Jugend Forscht 2024:

Verwendung von Field-Programmable Gate Arrays zur Simulation von Quantenschaltkreisen

Teilnehmer: Jonas Müller (20)

Erarbeitungsort: zu Hause

Fachgebiet: Mathematik/Informatik

Bundesland: Niedersachsen

Inhaltsverzeichnis

1 Fachliche Kurzfassung

In diesem Projekt entwickle ich einen Prozessor, der in der Lage ist, Quantencomputer zu simulieren. Dafür entwerfe ich alle Komponenten des Prozessors mit der Hardwarebeschreibungssprache VHDL und optimiere diese, mit dem Ziel, Quantenalgorithmen energieeffizient und mit geringer Laufzeit zu simulieren. Mit der von mir entworfenen Prozessor-Architektur bin ich in der Lage, den Zustandsvektor eines Quantencomputers mit 14 Qubits zu berechnen und auszugeben. Dafür wird ein von mir erstellter Befehlssatz verwendet, der es mir ermöglicht, H-, T- und CNOT-Operationen, welche ein universelles Gate-Set in Quantencomputern darstellen, auf beliebige Qubits anzuwenden. Meinen Prozessor testete ich sowohl in der Simulation als auch in der Praxis auf dem Basys3™ FPGA Board von Digilent, wobei in beiden Fällen die gewünschten Resultate erzielt wurden. Verglichen habe ich die Ergebnisse mit einer Softwaresimulation in Python mit der Bibliothek Qiskit, welche speziell für das Simulieren von Quantencomputern optimiert ist. Dabei liegt die Laufzeit in der aktuellen Version meiner Architektur zwar noch unter der eines modernen x86-Prozessors (was unter anderem auch auf die Hardwarelimitierungen meines FPGA-Boards zurückzuführen ist), die Energieeffizienz befindet sich allerdings schon auf einem ähnlichen Niveau. Verschiedene Optimierungsansätze und die Skalierbarkeit meiner Architektur stellen vielversprechende Möglichkeiten für eine weitere Verbesserung in der Zukunft dar. Weitere Informationen und der Quellcode können unter https://github.com/Sanotsch2003/Simulation-of-Quantum-Computers-using-FPGAs.git eingesehen werden.

2 Motivation und Fragestellung

Die Entwicklung von Quantencomputern verspricht, die Grenzen herkömmlicher Computer zu überwinden und bestimmte Probleme teilweise exponentiell schneller lösen zu können. Dafür notwendig ist auf der einen Seite die entsprechende Hardware, auf der anderen Seite aber auch die Entwicklung von Algorithmen, die von Quantencomputern ausgeführt werden können. Für die Entwicklung jener Algorithmen wird zum aktuellen Zeitpunkt größtenteils auf Softwaresimulationen zurückgegriffen, da echte Quantencomputer sowohl teuer und fehleranfällig als auch nicht weitläufig verfügbar sind. Obwohl es möglich ist, Quantenschaltkreise auf konventionellen Computern zu simulieren, ist dies aufgrund von beschränkter Rechenleistung auf relativ kleine Schaltkreise begrenzt. FPGAs (Field Programmable Gate Arrays) sind auf Hardwareebene programmierbare, integrierte Schaltkreise, die sich flexibel auf jegliche Art von Problem anpassen lassen und somit eine potenzielle Möglichkeit bieten, Quantenschaltkreise energieeffizient und schnell zu simulieren. Im Folgenden werde ich untersuchen, wie energieeffizient und zeitaufwändig Simulationen von Quantenschaltkreisen mittels FPGAs sind. Dafür werde ich meine eigene optimierte Prozessor-Architektur entwickeln und diese mit herkömmlichen Softwaresimulationen auf x86-Prozessoren vergleichen.

3 Hintergrund und theoretische Grundlagen

3.1 FPGAs

Mein Wissen zu FPGAs und die folgenden Informationen stammen aus den Büchern *Getting Started with FPGAs - Digital Circuit Design, Verilog, and VHDL for Beginners* von Russel Merrick [Mer24] und *VHDL by Example - A Concise Introduction for FPGA Design* von Blaine C. Readler [Rea14].

FPGAs lassen sich auf Logikebene programmieren. Anders als anwendungsspezifische integrierte Schaltkreise (ASICs), sind FPGAs nicht für eine bestimmte Anwendung konzipiert. Stattdessen enthalten sie eine bestimmte Menge von Logikzellen, die dem Anwender frei zur Verfügung stehen. Dadurch sind FPGAs sehr flexibel und lassen sich für eine Vielzahl von Anwendungen benutzen. Durch das flexible Verbinden von Logikzellen (Abbildung 1) kann der Entwickler die innere Struktur des FPGAs nach eigenen Wünschen verändern und anpassen. Diese Logikzellen bestehen aus mindestens einem LUT (Lookup Table), mindestens einem Flip-Flop und einem Multiplexer. In einem LUT kann eine beliebige Wahrheitstabelle gespeichert werden, die das Benutzen von kombinatorischer Logik ermöglicht. Das Flip-Flop ermöglicht es, die durch die LUTs implementierte rein kombinatorische Logik sequenziell auszuführen und zum Beispiel durch ein Taktsignal zu steuern. Dies ermöglicht die Erstellung von komplexen Logikmodulen und ganzen Prozessorarchitekturen. Mithilfe der Sel-Eingabe des Multiplexers kann zwischen rein kombinatorischer Logik und sequenzieller Logik umgeschaltet werden. Neben

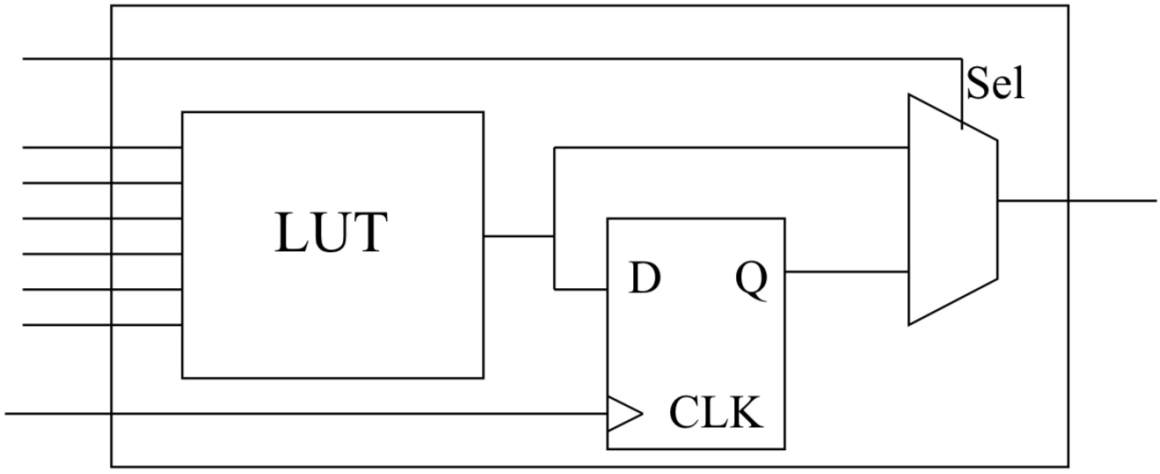

Abbildung 1: LUT basierte Logikzelle [Wan]

den Logikzellen besitzen viele moderne FPGAs Hardwareblöcke, die bestimmte häufig benutze Funktionen beinhalten und dazu beitragen, dass programmierbare Logikzellen eingespart werden können. Dazu zählen zum Beispiel RAM und Multiplikatoren. Um FPGAs zu programmieren, werden sogenannte Hardware-Beschreibungssprachen (HDLs) verwendet. Im Gegensatz zu herkömmlichen Programmiersprachen, wie C oder Python, führen HDLs Anweisungen nicht sequenziell, Zeile für Zeile aus. Stattdessen sind sie darauf ausgelegt, Strukturen so zu beschreiben, dass diese parallel operieren können. Die Entwicklung von Schaltkreisen besteht aus vier Schritten: Der erste Schritt ist das Programmieren mit einer HDL, wobei der gewünschte Schaltkreis mittels Programmcode entworfen und definiert wird. Als

