IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Masterarbeit

Power Optimization of Register File Accesses in a Hearing Aid Processor Using Genetic Optimization Algorithms

Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Mikroelektronische Systeme
Fachgebiet Architekturen und Systeme

erstellt von

René, Weinmann
Matrikelnummer: 389570
geb. 14.08.1989 in Stuttgart, Deutschland

Hannover, 17. August 2017

Statement of originality

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments. This applies also to all graphics, drawings and images included in the thesis.

_____

Hannover, Do 17 August, 2017

Rene Weinmann

Acknowledgment

Auch gibt es niemanden, der den Schmerz an sich liebt, sucht oder wünscht, nur, weil er Schmerz ist, es sei denn, es kommt zu zufälligen Umständen, in denen Mühen und Schmerz ihm große Freude bereiten können. Um ein triviales Beispiel zu nehmen, wer von uns unterzieht sich je anstrengender körperlicher Betätigung, außer um Vorteile daraus zu ziehen? Aber wer hat irgend ein Recht, einen Menschen zu tadeln, der die Entscheidung trifft, eine Freude zu genießen, die keine unangenehmen Folgen hat, oder einen, der Schmerz vermeidet, welcher keine daraus resultierende Freude nach sich zieht? Auch gibt es niemanden, der den Schmerz an sich liebt, sucht oder wünscht, nur, weil er Schmerz ist, es sei denn, es kommt zu zufälligen Umständen, in denen Mühen und Schmerz ihm große Freude bereiten können. Um ein triviales Beispiel zu nehmen, wer von uns unterzieht sich je anstrengender körperlicher Betätigung, außer um Vorteile daraus zu ziehen? Aber wer hat irgend ein Recht, einen Menschen zu tadeln, der die Entscheidung trifft, eine Freude zu genießen, die keine unangenehmen Folgen hat, oder einen, der Schmerz vermeidet, welcher keine daraus resultierende Freude nach sich zieht?

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

List of Algorithms

Kapitel 1

Einleitung

In den letzten Jahren steigt die Komplexität von Schaltungen kontinuierlich und scheint dabei dem Gesetz von Gordon Moor zu folgen, welches im Jahre 1965 veröffentlicht wurde. Jedoch hat diese Entwicklung auch eine Kehrseite, denn mit steigender Performance steigt auch gleichzeitig der Energiebedarf. Dies hat zur folge, dass nun der limitierende Faktor bei portablen Geräten bei der Stromaufnahme liegt und nicht mehr bei der Performance. Außerdem sind die Materialien für immer größer werdenden Batterien sehr selten und dementsprechend teuer. Aus diesem Grund ist die Verlustleistungsoptimierung von portablen Geräten ein immer wichtigeres Themengebiet und muss deshalb bestmöglich optimiert werden. Bei mobilen Geräten handelt es sich nicht nur um Smartphones oder Smartwatches sondern auch um medizinische Geräte wie in Fall dieser Arbeit um ein Hörgerät. Durch optimierte Verlustleistung ist es Nutzern möglich längere Zeit besser zu hören, welches die Lebensqualität deutlich erhöht. In dieser Arbeit soll die Verlustleistung eines solchen Hörgeräteprozessor minimiert werden, umso eine höhere Laufzeit zu ermöglichen. Hierbei soll die explizit die Verlustleistung von Registerspeicherzugriffen optimiert werden. Es gibt zwei Methoden die Leistungsaufnahme zu optimieren, zum Einen kann Veränderung der Hardware vorgenommen werden, oder zum Andern kann eine Verbesserung durch Software herbei geführt werden. Eine Optimierung der Hardware ist sehr zeitaufwändig und demzufolge teuer. Außerdem ist es bei vielen Geräten nicht möglich eine weitere Verbesserung durch Anpassen der Hardware

sicherzustellen. Aus diesem Grund wird auf eine Optimierung der Software gesetzt. Dies kann insbesondere bei gepipelineten Prozessoren zur Kompilierzeit geschehen. Diese Arbeit zeigt wie die Verlustleistung eines solchen Hörgeräteprozessor mithilfe eines genetischen Algorithmus zur Registerallokation optimiert werden kann.

