Entwurf und Inbetriebnahme eines CGRA basierten Simulators für Cavity-in-the-loop

Masterarbeit von David Scheunert
Tag der Einreichung: 30. Oktober 2023

1. Gutachten: Prof. Dr.-Ing. Christian Hochberger
2. Gutachten: M.Sc. Jonas Gehrunger
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

INFORMATIONS SYSTEM TECHNIK

Fachbereich Elektrotechnik und Informationstechnik
Institut für Datentechnik
FG Rechnersysteme

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, David Scheunert, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 30. Oktober 2023

David Scheunert

TECHNISCHE UNIVERSITÄT DARMSTADT

Aufgabenstellung Master of Science

Entwurf und Inbetriebnahme eines CGRA basierten Simulators für Cavity-in-the-loop

Institut für Datentechnik
Fachgebiet Rechnersysteme

Bearbeiter:
David Scheunert
Matrikel-Nr.: 2564324
david-scheunert@web.de

Betreuer:
Christian Hochberger
hochberger@rs.tu-darmstadt.de

Jonas Gehrunger
gehrunger@rs.tu-darmstadt.de

Zeitrahmen:
Beginn: 2. Mai 2023
Ende: 30. Okt 2023

Voraussetzungen:

1. HDL-Kenntnisse
2. CGRA Scheduling
3. FPGA Erfahrungen

Einleitung:

Im Bereich der Entwicklung von Steuergeräten und eingebetteten Systemen hat sich seit einigen Jahren die Methodik "Hardware-in-the-loop" etabliert. Dabei wird ein Steuergerät nicht mit dem realen System verbunden, welches es kontrollieren soll, sondern mit einer Simulation dieses Systems. Der Vorteil liegt darin, dass Sondersituation und Fehlerfälle in der Simulation ganz gezielt angesteuert werden können und somit das Steuergerät sehr viel komfortabler und umfangreicher getestet werden kann.

Diese Methodik lässt sich auch auf komplexe Regelungssysteme im wissenschaftlichen Bereich übertragen. So ist es denkbar, die Strahlregelung eines Cyclotrons nicht am realen Cyclotron auszutesten und zu entwickeln, sondern stattdessen ein Modell eines Strahls zu rechnen und somit die Strahlreaktion of die Regelungseingriffe nur zu simulieren. Voraussetzung hierfür ist natürlich einerseits die Möglichkeit die relevanten Steuereingriffe des Regelungssystems für die Simulation verfügbar zu machen und andererseits dem Regelungssystem dann eine vermeintliche Reaktion des Strahls vorzuspielen. Ebenso muss die Berechnung des zugrunde liegenden Modells in Echtzeit erfolgen.

Am Fachgebiet Rechnersysteme wird seit mehreren Jahren an Coarse Grained Reconfigurable Arrays (CGRA) als Rechenbeschleuniger geforscht. Dabei wurde in einem vorhergehenden Projekt auch schon ein physikalisches Modell in einem industriellen Umfeld in Echtzeit simuliert [1]. CGRAs haben also bereits bewiesen,

dass sie für den Einsatz in Echtzeitsystemen gut geeignet sind. Es bietet sich daher an, die Strahlsimulation ebenfalls mithilfe von CGRAs zu realisieren.

Aufgabe:

Im Rahmen dieser Arbeit soll ein CGRA basierter Strahlsimulator entworfen, anhand mehrerer Anwendungsfälle verifiziert und in der GSI an einem realen Strahlregelungssystem getestet werden. Dazu wurde bereits ein Prototyp gebaut, der zunächst schrittweise lauffähig gemacht und dann erweitert werden soll.

1. Zunächst soll der vorhandene Prototyp modifiziert werden, damit ein einzelner Bunch korrekt simuliert werden kann. Dies umfasst sowohl Korrekturen am ausgeführten Modell als auch an der Hardware-Implementation. Die korrekte Funktion der Simulation soll über Testfälle sichergestellt werden.
2. Im Anschluss soll ein neuer Simulationsfall implementiert werden: Der Ramp-Up. In diesem Fall wird ein simulierter Bunch durch externe Steuereingriffe von einer Ausgangsumlaufgeschwindigkeit auf eine Zielgeschwindigkeit beschleunigt. Dies soll schrittweise erfolgen, wobei zuerst die Hardware des Simulators erweitert werden muss, um die aktuelle Frequenz der Eingangssignale zu messen und mit einer variablen Frequenz umgehen zu können. Da die gemessene Frequenz starken Schwankungen unterliegt, soll diese ebenfalls durch eine Filterung geglättet werden. Außerdem muss die Erzeugung der Gauss-Pulse angepasst werden, um Pulse mit variabler Breite erzeugen zu können.
3. Mithilfe der modifizierten Hardware soll nun die Software um Unterstützung für den Ramp-Up erweitert werden. Dies umfasst sowohl die Modell-Implementation als auch das Diagnose-Interface im SpartanMC Softcore. So sind etwa bisher viele Faktoren im Modell für eine fixe Frequenz vorberechnet und müssen mit den Modifikationen nun in Echtzeit neu ermittelt werden. Auch hierbei ist die korrekte Funktion der Simulation durch Testfälle zu verifizieren.
4. Für ein sehr gutes Ergebnis soll die Simulation um zwei zusätzliche Fälle erweitert werden: Die Unterstützung mehrerer Bunches innerhalb einer Simulation und die Simulation mehrerer Macro-Partikel innerhalb eines Bunches. Dazu muss zunächst das im Prototyp enthaltene CGRA vergrößert und die Verteilung der Operatoren im CGRA angepasst werden. Anschließend kann schrittweise das Modell an die neuen Szenarien angepasst werden. Das korrekte Verhalten soll auch hier durch Testfälle verifiziert werden.

Der Code soll sich in das vorhandene Projekt modular einfügen. Der Code ist auch wartbar zu formulieren, zu kommentieren und dokumentieren. Die Entwicklung soll mit Git versioniert werden. Außerdem ist eine schriftliche Ausarbeitung im Stil der Rechnersysteme Vorlage [template] anzufertigen. Die Ausarbeitung sollte sowohl die Problemstellung als auch die geleistete Arbeit sowie die Resultate reflektieren.

Literatur:

[1] UltraSynth: https://doi.org/10.1007/978-3-030-17227-5_18

[template] RS Template: https://www.rs.tu-darmstadt.de/lehre/download/

Prof. Dr.-Ing. Christian Hochberger

David Scheunert

Inhaltsverzeichnis

1 Einleitung **5**
1.1 Aufbau der Arbeit . . . 6

2 Grundlagen **7**
2.1 Synchrotron . . . 7
2.1.1 Synchrotronmodell . . . 7
2.1.2 Mathematisches Modell - Stationärer Fall . . . 9
2.2 FPGA . . . 12
2.3 CGRA . . . 14
2.3.1 Programmierung und Scheduling . . . 17

3 Aufbau des Simulators **19**
3.1 FPGA Design . . . 19

4 Stationärer Fall **22**
4.1 C Modell . . . 22
4.2 Softwareanpassungen . . . 22
4.3 Hardwareanpassungen . . . 24

5 Ramp Up **26**
5.1 Modellanpassungen . . . 26
5.2 Hardwareanpassung . . . 28
5.2.1 DRAM-Ansteuerung . . . 30
5.3 Softwareanpassungen . . . 35
5.4 Testen . . . 38

6 Multi Bunch **40**
6.1 Anpassungen . . . 40
6.1.1 Codevervielfachung . . . 40
6.1.2 Spannungsmessung . . . 43
6.1.3 h zur Laufzeit berechnen . . . 44
6.1.4 System beschleunigen . . . 45
6.1.5 Modellanpassungen . . . 46

7 Quadrupolschwingung **48**
7.1 Hardwareanpassungen 48
7.2 Softwareanpassungen 50

8 Evaluation **51**
8.1 Stationärer Fall 51
8.2 Ramp Up 54
8.3 Multi Bunch 57
8.4 Quadrupolschwingung 57
8.4.1 Maximale Anzahl an Minibunches 62
8.5 Harwarebedarf 65

9 Fazit **67**
9.1 Ausblick 67

1 Einleitung

In der heutigen schnelllebigen Welt der Elektronikentwicklung wächst die Notwendigkeit, moderne elektronische Systeme effizienter zu entwickeln und zu testen, um die ständig steigenden Anforderungen an Leistung und Zuverlässigkeit zu erfüllen. Um diesem Trend gerecht zu werden, gewinnen sogenannte *Hardware-in-the-Loop* (HiL)-Versuchsaufbauten zunehmend an Bedeutung. Bei diesen Versuchsaufbauten wird die entwickelte Hardware mit einem System verbunden, das die Einsatzumgebung der Neuentwicklung in Hardware simuliert. Vor allem in der Entwicklung und Testung von elektronischen Steuergeräten gewinnt die Verwendung von HiL immer mehr an Beliebtheit, insbesondere auch deshalb, weil die immer weiter wachsende Leistungsfähigkeit modernere Rechnersysteme es ermöglicht, zunehmend komplexere Systeme in Echtzeit zu simulieren. Das Konzept von HiL hat somit Einzug in die Entwicklungsabteilungen vieler Branchen, wie z.B. der Automobil- oder Luftfahrtindustrie, erhalten.

Ein Bereich, in dem solche HiL-Systeme bisher noch keine Verwendung gefunden haben, sind Teilchenbeschleuniger. In dieser Arbeit wird daher ein HiL-System vorgestellt, das den *SIS18* Schwerionensynchrotron der *Gesellschaft für Schwerionenforschung* (GSI) in Darmstadt simulieren soll. Damit soll erreicht werden, dass die Weiterentwicklung der Steuerelektronik des *SIS18*, auch als *Low-Level-Radio-Frequency* (LLRF)-System [3] bezeichnet, mit Hilfe des Simulators erfolgen kann. Die Schnittstellen des LLRF-Systems werden hierfür direkt an den Simulator angeschlossen. Dadurch wird im besten Fall erreicht, dass keine kostbare Strahlzeit des Teilchenbeschleunigers mehr für die Entwicklung des LLRF-Systems benötigt wird, sodass der normale Betriebsablauf des Teilchenbeschleunigers nicht durch Versuche und Tests unterbrochen werden muss.

Voraussetzung dafür, dass die Ergebnisse des HiL-Versuchsaufbaus auch praxisrelevant sind, ist, dass das verwendete Modell des Teilchenbeschleunigers genau genug ist, um die Realität widerzuspiegeln und dass die Berechnung in Echtzeit ausgeführt werden kann. Der *SIS18* erreicht Umlauffrequenzen von bis zu $1.4\,\mathrm{MHz}$, was dazu führt, dass das Modell zum Erfüllen der Echtzeitanforderung in weniger als $1\,\mu\mathrm{s}$ ausgewertet werden muss. Um eine solch schnelle Berechnungszeit zu erreichen, wird ein am Fachgebiet Rechnersysteme entwickeltes *Coarse Grain Reconfigurable Array* (CGRA) [19] auf einem *Field Programmable Gate Array* (FPGA) für die Auswertung des Modells verwendet.

Bereits vor Beginn dieser Arbeit wurde am Fachgebiet Rechnersysteme ein Prototyp für das HiL-System entwickelt, auf dem diese Arbeit aufbaut. Dieser Prototyp umfasste lediglich den stationären Fall des Teilchenbeschleunigers und wies noch Verbesserungspotential auf. Ziel dieser Arbeit ist es, diesen Prototypen zu stabilisieren und ihn um komplexere Szenarien als den stationären Fall zu erweitern.

1.1 Aufbau der Arbeit

Zu Beginn dieser Arbeit werden grundlegende Konzepte über Synchrotrons und deren mathematische Modellierung erklärt. Weiterhin werden die Vorgehensweise und Struktur von FPGAs und CGRAs erläutert. Anschließend wird der interne Aufbau des HiL-Simulators vorgestellt. Es folgt eine Beschreibung der Änderungen am Simulator, um den stationären Fall besser zu emulieren. Danach werden noch drei weitere praxisrelevante Szenarien näher beleuchtet und ihre Implementierung beschrieben. Am Ende gibt es eine Einordnung der ermittelten Resultate.

2 Grundlagen

In diesem Kapitel soll zunächst ein Überblick über den Versuchsaufbau sowie die grundlegenden Konzepte der verwendeten Hardware vermittelt werden. Außerdem wird das Grundprinzip eines Synchrotrons erläutert.

2.1 Synchrotron

Ein Synchrotron ist ein leistungsstarker Teilchenbeschleuniger, der in der modernen Teilchenphysik und anderen wissenschaftlichen Anwendungen verwendet wird. Es basiert auf einem Ringdesign und nutzt starke Magnetfelder sowie Hochfrequenztechnologie, um geladene Teilchen auf extrem hohe Geschwindigkeiten zu beschleunigen. In einem Synchrotron arbeiten viele Elemente zusammen, um die geladene Teilchen auf diese extrem hohen Energien zu bringen und sie in stabilen Strahlen für wissenschaftliche Experimente zur Verfügung zu stellen. Die Fähigkeit, die Energie der Teilchen präzise zu kontrollieren und die Strahlqualität zu optimieren, macht Synchrotrone zu unverzichtbaren Werkzeugen in der modernen Teilchenphysik und anderen Bereichen wie bespielsweise der Strahlentherapie. Abbildung 2.1 zeigt den schematischen Aufbau eines Synchrotrons.

Die Beschleunigung der Teilchen erfolgt in Synchrotronen dadurch, dass an sogenannten Kavitäten eine hochfrequente Wechselspannung angelegt wird. Durch das Anlegen einer solchen Wechselspannung bilden sich Gruppen an Teilchen, sogenannte Bunches, die gemeinsam durch den vakuumierten Teilchenbeschleuniger fliegen. Im Regelfall folgt die Verteilung der einzelnen Teilchen eines Bunches einer Gaußkurve, allerdings sind auch andere Verteilungen möglich. Wird die Frequenz der angelegten Spannung auf ein ganzzahliges Vielfaches der Umlauffrequenz eines Bunches festgelegt, so bilden sich mehrere Bunche aus. Die Anzahl der Bunches entspricht dabei dem Faktor, mit dem die Umlauffrequenz multipliziert wurde, der sogenannten *harmonic number* h. Alle Bunches zusammen ergeben das Strahlsignal, welches auch das Ausgabesignal unseres Simulators ist.

2.1.1 Synchrotronmodell

Für die Bildung des verwendeten Synchrotronmodells wird die Longitudinal-Strahl-Dynamik-Theorie verwendet [4, 6, 11]. In dieser wird ein nicht reales Referenzteilchen definiert. Für

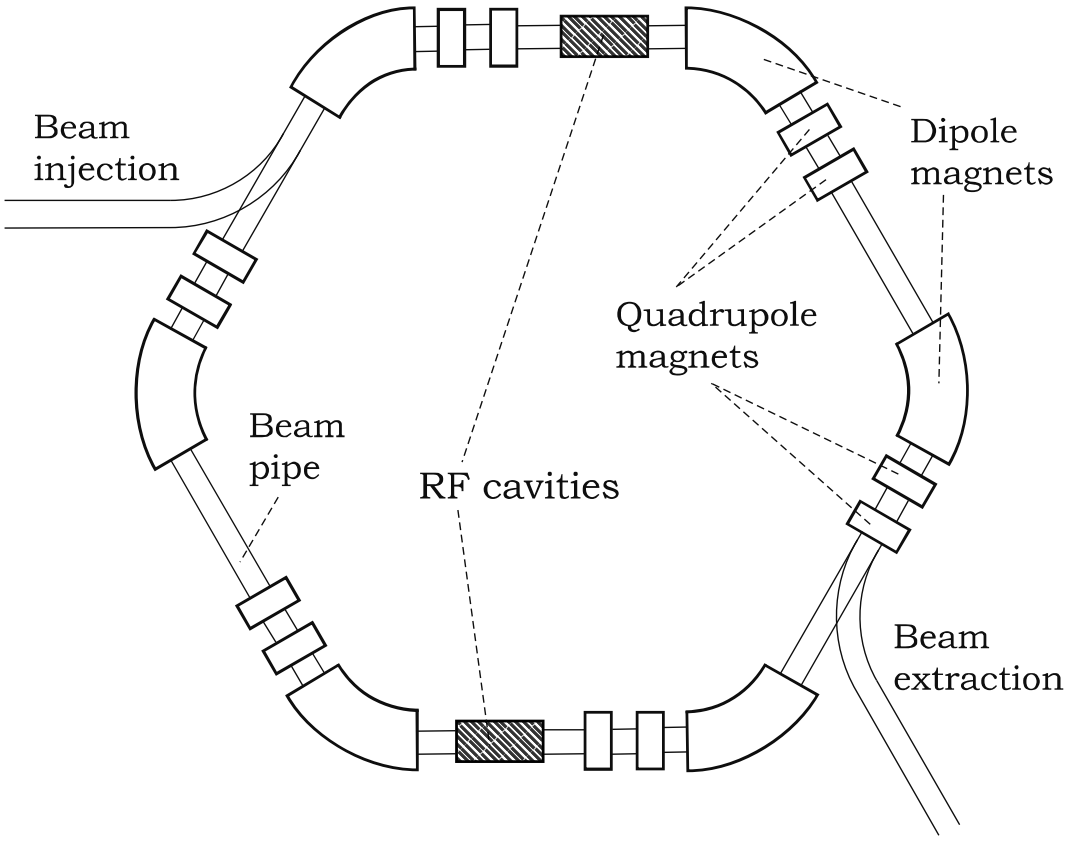


Abbildung 2.1: Schematische Darstellung eines Synchrotrons [10]

dieses Teilchen gilt, dass alle Parameter so eingestellt sind, dass es genau dem gewünschten Beschleunigungsszenario entspricht. Die Magnetfelder des Synchrotrons sind daher auch immer so konfiguriert, dass sie das Referenzteilchen exakt im selben Orbit halten. Für das Referenzteilchen wird weiterhin eine Spannung (Referenzspannung) definiert, die nicht identisch mit der real an der Kavität anliegenden Spannung (Gapspannung) ist und die exakt dem Verlauf folgt, der nötig ist, um das Referenzteilchen wie gewünscht zu beschleunigen. Da echte Teilchen gegenüber dem idealen Referenzteilchen versetzt sein können, werden sie im Folgenden auch als asynchrone Teilchen bezeichnet.

Wenn nun ein echtes Teilchen später als das Referenzteilchen an der Kavität ankommt, erfährt es eine positive Spannung und wird beschleunigt. Umgekehrt gilt für ein Teilchen, dass zu früh ankommt, dass es durch eine negative Spannung abgebremst wird. Eine Änderung der Geschwindigkeit führt zu einer Änderung des Orbits des Teilchens, weshalb ein nicht lineares Verhalten von Teilchen beobachtet werden kann, die gegenüber dem Referenzteilchen verschoben sind. Dieses Verhalten zeigt sich in einer Longitudinalschwingung, die die Position des Referenzpartikels als Zentrum aufweist.

Der einfachste anzunehmende Fall ist der sogenannte stationäre Fall. In diesem Fall durchquert das Referenzteilchen den Beschleuniger mit konstanter Energie. Daraus folgt, dass das Referenzteil-

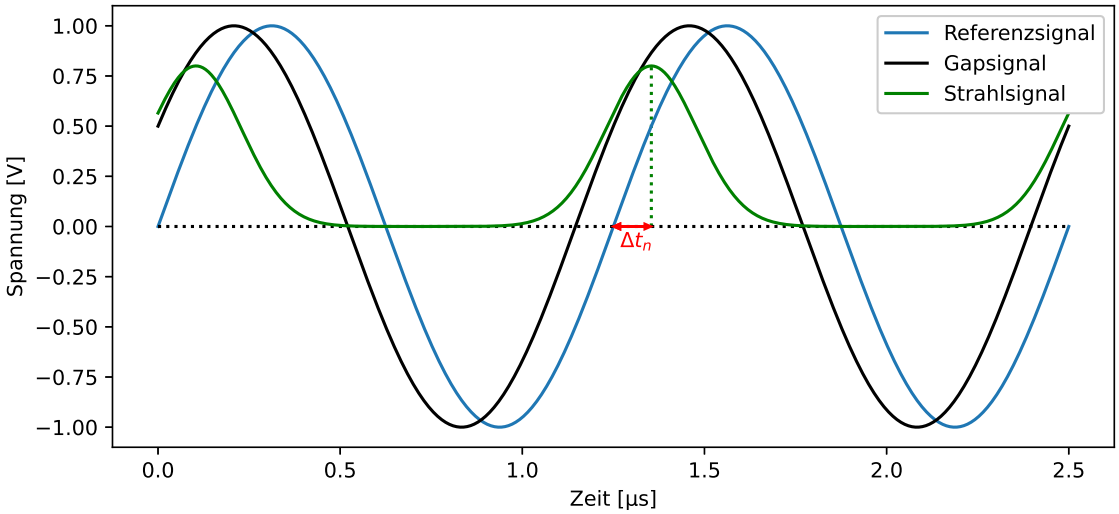


Abbildung 2.2: Beispielhafte Verläufe für Referenz- und Gapspannung sowie für das Strahlsignal im nicht Gleichgewichtszustand

chen die Kavität immer zum positiven Nulldurchgang der Referenzspannung passiert. Erreicht ein echter Bunch nun die Kavität um einen Zeitpunkt Δt_n gegenüber dem Referenzteilchen versetzt (vgl. Abbildung 2.2), so wird eine Longitudinalschwingung, wie eben beschrieben, angeregt. Ein solcher zeitlicher Versatz Δt_n kann gezielt durch eine Phasenverschiebung der realen Spannung gegenüber der Referenzspannung erzeugt werden. Die Aufgabe, eine so erzeugte Schwingung auszuregeln und das System zu stabilisieren, wird durch das LLRF-System erfüllt. In Abbildung 2.2 sind die Signale sowie Δt_n beispielhaft dargestellt.

2.1.2 Mathematisches Modell - Stationärer Fall

In dieser Sektion wird das mathematische Modell des Synchrotrons vorgestellt, das auf dem HiL-Simulator implementiert wird. Das mathematische Vorgehen sowie die Nomenklatur orientieren sich dabei an [10].

Vorbemerkungen

Größen, die sich auf das Referenzteilchen beziehen, werden mit dem Index R gekennzeichnet. Außerdem gibt es noch einen weiteren Index, n, der angibt, dass es sich um einen Variable handelt, die sich mit jedem Umlauf im Synchrotron ändert und sich auf ihren Wert in der n-ten Umdrehung bezieht. Die Abweichungen einer Größe eines echten Bunches zum Referenzbunch wird durch ein Δ-Symbol ausgedrückt, wie es beispielsweise beim zuvor erwähnten Δt_n bereits verwendet wurde.

Für eine korrekte mathematische Modellierung des Synchrotrons muss beachtet werden, dass das Modell auf Grund der hohen Geschwindigkeiten von bis zu 90% der Lichtgeschwindigkeit

c_0 relativistische Effekte berücksichtigen muss. Dafür werden zwei sogenannte Lorentzfaktoren gemäß Gleichung (2.1) eingeführt. Der eine Faktor, β, gibt hierbei an, wie viel Prozent der Lichtgeschwindigkeit das Teilchen bereits erreicht hat. Mit dem anderen Faktor, γ, wird angegeben, um welchen Faktor sich die Energie des Teilchen gegenüber seiner Energie im Ruhezustand aufgrund der speziellen Relativitätstheorie vergrößert hat. Aus Gleichung (2.1) geht auch hervor, dass die beiden Lorentzfaktoren voneinander abhängig sind, sodass das Wissen über einen Faktor ausreicht, um den anderen zu ermitteln.

$$\beta = \frac{v}{c_0}; \quad \gamma = \frac{1}{\sqrt{1-\beta^2}} \tag{2.1}$$

Weiterhin wird vereinfachend angenommen, dass jeder Bunch nur aus einem einzigen großen Makropartikel besteht, weshalb alle Berechnungen nur für dieses eine Makropartikel durchgeführt werden und nicht für jedes einzelne Teilchen innerhalb des Bunches.

Modell

Der erste Schritt in der Berechnung des Modells ist die Berechnung der Lorentzfaktoren des Makropartikels für den aktuellen Umlauf. Die Energie des Teilchens ändert sich um die potentielle Energie, die es beim Eintritt in die Kavität aufgrund des anliegenden elektrischen Feldes besitzt. Wegen der extrem hohen Geschwindigkeiten der Teilchen kann angenommen werden, dass sich die an der Kavität anliegende Spannung nicht verändert, solange das Teilchen diese durchquert. Gleichung (2.2) gibt eine rekursive Vorschrift vor, wie sich die Abweichung der Energie des asynchronen Bunches mit jeder Umrundung des Synchrotrons ändert. Die Spannung ΔV_{n-1} ist hierbei die Differenz zwischen den Spannungen $V_{R,n-1}$, die das Referenzpartikel an der Kavität erfährt, und V_{n-1}, die das asynchrone Teilchen sieht.

$$\gamma_n - \gamma_{R,n} = \Delta\gamma_n = \Delta\gamma_{n-1} + \frac{Q}{m_0 c_0^2}\Delta V_{n-1} \tag{2.2}$$

Der Lorentzfaktor γ_n des asynchronen Teilchens kann schließlich durch das Addieren von $\gamma_{R,n}$ mit $\Delta\gamma_n$ ermittelt werden. Im stationären Fall vereinfacht sich das Ganze dadurch, dass zum einen $V_{R,n}$ immer Null ist, weshalb anstelle von ΔV_{n-1} direkt V_{n-1} verwendet werden kann. Zum anderen ist $\gamma_{R,n}$ im stationären Fall unveränderlich, weshalb es als Konstante γ_R betrachtet wird.

Da das asynchrone Teilchen nicht zwangsläufig dieselbe Energie wie das Referenzteilchen besitzt, fliegt es im Gegensatz zu diesem auch nicht immer auf derselben Umlaufbahn durch den Teilchenbeschleuniger, denn die Magnetfelder des Synchrotrons sind für das Referenzteilchen kalibriert. Der Impulsverdichtungsfaktor α_c (*Momentum Compaction Factor*) ist ein Faktor, der eine Beziehung zwischen dem Verhältnis der Umlaufbahnlängen von Referenzteilchen und asynchronem Teilchen mit dem Verhältnis der Impulse der beiden Teilchen gemäß Gleichung (2.3) herstellt. α_c ist eine Konstante, die individuell für jeden Teilchenbeschleuniger ist, und hängt zusätzlich auch von der verwendeten Ionenoptik ab. In den meisten Fällen, wie auch beim SIS18 der GSI, ist α_c positiv.

$$\frac{\Delta l_n}{l_R} = \alpha_c \frac{\Delta p_n}{p_{R,n}} \tag{2.3}$$

Aus α_c lässt sich mit Hilfe von Gleichung (2.4a) der sogenannte Phasenschlupffaktor $\eta_{R,n}$ (*Phase Slip Factor*) ermitteln. Dieser besitzt eine ähnliche Bedeutung wie α_c, allerdings stellt er eine Verbindung zwischen dem Verhältnis der Umlaufdauern T (anstelle der Umlaufbahnlängen) der Teilchen mit dem Verhältnis ihrer Impulse her (siehe Gleichung (2.4b)). Weil weiterhin für den stationären Fall gilt, dass das Referenzteilchen seine Energie nicht ändert, ist $\gamma_{R,n}$ eine Konstante. Deshalb folgt aus Gleichung (2.4a), dass auch $\eta_{R,n}$ im stationären Fall als unveränderlich angesehen werden kann.

$$\eta_{R,n} = \alpha_c - \frac{1}{\gamma_{R,n}^2} \tag{2.4a}$$

$$\frac{\Delta T_n}{T_{R,n}} = \eta_{R,n} \frac{\Delta p_n}{p_{R,n}} \tag{2.4b}$$

Schließlich muss noch der neue Zeitversatz Δt_n zwischen dem Eintreffen des Referenzteilchens und dem des asynchronen Teilchens ermittelt werden. Ausgangsbasis ist dabei Gleichung (2.5a), welche eine rekursive Vorschrift für die Berechnung der neuen zeitlichen Abweichung darstellt. Δt_n wird ermittelt, indem Δt_{n-1} um die zeitliche Abweichung vergrößert wird, die im aktuellen Umlauf dazugekommen ist.

$$\begin{aligned}
\Delta t_n &= \Delta t_{n-1} + T_n - T_{R,n} \\
&= \Delta t_{n-1} + \frac{l_R + \Delta l_n}{\beta_n c_0} - \frac{l_R}{\beta_{R,n} c_0} \\
&= \Delta t_{n-1} + \frac{l_R}{\beta_n c_0} \left(\frac{\Delta l_n}{l_R} - \frac{\Delta \beta_n}{\beta_{R,n}} \right) && (2.5a) \\
&= \dots \\
&= \Delta t_{n-1} + \frac{l_R \eta_{R,n}}{\beta_n \beta_{R,n}^2 c_0} \frac{\Delta \gamma_n}{\gamma_{R,n}} && (2.5b)
\end{aligned}$$

Unter Verwendung von Gleichungen (2.3) und (2.4a) lässt sich schließlich mit Hilfe von einigem Umformen der Gleichung (2.5a) und folgenden Vereinfachungen

$$\frac{\Delta \gamma_n}{\gamma_{R,n}} \ll 1 \; ; \; \frac{\Delta \beta_n}{\beta_{R,n}} \ll 1 \; ; \; \frac{\Delta \beta_n}{\beta_{R,n}} \approx \frac{1}{\gamma_{R,n}^2 \beta_{R,n}^2} \frac{\Delta \gamma_n}{\gamma_{R,n}}$$

Gleichung (2.5b) erhalten. Auch hier können im stationären Fall wieder alle Größen des Referenzteilchens als konstant angesehen werden.

2.2 FPGA

Da der HiL-Simulator auf dem FPGA eines *AMD/Xilinx Virtex 7 VC707 Evaluation Kits* [1] implementiert wird, wird in diesem Abschnitt die grundlegende Funktionsweise von FPGAs erläutert.

FPGAs sind *Integrated Circuits* (IC), deren elektrische Schaltung nicht festgelegt ist, wenn sie hergestellt werden. Stattdessen kann der Benutzer sie mit Hilfe eines nach seinen Anforderungen entwickeltem Designs konfigurieren. Hauptmerkmal von FPGAs ist ihre Rekonfigurierbarkeit, sodass das aufgespielte Design beliebig oft geändert werden kann. Diese Flexibilität ermöglicht die Verwendung individuell gestalteter Chips, selbst in Fällen, in denen die Entwicklung und Produktion eines *Application-Specific Integrated Circuits* (ASIC) aufgrund geringer Stückzahlen nicht wirtschaftlich ist.

Im Laufe der Jahre wurden verschiedene Technologien in FPGAs eingesetzt, darunter beispielsweise die sogenannte „Anti-Fuse“-Technik, bei der durch Anlegen eines hohen Stroms an bestimmten Stellen eine dauerhafte Verbindung zwischen Metallschichten hergestellt wird, um das FPGA zu programmieren. Dieser Vorgang ist jedoch nicht reversibel, weshalb hierbei die Rekonfigurierbarkeit verloren geht. Am häufigsten werden *Static Random-Access Memory* (SRAM)-Zellen verwendet, um die Konfiguration des FPGAs zu speichern. Auch die FPGAs der *Virtex 7*-Serie von *AMD/Xilinx* verwenden diese Technologie. Ein Nachteil von SRAM ist jedoch, dass der Speicherinhalt verloren geht, wenn die Stromversorgung unterbrochen wird. Daher ist oft ein externer nichtflüchtiger Speicher erforderlich, aus dem das FPGA seine Konfiguration beim Einschalten lädt. Es gibt auch die Idee, innovative Speicherlösungen wie Memristoren in FPGAs zu verwenden, die im Gegensatz zum typischerweise verwendeten SRAM-Speicher ihre Konfiguration beibehalten, wenn das FPGA ausgeschaltet wird.

