Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelorarbeit

Daniel Röhrborn

Entwicklung eines FPGA basierten Triggergenerators
für Fehlerinjektionsattacken und
Seitenkanalanalysen von Smart Cards

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

**Daniel Röhrborn**

# Entwicklung eines FPGA basierten Triggergenerators für Fehlerinjektionsattacken und Seitenkanalanalysen von Smart Cards

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Michael Schäfers
Zweitgutachter : Dr. Wolfgang Tobergte

Abgegeben am 20.10.2015

**Daniel Röhrborn**

**Thema der Bachelorarbeit**

Entwicklung eines FPGA basierten Triggergenerators für Fehlerinjektionsattacken und Seitenkanalanalysen von Smart Cards

**Stichworte**

Triggererzeugung, Laufzeitkonfiguration, Chipkarte

**Kurzzusammenfassung**

Moderne mikrokontrollerbasierte SmartCards enthalten vertrauliche Algorithmen und Daten. Um diese vor unerlaubtem Zugriff zu schützen, entwickeln SmartCard-Hersteller Sicherheitsfunktionen und Gegenmaßnahmen. Zur Analyse und Verifizierung dieser Schutzmechanismen bedarf es komplexer Messaufbauten. Die an einer Messung beteiligten Komponenten müssen zeitlich präzise, mithilfe digitaler Triggersignale, gesteuert werden, welche abhängig von internen Zuständen der untersuchten SmartCard generiert werden.

Zeitgemäße Oszilloskope bieten hierzu die Möglichkeit durch Verknüpfung einzelner Messsignale, über die Definition einer Triggerbedingung, ein Triggersignal zu generieren. Die Triggerbedingung lässt sich jedoch selten aus mehr als vier einfachen Unterbedingungen formulieren und beschränkt sich zusätzlich auf ein Ausgangssignal. Dieses und weitere zur Triggererzeugung eingesetzte Geräte weisen zusammen zwei schwerwiegende Nachteile hinsichtlich der Konfigurierbarkeit auf:

Erstens ist die Anzahl der Triggerbedingungen sehr eingeschränkt. Zweitens ist die Konfiguration zeitaufwändig und nicht automatisierbar.

Ziel dieser Arbeit ist die Entwicklung eines Triggergenerators, der auf ein spezielles bestehendes System zugeschnitten ist und die Verarbeitung komplexerer Triggerbedingungen, auf Basis digital ausgewerteter Signale, ermöglicht. Zudem soll mit dem Ansatz die Möglichkeit bestehen, mehrere Ausgangssignale gleichzeitig generieren zu können. Zur Realisierung des Triggergenerators bietet sich eine FPGA basierte Entwicklungsplattform an. Kernstück dieser Arbeit ist die Implementierung elementarer Triggerkomponenten, die zur Laufzeit konfiguriert werden können, um zusammen komplexe Triggerbedingungen zu realisieren.

Praktische Anwendung findet der Triggergenerator als Komponente eines SmartCard-Lesegeräts. Dort wertet er die Signalleitungen zwischen SmartCard und Lesegerät, hinsichtlich benutzerdefinierter Triggerbedingungen, aus und erzeugt entsprechende Triggersignale. Letztere werden dann beispielsweise verwendet, um eine Seitenkanalanalyse zu starten, einen Angriff zum Auslesen geheimer Daten. Dieser und andere Angriffe werden von SmartCard-Testern durchgeführt, um die Wirksamkeit von Sicherheitsfunktionen und Abwehrmechanismen der SmartCard zu kontrollieren. Ein wichtiger Aspekt dieser Arbeit ist ein modularer Aufbau der Teilkomponenten und einfache Schnittstellen um ein Höchstmaß an Erweiterbarkeit zu gewährleisten.

**Daniel Röhrborn**

**Title of the paper**

Development of a FPGA based Triggergenerator for Fault Injection Attacks and Side Channel Analysis on Smart Cards

**Keywords**

Trigger generation, runtime configuration, smart card

**Abstract**

Modern microcontroller based smart cards contain confidential algorithms and data. For protection against unauthorized access security features and countermeasures are implemented by the smart card developer. Their efficiency is analyzed and verified using complex test setups. The involved measuring instruments must be controlled by precisely timed trigger signals which are generated according to various internal states of a smart card. Modern oscilloscopes provide the required function to combine measuring signals logically by defining multistage trigger conditions to generate trigger signals. In most cases it is, however, not possible to define a multistage trigger condition from more than four subconditions or use more than one trigger output. The goal of development is a triggergenerator which is compatible to a specific existing system and which allows the user to define more complex digitally processed trigger conditions. Several trigger signals shall also be computable simultaneously. For the implementation a FPGA based development platform is conceivable. Core work of this bachelor thesis is the implementation of runtime configurable elementary trigger components which can be combined to realize complex trigger conditions.

The triggergenerator will serve as component of a smart card reader. He analyzes signal lines about user-defined trigger conditions and generates appropriate trigger signals. These are subsequently used to start a side-channel analysis, an attack to gain confidential data. These and other attacks are performed by smart card tester to verify the efficiency of security functions and countermeasures of smart cards. An important aspect of implementation is a modular design of the components and their interfaces to ensure the best possible extensibility.

# Inhaltsverzeichnis

# 1 Einleitung

Die Einsatzgebiete moderner Prozessorkarten, sogenannter Smartcards, finden sich heutzutage überwiegend im Finanz- und Sicherheitssektor, wo sie beispielsweise als Bankkarte oder zur Zugangskontrolle von Gebäuden verwendet werden (vgl. Effing 2008, S. 9). Sie enthalten geheime Daten zur Authentifizierung von Personen, wie zum Beispiel Passwörter und implementieren Verschlüsselungsalgorithmen, die geheime Parameter verwenden. Diese Daten und Algorithmen müssen vor unerlaubtem Auslesen geschützt werden (vgl. Rüdinger 2009, S. 22). Verfügt eine Smartcard über unzureichende Sicherheitsfunktionen, können verschiedenartige Angriffe auf sie ausgeübt werden, um an die geheimen Daten zu gelangen. Eine Fehlerinjektionsattacke manipuliert etwa Teile der Hard- und Software der Smartcard, um Fehlfunktionen zu provozieren, beispielsweise die Ausgabe zufälliger (geheimer) Speicherinhalte. Alternativ wird die Hardware einer Smartcard im Zuge einer Seitenkanalanalyse mithilfe verschiedener Messgeräte überwacht, um Informationen über die ausgeführte Software zu erhalten. Die Entwicklung einer Smartcard mit huntertprozentig angriffssicherer Hard- und Software ist praktisch unmöglich. Um dennoch ein Höchstmaß an Sicherheit zu gewährleisten führen Smartcardhersteller, wie die Firma NXP Semiconductors[1], selber Angriffe durch, mit dem Ziel Sicherheitslücken aufzudecken. Diese werden dann schnellstmöglich geschlossen, um die Smartcards sicherer zu machen. Die Durchführung von Fehlerinjektionsattacken und Seitenkanalanalysen erfordert eine zeitlich sehr präzise Steuerung. Der Angriffszeitpunkt muss möglichst genau definiert und erreicht werden können. Denn als Annäherung an das Ziel einer absolut angriffssicheren Smartcard, wird versucht alle erdenklichen Angriffe durchzuführen und auszuwerten. Dies läuft darauf hinaus die Smartcard bei Ausführung jedes einzelnen Prozessorbefehls zu Manipulieren oder zu überwachen.

Zur Steuerung von Angriffen werden bestimmte Oszilloskope eingesetzt. Diese ermöglichen sowohl die Aufzeichnung mehrerer Signalverläufe, als auch deren Auswertung hinsichtlich verschiedener Triggerbedingungen und die Ausgabe eines Triggersignals, nachdem diese erfüllt wurden. Mithilfe eines Oszilloskops können somit die zwischen Smartcard und Smartcard-Lesegerät verlaufenden Signalleitungen beobachtet werden und ein Triggersignal ausgegeben werden, wenn bestimmte Signale übertragen wurden. Derartige Triggersignale zeigen den Zeitpunkt für einen Angriff an. Der Benutzer definiert einen Angriffszeitpunkt demzufolge über Triggerbedingungen.

Alternativ oder unterstützend zum Oszilloskop wird ein speziell für dieses Einsatzgebiet entwickeltes Gerät eingesetzt, der „VC-Glitcher“ von der Firma Riscure[2]. Dieser generiert ebenfalls ein Triggersignal, kann einige Angriffe jedoch auch direkt durchführen.

Die zur Triggererzeugung eingesetzten Geräte arbeiten einerseits sehr genau und zuverlässig, weisen andererseits jedoch zwei schwerwiegende Nachteile hinsichtlich der Konfigurierbarkeit auf:

Erstens ist der mögliche Umfang benutzerdefinierbarer Triggerbedingungen sehr eingeschränkt. Zweitens ist deren Konfiguration zeitaufwändig und zudem nicht automatisierbar. Dies wirkt dem Ziel entgegen möglichst viele Angriffe in begrenzter Zeit durchzuführen zu können. Denn idealerweise

[1] NXP Semiconductors - Internetpräsenz unter http://www.nxp.com

[2] Weitere Informationen zu Riscure unter https://www.riscure.com/security-tools/hardware/vc-glitcher

erfolgt ein Großteil der Angriffe automatisiert. Intelligente Algorithmen führen Angriffe durch, werten die Ergebnisse aus und bestimmen so die lohnendsten Ziele für Folgeangriffe. Wenn ein Angriff erfolgreich war, also eine Schwachstelle gefunden wurde, wird diese mithilfe weiterer, örtlich und zeitlich lokaler, Angriffe eingehender untersucht. Orte und Zeitpunkte der Angriffe werden also zum Teil zur Laufzeit bestimmt und sollen zur Laufzeit eingestellt werden können.

## 1.1 Aufgabenstellung

Gegenstand dieser Arbeit ist die Implementierung eines Triggergenerators, welcher es ermöglichen soll eine Vielzahl von Triggerbedingungen zu definieren, die nebenläufig Triggersignale berechnen und diese auf mehreren Ausgängen ausgeben.
Die unterschiedlichen Arten einsetzbarer Triggerbedingungen sollen sich an den Möglichkeiten moderner Oszilloskope orientieren. Diese ermöglichen zudem die Definition einer sogenannten mehrstufigen Triggerbedingung. Dabei handelt es sich um eine Sequenz aus Triggerbedingungen von denen eine nach der anderen erfüllt werden muss, damit die gesamte Triggerbedingung erfüllt ist.
Jede Unterbedingung[3] einer mehrstufigen Triggerbedingung beobachtet dazu eine Signalleitung innerhalb eines Zeitfensters, das beginnt nachdem die vorhergehende Unterbedingung erfüllt wurde. Das am Ende der Sequenz ausgegebene Triggersignal kann vom Benutzer bezüglich einer optionalen Ausgabeverzögerung und der gewünschten Ausgabelänge konfiguriert werden. Diese beiden zeitlichen Abstände sollen mit höchstmöglicher Auflösung konfiguriert und von der Schaltung mit maximaler Präzision umgesetzt werden.
Die Konfiguration von Unterbedingungen und deren Kombination zu einer mehrstufigen Triggerbedingung soll zur Laufzeit des Systems, in kürzester Zeit, durchgeführt werden können. Zur Datenübertragung ist eine I²C-Schnittstelle zu implementieren, um mit dem bestehenden automatisierten Angriffssystem kompatibel zu sein.
Der Triggergenerator wird für die Firma NXP Semiconductors entwickelt. Er unterstützt die Smartcard-Tester bei der Durchführung von Angriffen auf Smartcards und hilft damit Sicherheitslücken aufzudecken, welche anschließend von den Smartcard-Entwicklern beseitigt werden. Ein wichtiger Aspekt bei der Entwicklung des Triggergenerators ist eine benutzerfreundliche Bedienung, um dessen Einsatz auch Mitarbeitern zu ermöglichen, die nicht mit dem internen Aufbau und der Funktionsweise vertraut sind. Unterstützend soll ein User Guide verfasst werden, welcher alle Einstellmöglichkeiten im Detail erläutert.
Die zukünftige Weiterentwicklung des Projekts übernehmen ebenfalls Smartcard-Tester, wozu detaillierte Beschreibungen von Aufbau und Funktionsweise aller implementierten Komponenten gefordert werden.

[3] Gemäß dem Sprachgebrauch der Smartcard-Entwickler, bilden mehrere Unterbedingungen in einer Sequenz eine mehrstufige Triggerbedingung.

## 1.2 Gliederung der Arbeit

Der folgende Abschnitt, **Kapitel 2**, erklärt zunächst grundlegende Begriffe, deren Bedeutungen dem Leser, zum vollständigen Verständnis dieser Ausarbeitung, bekannt sein sollten.
In **Kapitel 3** werden die Anforderungen wiedergegeben. Das Kapitel ist in zwei Abschnitte unterteilt, die allgemeinen und die spezifischen Anforderungen. Die allgemeinen Anforderungen beschreiben alle Funktionen, die der Triggergenerator bieten soll. Die spezifischen Anforderungen enthalten, für jede dieser Funktionen spezifische Zeitabstände und Wertebereiche die eingehalten werden müssen. Während die allgemeinen Anforderungen dem Leser einen Überblick über die verfügbaren Funktionen bieten, dienen die spezifischen Anforderungen hauptsächlich zur Verifikation der Implementierung. In **Kapitel 4** werden alternative, bereits vorhandene, Möglichkeiten zur Triggergenerierung und deren Vor- und Nachteile betrachtet. Anschließend beschreibt und bewertet **Kapitel 5** drei grundlegend verschiedene Lösungsansätze zur Umsetzung des Triggergenerators. Auf einem dieser Lösungsansätze basierte der Entwurf des Triggergenerators, welcher in **Kapitel 6** erläutert wird. Hier wird das Konzept hinter den umgesetzten Komponenten und deren Relevanz im Bezug auf die Anforderungen beschrieben. In **Kapitel 7** wird auf die Implementierung der triggererzeugenden Teilschaltung des Triggergenerators eingegangen. Um Anpassungen der komplexen internen Verschaltung vornehmen zu können, wurden Maßnahmen zur deutlichen Erhöhung des Komforts für den Benutzer ergriffen, deren Umsetzung im Detail erläutert wird. Im Anschluss wird die Implementierung der Benutzerschnittstelle beschrieben. Anschließend beschreibt **Kapitel 8** verschiedene Tests, denen der Triggergenerator unterzogen wurde, inklusive deren Aufbau und Auswertung. Die nötigen Schritte zur Umsetzung dieser Anwendungsbeispiele werden detailliert im User Guide beschrieben, der sich in Anhang A befindet. Durch den Vergleich der Testergebnisse mit den Anforderungen wird die Implementierung verifiziert.
Das abschließende **Fazit** fasst die Errungenschaften des entwickelten Triggergenerators zusammen. Der **Ausblick** beschreibt mögliche Erweiterungen, die in Zukunft benötigt werden.
In **Anhang A** befindet sich der User Guide, wobei ergänzende Informationen in **Anhang B** und **Anhang C** ausgelagert wurden. **Anhang D** enthält detaillierte Beschreibungen aller implementierten Komponenten.

# 2 Begriffe und Techniken

Um an die geheimen Daten einer Smartcard zu gelangen, werden verschiedenartige Angriffe, in Form von Fehlerinjektionsattacken oder Seitenkanalanalysen, auf sie ausgeübt.
Für die Kommunikation einer Smartcard mit einem Smartcard-Lesegerät, dem Reader[4], stehen verschiedene Kommunikationsprotokolle zur Verfügung.

## 2.1 Smartcard und Reader

Eine Smartcard ist eine Chipkarte, die einen Mikrocontroller enthält auf dem ein proprietäres Betriebssystem läuft. Dadurch ist die Smartcard in der Lage Rechenaufträge bearbeiten zu können, wie zum Beispiel die Verschlüsselung zuvor übertragener Daten. Smartcards enthalten oft geheime Schlüssel und Verschlüsselungsalgorithmen. Diese dürfen unter keinen Umständen nach außen getragen werden, weder absichtlich, noch im Fehlerfall. In dieser Arbeit wird ein Mikrokontroller vom Typ SmartMX P40 der Firma NXP Semiconductors für Tests verwendet.

Der Reader stellt dem Host-PC eine Schnittstelle zur Smartcard zur Verfügung. Er empfängt Nachrichten, die zur Smartcard übertragen werden sollen vom Host-PC und sendet diese, übersetzt in ein Smartcard-Kommunikationsprotokoll, weiter zur Smartcard. Anschließend empfängt er die Antwort der Smartcard und reicht diese zum Host-PC weiter.

## 2.2 Fehlerinjektionsattacke

Eine Fehlerinjektionsattacke ist ein aktiver Angriff auf die Smartcard. Dabei wird die Hardware der Smartcard zur Laufzeit manipuliert, beispielsweise durch Bestrahlung der Chipoberfläche. Ziel ist es Fehlfunktionen aufzudecken, die dazu führen, dass die Smartcard geheime Daten ausgibt.
Es existieren unterschiedliche Arten von Fehlerinjektionsattacken, wobei der zu entwickelnde Triggergenerator primär zur Steuerung einer sogenannten Laserattacke eingesetzt werden soll. Zukünftig sind jedoch auch Anwendungen mit anderen Fehlerinjektionsarten denkbar.
Im Folgenden seien einige Beispiele für Fehlerinjektionsattacken genannt, gefolgt von einer Erläuterung der Auswertung eines Angriffs und einigen grundlegenden Mechanismen zur Erkennung von Angriffen und Gegenmaßnahmen.
Das Wissen über die genauen Vorgänge ist für das Verständnis dieser Ausarbeitung nicht notwendig. Dennoch soll dem interessierten Leser ein kurzer Einblick gewährt werden.
Fehler können beispielsweise auf folgende Arten injiziert werden (vgl. Tunstall 2012, S. 277-280):

- Ein Laser schießt zu einem bestimmten Zeitpunkt auf einen Teil des Chips, um für einen bestimmten Zeitraum (üblich sind 10 ns bis 1 µs) einen lokalen Speicher- oder Bus-Fehler zu verursachen.
- Eine Spule wird über einem bestimmten Teil des Chips angebracht, um diesen durch ein elektromagnetisches Feld zu stören.
- Ein Pulsgenerator erzeugt eine Spannungsspitze in der Stromzufuhr der Smartcard oder alternativ eine dauerhafte Unterversorgung.

[4] Im allgemeinen Sprachgebrauch wird das Lesegerät auch als Terminal oder Reader bezeichnet. Fortan wird hierfür der Begriff Reader verwendet.

- Verkürzung der Länge ausgewählter CPU-Takte oder dauerhafte Übertaktung, sofern der Chip den Arbeitstakt von außen empfängt.
- Ein Heißluftföhn erhitzt die Smartcard auf bis zu 200°C.

Es ist grundsätzlich nicht möglich zu ermitteln welche Auswirkung ein Angriff auf die Bits im Speicher oder einen Bus genau hatte, denn die Hardware verhält sich instabil. Bei Verwendung eines Lasers lassen sich, vereinfacht gesagt, Zeitpunkt, Dauer, Ort und Intensität der Bestrahlung, sowie der Durchmesser des Laserstrahls festlegen. Wird dann beispielsweise ein Angriff auf Speicherzellen ausgeführt, durchlaufen diese jeweils drei Zustände:

1. **Ursprungszustand**: Vor der Bestrahlung befinden sich die Speicherzellen in einem bekannten Ursprungszustand.
2. **Zwischenzustand**: Für die Dauer der Laser-Bestrahlung ändert sich der bekannte Ursprungszustand der Speicherzellen in einen unbekannten, instabilen Zwischenzustand.
3. **Endzustand**: Mit dem Ende der Bestrahlung werden sie wiederum in einen unbekannten, instabilen Endzustand überführt. Denn ein Teil der bestrahlten Zellen kehrt dann in seinen Ursprungszustand zurück und ein anderer Teil bleibt im Zwischenzustand. Die Zustände 2 und 3 sind als instabil anzusehen, da ein Auslesen der Daten zu einer erneuten Zustandsänderung führen kann.

Aufgrund dieser Instabilitäten wird zur Auswertung der Angriffe keine Analyse der Smartcard-Hardware durchgeführt, sondern lediglich eine Auswertung der Antwort der Smartcard. Üblicherweise beginnt ein Angriff mit der Übermittlung eines Rechenauftrags zur Smartcard. Im Zuge dieser Berechnung führt die Smartcard beispielsweise Zugriffe auf Speicherzellen aus, wobei einer dieser Zugriffe durch einen Laserschuss auf die gelesenen Speicherzellen manipuliert sei. Am Ende des Rechenauftrags sendet die Smartcard das Ergebnis zum Reader.

Ein Angriff wurde erfolgreich abgewendet, wenn das korrekt berechnete Ergebnis, ein Fehlercode oder gar nichts von der Smartcard zurückgegeben wird. Gibt diese jedoch ein verfälschtes Ergebnis oder gar beliebige Speicherinhalte aus, war der Angriff erfolgreich. Denn er hat Einfluss auf den Verlauf der Berechnung genommen und ein unzulässiges Verhalten der Smartcard herbeigeführt, ohne, dass diese den Angriff bemerkt hat.

Smartcard-Entwickler implementieren Gegenmaßnahmen, um die Wahrscheinlichkeit zu verringern, dass eine Smartcard als Folge eines Angriffs geheime Daten ausgibt. Zum Einen sollen Angriffe erkannt werden, um eine Berechnung abbrechen und dem Reader mit einer Fehlermeldung antworten zu können. Zum Anderen soll eine, durch Manipulation von Sprungbefehlen, außer Kontrolle geratene Smartcard so schnell wie möglich in einen sicheren Zustand gebracht werden.

Einige Beispiele für Gegenmaßnahmen seien im Folgenden erläutert:

- Zur Erkennung der Manipulation einer Berechnung, wird die gleiche Berechnung mehrmals ausgeführt und die Ergebnisse verglichen. Unterscheiden sich diese voneinander, hat ein Angriff stattgefunden und die Smartcard reagiert mit einer Fehlermeldung (vgl. Tunstall 2012, S. 74).
- Daten werden zwischen internen Komponenten mehrfach, nebenläufig übertragen (Dual Rail) um die Manipulation einer Leitung zu erkennen (vgl Tunstall 2012, S. 76).
- Die Manipulation von Daten innerhalb eines nicht-flüchtigen Speichers kann über CRC-Berechnungen (Cyclic Redundancy Checks) erkannt werden (vgl. Tunstall 2012, S. 75).
- Beliebige Instruktionen im Speicher können durch einen Angriff in andere Instruktionen umgewandelt werden. So können auch Sprungbefehle in andere Befehlsarten umgewandelt werden. Dies ist besonders dann ein Sicherheitsrisiko, wenn Sprünge außer Kraft gesetzt werden, die innerhalb einer Sicherheitsüberprüfung für den Sprung in einen Ausnahmebereich verwendet werden. Das Ausschalten bestimmter Sprünge würde verhindern, dass die Software auf der Smartcard Angriffe erkennt bzw. entsprechend darauf reagiert. Stattdessen würde der Prozessor einfach weiterlaufen und Instruktionsbereiche mit sensiblen Algorithmen auch dann ausführen, wenn die vorherige Sicherheitsüberprüfung bereits Unstimmigkeiten festgestellt hat. Als Gegenmaßnahme dazu werden u.A. Verschlüsselungsfunktionen auf

Assemblerebene in Blöcke zerteilt, die jeweils durch Rückwärtssprünge im Instruktionsspeicher miteinander verbunden sind, wie Abb. 1 zeigt.

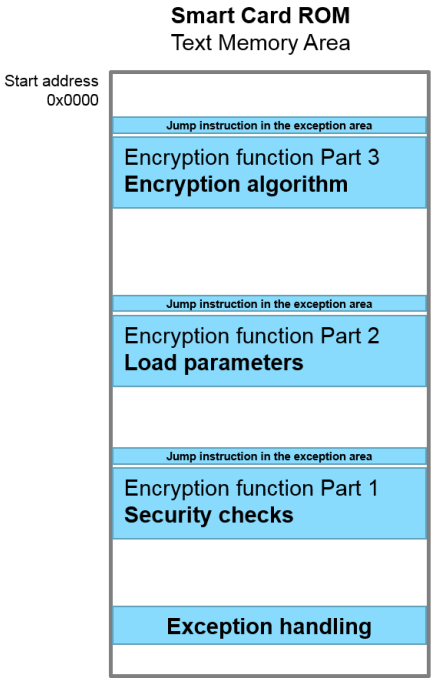


Abb. 1: Smartcard ROM – Gegenmaßnahme für Fehlerinjektionsattacken

Der Startblock einer Funktion, welcher Sicherheitsüberprüfungen enthält, liegt weiter hinten im Speicher, als der zweite Block, welcher Rechenparameter in die Register lädt. Der dritte Block, welcher einen vertraulichen Algorithmus enthält, liegt wiederum weiter vorne im Speicher, als der zweite Block. Somit ist es nicht mehr möglich Sicherheitsüberprüfungen am Anfang einer Funktion zu überspringen und dadurch die Ausführung sensibler Berechnungen zu erzwingen. Denn ein außer Kraft gesetzter Sprung in einem Funktionsblock im hinteren Teil des Speichers würde dazu führen, dass nur Instruktionen ausgeführt werden die noch weiter hinten im Speicher liegen.

Eine andere mögliche Absicht, anstelle Sicherheitsüberprüfungen zu überspringen, kann die gezielte Manipulation eines bestimmten Teils einer Verschlüsselungsberechnung sein. Abhängig vom verwendeten Verschlüsselungsalgorithmus, kann ein Angreifer mithilfe eines korrekten und eines fehlerhaft verschlüsselten Wertes auf mathematischem Weg auf die verwendeten Schlüsselwerte schließen. Ein Beispiel dafür ist die sogenannte Bellcore-Attacke auf den RSA-CRT Algorithmus. Eine ausführliche Erläuterung der mathematischen Hintergründe findet sich in Tunstall 2012 (S.59 - 62).

- Da durch einen Angriff beliebige Sprungbefehle erzeugt werden können, sind auch Sprünge an unbenutzte Speicherstellen, zwischen den Instruktionsbereichen, möglich. In diesem Fall soll verhindert werden, dass die Smartcard einen leeren Speicherbereich durchläuft und dann mit der Ausführung des folgenden Instruktionsblocks fortfährt. Dazu werden in die unbenutzten Speicherbereiche Sprungbefehle in Bereiche zur Ausnahmebehandlung eingefügt, wie in Abb. 1 zu sehen ist. Nach einem solchen Sprung, wird die Smartcard in einem Ausnahmezustand festgehalten. Nur durch einen Reset kann sie aus einem Ausnahmezustand wieder in einen betriebsbereiten Zustand gebracht werden.
- Für die Bestrahlung der Smartcard-Oberfläche mit einem Laser, muss der Chip freiliegen und darf durch nichts verdeckt werden. Um dies zu erkennen, werden tausende Helligkeitssensoren in den Chip integriert, welche die Smartcard für jegliche Funktion sperren, wenn Licht auf sie einfallen sollte.
- Die Anzahl der, von der Smartcard erkannten, Angriffe wird gespeichert. Nach einer modellabhängigen Anzahl von Angriffen, meist 3, deaktiviert sich die Smartcard irreversibel.

## 2.3 Seitenkanalanalyse

Eine Seitenkanalanalyse ist ein passiver Angriff auf die Smartcard. Verschiedene hardwarespezifische Informationen über den Mikrokontroller der Smartcard werden über nicht dafür vorgesehene Kanäle, die sogenannten Seitenkanäle, ermittelt.
Die erste Seitenkanalanalyse führte der amerikanische Kryptologe Paul Kocher im Jahr 1996 durch. Bei der als Rechenzeitangriff bezeichneten Angriffsmethode, wird die Tatsache ausgenutzt, dass die Laufzeit eines (Verschlüsselungs-) Algorithmus von den verwendeten Parametern abhängt. Ist grundsätzlich bekannt, welche Berechnungen ein Algorithmus durchführt, kann durch Messung der Rechenzeiten und unter Einsatz verschiedener Parameter, auf die geheimen Charakteristika des Algorithmus geschlossen werden. Im Falle eines Verschlüsselungsalgorithmus ist es auf diese Weise sogar möglich einen geheimen Schlüsselwert zu ermitteln (vgl. Kocher 1996, S. 1-2).
Weitere Arten von Seitenkanalanalysen sind:

- Die Messung der Stärke elektromagnetischer Felder über der Smartcard-Oberfläche, um Charakteristika des, von der Smartcard ausgeführten, Algorithmus aufzudecken. Außerdem können so die auf einem Bus übertragenen Daten mitgelesen werden. Dazu misst eine mikroskopische Spule für eine bestimmte Zeitspanne die Schwankungen in der elektromagnetischen Feldstärke über einer bestimmten Baugruppe des Chips.
- Überwachung des Stromverbrauchs der Smartcard (Differential Power Analysis). Dieser schwankt während einer Berechnung, abhängig vom ausgeführten Mikrokontrollerbefehl und durch die Verwendung verschiedener Baugruppen des Chips. Mit dem Wissen, welche Baugruppe, wie lange aktiv war, kann auf Charakteristika eines Algorithmus geschlossen werden (vgl. Tunstall 2012, S. 37). Zum Teil ist es sogar möglich die Eingabeparameter einer Berechnung zu ermitteln.

Verglichen mit den aktiven Angriffsarten ist die passive Seitenkanalanalyse, nach Erfahrung der Smartcard-Tester, die erfolgversprechendere Angriffsmethode. Denn die Smartcard ist nicht in der Lage eine Überwachung der Seitenkanäle als Angriff zu erkennen. Die Gegenmaßnahmen für passive Angriffe fallen entsprechend weniger wirkungsvoll aus, als die der aktiven Angriffe. Denn ohne die Möglichkeit einen Angriff zu erkennen, kann nicht aktiv gegen diesen vorgegangen werden (vgl. Rüdinger 2009, S. 3). Deshalb ist die Entwicklung von Gegenmaßnahmen für Seitenkanalanalysen wesentlich zeitaufwändiger, als es bei den aktiven Angriffsarten der Fall ist.
Die implementierten Gegenmaßnahmen dienen dazu die Wahrscheinlichkeit zu verringern, dass geheime Daten ausgelesen werden können. Da nicht erkennbar ist, wann ein Angriff stattfindet, wird versucht die Übertragung geheimer Daten und den Verlauf geheimer Berechnungen zu verschleiern.

Einige Beispiele für Gegenmaßnahmen seien im Folgenden erläutert:

- Algorithmen werden um irrelevante Berechnungen bzw. zufällige Wartezeiten erweitert. So wird es dem Angreifer erschwert, von den gemessenen Charakteristika des Algorithmus, auf die des relevanten Teils der Berechnung zu schließen (vgl. Tunstall 2012, S. 76).
- Signalleitungen werden in doppelter Anzahl eingebaut. Indem parallel zu einer Datenleitung im Chip eine weitere verlegt wird, welche stets den gegenteiligen Signalpegel überträgt, erreicht man einen, von Datenübertragungen unabhängigen, Stromverbrauch. So kann durch dessen Messung nicht mehr festgestellt werden welche Mikrokontrollerbefehle ausgeführt wurden bzw. wann und wie lange bestimmte Berechnungen stattfanden.
- Metallene Schutzschichten werden um Komponenten im Chip herum angebracht, um die Abgabe elektromagnetischer Strahlung einzudämmen.

## 2.4 Kommunikationsprotokolle

Für die Kommunikation zwischen Reader und Smartcard werden überwiegend die genormten Übertragungsprotokolle T=0 und T=1 eingesetzt. Während der Entwicklung und Tests von Smartcards werden jedoch auch proprietäre, nicht spezifizierte Protokolle eingesetzt.
Das I²C-Protokoll ist ein einfach zu implementierendes Übertragungsprotokoll und grundsätzlich für die Kommunikation verschiedener Schaltungsteile miteinander vorgesehen. In diesem Projekt wird es für die Konfiguration des Triggergenerators verwendet.

Der ISO/IEC 7816-3 Standard spezifiziert die beiden asynchronen Halbduplex-Datenübertragungsprotokolle T=0 und T=1 (Abkürzung für transport protocol 0 und 1), welche für die Kommunikation zwischen einer Smartcard und einem Reader verwendet werden. Jede Datenübertragung beginnt mit einer Anfrage, die der Reader zur Smartcard überträgt und endet mit einer Antwort durch die Smartcard. Der Standard definiert dazu unter Anderem die Formatierungen und Größen der Datenpakete und Mechanismen zur Fehlerkorrektur, worin sich beide Protokolle unterscheiden. Für weitere Informationen siehe Effing 2008 (S. 275). Die Übertragung auf Bitebene (bzw. Leitungsschicht) erfolgt stets nach dem Prinzip zur asynchronen Datenübertragung. Sie findet asynchron zum vorhandenen Kommunikationstakt statt, welcher lediglich zur Steuerung der Übertragungsgeschwindigkeit verwendet wird.
Der Kommunikationstakt wird vom Reader erzeugt und liegt bei einer Frequenz im Bereich 1 bis 5 MHz. Vor Beginn der Datenübertragung vereinbaren beide Teilnehmer einen Taktteiler, mit dessen Hilfe die tatsächliche Datenrate festgelegt wird (vgl. Effing 2008, S. 265). Der Taktteiler gibt folglich die Länge eines Datenbits als Anzahl von Kommunikationstakten an. In der Praxis erreichte Datenraten liegen grundsätzlich bei 9,6 kBit/s bis 300 kBit/s. Es sei angemerkt, dass die Taktleitung, bei Smartcards jüngerer Generation mit eigener Takterzeugung, nur noch aus Kompatibilitätsgründen vorhanden ist. Ursprünglich wurde die Smartcard von außen mit einem CPU-Takt versorgt. Damals war der Taktteiler notwendig, da nicht der selbe Takt sowohl als CPU-Takt, als auch als Kommunikationstakt verwendet werden kann.
Der Triggergenerator soll unabhängig vom verwendeten Kommunikationsprotokoll eingesetzt werden können. Daher soll hier nicht näher auf diese eingegangen werden. Für weitere Details zu den Protokollen siehe Effing 2008 (ab S. 275).

Das I²C-Protokoll (Abk. für Inter-Integrated Circuit, siehe I2C 2014) ist ein serielles und taktsynchrones Datenübertragungsprotokoll, welches von der Firma Philips Semiconductors entwickelt wurde. Der I²C-Bus besteht aus einer seriellen und bidirektionalen Datenleitung und einer seriellen Taktleitung (vgl. Effing 2008, S. 272). Die Taktleitung SCL (Serial CLock) überträgt einen Kommunikationstakt, während die Datenleitung SDA (Serial DAta) zum bidirektionalen Datenaustausch dient. Beide Leitungen werden durch Pullup-Widerstände mit der Versorgungsspannung verbunden. Eine Datenübertragung erfolgt, indem beide Kommunikationsteilnehmer die SDA-Leitung auf Masse ziehen. High-Pulse werden passiv gesendet, indem der sendende Kommunikationsteilnehmer seinen SDA-Ausgang in den hochohmigen Zustand versetzt und der PullUp-Widerstand die Leitung im High-Zustand hält. Eine Datenübertragung beginnt stets mit einer Startbedingung und endet mit einer Stoppbedingung. Dazwischen erfolgt eine byteweise Datenübertragung, wobei auf jedes übertragene Byte ein Bestätigungsbit vom jeweiligen Empfänger gesendet wird. Dem Teilnehmer, welcher die Übertragung initiiert, wird die Rolle des I²C-Masters zugeordnet, während der antwortende Teilnehmer als I²C-Slave bezeichnet wird. Das Taktsignal wird ausschließlich vom I²C-Master erzeugt, wobei der I²C-Slave diesen jedoch bei Bedarf verlangsamen kann (Clock Stretching genannt).

# 3 Anforderungen

Der Triggergenerator soll zwischen einer Smartcard und einem Reader eingesetzt werden, um Signalleitungen zur Auswertung von mehrstufigen Triggerbedingungen beobachten zu können und auch Einfluss auf die Signalleitungen nehmen zu können. Die von ihm ausgegebenen Triggersignale dienen dazu Fehlerinjektionsattacken und Seitenkanalanalysen zu starten.
Der folgende Abschnitt beschreibt das bereits existierende System, welches die Laufzeitumgebung des Triggergenerators bilden soll. Anschließend werden die daraus abgeleiteten allgemeinen Anforderungen beschrieben. Der Schwerpunkt liegt dabei auf den Funktionen, welche der Triggergenerator dem Benutzer zur Verfügung stellen soll. Im Anschluss daran folgen die spezifischen Anforderungen, welche eine Zusammenfassung der einzuhaltenden zeitlichen Grenzwerte darstellen. Diese dienen vorwiegend zum Vergleich mit Testergebnissen und deren Bewertung am Ende der Ausarbeitung.

## 3.1 Umgebendes System

Der Triggergenerator soll in das, in Abb. 2 dargestellte, bestehendes System eingefügt werden und muss daher seine Schnittstellen den gegebenen Umständen anpassen.

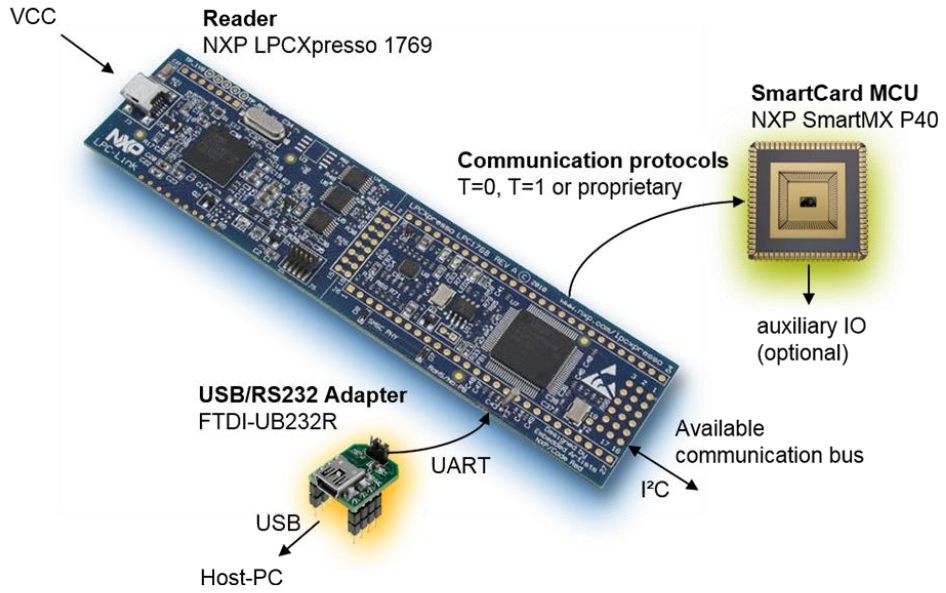


Abb. 2 Das bestehende System – Vereinfachte Darstellung

Die bei der Entwicklung des Triggergenerators zu verwendenden Hardwarekomponenten und deren Schnittstellen werden in Abb. 2 vereinfacht dargestellt:

- NXP LPCXpresso 1769-Board: Enthält die Reader-Software.
- NXP SmartMX P40: Enthält die Smartcard-Software. Abgebildet ist der Smartcard-Prozessor in einem CQFJ-Package (Abk. für Ceramic Leaded Chip), einem Gehäuse, das die Handhabung der Testchips vereinfacht. In dessen Mitte ist der Chip zu sehen, welcher auch in einer Smartcard enthalten ist.

- FTDI-UB232R: Adapterbaustein, der eine Übersetzung zwischen dem USB-Protokoll und dem UART-Protokoll (Universal Asynchronous Receiver Transmitter) vornimmt. Aus Sicht des Readers kommuniziert dieser über seine UART-Schnittstelle mit dem Host-PC. Äquivalent dazu kommuniziert auch das Daten an den Reader sendende Programm auf dem Host-PC aus seiner Sicht über eine (virtuelle) UART-Schnittstelle mit dem Reader. Da der USB/UART-Adapter somit transparent für die Kommunikation aus Sicht beider Teilnehmer ist, wird er in den folgenden Darstellungen, der Einfachheit halber, weggelassen.

Der Reader dient dazu eine Übersetzung zwischen verschiedenen Kommunikationsprotokollen vorzunehmen. Wie in Abb. 2 dargestellt, kommuniziert er über eine UART-Schnittstelle, mithilfe eines USB/RS232-Übersetzers (FTDI-UB232R), mit dem Host-PC. Über diese Schnittstelle erhält der Reader Daten und Kommandos für die Kommunikation mit der Smartcard oder für die Kommunikation über eine I²C-Schnittstelle. Da die I²C-Schnittstelle des Readers softwareseitig bereits für die Kommunikation mit externen Komponenten vorbereitet ist, soll sie auch für die Konfiguration des Triggergenerators eingesetzt werden. Die von der Smartcard oder der I²C-Schnittstelle empfangenen Antworten, leitet der Reader schließlich an den Host-PC weiter.
Für die Kommunikation mit der Smartcard werden entweder die Kommunikationsprotokolle T=0 / T=1 verwendet oder proprietäre, nicht standardisierte Protokolle, die zu Testzwecken während der Smartcard-Entwicklung eingesetzt werden. Anzahl und Funktionen der Signalleitungen zwischen Reader und Smartcard sind protokollunabhängig. Der Reader versorgt die Smartcard stets mit Strom, einen Resetsignal, einem Kommunikationstakt und kommuniziert mit ihr über eine bidirektionale Kommunikationsleitung. Abb. 3 zeigt die mit dem Reader verbundenen Signalleitungen in einer Übersicht.

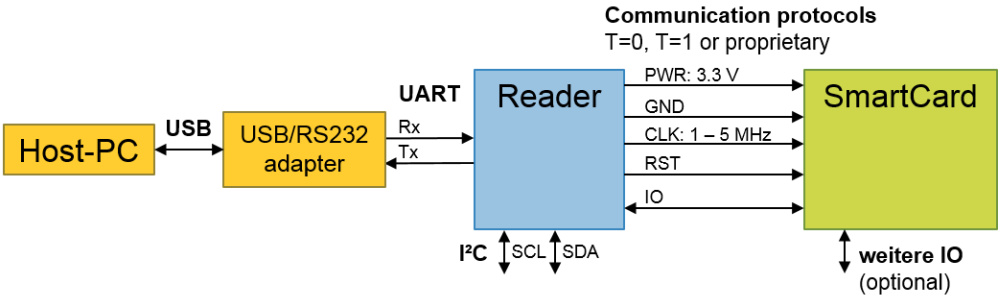


Abb. 3: Kommunikationsleitungen: Host-PC, Reader und Smartcard aus Sicht des Readers

1. **PWR (Power)**: Die Smartcard erhält vom Reader eine Versorgungsspannung von 3.3 V und nimmt dabei einen Strom von bis zu 100 µA auf.
2. **GND (Ground):** Masseleitung für die Smartcard.
3. **CLK (Clock)**: Unidirektionale Taktleitung vom Reader zur Smartcard. Zwischen Smartcard und Reader findet eine Absprache statt, um eine konkrete Taktrate zu vereinbaren. Der übertragene Takt entspricht einer Baudrate, die von beiden um einen vereinbarten Faktor heruntergeteilt wird um so einen Takt mit der zur Kommunikation verwendeten Datenrate zu erhalten. Der übertragene Takt beträgt 1 bis 5 MHz.
4. **RST (Reset)**: Low-aktive Steuerleitung über die der Reader einen Reset der Smartcard ausführen kann.
5. **IO**: Bidirektionale, Low-aktive Kommunikationsleitung die von Smartcard und Reader beschrieben und gelesen wird. Grundsätzlich werden dazu auf beiden Seiten Open-Collector Ausgangstreiber eingesetzt.

Zur bidirektionalen Kommunikation wird die IO-Signalleitung über einen Pullup-Widerstand im High-Zustand gehalten. Beide Teilnehmer kommunizieren dann, vergleichbar mit dem I²C-Bus, indem sie Nullen auf den Bus schreiben, das heißt diesen in den Low-Zustand versetzen. Dazu wird jeweils ein Open-Collector Ausgangstreiber verwendet. Dieser kann folgende Signale ausgeben:

1. **TriState** (hochohmig): Ausgang hat keinen Einfluss aus den Bus.
2. **Null**: Treibt ein Low-Signal. Strom fließt vom Pullup-Widerstand durch den Open-Collector Ausgangstreiber ab.

Keiner der Teilnehmer sendet aktiv einen High-Pegel (vgl. Effing 2008, S. 74). So ist gewährleistet, dass beide nicht gegeneinander treiben können, denn dies würde zu einem Kurzschluss und der Beschädigung der Hardware führen.
Der für diese Arbeit verwendete Reader ist diesbezüglich ein Ausnahmefall. Denn er verfügt hardwaretechnisch nicht über Open-Collector Ausgangstreiber. Stattdessen schreibt er sowohl High-Pegel, als auch Low-Pegel aktiv auf den Bus. Infolgedessen wird kein Pullup-Widerstand verwendet, sondern ein Widerstand zur Strombegrenzung, welcher am Ausgang des Readers sitzt. Auf die Sicht der Smartcard auf den Bus hat dieser Umstand keinen Einfluss.
Näheres zur Busbeschaltung und den verwendeten Widerständen folgt in Abschnitt „7.1 Testaufbau“.

Die IO-Signalleitung kann von der Smartcard, außer zur Datenübertragung, auch dazu verwendet werden zu Testzwecken Triggersignale auszugeben. Über diese wird beispielsweise das Erreichen eines bestimmten Zustands der Smartcard angezeigt. Der Reader kann derartige Triggersignale von übertragenen Datenbits anhand deren deutlich geringerer Länge unterscheiden und daraus entsprechende Triggersignale für Fehlerinjektionsattacken oder Seitenkanalanalysen generieren.
Ergänzend zur IO-Signalleitung kann eine Smartcard, abhängig vom verwendeten Mikrocontroller, weitere Ein- und Ausgänge haben, welche dann anstelle der IO-Signalleitung zur Triggerausgabe verwendet werden können. Es sei angemerkt, dass derartige zusätzliche Ports von fertig entwickelten Smartcards nicht benötigt werden und aus Sicherheitsgründen deaktiviert sind. Sie sind grundsätzlich vorhanden, da die für Smartcards verwendeten Mikrocontroller auch in anderen Anwendungsgebieten, zum Beispiel der Logistik, eingesetzt werden, wo eine Kommunikation mit zusätzlicher Hardware erforderlich ist. Dort dienen die zusätzlichen Ports zum Beispiel zur Realisierung einer I²C-Schnittstelle, über die verschiedene Sensoren ausgelesen werden.

### Pegelwandlung

Neben den mit 3.3 V arbeitenden Smartcards, die direkt mit dem Reader verbunden werden können, existieren Smartcard-Typen, die mit einer anderen Spannung im Bereich von 1 V bis 5 V arbeiten. Dies muss bei der Entwicklung des Triggergenerators jedoch nicht weiter berücksichtigt werden. Denn um die Kommunikation eines auf 3.3 V festgelegten Readers mit einer Smartcard, welche eine andere Spannung verwendet, zu ermöglichen, wird eine Schaltung zur Pegelwandlung eingesetzt. Diese wird zwischen Smartcard und Reader bzw. zwischen Smartcard und Triggergenerator geschaltet und nimmt die Umwandlung von der Smartcard-Spannung auf 3.3 V und umgekehrt vor. Bezüglich der Kommunikation verhält sich die Pegelwandlerschaltung für beide Teilnehmer transparent, weshalb sie vom Triggergenerator ignoriert werden kann. Aus Sicht des Triggergenerators, ist dieser immer direkt mit Smartcard und Reader verbunden.
Da diese Schaltung eine galvanische Entkopplung von Reader und Smartcard vornimmt, das heißt eine Trennung der elektrischen Verbindungen zwischen beiden, bleibt der Reader von Fehlerinjektionsattacken auf die Smartcard unberührt. Erzeugt beispielsweise ein Pulsgenerator eine Spannungsspitze in der Stromzufuhr der Smartcard (siehe „2.2 Fehlerinjektionsattacke“), gelangt diese Spannungsspitze nicht über die PWR-Leitung (siehe Abb. 3, S. 17) zum Reader. Die galvanische Entkopplung erfolgt über Optokoppler, welche elektrische Impulse in Form von Lichtimpulsen übertragen.

## Die Smartcard

Die Smartcard basiert auf einem Mikrokontroller der „NXP SmartMX P40“-Familie und besitzt unter anderem einen eigenen RAM, einen ROM und verwendet einen selbst-erzeugten Systemtakt. Die Prototypen der Smartcard-Mikrokontroller werden, zur einfacheren Handhabung während der Tests, in einem Keramikträger untergebracht, wie er in Abb. 2 (siehe S. 16) zu sehen ist.
Die Smartcard enthält eine Software, die Befehle, wie zum Beispiel Rechenaufträge, über ein Kommunikationsprotokoll empfängt, die entsprechende Berechnungen durchführt und das Ergebnis zurücksendet. Üblicherweise besteht ein Rechenauftrag aus mehreren Nachrichten: Im ersten Schritt werden der Smartcard Speicheraufträge mit Rechenparametern übermittelt, die sie in ihren Speicher lädt, woraufhin sie dem Reader jeweils mit einer Statusmeldung antwortet. Anschließend wird der Befehl zum Start einer Berechnung, unter Benutzung der zuvor gespeicherten Parameter, zur Smartcard übertragen. Daraufhin wird eine Rechenfunktion im Smartcard-Betriebssystem aufgerufen (z.B. eine Verschlüsselungsfunktion) und das berechnete Ergebnis an den Reader zurückgeschickt.

## Triggersignale der Smartcard

Die Ausgabe eines Triggersignals durch die Smartcard kann vom Smartcard-Tester, zu Testzwecken, vorübergehend an beliebige Stellen in die Software der Smartcard eingefügt werden. Üblicherweise werden derartige Triggersignale am Anfang zu testender Funktionen ausgegeben, um die Smartcard anzeigen zu lassen, wann sie mit der Ausführung der jeweiligen Funktion beginnt. So ist es möglich die Ausführung einer bestimmten Funktion gezielt zu manipulieren, um diese auf Sicherheitsmängel hin zu überprüfen.
Aufgrund von hardwarebedingten Einschränkungen, sind die ausgegebenen Triggersignale mindestens 1 µs lang. Da ein Trigger-Ausgabebefehl, innerhalb der Software der Smartcard, zwischen anderen Befehlen eingefügt wird, lässt sich nur der Beginn einer Instruktion über ein Triggersignal anzeigen, jedoch nicht die einzelnen Schritte einer mehrere CPU-Zyklen andauernden Instruktion. Zudem ist der Aufwand für eine Änderung der Position des Befehls zur Triggerausgabe sehr hoch, da die Umprogrammierung der Smartcard viel Zeit in Anspruch nimmt. Außerdem lässt sich ihr Speicher, hardwarebedingt, nur begrenzt oft überschreiben.
Aufgrund dieser Einschränkungen eignet sich ein von der Smartcard erzeugtes Triggersignal nur bedingt für den unmittelbaren Einsatz bei Fehlerinjektionsattacken oder Seitenkanalanalysen. Eine direkte Verwendung ist gar nicht möglich, wenn ein Triggersignal über die Kommunikations-Signalleitung übertragen wird. Dann ist in jedem Fall eine zusätzliche Filterung der Triggersignale aus den Datenübertragungen notwendig. Deshalb soll ein von der Smartcard erzeugtes Triggersignal lediglich dazu dienen dem Triggergenerator einen Ausgangszeitpunkt für die Berechnung eines genauer einstellbaren Triggersignals zu liefern.
Es sei angemerkt, dass sämtliche Befehle zur Triggerausgabe am Ende der Testphase wieder entfernt werden, da diese potentiellen Angreifern nützliche Informationen liefern würden. Zudem könnte die Kommunikation zwischen Smartcard und Reader durch die Ausgabe von Triggersignalen gestört werden.

## Ablauf der Kommunikation zwischen Smartcard und Reader

Der Reader initialisiert die Smartcard, indem er sie mit Strom versorgt und sie aus dem Resetzustand holt. Dazu setzt er die standardmäßig im Low-Zustand befindliche, Low-aktive Resetleitung auf High. Danach baut er eine Verbindung zur Smartcard auf und erfragt die höchstmögliche, unterstützte Übertragungsgeschwindigkeit. Anschließend senden beide, mit der vereinbarten Geschwindigkeit, abwechselnd Daten. Abb. 4 zeigt dazu ein vereinfachtes Beispiel.

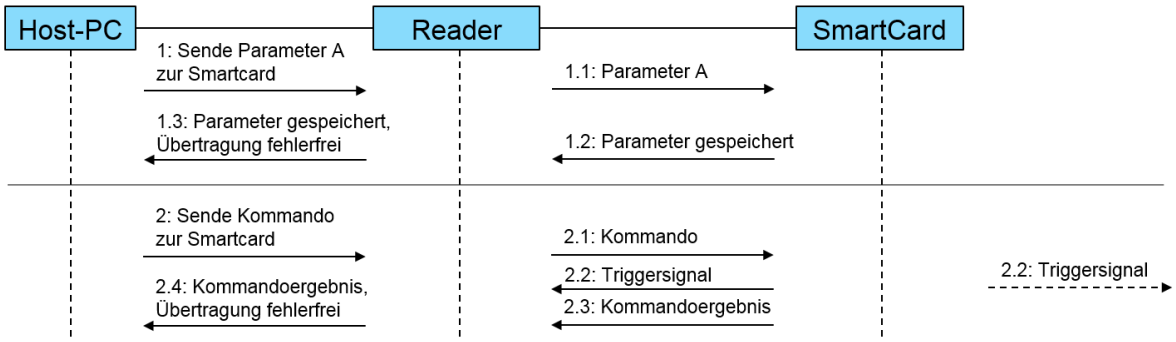


**Abb. 4: UML-Kommunikationsdiagramm: Host-PC, Reader und Smartcard Dialoge**

Der Datenaustausch zwischen Host-PC und Smartcard setzt sich üblicherweise aus mehreren Einzelübertragungen zusammen. Zuerst werden Rechenparameter zur Smartcard übertragen, anschließend folgt ein Befehl zum Start einer Berechnung mit diesen Parametern. In Abb. 4 wird der beispielhafte Ablauf zweier Dialoge zwischen Host-PC, Reader und Smartcard in Form eines UML-Kommunikationsdiagramms dargestellt. Die erste Übertragung dient dazu einen Parameter A in die Smartcard zu laden, die zweite Übertragung dazu eine Berechnung durchzuführen.
Beide Übertragungen haben folgenden Ablauf:

1. Der Reader sendet eine Anfrage an die Smartcard. Er hat einen Sendeauftrag vom Host-PC erhalten (1. und 2.) und leitet die zu übertragenden Daten, dem Kommunikationsprotokoll entsprechend, zur Smartcard weiter (1.1 und 2.1).
2. Die Smartcard empfängt die Anfrage und führt die geforderten Funktionen aus. Dafür benötigt sie mindestens 200 ms, bevor sie mit dem Senden einer Antwort beginnen kann (1.2 und 2.3). Die maximal benötigte Rechenzeit kann im Entwicklungsumfeld mehrere Minuten betragen, wenn besonders aufwändige Funktionen getestet werden.
   Während die Smartcard die geforderten Funktionen ausführt, kann sie an einen Punkt in ihrer Software gelangen, an dem der Entwickler die Ausgabe eines Triggerpulses einprogrammiert hat (2.2). Dieses Triggersignal wird dann auf der Kommunikations-IO-Leitung (2.2 durchgezogene Linie) oder einer gesonderten IO-Leitung (2.2 unterbrochene Linie) ausgegeben.
3. Der Reader empfängt die Antwort der Smartcard und sendet sie an den Host-PC weiter (1.3 und 2.4). Dabei wird die Antwort der Smartcard um Statusmeldungen des Readers zum Verlauf der Kommunikation mit der Smartcard erweitert.

## 3.2 Allgemeine Anforderungen

Die folgenden Abschnitte beschreiben die geforderten Funktionen des Triggergenerators anhand von zwei Anwendungsbeispielen. Ein Ziel der Entwicklung ist die Realisierung dieser beiden Beispiele. Für die Implementierung soll ein FPGA (Field Programmable Gate Array) verwendet werden, da dieser einem Mikrocontroller bei zeitkritischen Anwendungen überlegen ist.

Der Triggergenerator soll Triggersignale erzeugen, die anzeigen, wann Sequenzen von bestimmten Signalzuständen der Signalleitungen zwischen Smartcard und Reader, aufgetreten sind.
Im einfachsten Fall kann der momentan übertragene Pegel einer Signalleitung als deren Zustand bezeichnet werden. Über einen längeren Signalverlauf hinweg, nimmt eine Signalleitung unterschiedliche Zustände zu unterschiedlichen Zeitpunkten ein. Ein Triggersignal soll nun anzeigen, wann ein bestimmter Zustand in einem bestimmten Teil eines bekannten Signalverlaufs erreicht wurde. Es wird eine Triggerbedingung definiert, welche eine Signalleitung beobachtet. Sobald der gewünschte Pegel oder Signalverlauf übertragen wurde, ist die Bedingung erfüllt und ein Triggersignal wird ausgegeben. Um mehrere Signalleitungen zu betrachten, werden mehrere Triggerbedingungen definiert. Diese können entweder gleichzeitig arbeiten oder in einer Sequenz nacheinander. Eine solche Sequenz von Triggerbedingungen wird als mehrstufige Triggerbedingung bezeichnet.
Die in einer mehrstufigen Triggerbedingung enthaltenen Triggerbedingungen müssen in einer bestimmten Reihenfolge durchlaufen werden, damit die gesamte Bedingung erfüllt ist und ein Triggersignal erzeugt wird. Die Teilbedingung einer mehrstufigen Triggerbedingung wird im Folgenden als Unterbedingung bezeichnet.
Eine mehrstufige Triggerbedingung wird vom Anwender so definiert, dass die Smartcard nach Erfüllung aller Unterbedingungen in einen Zustand gerät, in dem ein Angriff mittels eines Triggersignals gestartet werden soll. Denn da kein Zugriff auf die interne Schaltung der Smartcard möglich ist, kann nur über die Signalverläufe zwischen Smartcard und Reader auf ihren Zustand geschlossen werden.
Nach Konfiguration einer mehrstufigen Triggerbedingung im Triggergenerator, beginnt dieser mit der Beobachtung der beteiligten Signalleitungen. Nachdem eine Unterbedingung erfüllt wurde, wird die nächste abgearbeitet, solange bis alle erfüllt sind und ein Triggersignal ausgegeben werden kann. Der Benutzer definiert wann das anschließende Triggersignal ausgegeben werden soll und wie lang es sein soll. Denn je nach Anwendungszweck kann es notwendig sein die Triggerausgabe zu verzögern.
Der Triggergenerator soll mehrere mehrstufige Triggerbedingungen gleichzeitig abarbeiten können, wobei jede aus einer individuellen Anzahl verschiedenartiger Unterbedingungen bestehen kann.

### 3.2.1 Unterbedingungsarten

Jede Unterbedingung beobachtet eine Signalleitung. Mehrere Unterbedingungen können die gleiche Signalleitung beobachten. Der Typ einer Unterbedingung bestimmt den Signalverlauf, der sie erfüllt. Folgende sechs Unterbedingungstypen sollen implementiert werden:

1. **Highstate**: Signal muss im Zustand High sein.
2. **Lowstate**: Signal muss im Zustand Low sein.
3. **HighEdgeCount**: Eine definierte Anzahl steigende Flanken muss aufgetreten sein.
4. **LowEdgeCount**: Eine definierte Anzahl fallende Flanken muss aufgetreten sein.
5. **HighPulseDetect**: Ein High-Puls mit einer definierten Länge muss aufgetreten sein.
6. **LowPulseDetect**: Ein Low-Puls mit einer definierten Länge muss aufgetreten sein.

Für das erste Anwendungsbeispiel sei eine mehrstufige Triggerbedingung definiert, welche aus einer Sequenz der folgenden vier Unterbedingungen besteht:

Unterbedingung 1: Highstate(VDD) → Signalleitung VDD muss im High-Zustand sein.
Unterbedingung 2: Highstate(RST) → Signalleitung RST muss im High-Zustand sein.
Unterbedingung 3: LowPulseDetect(IO) → Low-Puls wurde auf Signalleitung IO übertragen.
Unterbedingung 4: HighEdgeCount(CLK) → Flanken wurden auf Signalleitung CLK übertragen.

Jede Unterbedingung enabled die darauffolgende für einen unbegrenzten Zeitraum. Denn es ist nicht absehbar, wie lange es dauert bis die jeweils nächste erfüllt wird. Abb. 5 zeigt Beispielsignale, welche alle vier Unterbedingungen in der festgelegten Reihenfolge erfüllen. Die gelbe Linie gibt den Zeitpunkt an, zu dem ein Triggersignal vom Triggergenerator erzeugt werden soll.

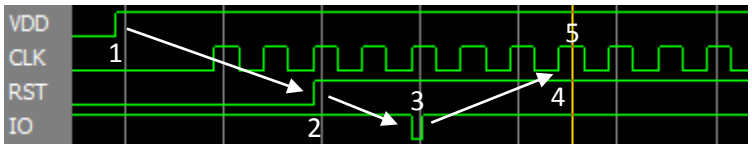


Abb. 5: Signale zur Erfüllung der mehrstufigen Triggerbedingung

In diesem vereinfachten Beispiel wurde eine mehrstufige Triggerbedingung definiert, um auf den Eintritt der Sequenz der in Abb. 5 dargestellten Ereignisse reagieren zu können:

1. VDD-Signalleitung ist High → Die Smartcard wird vom Reader mit Strom versorgt.
2. RST-Signalleitung ist High → Die Smartcard wurde vom Reader aus dem Resetzustand geholt und damit in den Betriebszustand versetzt.
3. IO-Signalleitung hat einen Low-Puls mit bestimmter Länge übertragen → Die Smartcard hat über einen Triggerpuls den Ausführungsbeginn einer bestimmten Funktion bekanntgegeben.
4. CLK-Signalleitung hat drei steigende Flanken übertragen → Der Reader hat 3 Kommunikationstakte ausgegeben, welche in diesem Beispiel zur Verzögerung der Triggerausgabe verwendet werden sollen.
5. Die Zeitspanne von der High-Flanke (4) bis zum Triggerzeitpunkt (5) ist abgelaufen → Der Triggergenerator hat eine bestimmte, möglichst genau aufgelöste, Zeitspanne gewartet um die Ausgabe des Triggersignals zu verzögern.

Obwohl die Unterbedingungen 1 und 2 in diesem Beispiel durch eine logische UND-Verknüpfung der Signalleitungen VDD und RST realisiert werden könnten, soll die Implementierung stets eine Sequenz von unabhängigen Einzelabfragen der Signalleitungen ausführen. Dies ermöglicht die im nächsten Abschnitt dargestellte Anforderung, zeitliche Abstände zwischen die einzelnen Unterbedingungen einfügen zu können.

Dieses Beispielszenario hat zum Zweck ein Triggersignal zu erzeugen, welches auf Basis eines Triggersignals von der Smartcard berechnet wird. Die Smartcard zeigt mit der Ausgabe eines Triggersignals auf der IO-Signalleitung das Erreichen eines bestimmten Zustands an. Beispielsweise, dass sie sich am Anfang der Ausführung einer bestimmten Funktion befindet. Von diesem Triggersignal ausgehend soll der Triggergenerator eine möglichst präzise definierbare Zeitspanne abwarten, in der die Smartcard mit der Ausführung ihrer Funktion fortfährt. Nach Ablauf dieser Zeitspanne soll der Triggergenerator ein Triggersignal erzeugen, um anzuzeigen, dass sich die Smartcard nun in einem bestimmten hinteren Teil ihrer Funktion befindet. Die Zeitspanne kann dabei eine Länge von bis zu zwei Minuten haben, weshalb die Taktflanken des Kommunikationstaktsignals verwendet werden sollen, um Zeitspannen einer Länge von mehr als 1 ms abzuwarten.
Die Unterbedingungen 1 und 2 sorgen dafür, dass die Vorbedingungen erfüllt sind. Das bedeutet, die Smartcard muss mit Strom versorgt werden (1) und darf nicht mehr im Resetzustand (2) sein. Unterbedingung 3 wartet auf das Triggersignal der Smartcard (3), welches verzögert werden soll. Unterbedingung 4 beginnt nach dessen Auftreten mit der Zählung von Kommunikationstakten und gibt das Triggersignal der Smartcard so um drei Takte (4) verzögert aus, zuzüglich einer kleinen, möglichst präzise programmierbaren, Ausgabeverzögerung (5) von bis zu 1 ms.
Der gesamte, zeitliche Abstand vom Triggersignal der Smartcard (3) bis zur Triggerausgabe durch den Triggergenerator (5) setzt sich also aus einer bestimmten Anzahl von Kommunikationstakten und einer zusätzlichen Ausgabeverzögerung, von bis zu 1 ms, zusammensetzen.

Einige Anwendungsgebiete machen es erforderlich, nicht nur am Ende einer mehrstufigen Triggerbedingung (4) eine zeitliche Verzögerung (Offset) zur Triggerausgabe (5) angeben zu können, sondern auch zwischen den Unterbedingungen. So muss es möglich sein einen zeitlichen Offset vom Erfüllen einer Unterbedingung bis zum Enable der nächsten Unterbedingung definieren zu können, worauf der folgende Abschnitt eingeht.

### 3.2.2 HoldOff- und Window-Offsets

Bei der Definition einer mehrstufigen Triggerbedingung ist es notwendig für jede Unterbedingung ein individuelles Zeitfenster angeben zu können innerhalb dessen die Unterbedingung erfüllt werden kann. Der Beginn eines solchen Zeitfensters, im Folgenden Window-Offset genannt, ist der Zeitpunkt zu dem die vorherige Unterbedingung erfüllt wurde, zuzüglich eines zeitlichen Offsets, im Folgenden HoldOff-Offset genannt. Eine Unterbedingung ist nur innerhalb ihres Window-Zeitfensters enabled. Abb. 6 zeigt am Beispiel einer zweistufigen Triggerbedingung, wann die beteiligten Unterbedingungen A und B, bei Verwendung von HoldOff und Window-Offsets, jeweils enabled wären.

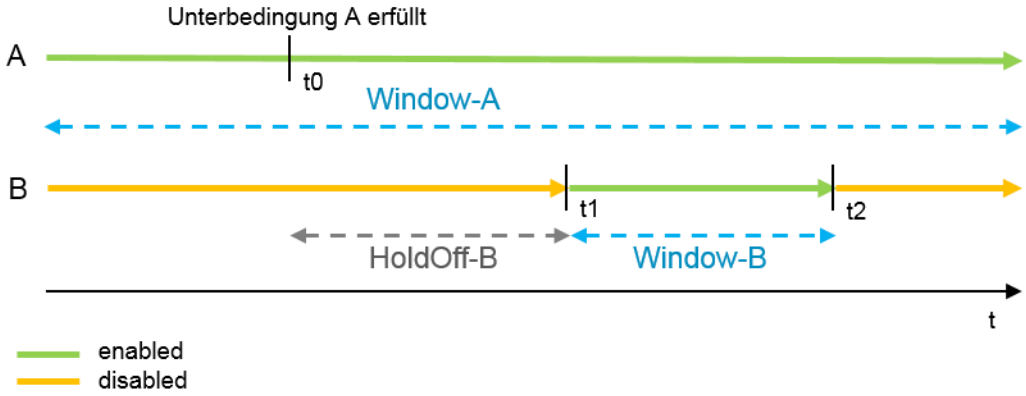


Abb. 6: Beispiel für HoldOff und Window-Offsets

Wie in Abb. 6 zu sehen ist, verfügt Unterbedingung A über ein Window-Zeitfenster von unbegrenzter Länge (Window-A) und kann damit jederzeit erfüllt werden. Denn als erste Unterbedingung, muss sie unbedingt enabled sein, da es keine andere Unterbedingung vor ihr gibt, von der sie enabled werden könnte. Zum Zeitpunkt t0 tritt ein, Unterbedingung A erfüllendes, Ereignis auf der von ihr beobachteten Signalleitung ein. Unterbedingung B soll jedoch nicht unmittelbar als Folge davon enabled werden. Dies wird um eine Zeitspanne (HoldOff-B) der Länge t1 - t0 verzögert. Erst dann soll ein Zeitfenster der Länge t2 – t1 beginnen, innerhalb dessen Unterbedingung B enabled ist (Window-B) und erfüllt werden kann.
Grundsätzlich sollen jeweils ein HoldOff-Offset und ein Window-Offset mit einer Länge von bis zu 1 ms für jede Unterbedingung individuell angegeben werden können und möglichst präzise umgesetzt werden. Dieser Wertebereich ist, laut Erfahrung der Smartcard-Tester, für alle praxisrelevanten Anwendungen ausreichend. Ein Window-Offset soll alternativ auch eine unbegrenzte Länge haben werden können, so wie es in diesem Beispiel bei Unterbedingung A notwendig ist.
Auf das letzte Beispiel (siehe Abb. 5, S. 22) zurückblickend, wurde dort für alle vier Unterbedingungen ein HoldOff-Offset der Länge 0 und ein Window von unbegrenzter Länge benötigt, da die Ereignisse, welche die Unterbedingungen erfüllten, nicht zu einem bestimmten Zeitpunkt erwartet werden konnten. Denn es war nur vorhersagbar in welcher Reihenfolge, nicht aber, in welchen zeitlichen Abständen die Unterbedingungen erfüllt werden würden.
Grundsätzlich darf ein einmal gestarteter Ablauf der beiden Offsets nicht durch ein erneutes triggern der Vorbedingung unterbrechbar sein. Auch, wenn Unterbedingung A in diesem Beispiel zwischen den Zeitpunkten t0 und t2 wiederholt erfüllt werden sollte, muss Unterbedingung B im Zeitraum t1 bis t2

enabled sein. Erst nach dem Zeitpunkt t2 kann Unterbedingung B erneut von Unterbedingung A enabled werden.
Ein praxisrelevantes Beispiel für die Verwendung der HoldOff und Window-Offsets zeigt Abb. 7.