1.1. Motivation

BLA BLA

1.2. State of the Art

BLA BLA

!Write something

1.3. Aufbau der Arbeit

For a better reading experience of this thesis, let us provide a short overview of the following chapters.

Chapter 2: Fundamentals !Write something

Chapter 3: Architecture !Write something

Chapter 4: Evaluation !Write something

Chapter 5: Conclusion !Write something

Kapitel 2

Grundlagen

Bla bla

2.1. SIMD Prozessor

2.1.1. VLIW

VLIW ist eine Eigenschaft von Mikroprozessor-Architekuren. VLIW steht hierbei fuer Very Long Instruction Word und bedeutet soviel wie sehr langes Instruktionsword. Das Ziel dieser Struktur ist eine schnelle Abarbeitung des Befehlsatzes, wobei hierbei einige Befehle parallel ausgefuehrt werden. Um dies zu ermoeglichen sind mehrere Instruktions-Dekoder vonnoeten. Der verwendete TUKUTURI-Prozessor besitzt zwei solcher Dekoder auch Issue-Slots genannt und kann somit zwei Befehle parallel ausfuehren. Eine VLIW-Architekur geht meist mit Pipelining einher.

2.1.2. Pipelining

Wie das Wort Pipelining schon besagt, handet es sich hierbei um eine Befehltabarbeitung am Fliessband(Pipeline). Wurde ein Befehl in Phase 1 abgearbeitet kann dieser an die naechste Phase weitergeleitet werden und der nachfolgende Befehl fuehrt

die Phase 1 aus. Somit ist eine parallele verarbeitung mehrerer Befehle moeglich. Die Phasen in einem VLIW Prozessor sind: 1. Fetch 2.Decode 3.Load 4.Execute 5.Write.