Nächstes folgt die Simulation des Schaltkreises, die zwar nicht dringend notwendig ist, um das Design auf einem FPGA zu implementieren, sich aber trotzdem bei dem Suchen von potenziellen Fehlern als nützlich erweist. Darauf folgt die Synthese, bei welcher der geschriebene Programmcode in eine Repräsentation des Schaltkreises auf Logikebene, also in Logikgatter, Flip-Flops, etc., übersetzt wird. Der letzte Schritt ist die Kompilierung, durch welche das im vorigen Schritt erstellte Design den Ressourcen auf dem FPGA-Chip zugeordnet und eine für diesen lesbare Datei erstellt wird. Diese Datei kann dann auf den Chip hochgeladen werden.

3.2 Computersimulation von Quantencomputern

Die folgenden Informationen eignete ich mir durch das Buch *Introduction to Classical and Quantum Computing* von Thomas G. Wong [Won22] an.

Wie ein klassischer Computer bestehen Quantencomputer aus Bits, sogenannte Qubits (Quantum-Bits). Anders als bei herkömmlichen Computern können Qubits allerdings nicht nur in einem von zwei Zuständen (0 oder 1) sein, sondern auch in einem beliebigen Zustand dazwischen. Ist das Qubit in einem Zustand zwischen 0 und 1, spricht man von einer sogenannten Superposition. Diese Superposition hält allerdings nur solange an, bis der Zustand des Qubits gemessen wird. Durch die Messung bricht die Superposition zusammen und das Qubit ist mit einer Wahrscheinlichkeit P im Zustand 0 und mit einer Wahrscheinlichkeit $1 - P$ im Zustand 1. Eine Messung eines Qubits bringt also immer entweder das Resultat 1 oder 0 und jede darauffolgende weitere Messung des gleichen Qubits bringt dasselbe Resultat. Die Wahrscheinlichkeit, mit der 0 oder 1 gemessen wird, hängt von der Superposition ab. Mathematisch lässt sich der Zustand eines einzelnen Qubits folgendermaßen beschreiben:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

wobei $\alpha, \beta \in \mathbb{C}$ gilt. Die Verwendung von komplexen Zahlen zum Beschreiben der Zustände ist notwendig, da sie es ermöglichen, Phaseninformationen der Zustände darzustellen. Dieser Aspekt ist essenziell in der Quantenmechanik, da Überlagerung zu desktruktiver und konstruktiver Interferenz führen kann. Diese Interferenzmuster werden durch Phasenunterschiede zwischen Zuständen beeinflusst. Durch die komplexe Darstellung kann sowohl die Amplitude als auch die Phase in einer einzigen Zahl kodiert werden. $|\alpha|^2$ und $|\beta|^2$ geben dabei die Wahrscheinlichkeiten an, das Qubit beim Messen im Zustand $|0\rangle$ beziehungsweise $|1\rangle$ zu finden. Eine Superposition, bei der das Messen von 1 und 0 jeweils gleich wahrscheinlich ist, könnte dementsprechend zum Beispiel so aussehen:

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle,$$

Mithilfe dieser Schreibweise lassen sich auch Systeme aus mehreren Qubits beschreiben: Nehmen wir als Beispiel zwei der eben beschriebenen Qubits, die sich jeweils in einer Superposition aus 1 und 0 (mit gleicher Wahrscheinlichkeit) befinden. Das resultierende System kann als Tensorprodukt der einzelnen Systeme beschrieben werden:

$$\begin{aligned}|\psi_{ges}\rangle = |\psi_1\rangle \otimes |\psi_2\rangle &= (\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle) \otimes (\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle) \\ &= \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle.\end{aligned}$$

In diesem Fall gibt es vier verschiedene mögliche Messresultate, die jeweils gleich wahrscheinlich sind.

Wie in klassischen Computern Bits durch Logikgatter beeinflusst werden können, so können Qubits durch sogenannte Quantum-Gates beeinflusst werden. Es gibt sowohl Gates, die auf einzelne Qubits anwendbar sind und diese in eine beliebige Superposition bringen können, als auch solche, die eine Verschränkung mehrerer Qubits bewirken können. Verschränkung ist ein quantenmechanisches Phänomen und führt dazu, dass der Zustand eines Qubits, unabhängig von dessen räumlicher Position, unmittelbar von dem Zustand eines anderen Qubits abhängig ist. Ein solches Gate kann zum Beispiel dazu führen, dass die Wahrscheinlichkeit, das erste Qubit im Zustand 0 oder 1 zu messen, immer noch gleichermaßen wahrscheinlich bleibt, die Messung des zweiten Qubits aber zu 100% der des ersten entspricht. Ein solches System lässt sich mathematisch folgendermaßen beschreiben:

$$|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + 0|01\rangle + 0|10\rangle + \frac{1}{\sqrt{2}}|11\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle.$$

Hier ist es nicht möglich, den Gesamtzustand des Systems in Zustände einzelner Qubits aufzuteilen, weil deren Zustände durch die Verschränkung voneinander abhängig sind.
Durch das gezielte Anwenden von Quantum Gates können Quantenalgorithmen modelliert werden. Das Ziel ist immer, dass sich die Wahrscheinlichkeit für eine bestimmte Messung verstärkt und alle anderen Wahrscheinlichkeiten minimiert werden. Dafür macht man Verschränkung und Interferenz in Quantensystemen zunutze.
Um Simulationen von Quantenschaltkreisen durchzuführen, werden Systeme aus einem oder mehreren Qubits als Spaltenvektoren und Gates als Matrizen dargestellt. So kann die Superposition

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

als Spaltenvektor folgendermaßen dargestellt werden:

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}.$$

Aus dem Anwenden eines Ein-Qubit-Gates U, welche als 2×2 Matrix dargestellt wird, resultiert nach der Multiplikation mit dem Zustandsvektor des betrachteten Qubits folgende Superposition:

$$U|\psi\rangle = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} a \cdot \alpha + b \cdot \beta \\ c \cdot \alpha + d \cdot \beta \end{pmatrix}$$

Wie bereits erwähnt, ergibt sich der Gesamtzustand aus dem Tensorprodukt der Zustände der einzelnen Qubits. Daraus folgt:

$$|\psi_{gesamt}\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes ... \otimes |\psi_n\rangle = \begin{pmatrix} \alpha_1 \\ \beta_1 \end{pmatrix} \otimes \begin{pmatrix} \alpha_2 \\ \beta_2 \end{pmatrix} \otimes ... \otimes \begin{pmatrix} \alpha_n \\ \beta_n \end{pmatrix} = \begin{pmatrix} \alpha_n \\ \beta_n \end{pmatrix}$$

$$= \begin{pmatrix} \alpha_1 \cdot \alpha_2 \cdot ... \cdot \alpha_n \\ \alpha_1 \cdot \beta 2 \cdot ... \cdot \alpha_n \\ \vdots \\ \beta 1 \cdot \beta 2 \cdot ... \cdot \beta n \end{pmatrix}$$