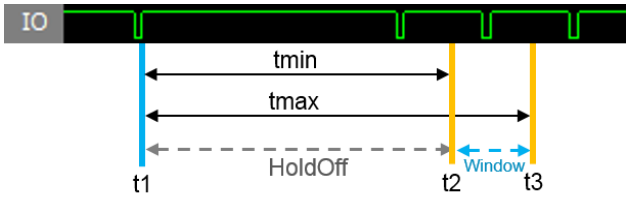


Abb. 7: HoldOff und Window-Offset Motivation

Das in Abb. 7 dargestellte Signal überträgt vier Pulse. Zwischen dem ersten und dem letzten Puls könnten noch weitere Pulse übertragen werden (nicht eingezeichnet). Der zeitliche Abstand zwischen den Pulsen ist bekannt und nahezu konstant. Ziel ist die Ausgabe eines Triggersignals, nachdem der dritte eingezeichnete Puls übertragen wurde. Vom ersten Puls zum Zeitpunkt t1 ausgehend, tritt dieser frühestens nach der Zeitspanne tmin innerhalb eines Zeitfensters der Länge $tmax - tmin$ auf.
Zur Erkennung sollen zwei Unterbedingungen des Typs LowPulseDetect eingesetzt werden. Während die Erste unbedingt enabled ist, um den ersten Puls zum Zeitpunkt t1 zu erkennen, wird die Zweite nach einem HoldOff-Offset, ab dem Zeitpunkt t2, bis zum Zeitpunkt t3 enabled. Dabei sind t2 und t3 so gewählt, dass diese so nah um den dritten Puls herum liegen, dass innerhalb des Window-Offsets keine anderen Pulse auftreten können.
Das in Abb. 7 gezeigte Signal ist Teil eines komplexeren Anwendungsbeispiels, welches in Abb. 8 dargestellt ist.

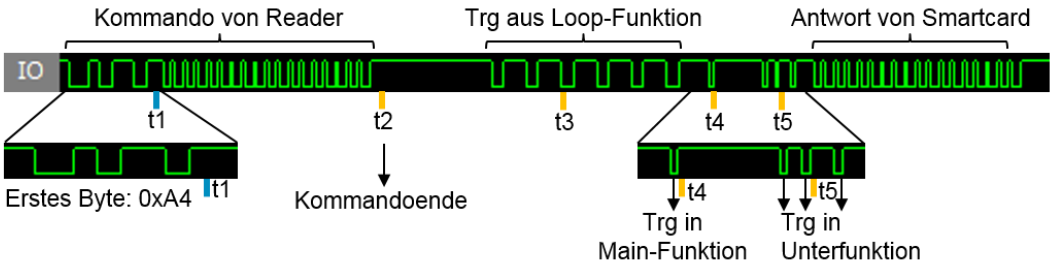


Abb. 8: Praxisrelevantes Beispiel für HoldOff/Window-Offsets

Die in Abb. 8 dargestellte Signalleitung IO überträgt sowohl Daten bidirektional zwischen Reader und Smartcard, als auch Triggersignale von der Smartcard, die vom Triggergenerator ausgewertet werden sollen. In diesem Beispiel wird zuerst ein Kommando vom Reader zur Smartcard übertragen. Daraufhin führt die Smartcard Berechnungen durch und sendet eine Antwort zum Reader.
Zum Hintergrund dieses Beispiels sei erwähnt, dass die Smartcard beauftragt wird eine ECC-Berechnung (Abk. für Elliptic Curve Cryptosystem, vgl. Tunstall 2012, S. 137) durchzuführen. Aus Platzgründen wird hier nicht näher auf das Verfahren eingegangen, zwei Details dazu fördern jedoch das Verständnis des übertragenen Signals. Zum Ersten empfängt diese Berechnung einen Parameter mit zufälliger Länge, weshalb die Länge des vom Reader zur Smartcard übertragenen Kommandos nicht vorhersehbar ist. Zum Zweiten hängt die von der Smartcard benötigte Rechenzeit von diesem zufälligen Parameter ab, weshalb nicht vorhersehbar ist, in welchem genauen Abstand die Antwort der Smartcard auf das Kommando des Readers folgt.
Während des Dialogs zwischen Smartcard und Reader sollen fünf Triggersignale, zu den Zeitpunkten t1 bis t5, ausgegeben werden. Zunächst muss das Kommando anhand seines ersten Bytes identifiziert werden, denn nur bei einer ECC-Berechnung sollen Triggersignale ausgegeben werden. In diesem Beispiel muss das Kommando mit dem Byte 0xA4 beginnen. Wurde dieses erkannt (t1) wird dies durch Ausgabe eines Triggersignals bestätigt. Auf das erste Byte folgt eine nicht bekannte Anzahl weiterer

Daten, nach deren Ende (t2) das nächste Triggersignal ausgegeben werden soll. Nach Übertragung des Kommandos beginnt die Smartcard mit ihren Berechnungen.
Der Smartcard-Entwickler hat, zur Anzeige verschiedener Zustände der Smartcard, zu Testzwecken mehrere Triggerausgaben in die Rechenfunktionen einprogrammiert. Es sei angemerkt, dass diese Triggerpulse um ein Zehnfaches kürzer sind, als der kürzest mögliche Datenpuls und dadurch vom Reader erkannt und ignoriert werden können. Die ersten Triggersignale der Smartcard werden von einer Schleife aus deren Funktion zur Parameterverarbeitung ausgegeben. Da es sich bei dem übermittelten Parameter um einen Zufallswert handelt, ist nicht vorhersagbar, wie oft diese Schleife durchlaufen wird und dabei jedesmal einen Triggerpuls ausgibt. Unabhängig davon soll ein Triggersignal zu Beginn des dritten Schleifendurchlaufs (t3) ausgegeben werden.
Nach Beendigung der Parameterverarbeitung kehrt die Smartcard in deren Main-Funktion zurück und gibt dort einen weiteren Triggerpuls (t4) aus. Dieser kann von den vorherigen Triggerpulsen durch dessen geringere Länge unterschieden werden.
Äquivalent zu dem Beispiel in Abb. 7 folgt auf den Trigger der Main-Funktion eine Reihe aus drei weiteren, nahe beieinanderliegenden Triggerpulsen, welche aus verschiedenen Unterfunktionen der Smartcard ausgegeben werden. Auch in diesem Beispiel soll ein Triggersignal ausgegeben werden, wenn der mittlere Smartcard-Trigger aufgetreten ist.
Bei der Umsetzung dieses Anwendungsbeispiels ist zu beachten, dass innerhalb verschiedener Kommandos das Byte 0xA4 vorkommt. Somit muss es möglich sein zu erkennen, ob ein erkanntes Byte 0xA4 wirklich am Anfang des jeweiligen Kommandos liegt. Zur Erkennung des Bytes an sich können zum Beispiel drei Unterbedingungen des Typs LowPulseDetect eingesetzt werden, wie Abb. 9 zeigt.

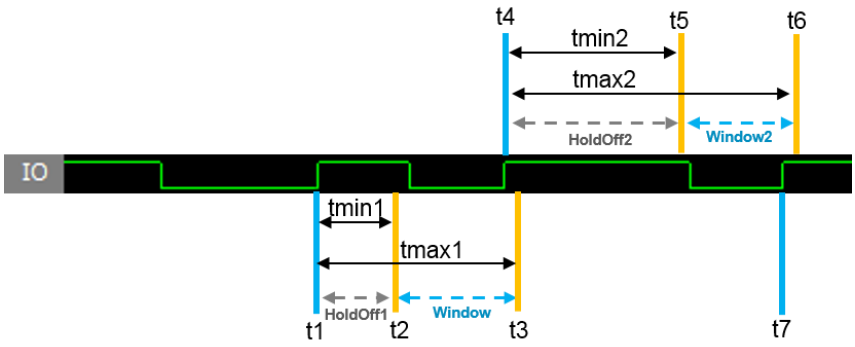


Abb. 9: HoldOff/Window-Offsets zur Erkennung einer Pulsfolge (0xA4)

Zu erkennen ist eine Folge aus drei Pulsen mit bestimmten Längen und Abständen zueinander. Die erste Unterbedingung sei in diesem Beispiel unbedingt enabled. Nachdem diese den ersten Low-Puls zum Zeitpunkt t1 erkannt hat, beginnt die HoldOff-Zeitspanne der Länge $tmax1 - tmin1$ bis zum enable der zweiten Unterbedingung. Sobald diese den zweiten Puls erkannt hat (t4), beginnt der HoldOff-Offset bis kurz vor den erwarteten Beginn (t5) der des dritten Pulses. Zum Zeitpunkt t7 erfolgt schließlich die Ausgabe des Triggersignals zur Bestätigung, dass die erwartete Pulsfolge erkannt wurde.
Auf Abb. 8 zurückblickend muss, nach Erkennung des Kommandoendes, auf den Beginn des dritten Schleifen-Triggerpulses (t3) gewartet werden. Hier könnte eine Unterfunktion des Typs LowEdgeCount zum Einsatz kommen.
Zur Erkennung des Triggerpulses in der Main-Funktion (t4) kann wieder eine LowPulseDetect-Unterbedingung benutzt werden, welche auf die individuelle Länge dieses Pulses eingestellt ist und zum Zeitpunkt des Kommandoendes (t2) enabled wird.
Das letzte Triggersignal in Abb. 8 wird zum Zeitpunkt t5 ausgegeben, nachdem der mittlere Smartcard-Trigger, wie in Abb. 7 beschrieben, über einen HoldOff-Offset zu t4 erkannt wurde.
Die verfügbaren Wertebereiche für HoldOff und Window-Offsets werden im Abschnitt „3.3 Spezifische funktionale Anforderungen“ definiert.

### 3.2.3 Signalweiterleitung

Die Position des Triggergenerators, zwischen Smartcard und Reader, in seiner späteren Laufzeitumgebung ist durch die Notwendigkeit begründet bestimmte Signale unterbrechen und filtern zu können. Abb. 10 zeigt eine vereinfachte Darstellung der Integration des Triggergenerators in das bestehende System.

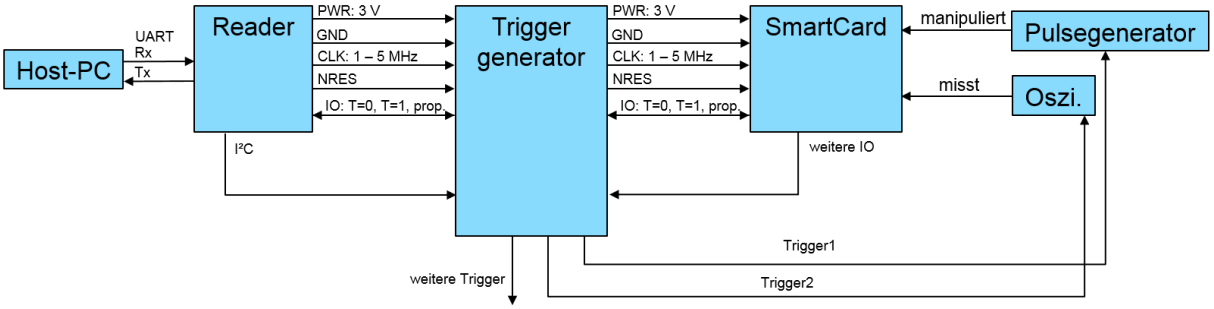


**Abb. 10: Vereinfachte Darstellung der Laufzeitumgebung des Triggergenerators**

Der Triggergenerator soll die zwischen Smartcard und Reader verlaufenden Signalleitungen uni- oder bidirektional durchleiten können. Die in Abb. 10 eingezeichneten Pfeile zeigen die Datenflussrichtungen der Signalleitungen an. Dadurch ist der Triggergenerator in der Lage Signalleitungen unterbrechen und bestimmte Pulse herauszufiltern zu können. Neben den bei der Durchleitung beteiligten Ein- und Ausgängen des Triggergenerators, verfügt dieser über weitere Eingänge zum Empfang von Triggersignalen, welche die Smartcard generiert. Die vom Triggergenerator erzeugten Triggersignale werden über gesonderte Ausgänge ausgegeben.
Zu Analysezwecken soll es möglich sein die Smartcard für eine bestimmte Zeitspanne in einem Zustand festzuhalten. Dies kann nur für einige, vom Kommunikationstakt abhängige, Zustände erreicht werden, indem die Übertragung des Clock-Signals, vom Clock-Ausgang des Readers zum Clock-Eingang der Smartcard, pausiert wird. Es soll eine mehrstufige Triggerbedingung definiert werden können um einen Zeitpunkt festzulegen, zu dem ein programmierbarer Pegel an den Clock-Eingang der Smartcard angelegt wird und nicht mehr das Clock-Signal des Readers.
Die Dauer der Signalumschaltung soll, äquivalent zum Wertebereich für Window-Offsets, für Längen von bis zu 1 ms möglichst präzise umgesetzt werden oder eine unbegrenzte Länge haben.
Die Smartcard kann programmiert werden Triggersignale auf der zur Kommunikation mit dem Reader verwendeten Daten-IO-Leitung auszugeben, sofern die Smartcard über keine weiteren IO-Signalleitungen verfügt. Es muss möglich sein die ausgegebenen Triggersignale herauszufiltern, so dass diese nicht den Reader erreichen.

Der Triggergenerator ist ein Werkzeug für Smartcard-Tester, ohne tiefgehende VHDL[5]-Kenntnisse, das an spezifische Testumstände angepasst werden können muss. Die für, jeden Test individuellen, Einstellungen sollen dabei zur Laufzeit übertragen werden können. Nur bei größeren und damit selteneren Anpassungen des Triggergenerators soll es nötig sein den Quellcode der Schaltung modifizieren und erneut übersetzen zu müssen.

[5] Abk. VHDL = Very High Speed Integrated Circuit Hardware Description Language

## 3.3 Spezifische funktionale Anforderungen

Der Triggergenerator muss zum bestehenden System kompatibel sein (siehe Abb. 2, S. 16). Sein Funktionsumfang kann, in Absprache mit den verantwortlichen Smartcard-Testern, aufgrund absehbarer Praxisrelevanz konzeptionell eingeschränkt werden. Dadurch können Hardware-Ressourcen eingespart werden, welche für Skalierungen der Schaltung zur Verfügung stehen.
Die folgenden Abschnitte definieren die erforderliche Dimensionierung der Funktionen des Triggergenerators. Die Schaltung muss mindestens diesen Anforderungen genügen, darf, sofern es sinnvoll erscheint, aber auch leistungsfähiger sein:

- 16 Eingänge zur Triggerberechnung, 8 Ausgänge zur Triggerausgabe,
  4 Ausgänge für weiterzuleitende Signale
- Filterung von Smartcard-Triggerpulsen:
  - Wertebereich des Filtergrenzwertes: 1 µs bis 1 ms
  - Schrittweite für Definition: < 100 ns
  - Tolerierte Abweichung eingestellter zu gefilterter Pulslänge: ± 100 ns
  - Tolerierte Schwankung der Länge nicht gefilterter Datenpulse: ± 100 ns
  - Zulässige Verzögerung des gefilterten Signals: Filtergrenzwert ± 1 µs
- Unterbrechung der Signaldurchleitung:
  - Dauer der Unterbrechung: 100 ns bis 1 ms
  - Schrittweite für Definition: < 100 ns
  - Tolerierte Abweichung eingestellter zu erreichter Dauer: ± 100 ns
  - Alternativ soll Unterbrechung mit unbegrenzter Länge möglich sein
- Unterbedingungen:
  - Rechenzeit zur Erkennung erfüllter Bedingung: < 100 ns
  - Tolerierte Abweichung von durchschnittlicher Rechenzeit: ± 10 ns
- Mehrstufige Triggerbedingung:
  - Ausgabelänge Triggersignal: 100 ns bis 1 ms oder unbegrenzt
  - Schrittweite für Definition: < 100 ns
  - Tolerierte Abweichung eingestellter zu erreichter Dauer: ± 100 ns
- HoldOff-Offset:
  - Länge des Offset: 0 ns bis 1 ms
  - Schrittweite für Definition: < 100 ns
  - Tolerierte Abweichung eingestellter zu erreichter Dauer: ± 100 ns
- Window-Offset:
  - Länge des Offset: 100 ns bis 1 ms oder unbegrenzt
  - Schrittweite für Definition: < 100 ns
  - Tolerierte Abweichung eingestellter zu erreichter Dauer: ± 100 ns
- I²C-Übertragung mit maximaler Übertragungsgeschwindigkeit 300 kBaud/s

### 3.3.1 Ein-/ und Ausgänge

Der Triggergenerator soll über 16 Eingänge und 8 Ausgänge verfügen. An den Eingängen liegen alle vom Reader kommenden Signalleitungen an, die zur Smartcard durchgeleitet werden sollen. Außerdem liegen hier (optional) zusätzliche, von der Smartcard kommende, IO-Leitungen an, auf denen diese Triggersignale ausgibt. Alle Eingangssignale können von Triggerbedingungen ausgewertet werden.
Bis zu vier Ausgänge geben die durchgeleiteten Signale an die Eingänge der Smartcard aus. Die übrigen acht Ausgänge dienen dazu Triggersignale auszugeben.
In Abb. 10 (siehe S. 26) ist folgende Beschaltung der Ein- und Ausgänge (bzw. Ports) des Triggergenerators dargestellt:

- Der Reader gibt die Signale: PWR (Power), GND (Ground), CLK (Clock) und NRES (Not-Reset) unidirektional an die Eingänge des Triggergenerators aus.
- Der Triggergenerator gibt diese über seine Ausgänge unidirektional an die Eingänge der Smartcard aus.
- Der Reader gibt das IO-Signal an einen bidirektionalen Port des Triggergenerators aus. Dieser gibt es über einen bidirektionalen Port an den Eingang der Smartcard aus.
- Die Smartcard gibt optional weitere IO-Signale an die Eingänge des Triggergenerators aus.
- Der Triggergenerator gibt erzeugte Triggersignale auf seinen Triggerausgängen aus.
- Die Einstellungsdaten erhält der Triggergenerator über zwei, von der Triggerfunktionalität unabhängige, Signalleitungen vom Reader. Einen unidirektionalen Eingang zum Empfang des SCL-Signals und einen bidirektionalen Port für die SDA-Signalleitung. Diese sollen, im Gegensatz zu allen anderen Eingängen, nicht für die Auswertung von Triggerbedingungen verwendet werden.

Zur Durchleitung der bidirektionalen IO-Leitung benötigt der Triggergenerator zwei bidirektionale Ports, einen auf Seite des Readers und einen auf Seite der Smartcard. Da, bei unveränderter Durchleitung, über beide Ports das selbe Signal läuft, würde es zur Verarbeitung des IO-Signals durch Triggerbedingungen ausreichen, wenn einer der beiden Ports als Eingang betrachtet wird und damit von Unterbedingungen beobachtet werden kann.
Sollte die Smartcard jedoch Triggerpulse auf der bidirektionalen IO-Signalleitung übertragen, könnte es nötig sein diese vor der Weiterleitung des Signals an den Reader herauszufiltern. Trotzdem sollen derartige Triggerpulse für die Auswertung von Triggerbedingungen verwendet werden können. Deshalb muss das IO-Signal in ungefilterter Form, wie es von der Smartcard empfangen wurde, von den Triggerbedingungen ausgewertet werden können.
Wenn eine Filterung von Triggerpulsen auf der IO-Signalleitung notwendig ist, muss somit der bidirektionale Port auf Seite der Smartcard zur Auswertung von Triggerbedingungen verwendet werden. Gibt die Smartcard keine Triggerpulse aus oder werden diese nicht herausgefiltert, kann alternativ auch der bidirektionale Port auf Seite des Readers zur Auswertung von Triggerbedingungen verwendet werden.

### 3.3.2 Signalweiterleitung

Die zwischen Reader und Smartcard verlaufende IO-Signalleitung wird bidirektional durch den Triggergenerator geleitet. Dabei müssen die von der Smartcard erzeugten Triggerpulse herausgefiltert werden können. In umgekehrter Richtung, vom Reader zur Smartcard hin, ist keine Filterung vorgesehen. Zudem soll eine durch Triggersignale ausgelöste Unterbrechung der Weiterleitung möglich sein.

Smartcard und Reader verfügen über Open-Collector Ausgangstreiber und übertragen Daten Low-Aktiv, weswegen ein Pullup-Widerstand im Reader die Signalleitung auf High zieht. Gefiltert werden sollen alle Low-Pulse, deren Länge einen programmierbaren Wert unterschreitet. High-Pulse dürfen auch gefiltert werden.
Die zu filternden Triggerpulse sind anhand ihrer Länge erkennbar, welche unter der kürzestmöglichen Datenpulslänge liegt. Die minimale Datenpulslänge entspricht nicht der Kommunikationstaktperiode von 1 bis 5 MHz, da Reader und Smartcard einen vereinbarten Taktteiler verwenden (siehe „2.4 Kommunikationsprotokolle“). Gemäß der Smartcard-Entwickler beträgt die maximal erreichbare Baudrate 300 kBaud/s. Damit ist jeder Datenpuls länger als 3,3 µs. In vielen Fällen wird eine Baudrate von 115 kBaud/s verwendet, wobei jeder Datenpuls länger als 8,6 µs ist.
Die von der Smartcard erzeugten Triggerpulse haben eine gemessene Länge von 1,4 µs ± 200 ns. Damit sind Triggerpulse deutlich von Datenpulsen unterscheidbar. Die Smartcard ist hardwarebedingt nicht in der Lage kürzere Triggerpulse zu erzeugen. Längere Triggerpulse könnten, bei entsprechend

herabgesetzter Übertragungsgeschwindigkeit eingesetzt werden. Die Obergrenze der Länge zu filternder Pulse kann, nach Absprache, auf 1 ms begrenzt werden. Somit ergibt sich ein Wertebereich von 1 µs bis 1 ms für die einstellbare maximale Länge von Pulsen, die herausgefiltert werden sollen. Diese soll mit einer Schrittweite von unter 100 ns angegeben werden können.
Der Filter muss auf einen Grenzwert aus diesem Wertebereich eingestellt werden können, welcher zwischen der Länge des kürzesten Datenpulses (z.B. 3,3 µs) und der Länge eines Smartcard-Triggerpulses (z.B. 1,4 µs) liegt, wobei alle Pulse kürzer dem Grenzwert herausgefiltert werden sollen. Toleriert wird das Filtern aller Pulse mit der Länge des Grenzwertes ± 100 ns. Der Grenzwert soll so gewählt werden, dass es zur Triggerpulslänge hin, sowie zur kürzesten Datenpulslänge hin einen zeitlichen Puffer von mindestens 1 µs gibt. Ein Puffer dieser Größe in beide Richtungen reicht aus, um Ungenauigkeiten bei der Länge von der Smartcard ausgegebener Triggerpulse auszugleichen. Bei einer beispielhaften Datenpulslänge von 3,3 µs und einer Triggerpulslänge von 1,4 µs könnte der Grenzwert auf 2 µs festgelegt werden. Bei einer Datenpulslänge von 8,6 µs und einer Triggerpulslänge von 4 µs könnte der Grenzwert auf einen Wert zwischen 5 µs und 7 µs festgelegt werden.

Die Filterung von Pulsen aus einem Signal bewirkt eine konstante Verzögerung des Signals um mindestens die Höchstlänge der herauszufilternden Pulse. Denn ein Puls muss zuerst vollständig vom Filter empfangen und seine Länge gemessen worden sein, bevor entschieden werden kann, ob er weitergeleitet werden soll oder nicht. Auf einer bidirektionalen Leitung muss der Filter so dimensioniert werden, dass es durch die Verzögerung nicht zu Datenkollisionen kommen kann.
Smartcard und Reader senden abwechselnd Daten. Grundsätzlich antwortet die Smartcard unverzüglich, nachdem sie eine Anfrage vom Reader bearbeitet hat. Dazu sei jedoch bemerkt, dass durch Messungen eine Bearbeitungszeit für jede Anfrage von mindestens 200 ms ermittelt wurde.
Der Reader wartet, nach Übertragung einer Anfrage, bis zu mehrere Minuten auf das Eintreffen der Antwort von der Smartcard. Empfängt er diese, leitet er sie an den Host-PC weiter. Dieser überträgt dann die nächste Anfrage an den Reader. Erst danach beginnt der Reader eine weitere Anfrage zur Smartcard zu übertragen.
Der Filter soll nur die von der Smartcard zum Reader übertragenen Pulse filtern. Deshalb verzögert er auch nur die Übertragung der Smartcard-Antwort zum Reader. Diese filterbedingte Verzögerung von einigen µs kann jedoch ignoriert werden, da der Reader mehrere Minuten auf eine Antwort der Smartcard wartet.
Die tatsächlich erreichte Verzögerung entspricht dem gewählten Grenzwert des Filters, welcher, zuzüglich der Rechenzeit des Filters, um höchstens 1 µs überschritten werden soll.
Der Filter soll nicht herausgefilterte Datenpulse mit unveränderter Länge ausgeben. Die Toleranz für das Verändern der Länge nicht gefilterter Datenpulse beträgt ± 100 ns.

Die zwischen Reader und Smartcard verlaufende CLK-Signalleitung wird unidirektional durch den Triggergenerator geleitet. Dabei soll eine konfigurierbare, mehrstufige Triggerbedingung verwendet werden können, um die Durchleitung des CLK-Signals zu steuern. Das Erfüllen der Bedingung bewirkt eine zeitlich begrenzte Ausgabe eines definierten, logischen Pegels auf den CLK-Eingang der Smartcard. Nach Ablauf dieser Zeitspanne soll wieder das CLK-Signal an der Smartcard anliegen.
Die Länge der Zeitspanne soll zur Laufzeit eingestellt werden können. Der gewünschte Wert soll aus dem Bereich von 100 ns bis 1 ms bei einer Schrittweite von unter 100 ns gewählt werden können und mit einer Abweichung von ± 100 ns erreicht werden. Alternativ soll für eine unbegrenzte Zeitspanne umgeschaltet werden können.

### 3.3.3 Mehrstufige Triggerbedingungen und Unterbedingungen

Der Triggergenerator soll über 16 Signaleingänge verfügen. Eine Unterbedingung kann genau eine Signalleitung beobachten. Verschiedene Unterbedingungen können jedoch die selbe Signalleitung beobachten. Nur Unterbedingungen, die enabled sind, werten die ihnen zugeordnete Signalleitung hinsichtlich einer Triggerbedingung aus und können somit erfüllt werden. Innerhalb einer mehrstufigen Triggerbedingung enabled eine erfüllte Unterbedingung die in der Sequenz darauffolgende Unterbedingung. Die erste Unterbedingung einer Sequenz muss unbedingt enabled sein. Die letzte Unterbedingung einer Sequenz bestimmt den Zeitpunkt zur Ausgabe des Triggersignals auf dem der mehrstufigen Triggerbedingung zugewiesenen Ausgang des Triggergenerators.
Jede Unterbedingung benötigt, neben einem zu beobachtenden Signal und ihrem individuellen Enabled-Signal, verschiedene weitere Parameter. Diese sind spezifisch für jeden Unterbedingungs-Typ:

- **Highstate / Lowstate**: Benötigt keine zusätzlichen Parameter.
- **HighEdgeCount / LowEdgeCount**: Anzahl Flanken, die gezählt werden sollen.
- **HighPulseDetect / LowPulseDetect**: Anzahl Takte, die ein Puls mindestens lang sein muss und Anzahl Takte, die ein Puls maximal lang sein darf.

Jede Unterbedingung soll höchstens 100 ns Rechenzeit zur Erkennung einer erfüllten Bedingung benötigen. Die tatsächlich benötigte Zeitspanne soll pro Unterbedingungsart konstant sein, um sie bei der praktischen Anwendung miteinbeziehen zu können.
Wenn beispielweise die Unterbedingundsart HighEdgeCount eine Rechenzeit von 50 ns benötigen sollte, um eine steigende Signalflanke zu erkennen, dann muss in allen Anwendungsfällen von HighEdgeCount eine Verzögerung von 50 ns entstehen.
Dies ist für die Reproduzierbarkeit einer Triggerausgabe und zur Berechnung eines ggf. zusätzlich benötigten HoldOff-Offsets erforderlich, wie folgendes Rechenbeispiel demonstriert:

$$Gewünschte\ Verzögerung\ der\ Triggerausgabe = 90\ ns$$

$$Fixe\ Rechenzeit\ einer\ HighEdgeCount\ Unterbedingung = 50\ ns$$

$$Benötigter\ HoldOff\ Offset = \frac{(90\ ns - 50\ ns)\ Restverzögerung}{20\ ns\ HoldOff\ Schrittweite\ (50\ MHz)} = 2\ CPU\ Takte$$

Toleriert werden Schwankungen der Rechenzeit von ± 10 ns. Auf dieses Beispiel bezogen, darf die Triggerausgabe frühestens 40 ns nach Auftritt der Signalflanke beginnen und muss spätestens nach 60 ns beginnen.

Je nach Anwendungszweck wird eine unterschiedliche Anzahl mehrstufiger Triggerbedingungen benötigt. Der Triggergenerator soll es ermöglichen mindestens 4 mehrstufige Triggerbedingungen zu definieren, welche jeweils mindestens aus bis zu 4 Unterbedingungen bestehen können.
Jede mehrstufige Triggerbedingung kann, zur Ausgabe eines Triggersignals, einem von 4 Ausgängen des Triggergenerators zugewiesen werden. Welche Ausgänge für die Triggerausgabe reserviert sind und zur Laufzeit einer mehrstufigen Triggerbedingung zugewiesen werden können, soll vor der Synthese der Schaltung festgelegt werden.
Die Ausgabe des Triggersignals einer erfüllten, mehrstufigen Triggerbedingung erfolgt in Form eines High-Pulses mit einer einstellbaren Länge. Diese soll auf einen Wert aus dem Bereich von 100 ns bis 1 ms, bei einer Schrittweite von 100 ns, eingestellt werden können und mit einer Abweichung von maximal ± 100 ns erreicht werden.
Alternativ soll ein Triggersignal auch mit unbeschränkter Länge ausgegeben werden können. Diese Funktion wird für Fälle benötigt, in denen eine mehrstufige Triggerbedingung nur einmal erfüllt werden können soll und eine wiederholte Ausgabe des Triggersignals zu einem unerwünschten Verhalten der triggerverarbeitenden Geräte führen würde. Beispielsweise könnte der in Abb. 10 (siehe

S. 26) eingezeichnete Pulsgenerator so eingestellt sein, dass er, als Reaktion auf die steigende Flanke eines empfangenen Triggerpulses, eine Fehlerinjektion in Form einer Spannungsspitze auf die Smartcard ausübt (siehe „2.2 Fehlerinjektionsattacke“). Der Triggergenerator soll in diesem Fall die Möglichkeit bieten die Ausgabe weiterer Triggerpulse grundsätzlich oder für eine Zeitspanne von bis zu 1 ms zu unterbinden, indem die Länge eines ausgegebenen Triggerpulses erhöht wird. Denn andernfalls würde jedes erneute Erfüllen der mehrstufigen Triggerbedingung zu einer weitere Fehlerinjektion führen, was verhindert werden muss.
Mit dem Ende der Ausgabe eines Triggersignals soll die mehrstufige Triggerbedingung in den Anfangszustand versetzt werden, um erneut erfüllt werden zu können. Wird ein Triggersignal von unbegrenzter Länge ausgegeben, soll sie mithilfe eines Kommandos über die I²C-Schnittstelle in den Anfangszustand versetzt werden. Dies soll am Ende einer Sequenz von Fehlerinjektionsattacken und Seitankanalanalysen geschehen, weshalb ein einziges Kommando für das Zurücksetzen aller mehrstufigen Triggerbedingungen implementiert werden soll.

Nachdem eine Unterbedingung erfüllt wurde, soll die Aktivierung der nächsten Unterbedingung der Sequenz um einen HoldOff-Wert verzögert werden können. Der gewünschte Wert soll aus dem Bereich von 0 ns bis 1 ms, bei einer Schrittweite von 100 ns, auswählbar sein. Die erreichte Verzögerungszeitspanne soll sich um weniger als 100 ns von dem beabsichtigten HoldOff-Wert unterscheiden. Ein HoldOff-Wert von Null gibt an, dass die Ausgabe gar nicht verzögert werden soll, wobei die Toleranz auch hier 100 ns beträgt. Für jede Unterbedingung soll ein individueller HoldOff-Wert zur Laufzeit eingestellt werden können.
Eine Unterbedingung muss unbedingt enabled sein, wenn sie die erste in einer mehrstufigen Triggerbedingung ist . Da der Benutzer entscheiden muss, welche Unterbedingung die Erste sein soll, muss jede unbedingt enabled werden können.
Alternativ soll eine Unterbedingung durch eine andere Unterbedingung für eine Window-Zeitspanne von 100 ns bis 1 ms enabled werden. Die erreichte Zeitspanne soll sich um weniger als 100 ns von dem beabsichtigten Wert unterscheiden. Für jede Unterbedingung soll ein individueller Window-Wert zur Laufzeit eingestellt werden können.

### 3.3.4 Konfiguration über I²C

Zur Programmierung mehrstufiger Triggerbedingungen soll der Triggergenerator zur Laufzeit verschiedene Parameter und Kommandos über eine I²C-Schnittstelle mit einer Übertragungsrate von bis zu 300 kBaud/s vom Reader empfangen können. Alle zu übermittelnden Parameter sollen in einer einzigen I²C-Übertragung vom Reader zum Triggergenerator gesendet werden können. Ebenso sollen alle gespeicherten Parameter in einer einzigen Übertragung in umgekehrter Richtung zurückgelesen werden können. Für nachträgliche Korrekturen soll auch auf einzelne Parameter lesend und schreibend zugegriffen werden können.
Das Kommando zum Zurücksetzen aller mehrstufigen Triggerbedingungen und ggf. auch weitere Kommandos, sollen sowohl als Teil einer Übermittlung von Parametern, als auch separat übertragen werden können.

## 3.4 Spezifische nicht-funktionale Anforderungen

Der Triggergenerator ist Teil eines Systems zum Testen der Sicherheitsfunktionen neuer Smartcard-Algorithmen. Um zukünftigen Anforderungen zu genügen, werden Anpassungen in verschiedenen Bereichen in absehbarer Zeit nötig sein. Bei der Entwicklung des Triggergenerators sollen Maßnahmen ergriffen werden, um den Aufwand solcher Anpassungen möglichst gering zu halten.

Zukünftig kann es erforderlich sein die maximale Anzahl mehrstufiger Triggerbedingungen bzw. die maximale Anzahl von Unterbedingungen zu erhöhen. Da die Ressourcen des FPGA begrenzt sind, muss dann eine Umdimensionierung einzelner Bestandteile des Triggergenerators durchgeführt werden. Ressourcen können durch Reduzierung der Bitbreiten verwendeter Konfigurationsparameter und damit durch Reduzierung der Anzahl von Signalleitungen, Speicherzellen und Rechenlogikzellen gespart werden. Die Reduzierungen der Bitbreiten müssen vom Entwickler so gewählt werden, dass die daraus resultierenden Einschränkungen nicht den jeweiligen Anforderungen widersprechen.
Der Triggergenerator soll eine Umdimensionierung begünstigen, indem die Bitbreite jedes Konfigurationsparameters für jede Instanz einer Komponente über generische Attribute separat angegeben werden kann.

**Design Constraints**

Für die Entwicklung sollen die in Abschnitt 1 beschriebenen Geräte verwendet werden (siehe Abb. 1. S. 16). Des Weiteren gelten folgende Rahmenbedingungen.
Zur Entwicklung wird ein „Nexys 2“-Board der Firma Digilent mit einem „Spartan-3E 500K“-FPGA von Xilinx verwendet. Der CPU-Takt beträgt 50 MHz.
Alle zwischen Reader und Smartcard verlaufenden Signale verwenden eine Spannung von 3.3 V. Der Stromfluss ist auf allen Signalleitungen kleiner als 100 µA. Der Takt auf der Clock-Leitung beträgt 1 – 5 MHz. Die Implementierung der I²C-Schnittstelle des Triggergenerators muss mit der I²C-Schnittstelle des „ARM Cortex-M3“-Mikrocontrollers auf dem Readerboard „LPCXpresso 1769“ von NXP bei einer Baudrate von 300 kBaud/s kommunizieren können.

# 4 Lösungsansätze

Die bereits existierenden Systeme zur Triggererzeugung bieten nur einen geringen Anteil der benötigten Funktionen.
Für die Umsetzung eines FPGA basierten Triggergenerators boten sich drei unterschiedliche Lösungsansätze an. Diese werden in den folgenden Abschnitten erläutert und jeweils deren Vor- und Nachteile aufgezählt. In der anschließenden Bewertung wird der vielversprechenste Ansatz ausgewählt, welcher die Entwurfsgrundlage bildet.

## 4.1 Existierende Systeme zur Triggergenerierung

Zwei auf dem Markt befindliche, kommerzielle Lösungen bieten bereits Teile der benötigten Funktionalitäten. Der „VC-Glitcher“ der Firma riscure wurde speziell für dieses Einsatzgebiet entwickelt. Oszilloskope ergänzen die Möglichkeit Triggerbedingungen definieren zu können.

Einige Oszilloskope bieten grundsätzlich die Funktionalitäten, um als Triggergenerator dienen zu können, jedoch nicht in dem benötigten Umfang. Mit ihnen ist es nicht möglich mehrstufige Triggerbedingungen mit einer ausreichenden Anzahl von Unterbedingungen zu definieren oder mehrere mehrstufige Triggerbedingungen zu definieren, um so mehrere Triggersignale auf unterschiedlichen Leitungen auszugeben. Die von den Smartcard-Testern verwendeten Oszilloskope bieten die Möglichkeit maximal vier Unterbedingungen zu definieren und verfügen zudem nur über einen einzigen Triggerausgang.
Zur Ansteuerung mehrerer Komponenten gleichzeitg, wie zum Beispiel einem Fehlerinjektions-Laser und einer Seitenkanalanalysemessung, müssen verschiedene Triggerbedingungen definiert werden können, die nebenläufig abgearbeitet werden und ihre Triggersignale auf unterschiedlichen Leitungen ausgeben.

Der „VC-Glitcher“ der Firma riscure stellt die zentrale Steuereinheit eines Systems zur Durchführung verschiedener Fehlerinjektionsattacken und Seitenkanalanalysen auf Smartcards dar, das sowohl Fehlerinjektions-Laser und Seitenkanalanalyse-Messspulen, als auch die „riscure Inspector“-Software zur Definition von Angriffen und der Programmierung des „VC-Glitcher“, umfasst.
Der „VC-Glitcher“ übernimmt hauptsächlich die Funktion des Readers und verfügt dazu über einen Steckplatz für eine Smartcard. Er enthält einen Mikrocontroller für die Kommunikation mit der Software auf dem Host-PC und über einen FPGA, welcher die Kommunikation mit der Smartcard übernimmt und auch ein Triggersignal erzeugen kann.
Es sind ein Triggereingang und ein Triggerausgang vorhanden. Ein am Triggereingang eingehender Puls kann verzögert auf dem Triggerausgang ausgegeben werden. Das eingehende Triggersignal wird dabei entweder von der Smartcard oder einem Oszilloskop erzeugt. Die Verzögerung lässt sich zweistufig definieren: Erstens ist eine Verzögerung um eine bestimmte Anzahl von Kommunikationstakten möglich, in zweiter Instanz eine anschließende Verzögerung mit 20 ns Schrittweise.
In den Anfängen der Smartcardentwicklung war diese Art der Verzögerung vollkommen ausreichend, da der Kommunikationstakt gleichzeitig der CPU-Takt für die Smartcard war. So konnten Triggersignale instruktionsgenau erzeugt werden. Da moderne Smartcards mit einem wesentlich höheren, intern erzeugten, CPU-Takt arbeiten, ist dies grundsätzlich nicht mehr möglich.
Hinzu kommen folgende weitere Einschränkungen bei Verwendung des „VC-Glitcher“:

- **Dynamisches Verhalten während eines Angriffs nicht möglich**: Um die Effizienz von Fehlerinjektionsattacken zu steigern soll ein intelligenter Algorithmus auf dem Host-PC den Verlauf eines Angriffs mitverfolgen und in dessen Ablauf eingreifen können. Durch das Erkennen und weglassen irrelevanter Teilangriffe soll Zeit gespart werden. Erfolgversprechende Angriffe sollen vertieft werden können. Dazu ist eine Umprogrammierung der bedingten Triggerausgabe notwendig. Der „VC-Glitcher“ bietet nicht die Möglichkeit zur Laufzeit Modifikationen am Verlauf eines Angriffs vorzunehmen. Dazu wäre eine zeitaufwändige, manuelle Änderung der Angriffseinstellungen in der „riscure Inspector“-Software mit anschließender Programmierung des „VC-Glitcher“ notwendig.
- **Reader unterstützt nur Standardprotokolle**: Während der Smartcard-Entwicklung werden unter anderem nicht-spezifizierte Protokolle zur Kommunikation mit einer Smartcard verwendet. Der „VC-Glitcher“ enthält einen Reader, der ausschließlich über die standardisierten Kommunikationsprotokolle T=0 und T=1 mit einer Smartcard kommunizieren kann. Um Fehlerinjektionsattacken oder Seitenkanalanalysen auf in der Entwicklung befindliche Smartcards durchführen zu können, muss jedoch über nicht-standardisierte Kommunikationsprotokolle mit ihnen kommuniziert werden können.
- **Reader unterstützt keine Extended Waiting-Timeouts**: Berechnungsfunktionen werden schon in frühen Entwicklungsstadien auf einer Smartcard getestet. Diese können noch Laufzeiten mit einer Länge von bis zu mehreren Minuten haben, die vom fertigen Produkt, am Ende der Entwicklung, nicht mehr erreicht werden. Deshalb sind kommerzielle Reader, inklusive dem im „VC-Glitcher“ enthaltene Reader, nicht dafür ausgelegt mit derartig langen Laufzeiten umzugehen. Erhält der Reader des „VC-Glitcher“ nach einer unveränderbaren, maximalen Wartezeit (auch Waiting-Timeout genannt) keine Antwort von der Smartcard, bricht er die Kommunikation ab.

In der Praxis werden sowohl Oszilloskope, als auch der „VC-Glitcher“ für Tests der Smartcard-Software verwendet, teilweise auch beide in Kombination. So wird eine mehrstufige Triggerbedingung von einem oder mehreren Oszilloskopen ausgewertet und das erzeugte Triggersignal vom „VC-Glitcher“ anschließend um eine bestimmte Anzahl von Kommunikationstakten verzögert. Zwei Arten von Anwendungsfällen sind zu unterscheiden:

1) Die Smartcard gibt ein Triggersignal auf einer gesonderten Triggersignalleitung aus. Dieses wird dann an den Triggereingang des „VC-Glitcher“ angelegt und von diesem mit einer definierten Verzögerung ausgegeben. Eine Änderung des Triggerzeitpunktes der Smartcard erfordert deren Umprogrammierung durch einen Smartcard-Entwickler. Da dies zeitaufwändig ist, gibt die Smartcard meistens mehrere Triggerpulse aus, wobei Oszilloskope eine Unterscheidung ermöglichen. Dies führt zur zweiten Art von Anwendungsfällen.
2) Die Smartcard verfügt über keine Triggersignalleitung und gibt Triggerpulse auf der IO-Signalleitung aus oder sie gibt gar keine Triggerpulse aus. Dann müssen die Kommunikationsleitungen zwischen Reader und Smartcard über mehrstufige Triggerbedingungen ausgewertet werden. Dies geschieht durch ein oder mehrere Oszilloskope. Das von ihnen generierte Triggersignal wird entweder direkt ausgegeben oder an den Triggereingang des „VC-Glitcher“ angelegt und von diesem mit einer definierten Verzögerung ausgegeben.

Das erzeugte Triggersignal geht dann entweder an einen Oszilloskop-Eingang und startet so eine Messung oder es wird zu einem Pulsgenerator weitergeleitet. Dieser verändert die Länge des Triggerpulses und gibt ihn an einen Fehlerinjektions-Laser weiter, wobei über die Länge des Triggerpulses die Dauer und Intensität einer Laserbestrahlung bestimmt wird.
Aufgrund der Einschränkungen der bisherigen Lösungen, soll der Reader, zusammen mit dem in dieser Bachelorarbeit implementierten Triggergenerator, eine automatisierbare und leistungsfähigere Alternative bilden. Der Reader ersetzt dabei den „VC-Glitcher“ und der Triggergenerator die Oszilloskope.

## 4.2 Individuelle Synthese für jeden Anwendungsfall

Die erste mögliche Entwurfsgrundlage zielt darauf ab den Benutzer Triggerbedingungen in möglichst abstrakter Form definieren zu lassen, ohne Kenntnisse über die eingesetzte Hardware. Der Benutzer definiert eine oder mehrere mehrstufige Triggerbedingungen in einer Konfigurationsdatei. Ein Programm wertet diese aus und erzeugt daraus eine Schaltung in Form von VHDL-Dateien. Über eine Programmierschnittstelle delegiert es anschließend die Synthese der Schaltung an die ISE-Entwicklungsumgebung der Firma Xilinx. Die FPGA-Konfigurationsdatei muss dann noch zum FPGA hin übertragen werden. Dies könnte mithilfe eines JTAG-Adapters geschehen. Alternativ könnte auch die Funktion des FPGA, Konfigurationsdaten von einem Mikrocontroller anzufordern, verwendet werden. Der Reader könnte die Aufgabe übernehmen die Konfigurationsdaten zwischenzuspeichern und auf Anfrage zum FPGA weiterzuleiten.

**Vorteile**

Diese Lösung ist vermutlich die anwenderfreundlichste, abhängig von der Komplexität der Syntax, mit der die mehrstufigen Triggerbedingungen definiert werden müssen. Der Benutzer muss dann kein Wissen über die Implementierung haben, sondern definiert Triggerbedingungen in möglichst abstrakter Form. Außerdem stehen die gesamten Hardware-Ressourcen des FPGA für die Umsetzung der vom Anwender definierten mehrstufigen Triggerbedingungen zur Verfügung.

**Nachteile**

Änderungen an den mehrstufigen Triggerbedingungen erfordern eine erneute Synthese, was zeitaufwändig ist. Dies erfordert zudem einen PC auf dem die Xilinx ISE-Entwicklungsumgebung läuft. Laut der Smartcard-Entwickler wäre dies inakzeptabel, da beabsichtigt wird Änderungen an den Triggerbedingungen möglichst schnell und direkt vom Reader durchführen zu lassen.

## 4.3 Bibliothek verschaltbarer Logikblöcke

Der FPGA enthält eine Menge einfacher Logikkomponenten, wie zum Beispiel UND- und ODER-Verknüpfungen, sowie Zähler und Timer. Diese sind, über einen Speicher im FPGA, welcher vom Benutzer beschrieben werden kann, untereinander verschaltbar.

**Vorteile**

Durch die Möglichkeit Schaltungen aus einfachsten logischen Verknüpfungen erstellen zu können, hat der Benutzer die größtmögliche Flexibilität bei der Definition und Änderung von Triggerbedingungen. Außerdem ist keine erneute Synthese für jeden Anwendungsfall notwendig, da alle Einstellungen zur Laufzeit vorgenommen werden.

**Nachteile**

Der Benutzer muss ein Schaltnetz aus der Bibliothek zur Verfügung stehender Logikblöcke entwerfen, dass die gewünschten mehrstufigen Triggerbedingungen umsetzt. Dies erfordert zum einen Wissen über die Bibliothek und zum anderen das nötige digitaltechnische Verständnis. Auch wenn beides vorhanden ist, kann es viel Zeit in Anspruch nehmen ein passendes Schaltnetz zu entwickeln.
Verknüpfungen zwischen einer großen Anzahl von Logikblöcken zu ermöglichen führt zudem zu einem nicht zu unterschätzenden Verwaltungsaufwand, der einen beträchtlichen Teil der Hardware-Ressourcen des FPGA einnehmen würde.

## 4.4 Einstellbare Schaltung universeller Triggerkomponenten

Als Kompromiss zwischen den ersten beiden Lösungsansätzen könnte der FPGA eine Menge universeller, elementarer Triggerkomponenten enthalten. Diese erfüllen verschiedene komplexe triggerbezogene Aufgaben und können zur Laufzeit vom Benutzer eingestellt werden. Der Benutzer müsste eine Verschaltung dieser Triggerkomponenten entwerfen, welche die mehrstufigen

Triggerbedingungen realisiert. Denkbar wäre es jede Unterbedingung auf eine separate Triggerkomponente abzubilden. Zur Laufzeit können die Triggerkomponenten eingestellt und in eine bestimmte Reihenfolge gebracht werden.
Um die Hardware-Ressourcen des FPGA effektiv auszunutzen hätte der Benutzer die Möglichkeit einzelne Triggerkomponenten vor der Synthese zu spezialisieren.

**Vorteile**

Der Benutzer kann seine mehrstufigen Triggerbedingungen relativ einfach in eine Schaltung aus Triggerkomponenten überführen, da nur eine Komponente pro Unterbedingung benötigt wird. Jede Triggerkomponente verfügt dabei über die nötigen Schaltungen um jede Unterbedingungsart umzusetzen. Dadurch ist der Triggergenerator sehr flexibel und benötigt keine anwendungsspezifische Synthese. Bei Bedarf ist es dem Benutzer jedoch möglich Triggerkomponenten spezialisiert zu synthetisieren. So kann Flexibilität gegen Hardware-Ressourcen getauscht werden, um mehr Triggerkomponenten erstellen zu können.

**Nachteile**

Mehrstufige Triggerbedingungen müssen vom Benutzer in ein Schaltnetz aus Triggerkomponenten überführt werden, was bei komplexen Triggerbedingungen, zeitaufwändig sein kann.
Die Verwaltung der Triggerkomponenten benötigt Hardware-Ressourcen des FPGA, welche für die Triggerberechnung nicht mehr zur Verfügung stehen. Gleiches gilt für die Flexibilität der Triggerkomponenten, da jeweils die Schaltungsteile für jede Unterbedingungsart vorhanden sind.

## 4.5 Auswahl eines Lösungsansatzes

Der Triggergenerator soll von Benutzern, mit wenig Kenntnissen über dessen interne Schaltung, schnell und einfach konfiguriert und eingesetzt werden können.
Als Grundlage der Umsetzung des Triggergenerators wurde der dritte Lösungsansatz gewählt, welcher einen praktikablen Kompromiss zwischen den beiden anderen Lösungsansätzen darstellt und keinen von deren gravierenden Nachteilen mit sich bringt. So ist eine zeitaufwändige erneute Synthese nicht notwendig, aber möglich, um den Triggergenerator an besonders umfangreiche Anwendungsfälle anzupassen. Der Hardwareaufwand für die Verbindung der Triggerkomponenten hält sich zudem in akzeptable Grenzen. Da nur eine relativ geringe Anzahl von Triggerkomponenten vorhanden ist, kann die Möglichkeit jede mit jeder, in beliebiger Reihenfolge, verbinden zu können, leicht umgesetzt werden.

# 5 Entwurf

Die Aufgaben des Triggergenerators können in drei Aspekte unterteilt werden:

- **Übertragung und Speicherung benutzerdefinierter Einstellungen**
- **Triggererzeugung über mehrstufige Triggerbedingungen**
- **Signalweiterleitung zwischen Reader und Smartcard**

Für jeden Aufgabenbereich lässt sich ein funktionaler Block definieren. Diese zuständigkeitenbezogene Aufteilung der Schaltung in Module hat zum Ziel die Übersicht zu erleichtern und die Wartbarkeit der Schaltung zu fördern. Abb. 11 zeigt die vier, nach Verantwortlichkeiten getrennten, Funktionsblöcke des Triggergenerators in einer Übersicht. Diese Darstellung soll vorerst dazu dienen die Schnittstellen zwischen den vier großen Teilbereichen der Schaltung aufzuzeigen. Auf die folgenden Abschnitte vorgreifend, enthält jeder Block bereits Details zu den enthaltenen Teilkomponenten, sowie die konkreten Namen implementierter Komponenten.

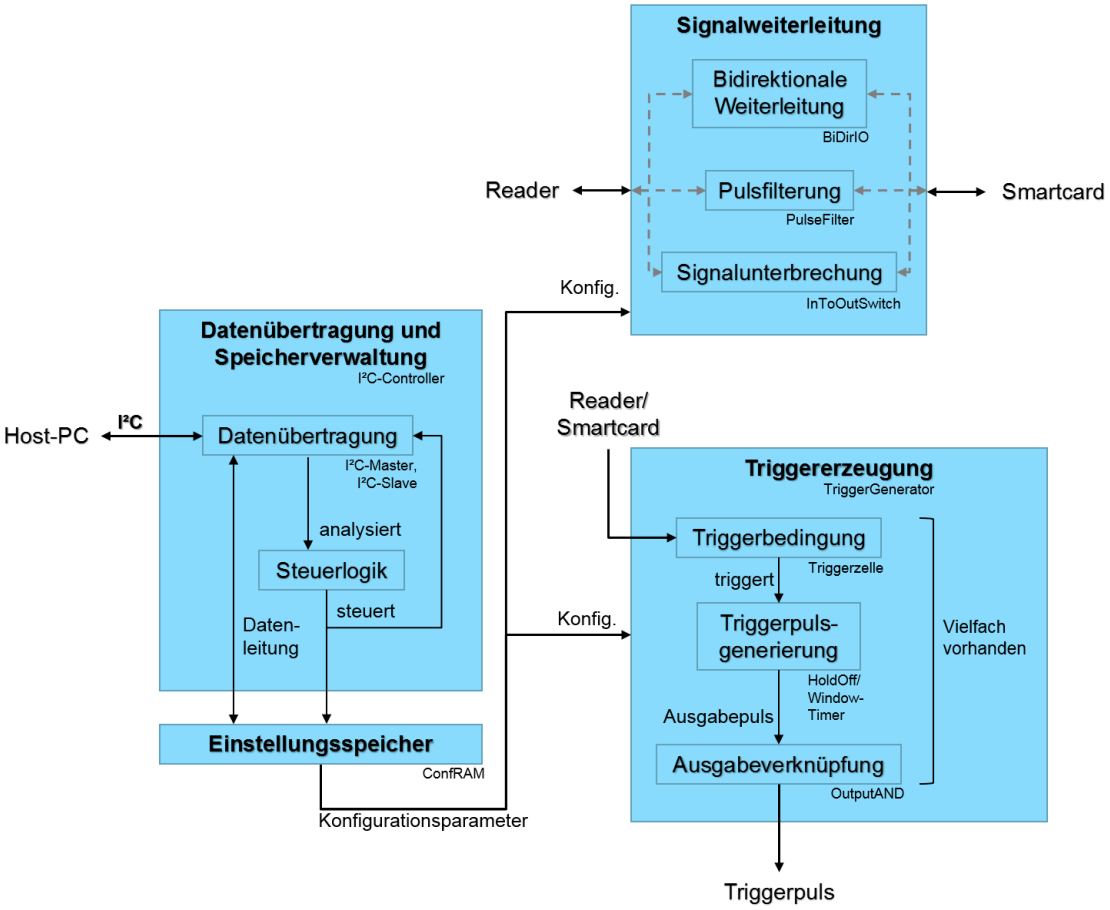


Abb. 11: Komponentenzuständigkeiten – Übersicht

Nachfolgend sollen Aufbau und Funktionsweise der implementierten Komponenten erläutert werden. Begonnen wird mit der Triggererzeugung, wobei die zwischen Reader und Smartcard verlaufenden Signalleitungen beobachtet und daraus Triggersignale berechnet werden, entsprechend der

benutzerdefinierten mehrstufigen Triggerbedingungen. Wie in Abb. 11 zu sehen ist, wurden die Auswertung einer Triggerbedingung, die Generierung eines Triggersignals und die Auswahl eines Ausgabesignals in separaten Komponenten gekapselt.
Anschließend wird auf die Umsetzung der Signalweiterleitung zwischen Smartcard und Reader eingegangen. Diese kann uni- oder bidirektional sein, optional mit zusätzlicher Pulsfilterung, sowie unterbrechbar sein.
Danach werden die implementierten Komponenten zur Konfiguration des Triggergenerators erläutert. Die benutzerdefinierten Konfigurationsparameter liegen in einem Einstellungsspeicher, auf den der Benutzer über eine I²C-Schnittstelle zugreifen kann. Die Steuerung der Kommunikation, sowie die Speicherverwaltung wurden in einer Komponente vereint.

## 5.1 Triggererzeugung

Eine mehrstufige Triggerbedingung besteht aus einer oder mehreren Unterbedingungen unterschiedlichen Typs. Gemäß dem dritten Lösungsansatz im Analysekapitel (siehe „4.4 Einstellbare Schaltung universeller Triggerkomponenten“), wird eine Menge universeller Triggerkomponenten implementiert, welche der Benutzer zur Laufzeit allozieren kann. Jede dieser, fortan als **Triggerzelle** bezeichneten, Komponenten kann grundsätzlich alle Arten von Unterbedingungen umsetzen. Mehrere Triggerzellen können sowohl unabhängig voneinander verwendet werden, als auch zusammenarbeiten. Einzeln verwendet realisiert eine Triggerzelle eine einzelne Triggerbedingung, die nebenläufig zu den anderen Triggerzellen abgearbeitet wird. Zu einer Sequenz verschaltet können hingegen mehrere Unterbedingungen nacheinander überprüft werden, um so mehrstufige Triggerbedingungen zu realisieren.

Wie aus Abb. 11 ersichtlich, wurden die Zuständigkeiten zur Auswertung einer Triggerbedingung und die Generierung eines Triggersignals auf zwei Komponenten aufgeteilt. Erstere Aufgabe übernimmt eine Triggerzelle, letztere ein **HoldOff/Window-Timer**. Instanzen beider Komponenten sind einander jeweils fest zugeordnet. Auch an dieser Stelle erfolgte eine Trennung nach Verantwortlichkeiten zur Verbesserung der Übersicht und Erweiterbarkeit.

Abb. 12 zeigt alle an der Triggererzeugung beteiligten Komponenten und deren Vernetzung in einer Beispielschaltung mit drei Triggerzellen (bezeichnet als TC0, TC1 und TC2).

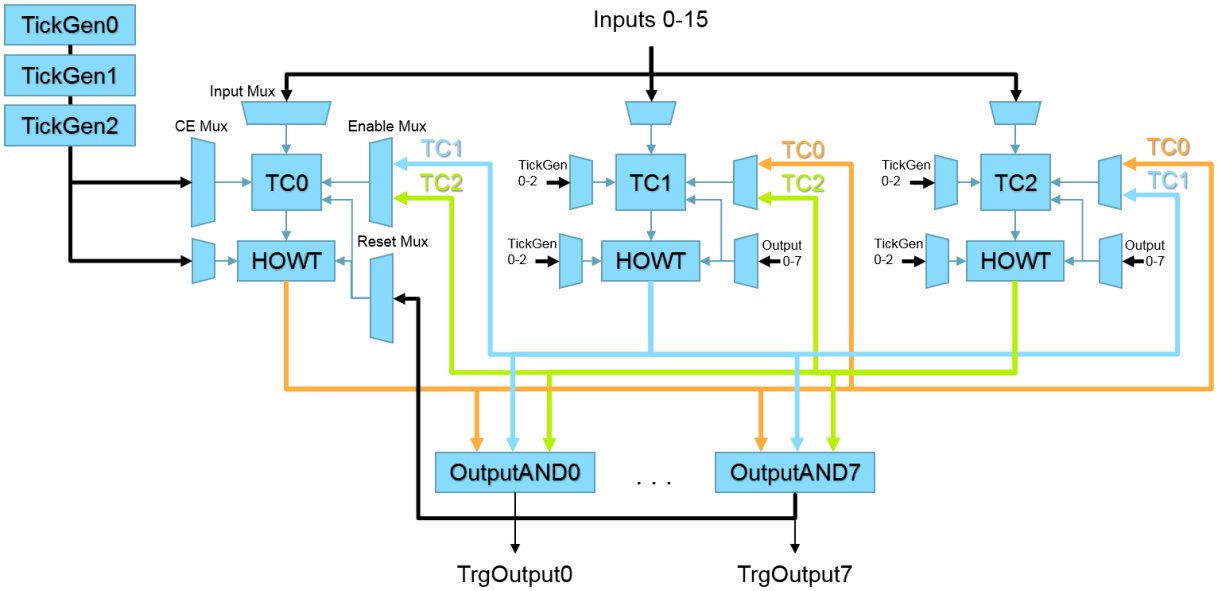


Abb. 12: Verschaltung von Triggerzellen – Konzept

Um die drei Triggerzellen TC0, TC1 und TC2 zur Laufzeit zu einer Sequenz verschalten zu können, sowie diese jeweils mit beliebigen Ein- und Ausgängen des Triggergenerators verbinden zu können, werden Multiplexer eingesetzt. Für jede Triggerzelle existiert ein Satz aus vier Eingangsmultiplexern:

- **Input Mux**: Dient zur Auswahl einer von 16 Datensignalleitungen, die hinsichtlich einer Unterbedingung ausgewertet werden soll.
- **Enable Mux**: Ermöglicht die Verkettung mehrerer Triggerzellen zu einer mehrstufigen Triggerbedingung. Eine Triggerzelle arbeitet nur, wenn sie von außen enabled wird. Es kann entweder ein permanentes Enable-Signal oder ein Triggersignal einer anderen Triggerzelle angelegt werden. In ersterem Fall ist die Triggerzelle unbedingt enabled, in letzterem wird sie erst enabled, nachdem eine vorherige Unterbedingung erfüllt wurde. An einem Enable-Multiplexer einer Triggerzelle liegen dazu die Triggersignalleitungen der anderen Triggerzellen an. Wie in Abb. 12 zu sehen ist, könnte TC0 somit von TC1 oder TC2 enabled werden. Alternativ kann TC0 auch unbedingt enabled oder disabled sein (nicht in Abb. 12 eingezeichnet).

Diese drei Triggerzellen könnten nun zu einer mehrstufigen Triggerbedingung verschaltet werden, bei der zuerst TC0, dann TC1 und anschließend TC2 erfüllt werden sollen. Dazu müsste TC0 unbedingt enabled sein, TC1 von TC0 enabled werden und TC2 von TC1 enabled werden.

- **Reset Mux**: Dient zur Auswahl eines Triggersignals, das einen Reset der Triggerzelle auslösen soll. Dadurch ist es möglich eine mehrstufige Triggerbedingung nach deren Triggern automatisch in den Ausgangszustand zurückzusetzen und diese damit für ein erneutes Triggern bereit zu machen. Um die Größe des Multiplexers gering zu halten, wird für den Reset eines der acht Ausgabesignale ausgewählt und nicht direkt das Triggersignal einer der, in größerer Anzahl vorhandenen, Triggerzellen.
- **CE[6] Mux**: Die Arbeitsgeschwindigkeit einer Triggerzelle kann durch Auswahl eines Ticksignals, von einem **Tickgenerator**, verlangsamt werden. Dieser zählt CPU-Takte und gibt nach einer benutzerdefinierten Anzahl einen Tick aus, das heißt einen Puls mit Länge eines CPU-Taktes.

Während eine Triggerzelle eine Triggerbedingung auswertet und so einen Triggerzeitpunkt berechnet, wird die Generierung des auszugebenden Triggersignals in einer weiteren Komponente gekapselt, dem **HoldOff/Window-Timer** (im Folgenden abgekürzt als **HOWT** bezeichnet). Für jede Triggerzelle existiert genau ein HOWT, welcher einen Triggerpuls empfängt und daraufhin einen Triggerpuls mit benutzerdefinierter Verzögerung und Länge erzeugt. Die Arbeitsgeschwindigkeit eines HOWT kann, unabhängig von der zugeordneten Triggerzelle, über einen CE-Mux verringert werden. Eine Trennung der Triggerberechnung und der Triggerausgabe in zwei Komponenten wurde vorgenommen, da schon aus den Anforderungen eine logische Trennung hervorgeht. Denn mit dem ersten Triggern einer Triggerzelle soll ein nicht-unterbrechbarer Ablauf aus Triggerverzögerung und anschließender Triggerausgabe beginnen, der unabhängig vom späteren Zustand der Triggerzelle abläuft. Das heißt unabhängig davon, ob die Triggerzelle zwischenzeitlich erneut triggert oder disabled wird.
Die HOWT-Komponente wird für zwei Aufgaben verwendet, abhängig von der Position der zugehörigen Triggerzelle innerhalb einer mehrstufigen Triggerbedingung: Entweder für das verzögerte Enable der nächsten Triggerzelle der Sequenz oder zur Erzeugung eines Ausgabetriggersignals, beides jeweils mit bestimmter Verzögerung und Länge.

### 5.1.1 Triggerzelle

Jede Triggerzelle implementiert alle benötigten Arten von Unterbedingungen, von denen jeweils eine ausgewählt werden kann, die dann von der Triggerzelle folgendermaßen realisiert wird.

1. **Highstate**: In diesem Modus soll ein Triggersignal ausgegeben werden, wenn sich die Eingangsleitung im High-Zustand befindet. Somit kann der Eingangswert direkt an den Triggerausgang durchgereicht werden.
2. **Lowstate**: Um ein Triggersignal auszugeben, wenn sich die Eingangsleitung im Low-Zustand befindet, wird das Eingangssignal invertiert auf dem Triggerausgang ausgegeben.
3. **HighEdgeCount**: Ein Triggersignal soll nach dem Auftreten einer bestimmten Anzahl steigender Flanken ausgegeben werden. Die Flankenerkennung erfolgt dabei über ein zweistufiges Schieberegister. Ein Zähler verfolgt die Anzahl erkannter Flanken. Stimmt der Zählwert mit dem Soll-Wert überein, wird ein Triggerpuls ausgegeben und der Zähler zurückgesetzt.
4. **LowEdgeCount**: In diesem Fall wird das Auftreten fallender Flanken gezählt. Der übrige Ablauf ist äquivalent zu HighEdgeCount.
5. **HighPulseDetect**: In diesem Modus zählt ein Zähler CPU-Takte, solange sich das Eingangssignal im High-Zustand befindet. Erreicht der Zähler einen unteren Grenzwert und ist unter einem oberen Grenzwert, wenn das Eingangssignal wieder in den Low-Zustand wechselt, wird ein Triggerpuls ausgegeben. Unabhängig davon wird der Zähler bei jedem Wechsel in den Low-

[6] CE (Abk. für Clock-Enable): Bezeichnung der Firma Xilinx für den FlipFlop-Eingang, welcher die taktsynchrone Übernahme des Eingangssignals steuert.

Zustand auf den Startwert zurückgesetzt. Ein eingehender Puls muss somit eine Mindestlänge haben und enden, bevor eine Höchstlänge erreicht wird. Nur dann wird ein Triggersignal ausgegeben.

6. **LowPulseDetect**: Zählt CPU-Takte, wenn sich das Eingangssignal im Low-Zustand befindet. Der übrige Ablauf ist äquivalent zu HighPulseDetect.

