohen Anforderungen an die Informationssicherheit zu einem wichtigen Thema geworden. Getrieben werden diese Anforderungen durch Vorgaben in Bezug auf den Schutz personenbezogener Daten und den Schutz geistigen Eigentums. Ein wichtiger Aspekt bei der Verbesserung des Vertrauens in Halbleiterbauelemente und die Halbleiterlieferkette ist die Erhöhung der physischen Sicherheit. Physically Unclonable Function (PUF) [1, 2] ermöglichen es, die physikalische Sicherheit von integrierten Schaltkreisen (ICs) gegen Hacker und unberechtigten Zugriff zu erhöhen.

PUFs nutzen die inhärenten Eigenschaften von Drähten und Transistoren aus, die aufgrund von Variationen im Herstellungsprozess von Chip zu Chip unterschiedlich sind [3]. Diese komplexen physikalischen Eigenschaften von integrierten Schaltkreisen werden verwendet, um einzigartige Signaturen zu erzeugen, die zufällig, unvorhersehbar und schwer zu reproduzieren sind. Ein PUF generiert eine Reihe von Responses, wenn er durch eine Reihe von Challenges stimuliert wird. Die Challenge-Response-Beziehung wird durch komplexe physikalische Eigenschaften des Materials definiert, wie z. B. der Prozessvariabilität von Halbleiterbauelementen.

PUFs erhöhen die physische Sicherheit, durch die Erzeugung flüchtiger Geheimnisse in digitaler Form während des Betriebs des Chips. Geheime Schlüssel sind für viele sicherheitsrelevante Anwendungen unerlässlich. Die Speicherung von Geheimnissen in einem nichtflüchtigen Speicher ist nicht nur kostspielig, sondern kann auch ein leichtes Ziel für invasive Angriffe darstellen [4]. Eine PUF bietet einen kostengünstigen und sicheren Ansatz für die Generierung geheimer Schlüssel. Für jede Challenge oder Eingabe, erzeugt eine PUF eine eindeutige Response oder Ausgabe. Diese Eigenschaft der PUF wird genutzt, um verschiedene Sicherheitsprobleme zu lösen, wie z. B. die Authentifizierung von Chips, die Generierung kryptografischer Schlüssel, Softwarelizenzen, den Schutz des geistigen Eigentums (IP) sowie die Erkennung und Verhinderung von gefälschten integrierten Schaltkreisen.

3.1 Qualität einer PUF

Die Zuverlässigkeit einer PUF wird durch Messungen zur Bewertung der grundlegenden PUF-Funktionen definiert. Die Einzigartigkeit, Zuverlässigkeit und Belastbarkeit der PUF-Responses stellen dabei maßgebliche Qualitätsfaktoren einer PUF dar [3, 5].

3.1.1 Einzigartigkeit

Die Einzigartigkeit ist die Schätzung der Eindeutigkeit, mit der durch eine PUF verschiedene Chips auf der Grundlage der generierten Response unterschieden werden können. Der Einzigartigkeitsfaktor ist ein Maß für die Inter-Chip-Variation, die Auskunft darüber gibt, wie viele der Ausgangsbits der PUF zwischen zwei verschiedenen PUFs unterschiedlich sind. Die Einzigartigkeit einer PUF wird durch die durchschnittliche Inter-Hamming-Distanz (HD) über eine Gruppe von Chips ermittelt.

Die Einzigartigkeit ist ein Maß für die Erwartung an die Hammingdistanz zwischen den Responses zweier Instanzen auf dieselbe Challenge. Sie kann aus dem Verlauf einer Probability Mass Function (PMF) oder Probability Density Function (PDF) der Hammingdistanzen zwischen den Antworten mehrerer Instanzen auf die gleiche Challenge ermittelt werden. Diese Verteilungen sind bei einer idealen PUF um die Hammingdistanz der halben Response-Breite zentriert. Bei binären Datenwörtern ist die Hamming-Distanz zwischen zwei Datenwörtern gleicher Länge die Anzahl der Bits, die sich in den beiden Datenwörtern unterscheiden.

Sei (i, j) ein Paar Chips mit i ≠ j in denen jeweils eine PUF integriert ist und Ri bzw. Rj sei die n-Bit-Response der PUF im Chip i bzw. im Chip j. Die erste Metrik ist die durchschnittliche Inter-Chip-Hamming-Distanz unter einer Gruppe von k Chips und wird wie folgt definiert [5]:

$$D_{INTER} = \frac{2}{k(k-1)} \sum_{i=j}^{k-1} \sum_{j=i+1}^{k} \frac{HD(Ri, Rj)}{n} \times 100\%$$

Wenn die PUF unabhängige, gleichmäßig verteilte Zufallsbits erzeugt, d.h. wenn jedes binäre Response-Bit einer PUF mit gleicher Wahrscheinlichkeit eine '0' oder eine '1' annehmen kann, dann sollten die Inter-Chip-Variationen im Durchschnitt 50% betragen. Echte Zufallsbits werden erzeugt, wenn nur die Variation des Zufallsprozesses und keine systematischen Einflüsse existieren.

3.1.2 Zuverlässigkeit

Die Zuverlässigkeit bewertet die Reproduzierbarkeit der PUF-Response und gibt an, wie viele der PUF-Ausgangsbits verändert werden, wenn die Bits derselben PUF mit oder ohne Umgebungsvariationen neu generiert werden. Die Responses einer idealen PUF sollten konsistent sein. Faktoren wie Temperaturschwankungen, Schwankungen der Versorgungsspannung und Fehler aufgrund von thermischem Rauschen beeinträchtigen jedoch die Reproduzierbarkeit der PUF- Response. Dementsprechend ist die Zuverlässigkeit ein Maß für die Konsistenz oder Stabilität der Ausgangs Response der PUF, wenn die PUF variablen Umgebungsbedingungen wie Schwankungen der Versorgungsspannung und der Temperatur sowie der gleichen Challenge ausgesetzt ist. Da die verglichenen Responses von selbem Chip erzeugt werden, wird diese Variation auch als Intra-Chip- oder Intra-Die-Variation bezeichnet. Eine n-Bit-Referenzantwort (Ri) wird unter normalen Betriebsbedingungen aus dem Chip i extrahiert. Die gleiche n-Bit-Response wird unter einer anderen Betriebsbedingung mit den Response bits R'i aus demselben PUF extrahiert. Sei R'i, y die y^{ste}

Stichprobe von R'i. Dann wird die durchschnittliche Intra-Die HD über x Abtastungen für den Chip i wie folgt definiert [5]:

$$D_{INTRA} = \frac{1}{x}\sum_{y=1}^{x}\frac{HD(Ri, R'i, y)}{n} \times 100\%$$

Ein niedrigerer Wert für den durchschnittlichen Intra-Chip-HD-Faktor führt zu einere zuverlässigeren PUF- Response. Die Intra-Chip-Schwankungen für eine ideale PUF sollten also 0% betragen.

3.1.3 Resilienz

Die Resilienz einer PUF ist die Fähigkeit der PUF, einen Gegner daran zu hindern, die Geheimnisse der PUF zu enthüllen. die ein Maß für die Widerstandsfähigkeit gegen Angriffe oder für die Sicherheit ist.

3.2 Klassifizierungen von PUFs

PUFs können nach ihren Konstruktionseigenschaften, ihrem Funktionsprinzip und unter Sicherheitsaspekten klassifiziert werden. Tabelle 1 fasst die verschiedenen PUFs in verschiedenen Kategorien zusammen.

Kategorien	Beispiele
Nicht-elektronische PUF	Optische PUF, Akustische PUF
Elektronische PUF	Beschichtungs-PUF, Stromverteilungs-PUF
Verzögerungsbasierte PUF	Arbiter PUF, Ring Oszillator PUF, Glitch PUF, Anderson-PUF
Speicherbasierte PUF	SRAM PUF, Butterfly PUF, Flip-Flop PUF

Tabelle 1: Verschiedene Arten von PUFs

3.2.1 Nicht-elektronische PUFs, elektronische PUFs und PUFs aus Silizium

Auf der Grundlage ihrer Konstruktions- und Funktionsprinzipien lassen sich PUFs in drei Kategorien einteilen: nicht-elektronische PUFs, elektronische PUFs und Silizium-PUFs [6]. Nicht-elektronische PUFs beziehen sich auf solche, die PUF-ähnliche Eigenschaften haben und deren Aufbau und/oder Funktionsweise von Natur aus nicht-elektronisch sind.

Ihr PUF-ähnliches Verhalten basiert auf nicht-elektronische Technologien oder Materialien, wie etwa der zufälligen Struktur der Fasern eines Blattes Papier oder der zufälligen Reflexion der Streueigenschaften eines optischen Mediums.

Zum Beispiel sind optische PUFs, die auf transparenten Medien basieren, wie in [7] vorgeschlagen, physikalische Einwegfunktionen. Abbildung 1 zeigt die grundlegende Implementierung einer optischen PUF. Die PUF Challenge-Response, bestehend aus der Ausrichtung des Lasers und dem resultierenden Hash, wird zur späteren Verwendung in einer öffentlichen Datenbank gespeichert.

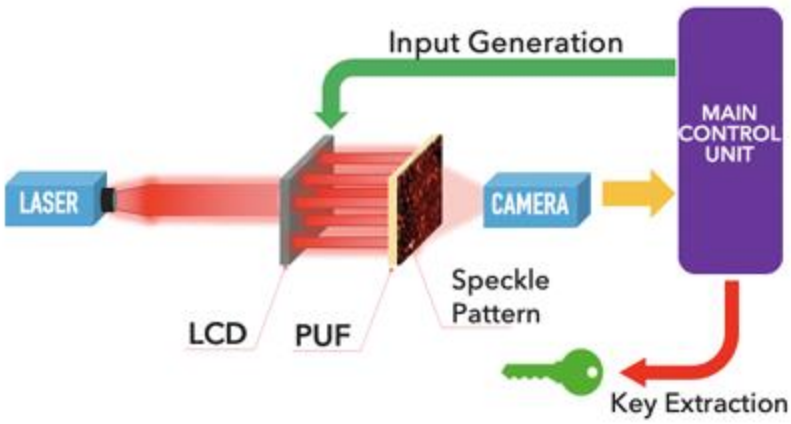


Abbildung 1: Optischer PUF [15]

Bei elektronischen PUFs besteht die Grundoperation in der analogen Messung einer elektrischen oder elektronischen Größe wie Leistung, Widerstand und Kapazität. Ein Beispiel für eine elektronische PUF ist die Beschichtungs-PUF [8], bei der die Zufälligkeit von Kapazitätsmessungen in kammförmigen Sensoren in der oberen Metallschicht eines ICs berücksichtigt wird.

Silizium-PUFs [1] zeigen das Verhalten von PUFs und sind in einem Siliziumchip integriert. Sie basieren auf verborgenen Zeit- und Verzögerungsinformationen von integrierten Schaltkreisen. Ein komplexer integrierter Schaltkreis kann als Silizium-PUF verwendet werden, um einzelne integrierte Schaltkreise zu identifizieren und zu authentifizieren. Silizium-PUFs können auch als Hardware-Baustein in kryptografische Systeme integriert werden. Silizium-PUFs nutzen Variationen im Herstellungsprozess von integrierten Schaltkreisen mit identischen Masken aus, um jeden integrierten Schaltkreis eindeutig zu charakterisieren. Silizium-PUFs sind für Sicherheitslösungen von besonderem Interesse und werden als eine wichtige Art von PUFs umfassend untersucht. Gerade Verzögerungs- und Speicher-PUFs werden auf Grund ihres einfachen Aufbaus als besonderes aussichtsreiche Silizium-PUFs betrachtet.

3.2.2 Starker PUF und schwacher PUF

Die Unterscheidung zwischen starken und schwachen PUFs wird anhand der Sicherheitseigenschaften ihres Challenge-Response-Verhaltens erläutert [9]. Eine PUF gilt als starke PUF, wenn die PUF so viele CRPs besitzt, dass ein Angriff, der auf einer umfassenden Messung der CRPs beruht, nur eine vernachlässigbare Erfolgswahrscheinlichkeit hat. Bei einer starken PUF ist es unmöglich, auf der Grundlage der beobachteten CRPs ein genaues Modell der PUF zu erstellen. Ist die Anzahl der CRPs gering, wird von einer schwachen PUF ausgegangen.

3.2.3 Intrinsische PUF und nicht-intrinsische PUF

Eine weitere Klassifizierung, die auf den Konstruktionseigenschaften der PUFs beruht, ist die Unterscheidung zwischen intrinsischen und nicht-intrinsischen PUFs. Intrinsische PUFs wurden ursprünglich von Guajardo et al. in [10] vorgeschlagen. Bei intrinsischen PUFs werden die Bewertungen intern durch integrierte Messgeräte durchgeführt und die spezifischen Eigenschaften der Zufallsinstanz werden implizit während des Herstellungsprozesses eingeführt. Alle Silizium-PUFs, die auf zufälligen Variationen im Herstellungsprozess von Siliziumchips beruhen, sind intrinsische PUFs. Zu diesen Silizium-PUFs gehören insbesondere verzögerungsbasierte und speicherbasierte PUFs. Nicht-

intrinsische PUFs werden extern ausgewertet und ihre zufälligen Eigenschaften werden explizit dargestellt. Optische PUFs und beschichtete PUFs sind Beispiel für nicht-intrinsische PUFs.

3.3 Ringoszillator PUF Schaltung

Da PUFs in den letzten Jahren als eines der potenziellen Innovationsthemen im Bereich der Hardwaresicherheit und Kryptografie viel Aufmerksamkeit auf sich gezogen haben, wurden verschiedene PUF-Techniken für die On-Chip-Implementierungen als anwendungsspezifischen integrierten Schaltkreis (ASICs) oder in FPGAs vorgeschlagen [2].

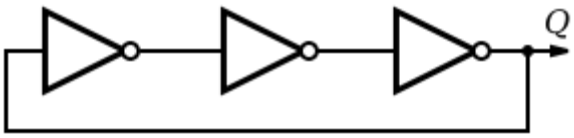


Abbildung 2: Schaltplan eines einfachen Ringoszillators mit 3 Wechselrichtern

Die Ringoszillator (RO) PUF besteht aus mehreren identisch aufgebauten Verzögerungsschleifen bzw. Ringoszillatoren, von denen jeder aufgrund von Variationen im Herstellungsprozess mit einer eigenen Frequenz schwingt [11]. Challenges wählen ein Oszillatorpaar zum Vergleich aus, um ein Response-Bit zu erzeugen. Die PUF erhält eine Reihe von Challenges und wählt dadurch eine Folge von Oszillatorpaaren aus, um eine Anzahl von Response Bits zu erzeugen.

Die Frequenzunterschiede werden durch Prozessvariationen bestimmt, wenn alle Oszillatoren identisch angeordnet sind, was dazu führt, dass bei einer zufälligen Variation die gleiche Wahrscheinlichkeit besteht, eine "1" oder "0" als Response-Bit zu erhalten. Die einfache Duplizierung eines Ringoszillators mit Hilfe von Hard-Makros hat seine Implementierung in FPGAs populärer gemacht. In [12] wurde ein konfigurierbarer Ringoszillator vorgeschlagen, um die Zuverlässigkeit eines RO-PUFs zu verbessern.

Die Autoren haben gezeigt, dass RO-PUFs sorgfältige Entwurfsentscheidungen erfordern, um systematische Prozessvariationen zu vermeiden. Beispielsweise verbessern deterministische Platzierungstechniken und eine optimale Auswahl von Ringoszillatorpaare die Einzigartigkeit der PUFs erheblich.

4. Verwendete Hardware:

GateMate ist eine FPGA-Familie von Cologne Chip mit Komponenten kleiner bis mittlerer Dichte. Das Unternehmen adressiert eine Vielzahl von Anwendungen in den Bereichen Automatisierung, Kommunikation, Sicherheit, Automobil, IoT, Beleuchtung etc.

Die Cologne Chip AG wurde 1996 gegründet und hat ihren Sitz in Köln. Das GateMate FPGA-Programm wird vom deutschen Bundesministerium für Wirtschaft und Energie im Rahmen des Important Project of Common European Interest on Microelectronics der Europäischen Kommission gefördert.

Die GateMate-Familie des Unternehmens kombiniert Logikdichte, Energieverbrauch und Gehäusegröße mit den "niedrigsten Kosten auf dem Markt", wodurch sich die Bauteile sowohl für akademische Projekte als auch für Anwendungen mit hohem Volumen eignen. Die Chips wurden nicht nur in Deutschland entworfen, sondern werden auch in Deutschland hergestellt.

4.1 GateMate FPGA Evaluation Board

In diesem Projekt wird das GateMate FPGA mit dem dazugehörigen Evaluation Board verwendet, da es eine funktionsreiche und sofort einsatzbereite Entwicklungsplattform für das CCGM1A1 darstellt.

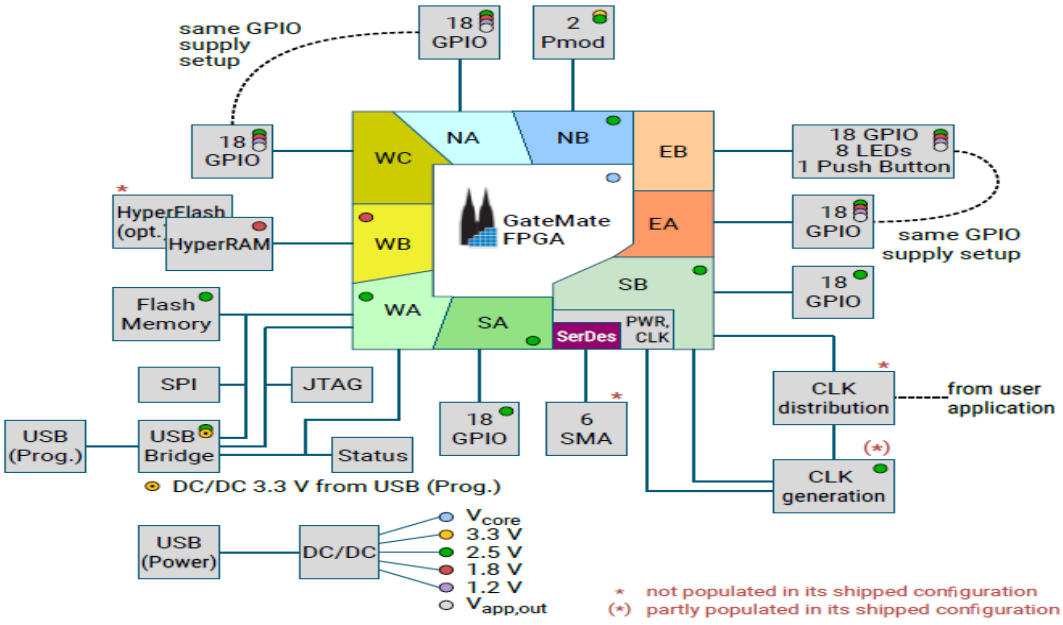


Abbildung 3: Überblick auf das GateMate FPGA Evaluation Board Version 3.1 [13]

Abbildung 3 ermöglicht einen Überblick auf die Eigenschaften des Evaluation Boards. Einige GPIO-Bänke haben je nach Funktion einen festen Spannungspegel. Andere können auf unterschiedliche Spannungen eingestellt werden.

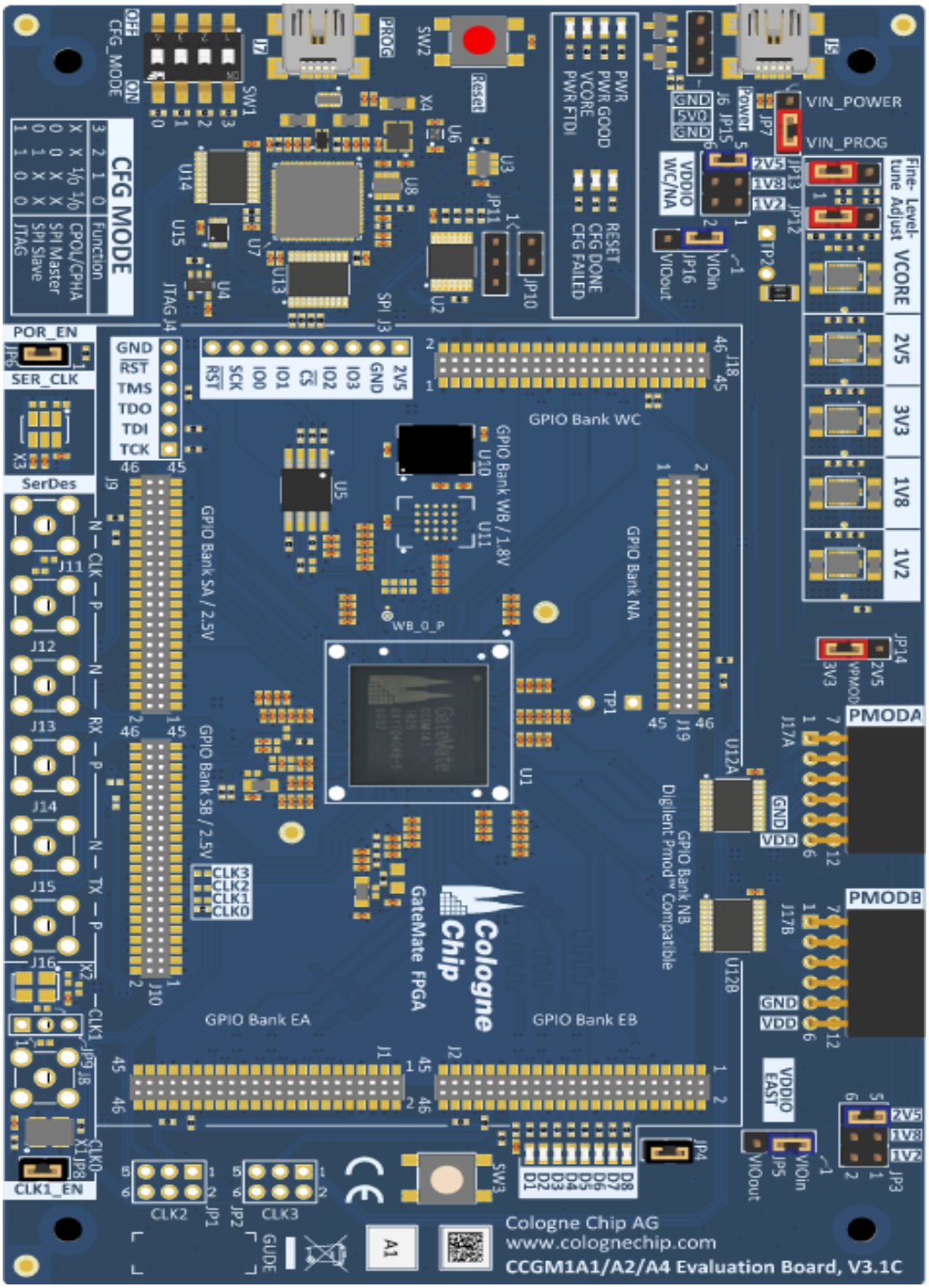


Abbildung 4: GateMate FPGA Evaluation Board Version 3.1 [14]

Abbildung 4 zeigt eine Draufsicht auf die Leiterplatte (PCB) des Evaluations-Bords. Diese Platine ist grundlegend bestückt und sieht Bauteile für erweiterte Funktionen vor.