Der resultierende Spaltenvektor, der den Zustand des gesamten Systems beschreibt, hat 2^n Elemente, wobei n die Anzahl der Qubits in dem System ist. Diese Vektorschreibweise macht es einfacher, den aktuellen Zustand des Systems in einem Computer zu speichern, was essenziell bei der Simulation von Quantenschaltkreisen ist. Aufgrund der möglichen Verschränkung von Qubits ist es nicht möglich, nur die einzelnen Zustände der Qubits zu speichern, stattdessen muss immer das gesamte System betrachtet werden. Dies ist der Grund, warum die Simulation von Quantensystemen auf konventionellen Computern so aufwendig ist. Neben dem Speichern des aktuellen Zustands muss eine Simulation allerdings auch in der Lage sein, ein Gate auf ein bestimmtes Qubit in dem System anzuwenden. Soll ein Gate U auf das zweite Qubit des Systems angewendet werden, kann dies mathematisch folgendermaßen beschrieben werden:

$$U_2|\psi_{gesamt}\rangle = I|\psi_1\rangle \otimes U|\psi_2\rangle \otimes ... \otimes I|\psi_n\rangle = I \begin{pmatrix} \alpha_1 \\ \beta_1 \end{pmatrix} \otimes U \begin{pmatrix} \alpha_2 \\ \beta_2 \end{pmatrix} \otimes ... \otimes I \begin{pmatrix} \alpha_n \\ \beta_n \end{pmatrix}$$
$$= \left(\begin{pmatrix} \alpha_1 \\ \beta_1 \end{pmatrix} \otimes \begin{pmatrix} \alpha_2 \\ \beta_2 \end{pmatrix} \otimes ... \otimes \begin{pmatrix} \alpha_n \\ \beta_n \end{pmatrix} \right) \cdot (I \otimes U \otimes ... \otimes I) \,.$$

Dabei ist U eine 2×2 Matrix, die die Transformation des Zustandes eines Qubits durch ein Gate repräsentiert. I ist die 2×2 Einheitsmatrix

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

welche keinen Einfluss auf die Zustände der anderen Qubits hat und nur dafür verwendet wird, eine Transformationsmatrix mit den richtigen Dimensionen zu erhalten. Die Transformationsmatrix, welche die Operation U auf das Qubit an der Position i in einem System aus n Qubits darstellt, sieht wie folgt aus:

$$U_i = \bigotimes_{j=1}^{n} \begin{cases} U & \text{falls } j = i \\ I & \text{sonst} \end{cases}.$$

Das Anwenden von Ein-Qubit-Gates allein ist allerdings nicht ausreichend, um einen Quantencomputer zu simulieren. Wie bereits erwähnt, ist das Prinzip der Verschränkung erst das, was es einem Quantencomputer ermöglicht, bestimmte Probleme exponentiell schneller zu lösen, als es auf klassischen Computern der Fall wäre. Um einen verschränkten Zustand herzustellen, werden Quantum Gates benötigt, die auf zwei Qubits gleichzeitig angewendet werden können. Diese Gates werden mathematisch durch 4×4 Matrizen dargestellt und lassen sich, ähnlich wie Ein-Qubit-Gates, auf den aktuellen Zustandsvektor anwenden. Dabei lässt sich die Transformationsmatrix folgendermaßen berechnen:

$$U_{i,i+1} = \bigotimes_{j=1}^{n} \begin{cases} U & \text{falls } j = i \\ I & \text{falls } j \neq i \text{ und } j \neq i+1 \end{cases}.$$

Wichtig zu beachten ist, dass sich Zwei-Qubit-Gates auf diese Weise nur auf benachbarte Qubits anwenden lassen. Soll ein Gate auf nicht benachbarte Qubits angewendet werden, müssen vorher mehrere Tausch-Operationen durchgeführt werden, welche die Zustände benachbarter Qubits so häufig austauschen, bis die gewünschten Zustände nebeneinander liegen. Erst dann kann die eigentliche Operation angewendet werden, wonach der Tauschvorgang rückgängig gemacht werden muss. Obwohl es in

der Theorie unendlich viele verschiedene Quantum Gates gibt, da es auch unendlich viele mögliche Superpositionen gibt, kann nach dem $Solovay - KitaevTheorem$ [DN05] mit einem relativ kleinen universellen Gate-Set jedes andere Gate mit beliebiger Genauigkeit konstruiert werden. Eines dieser universellen Gate-Sets besteht aus dem H-Gate, dem T-Gate und dem CNOT-Gate [Boy+99]. Die Matrix-Repräsentationen dieser Gates sind die folgenden:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix} \text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Die Tatsache, dass die Anzahl der möglichen Zustände und somit auch der Rechenaufwand exponentiell mit der Anzahl der Qubits ansteigt, macht die Simulation von Quantenschaltkreisen schwierig. Es gibt einige Möglichkeiten, die oben beschriebenen Matrix-Operationen zu vereinfachen und das Berechnen des Zustandsvektors zu beschleunigen. Auf diese Möglichkeiten und das Implementieren eines Prozessors, der die Zustandsvektoren effizient berechnen kann, werde ich mich in den folgenden Kapiteln fokussieren.

4 Materialien und Ausführung des Projekts

4.1 Materialien

Für dieses Projekt benutze ich die Hardware-Beschreibungssprache VHDL in der Entwicklungsumgebung Vivado. Vivado stellt Werkzeuge für das Schreiben des Codes, die Simulation des Designs, die Synthese, wie auch die Kompilierung zur Verfügung. Als FPGA-Board benutze ich das Basys3™ FPGA-Board von Digilent [Dig]. Der verbaute Chip, hergestellt von Xilinx, besitzt 33.280 Logikzellen, 225 Kilobyte BRAM (Block-RAM) und 90 DSP-Slices (Hardwareblöcke, die vor allem für das Verarbeiten von Signalen ausgelegt sind).

4.2 Vorgehensweise

Für die Architektur meines für die Simulation von Quantenschaltkreisen optimierten Prozessors habe ich mir folgende Ziele gesetzt:

- Der Prozessor soll einen Quantencomputer mit möglichst vielen Qubits simulieren können.
- Die Anzahl der nötigen Taktzyklen pro Gate-Operation soll minimiert werden.
- Der Prozessor soll so programmierbar sein, dass beliebige Quantenalgorithmen simuliert werden können, ohne dass eine Änderung der Hardware erforderlich ist.
- Es muss eine Möglichkeit geben, die berechneten Ergebnisse auszugeben.
- Die Architektur soll skalierbar sein, sodass die Leistung auf besserer Hardware gesteigert werden kann.

Bevor ich mit dem Entwurf der einzelnen Komponenten des Prozessors starten konnte, musste ich mir Gedanken über zwei grundlegende Fragen machen:

1. Wie sollen die nötigen komplexen Zahlen in meinem Prozessor dargestellt werden, also was für einen Datentyp sollte ich für die Zahlen nutzen, und mit welcher Genauigkeit sollte ich diese darstellen?

2. Wie genau sollen die Matrix-Multiplikationen durchgeführt werden? Gibt es Algorithmen, die schneller sind als das in 3.2 beschriebene Berechnen einer Tranformationsmatrix aus dem Tensor-Produkt der Gate-Matrix und den Einheitsmatrizen?

