Bachelorarbeit

Assembler Code-Generierung aus synchronen Aktionen

Marcel Heer

2. Januar 2014

Technische Universität Kaiserslautern
Fachbereich Informatik

Prüfer: Prof. Dr. Klaus Schneider
M. Sc. Maximilian Senftleben

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Assembler Code-Generierung aus synchronen Aktionen“ selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 2. Januar 2014

_____
Marcel Heer

Abstract

This thesis describes an implementation of a code generator for the Abacus processor architecture. From the intermediate format AIF, which was compiled from a synchronous programming language, program code is generated for a sequential processor of the Abacus architecture. The intermediate format is translated into a control flow graph and it is described how sequential code can be generated from synchronous programs efficiently. Register allocation is implemented by graph coloring.

Zusammenfassung

Diese Arbeit beschreibt die Implementierung eines Codegenerators für die Abacus- Prozessorarchitektur. Aus einem Zwischenformat AIF, welches aus einer synchronen Programmiersprache erzeugt wurde, wird Programmcode für einen sequentiellen Prozessor der Abacus-Architektur generiert. Dazu wird das Zwischenformat in einen Kontrollflussgraphen übersetzt und es wird beschrieben, wie aus synchronen Programmen effizient sequentieller Code erzeugt werden kann. Die Registerzuteilung geschieht durch Graphfärbung.

Inhaltsverzeichnis

1 Einleitung

Die meisten Computersysteme, die uns im Alltag begegnen, sind eingebettete Systeme. Allerdings werden interaktive Systeme, wie es sie z.B. an Geldautomaten gibt, eher wahrgenommen, da sie eine Schnittstelle zwischen Mensch und Maschine haben. Eingebettete Systeme werden hingegen kaum wahrgenommen, da es keine direkte Interaktion mit dem Menschen gibt. Bei einem überwiegenden Teil dieser eingebetteten Systeme handelt es sich um reaktive Systeme. Ein reaktives System muss auf Ereignisse seiner Umwelt zeitkritisch reagieren und das passende Antwortverhalten generieren. Das Verhalten des Systems ist daher abhängig von seiner Umwelt. Daher grenzen sich reaktive Systeme von transformationellen Systemen ab, die eine Transformation ohne Interaktionen durchführen. Außerdem unterscheiden sich reaktive Systeme von interaktiven Systemen, die nur zu vom System bestimmten Zeitpunkten und Bedingungen mit ihrer Umwelt interagieren.

Ein eingebettetes System kann aus Subsystemen bestehen, die wiederum aus mehreren Prozessoren bestehen können. Traditionelle Programmiersprachen für sequentielle Prozessoren haben keine Auffassung von Zeit und Nebenläufigkeit, die bei nebenläufigen Echtzeitsystemen von Bedeutung sind [Sch09].
Auf einem sequentiellen Prozessor wird eine Folge von Befehlen der Reihe nach abgearbeitet. Unterprogramme werden gestartet und nach ihrer Ausführung liegen ihre Ergebnisse vor. Dies geschieht nicht nebenläufig. Daher sind diese Programmiersprachen für die Konstruktion paralleler eingebetteter Systeme nur bedingt geeignet. Imperative synchrone Programmiersprachen hingegen besitzen viele Eigenschaften, die von eingebetteten Systemen an eine Sprache gefordert werden. Das Voranschreiten der Zeit wird durch Anweisungen im Programm explizit dargestellt [BS11].

Dennoch enthalten eingebettete Systeme aus Kostengründen häufig einen sequentiellen Prozessor. Um synchrone Programme auf einem solchen Prozessor auszuführen, ist es notwendig, die Nebenläufigkeit aus der synchronen Quellsprache in sequentieller Software zu simulieren oder für die Ausführung auf einem sequentiellen Prozessor zu übersetzen [Edw03].

Zielplattformen wechseln häufig. Daher ist es erstrebenswert, synchrone Programme in Zwischenformate zu übersetzen. Für die synchrone Programmiersprache Quartz gibt es einen Compiler, der Programme in das Zwischenformat AIF übersetzt [BS11].

Diese Arbeit beschäftigt sich mit der Codegenerierung aus dem Zwischenformat AIF für sequentielle Prozessoren. Es wird gezeigt, wie aus diesem Zwischenformat sequentieller Maschinencode generiert werden kann. Das geschieht am Beispiel der Abacus-Prozessorarchitektur [ES14].