Der Zähler für Signalflanken oder CPU-Takte ist in einer eigenen Komponente gekapselt, dem **EdgesAndPulsClocks**-Zähler (fortan abgekürzt als **ENPC** bezeichnet). Diese ist in den Modi Highstate und Lowstate disabled.
Eine Triggerausgabe erfolgt nur, wenn die Triggerzelle von außen enabled ist. Andernfalls wird der interne Zähler im Anfangszustand gehalten. Die Flankenerkennung läuft hingegen weiter. Auf diese Weise wird sichergestellt, dass die Auswertung einer Unterbedingung mit dem nächsten enable von vorne beginnt.
Das Clock-Enable Signal bestimmt die Abtastrate des flankenerkennenden Schieberegisters und die Geschwindigkeit des Zählers für CPU-Takte. Dadurch können größere Pulse mit einem kleineren Zähler erkannt werden, was Hardwareressourcen des FPGA spart. Mit steigendem zeitlichen Abstand zwischen zwei eingehenden Ticks sinkt somit die Abtastrate der Pulserkennung und damit die Genauigkeit mit der die Länge zu erkennender Pulse in den Modi HighPulseDetect und LowPulseDetect definiert werden kann.
Die Bitbreite des internen Zählers beträgt maximal 32 Bit. Ohne Verwendung eines Tickgenerators wird somit im CPU-Takt von 50 MHz mit einer Schrittweite von 20 ns gezählt. Daraus ergibt sich eine maximale Zeitspanne von über 80 Sekunden (32 Bit * 20 ns) für die Länge zu erkennender Pulse in den Modi HighPulseDetect und LowPulseDetect. Da dieser Wert weit über den Anforderungen liegt (siehe „3.3 Spezifische funktionale Anforderungen“), kann die Zählerbitbreite über benutzerdefinierte Konstanten komfortabel angepasst werden. Näheres dazu folgt im Abschnitt „5.4 Generische Attribute“.

### 5.1.2 Triggerausgabe

Ein von der Triggerzelle ausgegebenes Triggersignal soll eine, von der Triggerzelle unabhängig ablaufende, Sequenz aus Triggerausgabeverzögerung (HoldOff) und anschließender Triggerausgabe in bestimmter Länge (Window) starten, wie in Abschnitt „3.2.2 HoldOff- und Window-Offsets“ beschrieben. Zur Steuerung dieses Ablaufs wird ein Zustandsautomat verwendet. Dieser umfasst drei Zustände, die in folgender Reihenfolge durchlaufen werden:

1. **Idle**: Im Anfangszustand wird auf einen eingehenden Triggerpuls von der Triggerzelle gewartet.
2. **HoldOff**: Zur Umsetzung der Ausgabeverzögerung, zählt ein Zähler bis zu einem HoldOff-Wert, der im Einstellungsspeicher, dem ConfRAM, gespeichert ist. Für die Dauer dieses Zustands wird ein Low-Pegel auf dem Triggerausgang ausgegeben.
3. **Window**: Zur Umsetzung der Triggerausgabe in bestimmter Länge, zählt ein Zähler bis zu einem Window-Wert, der im ConfRAM gespeichert ist. Für die Dauer dieses Zustands wird ein High-Pegel auf dem Triggerausgang ausgegeben.

Äquivalent zu dem internen Zähler der Triggerzelle, zählt auch der HOWT-Zähler mit maximal 32 Bit im CPU-Takt von 50 MHz. Daraus ergibt sich eine Schrittweite von 20 ns bei der Definition des HoldOff- und es Window-Werts, weshalb eine Ungenauigkeit von ± 10 ns erwartet wird. Auch in diesem Fall übersteigt die erreichbare Zeitspanne für HoldOff- und Window-Längen, mit über 80 Sekunden, deutlich die in den Anforderungen genannte 1 ms (siehe „3.3 Spezifische funktionale Anforderungen“). Deshalb kann die Zählerbitbreite über eine benutzerdefinierte Konstante vor der Synthese reduziert werden, um Hardwareressourcen zu sparen.
Die Möglichkeiten des HOWT-Zählers erfüllen gleichzeitig die Anforderungen an die einstellbare Länge und Verzögerung eines an den Benutzer ausgegebenen Triggersignals. Außerdem werden damit auch

die Anforderungen an die mögliche Länge und Genauigkeit der Unterbrechung einer Signalweiterleitung erfüllt, da diese von einem HOWT bestimmt werden, worauf im folgenden Abschnitt „5.2 Signalweiterleitung“ eingegangen wird.

Die Triggerausgänge des Triggergenerators werden jeweils von einer Instanz der OutputAND-Komponente beschrieben. Eine beliebige benutzerdefinierte Kombination von Triggersignalen der HOWT-Komponenten werden jeweils UND-verknüpft und eine Triggerausgabe auf einem der Ausgänge TrgOutput0-7 findet nur statt, während alle relevanten HOWT-Komponenten gleichzeitig ein Triggersignal ausgeben. So ist es möglich voneinander unabhängige, nebenläufige mehrstufige Triggerbedingungen an der Berechnung eines Triggersignals teilhaben zu lassen.
Die in Abb. 12 gezeigte Schaltung wird in einer Komponente gekapselt, welche im Folgenden als **TriggerGenerator**-Komponente bezeichnet wird.

## 5.2 Signalweiterleitung

Für die unidirektionale Weiterleitung von Reader-Signalen zur Smartcard reicht es diese im FPGA von dessen Eingängen an dessen Ausgänge durchzureichen. Für die Triggerberechnung relevante Signalleitungen werden zudem an die TriggerGenerator-Komponente weitergegeben.
Die bidirektionale Weiterleitung der IO-Signalleitung erfordert TriState-Ausgangstreiber im FPGA, sowie PullUp-Widerstände. Für die Steuerung der Weiterleitung wird die **BiDirIO**-Komponente implementiert, welche die momentane Übertragungsrichtung erkennt und so eine Rückkopplungsschleife zwischen den Aus- und Eingängen des FPGA verhindern kann.
Nach Aussage der Smartcard-Tester finden nur sehr selten Änderungen bzgl. Anzahl und Art weiterzuleitender Signale statt, sodass die daran beteiligten Komponenten im FPGA fest verdrahtet werden können. Das heißt die Signalweiterleitung ist nicht zur Laufzeit beeinflussbar, abgesehen von Unterbrechungen durch die **InToOutSwitch**-Komponente. Diese ermöglicht eine triggergesteuerte Unterbrechung der Weiterleitung eines Signals. Zudem ermöglicht die **Pulsfilter**-Komponente das herausfiltern kurzer Pulse aus einem weitergeleiteten Signal.

### 5.2.1 Bidirektionale Weiterleitung

Die BiDirIO-Komponente ermöglicht die Weiterleitung von Low-Pulsen zwischen zwei bidirektionalen Ports. Um unabhängig von Übertragungsprotokollen und Geschwindigkeiten zu sein, findet die Weiterleitung pulsweise statt. Zur Erkennung der aktuellen Übertragungsrichtung wird ein Zustandsautomat verwendet. Liegt an beiden Eingängen ein PullUp-bedingtes High-Signal an, ist die Übertragungsrichtung unbekannt und es wird ein TriState-Signal in beide Richtungen ausgegeben. Die Weiterleitung von High-Signalen erfolgt somit indirekt durch die PullUp-Widerstände. Findet an einem Eingang ein Flankenwechsel in den Low-Zustand statt, kennzeichnet dies einen Wechsel der Übertragungsrichtung. Daraus lässt sich ein Zustandsautomat mit drei Zuständen ableiten, den Abb. 13 zeigt.

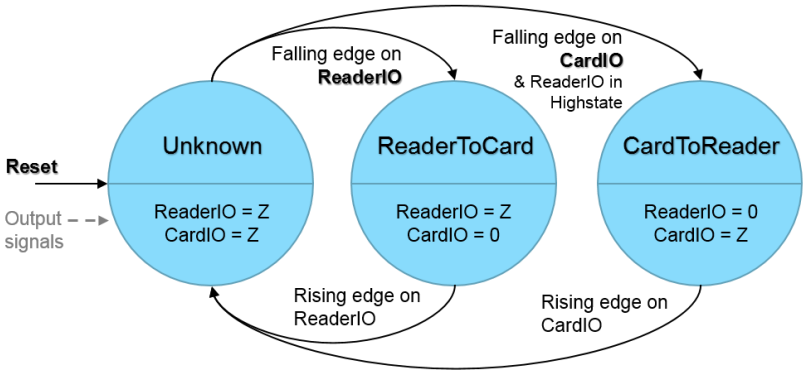


Abb. 13: Zustandsautomat zur Steuerung der BiDirIO-Komponente

1. **Unknown**: Übertragungsrichtung ist unbekannt. Beide Ausgangstreiber werden in den TriState-Zustand versetzt. Tritt eine fallende Flanke am Port **ReaderIO** auf, findet ein Wechsel in den Zustand **ReaderToCard** statt. Tritt alternativ eine fallende Flanke am Port **CardIO** auf, wird in den Zustand **CardToReader** gewechselt.
2. **ReaderToCard**: Der Reader sendet einen Low-Puls, der zur Smartcard weitergeleitet wird. Für die Dauer des Zustands wird deshalb ein Low-Pegel auf CardIO ausgegeben. ReaderIO befindet sich weiterhin im TriState-Zustand. Beendet der Reader den Low-Puls auf ReaderIO, ist die Übertragung beendet und es wird zurück in den Zustand **Unknown** gewechselt.
3. **CardToReader**: Die Smartcard sendet einen Low-Puls, der zum weitergeleitet wird. Dazu wird ein Low-Pegel auf ReaderIO ausgegeben, während CardIO im TriState-Zustand bleibt. Beendet die Smartcard den Low-Puls auf CardIO, wird zurück in den Zustand **Unknown** gewechselt.

Bislang unbetrachtet blieb der Fall, in dem Reader und Smartcard gleichzeitig einen Low-Puls senden wollen, das heißt es treten auf den Signalleitungen ReaderIO und CardIO gleichzeitig fallende Flanken auf. Dieses gleichzeitige Senden von Daten ist grundsätzlich verboten. Denn im Normalfall beginnt der Reader mit dem Senden einer Anfrage und wartet anschließend so lange, bis die Smartcard geantwortet hat. Die Angriffe, welche auf die Smartcard ausgeübt werden, können jedoch ein unvorhersehbares Verhalten provozieren. So kommt es, nach Erfahrung der Smartcard-Tester, auch vor, dass die Smartcard plötzlich beginnt beliebige Daten zu senden. Deshalb ist es sehr wichtig genauer auf diese Situation einzugehen.
Der Triggergenerator reagiert auf derartige Datenkollisionen, indem dem Readersignal der Vorzug gegeben wird. Das heißt ein Übergang in den Zustand CardToReader findet nur statt, wenn eine fallende Flanke auf CardIO auftritt und ReaderIO währenddessen im High-Zustand bleibt. Auf Seite der Smartcard, bzw. CardIO, kommt es in dem Fall zu einer Kollision zweier Low-Pegel, Triggergenerator und Smartcard senden beide ein Low-Signal. Durch die Open-Collector Schaltung ist dies, aus elektrotechnischer Sicht, unproblematisch. Sobald der Reader das Senden seines Low-Pulses einstellt, findet ein Wechsel zurück in den Zustand Unknown statt. Wenn die Smartcard dann immer noch einen Low-Pegel sendet, erfolgt unmittelbar danach ein Übergang in den Zustand CardToReader.
Für die Erkennung und Behandlung von Datenkollisionen ist der Reader verantwortlich, denn die Signalweiterleitung durch den Triggergenerator erfolgt für Reader und Smartcard transparent. Dementsprechend sind für den Triggergenerator keine Funktionen zur Kollisionsvermeidung vorgesehen. Problem wird lediglich weitergereicht.
Mithilfe des Zustandsautomaten wird zudem eine Rückkopplung der weitergeleiteten Pulse verhindert. Denn vom Triggergenerator ausgegebene Signale werden nicht zurückgelesen. So wird das Problem eines Signal-Echos umgangen.
Die Umschaltzeiten der BiDirIO-Komponente, das heißt die Veränderung der Länge weitergeleiteter Pulse, beeinflussen die Datenübertragung zwischen Smartcard und Reader nicht. Denn es ist ohnehin eine Pulsfilterung aus dem weitergeleiteten Signal vorgesehen, wobei laut den Anforderungen (siehe „3.3 Spezifische funktionale Anforderungen“), Schwankungen der Länge nicht gefilterter Pulse um ± 100 ns toleriert werden. Die erreichte Umschaltzeit der Implementierung liegt, mit einem CPU-Takt von 50 MHz, bei ± 10 ns. Die Weiterleitungsverzögerung der Pulsübertragung beträgt, bedingt durch die Flankenerkennung, zwei CPU-Takte und liegt damit weit unter der tolerablen Grenze von ± 1 µs.
Um die Filterung kurzer Pulse in der Übertragungsrichtung von der Smartcard zum Reader zu ermöglichen, enthält die BiDirIO-Komponente einen unidirektionalen Pulsfilter, auf den im folgenden Abschnitt eingegangen wird.

### 5.2.2 Pulsfilterung

Die Pulsfilter-Komponente ermöglicht die Filterung von Pulsen, die eine definierte Mindestlänge unterschreiten, aus einem unidirektionalen Signal. Das Ausgabesignal stammt dabei aus einem Zwischenspeicher. Liegt ein Eingangssignal an, das sich von dem Ausgabesignal unterscheidet, beginnt

eine Zählung von CPU-Takten. Erreicht der Zähler den Soll-Wert, wird das Eingangssignal in den Ausgabespeicher übernommen. Wenn Eingangssignal und Ausgabesignal gleich sind, wird der Zähler in den Startzustand zurückversetzt. Auf diese Weise werden High- und Low-Pulse kürzer dem Soll-Wert gefiltert.
Der verwendete Zähler von CPU-Takten zählt maximal 32 Bit mit 50 MHz, woraus sich eine Schrittweite von 20 ns, mit einer erwarteten Abweichung von ± 10 ns, für die Definition der Obergrenze zu filternder Pulse ergibt. Zur Abdeckung des in den Anforderungen genannten Wertebereichs von bis zu 1 ms reicht bereits ein 16 Bit Zähler. Die Bitbreite eines Pulsfilters kann deshalb als benutzerdefinierte Konstante festgelegt werden.

### 5.2.3 Signalunterbrechung

Zur Unterbrechung der Weiterleitung eines Signals, wird die InToOutSwitch-Komponente implementiert. Diese besteht aus lediglich drei Multiplexern, welche asynchron zum CPU-Takt, triggergesteuert zwischen den drei möglichen Ausgabesignalen umschaltet: Dem Eingangssignal, einem High-Pegel und einem Low-Pegel. Der Benutzer definiert hierzu ein Default-Ausgabesignal und ein Triggered-Ausgabesignal. Ersteres wird ausgegeben, wenn kein Triggerpuls anliegt, letzteres solange ein Triggerpuls anliegt. Als Triggerquelle kann eines der HOWT-Ausgabesignale ausgewählt werden.
Da der InToOutSwitch asynchron zum CPU-Takt arbeitet, beträgt die Reaktionsgeschwindigkeit der Schaltung auf ein eingehendes Triggersignal erwartungsgemäß weniger als eine Taktperiode (20 ns). Die Anforderungen an die mögliche Dauer und Schrittweite der Unterbrechungen werden somit vom triggererzeugenden HOWT erfüllt.

## 5.3 Übertragung und Speicherung benutzerdefinierter Einstellungen

Die Steuersignale der in Abb. 12 (siehe S. 39) gezeigten Multiplexer müssen permanent anliegen, um den nebenläufigen Betrieb aller Triggerzellen zu ermöglichen. Der Einstellungsspeicher wird deshalb als „Distributed RAM“ angelegt, welcher jeweils ein FlipFlop zur Speicherung eines Datenbits verwendet. Der Einstellungsspeicher, fortan als **ConfRAM** bezeichnet, wird somit einerseits asynchron von nebenläufig arbeitenden Komponenten ausgelesen, andererseits auch synchron über die Benutzerschnittstelle. Abb. 14 zeigt den ConfRAM mit dessen umgebenden Komponenten.

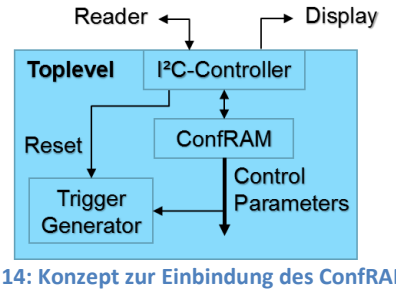


Abb. 14: Konzept zur Einbindung des ConfRAM

Der **I²C-Controller** wird implementiert, um Anfragen des Benutzers über eine I²C-Schnittstelle zu empfangen und die geforderten Aktionen auszuführen. Dazu zählen:

- Die Speicherung von Parametern im ConfRAM
- Das Auslesen von Parametern aus dem ConfRAM und deren Übertragung zum Benutzer
- Die Verarbeitung von Benutzerkommandos (Reset der TriggerGenerator-Komponente)

Für den Empfang und die Beantwortung von Anfragen durch den Benutzer wird ein **I²C-Slave** implementiert, der sich im I²C-Controller befindet und von diesem gesteuert wird.

Für das Speichern und Auslesen von Einstellungsdaten stellt der ConfRAM dem I²C-Controller eine taktsynchrone Schnittstelle zur Verfügung. Über diese kann ein Speicherwort adressiert und ausgelesen oder mit einem angelegten Wert überschrieben werden.
Zusätzlich zur I²C-Schnittstelle zum Benutzer, wird eine zweite für die Übertragung ausgewählter ConfRAM-Inhalte zu einem Display implementiert. Dazu wird ein **I²C-Master** implementiert, welcher sich ebenfalls im I²C-Controller befindet und von diesem gesteuert wird.
Während der Entwicklung wird das Display zum Debuggen eingesetzt. Im Anschluss kann es verwendet werden, um den Fortschritt automatisierter Angriffe überwachen zu können.

### 5.3.1 Einstellungsspeicher

Der ConfRAM besteht, von außen betrachtet, aus einer Menge von 32 Bit breiten Speicherworten. Jedes Speicherwort enthält dabei einen Konfigurationsparameter. Es sei erwähnt, dass die gegebenen Anforderungen mit dieser Bitbreite erfüllt werden.
Auf VHDL-Ebene umfasst der ConfRAM zusätzlich Mechanismen zur Optimierung der Speichergröße. Denn in den seltensten Fällen werden wirklich alle 32 Bit eines Speicherworts benötigt. Abhängig von der Art des gespeicherten Parameters wird dieser, vom Benutzer oder automatisch, auf eine bestimmte Bitbreite beschränkt. Details dazu folgen in späteren Abschnitten.
An dieser Stelle soll beschrieben werden, wie die datenwortweise Reduzierung der Speichergröße in VHDL umgesetzt wurde. Der Speicher innerhalb der ConfRAM-Komponente ist als Array aus 32 Bit-Vektoren angelegt. Zu speichernde Datenworte werden in diesem Array abgelegt. Daneben existiert ein Ausgabe-Array, das beim Auslesen des ConfRAM verwendet wird. Zur Optimierung werden nur die benötigten Bits aus dem speichernden Array in das Ausgabe-Array abgebildet. Auf diese Weise können nicht verwendete Speicherbits bei der Synthese erkannt und herausoptimiert werden, wie in Abb. 15 dargestellt ist.

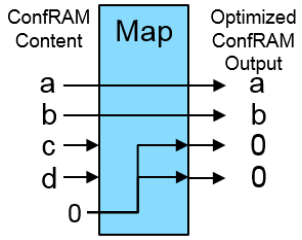


Abb. 15: ConfRAM Größenoptimierung

In diesem Beispiel sei ein Datenwort betrachtet, das aus den vier Bytes (a bis d) besteht. Davon werden jedoch nur die ersten beiden Bytes (a und b) benötigt. Das heißt die Bytes c und d werden weder mit relevanten Werten beschrieben, noch zur Konfiguration verwendet. Das diese Speicherbytes nicht gebraucht werden, muss dem Synthesetool explizit gezeigt werden. Ohne Optimierung wäre es dem Synthesetool nicht möglich unbenutzte Speicherbits zu erkennen, da die taktsynchrone Schnittstelle zum I²C-Controller es grundsätzlich ermöglicht alle in den ConfRAM geschriebenen 32 Bit Datenworte in voller Bitbreite auszulesen. Dazu werden alle benötigten Speicherbits (die Bytes a und b) innerhalb der Map-Komponente des ConfRAM in das Ausgabe-Array abgebildet. Anstelle der Bytes c und d enthält das Ausgabe-Array konstante Nullen. Somit werden die Bytes c und d des speichernden Arrays nie ausgelesen. Für das Synthesetool ist dies ein deutlicher Hinweis auf nicht benötigte Ressourcen.
Näheres zu den Berechnungen der Parameterbitbreiten folgt in Abschnitt „5.4 Generische Attribute“.
Alle Speicherbits des ConfRAM werden, vom Ausgabe-Array aus, über separate Leitungen nach außen gereicht und mit den Empfängerkomponenten verbunden. Auf VHDL-Ebene werden Records zur Strukturierung der Ausgabebits eingesetzt, um die Verknüpfungen übersichtlich zu gestalten.

### 5.3.2 I²C-Controller

Um einerseits den Zugriff auf beliebig adressierte Datenworte zu ermöglichen und andererseits einen sequentiellen Zugriff auf im Speicher hintereinanderliegende Datenworte zuzulassen, wird ein Speicheriterator implementiert. Dieser zeigt auf das als nächstes gelesene oder überschriebene Datenwort. Nach jedem Zugriff wird der Iterator auf das Folgewort gesetzt. Am Ende des ConfRAM angelangt, wird er auf dessen Anfang zurückgesetzt. Dadurch ist es möglich mehrere Datenworte oder den kompletten Inhalt des ConfRAM in einer einzigen I²C-Nachricht auszulesen oder zu überschreiben. Für den Zugriff auf beliebig adressierte Datenworte kann der Iterator über ein Kommando mit einer Adresse für den nächsten Zugriff belegt werden. Ein zweites Kommando wurde zur Durchführung eines Resets der TriggerGenerator-Komponente implementiert.
Empfangene Datenworte werden vom I²C-Controller analysiert, um zwischen Daten und Kommandos unterscheiden zu können. Denn Kommandos können an beliebigen Stellen innerhalb einer I²C-Nachricht übertragen werden und werden unmittelbar nach dem Empfang ausgeführt. Folgendes Kommandoformat wurde implementiert, hier in hexadezimaler Form angegeben:

$$Set\ Iterator\ Command: 0xff80XXXX$$
$$TriggerGenerator\ Reset\ Command: 0xff400000$$

Das höchstwertigste Byte auf „0xff“ gesetzt, entspricht der Kommandomarkierung. In allen anderen Fällen handelt es sich um reine Daten. Daraus folgt die Einschränkung, dass kein Parameterwert verwendet werden darf, dessen höchstwertigstes Byte aus „0xff“ ist. Diese Einschränkung wird in absehbarer Zukunft kein Nachteil sein, da keine Konfigurationsparameter dieser Größe benötigt werden. Das zweithöchstwertigste Byte dient zur Unterscheidung de verschiedenen Kommandos. Die „0x80“ steht dabei für das Kommando „Set Iterator Command“ zum Setzen des Iterators und die „0x40“ für das Kommando „TriggerGenerator Reset Command“ zur Erzeugung eines TriggerGenerator-Resets. Die beiden niederwertigsten Bytes enthalten kommandospezifische Parameter. Soll der Iterator gesetzt werden, muss die neue Datenwortadresse in die mit „XXXX“ markierten Kommandobytes eingesetzt werden.
Im Gegensatz zum I²C-Slave überträgt der I²C-Master nicht datenwortweise, sondern byteweise Daten zum Display. Ein Schreibvorgang auf das Display wird immer dann angestoßen, wenn der I²C-Slave eine Übertragung beendet hat. Denn nur dann kann sich der ConfRAM-Inhalt und damit die Displayausgabe verändert haben. Die zum Display übertragenen Daten sind in der **DisplayDataMemory**-Komponente gespeichert und werden vom I²C-Controller byteweise an den I²C-Master weitergereicht, bis der komplette Inhalt übertrtagen wurde.

#### I²C-Slave

Die Kommunikation mit dem Reader findet über zwei bidirektionale Signalleitungen unter Benutzung von TriState-Ausgangstreibern, in Kombination mit PullUp-Widerständen statt. Die TriState-Treiber werden dabei nur zum Schreiben von Low-Pulsen eingesetzt und damit als Open-Collector Ausgangstreiber eingesetzt. So wird sichergestellt, dass zwei Ausgänge niemals gegeneinander treiben, was einen Kurzschluss und damit eine Beschädigung der Hardware zur Folge hätte.
Der Reader ist der I²C-Master, welcher Lese- oder Schreib-Anfragen an den I²C-Slave sendet. Der Triggergenerator enthält den I²C-Slave, welcher auf Anfragen wartet und diese beantwortet.
Gemäß dem I²C-Protokoll beginnt eine I²C-Nachricht mit einer Startbedingung und der Übertragung einer Slave-Adresse durch den I²C-Master. Der I²C-Slave bestätigt, seine Adresse empfangen zu haben. Nachfolgend wird eine beliebige Anzahl von Datenbytes übertragen, jeweils mit einer Empfangsbestätigung zwischen zwei Bytes.
Es gibt zwei Arten von Anfragen: Schreibanfragen und Leseanfragen. Im ersten Fall sendet der I²C-Master zu speichernde Daten zum I²C-Slave und der I²C-Slave bestätigt deren Empfang byteweise. Im zweiten Fall sendet der I²C-Slave angeforderte Daten zum I²C-Master und der I²C-Master sendet byteweise Bestätigungen. Das Ende einer Übertragung markiert der I²C-Master über eine

Stoppbedingung. Der I²C-Slave verarbeitet diesen Ablauf über einen Zustandsautomaten, dessen Zustände in vereinfachter Kurzform beschrieben werden sollen:

- **Idle**: Slave wartet auf Startbedingung.
- **Init**: Slave wird in den Startzustand versetzt. Dann unbedingter Übergang nach Read_Address.
- **Read_Address**: Master überträgt Ziel-Adresse, Slave bestätigt diese zu besitzen.
    - Master will Daten zum Slave übertragen: Zustandsübergang nach Read_Data
    - Master will Daten vom Slave auslesen: Zustandsübergang nach Write_Data
- **Read_Data**: Slave empfängt Datenbyte, überträgt jeweils eine Empfangsbestätigung.
- **Write_Data**: Slave sendet Datenbyte, empfängt jeweils eine Empfangsbestätigung.

Das entsprechende Zustandsdiagramm befindet sich in Anhang D (siehe Abb. 70, S. 118).

Der Init-Zustand wird stets bei Empfang einer Startbedingung erreicht. Wenn dies während einer Datenübertragung geschieht, kann unmittelbar mit der Auswertung einer neuen Adresse begonnen werden. Denn nicht immer überträgt ein I²C-Master eine Stoppbedingung vor der nächsten Startbedingung. Der Idle-Zustand wird jeweils bei Empfang einer Stoppbedingung erreicht oder wenn eine unbekannte Slaveadresse empfangen wurde.
Die Datenwortgröße des ConfRAM beträgt vier Byte. Deshalb werden empfangene Bytes vom I²C-Slave zwischengespeichert und dem I²C-Controller als ganzes Datenwort übergeben. Sollte die Übertragung enden, bevor alle vier Datenwort-Bytes empfangen wurden, werden die bereits empfangenen Daten verworfen. In umgekehrter Richtung, wenn der I²C-Slave ein Datenwort zum Reader überträgt, muss letzterer den Empfang aller vier Datenwort-Bytes bestätigt haben, bevor dem I²C-Controller mitgeteilt wird seinen Speicheriterator auf das nächste Datenwort zeigen zu lassen. Bleibt eine Empfangsbestätigung durch den I²C-Master aus, werden keine weiteren Daten gesendet. Der I²C-Master im Reader hat so die Möglichkeit die Übertragung mithilfe einer Stoppbedingung zu beenden oder über eine Startbedingung erneut zu beginnen.

#### I²C-Master

Die Kommunikation mit dem Display findet auch über zwei bidirektionale Signalleitungen unter Benutzung von TriState-Ausgangstreibern und PullUp-Widerständen statt. Das Display ist der I²C-Slave, welcher Schreibanfragen vom I²C-Master erhält und die empfangenen Daten anzeigt.
Für den I²C-Master sieht das I²C-Protokoll vor eine Startbedingung zu senden, gefolgt von der Adresse eines I²C-Slave zu dem eine Verbindung aufgebaut werden soll, sowie der Information, ob Daten zum I²C-Slave gesendet oder von diesem empfangen werden sollen. Nach der Bestätigung der Adresse durch den I²C-Slave beginnt die eigentliche Datenübertragung. Zu einem beliebigen Zeitpunkt kann der I²C-Master eine Stoppbedingung senden, um die Übertragung zu beenden oder eine Startbedingung, um eine neue Verbindung aufzubauen.
Ab der Startbedingung erzeugt der I²C-Master einen Kommunikationstakt. Über diesen steuert er die Datenrate der Übertragung. Ein I²C-Slave kann diese bei Bedarf, über einen Clock-Stretching genannten Mechanismus, reduzieren. Dazu hält dieser die Taktsignalleitung im Low-Zustand und hindert den I²C-Master so daran weitere Kommunikationstakte zu senden. Der implementierte I²C-Master erkennt dies und pausiert die Datenübertragung solange der I²C-Slave dies wünscht.
Der I²C-Master steuert die Kommunikation über einen Zustandsautomaten, welcher im Detail in Anhang D dargestellt und erläutert wird (siehe Abb. 78, S. 126).

### 5.3.3 Displayausgabespeicher

Der I²C-Controller entnimmt die zum Display zu übertragenden Datenbytes der DisplayDataMemory-Komponente und reicht sie an den I²C-Master weiter. Im DisplayDataMemory sind sowohl displayspezifische Konfigurationsbytes (siehe EA 2014), zum Beispiel zur Steuerung der Hintergrundbeleuchtung, sowie die eigentlichen anzuzeigenden Zeichen gespeichert. Da taktsynchron und byteweise ausgelesen wird, wurde ein Block-ROM des FPGA als Speicher verwendet. Die

Verwendung eines dieser, im FPGA ohnehin vorhandenen Hardwarebausteine, von denen der Triggergenerator ansonsten keinen Gebrauch macht, spart eine große Menge Hardwareressourcen, gegenüber einer Speicherimplementierung nach Art des ConfRAM.
Während die konstanten Display-Bytes in einem Block-ROM liegen, werden die veränderlichen Ausgabewerte aus dem ConfRAM in die DisplayDataMemory-Komponente hineingereicht. Den veränderlichen Daten sind bestimmte Speicheradressen des DisplayDataMemory fest zugewiesen. Beim Auslesen des Speichers wird, abhängig von der gewählten Speicheradresse, entweder ein im Block-ROM gespeichertes oder ein von außen anliegendes Byte zurückgegeben. So wird dem I²C-Controller eine einzige Schnittstelle zur Verfügung gestellt, über die alle zum Display zu übertragenden Daten abgerufen werden können. Unabhängig davon, ob diese als Konstanten im Block-ROM liegen oder aus dem ConfRAM stammen.
Die VHDL-Datei der DisplayDataMemory-Komponente kann mithilfe des, in C++ implementierten, Programms „Text2Vhdl.exe“ und einer Konfigurationsdatei generiert werden. Der auf dem Display auszugebende Text kann so in lesbarer Form in der Konfigurationsdatei gespeichert werden. Beliebige Stellen im Text können für die Ausgabe veränderlicher Werte markiert werden. Die Generierung der VHDL-Datei erfolgt innerhalb des Programms durch Zusammensetzen von Quelltextblöcken. Es wird ein Array zur Speicherung der konstanten Zeichen erzeugt, sowie ein Adressdekoder, welcher beim Auslesen bestimmter Adressen nicht den Arrayinhalt ausgibt, sondern ein von außen anliegendes Datenbyte. Das Array im VHDL-Code wird zu einem Block-ROM synthetisiert.
Um ConfRAM-Inhalte in Form hexadezimaler Zeichen auf dem Display auszugeben, muss jeweils ein halbes ConfRAM-Byte in ein Zeichen-Byte (0-9 oder ‚a‘-‚f‘) umgewandelt werden, welches dann an die DisplayDateMemory-Komponente weitergegeben wird. Dazu wurde die **HexToChar**-Komponente implementiert. Diese rechnet die Werte von 0 bis 9 in das entsprechende ASCII-Byte (siehe Network Working Group 1969) um, sowie die Werte 10 bis 15 in das entsprechende ASCII-Byte der Zeichen ‚a‘ bis ‚f‘.

## 5.4 Generische Attribute

Um die Hardwareressourcen des FPGA möglichst effizient auszunutzen, wird die erforderliche Größe des ConfRAM mithilfe generischer Attribute bei der Synthese berechnet. Basis der Berechnungen sind benutzerdefinierte Konstanten zur Dimensionierung der Komponenten und deren Anzahl im System. Diese Konstanten stellen außerdem eine benutzerfreundliche Möglichkeit zur Umdimensionierung des Systems zur Verfügung. Diese ist notwendig, denn umfangreiche Anwendungsszenarien können es, aufgrund begrenzter Hardwareressourcen, erfordern den Hardwareverbrauch in einem Teilbereich des Systems zu reduzieren, um einen anderen Teilbereichs vergrößern zu können. Beispielsweise kann über diese Konstanten ein Kompromiss zwischen der Anzahl von Triggerzellen und deren Ausstattung gewählt werden. Da hierzu lediglich einige benutzerdefinierte Konstanten angepasst werden müssen, können auch Benutzer mit wenigen VHDL-Kenntnissen eine Umdimensionierung des Triggergenerators schnell und einfach vornehmen.

Aufgrund der teils komplexen Verschaltung der Komponenten, werden diese größtenteils automatisch, während der Synthese instantiiert und verschaltet. Auch dies geschieht auf Basis der benutzerdefinierten Konstanten. Insbesondere die Triggerzellen und deren Eingangsmultiplexer werden automatisch instantiiert und untereinander, sowie mit dem ConfRAM, verschaltet.

Da die generischen Berechnungen während der Synthese durchgeführt werden, müssen diese nicht synthetisierbar sein. Deren Umfang hat somit auch keinen Einfluss auf den Hardwarebedarf der Schaltung.

## 5.5 Zusammenfassung

Dieser Abschnitt hat gezeigt, dass die in den Anforderungen genannten Unterbedingungsarten alle von der selben universellen Triggerkomponente, der Triggerzelle, umgesetzt werden. Durch die Verknüpfung mehrerer Triggerzellen kann eine mehrstufige Triggerbedingung realisiert werden. Jeder Triggerzelle ist ein HoldOff- und Window-Timer zugeordnet, welcher es ermöglicht Unterbedingungen zeitlich versetzt für bestimmte Zeiträume zu enablen. Alternativ dazu ist es möglich eine Triggerausgabe zu verzögern, sowie deren Länge individuell festzulegen. Denn für beide Zwecke wird die gleiche HOWT-Komponente eingesetzt. Zusammen mit einem InToOutSwitch kann eine triggerbedingte Unterbrechung der Signalweiterleitung, von Kommunikationssignalen zwischen Reader und Smartcard erreicht werden.
Die BiDirIO-Komponente ermöglicht die bidirektionale Signalweiterleitung. Die Implementierung funktioniert protokollunabhängig und ist robust bezüglich Übertragungsfehlern, wie Datenkollisionen, die in Folge von Fehlerinjektionsangriffen zu erwarten sind.
Der Pulsfilter filtert kurze Pulse aus einem weitergeleiteten Signal und kann bei unidirektionaler, zusammen mit der BiDirIO-Komponente bei bidirektionaler, Signalweiterleitung eingesetzt werden.
Der I²C-Controller erfüllt, zusammen mit dem ConfRAM, die Anforderung benutzerdefinierte Einstellungen über eine I²C-Schnittstelle zur Laufzeit zum Triggergenerator übertragen zu können. Zudem können Teile der gespeicherten Einstellungen auf einem Display angezeigt werden. Obwohl die Displayausgabe zunächst nur zu Testzwecken integriert wurde, stellt sie im Nachhinein, laut der Smartcard-Tester, ein willkommenes Feature dar. Insbesondere, da die Speicherung auszugebender Daten, durch Verwendung eines Block-RAM, den Vorrat knapper Hardwareressourcen nicht belastet. Außerdem ermöglicht das Tool „Text2Vhdl“ eine komfortable Definition der Displayausgabe.
Benutzerdefinierte Konstanten im Quelltext ermöglichen es, auch Benutzern mit wenigen VHDL-Kenntnissen, Einfluss auf die Dimensionierung und den Hardwareverbrauch einzelner Schaltungsteile zu nehmen. Mithilfe generischer Attribute wird die benötigte Größe des ConfRAM berechnet, um das Syntheseprogramm bei der Optimierung der Schaltung zu unterstützen. Große Teile des Triggergenerators werden vor der Synthese automatisch generiert, entsprechend der benutzerdefinierten Konstanten. Der Triggergenerator kann so komfortabel in allen relevanten Teilbereichen umdimensioniert werden, um den Hardwareverbrauch gezielt zu beeinflussen. Die effiziente Ausnutzung der Hardwareressourcen begünstigt die Umsetzung besonders aufwändiger Anwendungsszenarien. Damit sind die im Abschnitt „3.2 Allgemeine Anforderungen“ genannten Anforderungen erfüllt.

# 6 Implementierung

Die implementierten Komponenten des Triggergenerators sind in Abb. 16 in einer vollständigen hierarchischen Übersicht dargestellt.

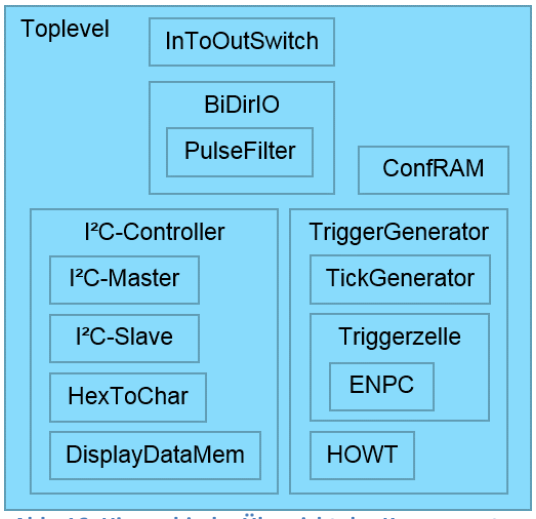


Abb. 16: Hierarchische Übersicht der Komponenten

Die im letzten Abschnitt angesprochenen generischen Attribute haben einen großen Einfluss auf die Komponenten auf der rechten Seite von Abb. 16, die TriggerGenerator-Komponente und den ConfRAM. Der ConfRAM wird vor der Synthese entsprechend dimensioniert und die in der TriggerGenerator-Komponente enthaltenen Triggerzellen instantiiert und miteinander vernetzt. Die Umsetzung dieses Mechanismus in VHDL soll in diesem Abschnitt im Detail erläutert werden.
Die im oberen Teil von Abb. 16 gezeigten Komponenten, der InToOutSwitch und die BiDirIO-Komponente, werden ebenfalls über benutzerdefinierte Konstanten dimensioniert, deren Instantiierung findet jedoch nicht automatisch statt, sondern erfolgt, da hier nur in seltenen Fällen Änderungen notwendig sind, durch den Benutzer.
Die auf der linken Seite von Abb. 16 dargestellten Komponenten, der I²C-Controller und dessen interne Komponenten, sind nicht umdimensionierbar. Die Ausnahme bildet der Speicher für die Displayausgabe, der DisplayDataMemory. Eine Änderung der, auf dem Display auszugebenden, Inhalte des ConfRAM erfordert eine Anpassung der Schnittstelle zwischen ConfRAM und DisplayDataMemory durch den Benutzer. Die Umsetzung des I²C-Controllers wird in diesem Abschnitt ebenfalls erläutert.
Die Firma NXP Semiconductors, in der diese Arbeit umgesetzt wurde, verlangte ausführliche Erläuterungen der Funktionsweise und Schnittstelle jeder implementierten Komponente. Diese befinden sich in Anhang D. In Top-Down Methode wird dort, angefangen beim Toplevel, jede Schaltung in Form eines Blockdiagramms dargestellt, um die Signalflüsse zwischen den enthaltenen Komponenten zu verdeutlichen. Die darauffolgenden Abschnitte begeben sich dann jeweils eine Hierarchieebene tiefer und erklären jede enthaltene Komponente anhand eines eigenen Blockdiagramms. Um den Smartcard-Testern die Verwendung des Triggergenerators ohne vorherige Einarbeitung in dessen Funktionsweise zu ermöglichen, wurde zudem ein User Guide gefordert. Dieser befindet sich in Anhang A und fasst die Einstellmöglichkeiten der Schaltung zusammen und beschreibt die Schritte zur Konfiguration anhand eines Beispielszenarios.

## 6.1 Dimensionierungsparameter des Systems

Die Dimensionierungsparameter werden an zentraler Stelle, einem VHDL-Package namens DEFINITIONS_pack, in Form von Konstanten zusammengefasst. Dieses wird von allen Komponenten importiert. Darin enthalten sind Konstanten zur Definition der Anzahlen verschiedener Komponenten, Konstanten zur Dimensionierung einzelner Komponenten, sowie Definitionen von Record-Typen zur Verwaltung des ConfRAM-Inhalts.

Dem Benutzer stehen folgende fünf Konstanten für die Dimensionierung des Gesamtsystems zur Verfügung:

- **numTriggerCells**: Definiert die Anzahl der Triggerzellen, die bei der Synthese automatisch instantiiert und miteinander verbunden werden sollen. Zulässig sind Werte von 1 bis 32.
- **numTGInputs**: Anzahl der Eingänge der TriggerGenerator-Komponente. Dieser Wert gibt an wieviele Signalleitungen zur Berechnung von Triggersignalen beobachtet werden können. Zulässig sind Werte von 1 bis 16, entsprechend der maximalen Anzahl Eingangssignale.
- **numTGTickGens**: Anzahl instantiierter Tickgeneratoren in der TriggerGenerator-Komponente. Zulässig sind Werte von 0 bis 32.
- **numTGOutputs**: Anzahl der Ausgänge der TriggerGenerator-Komponente. Dieser Wert gibt an, über wieviele Triggersignalleitungen die TriggerGenerator-Komponente verfügen soll. Zulässig sind Werte von 1 bis 8, entsprechend der maximalen Anzahl Triggersignale.
- **numInToOutSwitches**: Anzahl instantiierter InToOutSwitches. Über diesen Wert gibt der Benutzer bekannt wieviele Instanzen der InToOutSwitch-Komponente er erzeugt hat. Zulässig sind Werte von 1 bis 16.
- **numPulseFilters**: Anzahl instantiierter PulseFilter. Über diesen Wert gibt der Benutzer bekannt wieviele Instanzen der PulseFilter-Komponente er erzeugt hat. Zulässig sind ebenfalls Werte von 1 bis 16. Eine größere Anzahl als 16 ist nicht erforderlich, da es maximal 16 Weitergeleitete Signale geben kann.

Drei der in Abb. 16 gezeigten Komponenten können vor der Synthese vom Benutzer dimensioniert werden: Triggerzellen, Tickgeneratoren und Pulsfilter. Einstellbar sind die Bitbreiten komponenteninterner Zähler und die Bitbreiten der, im ConfRAM gespeicherter Parameter für jede Instanz einer Komponente. Das heißt die Anzahl der tatsächlich verwendeten Bits eines 32 Bit Datenworts im ConfRAM. Zur Angabe der Bitbreiten wurden drei Record-Typen definiert, deren enthaltene Parameter in den folgenden Abschnitten erläutert werden.

Für jede instantiierte Triggerzelle innerhalb der TriggerGenerator-Komponente existiert eine Instanz des in Abb. 17 dargestellten Record-Typs **tTcSpec**.

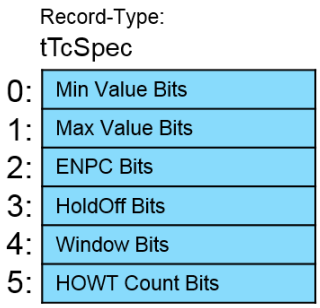


Abb. 17: Record-Type tTcSpec

Für jede Triggerzelle werden im ConfRAM unter anderem die Parameter „Min Value“ und „Max Value“ gespeichert. Diese geben die Grenzwerte für die Pulserkennung einer Triggerzelle (im High- und

LowPulseDetect Modus) an. Damit ein Trigger ausgegeben wird, muss ein Puls aufgetreten sein, dessen Länge, gemessen in CPU-Takten, zwischen „Min Value“ und „Max Value“ liegt. Die Parameter „Min Value Bits“ und „Max Value Bits“ legen fest wieviel Bit zur Speicherung der Grenzwerte im ConfRAM verwendet werden. Gültig sind Werte von 1 Bit bis 32 Bit.
Der Parameter „ENPC Bits“ bestimmt die Bitbreite des Zählers für Flanken oder CPU-Takte innerhalb der ENPC-Komponente einer Triggerzelle. Sinnvollerweise sollten „Max Value Bits“ und „ENPC Bits“ gleich sein, während „Min Value Bits“ auch kleiner sein kann.
Die Parameter „HoldOff“ und „Window“ beziehen sich auf den einer Triggerzelle zugeordneten HOWT und geben an um wieviele CPU-Takte eine Triggerausgabe verzögert wird (HoldOff) und wieviele CPU-Takte eine Triggerausgabe andauert (Window). Die Parameter „HoldOff Bits“ und „Window Bits“ legen fest wieviel Bit für deren Speicherung im ConfRAM verwendet werden.
Der Parameter „HOWT Count Bits“ bestimmt die Bitbreite des Zählers für CPU-Takte innerhalb des HOWT. Sinnvoll ist hier „HOWT Count Bits“ auf den jeweils höheren Wert von „HoldOff Bits“ und „Window Bits“ zu setzen.

Für jeden instantiierten Tickgenerator existiert eine Instanz des in Abb. 18 dargestellten Record-Typs **tTickGenSpec**.

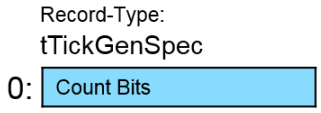


Abb. 18: Record-Type tTickGenSpec

Für jeden Tickgenerator wird der Parameter „Count Value“ im ConfRAM gespeichert. Der Tickgenerator zählt CPU-Takte bis zu diesem Grenzwert, gibt dann einen Tick-Puls aus und beginnt mit der Zählung von vorne.
Über den Parameter „Count Bits“ im Record tTickGenSpec, wird sowohl die Bitbreite von „Count Value“ definiert, als auch die Bitbreite des Zählers für CPU-Takte im Tickgenerator. Eine Trennung dieser beiden Angaben ist nicht erforderlich, da es im Tickgenerator immer nur einen Zähler gibt, der bis zu einem Grenzwert zählt.
Ein Record-Typ wurde für diesen einen Wert eingeführt, um zukünftige Erweiterungen um weitere Parameter zu vereinfachen.

Für jeden instantiierten Pulsfilter existiert eine Instanz des in Abb. 19 dargestellten Record-Typs **tPulseFilterSpec**.

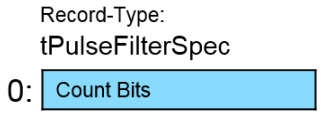


Abb. 19: Record-Type tPulseFilterSpec

Ein Pulsfilter zählt CPU-Takte und gibt einen eingehenden Puls nur aus, wenn vor dessen Ende der Grenzwert „Count Value“ erreicht wird, welcher pro Pulsfilter im ConfRAM gespeichert ist. Dessen Bitbreite, sowie die Bitbreite des Zählers im Pulsfilter wird über den Parameter „Count Bits“ definiert.

Für jeden der drei Record-Tyoen existiert ein Array, dessen Größe jeweils durch die Konstanten: numTriggerCells, numTGTickGens oder numPulseFilters bestimmt wird. Dadurch muss sich der Benutzer erstens nicht selber um die Instantiierung der drei beschriebenen Record-Typen kümmern, zweitens stehen ihm die VHDL-typischen Möglichkeiten zur Definition mehrerer Array-Elemente auf einmal zur Verfügung. So kann eine Standardbelegung für die Record-Werte definiert werden und diese über den „others“-Operator an alle Array-Elemente verteilt werden. Nur die davon abweichenden Elemente müssen vom Benutzer angepasst werden. Auf diese Weise wird der Aufwand

für den Benutzer minimiert, den eine Änderung der Komponentenanzahlen (zum Beispiel von numTriggerCells) mit sich bringt.
Ein Großteil der Bitbreiten im ConfRAM gespeicherter Parameter werden aus den Komponentenanzahlen errechnet und können daher nicht direkt vom Benutzer geändert werden. Auf diese wird im folgenden Abschnitt eingegangen.

## 6.2 Konfigurationsparameter der Komponenten

Die im ConfRAM gespeicherten Daten dienen zur Einstellung der Komponenten des Systems zur Laufzeit. Für jede Komponente wurde ein Record-Typ definiert, welcher die Reihenfolge und Anzahl der im ConfRAM gespeicherten Parameter festlegt. Die Bitbreiten der einzelnen Parameter werden entweder vom Benutzer direkt vergeben oder, wie im letzten Abschnitt angedeutet, aus benutzerdefinierten Konstanten errechnet.
Im Folgenden werden die für jede Komponente gespeicherten Parameter erläutert. Dabei wird insbesondere auf die Ermittlung der jeweiligen Bitbreite eingegangen. Es sei noch einmal darauf hingewiesen, dass die Reduzierung der ConfRAM-Größe, durch Verkleinerung von Datenworten, es erst ermöglicht besonders umfangreiche Anwendungsbeispiele umzusetzen. Ein derartiges Beispiel wird in einem späteren Kapitel im Detail beschrieben. Ohne diese Optimierung würde der ConfRAM einen wesentlich größeren Teil der Hardwareressourcen des FPGA einnehmen. Dies wäre eine Verschwendung von Hardware, die dann nicht mehr für sinnvolle Zwecke zur Verfügung steht.
Um die Datenwortgröße zu verringern wird dem Syntheseprogramm gezeigt, welche Bitbreiten benötigt werden. Denn dieses erkennt zwar nicht-verwendete Schaltungsteile und optimiert diese heraus, kann die Größe des ConfRAM jedoch nicht eigenmächtig beschränken, da alle Speicherbits scheinbar verwendet werden. Denn es werden stets 32 Bit Datenworte in den ConfRAM geschrieben und auch 32 Bit Datenworte ausgelesen. Erst durch die in Abschnitt 5.3 beschriebene Abbildung aller tatsächlich benötigten Bits des ConfRAM in ein Ausgabe-Array wird dem Syntheseprogramm gezeigt, welche Speicherbits tatsächlich benötigt werden, damit alle übrigen wegoptimiert werden können.

### 6.2.1 Triggerzelle

Ein zehn Parameter umfassender Datensatz konfiguriert eine Triggerzelle und die Komponenten in deren unmittalbarem Umfeld. Dazu zählen die in Abb. 12 (siehe S. 39) gezeigten Eingangsmultiplexer und auch der einer Triggerzelle zugeordnete HOWT. Diese zehn Parameter werden im Record-Typ **tTriggerCellMemMap** zusammengefasst, welcher in Abb. 20 dargestellt ist.

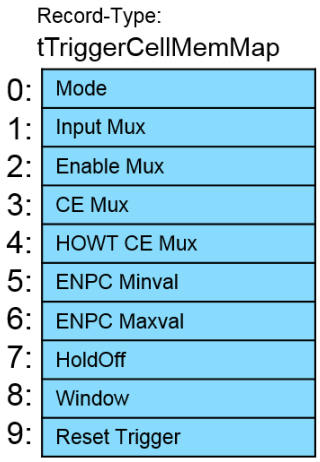


Abb. 20: Record-Type tTriggerCellMemMap

Die Bedeutung der einzelnen Parameter soll nun im Detail erläutert werden.

**Triggerzell-Parameter 0: Mode**

Jede Triggerzelle unterstützt sechs Trigger-Modi, von denen einer über den Parameter Mode ausgewählt werden kann. Die Speicherung eines Wertes von 0x0 bis 0x5 entspricht jeweils folgendem Modus:

$$0x0: Highstate$$
$$0x1: Lowstate$$
$$0x2: HighEdgeCount$$
$$0x3: LowEdgeCount$$
$$0x4: HighPulseDetect$$
$$0x5: LowPulseDetect$$

Die zur Speicherung von sechs Zuständen benötigte Bitzahl kann über den Zweierlogarithmus ermittelt werden:

$$Mode\ Bitbreite = log_2(6) = 2{,}58\ Bit$$

Dieser Wert wird aufgerundet, da mehr als zwei Bit benötigt werden:

$$Mode\ Bitbreite = \lceil log_2\ (6) \rceil = 3\ Bit$$

In VHDL werden diese Berechnungen mithilfe der IEEE-Bibliothek „math_real“ durchgeführt. Diese enthält die Funktion „log2“ zur Berechnung eines Logarithmus zur Basis 2, sowie die Funktion „ceil“ zum Aufrunden eines Real-Wertes auf die nächst größere ganze Zahl. Die Umsetzung der Berechnung im Quellcode zeigt Code 1.

```
constant numModes : integer := 6;
constant numModeBits : integer := integer(ceil(log2(real(numModes))));
```

Code 1: Berechnung des log2 zur Bestimmung der Datenwortbitbreite

Die Konstante numModes enthält die Anzahl implementierter Trigger-Modi (6) und wird in den Typ „real“ umgewandelt, da die Funktion „log2“ einen Real-Parameter verlangt. Der berechnete Logarithmus wird der Funktion „ceil“ übergeben und deren Ergebnis nach Umwandlung in den ganzzahligen Integer-Typ in der Konstanten numModeBits gespeichert.
In einem 3 Bit breiten Datenwort lassen sich $2^{3\,Bit} = 7\,Zustände$ unterscheiden. Die Triggerzelle erkennt die Auswahl des ungültigen siebten Zustands und erzeugt dann keine Triggersignale.
Der errechnete Wert wird einerseits zur Optimierung der Datenwortgröße im ConfRAM benutzt, andererseits zur Optimierung der Signalleitungen zwischen ConfRAM und Triggerzelle.

Die VHDL-Umsetzung zur Anpassung der Größe eines Mode-Datenworts im ConfRAM ist in Abb. 21 bildlich dargestellt.

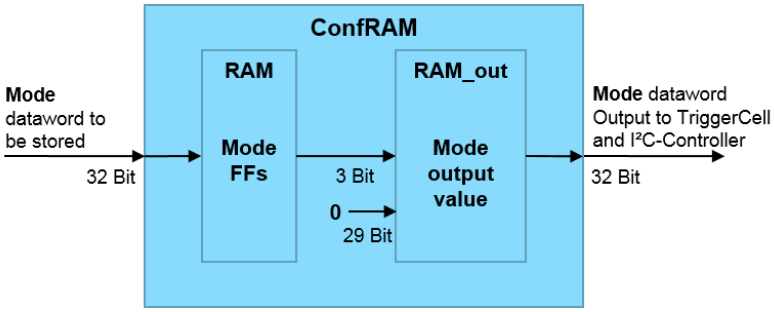


Abb. 21: Optimierung der Datenwortgröße (für Mode) im ConfRAM

Um ein Mode-Datenwort im ConfRAM zu speichern, wird dessen Speicheradresse dem ConfRAM mitgeteilt und das zu speichernde 32 Bit Datenwort übergeben. Dieses wird auf VHDL-Ebene in einem Array aus 32 Bit Datenworten abgelegt (in Abb. 21 als RAM bezeichnet). Ein zweites Array von identischer Größe (RAM_out) dient zum Auslesen des Speichers. Vom Datenwort Mode werden nur die ersten 3 Bit benötigt. Diese werden vom Array RAM an die gleiche Stelle im Array RAM_out weitergereicht. Die übrigen 29 Bit des Datenworts sind in RAM_out mit 0 initialisiert. Das Syntheseprogramm erkennt, dass nur 3 Bit tatsächlich gespeichert werden müssen und verbraucht deshalb nur 3 statt 32 FlipFlops zur Speicherung des Datenworts.
Ausgelesen wird das Mode-Datenwort taktsynchron vom I²C-Controller, welcher es erneut adressiert und es dann aus RAM_out ausliest, sowie asynchron von der Triggerzelle zu der es gehört.

Der ConfRAM gibt seinen Speicherinhalt asynchron in Form eines Arrays aus tTriggerCellMemMap-Instanzen aus, eine Instanz pro Triggerzelle. Alle Parameter des Record-Typs sind dabei 32 Bit breit. Bei zehn Parametern pro Triggerzelle werden somit zunächst 320 Signalleitungen vom ConfRAM zu jeder Triggerzelle geleitet, was erstmal einen großen Hardwareaufwand bedeutet. Eine Verkleinerung auf die tatsächlich benötigten Bitbreiten ist an dieser Stelle nicht möglich, da kein Array aus unterschiedlichen Versionen von tTriggerCellMemMap, mit individuellen Bitbreiten, erstellt werden kann. Ein Array wird zur Weitergabe der Parameter jedoch benötigt, da nur so die Ausgangsgröße des ConfRAM während der Synthese berechnet werden kann.
Auf VHDL-Ebene wird das gesamte tTriggerCellMemMap-Array vom ConfRAM an die TriggerGenerator-Komponente weitergereicht. In dieser werden bei der Synthese die Triggerzellen generiert, wobei sie mit den Signalleitungen aus dem ConfRAM verbunden werden. Bei der Anbindung werden dann nur die wirklich benötigten Bits berücksichtigt, wodurch es dem Syntheseprogramm möglich ist alle übrigen Signalleitungen zwischen ConfRAM und TriggerGenerator-Komponente wegzuoptimieren. Abb. 22 zeigt dies am Beispiel des Mode-Datenworts einer Triggerzelle.

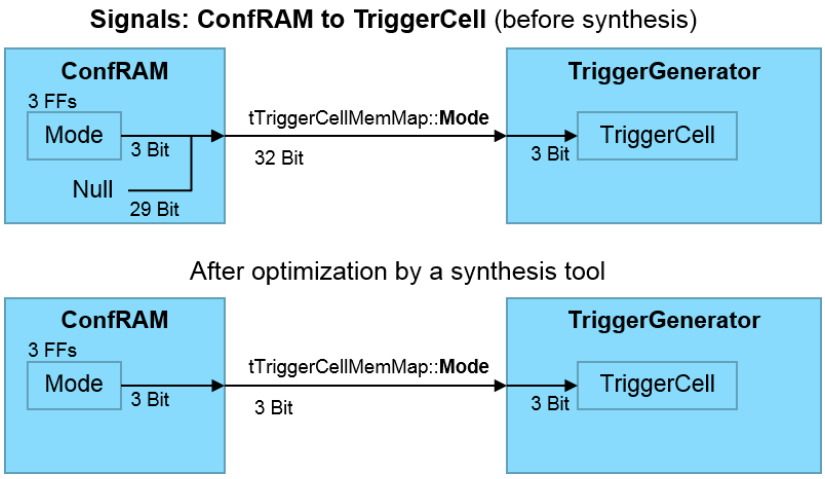


Abb. 22: Optimierung der Signalleitungen durch das Synthesetool

In der oberen Hälfte von Abb. 22 ist die Verbindung zwischen ConfRAM und TriggerGenerator-Komponente zu sehen, wie sie im Quellcode vorgenommen wird. Der 3 Bit Mode-Parameter wird mit Nullen auf die 32 Bit Datenwortbreite ausgefüllt und zur TriggerGenerator-Komponente weitergeleitet. Diese wählt die drei relevanten Bits aus und legt sie an die Eingänge der generierten Triggerzelle an. Das Syntheseprogramm erkennt, das nur diese 3 Bit verwendet werden und entfernt die 29 nicht-benötigten Signalleitungen.
Zusammen ermöglichen die beiden beschriebenen Optimierungen eine komfortable Erweiterung der Schaltung um weitere Trigger-Modi, das heißt eine Erweiterung des Triggergenerators um weitere Unterbedingungsarten. Eine Änderung der Konstanten „numModes“ führt zu einer Neuberechnung der benötigten Speicherbits im ConfRAM und zur Anpassung der Leitungsbitbreite zwischen ConfRAM und einer Triggerzelle.
Nach diesem Schema erfolgen alle Anpassungen, des ConfRAM und der Verbindungen zwischen ConfRAM und TriggerGenerator-Komponente, an die Bitbreite eines Parameters. Lediglich die Berechnung der jeweiligen Bitbreite unterscheidet sich von Parameter zu Parameter. Einige Bitbreiten können individuell für jede Instanz einer Komponente vom Benutzer definiert werden, wie in Abschnitt „6.1 Dimensionierungsparameter des Systems“ beschrieben wurde.

### Triggerzell-Parameter 1: Input Mux

Wie Abb. 12 (siehe S. 39) zeigt, dient ein Multiplexer je Triggerzelle zur Auswahl eines Eingangssignals, das von der Triggerzelle verarbeitet werden soll. Das Steuersignal dieses Multiplexers ist im Datenwort Input Mux des, zur Triggerzelle gehörenden, tTriggerCellMemMap-Records gespeichert.
Die Dimensionierung des Multiplexers erfolgt auf Basis der benutzerdefinierten Konstanten numTGInputs, welche die Anzahl der Eingänge der TriggerGenerator-Komponente angibt. Diese kann, den Requirements entsprechend, in einer Anzahl von 1 bis 16 Stück definiert werden (siehe „3.3 Ein-Ausgänge“).
Die benötigte Bitbreite für das Multiplexer-Steuersignal entspricht dem aufgerundeten Zweierlogarithmus der Anzahl der Eingangssignale. Verfügt die TriggerGenerator-Komponente beispielsweise über 10 Eingänge, da der Benutzer numTGInputs mit einem Wert von 10 belegt hat, ergibt sich daraus eine Bitbreite für das Steuersignal in Höhe von:

$$Input\ Mux\ Bitbreite = \lceil log_2(numTGInputs) \rceil = \lceil log_2(10) \rceil\ = 4\ Bit$$

Dieser Wert legt die Größe des Datenworts Input Mux im ConfRAM fest und die Größe des Multiplexers. Denn ein 4-Bit breites Steuersignal ermöglicht es mehr, als die vorhandenen 10 Eingänge zu unterscheiden, nämlich $2^{4\ Bit} = 16\ Zustände$.

Somit existieren sechs Steuersignalbelegungen, deren Verwendung ein undefiniertes Verhalten des Multiplexers bewirken könnte. Um dies zu verhindern, wird die Anzahl der Multiplexereingänge auf die Anzahl der adressierbaren Eingänge erhöht. Abb. 23 stellt den erweiterten Multiplexer in vereinfachter Form dar.

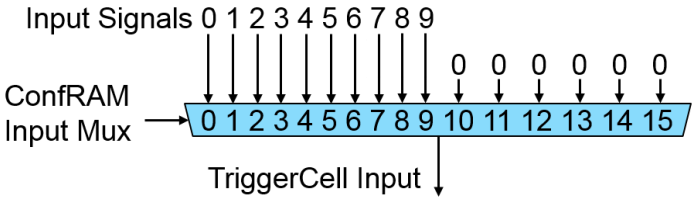


Abb. 23: Input-Multiplexer Blockdiagramm

Von den in diesem Beispiel adressierbaren 16 Eingängen sind die ersten 10 mit den Eingängen der TriggerGenerator-Komponente verbunden, während die übrigen Eingänge mit einem konstanten Low-Signal belegt sind. Diese Vorgehensweise wurde gewählt, da es sinnvoll erschien den Benutzer, im Falle einer Fehlkonfiguration, durch Beendigung der Funktion der Triggerzelle auf seinen Fehler hinzuweisen.
Zur Realisierung des Multiplexers in VHDL wird ein Vektor verwendet. Da dies einem unüblichen Verwendungszweck für einen Vektor darstellt und alle vier Multiplexer an den Eingängen einer Triggerzelle auf diese Weise generiert werden, sei die Vorgehensweise hier anhand des Quellcodes im Detail erläutert.

```
1:  gen_InputMux: for i in 0 to numTriggerCells - 1 generate
2:  signal mux_s : std_logic_vector(2**numTCsInputsMuxCtrlBits - 1 downto 0) := (others => '0');
3:  signal index_s : std_logic_vector(numTCsInputsMuxCtrlBits - 1 downto 0);
4:  begin
5:        mux_s(numTGInputs - 1 downto 0) <= inputs_s(numTGInputs - 1 downto 0);
6:        index_s <= RAM.triggercell(i).input_mux(numTCsInputsMuxCtrlBits - 1 downto 0);
7:        TCsInputIn_s(i) <= mux_s(to_integer(unsigned(index_s)));
8:  end generate gen_InputMux;
```

Code 2: Generierung der Triggerzellen-Eingangsmultiplexer

Code 2 zeigt einen Ausschnitt der TriggerGenerator-Komponente, welche die Triggerzellen und deren Eingangsmultiplexer generiert und mit den vom ConfRAM kommenden Steuersignalen verbindet. Für jede Triggerzelle wird ein Input-Multiplexer erstellt, weshalb der Schleifenkopf in Zeile 1 den Vorgang entsprechend wiederholt. In Zeile 2 wird der Multiplexer-Vektor erstellt und alle Felder mit 0 initialisiert. Die Größe des Vektors beträgt in diesem Beispiel 16 Bit. Zeile 3 zeigt den Steuersignalvektor index_s, welcher hier 4 Bit groß sei. In Zeile 5 werden die 10 Eingangssignalleitungen (inputs_s) mit den Multiplexereingängen 0 bis 9 verbunden. Nun fehlt lediglich noch die Anbindung einer Steuersignalleitung an den Multiplexer. In Zeile 6 nimmt dazu index_s, hier als Steuersignal verwendet, den ConfRAM-Parameter Input Mux auf. In Zeile 7 erfolgt schließlich die Zusammenführung aller Teile zum fertigen Multiplexer. Das Steuersignal index_s wird in einen ganzzahligen Typ (Integer) umgewandelt und indexiert den Multiplexer-Vektor mux_s. Der Wert i zur Identifikation einer Triggerzelle wird in Zeile 6 dazu verwendet den ConfRAM-Parameter Input Mux der i-ten Triggerzelle auszuwählen. In Zeile 7 dient er dazu das Ausgabesignal des Multiplexers an die i-te Stelle des Vektors TCsInputIn_s zu schreiben. Dieser bündelt alle Signalleitungen, die mit den Triggerzelleneingängen verbunden werden.
Die getroffenen Designentscheidungen haben zum Ziel dem Benutzer Skalierungen des Systems möglichst einfach zu machen. So führt eine Veränderung der Anzahl der Eingänge automatisch zur Anpassung der Anzahl von Steuerbits aus dem ConfRAM, die verwendet werden müssen. Die Benutzung eines Vektors als Multiplexer macht dies möglich, da dessen Größe bei der Synthese an die Anzahl der vorhandenen Eingänge angepasst werden kann.

Die Anzahl der Multiplexereingänge entspricht immer der Anzahl mit dem Steuersignal unterscheidbarer Zustände. Ein undefiniertes Verhalten bei Adressierung nicht-vorhandener Multiplexereingänge kann so ausgeschlossen werden.

### Triggerzell-Parameter 2: Enable Mux

Ein Multiplexer am Enable-Eingang einer Triggerzelle ermöglicht die Auswahl eines Triggersignals, welches steuert wann und wie lange eine Triggerzelle arbeitet. Auf diese Weise ist es möglich Triggerzellen zu einer Sequenz zu verketten, um so eine mehrstufige Triggerbedingung zu realisieren.
Das Steuersignal dieses Multiplexers ist im Datenwort Enable Mux des, zur Triggerzelle gehörenden, tTriggerCellMemMap-Records gespeichert.
Die Dimensionierung des Multiplexers wird durch die Anzahl vorhandener Triggerzellen bestimmt, welche über die benutzerdefinierte Konstante numTriggerCells festgelegt wird. Auch in diesem Fall wird der Zweierlogarithmus gebildet, um die Anzahl der benötigten Steuersignalbits zu bestimmen und die Größe des Multiplexers auf deren Zweierpotenz erweitert
Abb. 24 stellt die erzeugten Enable-Multiplexer innerhalb einer TriggerGenerator-Komponente vereinfacht dar, die vier Triggerzellen enthält.