Auf die erste Frage konnte ich relativ schnell eine Antwort finden: Da VHDL es zulässt, seinen Code flexibel mit Variablen zu gestalten, brauchte ich mich noch nicht mit der Genauigkeit der Zahlen beschäftigen, da diese bei richtiger Implementierung später schnell geändert werden kann. Deshalb entschied ich zu diesem Punkt, Zahlen mit n Bits darzustellen, wobei sich die Anzahl der Bits gleichermaßen auf den reellen und imaginären Anteil der komplexen Zahl aufteilen. Außerdem entschied ich mich für einen Festkommazahl-Datentyp, der aus einem Vorzeichenbit, einem Bit für den ganzzahligen Anteil, und den restlichen Bits für den Dezimalteil besteht. Dies ist möglich, da der Prozessor nur mit Zahlen im Bereich von -1 bis 1 arbeiten muss, was sowohl für den reellen als auch den imaginären Teil der Fall ist. Diese Vereinfachung ermöglicht es, die Nachkommastellen so genau wie möglich darzustellen. Des Weiteren wird das Durchführen von mathematischen Berechnungen im Vergleich zu Fließkommazahlen simplifiziert. Jegliche oben nicht aufgelisteten wichtigen Fragen bezüglich der genauen Struktur des Prozessors, des Befehlssatzes, etc. werde ich im Prozess der Implementierung beantworten. Obwohl ein genauer Plan vor dem Implementierungsprozess mit Sicherheit von Vorteil gewesen wäre, startete ich die praktische Arbeit an dem Code an diesem Punkt. Dies hatte den Grund, dass ich mich mitten im Lernprozess von VDHL befand und praktische Übung für dringend notwendig hielt. Somit war es unumgänglich, dass der Entwurfsprozess sehr viel Zeit in Anspruch nahm und viele der anfänglichen Resultate wieder verworfen wurden. Im Folgenden werde ich die wichtigsten Komponenten meines Prozessors vorstellen. Ein Überblick über die essenziellen Module ist in Abbildung 2 gegeben. Der Zustandsvekor-Speicher, der Speichercontroller, sowie die Recheneinheiten arbeiten bei einer Taktrate von 200MHz, die übrigen Module bei 100MHz. Es sei erwähnt, dass es sich bei der Abbildung und auch bei den folgenden Beschreibungen um teils starke Abstraktionen handelt und für die eigentliche Implementierung dieser Elemente ein Verständnis der Funktionsweise und Interaktion der einzelnen Elemente auf Bit-Ebene notwendig war. Sowohl der von mir geschriebene VHDL-Code, welcher meine Architektur beschreibt, als auch der Quellcode zur Interaktion mit der seriellen Schnittstelle kann unter https://github.com/Sanotsch2003/Simulation-of-Quantum-Computers-using-FPGAs.git eingesehen werden.

4.2.1 Speichercontroller und Recheneinheiten

Der Kern meines Prozessors ist die Recheneinheit, die in einem konventionellen Prozessor als Arithmetic-Logic-Unit (ALU) bezeichnet werden würde. Da mein Prozessor allerdings keine Arithmetik im allgemeinen Sinne ausführen muss, sondern sich auf wenige, immer wiederkehrende spezielle Operationen beschränkt, war es notwendig, diese speziellen Operationen so effizient wie möglich auszuführen. Wie

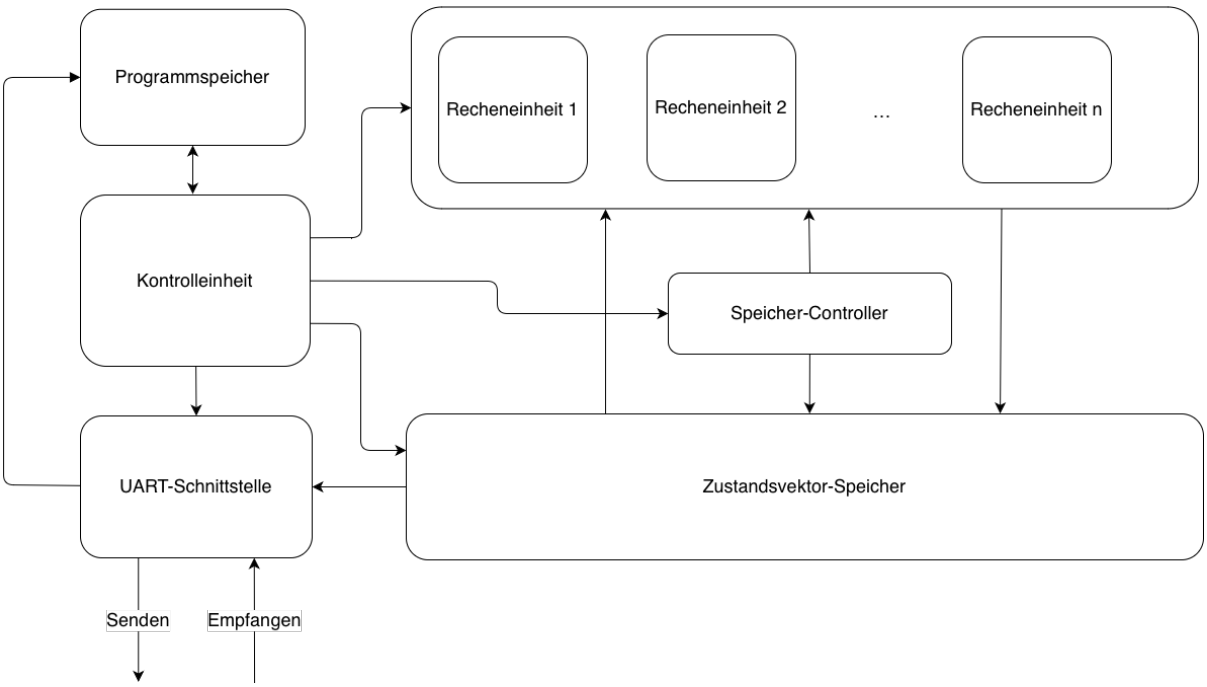

Abbildung 2: Prozessor-Struktur

bereits erwähnt, bestehen diese Operationen aus dem Bilden des Tensor-Produkts von mehreren 2×2, beziehungsweise 4×4 Matrizen. Die entstehende Transformationsmatrix wird dann mit dem aktuellen Zustandsvektor multipliziert. Es liegt nahe, dass sich diese Operationen vereinfachen lassen, da Multiplikation mit Einheitsmatrizen den Großteil der nötigen Berechnungen ausmachen. Deshalb habe ich mich nach einer effizienteren Methode umgeschaut. Ein von Adam Kelly [Kel18] benutzter Ansatz wird in Abbildung 3 dargestellt. Ein Vorteil dieses Algorithmus ist die Tatsache, dass er nicht auf dem Berechnen

Algorithm 1: GATE APPLICATION ALGORITHM (v, G, t)

Input: An n qubit quantum state represented by a column vector $v = (v_1, \ldots v_{2^n})^T$ and a single qubit gate G, represented by a 2×2 matrix, acting on the tth qubit.