4.1.1 Block Diagramm

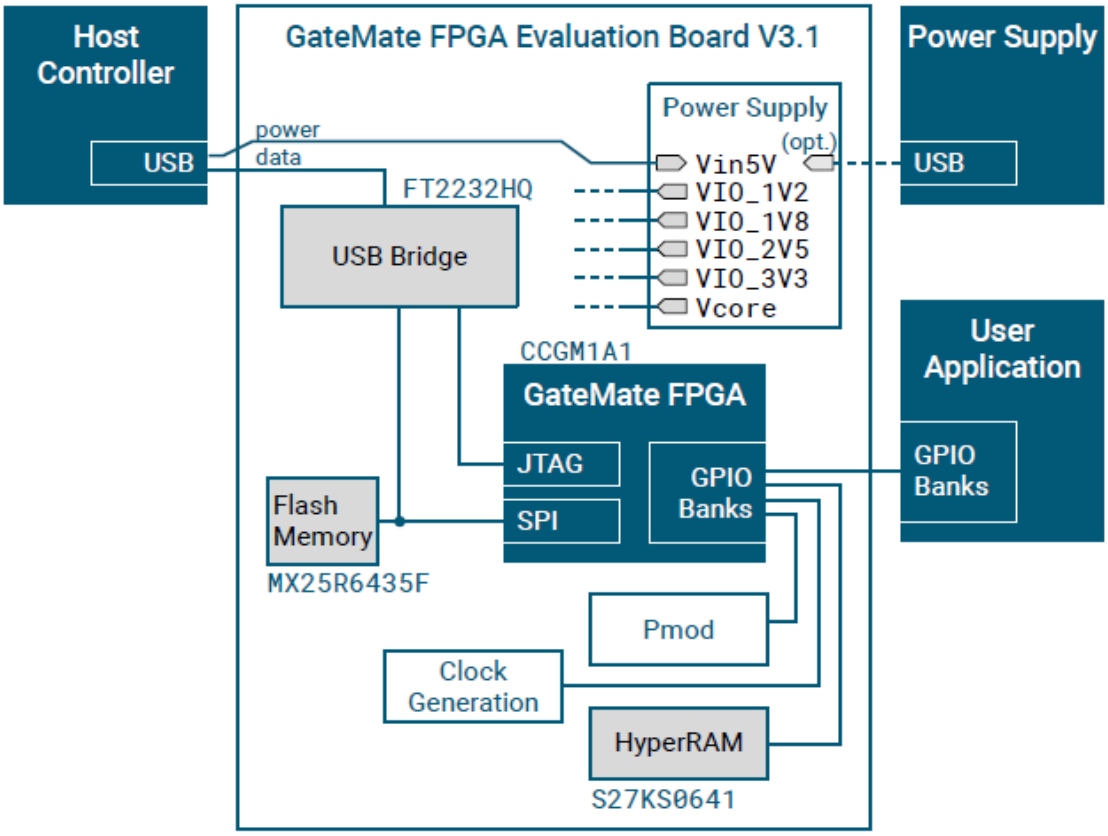


Abbildung 5: Block Diagramm des GateMate FPGA Evaluation Boards [16]

Abbildung 5 zeigt ein Blockdiagramm des Evaluation-Bords. Es wird eine einzige Versorgungsspannung von einem USB-Netzteil verwendet, um die Leiterplatte und die Benutzeranwendung zu versorgen. In diesem Projekt wird die FPGA-Konfiguration aus dem On-Board-Flash-Speicher eingelesen, so dass der FPGA nach einem Reset automatisch konfiguriert wird. Eine Benutzeranwendung kann über GPIO-Bänke (General Purpose Input/Output) mit dem GateMate FPGA verbunden werden. Außerdem existieren zusätzliche Signale für Reset und Clock um auch hohe Anforderungen der Anwendungen erfüllen zu können [15].

4.1.2 Evaluation Board Funktionen

Dieses Kapitel beschreibt die Funktionen des GateMate FPGA-Evaluierungsboards.

4.1.2.1 PCB Power Supply

Die Stromversorgungseinheit besteht aus fünf DC-DC -Wandlern. Alle Wandler werden von einer einzigen Quelle gespeist. Typischerweise wird in diesem Projekt ein USB-Netzteil verwendet [17].

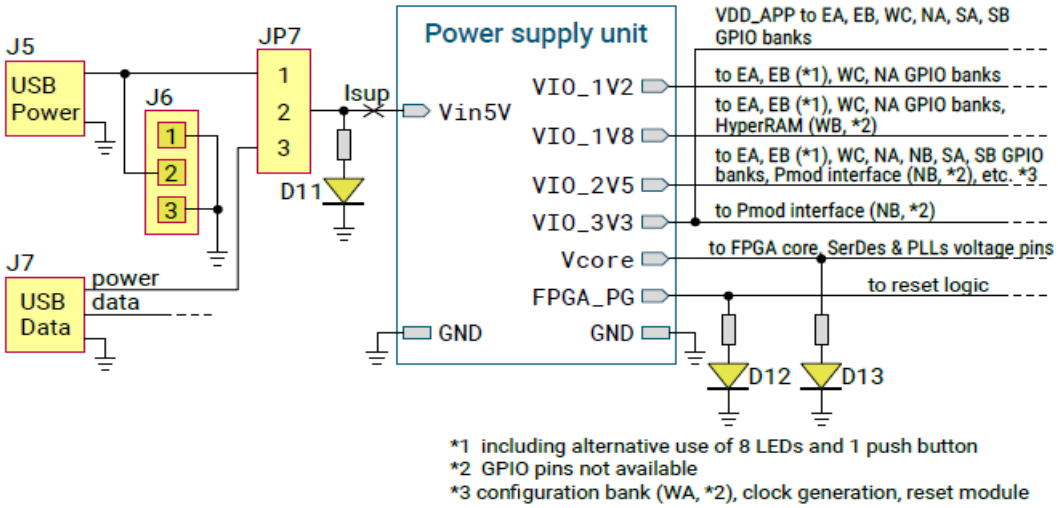


Abbildung 6: PCB-Stromversorgung. [17]

Abbildung 6 zeigt, dass die Stromversorgung über den USB-Anschluss J5 oder den USB-Datenanschluss J7 erfolgen kann.

4.1.2.2 GPIO Power Supply

Das CCGM1A1 FPGA bietet neun GPIO-Bänke (General Purpose Input Output). Deren GPIO-Versorgungsspannung kann auf der Evaluierungsplatine, wie in Tabelle 2 dargestellt, auf verschiedene Weise konfiguriert werden.

GPIO-Bänke	Eigenschaften des Systems
WC, NA, EA	GPIO- Spannung, ausgewählt aus drei On-Board-Quellen oder der Anwendungsspannung
EB	GPIO-Spannung wählbar aus drei On-Board-Quellen oder der Applikationsspannung, Zusatzfunktionen (LEDs und User-Button)
SA, SB	Einzelne 2,5 V Spannungsversorgung
WA	Konfigurationsbank, 2,5 V Versorgung
WB	HyperBusSpeicher, 1,8 V Versorgung
NB	Pmod-Schnittstelle, 2,5 V Versorgung

Tabelle 2: Zuordnung des GPIO-Stromversorgungsschemas zu den GPIO-Bänken [18]

Abbildung 6 zeigt die konfigurierbare GPIO-Stromversorgung für die GPIO-Banken WC, NA und EA. Diese Bänke können mit einer von drei OnBoard-Spannungen sowie der VIO_OUT-Spannung der Benutzeranwendung versorgt werden. Die ausgewählte OnBoard-Spannung wird auch an die

Benutzeranwendung (VIO_IN) geliefert und kann zur Versorgung der separaten Spannung VDD_APP verwendet werden [19].

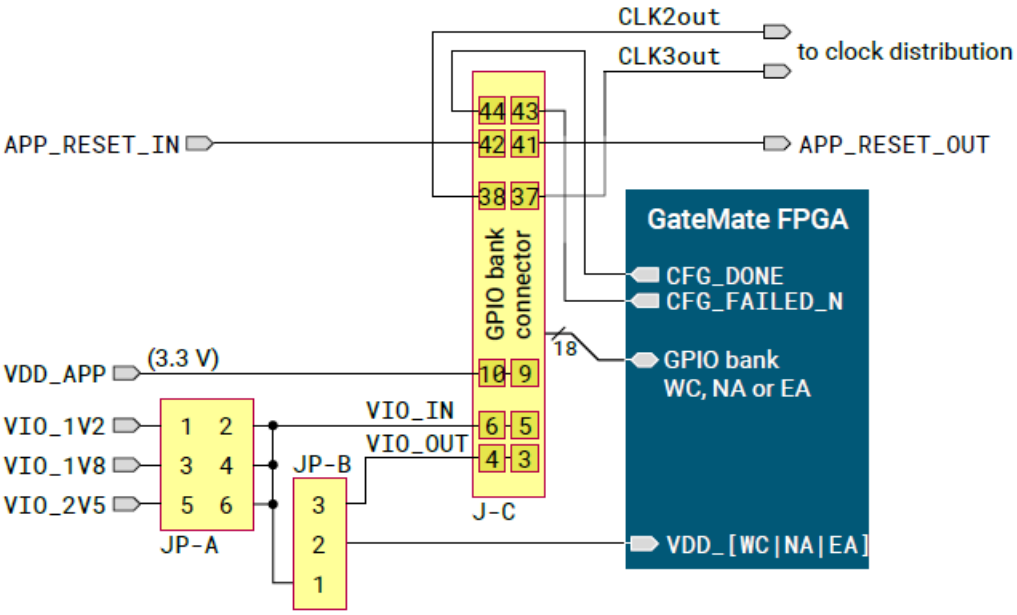


Abbildung 7: GPIO-Versorgung mit Spannungsauswahl [15]

4.1.2.3 SPI and JTAG Data Busses

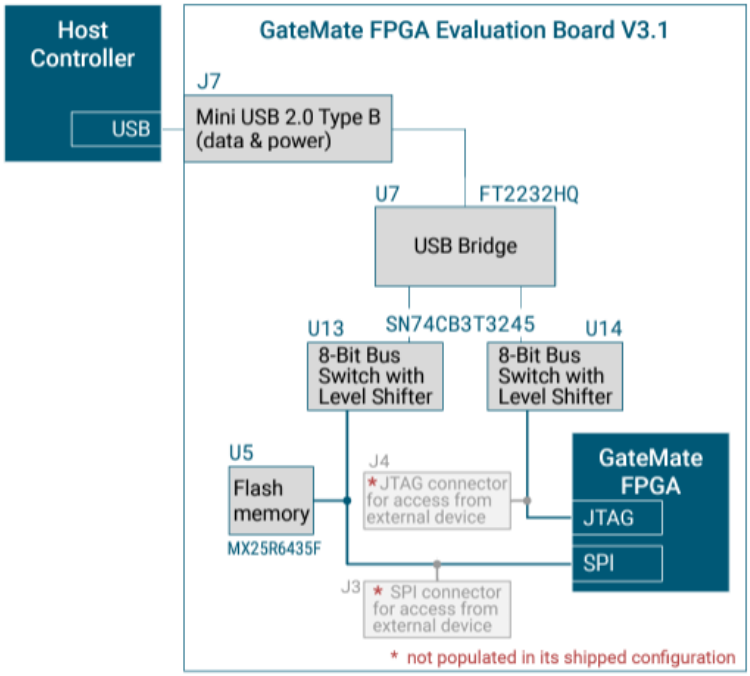


Abbildung 8: Blockschaltbild mit Details der SPI- und JTAG-Schnittstellen

Benutzergeräte können bei Bedarf direkt auf den SPI-Bus zugreifen. Dazu muss der Anschluss J3 (Raster 2,54 mm, 9-polig) belegt sein. Abbildung 9 zeigt die Pinbelegung des SPI-Steckers. Ebenso kann auch

die JTAG-Schnittstelle von Benutzergeräten direkt geschrieben und gelesen werden. Dazu muss der J4-Stecker (Raster 2,54 mm, 6-polig) belegt sein. Abbildung 10 zeigt die Pinbelegung des JTAG-Steckers.

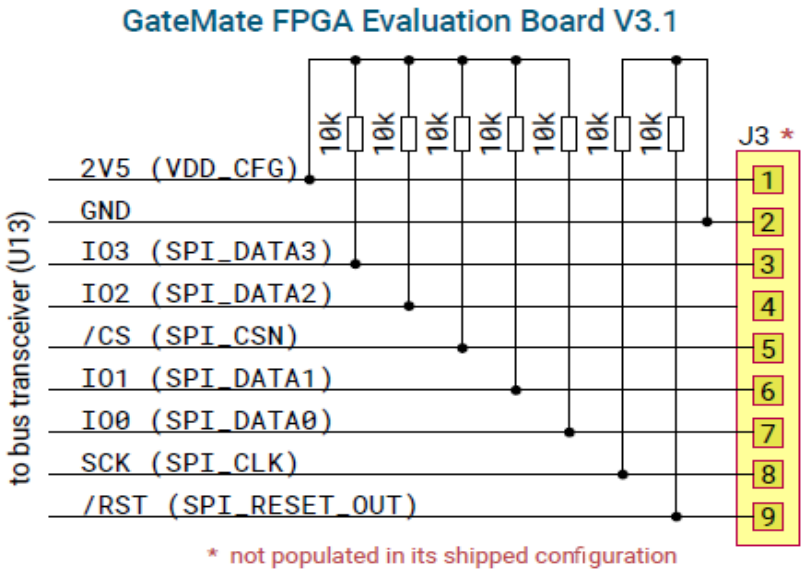


Abbildung 9: Optionaler SPI-Schnittstellenanschluss J3 [26]

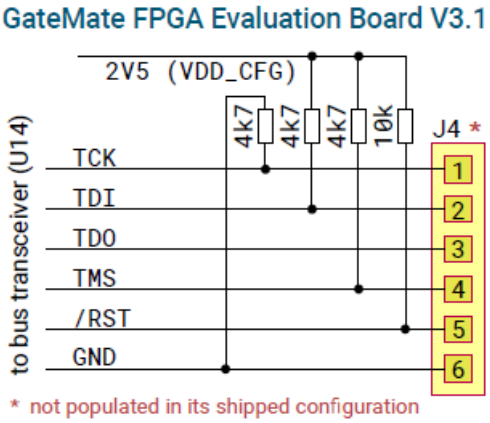


Abbildung 10: Optionaler JTAG-Schnittstellenanschluss J4 [26]

4.1.2.4 Konfiguration Mode and Reset

Der Konfigurationsmodus wird über den Schalter SW1 ausgewählt. Das Reset des FPGA hängt von verschiedenen Bedingungen ab:

- Schalter SW2 löst das Reset des FPGA aus.
- Nach dem Einschalten signalisieren alle DC-DC-Wandler, dass die Stromversorgung in Ordnung ist, um den Reset abzuschließen.
- Der Host-Controller löst den Reset über die SPI- oder JTAG-Schnittstelle aus. In diesem Fall sendet der U13- oder U14-Levenlshifter das Reset-Signal vom Host an den GateMate FPGA.
- Ein externes SPI-Gerät sendet ein Reset-Signal über Pin 9 des SPI-Anschlusses J3 an den FPGA.
- Ein externes JTAG-Gerät sendet ein Reset-Signal über Pin 5 des JTAG-Anschlusses J4 an den FPGA.
- Die Benutzeranwendung löst den Reset des FPGA über den GPIO Bank J-C-Anschluss Pin 41 (APP_RESET_OUT) aus. Dieses Signal wird durch den Jumper JP10 deaktiviert.

Der GPIO-Bank-Anschluss sendet das Rücksetzsignal APP_RESET_IN an die Anwendung des Benutzers. Dabei handelt es sich entweder um die Reset-Taste SW2 oder eine beliebige der oben genannten Reset-Bedingungen (siehe Abbildung 11).

Jede GPIO-Bankverbindung verfügt über ein eigenes APP_RESET_OUT-Signal. Alle GPIO-Bankverbindungen sind miteinander verbunden und werden in Abbildung 11 als 'User application reset out' bezeichnet [20].

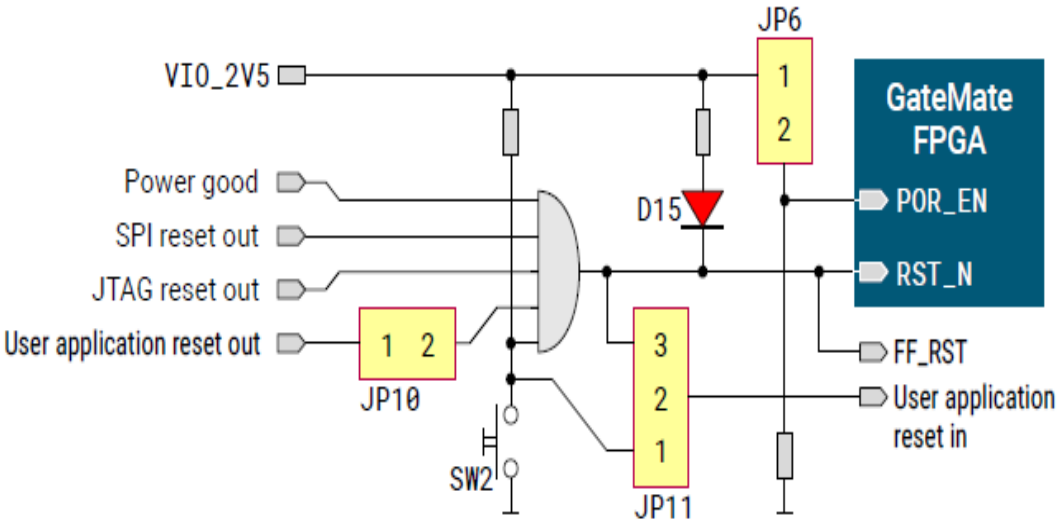


Abbildung 11: Reset Modul [21]

Nach der Konfiguration zeigt das GateMate FPGA an, ob der Datenstrom erfolgreich geladen wurde oder nicht.

- LED D10 leuchtet grün, wenn der Datenstrom vollständig übertragen wurde.

- LED D9 leuchtet rot, wenn ein Fehler aufgetreten ist. Beide Signale werden, wie in Abbildung 12 gezeigt, auch an den Host-Controller übertragen.

Wenn der Ladevorgang fehlschlägt, muss ein Reset ausgelöst werden, um den Datenstrom erneut zu laden [20].

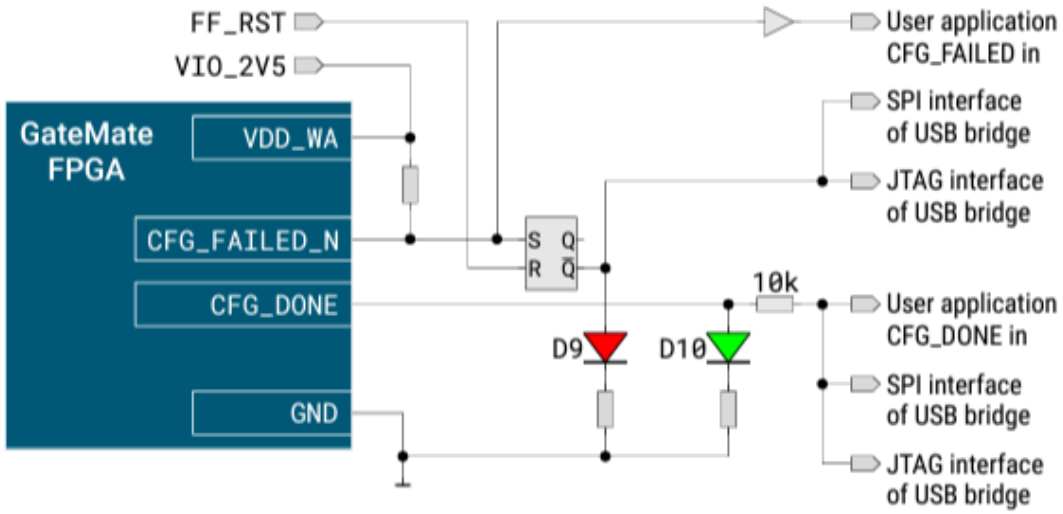


Abbildung 12: Konfiguration Statusmeldungen

4.1.2.5 Clock Generation and Distribution

Die Evaluierungsplatine verfügt über einen integrierten X1-Oszillator mit einer Frequenz von 10.000 MHz. Dieses Signal ist mit dem GPIO IO_SB_A8 verbunden, der dem Eingang für Takt 0 entspricht (siehe Abbildung 13). Der Oszillator kann mithilfe des Jumpers JP8 deaktiviert werden.