Die Logik in einem FPGA wird normalerweise mithilfe von sogenannten *Look Up Tables* (LUT) realisiert. Jede LUT verfügt über n Eingangssignale, mit denen der gewünschte Ausgabewert aus den 2^n SRAM-Zellen der LUT ausgewählt wird. Um Platz auf dem Chip zu sparen, werden anstelle von Multiplexern einzelne NMOS-Transistoren als Pass-Transistoren für die Ausgangsauswahl verwendet. Diese führen zwar zu einer gewissen Signaldegradation, da sie ein um die Schwellspannung reduziertes 1-Signal liefern, jedoch ist die Signaldegradation kalkulierbar, da der Hersteller die Anzahl der hintereinandergeschalteten Pass-Transistoren kennt. Durch das Verwenden von entsprechenden Bauteilen an den entscheidenden Stellen wird eine Regeneration des Signals durchgeführt, um die Funktionalität der Schaltung zu gewährleisten.

Abbildung 2.3 zeigt die typische Struktur eines FPGAs im sogenannten „Island-Style“, mit Bezeichnungen wie sie von *Xilinx* konventionell verwendet werden. An den Rändern des Chips befinden sich I/O-Blöcke, die für die Kommunikation mit externen Geräten benötigt werden. Die *Configurable Logic Blocks* (CLB) bestehen hauptsächlich aus LUTs und dienen zur Implementierung bitweiser Logik. Darüber hinaus enthalten CLBs auch mehrere Flipflops und Latches. Einige LUTs in einem CLB besitzen auch spezielle Funktionen wie *Fast Carry* oder die Verwendung einer LUT als Speicher.

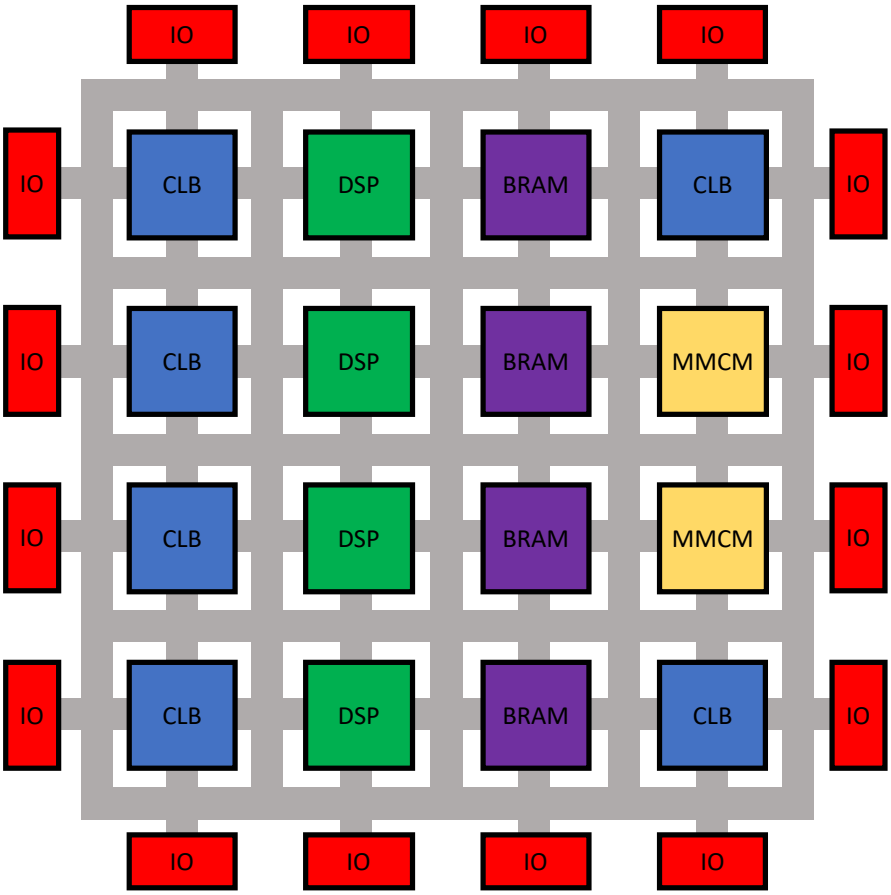


Abbildung 2.3: Struktur eines FPGAs (erstellt in Anlehnung an [9])

Da die meisten Anwendungen arithmetische Operationen erfordern, die mit LUTs nur ineffizient implementiert werden können, verfügen moderne FPGAs über spezielle *Digital Signal Processing*-Blöcke (DSP), die Additionen, Multiplikationen und komplexe Funktionen wie *Multiply-Accumulate* effizient unterstützen.

Um Daten zu speichern, bieten FPGAs spezielle Speicherblöcke namens *Block Random-Access Memory* (BRAM). Diese BRAMs bestehen aus SRAM-Zellen und ermöglichen somit einen Speicherzugriff in einem einzelnen Takt. Bei *Xilinx*-FPGAs verfügen die Speicherblöcke über zwei Zugriffsports, die unabhängig voneinander sowohl lesend als auch schreibend benutzt werden können.

Von großer Bedeutung sind auch die Blöcke zur Takterzeugung, die als *Mixed Mode Clock Manager* (MMCM) bezeichnet werden. Mit diesen Blöcken kann ein Takt mit nahezu beliebiger Frequenz aus dem Signal eines externen Oszillators generiert werden.

Um alle diese Blöcke miteinander zu verbinden und die Kommunikation zwischen ihnen zu ermöglichen, wird ein komplexes Routingsystem verwendet. Dabei werden nicht alle Blöcke direkt miteinander verbunden, sondern es kommen verschiedene Verbindungstypen wie *Single Lines*,

Double Lines und *Quad Lines* zum Einsatz, um Verbindungen zu benachbarten Blöcken herzustellen, die ein, zwei oder vier Blöcke weit entfernt sind (vgl. Abbildung 2.4). Die Auswahl, welche Verbindungen aktiv sind, wird wiederum durch Multiplexer oder Pass-Transistoren getroffen und kann nach Bedarf konfiguriert werden.

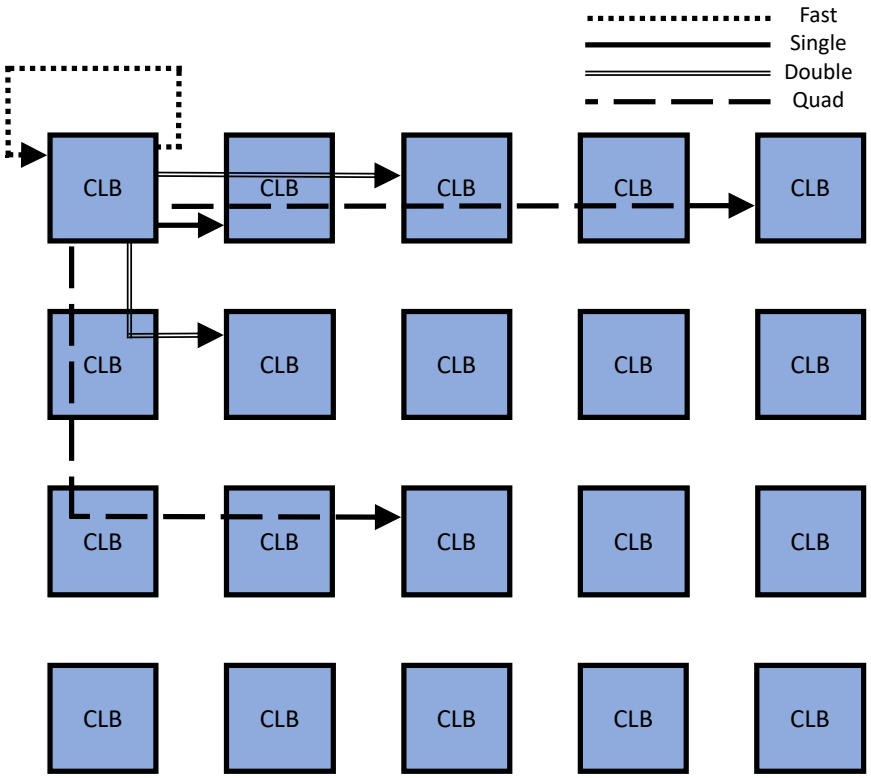


Abbildung 2.4: Interconnect-Aufbau (erstellt in Anlehnung an [17])

2.3 CGRA

Wie bereits einleitend erwähnt, wird ein CGRA-Overlay auf dem FPGA verwendet, um die Berechnung des Modells auszuführen. Aus diesem Grund wird nachfolgend ein kurzer Überblick über die Grundkonzepte eines CGRAs gegeben. Ein CGRA ist eine Rechenarchitektur, die für sehr rechenintensive Anwendungen mit viel Parallelität entwickelt wurde. Aufgrund ihrer hohen Performance werden CGRAs häufig als ein Hardwarebeschleuniger verwendet, der als Ergänzung zu einer herkömmlichen *Central Processing Unit* (CPU) dient.

Aufgebaut sind CGRAs aus sogenannten *Processing Elements* (PE), welche untereinander verbunden sind. Prinzipiell ist es dem Entwickler möglich, diese Verbindungen frei zu konfigurieren. Es gibt

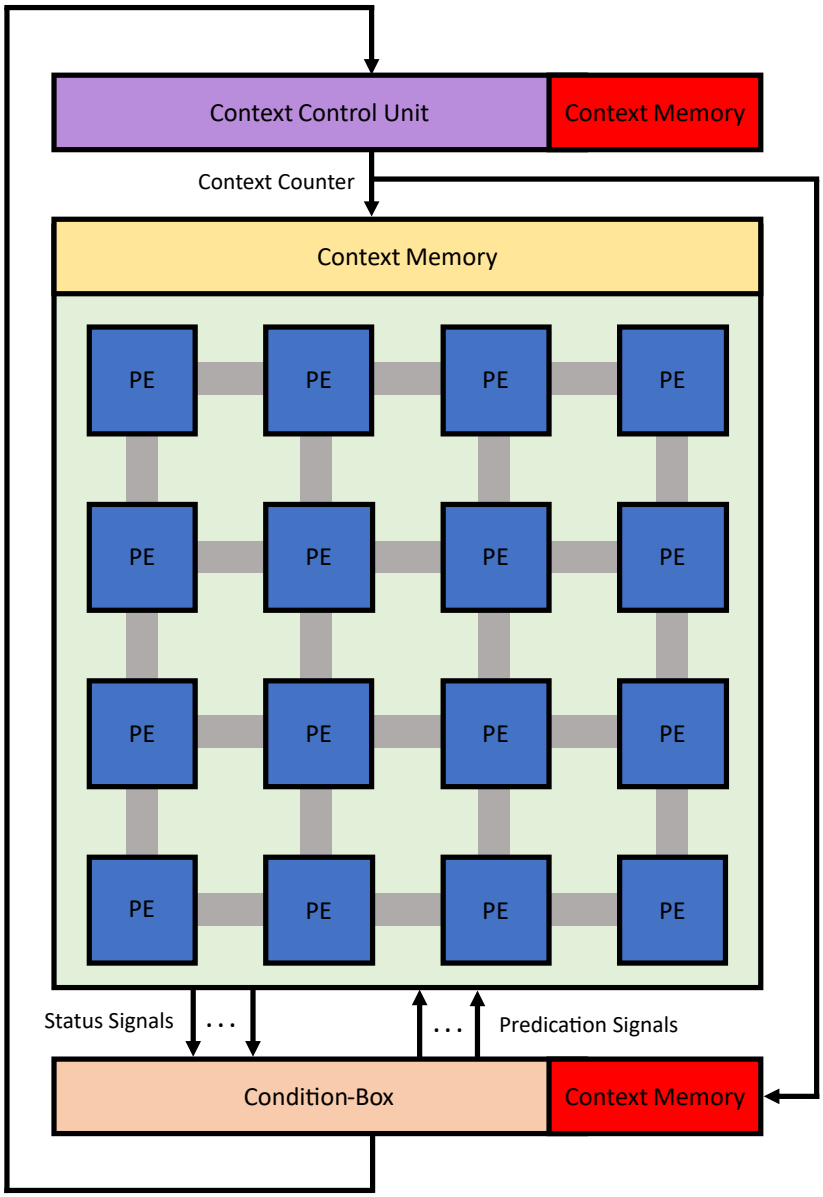


Abbildung 2.5: Aufbau eines CGRAs (erstellt in Anlehnung an [17])

allerdings typische Arten, wie diese Verbindungen konstruiert werden, z.B. immer die direkten Nachbarn zu verbinden (auch mit Wrap Arround über die Grenzen des CGRAs möglich) oder das sogenannte *Star Interconnet*, bei dem zusätzlich auch die diagonalen Nachbarn verbunden werden. Abbildung 2.5 zeigt den schematischen Aufbau eines CGRAs. Ähnlich wie ein FPGA besteht auch ein CGRA aus einem Array von Recheneinheiten, weshalb der Aufbau des CGRAs stark an den Aufbau eines FPGAs erinnert. Im Gegensatz zu den LUTs des FPGAs, die auf einzelnen Bits arbeiten, verarbeiten die PEs aber immer ganze Wörter. Daher muss immer nur für jedes Wort und nicht für jedes Bit abgespeichert werden, wie dieses verarbeitet oder weitergeleitet werden muss. Das hat den enormen Vorteil, dass deutlich weniger Informationen für die Konfiguration

des CGRAs (der sogenannte Kontext) gespeichert werden müssen, als es bei einem FPGA der Fall ist. Dadurch ist es ohne Probleme möglich, hunderte verschiedene Kontexte abzuspeichern und innerhalb eines einzelnen Taktes zwischen diesen zu wechseln. Ein CGRA kann somit in jedem Takt ändern, welche PEs mit welchen anderen PEs Daten austauschen und welche Funktionen die PEs ausführen. Zusätzlich ist durch die geringere Komplexität auch die Generierung einer Konfiguration für das CGRA um Größenordnungen schneller als die Synthese eines FPGA-Designs.

Die einzelnen PEs eines CGRAs besitzen in der Regel eine ganze Reihe an arithmetischen Funktionen, die sie ausführen können. Diese Funktionen können prinzipiell unterschiedlichster Natur sein und von einfachen Ganzzahloperationen bis hin zu numerischen Gleitkommaoperationen, wie das *Coordinate Rotation Digital Computer* (CORDIC)-Verfahren zum Berechnen trigonometrischer Funktionen, reichen. Der Entwickler kann frei wählen, welche Operationen auf welchem PE vorhanden sein sollen. Dadurch kann bspw. eine aufwendige Operation, die in der gegebenen Anwendung nicht häufig vorkommt, wie z.B. eine Quadratwurzelberechnung, von den meisten PEs entfernt werden, um Hardware zu sparen, ohne die Performance des CGRAs nennenswert zu verschlechtern.

Eine besonders wichtige Operation ist der Speicherzugriff, damit das CGRA auch mit der Außenwelt interagieren kann. Normalerweise haben nicht alle PEs einen eigenen Zugriff zum Speicher, weil das zu deutlich mehr Hardwarebedarf führen würde, da jedes PE, das den Speicher erreichen kann, mit einem eigenen Cache verbunden ist. Vor allem das Gewährleisten der Cachekohärenz ist bei vielen Caches äußerst aufwendig. Aus diesem Grund ist es sehr wichtig, die PEs, die einen Speicherzugriff besitzen, wohlüberlegt auf dem CGRA zu verteilen, um eine gute Performance zu erreichen.

Abbildung 2.6 zeigt den prinzipiellen Aufbau eines PEs mit Speicherzugriff. Die Operanden, die in der *Arithmetic Logic Unit* (ALU) des PEs verarbeitet werden, können entweder aus dem Registerfile

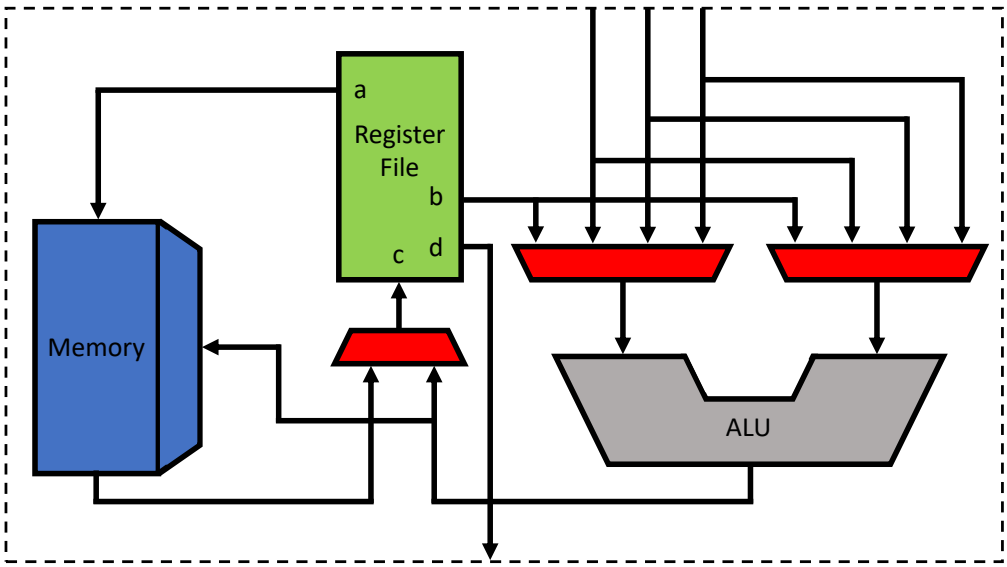


Abbildung 2.6: Aufbau eines PEs (erstellt in Anlehnung an [17])

des PEs selbst kommen oder von einem benachbarten PE stammen. Die benötigten Werte werden entsprechend über Multiplexer ausgewählt. Für das am Fachgebiet Rechnersysteme entwickelte CGRA gilt, dass bei Operationen, die zwei Operanden benötigen, der zweite Operand von einer benachbarten PE geliefert werden muss, weil nur ein Wert aus dem PE eigenen Registerfile gelesen wird, der dann an beide Eingänge des Multiplexers vor der ALU angelegt wird. Das Ergebnis der ALU kann entweder im Registerfile gespeichert werden oder es wird noch im selben Takt als Adresse für den Speicher benutzt. Dadurch wird es ermöglicht, einen Speicherzugriff in einem einzigen Takt durchzuführen. Ein zu speichernder Wert muss allerdings immer bereits im Registerfile vorhanden sein und kann nicht das Ergebnis der ALU sein. Das Ausgabesignal des PEs ist ein Wert aus dem Registerfile, der dann den benachbarten PEs zur Verfügung gestellt wird.

Zusätzlich bieten PEs auch einen Ausgang für Bedingungssignale. Diese werden an sogenannte *Condition-PEs* gesendet, welche den Kontrollfluss im CGRA steuern. Abbildung 2.5 zeigt eine ältere Version des CGRAs, die noch keine *Condition-PEs* sondern eine *Condition-Box* besitzt. Die *Condition-Box* erfüllt dabei dieselben Aufgaben wie die *Condition-PEs*. Zum Behandeln von Verzweigungen im Kontrollfluss verwendet das CGRA des Fachgebiets Rechnersysteme Spekulation. Das heißt, dass alle Pfade und die Berechnung des Auswahlkriteriums parallel ausgeführt werden und dass anschließend nur die Ergebnisse des korrekten Pfades übernommen werden. Dabei muss natürlich darauf geachtet werden, dass nur der korrekte Pfad den Verlauf des restlichen Programms beeinflusst. Aus diesem Grund dürfen die einzelnen Pfade der Verzweigung keine Seiteneffekte besitzen, wie sie beispielsweise von Speicher- oder Peripheriezugriffen verursacht werden. Zum einen werden daher von den Condition-PEs Prädikatsignale aus den Bedingungssignalen der PEs ermittelt, mit denen dafür gesorgt wird, dass nur die Ergebnisse des korrekten Pfades übernommen werden. Zum anderen werden die Bedingungssignale in den Condition-PEs auch verwendet, um den Kontext für den nächsten Takt zu ermitteln. Das wird zum Beispiel dann benötigt, wenn eine Schleife ausgeführt wird und die Schleifenabbruchsbedingung ausgewertet wird. Das Condition-PE muss dann entscheiden, ob der erste Kontext innerhalb der Schleife oder der Kontext unmittelbar nach der Schleife verwendet werden soll.

Insgesamt kann ein CGRA auch sehr gut mit einem *Very Large Instruction Word* (VLIW)-Prozessor verglichen werden. Der größte Unterschied ist hierbei, dass die einzelnen *Functional Units* (FU) des VLIW-Prozessors (vergleichbar mit den PEs des CGRAs) alle dasselbe Registerfile verwenden, während die PEs des CGRAs jeweils ihr eigenes Registerfile besitzen. Deshalb ist die maximale Anzahl an PEs in einem CGRA auch deutlich größer als die maximale Anzahl an FUs in einem VLIW-Prozessor, denn die Anzahl an Zugriffsports für ein Registerfile kann nicht beliebig weit erweitert werden.

2.3.1 Programmierung und Scheduling

Das Programmieren eines CGRAs ist eine äußerst wichtige Aufgabe, die einen erheblichen Einfluss auf die erzielte Performance des CGRAs hat. Dabei sollte versucht werden, immer so viel Parallelität wie möglich beim Scheduling auszunutzen, um die Struktur des CGRAs optimal zu verwenden. Dies ist keine triviale Aufgabe, denn es muss immer darauf geachtet werden, wo gerade ungenutzte PEs mit der notwendigen Hardware für die Operationen sind, deren Eingangswerte bereits zur

Verfügung stehen. Gleichzeitig muss dabei berücksichtigt werden, dass die Ergebnisse, im Takt nachdem die Berechnung abgeschlossen wurden, nur von den unmittelbar verbundenen PEs verwendet werden können. Muss der Wert also an ein weiter entferntes PE geliefert werden, so muss er über andere PEs weitergereicht werden, was die weiteren Berechnungen verzögert.

Um diese komplexe Programmierung durchzuführen, gibt es prinzipiell verschiedene Vorgehensweisen. Bei der manuellen Programmierung muss der Entwickler von Hand zu jedem Zeitschritt für jedes PE bestimmen, welche Operation dort ausgeführt wird. Das bietet potentiell die Möglichkeit, die beste und effizienteste Lösung zu finden, weil der Entwickler alles kontrollieren kann. Es ist allerdings auch sehr aufwendig und erfordert viel Spezialwissen, um ein gutes Ergebnis zu erzielen. Wird die Programmierung hingegen automatisch ausgeführt, so muss für den Fall, dass das CGRA als Beschleuniger für einen herkömmlichen Prozessor verwendet wird, zunächst ermittelt werden, welche Teile der Software auf das CGRA ausgelagert werden sollen. Anschließend muss auch hier festgelegt werden, wann welche Operation auf welchem PE ausgeführt wird. Hierfür wird in der Regel ein ressourcenbeschränkter Scheduling-Algorithmus verwendet, der um die Einschränkungen erweitert wird, dass Daten zunächst nur an unmittelbar benachbarten PEs verfügbar sind und ansonsten weitergeleitet werden müssen.

Die Programmierung des verwendeten CGRAs erfolgt in der Programmiersprache C. Der Code wird durch das Tool Code Parser [18] in einen *Control Flow Graph* (CFG) sowie einen *Scheduler Application Representation* (SCAR)-Graphen übersetzt. Auf diesen Graphen wird ein modifizierter *List Scheduler*-Algorithmus [19] mit den Ressourcenbeschränkungen durch die gegebene CGRA-Konfiguration angewendet, der auch die Verzögerung durch das Weiterreichen der Ergebnisse berücksichtigt. Dadurch werden die Kontextinformationen erstellt, die zum Programmieren des CGRAs in dessen Kontextspeicher hinterlegt werden müssen.

3 Aufbau des Simulators

Herzstück des HiL-Simulators ist ein *AMD/Xilinx Virtex 7 VC707 Evaluation Kit*. Die zentrale Recheneinheit auf diesem ist ein Virtex 7 *XC7VX485T-2FFG1761C* FPGA. Über eine *FPGA Mezzanine Card* (FMC)-Schnittstelle wird die Verbindung zur Außenwelt hergestellt. Dafür wird eine FMC151 Karte [14] verwendet, die zwei *Analog Digital Wandler* (ADC) und zwei *Digital Analog Wandler* (DAC) zur Verfügung stellt. Die beiden 14-Bit ADCs tasten jeweils mit einer Abtastrate von etwa $250\,$MHz das Referenz- bzw. das Gapsignal ab. Das Ausgabesignal, das gaußförmige Strahlsignal, wird über einen der beiden 16-Bit DACs ausgegeben. Mit Hilfe des zweiten DACs kann ein Überwachungssignal, wie bspw. die aktuelle Phase des Strahlsignals, ausgegeben werden, das zur Laufzeit vom Nutzer ausgewählt werden kann. Die Spannungsamplituden der analogen Signale von ADCs und DACs sind auf ein Maximum von $V_{PP} = 2\,$V beschränkt. Aus diesem Grund werden die mehrere $10\,$kV großen Spannungen, die am Beschleuniger anliegen, zunächst auf ein verträgliches Maß für die FMC151 Karte herunterskaliert.

Da es nicht möglich ist, den gesamten Versuchsaufbau der GSI für die Entwicklung des HiL-Simulators zu verwenden, wird ein *SDG 1062X* Funktionsgenerator von Siglent [13] verwendet, der zwei synchronisierte Ausgänge besitzt, mit deren Hilfe dann das Gap- und das Referenzsignal erzeugt werden können. Die Ausgangssignale aus den DACs sowie die Eingangssignale aus dem Funktionsgenerator werden zur Überwachung an ein *MSO3034* Oszilloskop von Textronix [15] angeschlossen.

3.1 FPGA Design

Das gesamte Hil-System ist im Toplevelmodul des FPGAs gebündelt. Der schematische Aufbau ist in Abbildung 3.1 als Blockdiagramm dargestellt.

Die zwei Eingangssignale werden nach der Konvertierung im ADC in einem *Ring Buffer*-Modul gespeichert, bzw. an ein Modul zur Erkennung positiver Nulldurchgänge übergeben. Der Ringpuffer speichert die Messdaten, damit diese bei der Berechnung des mathematischen Modells zur Verfügung stehen. Er muss dabei Platz für mehr als zwei Perioden auf einmal in seinem Speicher haben, damit auch für betraglich große Δt_n-Werte die Messergebnisse aus dem Puffer gelesen werden können. Im *Ring Buffer*-Modul wird die Adresse der letzten zwei positiven Nulldurchgänge gespeichert und bei einem Lesebefehl wird auf die angeforderte Adresse die Adresse des vorletzten Nulldurchgangs als Offset aufaddiert. Der Speicher ist aus BRAM aufgebaut, sodass es möglich

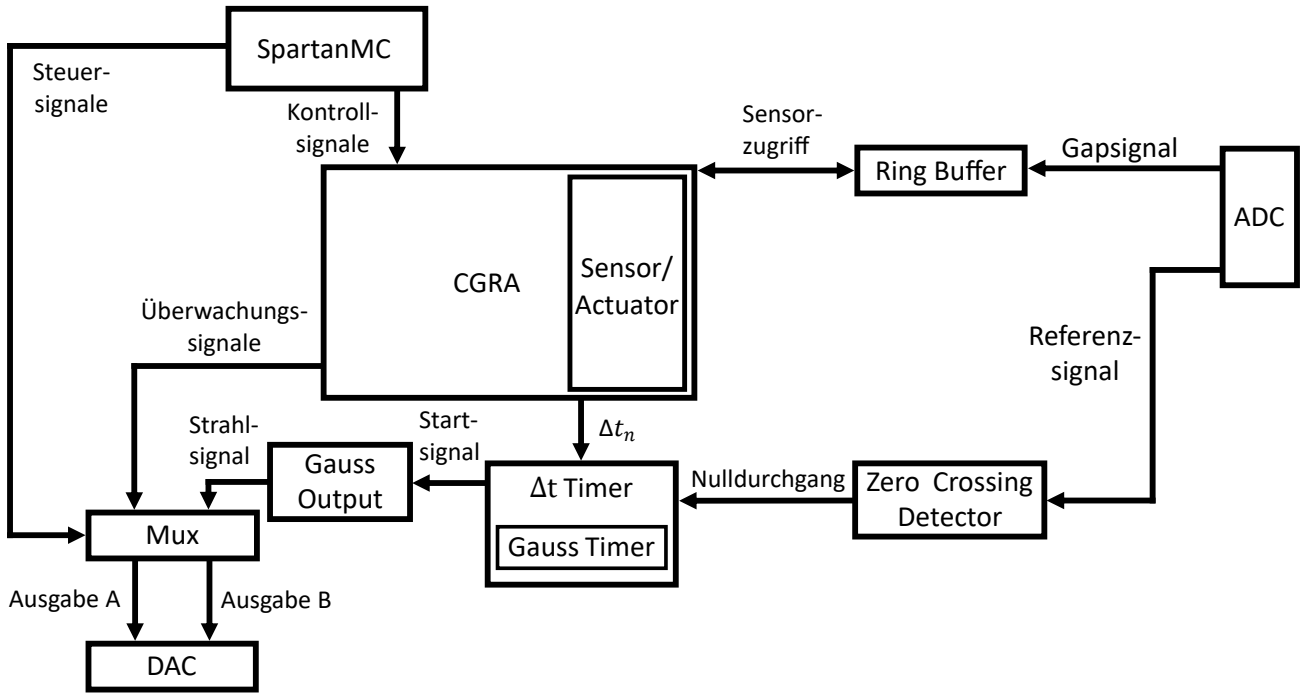


Abbildung 3.1: FPGA Blockdiagramm für den stationären Fall

ist, über den einen Port Werte aus dem Speicher auszulesen, ohne dabei das Aufzeichnen neuer Messwerte über den anderen Port zu behindern.

Das *Zero Crossing Detector*-Modul bestimmt die Nulldurchgänge, indem es die Vorzeichen der Messwerte ermittelt. Findet ein Wechsel der Vorzeichen statt, so liegt ein Nulldurchgang vor. Um robuster gegenüber Rauschen zu sein, verwendet das Module eine Hysterese, denn gerade im Bereich niedriger Spannungen ist das Rauschen betragsmäßig groß im Vergleich zur anliegenden Spannung und kann zu ungewollten Sprüngen der Messwerte über die Vorzeichengrenze führen.

Das CGRA besitzt ein *Sensor-/Actuator*-Modul für die Peripheriezugriffe. Ähnlich wie bei einem lesenden Speicherzugriff ermöglicht dieses dem CGRA, die Messergebnisse aus Peripheriekomponenten einzulesen. Auch die Ausgabewerte des CGRAs werden mit dem *Sensor-/Actuator*-Modul, ähnlich einem schreibenden Speicherzugriff, übermittelt.

Das CGRA liefert den berechneten Wert für Δt_n mittels des *Sensor-/Actuator*-Moduls an das Δt-*Timer*-Modul. Dieses erhält zusätzlich noch den Zeitpunkt des gemessenen positiven Nulldurchgangs und ermittelt damit den Startzeitpunkt für den Gaußimpuls. Dabei werden alle durch das System verursachten Verzögerungen und die Länge des Gaußimpulses berücksichtigt, damit der Scheitelpunkt des Ausgangssignals zur korrekten Zeit erreicht wird. Ist dieser Startzeitpunkt erreicht, so wird im *Gauss Output*-Modul ein vorab berechneter Gaußimpuls abgespielt. Dieser kann einfach ausgetauscht werden, um ein anderes Ausgangssignal, beispielsweise eine breitere oder spitzere Gaußkurve, zu erzeugen.