1 **for** $i \leftarrow 0$ **to** 2^{n-1} **do**
2 $\quad a \leftarrow$ the ith integer who's tth bit is 0;
3 $\quad b \leftarrow$ the ith integer who's tth bit is 1;
$\quad$ `// The following must be simultaneously updated`
4 $\quad v_a \leftarrow v_a \cdot G_{0,0} + v_b \cdot G_{0,1}$;
5 $\quad v_b \leftarrow v_b \cdot G_{1,1} + v_a \cdot G_{1,0}$;

Abbildung 3: Algorithmus von Adam Kelly [Kel18]

einer Transformationsmatrix basiert, sondern stattdessen in einer Schleife die Werte des Zustandsvektors in Paaren, bestehend aus zwei Elementen, aktualisiert (Abbildung 3, Z. 4–5). Auf der einen Seite eliminiert dies die Notwendigkeit der Zwischenspeicherung einer Transformationsmatrix, welche die quadrierte Menge des Speicherplatzes des Zustandsvektors einnehmen würde, auf der anderen Seite verspricht dieser Ansatz eine relativ simple Parallelisierung von Operationen, da jedes Paar unabhängig von allen anderen Paaren aktualisiert werden kann. So kann die Anzahl der verwendeten Rechenein-

heiten in der Theorie beliebig erhöht werden. Abbildung 4 stellt eine der Recheneinheiten im Detail dar. V_a_in und V_b_in repräsentieren dabei das zu aktualisierende Wertepaar. Der Speichercontroller hat

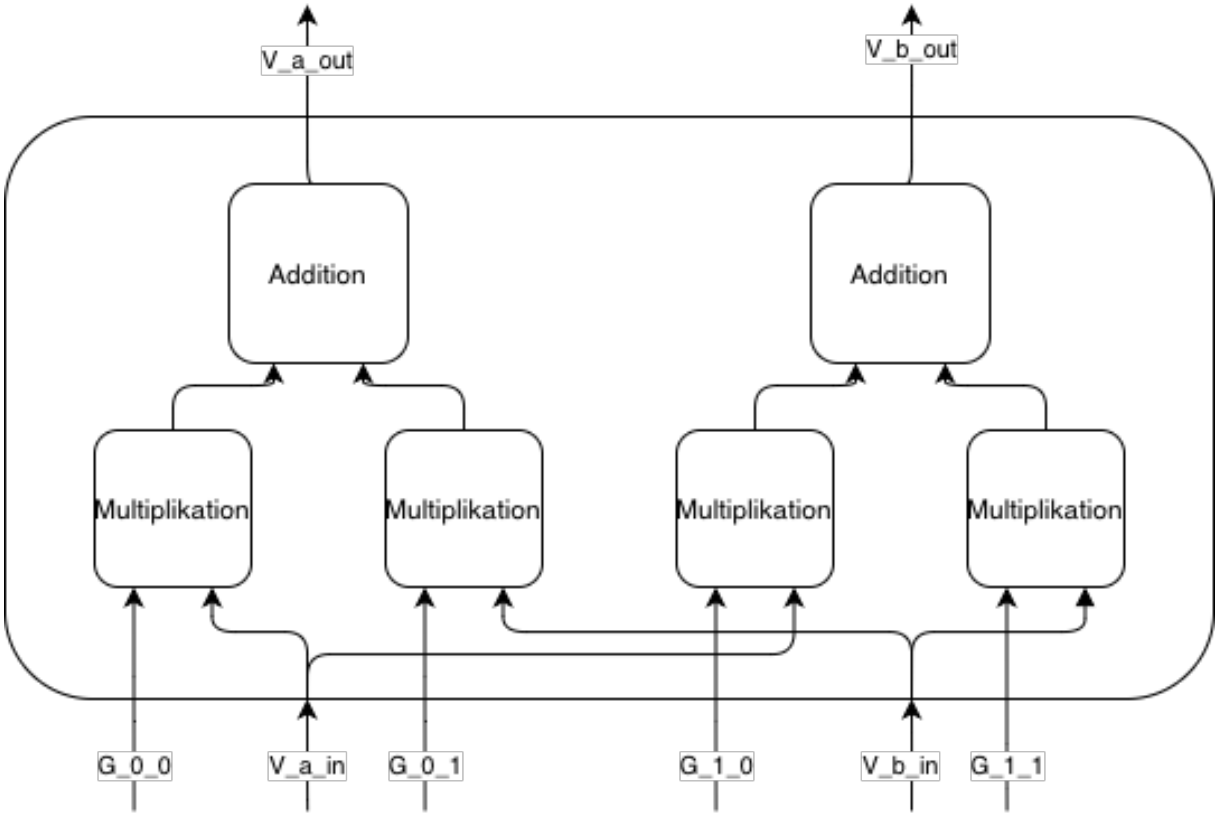

Abbildung 4: Einzelne Recheneinheit

die Aufgabe, die richtigen Werte aus dem Zustandsvektorspeicher zu laden. V_a_out und V_b_out sind die aktualisierten Werte. Diese werden mithilfe des Speichercontrollers nach der Aktualisierung wieder an die richtige Stelle im Zustandsvektorspeicher geschrieben. Die mit G_0_0 bis G_1_1 beschrifteten Eingänge sind die Elemente der Operationsmatrix und können von der Kontrolleinheit beliebig gesetzt werden. Dabei ist jeder Eingang 3 Bit weit, wobei unterschiedliche Bitfolgen von der Recheneinheit folgendermaßen interpretiert werden:

- 000 $\rightarrow 0$
- 001 $\rightarrow 1$
- 010 $\rightarrow -1$
- 011 $\rightarrow \frac{1}{\sqrt{2}}$
- 100 $\rightarrow -\frac{1}{\sqrt{2}}$
- 101 $\rightarrow e^{i\frac{\pi}{4}}$

Diese Werte reichen aus, um alle Matrizen des in Abschnitt 3.2 beschriebenen universellen Gate-Sets darzustellen, wobei das CNOT-Gate auf diese Weise noch nicht abgebildet werden kann, da dieses durch eine 4×4 Matrix dargestellt wird. Für dieses Problem wird in [Kel18] ebenfalls eine Lösung vorgeschlagen:

Das CNOT-Gate (Controlled-Not-Gate) tut nichts anders, als eine durch ein anderes Qubit kontrollierte X-Operation auf das Ziel-Qubit anzuwenden. Wenn wir in dem beschriebenen universellen Gate-Set das CNOT-Gate durch ein X-Gate austauschen, erhalten wir folgendes Set:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix} X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Soll eines dieser Gates als kontrolliertes Gate angewendet werden, werden bestimmte Elemente des Zustandsvektors bei der Aktualisierung übersprungen. Der Speichercontroller kann diese Überprüfung durchführen und gegebenenfalls der Recheneinheit mitteilen, dass die jeweiligen Werte nicht aktualisiert werden sollen. Auf diese Weise kann das universelle Gate-Set auf ausschließlich 2×2 Matrizen reduziert werden. Die genaue Berechnung der zu überspringenden Elemente kann in [Kel18, S.5] nachgelesen werden.

4.2.2 Zustandsvektorspeicher

Um den aktuellen Zustandsvektor speichern zu können, wird Arbeitsspeicher benötigt. Dafür benutze ich den im FPGA-Chip integrierten BRAM. Da das Basys3™ FPGA-Board BRAM mit einer Kapazität von 225 Kilobyte besitzt, können damit bei einer Präzision von 64 Bit $\lfloor \log_2 \left(\frac{225 \cdot 8192}{64}\right) \rfloor = 14$ Qubits simuliert werden. Für die Funktionalität des Speichers sind 64 Bit Ein- und Ausgänge, zwei 14 Bit Adress-Eingänge, ein Taktsignal, sowie Kontrollsignale zum Lesen und Schreiben notwendig. Durch das Anlegen entsprechender Adress- und Kontrollsignale kann vom Speicher gelesen, beziehungsweise zum Speicher geschrieben werden. Dies passiert synchron mit der positiven Flanke des Taktsignals, wobei es möglich ist, gleichzeitig vom Speicher an einer Adresse zu lesen und den Speicher an einer anderen Adresse zu beschreiben. Dies ist insofern essenziell, als eine hohe Speicherbandbreite nötig ist, um die Recheneinheiten auszulasten. Die aktuelle Implementierung des Speichers kann bis zu zwei Recheneinheiten gleichzeitig mit Daten versorgen. Um die Leistung in Zukunft zu steigern, kann ein sogenannter Burst-Modus implementiert werden. Dabei ist nur eine Initialisierung eines Lese- oder Schreibvorgangs notwendig, um daraufhin mehrere aufeinander folgende Elemente aus dem Speicher zu lesen oder mehrere Elemente an aufeinander folgende Adressen im Speicher zu schreiben. Bei richtiger Implementierung kann dies die Speicherbandbreite bei nicht-zufälligen Speicherzugriffen erhöhen und so eine effizientere Auslastung von mehr Recheneinheiten ermöglichen.