2 Verwandte Arbeiten

2.1 Ansätze zur Übersetzung synchroner Programmiersprachen

Eingebettete Systeme enthalten häufig einen traditionellen Prozessor, der sequentiellen Programmcode ausführen kann. Die Spezifikation eingebetteter Anwendungen erfolgt aber häufig durch nebenläufige Sprachen. Edwards [Edw03] beschreibt Techniken zur Übersetzung von nebenläufigen Spezifikationen in sequentiellen Programmcode. Die nebenläufige Beschreibung eines Systems ist natürlich, da ein eingebettetes System auf mehrere Prozesse antwortet oder diese kontrolliert. Eingebettete Systeme werden jedoch häufig mit einem einzelnen sequentiellen Prozessor implementiert, z.B. um Kosten zu senken.
Die Compiler, die Edwards beschreibt, bestehen aus zwei Übersetzungsphasen. In der ersten Phase wird aus einer Quellsprache wie z.B. Esterel ein einfacheres nebenläufiges Zwischenformat erzeugt. Aus diesem wird in der zweiten Phase schließlich das sequentielle Zielprogramm generiert.
Als effiziente Compiler beschreibt Edwards solche, die sich am CFG[1] des Quellprogramms orientieren, da ein CFG passend für das Verhalten eines sequentiellen Prozessors ist. Der Vorteil bei der Verwendung eines CFG besteht darin, dass nicht durchgehend alle Teile des Gesamtsystems ausgeführt werden [Edw03].
Bai [BBS11] beschreibt die Übersetzung eines Zwischenformats, das aus der synchronen Sprache Quartz übersetzt wurde, in einen erweiterten endlichen Zustandsautomaten. Dieser Zustandsautomat kann für die Erzeugung von effizientem sequentiellen Code verwendet werden [BBS11].

2.2 Ansätze zur Registerzuteilung

In vielen Veröffentlichungen wird das Graphfärben zur Optimierung des Problems der Registerzuteilung beschreiben. Chaitin [Cha81] beschreibt, wie die Registerzuteilung mit Hilfe des Graphfärbens implementiert werden kann. Smith [SRH04] beschreibt einen Algorithmus, der das Graphfärben verwendet und dabei die Registerzuteilung für Systeme mit unterschiedlichen Registerklassen allgemein durchführt [SRH04].

Daveau [DTLS04] beschreibt ein Framework zur Registerzuteilung. Dieses Framework ist ein Compiler, welcher das Retargeting[2] umsetzt. Dieser Ansatz wird verfolgt, da in eingebetteten Systemen häufig unterschiedliche und beschränkte Registersätze vorkommen [DTLS04].

[1] CFG: Control Flow Graph, Kontrollflussgraph
[2] Retargeting: Code soll für mehrere Plattformen generiert werden

2.3 Vergleich mit verwandten Arbeiten

Die Implementierung anhand des globalen Schedulings in dieser Arbeit betrachtet die Datenabhängigkeiten zwischen synchronen bedingten Aktionen. Dies wird von Edwards [Edw03] als eine Möglichkeit beschrieben, die den Nachteil hat, dass der erzeugte Code auf der Zielarchitektur langsam läuft. Das liegt daran, dass jeder Teil des Programms durchlaufen wird [Edw03].
Für die bedingten Aktionen bedeutet dies, dass auch solche Bedingungen ausgewertet werden, die aus Sicht des Kontrollflusses nicht relevant sind.
Die von Edwards [Edw03] beschriebenen Ansätze zur Übersetzung der Sprache Esterel auf einen sequentiellen Prozessor können auch für andere synchrone Sprachen verwendet werden. Daher wird in dieser Arbeit auch ein Ansatz verfolgt, bei dem ein Kontrollflussgraph verwendet wird, welcher auf der Implementierung des erweiterten endlichen Zustandsautomaten von Bai [BBS11] basiert, um Code zu erzeugen, der auf dem Zielsystem effizienter läuft.
In dieser Arbeit wird das Graphfärben zur Registerzuteilung ähnlich zu der von Chaitin [Cha81] beschriebenen Implementierung umgesetzt, allerdings mit weniger Registern, was an der verwendeten Prozessorarchitektur liegt.