Das GateMate FPGA hat drei weitere Takteingänge, die mit der GPIO-Bank WA verbunden sind. Diese Taktgeber sind für die Verwendung auf dem Evaluierungsboard vorbereitet, aber einige Komponenten müssen, wie in Abbildung 15 gezeigt, vom Benutzer bestückt werden.

Clock 1: Es kann ein weiterer Oszillator installiert oder eine externe Taktquelle über die SMA-Buchse J8 verwendet werden.

Clock 2 und 3: Die Benutzeranwendung kann zwei Taktsignale an die Eingänge des FPGAs senden. Hierfür müssen die Jumper JP1 und JP2 gesetzt werden, um die Taktquelle auszuwählen.

Wie in Abbildung 14 dargestellt, kann alternativ ein LVDS (Low Voltage Differential Signaling) -Taktoszillator installiert werden, um einen SerDes (Serializer/Deserializer) -Eingangstakt zu erzeugen [22].

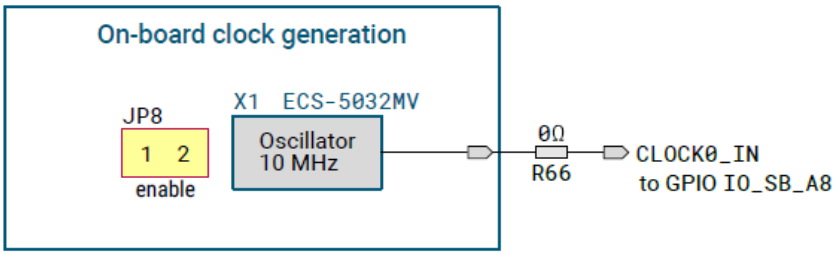


Abbildung 13: 10MHz On-Board Clock Oszillator [22]

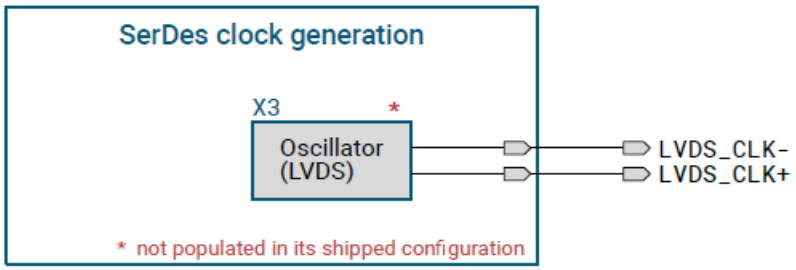


Abbildung 14: Optional SerDes Clock (LVDS Clock Oszillator) [22]

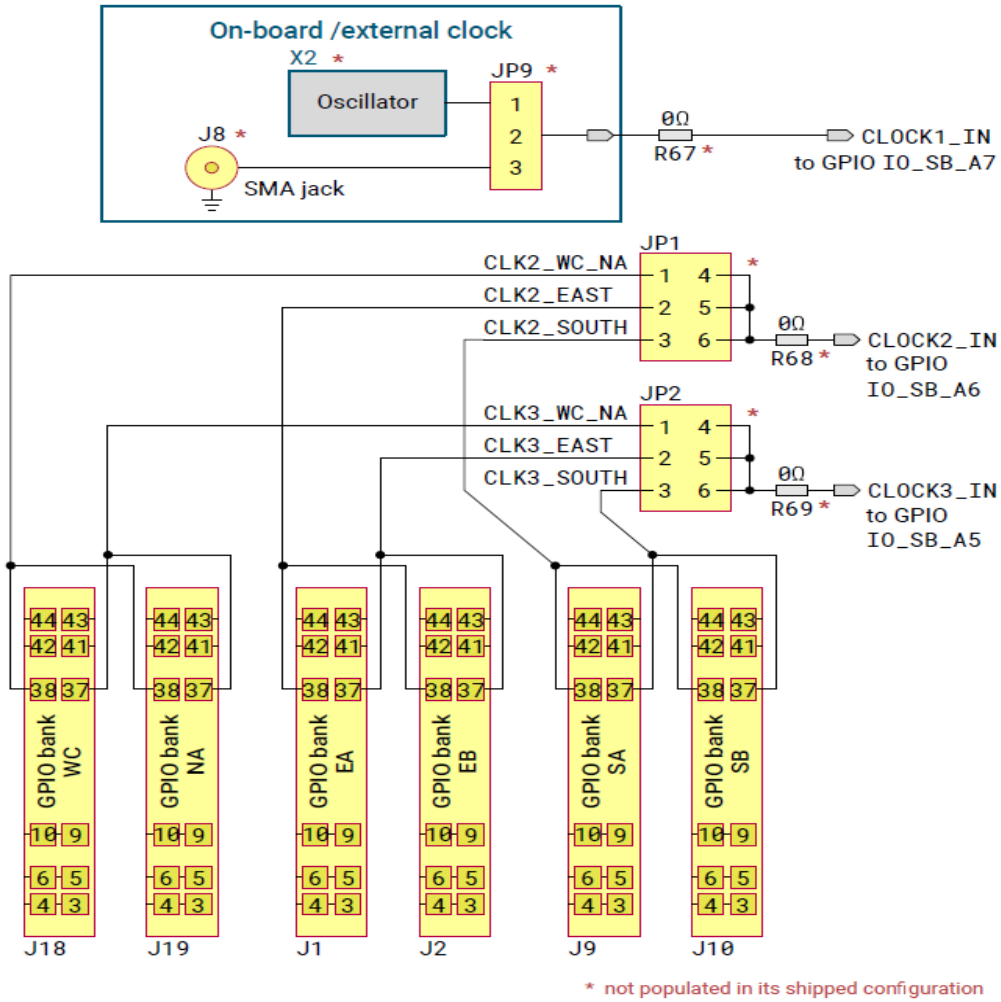


Abbildung 15: Optional Clock Signals [22]

4.1.2.6 GPIO-Verbindungen zur User Applikation

Die Anwendung des Benutzers kann mit einer oder mehreren GPIO-Banken verbunden werden, wobei bis zu 6 Banken verfügbar sind und die restlichen GPIO-Banken anderweitig zugeordnet sind.

Alle GPIO-Bank-Anschlüsse haben die gleiche Pinbelegung, wie in Abbildung 16 dargestellt. Die GPIOs sind auf zwei verschiedene Arten mit einem Levelshifter verbunden, um die freie Wahl einer GPIO-Bank für Benutzeranwendungen mit einer Bank zu ermöglichen. Das Routing der GPIO-Signale der Ae- und Be-Banken ist paarweise und nach Paarlängen angeordnet.

Die Benutzeranwendung wird über den eingebauten 3,3-V-DC/DC-Wandler mit Strom versorgt. Außerdem wird die GPIO-Spannung von dem Evaluierungs-Board an die Benutzeranwendung geliefert oder umgekehrt.

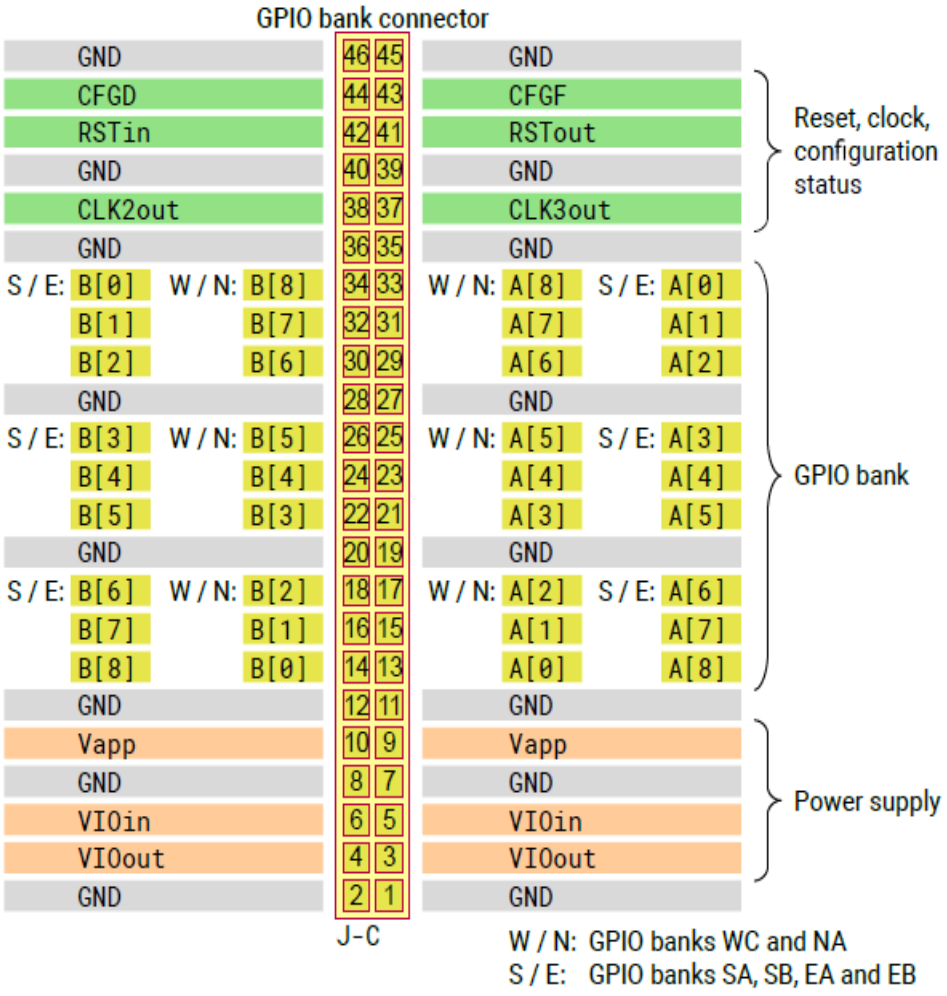


Abbildung 16: GPIO-Bank-Anschluss J-C [23]

4.1.2.7 PMOD-Interface

Die GPIO-Bank NB ist eine PMOD-Schnittstelle mit den 12-poligen Standardanschlüssen J17A und J17B.

Die Versorgungsspannung von 3,3 V muss über den Jumper JP14 ausgewählt werden, um der PMOD-Spezifikation zu entsprechen. Wenn bestimmte Anwendungen dies erfordern, kann alternativ eine Versorgungsspannung von 2,5 V gewählt werden.

In Tabelle 3 sind die GPIO-Signale aufgeführt, die mit den Anschlüssen J17A und J17B der PMOD-Schnittstelle verbunden sind [24].

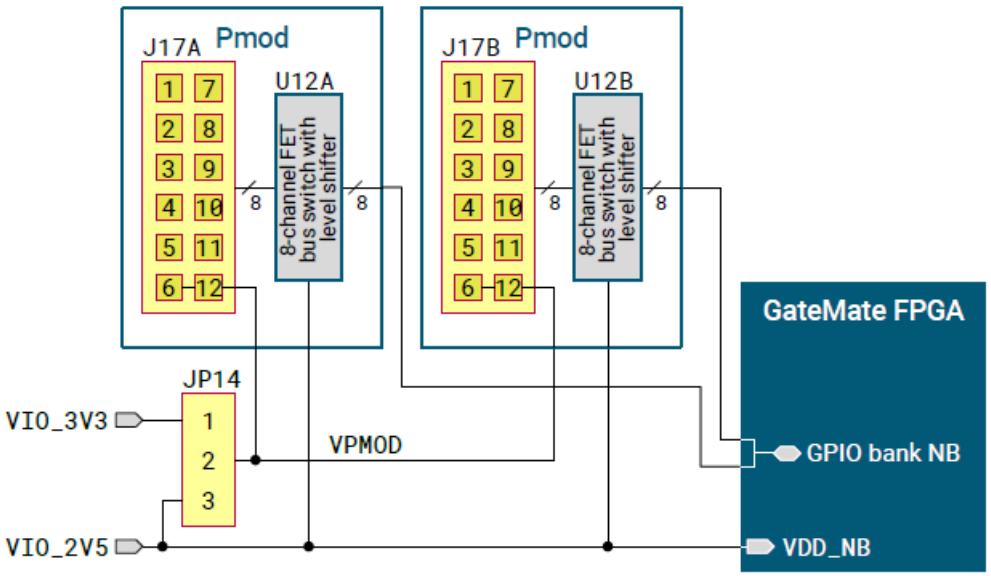


Abbildung 17: GPIO-Bank NB mit PMOD-Schnittstelle [24]

J17A/B pin	PMOD signal	GPIO PMOD A	GPIO PMOD B
1	IO1	IO_NB_A0	IO_NB_A4
2	IO3	IO_NB_A1	IO_NB_A5
3	IO5	IO_NB_A2	IO_NB_A6
4	IO7	IO_NB_A3	IO_NB_A7
5	GND		
6	VPMOD		
7	IO2	IO_NB_B0	IO_NB_B4
8	IO4	IO_NB_B1	IO_NB_B5
9	IO6	IO_NB_B2	IO_NB_B6
10	IO8	IO_NB_B3	IO_NB_B7
11	GND		
12	VPMOD		

Tabelle 3: PMOD-Signalbelegung der Anschlüsse J17A und J17B

4.1.2.8 HyperRAM Device

Die GPIO-Bank WB wird verwendet, um den Zugriff auf einen HyperRAM-Speicher bereitzustellen (siehe Abbildung 18). Die Komponente U10 entspricht einem 64-MB- Speicherbaustein, der an die GPIO-Bank WB angeschlossen ist (siehe Tabelle 4).

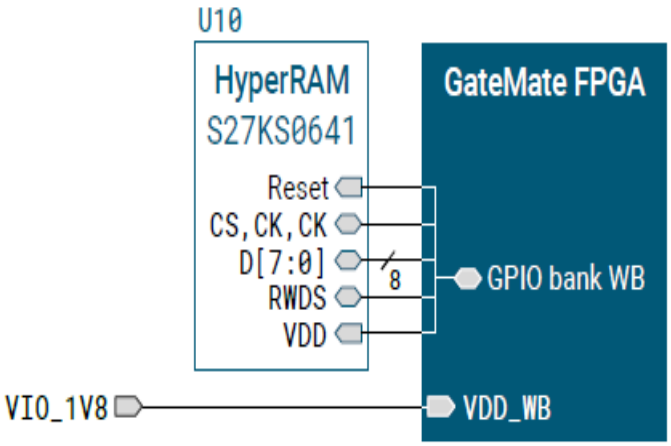


Abbildung 18: GPIO-Bank WB mit HyperRAM Baustein [27]

GPIO	HyperRAM Signal	GPIO	HyperRAM Signal
IO_WB_A0	–	IO_WB_B0	CS
IO_WB_A1	–	IO_WB_B1	–
IO_WB_A2	RESET	IO_WB_B2	–
IO_WB_A3	CK	IO_WB_B3	CK
IO_WB_A4	–	IO_WB_B4	RWDS
IO_WB_A5	DQ0	IO_WB_B5	DQ1
IO_WB_A6	DQ2	IO_WB_B6	DQ3
IO_WB_A7	DQ4	IO_WB_B7	DQ5
IO_WB_A8	DQ6	IO_WB_B8	DQ7

Tabelle 4: GPIO-Zuweisung an das On-Board-HyperRAM-Baustein [27]

4.1.2.9 SERDES Interface

Das Evaluation Board ist auf die Nutzung der im Gatemate integrierten SerDes-Schnittstelle vorbereitet Abbildung 19 zeigt, dass die SMA-Buchsen J13...J16 vom Benutzer bestückt werden müssen, um Zugriff auf die SerDes-Schnittstelle des CCGM1A1 zu erhalten.

Es gibt zwei Möglichkeiten, das SerDes-Taktsignal einzugeben:

1. Die SMA-Buchsen J11 und J12 werden verwendet, um einen externen LVDS-Taktgeber zu versorgen.

2. Ein LVDS-Oszillator und die Vorwiderstände R73 und R74 werden bestückt.

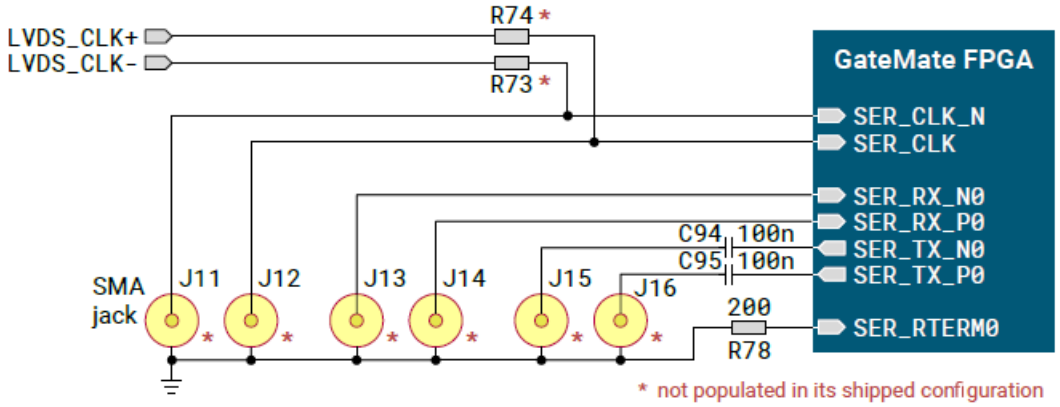


Abbildung 19: Optionale SerDes-Schnittstelle [27]

4.2 PMOD USBUART Modul

Das USBUART PMOD-Modul bietet eine Kreuzkonvertierung von USB zu UART unter Verwendung des FTDI FT232R-Bausteins (siehe Abbildungen 20 und 21). Das USBUART Modul bietet eine einfache Möglichkeit Daten in unterschiedlichen Datenformaten zu empfangen und zu senden. In diesem Projekt sollen die Daten aus dem FPGA entnommen und über das USBUART PMOD Modul per USB an einen PC gesendet werden.

Abbildung 20: Pmod USBUART [28]

4.2.1 Eigenschaften

- USB auf serielle UART-Schnittstellen Kreuzkopplung
- Micro-USB-Anschluss
- Möglichkeit, die Systemplatine über den FTDI-Chip zu versorgen
- Kleine Leiterplatte für flexible Ausführungen 1.0“ × 0.8” (2.5 cm × 2.0 cm)
- 6-poliger Pmod-Stecker
- Folgt der Digilent Pmod Schnittstellenspezifikation Type 4

4.2.2 UART

UART steht für Universal Asynchronous Receiver Transmitter und entspricht einem seriellen Datenübertragungsprotokoll bei dem die Daten asynchron, d.h. ohne Taktsignal übertragen werden. Eine Alternative Abkürzung lautet ACIA, die für Asynchronous Communication Interface Adapter steht, heute jedoch als Bezeichnung nicht mehr gebräuchlich ist. Umgangssprachlich wird damit das Bauteil bezeichnet, das verwendet wird, um einen Computer über einen seriellen Anschluss zu verbinden.

In Computern werden Daten als mehrbitige Datenwörter organsiert, und müssen erst so umgewandelt werden, dass sie über eine serielle Verbindung, die nur über einen einzigen Draht verfügt, nacheinander versendet werden können.

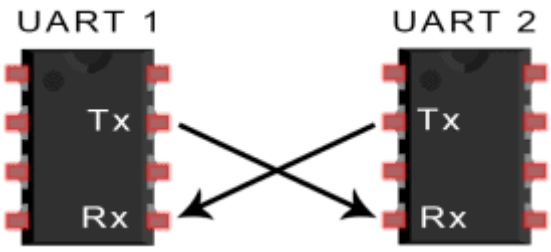


Abbildung 21: Kommunikationsprotokoll

Heutzutage werden UARTs in der Regel in Komponenten wie Mikrocontroller integriert. In diesen Fällen sind UARTs keine eigenständigen Bauteile mehr, sondern eine periphere Funktion des Mikrocontrollers.

Ein UART-Rahmen besteht aus den folgenden Bits:

- einem Startbit, das immer auf 0 gesetzt ist und zur Synchronisierung des Empfängers verwendet wird.
- Daten deren Größe zwischen 5 und 9 Bits liegen können. Die Bits werden vom LSB (niederwertigstes Bit) beginnend bis zum MSB (höchstwertigstes Bit) gesendet.
- ein Parität Bit, das optional ungerade, gerade oder weggelassen werden kann und zur Fehlererkennung dient.
- einem Stopp-Bit, das immer auf 1 gesetzt ist. Die Dauer dieses Bits variiert zwischen 1, 1,5 und 2 Bitzeiten, wobei die Dauer von den Benutzern frei wählbar ist.

Der logische Ruhepegel, wenn keine Daten versendet werden, entspricht einer logischen 1. Ein beispielhafter UART-Rahmen ist in Abbildung 23 dargestellt.