4.2.3 Programm-Speicher

Für den Programm-Speicher wird ebenfalls der im Chip integrierte BRAM verwendet. In diesem werden die Instruktionen zum Ausführen des aktuellen Programms gespeichert. Dabei führt die Kontrolleinheit die Instruktionen, beginnend von Adresse 0 und von da an aufsteigend, aus. Ich habe mich dazu entschieden, die Instruktionen auf eine Länge von 8 Bit zu standardisieren, wobei die ersten 4 Bit die Art der Operation bestimmen und die restlichen 4 Bit für bestimmte Parameter dieser Operation benutzt werden können. Der Programm-Speicher kann über eine serielle Schnittstelle direkt beschrieben werden, um das Hochladen und Ausführen verschiedener Simulationen innerhalb kürzester Zeit zu ermöglichen.

4.2.4 Kontrolleinheit

Die Kontrolleinheit steuert alle anderen Komponenten, indem sie die Instruktionen, eine nach der anderen, aus dem Programmspeicher in das Instruktionsregister lädt und anschließend ausführt. Für das Laden der Instruktionen ist ein Zähler notwendig, der die aktuelle Adresse des Programmspeichers beinhaltet und diese nach dem Laden einer Instruktion inkrementiert. Um Schleifen ausführen zu können, kann dieser Zähler durch Instruktionen auch dekrementiert werden. Nach dem Laden einer Instruktion wird diese durch das Setzten und Verarbeiten der Kontrollsignale der einzelnen Komponenten ausgeführt. Damit dies wie vorgesehen funktioniert, muss für jede Instruktion definiert werden, wie die anderen Module von der Kontrolleinheit durch die entsprechenden Kontrollsignale gesteuert werden sollen. Dabei kann das Ausführen einer Instruktion, je nach Komplexität, ein oder mehrere Taktzyklen dauern. Der Maschinencode, der einen Quantenschaltkreis mit 14 Qubits simuliert und eine einfache H-Operation auf das erste Qubit anwendet und anschließend den errechneten Zustandsvektor an einen Empfänger sendet, sieht wie folgt aus:

1. 0001 1110 →setze Schaltkreisgröße auf 14.
2. 0010 0000 →setze Ziel-Qubit auf 0.
3. 0011 0001 →setze Ziel-Matrix auf H-Matrix.
4. 0101 0000 →Führe die nötigen Berechnungen zum Anwenden der Ziel-Matrix-Operation aus.
5. 1000 0000 →setze Adresse des Zustandsvektorspeichers auf i (i ist 0 zu Beginn).
6. 0111 0000 →sende Element aus Zustandsvektorspeicher via UART.
7. 1001 0000 →erhöhe i um 1 und gehe zwei Schritte im Programm zurück, falls noch nicht alle Elemente gesendet wurden.
8. 1011 0000 →halte Programm an.

Zusätzlich zu den oben beschriebenen Instruktionen gibt es einige weitere Instruktionen, die es zum Beispiel ermöglichen, neben dem Ziel-Qubit ein weiteres Kontroll-Qubit zu setzen, um CNOT-Operationen durchführen zu können.

4.2.5 UART-Schnittstelle

Über eine serielle Schnittstelle kann mit dem FPGA kommuniziert werden. Dies ist zum einen notwendig, da Instruktionen in den Programmspeicher geladen werden können, die Simulationsergebnisse aber auch ausgegeben werden müssen. Dafür implementierte ich eine UART-Schnittstelle (Universal-Asynchronous-Receiver-Transmitter-Schnittstelle), über welche Daten mit einem anderen UART-fähigen Gerät ausgetauscht werden können. Dafür müssen Sender und Empfänger aufeinander abgestimmt sein, sodass die Frequenz, mit der Daten gesendet werden, genauso groß ist wie die Frequenz, mit der die Daten empfangen werden. Diese Frequenz wird als Baud-Rate bezeichnet. Der Transmitter sendet 10-Bit-Pakete über eine serielle Schnittstelle, wobei das erste und letzte Bit den Anfang und das Ende

des Pakets markieren und die restlichen 8 Bit die gesendeten Daten darstellen. Um eine einzige komplexe Zahl mit einer Größe von 64 Bit vom FPGA zu zu einem Computer senden zu können, werden die Daten auf 12 Datenpakete aufgeteilt, wobei das erste und letzte dieser Pakete Kontrollpakete sind, die den Anfang und das Ende einer Zahl markieren. Die anderen 10 Pakete enthalten die zu versendende komplexe Zahl. Dafür sind 10 Pakete notwendig, da eines der Datenbits pro Paket markiert, ob es sich um ein Datenpaket oder ein Kontrollpaket handelt. Für die Interaktion mit meiner Architektur über meinen Computer schrieb ich außerdem ein Programm in Python. Dieses ermöglicht es, seriell gesendete Daten zu empfangen, diese in komplexe Fließkommazahlen umzuwandeln und in der Konsole auszugeben.

5 Ergebnisse

Nachdem ich die einzelnen Module in VHDL implementiert und durch Simulation auf ihre Funktionsweise überprüft hatte, brachte ich die einzelnen Komponenten, wie in Abbildung 2 gezeigt, zusammen. Es folgte eine weitere Simulationsphase, die durch langwieriges Fehlersuchen geprägt war. Nachdem alle Fehler behoben waren, versuchte ich den Code zu synthetisieren und auf mein FPGA-Board hochzuladen. Es stellte sich heraus, dass, nur weil eine Architektur in der Simulation funktioniert, dies noch lange nicht in der Praxis der Fall sein sollte. Die Simulation geht von einer idealen Umgebung aus, wobei diese in der Praxis nicht gegeben ist: So verzeiht die Simulation zum Beispiel in einigen Fällen das falsche Initialisieren von Datentypen, wobei dies in der tatsächlichen Architektur zu einer Fehlfunktion führt. Außerdem haben alle Signale in der Realität eine gewisse Verzögerung, die ebenfalls von einer einfachen Simulation nicht beachtet wird. Nach der Anpassung vieler Parameter und dem Testen von unzähligen Abwandlungen desselben Codes, war ich schließlich in der Lage, meinen Prozessor auf dem FPGA-Chip laufen zu lassen. Als Nächstes entschied ich mich dazu, meine Architektur in der Praxis auf Funktionalität, Rechengeschwindigkeit und Energieeffizienz zu testen. Als Referenz verwendete ich die Python-Bibliothek Qiskit [IBM], welche die Möglichkeit einer softwareseitigen Zustandsvektorsimulation bietet. Dabei benutzt Qiskit zum Durchführen der Berechnungen die Programmiersprache Rust und teilt die Berechnungen auf den gesamten Prozessor auf, wodurch maximale Rechenleistung erreicht werden soll. Mit einem von mir in Python geschriebenen Programm lassen sich Quantenschaltkreise durch abstrahierte Funktionen schnell erstellen. Diese können dann entweder mit Qiskit oder einem von mir geschriebenen Simulator simuliert werden. Obwohl ich diesen Simulator nicht auf Effizienz optimiert habe, kann er bei der Fehlersuche in meiner Hardwarearchitektur hilfreich sein, da diese Simulation denselben Algorithmus benutzt und sich so Zwischenergebnisse gut vergleichen lassen. Alternativ können die erstellten Schaltkreise auch direkt in Maschinencode kompiliert werden, sodass sie über die serielle Schnittstelle direkt auf den FPGA hochgeladen und von diesem ausgeführt werden können.