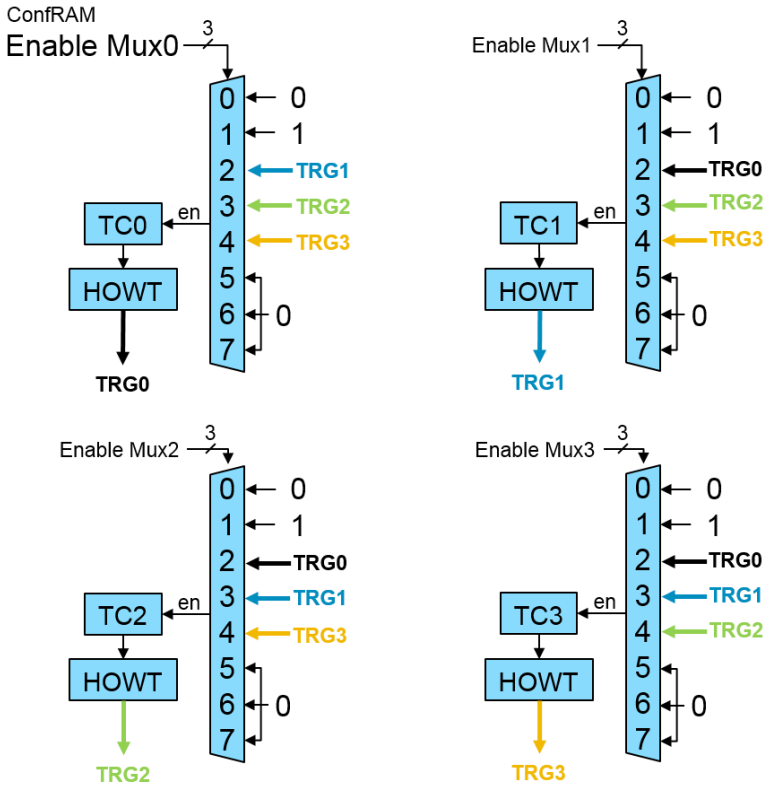


Abb. 24: Enable-Multiplexer Blockdiagramm

An den ersten beiden Eingängen eines Enable-Multiplexers liegen eine konstante Null und eine konstante Eins an, mithilfe derer die Triggerzelle für eine unbestimmte Zeitspanne disabled beziehungsweise enabled werden kann. An den darauffolgenden Eingängen liegen die Triggersignalleitungen aller HOWT-Komponenten des Systems in aufsteigender Reihenfolge an, ausgenommen der Ausgabeleitung des jeweils eigenen HOWT. Denn eine Rückkopplung des Triggersignals von diesem, an den Enable-Eingang der zugehörigen Triggerzelle, hätte keine Funktion. Eine Triggerzelle, die disabled ist, könnte sich nicht selbst enablen.

### Triggerzell-Parameter 3: CE Mux und 4: HOWT CE Mux

Multiplexer an den CE-Eingängen der Triggerzellen und HOWT-Komponenten dienen zur Auswahl eines Ticksignals von einem der Tickgeneratoren. Das Steuersignal für den Multiplexer am CE-Eingang der Triggerzelle ist im Datenwort CE Mux des, zur Triggerzelle gehörenden tTriggerCellMemMap-Records gespeichert. Das Steuersignal des zugehörigen HOWT ist im Datenwort HOWT CE Mux des selben Records gespeichert. Da für Triggerzelle und HOWT separate Multiplexer verwendet werden, können Ticksignale individuell zugewiesen werden.
Die Dimensionierung eines Multiplexers wird von der Anzahl vorhandener Tickgeneratoren bestimmt, welche über die benutzerdefinierte Konstante numTGTickGens festgelegt wird. Aus dieser Zahl wird, wiederum über die Logarithmus-Berechnung, die Anzahl der benötigten Bits aus dem ConfRAM ermittelt. Deren Zweierpotenz dient als der Anzahl der Multiplexereingänge.
Abb. 25 stellt den entstandenen Multiplexer in vereinfachter Form dar.

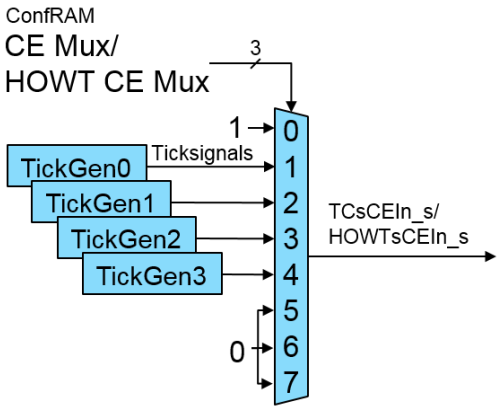


Abb. 25: CE-Multiplexer Blockdiagramm

In diesem Beispiel existieren vier Tickgeneratoren. Ein zusätzlicher Eingang, an dem ein konstanter High-Pegel anliegt, kann vom Benutzer ausgewählt werden, um die betreffende Triggerzelle oder den betreffenden HOWT ohne Verwendung eines Tickgenerators zu betreiben.

### Triggerzell-Parameter 5: ENPC Minval und 6: ENPC Maxval

Die Bitbreiten dieser beiden Parameter sind benutzerdefiniert, wie in Abschnitt „6.1 Dimensionierungsparameter des Systems“ beschrieben wurde. Beide liegen direkt an den Eingängen einer Triggerzelle an und werden als Vergleichsparameter eingesetzt. Deren Bedeutung hängt vom Modus ab, in dem sich eine Triggerzelle befindet. In den Modi zur Flankenzählung, HighEdgeCount und LowEdgeCount, definiert ENPC Maxval die Zählgrenze. Bei deren Erreichen wird ein Triggersignal ausgegeben und mit der Zählung von vorne begonnen. ENPC Minval wird dann nicht verwendet.
In den Modi HighPulseDetect und LowPulseDetect werden CPU-Takte zur Ermittlung einer Pulslänge gezählt. ENPC Minval gibt dabei die Mindestlänge und ENPC Maxval die maximale Länge eines Pulses in CPU-Takten an. Nur wenn ein Puls auftritt, dessen Länge zwischen diesen beiden Grenzwerten liegt, wird ein Triggersignal ausgegeben.
Die benutzerdefinierten Bitbreiten bestimmen sowohl Datenwortbitbreiten im ConfRAM, als auch die Bitbreiten der Triggerzell-Eingänge Minval und Maxval.

### Triggerzell-Parameter 7: HoldOff und 8: Window

Auch die Bitbreiten dieser beiden Parameter sind benutzerdefiniert. Beide liegen direkt an den Eingängen des HOWT einer Triggerzelle an und werden als Vergleichsparameter eingesetzt. Nachdem die Triggerzelle ein Triggersignal erzeugt hat, verzögert der HOWT es bei Bedarf und verändert dessen Ausgabelänge. HoldOff definiert dabei die Anzahl CPU-Takte, um die eine Ausgabe verzögert werden soll. Wird dieser Wert auf 0 gesetzt, erfolgt die Triggerausgabe unverzögert. Window definiert die Ausgabelänge des Triggersignals in CPU-Takten. Wird dieser Wert auf 0 gesetzt erfolgt eine Ausgabe

mit unbegrenzter Länge. Diese kann nur durch einen Reset der Triggerzelle und des HOWT beendet werden, worauf der folgende Abschnitt näher eingeht.

## Triggerzell-Parameter 9: Reset Trigger

Jeder Triggerzelle ist ein Reset-Multiplexer mit einigen zusätzlichen Schaltungselementen zugeordnet. Dies ermöglicht es, für jede Triggerzelle individuell, ein Triggersignal von den Ausgängen der TriggerGenerator-Komponente festzulegen, welches einen Reset einer Triggerzelle und des zugehörigen HOWT durchführt.
Diese Schaltung erfüllt die Anforderung nach einem triggerbedingten Reset. Abb. 26 stellt die Schaltung in Form eines vereinfachten Blockdiagramms dar.

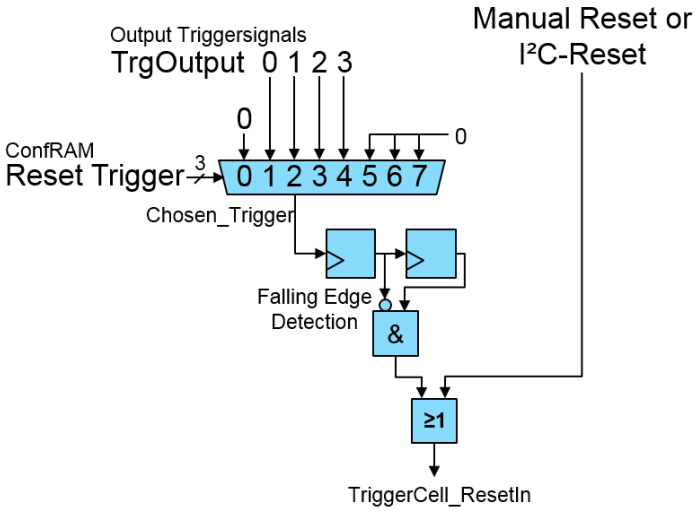


Abb. 26: Reset-Erzeugung für Triggerzelle und HOWT

In dem in Abb. 26 gezeigten Beispiel verfügt die TriggerGenerator-Komponente über vier Ausgänge (TrgOutput0-3), da die benutzerdefinierte Konstante numTGOutputs auf 4 gesetzt wurde. Diese 4 Triggersignalleitungen liegen an einem Multiplexer an, welcher zusätzlich über einen fünften Eingang verfügt an dem eine konstante 0 anliegt. Der ConfRAM-Parameter Reset Trigger liegt als Steuersignal am Multiplexer an. Das vom Reset-Multiplexer ausgegebene Triggersignal wird an eine Schaltung zur Flankenerkennung weitergereicht. Ein Reset der Triggerzelle und des HOWT wird erst bei einer fallenden Flanke, das heißt am Ende eines Triggerpulses, durchgeführt. Ein zweistufiges Schieberegister speichert dazu die Zustände der Triggersignalleitung und eine UND-Verknüpfung dient zur Erkennung einer fallenden Flanke. Der so errechnete Reset-Puls wird ODER-verknüpft mit einer weiteren Resetsignalleitung. Denn ein Reset kann zusätzlich auch manuell durch Betätigung eines Tasters auf dem Nexys2-Board, sowie über ein I²C-Kommando durchgeführt werden.
Ein Reset wird erst bei fallender Triggerflanke durchgeführt, da die Möglichkeit gefordert wird eine mehrstufige Triggerbedingung, nach deren Erfüllung und der daraus resultierenden Triggerausgabe, automatisch zurück setzen zu können. Das ausgegebene Triggersignal hat eine benutzerdefinierte Länge. Würde der Reset zu Beginn der Triggerausgabe durchgeführt werden, könnte dessen gewünschte Länge nicht erreicht werden, da sowohl Triggerzelle, als auch HOWT sofort zurückgesetzt werden würden. Zudem kann auch die Ausgabe eines unbegrenzt langen Triggersignals gewünscht sein. In diesem Fall darf kein automatischer Reset erfolgen, da die Ausgabelänge dann nicht unbegrenzt, sondern sehr kurz wäre. Es bleibt dem Benutzer überlassen die Ausgabe eines Triggersignals mit unbegrenzter Länge über das I²C-Kommando zu beenden, welches alle Triggerzellen und deren HOWT zurücksetzt.
Die Bitbreite des Reset-Multiplexers wird über den Zweierlogarithmus der Anzahl von Triggerausgängen (numTGOutputs), erhöht um eins, berechnet. Denn wenn kein Gebrauch von der

Funktion zum triggerbedingten Reset gemacht werden soll, kann anstelle eines Triggersignals auch der Multiplexereingang ausgewählt werden, an dem die konstante 0 anliegt.
Zur Berechnung eines Reset-Signals werden nicht die von den Triggerzellen erzeugten Triggersignale verwendet, das heißt von deren HOWT ausgegebene Triggersignale, sondern die auf den Ausgängen der TriggerGenerator-Komponente ausgegebenen Triggersignale. Grund dafür ist der hardwarebezogene Aufwand zur Umsetzung größerer Multiplexer, welcher hier zur Erfüllung der Anforderungen nicht erforderlich ist. Die Größe der Konstanten numTGOutputs (Anzahl der Triggerausgänge zum Benutzer) ist auf 8 beschränkt, während numTriggerCells bis zu 32 betragen kann. Es wird davon ausgegangen, dass das Triggersignal einer mehrstufigen Triggerbedingung vom Benutzer verwendet wird und somit auf einem Ausgang der TriggerGenerator-Komponente ausgegeben wird. Deren Ausgangsschaltungen (OutputAND-Blöcke, siehe Abb. 12, S. 39) können auch Teil der Triggerberechnung sein, nämlich wenn eine Triggerbedingung mehrere nebenläufig erfüllte mehrstufige Triggerbedingungen umfasst. Die vom Triggergenerator, zum Benutzer hin ausgegebenen Triggersignale für die Reset-Berechnung zu verwenden ist somit die sinnvollere Alternative.
Es sei angemerkt, dass diese Designentscheidung es gegebenenfalls notwendig macht mehr Triggerausgänge zu belegen, als für die reinen Triggerausgaben gebraucht würden. Möglicherweise möchte der Benutzer eine mehrstufige Triggerbedingung zu einem bestimmten Zeitpunkt zurücksetzen, welcher über eine andere mehrstufige Triggerbedingung ermittelt wird. In diesem Fall muss ein Ausgangsblock (OutputAND) des Triggergenerators verwendet werden um ersteres Triggersignal auszugeben, welches nur als Resetsignal dient und sonst nicht gebraucht wird. Dieser Kompromiss wurde eingegangen, da er nur in seltenen Fällen zu einem Nachteil wird, nämlich wenn der Benutzer 8 Triggersignale benötigt. Grundsätzlich werden bei dieser Art der Verschaltung Ressourcen gespart, die nicht mehr zur Verfügung stünden, wenn die Ausgabesignale aller Triggerzellen auf den Resetmultiplexer-Eingang jeder anderen Triggerzelle gelegt würden.
In einem späteren Abschnitt wird ein Anwendungsbeispiel gezeigt, das es notwendig macht Triggerbedingungen nur zu Reset-Zwecken zu verwenden.

### 6.2.2 Weitere Parameter im ConfRAM

Die übrigen, im ConfRAM gespeicherten Parameter seien hier der nur kurz erwähnt. Ausführliche Erläuterungen zu deren Bedeutungen und Wertebereichen befinden sich in Anhang D.

Für jeden OutputAND-Ausgabeblock der TriggerGenerator-Komponente (siehe Abb. 12, S. 39) wird ein Datenwort im ConfRAM gespeichert. Da zukünftige Erweiterungen weitere Parameter erfordern können, wurde dazu der Record-Typ **tTriggerGenOutputMemMap** definiert, welcher in Abb. 27 dargestellt ist.

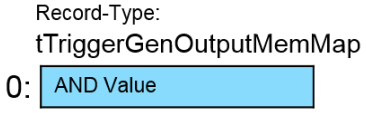


Abb. 27: Record-Type tTriggerGenOutputMemMap

Jeder OutputAND-Block ermöglicht es eine beliebige Kombination der Triggerzellen zu definieren, deren triggern Vorraussetzung für die Ausgabe eines Triggersignals zum Benutzer hin sind. Ein Triggersignal wird nur ausgegeben, solange alle relevanten Triggerzellen, bzw. deren HOWT, gleichzeitig ein Triggersignal ausgeben. Der Parameter AND Value speichert die relevanten Triggerzellen in Form eines Bitmap. Dessen Bitbreite entspricht somit der benutzerdefinierten Konstanten numTriggerCells. Ist beispielsweise nur das niederwertigste Bit gesetzt des Parameters AND Value gesetzt, wird das von Triggerzelle0 erzeugte Triggersignal direkt zum Benutzer hin ausgegeben. Wird auch noch das zweite Bit gesetzt, erfolgt eine Triggerausgabe zum Benutzer nur, solange Triggerzelle0 und Triggerzelle1 gleichzeitig ein Triggersignal ausgeben.

Als Anwendungsbeispiel sei eine einfache UND-Verknüpfung zweier Signalleitungen als Triggerbedingung erwähnt. Jede Triggerzelle wartet auf den High-Zustand einer Signalleitung.
Sind beide Signalleitungen gleichzeitig im High-Zustand, gibt der OutputAND-Block das Triggersignal für den Benutzer aus.

Der Record-Typ **tTickGenMemMap** ist in Abb. 28 dargestellt.

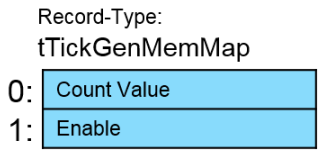


Abb. 28: Record-Type tTickGenMemMap

Dieser umfasst folgende zwei Parameter:

- **Count Value**: Zeitraum zwischen den Ticks in CPU-Takten. Die Bitbreite des Datenworts im ConfRAM ist benutzerdefiniert (siehe Abb. 18, S. 53).
- **Enable**: 1 Bit breiter Parameter zum Anhalten des Tickgenerators.

Der Record-Typ **tInToOutSwitchMemMap** ist in Abb. 29 zu sehen.

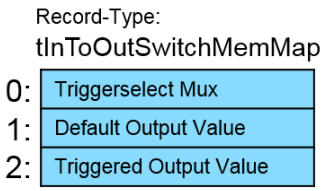


Abb. 29: Record-Type tInToOutSwitchMemMap

Folgende drei Parameter werden für jeden InToOutSwitch im ConfRAM gespeichert:

- **Triggerselect Mux**: Steuersignal eines Multiplexers zur Auswahl einer beliebigen Triggerzelle, deren Triggersignal eine Unterbrechnung des weitergeleiteten Signals bewirken soll. Dieser Multiplexer wird, abhängig von der Konstanten numTriggerCells, generiert.
- **Default Output Value**: Ausgabesignal des InToOutSwitch im nicht-getriggerten Zustand. Die Bitbreite dieses Parameters beträgt 2 Bit, da nur 3 Ausganesignale unterschieden werden. Ausgegeben wird: Ein Low-Pegel, ein High-Pegel oder das Eingangssignal des InToOutSwitch.
- **Triggered Output Value**: Ausgabesignal für die Dauer eines Triggerpulses von der ausgewählten Triggerquelle. Auch für diesen Parameter beträgt die Bitbreite 2 Bit.

Den Record-Typ **tPulseFilterMemMap** zeigt Abb. 30.

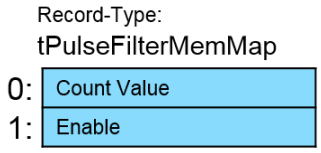


Abb. 30: Record-Type tPulseFilterMemMap

Ein Pulsfilter erlaubt die Konfiguration zur Laufzeit über folgende Parameter:

- **Count Value**: Maximale Pulslänge in CPU-Takten, bis zu der Pulse gefiltert werden. Nur längere Pulse werden vom Filter durchgelassen. Die Bitbreite dieses Parameters ist benutzerdefiniert.
- **Enable**: 1 Bit breiter Parameter zum Anhalten des Pulsfilters.

## 6.3 Organisation des ConfRAM

Die im vorherigen Abschnitt beschriebenen Record-Typen haben drei Aufgaben. Ihre Größe wird zur Berechnung der ConfRAM-Größe, als Anzahl von Datenworten benutzt. Die geben dem ConfRAM seine Struktur, sodass der Benutzer ermitteln kann an welcher Adresse welcher Parameter steht. Außerdem vereinfachen sie die Weitergabe der Parameter aus dem ConfRAM an die Empfängerkomponente. Zunächst soll die Struktur des ConfRAM betrachtet werden, welche in Abb. 31 dargestellt wird.

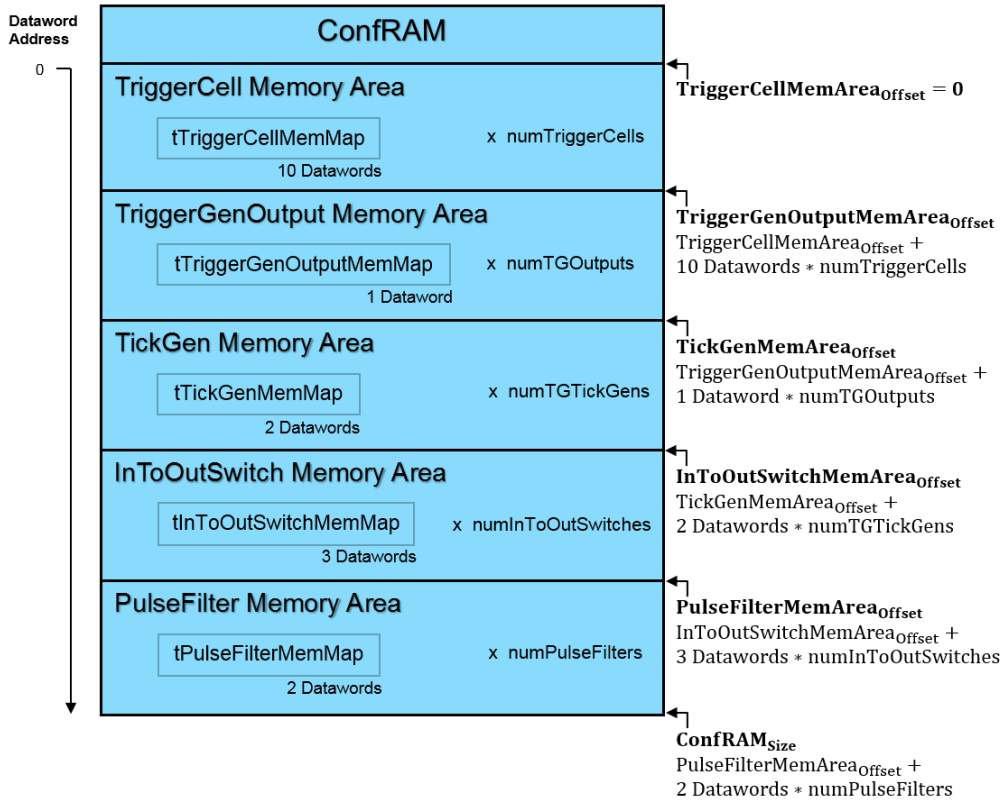


Abb. 31: ConfRAM Speicherformatierung

Für jeden Record-Typ existiert ein Bereich im ConfRAM, in dem mehrere Record-Instanzen hintereinander gespeichert sind. Deren Anzahl wird über die benutzerdefinierten Konstanten festgelegt. Mit dem ersten Datenwort des ConfRAM beginnt der Triggerzell-Speicherbereich. In diesem liegt für jede Triggerzelle eine Kopie des tTriggerCellMemMap-Records. Die Größe dieses Bereichs, in Datenworten, entspricht demnach der Anzahl von Triggerzellen (numTriggerCells), multipliziert mit der Anzahl von Datenworten pro Triggerzelle (10). Auf diese Weise werden die Größen und Offsets aller Speicherbereiche bestimmt, wie Abb. 31 zeigt.

## 6.4 Funktionsweise des I²C-Controllers

Die Aufgaben des I²C-Controllers umfassen die Verwaltung von I²C-Slave und I²C-Master, das Beschreiben und Auslesen des ConfRAM, sowie die Ausführung von Benutzerkommandos. Die Umsetzung dieser Aufgaben wird in den folgenden Abschnitten beschrieben.

### Verwaltung des I²C-Slave

Der I²C-Slave empfängt Lese- und Schreib-Anfragen vom Reader. Nach Eintreffen einer Anfrage teilt er dem I²C-Controller mit, ob der Reader Daten senden oder Daten aus dem ConfRAM auslesen will. Der I²C-Controller verbindet I²C-Slave und ConfRAM in der Art, dass das vom ConfRAM ausgegebene Datenwort direkt am Dateneingang des I²C-Slave anliegt und umgekehrt das vom Datenausgang des I²C-Slave ausgegebene Datenwort am Eingang des ConfRAM anliegt. Abb. 32 zeigt dies an einem Blockdiagramm.

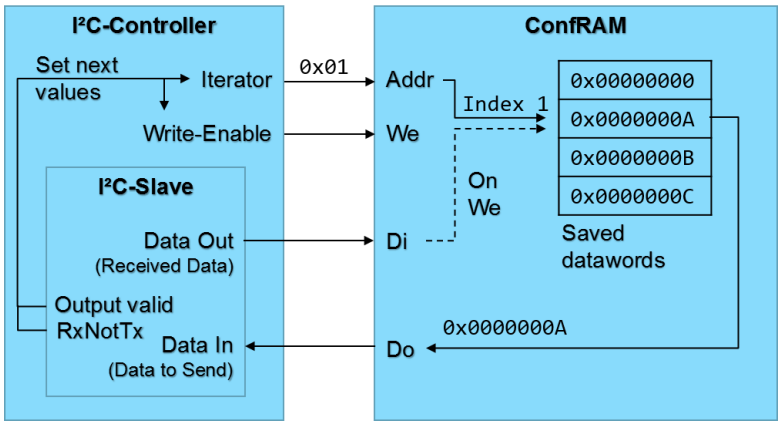


Abb. 32: Verbindung zwischen I²C-Slave und ConfRAM

Wann immer der Reader Daten vom I²C-Slave anfordert, bekommt er das aktuell adressierte Datenwort aus dem ConfRAM zurück. Der I²C-Controller ist dafür zuständig den Speicheriterator anschließend auf das nächste Speicherwort zu setzen, sodass hintereinander liegenden Datenworte nacheinander ausgelesen werden können. Dazu teilt der I²C-Slave dem I²C-Controller das Ende der Übertragung des Datenworts mit (über Output valid), welcher dann die nächste Position des Iterators berechnet. Sendet der Reader zu speichernde Daten zum I²C-Slave, berechnet der I²C-Controller ebenfalls die nächste Position des Iterators und sendet zudem ein Write-Enable Signal zum ConfRAM. Dadurch wird das zuletzte empfangene Datenwort, welches jederzeit am Dateneingang des ConfRAM anliegt, in den Speicher übernommen.
Abb. 32 zeigt eine beispielhafte Signalbelegung zur Verdeutlichung des Prinzips. Der Iterator speichert hier den Wert 0x1, wodurch das zweite Datenwort im ConfRAM adressiert wird. Der Do-Ausgang des ConfRAM gibt den an dieser Stelle gespeicherten Wert 0x0000000A aus. Sobald ein Write Enable-Signal zum ConfRAM gesendet wird, wird der gespeicherte Wert mit dem, am Di-Eingang anliegenden, Wert überschrieben. In diesem Beispiel handelt es sich dabei um die zuletzt auf dem I²C-Bus übertragenen Bits, welche der I²C-Slave stets auf seinem Data Out-Ausgang ausgibt. Einem vollständig übertragenen Datenwort entspricht diese Bitfolge nur, wenn der I²C-Slave ein Output Valid-Signal ausgibt. Nur dann darf demzufolge ein Write Enable-Signal zum ConfRAM gesendet werden.

## Verwaltung des Speicheriterators

Den Ablauf bei der Ermittlung des nächsten Iteratorwerts zeigt Abb. 33. Die verschachtelten Abfragen der dazu implementierten kombinatorischen Logik werden hier in Form eines Programmablaufplans dargestellt.

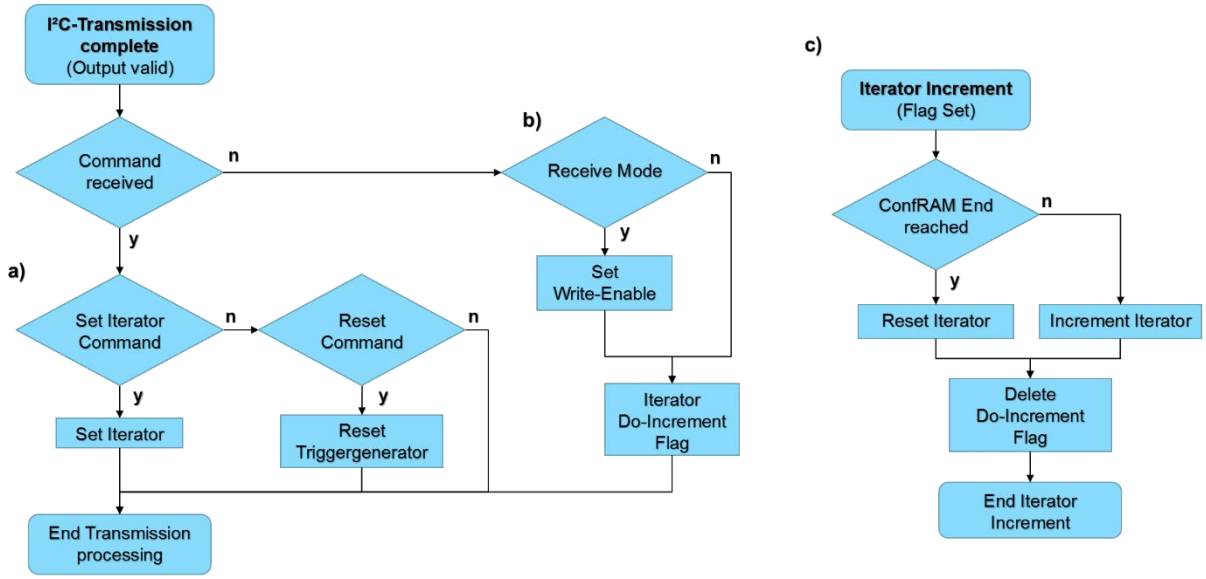


Abb. 33: Programmablaufplan zur Verwaltung des Speicheriterators

Der Reader sendet sowohl, im ConfRAM zu speichernde, Daten zum I²C-Slave, als auch Kommandos, die nicht gespeichert, sondern ausgeführt werden sollen. Alternativ liest der Reader, im ConfRAM gespeicherte, Daten aus. Jede vollständige Übertragung eines Datenworts bestätigt der I²C-Slave mit einem Output Valid-Signal. Dies zeigt dem I²C-Controller an, dass die vom I²C-Slave ausgegebenen Daten nun gültig sind. Zu diesem Zeitpunkt erfolgt die Aktualisierung der Eingangssignale des I²C-Slave. Beides zeigt Abb. 33. Mit dem Output Valid-Signal beginnt zudem die Verarbeitung des zuletzt übertragenen Datenworts, welches der I²C-Slave auf seinem Data Out-Ausgang ausgibt. Dabei kann es sich um ein empfangenes Kommando, ein empfangenes zu speicherndes Datenwort oder um ein, vom I²C-Slave zum Reader übertragenes, Datenwort handeln.
Es sein angemerkt, dass der I²C-Slave, auch wenn er selber Daten wegschickt, stets alle auf dem Bus übertragenen Daten mitliest und diese auf dem Data Out-Ausgang ausgibt. Somit könnte ein Kontrollalgorithmus implementiert werden, welcher überprüft, ob es zu Übertragungsfehlern gekommen ist.

Ein Kommando wird über das höchstwertigste Byte der Nachricht erkannt (siehe Abschnitt „5.3.2 I²C-Controller“). Es folgt die Unterscheidung der Kommandos, anhand des zweit-höchstwertigsten Bytes und anschließend die Ausführung der kommandospezifischen Aktionen (siehe Abb. 33-a):

- **Reset-Kommando**: Der I²C-Controller erzeugt einen Reset-Puls für die TriggerGenerator-Komponente. Der Iterator bleibt in diesem Fall unverändert.
- **Set Iterator-Kommando**: Der neue Wert für den Iterator wird direkt im Kommandowort mitgeliefert und überschreibt den aktuellen Iteratorwert.
- **Unbekanntes Kommando**: Der Iterator wird nicht verändert.

Wurde kein Kommando empfangen, hat der I²C-Slave ein zu speicherndes Datenwort empfangen oder ein gespeichertes Datenwort zum Reader gesendet, was jeweils auch zur Ausgabe eines Output Valid-Signals führt.
Über den RxNotTx-Ausgang zeigt der I²C-Slave an, ob er sich im Receive-Modus befindet und Daten vom Reader empfängt oder ob er sich im Send-Modus befindet und Daten zum Reader sendet. Nur

wenn sich der I²C-Slave im Receive-Modus befindet, gibt der I²C-Controller ein Write Enable-Signal zum ConfRAM hin aus (siehe Abb. 33-b), woraufhin der ConfRAM das vom I²C-Slave ausgegebene Datenwort an die Speicherstelle schreibt, auf die der Iterator zeigt.
Außerdem wird, unabhängig von der Übertragungsrichtung, die Hilfsvariable Do-Increment gesetzt. Diese verdient besondere Beachtung, da sie die Iteratorverwaltung besonders einfach macht. Die zugrunde liegende Problematik zeigt Abb. 34.

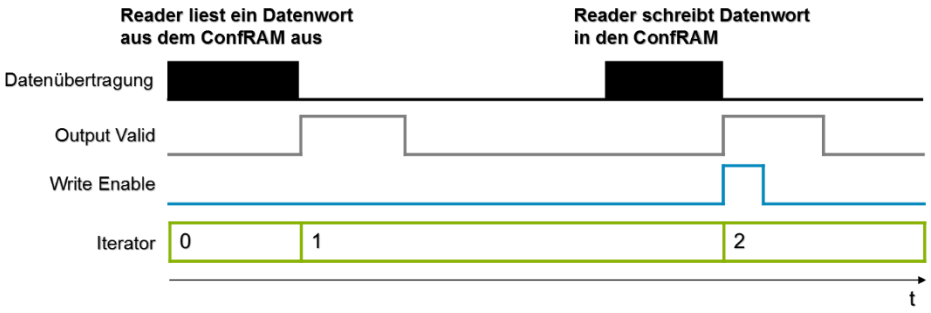


Abb. 34: Speicherzugriffsproblem das gelöst wurde

In diesem Beispiel liest der Reader zunächst das Datenwort an ConfRAM-Adresse 0 aus. Anschließend gibt der I²C-Slave das Output Valid-Signal aus, woraufhin der Iterator auf die nächste Adresse gesetzt wird. Bei der darauffolgenden Übertragung wünscht der Reader allerdings ein Datenwort in den ConfRAM zu schreiben. Da zuletzt Adresse 0 ausgelesen wurde, wird erwartet, dass dieser Schreibvorgang nun auf Adresse 1 erfolgt. Dem ist jedoch nicht so. Nach Empfang des zu speichernden Datenworts, gibt der I²C-Slave wieder ein Output Valid-Signal aus. Das führt dazu, dass der Iterator auf die Adresse 2 gesetzt wird. Da die Übertragungsrichtung diesmal zum I²C-Slave hin war, wird zeitgleich ein Write Enable-Signal zum ConfRAM hin ausgegeben. Dadurch wird das empfangene Datenwort nicht, wie erwartet, an Adresse 1 gespeichert, sondern an Adresse 2 des ConfRAM.
An dieser Stelle tritt die Hilfsvariable Do-Increment in Erscheinung. Diese Verzögert die Inkrementierung des Iterators um einen CPU-Takt, wie in Abb. 35 zu sehen ist.

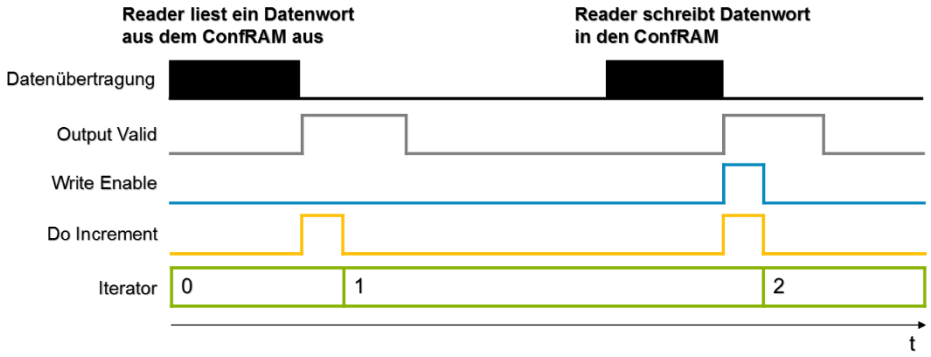


Abb. 35: Lösung des Speicherzugriffsproblems mit der Hilfsvariable Do-Increment

Das Output Valid-Signal führt nun nicht mehr zur unmittelbaren Inkrementierung des Iterators, sondern dazu, dass Do-Increment gesetzt wird. Dies startet die Neuberechnung des Iteratorwerts (siehe Abb. 33-c). Dort wird zuerst geprüft, ob der Iterator bereits auf das letzte Datenwort des ConfRAM zeigt. Ist dem so, wird der Iterator auf die Adresse 0 gesetzt, andernfalls wird er um eins inkrementiert. Anschließend wird Do-Increment wieder gelöscht.
In diesem Beispiel liest der Reader wieder das Datenwort von Adresse 0 aus. Anschließend wird der Iterator auf Adresse 1 gesetzt. Danach empfängt der I²C-Slave ein Datenwort vom Reader und gibt dann ein Output Valid-Signal aus. Dies bewirkt nun, dass ein Write Enable-Signal zum ConfRAM gesendet und zudem Do-Increment gesetzt wird. Der ConfRAM speichert das empfangene Datenwort

somit an Adresse 1. Erst einen CPU-Takt später wird Do-Increment wieder gelöscht und der Iterator auf Adresse 2 gesetzt.
Die Verzögerung der Iteratorinkrementierung um einen CPU-Takt hat keine negativen Auswirkungen auf eine mehrere Datenworte umfassende I²C-Übertragung. Denn durch das I²C-Protokoll bedingt, werden zwischen den Nutzdatenbytes Acknowledge-Bits übertragen, welche einen zeitlichen Puffer zur Ermittlung der im Folgenden übertragenen Nutzdaten darstellt. Dieser ist, aufgrund der relativ geringen Datenrate, mehr als ausreichend groß genug für diese Operation.

Das hier geschilderte Problem besteht darin, dass es nur einen Zeitpunkt zur Durchführung von Berechnungen gibt, welcher vom Output Valid-Signal angezeigt wird. Die Umsetzung der Berechnungen kann entweder vollständig zu diesem Zeitpunkt ausgeführt werden oder in Form einer Sequenz von Einzelberechnungen. Der erste Lösungsansatz scheint zunächst einfacher umsetzbar zu sein, die bei der Implementierung gemachten Erfahrungen wiesen jedoch auf das Gegenteil hin. Es hat sich gezeigt, dass Umstrukturierungen des gesamten Ablaufs, um alle Berechnungen zum selben Zeitpunkt durchführen zu können, wesentlich umständlicher umzusetzen sind, als die Implementierung der hier beschriebenen zweistufigen Berechnung. Der betroffene Quellcode konnte so um mehr als die Hälfte verkürzt werden und dessen Verständlichkeit hat sich ebenfalls verbessert.

**Verwaltung des I²C-Master**

Der I²C-Master überträgt byteweise Daten aus dem DisplayDataMemory zum Display. Innerhalb einer Übertragung, welche mit jedem Output Valid-Signal des I²C-Slave beginnt, wird der gesamte Inhalt des DisplayDataMemory übermittelt. Der I²C-Master gibt dabei nach jedem übertragenen Byte, äquivalent zum I²C-Slave, ein eigenes Output Valid-Signal aus. Der I²C-Controller inkrementiert zu diesem Zeitpunkt einen zweiten Iterator, welcher zur Adressierung des DisplayDataMemory verwendet wird. Wurde dessen gesamter Inhalt übertragen, wird der I²C-Master vom I²C-Controller abgeschaltet. Mit dem nächsten Output Valid-Signal des I²C-Slave beginnt die gleiche Übertragung von vorne. Auf diese Weise erfolgt eine Aktualisierung der Displayausgabe nach jeder Änderung des ConfRAM-Inhalts.

## 6.5 Bewertung der Implementierung

Die Generierung und Verknüpfung von Komponenten mithilfe generischer Berechnungen ermöglicht es auch Benutzern, die kaum mit dem internen Aufbau des Triggergenerators vertraut sind, über benutzerdefinierte Konstanten, weitreichende Anpassungen des Systems vorzunehmen.
Der Benutzer kann selber entscheiden, wie die Hardwareressourcen des FPGA eingesezt werden sollen und sich für einen Kompromiss entscheiden, zwischen wenigen Triggerbedingungen bei maximaler Anpassbarkeit zur Laufzeit und vielen Triggerbedingungen mit eingeschränkter Anpassbarkeit zur Laufzeit. Durch die Einschränkung der Möglichkeiten einzelner Triggerbedingungen kann eine spezifische Umverteilung der Hardwareressourcen vorgenommen werden, sodass auch komplexere Anwendungsszenarien umsetzbar sind. Unterstützend dazu können die Möglichkeiten der Hardware, durch die individuelle Reduzierung der Bitbreite jedes Datenworts des ConfRAM auf die tatsächlich benötigte Größe, effizient ausgenutzt werden.
Die implementierten Schnittstellen zwischen I²C-Slave und ConfRAM erlauben eine direkte, einfach nachvollziehbare Verknüpfung der Komponenten. Auch die Implementierung der Speicheradressierung unter Benutzung eines Iterators konnte, mithilfe des kleinen Tricks, aus einer großen Berechnung eine Sequenz aus zwei Einzelberechnungen zu macnen, stark vereinfacht werden. Dies erleichtert das Verständnis der Schaltung für Mitarbeiter, die an der zukünftigen Weiterentwiclung des Triggergenerators beteiligt sind.

# 7 Messungen

Die folgenden Abschnitte zeigen, dass die genannten Anforderungen von dem implementierten Triggergenerator erfüllt werden. Zunächst wird der verwendete Testaufbau beschrieben, worauf die Erläuterung zweier Anwendungsbeispiele folgt. Das erste Beispiel ist von geringem Umfang und zeigt, dass die geforderten grundlegenden Triggerfunktionalitäten vorhanden sind. Das zweite Beispiel stellt ein umfangreiches und praxisrelevantes Anwendungsszenario dar und dient dazu die Umsetzbarkeit komplexer, zusammengesetzter Bedingungen zu belegen, welche im späteren praktischen Einsatz benötigt werden. Dazu zählen beispielsweise die Erkennung eines bestimmten übertragenen Nachrichtenbytes oder die Ermittlung des Endes einer Nachricht. Das zweite Beispielszenario wurde von den Smartcard-Testern so definiert, dass alle derzeit bekannten, komplexeren Triggerbedingungen enthalten sind, die im praktischen Einsatz benötigt werden.
Im Anschluss werden weitere Testergebnisse, zur möglichen Dimensionierung des Systems und Genauigkeiten bei der Triggererzeugung, bewertet. Anhand dieser Tests wird gezeigt, dass die geforderten Zeitvorgaben, bezüglich Reaktionsgeschwindigkeiten und Reichweiten, erreicht werden.
Die notwendigen Schritte zur Konfiguration des Triggergenerators, bei Umsetzung der beiden Anwendungsbeispiele, werden detailliert im User Guide beschrieben, welcher in Anhang A zu finden ist. Auf Wunsch der Firma NXP Semiconductors, in der dieses Projekt umgesetzt wurde, sind dort alle Möglichkeiten zur Konfiguration des Systems im Detail beschrieben. Behandelt werden sowohl die Konfiguration mehrstufiger Triggerbedingungen zur Laufzeit, als auch die Dimensionierung einzelner Komponenten vor der Synthese. Auch die Weiterleitung von Signalleitungen, deren Konfiguration einen Eingriff in den Quellcode durch den Benutzer erfordert, wird dort erläutert. Außerdem wird die Anpassung der Displayausgabe beschrieben, inklusive der Anwendung des implementierten Programms zur Generierung der VHDL-Quellcodedatei, welche den Speicher der auszugebenden Zeichen enthält.

## 7.1 Testaufbau

Während der Entwicklung und der Tests wurde der in Abb. 36 gezeigte Aufbau verwendet.

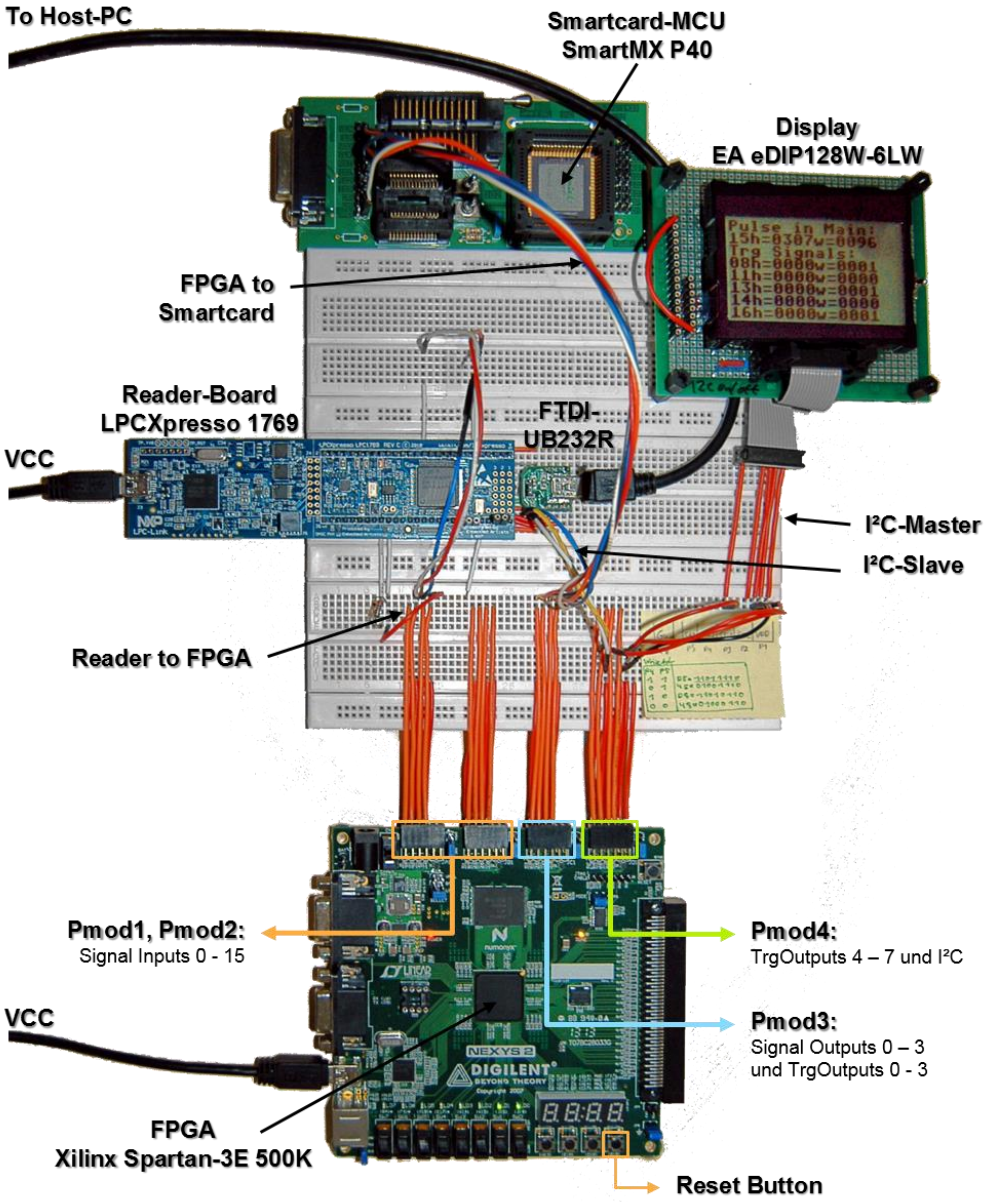


Abb. 36: Testaufbau

In der Mitte der Abbildung ist das Reader-Board zu sehen. Dieses ist über eine UART-Schnittstelle mit einem USB/UART-Adapter der Firma FTDI verbunden, welcher Daten vom Host-PC empfängt und zum Reader übermittelt.

Im unteren Teil der Abbildung ist das Nexys2-Board zu sehen. Dieses verfügt über vier Pmod-Anschlüsse, mit jeweils acht Signalleitungen. Die vom Reader abgehenden Smartcard-Kommunikationssignalleitungen (PWR, RST, IO und CLK) sind mit Pmod1 verbunden. Zusammen mit Pmod2 stehen insgesamt 16 Leitungen für Eingangssignale zur Verfügung, von denen in diesem Aufbau jedoch nur vier verwendet werden. Die I²C-Schnittstelle des Readers überträgt Einstellungsdaten vom Benutzer zum Triggergenerator und ist an Pmod4 angeschlossen.

Das Nexys2-Board gibt die durchgeleiteten Smartcard-Kommunikationssignale über die obere Hälfte von Pmod3 an die Smartcard aus. Die untere Hälfte von Pmod3 und die obere Hälfte von Pmod4 umfassen die acht Triggerausgänge des Triggergenerators.

Die untere Hälfte von Pmod4 enthält, neben der I²C-Schnittstelle zum Reader, auch die I²C-Schnittstelle zum Display.
Das Display wurde mit einer Platine geliefert, welche es ermöglicht die relevanten Pins des Displays über ein Flachbandkabel nach außen zu führen.
Zur Einbindung des Smartcardprozessors in den Aufbau wurde eine Adapterplatine verwendet, welche im oberen Teil des Bildes zu sehen ist.
Zur direkten Steuerung des Triggergenerators kommt der Button BTN0 des Nexys2-Boards zum Einsatz. Dessen Betätigung setzt alle Komponenten des Triggergenerators in deren Anfangszustand zurück, auch die benutzerdefinierten Einstellungen im ConfRAM werden dadurch gelöscht.
In diesem Aufbau vorhanden, jedoch unter dem Reader-Board angebracht und daher nicht sichtbar, sind drei Widerstände, die von den Smartcard-Testern auf jeweils 10kΩ dimensioniert wurden. Abb. 37 zeigt deren Positionen im Testaufbau.

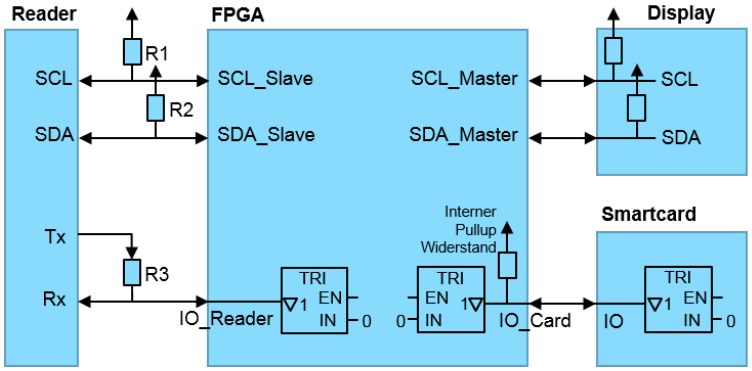


Abb. 37: Pullup-Widerstände im Testaufbau

Die I²C-Master-Implementierung des Readers stellt keine internen Pullup-Widerstände zur Verfügung, weshalb diese dem Testaufbau hinzugefügt werden mussten (siehe R1 und R2 in Abb. 37).
Die I²C-Slave-Implementierung des Displays verfügt hingegen über interne Pullup-Widerstände, weshalb dort keine externen benötigt werden.
Die IO-Signalleitung zwischen Reader und Smartcard wird vom Triggergenerator bidirektional weitergeleitet. Auf Seite des Readers ist ein Widerstand (R3) zur Strombegrenzung erforderlich, denn die Funktion des Pullup-Widerstandes übernimmt das Ausgabesignal des Readers. Dies ist eine Eigenheit des verwendeten Readers, da üblicherweise ein Pullup-Widerstand zum Einsatz kommt (vgl. Effing 2008, S. 74). Zur bidirektionalen Kommunikation zwischen FPGA und Reader, stellt der Reader ein High-Signal über den Ausgang Tx zur Verfügung. Um Daten zu senden, zieht der FPGA dieses über einen TriState-Treiber auf Low. Der dabei fließende Strom vom Reader zum Massepotential im FPGA wird über R3 begrenzt. Der Reader gibt, um Daten zu senden, die Low-Pulse direkt über den Ausgang Tx aus. Die Datenübertragung zwischen FPGA und Smartcard erfolgt nach dem Open-Collector-Prinzip. Die IO-Signalleitung wird über einen FPGA-internen Pullup-Widerstand im High-Zustand gehalten und zur Datenübertragung jeweils über TriState-Treiber von FPGA oder Smartcard auf Low gezogen.
Die zwischen Reader und Smartcard fließenden Ströme betragen weniger als 100 µA. Laut Datenblatt des verwendeten FPGAs, können dessen Ausgänge Ströme von bis zu 16 mA ausgeben. Aus diesem Grund und da alle Komponenten mit 3.3 V arbeiten, werden Reader und Smartcard direkt mit den Ein- und Ausgängen des FPGA (bzw. des Nexys2-Boards) verbunden.

## 7.2 Testfälle

In den folgenden Abschnitten wird jeweils eine mögliche Umsetzung der beiden Eingangs beschriebenen Beispielszenarien (siehe „3.2 Allgemeine Anforderungen“) vorgestellt und die daraus resultierenden Messergebnisse erläutert. Das erste Beispiel dient dazu die Anwendung der geforderten Grundfunktionalitäten zu demonstrieren, während das zweite Beispiel deren Einsatz in einem umfangreichen, praxisrelevanten Anwendungsszenario zeigt.

### 7.2.1 Beispiel 1: Demonstration der Unterbedingungsarten

Für die Umsetzung des Abschnitt „3.2.1 Unterbedingungsarten“ beschriebenen Anwendungsszenarios wurden fünf Triggerzellen eingesetzt. Abb. 38 zeigt deren Verschaltung.

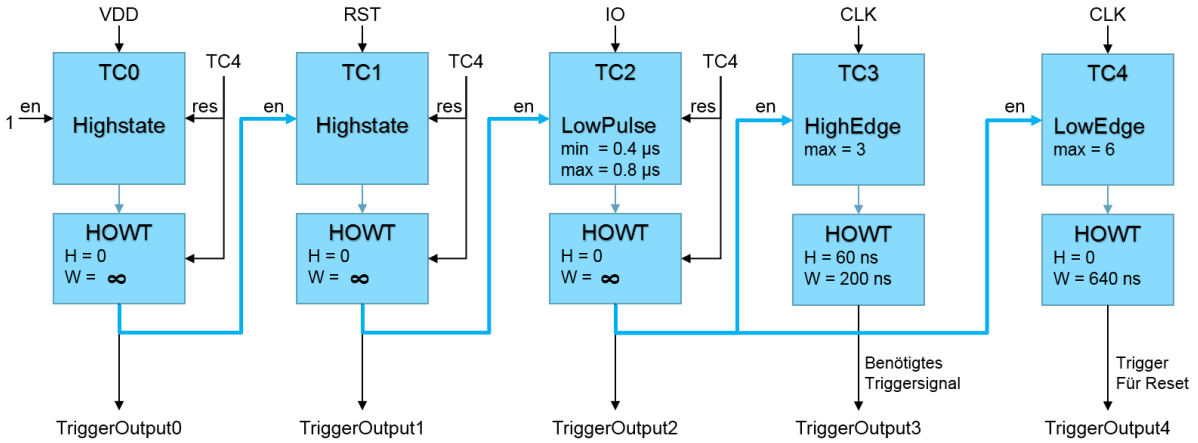


Abb. 38: Verschaltung der Triggerzellen in Beispiel 1

Die Triggerzellen haben in diesem Beispiel folgende Funktionen:

- **Triggerzelle TC0**: Wartet darauf, dass sich die Signalleitung VDD im High-Zustand befindet und gibt dann ein Triggersignal unbegrenzter Länge aus. Als erste Unterbedingung der mehrstufigen Triggerbedingung ist diese unbedingt enabled.
- **Triggerzelle TC1**: Wartet ab dem enable durch TC0 bis sich die Signalleitung RST im High-Zustand befindet. Gibt dann ebenfalls ein unbegrenzt langes Triggersignal aus.
- **Triggerzelle TC2**: Wartet ab dem enable durch TC1 auf einen, auf der Signalleitung IO übertragenen Low-Puls, welcher folgende Kriterien erfüllt: Die Mindestlänge muss 0,4 µs betragen und er darf nicht länger als 0.8 µs sein. Nach dessen Feststellung wird wiederum ein Triggersignal von unbegrenzter Länge ausgegeben.
- **Triggerzelle TC3**: Wartet ab dem enable durch TC2 bis die dritte steigende Flanke auf der Signalleitung CLK übertragen wurde. Anschließend wird ein Triggersignal mit einer Länge von 200 ns nach einer Verzögerung von 60 ns ausgegeben.
- **Triggerzelle TC4**: Wartet ab dem enable durch TC2 auf die sechste steigende Flanke auf der Signalleitung CLK. Ist diese aufgetreten, wird ein Triggersignal mit einer Länge von 640 ns ohne Verzögerung ausgegeben. Dieses dient dazu die gesamte mehrstufige Triggerbedingung in den Startzustand zurück zu versetzen und liegt deshalb an den Reset-Eingängen aller Triggerzellen an, die ein Triggersignal von unbegrenzter Länge ausgeben.

Die von den Triggerzellen TC0, TC1 und TC2 und deren HOWT realisierten Triggerbedingungen erzeugen jeweils ein Triggersignal mit unbegrenzter Länge, da nicht bekannt ist in welchen zeitlichen Abständen die Ereignisse eintreten. Die Ausgabe ihrer Triggersignale auf den Ausgängen TriggerOutput0, TriggerOutput1 und TriggerOutput2 erfolgt lediglich zur Veranschaulichung.

Der Konfiguration von TC2 liegt die Länge des, von der Smartcard erzeugten Triggerpulses, zugrunde, welcher eine Länge von ca. 0,5 µs hat.
Die Konfiguration von TC3 auf drei zu zählende Flanken, orientiert sich an dem eingangs genannten Beispiel (siehe Abb. 5, S. 22). Für die Verzögerung und Länge der Triggerausgabe gab es keine Vorgaben. Daher wurden Werte gewählt, die sich gut veranschaulichen lassen.
Bei der Konfiguration von TC4 war es wichtig, dass deren Trigger nach dem von TC3 erzeugt wird, um nicht zu früh einen Reset zu erzeugen, welcher die Triggererzeugung durch TC3 verhindern würde. Die Länge des ausgegebenen Triggersignals von TC4 wurde auf einen frei gewählten Wert gesetzt, um die Möglichkeiten der Schaltung zu verdeutlichen und da das Triggersignal gleichzeitig zur Steuerung eines InToOutSwitches verwendet wird. Letzterer unterbricht die Weiterleitung des CLK-Signals für eben diese Zeitspanne.
Je nach Anwendungsfall kann es auch sinnvoll sein die Ausgabelänge des als Reset verwendeten Triggersignals so zu dimensionieren, dass es nach dem Ende der Datenübertragung endet. So würde die Triggerschaltung erst nach Übertragungsende wieder in den Anfangszustand versetzt werden. Abhängig von den Anforderungen an die erzeugten Triggersignale wäre es auch denkbar TC4 einzusparen werden und das Triggersignal von TC3 zur Reseterzeugung zu verwenden. Damit soll verdeutlicht werden, dass der Benutzer viele Freiheiten beim Einsatz der Triggerzellen hat und es je nach Anwendungsfall auch mehrere Triggerzellenschaltungen geben kann, die zu ähnlichen Ergebnissen führen.
Dieses Beispiel zeigt zusätzlich die Verwendung eines Pulsfilters innerhalb der BiDirIO-Komponente, welche die bidirektionale Weiterleitung des IO-Signals durchführt. Der Pulsfilter wurde so konfiguriert, dass alle von der Smartcard kommenden Pulse mit einer Länge von unter 1,2 µs nicht zum Reader durchgelassen werden, welche eine Länge von ca. 0,5 µs haben.
Des Weiteren wurde, wie bereits angedeutet, ein InToOutSwitch integriert, um die Weiterleitung des CLK-Signals vom Reader zur Smartcard, mithilfe des Triggersignals von TC4, unterbrechen zu können.
Abb. 39 zeigt eine Übersicht der gemessenen Signale.

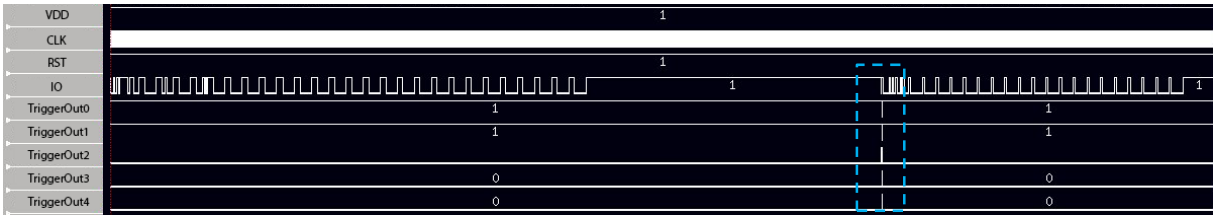


Abb. 39: Messergebnis von Beispiel 1 in der Übersicht

In dieser Darstellung ist die Übertragung eines Kommandos vom Reader zur Smartcard (linke Bildhälfte) und deren anschließende Antwort (rechte Bildhälfte) abgebildet. Innerhalb der Ausführungszeit des Kommandos auf der Smartcard, gibt diese ein Triggersignal auf der IO-Signalleitung aus. In diesem Beispiel erfolgt dessen Ausgabe kurz bevor die Smartcard mit dem Senden ihrer Antwort beginnt (siehe blau markierter Bereich). Abb. 40 zeigt eine Vergrößerung des blau umrandeten Bereichs.

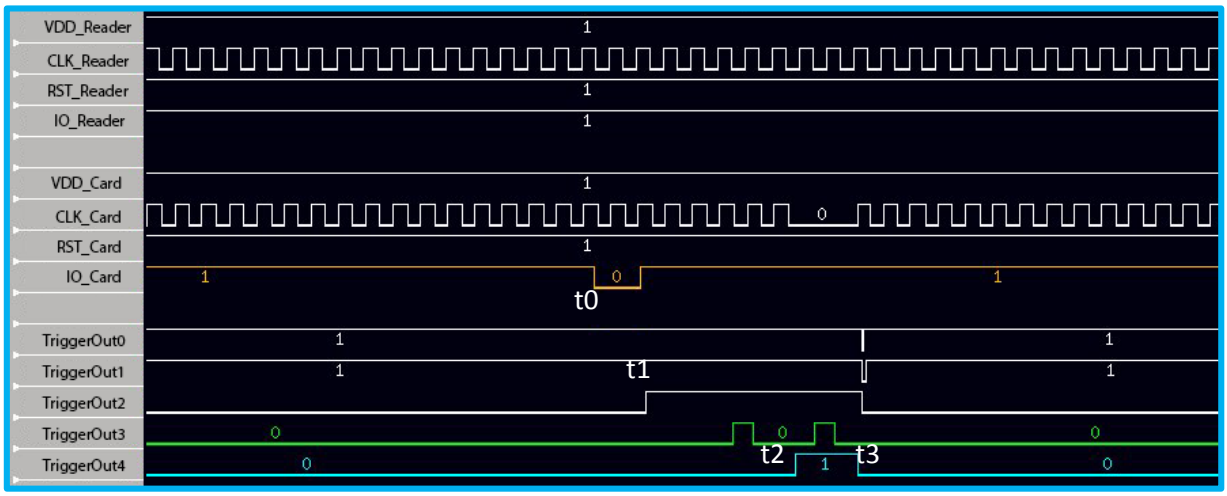


Abb. 40: Messergebnis von Beispiel 1 im Detail

Die Abbildung ist durch Abstände horizontal in drei Bereiche Unterteilt. Im oberen Bereich sind die vier Signalleitungen dargestellt, welche zwischen FPGA und Reader übertragen wurden. Der mittlere Bereich zeigt die vier zwischen FPGA und Smartcard übertragenen Signale und der untere Bereich fünf vom FPGA ausgegebene Triggersignale.

Wie in Abb. 39 zu erkennen ist, hat die Triggerausgabe von TC0 und TC1 bereits vor Beginn der Aufzeichnung begonnen, denn die Signalleitungen TriggerOut0 und TriggerOut1 geben schon zu Beginn der Datenübertragung ein High-Signal aus. Grund dafür ist, dass vor dem Senden des dargestellten Kommandos bereits andere Kommandos, unter anderem zur Aktivierung der Smartcard und zur Authentifikation gegenüber dieser, übertragen wurden. Infolge dessen wurde die Stromversorgung der Smartcard (VDD) seitens des Readers aktiviert und das Negativ-Resetsignal (RST) auf den High-Zustand gesetzt. Dies hat zur Folge, dass TC2 bereits während der Übertragung aller vorhergehenden Kommandos enabled war und auf einen Triggerpuls gewartet hat. Dieser wurde jedoch zum Zeitpunkt t0 erstmalig von der Smartcard ausgegeben.

Wie in Abb. 40 zu sehen ist, wird der auf der Signalleitung IO_Card von der Smartcard empfangene Triggerpuls herausgefiltert und nicht auf der Signalleitung IO_Reader zum Reader weitergeleitet.

Triggerzelle TC2 beginnt zum Zeitpunkt t1 mit der Ausgabe eines Triggersignals. Ab diesem Zeitpunkt zählt TC3 bis zur dritten steigenden Flanke auf der Signalleitung CLK_Reader und gibt dann ihrerseits ein Triggersignal aus. Dies wiederholt sich, solange sie enabled ist, mit jeder weiteren dritten Flanke.

Ebenfalls ab dem Zeitpunkt t1 beginnt TC4 bis zur sechsten fallenden Flanke auf CLK_Reader zu zählen. Nach deren Auftreten, startet TC4 die Triggerausgabe. Dies führt zum Umschalten des InToOutSwitches, welcher das Signal von CLK_Reader zu CLK_Card weiterleitet und damit zur Ausgabe eines Low-Signals auf CLK_Card für die Dauer der Triggerausgabe. Zum Zeitpunkt t3 endet diese und der InToOutSwitch schaltet in den Ausgangszustand zurück. Zusätzlich bewirkt die fallende Flanke des Triggersignals auf TriggerOut4 das Zurücksetzen der Triggerzellen TC0, TC1 und TC2 in deren Ausgangszustand. Wie in der Aufzeichnung zu erkennen ist, wird die Bedingung von TC0 unmittelbar danach wieder aktiviert, da VDD_Reader ein High-Signal überträgt und daraufhin auch die Bedingung von TC1, da RST_Reader ebenfalls im High-Zustand ist. TC2 ist hingegen nicht sofort wieder erfüllt, da diese nun den nächsten Low-Puls auf IO_Card wartet.

Anhand dieses Anwendungsbeispiels wurde die Funktion der implementierten Unterbedingungsarten demonstriert, welche in Abschnitt „3.2.1 Unterbedingungsarten“ gefordert wird. Darüber hinaus zeigte dieses Beispiel die in Abschnitt „3.2.3 Signalweiterleitung“ geforderte Möglichkeit zur triggergesteuerten Signalunterbrechung, sowie den Einsatz des Triggerfilters. Auf die im Abschnitt „3.3.4 Konfiguration über I²C“ geforderte Konfiguration von Triggerbedingungen zur Laufzeit, wird User Guide gesondert eingegangen (siehe Anhang A, S. 95).

### 7.2.2 Beispiel 2: Praxisrelevantes Anwendungsszenario

Für die Umsetzung des in Abschnitt „3.2.2 HoldOff- und Window-Offsets“ beschriebenen Anwendungsszenarios wurden 17 Triggerzellen eingesetzt. Abb. 41 zeigt deren Verschaltung.