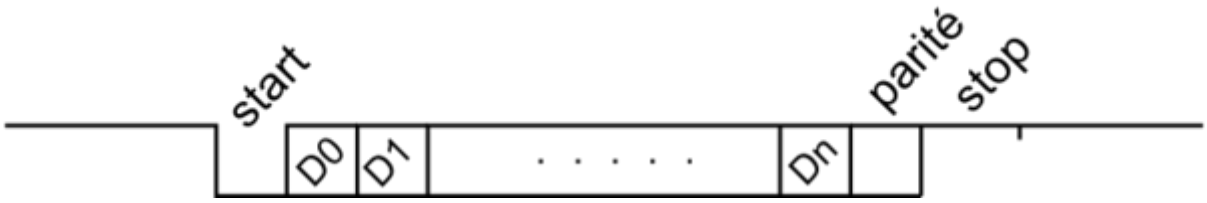


Abbildung 22: Aufbau eines UART-Rahmens [29]

4.2.2.1 Empfänger

Alle Operationen der UART-Hardware werden von einem internen Taktsignal gesteuert, das einem Vielfachen der Datenrate entspricht und typischerweise bei dem 8- oder 16-fachen der Bitrate liegt. Der Empfänger überprüft bei jedem Taktimpuls den Zustand des eingehenden Signals und sucht nach dem Beginn des Startbits. Wenn das Startbit mindestens die Hälfte der Bitzeit dauert, ist es gültig und signalisiert damit den Beginn eines neuen Zeichens. Andernfalls wird es als Störimpuls betrachtet und ignoriert. Nach dem Abwarten einer weiteren Bit-Zeit wird der Leitungszustand erneut abgetastet und der resultierende Pegel per Takt in ein Schieberegister gesendet. Nachdem die für die Zeichenlänge erforderliche Anzahl von Bitperioden von üblicherweise 5 bis 8 Bit verstrichen ist, wird der Inhalt des Schieberegisters parallel dem empfangenden System zur Verfügung gestellt. Der UART setzt ein Flag, das anzeigt, dass neue Daten verfügbar sind an dieser Stelle wird kein Komma benötigt. und kann auch einen Prozessorinterrupt erzeugen, um den Hostprozessor aufzufordern, die empfangenen Daten zu übertragen [29].

4.2.2.2 Sender

Der Sendevorgang ist einfacher, da das Timing nicht aus dem Zustand der Leitung ermittelt werden muss und nicht an feste Zeitintervalle gebunden ist. Sobald das Sendersystem ein Zeichen im Schieberegister ablegt und die Versendung des vorherigen Zeichens beendet hat, generiert der UART ein Startbit, verschiebt die erforderliche Anzahl von Datenbits auf die Leitung, generiert und sendet das Paritätsbit, falls die Verwendung eines Paritätsbits vorgesehen ist und sendet die Stoppbits. Da beim Vollduplexbetrieb die Zeichen gleichzeitig gesendet und empfangen werden müssen, verwenden UARTs zwei verschiedene Schieberegister für die gesendeten und empfangenen Zeichen. Hochleistungs-UARTs können einen FIFO-Puffer (First In First Out) für die Übertragung enthalten, damit eine CPU oder ein DMA-Controller mehrere Zeichen in einem Burst in den FIFO ablegen kann, anstatt ein Zeichen nach dem anderen in das Schieberegister ablegen zu müssen. Da die Übertragung eines oder mehrerer Zeichen im Verhältnis zur Geschwindigkeit der CPU sehr lange dauern kann, hält ein UART ein Flag bereit, das den Belegungszustand anzeigt, damit das Hostsystem weiß, ob sich mindestens ein Zeichen im Sendepuffer oder im Schieberegister befindet. Das Flag vermittelt die Information "bereit für das/die nächste(n) Zeichen", was auch durch ein Interrupt signalisiert werden kann [29].

4.2.2.3 Anwendung

Für einen ordnungsgemäßen Betrieb müssen der Sende- und der Empfangs-UART auf die gleiche Bitrate, Zeichenlänge, Parität und Stoppbits eingestellt sein. Der empfangende UART kann bestimmte nicht übereinstimmende Parameter erkennen und ein Indikatorbit "Rahmenfehler" für das Hostsystem setzen. In Ausnahmefällen erzeugt der empfangende UART einen erratischen Strom verstümmelter Zeichen und überträgt diese an das Hostsystem. Typische serielle Schnittstellen, die bei an Modems

angeschlossenen Personal Computern verwendet werden, verwenden acht Datenbits, keine Parität und ein Stoppbit; bei dieser Konfiguration entspricht die Anzahl der ASCII-Zeichen pro Sekunde der Bitrate geteilt durch 10 [29].

4.3 PMOD Button Module

4.3.1 PMOD Button Modul

Für die Erweiterung des Gatemate Evaluations-Bords um zusätzliche Bedienelemente wird ein PMOD-Modul mit vier Drucktasten verwendet. Die Drucktasten befinden sich auf einer kleinen Platine mit einer Größe von 3,0 cm × 2,0 cm. Der 6-polige PMOD-Anschluss entspricht der Schnittstellenspezifikation des Digilent PMOD Typs. Das PMOD-Modul wird über einen entsprechenden Stecker auf dem Evaluation-Board mit den GPIO-Eingängen des Gatemate FPGAS verbunden. Bei diesem PMOD-Modul werden Signale nur unidirektional zum FPGA gesendet. Sollten Signale auf Grund einer fehlerhaften Konfiguration vom FPGA in Richtung des PMOD-Moduls getrieben werden, werden diese Signale ignoriert. Wie in Abbildung dargestellt verfügt jede Taste über jeweils eine Datenleitung, einen Entprell Filter und einen Schmitt-Trigger, so dass jede Taste unabhängig von den anderen gedrückt werden kann. Der Benutzer kann dementsprechend die Tasten auch gleichzeitig drücken und dadurch bis zu 16 verschiedene Eingangskombinationen generieren. Die Pinbelegung und ein Schaltplan des PMOD-Moduls sind unten aufgeführt.

Abbildung 23: Pmod_Button [30]

4.3.2 Beschreibung der Funktionsweise

Der PmodBTN verwendet analoge Filter, die aus zwei Widerständen und einem Kondensator bestehen, um das Signalprellen, das beim Drücken einer Taste natürlich vorkommt, zu absorbieren und zu glätten. Das geglättete Spannungssignal wird dann an einen Schmitt-Inverter-Trigger gesendet, der eine hohe logische Spannung an die Systemplatine sendet, wenn ein Knopf gedrückt wird, oder eine niedrige logische Spannung, wenn der Knopf nicht gedrückt wird.

Pin	Signal	Beschreibung
1	BTN0	Taste 0
2	BTN1	Taste 0
3	BTN2	Taste 0
4	BTN3	Taste 0
5	GND	Masse der Spannungsversorgung
6	VCC	Stromversorgung (3,3V/5V)

Tabelle 5: Pin-Beschreibungen wie auf dem Pmod beschriftet

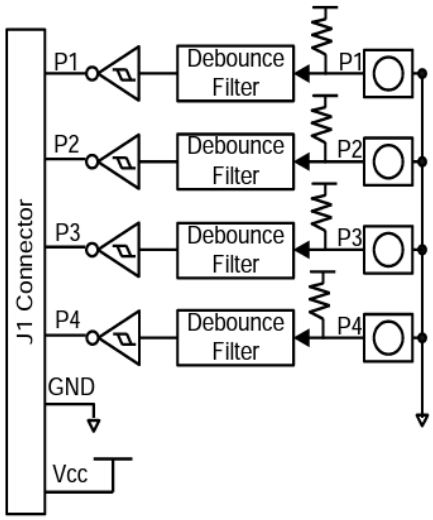


Abbildung 24: BTN-Schaltkreisdiagramm [30]

Eine externe Spannungsversorgung, die an das PMOD-Modul angelegt wird, muss zwischen 1,65 V und 5,5 V liegen. Es wird jedoch empfohlen, das PMOD-Modul mit 3,3 V zu betreiben.

5. Verwendete Software

5.1 VHDL

Very High Speed Integrated Circuit Hardware Description Language (auch VHSIC Hardware Description Language), kurz VHDL, ist eine Hardwarebeschreibungssprache, mit der es möglich ist, digitale Systeme textbasiert zu beschreiben. VHDL ist seit 1987 als IEEE-Standard festgelegt und es gibt inzwischen einige ebenfalls standardisierte Spracherweiterungen. Darüber hinaus gibt es Sprachderivate wie zum Beispiel VHDL-AMS, mit deren Hilfe auch analoge oder Mixed-Signal-Systeme beschrieben werden können.

Als Beschreibungssprache ist VHDL keine Programmiersprache. Da VHDL jedoch Objekte beschreibt, deren Aufgabe im Allgemeinen die Informationsverarbeitung ist, ist es möglich, Daten durch ihre Simulation zu verarbeiten, indem die "Eingabedaten", die für diese Simulationsausführung von der simulierten Hardware geliefert werden, in "Ergebnisdaten" umgewandelt werden. Durch diesen Umweg kann VHDL in Kombination mit einem Simulator wie eine Programmiersprache Turing-vollständige Datenverarbeitung beschreiben.

Durch fortschrittliche Schaltungsgeneratoren ist es mitunter sogar möglich, anstatt des Hardwareaufbaus für einen Algorithmus nur den Algorithmus selbst anzugeben. Die dazugehörige Schaltung wird dann vollautomatisch erzeugt. Dies nähert VHDL einer Programmiersprache weiter an [31].

5.2 Ubuntu

Ubuntu ist eine Linux-Distribution, die auf Debian basiert und hauptsächlich aus freier und quelloffener Software besteht. Ubuntu wird offiziell in drei Editionen veröffentlicht: Desktop, Server und Core für Geräte des Internets der Dinge und Roboter. Alle Editionen können auf dem Computer allein oder in einer virtuellen Maschine ausgeführt werden. Ubuntu ist ein beliebtes Betriebssystem für Cloud Computing, mit Unterstützung für OpenStack. Der Standard-Desktop von Ubuntu wechselte 2017 mit der Veröffentlichung von Version 17.10 nach fast 6,5 Jahren vom hauseigenen Unity zurück zu GNOME.

Ubuntu wird alle sechs Monate veröffentlicht wobei Long-Term-Support (LTS)-Releases alle zwei Jahre zur Verfügung gestellt werden. Im Oktober 2022 ist die jüngste Version 22.10 ("Kinetic Kudu"), und die aktuelle Version mit Langzeitunterstützung ist 22.04 ("Jammy Jellyfish").

Ubuntu wird von der britischen Firma Canonical und einer Gemeinschaft von anderen Entwicklern im Rahmen eines meritokratischen Verwaltungsmodells entwickelt. Canonical bietet Sicherheitsupdates und Support für jede Ubuntu-Version an und zwar ab dem Veröffentlichungsdatum und bis zum Erreichen des vorgesehenen End-of-Life-Datums (EOL) [32].

5.3 Hterm

HTerm von Tobias Hammer ist ein Terminalprogramm für die serielle RS232 Schnittstelle und richtet sich an Programmierer und Maker, die diese nutzen. So lässt sich mit der Software für Windows sowie Linux unter anderem ein Gerät zur Steuerung technischer Vorgänge, ein Mikrocontroller wie Arduino und insbesondere die UART-Schnittstelle ansprechen. HTerm ist kein Terminalprogramm im üblichen Sinne. In der grafischen Oberfläche kann man unter anderem den Port wie z.B. COM 1, die von Daten Baud-Rate und die Parität für Senden und Empfangen angeben. Außerdem ist im Interface eine Ein- und Ausgabe der Daten in Dezimal-, Hexadezimal- und Binärzahlen sowie als ASCII konfigurierbar, wobei die Hexadezimal-, Binär-, Dezimal- und ASCII-Darstellung das Debugging und die Bearbeitung in HTerm vereinfacht. Des Weiteren sind noch verschiedene Newline-Codes und Font-Größen einstellbar [33].

5.4 Vivado Simulation

Vivado ist ein Hardware Description Language (HDL)-Simulator, mit dem Verhaltens-, Funktions- und Zeitsimulationen für Designs in VHDL, Verilog und Mixed Language durchgeführt werden können [34].

5.5 Yosys

Yosys ist ein Softwarepaket für die Logiksynthese (engl. RTL synthesis), mit dem man eine logische Schaltung aus ihrer Beschreibung in der Hardwarebeschreibungssprache Verilog in eine Netzliste aus technologiespezifischen Gattern umwandeln kann. Eine derartige Gatternetzliste kann bei der Implementierung einer digitalen Schaltung als anwendungsspezifischer integrierter Schaltkreis (engl. Application Specific Integrated Circuit, ASIC) einem Place & Route Tool zur Platzierung und Verdrahtung und Generierung eines Layouts übergeben werden. Bei der Implementierung auf einem FPGA kann die Netzliste auf die vorhandenen Ressourcen abgebildet werden und zur Generierung eines Konfigurationsdatenstrom herangezogen werden. Yosys führt auch formale Verifikationsaufgaben durch. Er wurde von Clifford Wolf entwickelt. Im Jahr 2020 kündigte der deutsche Hersteller Cologne Chip AG an, Yosys als Werkzeug für die RTL-Synthese ihrer FPGAs zu unterstützen. openFPGAloader dient der Konfiguration dieser FPGAs und ist auf Github verfügbar [35].

5.5.1 Yosys Funktionen

Yosis verfügt über die folgenden Funktionen:

- Verarbeitung des größten Teils des Sprachumfangs von Verilog-2005.
- VHDL kann mit Hilfe des GHDL-Plugins ghdl-yosys-plugin verarbeitet werden.
- BLIF / EDIF/ BTOR / SMT-LIB / simple RTL-Netzlisten können nach Verilog konvertiert werden.
- Formale Verifikation durch Prüfung von Eigenschaften und Äquivalenzen.
- Mapping für ASICs im Liberty File Format.
- Mapping für FPGAs der Xilinx 7- und Lattice iCE40-Serie.
- Fundament oder Frontend für benutzerdefinierte Flows.

In diesem Projekt ist insbesondere die Funktion zur Verarbeitung von VHDL relevant.

Yosys kann an jede beliebige Syntheseaufgabe angepasst werden, indem die vorhandenen Durchläufe zu Algorithmen mit Hilfe von Synthese-Skripten kombiniert und bei Bedarf zusätzliche Durchläufe durch Erweiterung der Yosys C++-Codebasis hinzugefügt werden.

5.6 GHDL

GHDL ist eine Abkürzung für G Hardware Design Language, Derzeit hat G keine Bedeutung. Dieses Programm ist ein VHDL-Compiler, der fast jedes VHDL-Programm ausführen kann. GHDL ist kein Synthesewerkzeug.

Im Gegensatz zu einigen anderen Simulatoren ist GHDL ein Compiler. Er übersetzt eine VHDL-Datei direkt in Maschinencode, indem er das GCC-Backend verwendet und keine Zwischensprache wie C oder C++ benutzt. Daher sollte der kompilierte Code schneller sein und die Analysezeit sollte kürzer sein als bei einem Compiler, der eine Zwischensprache verwendet [36].

5.7 Place & Route

Die eigentliche Implementierung wird durch das Place&Route-Tool vom Cologne Chip durchgeführt Im Allgemeinen akzeptiert das Tool Verilog-Netzlisten aus architekturspezifischen Primitiven, die mit dem Yosys synth_gatemate-Pass generiert wurden. Die Primitiven werden auf die Ressourcen des FPGA verteilt und die Verbindungskomponenten so konfiguriert, dass die benötigten Verbindungen etabliert werden. Anschließend wird ein Bitstream generiert. Die Planung der Eingangs- und Ausgangsportswird mit Hilfe von Constraint-Dateien im CCF-Format verwaltet [37].

5.8 openFPGAloader

Bei openFGPAloader handelt es sich um ein universelles Dienstprogramm für die Programmierung von FPGAs. Das Tool ist kompatibel zu vielen Boards, Kabeln und FPGAs der wichtigsten Hersteller (Xilinx, Altera/Intel, Lattice, Gowin, Efinix, Anlogic, Cologne Chip). openFPGALoader funktioniert unter Linux, Windows und macOS [38].

5.9 Zadig USB

Zadig ist eine Windows-Anwendung, die generische USB-Treiber wie WinUSB, libusb-win32/libusb0.sys oder libusbK installiert, um den Zugriff auf USB-Geräte zu ermöglichen [39].

Dies kann besonders dann nützlich sein, wenn:

- der Zugriff auf ein Gerät über eine libusb-basierte Anwendung ermöglicht werden soll.
- ein generischer USB-Treiber aktualisiert werden soll.
- der Zugriff auf ein Gerät über WinUSB etabliert werden soll.

Der Zadig USB Installer wird in diesem Projekt verwendet, um den Treiber zu installieren, der den Zugriff auf das Gatemate-Evaluierungsboard ermöglicht.

6. Arbeitsablauf

6.1 Entwurfs-Flow des GateMate FPGAs

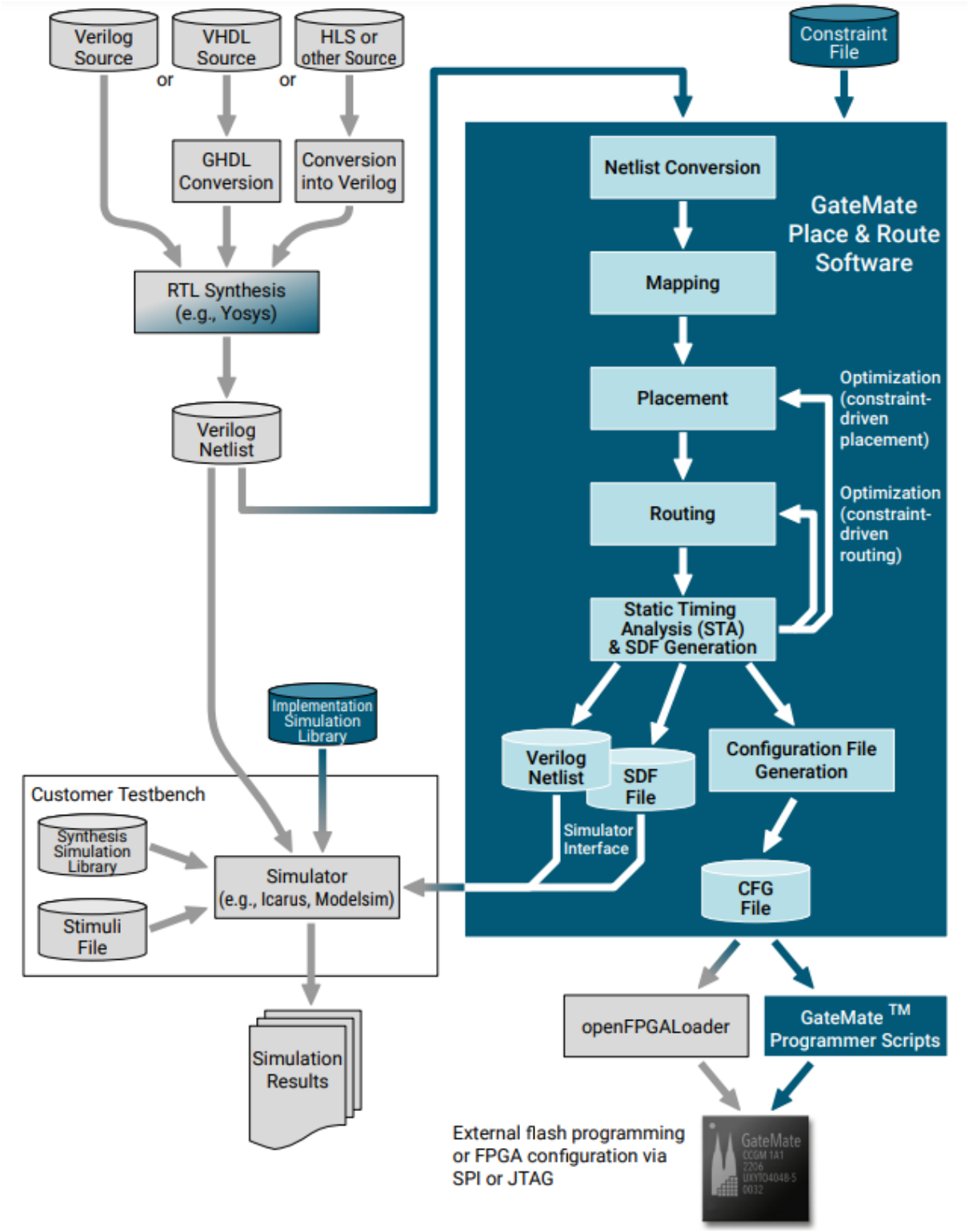


Abbildung 25: Ablauf der Synthese und Implementierung [40]

Der Arbeitsablauf von der Eingabe des Designs bis zur Konfigurationsdatei des Gatemate FPGAs ist in Abbildung 23 dargestellt. Die Eingabe des Designs erfolgt mithilfe von Eingabewerkzeugen in einer beliebigen Hardwarebeschreibungssprache (HDL). Die Yosys Open SYnthesis Suite (Yosys) wird verwendet, um die Synthese auf der Ebene des Registertransfer-Levels (RTL) durchzuführen. Yosys bietet erweiterte Unterstützung für Verilog. VHDL-Quellen können mit dem Yosys-ghdl-Plugin über GHDL synthetisiert werden. Andere HDLs und Tools mit einem Verilog-Backend können ebenfalls verwendet werden. Um die Bedienung zu vereinfachen, stellt Cologne Chip vordefinierte Skripte zur Verfügung, deren Ausführung durch kurze Befehle mithilfe eines Makefiles gesteuert wird. In der Datei Synth.tcl, die sich im Verzeichnis tcl/ befindet, sind alle Entwurfsdateien in der Form: src/File.vhd gespeichert. Der vollständige Eintrag für dieses Projekt in der Datei synth.tcl lautet wie folgt.

ghdl --warn-no-binding -C --ieee=synopsys src/FSM.vhd src/inverter.vhd src/PUF.vhd src/Ring_osc.vhd src/Timer.vhd src/Push_Button src/Comparator.vhd src/serial_tx.vhd src/MUX_Uart.vhd -e ${top}.