2.2. Aufbau der Architektur

2.2.1. MIPS Prozessor

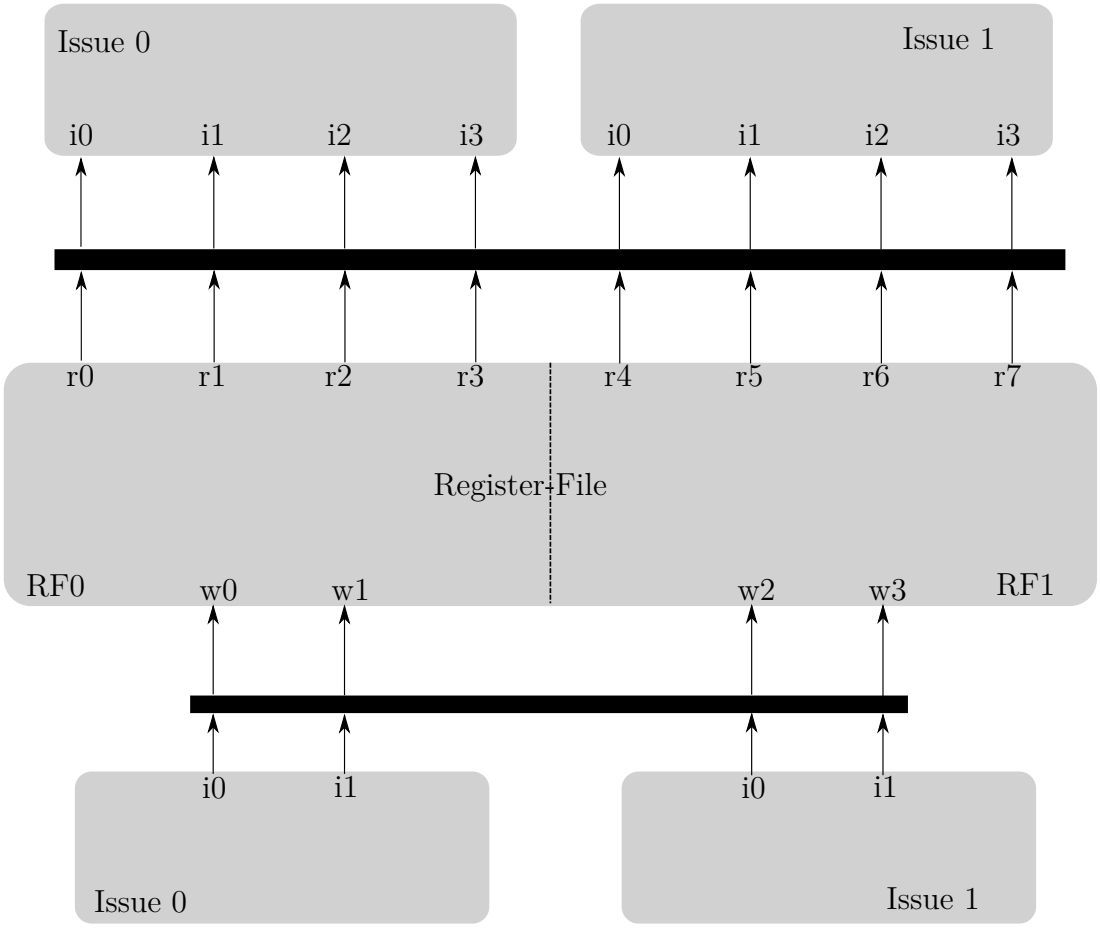
2.2.2. Register File Organisation

In dieser Arbeit wird eine Architektur mit 64Byte Register verwendet. Dabei handelt es sich nicht um ein monolithisches Register-File sondern om ein Multishared Register-File. Hierbei wird das Register in zwei oder mehr Teile getrennt, die Anzahl haengt hierbei von den Issue Solts ab. Durch diese Aufteilung koennen Einsparungen in der Logik fuer die Schreib- und Leseports generiert werden. Durch die Trennung ist es jedoch immernoch moeglich, dass Instuktionen aus dem Issue-Slot 1 in das Register-File 0 schreiben oder lesen koennen. Die Aufteilung und die Zuordnung der Schreib- und Leseports koennen aus der Abbildung XXX und Tabelle TTT entnommen werden.

In this thesis an architecture with an 64 byte register file (RF) is used. The two instruction decoders for issue 0 and issue 1 share this flexible register file (RF). In order to enlarge the bottleneck of the register file ports in an VLIW architecture, the register file is divided into two smaller files.

2.2.3. Compiler

Um von einer Programmiersprache (in diesem Fall Assembler) zu einem von dem Prozessor ausfuehrbaren Programm zu gelangen, ist es noetig die Programmiersprache in Maschinencode zu uebersetzten. Dies ist die Aufgabe des Compilers. Der verwendete Compiler stueckelt den Code auf in so genannte Micro Instructions (MI). Hierbei


Issue 0
Issue 1
i0
i1
i2
i3
i0
i1
i2
i3
r0
r1
r2
r3
r4
r5
r6
r7
Register-File
RF0
RF1
w0
w1
w2
w3
i0
i1
i0
i1
Issue 0
Issue 1

Schreib-Instruktion		Schreib-Ports			
0	1	0	1	2	3
0	0	0	1		
0	1	0		1	
1	0	1		0	
1	1			0	1

Tabelle 2.1.: Schreib-Port

Lese-Inst.				Lese-Ports							
0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	1	2	3				
0	0	0	1	0	1	2		3			
0	0	1	0	0	1	3		2			
0	0	1	1	0	1			2	3		
0	1	0	0	0	2	3		1			
0	1	0	1	0	2			1	3		
0	1	1	0	0	3			1	2		
0	1	1	1	0				1	2	3	
1	0	0	0	1	2	3		0			
1	0	0	1	1	2			0	3		
1	0	1	0	1	3			0	2		
1	0	1	1	1				0	2	3	
1	1	0	0	2	3			0	1		
1	1	0	1	2				0	1	3	
1	1	1	0	3				0	1	2	
1	1	1	1					0	1	2	3