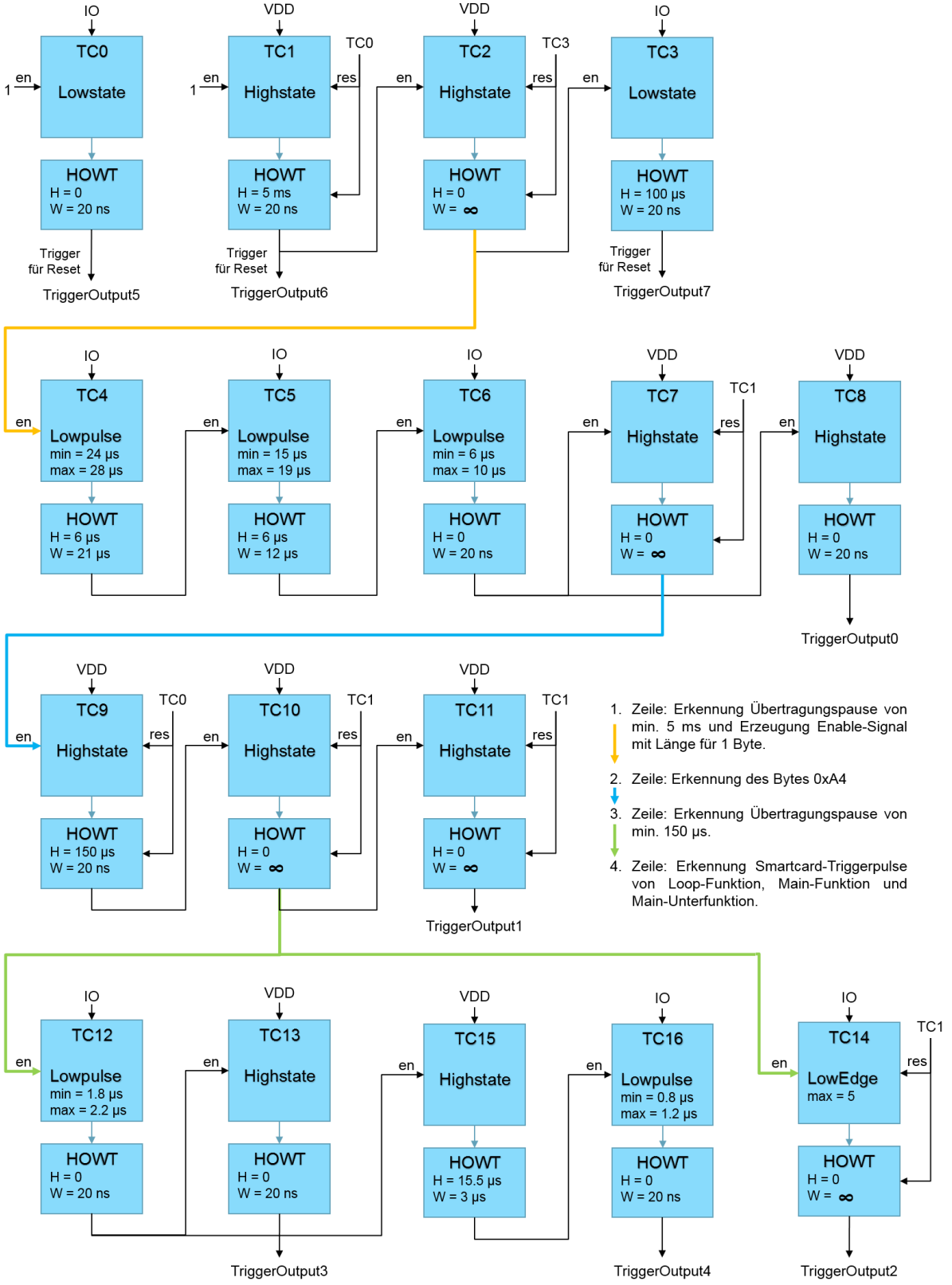


Abb. 41: Verschaltung der Triggerzellen in Beispiel 2

Die Triggerzellen TC0, TC1 und TC2 bilden den ersten Funktionsblock zur Erkennung einer Übertragungspause mit mindestens 5 ms Länge auf der Signalleitung IO:

- **TC0**: Gibt immer dann ein Triggersignal aus, wenn sich die IO-Signalleitung im Low-Zustand befindet. Das erzeugte Triggersignal wird auf TriggerOutput5 ausgegeben, um von anderen Triggerzellen (TC1 und TC9) als Reset-Signal verwendet werden zu können.
- **TC1**: Gibt ein Triggersignal mit einer Verzögerung von 5 ms aus, sofern bis dahin kein Reset durch den Trigger von TC0 erfolgt. Der Trigger von TC1 bedeutet, dass die IO-Signalleitung für min. 5 ms nicht im Low-Zustand war, was das Ende einer Übertragung bedeutet und die gesamte mehrstufige Triggerbedingung in den Ausgangszustand zurückversetzt. Um als Reset-Signal dienen zu können wird das Triggersignal auf TriggerOutput6 ausgegeben.
- **TC2**: Gibt unmittelbar nach dem enable durch TC1 ein Triggersignal unbegrenzter Länge aus, um die Datenbyteerkennung (von 0xA4) zu enablen (beginnend mit TC4).
- **TC3**: Wartet nach dem enable durch TC2 auf die Übertragung von Daten auf der IO-Signalleitung. Mit Beginn der Übertragung wird ein Trigger verzögert ausgegeben, um TC2 zu resetten, nachdem das erste Datenbyte übertragen wurde. So wird gewährleistet, dass nur Nachrichten akzeptiert werden, die das Byte 0xA4 als erstes Byte haben. Andere Nachrichten, die 0xA4 gar nicht oder an anderer Stelle übertragen werden ignoriert. Um als Reset-Signal dienen zu können wird das Triggersignal auf TriggerOutput7 ausgegeben.

Zu Beginn einer Datenübertragung sind die Triggerzellen TC4, TC5 und TC6 enabled, um das Datenbyte 0xA4 anhand seiner charakteristischen Pulslängen zu erkennen:

- **TC4**: Erkennt den erste Low-Puls der Übertragung von 0xA4. enabled TC5 verzögert und nur für die zeitliche Länge des zweiten Pulses von 0xA4.
- **TC5**: Erkennt den zweiten Low-Puls von 0xA4.
- **TC6**: Erkennt den dritten Low-Puls von 0xA4.
- **TC7**: Speichert den Zustand, dass 0xA4 am Anfang der Nachricht erkannt wurde und gibt einen Trigger unbegrenzter Länge zum enablen von TC9 aus. Die Ausgabe des Triggers wird von TC1 beendet, sobald diese die nächste Übertragungspause von mehr als 5 ms erkannt hat.
- **TC8**: Dient zur Ausgabe eines Triggers von benutzerdefinierbarer Länge und Verzögerung, welcher auf TriggerOutput0 ausgegeben wird. Dieser zeigt an, dass eine Nachricht, welche mit 0xA4 beginnt erkannt wurde.

Ab der Erkennung des Bytes 0xA4 am Anfang der Datenübertragung, sind die Triggerzellen TC9 und TC10 enabled, um das Ende der Nachricht zu erkennen:

- **TC9**: Gibt einen Trigger um 150 µs verzögert aus, wenn bis dahin kein Reset durch TC0 erfolgte. So wird das Ende des Kommandos abgewartet. 150 µs liegen über der Länge aller High-Pulse der Datenübertragung und unter der Länge des zeitlichen Abstands zwischen der Übertragung des Kommandos vom Reader zur Smartcard und deren Antwort.
- **TC10**: Speichert den Zustand, dass das Ende der Kommandoübertragung erreicht wurde. Der ausgegebene Trigger unbegrenzter Länge enabled TC12 und TC14. Die Ausgabe wird durch den Reset von TC1 beendet, sobald diese eine Übertragungspause von 5 ms festgestellt hat.
- **TC11**: Dient zur Ausgabe eines Triggers von benutzerdefinierbarer Länge und Verzögerung, welcher auf TriggerOutput1 ausgegeben wird. Dieser zeigt an, dass das Ende der Kommandonachricht vom Reader erkannt wurde.

Nach dem Ende der Datenübertragung werden die Triggerzellen TC12 und TC14 enabled, um auf den zu Beginn der Main-Funktion der Smartcard ausgegebenen Triggerpuls zu warten, sowie auf die noch davor in der Loop-Funktion erzeugten Triggerpulse.

- **TC12**: Wartet auf einen Low-Puls mit 2 µs Länge, welcher zu Beginn der Main-Funktion von der Smartcard ausgegeben wird.
- **TC13**: Dient zur Ausgabe eines Triggers von benutzerdefinierbarer Länge und Verzögerung, welcher auf TriggerOutput3 ausgegeben wird. Dieser zeigt an, der 2 µs lange Puls der Main-Funktion erkannt wurde.
- **TC14**: Wartet nebenläufig zu TC12 darauf, dass die fünfte fallende Flanke von der Smartcard ausgegeben wurde. So wird der Beginn des fünften loop-Pulses festgestellt, den die Smartcard ausgibt, bevor sie ihre Antwort and den Reader überträgt. Diese Bedingung wird erwartungsgemäß vor TC12 erfüllt und führt zur Ausgabe eines Triggers unbegrenzter Länge auf TriggerOutput2. Die Triggerausgabe endet mit der nächsten Übertragungspause von 5 ms, die von TC1 erkannt wird.
- **TC15**: Erzeugt nach Auftreten des Triggerpulses zu Beginn der Main-Funktion der Smartcard ein Zeitfenster, in dem die Ausgabe eines Triggers durch eine Unterfunktion der Smartcard erwartet wird. Dieser tritt nach einer HoldOff-Zeitspanne von 15,5 µs innerhalb eines Zeitfensters von 3 µs, weshalb ein entsprechendes Enable-Signal für TC16 erzeugt wird.
- **TC16**: Wartet auf einen Low-Puls mit 1 µs Länge um nach dessen Auftreten einen Trigger von benutzerdefinierter Länge und Verzögerung auf TriggerOutput4 auszugeben.

In diesem Beispiel werden einige Triggerzellen lediglich dazu eingesetzt ein bereits erzeugtes Triggersignal verzögert weiterzugeben. Zu erkennen sind diese daran, dass sie die Signalleitung VDD beobachten, welche durchgehend im High-Zustand ist, und auf den Modus Highstate eingestellt sind. Die Bedingung (anliegendes High-Signal) einer auf Highstate eingestellten Triggerzelle ist unmittelbar erfüllt, sobald sie enabled wird, wobei der HOWTimer den Zeitpunkt und Länge der Triggerausgabe bestimmt. Eine derartige Verwendung von Triggerzellen macht Sinn, wenn ein Triggersignal sowohl zum Enablen weiterer Triggerzellen, als auch zur Ausgabe an den Benutzer dienen soll. Typischerweise sind die Anforderungen, also Verzögerung und Dauer, an das benötigte Enable-Signal andere, als an das vom Benutzer gewünschte Triggersignal. Zusätzlich kann es erforderlich ein Reset-Signal mit wiederum anderer Verzögerung und Dauer zu erzeugen. Auf diese Weise kommen auf eine Trigger erzeugende Triggerzelle bis zu drei Weitere, von denen lediglich der HOWTimer benötigt wird. In diesem Beispiel sind alle drei Varianten anzutreffen:
Beispielweise erzeugt TC1 einen maximal kurzen Triggerpuls, der TC2 enabled. TC2 hat die Aufgabe eines Zustandsspeichers. Dass TC1 einen Triggerpuls erzeugt hat, wird so für eine unbestimmte Dauer gespeichert und dient als Enable-Signal für weitere Triggerbedingungen. TC3 verwendet den gleichen Triggerpuls von TC1 zur Erzeugung eines Reset-Signals.
Als Beispiel für die Erzeugung benutzerspezifischer Triggersignale seien TC12 und TC13 genannt: Nachdem TC12 einen Puls erkannt hat, gibt diese ein maximal kurzes, unverzögertes Triggersignal aus. TC13 dient dazu eine Verzögerung und eine Verlängerung von diesem vorzunehmen. Da es separate Triggerzellen zur Triggererzeugung und zur Triggerausgabe gibt, können die Anforderungen der in der Schaltung folgenden Triggerzellen unabhängig von denen des Benutzers definiert werden. In diesem Beispiel nimmt TC13 keine Verzögerung oder Verlängerung an dem von TC12 erzeugten Triggersignal vor, möglich wäre dies jedoch und hätte keinen Einfluss auf den Rest der Schaltung. Sollte jedoch TC12 zur Ausgabe eines benutzerdefinierten Triggersignals und gleichzeitig zum Enablen von TC15 verwendet werden, müsste TC15 an die benutzerdefinierte Triggerverzögerung angepasst werden. Diese Art der Verschaltung kann dennoch sinnvoll sein, wenn eine umfangreiche Schaltung es, aufgrund von Hardwareknappheit, notwendig macht TC13 für andere Zwecke zu verwenden.
Abb. 42 zeigt eine Übersicht der gemessenen Signale.

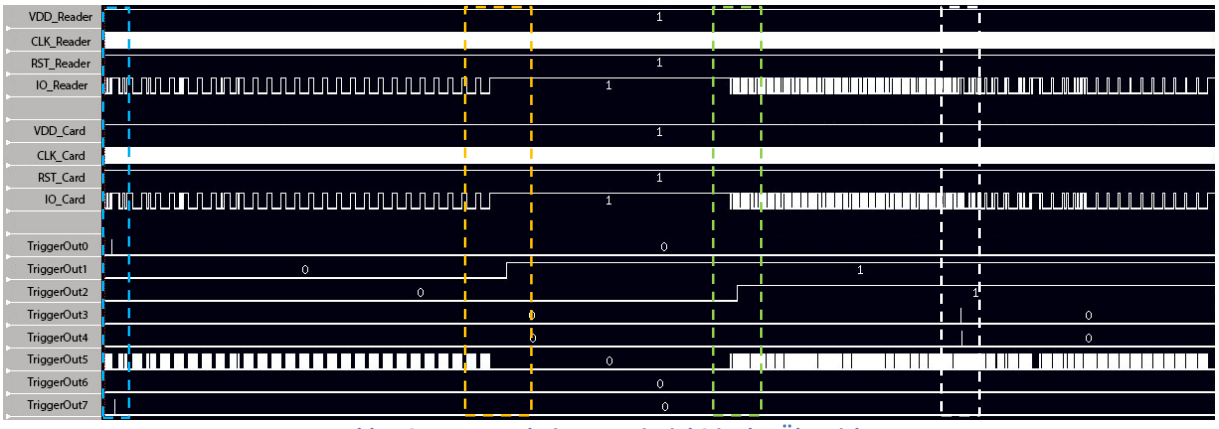


Abb. 42: Messergebnis von Beispiel 2 in der Übersicht

In dieser Darstellung ist die Übertragung eines Kommandos vom Reader zur Smartcard (linke Bildhälfte) und deren anschließende Antwort (rechte Bildhälfte) abgebildet. Das Kommando beginnt mit dem Datenbyte 0xA4 (blau markierter Bereich). Nach dem Ende des Kommandos (orange markierter Bereich) folgt eine Übertragungspause, in der die Smartcard ihre Antwort berechnet. Teil dieser Berechnungen ist eine Loop-Funktion, welche die Ausgabe von Triggerpulsen auslöst (grün markierter Bereich). Danach werden ein bestimmter Teil der Main-Funktion der Smartcard und anschließend eine Unterfunktion erreicht, in denen weitere Triggerpulse von der Smartcard auf der Signalleitung IO_Card ausgegeben werden (weiß markierter Bereich). Die Inhalte der vier markierten Bereiche werden im Folgenden anhand separater Abbildungen im Detail erläutert.
Abb. 43 zeigt eine Vergrößerung des blau umrandeten Bereichs.

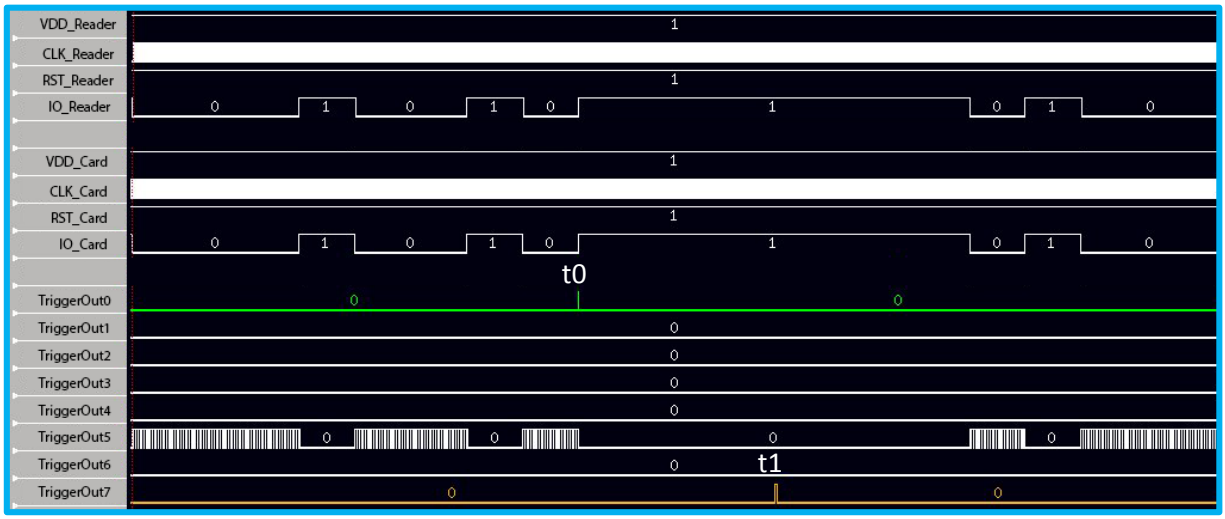


Abb. 43: Erkennung des ersten Kommandobytes im Detail

In der linken Bildhälfte ist das erste Byte des vom Reader kommenden Kommandos zu erkennen. Die genaue Pulsfolge und deren Länge sind vom Übertragungsprotokoll und der Übertragungsgeschwindigkeit abhängig und sind dem Smartcard-Tester entweder bekannt oder müssen durch Messungen ermittelt werden. Für die Erkennung der Pulsfolge des Datenbytes 0xA4 ist es in diesem Beispiel notwendig drei Low-Pulse zu erkennen, auf deren Übertragung das bestätigende Triggersignal auf TriggerOut0 zum Zeitpunkt t0 folgt. Zum Zeitpunkt t1 folgt das als Resetsignal dienende Triggersignal auf TriggerOut7. Dieses zeigt das Ende des Zeitraums an, in dem eine Erkennung des Datenbytes 0xA4 möglich ist.
Mit dem Triggersignal auf TriggerOut0 beginnt das Warten auf das Ende der Kommandoübertragung, das in Abb. 44 zu sehen ist.

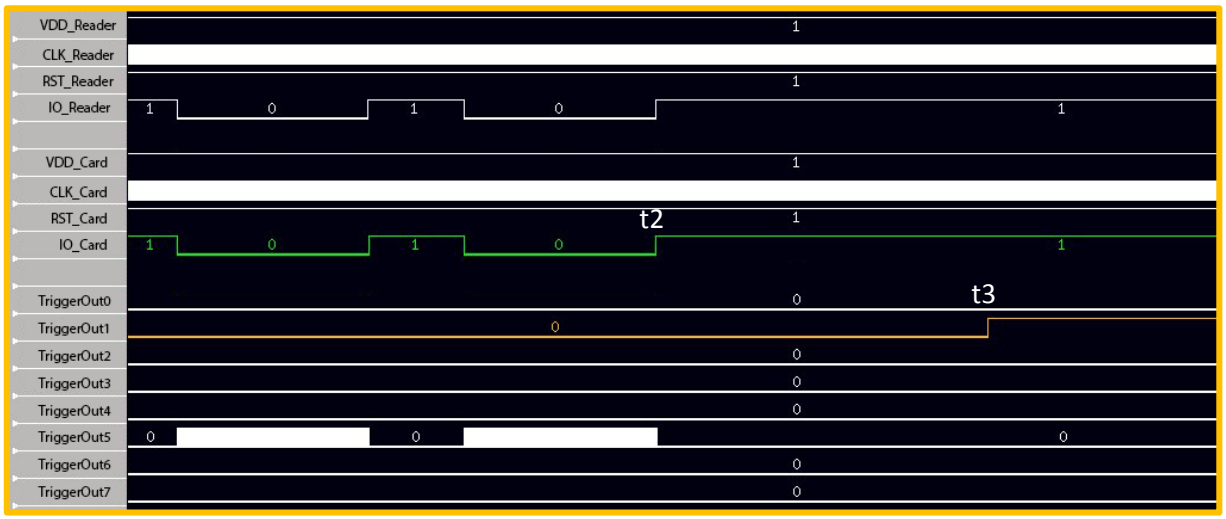


Abb. 44: Erkennung des Endes der Kommandoübertragung

Das Kommandoende ist erreicht, wenn eine bestimmte Zeitspanne lang nichts mehr übertragen wurde, sich die Signalleitung IO also durchgängig im High-Zustand befand. Deren Länge ist wiederum vom Kommunikationsprotokoll und der Übertragungsgeschwindigkeit abhängig. In diesem Beispiel haben Messungen ergeben, dass alle High-Pulse innerhalb des Kommandos kürzer als 150 µs sind. Deshalb wird auf eine Übertragungspause dieser Länge gewartet.
Zum Zeitpunkt t2 in Abb. 44 ist die Kommandoübertragung beendet. Die Ausgabe des bestätigenden Triggersignals erfolgt zum Zeitpunkt t3, 150 µs später, auf der Triggersignalleitung TriggerOut1.
Ab dem Zeitpunkt des Triggers wird auf den Beginn des fünften Triggerpulses gewartet, der von der Loop-Funktion der Smartcard, auf der Signalleitung IO_Card, erzeugt wird. Abb. 45 zeigt diesen im Detail.

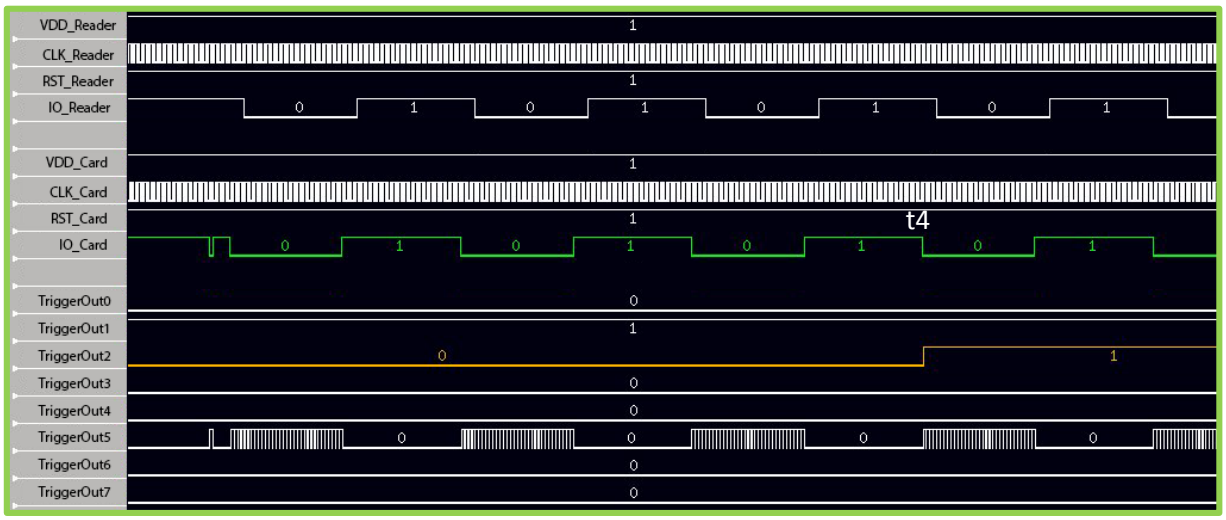


Abb. 45: Erkennung der fünften fallenden Triggerflanke

Auf der Signalleitung IO_Card in Abb. 45 sind die von der Smartcard, während der Kommandobearbeitung, erzeugten Triggerpulse zu erkennen. Der erste ist dabei wesentlich kürzer, als die darauffolgenden und zeigt den Beginn der Loop-Funktion an. Um den Beginn des fünften Triggerpulses anzuzeigen, wird auf die fünfte fallende Flanke auf IO_Card gewartet. Diese tritt zum Zeitpunkt t4 auf, was die Ausgabe eines Triggersignals auf TriggerOut2 bewirkt.

Beim Vergleich der von der Smartcard empfangenen Signale auf IO_Card mit den an den Reader weitergeleiteten Signalen auf IO_Reader ist erkennbar, dass der erste, kurze Triggerpuls herausgefiltert wird, die längeren jedoch nicht. Durch eine Umkonfiguration des Pulsfilters ließen sich auch die längeren Triggerpulse aus der Übertragung filtern, was in diesem Beispiel jedoch nicht gefordert war. Des Weiteren sei angemerkt, das die durch den Pulsfilter bedingte Verzögerung der Weiterleitung des IO-Signals in dieser Abbildung gut erkennbar ist.
Nach Abarbeitung der Loop-Funktion werden weitere Funktionen aufgerufen, welche ebenfalls Triggerpulse erzeugen, wie in Abb. 46 gezeigt wird.

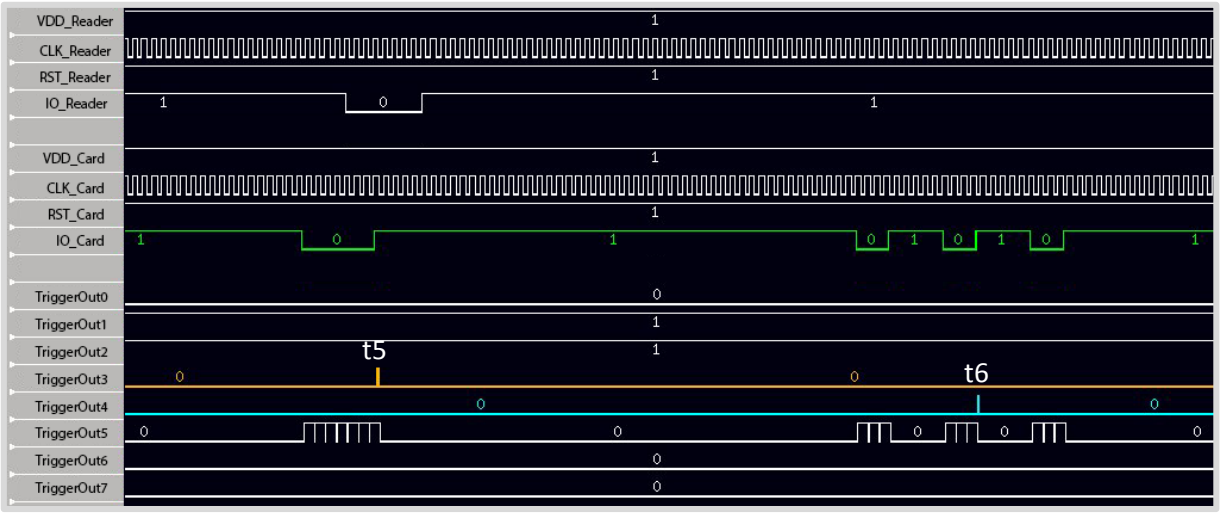


Abb. 46: Erkennung von Triggerpulsen aus der Main-Funktion und Unterfunktionen

Ab dem Ende der Kommandoübertragung (Zeitpunkt t3 in Abb. 44) wird auf die Ausgabe des Triggerpulses der Main-Funktion gewartet. Dieser kann durch seine individuelle Länge von 2 µs von allen anderen, durch die Smartcard erzeugten, Triggerpulsen unterschieden werden. Das auf TriggerOut3 zum Zeitpunkt t5 ausgegebene Triggersignal bestätigt dessen Erkennung. Auf den von der Main-Funktion ausgegebenen Triggerpuls folgen drei von einer Unterfunktion ausgegebene Triggerpulse, welche in der rechten Bildhälfte zu sehen sind. Die Abstände aller vier Triggerpulse zueinander sind als konstant anzusehen.
Ab dem Zeitpunkt t5 wird eine definierte Zeitspanne, bis kurz vor Beginn des mittleren Triggerpulses, abgewartet, bevor die Pulserkennung beginnt. Diese endet wiederum nach einer definierten Zeitspanne, kurz nach dem erwarteten Ende des mittleren Triggerpulses. Auf diese Weise ermöglichen es HoldOff/Window-Timer nah beineinander liegende Pulse zu unterscheiden. Die Erkennung des mittleren Triggerpulses wird zum Zeitpunkt t6 über ein Triggersignal auf TriggerOut4 bestätigt.
Im Anschluss gibt die Smartcard ihre Antwort an den Reader aus, währenddessen keine weiteren Triggersignale erzeugt werden, wie in Abb. 42 (siehe rechts vom weiß markierten Bereich auf TriggerOut0 bis 4) zu sehen ist. Lediglich die Triggersignale, deren Ausgabelänge als unbegrenzt definiert wurde (siehe TriggerOut1 und 2) werden bis zum Reset nach 5 ms Übertragungspause aufrechterhalten.
Die Funktion des, nach 5 ms Übertragungspause, resetauslösenden Triggersignals stellt Abb. 47 dar.

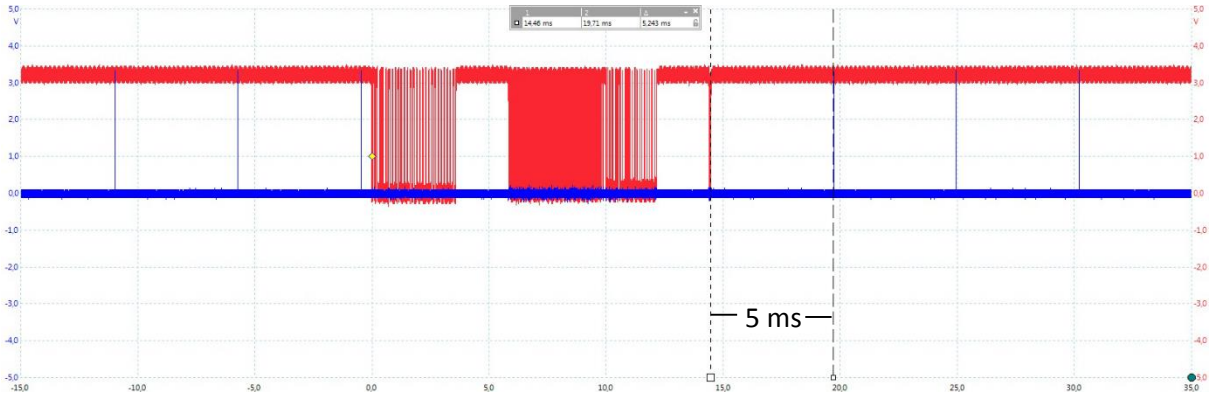


Abb. 47: Resetsignal nach 5 ms Übertragungspause

Abb. 47 zeigt die gleiche Datenübertragung wie Abb. 42, jedoch in verkleinerter Darstellung. Die Signalleitung IO_Card ist in rot eingezeichnet, während die Triggersignalleitung TriggerOut6 blau dargestellt ist. In der Abbildung ist erkennbar, dass vor der Übertragung des Kommandos durch den Reader und nach der Antwort durch die Smartcard Triggerpulse im Abstand von 5 ms erzeugt werden. Diese setzen die Triggerzellenschaltung in den Anfangszustand zurück. Der Wert von 5 ms wurde gewählt, da innerhalb eines Dialogs zwischen Smartcard und Reader mit keiner Übertragungspause dieser Länge zu rechnen ist.
Anhand dieses Anwendungsbeispiels wurde die Funktion der in Abschnitt „3.2.2 HoldOff- und Window-Offsets“ geforderte Möglichkeit HoldOff und Window-Zeitspannen für eine Unterbedingung definieren zu können in einem praxisrelevanten Szenario demonstriert.

## 7.3 Weitere Testergebnisse

Die folgenden Abschnitte erläutern den möglichen Umfang des Systems und dessen Leistungsfähigkeit in Bezug auf Reaktionsgeschwindigkeiten und Jittereffekte (d.h. Schwankungen zeitlicher Abstände).

### Erreichter Schaltungsumfang

Die Anzahl synthetisierbarer Triggerzellen, bei gegebener Hardware, hängt maßgeblich von deren Ausstattung ab. Die Hardwareauslastung bei der für Beispiel 2 gewählten Dimensionierung lag in den Tests bei 80 – 90 %. Somit ist es nicht möglich wesentlich mehr als die verwendeten 24 Triggerzellen zu synthetisieren. Testweise gelang es unter Reduzierung von deren Umfang, auf weniger als 8 Bit Zählerbitbreiten, bis zu 32 Triggerzellen zu erstellen.
Je nach Anwendungzweck muss ein Kompromiss zwischen der Anzahl von Triggerzellen und Pulsfiltern und deren Zählerbitbreiten eingegangen werden. Einen Ausgleich geringer Zählerbitbreiten kann ein Tickgenerator bewirken, welcher als, von mehreren Komponenten gemeinsam benutzter, Zähler fungiert. Dieser reduziert zwar nicht die Reaktionsgeschwindigkeit der Triggerzellen und Pulsfilter, aber deren Genauigkeit bei der Bestimmung von Zeitspannen.
Als praktikabel, da ebenfalls ressourcenschonend, hat sich der Einsatz unterschiedlich dimensionierter Triggerzellen herausgestellt. Denn für einen Großteil der späteren Anwendungszwecke gelten, gemäß der Smartcard-Tester, ähnliche Anforderungen, welche jeweils nur wenige große Zähler benötigen. Auf diese Weise sind die benötigten Kapazitäten, an den wenigen Stellen, an denen sie gebraucht werden, vorhanden. Es muss bei der Konfiguration des Triggergenerators lediglich darauf geachtet werden die richtige Triggerzelle für jeden Zweck auszuwählen.

### Verzögerungen

Die Reaktionsgeschwindigkeit des Triggergenerators wird durch die anwendungsspezifischen Rechenzeiten limitiert. Verzögerungen gibt es einerseits bei der Weiterleitung von Signalen. Andererseits werden auch Triggersignale verzögert nach Erfüllung der Triggerbedingung ausgegeben. Die folgenden Abschnitte zeigen, dass die in „3.3 Spezifische funktionale Anforderungen“ genannten Anforderungen an die Rechenzeiten von der Schaltung erfüllt werden.

Die zwischen Smartcard und Reader verlaufenden Signalleitungen werden bei der Weiterleitung durch den FPGA verzögert. Messungen ergeben, dass die Länge der Verzögerung, abhängig von der Art der Weiterleitung (direkt, über InToOutSwitch oder durch BiDirIO) stets unter der Länge eines CPU-Taktes von 20 ns liegt. Da Verzögerungen in dieser Größenordnung nicht relevant für das Funktionieren der Kommunikation zwischen Smartcard und Reader sind, wird nicht weiter darauf eingegangen.
Die Ausnahme bildet lediglich die Pulsfilterung. Denn diese erzeugt eine Verzögerung in Länge der Filterbreite ± 10 ns. Ausgehend von einer Triggerpulslänge von beispielsweise 1 - 2 µs wird das gefilterte IO-Signal somit um eben diese Zeitspanne verzögert, je nach Einstellung des Pulsfilters. Abhängig vom Kommunikationsprotokoll und der Übertragungsgeschwindigkeit kann dies zu Problemen führen. Bei Verwendung der Protokolle T=0 oder T=1 ist dies nicht der Fall, da es sich um asynchrone Übertragungsprotokolle handelt, die den Kommunikationstakt lediglich zur Angleichung der Datenrate verwenden, ähnlich wie das UART-Protokoll.
Relevant für das Funktionieren der Kommunikation ist hingegen, dass Datenpulsbreiten nicht verändert werden. Messungen dazu ergaben eine Verzerrung von ± 10 ns, was, gemäß der Smartcard-Tester, eine tolerierbare Größe bei Datenraten von bis zu 300 kBit/s ist.

Die von einer Triggerzelle benötigte Zeitspanne zur Berechnung eines Triggersignals hängt von deren Modus. Abb. 48 zeigt die gemessenen Verzögerungen.

| TriggerCell-Mode | Verzögerung |
|---|---|
| High-/LowState | 30 ns ± 10 ns |
| High-/LowEdgeCount | 80 ns ± 10 ns |
| High-/LowPulseDetect | 50 ns ± 10 ns |

Abb. 48: Verzögerung der Triggerausgabe

Die dargestellten Werte entsprechen jeweils der Verzögerung der Triggerausgabe bei einem HoldOff-Wert von 0. Bei Verwendung größerer HoldOff-Werte, erhöht sich die Verzögerung um jeweils eine CPU-Taktperiode (20 ns bei 50MHz Systemtakt). Die Ungenauigkeit beträgt stets ± 10 ns.
Die Ausgabelänge eines Triggersignals beträgt mindestens eine CPU-Taktperiode, bei einem Window-Wert von 1, ebenfalls mit einer Ungenauigkeit von ± 10 ns.
Die Messung der Triggerausgabeverzögerung und deren Abweichung für den Modus HighEdgeCount ist in Abb. 49 dargestellt.

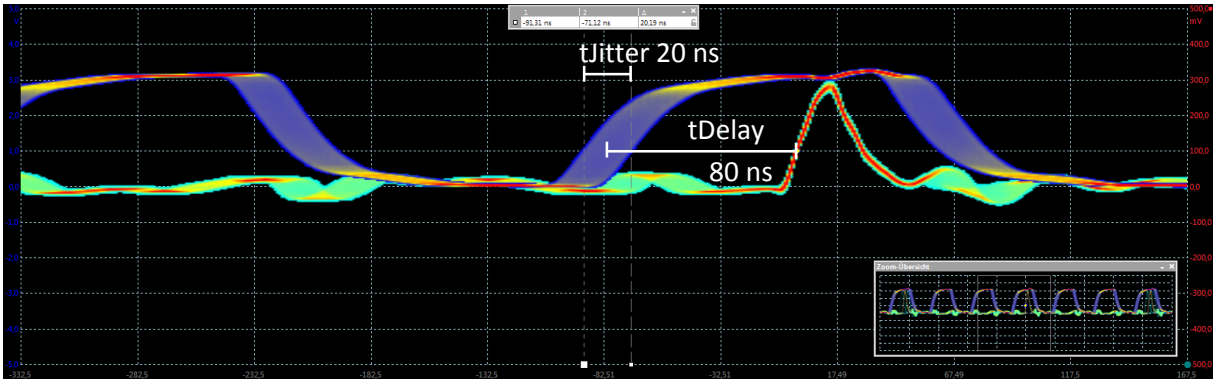


Abb. 49:Triggerverzögerung im Modus HighEdgeDetect (HoldOff=0, Window=1)

In Abb. 49 ist eine Aufzeichnung vieler Messungen zu sehen, deren Ergebniskurven übereinander gelegt wurden (im Persistence Acquisition Mode des Oszilloskops). Über die Breite und Farben der Kurven ist erkennbar welche Verläufe bei den Messungen häufiger aufgetreten sind (rötlich dargestellt) und welche seltener (bläulich dargestellt).
Gemessen wurde die zeitliche Verschiebung eines Taktsignals mit 3,5 MHz von einem nach jeder dritten steigenden Flanke erzeugten Triggersignal aus. Es wurde keine HoldOff-Ausgabeverzögerung verwendet und eine Ausgabelänge von einem CPU-Takt (entspricht 20 ns bei 50 MHZ CPU-Takt und einem Window-Wert von 1). Abb. 49 bestätigt diese Einstellungen.
Die Zeitspanne tJitter von 20 ns entspricht der Größe des zeitlichen Fensters innerhalb dessen der Triggerpuls nach dem Auftreten jeder dritten Flanke ausgegeben wird. Von der gemittelten Ungenauigkeit ausgehend ergibt sich eine Verzögerung in Länge der Zeitspanne tDelay, die mit einer Genauigkeit von ± 10 ns erreicht wird.
Die Messungen der übrigen Modi erfolgten auf vergleichbare Weise und lieferten die in Abb. 48 zusammengefassten Ergebnisse. Somit sind die in Abschnitt „3.3.3 Mehrstufige Triggerbedingungen und Unterbedingungen“ genannten Anforderungen erfüllt.

## Erhöhte Ungenauigkeit bei langsam steigender Signalflanke

Die Erzeugung der für die Triggerberechnung verwendeten Signale, deren Messwerte in Abb. 48 zusammengefasst sind, wurde ein Funktionsgenerator verwendet. Bei Verwendung der Kommunikationssignale zwischen Smartcard und Reader zur Triggerberechnung konnte eine erhöhte Ungenauigkeit des High-/LowPulseDetect-Modus gemessen werden. Diese resultiert aus der Beschaffenheit des erkannten Pulses, welchen Abb. 50 zeigt.

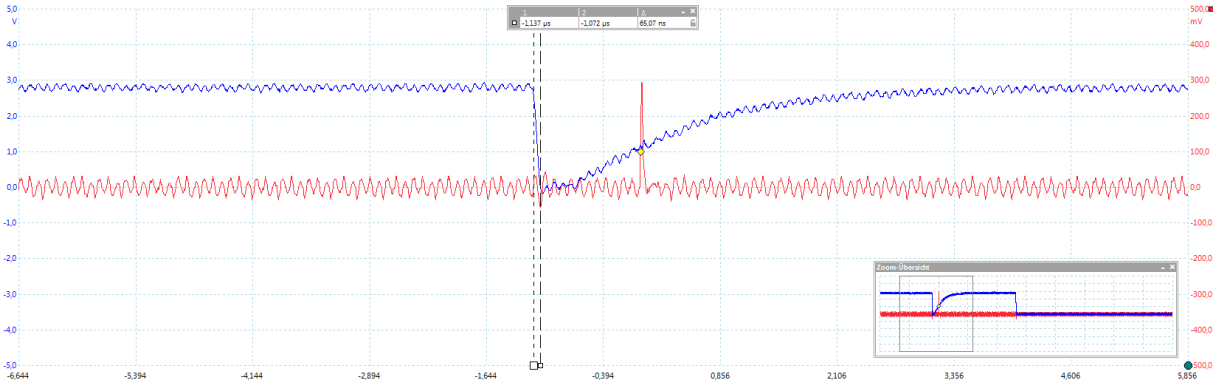

Abb. 50: Von der Smartcard erzeugter Triggerpuls

Der in Abb. 50 gezeigte Triggerpuls wurde von der Smartcard erzeugt. Dieser hat eine relativ steile fallende Flanke an dessen Beginn, jedoch nur eine schwach ansteigende Flanke am Ende. Zudem ist das Signal sichtbaren Spannungsschwankungen unterworfen. Beides führt in Kombination dazu, dass um die Spannungsschwellwert herum, ab dem der FPGA ein Signal als High interpretiert, ein Prell-Effekt zu beobachten ist. Der Zeitpunkt der Erkennung des Pulsendes schwankt dadurch um 20 ns, d.h. einen CPU-Takt. Somit ergibt sich bei Benutzung von Smartcard-Triggerpulsen die in Abb. 51 gezeigte Genauigkeit.

| TriggerCell-Mode | Verzögerung |
|---|---|
| High-/LowPulseDetect | 50 ns ± 20 ns |

Abb. 51: Verzögerung der Triggerausgabe mit Smartcard-Trigger

Zuzüglich zur vorhandenen Ungenauigkeit kommt eine weitere Schwankung um ± 10 ns. Nach Aussage der Smartcard-Tester ist die Smartcard hardwarebedingt nicht dazu in der Lage eine steilere Flanke am Ende des Triggerpulses zu erzeugen.

## 7.4 Zusammenfassung der Testergebnisse

Im diesem Kapitel wurde gezeigt, dass die Anforderungen mit den gegebenen Mitteln umgesetzt werden konnte und gleichzeitig noch Leistungsreserven für zukünftige Erweiterungen übrig lassen. Die implementierte Schaltung verknüpfbarer Triggerzellen ermöglichte es die im Abschnitt „3.2 Allgemeine Anforderungen“ geforderten Anwendungsbeispiele zur Laufzeit zu konfigurieren. Die Reaktionszeiten und Genauigkeiten der so entstandenen Schaltung zur Triggerberechnung übertrafen die Anforderungen größtenteils deutlich.

Die Möglichkeiten zur Verschaltung von Triggerzellen bieten dem Benutzer ein hohes Maß an Flexibilität bei der Definition von Triggerbedingungen. Infolgedessen fallen die entstehenden Triggerschaltungen, für umfangreiche Anwendungsszenarien, relativ komplex aus. Deren Entwicklung kann somit recht zeitaufwändig werden.

# Zusammenfassung und Ausblick

Diese Bachelorarbeit beschäftigt sich mit einer Thematik aus der Abteilung für Smartcard-Security der Firma NXP Semiconductors. Dort werden Sicherheitsmechanismen für Smartcards entwickelt und auf deren Wirksamkeit hin getestet. Man unterscheidet zwischen Seitenkanalanalysen und Fehlerinjektionsattacken als Angriffsarten auf Smartcards, welche im Rahmen der Tests durchgeführt werden. Die Anzahl der möglichen Sicherheitsüberprüfungen ist grundsätzlich unbegrenzt. Das Bestreben ist, in einem festgelegten Zeitraum eine möglichst hohe Testabdeckung zu erreichen, um die Zuverlässigkeit der Smartcards zu maximieren.
Das bislang verwendete Testsystem erfordert es, jede Testreihe von Hand zu konfigurieren, was zusätzlich Entwicklungszeit kostet. Grundsätzlich sind manuelle Eingaben unzuverlässig, da Fehler passieren können. Die Reproduzierbarkeit ist durch mögliche Eingabefehler ebenfalls geringer und erfordert zudem eine Dokumentation. Es ist nicht möglich eine Testreihe in der Hinsicht dynamisch zu gestalten, dass auf vorangegangene Testergebnisse reagiert wird, indem beispielsweise vertiefende Testangriffe an einer erfolgversprechenden Stelle durchgeführt werden. Eine derartige dynamische Steuerung bewirkt jedoch eine erhebliche Effizienzsteigerung der Tests. Zur Steuerung der Seitenkanalanalysen und Fehlerinjektionsattacken wurden bislang Oszilloskope eingesetzt. Diese können Signalleitungen hinsichtlich einer mehrstufigen Triggerbedingung, bestehend aus bis zu vier Unterbedingungen, auswerten und ein Triggersignal ausgeben. Manipulationen von Signalleitungen sind gar nicht möglich. Die Beschränkungen auf wenige Eingänge, einfachste Triggerbedingungen und nur einen Triggerausgang machten umfangreiche Tests aufwändig oder gar undurchführbar.
Im Rahmen dieser Bachelorarbeit wurde ein Triggergenerator entwickelt, um eine Automatisierung des Testsystems zu ermöglichen. Er dient zur Synchronisation der Seitenkanalanalysen und Fehlerinjektionsattacken an die Abläufe auf einer Smartcard. Über 16 Eingänge empfängt der Triggergenerator Signale von Smartcard und Reader und generiert daraus, über mehrstufige Triggerbedingungen, auf 16 Ausgängen Triggersignale. Zudem können Eingangssignale in manipulierter Form, unterbrochen oder gefiltert, weiterverwendet werden. Die Konfiguration erfolgt zur Laufzeit, über eine I²C-Schnittstelle. Als Hardwareplattform wurde ein FPGA gewählt, da er nebenläufige Berechnungen in Echtzeit erlaubt und seine Reaktionszeiten die eines Mikrocontrollers übertreffen. Zur Realisierung mehrstufiger Triggerbedingungen wurde eine elementare Triggerkomponente, die Triggerzelle, implementiert. Von dieser existieren bis zu 32 Instanzen, welche beliebig aneinandergereiht werden können, um eine oder mehrere mehrstufige Triggerbedingungen zu bilden. Die dadurch erreichte hohe Flexibilität ermöglicht die Umsetzung komplexer Triggerbedingungen. Die Entwicklung eines geeigneten Schaltnetzes aus Triggerzellen ist Aufgabe des Anwenders. Der Triggergenerator ist ein Werkzeug, dass an zukünftige Anforderungen angepasst werden muss. Der weitreichende Einsatz generischer Attribute erlaubt es dazu, auch Anwendern ohne tiefgehende VHDL-Kenntnisse, Komponenten zu dimensionieren und so den Verbrauch von Hardwareressourcen zu steuern. Der Triggergenerator stellt eine wertvolle Ergänzung zu den hochpreisigen, genauer arbeitenden, Oszilloskopen dar. Mit seiner Hilfe kann der Versuchsaufbau verkleinert werden, bei gleichzeitig steigendem Funktionsumfang. Als Teil eines automatisierten Testsystems ermöglicht er eine höhere Testabdeckung, denn die Testangriffe können schneller hintereinander ausgeführt werden. Zudem ermöglicht er die Umsetzung wesentlich komplexerer Testfälle, als bisher möglich waren. Der modulare Ansatz und die Flexibilität und komfortable Anpassbarkeit der Schaltung an neue Testfälle wurde von den Mitarbeitern von NXP Semicunductors begrüßt.

Zur komfortablen Integration des Triggergenerators in das bestehende Testsystem soll eine Adapterplatine entworfen werden. Kurzfristig bietet diese eine Schnittstelle zur Anbindung des FPGA-Boards. Langfristig wird der FPGA direkt in das Reader-Board integriert.
Um auch zukünftigen Anwendungsszenarien zu entsprechen, ist es möglich die Funktionen des Triggergenerators in bestimmten Bereichen auszubauen. Beispielsweise kann eine Vergrößerung der Anzahl verfügbarer Triggerbedingungen erforderlich sein.
Durch den kontinuierlichen Funktionszuwachs während der Entwicklung haben die Zuständigkeiten der Komponenten schrittweise zugenommen und sich teilweise verändert. Einige Funktionsblöcke und Benennungen können, zugunsten der Lesbarkeit, angepasst werden. Da sich daraus keine funktionalen Unterschiede ergeben, wurde auf eine nachträgliche Optimierung im Rahmen der Bachelorarbeit verzichtet.
Eine grafische Oberfläche zur Darstellung und Anpassung der Triggerzellen hilft die Übersichtlichkeit zu steigern und Änderungen schneller vorzunehmen. Zum Beispiel kann die Verzögerung einer Triggerausgabe dann mithilfe eines Schiebereglers eingestellt werden. Auf diese Weise werden dem Benutzer die Ermittlung von Speicheradressen des Konfigurationsspeichers und Parameterwerten abgenommen.
Ergänzend dazu sollen Informationen zur Dimensionierung des Triggergenerators über die I²C-Schnittstelle ausgelesen werden können. Mit deren Hilfe können die Einstellmöglichkeiten auf der grafischen Oberfläche an den jeweiligen Funktionsumfang des Triggergenerators angepasst werden.
Zukünftige Tests schnellerer Smartcard-Mikrocontroller erfordern kürzere Rechen- und Reaktionszeiten seitens des Triggergenerators. Der Umstieg auf einen FPGA, der höhere Systemtakte unterstützt, kann zusätzliche Reduzierungen der Signallaufzeiten in der Schaltung des Triggergenerators erfordern. Zum Beispiel durch Aufteilung langer Pfade kombinatorischer Logik durch Abtaktung mit FlipFlops.

# Literaturverzeichnis

**Rüdinger 2009**
RÜDINGER, Jens: *Auswirkungen von Seitenkanalangriffen auf das Design kryptographischer Algorithmen*. Dresden : Jörg Vogt Verlag, 2009. – ISBN 978-3-938860-27-4

**Kocher 1996**
KOCHER, Paul C.: *Timing Attacks on Implementations of Diddie-Hellman, RSA, DSS, and Other Systems*. – Online verfügbar unter: http://www.cryptography.com/public/pdf/TimingAttacks.pdf Abruf: 2015-06-01

**Network Working Group 1969**
NETWORK WORKING GROUP (Hrsg.): *ASCII format for Network Interchange*. – Online verfügbar unter: http://tools.ietf.org/html/rfc20 Abruf: 2015-06-01

**Tunstall 2012**
TUNSTALL, Michael ; JOYE, Marc (Mitarb.): *Fault Analysis in Cryptography* : *Information Security and Cryptography*. New York : Springer-Verlag Heidelberg, 2012. – ISBN 978-3-642-29655-0

**Effing 2008**
EFFING, Wolfgang ; RANKL, Wolfgang: *Handbuch der Chipkarten* : *Aufbau – Funktionsweise – Einsatz von Smart Cards*. 5., überarbeitete und erweiterte Auflage. München : Carl Hanser Verlag, 2008. – ISBN 978-3-446-40402-1

**EA 2014**
ELECTRONIC ASSEMBLY (Hrsg.): *EA eDIP128-6* : *Operating Unit 128x64 with touch panel*. Stand: 2014-01-01. – Online verfügbar unter: http://www.lcd-module.de/fileadmin/eng/pdf/grafik/edip128-6e.pdf Abruf: 2015-06-01

**I2C 2014**
NXP SEMICONDUCTORS (Hrsg.): *I²C-bus specification and user manual* : *UM10204*. Stand: 2014-04-06. – Online verfügbar unter: http://www.nxp.com/documents/user_manual/UM10204.pdf Abruf: 2015-06-01

# Abbildungsverzeichnis

# Verzeichnis der Quellcodeausschnitte

# Inhaltsverzeichnis der beigelegten CD

Die vorliegende Bachelorarbeit ist auf CD1 enthalten.
Zusätzlich enthält CD2 den vollständigen Quellcode der Arbeit, sowie die in Abschnitt „7 Messungen“ gezeigten Bilder zu Testaufbau und Messergebnissen in voller Auflösung.

- **Messungen** — Bilder vom Testaufbau, den Beispielszenarien, sowie Rechenzeit und Jitter der Triggermodi
  - Scenario1
  - Scenario2
  - TriggerMode_State
  - TriggerMode_EdgeCount
  - TriggerMode_PulseDetect
- **Quellcode** — Quellcode aller implementierten Komponenten
  - ConfRAM
  - HoldOffAndWindowTimer
  - I2CControlCRAM
  - I2CDisplayDataMemory
  - I2CMaster
  - I2CMasterAndSlave
  - I2CMasterAskSlave
  - I2CSlave
  - InToOutSwitch
  - PulseFilter
  - Text2Vhdl
  - TickGen
  - Toplevel
  - TriggerCell
  - TriggerGenerator

# Anhang A

Die folgenden Abschnitte erläutern die Vorgehensweise bei der Konfiguration des Triggergenerators und demonstrieren diese anhand der Umsetzung des ersten Beispiels (siehe Abschnitt „7.2.1 Beispiel 1: Demonstration der Unterbedingungsarten“):

**Verwendung des ConfRAM** (S. 93):

- Zugriff auf den ConfRAM
- Identifikation von ConfRAM-Inhalten

**Realisierung der Beispielszenarien** (S. 95):

- Konfiguration des ersten Beispiel
- Übertragung der Konfiguration in den ConfRAM

**Umdimensionierung der Komponenten** (S. 99):

- Dimensionierung des gesamten Systems (Komponentenanzahlen)
- Dimensionierung einzelner Komponenten

**Konfiguration der Ein-/Ausgänge und Signalweiterleitung** (S. 101):

- Signale uni-/bidirektional durch den FPGA leiten:
  - Wahlweise unterbrechbar (über IntToOutSwitch)
  - Wahlweise mit Pulsfilterung (über Pulsfilter)
- Einbindung der TriggerGenerator-Komponente

**Displayausgabe** (S. 103):

- Erstellung einer Konfigurationsdatei für Text2Vhdl.exe
- Einbindung der generierten I2CDisplayDataMemory-Komponente

## Verwendung des ConfRAM

Der ConfRAM kann sowohl sequentiell, als auch beliebig beschrieben und gelesen werden. Dazu verwaltet der I²C-Controller einen Iterator, der mithilfe eines Kommandos manipuliert werden kann (siehe Abb. 65, S. 113). Dieser zeigt jeweils auf das Datenwort, welches mit dem nächsten Schreibbefehl beschrieben bzw. dem nächsten Lesebefehl ausgelesen wird. Grundsätzlich ist es möglich eine Datenwortsequenz beliebiger Länge in beiden Richtungen zu übertragen. Wird dabei das Ende des ConfRAM erreicht, findet ein Umbruch an dessen Anfang statt. Im Falle eines Schreibvorgangs, können Nutzdaten und Kommandos gemischt werden. Die empfangenen Kommandos werden unmittelbar nach Empfang ausgeführt.

Die im ConfRAM enthaltenen Daten müssen immer bis zu einer Datenwortgrenze ausgelesen werden, bzw. dieser muss immer mit ganzen Datenwörtern beschrieben werden. Andernfalls wird die letzte Datenwortübertragung als fehlerhaft betrachtet und der Iterator wird nicht verändert.

Die Datenwortlänge beträgt 32-Bit. Demzufolge muss immer eine restlos durch 4 teilbare Byteanzahl innerhalb einer I²C-Nachricht übertragen bzw. ausgelesen werden.

Das in Abb. 65 (siehe S. 113) in binärer Schreibweise dargestellte Kommando zur Iteratormanipulation entspricht dem hexadezimalen $0xff800001$. Während die höherwertigen 16-Bit zur Identifikation des Kommandos dienen, stehen die niederwertigen 16-Bit der Übertragung einer Datenwortadresse zur

Verfügung. In diesem Fall wird der Iterator auf den Wert 1 gesetzt werden. Als umfangreicheres Beispiel soll folgende Datenwortsequenz dienen:

$$ff800000\ 00000004\ 00000005\ ff800001\ 00000006\ 00000007\ ff800000$$

Die der Reihe nach übertragenen Datenwörter würden folgendes bewirken:

$ff800000$: Iterator = 0
$00000004$: ConfRAM#0 = 4, Iterator = 1
$00000005$: ConfRAM#1 = 5, Iterator = 2
$ff800001$: Iterator = 1
$00000006$: ConfRAM#1 = 6, Iterator = 2
$00000007$: ConfRAM#2 = 7, Iterator = 3
$ff800000$: Iterator = 0

Ein anschließendes Auslesen von 3 Datenwörtern ($3 * 32\ Bit = 12\ Byte$) würde folgendes Ergebnis liefern:

| Ausgelesene Werte: | 00000004 | 00000006 | 00000007 |
|---|---|---|---|
| Ausgelesen von: | (ConfRAM#0) | $(ConfRAM\#1)$ | $(ConfRAM\#2)$ |

Es sein angemerkt, dass beim Auslesen nur die im ConfRAM gespeicherten Werte zurückgeliefert werden. Diese sollten, müssen aber nicht mit den zuvor hineingeschriebenen Werten übereinstimmen. Denn werden beispielsweise 32-Bit in ein ConfRAM-Datenwort geschrieben, von dem nur 2-Bit verwendet werden, werden die oberen 30-Bit verworfen. Die Anzahl der verwendeten Bits pro Datenwort kann der Benutzer in einigen Fällen individuell für jede Komponente im Package DEFINITIONS_pack angeben, in anderen Fällen werden diese aus bestimmten Abhängigkeiten errechnet.

## Bestimmung der ConfRAM-Speicherbelegung

Die Größe der im ConfRAM befindlichen Speicherbereiche kann mithilfe der im Package DEFINITIONS_pack definierten Konstanten (siehe 0 Generische Attribute und Formatierung) errechnet werden. Ausgangspunkt sind die Anzahlen verwendeter Komponenten:

$$numTriggerCells = 24$$
$$numTGOutputs = 8$$
$$numTGTickGens = 2$$
$$numInToOutSwitches = 1$$
$$numPulseFilters = 1$$

Da die Komponenten in dieser Reihenfolge im ConfRAM liegen, kann über die Anzahl der jeweils benötigten Datenwörter auf die Bereichsgrenzen geschlossen werden:

$$TriggerCellMem_{numWords} = 10$$
$$TriggerGenOutputMem_{numWords} = 1$$
$$TickGenMem_{numWords} = 2$$
$$InToOutSwitchMem_{numWords} = 3$$
$$PulsFilterMem_{numWords} = 2$$

Die resultierenden Bereichsgrenzen können wie folgt berechnet werden:

$\mathrm{TriggerCellMemArea}_{\mathrm{Offset}} = 0$ Der TriggerCell-Speicherbereich beginnt mit Datenwort 0.

$$\mathrm{TriggerGenOutputMemArea}_{\mathrm{Offset}} = \mathrm{TriggerCellMemArea}_{\mathrm{Offset}} + \mathrm{TriggerCellMem}_{\mathrm{numWords}} * \mathrm{numTriggerCells} = 0 + 10 * 24 = 240$$

Der Speicherbereich für die Triggergeneratorausgangskonfiguration beginnt mit Datenwort 240. Denn der vorhergehende TriggerCell-Speicherbereich hat eine Größe von „Anzahl der Datenwörter pro Triggerzelle“ * „Anzahl der Triggerzellen“. Auf die gleiche Weise werden die Größen der übrigen Speicherbereiche berechnet.

$$\begin{aligned}&\mathrm{TickGenMemArea}_{\mathrm{Offset}}\\&\quad=\mathrm{TriggerGenOutputMemArea}_{\mathrm{Offset}}+\mathrm{TriggerGenOutputMem}_{\mathrm{numWords}}\\&\quad*\mathrm{numTGOutputs}=240+1*8=248\end{aligned}$$

Der TickGenerator-Speicherbereich beginnt mit Datenwort 248.

$$\begin{aligned}&\mathrm{InToOutSwitchMemArea}_{\mathrm{Offset}}\\&\quad=\mathrm{TickGenMemArea}_{\mathrm{Offset}}+\mathrm{TickGenMem}_{\mathrm{numWords}}*\mathrm{numTGTickGens}\\&\quad=248+2*2=252\end{aligned}$$

Der InToOutSwitch-Speicherbereich beginnt mit Datenwort 252.

$$\begin{aligned}&\mathrm{PulseFilterMemArea}_{\mathrm{Offset}}\\&\quad=\mathrm{InToOutSwitchMemArea}_{\mathrm{Offset}}+\mathrm{InToOutSwitchMem}_{\mathrm{numWords}}\\&\quad*\mathrm{numInToOutSwitches}=252+3*1=255\end{aligned}$$

Der PulsFilter-Speicherbereich beginnt mit Datenwort 255.

$$\begin{aligned}&ConfRAM_{Total}=PulseFilterMemArea_{Offset}+PulsFilterMem_{numWords}*numPulseFilters\\&\quad=255+2*1=257\end{aligned}$$

In diesem Beispiel enthält der ConfRAM 257 Datenwörter (0 bis 256).

## Realisierung der Beispielszenarien

Um die Konfiguration des Triggergenerators dem ersten Anwendungsbeispiel (siehe „7.2.1 Beispiel 1: Demonstration der Unterbedingungsarten“) entsprechend durchzuführen, müssen die in Abb. 38 bildlich dargestellten Einstellungen in den ConfRAM übertragen werden.

### Konfiguration der Triggerzellen

Zur Konfiguration der Triggerzellen sind die in dem Record-Typ tTriggerCellMemMap (siehe Abb. 82, S. 131) enthaltenen Parameter folgendermaßen zu definieren:

| Record-Type: tTriggerCellMemMap | TC0 Addr Value | TC1 Addr Value | TC2 Addr Value | TC3 Addr Value | TC4 Addr Value |
|---|---|---|---|---|---|
| Mode | 00: 0 (Highstate) | 0a: 0 (Highstate) | 14: 5 (LowPulse) | 1e: 2 (HighEdge) | 28: 3 (LowEdge) |
| Input Mux | 01: 0 (VDD) | 0b: 2 (RST) | 15: 3 (IO) | 1f: 1 (CLK) | 29: 1 (CLK) |
| Enable Mux | 02: 1 (uncond.) | 0c: 2 (TC0) | 16: 3 (TC1) | 20: 4 (TC2) | 2a: 4 (TC2) |
| CE Mux | 03: 0 (const. 1) | 0d: 0 (const. 1) | 17: 0 (const. 1) | 21: 0 (const. 1) | 2b: 0 (const. 1) |
| HOWT CE Mux | 04: 0 (const. 1) | 0e: 0 (const. 1) | 18: 0 (const. 1) | 22: 0 (const. 1) | 2c: 0 (const. 1) |
| ENPC Minval | 05: 0 (not used) | 0f: 0 (not used) | 19: 14 (0.4 µs) | 23: 0 (not used) | 2d: 0 (not used) |
| ENPC Maxval | 06: 0 (not used) | 10: 0 (not used) | 1a: 28 (0.8 µs) | 24: 3 (Third Edge) | 2e: 6 (Sixth Edge) |
| HoldOff | 07: 0 (no delay) | 11: 0 (no delay) | 1b: 0 (no delay) | 25: 3 (60 ns) | 2f: 0 (no delay) |
| Window | 08: 0 (∞) | 12: 0 (∞) | 1c: 0 (∞) | 26: A (200 ns) | 30: 20 (640 ns) |
| Reset Trigger | 09: 5 (TrgOut4) | 13: 5 (TrgOut4) | 1d: 5 (TrgOut4) | 27: 0 (no reset) | 31: 0 (no reset) |

Abb. 52: ConfRAM-Inhalt für die Triggerzellen in Beispiel 1

Abb. 52 zeigt die für jede Triggerzelle gespeicherten Werte, sowie die jeweilige Datenwortadresse im ConfRAM, beides in hexadezimaler Darstellung. Die Bedeutungen und Wertebereiche der einzelnen Parameter seien im Folgenden zusammengefasst:

- **Mode**: Die verfügbaren Triggerzell-Betriebsmodi sind linear durchnummeriert:
  0 = Highstate, 1 = Lowstate, 2 = HighEdgeCount, 3 = LowEdgeCount, 4 = HighPulseDetect, 5 = LowPulseDetect.
- **Input Mux**: Indexiert den Eingangsvektor der Triggergenerator-Komponente. Dessen Zuweisung zu den durchgeleiteten Signalleitungen muss innerhalb der Toplevel-Komponente vorgenommen werden. In diesem Beispiel ist der Eingangsvektor vier Signalleitungen breit, welche folgendendermaßen belegt wurden:
  TGInput(0) = PWR_Reader, TGInput(1) = CLK_Reader, TGInput(2) = RST_Reader, TGInput(3) = IO_Card.
- **Enable Mux**: Wählt die Triggerzelle, deren Triggersignal zum enablen verwendet werden soll:
  0 = Unbedingt Disabled, 1 = Unbedingt Enabled, 2 = TC0-Trigger, 3 = TC1-Trigger usw. ausgenommen der jeweils konfigurierten Triggerzelle (siehe Abb. 97, S. 143).
- **CE Mux/ HOWT CE Mux**: Auswahl eines TickGenerators:
  0 = CE-Signal ist konstant 1 (kein TickGenerator), 1 = TickGenerator0 usw.
- **ENPC Minval / ENPC Maxval**: Je nach Modus zur Definition von Pulslängen als Anzahl von CPU-Takten bzw. ENPC Maxval alternativ für eine Anzahl zu erkennender Flanken. In diesem Beispiel verwendet TC2 beide Parameter, für die folgende Werte berechnet wurden:
  $$0.4\mu s/20ns = 20\ (0x14)\ CPU\ Takte(bei\ 50MHz)$$
  $$0.8\mu s/20ns = 40\ (0x28)\ CPU\ Takte$$
  TC3 und TC4 zählen 3 bzw. 6 Flanken, wobei der in ENPC Minval stehende Wert ignoriert wird.
- **HoldOff**: Länge der Triggerverzögerung in CPU-Takten. TC3 verwendet einen Wert von:
  $$60ns/20ns = 3\ CPU\ Takte$$
- **Window**: Länge der Triggerausgabe in CPU-Takten. Ein Wert von 0 bedeutet eine unbegrenzt lange Ausgabe, bis zum nächsten Reset. Dieser kann durch andere Triggerzellen oder das I²C-Resetkommando ausgelöst werden. TC3 und TC4 verwenden folgende Window-Werte:
  $$200ns/20ns = 10\ (0xA)\ CPU\ Takte$$
  $$640ns/20ns = 32\ (0x20)\ CPU\ Takte$$
- **Reset Trigger**: Auswahl einer Trigger-Ausgabesignalleitung der TriggerGenerator-Komponente, deren fallende Triggerflanke als Resetsignal verwendet werden soll. Ein Wert von 0 bedeutet, dass kein triggerbasierter Reset für die jeweilige Triggerzelle konfiguriert wurde. Werte größer 0 selektieren hingegen einen der Triggerausgänge (siehe Abb. 95, S. 141):
  1 = TriggerOut0, 2 = TriggerOut1, 3 = TriggerOut2, 4 = TriggerOut3, 5 = TriggerOut4 usw.

## Konfiguration der Triggerausgänge

Zur Konfiguration der Triggerausgänge ist der im Record-Typ tTriggerGenOutputMemMap (siehe Abb. 83, S. 131) enthaltene Parameter für die fünf verwendeten Ausgänge folgendermaßen zu definieren:

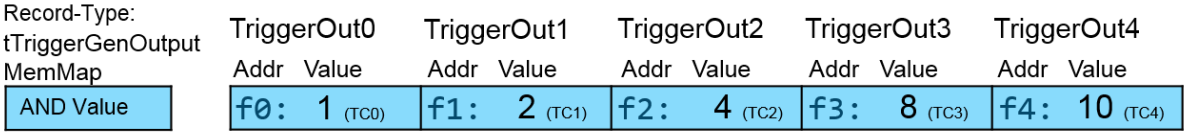


Abb. 53: ConfRAM-Inhalt für die Triggerausgänge in Beispiel 1

Abb. 53 zeigt den für jeden Triggerausgang gespeicherten Wert. Dieser dient zur Selektion einer oder mehrerer Triggersignale über eine Bitmap. Diese speichert jeweils ein Bit pro Triggerzelle. Bei einem hexadezimalen Wert von z.B. 1 ist das erste Bit gesetzt und damit TC0 dem Ausgang zugewiesen. Der Parametername AND-Value weist darauf hin, dass eine UND-Verknüpfung der gewählten Triggerzellen möglich ist. Sind mehrere Bits gesetzt, müssen alle entsprechenden Triggerzellen ein Triggersignal

ausgeben, um die Ausgabe eines Triggersignals auf einem Ausgang zu bewirken. Ein hexadezimaler Wert von z.B. 3 würde bedeuten, dass die ersten beiden Bits der Bitmap gesetzt sind, also TC0 und TC1 dem Ausgang zugewiesen wurden. Nur für die Zeitspanne, in der beide ein Triggersignal ausgeben, wird dieses auch über den Ausgang ausgegeben. Eine derartige Verwendung macht beispielsweise dann Sinn, wenn das Eintreten zweier Ereignisse gewartet werden soll, bei denen die Reihenfolge des Eintritts nicht bekannt ist.
Enthält die Bitmap eines Ausgangs den Wert 0, das heißt diesem wurde keine Triggerzelle zugewiesen, dann gibt dieser Ausgang ein konstantes High-Signal aus. Dieses bleibt auch nach Ausführung des I²C-Resetkommandos bestehen. Auf diese Weise können nicht benutzte Triggerausgänge bzw. eine Fehlkonfiguration erkannt werden.