Im Makefile muss die Variable top auf das Toplevel-Modul gesetzt werden. In diesem Projekt ist der Name des Toplevel-Moduls RO_PUF. Normalerweise erkennt Yosys aber das Toplevel-Modul automatisch in der Menge der Eingabedateien. In der Datei config.mk befinden sich alle Befehle, die über make für die Implementierung eines Entwurfs auf dem Gatemate FPGA aufgerufen werden müssen. Der Switch -noclkbuf wird dem synth_gatemate-Pass hinzugefügt, um die automatische BUFGs Generierung zu unterbinden, da diese zu einem Absturz des Tools führt. Für die Synthese wird der Befehl "make synth_vhdl" verwendet. Nach Eingabe des Befehls "make synth_vhdl" wird unter anderem das Skript tcl/synth.tcl ausgeführt und alle notwendigen Parameter übergeben, um die Entwurfsdateien zu öffnen, die Synthese zu starten und das Ergebnis der Synthese in ein Unterverzeichnis zu schreiben. Die Synthese erzeugt eine Gate-Level Darstellung des Schaltungsentwurfs in Form einer Verilog-Netzliste, die aus architekturspezifischen Primitiven besteht. Nach erfolgreichem Abschluss der Synthese wird die Datei RO_PUF_synth.v im Unterverzeichnis net/ erstellt.

Der nächste Schritt besteht darin, den Befehl "make impl" für die Implementierung zu verwenden. Der Befehl "make impl" ruft das Place & Route Werkzeug von Cologne Chip auf und übergibt alle notwendigen Parameter, um die Netzwerkliste des Designs zu öffnen und das Mapping auf die FPGA-Ressourcen durchzuführen. Im Unterverzeichnis src/ befindet sich außerdem die Datei RO_PUF_00.ccf, mit der die Ein- und Ausgänge des Top-Level-Moduls des Schaltungsdesigns auf die Ports des Gatemate FPGAs abgebildet werden. Die Einträge in der CCF-Datei haben das folgende Format:

<pin-direction> "<pin-name>" Loc = "<pin-location>" | <opt.-constraints>;

Abhängig von den Einstellungen erzeugt das Tool mindestens die folgenden Ausgabedateien:

- Configuration_Bitsream: RingOscillator_PUF/RO_PUF_00.cfg.bit.
- Verilog-Netzliste für die Post-Implementierungssimulation: net/RO_PUF_00.v
- SDF-Timing-Datei für die Post-Implementierungssimulation: RingOscillator_PUF/RO_PUF_00-.SDF
- Pin-Datei (falls aktiviert): RO_PUF_00.pin
- Platzierungsdatei (falls aktiviert): RingOscillator_PUF/RO_PUF_00.place
- Querverweisdatei (falls aktiviert): RingOscillator_PUF/RO_PUF_00.crf

6.2 Projektstrukturen

Cologne Chip empfiehlt für die Durchführung von Implementierungsprojekten die in Abbildung 26 dargestellte, genau festgelegte Ordnerstruktur. Im Verzeichnis cc-tool-win/ befinden sich die beiden Unterverzeichnisse bin und workspace. Im Unterverzeichnis bin/ befinden sich die drei weiteren Ordner openFPGALoader, p_r und Yosys, in denen die verwendeten EDA-Tools installiert sind. Diese Werkzeuge werden während der Synthese, der Implementierung und der Konfiguration des Projekts aufgerufen. Im Unterverzeichnis workspace/ sind die verschiedenen Anwendungsprojekte in Unterverzeichnissen abgelegt. In diesem Projekt wurde ein Verzeichnis RingOscillator_PUF erstellt. Außerdem befindet sich in diesem Verzeichnis die Datei config.mk. Dabei handelt es sich um eine Datei, die Konfigurationen aller Projekte enthält, die mit dem Befehl make verwendet werden sollen. Unter anderem sind die make-Regeln mit den Befehlszeilenbefehlen und den verwendeten Optionen und Argumenten formuliert. Das Unterverzeichnis RingOscillator_PUF/ enthält die vier weiteren Verzeichnisse log, net, src und tcl sowie die eigentliche makefile-Datei. Im Unterverzeichnis net/ werden die Ergebnisse der Synthese in der Datei RO_PUF_synth.v gespeichert. Im Unterverzeichnis src/ befinden sich alle VHDL-Quellen sowie die CCF-Datei, mit der die verwendeten FPGA-Pins konfiguriert werden. Im Unterverzeichnis tcl/ befindet sich die Datei synth.tcl. In dieser Datei müssen unter anderem alle Module des Schaltungsentwurfs in einer bestimmten Form eingetragen werden. Im Makefile ist die Datei config.mk eingebettet, in der weitere projektspezifische Konfigurationen eingetragen werden.

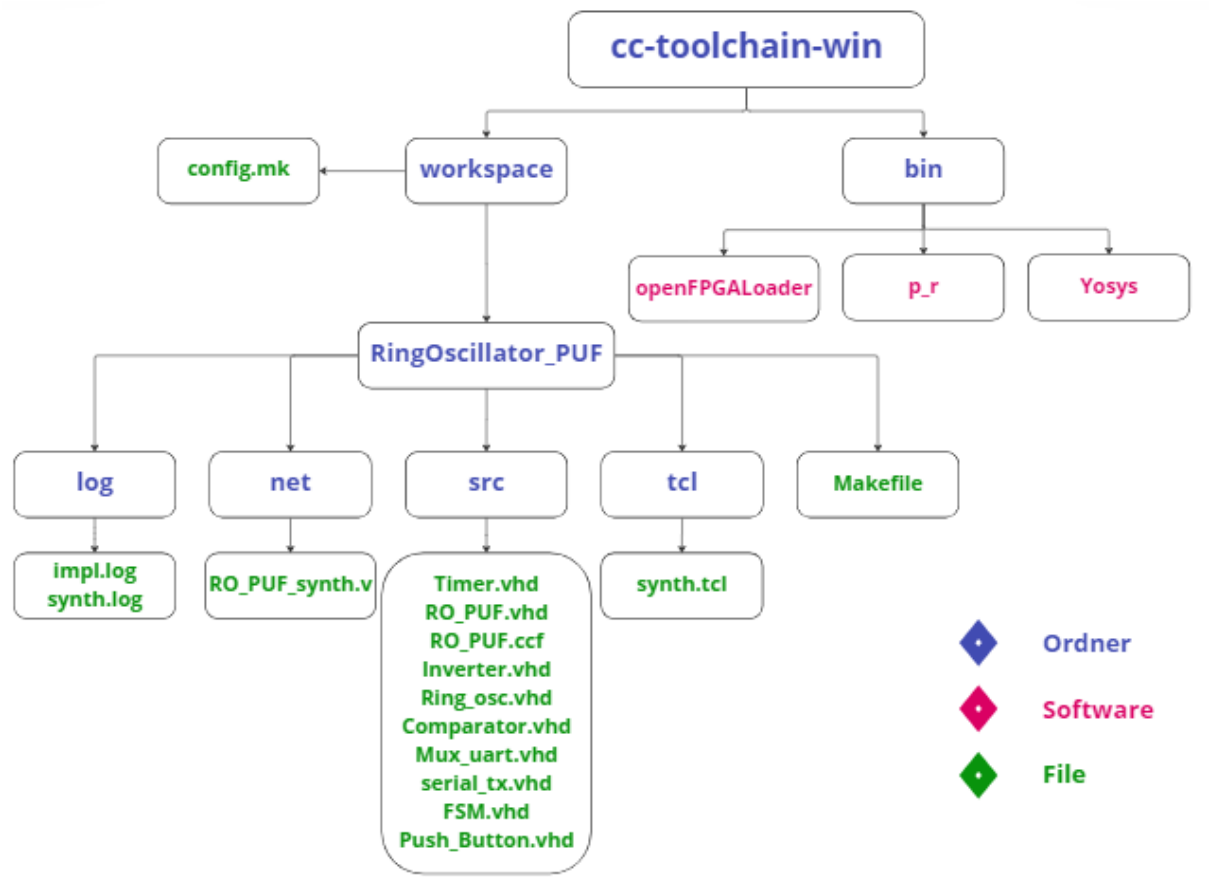


Abbildung 26: Baumstruktur des Unterverzeichnis cc-tool-win

6.3 Beschreibung des Designs der Schaltung

Das Ziel dieses Teils der Arbeit ist es, den Entwurf der Schaltung zu beschreiben, mit der die Komponenten des Ringoszillator parametrisiert instanziiert und eine Response aus dem Vergleich der Ringoszillator Frequenzen generiert und die Daten über eine UART-Schnittstelle gesendet werden können. Zunächst wird der Aufbau des Entwurfs anhand eines Blockdiagramms erläutert. Anschließend werden die einzelnen in VHDL entworfenen Module genauer beschrieben. Der vollständige VHDL-Quellcode ist im Anhang beigefügt.

6.3.1 Blockdiagramm

Die Struktur des Schaltungsentwurfs ist im Blockschaltbild in Abbildung 27 dargestellt. Der Entwurf besteht aus 10 Ringoszillatoren, die jeweils 15 Inverter enthalten. Die Ringoszillatoren haben Zählerausgänge, welche zu einem Komparator geführt werden, wo die Zählerstände miteinander verglichen werden. Nach dem Vergleich erzeugt der Komparator eine binäre Codeausgabe (Response), die aus 45 Bits besteht. Diese Response wird in der Multiplexer gegeben, um ihn in 6 Teile zu unterteilen, die jeweils mit dem UART-Modul übertragen werden, das 8 Bits bzw. 1 Byte an seinem Eingang einlesen kann. All dies wird von einer Finite-State-Machine konfiguriert, die das gesamte System steuert, indem sie alle Module durch entsprechende Steuersignale in der richtigen Reihenfolge aktiviert. Zunächst werden die Ringoszillatormodule aktiviert und die durch die Ringoszillatoren getakteten Zähler beginnen zu zählen. Nach einer gewissen Zeit beendet die FSM den Zählprozess, um mit Hilfe eines Komparators die Zählerstände zu vergleichen und daraus eine Response zu erstellen. Anschließend steuert die FSM das Multiplexer-Modul so an, dass die Response dem UART in 6 Teilen sequenziell übergeben wird. Der Prozess wird beendet, sobald die Response über die UART-Schnittstelle übertragen wurde und kann durch ein externes Signal erneut gestartet werden.

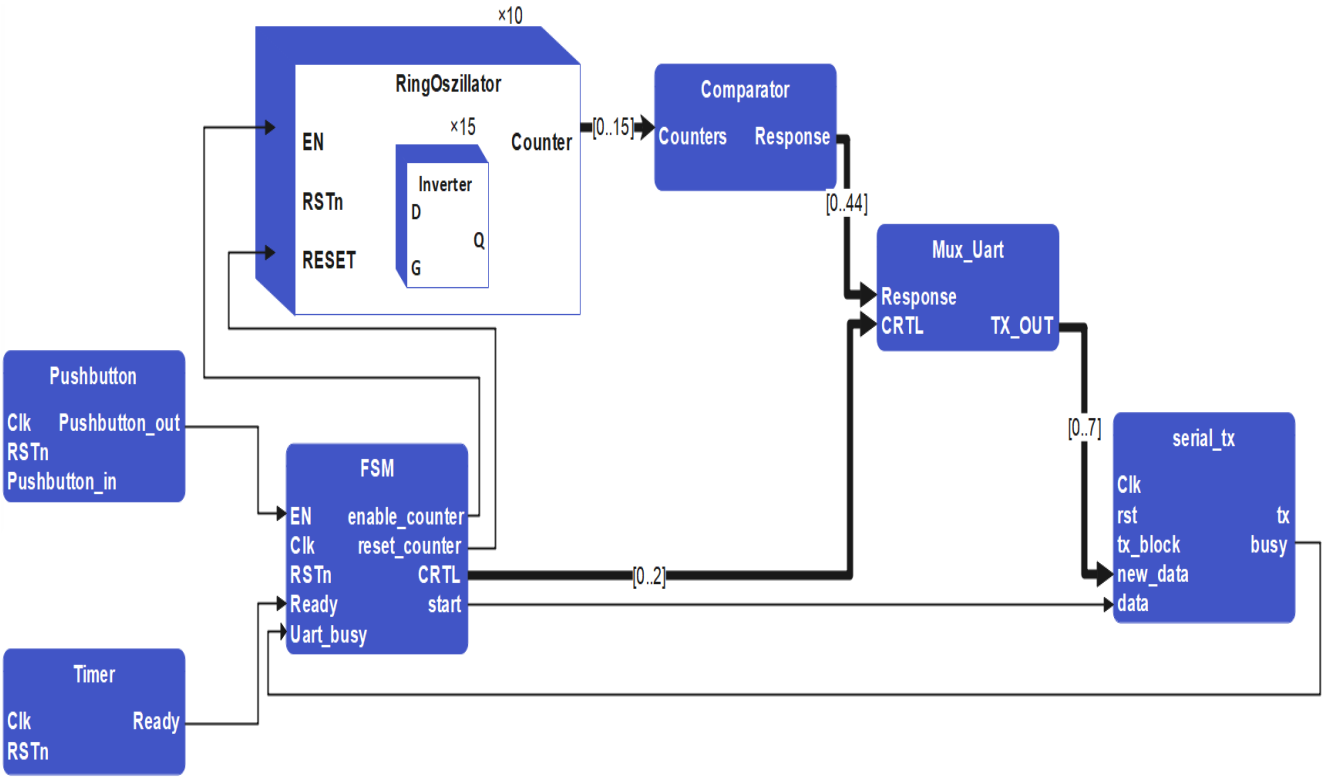


Abbildung 27: Blockdiagramm

6.3.2 Projektmodule

6.3.2.1 CC_DLT

Das CC_DLT ist ein D-Latch aus der Gatemate-Primitivenbibliothek. Die Invertierung der EN- und SR-Pins wird über Parameter gesteuert. Bis zu zwei Latches mit identischen Parametereinstellungen können in einem einzigen CPE kombiniert werden. Das Modul CC_DLT besteht aus drei Eingängen D, SR, G und einem Ausgang Q. In diesem Modul verhält sich der Eingang G wie ein Schalter, wenn G gleich '1' ist, erhält der Ausgang Q den Wert von D, und wenn G gleich '0' ist, behält der Ausgang Q unabhängig vom Dateneingang D seinen Wert bei. Der Eingang SR entspricht einem konfigurierbaren Set/Reset Signal, der das Latch asynchron zum Eingang G je nach Konfiguration entweder setzen oder zurücksetzen kann.

Eine Verilog-Implementierung dieses Moduls liegt in der Yosys-Technologiebibliothek des Gatemate-FPGAs vor. Diese Implementierung wird in diesem Projekt verwendet, um das Modul in Vivado simulieren zu können, obwohl die CC-Primitiven dort nicht standardmäßig verfügbar sind. [41]

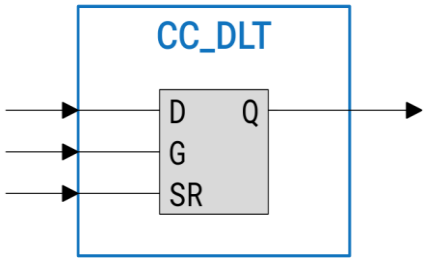


Abbildung 28: Modul CC_DLT

6.3.2.2 Inverter

Das Inverter Modul besteht aus einer CC_DLT Primitiven, die mit dem Eingang eines Nicht-Gates verbunden ist. Diese Reihenschaltung aus Latch und Inverter wird verwendet, um zu verhindern, dass die Synthese nachgeschaltete Inverter zur Optimierung der Signallaufzeit entfernt. Das Modul verfügt über zwei Eingängen D und G und einem Ausgang Q. Wenn der Wert von G gleich '1' ist, erhält der Ausgang Q den Wert des Eingang D und leitet diesen Wert invertiert auf den Ausgang des Moduls weiter. Nimmt der Eingang G des Latches den Wert Null an, wird die Latch-Funktion aktiviert und das Ausgangssignal ignoriert nachfolgende Änderungen am Eingangssignal D. Dementsprechend wird das Signal G im Ring-Oszillator verwendet, um die Oszillation zu aktivieren bzw. zu deaktivieren.

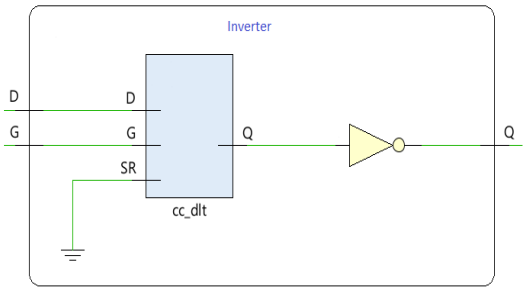


Abbildung 29: Modul Inverter

6.3.2.3 Ring_osc

Das Modul Ring_osc entspricht dem eigentlichen Ringoszillator-Entwurf, aus dem die PUF zusammengesetzt wird, die in diesem Projekt untersucht werden soll. Dieses Modul hat drei Eingänge und einen Ausgang.

Port-Name	Richtung	Funktion
EN	Eingang	aktiviert den Ringoszillator, wenn sein Wert gleich '1' ist
RESET	Eingang	synchroner Reset
RSTn	Eingang	asynchroner Reset
Counter	Ausgang	zählt die steigenden Flanken des letzten Invertersignals

Tabelle 6: Modul Eingänge und Ausgang

Dieses Modul enthält eine Anzahl von Invertern, die über einen Parameter data_width angepasst werden kann, wobei der Ausgang jedes Inverters mit dem Eingang des nächsten Inverters verbunden wird (Ausgang (i) => Eingang (i+1)) und der Ausgang des letzten Inverters mit dem Eingang des ersten Inverters verbunden wird. Für eine höhere Flexibilität wird der Ringoszillators mit einer konfigurierbaren Anzahl von Invertern über das generate statements erstellt.

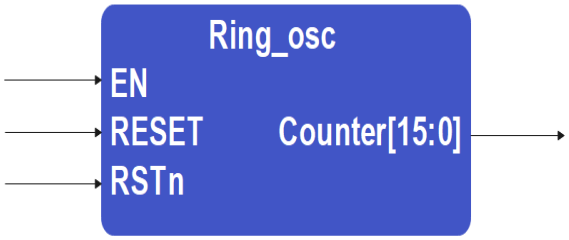


Abbildung 30: Modul Ring_osc

Der Ausgang des letzten Inverters fungiert als Taktgeber eines Zählers der bei jeder steigenden Flanke des Taktsignals seinen Zählerstand inkrementiert bzw. eine "1" zu seinem ursprünglichen Wert hinzuaddiert. Der Zähler enthält eine Anzahl von Bits, die über den Parameter cnt_width angepasst werden kann.

6.3.2.4 Comparator

Der Comparator enthält die Zählerstände der zehn Zähler als Eingang und generiert durch den Vergleich der Zählerstände eine Response als Ausgangssignal. Wenn der Wert des Zählers(i) größer oder gleich dem Wert des Zählers(j) ist, erhält das entsprechende Bit der Response den Wert '1', wenn nicht, erhält es den Wert '0'. Durch den Vergleich aller möglichen Zählerstandkombinationen entsteht als Ausgabe eine Response, welche aus 45 Bits besteht.

Der Vergleichsvorgang wird anhand der Vergleichsmatrix C_{10*10} durchgeführt, die in Abbildung 32 visualisiert wird, wobei die Felder $C_{i.j}$ der Matrix das Ergebnis des Vergleichs zwischen dem Zähler i und dem Zähler j enthalten. $C_{i.j}$ enthält die gleiche Information wie $C_{j.i}$, weshalb die Matrix nur für j > i ausgefüllt werden muss.

Dementsprechend besteht der Responsevektor aus den Elementen der Matrix, die sich oberhalb der Matrixdiagonalen befinden und in der Abbildung schraffiert dargestellt sind. Dadurch wird sichergestellt, dass jedes Zählerpaar nur einmal in das Ergebnis eingeht.

$$\text{Es gilt } C_{i,j} = \begin{cases} 1, wenn\ counter\ i > counter\ j \\ 0\ sonst \end{cases}$$

$C_{i,j}$ ist einzigartig für j ≥ i+1 und 0 ≤ i, j≤ 9

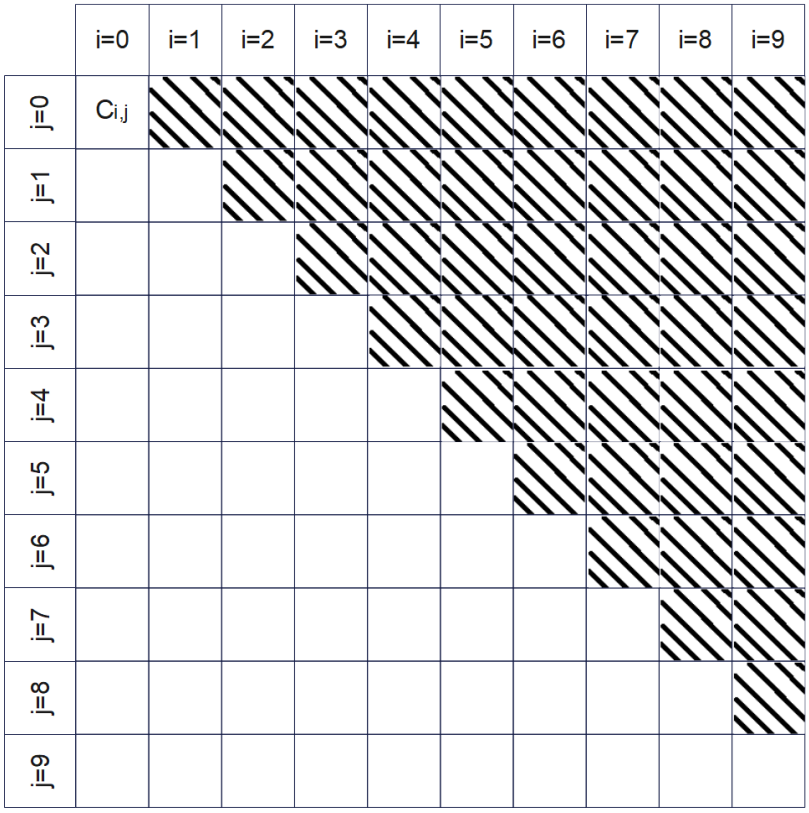


Abbildung 31: Compare-Matrix

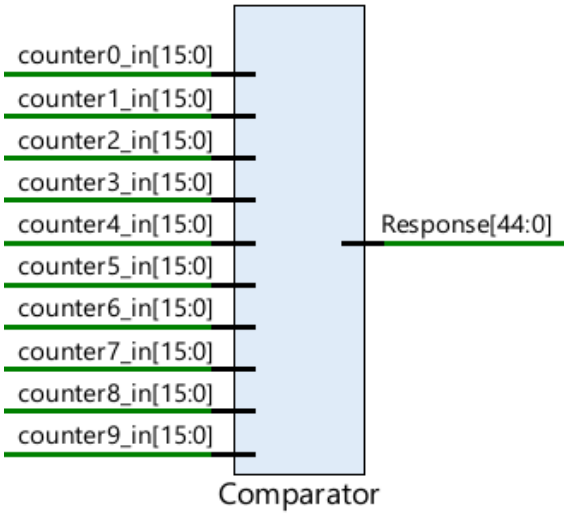


Abbildung 32: Modul Comparator

6.3.2.5 MUX_Uart

Das Modul MUX_Uart hat die Aufgabe, die 45-Bit- Response, die vom Komparator generiert wird und am Response -Eingang des Multiplexers anliegt, in 6 Teile zu partitionieren und diese über den TX_OUT-Port an den Eingang der UART-Schnittstelle weiterzuleiten. Jeder Teil enthält 8 Bits, da der UART nur diese Anzahl von Bits übertragen kann. Der Befehlseingang CRTL legt fest, welcher Teil der Response übertragen werden soll. Der CTRL-Anschluss hat drei Bits, der Wert "000" bewirkt, dass der erste Teil der Response [7:0] übertragen wird, mit dem Wert "001" werden die Bits [15:8] übertragen, mit dem Wert "010" wird der Teil [23:16] übertragen, mit dem Wert "011" wird der Teil [31:24] übertragen, mit dem Wert "100" wird der Teil [40:32] übertragen und mit dem Wert "101" wird der Teil [45:40] übertragen.