Tabelle 2.2.: Lese-Port

ist eine MI eine Anweisung welche der Prozessor in einem Taktzyklus ausfuehren kann, dabei kann es sich auch um mehrere Micro Operation (MO) handeln. Im Falle der verwendeten MOAI-Architektur handelt es sich um zwei MOs die parallel ausgefuehrt werden koennen. Diese MOs werden wiederum gruppiert in sogenannte Straight Line Microcode (SLM). Ein SLM ist hierbei so definiert, dass es nur eine Einsprungstelle und eine Austrittsstelle gibt. Ausserdem duerfen sich keine Spruenge und Verzweigungen in einer SLM befinden.

2.2.4. Scheduling

Das Scheduling ist zustaendig fuer die Anordnung der MIs bzw. der MOs die sich in einem SLM befinden. Hierbei geht der Algorithmus so vor, dass er die Anordnung sucht die den geringesten Kritischenpfad aufweist. Hierbei gibt es verschiedene Ansatzmoeglichkeiten. Die einfachste Methode ist das List-Scheduling. Hierbei wird bei jedem einfuegen eines MOs in ein MI ueberprueft, ob die zugehoerigen Register allokiert werden koennen, ist dies nicht der Fall wird mit der naechsten MO begonnen. Im darauffolgenden Schritt wird nun nochmals versucht die MO einzufuegen. Dies

wird solange wiederholt bis alle MOs einer MI zugeortnet sind. Diese Art von ALgorithmus findet jedoch nicht immer eine optimale Loesung und ist gerade fuer grosse Programme nicht geeignet. Aus diesem Grund wurde zusaetzlich ein genetischer Algroithmus eingesetzt der die laenge der SLM reduziert.

2.2.5. Hamming-Distanze

Die Hamming-Distanz ist nach dem amerikanischen Mathematiker Richard Wesley Hamming benannt und gibt ein Mass fuer die Unterschiedlichkeit zweier Zeichenketten an. Hierbei ist die Hamming-Distanz die Anzahl der unterschiedlichen Stellen der beiden Codewoerter.

$$00110 \text{ und } 00100 -> \text{Hamming-Distanze} = 1 \tag{2.1}$$

2.3. Register Allokation

2.3.1. Virtuelle Register

Bei virtuellen Registern handelt es sich um Register die an beliebiger Stelle im Register-File allokiert werden können, das heisst der Compiler kann selbst entscheiden wo im Registerfile er diese Variable platziert. Die Idee hierbei ist es, dem Compiler die Aufgabe zu uebergeben ein geeignetes Register zu suchen. Das Codebeispiel 2.1 zeigt anhand einer einfachen Addition diese Funktion. Da der Prozessor nicht mit Registeradressen addieren kann, muessen zwei Hilfsvariabeln verwendet werden. In diesem Fall wurden die Regsiter V0R0 und V0R1 gewaehlt. Mithilfe dieser Register kann der Prozessor nun eine Addition in Zeile drei durchfuehren.

```
1 MV V0R0 0x100
2 MV V0R1 0x101
3 ADD V0R0 V0R0 V0R1
4 STORE 0x100 V0R0
```

Codebeispiel 2.1: physikalische Register

Um nun den Code etwas flexibler zu gestallten, wird nun dem Compiler ueberlassen welches Register er benutzt. Die selbe Addition ist in 2.2 mit virtuellen Registern realisiert. Hierbei wird dem Compiler durch ein x gekennzeichnet, dass es sich um ein virtuelles Register handelt. Dieser waehlt anschliessend ein optimales Register aus, so dass sich der Programmierer um diese Aufgabe nich bemuehen muss. Dies hat den Vorteile, dass somit fuer den Code geeignete/optimale Register ausgewaehlt werden koennen. Hierbei wird darauf geachtet, dass beide Register-Files gleich ausgelastete sind und dass es moeglich beibt X2- oder MAC-Befehle (siehe Kapitel 2.3.2 ff.) allokiert werden koennen. Ausserdem sind die Register in diesem Fall so allokiert, dass die Verlustleistungsaufnahme minimal ist. Wie die Register fuer eine optimale Verlustleistung ausgewaehlt werden muessen wird in dieser Arbeit evaluiert und aufgezeigt.

```
1 MV VxR0 0x100
2 MV VxR1 0x101
3 ADD VxR0 VxR0 VxR1
4 STORE 0x100 VxR0
```