Steuern lässt sich das System mit Hilfe des *SpartanMC*-Softcoreprozessors [7]. Dies ist ein speziell für FPGAs entwickelter 18 Bit Prozessor. Der *SpartanMC* ist mit der *Universal Asynchronous Receiver/Transmitter* (UART)-Schnittstelle des *VC707 Evaluation Kits* verbunden, sodass er mit einem externen Gerät kommunizieren kann. Zum einen sendet der Softcoreprozessor regelmäßig Statusinformationen über das System an das externe Gerät. Zum anderen können von außen Befehle entgegengenommen werden, die das HiL-System steuern. Damit kann beispielsweise das Überwachnungssignal ausgewählt oder die Skalierung der Ausgabesignale angepasst werden. Für die Steuerung gibt es ein Bedienungsmenü, durch das navigiert werden kann, indem vom Anwender Zeichen über die UART-Schnittstelle gesendet werden.

Das System verfügt insgesamt über drei verschiedene *Clock Domains*. Sowohl der *SpartanMC* als auch der CGRA besitzen ihre eigene *Clock Domain*, in denen das Taktsignal eine Periodendauer von 5 ns respektive $8,75$ ns besitzt. Das restliche System arbeitet mit dem Systemtakt, der eine Periodendauer von 4 ns aufweist. Für Signale, die zwischen diesen drei Domänen wechseln, muss daher immer ein *Clock Domain Crossing*-Modul verwendet werden, um die korrekte Übertragung sicherzustellen.

4 Stationärer Fall

Dieses Kapitel befasst sich mit dem stationären Fall. Zunächst wird kurz erläutert, wie das mathematische Modell aus Kapitel 2.1.2 in ein C-Modell überführt wird. Anschließend wird auf die Änderungen eingegangen, die am bereits vorhandenen Prototypen des Fachgebiets Rechnersysteme vorgenommen wurden, um die Genauigkeit der Simulation des stationären Betrachtungsfalls zu verbessern und um Fehler zu beheben.

4.1 C Modell

Zur besseren Übersicht ist in Abbildung 4.1 der Programmablauf für den stationären Fall als Ablaufdiagramm dargestellt. Das in der Programmiersprache C geschriebene Modell des Beschleunigers für den stationären Fall beginnt mit einem Zugriff auf den Ringpuffer, um den gemessenen Spannungswert an der um Δt_n vom positiven Nulldurchgang entfernten Stelle auszulesen. Anschließend wird gemäß Gleichung (2.2) ein Wert für $\Delta\gamma_n$ bestimmt und aus diesem mit Gleichung (2.1) β_n ermittelt. Die beiden Lorentzfaktoren werden nun in Gleichung (2.5b) eingesetzt, um daraus den neuen Wert für Δt_n zu berechnen. Anschließend wird dieser zurückgeschrieben und es wird auf das Feststellen des nächsten positiven Nulldurchgangs der Referenzspannung gewartet, bevor die Rechnung von vorne losgeht.

4.2 Softwareanpassungen

Ein Test der Mitarbeitenden des Fachgebiets Rechnersysteme bei der GSI mit dem Prototypen hat gezeigt, dass ein kontinuierliches Springen zwischen den Phasendifferenzen +45° und -45° der Gapspannung gegenüber der Referenzspannung zu einem unendlich anwachsendem Δt_n geführt hat. Dies lässt sich mit Abbildung 4.2 erklären. Die Abbildung zeigt verschiedene mögliche Orbits des Bunches. Innerhalb des schwarzen Orbits sind die Kurven geschlossen, d.h. die Phase bleibt beschränkt. Steigt die Energie durch die Phasensprünge allerdings weit genug an, kann der Bunch auch auf eine der äußeren Kurven gelangen. In diesem Fall steigt die Phase $\Delta\varphi_{RF}$, und somit natürlich auch Δt_n, immer weiter an, weil der Bunch nicht in einem *Bucket* verbleibt, sondern immer wieder in den nächsten *Bucket* wechselt. Ein *Bucket* ist dabei der Bereich $(-180°, 180°]$ um

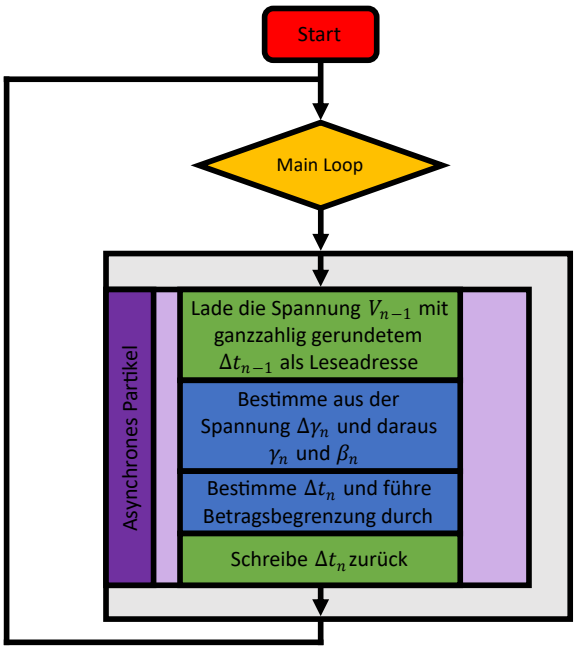


Abbildung 4.1: Ablaufdiagramm des Programms im stationären Fall. Sensor-/Aktuatorzugriffe sind grün dargestellt, Schleifenköpfe sind in orange und Berechnungen sind in blau abgebildet

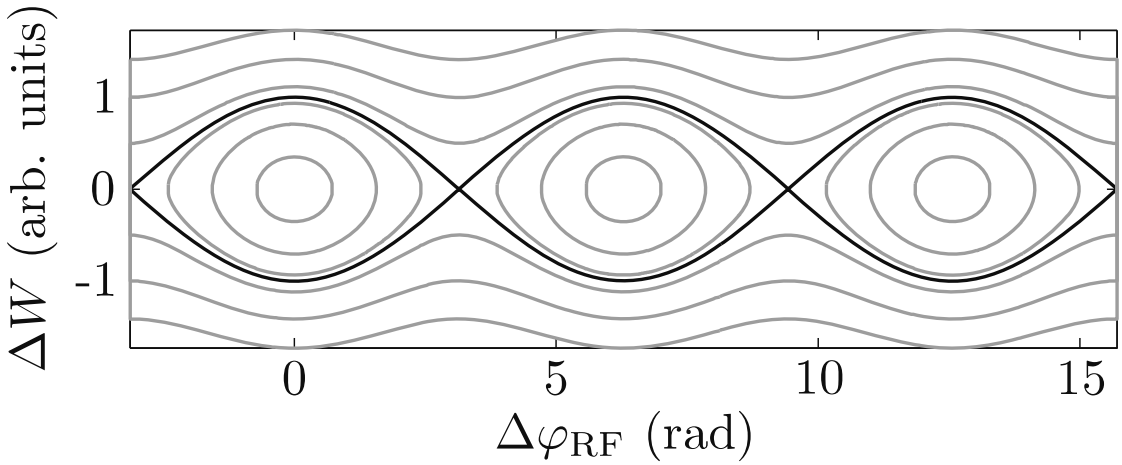


Abbildung 4.2: Abbildung verschiedener Orbits im Longitudinal-Phasen-Raum [10]

den Nulldurchgang des Referenzteilchens, also der Bereich, in dem die Spannungen so wirken, dass das Teilchen wieder in Richtung des Referenzteilchens beschleunigt wird.

Um dieses Problem zu lösen, wurde eine einfache Änderung im Programm vorgenommen, die den Betrag von Δt_n um $\frac{1}{f_R}$ reduziert, wenn der Betrag größer als $\frac{1}{f_R}$ ist. Somit ist die Phase $\Delta\varphi_{RF}$ immer im Bereich $[-2\pi, +2\pi]$. Dies sorgt allerdings dafür, dass bei Sprüngen um $+\frac{1}{f_R}$ ein Gaußimpuls verloren geht, während durch Sprünge um $-\frac{1}{f_R}$ zwei Gaußimpulse zur gleichen Zeit stattfinden. Letzteres ist kein Problem, da trotzdem nur ein Gaußimpuls erzeugt wird. Für ersteres wurde eine Spezialfallbehandlung in Hardware umgesetzt.

4.3 Hardwareanpassungen

Eine wichtige Anpassung der Hardware ist die Korrektur eines Fehlers im *Gaus Timer*-Modul. Hier trat ein Fehler auf, wenn Δt_n negativ und betragsmäßig so groß war, dass das Startsignal für das *Gauss Output*-Modul vor dem aktuellen Nulldurchgang der Referenzspannung gesendet werden müsste. Das hatte dann zur Folge, dass nur in jeder zweiten Schwingungsperiode der Referenzspannung ein Gaußimpuls ausgelöst wurde und zwar immer zum Zeitpunkt des positiven Nulldurchgangs.

Das lag daran, dass im *Gaus Timer*-Modul ein Trigger verwendet wird, um den Startzeitpunkt zu bestimmen. Wenn der Timer denselben Wert wie der Trigger aufweist, wird das Startsignal gesendet. In diesem Problemfall war ein negativer Wert im Trigger gespeichert und somit wies der Wert eine Eins als *Most Significant Bit* (MSB) auf. Die Periodendauer ist allerdings nie so lang, dass der Timer einen Wert mit einer Eins als MSB erreichen könnte. Dementsprechend sollte nie ein Gaußimpuls abgespielt werden. Es wurde dann aber eine Spezialfallbehandlung angewendet, die für den Fall gedacht war, dass der Impuls genau zum Zeitpunkt des nächsten Nulldurchgangs gestartet werden soll. Dort konnte der Trigger verloren gehen, wenn die nächste Periode einen Takt kürzer als die aktuelle war. Deshalb hat die Spezialfallbehandlung einen Impuls zum Zeitpunkt des positiven Nulldurchgangs ausgelöst, wenn während des gesamten vorherigen Umlaufs kein Gaußimpuls gestartet wurde. Für die negativen Triggerwerte, die nie erreicht werden, hat die Spezialfallbehandlung dann dementsprechend in jedem zweiten Umlauf einen Gaußimpuls ausgelöst, weil in der gesamten vorherigen Periode kein Impuls abgespielt wurde. Ursache für den Fehler war, dass nicht davon ausgegangen wurde, dass so große Phasenverschiebungen auftreten können (zum Auslösen des Bugs muss die Phase kleiner als $-180°$ sein).

Behoben wurde das Problem, indem der Triggerwert mit der Periodendauer addiert wird. Dadurch werden die negativen Zahlen wieder zu positiven Zahlen. Allerdings vergrößert sich somit die Latenz um eine zusätzliche Periodendauer, und es müssen zwei Timer und zwei Trigger existieren, weil der neu berechnete Trigger erst für die nächste Periode den Impuls auslöst, aber der vorherige Wert ist für die aktuelle Periode noch relevant und kann nicht überschrieben werden.

Eine weitere Spezialfallbehandlung wird für den Fall eingeführt, dass Δt_n zu groß wird und somit der Impuls in die nächste Sinusperiode fallen würde. Um mit diesem Fall umzugehen, wird für große Werte von Δt_n die zuvor beschriebene Addition einer Periodendauer zum Triggersignal ausgelassen. Ein Hin- und Herspringen wird dabei vermieden, indem für den Wechsel zwischen Periodendauer Addieren oder Nicht-Addieren eine Hysterese eingeführt wird. Eine zusätzliche Sicherheitsmaßnahme ist, dass nicht erst dann auf die Addition der Periodendauer verzichtet wird, wenn es unbedingt notwendig wäre. Stattdessen wird bereits auf den kleineren Triggerwert gewechselt, sobald der Trigger einen Schwellwert überschreitet. Der Schwellwert hat immer einen festen Sicherheitsabstand zur doppelten Periodendauer. Dieser Sicherheitsabstand kann als Parameter übergeben werden. Somit wird festgelegt, wie viel früher dieser Wechsel erfolgen soll.

Eine solche Verschiebung um eine Periodendauer entspricht demselben Verhalten, dass auch schon in Kapitel 4.2 bei der Beschränkung der Phase auf den Bereich $[-2\pi, +2\pi]$ beschrieben wurde. Daher geht in der soeben beschriebenen Spezialfallbehandlung beim Wechsel von dem

Trigger 1 und 2	Trigger 3 (Phasensprungtrigger)
Für den Normalbetrieb zuständig. Ein Trigger erzeugt nur jeden zweiten Impuls. Wechselt den Offset, um sowohl große als auch kleine Werte von Δt_n entgegennehmen zu können.	Füllt die Lücke bei Sprüngen vom kleinen zum großen Offset und bei positiven Sprüngen von Δt_n (softwarebedingt).

Tabelle 4.1: Übersicht über die verwendeten Trigger.

Fall, in dem keine Periodendauer auf den Trigger addiert wird, hin zu dem Fall, bei dem sie addiert wird, ebenfalls ein Gaußimpuls verloren. Deshalb wird ein dritter Trigger eingeführt, der bei einem solchen Wechsel aktiviert wird. Dieser Trigger wird auf den Wert ohne Addition der Periodendauer gesetzt, sodass er die Lücke füllt. Außerdem wird dieser Trigger auch aktiviert, wenn in der Software der Phasensprung um $+2\pi$ durchgeführt wird. Da sich Δt_n im Regelbetrieb nur kontinuierlich ändert, kann ein solcher Sprung einfach festgestellt werden, indem die Differenz zwischen dem vorherigen Δt_n-Wert und dem aktuellen gebildet wird. Ist der Wert größer als ein Schwellwert, muss ein zusätzlicher Gaußimpuls ausgelöst werden. In diesem Fall wird der Trigger auf den Wert vor dem Sprung gesetzt, um den fehlenden Impuls zu erzeugen. Zur besseren Übersicht sind die drei Trigger in Tabelle 4.1 nochmal aufgelistet.

Zusätzlich wurde eine verbesserte Implementierung der Gleitkommaoperationen auf dem CGRA in das Projekt eingepflegt. Diese sogenannten Floating-Point Cores (FloPoCo)-Operationen [5] arbeiten auf 34 Bit breiten Worten anstatt mit der üblichen Bitbreite von 32 Bit. Die zusätzlichen zwei Bits dienen dazu, Spezialfälle der Gleitkommaarithmetik effizienter darzustellen. Die vier Zustände, die unterschieden werden können, sind: *Null*, *Not a Number* (NaN), *Unendlich* (positiv oder negativ) und *gewöhnliche Zahl*. Der letzte Fall entspricht einer herkömmlichen Behandlung der nachfolgenden 32 Bit gemäß des IEEE Floating-Point-Standards [8]. Das Abfangen der drei Spezialfälle macht die Hardware der Gleitkommaarithmetik wesentlich schneller und kleiner. Das Abspeichern der zwei zusätzlichen Bits ist auch kein nennenswerter Nachteil, weil der BRAM auf dem FPGA ohnehin 36-Bit breit ist.

Ein weitere Fehler, der behoben wurde, ist, dass die Hysterese zur Bestimmung der Nulldurchgänge des Referenzsignals nur in eine Richtung funktioniert hat. Ist ein einziger negativer Wert aufgetreten, so wurde das Signal sofort als negativ angenommen. In die andere Richtung wurde immer eine Hysterese durchlaufen, es mussten also mehrere Werte hintereinander positiv sein, damit das Signal auch als positiv angenommen wurde. Das hat zu Fehlern geführt, wenn kurz nach dem positiven Nulldurchgang (also bei niedriger Spannungsamplitude) ein stark negativer Rauschwert auftritt. Dieser hat dann das Signal einen Takt lang negativ werden lassen, sodass es mangels Hysterese auch als negativ angenommen wurde. Da die Messwerte danach wieder zumeist positiv sind, wird ein weiterer Nulldurchgang detektiert. Aus diesem Grund wird die Hysterese auch für den Wechsel zu negativen Werten eingeführt.

Die Hardwareanpassungen wurden ausführlich mit *Modelsim* [12] anhand mehrere Testbenches validiert und wurden im Laborexperiment mit einem Oszilloskop auf ihre Richtigkeit überprüft.

5 Ramp Up

Nachdem das Modell für den stationären Fall angepasst worden ist, ist der nächste Schritt, es um den Beschleunigungsfall zu erweitern. In diesem sogenannten *Ramp Up*-Szenario wird die Umlauffrequenz des Bunches immer weiter erhöht, indem die Frequenzen und die Amplituden der Gap- sowie der Referenzspannung kontinuierlich gemäß eines vorgegebenen Verlaufs, wie beispielhaft in Abbildung 5.1 dargestellt, angepasst werden. Um dies realisieren zu können, müssen eine Reihe an Änderungen am Modell, der Sofwareimplementierung und an der Hardware durchgeführt werden.

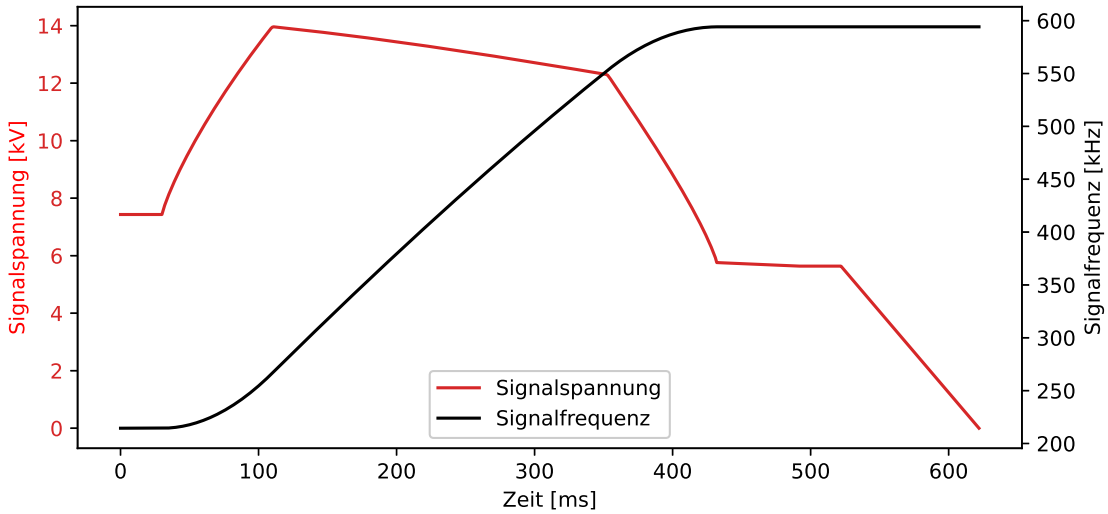


Abbildung 5.1: Verlauf von Spannung und Frequenz eines *Ramp Up*-Szenarios, wie er in einer Simulation der GSI verwendet wurde

5.1 Modellanpassungen

Aufgrund der nun variablen Frequenz der Eingangsspannungen sind einige Vereinfachungen, die in Kapitel 2.1.2 getroffen wurden, nicht mehr gültig. Dies betrifft vor allem die Parameter des Referenzpartikels, die nun nicht mehr als Konstanten betrachtet werden können und fortan zur Laufzeit berechnet werden müssen.

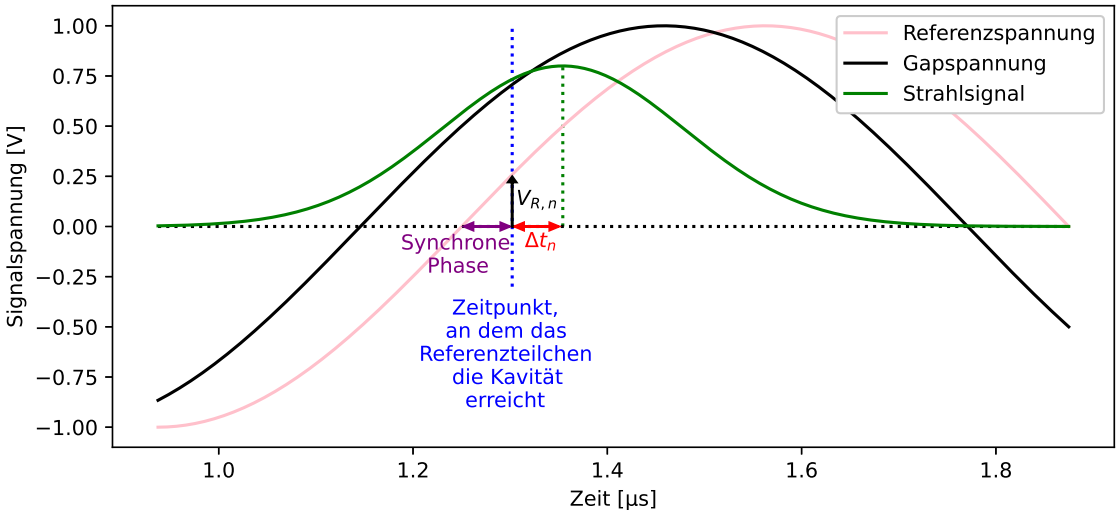


Abbildung 5.2: Beispielhafte Verläufe für Referenz- und Gapspannung sowie für das Strahlsignal während des *Ramp Up*-Szenarios

Ausgangspunkt für die Modellanpassungen ist, dass die Spannung $V_{R,n}$ nicht mehr dauerhaft als Null angesehen werden kann, denn durch die sich ändernde Frequenz gilt nicht mehr, dass das Referenzpartikel immer zum positiven Nulldurchgang der Referenzspannung die Kavität erreicht.

Eine Folge von $V_{R,n} \neq 0V$ ist, dass die Lorentzfaktoren des Referenzpartikels variabel sind und nach jedem Umlauf neu ermittelt werden müssen. $\gamma_{R,n}$ kann dabei aus Gleichung (5.1) gewonnen werden, und daraus kann anschließend mit Hilfe von Gleichung (2.1) $\beta_{R,n}$ bestimmt werden.

$$\gamma_{R,n} = \gamma_{R,n-1} + \frac{Q}{m_0 c_0^2} V_{R,n-1} \tag{5.1}$$

Da $\gamma_{R,n}$ nicht mehr konstant ist, muss auch $\eta_{R,n}$ in jedem Umlauf gemäß Gleichung (2.4a) neu berechnet werden.

Des Weiteren muss die Spannung $V_{R,n}$ auch in Gleichung (2.2) berücksichtigt werden. Daher gilt die in Kapitel 2.1.2 beschriebene Vereinfachung, dass ΔV_{n-1} mit V_{n-1} ersetzt werden kann, nicht mehr. Stattdessen muss ΔV_{n-1} nun aus der Differenz der Spannungen, die das Referenzteilchen und das asynchrone Teilchen an der Kavität sehen, ermittelt werden. In Abbildung 5.2 ist ein Signalverlauf für die Ein- und Ausgangssignale des Systems, wie es im *Ramp Up*-Szenario vorkommen kann, beispielhaft dargestellt. An dieser Abbildung lässt sich erkennen, dass $V_{R,n} = 0V$ nicht mehr vorausgesetzt werden kann, wenn das Referenzteilchen nicht beim positiven Nulldurchgang der Referenzspannung die Kavität erreicht.

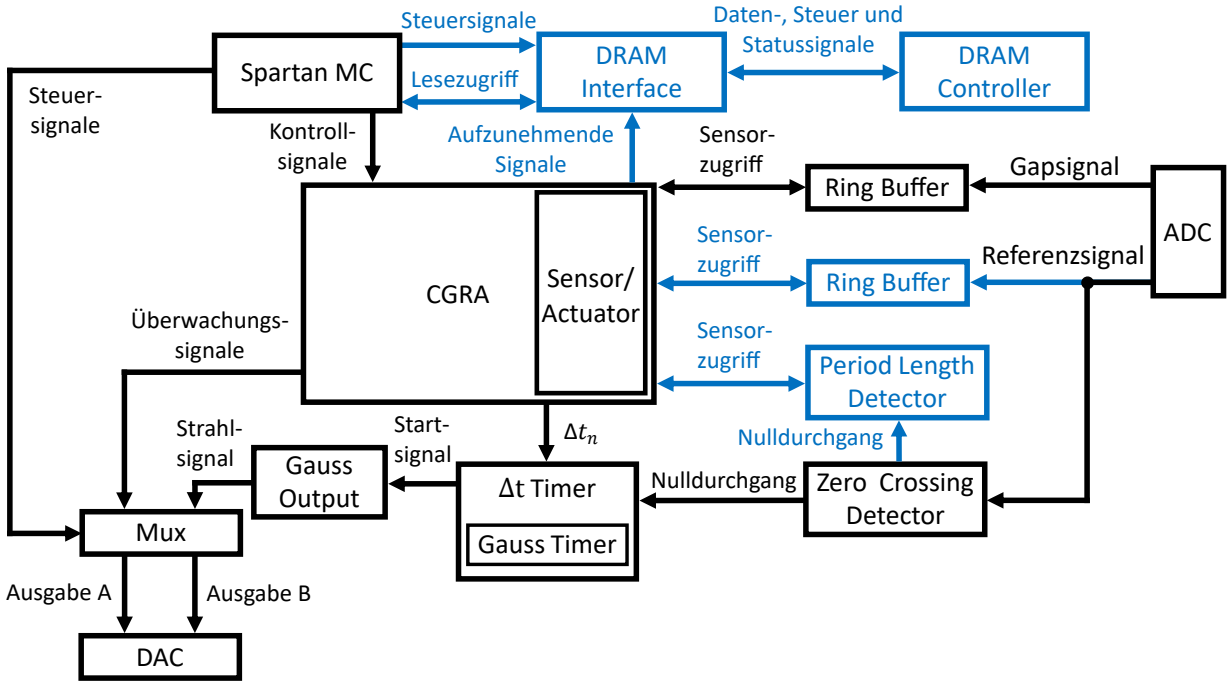


Abbildung 5.3: FPGA Blockdiagramm für das *Ramp Up*-Szenario. Neue Elemente sind in blau dargestellt

5.2 Hardwareanpassung

Für den *Ramp Up*-Fall werden einige Hardwareänderungen vorgenommen, damit die Änderungen im Modell umgesetzt werden können. Einen Überblick über das Gesamtsystem liefert Abbildung 5.3, die das FPGA-Design nach den Änderungen für das *Ramp Up*-Szenario darstellt.

Zunächst wird ein zweiter Ringpuffer ergänzt, der die Referenzspannung aufzeichnet, damit diese vom CGRA für die Berechnung des aktualisierten Modells ausgelesen werden kann. Damit das CGRA auf den neuen Ringpuffer zugreifen kann, wird das CGRA um einen zweiten Sensorleseport ergänzt. Dafür werden kleine Anpassungen an den Dateien *CfgExpressionVisitor.java*, *GsiCgra.json*, *SensorActuatorAccess.sv* und *SensorActuatorAccessOperations.kt* vorgenommen. Dadurch wird ein neuer Opcode für den Zugriff auf den zweiten Ringpuffer zum Befehlssatz hinzugefügt sowie die Bezeichnung der Methode festgelegt, mit der im C-Code der Lesebefehl ausgeführt werden kann. Das *SystemVerilog*-Modul sorgt dann dafür, dass das CGRA korrekt mit der Außenwelt interagiert, wenn der Befehl für das Lesen des zweiten Ringpuffers ausgeführt wird.

Des Weiteren wird ein neues Modul ergänzt, das die Periodendauer des Referenzsignals bestimmt. Dieses Modul speichert die letzten vier gemessenen Periodenlängen des Referenzsignals ab und liefert die Addition dieser vier Periodenlängen zurück. Durch die Addition wird ein Mittelwert gebildet, der in einer Fixkommadarstellung mit zwei Bit Nachkommastellen angegeben ist. Damit kann die Periodenlänge etwas genauer dargestellt werden, weil sie jetzt in Vierteltaktperioden

Signal	Reservierte Adressen
Nächster Nulldurchgang?	0x7FFFFFFF
Umlaufperiodendauer	0x7FFFFFFE
Gültiges Eingangssignal?	0x7FFFFFFD
DRAM Schreibadressen	0x7FFFFFE0-0x7FFFFFE4

Tabelle 5.1: Reservierte Adressen im *Ramp Up*-Szenario

quantisiert ist, obwohl sie immer nur als ganzzahliges Vielfaches der Abtasttaktperiode gemessen werden kann. Eingelesen werden kann die gemessen Periodendauer vom CGRA, indem eine Leseanfrage an eine dafür reservierte Adresse geschickt wird (vgl. Tabelle 5.1).

Im neuen Ringpuffer wird außerdem ermittelt, ob ein gültiges Nutzsignal anliegt. Das Signal gilt als gültig, wenn die Spannung innerhalb einer Periode einen Wert erreicht, der über einem Schwellwert liegt. Sobald das Signal in der gesamten vorherigen Periode den Schwellwert nicht überschritten hat, wird das Gültigkeitssignal wieder zurückgesetzt. Dieses Gültigkeitssignal wird an drei Module weitergeleitet.

Zum einen wird es im *Period Length Detector*-Modul verwendet. Solange das Gültigkeitssignal nicht anliegt, werden alle gespeicherten Umlaufperiodenlängen auf Null gesetzt. Die erste gemessene Periodenlänge, bei der das Gültigkeitssignal vorhanden ist, wird immer verworfen, da es sein kann, dass in diesem Umlauf das Resetsignal noch gesetzt war, weshalb es möglich ist, dass nicht die gesamte Periodendauer erfasst worden ist. Das Zurücksetzen der gemessen Werte auf Null ermöglicht auch das einfache Überprüfen, ob eine gültige gemessene Periodendauer vorhanden ist, indem überprüft wird, ob die erste abgespeicherte Periodenlänge ungleich Null ist. Dies ist eine wichtige Information, da die Anpassungen am Modell eine Initialisierung der Lorentzfaktoren erfordert. Das Wissen, ob gültige Messwerte vorliegen, wird daher benötigt, um festzustellen, wann die Initialisierung gestartet werden kann. Das Abfragen dieses Wertes erfolgt wieder über eine Leseanfrage des CGRAs an eine hierfür reservierte Adresse (vgl. Tabelle 5.1).

Weiterhin wird auch das CGRA zurückgesetzt, wenn das Gültigkeitssignal nicht mehr gesetzt ist. Damit wird dafür gesorgt, dass die im CGRA intern abgespeicherten Werte zurückgesetzt werden und dass, sobald ein neues Nutzsignal anliegt, der Code zur Initialisierung neu durchgeführt wird. Das ermöglicht es, dass die FPGA-Konfiguration nicht jedes Mal neu geladen werden muss, bevor ein neues Signal angelegt werden kann.

Schließlich wird das Gültigkeitssignal auch noch verwendet, um das Ausgeben von Gaußimpulsen zu unterdrücken, solange kein gültiges Eingangssignal vorliegt.

In einem weiteren Optimierungsschritt wird die Bestimmung des Gültigkeitssignals und das damit verbundene Setzen des Resetsignals, wenn das Gültigkeitssignal nicht vorhanden ist, in ein eigenes Modul, das *Signal Gone Detector*-Modul, ausgelagert. Um die Übersichtlichkeit in Abbildung 5.3 und den später folgenden Blockdiagrammen zu verbessern, wird das *Signal Gone Detector*-Modul nicht in den Blockdiagrammen dargestellt.