## Konfiguration der Tickgeneratoren

Zur Konfiguration der TickGeneratoren sind die im Record-Typ tTickGenMemMap (siehe Abb. 84, S. 131) enthaltenen Parameter zu definieren:

| Record-Type: tTickGenMemMap | TickGen0 Addr Value | TickGen1 Addr Value |
|---|---|---|
| Count Value | f8: 1 (Clock / 2) | fa: 3 (Clock / 4) |
| Enable | f9: 1 (Enabled) | fb: 0 (Disabled) |

Abb. 54: ConfRAM-Inhalt der Tickgeneratoren (nicht aus Beispiel 1)

In der für die Beispiele verwendeten Schaltungskonfiguration sind zwei Tickgeneratoren untergebracht, welche jedoch nicht verwendet werden. Die in Abb. 54 gezeigte Konfiguration ist für Beispiel 1 nicht erforderlich. Die Bedeutungen und Wertebereiche der beiden Parameter seien im Folgenden zusammengefasst:

- **Count Value**: Grenzwert für den internen Zähler des Tickgenerators. Die dargestellten Werte haben folgende Bedeutungen:
  1 = Es wird im CPU-Takt von 0 bis 1 gezählt. Dadurch entsteht ein Ticksignal mit halber Taktfrequenz,
  3 = Ticksignal mit einem Viertel der Taktfrequenz,
  0 = Ausgabe eines konstanten High-Signals.
- **Enable**: Ermöglicht es einen Tickgenerator zu disablen, wenn dieser nicht gebraucht wird. Wenn disabled, wird ein konstantes High-Signal ausgegeben.

## Konfiguration der InToOutSwitches

Zur Konfiguration des in Beispiel 1 verwendeten InToOutSwitches sind die im Record-Typ tInToOutSwitchMemMap (siehe Abb. 85, S. 132) enthaltenen Parameter folgendermaßen zu definieren:

| Record-Type: tInToOutSwitchMemMap | ITOS0 Addr Value |
|---|---|
| Triggerselect Mux | fc: 5 (TC4) |
| Default Output Value | fd: 2 (Input) |
| Triggered Output Value | fe: 0 (const. 0) |

Abb. 55: ConfRAM-Inhalt des InToOutSwitches in Beispiel 1

Abb. 55 zeigt die für den InToOutSwitch im ConfRAM gespeicherten Parameter in hexadezimaler Darstellung. Die Bedeutungen und Wertebereiche der einzelnen Parameter seien im Folgenden zusammengefasst:

- **Triggerselect Mux**: Auswahl einer Triggerzelle, deren Triggersignal zum Umschalten der Ausgangsbelegung verwendet werden soll. Der Wert 1 würde beispielsweise die erste

Triggerzelle (TC0) auswählen, 2 die zweite (TC1) usw. Der Wert 0 disabled den InToOutSwitch (siehe Abb. 102, S. 150). In diesem Fall wird nie umgeschaltet und durchgängig der Default Output Value ausgegeben.

- **Default Output Value**: Auswahl des Ausgabewertes im nicht-geschalteten Zustand: 0 = Low-Signal, 1 = High-Signal, 2 = Input-Signal.
- **Triggered Output Value**: Auswahl des Ausgabewertes im geschalteten Zustand.

Der in Beispiel 1 verwendete InToOutSwitch gibt im Ausgangszustand das Input-Signal aus. Für die Dauer eines Triggersignals von TC4 wird ein Low-Signal ausgegeben.

## Konfiguration der Pulsfilter

Zur Konfiguration des in Beispiel 1 verwendeten Pulsfilters sind die im Record-Typ tPulseFilterMemMap (siehe Abb. 86, S. 132) enthaltenen Parameter folgendermaßen zu definieren:

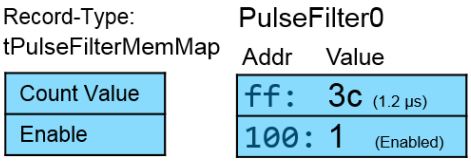


Abb. 56: ConfRAM-Inhalt des PulseFilters in Beispiel 1

Die in Abb. 56 gezeigten Parameter haben folgende Bedeutung:

- **Count Value**: Grenzwert für den internen Zähler von CPU-Takten. Pulse deren Länge diesen Wert unterschreitet, werden herausgefiltert. Der in diesem Beispiel verwendete Wert wurden folgendermaßen ermittelt: $1{,}2\mu s \,/\, 20ns = 60\ (0x3c)\ CPU\ Takte$
- **Enable**: Ermöglicht das disablen des Pulsfilters. Der Zustand des Ausgabesignals bleibt bis zum nächsten enablen erhalten. Solange werden alle eingehenden Pulse ignoriert.

## Übertragung in den ConfRAM

Im Resetzustand werden Datenwörter im ConfRAM auf Null gesetzt. Um die gezeigten Parameter mit geringstmöglichem Aufwand in den ConfRAM zu schreiben bietet es sich somit an einen Reset des Systems vorzunehmen und anschließend nur die Parameter zu übertragen, die einen Wert ungleich Null erhalten sollen:

**Konfiguration der Triggerzellen**:

**TC0**: `ff800000 00000000 00000000 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000005`
**TC1**: `ff80000a 00000000 00000002 00000002 00000000 00000000 00000000 00000000 00000000 00000000 00000005`
**TC2**: `ff800014 00000005 00000003 00000003 00000000 00000000 00000014 00000028 00000000 00000000 00000005`
**TC3**: `ff80001e 00000002 00000001 00000004 00000000 00000000 00000000 00000003 00000003 0000000a 00000000`
**TC4**: `ff800028 00000003 00000001 00000004 00000000 00000000 00000000 00000006 00000000 00000020 00000000`

**Konfiguration der Triggerausgänge**:

**TriggerOut0-4:** `ff8000f0 00000001 00000002 00000004 00000008 00000010`

**Konfiguration des InToOutSwitches**:

**ITOS0:** `ff8000fc 00000005 00000002`

**Konfiguration des Pulsfilters**:

**Pf0:** `ff8000ff 0000003c 00000001`

Im Anschluss an die Übertragung der Daten sollte ein Reset über das Kommando `ff400000` durchgeführt werden. Sonst kann es zu unerwarteten Triggerausgaben kommen, insbesondere da eine Belegung des Window-Parameters der HOWTimer eine Triggerausgabe mit unbegrenzter Länge veranlasst.
Für eine Auflistung der in Beispiel 2 übertragenen Parameter siehe Anhang A.

# Umdimensionierung der Komponenten

Die Wertebereiche der im ConfRAM gespeicherten Parameter wurden zur Einsparung von Hardwareressourcen den Anforderungen von Beispiel 2 angepasst. Die damit einhergehenden Einschränkungen (z.B. geringe Zählerbitbreiten) der in den einzelnen Triggerzellen waren nötig, um die erforderliche Anzahl von Triggerzellen in der Hardware unterbringen zu können. Andere Anwendungsszenarien können Umdimensionierungen erfordern. Diese werden im Package DEFINITIONS_pack vorgenommen.

## Anzahl verfügbarer Komponenten

Die Anzahlen der Komponenten, die bei der Synthese automatisch mehrfach instantiiert werden, können über die in Code 8 (siehe S. 110) gezeigten Konstanten festgelegt werden. Für die Anwendungsbeispiele wurden folgende Werte gewählt:

- **numTriggerCells = 24**
  Die Triggerzellen werden automatisch integriert.
- **numTGInputs = 16**
  Die TriggerGenerator-Komponente liest 16 Signalleitungen, die von den Triggerzellen beobachtet werden können. Eine Änderung des Wertes erfordert eine Anpassung der Instantiierung der TriggerGenerator-Komponente innerhalb der Toplevel-Komponente.
- **numTGTickGens = 2**
  Die gewählte Anzahl von Tickgeneratoren wird automatisch integriert.
- **numTGOutputs = 8**
  Die TriggerGenerator-Komponente gibt 8 Triggersignalleitungen aus. Eine Änderung erfordert es die von der Toplevel-Komponente ausgegebenen Triggersignalleitungen anzupassen.
- **numInToOutSwitches = 1**
  Die InToOutSwitch-Komponente wird nicht automatisch instantiiert, sondern vom Benutzer zwischen die gewünschten Ein- und Ausgänge der Toplevel-Komponente geschaltet. Die Konstante dient zur Dimensionierung des ConfRAM und dessen Ausgängen.
- **numPulseFilters = 1**
  Auch die Pulsfilter werden nicht automatisch bei der Synthese instantiiert, sondern durch den Benutzer, innerhalb der Toplevel-Komponente. Die Konstante wird wiederum zur Dimensionierung des ConfRAM verwendet.

Alle Triggerzellen, Tickgeneratoren und Pulsfilter können über weitere Konstanten individuell dimensioniert werden.

## Dimensionierung der Triggerzellen

Abb. 57 zeigt einen Ausschnitt der Konstanten, die zur Dimensionierung der Triggerzellen verwendet wurden.

| Record-Type: tTcSpec | TC0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10-14 | 15 | 16-24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ENPC Minval Bits | 8 | 8 | 8 | 8 | 16 | 16 | 16 | 8 | 8 | 8 | 8 | 8 | 8 |
| ENPC Maxval Bits | 8 | 8 | 8 | 8 | 16 | 16 | 16 | 8 | 8 | 8 | 8 | 8 | 8 |
| ENPC Count Bits | 8 | 8 | 8 | 8 | 16 | 16 | 16 | 8 | 8 | 8 | 8 | 8 | 8 |
| HoldOff Bits | 8 | 18 | 8 | 16 | 16 | 16 | 16 | 8 | 8 | 16 | 8 | 16 | 8 |
| Window Bits | 8 | 8 | 8 | 8 | 16 | 16 | 16 | 8 | 8 | 8 | 8 | 8 | 8 |
| HOWT Count Bits | 8 | 18 | 8 | 16 | 16 | 16 | 16 | 8 | 8 | 16 | 8 | 16 | 8 |

Abb. 57: Dimensionierung der Triggerzellen

Während die Konstanten „ENPC Minval Bits/Maxval Bits“, „HoldOff Bits“ sowie „Window Bits“ die Bitbreiten der im ConfRAM gespeicherten Parameter bestimmen, legen „ENPC Count Bits“ und „HOWT Count Bits“ die Bitbreiten der jeweiligen Zähler und die Anzahl der tatsächlich verwendeten Bits aus dem ConfRAM fest. Bei Vergabe der Bitbreiten sind folgende Abhängigkeiten zu beachten:

$$ENPC\ Count\ Bits = ENPC\ Maxval\ Bits \geq ENPC\ Minval\ Bits$$

- **ENPC Count Bits**: Die Bitbreite des Zählers eines Edges-And-Pulse-Counters innerhalb einer Triggerzelle sollte gleich der Bitbreite des oberen Zählgrenzwertes (ENPC Maxval Bits) sein.
- **ENPC Minval Bits**: Die Bitbreite des unteren Zählergrenzwertes sollte kleiner oder gleich „ENPC Maxval Bits“ sein.

$$HOWT\ Count\ Bits = MAX(HoldOff\ Bits, Window\ Bits)$$

- **HOWT Count Bits**: Die Bitbreite des Zählers eines HOWTimers sollte der jeweils größeren Bitbreite des zugehörigen HoldOff- und Window-Parameters entsprechen.

Eine andersartige Konfiguration der Bitbreiten kann den Hardwareaufwand erhöhen, ohne einen Nutzen zu bieten.

## Dimensionierung der Tickgeneratoren

In diesem Beispiel wurden zwei Tickgeneratoren instantiiert, deren Bitbreiten Abb. 58 zeigt.

| Record-Type: tTickGenSpec | TickGen0 | TickGen1 |
|---|---|---|
| Count Bits | 10 | 10 |

Abb. 58: Dimensionierung der Tickgeneratoren

Die Konstante „Count Bits“ kann, wie alle anderen Bitbreiten auch, auf einen Wert aus dem Bereich von 1 bis 32 gesetzt werden. Dieser bestimmt die Anzahl der jeweils im ConfRAM gespeicherten Bits.

## Dimensionierung der Pulsfilter

Der in diesem Beispiel verwendete Pulsfilter wurde Abb. 59 entsprechend konfiguriert. „Count Bits“ hat hier die gleiche Bedeutung, wie in Abb. 58.

| Record-Type: tPulseFilterSpec | Pf0 |
|---|---|
| Count Bits | 8 |

Abb. 59: Dimensionierung des Pulsfilters

# Konfiguration der Ein-/Ausgänge und Signalweiterleitung

Die Toplevel-Komponente stellt die Ein- und Ausgänge des Triggergenerators zur Außenwelt. Deren Anzahl kann über die Entity der Toplevel-Komponente verändert werden. Die bei der Synthese verwendete UCF-Datei (User Constraint File) ist entsprechend anzupassen.
Die Signalweiterleitung zwischen zwei Ports kann uni- oder bidirektional (im Open-Collector Betrieb) ablaufen:

- **Unidirektional**: Es reicht einen Toplevel-Eingang direkt mit einem Ausgang zu verbinden, wie Code 3 zeigt.

```
PWR_Card <= PWR_Reader;
RST_Card <= RST_Reader;
```

Code 3: Direkte Signalweiterleitung

Alternativ kann die Weiterleitung über einen InToOutSwitch unterbrechbar gemacht werden, siehe Code 4.

```
ITOS0: InToOutSwitch
port map (      signal_in => CLK_Reader,
                trigger_in => TGOutputs_s,
                signal_out => CLK_Card,
                RAM => RAM_to_intoutswitch_s(0)
);
```

Code 4: Weiterleitung über InToOutSwitch

In diesem Beispiel wurden dessen Ein- und Ausgänge folgendermaßen belegt:

- **Signal_In**: Vom Reader kommendes Clock-Signal das unterbrochen werden soll.
- **Trigger_In**: Triggerausgabevektor der TriggerGenerator-Komponente.
- **Signal_Out**: Zur Smartcard weitergeleitetes Clock-Signal.
- **RAM**: Vom ConfRAM ausgegebene Konfigurationsdaten. Die Anzahl der Elemente des Vektors „RAM_to_intooutswitch_s“ entspricht der Konstanten „numInToOutSwitches“ (siehe Code 8, S. 110). Der bei der Instantiierung gewählte Index von 0 bewirkt, dass ITOS0 über den ersten Datensatz im InToOutSwitch-Speicherbereich des ConfRAM zu konfigurieren ist.

- **Bidirektional**: Zwei bidirektionale Ports des FPGA können über die BiDirIO-Komponente miteinander verbunden werden, wie in Code 5 zu sehen ist.

```
BDIO: BiDirIO
generic map ( PFID => 0 )
port map (
        clk => clk,
        res => reset,
        PfRAM => RAM_to_pulsefilter_s,
        ReaderIO => IO_Reader,
        CardIO => IO_Card
);
```

Code 5: Weiterleitung über BiDirIO mit optionaler Pulsfilterung

In diesem Beispiel wurden dessen Ein- und Ausgänge folgendermaßen belegt:

- **ReaderIO**: Bidirektionale, zum Reader führende IO-Signalleitung.
- **CardIO**: Bidirektionale, zur Smartcard führende IO-Signalleitung.
- **PfRAM**: Vom ConfRAM ausgegebene Konfigurationsdaten für die Pulsfilter.

Die BiDirIO-Komponente enthält einen Pulsfilter, der von CardIO kommende, zu ReaderIO weiterzuleitende Pulse filtert.

Über den generischen Wert PFID ist dem internen Pulsfilter eine ID zuzuweisen. Abhängig von dieser wird ein Datensatz aus dem ConfRAM für dessen Konfiguration ausgewählt. In diesem Beispiel wurde festgelegt, dass die Schaltung nur einen Pulsfilter enthalten soll (über die Konstante numPulseFilters). Somit speichert der ConfRAM nur Daten für einen Pulsfilter (PFID kann maximal 0 sein). Alternativ ist es auch möglich eine BiDirIO-Instanz ohne Pulsfilter generieren zu lassen. In diesem Fall ist PFID auf -1 zu setzen.

## Instantiierung der TriggerGenerator-Komponente

Unabhängig von einer Signalweiterleitung, werden beliebige Eingangssignale der Toplevel-Komponente an die Eingänge der TriggerGenerator-Komponente gelegt, um diese mit den Triggerzellen beobachten zu können. Die Anzahl der Eingänge der TriggerGenerator-Komponente wird über die Konstante numTGInputs festgelegt, in diesem Beispiel 16. Code 6 zeigt die vorgenommenen Signalzuweisungen.

```
TGInputs_s(0) <= PWR_Reader;
TGInputs_s(1) <= CLK_Reader;
TGInputs_s(2) <= RST_Reader;
TGInputs_s(3) <= IO_Card;
TGInputs_s(numTGInputs - 1 downto 4) <= OtherInputs(numTGInputs - 4 - 1 downto 0);

TG: TriggerGenerator
port map (      clk => clk,
                reset => TGReset_s,
                inputs => TGInputs_s,
                outputs => TGOutputs_s,
                RAM => RAM_to_triggergen_s
);
```

Code 6: Instantiierung der TriggerGenerator-Komponente

Die Ein- und Ausgänge der TriggerGenerator-Komponente wurden wie folgt belegt:

- **Inputs**: Diesem Eingang werden die zur Triggererzeugung verwendeten Signalleitungen zugewiesen. In diesem Beispiel werden vier Stück verwendet: PWR, CLK und RST, welche vom Reader erzeugte Signale übermitteln und das IO-Signal von Seite der Smartcard. Die Signalleitung IO wurde auf Seite der Smartcard abgegriffen, da die dort übertragenen Triggerpulse durch die Pulsfilterung nicht auf die Seite des Readers gelangen.
  Die übrigen 12 der 16 Eingänge des Toplevels werden ebenfalls mit der TriggerGenerator-Komponente verbunden, wurden in den Beispielen jedoch nicht verwendet.
- **Outputs**: Die Triggersignalleitungen werden direkt mit den Triggerausgängen des Toplevels verbunden.
- **RAM**: Vom ConfRAM ausgegebene Konfigurationsdaten.

# Displayausgabe

Die auf dem Display ausgegebenen Daten werden innerhalb der I2CDisplayDataMemory-Komponente definiert. Die entsprechende VHDL-Datei kann mithilfe des für diesen Zweck entwickelten Programms „Text2Vhdl.exe“ aus einer Konfigurationsdatei generiert werden.
Um Beispiel 2 möglichst realitätsnah zu gestalten, wurde das Display in diesem Fall dazu verwendet die HoldOff und Window-Längen bestimmter Triggersignale auszugeben. Dies sind die Werte, die in der Praxis automatisiert verändert werden. Ein Blick auf das Display genügt dann zur Kontrolle des Fortschritts des steuernden Algorithmus. Abb. 60 zeigt die gewählte Displayausgabe.

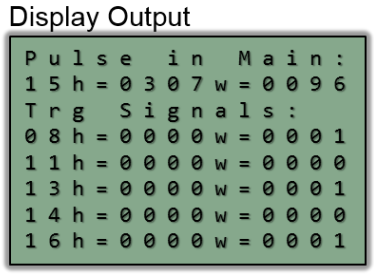


Abb. 60: Displayausgabe für Beispiel 2

In der zweiten Displayzeile werden die Zeitspannen, welche zum Erreichen des mittleren Pulses nach Beginn der Main-Funktion benutzt werden, angezeigt. Falls diese von einem automatisierten System verändert werden, um statt des mittleren, den ersten oder dritten Puls zu erreichen, kann dies auf dem Display beobachtet werden. Die restlichen Zeilen geben die Verzögerungen und Längen der relevanten Triggersignale aus.

## Generierung der I2CDisplayDataMemory-Komponente

Der Inhalt der Text2Vhdl-Konfigurationsdatei ist in drei Abschnitte unterteilt:

- **[VhdlFile]**: Hier definiert das Attribut path den Speicherort der generierten VHDL-Datei. In diesem Beispiel wurde folgender Wert benutzt:
  $$\boldsymbol{path} = \text{"}\boldsymbol{C}{:}\backslash\boldsymbol{vulnerabilitygroup\_students}\backslash\boldsymbol{TriggerGenerator}\backslash\boldsymbol{I2CDisplayDataMemory}\backslash\boldsymbol{src}\backslash\boldsymbol{I2CDisplayDataMemory.vhd}\text{"}$$

- **[ExternalSignals]**: Im zweiten Abschnitt werden die jeweils ein Byte breiten Eingangsvektoren der Komponente definiert. Diese empfangen die auf dem Display auszugebenden ConfRAM-Daten. Der folgende Ausschnitt zeigt die Definition der vier Bytes des HoldOff-Datenworts von TC15. Die übrigen Definitionen finden sich in Anhang B.
  $$\boldsymbol{sig\#0} = \boldsymbol{tc15\_holdoff\_b0}$$
  $$\boldsymbol{sig\#1} = \boldsymbol{tc15\_holdoff\_b1}$$
  $$\boldsymbol{sig\#2} = \boldsymbol{tc15\_holdoff\_b2}$$
  $$\boldsymbol{sig\#3} = \boldsymbol{tc15\_holdoff\_b3}$$
  Der frei wählbare Name eines Eingangssignals sollte möglichst selbsterklärend sein, um die Übersicht bei der Instantiierung der Komponente zu verbessern.

- **[RomContent]**: Der letzte Abschnitt enthält drei verschiedene Attribute:
  - Header gibt eine Bytefolge an, die vor den Nutzdaten an das Display geschickt werden sollen:
    $$\boldsymbol{header} = \boldsymbol{1b5948001b5443000c}$$

    Dieser Wert initialisiert das Display und schaltet u.A. dessen Hintergrundbeleuchtung an. Die Bedeutung der einzelnen Werte kann dem Datenblatt des Herstellers (siehe EA 2014) entnommen werden.

- Delimiter definiert eine Bytefolge, die zur Trennung der Zeilen dienen soll. In diesem Beispiel wurde folgender Wert verwendet:

$$\boldsymbol{delimiter = 0a0d}$$

- Die auf dem Display auszugebenden Zeilen können konstante Zeichen und Eingangssignalbezeichner enthalten. Letztere werden aus einem vorangestellten Prozentzeichen ‚%' und nachfolgender Eingangssignalnummer zusammengesetzt. Es ist darauf zu achten, dass auf dem Display maximal sieben Zeilen mit jeweils bis zu 14 Zeichen dargestellt werden können. Die in Abb. 60 gezeigte Ausgabe wurde über folgende Zeilendefinitionen erreicht:

$$\boldsymbol{line\#0 = Pulse\ in\ Main}:$$
$$\boldsymbol{line\#1 = 15h = \%3\%2\%1\%0w = \%7\%6\%5\%4}$$
$$\boldsymbol{line\#2 = Trg\ Signals}:$$
$$\boldsymbol{line\#3 = 08h = \%11\%10\%9\%8w = \%15\%14\%13\%12}$$
$$\boldsymbol{line\#4 = 11h = \%19\%18\%17\%16w = \%23\%22\%21\%20}$$
$$\boldsymbol{line\#5 = 13h = \%27\%26\%25\%24w = \%31\%30\%29\%28}$$
$$\boldsymbol{line\#6 = 14h = \%35\%34\%33\%32w = \%39\%38\%37\%36}$$
$$\boldsymbol{line\#7 = 16h = \%43\%42\%41\%40w = \%47\%46\%45\%44}$$

Nach Erstellung der Konfigurationsdatei, kann das Programm „Text2Vhdl.exe" mit ihr als einzigem Parameter gestartet werden. Wird das Programm ohne Parameter gestartet, versucht es eine Konfigurationsdatei mit dem Namen „displayContent.txt" aus seinem Arbeitsverzeichnis zu öffnen.

## Instantiierung der I2CDisplayDataMemory-Komponente

Die Instantiierung erfolgt innerhalb des I²C-Controllers (der I2CControlCRAM-Komponente). An die Eingänge der I2CDisplayDataMemory-Komponente sind die an das Display zu übertragenden Bytes anzulegen. Das verwendete Display empfängt ASCII-Bytes und stellt deren repräsentative Zeichen dar. Bei der hier gewählten hexadezimalen Ausgabeform von ConfRAM-Daten wird jeweils ein Zeichen zur Darstellung von vier Datenbits ausgegeben. Zur Konvertierung eines Nibbles (4 Bit) in das entsprechende ASCII-Byte kommt die ToHexChar-Komponente zum Einsatz. Abb. 61 zeigt deren Verwendung zur Konvertierung der ersten vier Bit des HoldOff-Wertes von Triggerzelle TC15.

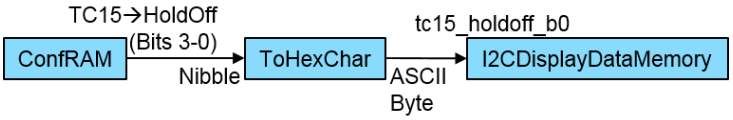


Abb. 61: Verwendung der ToHexChar-Komponente als Blockdiagramm

Die Umsetzung im VHDL-Code zeigt Code 7.

```
HexConv0:
ToHexChar port map (
        Nibble => ConfRAM(TriggerCellMemArea_Offset + TriggerCellMem_numWords * 15 + 7)(3 downto 0),
        HexByte => rom_val_s(0)
);
```

Code 7: Verwendung der ToHexChar-Komponente in VHDL

Die Berechnung einer Datenwortadresse des ConfRAM erfolgt über die Konstanten der Speicherbereich-Offsets. In diesem Fall wird vom Beginn des Triggerzell-Speicherbereichs aus der Beginn des 15 Triggerzell-Datensatzes berechnet und aus dessen achtem Element die ersten vier Bits an die ToHexChar-Komponente weitergereicht. Deren ASCII-Ausgabebyte wird über das Hilfssignal „rom_val_s(0)" an den Eingang „tc15_holdoff_b0" des I2CDisplayDataMemory weitergeleitet.

# Anhang B

Auflistung aller I²C-Nachrichtenpakete zur Übertragung der Konfigurationsparameter von Beispiel 2 in den ConfRAM.

**Konfiguration der Triggerzellen**:

**TC0**: ff800000 00000001 00000003 00000001 00000000 00000000 00000000 00000000 00000000 0000000f 00000000
**TC1**: ff80000a 00000000 00000002 00000001 00000000 00000000 00000000 00000000 0003ffff 0000000f 00000006
**TC2**: ff800014 00000000 00000002 00000003 00000000 00000000 00000000 00000000 00000000 00000000 00000008
**TC3**: ff80001e 00000001 00000003 00000004 00000000 00000000 00000000 00000000 00001388 0000000f 00000000
**TC4**: ff800028 00000005 00000003 00000004 00000000 00000000 000004b0 00000578 0000012c 0000041a 00000000
**TC5**: ff800032 00000005 00000003 00000006 00000000 00000000 000002ee 000003b6 0000012c 00000258 00000000
**TC6**: ff80003c 00000005 00000003 00000007 00000000 00000000 0000012c 000001f4 00000000 00000001 00000000
**TC7**: ff800046 00000000 00000002 00000008 00000000 00000000 00000000 00000000 00000000 00000000 00000007
**TC8**: ff800050 00000000 00000002 00000008 00000000 00000000 00000000 00000000 00000000 00000001 00000000
**TC9**: ff80005a 00000000 00000002 00000009 00000000 00000000 00000000 00000000 00001d4c 00000001 00000006
**TC10**: ff800064 00000000 00000002 0000000b 00000000 00000000 00000000 00000000 00000000 00000000 00000007
**TC11**: ff80006e 00000000 00000002 0000000b 00000000 00000000 00000000 00000000 00000000 00000000 00000007
**TC12**: ff800078 00000005 00000003 0000000c 00000000 00000000 0000005a 0000006e 00000000 00000001 00000000
**TC13**: ff800082 00000000 00000002 0000000e 00000000 00000000 00000000 00000000 00000000 00000001 00000000
**TC14**: ff80008c 00000003 00000003 0000000c 00000000 00000000 00000000 00000005 00000000 00000000 00000007
**TC15**: ff800096 00000000 00000002 0000000e 00000000 00000000 00000000 00000000 00000307 00000096 00000000
**TC16**: ff8000a0 00000005 00000003 00000011 00000000 00000000 00000028 0000003c 00000000 00000001 00000000

**Konfiguration der Triggerausgänge**:

**TrgOut0-7:** ff8000f0 00000100 00000800 00004000 00002000 00010000 00000001 00000002 00000008

**Konfiguration des InToOutSwitches**: (Default Out Val ist Input-Signal, keine Umschaltung)

**ITOS0:** ff8000fd 00000002

**Konfiguration des Pulsfilters**:

**Pf0:** ff8000ff 0000003c 00000001

# Anhang C

Vollständige Konfigurationsdatei für die Generierung der in den Beispiele verwendeten I2CDisplayDataMemory- Komponente.

```
[VhdlFile]
path =
"C:\vulnerabilitygroup_students\TriggerGenerator\I2CDisplay
DataMemory\src\I2CDisplayDataMemory.vhd"

[ExternalSignals]
sig#0 = tc15_holdoff_b0
sig#1 = tc15_holdoff_b1
sig#2 = tc15_holdoff_b2
sig#3 = tc15_holdoff_b3
sig#4 = tc15_window_b0
sig#5 = tc15_window_b1
sig#6 = tc15_window_b2
sig#7 = tc15_window_b3
sig#8 = tc8_holdoff_b0
sig#9 = tc8_holdoff_b1
sig#10 = tc8_holdoff_b2
sig#11 = tc8_holdoff_b3
sig#12 = tc8_window_b0
sig#13 = tc8_window_b1
sig#14 = tc8_window_b2
sig#15 = tc8_window_b3
sig#16 = tc11_holdoff_b0
sig#17 = tc11_holdoff_b1
sig#18 = tc11_holdoff_b2
sig#19 = tc11_holdoff_b3
sig#20 = tc11_window_b0
sig#21 = tc11_window_b1
sig#22 = tc11_window_b2
sig#23 = tc11_window_b3
sig#24 = tc13_holdoff_b0
sig#25 = tc13_holdoff_b1
sig#26 = tc13_holdoff_b2
sig#27 = tc13_holdoff_b3
sig#28 = tc13_window_b0
sig#29 = tc13_window_b1
sig#30 = tc13_window_b2
sig#31 = tc13_window_b3
sig#32 = tc14_holdoff_b0
sig#33 = tc14_holdoff_b1
sig#34 = tc14_holdoff_b2
sig#35 = tc14_holdoff_b3
sig#36 = tc14_window_b0
sig#37 = tc14_window_b1
sig#38 = tc14_window_b2
sig#39 = tc14_window_b3
sig#40 = tc16_holdoff_b0
sig#41 = tc16_holdoff_b1
sig#42 = tc16_holdoff_b2
sig#43 = tc16_holdoff_b3
sig#44 = tc16_window_b0
sig#45 = tc16_window_b1
sig#46 = tc16_window_b2
sig#47 = tc16_window_b3

[RomContent]
header = 1b5948001b5443000c
delimiter = 0a0d
line#0 = Pulse in Main:
line#1 = 15h=%3%2%1%0w=%7%6%5%4
line#2 = Trg Signals:
line#3 = 08h=%11%10%9%8w=%15%14%13%12
line#4 = 11h=%19%18%17%16w=%23%22%21%20
line#5 = 13h=%27%26%25%24w=%31%30%29%28
line#6 = 14h=%35%34%33%32w=%39%38%37%36
line#7 = 16h=%43%42%41%40w=%47%46%45%44
```

# Anhang D

Dieser Abschnitt enthält die detaillierten Beschreibungen aller Komponentenimplementierungen. Bei der Reihenfolge wird dabei nach den Hierarchiestufen der Komponenten vorgegangen, welche in Abb. 16 auf Seite 51 zu sehen ist. Begonnen wird dementsprechend mit dem **Toplevel** (S. 108), welcher alle anderen Komponenten enthält. Der **I²C-Controller** (S. 111) ist für die Kommunikation mit dem Reader verantwortlich und enthält dazu einen **I²C-Slave** (S. 116), welcher Gebrauch von der Hilfskomponente **HexToChar** (S. 120) macht. Die auf dem Display auszugebenden Daten sind im **DisplayDataMemory** (S. 122) gespeichert, welcher vom **I²C-Master** (S. 124) ausgelesen wird. Die enthaltenen Daten werden unter Kontrolle des I²C-Controller zum Display übertragen. Vom Reader empfangene Daten werden im **ConfRAM** (S. 128) gespeichert. Dieser versorgt die Komponenten vom **TriggerGenerator** (S. 135) mit den benötigten Konfigurationssignalen. Dazu zählen die **Triggerzelle** (S. 144), welche Teilberechnungen an den **Edges and Pulse Clocks Counter** (S. 146) delegiert, sowie der **HoldOffAndWindowTimer** (S. 148), welcher für die Erzeugung von auszugebenden Triggersignalen zuständig ist. Der **InToOutSwitch** (S. 150) liegt direkt im Toplevel und ermöglicht die Unterbrechung weitergeleiteter Signale. Daneben dient **BiDirIO** (S. 153) der bidirektionalen Signelweiterleitung und der **Pulsefilter** (S. 155) der Filterung kurzer Pulse aus einem Signal. Schließlich kann ein **Tickgenerator** (S. 157) die Arbeitsgeschwindigkeit verschiedener Komponenten verringern. Im TriggerGenerator können die Triggerzellen oder HoldOffAndWindowTimer ein Ticksignal verwenden, im Toplevel kann ein Pulsefilter von einem Ticksignal gesteuert werden.

# Toplevel

Die Toplevel-Komponente stellt die Ein- und Ausgänge des Systems zur Verfügung und verbindet diese mit den in ihr enthaltenen Komponenten. Auch Verbindungen zwischen den enthaltenen Komponenten werden hergestellt. Abb. 62 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

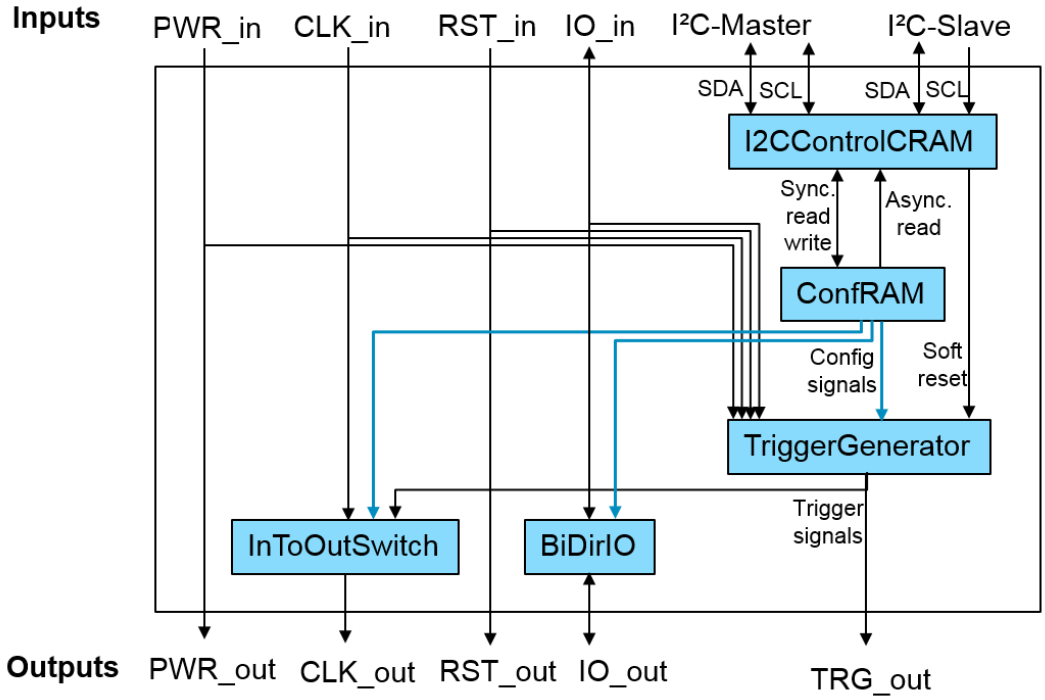


Abb. 62: Toplevel Blockdiagramm

## Ein-/ und Ausgänge

Die Toplevel-Schaltung verfügt über folgende, in Abb. 62 dargestellte, Eingangssignalleitungen:

- **Benutzerdefinierte Eingangssignale**: Maximal 16 Eingänge stehen dem Benutzer zur Belegung mit Signalleitungen zur Verfügung, welche zur Generierung von Triggersignalen ausgewertet werden können. Ein Teil dieser Signalleitungen kann zudem an die Ausgänge durchgeleitet werden. In diesem Beispiel werden vier benutzerdefinierte Signalleitungen verwendet, welche sowohl durchgeleitet, als auch zur Triggergenerierung verwendet werden: PWR_in, CLK_in, RST_in und IO_in.
- **I²C-Slave Schnittstelle**: Vom Reader wird ein unidirektionales Clock-Signal (SCL) empfangen und über ein bidirektionales Datensignal (SDA) empfangen und gesendet.
- **I²C-Master Schnittstelle**: Zur Kommunikation mit dem Display wird eine bidirektionale Clock-Signalleitung (SCL) und eine bidirektionale Daten-Signalleitung (SDA) verwendet.

Des Weiteren existieren bis zu 8 Ausgangssignalleitungen, welche jeweils zur Ausgabe einer der folgenden Signaltypen dienen:

- **Benutzerdefinierte Ausgangssignale**: Dienen zur Ausgabe durchgeleiteter, benutzerdefinierter Eingangssignale. In diesem Beispiel werden alle vier benutzerdefinierten Eingangssignale auf benutzersedinierten Ausgangssignalleitungen ausgegeben: PWR_out, CLK_out, RST_out und IO_out.
- **Triggersignale**: Dienen zur Ausgabe erzeugter Triggersignale. Abb. 62 zeigt nur eine ausgehende Triggersignalleitung (TRG_out), welche stellvertretend für die Menge

verwendeter Signalleitungen steht. In diesem Beispiel stünden bis zu vier Ausgänge zur Ausgabe von Triggersignalen zur Verfügung, da von den acht verfügbaren Ausgängen bereits vier Ausgänge für benutzerdefinierte Ausgangssignale verwendet werden.

## Schaltung

Innerhalb der Toplevel-Komponente werden folgende, in Abb. 62 dargestellte, Komponenten über Signalleitungen miteinander verbunden:

### I²C-Controller (I2CControlCRAM):

Der I²C-Controller verwaltet beide I²C-Busse, einen für die Kommunikation mit dem Reader und einen weiteren für die Kommunikation mit dem Display. Abhängig von den vom Reader empfangenen Kommandos, schreibt der I²C-Controller empfangene Daten synchron in den ConfRAM oder liest diesen synchron aus, um angeforderte Daten über die I²C-Slave Schnittstelle zum Reader zu senden. Dazu nebenläufig wird der ConfRAM auch asynchron ausgelesen, um das Display über die I²C-Master Schnittstelle zu beschreiben. Über eine weitere Signalleitung gibt der I²C-Controller ein Reset-Signal für den Triggergenerator aus. Dies erfolgt immer dann, wenn der Benutzer ein Kommando zum Zurücksetzen der Triggerzellen sendet.

### Konfigurationsspeicher (ConfRAM):

Der Speicher für die, vom Benutzer übermittelten, Konfigurationsdaten gibt diese asynchron an InToOutSwitch, PulseFilter und TriggerGenerator aus.

### Triggererzeugung (TriggerGenerator):

Empfängt benutzerdefinierte Eingangssignale und Konfigurationsdaten vom ConfRAM und gibt erzeugte Triggersignale an Triggerausgänge aus.

### Signalunterbrechung (InToOutSwitch):

Neben dem durchzuleitenden Signal von einem benutzerdefinierten Eingang, werden Konfigurationsdaten vom ConfRAM und alle vom TriggerGenerator ausgegebenen Triggersignale empfangen. Ausgegeben wird entweder das durchgeleitete Eingangssignal oder ein Ersatzsignal auf einem der benutzerdefinierten Ausgänge.

### Bidirektionale Weiterleitung und Pulsfilterung (BiDirIO):

Der Pulsfilter ist in der BiDirIO-Komponente enthalten. Diese sitzt zwischen einem unidirektionalen oder bidirektionalen, benutzerdefinierten Eingang und Ausgang, wobei das Signal in einer Richtung gefiltert und in der anderen unverändert durchgeleitet wird. Zusätzlich werden Konfigurationsdaten vom ConfRAM empfangen. In diesem Beispiel wird das von IO_in kommende Signal gefiltert auf IO_out ausgegeben und das von IO_out kommende Signal unverändert über IO_in ausgegeben. Eine alternative Implementierung des PulseFilters ermöglicht es in beiden Übertragungsrichtungen zu filtern.

Die Anzahl der im Toplevel enthaltenen InToOutSwitch- und PulseFilter-Komponenten wird vom Benutzer bestimmt, indem dieser sie instantiiert und mit den benutzerdefinierten Ein- und Ausgängen, sowie dem ConfRAM verbindet. Alle anderen Komponenten sind jeweils nur einmal vorhanden.

## Generische Attribute

Der Benutzer kann verschiedene Teile des Systems während der Synthese automatisch dimensionieren lassen, indem er bestimmte Parameter vor der Synthese entsprechend definiert. Diese Parameter und einige der Berechnungen, welche die Parameter verarbeiten, werden im DEFINITIONS_pack Package

zentral zusammengefasst und von allen beteiligten Komponenten importiert. Folgende Parameter können vom Benutzer gesetzt werden:

- **numTriggerCells**: Anzahl Triggerzellen, die automatisch erzeugt werden sollen. Zulässig sind Werte von 1 bis 32.
- **numTGInputs**: Anzahl der Eingänge des TriggerGenerators. Dieser Wert gibt an wieviele der benutzerdefinierten Eingangssignale von der TriggerGenerator-Komponente zur Triggererzeugung verwendet werden sollen. Zulässig sind Werte von 1 bis 16, entsprechend dem Wertebereich für die Anzahl benutzerdefinierter Eingangssignale.
- **numTGOutputs**: Anzahl der Ausgänge des TriggerGenerators. Dieser Wert gibt an, über wieviele Triggersignalleitungen die TriggerGenerator-Komponente verfügen soll. Zulässig sind Werte von 1 bis 8, wobei die gesamte Ausgangsanzahl mit den benutzerdefinierten Ausgängen maximal 8 betragen darf.
- **numInToOutSwitches**: Anzahl instantiierter InToOutSwitche. Über diesen Wert gibt der Benutzer bekannt wieviele Instanzen der InToOutSwitch-Komponente er erzeugt hat. Zulässig sind Werte von 1 bis 16, entsprechend dem Wertebereich für die Anzahl benutzerdefinierter Eingangssignale.
- **numPulseFilters**: Anzahl instantiierter PulseFilter. Über diesen Wert gibt der Benutzer bekannt wieviele Instanzen der PulseFilter-Komponente er erzeugt hat. Zulässig sind Werte von 1 bis 16, entsprechend dem Wertebereich für die Anzahl benutzerdefinierter Eingangssignale.
- **numModes**: Anzahl der Modi einer Triggerzelle. Dieser Wert gibt an, wieviele Unterbedingungsarten der Benutzer in den Triggerzellen implementiert hat.

Code 8 zeigt einen Codeausschnitt mit der Definition der Parameter.

```
package DEFINITIONS_pack is
        constant numTriggerCells : integer := 16;
        constant numTGInputs : integer := 4;
        constant numTGOutputs : integer := 4;
        constant numInToOutSwitches : integer := 1;
        constant numPulseFilters : integer := 1;
        constant numModes : integer := 6;
```

Code 8: Definition generischer Attribute.

Die in Code 8 definierten Werte entsprechen der in Abb. 62 dargestellten Schaltung. Die TriggerGenerator-Komponente enthält 16 Triggerzellen, hat 4 Eingänge um die benutzerdefinierten Eingangssignale PWR_in, CLK_in, RST_in und IO_in zu empfangen und kann Triggersignale auf 4 Ausgangsleitungen ausgeben. Außerdem hat der Benutzer einen InToOutSwitch und einen PulseFilter instantiiert. Jede Triggerzelle implementiert zudem 6 Unterbedingungsarten (numModes).
Die komponentenspezifischen Datentypen und Berechnungen innerhalb des DEFINITIONS_pack Package, werden in den folgenden Kapiteln für jede Komponente separat besprochen.

# I²C-Controller

Die I2CControlCRAM-Komponente verwaltet die enthaltenen I²C-Slave und I²C-Master Implementierungen und ermöglicht so den Datenaustausch zwischen Benutzer, ConfRAM und dem während der Entwicklung verwendeten Display. Die Implementierung des I²C-Protokolls innerhalb der I²C-Master und der I²C-Slave-Komponente erfolgte auf Basis der I²C-Spezifikation (siehe I2C 2014). Abb. 63 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

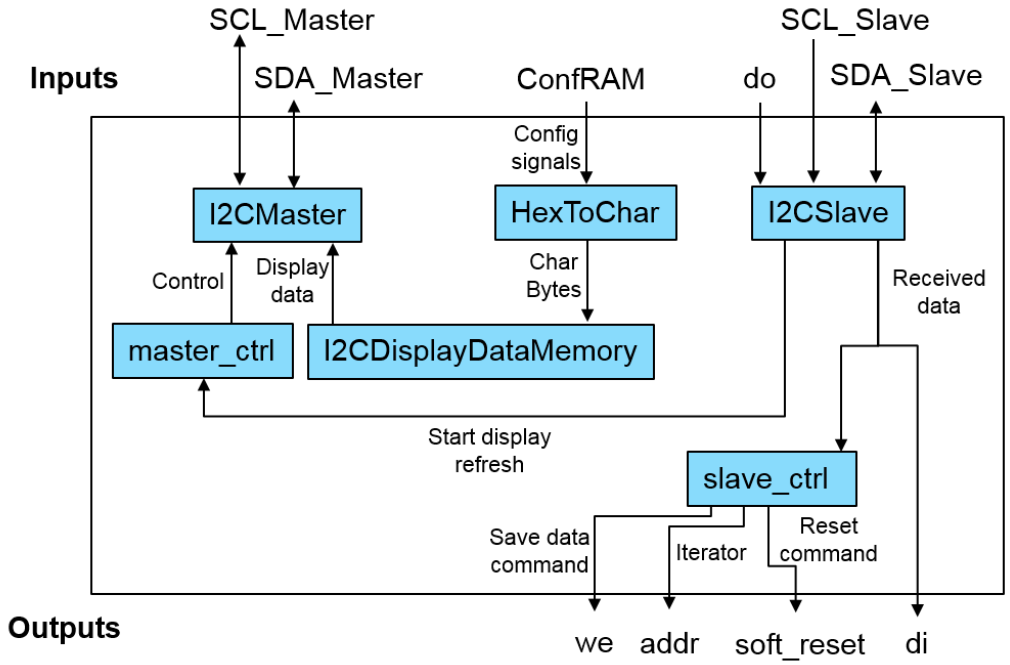


Abb. 63: I²C-Controller Blockdiagramm

## Ein-/ und Ausgänge

Die I2CControlCRAM-Schaltung verfügt über die, in Abb. 63 dargestellten, Eingangssignalleitungen:

- **I²C-Master Schnittstelle**: Eine bidirektionale Clock-Signalleitung (SCL_Master) und eine bidirektionale Daten-Signalleitung (SDA_Master) zur Kommunikation mit dem Display.
- **I²C-Slave Schnittstelle**: Eine unidirektionale Clock-Signalleitung (SCL_Slave) und eine bidirektionale Daten-Signalleitung (SDA_Slave) zur Kommunikation mit dem Reader. Die Clock-Signalleitung ist, entgegen dem I²C-Standard, unidirektional, da die I²C-Slave-Implementierung aus Ersparnisgründen nicht über die Möglichkeit des Clock-Stretchings verfügt.
- **ConfRAM**: Alle im ConfRAM gespeicherten Daten werden asynchron empfangen, um vom I²C-Master ausgelesen und an das Display übertragen zu werden.
- **do**: Ausgehende Datenleitung des ConfRAM. Ein Datenwort aus dem ConfRAM wird mithilfe des I²C-Slave zum Reader übertragen.

Des Weiteren existieren folgende Ausgänge:

- **addr**: Zur Addressierung eines Datenworts im ConfRAM, das gelesen oder überschrieben werden soll.
- **we**: Write-Enable zur Einleitung eines Schreibvorgangs auf den ConfRAM.
- **di**: Eingehende Datenleitung des ConfRAM. Das anliegende Datenwort wird, abhängig von Write-Enable, in den ConfRAM geschrieben.
- **soft_reset**: Resetsignal zum Zurücksetzen der Triggerzellen in der TriggerGenerator-Komponente.

# Schaltung

Die I2CControlCRAM-Komponente enthält folgende, in Abb. 63 dargestellte, Komponenten:

## I²C-Slave (I2CSlave):

Der I²C-Slave kommuniziert direkt mit dem Reader. Empfangene Daten und Kommandos werden über den Ausgang di an den ConfRAM weitergereicht. Nach jeder abgeschlossenen Übertragung gibt der I²C-Slave ein Signal an die slave_ctrl-Komponente aus, welche die empfangenen Daten analysiert und abhängig davon die Ausgänge addr und we steuert. Fordert der Reader Daten an, antwortet der I²C-Slave mit dem am Eingang do anliegenden Datenwort.

## I²C-Slave Controller (slave_ctrl):

Prozess zur Auswertung vom I²C-Slave empfangener Daten und Steuerung des ConfRAM über die Ausgänge addr und we. Zudem findet hier die Auswertung aller Kommandos statt. Implmentiert sind:

- Set Iterator: Kommando zum Setzen des ConfRAM-Iterators.
- Reset Triggercells: Kommando zum Zurücksetzen aller Triggerzellen (siehe letztgenannter Abschnitt). Das erzeugte Resetsignal wird auf dem Ausgang soft_reset ausgegeben.

## I²C-Master (I2CMaster):

Der I²C-Master kommuniziert direkt mit dem Display. Er überträgt ein, vom I2CDisplayDataMemory anliegendes Datenwort an das Display und fordert anschließend, äquivalent zum I²C-Slave, das nächste zu übertragende Datenwort von der master_ctrl-Komponente an.

## Generierung auf dem Display auszugebender Zeichen (HexToChar):

Um die aus dem ConfRAM anliegenden Daten in Form hexadezimaler Zeichen an das Display übermitteln zu können führt die HexToChar-Komponente die entsprechende Umwandlung durch.

## Datenspeicher für das I²C-Display (I2CDisplayDataMemory):

Speichert den konstanten Teil der Daten, die auf dem Display ausgegeben werden sollen. Der Anteil der auszugebenden Werte aus dem ConfRAM wird asynchron aus diesem ausgelesen und von der I2CDisplayDataMemory-Komponente in ihren Speicheradressraum gemappt. So können alle Daten, die auf dem Display angezeigt werden sollen, über die I2CDisplayDataMemory-Schnittstelle ausgelesen und an den I²C-Master weitergereicht werden.

## I²C-Master Controller (master_ctrl):

Prozess zur Steuerung der Übertragung von Daten aus dem I2CDisplayDataMemory über den I²C-Master zum Display. Jede abgeschlossene Übertragung des I²C-Slave führt zu einer erneuten Übertragung aller Daten aus dem I2CDisplayDataMemory über den I²C-Master zum Display.
Hat der I²C-Slave ein Datenwort vom Reader empfangen, startet dies zwei nebenläufige Prozesse. Slave_ctrl analysiert das Datenwort und führt davon abhängige Verarbeitungsschritte durch. Master_ctrl steuert die Übertragung der Displaydaten über den I²C-Master.

## Verarbeitung vom I²C-Slave empfangener Daten

Der Reader sendet Datenworte an den I²C-Slave. Dabei kann es sich entweder um Kommandos handeln oder um Daten die im ConfRAM gespeichert werden sollen. Zunächst soll der Datenempfang betrachtet werden.
Um sowohl einen sequentiellen Zugriff auf mehrere hintereinander liegende Datenworte, als auch den Zugriff auf beliebige, einzelne Datenworte im ConfRAM zu ermöglichen implementiert der I²C-Controller den eingangs erwähnten Speicheriterator, welcher mithilfe eines Kommandos steuerbar ist. Abb. 64 zeigt die Verwendung des Iterators an einem vereinfachten Beispiel.

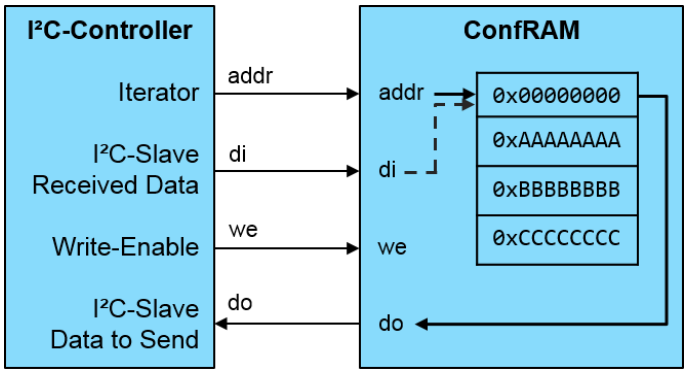


Abb. 64: Beispiel zur Funktion des ConfRAM-Iterator beim Speichern von Daten

Der Wert des Iterators wird über den Ausgang addr an den ConfRAM ausgegeben und dient der Adressierung eines Datenworts, welches der I²C-Controller auszulesen oder zu überschreiben wünscht. Das so adressierte Datenwort wird vom ConfRAM ausgegeben und liegt am Eingang do des I²C-Controllers an. Da der I²C-Slave zu sendende Daten über den Eingang do erhält, würde eine Leseanfrage des Readers dazu führen, dass er eben dieses Datenwort zurückgeliefert bekommt.
Jedes Datenwort, das der I²C-Slave empfangenen hat, wird über den Ausgang di an den ConfRAM ausgegeben und kann durch Ausgabe eines High-Pulses auf dem Ausgang we an die Speicherstelle des ConfRAM kopiert werden, auf die der Iterator jeweils zeigt.
Der Prozess slave_ctrl entscheidet, wann letzteres zu tun ist. Er prüft, ob es sich bei einem empfangenen Datenwort um ein Kommando oder um zu speichernde Daten handelt. Zur Speicherung von Daten, wird zunächst der Speicheriterator inkrementiert und anschließend ein High-Puls auf dem Ausgang we ausgegeben. Auf diese Weise können beliebig viele Datenworte sequentiell vom Reader empfangen und im ConfRAM gespeichert werden.
Fordert der Reader hingegen Daten an, wird kein Write-Enable-Signal erzeugt. In diesem Fall überträgt der I²C-Slave jeweils die aus dem ConfRAM anliegenden Daten. Wie gehabt, wird der Iterator am Ende jeder Übertragung inkrementiert, damit das nächste Speicherwort adressiert wird und so beliebig viele Datenworte vom Reader sequentiell, in einer Übertragung, ausgelesen werden können.

Der Zugriff auf beliebige Speicheradressen wird durch Verwendung des set_Iterator-Kommandos möglich. Zusätzlich ermöglicht das reset_Triggercells-Kommando die Erzeugung eines Resetsignals für alle Triggerzellen. Abb. 65 zeigt die Bedeutung der einzelnen Bits der beiden implementierten Kommandos. Da es keine genauen Auflagen zur Formatierung gab, wurde ein Format gewählt, das den Wertebereich der Konfigurationsparameter nur unwesentlich einschränkt und möglichst flexibel bezüglich der Anzahl und Länge möglicher Kommandoparameter ist.

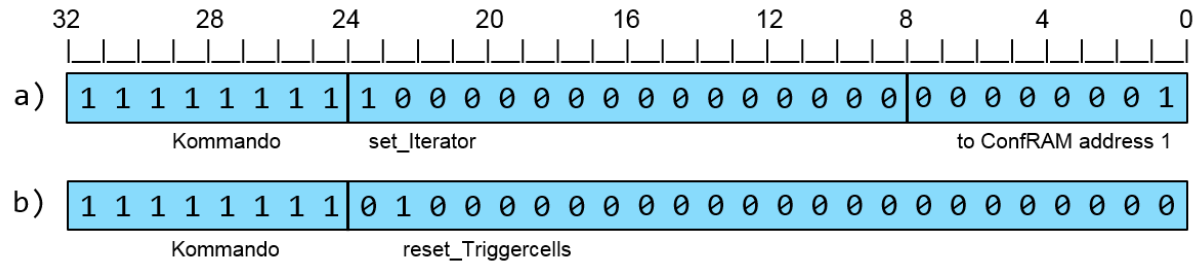


Abb. 65: I²C-Kommandoformat

Empfängt der I²C-Slave ein Kommando vom Reader, liegt dieses zwar am Dateneingang des ConfRAM an, soll aber nicht im ConfRAM abgespeichert, sondern ausgeführt werden. Slave_ctrl erkennt dazu ein Kommando anhand seiner Struktur und führt die entsprechenden Aktionen aus. In dem Fall werden weder der Iterator verändert, noch wird ein Write-Enable-Puls ausgegeben.

Jedes zum I²C-Slave übertragene Datenwort und Kommando hat eine Länge von 32 Bit. Gleiches gilt für die im ConfRAM gespeicherten und die vom I²C-Slave zum Reader übermittelten Datenworte. Erkennungszeichen aller Kommandos ist das, in Abb. 65 dargestellte, High-Byte, welches ab dem most-significant-Bit 31 abwärts bis Bit 24 reicht. Dieses Erkennungsmuster wurde so gewählt, da nicht erwartet wird, dass im späteren Gebrauch 32-Bit Werte dieser Größe verwendet werden.
Die Bits 23 und 22 werden zur Auswahl eines Kommandos benutzt. Eine Eins in Bit 23 steht dabei für das Kommando set_Iterator (siehe Abb. 65-a) und eine Eins in Bit 22 für das Kommando reset_Triggercells (siehe Abb. 65-b). Zur Identifikation eines zukünftig zu implementierenden Kommandos könnte beispielsweise Bit 21 verwendet werden.
Jedes Kommando kann zudem zusätzliche Parameter benötigen, welche ab dem least-significant-Bit 0 aufwärts untergebracht werden. Wie in Abb. 65-b zu sehen ist, verwendet das Kommando reset_Triggercells keine zusätzlichen Parameter, das Kommando set_Iterator benötigt hingegen einen Parameter zur Angabe eines neuen Iteratorwertes. Dieser beginnt bei Bit 0 und hat eine Länge von 8 Bit. Slave_ctrl kopiert diese Bits in den Iteratorspeicher, wobei Bitfolgen außerhalb des ConfRAM-Adressraums abgeschnitten werden. In diesem Beispiel hat der Parameter einen Wert von 1, weshalb der Iterator nach Ausführung des Befehls auf Speicherposition 1 des ConfRAM zeigt.

## Übertragung von Anzeigedaten

Der master_ctrl-Prozess steuert den I²C-Master zur Übertragung der Anzeigedaten aus dem I2CDisplayDataMemory an das Display. Der dazu implementierte Zustandsautomat wird in Abb. 66 vereinfacht dargestellt.

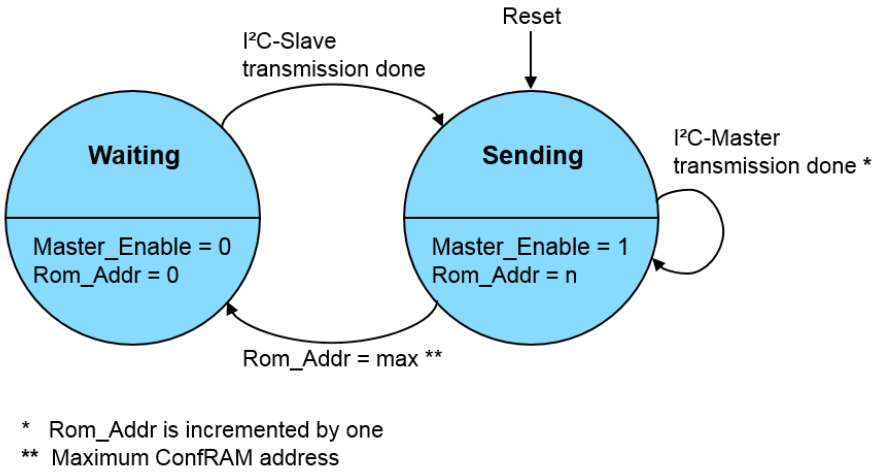


Abb. 66: Zustandsautomat zur Steuerung des I2CDisplayDataMemory

Nach dem Reset befindet sich der master_ctrl-Zustandsautomat im Zustand Sending. In diesem wird der I²C-Master über das Signal Master_Enable in den Sendemodus versetzt und überträgt das aus dem I2CDisplayDataMemory anliegende Datenwort an das Display. Die Adressierung eines Speicherwortes erfolgt über den jeweiligen Wert in Rom_Addr. Nach jeder vollständigen Übertragung eines Datenwortes, signalisiert der I²C-Master dies dem master_ctrl-Zustandsautomaten, welcher den Speicheriterator Rom_Addr daraufhin inkrementiert. Auf diese Weise wird der gesamte Inhalt des I2CDisplayDataMemory sequentiell abgerufen und vom I²C-Master an das Display übertragen. Ist die letzte Speicheradresse erreicht, wechselt der Zustandsautomat in den Zustand Waiting. In diesem ist der I²C-Master nicht enabled und der Speicheriterator wird in den Startzustand zurückversetzt. Nun wird solange gewartet, bis der I²C-Slave den Abschluss einer Übertragung signalisiert, bevor die Anzeigedaten erneut an das Display übertragen werden.

## Mapping von ConfRAM-Daten in den I2CDisplayDataMemory-Adressraum

Über die Schnittstelle des I2CDisplayDataMemory werden alle auf dem Display anzuzeigenden Daten abgerufen, welche sich aus konstanten Werten, wie beispielsweise Überschriften, sowie

veränderbaren Inhalten des ConfRAM zusammensetzen können. Der Speicher für konstante Werte innerhalb des I2CDisplayDataMemory ist als Read-Only-Speicher implementiert, sein Inhalt kann nicht verändert werden. Die Daten des ConfRAM, welche auf dem Display ausgegeben werden sollen, liegen an bestimmten Eingängen des I2CDisplayDataMemory an und werden mit vorher vereinbarten Stellen des Speichers verbunden. Durch dieses Mapping veränderlicher Eingangssignale in den Adressraum des Read-Only Speichers, ist es möglich alle Displaydaten über die selbe Schnittstelle abzufragen.

# I²C-Slave

Die I2CSlave-Komponente implementiert den Teil des I²C-Protokolls, welcher zur Beantwortung von Anfragen eines I²C-Masters notwendig ist und ermöglicht so eine Kommunikation mit dem Reader. Abb. 16 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

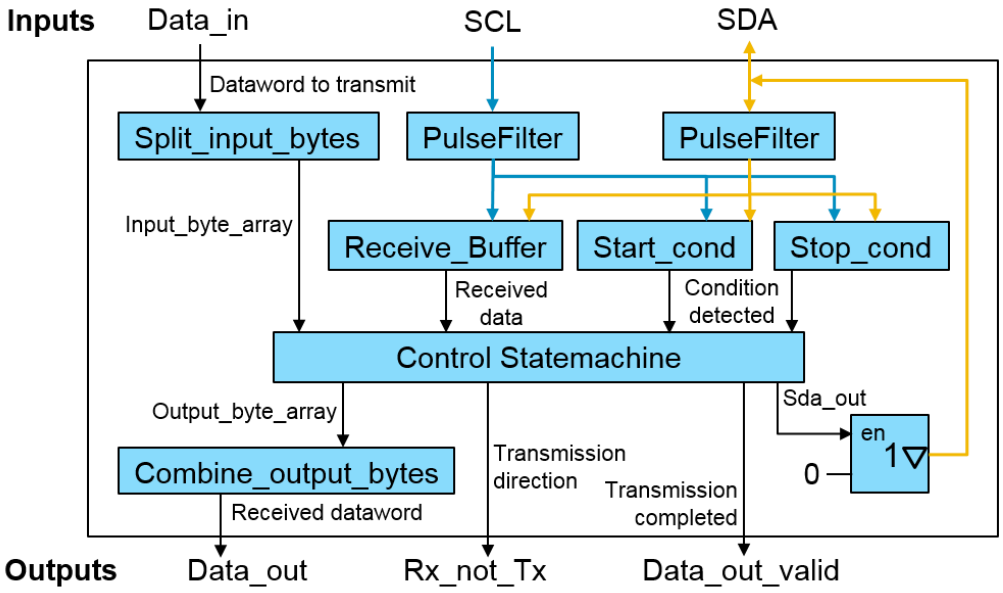


Abb. 67: I²C-Slave Blockdiagramm

## Ein-/ und Ausgänge

Die I2CSlave-Schaltung verfügt über die, in Abb. 67 dargestellten, Eingangssignalleitungen:

- **Data_in**: Das anliegende Datenwort wird auf Anfrage eines I²C-Masters zu diesem übertragen.
- **SCL**: Empfängt den vom I²C-Master generierten Kommunikationstakt.
- **SDA**: Datenleitung zur bidirektionalen Datenübertragung zwischen I²C-Slave und I²C-Master.

Des Weiteren existieren folgende Ausgänge:

- **Data_out**: Gibt empfangene Datenwörter aus.
- **Rx_not_Tx**: Zeigt Übertragungsrichtung an. Der I²C-Controller schreibt vom I²C-Slave ausgegebene Daten nur in den ConfRAM, wenn diese vom I²C-Master kommen, bzw. der I²C-Slave im Empfangsmodus ist.
- **Data_out_valid**: Zeigt an, wann ein Datenwort komplett übertragen wurde.

## Schaltung

Die I2CSlave-Komponente enthält folgende, in Abb. 67 dargestellte, Komponenten:

### Glättung der Signalleitungen SCL und SDA (PulseFilter):

Zwei PulseFilter-Komponenten dienen dazu die Signalleitungen SDA und SCL von Glitches zu befreien. Tests haben ergeben, dass der I²C-Master im Reader Glitches erzeugt, welche den I²C-Slave stören. Durch Einsatz der Pulsfilter wurde dieses Problem behoben.

## Erkennung einer Start-Condition (Start_cond):

Prozess zur Erkennung einer, vom I²C-Master empfangenen, Startbedingung. Diese wird im I²C-Protokoll vom I²C-Master gesendet, um eine Datenübertragung einzuleiten. Abb. 68 zeigt Beispielsignale, welche die Startbedingung erfüllen.


Abb. 68: I²C-Startbedingung

Wie in Abb. 68 dargestellt, entspricht eine I²C-Startbedingung einer fallenden Flanke auf der Datensignalleitung SDA, bei gleichzeitigem High-Zustand der Taktsignalleitung SCL.

## Erkennung einer Stop-Condition (Stop_cond):

Prozess zur Erkennung einer, vom I²C-Master empfangene, Stopbedingung. Diese wird verwendet, um eine Datenübertragung zu beenden. Abb. 69 zeigt Beispielsignale, welche die Stopbedingung erfüllen.


Abb. 69: I²C-Stopbedingung

Wie in Abb. 69 dargestellt, ist die I²C-Stopbedingung bei steigender Flanke der Datensignalleitung SDA, bei gleichzeitigem High-Zustand der Taktsignalleitung SCL erfüllt.

## Empfangsdatenspeicher (Receive_Buffer):

Schieberegister zur Zwischenspeicherung empfangener Datenbits. Dazu wird der am Eingang SDA anliegende Wert mit jeder steigenden Kommunikationstaktflanke, am Eingang SCL, in das Schieberegister geschoben.

## Steuerlogik (Control Statemachine):

Zustandsautomat zur Steuerung der Ausgänge des I²C-Slave. Dieser verarbeitet auftretende Start- und Stopbedingungen, führt eine Adressdekodierung durch und steuert den Empfang und Versand von Datenbits.

## Aufteilung Datenwort in Bytesequenz (Split_input_bytes):

Prozess zur Aufteilung des anliegenden Datenwortes in einzelne Bytes.

## Vereinigung Bytes zu Datenwort (Combine_output_bytes):

Prozess zur Zusammenfassung übertragener Bytes zu einem Datenwort.