Codebeispiel 2.2: virtuelle Register

2.3.2. X2 Betriebsmodus

Der X2-Betriebsmodus ermoeglicht es mehrere Funktionseinheiten zu nutzen ohne dabei die Anzahl der zu decodierenden Anweisungen zu erhoehen. Hierbei wird einem Befehl die doppelte Anzahl an Target und Source Registern uebergeben, somit steigt die Zahl der Parameter von drei auf sechs. Beide Instruktionen eines X2-Befehl koenen auf das selben Schreiberegister zugreifen. Bei den Lese Registern gibt es die Vorgabe, dass die geraden Instruktionen auf gerade Register zugreifen und die ungeraden Instruktionen auf ungerade.

2.3.3. MAC Betriebsmodus

2.4. Verlustleistung

Unter Verlustleistung in integrierten Schaltungen versteht man die in den Transistoren umgesetzte Leistung die in Form von Wärme verloren geht. Hierbei wird in statische P_{stat} und dynamische Verlustleistung P_{dyn} unterschieden.

2.4.1. Dynamische Verlustleistung

Jedes mal wenn eine Kapazität geladen oder entladen werden muss, entsteht eine dynamische VerlustleistungP_C. Die zweiter dynamische VerlustleistungP_{SC} entsteht beim Schaltevorgang von Transistoren insbesondere von Inverter-Schaltungen. In diesem Fall existiert beim Umschalten eine kurze Zeitspanne in der beide Transistoren durchgeschaltet sind. In diesem Fall besteht ein Kurzschlussstrom I_{SC}zwischen Versorgungsspannung V_{DD} und Masse V_{SS}. Die dynamische Verlustleistung ist proportional zur Schaltaktivitätα und somit auch zur Taktfrequenz f.

$$P_{dyn} = \alpha C_L V_{dd}^2 f \tag{2.2}$$

2.4.2. Statische Verlustleistung

Sobald die Verlustleistung unabhängig von der Taktrate wird, kann diese als statische P_{Stat} bezeichnet werden. Dies ist der Fall, wenn die Transistoren so konzeptioniert sind, dass ein konstanter Strom zwischenV_{DD} undV_{SS}besteht. Dieser Strom ist untabhängig von der angelegten Gatespannung. Der dadurch auftretende Srom wird Leakagestrom genannt. Mit immer kleiner werdenden Strukturen wird dieser Strom deutliche wichtiger. In diesem Fall hängt der Strom von der Gatespannung ab und die statische wird eine dynamische Verlustleistung.

Die gesamte Verlustleistung ist die Summer der drei erwähnten Verlusten.

$$\begin{aligned} P &= P_C + P_{SC} + P_{Stat} \\ P &= P_{dyn} + P_{Stat} \end{aligned} \tag{2.3}$$