5.2.1 DRAM-Ansteuerung

Im Gegensatz zum stationären Fall ist es beim *Ramp Up*-Szenario nicht trivial, die Qualität des HiL-Simulors am Oszilloskop zu beurteilen, denn wie in Abbildung 5.1 zu erkennen, dauert das ganze *Ramp Up*-Szenario nur den Bruchteil einer Sekunde. Gleichzeitig ist das *Ramp Up*-Szenario aber auch zu lang, um es mit Hilfe von *Modelsim* vollständig analysieren zu können. Aus diesem Grund wurde das FPGA-Design um eine Ansteuerung des *Dynamic Random Access Memorys* (DRAM) auf dem VC707 Evaluation Kit erweitert, wodurch es möglich wird, Ergebnisse aus der Modellberechnung aufzuzeichnen und und diese später wieder auszulesen.

MIG IP-Core

Für die Kommunikation mit dem DRAM-Chip wird der *Memory Interface Generator* [16] (MIG)-IP-Core von Xilinx verwendet. Dieser IP-Core kann mit Hilfe eines Wizards direkt in Vivado erzeugt werden. Dabei muss unter anderem der konkrete auf dem Board verbaute DRAM-IC angegeben werden, und es muss festgelegt werden, welche Pins des FPGAs mit welchen Pins des DRAMs verbunden sind. Außerdem muss bestimmt werden, ob der DRAM mit der doppelten oder der vierfachen Frequenz des MIG-Controllers betrieben wird. In dieser Arbeit wird die vierfache Frequenz verwendet. Daraus ergibt sich eine Wortbreite für Lese- und Schreibdaten von jeweils 512 Bit. Im MIG-IP-Core ist ein Modul zur Takterzeugung verbaut, das aus dem Takt des externen Oszillators den Systemtakt für den DRAM und den Controller erzeugt. Der DRAM wird mit 800 MHz betrieben. Dementsprechend arbeitet der Controller mit einer Frequenz von 200 MHz. Da bis jetzt noch keine Systemkomponente diese Frequenz verwendet hat, gibt es nun eine weitere *Clock Domain* und beim Übertragen von Signalen vom oder zum *DRAM Controller* muss der Wechsel der *Clock Domain* beachtet werden.

Der MIG bietet drei verschiedene Optionen, wie die Schnittstelle aussehen soll. Ein Möglichkeit ist die Verwendung eines *Advanced eXtensible Interface* (AXI4) in der Slave-Konfiguration [2], welche einfach von einem AXI-Master angesprochen werden kann. Da im bisherigen Design noch kein AXI-Bus verbaut ist, wurde diese Lösung nicht umgesetzt.

Die anderen beiden Schnittstellenmöglichkeiten sind das *Native Interface* und das *User Interface*. Diese ähneln sich sehr und unterscheiden sich hauptsächlich dadurch, dass das *Native Interface* mehr Low-Level-Kontrolle bietet, die in diesem Anwendungsfall allerdings nicht benötigt wird. Daher wird das *User Interface* des MIGs für die Anbindung an das bisherige Design verwendet. Tabelle 5.2 liefert einen Überblick über die Signale des *User Interfaces*. Nicht benötigte Signale, die auch in der Implementierung nicht angeschlossen sind, wie z.B. das Signal zum manuellen Refresh des DRAMs, werden dabei weggelassen.

Die Abbildungen 5.4 und 5.5 zeigen bei einem Frequenzverhältnis von 4:1 den zeitlichen Signalablauf für Lese- und Schreibzugriffe auf das *User Interface*, bei denen jeweils immer nur ein Wort gelesen oder geschrieben wird. Für einen Schreibzugriff müssen **app_rdy** und **app_wdf_rdy** gesetzt sein. Gleichzeitig müssen im selben Takt **app_en** auf 1, **app_cmd** auf 000 und **app_addr** auf die richtige Adresse gesetzt werden. Die Schreibdaten in **app_wdf_data** müssen gleichzeitig

mit dem **app_wdf_wren**- und dem **app_wdf_end**-Signal angelegt werden. Dies kann bis zu zwei Takte nach dem Setzen von **app_en**, **app_cmd** und **app_addr** geschehen, aber auch gleichzeitig oder vorher.

Beim Lesezugriff müssen wie beim Schreiben **app_en**, **app_cmd** und **app_addr** im selben Takt gesetzt werden und dabei muss **app_rdy = 1** gelten. In einem späteren Takt liegen dann die Lesedaten an **app_rd_data** an und das **app_rd_data_valid**-Signal zeigt an, dass die anliegenden Daten korrekt sind.

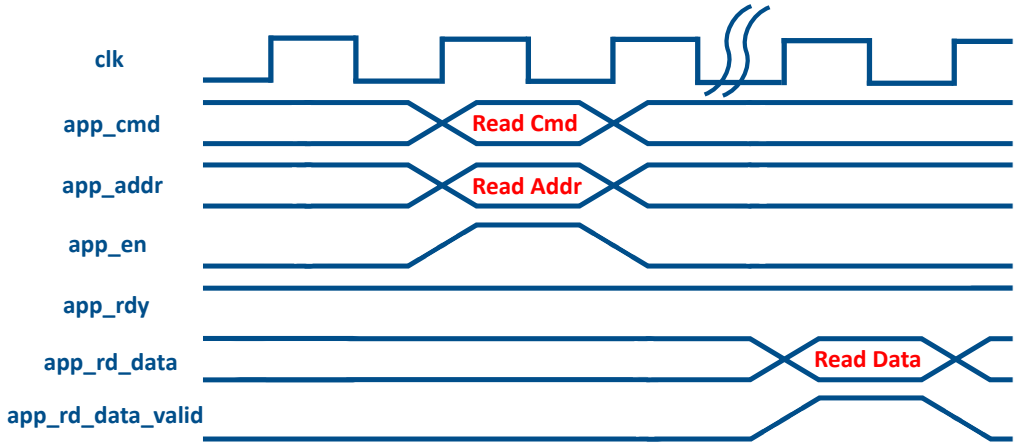


Abbildung 5.4: Signalverlauf zum Lesen von Daten aus dem DRAM mit Hilfe des *User Interfaces* (erstellt in Anlehnung an [16])

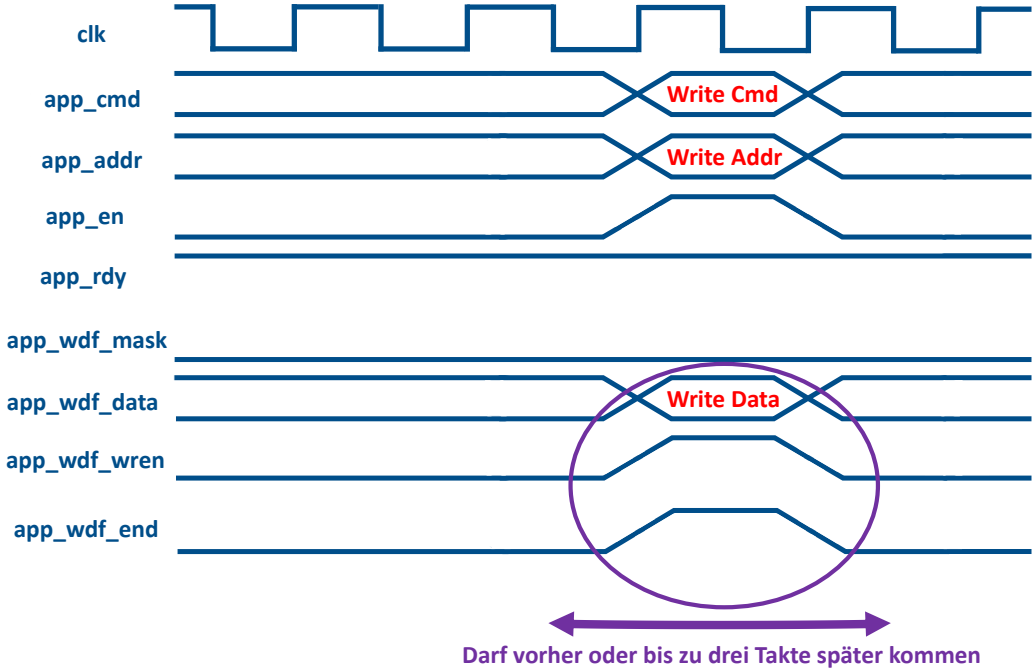


Abbildung 5.5: Signalverlauf zum Schreiben von Daten in den DRAM mit Hilfe des *User Interfaces* (erstellt in Anlehnung an [16])

Signal	Richtung	Beschreibung
app_addr [ADDR_WIDTH-1:0]	Input	Adressignal für die aktuelle Anfrage
app_cmd[2:0]	Input	wählt den aktuellen Befehl aus (001 für Lesen, 000 für Schreiben)
app_en	Input	Enablesignal für **app_cmd** und **app_addr**
app_rdy	Output	zeigt an, dass neue Befehle entgegengenommen werden können
app_rd_data [APP_DATA_WIDTH-1:0]	Output	gelesene Daten aus dem DRAM
app_rd_data_end	Output	signalisiert, dass das letzte Lesedatum gesendet wurde
app_rd_data_valid	Output	signalisiert, dass die Daten in **app_rd_data** korrekt sind
app_wdf_data [APP_DATA_WIDTH-1:0]	Input	Daten, die in den DRAM geschrieben werden sollen
app_wdf_data_end	Input	signalisiert, dass die anliegende Schreibdaten die letzten für den aktuellen Schreibzyklus sind
app_wdf_mask [APP_MASK_WIDTH-1:0]	Input	Maske, die angibt, welche Wörter von **app_wdf_data** geschrieben werden sollen
app_wdf_rdy	Output	gibt an, dass im Schreib-FIFO neue Daten angenommen werden können
app_wdf_wren	Input	Enablesignal für Schreibzugriff
ui_clk	Output	Taktsignal, mit dem das User-Interface betrieben wird
init_calib_complete	Output	signalisiert, dass die Kalibrierung des DRAMs abgeschlossen ist

Tabelle 5.2: Signale des MIG-*User Interfaces* [16]

DRAM-Controller-Modul

Die Schnittstelle zum MIG-*User Interface* wird vom *DRAM Controller*-Modul angesteuert. Dieses Modul nimmt dabei ein Signal vom DRAM-Interface-Modul (vgl. nächsten Abschnitt) entgegen, das ihm mitteilt, ob gerade Daten aufgezeichnet oder ausgelesen werden.

Die Schreibdaten werden dem *DRAM Controller* in Form von 32 Bit breiten Wörtern übergeben. Wenn korrekte Daten anliegen, muss ein *Valid*-Bit gesetzt werden, um die Gültigkeit der Daten anzuzeigen. Der Controller bestätigt den Empfang durch das Setzen eines *Received*-Bits. Die 32 Bit breiten Wörter werden im *DRAM Controller*-Modul zu 512 Bit breiten Wörtern zusammengesetzt und anschließend in einer *First In First Out* (FIFO)-Warteschlange gespeichert. Solange diese nicht leer ist, überprüft der Controller die **app_rdy**- und **app_wdf_rdy**-Signale und sendet immer das erste Element der Warteschlange, falls diese Signale beide gesetzt sind.

Eine Aufzeichnung wird initialisiert, indem das *DRAM Controller*-Modul vom *DRAM-Interface-Modul* das Signal zum Start einer Aufzeichnung erhält. Wenn somit in den Schreibmodus gewechselt wird, wird die aktuelle Schreibadresse auf Null zurückgesetzt, um den ganzen Adressraum zur Verfügung zu haben. Ansonsten wird die Adresse nach jedem gesendeten Wort um 8 inkrementiert. Das liegt daran, dass der auf dem VC707 verbaute DRAM-IC über 64 Datenleitungen mit dem FPGA verbunden ist, sodass bei vierfacher Frequenz mit Double Data Rate insgesamt 8 mal 64 Bit in einem Takt der langsameren *Clock Domain* des *DRAM Controllers* übertragen werden und für jedes dieser 64 Bit Wörter wird eine eindeutige Adresse verwendet. Da immer alle 512 Bit bei einem Schreibzugriff gesendet werden sollen, wird die **app_wdf_mask** dauerhaft auf null gesetzt. Das *DRAM Controller*-Modul besitzt als zusätzlichen Eingang auch das in Kapitel 5.2 beschriebene Signal, das angibt, ob ein gültiges Nutzsignal anliegt. Wenn dieses Signal auf Null wechselt, wird nach außen das Signal gesetzt, dass der Speicher voll ist, sodass der Aufzeichnungsprozess beendet wird.

Ist der Controller im Lesemodus, überprüft er, ob seine FIFO-Warteschlange für Lesedaten nicht voll ist. Ist das der Fall, sendet er, wenn das **app_rdy**-Signal gesetzt ist, eine Leseanfrage, wie sie in Abbildung 5.4 dargestellt wird. Die erhaltenen Daten werden in der FIFO-Warteschlange abgespeichert. Der Controller besitzt ein Signal, mit dem neue Lesedaten angefordert werden können. Ist die FIFO-Warteschlange nicht leer, so legt der Controller bei einer externen Anfrage das erste Datum aus der Warteschlange an den Ausgang an und setzt das Signal, dass diese korrekt sind. Die Sendedaten sind dabei wieder nur 32 Bit breit, weshalb der Controller die gelesenen 512 Bit breiten Wörter in 16 Wörter zerteilt, bevor er sie sendet. Auch der Start des Auslesens wird wieder durch das Signal vom *DRAM-Interface-Modul* signalisiert. Die Leseadresse wird bei einem Wechsel in den Lesemodus auf Null zurückgesetzt, damit die ganze Aufzeichnung ausgelesen wird. Wie beim Schreiben auch, wird die Adresse nach jedem Lesezugriff um 8 inkrementiert. Beim Aufzeichnen der Daten wird sich gemerkt, was die letzte beschriebene Adresse ist, damit die Daten beim Auslesen nur bis zu dieser Adresse wieder gelesen werden.

DRAM Interface-Modul

Die Kommunikation zwischen dem *DRAM Controller* und dem CGRA bzw. dem *SpartanMC* wird über das *DRAM Interface*-Modul ermöglicht. Vom *SpartanMC* werden zwei Steuersignale an dieses Modul gesendet, um ihm entweder den Befehl zum Datenaufzeichnen oder -auslesen geben zu können. Die Signale werden vom Nutzer gesetzt, indem im Bedieninterface vom *SpartanMC* eine Messung gestartet wird. Dann wird das Signal zum Aufzeichnen gesetzt. Nach Abschluss der Aufzeichnung sendet der *SpartanMC* automatisch das Signal zum Auslesen. Diese zwei Steuersignale werden an das *DRAM Controller*-Modul weitergereicht, um in den Lese- bzw. den Schreibmous zu wechseln.

Das *DRAM Interface*-Modul nimmt die Daten- und Adressleitungen vom *Sensor-/Actuator*-Moduls des CGRAs sowie das dazugehörige *Valid*-Signal entgegen. Das CGRA sendet pro Iteration der Hauptschleife des Modells fünf Werte zum Aufzeichnen an den DRAM (V_n; $V_{R,n}$; die Umlauffrequenz des Referenzpartikels; die Anzahl der Takte, in der das Referenzpartikel bzw. das asynchrone Partikel gegenüber dem Nulldurchgang der Referenzspannung versetzt die Kavität erreichen).

Ist die CGRA-Schreibadresse bei gesetztem *Valid*-Signal eine der fünf reservierten Adressen für das Schreiben der aufzuzeichnenden Daten (vgl. Tabelle 5.1), werden die Daten gespeichert. Um Speicherplatz zu sparen, wird dabei nicht für jedes Datum das ganze 32 Bit Wort gespeichert. Stattdessen wird das Wissen über die Maximalwerte der jeweiligen Daten ausgenutzt, um die benötigte Bitbreite zu ermitteln, und nur entsprechend große Wörter werden gespeichert. Insgesamt müssen somit nur 75 Bit abgespeichert werden. Sind alle Daten für diese Iteration der Hauptschleife des Modells erfasst worden, werden sie zu einem einzigen 75 Bit breitem Wort zusammengesetzt. Dieses wird über einen *Clock Domain Crossing Handshake* von der CGRA *Clock Domain* auf die *Clock Domain* des *DRAM Controllers* übersetzt. Erhält das *DRAM Interface*-Modul den Befehl zum Aufzeichnen der Daten, werden die Daten nach dem *Clock Domain Crossing* an den *DRAM-Controller*, wie in Abschnitt 5.2.1 beschrieben, übermittelt. Durch das Reduzieren der Bitbreiten und das anschließende Zusammenführen der fünf Werte wird erreicht, dass die 75 Bit mit Hilfe von nur drei anstatt von fünf 32 Bit breiten Wörtern an den *DRAM Controller* gesendet werden können.

Ist der Befehl zum Lesen gesetzt, so werden Daten vom *DRAM Controller* angefordert. Sind drei Wörter gelesen worden, werden sie wie zuvor zu einem 75 Bit breiten Wort zusammengesetzt und über einen *Clock Domain Crossing Handshake* in die *Clock Domain* des *SpartanMCs* überführt. Das lange Wort wird nun wieder in die fünf abgespeicherten Werte aufgeteilt. Diese fünf Wörter werden zusammen mit einem *Valid*-Signal an die Ports des *SpartanMCs* angelegt. Sobald der *SpartanMC* diese eingelesen hat, signalisiert er über ein weiteres Signal, dass die nächste Anfrage durchgeführt werden soll.

SpartanMC-Anpassungen

Wie in der Sektion zum **DRAM Interface-Modul** beschrieben, erfolgt die Ansteuerung des *DRAM Interface*-Moduls sowie das Auslesen der Daten über den *SpartanMC*. Um dies zu erreichen, wird der *SpartanMC* um mehrere neue Ports erweitert. Dies geschieht mit dem *jConfig*-Tool [7]. Für die fünf aufgezeichneten Signale ist jeweils ein Eingangsport ergänzt worden, der die jeweils benötigte Bitbreite aufweist. Die Darstellung der Umlauffrequenz benötigt allerdings mehr als die vom *SpartanMC* verwendete Wortbreite von 18 Bit, weshalb für sie zwei Ports benötigt werden. Zwei weitere Eingangsports ermöglichen das Einlesen des **init_calib_complete**-Signals des MIG-*User Interfaces* und des Signals, das angibt, ob der DRAM voll ist. Drei neue Ausgangsports liefern die Steuersignale des *DRAM Interface*-Moduls: Eines um anzuzeigen, dass Daten aufgezeichnet werden sollen, ein zweites um den Leseprozess einzuschalten, ein drittes Signal gibt an, dass alle fünf Werte beim *SpartanMC* angekommen sind, und dass deshalb neue Daten aus dem DRAM angefordert werden sollen.

Die Firmware des SpartanMCs wird um die zwei Dateien *record_data_to_dram.cpp* und *record_data_to_dram.h* erweitert. Damit ist es möglich, eine Messung zu starten. Diese Messung läuft dann so lange, bis der DRAM voll ist bzw. bis kein Eingangssignal mehr anliegt. Da die Messwerte nicht der 18 Bit Wortbreite des *SpartanMCs* entsprechen, müssen die vorzeichenbehafteten Werte noch nach dem Einlesen vorzeichenerweitert werden. Außerdem muss der Wert der Umlauffrequenz wieder aus den zwei eingelesenen Werten korrekt zusammengefügt werden. Dabei muss darauf

geachtet werden, dass der Datentyp long verwendet wird, damit der zusammengesetzte Wert breiter als 18 Bit sein kann. Anschließend werden die gemessenen Werte ausgelesen und über die UART-Schnittstelle an einen PC gesendet.

Das Bedienungsmenü wird so erweitert, dass der Nutzer das Aufzeichnen der Daten starten kann, indem von dort die eben beschriebenen Funktionen aufgerufen werden können.

Analog hierzu wird der *SpartanMC* zusätzlich um einen Port erweitert, mit dem die letzten 2^{12} gemessenen Periodendauern des Referenzsignals aus dem *Period Length Detector*-Modul, welches, wie zuvor beschrieben, für das *Ramp Up*-Szenario ergänzt wurde, ausgelesen werden können. Da dies jedoch wesentlich weniger Speicherplatz benötigt, werden die Periodendauern in BRAM gespeichert. Daher werden auch keine Kontrollsignale benötigt, sondern die Periodendauern werden immer abgespeichert, es sei denn, es wird ein Lesezugriff ausgeführt. Falls die letzte Adresse des BRAMs beschrieben wurde, wird für den nächsten Wert wieder die ersten Adresse verwendet. Auch diese Daten können über UART an einen PC gesendet werden. Dafür werden in der Firmware des *SpartanMCs* die zwei Dateien *print_estimated_frequency.cpp* und *print_estimated_frequency.h* ergänzt. Sinn dieser Erweiterung ist es, feststellen zu können, ob Schwankungen in der ermittelten Frequenz zu beobachten sind. Die in Tests aufgezeichneten Daten konnten zeigen, dass das Modul zur Periodendauerermittlung hervorragend funktioniert und äußerst robuste Ergebnisse liefert.

5.3 Softwareanpassungen

Die in Abschnitt 5.1 vorgestellten Modellanpassungen werden im Folgenden in konkrete Änderungen im C-Code überführt. Dafür müssen die Methoden zur Interaktion mit dem *Sensor-/Actuator*-Modul erweitert werden.

Zunächst wird die Möglichkeit ergänzt, die in Hardware gemessenen Umlaufzeit der Referenzspannung zu erhalten. Dafür werden zwei Adressen reserviert, die mit der *load_from_sensor()*-Funktion ausgelesen werden können, um zum einen das Gültigkeitssignal überprüfen zu können und zum anderen die gemessene Periodendauer einlesen zu können (vgl. Tabelle 5.1). Wird eine dieser beiden Adressen gelesen, so liefert die Hardware die angeforderte Information an den Dateneingang des *Sensor*-Moduls. Ein weiterer Adressraum wird für die Schreiboperationen auf den DRAM reserviert. Um den Zugriff auf den zweiten Ringpuffer zu erhalten, wird ein Methodenrumpf für die zweite *load_from_sensor()*-Funktion deklariert. Das Tool zum Programmieren des CGRAs benötigt diesen Methodenrumpf, ignoriert allerdings alles, was innerhalb dieser Methoden geschrieben steht, und ersetzt die Aufrufe durch den Opcode des Sensoraufrufs. Der Compiler behandelt diese Funktionen als *Intrinsics*.

Der Ablauf des Programms nach den Anpassungen ist schematisch in Abbildung 5.6 dargestellt.

Das Programm wird um eine Initialisierung ergänzt. Dabei wird gewartet, bis das Gültigkeitssignal gesetzt ist, was anzeigt, dass eine Periodendauer gemessen wurde. Aus der gemessenen Umlaufdauer werden dann die initialen Werte für die Lorentzfaktoren des Referenzpartikels berechnet.

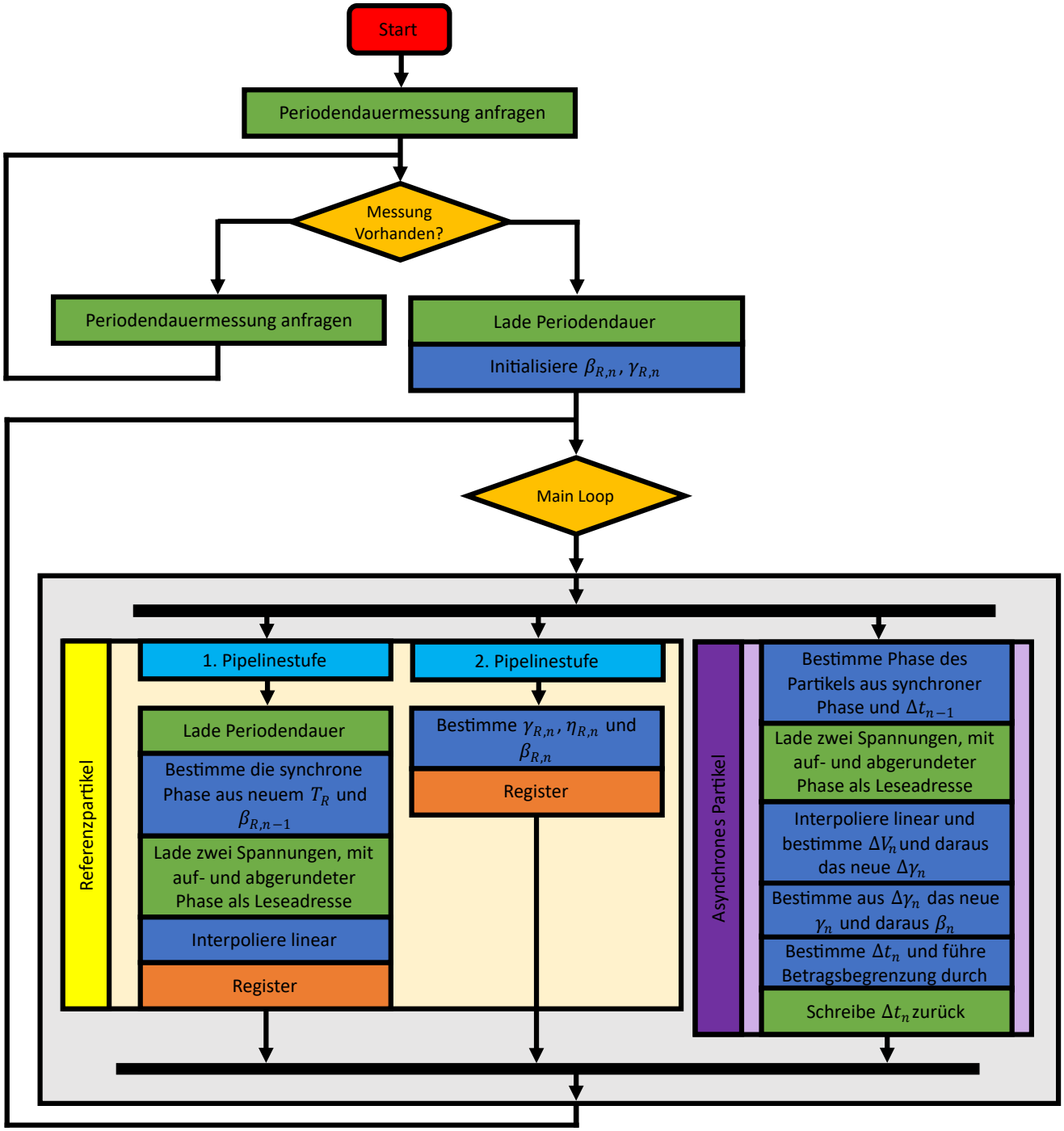


Abbildung 5.6: Ablaufdiagramm des Programms im *Ramp Up*-Fall. Sensor-/Aktuatorzugriffe sind grün dargestellt, Schleifenköpfe sind in orange und Berechnungen sind in blau abgebildet

Zu beachten ist hierbei, dass die Umlaufdauer als ganzzahliges Vielfaches der Abtastperiodendauer eingelesen wurde und in Fixpunktarithmetik mit zwei Nachkommastellen vorliegt. Dies muss bei der Ermittlung der Lorentzfaktoren berücksichtigt werden, indem die Umlaufdauer in eine Gleitkommazahl überführt wird und die Abtastfrequenz für die Berechnung vervierfacht wird. Damit ist die Initialisierung abgeschlossen und es wird in die Hauptschleife des Programms übergegangen.

In der Hauptschleife des Programms, die einmal pro Periode ausgeführt wird, wird zunächst die aktuelle Periodenlänge der Gapspannung ermittelt. Aus der Differenz dieser Periodenlänge mit der aus der relativistischen Geschwindigkeit des Referenzpartikels der vorherigen Iteration berechneten Umlaufdauer ergibt sich die synchrone Phase. Die synchrone Phase gibt an, mit welcher zeitlichen Verzögerung das Referenzteilchen gegenüber dem Nulldurchgang der Referenzspannung an der Kavität des Beschleunigers ankommt (vgl. Abbildung 5.2). Anschließend wird über den zweiten Sensorport ausgelesen, welche Spannung beim Eintreffen des Referenzteilchens an der Kavität anliegt. Da die synchrone Phase im Regelfall keinem ganzzahligem Vielfachen des Systemtakts entspricht, mit dem die Eingangssignale abgetastet werden, werden, um die Genauigkeit der Rechnung zu erhöhen, die zwei zeitlich am nächsten gelegenen Spannungen ausgelesen und linear dazwischen interpoliert. Die somit ermittelte Spannung $V_{R,n-1}$ ist diejenige Spannung, die das Referenzteilchen beim Eintritt in die Kavität sieht und die zu dessen Beschleunigung führt. Mit Hilfe dieser Spannung können nach Gleichung (5.1) neue Werte für $\gamma_{R,n}$ ermittelt werden. Daraus werden die Werte für $\eta_{R,n}$ und $\beta_{R,n}$ gemäß Gleichungen (2.4a) und (2.1) berechnet.

Durch das Aufsummieren von Δt_{n-1} auf die synchrone Phase ergibt sich nun der Zeitpunkt, zu dem das asynchrone, gegenüber dem Referenzteilchen um Δt_{n-1} verschobene Teilchen an der Kavität eintrifft (vgl. Abbildung 5.2). Für dieses wird die Spannung an der Kavität aus dem ersten Sensorport ausgelesen. Auch hier wird wieder für erhöhte Genauigkeit auf eine lineare Interpolation zurückgegriffen. Um die im Abschnitt 5.1 beschriebene Änderung für Gleichung (2.2) zu berücksichtigen, wird die Differenz ΔV_{n-1} der Spannungen $V_{R,n-1}$ (auf das Referenzpartikel wirkend) und V_{n-1} (auf das asynchrone Partikel wirkend) gebildet, um die Abweichung $\Delta\gamma_n$ zwischen γ_n und $\gamma_{R,n}$ zu ermitteln. Daraus wiederum werden die Lorentzfaktoren des asynchronen Partikels berechnet und abschließend die neue zeitliche Differenz Δt_n gemäß Gleichung (2.5b) für den nächsten Umlauf ermittelt. Zum Schluss wird noch die Begrenzung von Δt_n auf den Bereich $[-2\pi, +2\pi]$ durchgeführt und die Addition aus Δt_n mit der synchronen Phase über das *Sensor-Actuator*-Modul geschrieben, damit im Ausgabesignal ein Gaußimpuls mit Scheitelpunkt an dieser Stelle erzeugt wird. Zusätzlich werden auch die Signale, die im DRAM aufgezeichnet werden sollen, an die dafür reservierten Adressen zurückgeschrieben.

Die Ausführung des Programms erfordert durch die Erweiterungen mehr Zeit, als in einer Periodendauer bei maximaler Frequenz zur Verfügung steht. Aus diesem Grund wird für das Programm manuelles *Loop Pipelining* benutzt, um die Ausführungszeit auf Kosten einer um eine Umlaufdauer vergrößerten Latenz zu reduzieren. Beim *Loop Pipelining* wird ein Teil der Schleife in die nächsten Iteration verschoben. Dadurch werden in der ersten Periode nur die gemessenen Spannungen ermittelt und interpoliert. In der darauffolgenden Periode werden alle weiteren Berechnungen durchgeführt. Durch die Unabhängigkeit des Spannungenauslesens der aktuellen Periode und den Berechnungen, die mit den gemessenen Werten der vorherigen Periode durchgeführt werden,

ist es nun möglich diese zwei Aufgaben parallel auszuführen, wodurch die Rechenzeit weit genug gesenkt wird, sodass alles in der Zeit einer Umlaufperiode berechnet werden kann.