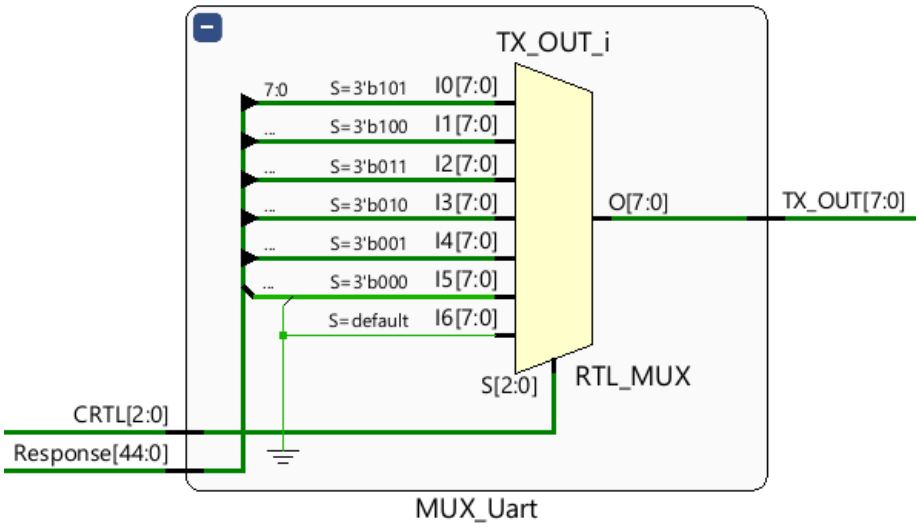


Abbildung 33: Modul MUX_Uart

6.3.2.6 Serial_tx

Das serial_tx-Modul führt das Senden von seriellen Daten nach dem UART-Standard aus. Es besteht aus einer Zustandsmaschine und einem Schieberegister. Da dieses Modul aus dem Mojo-Projekt übernommen wurde, wird es hier nicht näher beschrieben. Im Zusammenhang mit der Mojo-Board wird das Modul verwendet, um den Datenaustausch zwischen einem FPGA und einem Mikrocontroller auf dem Mojo-Board über eine UART-Schnittstelle zu implementieren. In diesem Projekt erhält das Modul serial_tx 8 Bits des Moduls MUX_Uart und überträgt die Daten über den Ausgangspin tx an das USBUART Modul, das über den PMOD-Anschluss mit der Gatemate-Evaluierungsplatine verbunden wurde. Das serial_tx-Modul besitzt fünf Eingänge clk, rst, tx_block, new_data und data und zwei Ausgänge tx und busy. Der Sendevorgang wird durch das Signal new_data eingeleitet. Solange das Signal busy gesetzt ist, ist das Modul mit dem Senden eines Bytes beschäftigt.

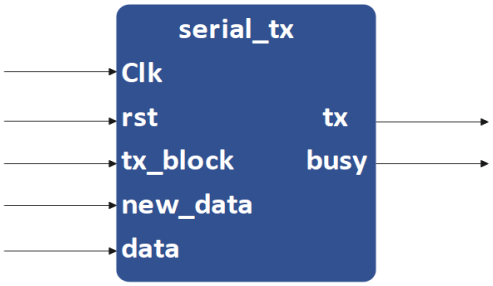


Abbildung 34: Modul serial_tx

6.3.2.7 Push_Button

Das Modul push_button wurde zur Entrprellung des Push-Taster-Signals erstellt und verfügt über drei Eingänge clk, push_button_in und RSTn sowie einen Ausgang push_button_out. Wenn der an das push_button_in-Signal angeschlossene Taster gedrückt wird, wird das push_button_out-Signal für mindestens 500 Zyklen unabhängig vom Zustand des push_button_in-Signals auf high gehalten.

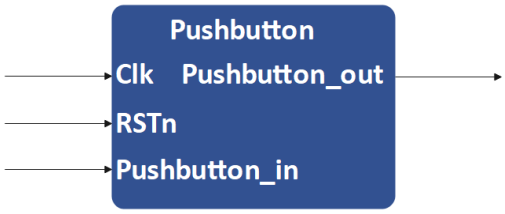


Abbildung 35: Modul Push_Button

6.3.2.8 Timer

Das Timer-Modul enthält zwei Eingänge clk, RSTn und einen Ausgang Ready. Das Modul entspricht einem Zähler, der seinen Zählerstand mit jedem Takt inkrementiert. Wenn der Zähler den Maximalwert erreicht, wird der Ausgang Ready aktiviert. Dieser Zähler enthält eine über einen Parameter cnt_clk angepasste Anzahl von Bits.

Abbildung 36: Modul Timer

6.3.2.9 FSM

Die Zustandsmaschine hat die Aufgabe, die Erstellung der Response und die Datenübertragung über die UART-Schnittstelle zu verwalten. Wie in Abbildung 32 dargestellt, wartet die Zustandsmaschine zunächst darauf, dass der Eingang "EN" aktiviert wird. Danach wird das Timer-Modul aktiviert, bis der Ready-Schalter gesetzt wird. Anschließend werden die Module MUX_Uart und serial_tx so gesteuert, dass alle sechs Teile der aktuellen Response übertragen werden. Mit Hilfe des Signals start_comb wird die Versendung eines Bytes über die UART-Schnittstelle ausgelöst. Die vollständige Versendung eines Bytes über die UART-Schnittstelle wird durch das Signal busy_uart signalisiert. Nach der Übertragung des sechsten Bytes geht die Zustandsmaschine in den Anfangszustand "Stop" über.

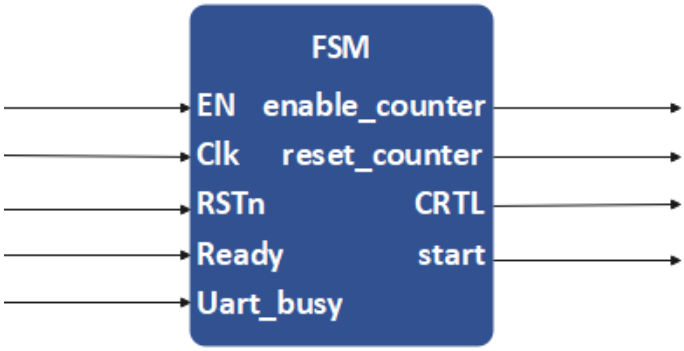


Abbildung 37: Modul FSM

Das FSM-Modul hat fünf Eingänge RSTn, EN, Ready, clk und Uart_busy und vier Ausgänge CRTL, start, enable_counter und reset_enable.

Das Zustandsdiagramm in Abbildung 32 stellt die Zustände der Zustandsmaschine grafisch dar. Der Anfangszustand ist Stop. Nach dem Drücken der Taste EN wechselt die Statusmaschine in den zweiten Zustand Count_time, wo durch das Signal Counter_enable die Ringoszillatoren und die dazugehörigen Zähler aktiviert werden. Gleichzeitig wird der Timer aktiviert, um die Ringoszillatoren für einen festen Zeitraum laufen zu lassen. Wenn das Signal Ready gesetzt ist, ist der Zeitraum abgelaufen und die Ringoszillatoren und Zähler werden deaktiviert. Dann findet der Vergleich der Zählerstände statt und die Versendung der Response beginnt. Dafür wechselt die Maschine in den Zustand Wait_Uart_1, in dem gewartet wird, bis das Signal busy_uart gleich null ist. Danach wechselt die Maschine in den Zustand Uart_1 und wartet, bis der erste Teil der Response (7:0) vollständig gesendet wurde. Anschließend geht die FSM in den Zustand Wait_Uart_2 über und wartet, bis das Signal busy_uart gleich null ist. Als nächstes geht die Zustandsmaschine in den Zustand Uart_2 über, und wartet, bis der nächste Teil der Response vollständig gesendet wurde (15:8). Dann erreicht die Zustandsmaschine den Zustand Wait_Uart_4. Wenn das Signal busy_uart gleich null ist, wechselt die Maschine in den Zustand Uart_4, um auf das vollständige Senden des nächsten Teils der Response zu warten (31:24). Im Anschluss geht die Zustandsmaschine in den Zustand Wait_Uart_5 über und wartet bis das Signal busy_uart gleich null ist, um in den Zustand Uart_5 zu wechseln. Dort wird die Übertragung des nächsten Teils der Response (39:32) ausgelöst. Dann wird Der Zustand Wait_Uart_6 angenommen und gewartet, bis das Signal busy_uart gleich null ist. Schließlich wechselt die Zustandsmaschine in den Zustand Uart_6 und löst die Versendung des letzten Teils der Response (44:40) aus und geht dann in den Ausgangszustand Stop über.

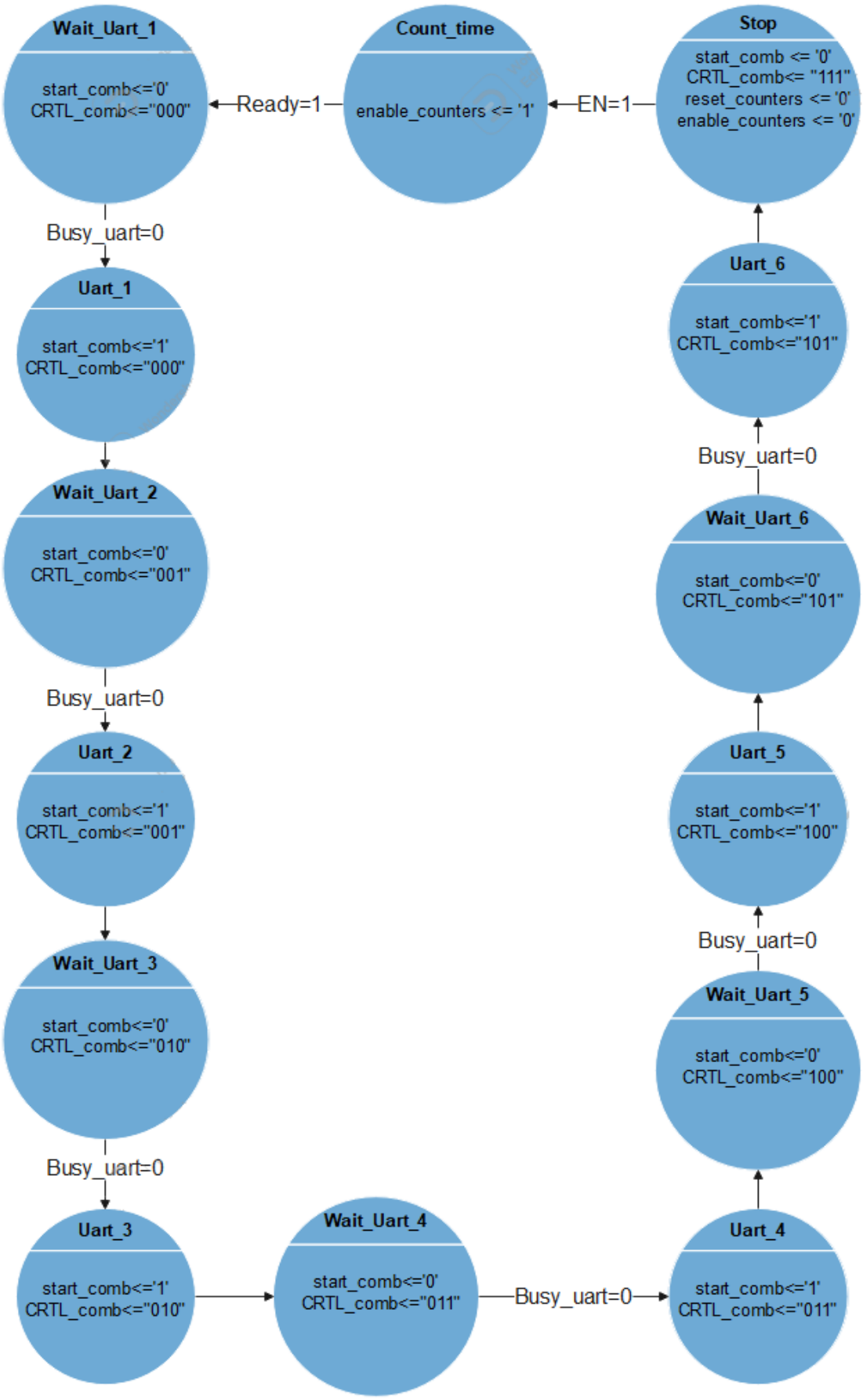


Abbildung 38: Zustandsdiagramm

6.3.2.10 RO_PUF

Der RO_PUF besteht aus den sechs Modulen Timer, Push_Button, FSM, MUX_Uart, Comparator und serial_tx und zehn Ring_osc Modulen. Das Modul hat die drei Eingänge clk_i, RSTn und Pushbutton und den Ausgang tx. Der Systemtakt wird an den Anschluss clk angelegt. RSTn entspricht einem asynchronen Reset mit geringer Aktivität und pushbutton entspricht dem gestörten Signal, mit dem der Benutzer die Erstellung der Response über einen Pushbutton starten kann. UART-Daten werden dann auf dem tx-Signal übertragen. Dieses Modul entspricht dem Top-Level-Modul des Schaltungsdesigns.

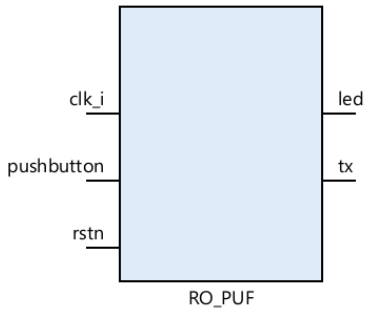


Abbildung 39: Modul RO_PUF

6.4 Versuchsbeschreibung

In diesem Abschnitt werden die Simulationsergebnisse, welche mit Hilfe der Software Vivado generiert wurden, vorgestellt. Nachdem der Entwurf synthetisiert, implementiert und über JTAG auf dem FPGA konfiguriert wurde, konnte eine Response generiert werden, die ebenfalls in diesem Kapitel vorgestellt wird.

6.4.1 Simulation

Da sich in der rein funktionalen Simulation keine prozessbedingten Variationen in Bezug auf die Frequenz der Ringoszillatoren einstellt, würden ohne zusätzliche Maßnahmen alle Zähler mit der gleichen Frequenz getaktet und damit auch die gleichen Zählerstände an das Comparator-Modul weitergeben. Aus diesem Grund wird das CC_DLT-Modul, welches die Funktion des verwendeten Latches modelliert, modifziert und eine parametrisierbare Ausgangsverzögerung eingeführt. (cc_dlt_dly.sv). Bei der Instanziierung der Ringoszillatoren in RO_PUF.vhd wird der Parameter für jede Instanz mit einem anderen Wert belegt, sodass sich unterschiedliche Oszillationsfrequenzen einstellen. Die in Abbildung 41 dargestellte Simulation beginnt nach Abschluss des Zeitraums, in dem die Ringoszillatoren aktiviert sind und die angeschlossenen Zähler takten. Zu Beginn ist zu erkennen, dass durch die Vergleiche im Komparator Block das Signal Response generiert wird. Diese Response wird Byte-Weise über das Signal TX_OUT an den UART übergeben und über das Signal tx seriell versendet.

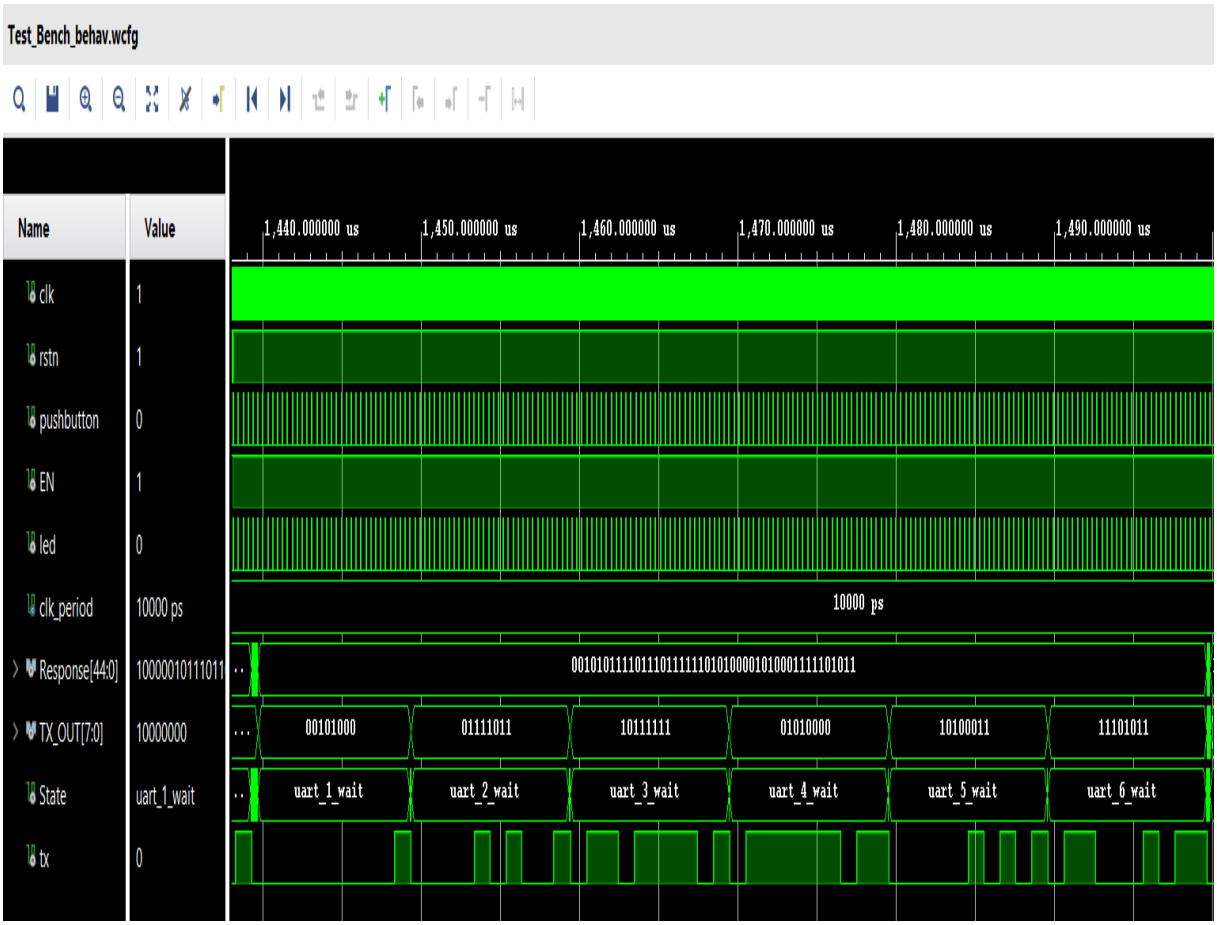


Abbildung 40: Simulation RO_PUF

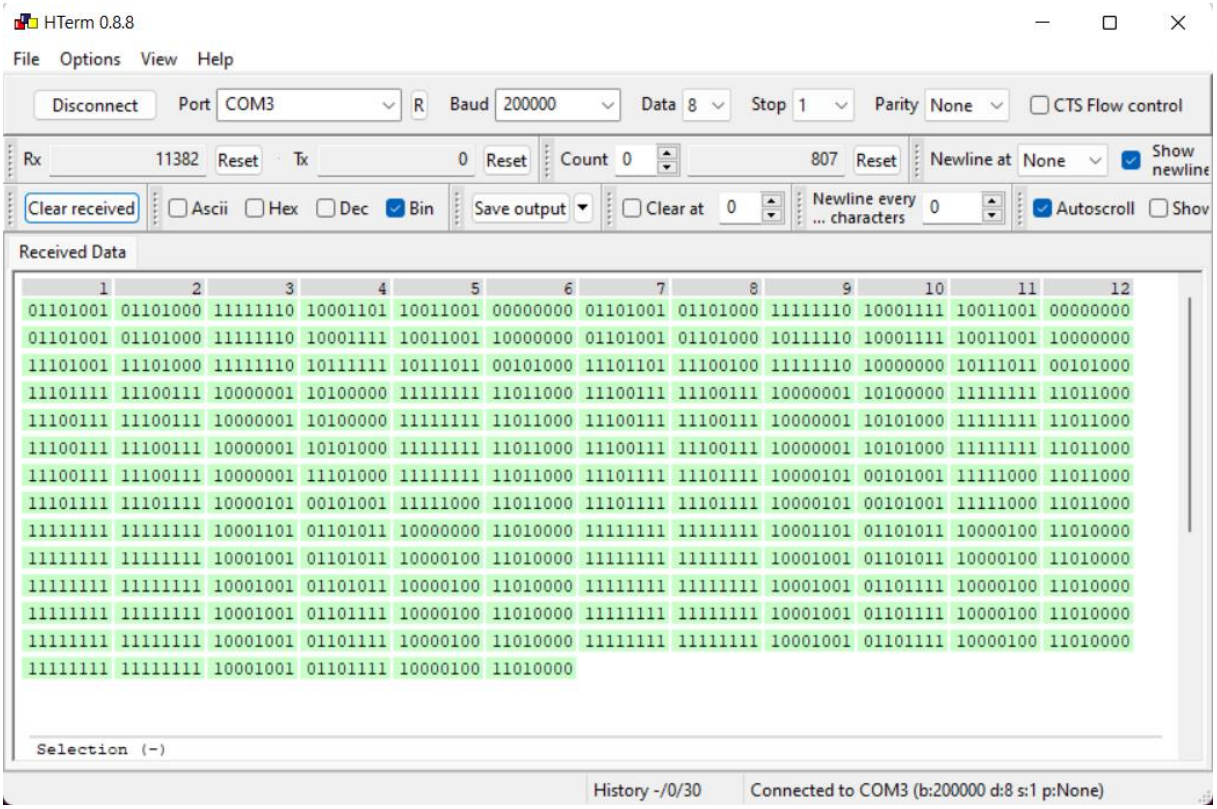


Abbildung 41: Die Anzeige des RO_PUF mit hterm

Die GateMate wurde mit dem vorgestellten Entwurf konfiguriert und über das USBUART PMOD Modul an den PC verbunden. Auf dem wird Hterm gestartet und auf eine Übertragungsrate von 200 Baud/s konfiguriert. Durch Druck auf einem der Taster auf dem Button PMOD Modul wird eine Response der Ringoszillator PUF, an den PC übermittelt und in Hterm dargestellt. In Abbildung 41 sind mehrere Responses der Ringoszillator PUF im Gatemate FPGA dargestellt.