5.1 Funktionalität

Für die Funktionalitätstests ließ ich zufällig generierte Quantenschaltkreise sowohl auf meiner Hardwarearchitektur, als auch in der Qiskit-Simulation laufen und verglich die Ergebnisse. Dabei stimmen diese in allen Fällen miteinander überein, wobei die Genauigkeit der Ergebnisse leicht voneinander abweicht. Dies ist der Benutzung verschiedener Datentypen geschuldet. Während Qiskit für die Simulation kom-

plexe Fließkommazahlen mit doppelter Genauigkeit verwendet (128 Bit), verwendet meine Simulation selbst definierte komplexe Fixed-Point-Zahlen mit 64 Bit. Dies hat zur Folge, dass Qiskit mit einer Genauigkeit von 15 bis 17 Dezimalstellen rechnet, wobei ich in meiner Hardwaresimulation nur Zahlen mit einer Genauigkeit von etwa 9 Nachkommastellen darstellen kann. Um auf eine ähnliche Genauigkeit zu kommen, könnten Fixed-Point-Zahlen mit mehr als 64 Bit verwendet werden. Dabei wären weniger als 128 Bit notwendig, um auf eine ähnliche Genauigkeit zu kommen, da Fixed-Point-Zahlen im Gegensatz zu Fließkommazahlen keine Bits für den Exponenten benötigen. Zwar können dadurch mit Fließkommazahlen deutlich kleinere Zahlen dargestellt werden als mit Fixed-Point-Zahlen, diese besitzen allerdings keine höhere Genauigkeit. Aufgrund der kleineren Datengröße und der schnelleren Verarbeitung von Fixed-Point-Zahlen werde ich diese auch in den nächsten Versionen meiner Architektur verwenden.

5.2 Rechenleistung

Um die Simulationsgeschwindigkeiten vergleichen zu können, implementierte ich in meiner Architektur einen Timer, der durch weitere Instruktionen an einem Punkt des Programms gestartet und auch wieder gestoppt werden kann. Der Timer zählt die für das Ausführen eines Programmabschnitts nötigen Taktzyklen und gibt die Zahl auf dem 7-Segment-Display meines FPGA-Boards aus. Daraus lässt sich die Laufzeit des Programms errechnen. Um die Rechenleistung vergleichen zu können, ließ ich erneut zufällig erzeugte Algorithmen sowohl auf meiner Architektur als auch in der Qiskit-Simulation laufen. Aus den Laufzeiten berechnete ich die für das Durchführen einer Operation benötigte Zeit, wobei ich eine Operation als das Aktualisieren eines Elements des Zustandsvektors definiere. Die errechneten Werte werden in Abbildung 5 dargestellt. FPGA Version 3.0 ist meine erste voll funktionsfähige Architektur, wobei ich in den folgenden Versionen einige Verbesserungen implementierte. Dazu zählen das Optimieren von Mikroinstruktionen, das Erhöhen der Taktfrequenzen und die Möglichkeit, mehrere der spezialisierten Mathematik-Einheiten parallel zu nutzen, um die Gesamtrechenleistung zu erhöhen (dies war in den ersten Versionen nicht möglich). Zwar macht diese Verbesserung meine Architektur in der Theorie noch weiter skalierbar, aufgrund der Hardwarelimitierungen meines FPGA-Boards bin ich trotzdem noch nicht in der Lage, die in Abbildung 5 genannten x86-Prozessoren in der Rechenleistung zu übertreffen.

5.3 Energieeffizienz

Um die Energieeffizienz der unterschiedlichen Simulationen zu bestimmen, wird die jeweils notwendige Zeit pro Operation mit der Leistungsaufnahme der Hardware multipliziert. Dazu verwendete ich den in der von mir benutzten Entwicklungsumgebung Vivado angezeigten Stromverbrauch meiner Hardwarearchitektur, beziehungsweise die Differenz zwischen der Idle- und der Lastleistung der jeweiligen Prozessoren während der Qiskit-Simulation, und multiplizierte diese mit den Werten aus Abbildung 5. Die resultierenden Daten, welche den Energieverbrauch pro Operation angeben, werden in Abbildung 6 dargestellt. Dabei fällt auf, dass sich die Energieeffizienz insgesamt seit der ersten Version deutlich verbessert hat, wobei Version 3.1 aus der Reihe fällt. Dies ist der Tatsache geschuldet, dass ich zuvor direkt auf das 100MHz Basistaktsignal des Chips zugegriffen habe. Ab Version 3.1 wird auf ein weiteres Modul zur Takterzeugung zurückgegriffen, was höhere Taktfrequenzen ermöglicht. Der Stromverbrauch dieses Moduls führt anfänglich zu einer leichten Verschlechterung der Energieeffizienz, wobei sich diese

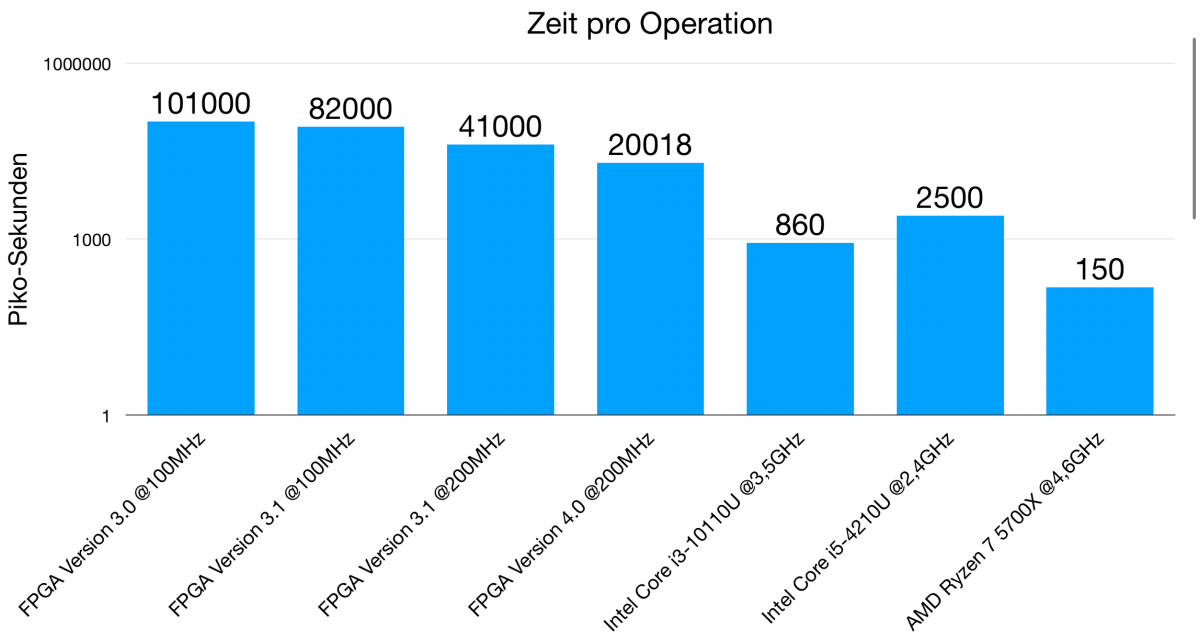

Abbildung 5: Simulationszeit

aber durch die nun ermöglichte Verdopplung der Taktfrequenz auf 200MHZ langfristig verbessert. Weitere Erhöhungen der Taktfrequenz sind prinzipiell möglich, können aber zu Timing-Fehlern innerhalb der Hardware führen, wenn interne Signale nicht innerhalb der gegebenen Zeit von ihrem Anfangs- zum Endpunkt propagieren können.