5.4 Testen

Um die Authentizität des Ausgabesignals des Simulators für das *Ramp Up*-Szenario festzustellen, muss für die beiden Eingangsspannungen ein Testsignal generiert werden. Bei diesem Testsignal entsprechen die Eingangsspannungen des Simulators dem in Abbildung 5.1 dargestellten Verlauf. Dafür wird zunächst ein Ansatz verfolgt, bei dem dem Funktionsgenerator während des gesamten Tests von außen vorgegeben wird, welche Amplitude und Frequenz er zum aktuellen Zeitpunkt für die Eingangssignale erzeugen soll. Um dies zu erreichen, wird der Funktionsgenerator per USB-Kabel an einen Computer angeschlossen und mit einem Pythonskript angesteuert, das Befehle mit Hilfe des *Universal Serial Bus Test & Measurement Class* (USBTMC)-Protokolls versendet. Dabei hat sich jedoch gezeigt, dass der Funktionsgenerator nicht in der Lage ist, die gesendeten Befehle schnell genug abzuarbeiten, um das *Ramp Up*-Szenario realitätsnah abzubilden, weshalb sich die Befehle in einer Warteschlange aufgestaut haben. Ein Durchlaufen des gesamten *Ramp Up*-Szenarios dauert so über 30 Sekunden, anstatt nur den Bruchteil einer Sekunde zu benötigen.

Aus diesem Grund wird ein anderer Ansatz umgesetzt, bei dem die Eingangssignale im FPGA-Design selbst generiert werden. Dafür wird zunächst der Verlauf der Signale vereinfacht, wie es in Abbildung 5.7 dargestellt ist, indem die Kurven linear angenähert werden. Die Amplitude ist in sieben lineare Abschnitte aufgeteilt, die Frequenz in drei.

Im Hardwaremodul zur Signalerzeugung läuft ein Timer während der Signalgenerierung. Mit Hilfe des Timers wird jeweils ermittelt, in welchem linearen Bereich der Kurve das Signal gerade ist. Außerdem wird der Timer für die Berechnung der linearen Funktion innerhalb dieses Bereichs verwendet. Die Frequenzen werden hierbei in Fixpunktarithmetik dargestellt, um die Berechnung zu vereinfachen. Der Wert der Frequenzen wird dabei um 11 Bit nach links verschoben. Für die Amplituden wird ebenfalls eine Umrechnung vorgenommen, sodass der größte darstellbare Wert, $14400\,\mathrm{V}$ entspricht.

In jedem Systemtakt wird aus der aktuellen Frequenz mit der Dauer eines Systemtakts (in Fixpunktdarstellung; um 58 Bit nach links geschoben) die Änderung der Phase des Signals (in Fixpunktdarstellung; um 31 Bit nach links geschoben) ermittelt. Diese wird auf die aktuelle Phase aufsummiert, um die neue Phase zu erhalten. Überschreitet die neue Phase den Wert 2^{31}, was dem Wert $2\,\pi$ entspricht, so ist eine Periode abgeschlossen und die neue Phase wird um 2^{31} reduziert, damit sie nicht unendlich anwächst.

Mit der Phase wird der Wert des Sinus ermittelt, indem der entsprechende Wert nachgeschlagen wird. Dafür ist eine Tabelle erstellt worden, in der die aufsteigende Flanke eines Sinus in 256 Datenpunkten hinterlegt ist. Mit Hilfe der obersten zwei Bits der Phase wird bestimmt, ob sich das Signal gerade in der aufsteigenden oder in der abfallenden Flanke des Sinus befindet und welches Vorzeichen das Signal besitzt. Die nächsten acht Bits sind dann der Index, an dem in der Tabelle der Wert des Sinus nachgeschlagen wird. Für erhöhte Genauigkeit werden die unteren 21

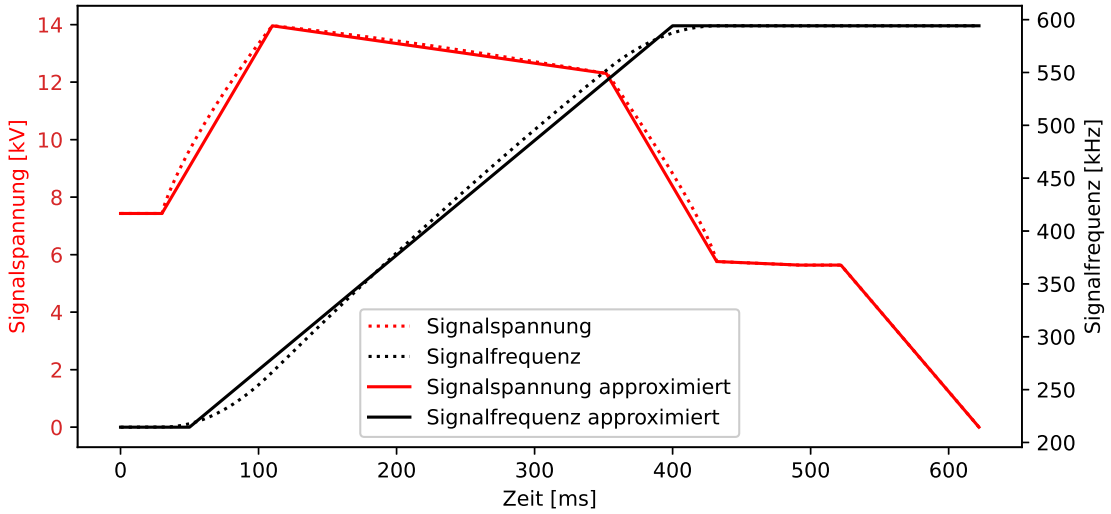


Abbildung 5.7: Näherung des Verlauf von Spannung und Frequenz eines *Ramp Up*-Szenarios, wie es in einer Simulation der GSI verwendet wurde

30	29	28 27 26 25 24 23 22 21	20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Vor-zei-chen	Flan-ke	Tabellenindex	Für lineare Interpolation

Tabelle 5.3: Bedeutung der einzelnen Bits bei der Bestimmung des Sinuswertes aus der Tabelle. Das Flankenbit gibt an, ob die aufsteigende (0) oder die abfallende (1) Flanke benötigt wird.

Bits noch verwendet, um eine lineare Interpolation auszuführen (vgl. Tabelle 5.3). Abschließend wird der Amplitudenwert mit dem Sinuswert multipliziert und das auf 16 Bit begrenze Ergebnis an den Ausgang des Moduls angelegt. Da die Latenz der Signalerzeugung irrelevant ist, es aber sehr wichtig ist, dass in jedem Takt ein neuer Wert generiert wird, ist die Signalerzeugungskette in viele Stufen gepipelined.

Damit das so generierte Eingangssignal realistischer wird, wird es an die beiden Multiplexer vor den DACs der FMC-Karte geschickt und im DAC in ein analoges Signal umgewandelt. Anschließend wird das analoge Signal per Kabel wieder als Eingangssignal an die ADCs angeschlossen. Somit wird gewährleistet, dass auch das im FPGA selbst erzeugte Signal eine analoge Strecke durchläuft und rauschbehaftet ist.

Um eine Aufzeichnung der Daten in den DRAM während des *Ramp Up*-Szenarios zu ermöglichen, gibt es im *SpartanMC* die Möglichkeit, beim Starten einer Aufzeichnung die Multiplexer vor den DACs umzuschalten, sodass das approximierte Eingangssignal des *Ramp Up*-Szenarios am DAC anliegt. Das Signal kann dabei entweder an einen, keinen oder an beide Ausgänge angelegt werden. Im Anschluss an die Messung wird der Multiplexer wieder auf seinen Standardwert zurückgeschaltet. Das Anlegen des Multiplexersignals, mit dem das Signal des *Ramp Up*-Szenarios als Eingangssignal des DACs ausgewählt wird, dient auch als Signal für das Testsignalgenerator-Modul zum Starten der Erzeugung des *Ramp Up*-Signals.

6 Multi Bunch

Neben der in den Kapiteln 4 und 5 beschriebenen Umsetzung des stationären Falls und des *Ramp Up*-Szenarios soll der HiL-Simulator um weitere praxisrelevante Szenarien erweitert werden. Dieses Kapitel befasst sich mit dem *Multi Bunch*-Szenario, in dem mehr als ein Bunch im Beschleuniger vorhanden ist. Dabei muss gelten, dass die Frequenz der an der Kavität anliegenden Spannung ein ganzzahliges Vielfaches h (die *harmonic number*) der Frequenz des bisherigen Betrachtungsfalles ist. Da sich die Dauer, die ein Teilchen für einen Umlauf im Beschleuniger benötigt, nicht verändert hat, bleibt auch das Referenzteilchen und damit die Referenzspannung unbetroffen von dieser Änderung. Dies hat zur Folge, dass in einer Periodendauer des Referenzteilchens insgesamt h asynchrone Teilchen berechnet und ihre entsprechenden Ausgabesignale erzeugt werden müssen.

6.1 Anpassungen

Für den *Multi Bunch*-Fall wird nach wie vor dasselbe Modell benutzt, das bereits im *Ramp Up*-Szenario in Verwendung war. Dennoch müssen Anpassungen an der Soft- und an der Hardware vorgenommen werden. In Abbildung 6.1 ist das Blockdiagramm des FPGAs mit den für das *Multi Bunch*-Szenario eingeführten Änderungen dargestellt. Der Ablauf des Programms nach den Anpassungen ist schematisch in Abbildung 6.2 dargestellt.

6.1.1 Codevervielfachung

Zunächst wird festgelegt, dass unabhängig von h nur ein einziges Referenzteilchen betrachtet wird. Das bedeutet, dass alle Berechnungen für das Referenzteilchen nur einmal ausgeführt werden und dass die ermittelten Werte für die Berechnung aller h asynchronen Partikel verwendet werden. Die Berechnung für ein asynchrones Partikel hingegen muss im Folgenden insgesamt h-mal ausgeführt werden.

Die naheliegende Lösung dies umzusetzen, ist, die Variablen des asynchronen Partikel als Arrays zu deklarieren und die Berechnung innerhalb einer Schleife auszuführen. Allerdings werden Arrays in der Toolchain des CGRAs im Speicher angelegt, sodass ein Arrayzugriff immer mit einem Speicherzugriff verbunden ist. Dies ist insofern ungeeignet, da bis jetzt kein Speichermodul an das CGRA angeschlossen wurde, denn alle Variablen sind in den Registerfiles des CGRAs

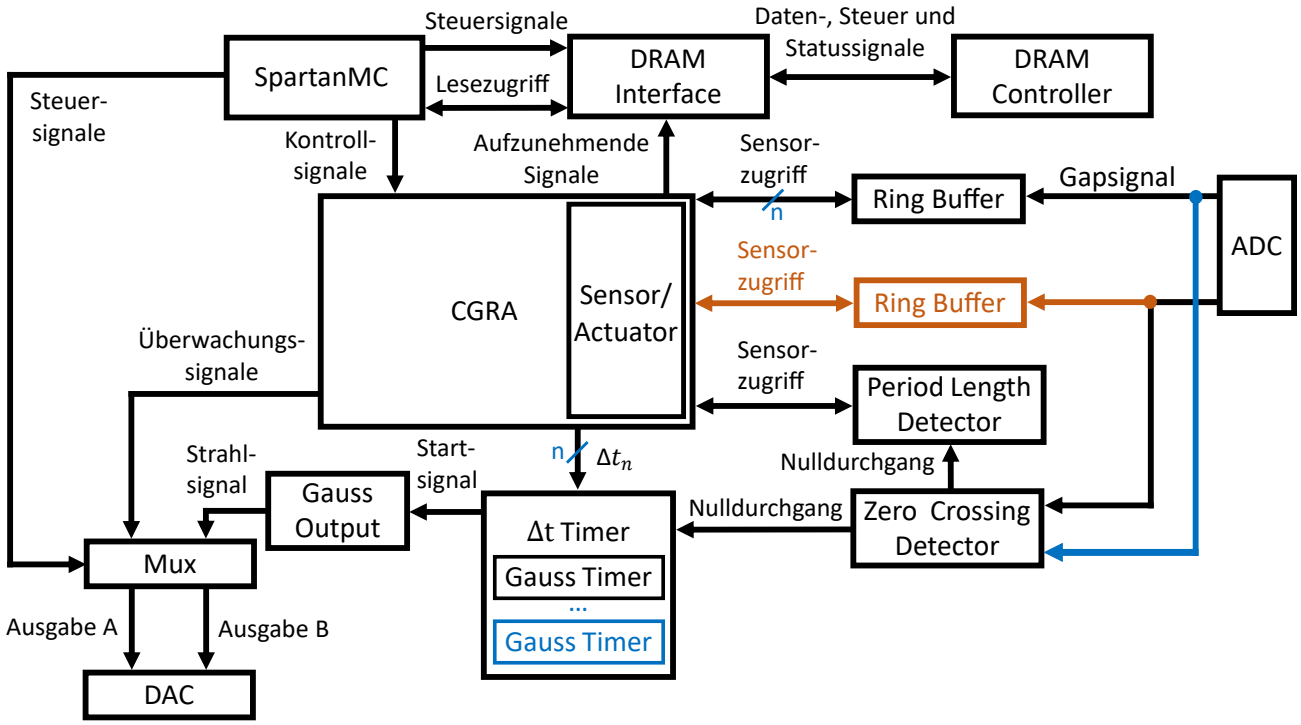


Abbildung 6.1: FPGA Blockdiagramm für das *Multi Bunch*-Szenario. Neue Elemente sind in blau, entfernte Elemente in rot dargestellt

abgespeichert und die Ein- und Ausgabewerte werden über das *Sensor-/Actuator*-Modul in das System hinein- oder aus dem System herausgeführt. Es ist auch nicht sinnvoll für die Arrays einen Speicher einzuführen, denn die Abspeicherung außerhalb der Registerfiles bringt lediglich längere Ausführungszeiten durch die Speicherzugriffe und durch das aufwendigere Routing mit sich, da Speicherzugriffe nur von PEs mit Speicherzugriff durchgeführt werden können. Zusätzlich entsteht weiterer Hardwareaufwand durch den Speicher, vor allem durch das Einhalten von Cachekohärenz, wenn mehrere PEs einen Speicherzugriff erhalten.

Aus diesem Grund wird der Ansatz, Arrays zu benutzen, verworfen und eine andere Vorgehensweise verwendet, um die Vervielfältigung des Abschnittes für die asynchronen Partikel umzusetzen. Es werden Makros im C-Code definiert, die zum einen die Deklaration der Variablen und zum anderen den Codeabschnitt zur Berechnung eines einzelnen asynchronen Partikels beinhalten. Die Namen der jeweiligen Variablen für das asynchrone Partikel werden mit Hilfe der Makros parametrisiert, sodass bei mehrfachem Aufrufen des Makros mehrere Kopien desselben Codes mit unterschiedlichen Variablennamen erstellt werden. Damit kann die Deklaration der Variablen respektive die Berechnung für alle h asynchronen Partikel durch das h-fache Aufrufen des jeweiligen Makros durchgeführt werden. Durch dieses Vorgehen werden alle Variablen wieder in den Registerfiles der PEs abgespeichert und das System verhält sich so wie zuvor.

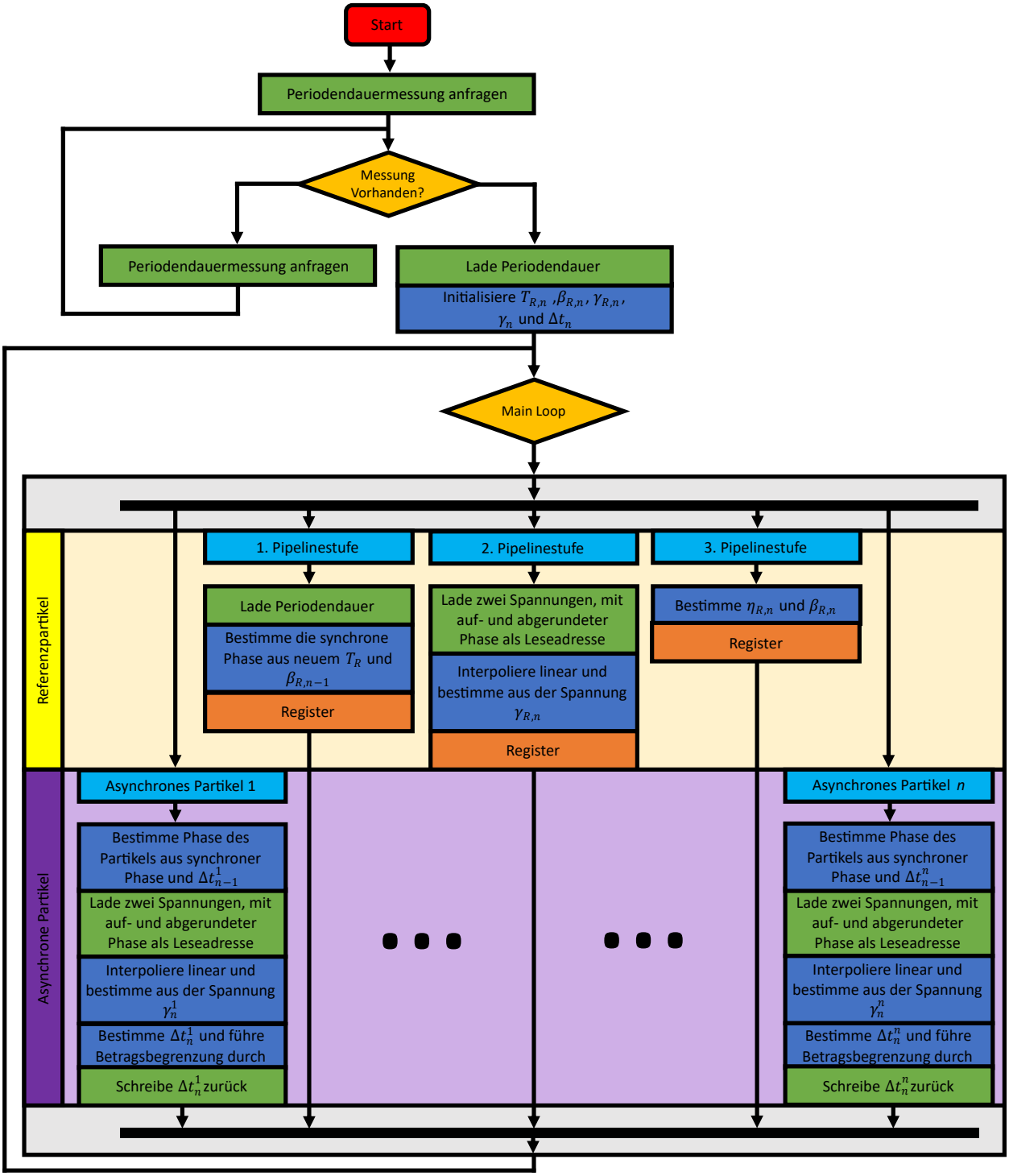


Abbildung 6.2: Ablaufdiagramm des Programms im *Multi Bunch*-Fall mit n Bunches. Sensor-/Aktuatorzugriffe sind grün dargestellt, Schleifenköpfe sind in orange und Berechnungen sind in blau abgebildet. Der hochgestellt Index zeigt an, auf welches asynchrone Partikel sich der Wert bezieht.

6.1.2 Spannungsmessung

Eine weitere Änderung, die am Programm vorgenommen wurde, liegt in der unterschiedlichen Frequenz der zwei Eingangssignale begründet. Bei der Ermittlung von $\Delta\gamma_n$ wird die auf das Referenzteilchen wirkende Spannung benötigt. Dafür wurde in Kapitel 5 ein zweiter Ringpuffer eingeführt, aus dem die auf das Referenzteilchen wirkende Spannung ausgelesen wurde. Da sich die Spannung an der Kavität aber im *Multi Bunch*-Szenario mit einer h-mal größeren Frequenz ändert, muss auch für die Spannung, die auf das Referenzteilchen wirkt, gelten, dass sie sich mit der größeren Frequenz ändert. Aus diesem Grund wird zunächst die Änderung vorgenommen, dass die synchrone Phase in Software mit h multipliziert wird, um die Leseadresse für $V_{R,n-1}$ zu ermitteln.

Dieser Ansatz hat zu Problemen geführt, weil die beiden Eingangssignale unabhängig voneinander mit Rauschen belastet werden und weil sie außerdem nicht perfekt synchronisiert im Funktionsgenerator erzeugt werden, gerade dann, wenn sie nicht dieselbe Frequenz aufweisen. Der in Kapitel 5 eingeführte Ringpuffer wurde daher wieder entfernt. Dafür wird im verbleibenden Ringpuffer, der die Gapspannung aufzeichnet, eine zweite Offsetadresse gespeichert, wenn die Gapspannung einen positiven Nulldurchgang aufweist. Wird nun die Leseadresse für $V_{R,n-1}$ angelegt, so wird diese neue Offsetadresse verwendet. Dadurch wird die Spannung, die das Referenzteilchen sieht, von einem Signal mit der richtigen Frequenz ermittelt. Außerdem werden jetzt alle Spannungen mit demselben Eingangssignal bestimmt, sodass sich Rauschen bei den Δ-Größen zum Teil gegenseitig kompensiert. Um den Nulldurchgang der Gapspannung zu ermitteln, wurde zunächst ein zweites *Zero Crossing Detector*-Modul instanziiert. Dieses wurde in einem weiteren Optimierungsschritt mit dem bereits existierenden *Zero Crossing Detector*-Modul zusammengefasst.

Auch für die Ermittlung der Spannung V_{n-1} müssen Anpassungen vorgenommen werden, denn für diese müssen im *Multi Bunch*-Szenario h verschiedene Werte ermittelt werden. Dafür wird zunächst der Ansatz verfolgt, dass in Software die gemessene Periodendauer durch die Anzahl der Bunches geteilt wird. Bei der Berechnung des k-ten Bunches wird dann auf die Leseadresse das k-fache des Quotienten addiert. Dieser Ansatz weist das Problem auf, dass der ermittelte Quotient Ungenauigkeiten aufweist und dass eine weitere Division sowie pro Bunch eine weitere Multiplikation und Addition ausgeführt werden müssen.

Es wird deshalb ein anderer Ansatz umgesetzt, bei dem im *Ringpuffer*-Modul für jeden Bunch im System ein Offsetwert bestimmt wird, der auf die entsprechende Leseadresse aufaddiert wird. Dafür wird ein Timer eingebaut, der beim ersten Nulldurchgang der Gapspannung zurückgesetzt wird. Bei den folgenden k Nulldurchgängen der Gapspannung wird der Timerwert auf die Adresse des Nulldurchgangs der Referenzspannung addiert, um den Offset der Leseadresse für den k-ten Bunch zu ermitteln. Wenn k den Wert der Anzahl an Bunches im System minus eins erreicht, werden k und der Timer wieder auf Null gesetzt. Um auswählen zu können, welcher Offset auf die Leseadresse aufaddiert werden soll, werden, wie in Kapitel 5.2, neue Sensorleseports ergänzt. Dieser neue Ansatz bietet den Vorteil, dass die Leseoffsets genau soweit auseinanderliegen wie die Nulldurchgänge der Gapspannung und dass keine zusätzlichen Berechnungen in Software

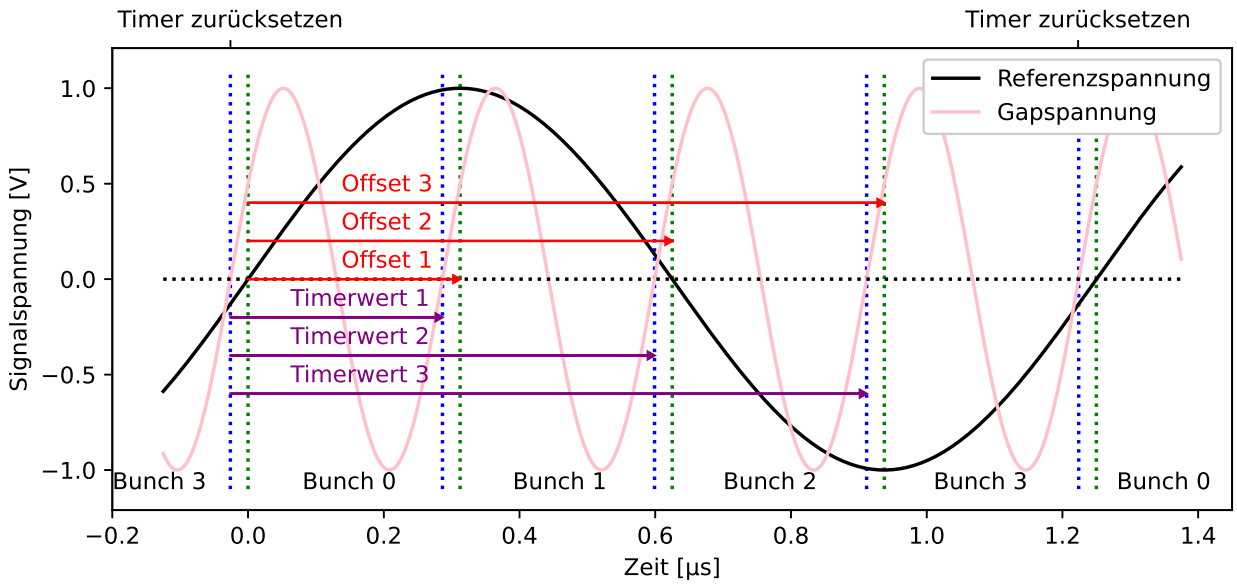


Abbildung 6.3: Beispielhafte Darstellung, wie für $n = 4$ der Leseoffset mit dem neuen System ermittelt wird. Für Bunch 0 wird direkt der Nulldurchgang des Referenzpartikels als Offset verwendet.

durchgeführt werden müssen. In Abbildung 6.3 ist das System zur Offsetbestimmung grafisch für $n = 4$ dargestellt.

6.1.3 h zur Laufzeit berechnen

Eine weitere Verbesserung, die vorgenommen wurde, ist, dass die Anzahl h der aktiven Bunches zur Laufzeit ermittelt und nicht fest vorgegeben wird. Die maximale Anzahl an Bunches wird allerdings weiterhin als Konstante festgelegt, damit für jeden potentiellen Bunch die notwendige Hardware existiert. Das Modul zur Periodendauererkennung wird erweitert, indem dort auch das Signal der detektierten positiven Nulldurchgänge der Gapspannung angelegt wird, mit dem die Periodenlänge der Gapspannung ermittelt wird. Da eine Division der beiden Periodenlängen nicht in einem Systemtakt durchgeführt werden kann, wird eine Fallunterscheidung durchgeführt, in der die Periodendauer der Referenzspannung mit Vielfachen der Periodendauer der Gapspannung verglichen wird. Für eine robustere Bestimmung wird als Vergleichswert die Periodendauer der Referenzspannung mit der um drei Bits nach rechts geschobenen Periodendauer der Gapspannung addiert. Dadurch liefert der Vergleich mit den Vielfachen der Periodendauer zuverlässig, dass der Wert größer als das h-fache, aber kleiner als das $h + 1$-fache der Periodendauer der Gapspannung ist.

In der Software wird weiterhin von der maximalen Anzahl an Bunches ausgegangen, weshalb das CGRA n Werte für Δt_n an das Δt *Timer*-Modul sendet. Das Δt *Timer*-Modul wird so angepasst, dass für jeden Bunch das an ihn adressierte Δt_n entgegengenommen wird und für jeden Bunch eine Instanz des *Gauss Timer*-Modul erstellt wird(vgl. Abbildung 6.1). Dieses Modul berechnet

den Startzeitpunkt, an dem der jeweilige Gaußimpuls abgespielt werden soll, und sendet zum entsprechenden Zeitpunkt ein Startsignal. Allerdings wird eine Maske über die Startsignale der Gaußimpulserzeugung gelegt, um nur die h aktiven Bunches darzustellen. Das heißt, dass nur die Gaußimpulse der ersten h Berechnungen erzeugt und alle anderen ausgeblendet werden. Die Startsignale werden anschließend alle verodert und zu einem einzigen Startsignal zusammengefasst. Das ist möglich, da die einzelnen Gaußimpulse zeitlich voneinander getrennt sein sollten. Wenn sie sich an den Rändern der Gaußkurve etwas überlagern, wird derjenige Wert des Gaußimpulses abgespielt, der größer ist. Die Bestimmung von h zur Laufzeit hat sich im Laborexperiment und in der Simulation als absolut zuverlässig erwiesen.

6.1.4 System beschleunigen

Durch die größere Anzahl an asynchronen Teilchen, die im *Multi Bunch*-Fall berechnet werden müssen, hat sich die Ausführungsdauer des Modells auf dem CGRA wieder etwas verschlechtert, sodass weitere Maßnahmen ergriffen werden müssen, um die Berechnung in weniger als $1\,\mu s$ ausführen zu können. Dafür wird das bereits im Kapitel 5.3 beschriebene *Loop Pipelining* weiter ausgebaut (vgl. Abbildung 6.2). Die Berechnung des Referenzpartikels wird nun in drei Teile geteilt: Zunächst wird die Periodendauer gemessen und daraus die synchrone Phase ermittelt. Im zweiten Teil wird die Spannung $V_{R,n-1}$ gemessen und interpoliert. Außerdem wird daraus noch $\gamma_{R,n}$ berechnet. Der letzte Teil dient schließlich der Bestimmung von $\beta_{R,n}$ sowie $\eta_{R,n}$. Zusätzlich wird hier noch der Teil von Gleichung (2.5b) ermittelt, der nicht vom asynchronen Teilchen abhängt, damit er nur einmal berechnet wird.

Auch für die Berechnung des asynchronen Partikels wurde versucht, *Loop Pipelining* zu verwenden. Es hat sich jedoch gezeigt, dass es nicht möglich ist, die Spannung V_{n-1} mit Hilfe von Δt_{n-2} oder einem noch älteren Wert zu ermitteln. Selbst in einer Java-Simulation des Modells mit exakten, nicht quantisierten Eingangswerten hat sich die Energie immer weiter aufgeschaukelt, falls nicht der mit den Messergebnissen der vorherigen Iteration bestimmte Wert von Δt für die Messung der neuen Spannungen verwendet wird. Somit ist es nicht möglich, die Berechnung des asynchronen Partikel über mehrere Schleifeniterationen aufzuteilen.

Es hat sich aber herausgestellt, dass die für das Referenzteilchen ermittelten Werte aus vorherigen Iterationen verwendet werden können, ohne das System zu destabilisieren. Dementsprechend können alle Berechnungen der asynchronen Teilchen bereits ganz am Anfang jeder Iteration starten und müssen nicht auf die neu ermittelten Werte des Referenzpartikels warten. Es kann daher die volle Umlaufdauer für ihre Berechnung ausgenutzt werden, was ausreichend lange ist, um die Berechnung rechtzeitig abzuschließen.

Da die asynchronen Partikel alle unabhängig voneinander sind, können prinzipiell unbegrenzt viele Teilchen ergänzt werden. Allerdings ist die Verteilung der Werte des Referenzpartikels dabei ein Flaschenhals. Das Gleiche gilt auch für den Sensorzugriff, der nur von einem einzigen PE ausgeführt werden kann, von dem aus dann die Messwerte an die anderen PEs weitergegeben werden müssen. Aus diesem Grund kann die Anzahl an Bunches nicht beliebig weit erhöht werden,

wenn die Echtzeitbedingung erfüllt bleiben soll. Das System wird daher auf eine maximale Anzahl von acht Bunches beschränkt.

Um dem Scheduler mehr Freiraum zu bieten wird außerdem die erzwungene Ordnung der Sensorzugriffe entfernt, wodurch es unter anderem auch möglich ist, die Zugriffe zu pipelinen. Daher muss nicht auf den Abschluss des letzten Zugriffs gewartet werden, sondern es kann bereits der nächste Zugriff stattfinden, während der vorherige noch ausgeführt wird. Somit kann ein Lesezugriff pro Takt stattfinden, anstatt wie vorher nur einer alle zwei Takte, was die Gesamtdauer des Schedules etwas weiter reduziert hat. Die erzwungene Ordnung, die vorher verwendet wurde, stellte sicher, dass die einzelnen Peripheriezugriffe exakt in der Ordnung stattfinden, in der sie im Programmcode auftauchen. Außerdem wird garantiert, dass der vorherige Zugriff abgeschlossen ist, bevor der nächste stattfindet. Das ist beim Zugriff auf Peripherie durchaus sinnvoll, weil eine Änderung der Reihenfolge der Zugriffe bei bspw. einer FIFO-Warteschlange durchaus zu anderen Ergebnissen führen kann. In diesem Fall ist die Reihenfolge allerdings unerheblich, weil die Sensorlesezugriffe sich nicht beeinflussen und die Aktuatorschreibzugriffe ebenfalls unabhängig sind.