3 Grundlagen

3.1 Aufbau eines Compilers

Ein Compiler hat zwei Aufgaben zu bewältigen: die Analyse[1] der Eingabe, eines Zeichenstroms, auf der einen Seite, und die Synthese[2] auf der anderen Seite, bei welcher das Zielprogramm konstruiert wird [Aho08, S.6].

Einem Compiler wird ein Programm übergeben, jedoch keine Eingabefolge. Es wird „[...] unabhängig von irgendwelchen Eingabedaten analysiert und in eine andere Form überführt, welche die effizientere Ausführung mit beliebigen Eingabefolgen erlaubt."[WM97, S.3]

In der Analysephase wird das Programm zerlegt und eine grammatische Struktur aufgebaut, aus der eine Zwischendarstellung des ursprünglichen Quellprogramms erstellt wird. Bei „[...] syntaktisch oder semantisch nicht wohlgeformten [...]"[Aho08, S.7] Quellprogrammen muss der Nutzer benachrichtigt werden. Aus dem Quellprogramm werden eine Symboltabelle und eine Zwischendarstellung erzeugt, welche dann im nächsten Schritt zur Synthese weiterverwendet werden [Aho08, S.6-7].

In der Synthesephase wird aus der Zwischendarstellung das Zielprogramm konstruiert, dazu werden die Informationen der Symboltabelle verwendet [Aho08, S.7].

Der Kompilierungsvorgang kann in einzelne Phasen zerlegt werden, „die jeweils eine Darstellung des Quellprogramms in eine andere umwandeln." Die Symboltabelle wird von all diesen Phasen genutzt. Zwischen Front-End und Back-End können in einer separaten Phase außerdem noch maschinenunabhängige Optimierungen durchgeführt werden [Aho08, S.7].

Die Zielsprache des Compilers hängt vom verwendeten Prozessortyp ab. Das Zielprogramm kann nach dem Kompilierungsvorgang auf einer realen oder abstrakten Maschine ausgeführt werden [WM97, S.4].

3.1.1 Analysephase

Lexikalische Analyse In der lexikalischen Analyse (Scanner) wird das Quellprogramm als Zeichenfolge eingelesen und in eine Folge von Symbolen zerlegt, die durch die Programmiersprache festgelegt werden. Typische Symbole sind z.B. Standardbezeichnungen von Typen, Kommentare oder Sonderzeichen. Symbole mit ähnlicher Struktur, wie z.B. die Menge aller Integer-Konstanten, werden dabei in Symbolklassen zusammengefasst. Nach der lexikalischen Analyse durch den Scanner wird ein Sieber eingesetzt, dessen Aufgabe es ist, zu erkennen, welche Symbole in der Programmiersprache von besonderer Bedeutung und welche für die weitere Verarbeitung irrelevant und somit zu eliminieren sind. Der Sieber kann auch einzelne

[1] Analyse: auch Front-End
[2] Synthese: auch Back-End

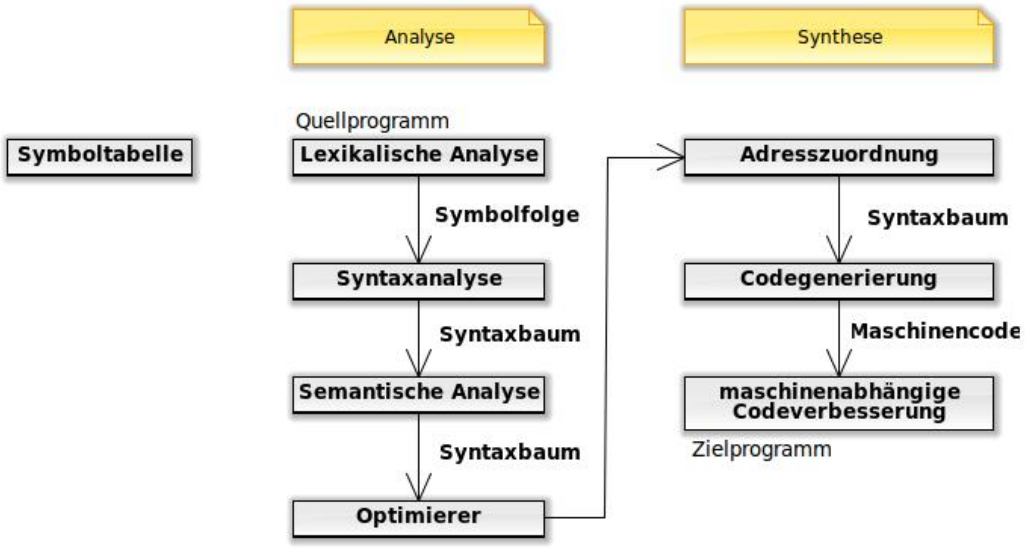

Abbildung 3.1: Aufbau eines Compilers [Aho08]

Symbolklassen eindeutig kodieren, wie z.B. Bezeichner, deren Zeichenkettendarstellung dann nur noch einmal abgespeichert werden muss [WM97, S.227,239].