7. Auswertung

Die entwickelte PUF wird in diesem Abschnitt charakterisiert, dabei werden die in Abschnitt 3.1 eingeführten Kriterien betrachtet.

7.1 Versuchsbeschreibung

Das PUF-Design wird synthetisiert und implementiert, es werden die folgenden Parameter verwendet:

- Anzahl der Inverter pro Ringoszillator: 15
- Breite der Zähler: 16 bit
- Dauer der Zählphase: 10 µs
- Intervall zwischen Abtastungen: 0,25 s

Der erstellte Bitstream wird nacheinander auf drei baugleiche Entwicklungsboards aufgespielt, die mit dem Versuchsrechner über die PMOD UART-USB-Brücke verbunden sind. Der PushButton, der die Abtastung der PUF und die Übertragung der Response auslöst, wird gedrückt gehalten, bis je Board 100 Responses empfangen wurden. Anschließend wird die Spannungsversorgung der Boards für etwa 10 Sekunden unterbrochen und der Versuch wiederholt, sodass schließlich für jede PUF-Instanz zweimal 100 Responses zur Auswertung vorliegen.

7.2 Ergebnisse

7.2.1 Zuverlässigkeit

Um die Reproduzierbarkeit der PUF-Response zu bewerten, werden die Compare-Matrizen aus den Response-Bitvektoren rekonstruiert.

Abbildung 46 zeigt die Häufigkeiten, mit denen einzelne Bits der Compare-Matrizen in den Versuchszyklen die Werte 0 oder 1 angenommen haben. Ein grünes Feld in Zeile a und Spalte b bedeutet demnach, dass der Zähler a immer größer gleich dem Zähler b ist, ein rotes Feld bedeutet das Gegenteil und ein blaues Feld bedeutet, dass beide Fälle aufgetreten sind. Für eine PUF mit perfekter Reproduzierbarkeit würden in dieser Darstellung nur grüne und rote Felder auftreten, je mehr blaue Felder und je näher die Auftrittshäufigkeiten dieser Bits bei 50% liegen, desto schlechter die Reproduzierbarkeit der PUF.

Der durchschnittliche Hamming Abstand zwischen zwei Antworten der PUF auf dem gleichen Board (Intra-Die) wird in Tabelle 7 dargestellt.

	Board 1	*Board 2*	*Board 3*
Hammingabstand	5,8	6,9	9,8

Tabelle 7: Durchschnittliche Intra-Die Hamming Distanzen der drei PUF-Instanzen

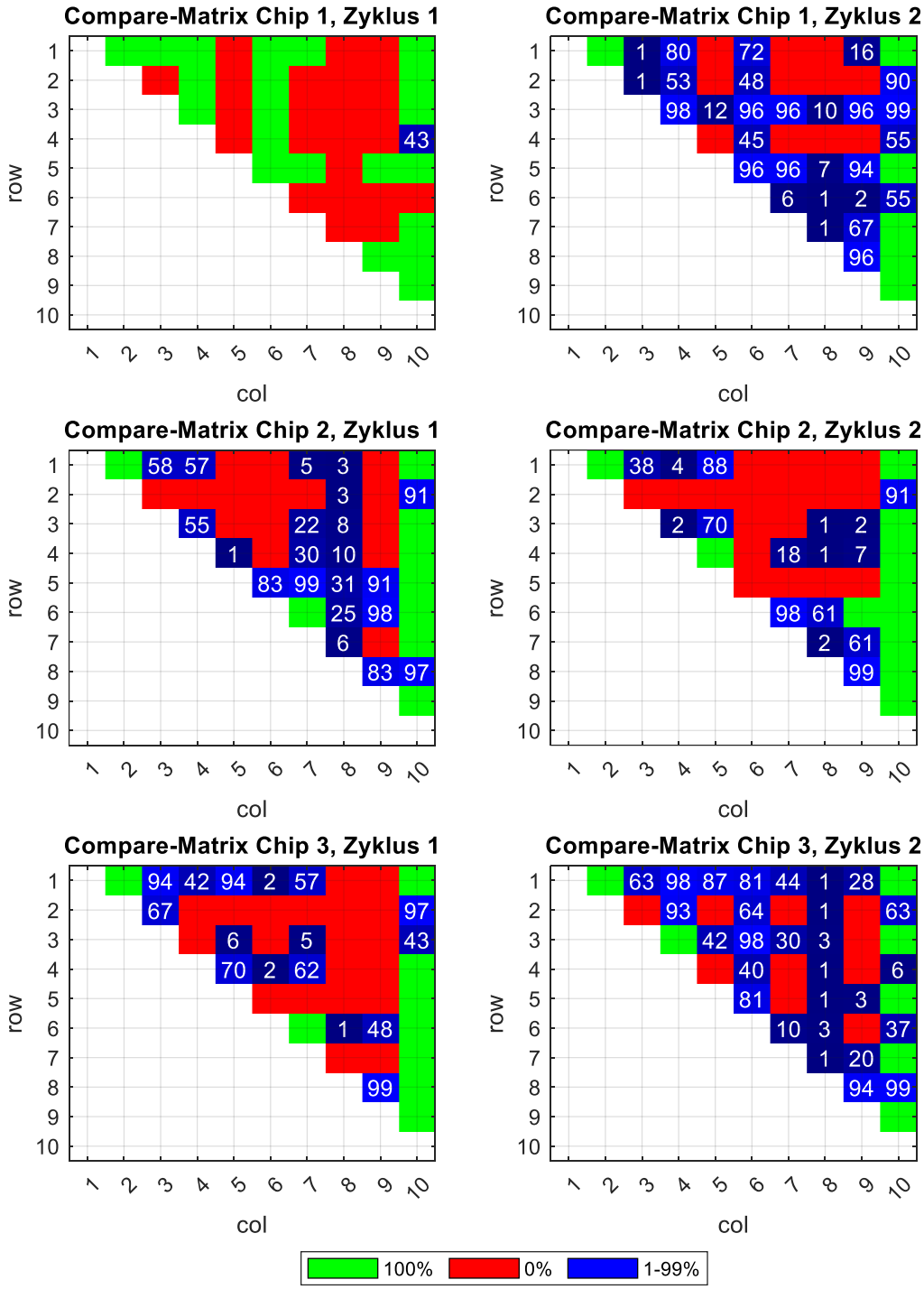


Abbildung 42: Compare-Matrizen der Versuche

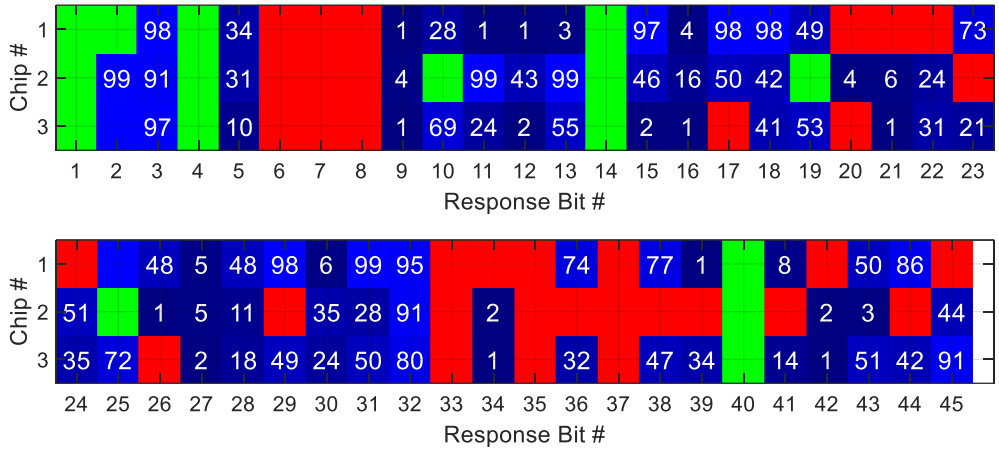
7.2.2 Eindeutigkeit

Um die Eindeutigkeit der PUF-Responses zu bestimmen, werden die durchschnittlichen Inter-Die Hamming Distanzen zwischen den drei Boards ermittelt und gemeinsam mit den Intra-Die-Distanzen in Tabelle 8 dargestellt.

	Board 1	*Board 2*	*Board 3*
Board 1	5,8	15,4	13,5
Board 2	15,4	6,9	11,9
Board 3	13,5	11,9	9,8

Tabelle 8: Durchschnittliche Intra- und Inter-Die Hamming Distanzen für den durchgeführten Versuch

Abbildung 47 gibt einen Überblick auf die Ähnlichkeiten und Unterschiede der einzelnen Bits in den PUF-Responses der drei Chips. Es ist zu erkennen, dass einzelne Bits (z.B. Positionen 1, 6, 7, 8, 33) in allen drei Instanzen immer den gleichen Zustand annehmen. Diese Bits sind unter dem Gesichtspunkt der Reprozierbarkeit vorteilhaft, unter dem Gesichtspunkt der Eindeutigkeit aber nicht, da sie nicht nur innerhalb einer Instanz, sondern über alle geprüften Instanzen ihren Wert beibehalten.



- Grün: immer 1
- Rot: immer 0
- Blau: Wahrscheinlichkeit für 1 als Zahlenwert angegeben

Abbildung 43: Auftrittswahrscheinlichkeiten für die logische 1 an den einzelnen Positionen in den Responses der drei Chips

7.2.3 Zusammenfassung

Der Vergleich der durchschnittlichen Intra- und Inter-Die Hamming Distanzen zeigt, dass sich zwei PUF-Antworten desselben Chips durchschnittlich mehr ähneln als zwei PUF-Antworten von unterschiedlichen Chips. Dies deutet darauf hin, dass Chip-spezifische Variationen im untersuchten Design einen größeren Einfluss auf die PUF-Response haben als die während des Versuchs auftretenden Effekte in den einzelnen Chips, die für die Intra-Die Variationen verantwortlich sind.

Tabelle 9 zeigt eine Übersicht über die minimal und maximal auftretenden Intra- und Inter-Die Hamming Distanzen. Es ist erkennbar, dass sich zwei Antworten unterschiedlicher Boards in Einzelfällen nur in einem Bit unterscheiden können (Board 2 – Board 3). Andererseits existieren auch Fälle, in denen sich zwei Antworten desselben Chips (Board 3) mehr unterscheiden als eine Antwort des Chips von einer Antwort eines anderen Chips (Board 3 – Board 1).

		Board 1	*Board 2*	*Board 3*
Board 1	max	22	27	26
	min	0	4	3
Board 2	max	27	19	32
	min	4	0	1
Board 3	max	26	32	28
	min	3	1	0

Tabelle 9: Maximale und Minimale Hamming Distanzen zwischen Antworten der drei PUF-Instanzen

Die Compare-Matrix, die in der 45 bit breiten PUF-Response enthalten ist, kodiert die Sortierungsreihenfolge der zehn Counter nach deren Frequenz. Die Menge der möglichen Responses hat damit $10! \approx 3{,}6 \cdot 10^6 \approx 2^{22}$ Elemente. Bei einer Beurteilung der PUF z.B. hinsichtlich ihrer Resilienz muss daher berücksichtigt werden, dass die 45 Reponsebits nicht statistisch unabhängig voneinander sind und tatsächlich nur etwa 22 bit nicht-redundanter Information kodieren. Dies ist auch bei der Bewertung von systematischen Einflüssen zu beachten: Wäre beispielsweise ein Bit der PUF-Response über alle Instanzen konstant, z.B. weil durch das Routing der Ringoszillatoren ein Oszillator immer schneller schwingt als ein anderer, so gehen tatsächlich knapp 5% statt nur 2,2% des Response-Raumes verloren.

8. Fazit

Das Ziel dieser Arbeit war die Implementierung einer PUF auf Basis von Ringoszillatoren in einem FPGA und unter Verwendung des Open-Source-Synthesewerkzeugs Yosys. In der Einleitung wurde ein Überblick über diese Arbeit und die Firma Cologne Chip gegeben. Im dritten Kapitel wurde der Begriff PUF definiert und die Funktionsweise von PUFs auf Basis des Ringoszillators erläutert. Anschließend wurden in den Kapiteln 4 und 5 die verwendete Hardware und Software vorgestellt. In Kapitel 6 wurden der Aufbau des Schaltungsdesigns, die Funktionsweise der einzelnen Module, Simulationsergebnisse und das Ergebnis eines grundlegenden Funktionstest vorgestellt. Schließlich wurde im letzten Kapitel Auswertung die Ringoszillator PUF im Gatemate FPGA hinsichtlich Reproduzierbarkeit und Eindeutigkeit untersucht. Es wurde gesehen, dass eine Untermenge der PUF-Response reproduzierbar ist. Hier ist zu untersuchen, ob durch eine optimale Wahl der Entwurfsparameter eine höhere Ausbeute erzielt werden kann. Des Weiteren muss der Einfluss von Umwelt- und Betriebsparametern auf die Reproduzierbarkeit untersucht werden. Außerdem wurde bei einem Vergleich von 3 Gatemate Bausteinen der Einfluss von Prozessvariationen sichtbar. Für eine bessere Bewertung der Eindeutigkeit ist die Untersuchung einer höheren Anzahl an Gatemate Bausteinen notwendig.

9. Literaturverzeichnis

[1] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, "Silicon physical random functions," in Proceedings of the 9th ACM conference on Computer and communications security, 2002, pp. 148-160.

[2] J. W. Lee, D. Lim, B. Gassend, G. E. Suh, M. Van Dijk, and S. Devadas, "A technique to build a secret key in integrated circuits for identification and authentication applications," in VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on, 2004, pp. 176-179.

[3] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in Proceedings of the 44th annual Design Automation Conference, 2007, pp. 9-14.

[4] S. Drimer, "Volatile FPGA design security–a survey," IEEE Computer Society Annual Volume, pp. 292-297, 2008.

[5] A. Maiti and P. Schaumont, "Improved ring oscillator PUF: an FPGA-friendly secure primitive," Journal of cryptology, vol. 24, pp. 375-397, 2011.

[6] R. Maes and I. Verbauwhede, "Physically unclonable functions: A study on the state of the art and future research directions," in Towards Hardware-Intrinsic Security, ed: Springer, 2010, pp. 3-37.

[7] R. S. Pappu, "Physical one-way functions," Massachusetts Institute of Technology, 2001.

[8] P. Tuyls, G.-J. Schrijen, B. Škorić, J. van Geloven, N. Verhaegh, and R. Wolters, "Read-proof hardware from protective coatings," in Cryptographic Hardware and Embedded Systems-CHES 2006, ed: Springer, 2006, pp. 369-383.

[9] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Physical unclonable functions and public-key crypto for FPGA IP protection," in Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on, 2007, pp. 189-195.

[10] J. Guajardo, S. Kumar, G.-J. Schrijen, and P. Tuyls, "FPGA intrinsic PUFs and their use for IP protection," Cryptographic Hardware and Embedded SystemsCHES 2007, pp. 63-80, 2007.

[11] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in Proceedings of the 44th annual Design Automation Conference, 2007, pp. 9-14.

[12] A. Maiti and P. Schaumont, "Improving the quality of a physical unclonable function using configurable ring oscillators," in Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, 2009, pp. 703-707.

[13] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 12 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[14] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S13 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[15] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 16 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[16] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 15 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[17] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 21 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[18] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 23 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[19] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 28 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[20] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 29 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[21] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 30 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[22] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 32 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[23] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 34 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[24] 1. Physically Unclonable Functions: a Study on the State of the Art and Future Research Direc-tions. Roel Maes, Ingrid Verbauwhede. Seite 17.

[25] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 27 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[26] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 35 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[27] Cologne Chip. (April 2022) Köln. GateMateTM FPGA Evaluation Board and Schematics S 36 https://colognechip.com/docs/ds1003-gatemate1-evalboard-3v1-latest.pdf

[28] Pmod USBUART: USB-zu-UART-Schnittstelle, Digilent A National Instruments Company https://shop.trenz-electronic.de/de/24242-Pmod-USBUART-USB-zu-UART-Schnittstelle4

[29] Uart https://fr.wikipedia.org/wiki/UART

[30] Pmod BTN: 4 User Pushbuttons https://digilent.com/shop/pmod-btn-4-user-pushbuttons/

[31] Very High-Speed Integrated Circuit Hardware Description Language https://de.wikipedia.org/wiki/Very_High_Speed_Integrated_Circuit_Hardware_Description_Language

[32] Ubuntu https://en.wikipedia.org/wiki/Ubuntu

[33] HTerm https://www.heise.de/download/product/hterm-53283

[34] Vivado Simulation https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2022_ 1/u g937 -vivado-design-suite-simulation-tutorial.pdf

[35] Yosys https://fr.wikipedia.org/wiki/Yosys

[36] GHDL http://ghdl.free.fr/site/pmwiki.php?n=Main.UserGuide

[37] Place and route https://colognechip.com/docs/ds1001-gatemate1-datasheet-latest.pdf S38

[38] openFPGALoader https://github.com/trabucayre/openFPGALoader

[39] zadig https://zadig.akeo.ie/

[40] Gatemate1-Datasheet-Latest, 22.Mai.2022 https://colognechip.com/docs/ds1001-gatemate1-datasheet-latest.pdf

[41] CC_DLT Primitive https://www.colognechip.com/docs/ug1001-gatemate1-primitives-library-latest.pdf

10. Code-Anhang

- **CC_DLT**

```
`TIMESCALE 1PS / 1PS

MODULE CC_DLT #(
	PARAMETER [0:0] G_INV = 1'B0,
	PARAMETER [0:0] SR_INV = 1'B0,
	PARAMETER [0:0] SR_VAL = 1'B0,
	PARAMETER DELAY_PS
)(
	INPUT D,
	INPUT G,
	INPUT SR,
	OUTPUT REG Q
);
	WIRE EN, SR;
	ASSIGN EN = (G_INV)? ~G: G;
	ASSIGN SR = (SR_INV)? ~SR: SR;

	INITIAL Q = 1'BX;

	ALWAYS @ (*)
	BEGIN
		IF (SR) BEGIN
			Q <= #(DELAY_PS) SR_VAL;
		END
		ELSE IF (EN) BEGIN
			Q <= #(DELAY_PS) D;
		END
	END

ENDMODULE
```