6.1.5 Modellanpassungen

Das Modell erzeugt bis jetzt, wenn ein Bunch die Kavität passiert, unabhängig von der Anzahl an Bunches im System, dieselbe Gaußimpulsform, die auch zuvor verwendet wurde. Offensichtlich ist dies nicht sinnvoll, vor allem, weil sich die so erzeugten Gaußimpulse bei höheren Frequenzen und größerem h erheblich überlagern. Im Modell wird daher eine neue vereinfachende Annahme getroffen, dass die Breite des Gaußimpulses mit h skaliert wird, sodass insgesamt pro Umlauf derselbe Ionenstrom wie im Fall $h = 1$ modelliert wird. Dies ist einfach umsetzbar, indem im *Gauss Output*-Modul der Zähler beim Abspielen des vorberechneten Gaußimpulses in jedem Takt um h anstatt um 1 inkrementiert wird.

Bei der Ausweitung des *Loop Pipelinings* wurde auch das mathematische Modell des Synchrotrons überdacht, um Verbesserungspotential zu finden. Ein potentieller Verbesserungspunkt wurde in der Berechnung von γ_n gefunden.

Um γ_n zu ermitteln, wird momentan zunächst $\gamma_{R,n}$ gemäß Gleichung (5.1) bestimmt. Anschließend wird mit Hilfe von Gleichung (2.2) ein neuer Wert für $\Delta\gamma_n$ berechnet, wofür die Differenz der gemessenen Spannungen benötigt wird. Durch die Addition von $\gamma_{R,n}$ mit $\Delta\gamma_n$ wird dann schließlich der aktuelle Wert für γ_n erhalten. Diese Vorgehensweise ist daraus erwachsen, dass zunächst nur der stationäre Fall betrachtet wurde. Dort war dieser Ansatz sinnvoll, da $\gamma_{R,n}$ konstant war und $V_{R,n} = 0V$ galt, sodass die Berechnung sich vereinfacht hat.

Da diese Vereinfachungen im aktuellen Modell nicht mehr gelten, ist es nicht sinnvoll, den Umweg über $\Delta\gamma_n$ und $\gamma_{R,n}$ zu gehen, um γ_n zu bestimmen. Stattdessen wird nun γ_n, genau wie $\gamma_{R,n}$ in Gleichung (5.1), direkt bestimmt. Gleichung (6.1) stellt die neue Berechnungsvorschrift dar.

$$\gamma_n = \gamma_{n-1} + \frac{Q}{m_0 c_0^2} V_{n-1} \tag{6.1}$$

Diese Änderung hat zunächst aus numerischen Gründen zur Instabilität des Systems geführt. Der Vorfaktor $\frac{Q}{m_0 c_0^2}$ in Gleichung (6.1) liegt für das im Modell verwendete $^{238}U^{73+}$ Ion in der Größenordnung 10^{-10}. $\Delta\gamma_n$ kann dementsprechend auch in dieser Größenordnung liegen. Es kann aber auch deutlich größer sein, wenn viele Werte aufsummiert wurden, und Werte größer 10^{-3} erreichen. In Gleichung (2.5b) wird $\Delta\gamma_n$ für die Berechnung von Δt_n verwendet. Mit der neuen Modellanpassung ist $\Delta\gamma_n$ noch nicht bekannt und muss zunächst berechnet werden, indem die Differenz der beiden γ-Werte gebildet wird. Beide γ-Werte sind immer größer als eins und sind, abhängig von der Geschwindigkeit des Teilchens, im Bereich zwischen $1,2$ und $1,4$. Die γ-Werte haben daher immer denselben Exponent in der IEEE-Gleitkommadarstellung. Dementsprechend ungenau ist das Bilden der Differenz dieser beiden Werte, wenn nur kleine Unterschiede in der Größenordnung 10^{-10} vorliegen. Diese Ungenauigkeiten führen zu Fehlern in der Bestimmung von Δt_n, die sich schnell akkumulieren und das System unbrauchbar machen. In der zuvor verwendeten Berechnung von $\Delta\gamma_n$ aus ΔV_n gemäß Gleichung (2.2), konnten diese kleineren Unterschiede deutlich genauer dargestellt werden, weil sich hier der Exponent in der IEEE-Gleitkommadarstellung ändert, wenn $\Delta\gamma_n$ größer oder kleiner wird.

Um das Aufschaukeln durch die numerischen Fehler zu beheben, wird zunächst ein Unterdrückungsfaktor in Form eines Schwellwertes festgelegt. Falls $\Delta\gamma_n$ diesen Schwellwert nicht überschreitet, so werden beide γ-Werte als identisch angenommen und γ_n auf den Wert von $\gamma_{R,n}$ gesetzt. Dadurch kann das System wieder stabilisiert werden. Allerdings ist die Wahl der Größe des Unterdrückungsfaktors kritisch und muss mit viel Bedacht erfolgen, da ungewollte numerische Fehler unterdrückt werden sollen, aber das restliche Verhalten des Systems unverändert bleiben soll. Daher wird der *SpartanMC* erweitert, um den Unterdrückungsfaktor zur Laufzeit anpassen zu können, damit verschiedene Werte auf ihre Tauglichkeit überprüft werden können. In jeder Iteration der Hauptsschleife wird vom CGRA der aktuelle Unterdrückungsfaktor eingelesen, um vom *SpartanMC* zur Laufzeit geschriebene Werte zu berücksichtigen. Dabei hat sich herausgestellt, dass es keinen idealen Wert für den Unterdrückungsfaktor gibt, der für alle Spannungsamplituden und Frequenzen geeignet ist, sodass es schwierig ist, das System zu stabilisieren, ohne seine Verhalten bezüglich der Longitudinalschwingungen zu beeinflussen.

Deshalb wurde die Idee des Unterdrückungsfaktors wieder verworfen. Stattdessen wird das Modell weiter angepasst, indem in den Gleichungen (6.1) und (5.1) der Wert von γ_0 bzw. $\gamma_{R,0}$ auf Null anstatt auf den Initialwert gesetzt wird. Dadurch ist der akkumulierte Wert, im Folgenden $\gamma_{R,n}^{'}$ bzw. $\gamma_n^{'}$ genannt, wieder in der Lage, seinen Exponenten in der IEEE-Gleitkommadarstellung anzupassen, wenn er größer oder kleiner wird. Zudem wird die Berechnung von $\Delta\gamma_n$ wieder genauer, indem die Differenz von $\gamma_{R,n}^{'}$ und $\gamma_n^{'}$ gebildet wird. Natürlich werden auch weiterhin die tatsächlichen Werte von $\gamma_{R,n}$ und γ_n benötigt. Um diese zu erhalten, muss lediglich der Initialwert mit $\gamma_{R,n}^{'}$ bzw. $\gamma_n^{'}$ addiert werden.

7 Quadrupolschwingung

Ein weiteres Szenario, um das der Simulator erweitert wird, dient dazu, die Schwingungen innerhalb eines Bunches darzustellen. Bei dieser sogenannten Quadrupolschwingung ist das Ausgabesignal kein starrer Gaußimpuls. Stattdessen oszilliert das Ausgabesignal zwischen den zwei Extremfällen eines sehr breiten, flachen Gaußimpuls und einem spitzen, hohen Gaußimpuls hin und her. Dieses Verhalten liegt darin begründet, dass ein Bunch nicht nur aus einem, sondern aus vielen Teilchen besteht, die alle unabhängig voneinander durch die Gapspannung beeinflusst werden. Diejenigen Teilchen, die weiter außen in der Gaußglockenkurve angeordnet sind, werden daher anders von der Gapspannung beeinflusst als diejenigen in der Mitte der Gaußverteilung. Modelliert wird diese Quadrupolschwingung deshalb, indem n asynchrone Partikel innerhalb eines Bunches angenommen werden. Die einzelnen überlagerten asynchronen Partikel werden im Folgenden als *Minibunches* bezeichnet, da sie nur einen Teil des gesamten Bunches darstellen. Für jeden dieser *Minibunches* wird dann ein Gaußimpulssignal erzeugt und diese Signale werden mittels Superposition überlagert. Prinzipiell verhält sich dieses Szenario daher ähnlich wie das zuvor beschriebene *Multi Bunch*-Szenario, denn auch hier wird der Teil des Modells, der ein asynchrones Teilchen betrifft, mehrfach pro Umlaufperiode berechnet. Deshalb müssen nur wenige Änderungen vorgenommen werden, um den Simulator so zu erweitern, sodass er Quadrupolschwingungen emulieren kann, die in diesem Kapitel genauer beschrieben werden.

7.1 Hardwareanpassungen

Die einzigen Anpassungen, die an der Hardware vorgenommen werden müssen, sind Anpassungen bezüglich der Erzeugung des Ausgabesignals. Im Δt *Timer*-Modul wird, analog zum *Multi Bunch*-Szenario, für jeden *Minibunch* das an ihn adressierte Δt_n entgegengenommen und in der jeweiligen Instanz des *Gauss Timer*-Modul verarbeitet. Das *Gauss Timer*-Modul berechnet den Startzeitpunkt, an dem der jeweilige Gaußimpuls abgespielt werden soll, und sendet zum entsprechenden Zeitpunkt ein Startsignal. Die Startsignale werden im Δt *Timer*-Modul gebündelt und gemeinsam an das Toplevelmodul weitergereicht. Zuvor war es nicht nötig, mehrere Startsignale getrennt voneinander zu erstellen, denn die einzelnen Bunches waren zeitlich weit voneinander getrennt. Im Modell der Quadrupolschwingung ist die Überlagerung der Gaußimpulse der einzelnen *Minibunches* aber der Effekt, der dargestellt werden soll. Daher kann es durchaus passieren, dass mehrere oder alle *Minibunches* zum exakt selben Zeitpunkt die Kavität erreichen. Dementsprechend muss für jeden *Minibunch* ein eigenes Startsignal an das *Gauss Output*-Modul

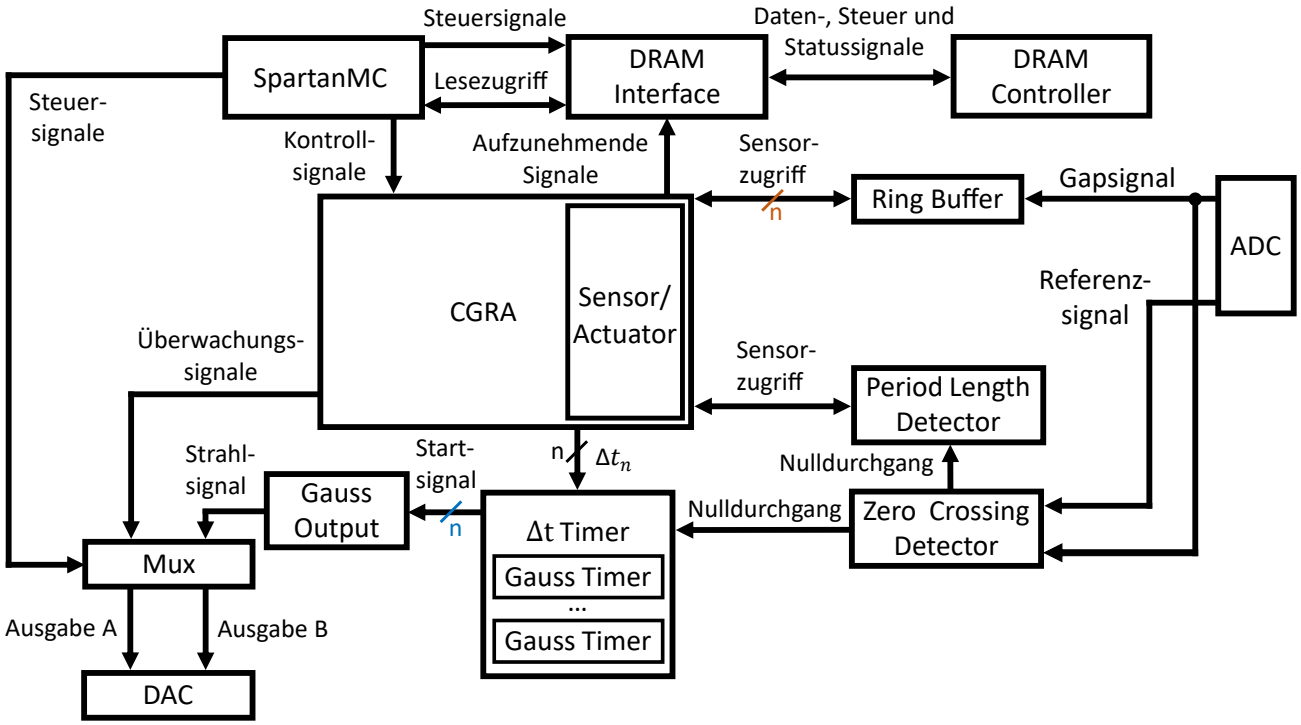


Abbildung 7.1: FPGA Blockdiagramm für das Szenario der Quadrupolschwingung. Neue Elemente sind in blau, entfernte Elemente in rot dargestellt

gesendet werden, weil ansonsten Startsignale verloren gehen könnten, wenn mehrere Startsignale zum gleichen Zeitpunkt ausgelöst werden.

Die Startsignale werden über das Toplevelmodul an das *Gauss Output*-Modul weitergeleitet, in dem anschließend für jeden *Minibunch* ein eigener Gaußimpuls generiert wird. Dafür wird innerhalb des *Gauss Output*-Modul die Hardware des bisherigen *Gauss Output*-Moduls n-mal vervielfältigt. Außerdem wird die Höhe jedes einzelnen Gaußimpulses um Faktor n verringert. Die n erzeugten Signale werden im Anschluss in einem Addierer aufsummiert und zu einem finalen Ausgabesignal zusammengefasst, wodurch insgesamt die Fläche unterhalb des Gesamtsignals, und damit der Ionenstrom, identisch mit der Fläche unterhalb der Gaußkurve in den vorherigen Szenarien ist. Der Addierer ist dabei als Pipeline aufgebaut, die die Struktur eines Binärbaums besitzt, und nach jeder Addition ein Register aufweist. Er besitzt daher eine Tiefe von $Depth = log_2(n)$, um die n Signale aufzuaddieren. Falls n keine Zweierpotenz ist, werden die nicht vorhandenen Eingangssignale in den Addierer auf Null gesetzt, um die Binärbaum-Struktur beizubehalten. Um das System flexibel zu halten, wird der Addierer generisch erstellt, sodass n einfach geändert werden kann und der Addierer automatisch angepasst wird, sodass er die richtige Größe hat, um alle n Signale addieren zu können.

Außerdem wird auch das *DRAM Interface*-Modul angepasst, sodass es eine generische Anzahl an Messwerten pro Umlaufperiode aufzeichnen kann, da es sich beim Erweitern des Gesamtsystems herausgestellt hat, dass es unpraktisch ist, wenn die Anzahl der Messwerte fest vorgegeben ist

und sich nicht einfach erweitern lässt. Das Modul wird so angepasst, dass mit einem Parameter die Anzahl der aufzuzeichnenden Messwerte vorgegeben wird. Leider führt diese Änderung dazu, dass es nicht mehr einfach möglich ist, Signale mit kleinerem Wertebereich zusammenzufassen, weshalb für jedes Wort immer alle 32 Bits abgespeichert werden, unabhängig davon, ob alle Bits benötigt werden oder nicht.

Das vereinfacht auch das Auslesen auf der Seite des *SpartanMCs*, denn das Hardwareinterface wird generisch gestaltet, indem zwei Leseports erstellt werden, die jeweils die unteren 18 bzw. die oberen 14 Bit des Wortes lesen. Zusätzlich wird ein Ausgangsport erstellt, der angibt, welches der aufgezeichneten Wörter gelesen werden soll. Dadurch muss nicht mehr für jede Variable, die der *SpartanMC* ausliest, ein eigener Port erstellt werden. Die Firmware des *SpartanMCs* vereinfacht sich dadurch ebenfalls, weil das Vorzeichenerweitern von Werten mit weniger als 18 Bit Breite entfällt. Allerdings muss jetzt vor jedem Lesezugriff zunächst geschrieben werden, welches Wort eingelesen wird. Damit es hier zu keinen Timingproblemen kommt, wird nach jedem Schreibzugriff kurz gewartet, bevor der Lesezugriff stattfindet.

In Abbildung 7.1 ist auch zu erkennen, dass nur noch ein *Sensor-/Actuator*-Modul Zugriff auf das *Ring Buffer*-Modul hat. Dies liegt darin begründet, dass nicht mehr wie im *Multi Bunch*-Szenario der Zugriff auf verschiedene Bunches ermöglicht werden muss. Alle *Minibunches* gehören zum selben Makropartikel und beziehen sich daher auch auf denselben Nulldurchgang, weshalb die Unterscheidung beim Lesezugriff nicht mehr nötig ist. Umgesetzt wird dies, indem alle Lesezugriffe in der Software die Methode zum Lesen vom ersten Sensor verwenden.

Da die in diesem Szenario vorgestellten Änderungen im derzeitigen Entwicklungsstadium nicht kompatibel mit dem System aus Kapitel 6 sind, wird im Toplevelmodul eine Compilerdirektive ergänzt, die vorgibt, ob das *Quadrupol*-System oder das *Multi Bunch*-System verwendet werden soll.

7.2 Softwareanpassungen

In der Software müssen nur zwei kleine Änderungen vorgenommen werden, um die Quadrupolschwingung zu unterstützen. In der Initialisierung muss dafür gesorgt werden, dass die *Minibunches* leicht zueinander versetzt sind, da sie sonst alle immer an derselben Position wären und das Ausgangssignal dasselbe wie vorher wäre. Dies lässt sich einfach realisieren, indem die Inititialwerte für Δt vom Nutzer vorgegeben werden. Des Weiteren muss der Leseport bei den Lesezugriffen für alle *Minibunches* auf Port 1 gesetzt werden, weil nur ein Bunch existiert. Zusätzlich wird außerdem das neu angepasste *DRAM Interface* genutzt, um das System auf seine Richtigkeit zu überprüfen, indem für alle *Minibunches* die Δt-Werte aufgezeichnet werden.

8 Evaluation

Das Evaluationskapitel nimmt eine zentrale Position in dieser Arbeit ein. Hier werden die Ergebnisse und Leistungen des entwickelten HiL-Simulators analysiert und bewertet. Ziel der Evaluation ist es, zu zeigen, dass die Simulation eines Teilchenbeschleunigers für praxisrelevante Szenarien unter vereinfachenden Modellannahmen in Echtzeit möglich ist. Weitergehend bietet dieser Abschnitt eine tiefgehende Einsicht in die Wirksamkeit und Effizienz der vorgestellten Ansätze.

8.1 Stationärer Fall

Die Qualität der Simulation des stationären Falls wird anhand eines Vergleichs mit einem experimentellen Test der GSI durchgeführt. Um einen fairen Vergleich zu erhalten, ist ein Test des HiL-Systems an der GSI mit der sich dort im Einsatz befindenden Mess- und Regelungstechnik sowie den dort verwendeten Methoden zur Signalerzeugung durchgeführt worden. Der experimentelle Aufbau ist in Abbildung 8.1 dargestellt. Mit Hilfe von zwei *Direct Digital Synthesis* (DDS)-Modulen werden die synchronisierten Eingangssignale erzeugt. Das gleichzeitige Resetten der Phase der beiden DDS-Module sowie das Verwenden des gleichen Taktsignals ermöglichen die extrem hohe Synchronität der Eingangssignale. Das Gapsignal wird alle $50\,\mathrm{ms}$ mit einem Phasensprung um $+/-25°$ belastet, um eine Anregung der Longitudinalschwingung zu erzielen. Dafür wird mittels eines HF-Combiners und Dämpfungsgliedern das phasenverschobene Signal aus der Inphase- und der Quadraturkomponente des Gapsignals erzeugt. Ein Hochfrequenzschalter ermöglicht das Umschalten zwischen dem originalen und dem phasenverschobenen Signal. Die Phasendifferenz zwischen dem Strahlsignal und dem Referenzsignal wird in einem DSP-System aufgezeichnet. Außerdem wird über das DSP-System der Regelkreis geschlossen und ein Signal zur Frequenzkorrektur an das für das Gapsignal verantwortliche DDS-System geschickt.

Abbildung 8.2 (oben) zeigt die aufgezeichnete Phasendifferenz zwischen Strahl- und Referenzsignal, wie sie mit dem HiL-Simulator gemessen wurden. Im unteren Teil zeigt die Abbildung zum Vergleich die Aufzeichnung eines Experiments der GSI am echten SIS18 Teilchenbeschleuniger mit einem sehr ähnlichen Setup.

Es lässt sich erkennen, dass die Reaktion des Simulators auf Phasensprünge sehr stark dem Verhalten der Ionen im echten Teilchenbeschleuniger ähneln. Wie zu erwarten, reagiert der Simulator auf den Phasensprung von $25°$ mit einer Schwingung, die eine Amplitude von $25°$ und ein um $25°$ verschobenes Schwingungszentrum aufweist. Zunächst unerwartet ist, dass die Dämpfung der

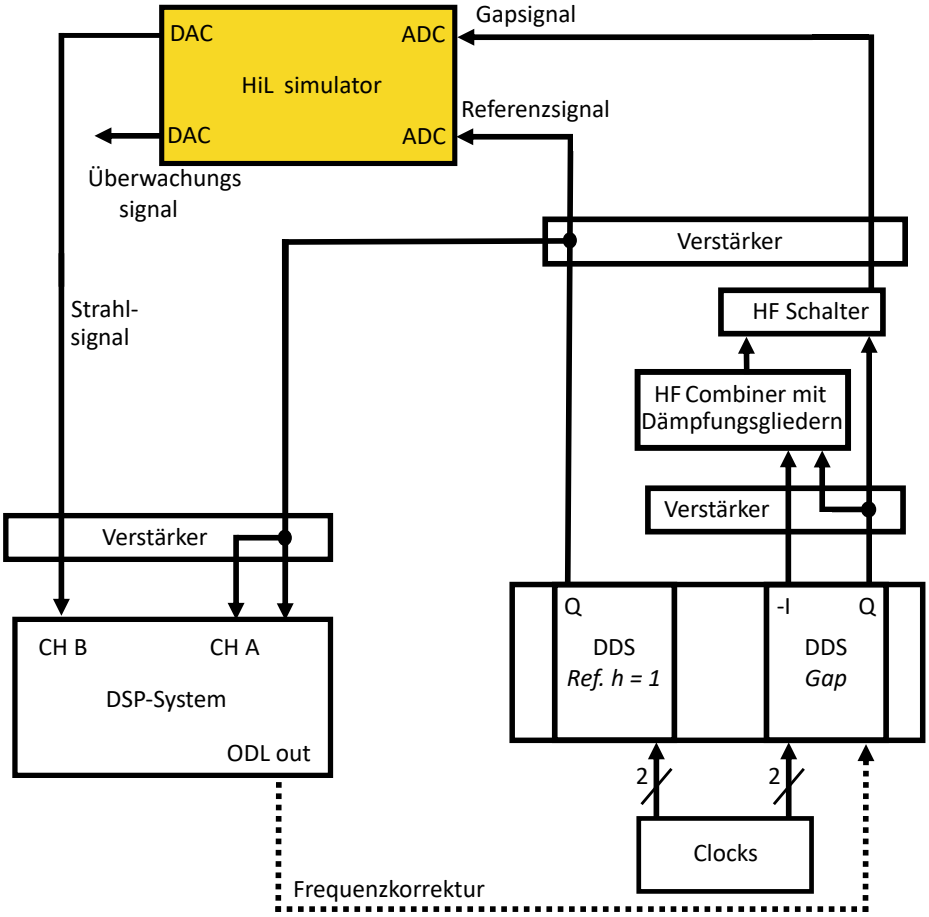


Abbildung 8.1: Blockschaltbild des Versuchaufbaus bei der GSI. Das in dieser Arbeit entwickelte System ist in gelb hervorgehoben.

Schwingung bei positivem und negativem Phasensprung im Simulator unterschiedlich schnell erfolgt. Dies lässt sich jedoch damit erklären, dass die Gapspannung nach dem negativen Sprung eine größere Amplitude besitzt, da im HF-Combiner ein gedämpfter Teil der Quadraturkomponenten des im DDS-System erzeugten Signals auf die Inphasekomponente dazuaddiert wird, um die Phasenverschiebung um $-25°$ zu erzielen. Beim positiven Phasensprung wird wieder auf das ursprüngliche im DDS-System erzeugte Signal zurückgeschaltet, weshalb die Amplitude wieder sinkt. Aus den unterschiedlichen Spannungsamplituden folgt eine unterschiedliche Frequenz der Longitudinalschwingung. Es lässt sich zeigen, dass die Frequenz proportional zur Quadratwurzel der Spannungsamplitude ist. Gleichung 8.1 zeigt, dass sich die gemessenen Werte der unterschiedlich großen Frequenzen der Longitudinalschwingung bei positiven und negativem Phasensprung durch das Verhältnis der Gapspannungen erklären lassen.

$$\sqrt{\frac{V_{negativer Phasensprung}}{V_{positiver Phasensprung}}} = \sqrt{\frac{436mV}{340mV}} \approx \frac{680Hz}{610Hz} = \frac{f_{negativer Phasensprung}}{f_{positiver Phasensprung}} \tag{8.1}$$

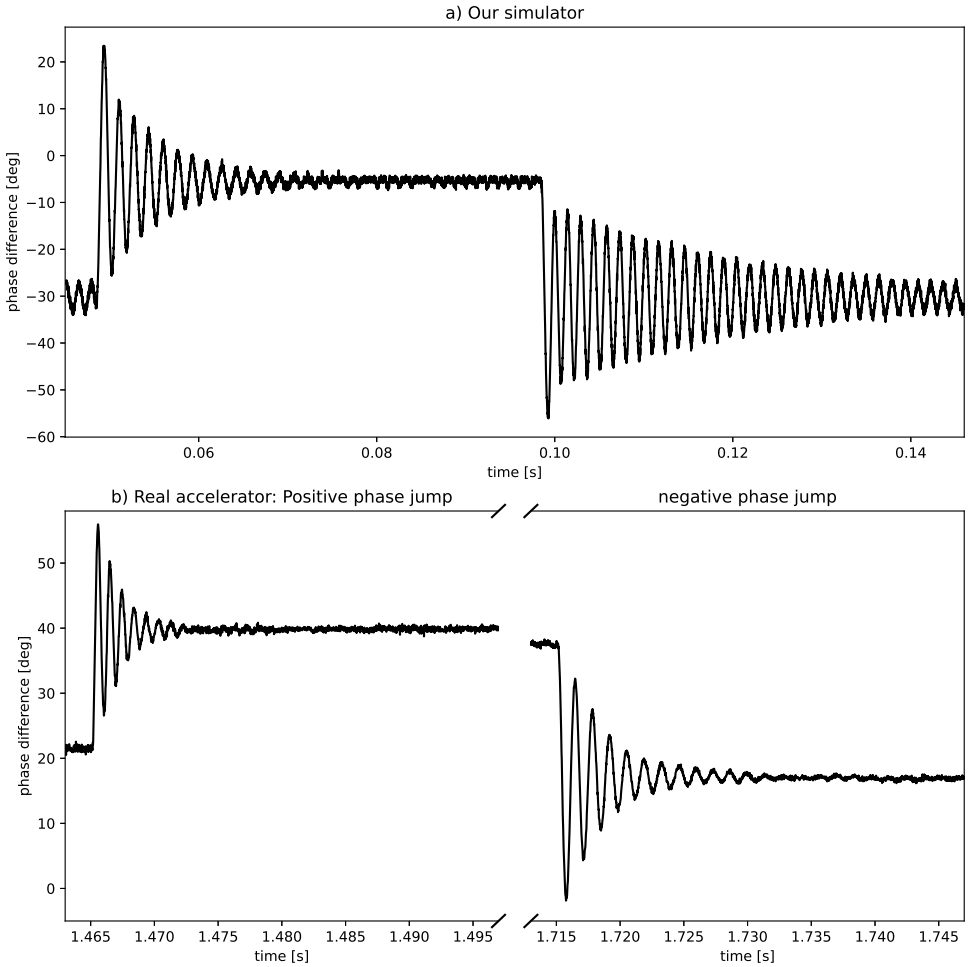


Abbildung 8.2: Messungen der Phasendifferenz zwischen Referenz- und Strahlsignal für den HiL-Simulator (oben) und dem Experiment vom 08.05.2022 der GSI mit dem SIS18, bei dem $^{238}U^{73+}$ Ionen beschleunigt wurden (unten)

Mit Hilfe der unterschiedlichen Schwingungsfrequenz kann schließlich im Frequenzgang des im DSP-System verwendeten *Finite Impulse Response* (FIR)-Filters (Abbildung 8.3) die Antwort auf die unterschiedlich starke Dämpfung gefunden werden. Durch die niedrigere Amplitude im Frequenzgang wird die Schwingung mit der leicht höheren Frequenz weniger schnell unterdrückt als die Schwingung bei etwas kleinerer Frequenz. Dieses Verhalten kann auch in der Realität beobachtet werden, wenn die Parameter des FIR-Filters noch nicht korrekt bestimmt worden sind.

Aus der experimentellen Untersuchung geht daher hervor, dass der Simulator im stationären Fall die im Synchrotron auftretenden Longitudinalschwingungen realitätsgetreu emulieren kann. Auch die Unterdrückung der Schwingung mit der an der GSI verwendeten Regelung funktioniert wie beim echten Strahl. Die hierbei durchgeführte Suche nach der Ursache für die unterschiedlich starke Dämpfung der Schwingungsamplitude für die verschiedenen Phasensprünge zeigt das enorme Potential des Simulators bei der Weiterentwicklung, Kalibrierung und Anpassung des LLRF-Systems.