Abb. 70 zeigt den Zustandsautomaten des I²C-Slave in vereinfachter Form.

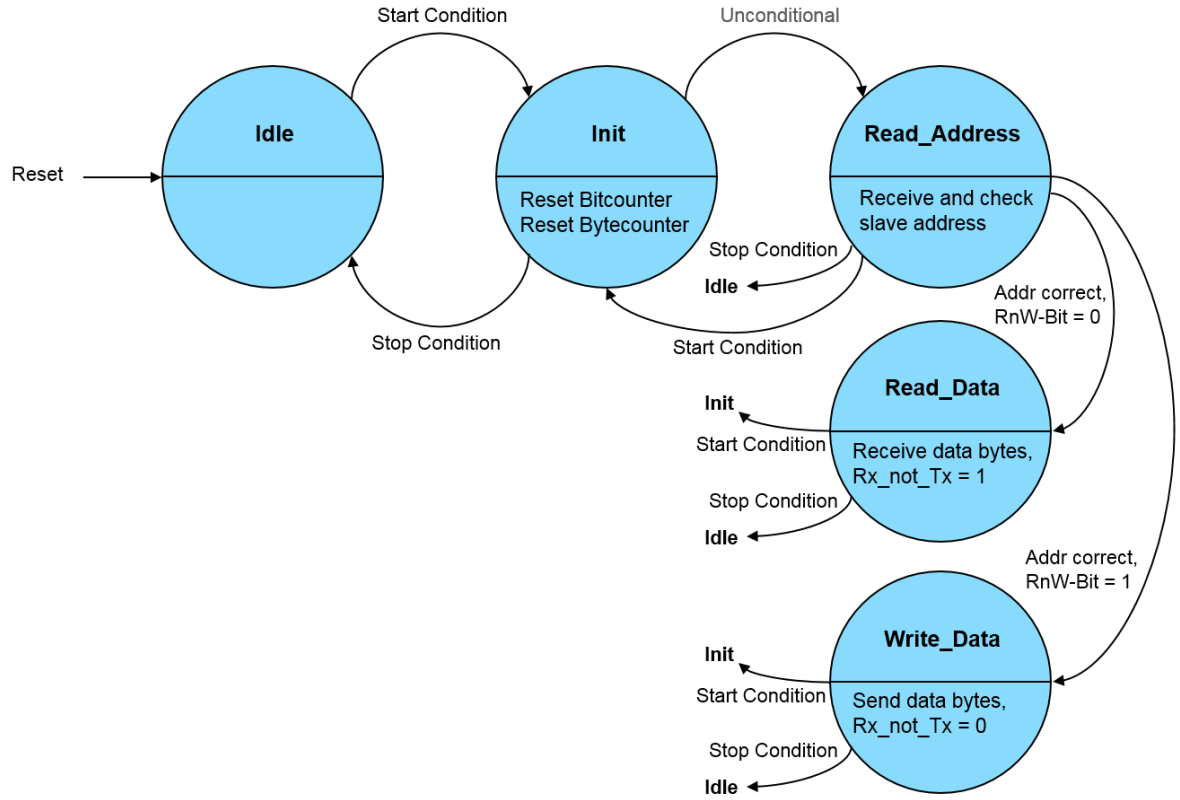


Abb. 70: Zustandsautomat zur Steuerung des I²C-Slave

Der in Abb. 70 dargestellte Zustandsautomat verfügt über folgende Zustände:

- **Idle**: Der Resetzustand, in dem auf das Eintreten der Startbedingung gewartet wird.
- **Init**: In diesem Zustand werden alle verwendeten Register in den Startzustand versetzt, u.a. Counter zur Zählung empfangener Bits und Bytes. Anschließend findet ein unbedingter Zustandsübergang nach Read_Address statt.
- **Read_Address**: In diesem Zustand werden die empfangenen Adressbits gezählt und sobald alle empfangen wurden, mit der Slave-Adresse verglichen. Stimmen beide überein, wird ein Acknowledge-Bit zur Bestätigung gesendet. Anschließend. entscheidet das übertragene ReadNotWrite-Bit über den Folgezustand. Hat dieses den Wert Null, wünscht der I²C-Master Daten zum I²C-Slave zu übermitteln. In diesem Fall findet ein Übergang in den Zustand Read_Data statt, in dem der I²C-Slave die vom I²C-Master kommenden Daten einliest. Wurde ein ReadNotWrite-Bit mit dem Wert Eins übertragen, fordert der I²C-Master Daten vom I²C-Slave an. In diesem Fall findet ein Übergang in den Zustand Write_Data statt, in dem der I²C-Slave Daten aussendet.
- **Read_Data**: In diesem Zustand werden die empfangenen Datenbits gezählt, bis ein Byte übertragen wurde. Daraufhin wird der Empfang des Bytes mit einem Acknowledge-Bit bestätigt und das empfangene Byte in einen Datenwort-Puffer kopiert. Nachdem alle vier Bytes eines Datenwortes empfangen wurden, teilt der I²C-Slave über einen High-Puls auf dem Ausgang Data_out_valid mit, dass das auf dem Ausgang Data_out gültig ist und gelesen werden kann. Das ausgegebene Datenwort ist nur für die Länge des Data_out_valid-Pulses gültig. Aus Ersparnisgründen wurde darauf verzichtet empfangene Datenwörter länger verfügbar zu machen. Der Ausgang Rx_not_Tx gibt ein High-Signal aus, solange sich der I²C-Slave in diesem Zustand befindet.
- **Write_Data**: In diesem Zustand werden die empfangenen Datenbits genauso gezählt und ausgegeben wie im Zustand Read_Data. Zeitgleich findet zudem eine Ausgabe der am Eingang Data_in anliegenden Datenwort-Bits auf dem Ausgang SDA statt. Das auf jedes übertragene

Datenbyte folgende Acknowledge-Bit des I²C-Master wird abgefragt, um so Acknowledge-Fehler feststellen zu können, welcher dazu führen würde, dass der I²C-Slave bis zum Ende der aktuellen Datenübertragung keine weiteren Ausgaben über den Ausgang SDA macht.

Die implementierte Schaltung liest die auf der Signalleitung SDA übertragenen Daten mit steigender Flanke des Kommunikationstaktes ein und führt daraus resultierende Berechnungen durch. Dazu zählen beispielsweise die Zählung empfangener Bits oder die Ermittlung des nächsten, auszugebenden Bits auf dem Ausgang SDA. Die Ausgabe des nächsten Daten- oder Acknowledge-Bits erfolgt anschließend mit der nächsten fallenden Flanke des Kommunikationstaktes. Auf diese Weise bleibt dem I²C-Master ein maximal großes Zeitfenster zum Auslesen der vom I²C-Slave ausgegebenen Daten. Abb. 71 stellt dies grafisch dar.


Abb. 71: Sende- und Empfangszeitpunkte des I2C-Slave

Innerhalb einer Datenübertragung berechnet der I²C-Slave bei der steigenden Taktflanke (t1) das nächste auszugebende Datenbit. Dessen Ausgabe beginnt mit der folgenden fallenden Taktflanke (t2) und endet bei der nächsten fallenden Taktflanke (t3). Somit kann der I²C-Master das auf der Signalleitung SDA liegende Datenbit innerhalb des Zeitraums t2 bis t3 auslesen.
Der I²C-Slave wurde für die Kommunikation mit dem Reader entworfen. Da dieser Daten mit einer Baudrate von maximal 300 kBaud/s überträgt, der implementierte I²C-Slave dieser Anforderung genügt und gemäß der Smartcard-Entwickler keine höheren Übertragungsraten in absehbarer Zeit erforderlich sein werden, wurde auf die Implementierung der Clock-Stretching Funktion verzichtet. Diese würde es dem I²C-Slave ermöglichen eine Datenübertragung durch den I²C-Master zu verlangsamen.

## Generische Attribute

Die I2CSlave-Komponente verwendet zwei generische Attribute:

- **Slave_Adresse**: Die Adresse mit welcher der I²C-Slave angesprochen werden kann.
- **Datenwort_Bytes**: Anzahl der Bytes eines Datenwortes. Das I²C-Protokoll überträgt Daten immer byteweise mit abschließendem Acknowledge-Bit. Mehrere Bytes werden zwischengespeichert, zu einem Datenwort zusammengesetzt und auf dem Ausgang Data_out ausgegeben. Zudem wird das Data_out_valid-Signal einmal pro empfangenem Datenwort ausgegeben. Durch Änderung dieses generischen Attributes werden alle Abhängigkeiten bei der Instantiierung entsprechend angepasst.

# HexToChar

Mithilfe der HexToChar-Komponente findet innerhalb des I²C-Controllers eine Umwandlung der, aus dem ConfRAM erhaltenen, Daten in ASCII-Zeichen statt. Jedes auf dem Display auszugebende Daten-Nibble (entspricht 4-Bit) wird von einer Instanz der HexToChar-Komponente taktasynchron in ein ASCII-Byte umgewandelt, welches dann an den Displaydatenspeicher (DisplayDataMemory) weitergereicht wird. Abb. 72 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

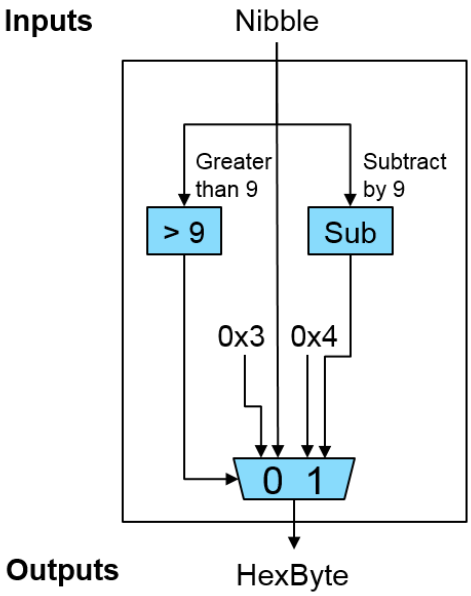


Abb. 72: HexToChar Blockdiagramm

## Ein- und Ausgänge

Die HexToChar-Schaltung verfügt über den Eingang Nibble zum Empfang der vier umzuwandelnden Bits und einen Ausgang HexByte zur Ausgabe des berechneten ASCII-Zeichens.

## Schaltung

Ziel der Umwandlung ist es für ein anliegendes Nibble aus dem Wertebereich 0 bis 9 die ASCII-Repräsentation der entsprechenden Zahl zurückzugeben und für Werte aus dem Bereich 10 bis 15 den ASCII-Wert eines Buchstabens aus dem Bereich A bis F. Abb. 73 gibt den relevanten Ausschnitt der ASCII-Tabelle (vgl. EA 2014, S. 12) wieder.

| Hex | Char | Hex | Char |
|---|---|---|---|
| 30 | ‘0‘ | | |
| 31 | ‘1‘ | 41 | ‘A‘ |
| 32 | ‘2‘ | 42 | ‘B‘ |
| 33 | ‘3‘ | 43 | ‘C‘ |
| 34 | ‘4‘ | 44 | ‘D‘ |
| 35 | ‘5‘ | 45 | ‘E‘ |
| 36 | ‘6‘ | 46 | ‘F‘ |
| 37 | ‘7‘ | | |
| 38 | ‘8‘ | | |
| 39 | ‘9‘ | | |

Abb. 73: Ausschnitt ASCII-Tabelle

Diesem Ausschnitt ist zu entnehmen, dass ein Nibble aus dem Bereich 0 bis 9 direkt auf die unteren vier Bit des Ausgabe-Bytes geleitet werden kann, während in den oberen vier Bit eine 3 stehen muss (siehe linker Teil der Tabelle).
Für ein Nibble aus dem Bereich 10 bis 15, wird von diesem die 9 subtrahiert bevor es ebenfalls an die unteren vier Bit des Ausgabe-Bytes weitergeleitet wird. Auf den oberen vier Bit wird in diesem Fall eine 4 ausgegeben.
Da diese ggf. viele Instanzen dieser Komponente benötigt werden (während des Funktionstests wurden beispielsweise 48 Instanzen benötigt, um zwölf 16-Bit-Werte auf dem Display auszugeben) wird der Hardwarebedarf reduziert, indem der größer-als-Neun Funktionsblock in Form einer boolschen Funktion realisiert wird.

| abcd | out | abcd | out |
|---|---|---|---|
| 0000 | 1 | 1000 | 1 |
| 0001 | 1 | 1001 | 1 |
| 0010 | 1 | 1010 | 0 |
| 0011 | 1 | 1011 | 0 |
| 0100 | 1 | 1100 | 0 |
| 0101 | 1 | 1101 | 0 |
| 0110 | 1 | 1110 | 0 |
| 0111 | 1 | 1111 | 0 |

Abb. 74: Bedingung für Nibbles ≤ 9

Die in Abb. 74 dargestellte Bedingung lässt zur Ermittlung, ob ein Nibble kleiner oder gleich Neun ist, lässt sich mithilfe der Boolschen Algebra vereinfachen, woraus sich folgende Bedingung ergibt:

$$Vereinfachte\ Bedingung\ (DNF): \sim b \sim c\ +\ \sim a$$

Code 9 zeigt schließlich die Umsetzung der Bedingung zur Zusammenstellung des Ausgabesignals HexByte.

```
HexByte <= X"3" & Nibble when b3 = '0' or (b1 = '0' and b2 = '0')
        else X"4" & std_logic_vector(to_unsigned(Nibble_int, 4));
```

Code 9: Berechnung des Ausgabesignals HexByte

Das in diesem Codeausschnitt als Nibble_int bezeichnete Signal überträgt den um 9 verringerten Wert vom Eingang Nibble.

# DisplayDataMemory

Die I2CDisplayDataMemory-Komponente speichert den, auf dem Display auszugebenden, Text und erlaubt zusätzlich das Einfügen externer Signale in den Adressbereich des Speichers. Abb. 75 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

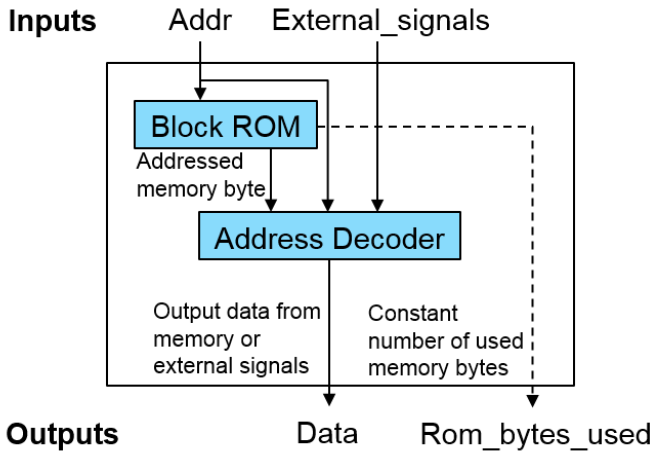


Abb. 75: I²C-DisplayDataMemory Blockdiagramm

## Ein- und Ausgänge

Die I2CDisplayDataMemory-Schaltung verfügt über die, in Abb. 75 dargestellten, Eingangssignalleitungen:

- **Addr**: Das anliegende Signal adressiert das Speicherwort, welches ausgegeben werden soll
- **External_signals**: Eine variable Anzahl externer Signalvektoren, welche alternativ zu Inhalten des Speichers ausgegeben werden können.

Des Weiteren existieren folgende Ausgänge:

- **Data**: Gibt das adressierte Speicherwort aus.
- **Rom_bytes_used**: Gibt die konstante Anzahl der verwendeten Speicheradressen aus.

## Schaltung

Die I2CDisplayDataMemory-Komponente enthält folgende, in Abb. 21 dargestellte, Komponenten:

### Displaydatenspeicher (Block ROM):

Dies ist der Speicher, welcher die an das Display zu übertragenden Daten enthält. Dieser wird als Array von konstanten Bytes angelegt und wird vom Synthesetool in Form eines vorinitialisierten und nicht überschreibbaren Block-RAM realisiert. Es wurde entschieden einen Block-RAMs zu diesem Zweck zu verwenden, um die übrigen Ressourcen des FPGA zu schonen. Für die Verwendung eines Block-RAM ist es notwendig auf die gespeicherten Daten ausschließlich taktsynchron zuzugreifen. Im Gegensatz zu der ConfRAM-Komponente, stellt das in diesem Fall jedoch kein Problem dar und wurde so implementiert. Der am Eingang Addr anliegende Adresswert führt somit zu einer taktsynchronen Ausgabe des im Block-RAM gespeicherten Datenbytes an den Adressdekoder.

## Adressdekoder

Der Adressdekoder wählt, je nach anliegendem Addr-Wert entweder das vom Block-RAM kommende Datenbyte oder ein externes Signal zur Ausgabe auf dem Ausgang Data aus. Code 10 zeigt ein Implementierungsbeispiel des Adressdekoders.

```
type ext_sig_map is array (0 to 1) of integer range 0 to 255;
constant sig_map : ext_sig_map := (20, 26);
                 :
data <=
        Ext_Sig1 when addr = sig_map(0) else
        Ext_Sig2 when addr = sig_map(1) else
        rom_data_s;
```

Code 10: Implementierung des Adressdekoders

Zunächst wird für jedes, in den Adressraum einzubindende, externe Signal ein Eintrag im Array sig_map angelegt, welcher der Adresse entspricht, unter der das externe Signal abgerufen werden können soll. In diesem Beispiel existieren die zwei externen Signale Ext_Sig1 und Ext_Sig2. Dem ersten Signal Ext_Sig1 ist der erste Eintrag des Arrays sig_map zugeordnet (hier 20), der zweite Eintrag dem Signal Ext_Sig2 (hier 26).
Anschließend findet eine Entscheidung für ein auf dem Ausgang data auszugebendes Signal statt. Liegt am Eingang Addr die Adresse 20 an, wird Ext_Sig1 ausgegeben, bei der Adresse 26 entsprechend Ext_Sig2. Für alle anderen Adressen wird das aus dem Block-RAM anliegende Datenbyte (hier rom_data_s genannt) ausgegeben.

## Generierung der Komponente durch Text2Vhdl.exe

Die Initialisierung des Block-RAM mit den auf dem Display auszugebenden, konstanten Zeichen ist mühsam und zeitaufwändig und wird dem Benutzer durch das eigens zu diesem Zweck entwickelte Programm Text2Vhdl abgenommen. Dieses liest eine Konfigurationsdatei ein und setzt Textbausteine entsprechend zu einer VHDL-Datei zusammen, welche dann die komplette Definition der I2CDisplayDataMemory-Komponente enthält. Die Konfigurationsdatei gibt dem Benutzer die Möglichkeit die auf dem Display auszugebenden Texte in leserlicher Form zu definieren, inklusive der Positionen für die Ausgabe externer Signale. Der Userguide erläutert die Verwendung im Detail (siehe Anhang A, S. 103).

# I²C-Master

Die I2CMaster-Komponente implementiert den Teil des I²C-Protokolls, welcher bei der Erzeugung von Anfragen an einen I²C-Slave zu berücksichtigen ist. Mithilfe eines Read-Requests können beliebig lange Folgen von Datenbytes von einem I²C-Slave angefordert und empfangen werden. Ein Write-Request ermöglicht es hingegen Folgen von Datenbytes zum I²C-Slave zu übertragen. Abb. 76 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

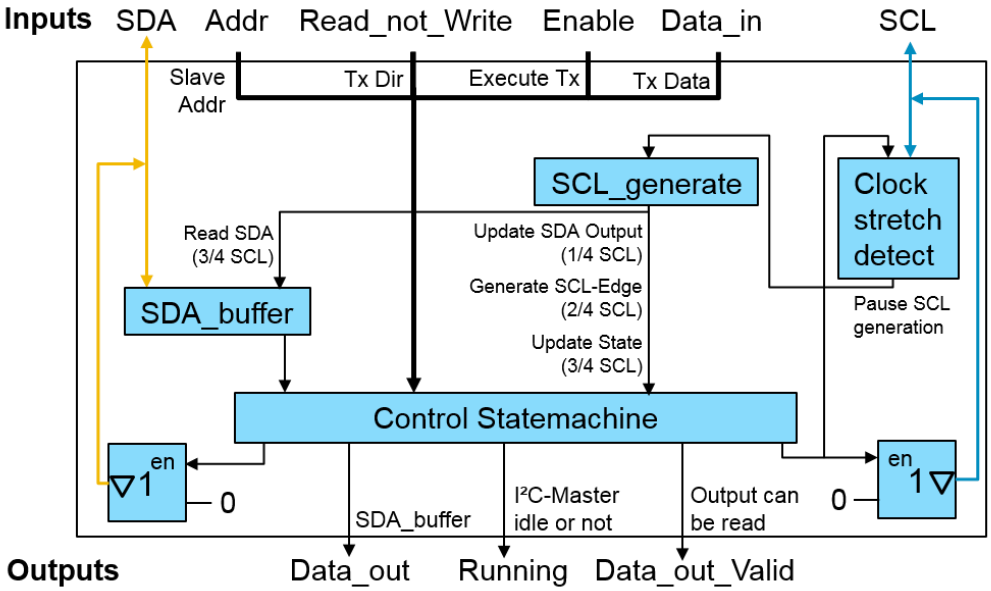


Abb. 76: I²C-Master Blockdiagramm

## Ein- und Ausgänge

Die I2CMaster-Schaltung verfügt über die, in Abb. 76 dargestellten, Eingangssignalleitungen:

- **SCL**: Bidirektionale Kommunikationstaktleitung. Der erzeugte Takt wird zum I²C-Slave hin ausgegeben. Vom I²C-Slave verursachtes Clock-Stretching wird erkannt und die Datenverarbeitung im I²C-Master entsprechend pausiert.
- **SDA**: Datenleitung zur bidirektionalen Datenübertragung zwischen I²C-Slave und I²C-Master.
- **Addr**: Adresse des I²C-Slave mit dem kommuniziert werden soll.
- **Read_not_Write**: Spezifiziert die Art der Anfrage, welche zum I²C-Slave gesendet wird. Es können Daten vom I²C-Slave angefordert oder zu diesem übertragen werden. Dieser Wert darf nur verändert werden, während keine Übertragung stattfindet. Eine Änderung der Übertragungsrichtung innerhalb einer Übertragung ist im I²C-Protokoll nicht vorgesehen.
- **Data_in**: Das anliegende Datenbyte wird zum I²C-Slave übertragen, sofern Read_not_Write den Sendemodus vorgibt. Im Empfangsmodus wird dieses Datenbyte ignoriert.
- **Enable**: Flag zum Starten und Fortführen einer Datenübertragung. Wenn gesetzt, wird eine Verbindung zum I²C-Slave aufgebaut und das Datenbyte Data_in zu diesem übertragen. Wenn danach immer noch gesetzt, wird direkt ein weiteres Byte übertragen, sonst wird die Kommunikation beendet.

Des Weiteren existieren folgende Ausgänge:

- **Data_out**: Zur Ausgabe des Buffers der SDA-Signalleitung, d.h. der letzten acht Bits, die auf dieser Signalleitung übertragen wurden.
- **Running**: Zeigt an, ob gerade eine Datenübertragung durchgeführt wird oder nicht.

- **Data_out_Valid**: Zeigt an, wann ein Datenbyte komplett übertragen wurde. Dann kann, je nach Kommunikationsrichtung, der Ausgang Data_out ausgelesen werden oder das am Eingang Data_in anliegende Datenbyte erneuert werden.

# Schaltung

Die I2CMaster-Komponente enthält folgende, in Abb. 76 dargestellte, Komponenten:

## Erzeugung des Kommunikationstaktes (SCL_generate):

Prozess, der durch Zählung der CPU-Takte drei Flags berechnet, welche anzeigen, wann eine bestimmte, zeitliche Position innerhalb einer Kommunikationstaktperiode erreicht wurde. Diese Flags werden zur Erzeugung des Kommunikationstaktes, zur Koordination des Einlesens und Ausgebens von Daten, sowie zum Auslösen von Zustandsübergängen des implementierten Zustandsautomaten verwendet. Abb. 77 stellt die Unterteilung der Kommunikationstakteperiode grafisch dar.

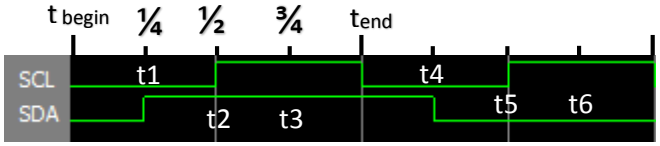


Abb. 77: Sende- und Empfangszeitpunkte des I²C-Master

$t_{begin}$ und $t_{end}$ in Abb. 77 markieren Perioden-Anfang und Ende. Dazwischen werden drei Flags im Abstand von jeweils einem Viertel der Taktperiode voneinander erzeugt:

- $^1/_4$ **SCL**: Das erste Flag wird nach dem ersten Viertel der Taktperiode erzeugt (t1 und t4) und gibt den Zeitpunkt für die Aktualisierung des Ausgabesignals auf der Signalleitung SDA vor. Der vorberechnete, zur Ausgabe anstehende, Wert wird ausgegeben.
- $^1/_2$ **SCL**: Das zweite Flag wird nach der halben Taktperiode (t2 und t5) generiert und zur Erzeugung der High-Flanke des Kommunikationstaktes auf SCL verwendet. Jeweils am Ende einer Taktperiode (tend) erfolgt die umgekehrte Flanke von High nach Low.
- $^3/_4$ **SCL**: Das dritte Flag wird nach dem dritten Viertel der Taktperiode erzeugt (t3 und t6) und löst zwei Aktionen aus. Das zu diesem Zeitpunkt auf der Signalleitung SDA übertragene Datenbit wird in den SDA_buffer eingelesen und außerdem der anstehende Zustandsübergang im implementierten Zustandsautomaten durchgeführt. Als direkte Folge des Zustandsübergangs werden sämtliche Abfragen und Berechnungen zum nächsten Folgezustand und zum nächsten Wert, der auf SDA ausgegeben werden soll ausgeführt. Auch die Ausgangssignale Running und Data_out_Valid werden zu diesem Zeitpunkt dem aktuellen Zustand entsprechend ausgegeben.

Meldet die Clock_stretch_detect-Komponente ein Clock-Stretching durch den I²C-Slave, wird das Zählen der CPU-Takte und damit die Flag-Erzeugung pausiert. Aufgrund der Abhängigkeiten von den Flags wird so der komplette I²C-Master pausiert.

## Datenspeicher (SDA_buffer):

Schieberegister, das jeweils zum Zeitpunkt einer dreiviertel Kommunikationstaktflanke den auf der Signalleitung SDA liegenden Wert speichert.

## Clock-Stretch-Erkennung (Clock stretch detect):

Prozess zur Ermittlung eines vom I²C-Slave erzeugten Clock-Stretchings. Ein solches liegt vor, wenn der I²C-Slave das Clock-Signal im Low-Zustand festhält, während der I²C-Master versucht es in den High-Zustand zu versetzen, indem er seinen Open-Collector-Ausgangstreiber auf Tri-State-Ausgabe

umschaltet. Es wird durch den Vergleich des vom I²C-Master ausgegebenen Signals mit dem auf der Signalleitung liegenden Signal ermittelt und dient dazu die SCL_generate-Komponente zu pausieren.

## Steuerlogik (Control Statemachine):

Zustandsautomat zur Steuerung der Ausgänge des I²C-Master. Dieser führt die Übertragung von Anfragen an einem I²C-Slave durch und wertet dessen Antwort aus.
Abb. 78 zeigt den Zustandsautomaten des I²C-Master in vereinfachter Form.

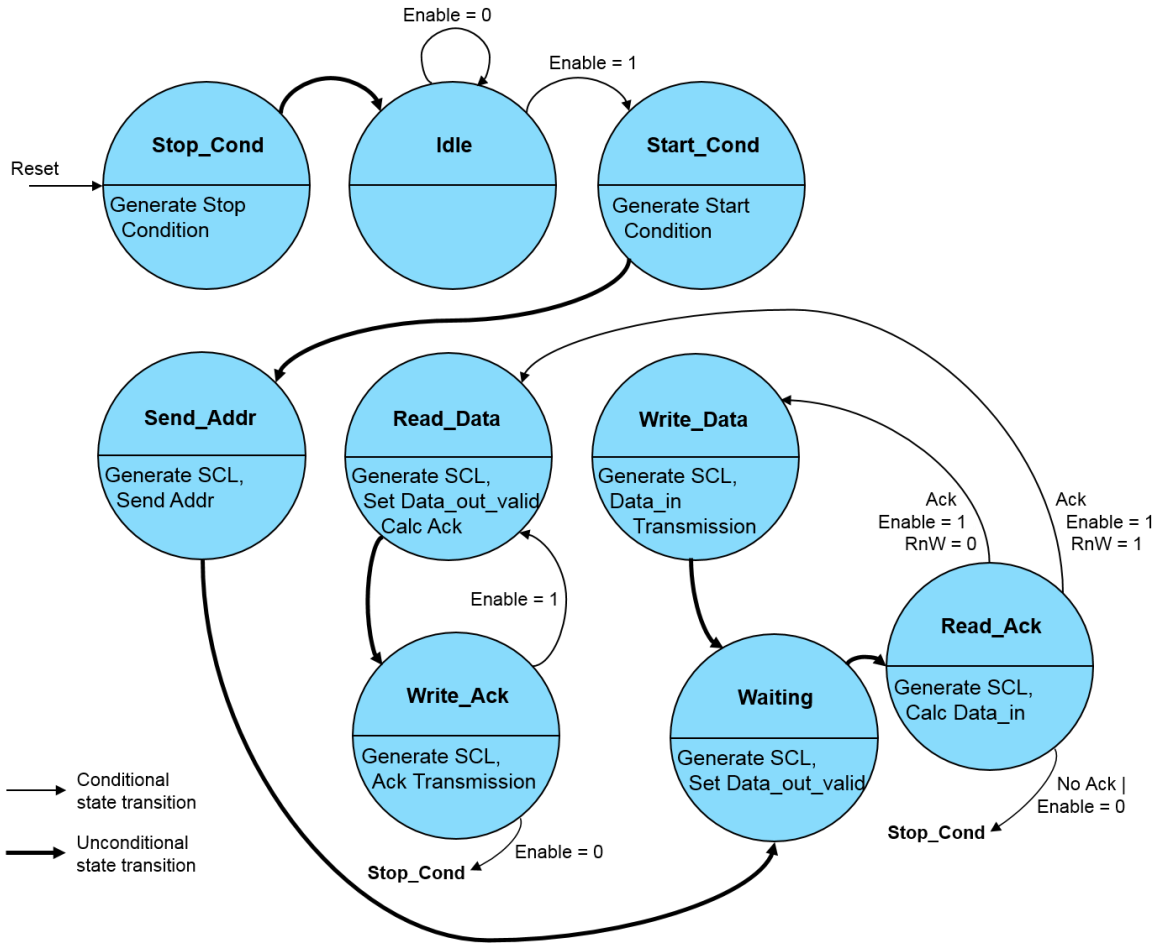


Abb. 78: Zustandsautomat zur Steuerung des I²C-Master

Der in Abb. 78 dargestellte Zustandsautomat verfügt über folgende Zustände:

- **Stop_Cond**: Dies ist der Resetzustand. In diesem wird die Ausgabe einer Stop-Condition vorbereitet, indem auf der Signalleitung SCL ein High-Signal und auf SDA ein Low-Signal ausgegeben werden. In dem unbedingt darauffolgenden Zustand Idle wird die Signalleitung SDA auf High gesetzt, während SCL High bleibt. Die entstehende Flanke auf der Signalleitung SDA entspricht der in Abb. 69 (siehe S. 117) dargestellten Stop-Condition.
- **Idle**: Im Idle-Zustand wird auf das Eintreffen eines Enable-Signals gewartet. Beide Signalleitungen SCL und SDA werden nicht beschrieben und sind damit im High-Zustand.
- **Start_Cond**: Erzeugt eine Start-Condition, indem die Signalleitung SCL im High-Zustand gelassen wird, während SDA auf Low gesetzt wird. Die entstehende Flanke entspricht der in Abb. 68 (siehe S. 117) dargestellten Start-Condition. Anschließend folgt ein unbedingter Übergang in den Zustand Send_Addr.
- **Send_Addr**: Überträgt die am Eingang Addr anliegende I²C-Slave-Adresse. Der von der SCL_generate-Komponente erzeugte Kommunikationstakt wird auf SCL ausgegeben, während

die am Eingang Addr anliegenden Adressbits nacheinander auf SDA ausgegeben werden. Anschließend folgt ein unbedingter Übergang in den Zustand Waiting.

- **Waiting**: Dieser Zustand dient dazu den High-Pulse auf der Signalleitung SDA zu erzeugen, welcher vom I²C-Slave auf Low gezogen wird, um ein Acknowledge-Bit zu senden. Dies geschieht nach der Übertragung der I²C-Slave-Adresse zu selbigem, sowie nach der Übertragung eines Datenbytes im Zustand Write_Data. Zusätzlich findet in diesem Zustand die Generierung des Data_out_Valid-Signals statt, welches anzeigt, dass der am Eingang Data_in anliegende Wert aktualisiert werden kann. Anschließend folgt ein unbedingter Übergang in den Zustand Read_Ack.
- **Read_Ack**: In diesem Zustand wird das vom I²C-Slave gesendete Acknowledge-Bit ausgelesen. Entsprechend dem Wert am Eingang Read_not_Write findet anschließend ein Übergang in den Zustand Read_Data oder Write_Data statt. Voraussetzung dafür ist ein empfangenes Acknowledge-Bit zur letzten Übertragung, sowie ein anliegendes Enable-Signal. Sollte kein Acknowledge-Bit empfangen worden sein oder kein Enable-Signal anliegen, findet ein Übergang in den Zustand Stop_Cond statt.
- **Write_Data**: Vergleichbar mit dem Zustand Send_Addr, wird auch in diesem Zustand ein Byte übertragen, jedoch nicht das Adress-Byte, sondern das am Eingang Data_in anliegende Datenbyte. Anschließend folgt wiederum der unbedingte Übergang in den Zustand Waiting.
- **Read_Data**: In diesem Zustand wird darauf gewartet, dass die zum Zustandsautomaten nebenläufig beschriebene SDA_buffer-Komponente ein komplettes Datenbyte empfangen hat. Am Ende findet zudem die Generierung des Data_out_Valid-Signals statt, welches anzeigt, dass das auf dem Ausgang Data_out ausgegebene Datenbyte dem empfangenen Datenbyte entspricht und ausgelesen werden kann. Außerdem wird die Ausgabe des Acknowledge-Bits ausgeführt. Anschließend findet ein Übergang in den Zustand Write_Ack statt.
- **Write_Ack**: In diesem Zustand wird die Ausgabe des Acknowledge-Bits beendet. Anschließend findet ein Übergang in den Zustand Read_Data statt, sofern ein Enable-Signal anliegt. Sollte kein Enable-Signal anliegen, folgt ein Übergang in den Zustand Stop_Cond.

## Generische Attribute

Für die Berechnung der Flags in der SCL_generate-Komponente, werden zwei Ausgangswerte benötigt: Die CPU-Taktfrequenz ClkFreq_Hz und die gewünschte Kommunikationstaktfrequenz SCLFreq_Hz. Aus diesen kann, während der Synthese, die Anzahl der CPU-Takte berechnet werden, welche einem Viertel der Kommunikationstaktperiode entsprechen.
Dazu wird die Formel $CPU\_Takte = (ClkFreq\_Hz/SCLFreq\_Hz)/4$ angewendet.

## Details zum Display

Wie eingangs erwähnt wurde, ist es möglich im ConfRAM enthaltene Werte zusammen mit den im I²C-DisplayDataMemory enthaltenen, konstanten Zeichen mithilfe des I²C-Master zu einem Display zu übertragen. Bei diesem handelt es sich um ein Display des Typs EAeDIP128W-6LW der Firma Electronic Assembly. Dieses wurde gewählt, da es von den Entwicklern des Readers ebenfalls eingesetzt wird. Es definiert eine Reihe von Steuerkommandos, sowie eine eigene Zeichentabelle, auf die hier jedoch nicht weiter eingegangen werden soll. Genauere Informationen finden sich im Datenblatt (siehe EA 2014).

# ConfRAM

Der ConfRAM dient dazu vom Benutzer übermittelte Konfigurationsparameter zu speichern. Diese werden über spezifische Ausgänge an die jeweiligen Komponenten weitergereicht. Außerdem findet hier eine Optimierung des Speicherbedarfs unter Benutzung generischer Attribute statt. Abb. 79 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

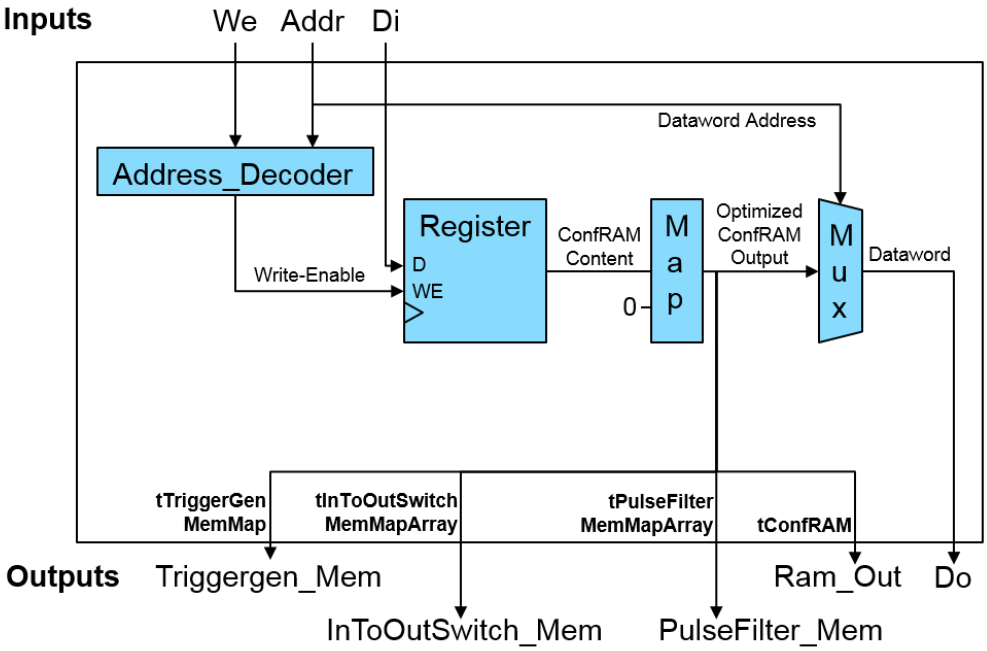


Abb. 79: ConfRAM Blockdiagramm

## Ein- und Ausgänge

Die ConfRAM-Schaltung verfügt über die, in Abb. 79 dargestellten, Eingangssignalleitungen:

- **Addr**: Adresse des zu beschreibenden Datenworts.
- **Di**: Datenwort, das in den ConfRAM geschrieben werden soll.
- **We**: Write-Enable zum Starten des CPU-Takt synchronen Schreibvorgangs.

Des Weiteren existieren folgende Ausgänge:

- **Do**: Gibt das über den Eingang Addr adressierte Datenwort aus. Dieser Ausgang wird vom I²C-Slave zum Auslesen des ConfRAM verwendet.
- **Triggergen_Mem**: Gibt die Konfigurationsparameter für die TriggerGenerator-Komponente aus. Die dafür benötigten Signalleitungen werden durch den Record-Typ tTriggerGenMemMap spezifiziert.
- **InToOutSwitch_Mem**: Gibt die Konfigurationsparameter für die InToOutSwitch-Komponente aus, spezifiziert durch den Record-Typ tInToOutSwitchMemMapArray.
- **PulseFilter_Mem**: Gibt die Konfigurationsparameter für die PulseFilter-Komponente aus, spezifiziert durch den Record-Typ tPulseFilterMemMapArray.
- **Ram_Out**: Gibt den gesamten Inhalt des ConfRAM aus, spezifiziert durch den Record-Typ tConfRAM. Es sei angemerkt, dass dieser Ausgang ausschließlich vom I²C-Controller für das Mapping von Inhalten des ConfRAM in den I²C-DisplayDataMemory verwendet wird. Es ist nicht möglich den Ausgang Do hierfür zu verwenden, da I²C-Master und I²C-Slave nebenläufig auf verschiedene Teile des ConfRAM zugreifen.

Erläuterungen der zur Ausgabe verwendeten Record-Typen folgen in den Abschnitten der jeweiligen Komponenten.

# Schaltung

Die ConfRAM-Komponente enthält folgende, in Abb. 79 dargestellte, Komponenten:

## Speicherwortadressierung (Address_Decoder):

Prozess zur Auswahl eines Datenwort-Registers als Speicherziel. Das anliegende Write-Enable-Signal wird an das adressierte Teilregister weitergeleitet.

## Der Speicher (Register):

Register, welches alle im ConfRAM gespeicherten Datenworte enthält. Diese werden CPU-Takt synchron gespeichert, jedoch zum Takt asynchron ausgegeben, damit sie nebenläufig, innerhalb verschiedener Komponenten, verarbeitet werden können. Im Gegensatz zu der I²C-DisplayDataMemory-Komponente, ist die Verwendung eines Block-RAM in diesem Fall nicht möglich. Denn dieser erlaubt kein asynchrones Auslesen von Daten. Aus diesem Grund wird bei der Synthese ein Distributed-RAM erzeugt, welcher einzelne Flip-Flops zur Speicherung verwendet. Dieser Umstand macht es möglich, aber auch notwendig Maßnahmen zur Reduzierung der Hardwarekosten zu ergreifen. Dazu markiert die im Folgenden erläuterte Map-Komponente nicht benötigte Speicherbits, bzw. Flip-Flops, so, dass diese vom Synthesetool als herausoptimierbar erkannt werden.

## Mux

Prozess zur Auswahl eines auszugebenden Datenwortes aus dem Register, das auf dem Ausgang Do ausgegeben wird.

## Map

Konstantes Mapping von Register-Bits und Null-Bits in den Ausgabeadressraum. Abhängig von generischen Attributen, werden individuelle Bereiche innerhalb jedes Datenwortes nicht ausgegeben, sondern durch die Ausgabe von Nullen ersetzt. Auf diese Weise wird das Wegoptimieren nicht benötigter Speicherbits während der Synthese erreicht. Abb. 80 zeigt die Vorgehensweise an einem Beispiel.

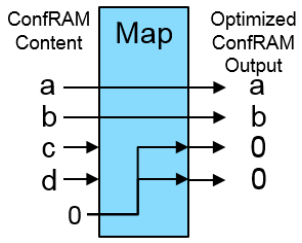


Abb. 80: Mapping benutzter und nicht benutzter Bits des ConfRAM

Von den vier beispielhaften, im ConfRAM vorhandenen Bits a bis d, werden nur die Bits a und b benötigt. Deshalb werden nur diese an die Ausgänge des ConfRAM durchgeleitet. Anstelle der nicht benötigten Bits c und d werden konstante Nullen ausgegeben.
Die Einsparung der Ausgabeleitungen, welche konstante Nullen ausgeben, ist an dieser Stelle nicht möglich, sondern erst innerhalb der datenverarbeitenden Komponenten. Bei der Synthese werden alle Signalleitungen, die eine konstante Null übertragen, wegoptimiert.
Der Benutzer schreibt und liest immer ganze Datenworte in und aus dem ConfRAM. Die bei der Synthese durchgeführten Datenwort-Verkleinerungen müssen bei Zugriffen auf den ConfRAM nicht berücksichtigt werden.

# Generische Attribute und Formatierung

In der ConfRAM-Komponente wird eine Optimierung der Speichergröße vorgenommen, indem jedes Datenwort in seiner Länge an die individuellen Erfordernisse angepasst wird. Außerdem werden die einzelnen Datenworte, den Ausgangssignaltypen entsprechend, gruppiert. Beides geschieht auf Basis generischer Attribute, die im Package DEFINITIONS_pack definiert sind.

## Formatierung des ConfRAM

Um die gespeicherten Datenworte abhängig von ihrer Adresse einem bestimmten Zweck zuordnen zu können, wurde ein Format zur Strukturierung des ConfRAM entwickelt. Abb. 81 zeigt die Aufteilung des Speichers in verschiedene Bereiche.

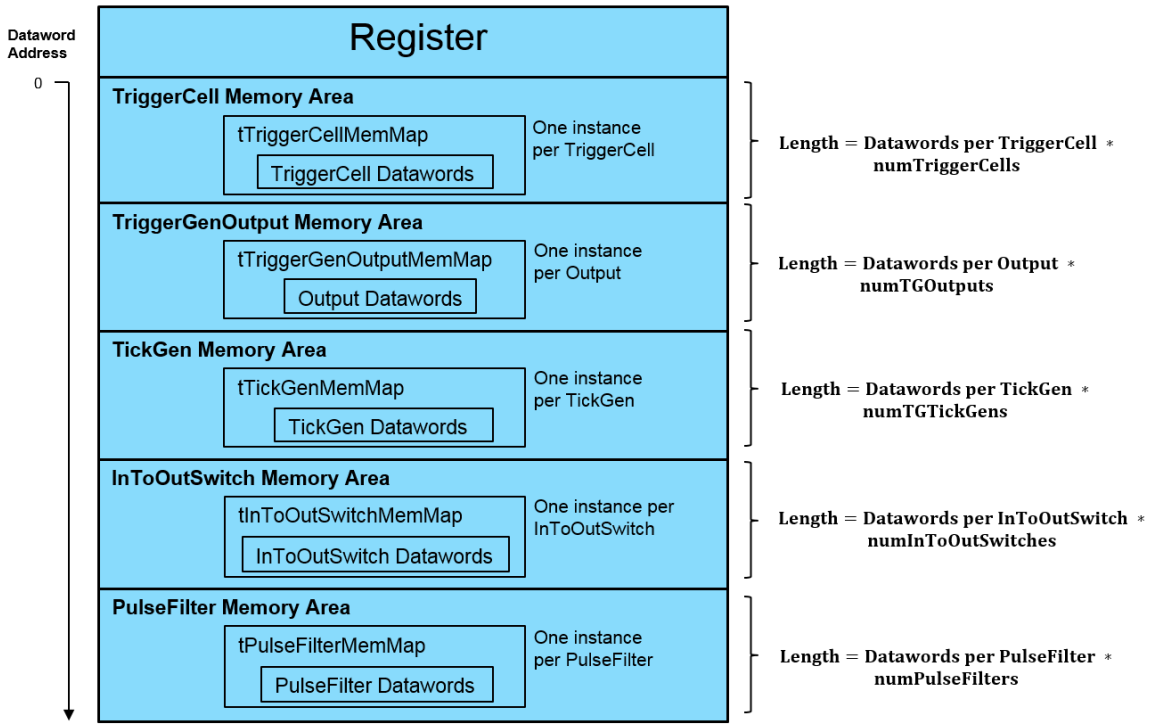


Abb. 81: ConfRAM Speicherformatierung

Für jede Art von Komponente, die Parameter vom ConfRAM bezieht, existiert ein eigener Speicherbereich (in Abb. 81 Memory Area genannt) im ConfRAM. Innerhalb eines Speicherbereichs liegt für jede Instanz der jeweiligen Komponente, eine Instanz eines Record-Typs zur Speicherung der benötigten Konfigurationsparameter. Die Größe eines Speicherbereichs wird mithilfe generischer Attribute, dazu zählen die Anzahl instantiierter Records und deren Länge, errechnet. Die erforderliche Gesamtgröße des ConfRAM, d.h. die Anzahl benötigter Datenwörter, entspricht der Summe aller Speicherbereiche. Der ConfRAM ist in folgende, in Abb. 81 dargestellte, Speicherbereiche unterteilt:

- **TriggerCell Memory Area**: Im ersten Speicherbereich des ConfRAM, beginnend bei Datenwortadresse 0, liegen Instanzen des Record-Typs tTriggerCellMemMap; eine pro instantiierter Triggerzelle. Abb. 82 stellt die darin enthaltenen Datenworte dar.

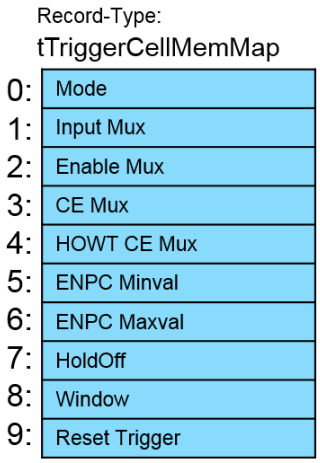


Abb. 82: Record-Type tTriggerCellMemMap

Die von diesem Record-Typ zusammengefassten Datenwörter werden zur Konfiguration einer Triggerzelle und deren direkter Umgebung verwendet. Wie in Abb. 82 zu erkennen ist, enthält der Record zehn Datenwörter. Die Größe des Speicherbereichs wird mithilfe des generischen Attributs numTriggerCells errechnet, welches der Anzahl instantiierter Triggerzellen entspricht und beläuft sich damit auf:

$$Länge = numTriggerCells * 10\ Datenwörter$$

- **TriggerGenOutput Memory Area**: Der zweite Speicherbereich enthält Instanzen des Record-Typs tTriggerGenOutputMemMap, welcher in Abb. 83 dargestellt wird.

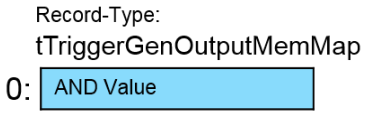


Abb. 83: Record-Type tTriggerGenOutputMemMap

Dieser Record umfasst ein Datenwort. Pro Ausgang der TriggerGenerator-Komponente wird eine Instanz des Typs benötigt, woraus sich eine Größe von:

$$Länge = numTGOutputs * 1\ Datenwort$$

für diesen Speicherbereich ergibt.

- **TickGen Memory Area**: Der dritte Speicherbereich enthält Instanzen des Record-Typs tTickGenMemMap, welcher in Abb. 84 dargestellt wird.

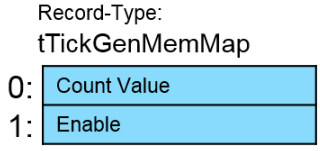


Abb. 84: Record-Type tTickGenMemMap

Dieser Record enthält Parameter zur Konfiguration der Tick-Generatoren. Pro in der TriggerGenerator-Komponente enthaltenem Tickgenerator werden zwei Datenwörter gespeichert, was eine Größe von:

$$Länge = numTGTickGens * 2\ Datenwörter$$

für diesen Speicherbereich ergibt.

- **InToOutSwitch Memory Area**: Der vierte Speicherbereich enthält Instanzen des Record-Typs tInToOutSwitchMemMap, welcher in Abb. 85 dargestellt wird.

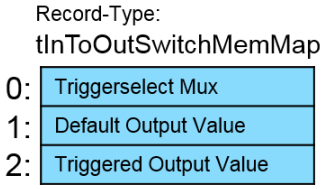


Abb. 85: Record-Type tInToOutSwitchMemMap

Für jede instantiierte InToOutSwitch-Komponente speichert eine Instanz, des in Abb. 31 dargestellten Record-Typs tInToOutSwitchMemMap, drei Datenwörter zur Konfiguration. Dadurch ergibt sich eine Speicherbereichgröße von:

$$Länge = numInToOutSwitches * 3\ Datenwörter$$

- **PulseFilter Memory Area**: Der fünfte und letzte Speicherbereich enthält Instanzen des Record-Typs tPulseFilterMemMap, welcher in Abb. 86 dargestellt wird.

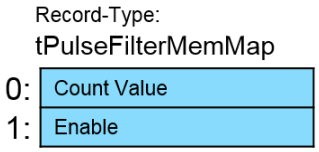


Abb. 86: Record-Type tPulseFilterMemMap

Für jede instantiierte PulseFilter-Komponente speichert eine Instanz von tPulseFilterMemMap zwei Datenwörter zur Konfiguration eines Pulsfilters. Die Größe des Speicherbereichs liegt somit bei:

$$Länge = numPulseFilters * 2\ Datenwörter$$

Da die Positionen der Konfigurationsparameter im ConfRAM nun bekannt sind, können die Gruppierungen der Ausgabetypen vorgenommen werden.

## Ausgabetypen

Der ConfRAM gibt Datensätze aus, welche von der Toplevel-Komponente an die entsprechenden Empfänger-Komponenten weitergeleitet werden.
Folgende, in Abb. 81 (siehe S. 130) genannte Typen werden zur Ausgabe verwendet:

- **tTriggerGenMemMap**: Die TriggerGenerator-Komponente verwendet alle in den ersten drei Speicherbereichen des ConfRAM liegenden Datenwörter. Deshalb wird ein Record-Typ zur Zusammenfassung von Arrays der drei Typen tTriggerCellMemMap, tTriggerGenOutputMemMap und tTickGenMemMap verwendet, wie Abb. 87 zeigt.

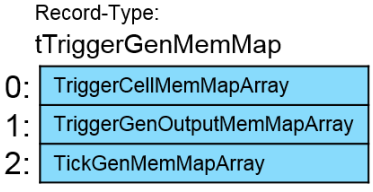


Abb. 87: Record-Type tTriggerGenMemMap

Die Größen der enthaltenen Arrays richten sich nach der Anzahl der Records in den jeweiligen Speicherbereichen des ConfRAM.

- **tInToOutSwitchMemMapArray**: Die im Speicherbereich InToOutSwitch Memory Area liegenden Instanzen des Record-Typs tInToOutSwitchMemMap werden in Form eines Arrays ausgegeben. Jede InToOutSwitch-Instanz empfängt ein Element dieses Arrays.

- **tPulseFilterMemMapArray**: Die im vierten Speicherbereich liegenden Instanzen von tPulseFilterMemMap werden ebenfalls in Form eines Arrays ausgegeben. Je ein Element dieses Arrays wird in der Toplevel-Komponente an jede PulseFilter-Instanz weitergeleitet.

- **tConfRAM**: Dieser Array-Typ umfasst den Speicher des ConfRAM in voller Länge.

Diese Ausgabetypen werden innerhalb des ConfRAM jedoch nicht direkt aus den Datenwörtern des Speichers zusammengesetzt. Zuvor werden nicht verwendete Speicherbits von der eingangs erwähnten Map-Komponente durch konstante Nullen ersetzt.

## Komponentendimensionierung

Die Anzahl der tatsächlich benötigten Bits pro Datenwort hängt im Einzelfall entweder von der Anzahl bestimmter, anderer Komponenten ab und wird aus dieser errechnet oder sie wird direkt vom Benutzer für jede Instanz einer Komponente individuell festgelegt.
Zur Definition durch den Benutzer existieren drei Record-Typen, mithilfe derer verschiedene Bitbreiten und andere konstante Parameter festgelegt werden:

- **tTcSpec**: Für jede instantiierte Triggerzelle innerhalb der TriggerGenerator-Komponente existiert eine Instanz des in Abb. 88 dargestellten Record-Typs tTcSpec.

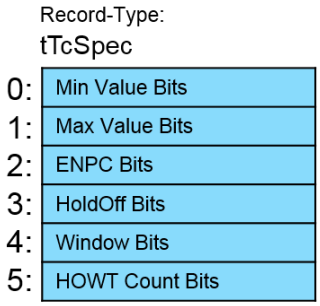


Abb. 88: Record-Type tTcSpec

Die enthaltenen Werte werden sowohl von der Map-Komponente zur Bestimmung nicht benötigter Bits im ConfRAM verwendet, als auch von der TriggerGenerator-Komponente bei der Instantiierung der Triggerzellen.

- **tTickGenSpec**: Pro Tickgenerator definiert der Benutzer es eine Instanz des Record-Typs tTickGenSpec aus Abb. 89.

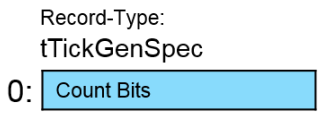


Abb. 89: Record-Type tTickGenSpec

Der enthaltene Wert wird ebenfalls sowohl bei der Speicheroptimierung durch die Map-Komponente, als auch bei der Instantiierung eines Tickgenerators, innerhalb der TriggerGenerator-Komponente, verwendet, um die Bitbreiten der aus dem ConfRAM empfangenen Parameter festzulegen .

- **tPulseFilterSpec**: Für jeden Pulsfilter existiert eine Instanz des Record-Typs tPulseFilterSpec aus Abb. 90.

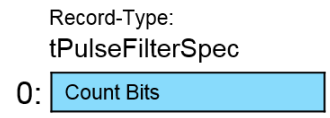


Abb. 90: Record-Type tPulseFilterSpec

Äquivalent zu tTickGenSpec, wird die Konstante Count Bits zur Dimensionierung des Zählers innerhalb eines Pulsfilters, sowie zur Festlegung der Bitanzahl, die aus dem ConfRAM empfangen wird.

# TriggerGenerator

Die TriggerGenerator-Komponente dient dazu benutzerdefinierte, mehrstufige Triggerbedingungen umzusetzen. Dazu werden die Eingangssignalleitungen durch Triggerzellen überwacht, welche mithilfe von Multiplexern zur Laufzeit verkettet werden können. Die so entstehenden Sequenzen von Triggerzellen können nebenläufige, mehrstufige Triggerbedingungen realisieren und entsprechende Triggersignale in variabler Länge generieren. Abb. 91 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

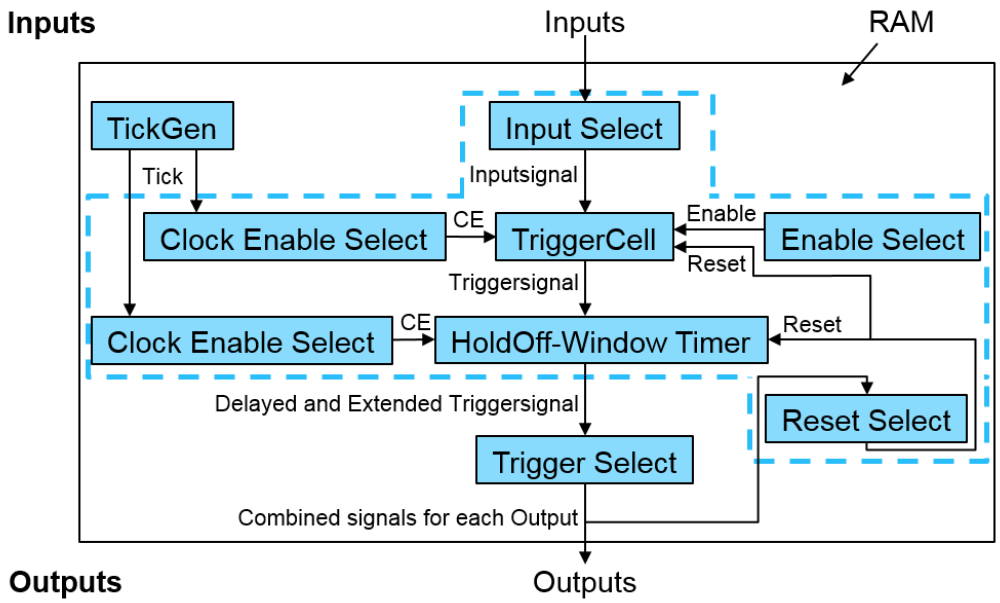


Abb. 91: TriggerGenerator Blockdiagramm

## Ein- und Ausgänge

Die TriggerGenerator-Schaltung verfügt über die, in Abb. 91 dargestellten, Eingangssignalleitungen:

- **Inputs**: Vektor von Eingangssignalleitungen, welche zur Generierung von Triggersignalen beobachtet werden. Die Anzahl der Signalleitungen wird über den generischen Parameter numTGInputs festgelegt (siehe Code 8, S. 110).
- **RAM**: Konfigurationsdaten aus dem ConfRAM. Die an diesem Eingang anliegenden Daten sind dem Record-Typ tTriggerGenMemMap entsprechend geordnet (siehe Abb. 87, S. 132). Jede der in Abb. 37 gezeigten Komponenten erhält die ihr zugedachten Konfigurationsparameter von diesem Eingang. Aus Gründen der Übersichtlichkeit wurden die entsprechenden Signalleitungen nicht eingezeichnet.

Des Weiteren existiert der Ausgang **Outputs**, welcher alle Signalleitungen zur Ausgabe von Triggersignalen umfasst. Deren Anzahl wird über den generischen Parameter numTGOutputs festgelegt (siehe Code 8, S. 110).

## Schaltung

Die TriggerGenerator-Komponente enthält folgende, in Abb. 91 dargestellte, Komponenten:

### Triggerberechnung (TriggerCell):

Die Triggerzellen-Komponente analysiert das anliegende Inputsignal und gibt ein daraus berechnetes Triggersignal aus. Außerdem empfängt sie folgende, weitere Signale:

- **Enable**: Bestimmt wann und wie lange die Triggerzelle arbeitet.
- **Reset**: Setzt die Triggerzelle in den Anfangszustand zurück.
- **CE** (Clock-Enable): Dient dazu die Arbeitsgeschwindigkeit interner Schaltungen zu steuern.

Des Weiteren liegen drei im ConfRAM gespeicherte Datenwörter an den Eingängen der Triggerzelle an:

- **Mode**: Bestimmt die Art der Triggerbedingung, welche durch die Triggerzelle realisiert wird.
- **ENPC Minval**: Untere Zählgrenze des internen Counters.
- **ENPC Maxval**: Obere Zählgrenze des internen Counters.

Diese drei Werte sind für jede Triggerzelle in der TriggerCell Memory Area des ConfRAM gespeichert und gelangen über den Eingang RAM der TriggerGenerator-Komponente zu den Eingängen der Triggerzelle.
Zusätzlich zu diesen zur Laufzeit änderbaren Parametern, werden drei konstante Parameter aus der tTcSpec-Instanz der Triggerzelle (siehe Abb. 88, S. 133) während der Instantiierung der Triggerzelle verwendet:

- **Min Value Bits**: Anzahl der verwendeten Bits des Datenworts ENPC Minval aus dem ConfRAM.
- **Max Value Bits**: Anzahl der verwendeten Bits des Datenworts ENPC Maxval aus dem ConfRAM.
- **ENPC Bits**: Bitbreite des internen Counters.

Diese drei Werte dienen dazu den Ressourcenbedarf der Schaltung zu reduzieren. Jede Triggerzelle enthält einen Zähler, der mit einer Datenwortbreite von bis zu 32 Bit arbeiten kann. Die verwendete Bitbreite kann für jede Triggerzelle individuell definiert werden. Dabei sollte darauf geachtet werden, dass die Zähler-Bitbreite ENPC Bits auf den gleichen Wert wie Max Value Bits gesetzt wird. Min Value Bits kann entweder auch auf den gleichen Wert gesetzt werden oder kleiner sein.
Aus Gründen der Übersichtlichkeit und Erweiterbarkeit wurden sämtliche Magic-Numbers im Quelltext durch die im Package DEFINITIONS_pack definierten Konstanten ersetzt. Dazu zählen auch die im Record tTcSpec enthaltenen Werte. Der Benutzer hat so die Möglichkeit an einer zentralen Stelle alle relevanten Bitbreiten nach belieben zu definieren, muss umgekehrt aber auch auf deren Sinnhaftigkeit achten. Auf dieses Beispiel bezogen könnten die Bitbreiten für Min Value und Max Value auch größer gewählt werden, als die Bitbreite des Zählers (ENPC Bits). Dies würde dazu führen, dass mehr Bits im ConfRAM gespeichert werden, als später, für den Vergleich mit dem Zähler, verwendet werden. Dieser Fehler könnte verwirrend für den Benutzer sein, da dieser große Grenzwerte im ConfRAM speichert und sie aus diesem auch wieder auslesen kann, der Zähler jedoch mit viel kleineren Grenzwerten arbeitet.
Die TriggerGenerator-Komponente erzeugt Triggerzellen, entsprechend der in numTriggerCells definierten Anzahl (siehe Code 8, S. 110), welche einen Wert von 1 bis 32 haben kann. Außerdem werden die zur Verknüpfung der Triggerzellen nötigen Signalleitungen und Schaltungsteile erzeugt. Für jede instantiierte Triggerzelle existiert ein Umfeld aus Instanzen der in Abb. 91 dargestellten Komponenten (siehe blau umrandeter Bereich), welche im Folgenden beschrieben werden.

## Auswahl eines Eingangssignals (Input Select):

Ein Multiplexer dient zur Auswahl eines Eingangssignals aus dem Inputs-Vektor, das von der Triggerzelle verarbeitet werden soll. Das Steuersignal dieses Multiplexers stammt aus dem zweiten Datenwort (Input Mux) des, zur Triggerzelle gehörenden, Speicherbereichs in der TriggerCell Memory Area im ConfRAM (siehe Abb. 82, S. 131).
Die Dimensionierung des Multiplexers geschieht auf Basis des generischen Parameters numTGInputs, welcher die Anzahl der Eingänge der TriggerGenerator-Komponente angibt. Diese können, den Requirements entsprechend, in einer Anzahl von 1 bis 16 Stück definiert werden (siehe „3.3 Ein-/ Ausgänge“).

Aus der Anzahl der Eingänge resultiert eine Mindestbitbreite für das Multiplexer-Steuersignal in Höhe des aufgerundeten Zweierlogarithmus der Eingangsanzahl. Verfügt die TriggerGenerator-Komponente beispielsweise über 10 Eingänge, d.h. der Inputs-Vektor hat eine Breite von 10 Bit, da der Benutzer numTGInputs mit einem Wert von 10 belegt hat, ergibt sich daraus eine Bitbreite für das Steuersignal in Höhe von:

$$Bitbreite: \quad numTCsInputsMuxCtrlBits = \lceil log_2\, numTGInputs \rceil = 4$$

Code 11 zeigt die Umsetzung der Logarithmusberechnung, unter Benutzung der IEEE Bibliothek math_real, in VHDL. Wie in der Vorbetrachtung erwähnt, werden generische Berechnungen wie diese vor der Synthese ausgeführt und müssen daher nicht synthetisierbar sein. Ebensowenig hat ihr Umfang Einfluss auf den Hardwarebedarf der Schaltung.

```
constant numTCsInputsMuxCtrlBits : integer := integer(ceil(log2(real(numTGInputs))));
```

Code 11: Berechnung der Steuersignalbitbreite

Die hier verwendete Bibliotheksfunktion „log2“ verlangt einen Parameter vom Typ „real“, weshalb die ganzzahlige Konstante numTGInputs entsprechend umgewandelt werden muss. Der berechnete Logarithmus wird anschließend mithilfe der Funktion „ceil“ aufgerundet und das Ergebnis in den ganzzahligen Typ integer umgewandelt und in der Konstanten numTCsInputMuxCtrlBits abgelegt. Diese enthält somit die Anzahl der Steuerbits der Eingangsmultiplexer der Triggerzellen.

Entsprechend der berechneten Bitbreite, in diesem Beispiel 4, werden dem Datenwort (Input Mux) im ConfRAM vier Bits entnommen, um später als Steuersignal zu dienen, wie in Code 12 zu sehen ist.

```
signal index_s : std_logic_vector(numTCsInputsMuxCtrlBits - 1 downto 0);
                                  :
                                  :
                                  :
index_s <= RAM.triggercell(i).input_mux(numTCsInputsMuxCtrlBits - 1 downto 0);
```

Code 12: Empfang des Multiplexer Steuersignals aus dem ConfRAM

Das 4-Bit breite Steuersignal index_s ermöglicht es mehr, als die vorhandenen 10 Eingänge zu unterscheiden, wie die Berechnung der Zweierpotenz von 4 ergibt:

$$Unterscheidbare\ Zustände = 2^{numTCsInputsMuxCtrlBits} = 2^4 = 16$$

Somit existieren 6 Steuersignalbelegungen, deren Verwendung ein undefiniertes Verhalten des Multiplexers bewirken würden. Um dies zu vermeiden, wird die Anzahl der Multiplexereingänge auf die Anzahl der adressierbaren Eingänge vergrößert. Abb. 92 stellt den vergrößerten Multiplexer in vereinfachter Form dar.