2.5. Genetische Algorithmen

Genetische Algorithmen wurden ursprünglich entwickelt um evolutionäre Prozesse aus der Natur nachzuempfinden. Erstmals wurde diese Art von Algorithmen von John Holland 1975 entwickelt und untersucht. In der Natur müssen sich Lebewesen ständig an ihren Lebensraum anpassen und mit den Problemen der Natur leben. Um dies zu ermöglichen haben sich Lebewesen über Jahrtausende an ihre Umgebungen angepasst. Der Aufbau, die Fähigkeiten und das Erscheinungsbild eines Lebewesen ist von Geburt an vorgegeben, diese Information befindet sich in den Chromosomen verschlüsselt. Eine Evolution ist hierbei die Weitergabe dieser Informationen. Durch das decodieren der Chromosomen entsteht eine neue Lebensform. Natürliche Selektion ist hierbei die Anpassungsfähigkeit des Lebewesens an den vorgegebenen Lebensraum. Demzufolge überleben bzw. pflanzen sich nur die Generationen fort, welche sich gut an die Umstände der Umgebung angepasst haben. Die Mechanismen hinter der Evolution sind noch nicht komplett entschlüsselt, jedoch sind einige Verfahren bekannt welche im weiteren betrachtet werden. Durch das Vorbild der Natur sollen mit genetischen Algorithmen schwierige Sachverhalte lösen können. Hierbei stellen die Chromosomen eine Abbildung einer Lösung auf ein Problem dar. Durch eine sogenannte Fitness-Funktion, kann ermittelt werden wie gut sich ein Chromosom an das gegebene Problem angepasst hat. Wie auch in der Natur müssen werden einzelne Chromosomen fortgepflanzt und bilden neue Generationen. Betrachtet man eine gewisse Anzahl an Generationen so spricht man von einer Population. Zu Beginn des Algorithmus startet man mit zufälligen Chromosomen, bis eine bestimmte Anzahl Generationen entstanden ist. Mit dieser Population kann nun die Fortpflanzung betrieben werden. Durch die Fortpflanzung werden die Chromosomen zweier Lebewesen an eine neue Generation weitergegeben, auch dieser Prozess ist bis

heute nicht genau entschlüsselt jedoch können einige Merkmale abgebildet werden. Darunter fällt das Crossover und die Mutation.

2.5.1. Crossover

Bei dem Crossover-Prozess, handelt es sich um die Fortpflanzung von Generationen. Hierbei ist ausschlaggebend welche Generationen miteinander gepaart werden. Auf den ersten Blick scheint es einleuchtend immer die Generation mit der besten Fitness zu paaren. Dies ist jedoch nicht immer sinnvoll, da so schnell ein lokales Minimum erreicht wird. Aus diesem Grund gibt es verschiedene Ansätze um geeignete Eltern für die neue Generation zu finden. Die am weitesten verbreitete Methode ist die Roulette Wheel Selection. Hierbei wird zufällig ein Elternpaar gewählt, wobei die Chance einer jeden Generation ausgewählt zu werden proportional zur Fitness ist. Dieses Verfahren trägt ihren Namen daher, dass es einem Roulette-Rad gleicht, wobei das Rad in Stücke unterteilt ist, welche die Größe proportional zur Fitness haben. Die Auswahl kann nun einem Drehen an einem Rad gleichgesetzt werden. Hierbei ist es wahrscheinlicher, dass die Generationen mit höher Fitness ausgewählt werden. Es ist jedoch auch möglich, dass Populationsmitglieder mit niedriger Fitness gepaart werden.

2.5.2. Mutation

Auch der Prozess der Mutation stammt aus der Natur. Hierbei besteht eine geringe Chance, dass Chromosomen verändert werden und Eigenschaften aufweisen die in keinem der Elternteile aufzufinden sind. Hierzu werden in dem durch Crossover erzeugtem Chromosom einzelne Gene durch Zufall verändert. Die Wahrscheinlichkeit einer zufälligen Veränderung ist hierbei wie in der Natur sehr gering.

2.5.3. Fitness

Die Fitness einer Generation gibt an wie gut sich das Chromosom an das gegebene Problem angepasst hat. Diese Bewertung ist sehr wichtig für die richtige Funktion des Algorithmus. Dabei muss die Fitness-Funktion so entwickelt werden, dass die Generationen unterscheidbar sind und eine Einordnung in der Population möglich ist.