Syntaktische Analyse In der syntaktischen Analyse, die vom Parser realisiert wird, wird aus der Symbolfolge, die aus der lexikalischen Analyse entstanden ist, die Struktur des Programms abgeleitet. Dazu ist dem Parser der Aufbau der Ausdrücke der Sprache bekannt. Er „[...] muss in der Lage sein, Fehler in der syntaktischen Struktur zu erkennen."[WM97, S.228] Die Ausgabe ist der Syntaxbaum des Programms oder eine äquivalente Form [WM97, S.228,271].

Semantische Analyse In der Semantischen Analyse werden statische semantische Eigenschaften des Programms bestimmt. Dies sind Eigenschaften wie z.B. Typkorrektheit und Typkonsistenz, die unabhängig von konkreten Eingabewerten sind. Sie können nicht durch eine kontextfreie Grammatik beschrieben werden, da Kontextinformationen benötigt werden [WM97, S.228,403].

Maschinencodeunabhängige Optimierung Durch Datenflussanalyse werden unerreichbare Programmteile und nie aufgerufene Funktionen erkannt und eliminiert. Außerdem können effizienzsteigernde Programmtransformationen durchgeführt werden, wie z.B. in Schleifen das Herausziehen von Berechnungen, die von Schleifenvariablen unbeeinflusst sind oder auch das Eliminieren von redundanten Berechnungen [WM97, S.230].

3.1.2 Synthesephase

Adresszuordnung Die Adresszuordnung ist Teil der Synthesephase, es werden Maschinenparameter wie z.B. Wortlänge und Adresslänge benötigt. Die Zuordnung von Speicherein-

heiten zu elementaren Typen wird durch diese Parameter bestimmt. Außerdem besteht die Möglichkeit, Typen mit wenig Speicherplatzbedarf, wie z.B. Boolean, in größeren Einheiten unterzubringen [WM97, S.232].

Codegenerierung In der Codegenerierung werden die Befehle des Zielprogramms erzeugt, zur Adressierung von Variablen wird dabei auf die Adresszuordnung zurückgegriffen. „Allerdings kann die Zeiteffizienz des Zielprogramms oft gesteigert werden, wenn es gelingt, die Werte von Variablen und Ausdrücken in den Registern der Maschine zu halten. Der Zugriff darauf ist i.Allg. schneller als der Zugriff auf Speicherzellen“[WM97, S.232]. Die Registerzuteilung muss also in der Codegenerierung die Register möglichst nutzbringend verwenden, indem beispielsweise häufig benutzte Werte in den Registern gehalten werden.
Werden die Ausdrücke des Quellprogramms in Befehlsfolgen des Zielprogramms übersetzt, müssen bei dieser Codeselektion möglichst gute solcher Befehlsfolgen bezüglich Ausführungszeit, Speicherplatzbedarf und Befehlsfolgenlänge gefunden werden.
Bei Prozessoren mit Pipeline-Architekturen ist außerdem noch die Instruktionsanordnung eine wichtige Teilaufgabe, bei der Befehle so angeordnet werden, dass sie Pipeline-Hazards[3] vermeiden [WM97, S.232,541,569].