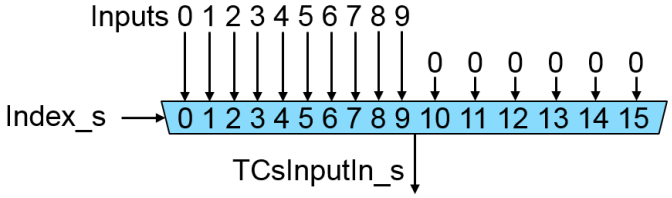


Abb. 92: Input-Multiplexer Blockdiagramm

Von den in diesem Beispiel adressierbaren 16 Eingängen sind die ersten 10 mit den Eingängen der TriggerGenerator-Komponente (Inputs-Vektor) verbunden, während bei Auswahl der übrigen 6 ein konstantes Low-Signal vom Multiplexer zum Eingang der Triggerzelle hin ausgegeben wird. Diese Vorgehensweise wurde so gewählt, da es sinnvoll erschien den Benutzer, im Falle einer

Fehlkonfiguration des ConfRAM, durch Beendigung der Funktion der Triggerzelle auf seinen Fehler hinzuweisen.
Zur Realisierung des Multiplexers in VHDL wird ein Vektor verwendet, dessen Länge der Zweierpotenz der ermittelten Steuersignalbitbreite entspricht. Code 13 zeigt dessen Definition und anschließende Verwendung als Multiplexer.

```
signal mux_s : std_logic_vector(2**numTCsInputsMuxCtrlBits - 1 downto 0) := (others => '0');
                                :
                                :
        mux_s(numTGInputs - 1 downto 0) <= inputs_s(numTGInputs - 1 downto 0);
                                :
                                :
        TCsInputIn_s(i) <= mux_s(to_integer(unsigned(index_s)));
```

Code 13: Input-Multiplexer - Definition und Adressierung

Die erste Zeile in Code 13 zeigt die Definition des Vektors mux_s und dessen Initialisierung mit konstanten Nullen. In der zweiten Zeile ist die Zuweisung der Eingangssignalleitungen zu den ersten 10 Vektorfeldern zu sehen, wobei die nicht verwendeten 6 Felder weiterhin mit Null belegt bleiben. Zeile drei zeigt schließlich die Anwendung des in Code 12 definierten Steuersignals index_s zur Auswahl eines Multiplexer-Eingangssignals, das dann auf dem Ausgang TCsInputIn_s zur Triggerzelle hin ausgegeben wird. Damit der Vektor index_s zur Indizierung des Vekors mux_s verwendet werden kann, muss er zuvor in den ganzzahligen Wert vom Typ integer umgewandelt werden. Da der enthaltene Indexwert vorzeichenlos gespeichert wurde, wird index_s zunächst in einen vorzeichenlosen Wert transformiert und anschließend mithilfe der Funktion to_integer in einen ganzzahligen Wert umgewandelt.

Die getroffenen Designentscheidungen haben zum Ziel dem Benutzer Skalierungen des Systems möglichst einfach zu machen. So führt eine Veränderung der Anzahl der Eingänge automatisch zur Anpassung der Anzahl von Steuerbits aus dem ConfRAM, die verwendet werden müssen. Die Benutzung eines Vektors als Multiplexer macht dies möglich, da dessen Größe bei der Synthese an die Anzahl der vorhandenen Eingänge angepasst werden kann.
Die Berechnung und Verwendung von ausschließlich benötigten Steuerbits aus dem ConfRAM begünstigt die Wegoptimierung der übrigen Bits des Datenwortes (Input Mux) bei der Synthese, wodurch Hardwareressourcen gespart werden können.
Die Vergrößerung des Multiplexers auf die nächst höhere Zweierpotenz der Steuersignalbitanzahl verhindert ein undefiniertes Verhalten bei Adressierung nicht vorhandener Eingänge.

## Triggerausgabe (HoldOff-Window Timer):

Jede Triggerzelle ist mit einer Instanz der HoldOffAndWindowTimer-Komponente verknüpft. Diese nimmt das von der Triggerzelle generierte Triggersignal entgegen und gibt es zeitlich verzögert und mit individuell definierbarer Länge aus.
Außerdem verfügt sie über folgende, weitere Eingänge:

- **Reset**: Setzt den HoldOffAndWindowTimer in den Anfangszustand zurück.
- **CE** (Clock-Enable): Dient dazu die Arbeitsgeschwindigkeit interner Schaltungen zu steuern.

Die verwendeten Konfigurationsdaten werden dem achten und neunten Datenwort (HoldOff und Window) der TriggerCell Memory Area (siehe Abb. 82, S. 131) entnommen:

- **HoldOff**: Gibt die Verzögerungszeitspanne der Triggerausgabe in CPU-Takten an.
- **Window**: Definiert die Anzahl der CPU-Takte, die eine Triggerausgabe andauert.

Da jeder HoldOffAndWindowTimer einer Triggerzelle fest zugewiesen ist, werden seine beiden Konfigurationsparameter im Speicherbereich der zugehörigen Triggerzelle abgelegt. Gleiches gilt für

drei konstante Parameter, welche in der tTcSpec-Instanz (siehe Abb. 88, S. 133) der jeweiligen Triggerzelle untergebracht sind:

- **HoldOff Bits**: Anzahl der verwendeten Bits aus dem Datenwort HoldOff.
- **Window Bits**: Anzahl der verwendeten Bits aus dem Datenwort Window.
- **HOWT Count Bits**: Bitbreite des internen Counters.

Äquivalent zu den konstanten Parametern der Triggerzellen, ermöglichen es diese drei Parameter Ressourcen zu sparen, indem die Bitbreite des im HoldOffAndWindowTimer enthaltenen Counters und die der im ConfRAM gespeicherten Konfigurationsparameter reduziert werden.

## Auswahl eines Ausgabesignals (Trigger Select):

Eine logische UND-Schaltung pro Ausgang der TriggerGenerator-Komponente dient dazu alle relevanten Triggersignale zu beobachten und daraus ein Triggersignal zu generieren, welches auf dem betreffenden Ausgang ausgegeben wird. Für jeden Ausgang existiert eine TriggerSelect-Schaltung, deren Gesamtanzahl demzufolge der Konstanten numTGOutputs (siehe Code 8, S. 110) entspricht. Abb. 93 stellt die Ausgangsbeschaltung vereinfacht dar.

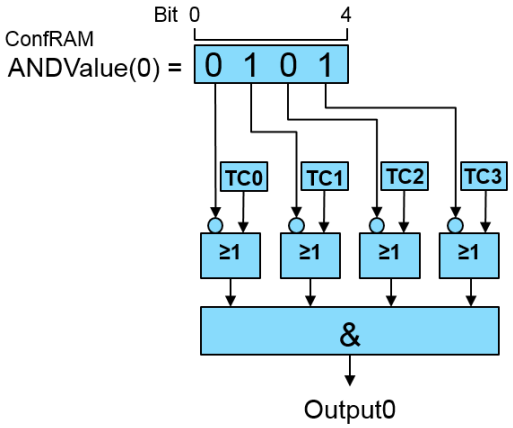


Abb. 93: Ausgangsbeschaltung der TriggerGenerator-Komponente

Die in Abb. 93 gezeigte Ausgangsbeschaltung erlaubt es zur Laufzeit festzulegen welche Triggerzellen gleichzeitig ein Triggersignal ausgeben müssen, damit eine Triggerbedingung erfüllt ist und ein Triggersignal auf einem Ausgang TriggerGenerator-Komponente ausgegeben wird.
Diese Möglichkeit wurde implementiert, da aus Gesprächen mit den Smartcard-Entwicklern deutlich wurde, dass zukünftige Anwendungsfälle eine nebenläufige Abarbeitung mehrstufiger Triggerbedingung zur Erzeugung eines einzigen Triggersignals erfordern. Abb. 94 veranschaulicht die Zusammenhänge.

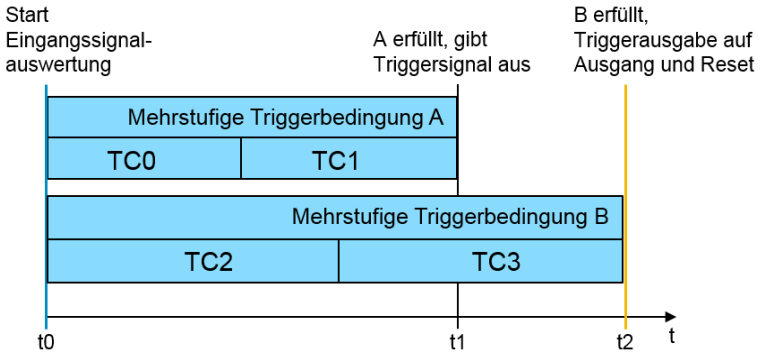


Abb. 94: Auswertung nebenläufiger, mehrstufiger Triggerbedingungen

Es sind Anwendungsfällen denkbar, in denen es nicht möglich ist alle zu einer mehrstufigen Triggerbedingung gehörenden Unterbedingungen innerhalb einer einzigen Sequenz abzuarbeiten. Beispielsweise im Falle nebenläufiger Auswertung verschiedener Eingangssignale oder wenn die Reihenfolge nicht vorhersagbar ist, in der die mehrstufigen Triggerbedingungen erfüllt werden. Deshalb müssen mehrere voneinander unabhängige, mehrstufige Triggerbedingungen zusammenarbeiten können, wie in Abb. 94 zu sehen ist.
Zum Zeitpunkt t0 werden alle Triggerzellen in ihren Anfangszustand versetzt. Zum Zeitpunkt t1 ist die erste mehrstufige Triggerbedingung erfüllt und beginnt mit der Ausgabe eines Triggersignals von unbegrenzter Länge. Zum Zeitpunkt t2 ist auch die zweite beteiligte mehrstufige Triggerbedingung erfüllt und beginnt ihrerseits mit der Ausgabe eines Triggersignals von unbegrenzter Länge. Da nun beide, für den Ausgang Output0 (siehe Abb. 93) relevanten, Triggerzellen ein Triggersignal ausgeben, beginnt die TriggerSelect-Schaltung mit der Ausgabe eines Triggersignals über den Ausgang Output0.
Abb. 93 zeigt die zu diesem Beispiel passende Konfiguration der TriggerSelect-Schaltung über den ConfRAM. Jeder TriggerSelect-Schaltung ist eine Datenwort ANDValue aus der TriggerGenOutput Memory Area des ConfRAM zugeordnet (siehe Abb. 83, S. 131). Jedes Bit steht für eine der bis zu 32 Triggerzellen, deren Anzahl durch die Konstante numTriggerCells (siehe Code 8, S. 110) festgelegt wird. Durch das Setzen eines Bits wird die jeweilige Triggerzelle für einen Ausgang relevant. In diesem Beispiel existieren vier Triggerzellen TC0 bis TC3, weshalb nur die ersten vier Bits des Datenworts ANDValue verwendet werden. Von diesen sind die beiden Bits 1 und 3 gesetzt. Das bedeutet, dass die beiden Triggerzellen TC1 und TC3 gleichzeitig ein Triggersignal ausgeben müssen, damit eines auf Output0 ausgegeben wird. Die Wahl fiel auf diese beiden Triggerzellen, da sie die jeweils letzte Unterbedingung der beiden in Abb. 94 gezeigten mehrstufigen Triggerbedingungen repräsentieren. Die Erfüllung der letzten Unterbedingung bedeutet die Erfüllung einer gesamten mehrstufigen Triggerbedingung.

## Erzeugung eines Resetsignals (Reset Select):

Resetleitungen wurden aus Gründen der Übersichtlichkeit in keinem Schaltungsbild der bisherigen Abschnitte berücksichtigt. Denn es wurde stets lediglich ein von außen kommendes Resetsignal an die Unterkomponenten durchgereicht. Abb. 91 macht diesbezüglich eine Ausnahme, da die ResetSelect-Schaltung ihrerseits Resetsignale erzeugt, welche innerhalb der TriggerGenerator-Komponente weitergereicht werden.
Jeder Triggerzelle ist eine ResetSelect-Schaltung zugeordnet. Dies ermöglicht es, für jede Triggerzelle individuell, ein Triggersignal von den Ausgängen der TriggerGenerator-Komponente festzulegen, welches einen Reset der Triggerzelle und deren HoldOffAndWindowTimer durchführt.
Abb. 95 stellt die Schaltung in Form eines vereinfachten Blockdiagramms dar.

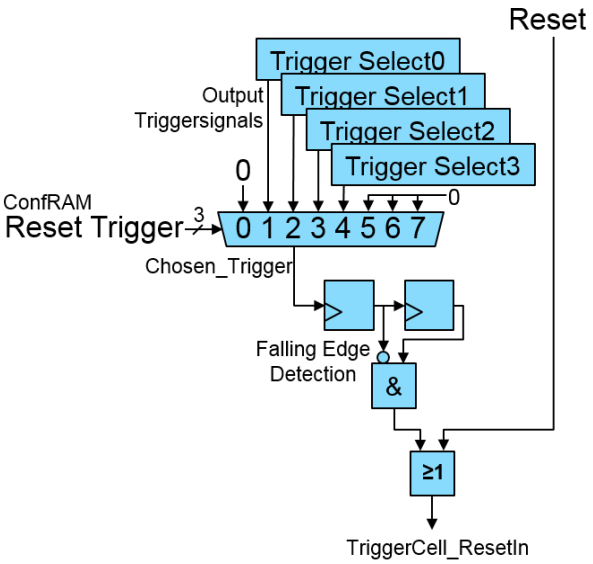

Abb. 95: Reset-Erzeugung für Triggerzelle und HOWTimer

In dem in Abb. 95 gezeigten Beispiel verfügt die TriggerGenerator-Komponente über vier Ausgänge (d.h. numTGOutputs wurde auf 4 gesetzt). Diese vier Triggersignalleitungen liegen an einem Multiplexer an, welcher zusätzlich über einen fünften Eingang verfügt an dem eine konstante Null anliegt. Datenwort Reset Trigger aus der TriggerCell Memory Area (siehe Abb. 82, S. 131) liegt zur Auswahl eines Triggersignals als Steuersignal am Multiplexer an. Eine fallende Flanke des ausgewählten Triggersignals führt zur Erzeugung eines Resetsignals. Deshalb wird das Triggersignal in ein zweistufiges Schieberegister geleitet und dessen Inhalt auf eine fallende Flanke hin überprüft. Ist eine solche aufgetreten oder ein vom Benutzer erzeugtes Resetsignal liegt an, wird ein Resetsignal zur Triggerzelle und dem HoldOffAndWindowTimer hin ausgegeben.
Die Erzeugung des Multiplexers findet nach dem selben Schema statt, wie in der InputSelect-Schaltung: Es wird der Zweierlogarithmus gebildet, um die Anzahl der Bits zu bestimmen, die dem Datenwort Reset Trigger aus dem ConfRAM entnommen und als Steuersignal verwendet werden.

$$Bitbreite: \quad numTCsResetMuxCtrlBits = \lceil log_2\, numTGOutputs + 1 \rceil = 3$$

Anschließend wird ein Multiplexer mit Eingängen in Anzahl der Zweierpotenz des soeben berechneten Logarithmus erzeugt.

$$Unterscheidbare\ Zustände = 2^{numTCsResetMuxCtrlBits} = 2^3 = 8$$

Die überschüssigen Eingänge werden wiederum mit einer konstanten Null belegt.
Somit werden nicht benötigte Speicherbits des ConfRAM eingespart und ein undefiniertes Verhalten des Multiplexers durch Auswahl nicht vorhandener Eingänge unterbunden.
Sollte die triggerbedingte Resetfunktion bei einer Triggerzelle nicht benötigt werden, kann sie deaktiviert werden. Durch Auswahl des Multiplexereingangs an dem die konstante Null liegt, ist die betreffende Triggerzelle unabhängig von allen erzeugten Triggersignalen. In diesem Fall bleibt es dem Benutzer überlassen alle Triggerzellen über das entsprechende I²C-Kommando zurückzusetzen. Dies findet insbesondere dann Anwendung, wenn Triggersignale mit unbegrenzter Länge ausgegeben werden. Nur bei Verwendung zeitlich begrenzter Triggersignale kann die Reset-Erzeugung verwendet werden, da diese wie beschrieben mit fallenden Triggerflanken arbeitet.
Die Verwendung eines Triggerausgangssignals als Resetsignal macht es ggf. notwendig mehr Triggerausgänge zu belegen, als für die reinen Triggerausgaben gebraucht würden: Wird eine Triggerzelle für die Erzeugung eines Resetsignals verwendet, deren Triggersignal ansonsten nicht gebraucht wird, muss dieses dennoch auf einem Triggerausgang ausgegeben werden. Dieser Kompromiss wurde gewählt, da er nur in seltenen Fällen (wenn viele Triggersignale benötigt werden) zu einem Nachteil wird. Grundsätzlich werden bei dieser Art der Verschaltung Ressourcen gespart, die nicht mehr zur Verfügung stünden, wenn die Ausgabesignale aller Triggerzellen auf den Resetmultiplexer-Eingang jeder anderen Triggerzelle gelegt würden.

## Erzeugung eines Clock-Enable Signals (TickGen):

Die TickGenerator-Komponente erzeugt ein CE (Clock-Enable)-Signal, welches von den Triggerzellen und HOWTimern verwendet werden kann, um deren Arbeitsgeschwindigkeit zu reduzieren. Durch deren Verwendung können Hardwareressourcen gespart werden, nämlich genau dann, wenn mehrere Triggerzellen oder HOWTimer mit relativ großen Zeitspannen arbeiten. Gleichzeitig vergrößert sich bei deren Verwendung die Ungenauigkeit, denn Zeitabstände werden dann nicht mehr als Anzahl von CPU-Takten angegeben, sondern als Anzahl von Ticks. Die Einbindung von TickGeneratoren wurde implementiert, um bei Knappheit von Hardwareressourcen die Möglichkeit zu haben Genauigkeit bei der Messung von Zeitabständen zugunsten gesparter Ressourcen zu verringern. Die Anzahl

instantiierter TickGeneratoren wird über die Konstante numTGTickGens (siehe Code 8, S. 110) festgelegt. Sofern keine TickGeneratoren benötigt werden, kann deren Anzahl auch auf 0 gesetzt werden.
An den Eingängen der TickGenerator-Komponente liegen zwei Konfigurationsdatenwörter, welche in der TickGen Memory Area (siehe Abb. 84, S. 131) des ConfRAM gespeichert sind:

- **Count Value**: Definiert die Zählobergrenze des internen Counters.
- **Enable**: Zur Aktivierung/Deaktivierung des internen Counters.

Zusätzlich zu diesen zur Laufzeit änderbaren Parametern, wird der konstante Parameter **Count Bits** aus der tTickGenSpec-Instanz des TickGenerators (siehe Abb. 89, S. 133) während dessen Instantiierung verwendet. Dieser definiert die Bitbreite des internen Counters und die Anzahl der verwendeten Bits aus dem Datenwort Count Value im ConfRAM.

## Auswahl eines Clock-Enable Signals (Clock Enable Select):

Multiplexer an den CE-Eingängen der Triggerzellen und HOWTimern dienen zur Auswahl eines TickSignals von einem der TickGeneratoren. Die Steuersignale dieser Multiplexer stammen aus dem vierten und fünften Datenwort (CE Mux und HOWT CE Mux) des, zur Triggerzelle gehörenden, Speicherbereichs in der TriggerCell Memory Area des ConfRAM (siehe Abb. 82, S. 131). Da für Triggerzelle und HOWTimer separate Multiplexer verwendet werden, können TickSignale individuell zugewiesen werden. Das vierte Datenwort (CE Mux) legt das von der Triggerzelle verwendete Signal fest und das fünfte Datenwort (HOWT CE Mux) das des zugehörigen HOWTimers.
Die Dimensionierung eines Multiplexers wird von der Anzahl vorhandener TickGeneratoren bestimmt. Aus dieser Zahl wird, wie auch beim Input Multiplexer (siehe Abb. 92, S. 137), der Logarithmus zur Basis 2 gebildet, um die Anzahl der benötigten Bits aus dem ConfRAM zu bestimmen. Deren Zweierpotenz entspricht der Anzahl der Multiplexereingänge.
Abb. 96 stellt den entstandenen Multiplexer in vereinfachter Form dar.

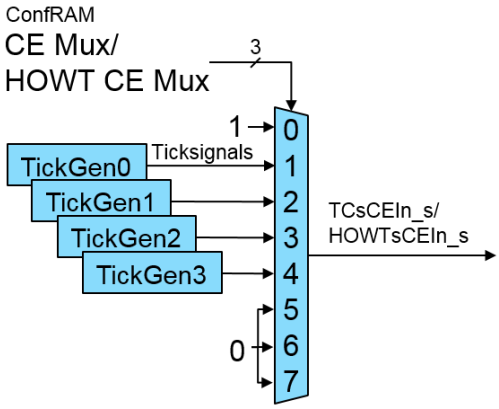


Abb. 96: CE-Multiplexer Blockdiagramm

In diesem Beispiel existieren vier TickGeneratoren. Ein zusätzlicher Eingang, an dem eine konstante Eins liegt, kann vom Benutzer selektiert werden, um die betreffende Triggerzelle oder den betreffenden HOWTimer im CPU-Takt, ohne Verwendung eines TickGenerators, zu betreiben.

## Verkettung von Triggerzellen (Enable Select):

Ein Multiplexer am Enable-Eingang einer Triggerzelle ermöglicht die Auswahl eines Triggersignals, welches vom HOWTimer einer anderen Triggerzelle ausgegeben wird. Dieses steuert damit wann und wie lange eine Triggerzelle arbeitet. Auf diese Weise ist es möglich Triggerzellen zu einer Sequenz zu verketten, um so eine mehrstufige Triggerbedingung zu realisieren.

Das Steuersignal eines Enable-Multiplexers befindet sich im dritten Datenwort (Enable Mux) des, zur Triggerzelle gehörenden, Speicherbereichs in der TriggerCell Memory Area des ConfRAM (siehe Abb. 82, S. 131). Die Dimensionierung des Multiplexers wird durch die Anzahl existierender Triggerzellen numTriggerCells (siehe Code 8, S. 110) bestimmt. Auch in diesem Fall wird der Zweierlogarithmus gebildet, um die Anzahl der benötigten Bits aus dem Datenwort (Enable Mux) zu bestimmen und dessen Zweierpotenz berechnet, um die Anzahl der Multiplexereingänge festzulegen.
Abb. 97 stellt die erzeugten Enable-Multiplexer innerhalb einer TriggerGenerator-Komponente, die vier Triggerzellen enthält, vereinfacht dar.

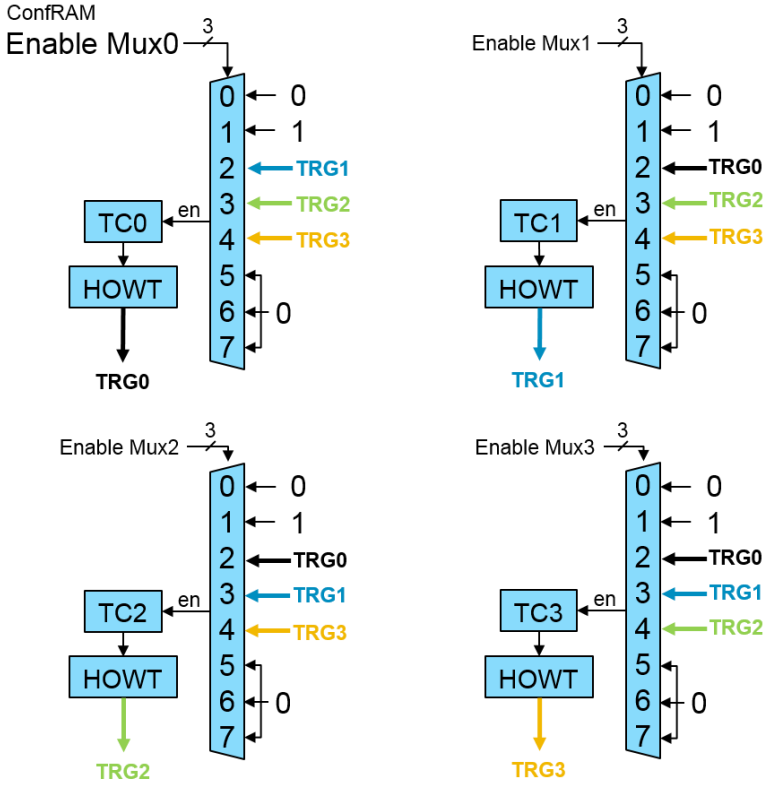


Abb. 97: Enable-Multiplexer Blockdiagramm

An den ersten beiden Eingängen eines Enable-Multiplexers liegen eine konstante Null und eine konstante Eins an, um die Triggerzelle für eine unbestimmte Zeitspanne unbedingt disablen bzw. enablen zu können. An den darauffolgenden Eingängen liegen die Triggersignalleitungen aller HOWTimer in aufsteigender Reihenfolge an. Eine Ausnahme bildet hier die Ausgabeleitung des jeweils eigenen HOWTimers. Denn eine Rückkopplung des Triggersignals von diesem, an den Enable-Eingang der zugehörigen Triggerzelle hätte keine Funktion, da sich eine disablete Triggerzelle nicht selbst enablen könnte.

## Reflektion

Zusammenfassend sei erwähnt, dass in diesem Abschnitt mehrfach besprochenen Logarithmusberechnungen zwingend notwendig sind, um den Speicher des ConfRAM optimieren zu können. Bezüglich der beschriebenen Multiplexer in der TriggerGenerator-Komponente, könnte ggf. auf eine Dimensionierung mithilfe der Zweierpotenzen verzichtet werden. Beispielsweise wäre es denkbar komplette 32-Bit Datenworte als Steuersignale zu verwenden und die Multiplexer mit einer einheitlichen, möglichst großen Anzahl an Eingängen auszustatten. Das verwendete Synthesetool XST der Firma Xilinx war in entsprechenden Testfällen in der Lage die überflüssigen Schaltungsteile selbstständig wegzuoptimieren. Um dies auch für zukünftige Versionen des Systems zu begünstigen und das Erreichen undefinierter Zustände der Multiplexer zu verhindern, wurden die Steuersignalbitbreiten und Multiplexergrößen dennoch individuell berechnet.

# Triggerzelle

Die TriggerCell-Komponente analysiert eine Signalleitung hinsichtlich der Signalverläufe, welche es gemäß der verschiedenen Unterbedingungsarten zu erkennen gilt. Abhängig von Art der Unterbedingung, die eine Triggerzelle umsetzt, wird die Erzeugung eines Triggersignals entweder von der Triggerzelle selbst übernommen oder an einen EdgesAndPulseClocks-Counter (kurz ENPC) delegiert.
Abb. 98 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

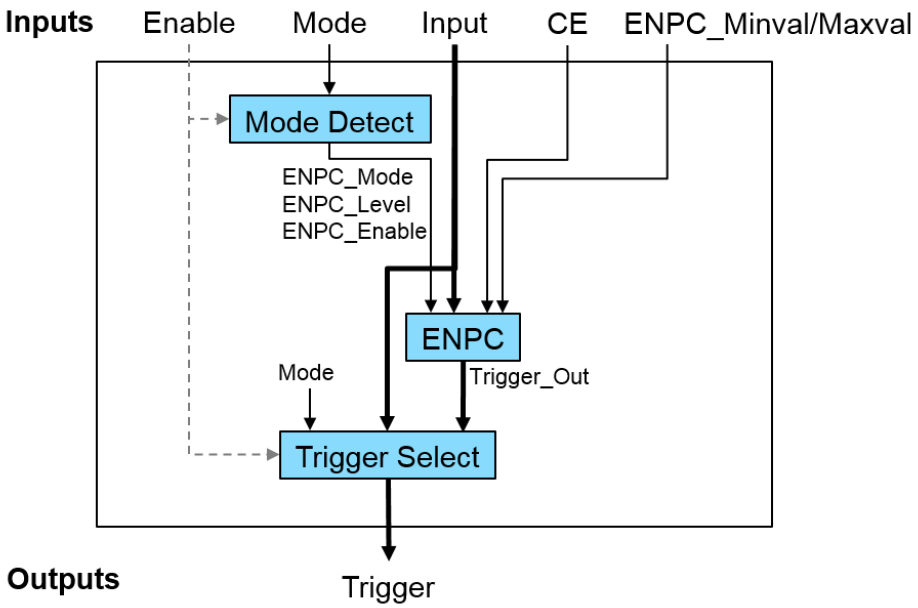


Abb. 98: TriggerCell Blockdiagramm

## Ein- und Ausgänge

Die TriggerCell-Schaltung verfügt über die, in Abb. 98 dargestellten, Eingangssignalleitungen:

- **Enable**: Zum enablen und Disablen der Triggerzelle. Eine disablete Triggerzelle gibt keine Triggersignale aus.
- **Mode**: Wählt den Arbeitsmodus der Triggerzelle aus. Jeder Modus entspricht einer Art von Unterbedingung.
- **Input**: Signal, das analysiert wird.
- **CE**: Clock-Enable zur Reduzierung der Arbeitsgeschwindigkeit.
- **ENPC_Minval/Maxval**: Grenzwerte, deren Bedeutung jeweils vom gewählten Modus abhängt.

Des Weiteren existiert der Ausgang **Trigger**, welcher das erzeugte Triggersignal ausgibt.

## Schaltung

Die TriggerCell-Komponente enthält folgende, in Abb. 98 dargestellte, Komponenten:

### Auswahl einer Methode zur Triggererzeugung (Mode Detect):

Dieser Prozess unterscheidet zwischen den 6 implementierten Betriebsmodi einer Triggerzelle und beschaltet die Eingänge der ENPC-Komponente in Abhängigkeit davon:

- **HighState** / **LowState**: In diesen beiden Modi ist die ENPC-Komponente Disabled. In den folgenden Modi ist sie enabled, sofern die Triggerzelle von außen enabled wird.

- **HighEdgeCount / LowEdgeCount**: Der ENPC_Mode wird auf Edge gesetzt und das ENPC_Level auf High (bei HighEdgeCount) und auf Low (bei LowEdgeCount).

- **HighPulseDetect / LowPulseDetect**: Der ENPC_Mode ist in diesem Fall Pulse und das ENPC_Level wiederum High (bei HighEdgeCount) oder Low (bei LowEdgeCount).

Die Betriebsmodi entsprechen den unterschiedlichen Arten von Unterbedingungen, wie sie in den Anforderungen gefordert sind.

### Flankenzählung und Zeitmessung (ENPC):

Die ENPC-Komponente dient dazu entweder Flanken auf der Input-Signalleitung oder CPU-Takte zu zählen. Abhängig von den Zählergrenzwerten ENPC_Minval und ENPC_Maxval generiert sie ein Triggersignal, welches dann von der Triggerzelle ausgegeben wird.
Bei der Instantiierung wird die Bitbreite des internen Zählers dimensioniert. Diese legt der Benutzer über die Konstante ENPC Bits in der zur Triggerzelle gehörenden Instanz von tTcSpec fest (siehe Abb. 88, S. 133).

### Auswahl einer Triggerquelle (Trigger Select):

Dieser Prozess unterscheidet zwischen den Betriebsmodi, um die Quelle des auszugebenden Triggersignals zu bestimmen:

- **HighState**: In diesem Modus wird ein Triggersignal erzeugt, wenn sich das Input-Signal im Zustand High befindet. Deshalb wird es direkt an den Ausgang durchgeleitet.

- **LowState**: Erzeugt ein Triggersignal, wenn sich das Input-Signal im Zustand Low befindet. Dazu wird das invertierte Input-Signal auf dem Ausgang ausgegeben.

- **HighEdgeCount, LowEdgeCount, HighPulseDetect, LowPulseDetect**: Die übrigen vier Betriebsmodi verwenden die ENPC-Komponente zur Triggererzeugung, weshalb in diesen Fällen deren Trigger_Out Ausgabesignal an den Ausgang weitergereicht wird.

Das ausgewählte Triggersignal wird jeweils nur ausgegeben, wenn die Triggerzelle enabled ist.

## Generische Attribute

Die Anzahl der benötigten Bits aus dem Datenwort Mode, des zur Triggerzelle gehörenden Speicherbereichs in der TriggerCell Memory Area (siehe Abb. 82, S. 131), wird aus der Anzahl implementierter Modi numModes (siehe Code 8, S. 110) durch Berechnung deren von Logarithmus zur Basis 2 bestimmt.

# Edges and Pulse Clocks Counter

Die ENPC-Komponente beobachtet eine Signalleitung, um ein Triggersignal nach Auftreten einer bestimmten Anzahl Flanken oder einem Puls von bestimmter Länge, zu erzeugen. Abb. 99 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

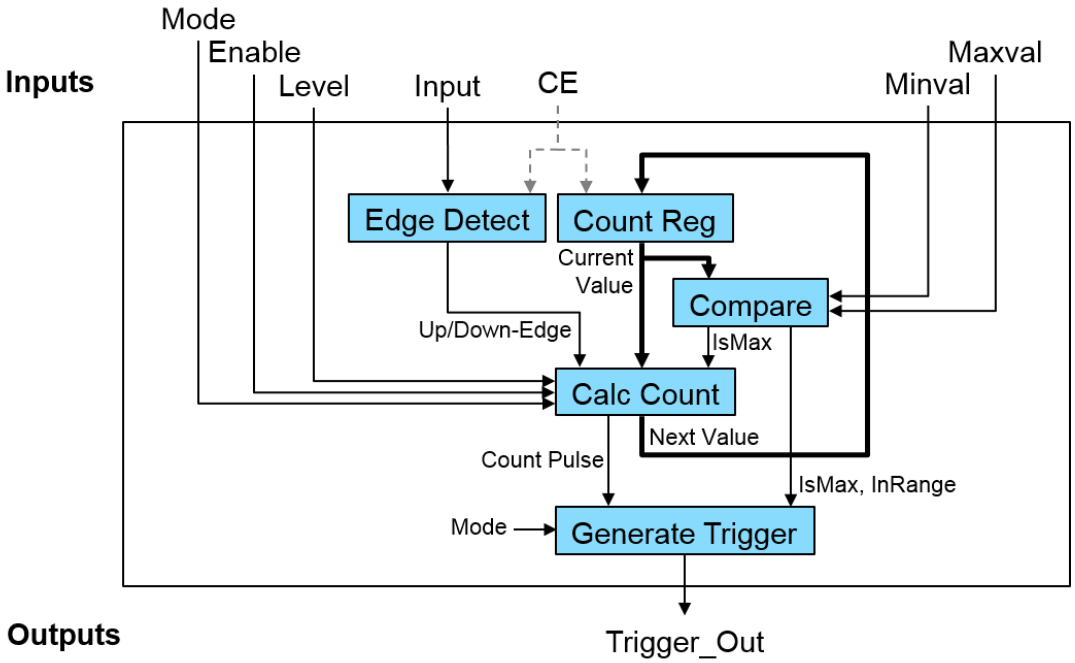


Abb. 99: ENPC Blockdiagramm

## Ein- und Ausgänge

Die ENPC-Schaltung verfügt über die, in Abb. 99 dargestellten, Eingangssignalleitungen:

- **Mode**: Auswahl des Betriebsmodus (Edge oder Pulse).
- **Enable**: Zum enablen und disablen des ENPC.
- **Level**: Definiert den Pegel der zu zählenden Flanken oder Pulse.
- **Input**: Zu analysierende Eingangssignalleitung.
- **CE**: Clock-Enable zur Reduzierung der Arbeitsgeschwindigkeit.
- **Minval**, **Maxval**: Zählergrenzwerte, deren Bedeutung vom gewählten Modus abhängt.

Des Weiteren existiert der Ausgang **Trigger_Out**, welcher das erzeugte Triggersignal ausgibt.

## Schaltung

Die ENPC-Komponente enthält folgende, in Abb. 99 dargestellte, Komponenten:

### Flankenerkennung (Edge Detect):

Dieser Prozess erkennt steigende und fallende Flanken des Input-Signals. Dazu wird ein 2-Bit breites Schieberegister zur Speicherung des Input-Signals verwendet. Dessen gespeicherte Werte können dann zur Erkennung einer Flanke miteinander verglichen werden.

### Zählerspeicher (Count Reg):

Register zur Speicherung des Zählerwertes. Dessen Bitbreite wird durch den generischen Parameter ENPC Bits festgelegt.

### Grenzwertvergleich (Compare):

Prozess zum Vergleichen des aktuellen Zählerwertes mit den Grenzwerten Minval und Maxval. Auf diese Weise werden die zwei Zustandssignale isMin und isMax ermittelt. Zur Berechnung des dritten Zustandssignals inRange wird ein Register verwendet. Um Hardwareressourcen zu sparen wurde in diesem Fall auf die Benutzung von größer-gleich und kleiner-gleich Vergleichsoperatoren verzichtet. Stattdessen wird der aktuelle inRange-Wert mit den ohnehin vorhandenen Signalen isMin und isMax verglichen.

### Zählersteuerung (Calc Count):

Dieser Prozess dient dazu den Zählerwert, abhängig vom gewählten Modus, zu aktualisieren. Es wird zwischen zwei Modi unterschieden:

- **Edge**: In diesem Modus werden Flanken gezählt. Wenn eine steigende Flanke aufgetreten ist und der Eingang Level ein High-Signal empfängt bzw. eine fallende Flanke erkannt wurde und an Level ein Low-Signal anliegt, wird der Zähler inkrementiert.
  Hat dieser den Wert von Maxval erreicht, wird er in den Startzustand zurückversetzt, um mit der Zählung von vorne zu beginnen.

- **Pulse**: In diesem Modus werden CPU-Takte gezählt, um die Dauer eines Pulses zu ermitteln. Wenn am Eingang Input der selbe Pegel anliegt, wie am Eingang Level, wird der Zähler mit jedem Takt inkrementiert. Unterscheiden sich die Pegel, wird er in den Startzustand zurückversetzt, um mit der Analyse des nächsten Pulses fortfahren zu können.

Die Beschriebene Zählerinkrementierung findet nur statt, wenn der ENPC enabled ist. Andernfalls wird der Zähler im Anfangszustand gehalten, um beim nächsten enable mit einer neuen Zählung beginnen zu können.

### Triggererzeugung (Generate Trigger):

Dieser Prozess erzeugt ein Triggersignal, wiederum abhängig vom gewählten Modus:

- **Edge**: Hat der Zähler den Wert von Maxval erreicht, wird ein Triggersignal ausgegeben. Der am Eingang Minval anliegende Wert hat in diesem Modus keine Funktion.
- **Pulse**: Befindet sich der Zähler zwischen den Grenzwerten in Minval und Maxval während der Puls, dessen Länge gemessen wurde, endet, wird ein Triggersignal ausgegeben.

## Generische Attribute

Bei der Instantiierung nimmt der ENPC das generische Attribut ENPC Bits entgegen, welches zur Dimensionierung des Zählerregisters verwendet wird. Auf die gleiche Bitbreite werden auch die Grenzwerteingänge Minval und Maxval gesetzt, welche für den Vergleich mit dem Zähler verwendet werden. Die Anzahl der für Minval und Maxval gespeicherten Bits im ConfRAM ist somit unabhängig von der Anzahl der Bits, die tatsächlich zum Vergleichen verwendet werden. Schon in der Triggerzelle werden die aus dem ConfRAM kommenden Werte zuerst bis zu einer Breite von 32 Bit mit Nullen aufgefüllt und anschließend auf die Zählerbitbreite des ENPC zugeschnitten.

# HoldOffAndWindowTimer

Die HOWTimer-Komponente nimmt einn Triggerflanke entgegen und erzeugt daraus ein Triggersignal mit Länge einer Window-Zeitspanne, welches nach einer optionalen HoldOff-Zeitspanne ausgegeben wird. Abb. 100 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

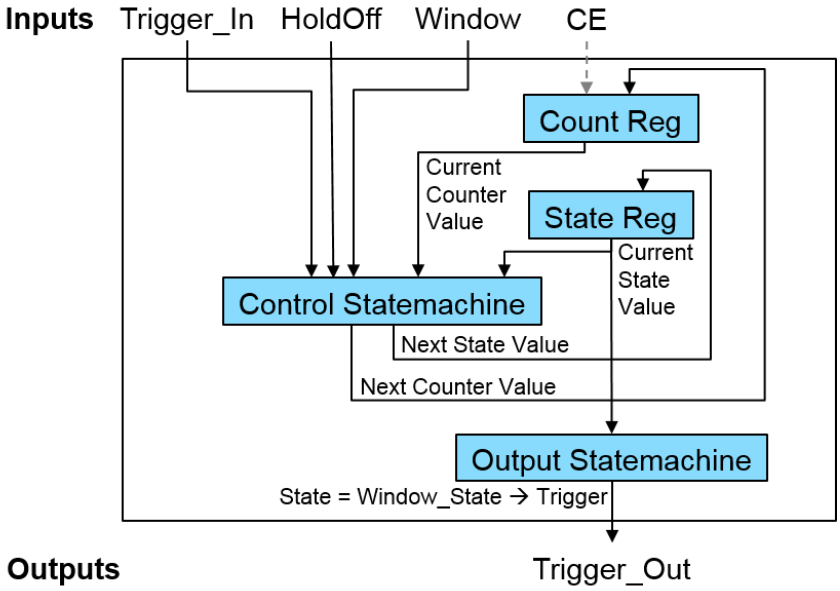


Abb. 100: HOWTimer Blockdiagramm

## Ein- und Ausgänge

Die HOWTimer-Schaltung verfügt über die, in Abb. 100 dargestellten, Eingangssignalleitungen:

- **Trigger_In**: Empfängt ein Triggersignal, das Verzögert und/oder verlängert werden soll.
- **HoldOff**: Zählergrenzwert zur Verzögerung der Triggerausgabe.
- **Window**: Zählergrenzwert für die Dauer der Triggerausgabe.
- **CE**: Clock-Enable zur Reduzierung der Arbeitsgeschwindigkeit.

Des Weiteren existiert der Ausgang **Trigger_Out**, welcher das erzeugte Triggersignal ausgibt.

## Schaltung

Die HOWTimer-Komponente enthält folgende, in Abb. 100 dargestellte, Komponenten:

### Zustandsspeicher (State Reg):

Register zur Speicherung des Zustands. Es existieren die Zustände: Idle, HoldOff, Window.
Die Dauer von Zustandsübergängen ist vom ggf. verwendeten Clock-Enable Signal unabhängig, um auch auf kürzeste Triggerpulse am Eingang Trigger_In reagieren zu können.

### Zählerspeicher (Count Reg):

Register zur Speicherung des Zählerwertes. Dieser dient dazu eine Zeitspanne in Form von CPU-Takten zu messen. Die Bitbreite des Zählers wird durch den generischen Parameter HOWT Count Bits festgelegt.
Die Geschwindigkeit des Zählers kann über ein Clock-Enable Signal verringert werden, um mit gleicher Zählerbitbreite größere Zeiträume abdecken zu können.

## Übergangsschaltnetz (Control Statemachine):

Zustandsautomat zur Steuerung einer Zähl-Sequenz in der der selbe Zähler zweimal hintereinander für unterschiedliche Zwecke eingesetzt wird. Zunächst wird die HoldOff-Zeitspanne abgewartet, in der noch kein Triggersignal ausgegeben wird. Dieses wird anschließend für die Länge der Window-Zeitspanne ausgegeben.
Abb. 101 zeigt den Zustandsautomaten des HOWTimers in vereinfachter Form.

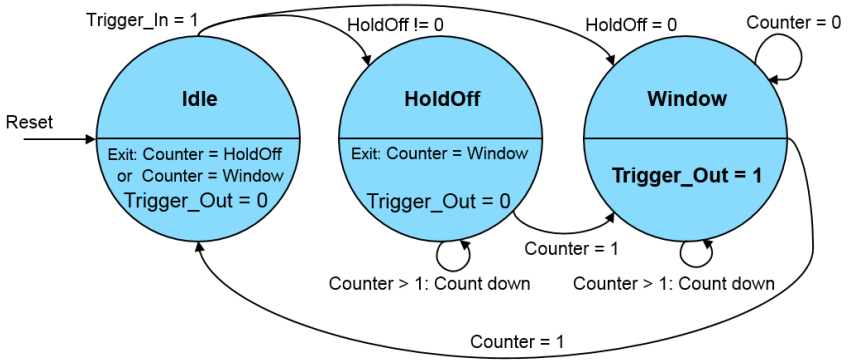


Abb. 101: Zustandsautomat zur Steuerung des HoldOffAndWindow Timers

Der in Abb. 101 dargestellte Zustandsautomat verfügt über folgende Zustände:

- **Idle**: Dies ist der Resetzustand. In diesem wird auf das Eintreffen eines Triggerpulses am Eingang Trigger_In gewartet. Liegt am Eingang HoldOff ein Wert größer Null an, wird dieser in den Zähler geladen und in den Zustand HoldOff gewechselt. Beträgt der HoldOff-Wert hingegen Null, wird der am Eingang Window anliegende Wert in den Zähler geladen und in den Zustand Window gewechselt.
- **HoldOff**: In diesem Zustand findet die zeitliche Verzögerung vor der Triggerausgabe statt. Dazu wird der Zähler bis zum Wert Eins im CPU-Takt heruntergezählt, ein ggf. vorhandenes Clock-Enable Signal verlängert die Dauer jedes Zählschrittes. Hat der Zähler den Wert Eins erreicht, wird der am Eingang Window anliegende Wert in den Zähler geladen und in den Zustand Window gewechselt. Da ein Zustandswechsel einen Takt dauert, findet der Zustandswechsel schon beim Zählerwert Eins statt und nicht erst bei Null.
- **Window**: In diesem Zustand wird das Triggersignal erzeugt. Zur Begrenzung der Ausgabedauer wird der Zähler bis zum Wert Eins heruntergezählt, bevor wieder in den Zustand Idle zurückgewechselt wird. Eine besondere Bedeutung hat in diesem Zustand der Zählerwert von Null. Dieser kann nur erreicht werden, indem am Eingang Window eine Null anliegt. In diesem Fall wird nicht weiter heruntergezählt und es findet kein Übergang in den Zustand Idle statt. Die Triggerausgabe ist somit von unbestimmter Länge. Nur ein Reset der Komponente beendet die Triggerausgabe.

## Ausgangsschaltnetz (Output Statemachine):

Das Ausgangsschaltnetz besteht lediglich aus einer Abfrage nach dem Zustand Window. Ist dieser erreicht, wird ein Triggersignal auf dem Ausgang Trigger_Out ausgegeben. Die Darstellung dieses Verhaltens wurde in Abb. 101 integriert.

# InToOutSwitch

Ein unidirektional durch den Triggergenerator geleitetes Signal kann mithilfe der InToOutSwitch-Komponente unterbrochen werden, wobei ein anliegendes Triggersignal entscheidet wann und wie lang diese Unterbrechung sein soll. Abb. 102 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

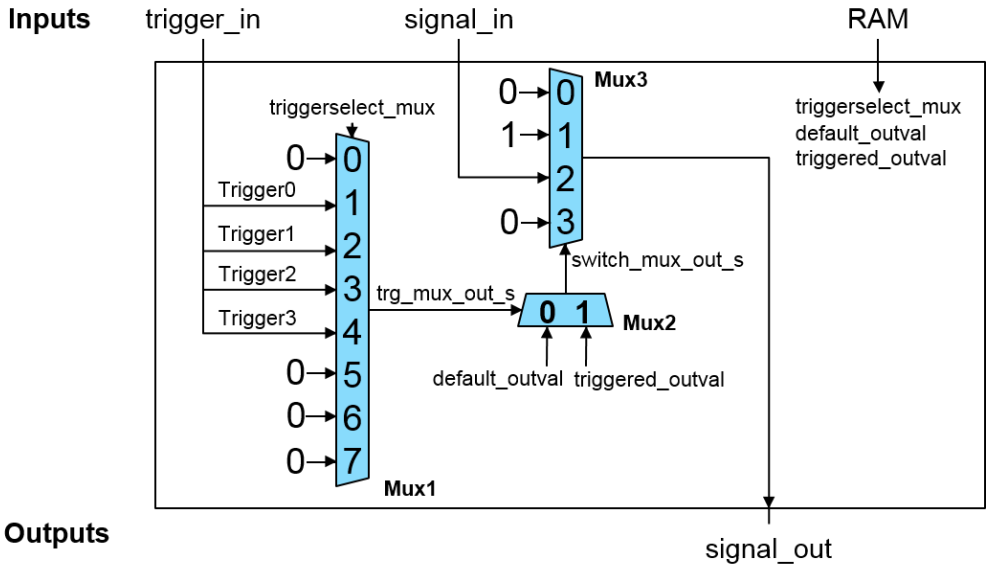


Abb. 102: InToOutSwitch Blockdiagramm

## Ein- und Ausgänge

Der InToOutSwitch verfügt über folgende, in Abb. 102 dargestellte, Eingangssignalleitungen:

- **signal_in**: Eingehendes Signal, das durchgeleitet und unterbrochen werden können soll.
- **trigger_in**: Vektor aller vom Triggergenerator erzeugten Triggersignale, von denen eines den Zeitpunkt zum Unterbrechen der Durchleitung anzeigen soll.
- **RAM**: Konfigurationsparameter vom Typ tInToOutSwitchMemMap. Dieser enthält die in Abb. 102 dargestellten Parameter triggerselect_mux, default_outval und triggered_outval.

Des weiteren verfügt der InToOutSwitch über die Ausgangssignalleitung signal_out zur Ausgabe des durchgeleiteten Signals oder eines Ersatzsignals, sollte die Durchleitung unterbrochen sein.

Die benötigten Konfigurationsparameter empfängt ein InToOutSwitch über einen Eingang vom Record-Typ tInToOutSwitchMemMap. Dieser enthält drei Parameter:

- **triggerselect_mux**: Multiplexersteuersignal für Mux1 zur Auswahl des Triggersignals, das eine Unterbrechung der Durchleitung auslöst.
- **default_outval**: Multiplexereingangssignal für Mux2 zur Auswahl des Ausgabesignals in nicht geschaltetem Zustand.
- **triggered_outval**: Multiplexereingangssignal für Mux2 zur Auswahl des Ausgabesignals in geschaltetem Zustand.

## Schaltung

Die Schaltung des InToOutSwitch enthält die drei in Abb. 102 dargestellten Multiplexer Mux1, Mux2 und Mux3.

Der Multiplexer Mux1 dient zur Auswahl des Triggersignals, das Zeitpunkt und Dauer der Umschaltung anzeigt. Deshalb liegen alle vom Triggergenerator erzeugten Triggersignale an seinen Eingängen an.
Zusätzlich liegt an einem Eingang eine konstante Null an, die ausgewählt werden kann, um es dem Benutzer zu ermöglichen die Umschaltfunktion eines vorhandenen, aber nicht benötigten InToOutSwitch zu deaktivieren. Dieser leitet, bei entsprechender Definition von default_outval, das Eingangssignal singal_in dann permanent an den Ausgang durch und ist damit transparent für die Datenübertragung, ohne dass eine Entfernung des InToOutSwitch aus dem Quellcode und eine erneute Synthese der Schaltung notwendig war. Durch Auswahl eines Triggersignals mit Mux1 kann die Umschaltfunktion jederzeit wieder verwendet werden.
Die Eingangs-Bitbreite von Mux1, sowie die Bitbreite seines Seteuersignals triggerselect_mux, hängen von der Anzahl der Ausgänge des Triggergenerators numTGOutputs (siehe Code 8, S. 110) ab und werden bei der Synthese aus dieser berechnet, wie in Code 14 zu sehen ist.

```
constant numITOSTrgSelMuxCtrlBits : integer := integer(ceil(log2(real(numTGOutputs+1))));
```

Code 14: Berechnung der Steuersignal-Bitbreite von Mux1

Die in numTGOutputs vom Benutzer festgelegte Anzahl von Ausgängen des Triggergenerators wird, zuzüglich einer weiteren Stelle für die konstante Null, zur Berechnung eines Zweierlogarithmus verwendet. Das Ergebnis wird anschließend aufgerundet, um so die ganzzahlige Anzahl von Bits zu erhalten, die zur Unterscheidung der Mux1-Eingangssignale mindestens gebraucht werden.
Da durch das Aufrunden eine Bitbreite entstehen kann, mit der mehr als nur die beabsichtigten Eingangssignale adressiert werden können, wird die soeben berechnete Steuersignal-Bitbreite zur Berechnung der Anzahl der Eingänge von Mux1 verwendet, wie Code 14 zeigt. So soll verhindert werden, dass die Auswahl nicht vorhandener Eingänge von Mux1 zu einem undefinierten Verhalten des Multiplexers führt.

```
signal trg_mux_s : std_logic_vector(2**numITOSTrgSelMuxCtrlBits - 1 downto 0) := (others => '0');
```

Code 15: Deklaration von Mux1

Code 15 zeigt die Deklaration eines Vektors, dessen Bitbreite der Zweierpotenz der in Code 14 berechneten Steuersignal-Bitbreite entspricht. Dieser Vektor wird anschließend mit den Eingangssignalen von trigger_in und einer konstanten Null beschrieben. Das gewünschte Eingangssignal wird über das Steuersignal triggerselect_mux ausgewählt, wie in Code 16 zu sehen ist.

```
trg_mux_s(numTGOutputs downto 0) <= trigger_in(numTGOutputs - 1 downto 0) & '0';
trg_mux_out_s <= trg_mux_s(to_integer(unsigned(triggerSelect_s)));
```

Code 16: Definition von Mux1 und Eingangsauswahl durch Steuersignal

Auf diese Weise wurde Mux1 in Form eines Vektors implementiert. Dadurch können die Anzahl seiner Eingänge und die Bitbreite des benötigten Steuersignals während der Synthese aus der Anzahl verfügbarer Triggersignale errechnet werden.
In dem in Abb. 102 (siehe S. 150) gezeigten Beispiel existieren vier Triggersignalleitungen, welche als Vektor trigger_in am InToOutSwitch anliegen. Zuzüglich eines Eingangs für die konstante Null, ergibt sich für Mux1, über den aufgerundeten Zweierlogarithmus, eine Steuersignal-Bitbreite von 3 Bits. Denn es werden mindestens drei Bit benötigt um fünf Eingangssignale unterscheiden zu können. Entsprechend der Zweierpotenz von drei, können mit diesem Signal tatsächlich acht Eingangssignale unterschieden werden. Deshalb wird zur Erzeugung von Mux1 ein Vektor mit acht adressierbaren Feldern erstellt, von denen die ersten fünf die Triggersignalleitungen und die konstante Null zugewiesen bekommen und die übrigen drei Felder ebenfalls eine konstante Null. Somit ist für jede mögliche Belegung des Steuersignals triggerselect_mux festgelegt, welches Signal von Mux1 ausgegeben wird.

Der Multiplexer Mux2 dient zur Unterscheidung zwischen dem Ausgabesignal für den ungeschalteten und für den geschalteten Zustand. Er hat eine feste Anzahl von zwei Eingängen, von denen einer über das ein Bit breite Steuersignal, welches von Mux1 kommt, ausgewählt wird. An den Eingängen liegen die, im RAM gespeicherten Werte default_outval und triggered_outval an, welche beide eine feste Größe von zwei Bit haben.

Der Multiplexer Mux3 dient zur Auswahl des über den Ausgang signal_out ausgegebenen Signals. Unterschieden werden kann zwischen einem konstanten Low-Pegel, einem konstanten High-Pegel und dem Eingangssignal signal_in. Da zur Unterscheidung dieser drei Signale ein zwei Bit breites Steuersignal benötigt wird, welches jedoch vier Eingänge unterscheiden kann, liegen an Mux3 zwei konstante Nullen an. So ist der Ausgabewert für jedes, von Mux2 kommende, Steuersignal definiert.
Beide Multiplexer Mux2 und Mux3 sind als with-select Auswahl implementiert.

## Anwendungsbeispiel

Das vom Reader zur Smartcard übertragene CLK-Signal (siehe „3.3 Signalunterbrechung“) soll durch das Triggersignal von Triggerzelle0 unterbrochen werden, und durch die Ausgabe eines konstanten Low-Pegels ersetzt werden.
Dazu wird signal_in mit dem vom Reader kommenden CLK-Signal beschaltet. default_outval wird auf 2 gesetzt und triggered_outval wird auf 0 gesetzt. Dies bedeutet, dass im ungeschalteten Zustand signal_in auf signal_out ausgegeben wird und im geschalteten Zustand ein Low-Pegel ausgegeben wird. Außerdem wird triggerselect_mux auf 1 gesetzt, um das Triggersignal von Triggerzelle0 zum Umschalten der Ausgabe zu verwenden.
Solange kein Triggerpuls von Triggerzelle0 ausgegeben wird, liegt ein Low-Pegel am Steuereingang von Mux2 an. Dieser leitet somit den im RAM befindlichen Wert default_outval (hier 2) an Mux3 weiter, welcher signal_in zur Ausgabe auswählt. Erreicht ein Triggerpuls den Steuereingang von Mux2, leitet dieser nun den im RAM befindlichen Wert triggered_outval (hier 0) an Mux3 weiter, welcher deshalb einen Low-Pegel ausgibt.

# BiDirIO

Die BiDirIO-Komponente dient dazu zwei bidirektionale Ports des FPGA so miteinander zu verbinden, dass die auf einem Port empfangenen Signale über den jeweils anderen Port ausgegeben werden. Zudem können Pulse bestimmter Länge in einer Übertragungsrichtung herausgefiltert werden. Die Ausgabe findet über TriState-Ausgangstreiber statt, welche als Open-Collector-Treiber eingesetzt werden, indem diese lediglich ein Low- oder TriState-Signal ausgeben. Beide Signalleitungen müssen über Pullup-Widerstände im High-Zustand gehalten werden, wenn nichts übertragen wird.
Abb. 103 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

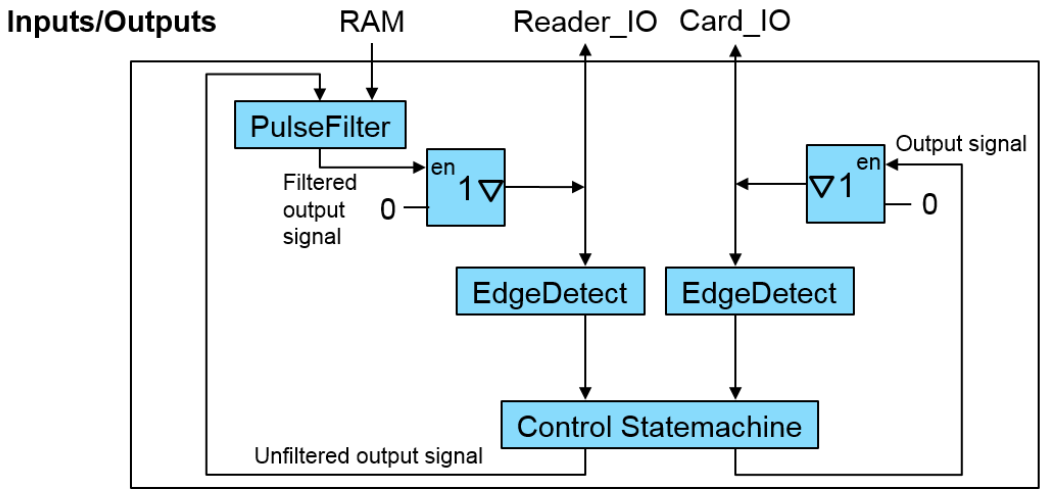


Abb. 103: Bidirektional IO Blockdiagramm

## Ein- und Ausgänge

Die BiDirIO-Schaltung verfügt neben den beiden bidirektionalen Ports: Reader_IO und Card_IO über den Eingang RAM. An diesem liegt ein aus dem ConfRAM stammendes Signal des Typs tPulseFilterMemMapArray (siehe S. 132) an, welches die von der PulseFilter-Komponente benötigten Konfigurationsparameter enthält.
Des Weiteren enthält die BiDirIO-Komponente das generische Attribut PFID. Über dieses wird ihr bei der Instantiierung die Nummer des internen Pulsfilters mitgeteilt. Diese Nummer entscheidet darüber welcher Datensatz des Record-Typs tPulseFilterMemMap (siehe Abb. 86, S. 132) aus dem am Eingang RAM anliegenden Datensatzarray an den Pulsfilter weitergereicht wird. Durch Angabe einer PFID < 0 ist es zudem möglich eine BiDirIO-Instanz ohne Pulsfilter zu erstellen. Der mögliche Wertebereich für die PFID reicht von 0 bis zur Konstanten numPulseFilters (siehe Code 8, S. 110)

## Schaltung

Die BiDirIO -Komponente enthält folgende, in Abb. 103 dargestellte, Komponenten:

### Erkennung eines Lowpulses (Edge Detect):

Der Beginn einer Datenübertragung wird mithilfe eines 2-Bit Registers erkannt. Eine fallende Flanke bedeutet den Beginn eines Low-Pulses, eine steigende dessen Ende. Diese Information wird an folgenden Zustandsautomaten weitergeleitet.

### Erkennung der Übertragungsrichtung (Control Statemachine):

Zustandsautomat zur Ermittlung der Übertragungsrichtung und Steuerung der Signalweiterleitung, wie bereits in Abschnitt „5.2.1 Bidirektionale Weiterleitung“ erläutert wurde.

## Filterung von Triggerpulsen auf Card_IO (PulseFilter):

Während das auf dem Port Card_IO auszugebende Signal direkt vom Zustandsautomaten zum Ausgangstreiber geleitet wird, findet in der anderen Übertragungsrichtung eine Filterung statt. Jeder auf dem Port Card_IO empfangene Low-Puls wird vor der Ausgabe durch die PulsFilter-Komponente geleitet. Diese leitet empfangene Pulse nur weiter, wenn deren Länge einen Grenzwert überschreiten. Dieser Grenzwert kann zur Laufzeit vom Benutzer in den ConfRAM eingespeichert werden.

# Pulsefilter

Die PulseFilter-Komponente dient dazu Pulse, deren Länge einen Grenzwert unterschreitet, aus einer Signalleitung herauszufiltern. Dazu wird die zeitliche Länge eines Pulses auf der Eingangsleitung durch CPU-Taktzählung ermittelt und eine Pegeländerung nur auf den Ausgang übernommen, wenn die Anzahl der gezählten Takte größer dem anliegenden Grenzwert ist. Abb. 104 zeigt den Aufbau der implementierten implementierten Schaltung in vereinfachter Form.

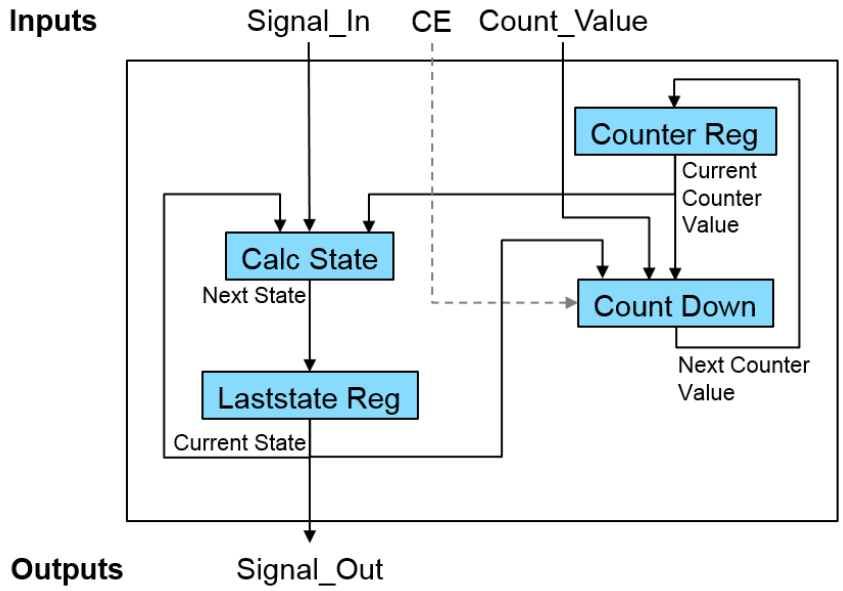


Abb. 104: PulseFilter Blockdiagramm

## Ein- und Ausgänge

Die PulseFilter-Schaltung verfügt über die, in Abb. 104 dargestellten, Eingangssignalleitungen:

- **Signal_In**: Signalleitung aus der Pulse gefiltert werden sollen.
- **Count_Value**: Zählergrenzwert zur Definition der Größe zu filternder Pulse.
- **CE**: Clock-Enable zur Reduzierung der Arbeitsgeschwindigkeit.

Des Weiteren existiert der Ausgang **Signal_Out**, welcher das gefilterte Signal ausgibt.

## Schaltung

Die PulseFilter-Komponente enthält folgende, in Abb. 104 dargestellte, Komponenten:

### Pulslängenermittlung durch CPU-Taktzählung (Count Down):

Prozess zur Steuerung des Taktzählers. Unterscheidet sich das Eingangssignal Signal_In vom zuletzt akzeptierten Eingangssignal, das im Laststate Register gespeichert ist, wird der Zähler heruntergezählt. Sind hingegen beide Signale gleich, wird der Zähler auf den am Eingang Count_Value anliegenden Wert gesetzt.

### Ausgabespeicher (Laststate Reg):

Register zur Speicherung des letzten akzeptierten Ausgabewertes. Der hierin gespeicherte Signalpegel wird auf dem Ausgang Signal_Out ausgegeben.

### Zählerspeicher (Counter Reg):

Register zur Speicherung des Zählerwertes. Dieser dient dazu eine Zeitspanne in Form von CPU-Takten zu messen. Die Bitbreite des Zählers wird durch den generischen Parameter Count Bits festgelegt.

### Eingangs- und Zählerauswertung (Calc State):

Prozess zur Aktualisierung des Ausgabewertes. Hat der Zähler bis zur Null heruntergezählt, ist der am Eingang Signal_In anliegende Wert der neue akzeptierte Wert und wird Ausgaberegister übernommen.

Die PulseFilter-Komponente wurde implementiert, um die geforderte Möglichkeit zur Filterung von Triggerpulsen zu bieten, welche von der Smartcard erzeugt werden und nicht zum Reader gelangen sollen.

# Tickgenerator

Die TickGenerator-Komponente dient dazu Clock-Enable Signale zu erzeugen. Diese können von Triggerzellen, HoldOffAndWindowTimern und Pulsfiltern verwendet werden, um lange Zeiträume mit kleinen Zählern zu messen. Abb. 105 zeigt den Aufbau der implementierten Schaltung in vereinfachter Form.

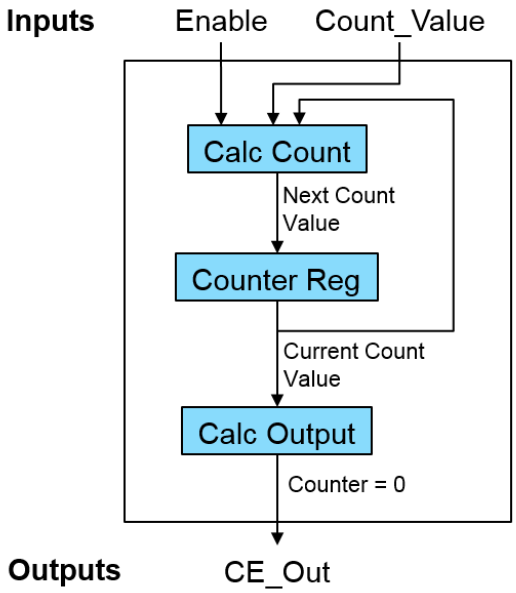


Abb. 105: Tickgenerator Blockdiagramm

## Ein- und Ausgänge

Die TickGenerator-Schaltung verfügt über die, in Abb. 105 dargestellten, Eingangssignalleitungen:

- **Enable_In**: Zum Enablen und Disablen des Tickgenerators.
- **Count_Value**: Zählergrenzwert zur Definition der Größe zu filternder Pulse.

Des Weiteren existiert der Ausgang **CE_Out**, welcher das Clock-Enable Signal ausgibt.

## Schaltung

Die TickGenerator -Komponente enthält folgende, in Abb. 105 dargestellte, Komponenten:

### Zählerspeicher (Counter Reg):

Register zur Speicherung des Zählerwertes. Dieser dient dazu eine Zeitspanne in Form von CPU-Takten zu messen. Die Bitbreite des Zählers wird durch den generischen Parameter Count Bits festgelegt.

### Zeitabstandsberechnung durch CPU-Taktzählung (Calc Count):

Prozess zur Steuerung des Taktzählers. Ist der Tickgenerator enabled, wird der am Eingang Count_Value anliegende Wert in den Zählerspeicher geladen. Dieser wird anschließend mit jedem CPU-Takt bis auf Null heruntergezählt und anschließend wieder mit dem an Wert von Count_Value überschrieben. Auf diese Weise ist es möglich zur Laufzeit auf benutzerdefinierte Änderungen zu reagieren.

### Tickerzeugung (Calc Output):

Prozess zur Erzeugung von einen CPU-Takt langen Clock-Enable Pulsen. Hat der Zähler auf Null runtergezählt, wird ein Tick ausgegeben.

Die Verwendung eines Tickgenerators macht nur dann Sinn, wenn mehrere Komponenten ihn benutzen sollen und es für diese nicht erforderlich ist die zu ermittelnden Zeiträume mit maximaler Genauigkeit zu erreichen. Die Requirements erforderten die Implementierung dieser Komponente nicht. Sie wurde dennoch implementiert, da sie, nach Aussagen der Smartcard-Entwickler, in absehbarer Zeit Anwendung finden könnte.

# Versicherung über Selbstständigkeit

*Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.*

*Hamburg, den* _____ _____