- **Inverter**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity inverter is
  generic(delay_ps: integer);
  Port (D:in std_logic;
        G:in std_logic;
        Q:out std_logic );
end inverter;

architecture Behavioral of inverter is

component CC_DLT is
 generic (delay_ps: integer);
 port (D : in std_logic;
          G : in std_logic;
          SR: in std_logic;
          Q : out std_logic);
end component;

signal out_dlt,out_nox : std_logic;

begin
u: CC_DLT generic map(delay_ps => delay_ps) port map (D=>d,G=>g,SR=>'0',Q=>out_dlt);

process (out_dlt, out_nox) begin
   if out_dlt = 'X' or out_dlt = 'U' then
      out_nox <= '0';
   else
      out_nox <= out_dlt;
   end if;
   q <= not out_nox  ;

end process;

end Behavioral;
```

- **Ring_osc**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity Ring is

   generic (data_width : integer :=15;
            cnt_width: integer:=16;
            inv_delay_ps: integer:=100);

   Port (       En    : in  STD_LOGIC;
                RSTn   : in  STD_LOGIC;
                RESET  : in  STD_LOGIC;
                counter: out std_logic_vector(cnt_width - 1 downto 0)
            );
end Ring;

architecture Behavioral of Ring is

component inverter is
            generic (delay_ps: integer);
              port (     D: in std_logic;
                         G: in std_logic;
                         Q: out std_logic);
end component ;

signal counter_reading  : unsigned(cnt_width - 1 downto 0):=(others=>'0');
signal chain_in :std_logic_vector(data_width downto 0);
signal chain_out :std_logic_vector(data_width - 1 downto 0);
signal loc_clk : std_logic;
```

```
begin

 gen_chain:
for i in 0 to data_width -1 generate

  begin

   u1:inverter

   generic map(delay_ps => inv_delay_ps)

    port map

    (       Q =>  chain_out(i),
            D => chain_in(i),
            G => EN);
            chain_in(i+1)<=chain_out(i);

 end generate;

            chain_in(0) <= chain_out(data_width-1);
            loc_clk <= chain_out(data_width-1);

count: process (RSTn,loc_clk)

begin

if RSTn = '0' then
            counter_reading <= (others => '0');
elsif (rising_edge(loc_clk)) then
            if RESET = '1' then
                        counter_reading <= (others=>'0');
            elsif EN='1' then
                        counter_reading<= counter_reading +1;

   end if;
end if;
end process count;

counter<=std_logic_vector(counter_reading);

end Behavioral;
```

- **Comparator**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Comparator is

  generic (
              cnt_width      : integer := 16
            );

  Port (

        counter0_in    : in std_logic_vector(cnt_width - 1 downto 0);
        counter1_in    : in std_logic_vector(cnt_width - 1 downto 0);
        counter2_in    : in std_logic_vector(cnt_width - 1 downto 0);
        counter3_in    : in std_logic_vector(cnt_width - 1 downto 0);
        counter4_in    : in std_logic_vector(cnt_width - 1 downto 0);
        counter5_in    : in std_logic_vector(cnt_width - 1 downto 0);
        counter6_in    : in std_logic_vector(cnt_width - 1 downto 0);
        counter7_in    : in std_logic_vector(cnt_width - 1 downto 0);
        counter8_in    : in std_logic_vector(cnt_width - 1 downto 0);
        counter9_in    : in std_logic_vector(cnt_width - 1 downto 0);
        Response       : out std_logic_vector(44 downto 0)

  );

end Comparator;


architecture Behavioral of Comparator is

type counters_t is array (0 to 9) of unsigned (cnt_width - 1 DOWNTO 0);
signal counters : counters_t;
signal compare_matrix : std_logic_vector (99 downto 0);

begin

  comparator_proc : process(counter0_in,counter1_in,counter2_in,counter3_in,counter4_in,
                            counter5_in,counter6_in,counter7_in,counter8_in,counter9_in)

  begin

        Response <= (others => '0');
        counters <= (others => (others => '0'));
        counters(0) <= unsigned(counter0_in);
```

```
            counters(1) <= unsigned(counter1_in);
            counters(2) <= unsigned(counter2_in);
            counters(3) <= unsigned(counter3_in);
            counters(4) <= unsigned(counter4_in);
            counters(5) <= unsigned(counter5_in);
            counters(6) <= unsigned(counter6_in);
            counters(7) <= unsigned(counter7_in);
            counters(8) <= unsigned(counter8_in);
            counters(9) <= unsigned(counter9_in);

        compare_matrix <= (others => '0');

        for row in 0 to 9 loop

           for col in row + 1 to 9 loop

              if counters(row) >= counters(col) then

                 compare_matrix(row * 10 + col) <= '1';

              end if ;

           end loop;

        end loop;

Response <= (others => '0');
Response(8  downto 0)   <= compare_matrix(9  downto 1);
Response(16 downto 9)   <= compare_matrix(19 downto 12);
Response(23 downto 17)  <= compare_matrix(29 downto 23);
Response(29 downto 24)  <= compare_matrix(39 downto 34);
Response(34 downto 30)  <= compare_matrix(49 downto 45);
Response(38 downto 35)  <= compare_matrix(59 downto 56);
Response(41 downto 39)  <= compare_matrix(69 downto 67);
Response(43 downto 42)  <= compare_matrix(79 downto 78);
Response(44)            <= compare_matrix(89);

    end process;

end Behavioral;
```

- **MUX_Uart**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MUX_Uart is

port(
Response: in std_logic_vector (44 downto 0);
TX_OUT : out std_logic_vector (7 downto 0);
CRTL : in std_logic_vector(2 downto 0)
);

end MUX_Uart;

architecture rtl of MUX_Uart is

begin

p_MUX_Uart : process(CRTL,Response)

begin

case CRTL is

when "101" => TX_OUT <= Response(7 downto 0);
when "100" => TX_OUT <= Response(15 downto 8);
when "011" => TX_OUT <= Response(23 downto 16);
when "010" => TX_OUT <= Response(31 downto 24);
when "001" => TX_OUT <= Response(39 downto 32);
when "000" => TX_OUT <= Response(44 downto 40)&"000" ;
when others => TX_OUT <= "00000000";

end case;

end process p_MUX_Uart;

end rtl;
```

- **serial_tx**

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity serial_tx is

	generic (
		CLK_PER_BIT	: natural := 50;
		CTR_SIZE : natural := 6
		);

	port (
		clk				: in  std_logic;
		rst				: in  std_logic;
		tx				: out std_logic;
		tx_block  : in  std_logic;
		busy			: out std_logic;
		new_data		: in  std_logic;
		data			: in  std_logic_vector(7 downto 0)
		);

end entity serial_tx;

architecture RTL of serial_tx is

type tx_state_type is (IDLE, START_BIT, DATA_BITS, STOP_BIT);

signal state_d, state_q			: tx_state_type;
signal ctr_d, ctr_q			: unsigned(CTR_SIZE-1 downto 0);
signal bit_ctr_d, bit_ctr_q	: unsigned(2 downto 0);
signal data_d, data_q			: std_logic_vector(7 downto 0);
signal tx_d, tx_q			: std_logic;
signal busy_r				: std_logic;
signal block_d, block_q			: std_logic;

Begin
	tx		<= tx_q;
	busy	<= busy_r;

tx_comb: process(tx_block, block_q, ctr_q, bit_ctr_q, data_q, state_q, data, new_data)

begin

	block_d	<= tx_block;
	ctr_d		<= ctr_q;
	bit_ctr_d	<= bit_ctr_q;
	data_d		<= data_q;
	state_d	<= state_q;
	tx_d		<='0';
	busy_r <= '0';

	case state_q is
		when IDLE =>
			if block_q = '1' then
				busy_r		<= '1';
				tx_d		<= '1';
			else
```

```
                busy_r          <=      '0';
                tx_d            <=      '1';
                bit_ctr_d <= "000";
                ctr_d           <=      (others => '0');
                if new_data = '1' then
                        data_d          <= data;
                        state_d   <= START_BIT;
                        busy_r          <= '1';
                end if;
            end if;
        when START_BIT =>
            busy_r    <= '1';
            ctr_d     <= ctr_q + 1;
            tx_d      <= '0';
            if ctr_q = (CLK_PER_BIT-1) then
                ctr_d     <= (others => '0');
                state_d   <= DATA_BITS;
            end if;

        when DATA_BITS =>
            busy_r    <= '1';
            tx_d      <= data_q(0);
            ctr_d     <= ctr_q + 1;
            if ctr_q = (CLK_PER_BIT-1) then
                data_d          <= "0" & data_q(7 downto 1);
                ctr_d           <= (others => '0');
                bit_ctr_d <= bit_ctr_q + 1;
                if bit_ctr_q = 7 then
                        state_d   <= STOP_BIT;
                end if;
            end if;
        when STOP_BIT =>
            busy_r    <= '1';
            tx_d      <= '1';
            ctr_d     <= ctr_q + 1;
            if ctr_q = (CLK_PER_BIT-1) then
                state_d   <= IDLE;
            end if;
        when others =>
            state_d   <= IDLE;
    end case;

end process tx_comb;

tx_seq: process(clk, rst)

begin
    if rst = '1' then
        state_q   <= IDLE;
        tx_q      <= '1';
        --busy    <= '0';
    elsif rising_edge(clk) then
        state_q   <= state_d;
        tx_q      <= tx_d;
        --busy    <= busy_r;
        block_q         <= block_d;
        data_q          <= data_d;
        bit_ctr_q <= bit_ctr_d;
        ctr_q           <= ctr_d;
    end if;

end process tx_seq;

end RTL;
```

- **FSM**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity FSM is

   Port (
            EN    : in std_logic;
            clk   : in std_logic;
            RSTn  : in std_logic;
            Ready : in std_logic;
            enable_counters     : out std_logic;
            reset_counters      : out std_logic;
            uart_busy: in std_logic;
            CRTL : out std_logic_vector(2 downto 0);
            start : out std_logic
            );
end FSM;

architecture Behavioral of FSM is

type t_State is (stop,count_time,uart_1,uart_1_wait, uart_2,uart_2_wait,uart_3,
            uart_3_wait,uart_4,uart_4_wait,uart_5,uart_5_wait,uart_6_wait,uart_6);

signal State,next_state : t_State;
signal start_comb: std_logic;
signal stop_cnt: std_logic;
signal crtl_comb: std_logic_vector(2 downto 0);
```

```
begin

 process (clk,RSTn)

   begin
            if RSTn='0' then
            State <= stop;
            start <= '0';
            CRTL <= "000";
            elsif (rising_edge(clk)) then
                        State <= next_state;
                        start <= start_comb;
                        CRTL <= crtl_comb;
            end if;
end process;

 FSM_proc :process (State,EN,Ready,uart_busy)

Begin
case State is
             when Stop =>
                        if EN='1' then
                                    next_state<=count_time;
                        else
                                    next_state<=stop;
                        end if;
            when count_time =>
                        if Ready='1' then
                                    next_state<=uart_1_wait;
                        else
                                    next_state<=count_time;
                        end if;
```

```
 when uart_1_wait =>
        if uart_busy = '0' then
                next_state<= uart_1;
        else
                next_state <= uart_1_wait;
        end if;

when uart_1 =>
        next_state<= uart_2_wait;

when uart_2_wait =>
        if uart_busy = '0' then
                next_state<= uart_2;
        else
                next_state <= uart_2_wait;
        end if;

when uart_2 =>
        next_state<= uart_3_wait;

 when uart_3_wait =>
        if uart_busy = '0' then
                next_state<= uart_3;
        else
                next_state <= uart_3_wait;
        end if;

when uart_3 =>
        next_state<= uart_4_wait;

 when uart_4_wait =>
         if uart_busy = '0' then
                next_state<= uart_4;
        else
                next_state <= uart_4_wait;
        end if;
```

```
    when uart_4 =>
            next_state<= uart_5_wait;

    when uart_5_wait =>
            if uart_busy = '0' then
                    next_state<= uart_5;
            else
                    next_state <= uart_5_wait;
            end if;

    when uart_5 =>
            next_state<=uart_6_wait;

   when uart_6_wait =>
            if uart_busy = '0' then
                    next_state<= uart_6;
            else
                    next_state <= uart_6_wait;
            end if;

   when uart_6 =>
            next_state<=stop;
end case;
end process;

process(State)

   begin
            start_comb <= '0';
            CRTL_comb<= "111";
            reset_counters <= '0';
            enable_counters <= '0';
```

```
case State is

   when count_time =>
         enable_counters <= '1';

   when uart_1 =>
         start_comb<='1';
         CRTL_comb<="000";

   when uart_1_wait =>
         start_comb<='0';
         CRTL_comb<="000";

   when uart_2 =>
         start_comb<='1';
         CRTL_comb<="001";

   when uart_2_wait =>
         start_comb<='0';
         CRTL_comb<="001";

   when uart_3 =>
         start_comb<='1';
         CRTL_comb<="010";

   when uart_3_wait =>
         start_comb<='0';
         CRTL_comb<="010";

   when uart_4 =>
         start_comb<='1';
         CRTL_comb<="011";

   when uart_4_wait =>
         start_comb<='0';
         CRTL_comb<="011";
```

```
        when uart_5 =>
                start_comb<='1';
                CRTL_comb<="100";

        when uart_5_wait =>
                start_comb<='0';
                CRTL_comb<="100";

        when uart_6 =>
                start_comb<='1';
                reset_counters <= '1';
                CRTL_comb<="101";

         when uart_6_wait =>
                start_comb<='0';
                CRTL_comb<="101";

        when others =>
                start_comb <= '0';
                CRTL_comb<= "111";
                stop_cnt<='0';
end case;
end process;
end Behavioral;
```

- **Timer**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Timer is
generic (cnt_clk:    integer:=31);
Port (
        RSTn: in  STD_LOGIC;
        clk : in std_logic;
        Ready: out std_logic);
end Timer;

architecture Behavioral of Timer is
signal clk_counter : unsigned(0 to cnt_clk):=(others=>'0');

begin
process(clk, RSTn)
begin
        if (RSTn = '0') then
                clk_counter <= (others => '0');
        else
        if (rising_edge(clk)) then
                if clk_counter = 30 then
                        Ready <= '1';
                        clk_counter <= (others => '0');
                else
                        clk_counter <= clk_counter + 1;
                        Ready <= '0';
                end if;
        end if;
        end if;
end process;
end Behavioral;
```

- **Push_Button**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Push_Button is

 generic ( top : natural := 10000000 );

 Port (
          clk           : in std_logic;
          push_button_in  : in std_logic;
          RSTn          : in std_logic;
          push_button_out : out std_logic
          );

end Push_Button;

architecture Behavioral of Push_Button is

signal counter_up: integer range 0 to top;
signal s_push_button_out:std_logic;

begin

   push_button_out<=s_push_button_out;
```

```
process(clk,RSTn)
begin
	if RSTn = '0' then
		counter_up <= 0;
		s_push_button_out <= '0';
	elsif (rising_edge(clk)) then
		s_push_button_out <= '0';
	if counter_up = 0 then
		if push_button_in = '1' then
			s_push_button_out <= '1';
			counter_up <= 1;
		end if;
	else
		counter_up <= counter_up + 1;
	if counter_up = top then
		counter_up <= 0;
	end if;
	end if;
	end if;
end process;
end Behavioral;
```

- **RO_PUF**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity RO_PUF is

   generic (
            data_width : integer := 15;
            cnt_width  : integer := 16;
            cnt_clk    : integer := 8
            );

   Port (
            rstn          : in STD_LOGIC;
            pushbutton    : in std_logic;
            clk_i         : in std_logic;
            tx            : out STD_LOGIC;
            led           : out std_logic
            );

end RO_PUF;

architecture Behavioral of RO_PUF is

component Ring is

   generic (
            data_width    : integer := 15;
            cnt_width     : integer := cnt_width;
            inv_delay_ps  : integer
            );

   Port (
            RSTn          : in  STD_LOGIC;
            RESET         : in  STD_LOGIC;
            EN            : in std_logic;
            counter       : out std_logic_vector(cnt_width - 1 downto 0)
            );

end component;
```

```
component Comparator is

   generic (
            cnt_width   : integer := cnt_width
            );

   Port (
            counter0_in,
            counter1_in,
            counter2_in,
            counter3_in,
            counter4_in,
            counter5_in,
            counter6_in,
            counter7_in,
            counter8_in,
            counter9_in    : in std_logic_vector(cnt_width - 1 downto 0);
            Response      : out std_logic_vector(44 downto 0)
            );

end component;

component FSM is

   Port (
            EN         : in std_logic ;
            clk        : in std_logic ;
            RSTn       : in std_logic ;
            uart_busy   : in std_logic;
            Ready      : in std_logic ;
            enable_counters  : out std_logic;
            reset_counters     : out std_logic;
            start      : out std_logic;
            CRTL       : out std_logic_vector(2 downto 0)
            );

end component;
```

```
component Timer is
generic (
        cnt_clk       : integer := 31
        );
Port (
        RSTn          : in STD_LOGIC;
        clk           : in std_logic;
        Ready         : out std_logic
        );
end component;

component serial_tx is
generic (
        CLK_PER_BIT    : natural := 50;
        CTR_SIZE       : natural := 6
        );
Port (
        clk       : in  std_logic;
        rst       : in  std_logic;
        tx        : out std_logic;
        tx_block   : in  std_logic;
        busy      : out std_logic;
        new_data   : in  std_logic;
        data      : in  std_logic_vector(7 downto 0)
        );
end component;

component MUX_Uart is
Port (
        Response  : in std_logic_vector(44 downto 0);
        TX_OUT    : out std_logic_vector(7 downto 0);
        CRTL      : in std_logic_vector(2 downto 0)
        );
end component;

component Push_Button is
generic ( top : natural := 250000 );
Port (
        clk           : in std_logic;
        push_button_in  : in std_logic;
        RSTn          : in std_logic;
        push_button_out : out std_logic
        );
end component;
```

```
component Push_Button is

 generic ( top : natural := 250000 );

 Port (
            clk           : in std_logic;
            push_button_in  : in std_logic;
            RSTn          : in std_logic;
            push_button_out : out std_logic
            );

end component;

signal counter1,counter2,counter3,counter4,counter5,counter6,counter7,
            counter8,counter9,counter10:  std_logic_vector(cnt_width - 1 downto 0);
signal ready          : std_logic;
signal start          : std_logic;
signal enable_counters, reset_counters      : std_logic;
signal busy           : std_logic;
signal tx_block       : std_logic;
signal Response       : std_logic_vector(44 downto 0);
signal TX_OUT         : std_logic_vector(7 downto 0);
signal CRTL           : std_logic_vector(2 downto 0);
signal reset          : std_logic;
signal EN             : std_logic;
signal CLK            : std_logic;

begin

clk <= clk_i;
reset <= not rstn;
led <= pushbutton;
R1:Ring generic map(inv_delay_ps => 100) port map(RSTn=>RSTn,RESET=>reset_counters,EN=>enable_counters,counter=>counter1);
R2:Ring generic map(inv_delay_ps => 110) port map(RSTn=>RSTn,RESET=>reset_counters,EN=>enable_counters,counter=>counter2);
R3:Ring generic map(inv_delay_ps => 120) port map(RSTn=>RSTn,RESET=>reset_counters,EN=>enable_counters,counter=>counter3);
R4:Ring generic map(inv_delay_ps => 130) port map(RSTn=>RSTn,RESET=>reset_counters,EN=>enable_counters,counter=>counter4);
R5:Ring generic map(inv_delay_ps => 140) port map(RSTn=>RSTn,RESET=>reset_counters,EN=>enable_counters,counter=>counter5);
R6:Ring generic map(inv_delay_ps => 150) port map(RSTn=>RSTn,RESET=>reset_counters,EN=>enable_counters,counter=>counter6);
R7:Ring generic map(inv_delay_ps => 160) port map(RSTn=>RSTn,RESET=>reset_counters,EN=>enable_counters,counter=>counter7);
R8:Ring generic map(inv_delay_ps => 170) port map(RSTn=>RSTn,RESET=>reset_counters,EN=>enable_counters,counter=>counter8);
R9:Ring generic map(inv_delay_ps => 180) port map(RSTn=>RSTn,RESET=>reset_counters,EN=>enable_counters,counter=>counter9);
R10:Ring generic map(inv_delay_ps => 190) port map(RSTn=>RSTn,RESET=>reset_counters,EN=>enable_counters,counter=>counter10);
```

```
compar:Comparator port map(          Response =>Response,
                                     counter0_in=>counter1,
                                     counter1_in=>counter2,
                                     counter2_in=>counter3,
                                     counter3_in=>counter4,
                                     counter4_in=>counter5,
                                     counter5_in=>counter6,
                                     counter6_in=>counter7,
                                     counter7_in=>counter8,
                                     counter8_in=>counter9,
                                     counter9_in=>counter10);

FS_M:FSM port map(                   RSTn=>RSTn,
                                     EN=>EN,
                                     clk=>clk,
                                     ready=>ready,
                                     start=>start,
                                     enable_counters => enable_counters,
                                     reset_counters => reset_counters,
                                     uart_busy=>busy,
                                     CRTL=>CRTL);

counter_T:Timer port map(            RSTn=>RSTn,
                                     clk=>clk,
                                     ready=>ready);

uart:serial_tx port map (            clk=>clk,
                                     rst=>reset,
                                     tx=>tx,
                                     tx_block=>'0',
                                     new_data=>start,
                                     busy=>busy,
                                     data=>TX_OUT);

MUX:MUX_Uart port map (              Response=>Response,
                                     TX_OUT=>TX_OUT,
                                     CRTL=>CRTL);

Push:Push_button port map (clk=>clk,
                                     RSTn=>RSTn,
                                     Push_button_in=>Pushbutton,
                                     Push_Button_out=>EN);

end Behavioral;
```

- **RO_PUF.ccf**

```
## PUF.ccf

Pin_in "clk_i" Loc = "IO_SB_A8" | SCHMITT_TRIGGER=true;
Pin_in "rstn" Loc = "IO_EB_B0"; # SW3
Pin_out "tx" Loc = "IO_NB_A5"; # tx
Pin_in "pushbutton" Loc = "IO_NB_A0"; # BTN0 PMOD A
Pin_in "pushbutton2" Loc = "IO_NB_B0"; # BTN1 PMOD A
Pin_out "led" Loc = "IO_EB_B1";#led
```

- **synth.tcl**

```
## synth.tcl

## argparse

set top [lindex $argv 0]
set src [lrange $argv 1 end]

## import yosys commands

yosys -import

## vhdl frontend

ghdl --warn-no-binding -C --ieee=synopsys src/FSM.vhd src/inverter.vhd src/RO_PUF.vhd src/Ring_osc.vhd
src/Timer.vhd src/Push_Button src/Comparator.vhd src/serial_tx.vhd src/MUX_Uart.vhd -e ${top}

## synthesis

synth_gatemate -top ${top} -nomx8

## outputs

write_verilog -noattr net/${top}_synth.v
```