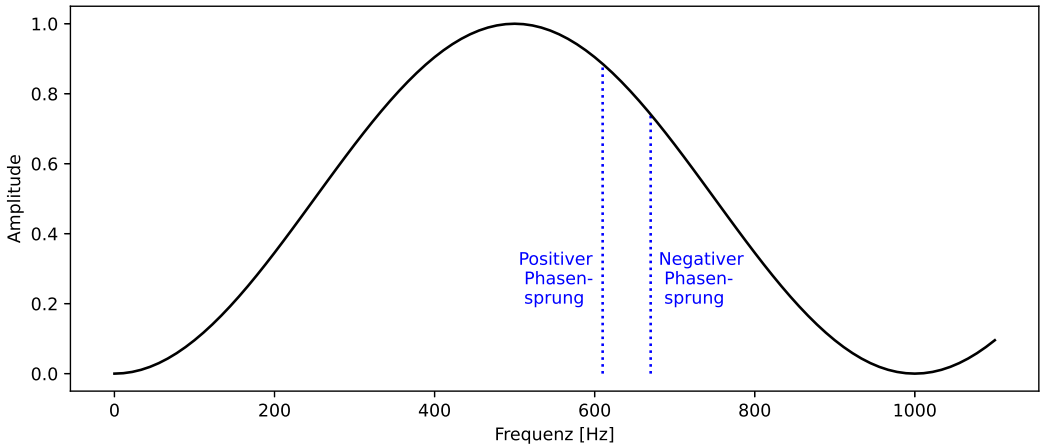


Abbildung 8.3: Frequenzgang des im DSP-System verwendeten FIR-Filters

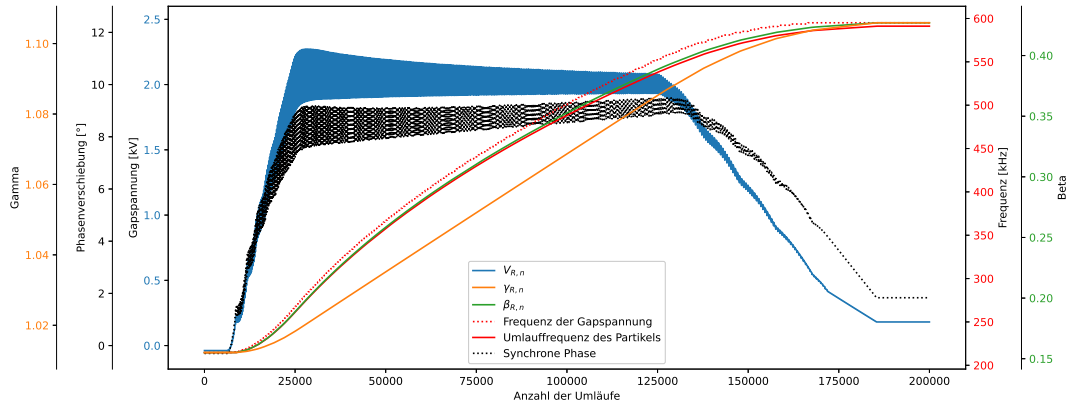
8.2 Ramp Up

Die Evaluation des *Ramp Up*-Szenarios erfolgt, wie bereits in Kapitel 5.4 erwähnt, mit Hilfe des im FPGA selbst erzeugten Testsignals. Aus diesem Grund werden zunächst die Auswirkungen der dort eingeführten Linearisierung des Spannungs- und des Frequenzverlaufs untersucht. Dafür sind in Abbildung 8.4 die Ergebnisse von drei verschiedenen Szenarien simuliert worden.

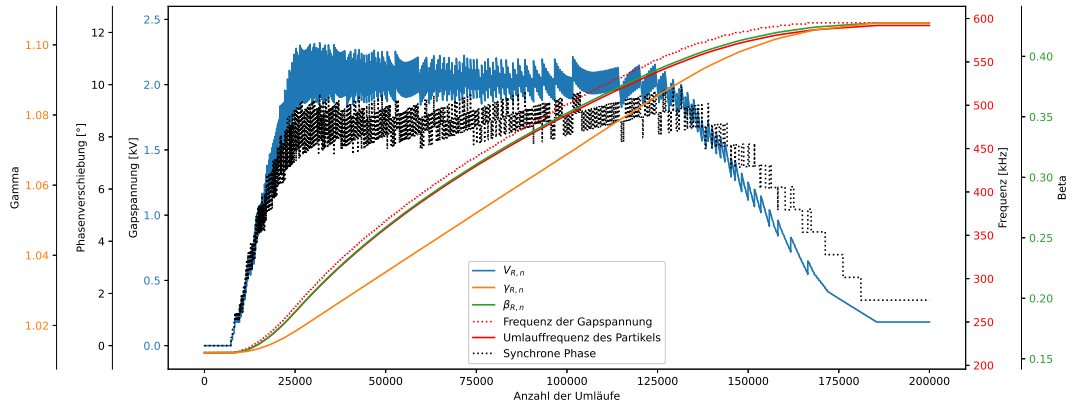
In Abbildung 8.4a ist das ideale Verhalten dargestellt. Im Modell wird hier für $V_{R,n-1}$ und V_{n-1} die exakte Spannung eingesetzt, die an der Kavität beim Eintreffen des jeweiligen Teilchens anliegt. Auch die aktuelle Frequenz wird als exakter Wert übergeben. Das hier zu sehende Verhalten entspricht dem gewünschten Verhalten.

Von diesem idealen Modell ausgehend, wird im nächsten Schritt, in Abbildung 8.4b, eine Quantisierung durchgeführt. Im Modell wird, wie im echten HiL-Simulator durch den ADC, auch die Spannung quantisiert. Zusätzlich wird die Spannung auch nicht mehr am exakten Zeitpunkt des Eintreffens des Teilchens abgefragt, sondern an den zwei am nähesten liegenden ganzzahligen Vielfachen der Taktfrequenz. Die dargestellte synchrone Phase wird hier auch, wie im echten System, quantisiert, indem sie in ein ganzzahliges Vielfaches der Systemtaktperiodendauer überführt wird. Selbiges gilt für die gemessene Periodendauer der Eingangsspannung, die in die entsprechende Frequenz umgerechnet wird. Anschließend wird, wie in Kapitel 5.3 beschrieben, eine lineare Interpolation bei der Spannungsermittlung durchgeführt. Im Vergleich zum idealen Modell in Abbildung 8.4a lässt sich erkennen, dass die Quanitisierung das Modell ungenauer gemacht hat. Vor allem der Verlauf der Spannung $V_{R,n}$ ist deutlich „zackiger“ als zuvor. Nichtsdestotrotz ist das Gesamtbild nach wie vor nahezu identisch mit dem idealen Modell.

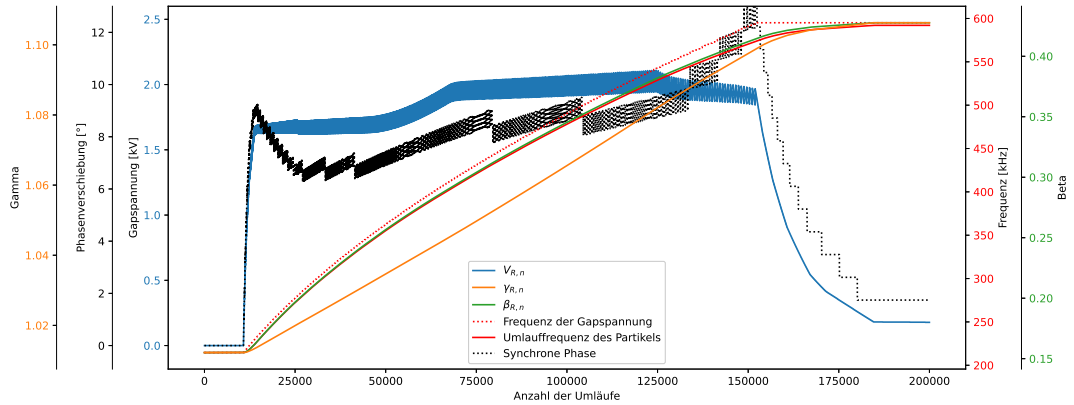
Das ändert sich, wenn die Kurven des Spannungs- und Frequenzverlaufs wie in Abbildung 5.7 linearisiert werden, obwohl die Abweichungen in Abbildung 5.7 nur sehr gering ausfallen. In Abbildung 8.4c ist das Simulationsergebnis mit den linearisierten Kurven dargestellt. Besonders auffällig sind die zwei Spitzen im Verlauf der synchronen Phase. Diese lassen sich auf die Knicke im



(a) Java-Simulation mit idealen, nicht quantisierten Eingangsgrößen (auch an nicht quantisierten Stellen gelesen)



(b) Java-Simulation mit rauschfreien, quantisierten Ein- und Ausgangsgrößen



(c) Java-Simulation mit rauschfreien, quantisierten Ein- und Ausgangsgrößen (lineare Näherung des Frequenz- und Spannungsverlaufs gemäß Abbildung 5.7)

Abbildung 8.4: Ergebnisse der Simulation des *Ramp Up*-Szenarios

linearisierten Frequenzverlauf zurückführen. Außerdem fällt auf, dass der Verlauf der Spannung nun nicht mehr so „zackig“ ist wie in 8.4b. Das lässt sich damit erklären, dass sich die vorgegebenen Daten von Frequenz und Spannungsamplitude gemäß Abbildung 5.1 im Gegensatz zu den linearen interpolierten Werten nicht kontinuierlich ändern. Insgesamt besteht der vorgegebene Datensatz aus 407 Punkten, die meist einen Abstand von 1 ms zueinander aufweisen. Frequenz und Spannungsamplitude bleiben daher für diese Dauer konstant und springen anschließend zum nächsten Wert, wodurch die Sprünge in Abbildung 8.4b entstehen. Dieses Phänomen wird durch die lineare Annäherung der Eingangsgrößen unterbunden.

Auch wenn sich das Ergebnis dieses Modells erheblich vom idealen Modell in 8.4a unterscheidet, ist es nach wie vor durchaus in der Lage, die Beschleunigung der Teilchen sinnvoll darzustellen. Der Verlauf der Lorentzfaktoren und der Umlauffrequenz des Teilchens ist hier noch immer nahezu identisch mit dem Verlauf im idealen Modell. Dementsprechend kann das selbst erzeugt Signal für eine Bewertung der Qualität des HiL-Simulators beim Emulieren des *Ramp Up*-Szenarios genutzt werden. Es muss dabei lediglich darauf geachtete werden, die Messergebnisse nicht mit dem idealen Verlauf aus 8.4a, sondern mit den Signalen aus 8.4c zu vergleichen.

Die aufgezeichneten Daten aus dem Test mit dem HiL-Simulator sind in Abbildung 8.5 zu sehen. Der Verlauf der gemessenen Spannung ist etwas mit Rauschen belastet, aber das Gesamtbild entspricht qualitativ genau dem, was in der Java-Simulation berechnet wurde. Wie auch in der Java-Simulation sind die zwei Spitzen in der synchronen Phase zu erkennen. Die Beschleunigung des Partikels erfolgt ebenfalls wie gewünscht, was am quasi identischen Verlauf der Kurve der Umlauffrequenz des Partikels in der Java-Simulation im Vergleich zu den am HiL-Simualtor gemessenen Daten erkannt werden kann.

Die Ergebnisse sind äußerst vielversprechend und belegen, dass die in Kapitel 5 vorgestellten Anpassungen am HiL-System und am Modell eine Simulation des *Ramp Up*-Szenarios ermöglichen.

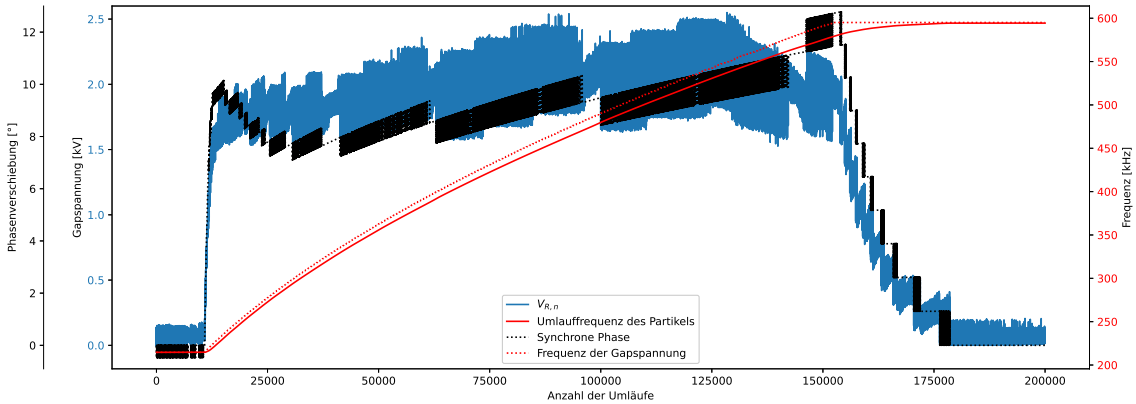


Abbildung 8.5: Aufgezeichnete Daten aus dem HiL-Simulator

8.3 Multi Bunch

Die Evaluation der Implementation für das Multi Bunch-Szenarios erfolgt in Bezug darauf, wie viele Bunches mit dem HiL-Simulator stabil simuliert werden können.

Zunächst wird dabei auf die Zuverlässigkeit der Bestimmung der Anzahl der Bunches im System zur Laufzeit eingegangen. Wie bereits im Kapitel 6.1.3 erwähnt, arbeitet das System hierbei sehr zuverlässig. Es ist in keinem einzigen Versuch vorgekommen, dass die Anzahl falsch ermittelt wurde.

Die eigentliche Simulation des *Multi Bunch*-Szenarios hingegen stellt sich als nicht ganz so stabil heraus. Prinzipiell ist das System in der Lage, bis zu acht Bunches gleichzeitig zu simulieren und unabhängig voneinander zu berechnen. Für eine kleinere Anzahl an Bunches funktioniert das auch recht zuverlässig. Das System wird ab $h \geq 4$ allerdings zunehmend instabiler und fängt mit größer werdender Bunchzahl an, sich immer schneller aufzuschaukeln.

Dieses Instabilitätsproblem ist bereits während der Entwicklung aufgefallen, weshalb, wie in Kapitel 6.1.2 beschrieben, mehrere Methoden zur Bestimmung der Spannungen ausprobiert wurden. Allerdings konnte keine dieser Methoden das System für größere Werte von h hinreichend stabilisieren. Das HiL-System muss daher noch weitergehend optimiert werden, um diesen Fall zuverlässiger simulieren zu können.

Die Ursache des Problems liegt vermutlich in Ungenauigkeiten bei der Bestimmung der Nulldurchgänge. Durch leichte Verschiebungen werden aufgrund der h-fach höheren Frequenz der Gapspannung bereits relativ große Spannungen gemessen, die dem System immer weiter Energie hinzufügen.

8.4 Quadrupolschwingung

Um das Modell zum Emulieren der Quadrupolschwingungen zu beurteilen, wird im Folgenden die Qualität des erzeugten Ausgabesignals eingeschätzt. Dafür wird das Modell zum einen in einer Java-Simulation ausgewertet und zum anderen werden im HiL-Simulator die Δt_n-Werte der einzelnen *Minibunches* aufgezeichnet. In beiden Fällen wird ein Bunch durch neun *Minibunches* dargestellt. Die Schwingung der *Minibunches* wird dadurch angeregt, dass alle *Minibunches* identisch initialisiert werden und im Abstand von jeweils 10 Taktperioden des Systemtakts angeordnet sind. Das gewünschte Verhalten des Ausgabesignals ist, dass zwischen einer schmalen, spitzen Gaußkurve und einer breiten, flachen Gaußkurve hin und her geschwungen wird. Abbildung 8.6 zeigt die dabei erzielten Ergebnisse. Um die kontinuierliche Schwingung darstellen zu können, wird das Ausgabesignal an den zwei Extrema (grüne und schwarze Kurve) sowie an zwei dazwischen liegenden Stellen (grüne und rote Kurve) abgebildet. Es ist erkennbar, dass beide Plots das gewünschte Verhalten des Ausgabesignals zeigen. Das Modell ist daher in der Lage, Quadrupolschwingungen erfolgreich zu emulieren. Allerdings sind auch Unterschiede zwischen der Java-Simulation und dem HiL-System zu erkennen.

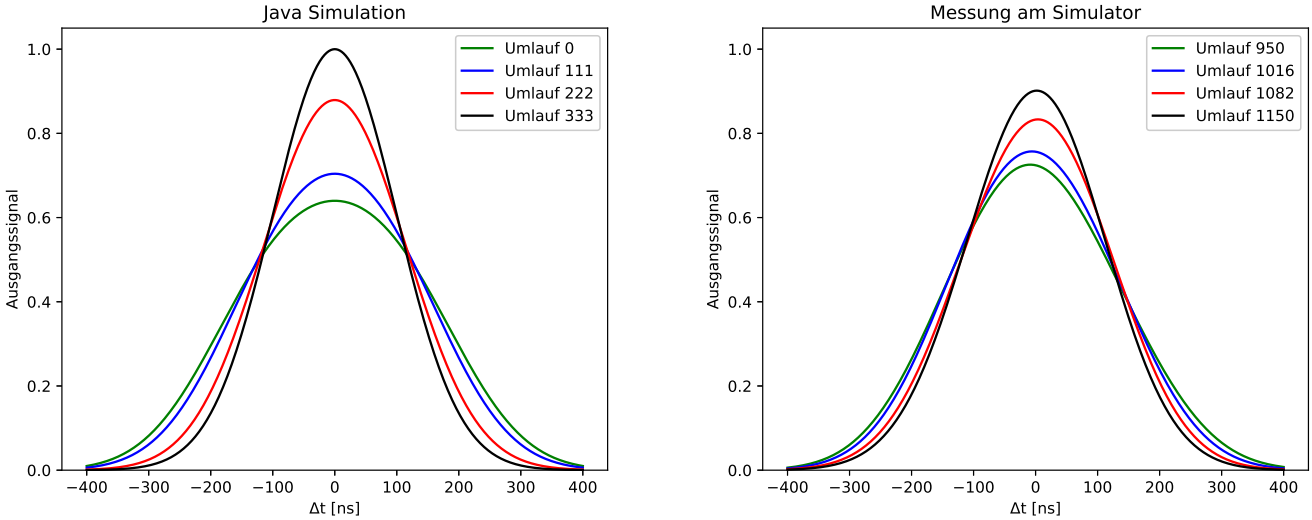
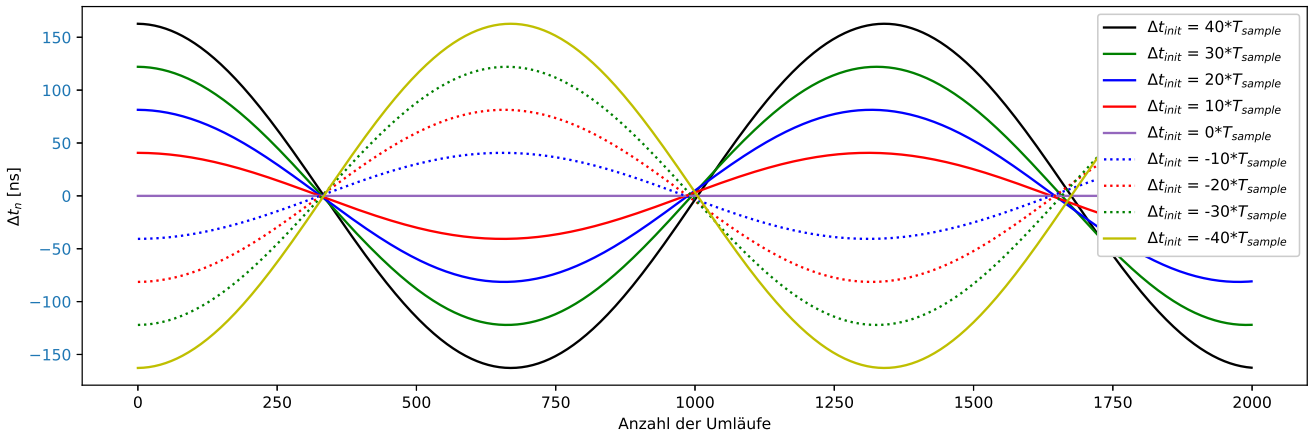


Abbildung 8.6: Verhalten des Quadrupolmodels mit neun *Minibunches* in der Java-Simulation und aufgezeichnet vom HiL-Simulator

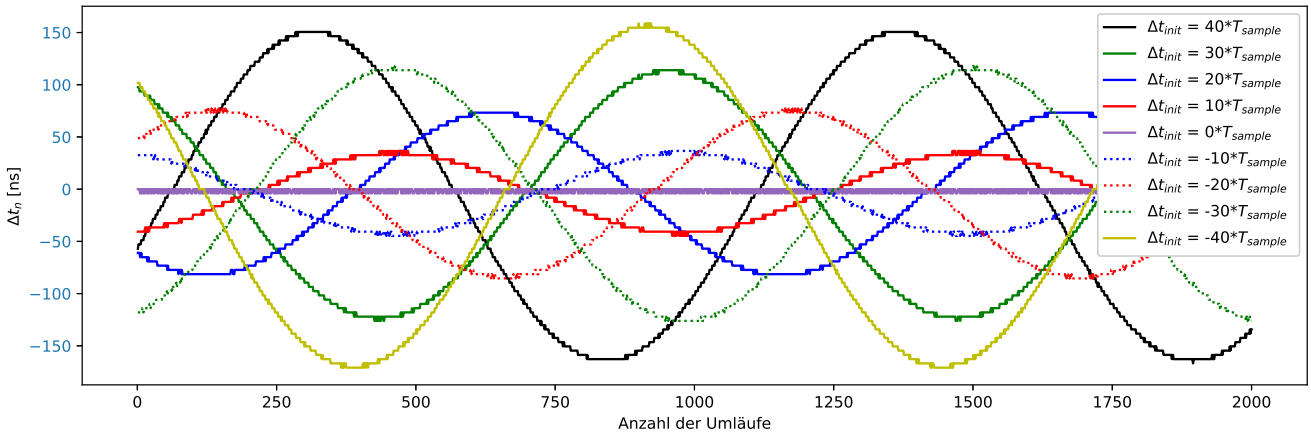
In der Java-Simulation sind die Extrema der Schwingung stärker ausgeprägt als in der Messung am HiL-System. Außerdem ist die Zeit zwischen den Extrema unterschiedlich (333 Umläufe bzw. 200 Umläufe). Begründen lässt sich das damit, wie die Initialisierung durchgeführt wurde. Alle *Minibunches* werden mit derselben kinetischen Energie initialisiert. Durch die unterschiedlichen Startpositionen der *Minibunches* besitzen sie aber unterschiedlich viel potentielle Energie und verlaufen auf unterschiedlichen Orbits. Das führt zu leicht unterschiedlichen Frequenzen, mit denen die *Minibunches* um das Referenzpartikel schwingen. In Abbildung 8.7 ist für alle neun *Minibunches* der Verlauf von Δt_n für 2000 Synchrotrondurchquerungen für beide Modelle dargestellt.

In der Java-Simulation beginnt die Aufzeichnung sofort nach der Initialisierung. Hier kann man erkennen, dass die einzelnen *Minibunches* zunächst noch quasi synchronisiert sind, aber schon beim dritten Nulldurchgang kann deutlich erkannt werden, dass die Kurven unterschiedliche Frequenzen haben, da sie nicht zum selben Zeitpunkt die x-Achse schneiden. Die Messung am HiL-Simulator ist im Gegensatz zur Java-Simulation nicht zum Zeitpunkt $t = 0s$ gestartet worden. Es ist schon etwas Zeit vergangen, bevor die ersten Werte aufgezeichnet werden. Dementsprechend sind die Schwingungen der einzelnen *Minibunches* nicht so snychronisiert wie in der Java-Simulation. Damit lässt sich erklären, warum die Extrema der Gaußkurve in Abbildung 8.6 bei den gemessenen Werten nicht so scharf sind wie bei der Java-Simulation, denn die einzelnen *Minibunches* sind während der Messung nie äquidistant verteilt oder alle gleichzeitig bei $\Delta t_n = 0$.

Auch die unterschiedliche Zeit zwischen den Extrema lässt sich so nachvollziehen. In Abbildung 8.7 ist klar erkennbar, dass die Schwingungsfrequenzen der einzelnen *Minibunches* in der Java-Simulation sehr ähnlich, wenn nicht sogar identisch, mit denen aus der Messung sind. Es ist also keine unterschiedlich schnelle Schwingung der einzelnen *Minibunches*, die zu einer unterschiedlichen Dauer zwischen den Extrema führt. Da die einzelnen *Minibunches* in der Java-Simulation quasi synchron sind, ist die Zeit zwischen den Extrema der Gaußkurve fast exakt ein Viertel der



(a) Java-Simulation



(b) Messung am HiL-Simulator

Abbildung 8.7: Verlauf vonΔt_n der einzelnen *Minibunches*

Schwingungsdauer der *Minibunches*, wie in Abbildung 8.7a erkennbar ist. Die nicht synchronen *Minibunches* sorgen aber auch dafür, dass es keinen scharfen Zeitpunkt gibt, an dem das Ausgabesignal am spitzesten oder am flachsten ist. Dementsprechend länger ist die Gaußkurve in einem Extremum (das dafür weniger scharf ausgeprägt ist) und deshalb wird die Zeit, zu der eine nennenswerte Änderung der Form der Glockenkurve beobachtet werden kann, geringer.

Zu diesem Effekt trägt ebenfalls bei, dass die Messungen von Δt immer in ganzzahligen Vielfachen der Dauer des Systemtakts angegeben sind. Diese Quanitisierung sorgt zusätzlich für die leichte Unschärfe in der Form der Gaußkurve, ist aber im Vergleich zu dem eben beschriebenen Verhalten vernachlässigbar.

Des Weiteren kann auch beobachtet werden, dass sich das Zentrum der Gaußkurve leicht bewegt (z.b. bei der roten Kurve in 8.6). Dies lässt sich einfach damit erklären, dass es Zeitpunkte gibt, an denen sich, auf Grund der Asynchronität, keine gleichmäßige Verteilung der *Minibunches* einstellt. Dadurch wird das Zentrum der Glockenkurve in die eine oder in die andere Richtung verschoben.

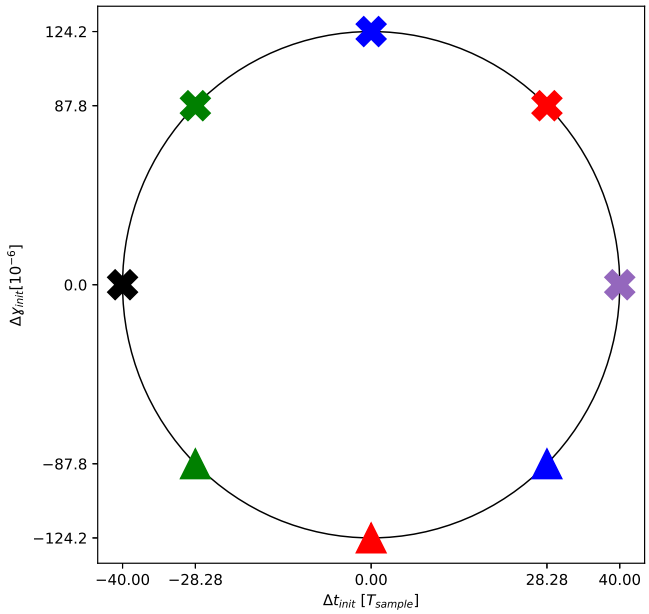
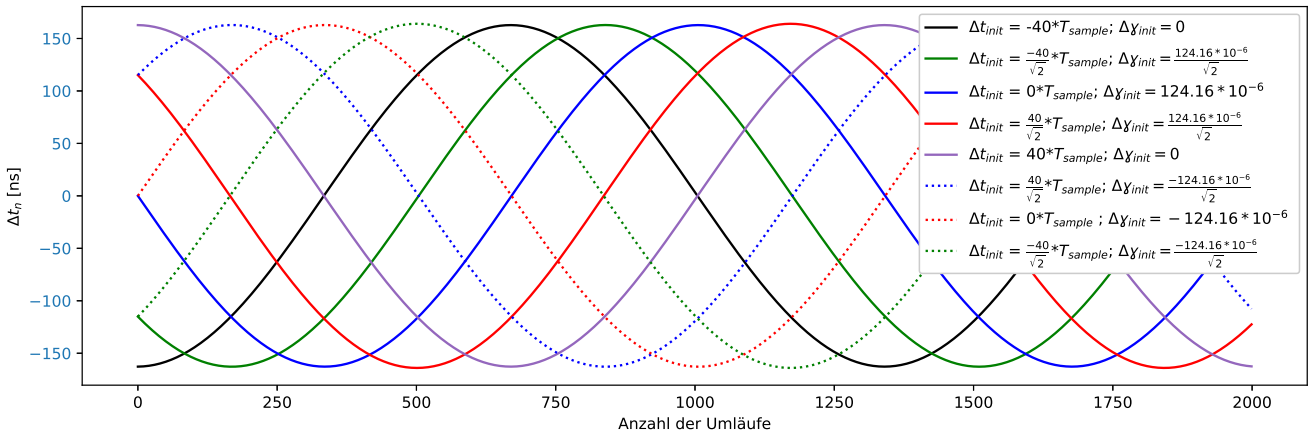


Abbildung 8.8: Die verwendeten Initialisierungswerte entlang eines Kreises. Die Farben korrespondieren zu den Farben in Abbildung 8.9. Mit einem X markierten Punkte gehören dort zu den durchgezogenen Linien, die Dreiecke zu den gepunkteten Linien.

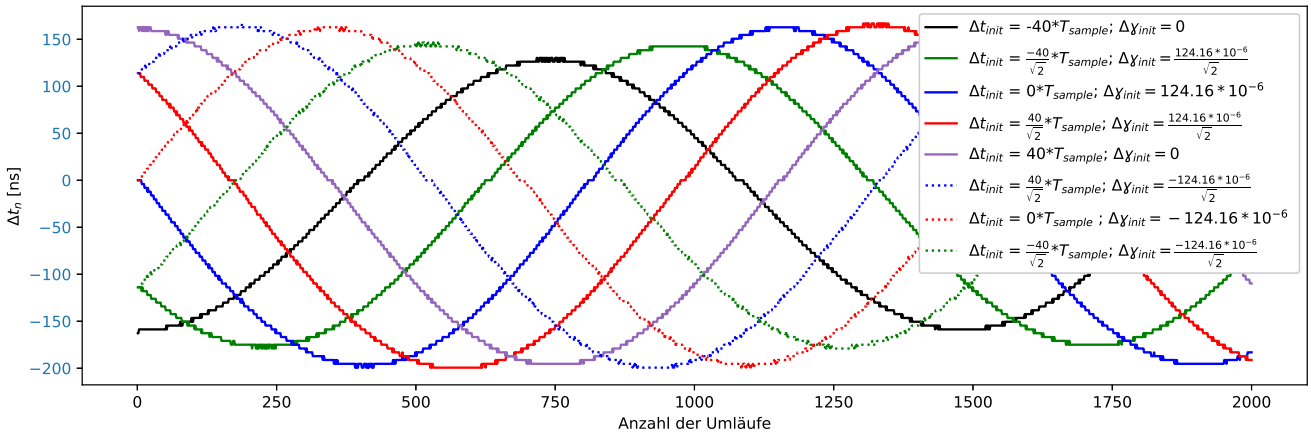
In einem weiteren Experiment wird versucht, die Initialisierung der *Minibunches* mit unterschiedlich viel Energie zu vermeiden. Dafür werden die einzelnen *Minibunches* entlang eines Kreises angeordnet, der den Verlauf von potentieller und kinetischer Energie entlang eines Orbits darstellen soll. Die einzelnen *Minibunches* werden mit einem Abstand von $45°$ zueinander entlang des Kreises verteilt. Dementsprechend gibt es in diesem Experiment nur acht anstatt der vorher verwendeten neun *Minibunches*. Abbildung 8.8 stellt diese Anordnung grafisch dar.

Die gemessenen Ergebnisse dieses Experimentes sowie die Ergebnisse der Java-Simulation mit der neuen Initialisierung sind in Abbildung 8.9 zu sehen. Aus dem Ergebnis der Java-Simulation lässt sich erkennen, dass nun alle *Minibunches* mit derselben Frequenz schwingen, da sich die Kurven zweier jeweils um 180° versetzter *Minibunches* immer exakt auf der x-Achse schneiden. Es tritt außerdem kein Schwingen mehr zwischen einem breiten und einem schmalen Gaußimpuls auf. Dieses Verhalten, dass das Ausgangssignal quasi konstant bleibt, ergibt sich durch die gleichmäßige Verteilung der einzelnen *Minibunches* entlang des Kreises in Abbildung 8.8. Im Vergleich zu Abbildung 8.7a lässt sich erkennen, wie somit die Schwingung des Ausgangssignals unterdrückt wird. Die breite Gaußkurve liegt dort im ersten Umlauf vor, wenn alle *Minibunches* äquidistant verteilt sind, das spitze Ausgangssignal im 333. Umlauf, wenn alle *Minibunches* auf der x-Achse liegen. Der letzte Fall kann im neuen Experiment nie auftreten, denn wenn hier ein *Minibunch* weiter nach außen wandert, gibt es immer einen anderen *Minibunch*, der dies kompensiert, indem er näher in die Mitte rückt. Daher kann hier nie eine derart spitze Gaußkurve erzeugt werden, da die *Minbunches* immer gleichmäßig verteilt sind.