Kapitel 3

Implementierung

Nach genauerem betrachten der Formel für die dynamische Verlustleistung 2.2 hängt diese von vier Faktoren ab, Schaltaktivitäten, Lastkapazität, Versorgungsspannung und der Frequenz. Die statische Verlustleistung wird nicht weiter betrachtet, da diese nur durch eine Veränderung der Hardware verbessert werden könnte. Dies ist jedoch nicht Teil dieser Arbeit. Da ebenfalls die Spannung und die Frequenz von der Architektur und anderen nicht beeinflussbaren Faktoren vorgegeben wurde kann auch an diesen Parametern nichts geändert werden. Im folgenden wird nun erst darauf eingegangen wie, durch Minimierung der Schaltaktivitäten, die Verlustleistung optimiert werden kann. Im Weiteren Verlauf des Textes wird dann ebenfalls der Einfluss der Lastkapazitäten überprüft.
Um in ein Register zu schreiben, muss eine geeignete Adresse an den Adressbus angelegt werden. Da es sich bei der Architektur um 32-bit-Register handelt muss hierfür eine 5-bit Adresse angelegt werden. Dieser Adressbus wird nun auf Schaltaktivität optimiert. Schreibt beispielsweise eine Anweisung an die Adresse Null und die nachfolgende Instruktion an die Adresse 31, so wäre dies der Worst-Case, da in diesem Fall alle 5-Bit umgeladen werden müssten. Befinden sich im Code jedoch virtuelle Register so können diese an beliebiger Stelle zugewiesen und somit die Adresse frei wählbar machen. Dadurch kann die Schaltaktivitäten der Leitungen verringert werden. Aus diesem Grund wurde zu Beginn eine Heuristik entwickelt bei der die Hammingdistanz der Adressleitungen minimiert wird.

3.1. Heuristik

Um die Hammingdistanz zu berechnen werden die Adressen der letzten Anweisung gespeichert. Da die Architektur zwei Issue-Slots aufweist und diese auf beide Register-Files zugreifen können muss es dementsprechend vier Target-Adressleitungen geben(siehe Abbildung XXX). Wollen beide Instruktionen an das selbe Register-File schreiben, so werden entweder die alle zehn Adressleitungen für die Adressierung verwendet. Verwenden beiden Instruktionen unterschiedliche Register werden jeweils die unteren fünf Adressleitungen des jeweiligen Registers verwendet. Die Adressen bleiben solange in an der Adressleitung angelegt, bis ein neuer Befehl diese Leitungen benutzen muss. Die Distanz wird nun immer mit den zuletzt verwendeten Adressen bestimmt.

3.2. Genetischer Algorithmus

Die Gene des implementierten genetischen Algorithmuses wurden so gewaehlt, dass diese die zu allokierenden Register repraesentieren. Die Anzahl der Gene haengt hierbei jeweils von der Menge der virtuellen Registern ab. Jedes Gen enspricht einem Mapping von virtuellen zu reelen Registern. Die Populationsgroesse wurde aus der Literatur [LLL] entnommen. Im Kapitel XXX wird der Einfluss der Population auf die Ergenisse genauer untersucht.

3.2.1. Fitness-Funktion

Da die Registeralokation dem Scheduling untergeordnet ist und von diesem aufgerufen wird, muss die Fitnessfunktion an die des Scheduling angepasst werden. Aus diesem Grund wurde zum einfacheren Vorgehen zwei Verschiedene Fitness-Funktionen implementiert. Zum einen der Wert der an die uebergeornete Funktion weitergegeben wird und zum Andern eine interene Fitness um die Ergebnisse zu evaluieren. Der Uebergabewert besteht lediglich aus der Anzahl der Register die nicht im Register-

File allokiert werden konnten. Bei der internen Fitness-Funktion wurden mehrere Ansaetze evaluiert.

- Hamming-Distanz
  Als erste Variante wurde die Summe der Hamming-Distanzen als Fitness-Funktion verwendet. Hierzu wurden die Hammingdistanzen der Target- sowie Source-Register ermittelt. Wie im Falle der Heueristik mussten die verschiedenen Issue-Slots berueksichtigt werden. Da jeden Adress-Dekoder zwei Source-Adressen aufweist, gibt es in diesem Fall vier Moeglichkeiten der Allokation.
- gewichtete Hamming-Distanz
  blabla
- Lastkapazitaet
  blabla
- Hamming-Distanz und Lastkapazitaet
  blabla
- Adresssumme
  blabla