6 Ergebnisdiskussion

Meine Ergebnisse haben gezeigt, dass es möglich ist, Quantenschaltkreise mit FPGAs zu simulieren. In Bezug auf die Simulationsgeschwindigkeit konnte ich allerdings noch keine Verbesserungen gegenüber x86-Prozessoren verzeichnen, wobei sich die Energieeffizienz meiner Architektur schon jetzt auf einem vergleichbaren, beziehungsweise in vielen Fällen besseren Niveau befindet. Seit meiner ersten funktionierenden Architektur konnte ich die Simulationszeit schrittweise auf 20% der ursprünglichen Zeit verringern. Es ist wahrscheinlich, dass damit noch nicht die maximal mögliche Rechenleistung mit dieser Hardware ausgeschöpft ist und fortschreitende Weiterentwicklung zu einer weiteren Verringerung der Simulationszeit führen könnte. Darüber hinaus können mehr Hardwareressourcen aufgrund der Skalierbarkeit meiner Architektur ebenfalls zu einer erhöhten Rechenleistung beitragen. Mit einer größeren Speicherkapazität wäre außerdem die Simulation von mehr als 14 Qubits möglich. Das Senden des errechneten Zustandsvektors über die serielle Schnittstelle ist vergleichsweise langsam und nimmt deutlich mehr Zeit in Anspruch als das eigentliche Durchführen der Berechnungen, wodurch der aktuelle Prototyp nicht alltagstauglich ist. Insofern bin ich durch die mir aktuell zur Verfügung stehende Hardware sowohl in Bezug auf den Speicher und Hardwareressourcen, als auch in Bezug auf schnelle Schnittstellen limitiert.

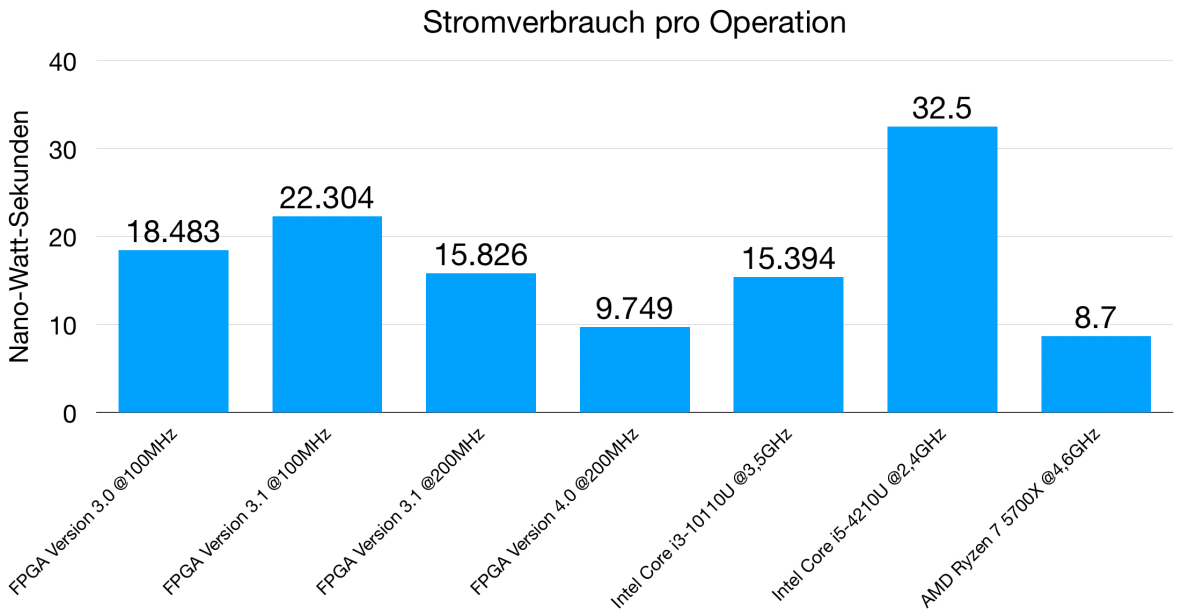

Abbildung 6: Energieeffizienz

7 Fazit und Ausblick

Dieses Projekt half mir, ein tieferes Verständnis für die Funktionsweise von sowohl konventionellen als auch Quantencomputern zu gewinnen. Das Lernen der Hardwarebeschreibungssprache VHDL trug dabei stark dazu bei und wird mir bei einer Fortsetzung des Projektes erlauben, meine Ideen deutlich schneller in die Tat umsetzen zu können, als es bis jetzt der Fall war. Viele Stunden verbrachte ich damit, Fehler in meinem Code zu suchen, die mir jetzt banal erscheinen. In Zukunft werde ich meine Architektur weiter optimieren und Lösungen für die im vorherigen Abschnitt beschriebenen Limitierungen suchen. Für eine schnellere Kommunikation mit der Hardware können entsprechende FPGA-Boards über die PCIe-Schnittstelle mit anderer Hardware kommunizieren. Dies würde auf der einen Seite die Limitierungen der aktuell verwendeten UART-Schnittstelle eliminieren, könnte auf der anderen Seite aber auch zur Lösung des Speicherproblems beitragen, da über die PCIe-Schnittstelle direkt auf den System-Arbeitsspeicher zugegriffen werden kann. Die effiziente Verwaltung von Speicher wird entscheidend sein, um das volle Potenzial der FPGA-Hardware ausnutzen zu können. Inwiefern der System-Arbeitsspeicher tatsächlich in Verbindung mit der PCIe-Schnittstelle die nötigen Datentransferraten liefern kann, muss untersucht werden. Alternativ liegt die Entwicklung von maßgefertigten Platinen auf der Hand, die ausreichend schnellen Speicher in physischer Nähe zum FPGA-Chip implementieren.

Literatur

[Boy+99] P. Oscar Boykin u. a. *On Universal and Fault-Tolerant Quantum Computing*. 1999. arXiv: quant-ph/9906054 [quant-ph].

[DN05] Christopher M. Dawson und Michael A. Nielsen. *The Solovay-Kitaev algorithm*. 2005. arXiv: quant-ph/0505030 [quant-ph].

[Dig] Digilent. URL: https://www.farnell.com/datasheets/1884379.pdf.

[IBM] IBM. *IBM Quantum Computing | Qiskit*. URL: https://www.ibm.com/quantum/qiskit.

[Kel18] Adam Kelly. *Simulating Quantum Computers Using OpenCL*. 2018. arXiv: 1805.00988 [quant-ph]. URL: https://arxiv.org/pdf/1805.00988.pdf.

[Mer24] Russell Merrick. *Getting started with fpgas digital circuit design, Verilog, and VHDL for Beginners*. No Starch Press, 2024.

[Rea14] Blaine C. Readler. *VHDL by example: A concise introduction for fpga design*. Full Arc Press, 2014.

[Wan] Haibo Wang. *FPGA Logic Cells and Architecture*. URL: https://www.engr.siu.edu/haibo/ece428/notes/ece428_logcell.pdf.

[Won22] Thomas Giechaung Wong. *Introduction to classical and quantum computing*. Rooted Grove, 2022.