Die gemessenen Ergebnisse am HiL-Simulator geben ein der Java-Simulation qualitativ sehr ähnli-



(a) Java-Simulation



(b) Messung am HiL-Simulator

Abbildung 8.9: Verlauf vonΔt_n der einzelnen *Minibunches* im neuen Experiment

ches Bild wieder. Es lässt sich aber erkennen, dass die grünen und die schwarze Kurve eine etwas geringere Amplitude aufweisen. Dies lässt sich auf den nicht perfekt bestimmten Nulldurchgang der Gapsspannung zurückführen, sodass ein negatives Δt_n etwas weniger potentielle Energie darstellt als ein positives Δt_n. Deshalb hat die schwarze Kurve auch die geringste Amplitude, denn ihre Energie stammt ausschließlich aus dem initialen Wert von Δt_n. Die grünen Kurven haben nur die Hälfte ihrer Energie aus dem Initialwert von Δt_n und die andere Hälfte aus $\Delta\gamma_{n,init}$, weshalb ihre Amplitude zwischen der der schwarzen Kurve und den anderen Amplituden liegen. Diese leicht unterschiedlichen Amplituden führen auch wieder zu sich unterscheidenden Umlauffrequenzen, weshalb bei der Messung für diese Abbildung sichergestellt werden musste, dass die Aufzeichnung mit der ersten Auswertung des Modells beginnt. Eine durchgeführte Messung, die nicht sofort gestartet wurde, wie im Experiment mit den neun *Minibunches*, hat gezeigt, dass sich durch die unterschiedlichen Frequenzen die Kurven bereits verschoben haben und sich daher ein ungleichmäßigeres Bild zeigt.

CGRA Struktur \ Anzahl der *Minibunches*	9	10	11	12	13
4x4	118	122	135	138	144
5x5	113	121	124	130	128

Tabelle 8.1: Ergebnisse der Suche nach der maximal möglichen Anzahl an simulierten *Minibunches*. Dargestellt ist die Anzahl der benötigten Takte für die Hauptschleife.

8.4.1 Maximale Anzahl an Minibunches

Im Anschluss an die Analyse des Ausgangsignals der Quadropolschwingung ist eine weitere interessante Fragestellung, wie viele *Minibunches* potentiell im System simuliert werden können, ohne die Echtzeitanforderungen zu verletzen. Dafür wird zunächst einfach die Anzahl der *Minibunches* in der Software vergrößert. In Tabelle 8.1 sind die Ergebnisse dieser Untersuchung für zwei verschiedene CGRA-Konfigurationen mit 16 bzw. 25 PEs dargestellt. Das *Sensor-/Actuator*-Modul wurde dabei an allen Positionen ausprobiert, was aber aufgrund der symmetrischen CGRA-Struktur mit *Star-Interconnect* mit *Wrap Around* immer zum selben Ergebnis geführt hat. Ein Schedule, der die Echtzeitanforderungen erfüllen soll, darf eine maximale Ausführungsdauer für die Hauptschleife von 113 Takten besitzen. Dementsprechend ist es zunächst nicht möglich, mehr als die zuvor bereits verwendeten neun *Minibunches* mit diesen CGRA-Konfigurationen zu simulieren.

Beim Inspizieren der Schedules ist dabei aber aufgefallen, dass die meisten PEs aufwendige Funktionen wie die Gleitkommadivision oder die Quadratwurzelberechnung nie benötigen. Um das Timing zu verbessern wird daher die Hardware für diese Operationen von einigen PEs entfernt, in der Hoffnung, das CGRA schneller takten zu können, falls weniger Hardware in den PEs verbaut wird. Dadurch sollten Placement und Routing besser werden, was zu einer Verbesserung des kritischen Pfades führen müsste. Unglücklicherweise konnte keine nennenswerte Verkürzung des kritischen Pfades erreicht werden, sodass die Taktperiode nach wie vor $8,875$ ns beträgt.

Allerdings konnte ein anderer, seltsamer Effekt beobachtet werden. Wenn Ressourcen entfernt werden, wird manchmal ein kürzerer Schedule gefunden, selbst dann, wenn diese Ressourcen vorher nie benutzt wurden. Die plausibelste Erklärung hierfür ist, dass die Operationen nicht zu weit auf dem CGRA verteilt werden, weil die vorhandenen Ressourcen einen Sammelpunkt vorgeben, zu dem die Daten gelangen müssen. Dadurch bilden sich Cluster und es müssen weniger Verzögerungen dafür eingebaut werden, Daten an weiter entfernte PEs zu liefern. Dies ist dennoch ein Verhalten, das so nicht auftreten sollte und ein Anzeichen für einen Fehler im Schedulingalgorithmus ist, da die Implementierung auf einer Abänderung des *List Scheduler*-Algorithmus basiert, welcher durch zusätzliche Ressourcen niemals ein schlechteres Ergebnis liefert. Nichtsdestotrotz kann diese Besonderheit ausgenutzt werden, um einen schnelleren Schedule zu finden, der die Simulation zusätzlicher *Minibunches* ermöglicht.

Das Finden der idealen Konfiguration für einen möglichst kurzen Schedule ist dabei keine triviale Aufgabe. Wenn nicht alle PEs alle Ressourcen besitzen, vergrößert sich die Anzahl an möglichen Konfigurationen extrem schnell. Bei n PEs ist die Anzahl kombinatorisch möglicher Konfigurationen in $\mathcal{O}(n!)$, selbst wenn nur eine Ressource betrachtet wird. Dementsprechend ist eine

erschöpfende Suche keine praktisch realisierbare Option. Es kann auch kein sinnvoller Zusammenhang zwischen den Änderungen an der Konfiguration und der Qualität des Ergebnisses erkannt werden. Das Ergebnis ändert sich völlig zufällig, wenn die Konfiguration geändert wird, sodass es leider nicht möglich ist, ein iteratives Lösungsverfahren, wie das *Simulated Annealing*, oder ein analytischen Verfahren anwenden zu können. Stattdessen wird eine kleine *Design Space Exploration* durchgeführt, bei der für eine vorgegebene Anzahl an Ressourcen jeweils 3000 zufällige Anordnungen dieser Ressourcen auf dem CGRA ausprobiert werden.

In den Tabellen 8.2 und 8.3 sind die Ergebnisse dieser Suche dargestellt. Untersucht wurde die Anzahl und die Anordnung der zwei aufwendigeren, nicht sonderlich häufig benötigten Operationen der Gleitkommadivision und des Quadrahtwurzelziehens sowie die Positionierung des *Sensor-/Actuator*-Moduls. Die Anzahl der PEs mit Wurzel-(SQRT) und Divisionsoperatoren (DIV) wird dabei von zwei bis fünf bzw. von drei bis zehn variiert. Ausgangspunkt dafür war, dass beim System mit neun *Minibunches* für die Untersuchung des Strahlsignals insgesamt nur auf sieben PEs die Division und auf vier PEs das Wurzelziehen verwendet wurde.

	Wurzel \ Division	3	4	5	6	7	8	9	10
10 Bunches	2	116	116	116	116	115	116	115	117
	3	117	115	115	115	114	115	116	115
	4	116	115	116	115	116	115	115	115
	5	116	116	116	116	116	116	114	115
11 Bunches	Wurzel \ Division	3	4	5	6	7	8	9	10
	2	121	121	121	121	121	119	121	120
	3	119	119	119	120	120	120	120	120
	4	121	119	120	120	119	119	120	121
	5	119	120	120	119	118	119	120	119
12 Bunches	Wurzel \ Division	3	4	5	6	7	8	9	10
	2	124	125	124	125	125	125	124	123
	3	124	125	124	125	124	124	124	124
	4	124	124	123	123	124	124	124	125
	5	124	124	124	122	124	122	123	124
13 Bunches	Wurzel \ Division	3	4	5	6	7	8	9	10
	2	128	129	128	128	126	126	127	128
	3	128	128	129	127	126	126	128	127
	4	129	128	128	127	128	128	127	126
	5	130	127	127	127	127	127	127	127

Tabelle 8.2: Ergebnisse der Suche nach der minimalen Schedulelänge für ein CGRA mit 4x4 PEs. Dargestellt ist die Anzahl der benötigten Takte für die Hauptschleife.

10 Bunches	Wurzel \ Division	3	4	5	6	7	8	9	10
	2	114	113	111	113	112	112	113	113
	3	113	113	113	112	112	112	113	112
	4	112	113	113	112	112	113	111	113
	5	113	113	113	111	112	112	111	113

11 Bunches	Wurzel \ Division	3	4	5	6	7	8	9	10
	2	116	115	113	115	113	115	115	115
	3	116	115	114	115	115	114	115	116
	4	116	113	115	115	114	114	116	115
	5	113	116	116	115	114	114	115	114

12 Bunches	Wurzel \ Division	3	4	5	6	7	8	9	10
	2	120	120	120	117	119	118	119	118
	3	119	119	119	118	118	118	118	119
	4	118	118	117	119	118	118	119	119
	5	119	119	119	119	118	117	118	119

13 Bunches	Wurzel \ Division	3	4	5	6	7	8	9	10
	2	124	123	123	122	123	123	122	121
	3	122	122	123	123	121	123	122	122
	4	121	122	121	120	123	122	120	120
	5	123	122	122	122	121	121	122	121

Tabelle 8.3: Ergebnisse der Suche nach der minimalen Schedulelänge für ein CGRA mit 5x5 PEs. Dargestellt ist die Anzahl der benötigten Takte für die Hauptschleife.

Die Ergebnisse in den Tabellen sind allesamt bis auf eine Ausnahme (5x5 CGRA mit 2 SQRT- und 3 DIV-Ressourcen bei 10 Bunches) mindestens genau so gut wie und in den meisten Fällen deutlich besser als der gefundene Schedule, wenn alle PEs über alle Ressourcen verfügen. Vor allem wenn mehr *Minibunches* berechnet werden sollen, kann ein deutlicher Unterschied in der Ergebnisqualität festgestellt werden. Die Verbesserung geht so weit, dass für ein 5x5 CGRA Konfigurationen gefunden wurden, mit denen elf statt wie zuvor neun *Minibunches* simuliert werden können, ohne die Echtzeitanforderungen zu verletzen. Leider lässt sich kein klarer Trend erkennen, dass das Hinzufügen von mehr Ressourcen eines Types bessere Ergebnisse liefern würde. Die besten Ergebnisse sind teilweise sogar in Konfigurationen mit den wenigsten SQRT- bzw. DIV-Ressourcen gefunden worden. Das Einzige, was relativ deutlich wird, ist, dass die Kombination aus drei DIV- und zwei SQRT-Ressourcen zu wenige Ressourcen zur Verfügung stellt, denn hier werden nie gute Ergebnisse erzielt. Prinzipiell müsste die Suche auch für Konfigurationen mit mehr SQRT-Ressourcen durchgeführt werden, aber da sich bisher kein klarer Trend herauskristalisiert hat, ist nicht davon auszugehen, dass sich das Bild maßgeblich ändert, wenn weitere Konfigurationen untersucht werden. Vor allem auch deshalb, weil sich bei wiederholter Ausführung herausstellt, dass

andere Ergebnisse erzielt wurden. Die 3000 betrachteten Verteilungen der jeweiligen Ressourcen sind nicht ansatzweise genügend, um zuverlässig eine der besten Lösungen in diesem riesigen Suchraum zu finden, sodass das letztlich erzielt Ergebnis vom Zufall abhängt.

Die nur geringen Abweichungen in der erzielten Schedulelänge und die Tatsache, dass die beste Konfiguration an völlig verschiedenen Punkten in der Tabelle gefunden werden kann, lässt zusätzlich darauf schließen, dass die eigentliche Anzahl der Ressourcen keine wesentliche Rolle in der gefundenen Ergebnisqualität spielt. Stattdessen hängt diese eher davon ab, wie gut die Anordnung der Ressourcen die eigentümliche Besonderheit des Schedulingalgorithmus unterstützt oder behindert. Leider konnte allerdings kein Muster in den Anordnungen, die zu besonders guten Ergebnissen führen, erkannt werden, um daraus ein System entwickeln zu können, mit dem gezielt nach neuen Anordnungen gesucht werden könnte. Untersucht wurde auch, ob eine CGRA-Konfiguration, die bei einer Suche ein gutes Ergebnis geliefert hat, für mehr oder weniger *Minibunches* auch einen guten Schedule erstellt. Es hat sich dabei gezeigt, dass sich die gute Ergebnisqualität nicht einfach so übertragen lässt.

8.5 Harwarebedarf

Abschließend wird noch ein Überblick über den Hardwarebedarf für das Gesamtsystem auf dem *VC707 Evalutaion Kit* gegeben. In Tabelle 8.4 ist eine Auflistung der verwendeten Ressourcen zu sehen. Abbildung 8.10 zeigt in grafischer Form, welche Elemente des FPGAs benutzt werden. Es lässt sich erkennen, dass das FPGA noch lange nicht ausgelastet ist, sodass das System noch deutlich vergrößert werden könnte, um beispielsweise mehr Bunches simulieren zu können. Problematisch ist bei einer Vergrößerung des CGRAs aber voraussichtlich nicht die Menge an verfügbaren LUTs und DSP-Blöcken, sondern eher die Routing-Ressourcen. Beim Vergrößern des CGRAs wird das Routing immer schwieriger und dadurch wird der kritische Pfad länger, sodass Timingprobleme auftreten können.

Ressource	Relative Nutzung	Absolute Anzahl
LUT	26%	79576
LUTRAM	3%	4263
Flip Flops	6%	35761
BRAM	18%	188
DSP	2%	69
BUFG	34%	11
MMCM	29%	4
PLL	7%	1

Tabelle 8.4: Hardwarebedarf des FPGA-Designs in absoluten und relativen Zahlen.

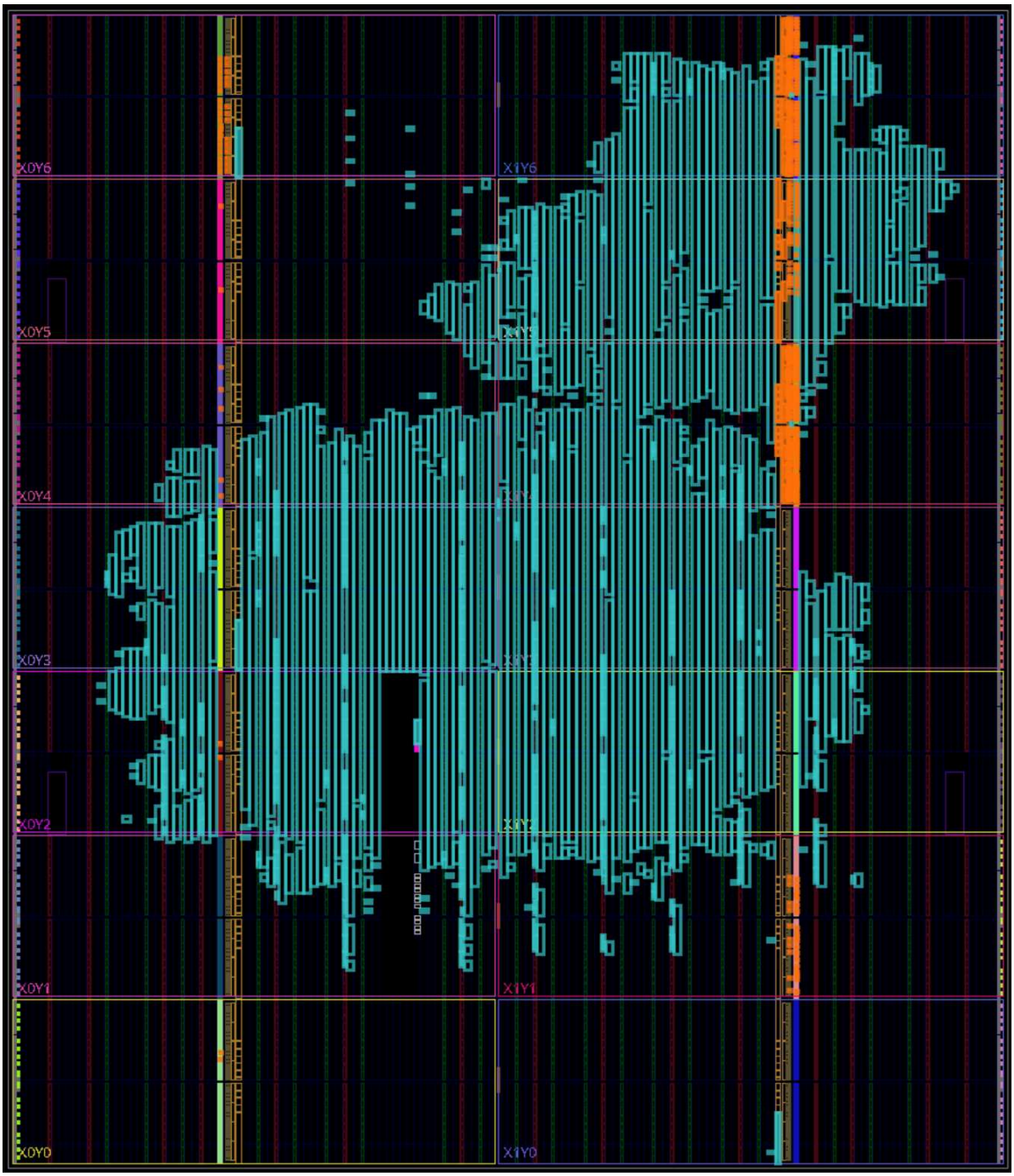


Abbildung 8.10: Grafische Darstellung des Ressourcenverbrauchs auf dem FPGA

9 Fazit

In dieser Masterarbeit wurde erfolgreich ein HiL-System weiterentwickelt, um die Simula-tion eines Teilchenbeschleunigers unter Einhaltung strenger Echtzeitanforderungen zu ermöglichen. Dieses System kann dabei nicht nur für die Simulation des stationären Falls des Teilchenbeschleunigers genutzt werden, sondern bietet durch neue Anpassungen die Möglichkeit, auch verschiedene komplexere, äußerst relevante Szenarien unter vereinfachenden Modellannahmen darzustellen. Die Simulation des *Ramp Up*-Szenarios und der Quadrupolschwingungen haben dabei sehr vielversprechende Ergebnisse geliefert. Ein positiv überraschendes Ergebnis war dabei, wie realitätsnah die Quadrupolschwingungen selbst mit einer nur einstelligen Anzahl an *Minibunches* dargestellt werden konnten.

Das wichtigste Ergebnis dieser Arbeit liegt im Darlegen der potentiellen Möglichkeiten, die das HiL-System in der Weiterentwicklung der Steuerelektronik für ein äußerst komplexes System wie einen Teilchenbeschleuniger bieten könnte. Die Aussicht, die Steuerelektronik in einem sicheren und kontrollierten Umfeld zu testen und zu kalibrieren, bevor sie im tatsächlichen Betrieb des Teilchenbeschleunigers eingesetzt wird, könnte sich als äußerst wertvoll erweisen.

Durch die Vorab-Kalibrierung und Optimierung der Steuerelektronik mit Hilfe des HiL-Systems können Experimente effizienter und zuverlässiger durchgeführt und wertvolle Strahlzeit eingespart werden. Dies hat das Potenzial, die Forschung in der Teilchenphysik und anderen verwandten Bereichen erheblich zu beschleunigen und neue Erkenntnisse zu gewinnen.

Zusammenfassend zeigt diese Arbeit, wie der Einsatz von HiL-Systemen in der Entwicklung und Optimierung komplexer elektronischer Systeme, insbesondere in der Teilchenphysik, bahnbrechende Fortschritte ermöglichen kann. Die gewonnenen Erkenntnisse und die entwickelte Methodik tragen dazu bei, die Effizienz und Präzision von Teilchenbeschleunigerexperimenten zu steigern und eröffnen vielversprechende Perspektiven für zukünftige Entwicklungen in diesem Bereich.

9.1 Ausblick

Bei zukünftigen Forschungen an dem HiL-System müssen zur besseren Beurteilung der Implementierung von Quadropolschwingungen weitere Experimente mit den an der GSI verwendeten

Systemen durchgeführt und die dort erhaltenen Ergebnisse mit echten Messdaten aus dem Teilchenbeschleuniger verglichen werden. Außerdem ist es erforderlich ein Konzept zum Stabilisieren des *Multi Bunch*-Szenarios zu entwickeln. Des Weiteren muss versucht werden, die Berechnungen im System deutlich weiter zu beschleunigen, damit potentiell hunderte bis tausende Teilchen gleichzeitig simuliert werden können, um physikalische Phänomene im Teilchenbeschleuniger noch genauer darstellen zu können. Eine erste Erweiterung hierbei wäre, das CGRA um weitere *Sensor-/Actuator*-Module zu erweitern, damit der Zugriff auf dieses kein Flaschenhals mehr ist.

Abkürzungsverzeichnis

ADC Analog - Digital Converter

ALU Arithmetic Logic Unit

ASIC Application-Specific Integrated Circuits

AXI Advanced eXtensible Interface

BRAM Block Random-Access Memory

CLB Configurable Logic Blocks

CORDIC Coordinate Rotation Digital Computer

CFG Control Flow Graph

CGRA Coarse Grain Reconfigurable Array

CPU Central Processing Unit

DAC Digital - Analog Converter

DDS Direct Digital Synthesis

DRAM Dynamic Random Access Memory

DSP Digital Signal Processor

FIFO First In First Out

FIR Finite Impulse Response

FloPoCo Floating-Point Cores

FMC FPGA Mezzanine Card

FPGA Field Programmable Gate Array

FU Functional Units

GSI Gesellschaft für Schwerionenforschung

HiL Hardware-in-the-Loop

IC Integrated Circuit

LLRF Low-Level-Radio-Frequenc

LUT Look Up Table

MIG Memory Interface Generator

MMCM Mixed Mode Clock Manager

MSB Most Significant Bit

PE Processing Elements

SCAR Scheduler Application Representation

SRAM Static Random-Access Memory

UART Universal Asynchronous Receiver/Transmitter

USBTMC Universal Serial Bus Test & Measurement Class

VLIW Very Large Instruction Word

Abbildungsverzeichnis

2.1 Schematische Darstellung eines Synchrotrons [10] 8
2.2 Beispielhafte Verläufe für Referenz- und Gapspannung sowie für das Strahlsignal im nicht Gleichgewichtszustand 9
2.3 Struktur eines FPGAs (erstellt in Anlehnung an [9]) 13
2.4 Interconnect-Aufbau (erstellt in Anlehnung an [17]) 14
2.5 Aufbau eines CGRAs (erstellt in Anlehnung an [17]) 15
2.6 Aufbau eines PEs (erstellt in Anlehnung an [17]) 16

3.1 FPGA Blockdiagramm für den stationären Fall 20

4.1 Ablaufdiagramm des Programms im stationären Fall. Sensor-/Aktuatorzugriffe sind grün dargestellt, Schleifenköpfe sind in orange und Berechnungen sind in blau abgebildet 23
4.2 Abbildung verschiedener Orbits im Longitudinal-Phasen-Raum [10] 23

5.1 Verlauf von Spannung und Frequenz eines *Ramp Up*-Szenarios, wie er in einer Simulation der GSI verwendet wurde 26
5.2 Beispielhafte Verläufe für Referenz- und Gapspannung sowie für das Strahlsignal während des *Ramp Up*-Szenarios 27
5.3 FPGA Blockdiagramm für das *Ramp Up*-Szenario. Neue Elemente sind in blau dargestellt 28
5.4 Signalverlauf zum Lesen von Daten aus dem DRAM mit Hilfe des *User Interfaces* (erstellt in Anlehnung an [16]) 31
5.5 Signalverlauf zum Schreiben von Daten in den DRAM mit Hilfe des *User Interfaces* (erstellt in Anlehnung an [16]) 31
5.6 Ablaufdiagramm des Programms im *Ramp Up*-Fall. Sensor-/Aktuatorzugriffe sind grün dargestellt, Schleifenköpfe sind in orange und Berechnungen sind in blau abgebildet 36
5.7 Näherung des Verlauf von Spannung und Frequenz eines *Ramp Up*-Szenarios, wie es in einer Simulation der GSI verwendet wurde 39

6.1 FPGA Blockdiagramm für das *Multi Bunch*-Szenario. Neue Elemente sind in blau, entfernte Elemente in rot dargestellt 41

6.2 Ablaufdiagramm des Programms im *Multi Bunch*-Fall mit n Bunches. Sensor-/Aktuatorzugriffe sind grün dargestellt, Schleifenköpfe sind in orange und Berechnungen sind in blau abgebildet. Der hochgestellt Index zeigt an, auf welches asynchrone Partikel sich der Wert bezieht. 42
6.3 Beispielhafte Darstellung, wie für $n = 4$ der Leseoffset mit dem neuen System ermittelt wird. Für Bunch 0 wird direkt der Nulldurchgang des Referenzpartikels als Offset verwendet. 44

7.1 FPGA Blockdiagramm für das Szenario der Quadrupolschwingung. Neue Elemente sind in blau, entfernte Elemente in rot dargestellt 49

8.1 Blockschaltbild des Versuchaufbaus bei der GSI. Das in dieser Arbeit entwickelte System ist in gelb hervorgehoben. 52
8.2 Messungen der Phasendifferenz zwischen Referenz- und Strahlsignal für den HiL-Simulator (oben) und dem Experiment vom 08.05.2022 der GSI mit dem SIS18, bei dem $^{238}U^{73+}$ Ionen beschleunigt wurden (unten) 53
8.3 Frequenzgang des im DSP-System verwendeten FIR-Filters 54
8.4 Ergebnisse der Simulation des *Ramp Up*-Szenarios 55
8.5 Aufgezeichnete Daten aus dem HiL-Simulator 56
8.6 Verhalten des Quadrupolmodels mit neun *Minibunches* in der Java-Simulation und aufgezeichnet vom HiL-Simulator 58
8.7 Verlauf vonΔt_n der einzelnen *Minibunches* 59
8.8 Die verwendeten Initialisierungswerte entlang eines Kreises. Die Farben korrespondieren zu den Farben in Abbildung 8.9. Mit einem X markierten Punkte gehören dort zu den durchgezogenen Linien, die Dreiecke zu den gepunkteten Linien. . . 60
8.9 Verlauf vonΔt_n der einzelnen *Minibunches* im neuen Experiment 61
8.10 Grafische Darstellung des Ressourcenverbrauchs auf dem FPGA 66

Tabellenverzeichnis

4.1 Übersicht über die verwendeten Trigger. 25

5.1 Reservierte Adressen im *Ramp Up*-Szenario 29
5.2 Signale des MIG-*User Interfaces* [16] 32
5.3 Bedeutung der einzelnen Bits bei der Bestimmung des Sinuswertes aus der Tabelle. Das Flankenbit gibt an, ob die aufsteigende (0) oder die abfallende (1) Flanke benötigt wird. 39

8.1 Ergebnisse der Suche nach der maximal möglichen Anzahl an simulierten *Minibunches*. Dargestellt ist die Anzahl der benötigten Takte für die Hauptschleife. . . 62
8.2 Ergebnisse der Suche nach der minimalen Schedulelänge für ein CGRA mit 4x4 PEs. Dargestellt ist die Anzahl der benötigten Takte für die Hauptschleife. 63
8.3 Ergebnisse der Suche nach der minimalen Schedulelänge für ein CGRA mit 5x5 PEs. Dargestellt ist die Anzahl der benötigten Takte für die Hauptschleife. 64
8.4 Hardwarebedarf des FPGA-Designs in absoluten und relativen Zahlen. 65

Literatur

[1] Inc. Advanced Micro Devices. *AMD Virtex 7 FPGA VC707 Evaluation Kit*. https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html. [Aufgerufen am 16.08.2023].

[2] ARM. *AXI4*. https://developer.arm.com/documentation/ihi0022/e/. [Aufgerufen am 04.10.2023].

[3] P. Baudrenghien. „Low-level RF - Part I: Longitudinal dynamics and beam-based loops in synchrotrons“. In: (2011). CERN Accelerator School: Specialised Course on RF for Accelerators; 8 - 17 Jun 2010, Ebeltoft, Denmark. DOI: 10.5170/CERN-2011-007.341. arXiv: 1201.2597.

[4] Philip J Bryant und Kjell Johnsen. *The Principles of Circular Accelerators and Storage Rings*. Cambridge: Cambridge Univ. Press, 1993. DOI: 10.1017/CBO9780511563959.

[5] Florent de Dinechin und Bogdan Pasca. „Designing Custom Arithmetic Data Paths with FloPoCo“. In: *IEEE Design & Test of Computers* 28.4 (Juli 2011), S. 18–27.

[6] D. A. Edwards und M. J. Syphers. *An Introduction to the Physics of High Energy Accelerators*. WILEY-VCH Verlag GmbH & Co. KGaA, 2004. ISBN: 9783527617272. DOI: https://doi.org/10.1002/9783527617272.

[7] Gerald Hempel und Christian Hochberger. „A resource optimized Processor Core for FPGA based SoCs“. In: *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. 2007, S. 51–58. DOI: 10.1109/DSD.2007.4341449.

[8] „IEEE Standard for Floating-Point Arithmetic“. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), S. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[9] Mohd Nazrin Md Isa und Sohiful Anuar Zainol Murad. „Field Programmable Gate Array (FPGA): From Conventional to Modern Architectures“. In: Jan. 2015, S. 53. ISBN: 978-967-5415-98-2.

[10] Harald Klingbeil, Ulrich Laier und Dieter Lens. *Theoretical Foundations of synchrotron and Storage Ring RF Systems*. Springer Cham, 2015.

[11] S Y Lee. *Accelerator Physics*. World Scientific, 1999. DOI: 10.1142/3977.

[12] Siemens. *Modelsim*. https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/. [Aufgerufen am 24.10.2023].

[13] Siglent. *SDG 1062X*. https://www.siglenteu.com/waveform-generators/sdg1000x-series-functionarbitrary-waveform-generators/. [Aufgerufen am 04.10.2023].

[14] Abaco Systems. *FMC151 FPGA Mezzanine Card*. https://www.abaco.com/products/fmc151-fpga-mezzanine-card. [Aufgerufen am 16.08.2023].

[15] Tektronix. *MSO3034*. https://www.tek.com/en/datasheet/mso3000-dpo3000-mixed-signal-oscilloscopes-datasheet. [Aufgerufen am 04.10.2023].

[16] *User Guide UG586, Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v4.2*. URL: https://docs.xilinx.com/v/u/en-US/ug586_7Series_MIS.

[17] *Vorlesungsfolien von Rechnersysteme 2 (Sommersemester 2022) der TU Darmstadt von Prof. Hochberger*.

[18] Mario Wetzel. *Graph Generation from C-Code*. Masterarbeit von 2020 am Fachgebiet Rechnersysteme der TU Darmstadt.

[19] Ramon Wirsch und Christian Hochberger. „Towards Transparent Dynamic Binary Translation from RISC-V to a CGRA“. In: *Architecture of Computing Systems*. Hrsg. von Christian Hochberger, Lars Bauer und Thilo Pionteck. Cham: Springer International Publishing, 2021, S. 118–132. ISBN: 978-3-030-81682-7.