3.2.2. Startpopulation

Die Startpopulation besteht im Normalfall aus zufaellig gewaehlten Genen. Hierbei werden nur Register verwendet die nicht bereits von reelen Registern blockiert sind. Um eine schnellere Konvergenz zu einem globalen Minimum zu geweehrleisten wird ein Gene der Startpopulation durch ein Heuristik-Algorithmus bestimmt. Dadurch wird bereits an einem lokalen Minimum gestartet und der genetische Algorithmus muss dies nicht selbst finden. Nachteilig ist hierbei jedoch das durch ungeeignete Parameter der Algorithmus bei dem bereis sehr starken lokalen Minimum stagniert und keine Verbesserung mehr finden kann. Aus diesem Grund sollte beim Start mit der Heueristik eine hoehere Mutationswahrscheinlichkeit eingesetzt werden um dieses Minimu dennoch zu ueberwinden.

3.2.3. dynamische Anpassung

3.2.4. Algorithmus Modi

-M -> Allocation Mode

Algorithmus-Konfig.	Modi -E	-M	-R	-D	-H	-U	-o
Heuristik alt	0	1	0	X	X	X	1
Heuristik neu	1	1	0	X	X	X	1
Genetischer Algo. Hamming-Fitness	1	1	1	0	0	0	3
	1	1	1	0	1	0	3
Heuristik als Startpop.	1	1	1	0	1	0	3
dynamischer Genetischer Algo.	1	1	1	1	0	0	3
Genetischer Algo. Load-Fitness	1	2	1	0	0	0	3
Genetischer Algo. Hamming+Load-Fitness	1	3	1	0	0	0	3

Tabelle 3.1.: Algorithmus-Konfiguration

Kapitel 4

Evaluation

Implemented techniques/designs/hardware/software need some sort of validation regarding propriety and performance. The way chosen to establish the correct functionality and/or performance have to be described and justified regarding their suitability.

4.1. Evaluation

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim.

Kapitel 5

Conclusion

This chapter contains concluding remarks whether the desired results could be met or why this wasn't the case. Furthermore future optimizations can be documented here.

5.1. Evaluation

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim.

Anhang A

Source Code

Codebeispiel A.1: Combinational part of the checker for state transition inspection

```
1  ...
2   -- combinational transition checking
3   process (next_state, curr_state, rst_n)
4   begin  -- process
5
6     transition_error <= '0';
7
8     case next_state is
9
10      when INIT =>
11        null;
12
13      when READY =>
14        if curr_state /= INIT
15          and curr_state /= READY
16          and curr_state /= IFG then
17          transition_error <= '1';
18        end if;
19
20      -- we check the next state against the current state
21      when PREAMBLE =>
22        if curr_state /= READY
23          and curr_state /= PREAMBLE then
24          transition_error <= '1';
25        end if;
26
27      when SFD =>
28        if curr_state /= PREAMBLE then
29          transition_error <= '1';
30        end if;
31
32      when DEST =>
33        if curr_state /= SFD
34          and curr_state /= DEST then
35          transition_error <= '1';
36        end if;
37
38      when SRC =>
39        if curr_state /= DEST
40          and curr_state /= SRC then
```

```
41            transition_error <= '1';
42          end if;
43
44        when PAYLOAD =>
45          if curr_state /= SRC
46            and curr_state /= PAYLOAD then
47            transition_error <= '1';
48          end if;
49
50        when PAD =>
51          if curr_state /= PAYLOAD
52            and curr_state /= PAD
53            and curr_state /= FLUSH then
54            transition_error <= '1';
55          end if;
56
57        -- flush state can always be accessed in case of tx-error
58        when FLUSH =>
59          null;
60
61        when CRC =>
62          if curr_state /= PAYLOAD
63            and curr_state /= PAD
64            and curr_state /= FLUSH
65            and curr_state /= CRC then
66            transition_error <= '1';
67          end if;
68
69        -- ifg state can always be accessed in case of tx-error
70        when IFG =>
71          null;
72
73        when others => transition_error <= '1';
74      end case;
75
76    end process;
77  ...
```