3.2 Eingebettete Systeme und synchrone Programmiersprachen

Eingebettete Systeme Eingebettete Systeme werden verstärkt eingesetzt und es gibt den Trend, dass mehrere Prozessoren auf einem Chip integriert werden. So können z.B. MPSoCs[4] beim Design eingebetteter Systeme verwendet werden. Die Konstruktion eingebetteter Systeme ist nicht weit entwickelt, was daran liegt, dass die unterschiedlichen Fachrichtungen, die damit in Zusammenhang stehen, nicht nahtlos zusammenarbeiten. Ein eingebettetes System besteht aus Subsystemen, die wiederum aus mehreren Prozessoren bestehen. Die Kommunikation zwischen diesen eingebetteten Systemen geschieht über Bussysteme wie z.B. einen CAN Bus[5] oder FlexRay. Eingebettete Systeme kommunizieren mit ihrer Umgebung, weshalb es neben interaktiven Systemen auch reaktive Systeme gibt, welche auf Ereignisse aus der Umgebung reagieren. Da Eingebettete Systeme häufig in sicherheitskritischen Bereichen eingesetzt werden, ist die Korrektheit dieser Systeme von essentieller Wichtigkeit. Diese Korrektheit kann durch formale Verifikation z.B. mit Model Checking gezeigt werden. Es ist daher nötig, dass die verwendete Sprache eine formale Semantik besitzt, die eine direkte Übersetzung in Transitionssysteme[6] erlaubt, welche von Verifikationsmethoden verwendet werden [Sch09].

Nebenläufige Berechnung Bei traditionellen sequentiellen Programmiersprachen fehlen Eigenschaften, die für die Konstruktion reaktiver Echtzeitsysteme von Bedeutung sind, wie beispielsweise die Auffassung von Zeit und Nebenläufigkeit. Von den Sprachen mit diesen Eigenschaften erlauben nur wenige eine Hardware- und Softwaresynthese desselben Programms oder eine Übersetzung in Transitionssysteme. Die Programmiersprache Quartz [Sch09] hingegen bietet diese Möglichkeiten [Sch09].

[3] Pipeline-Hazard: Konflikt bei der Abarbeitung von Befehlen in der Pipeline
[4] MPSoC: heterogenous multiprocessor systems on a single chip
[5] CAN: Controller Area Network, standardisiertes Feldbussystem
[6] Transitionssystem/state transition system: besteht aus Zuständen und deren Übergängen

3.3 Die synchrone Sprache Quartz

Synchrone Sprachen Die Sprache Quartz ist eine imperative synchrone Programmiersprache, die aus der Sprache Esterel entstanden ist. Bei synchronen Sprachen wird von perfekter Synchronität ausgegangen. Aus Sicht des Programmierers nehmen Kommunikation und Berechnung keine Zeit in Anspruch. Außerdem werden alle Komponenten parallel berechnet. Nur wenn eine *pause*-Anweisung auftritt, vergeht eine logische Zeiteinheit. An einer *pause*-Anweisung müssen die nebenläufig ausgeführten Aktionen synchronisiert werden. Die Sprache Quartz hat im Vergleich zu Esterel zusätzliche Anweisungen für die asynchron parallele Ausführung von Aktionen und verzögerte Zuweisungen. Für Quartz existiert der Compiler „qrz2aif“, der ein Quartz-Programm in eine Menge bedingter Aktionen[7] übersetzt. Zur besseren Nutzbarkeit für die Verifikation wird zwischen Kontroll- und Datenfluss eines Programms unterschieden. Der Kontrollfluss eines synchronen Programms kann außerdem in einen endlichen Zustandsautomaten übersetzt werden [Sch09].

Makroschritt

Die Menge der Aktionen zwischen zwei *pause*-Anweisungen wird als Makroschritt bezeichnet, die einzelnen Aktionen als Mikroschritte. In einem Makroschritt werden alle Eingaben gelesen und alle Ausgaben parallel berechnet. Mikroschritte werden dynamisch anhand der Datenabhängigkeiten ausgewertet. Eine Umsortierung der Mikroschritte genügt nicht, da semantische Probleme (siehe 3.3) auftreten können. Jede Variable einfachen Datentyps darf innerhalb eines Makroschrittes nur einmal geschrieben werden, bei zusammengesetzten Datentypen dürfen unterschiedliche Komponenten allerdings in unterschiedlichen Mikroschritten geschrieben werden [Sch09].

Verzögerte Zuweisungen Neben direkten Zuweisungen der Form $x = \tau;$ gibt es in Quartz verzögerte Zuweisungen der Form $next(x) = \tau;$. Bei einer verzögerten Zuweisung wird τ mit den Werten des Makroschrittes berechnet, in dem sich die Zuweisung befindet, $next(x)$ weist den Wert jedoch erst im folgenden Makroschritt zu [Sch09].

Threads Synchrone Sprachen wie Quartz sind nebenläufige Sprachen, in denen Ausdrücke für die Ausführung paralleler Threads existieren:

- $S_1||S_2$ (synchron parallele Ausführung von S_1 und S_2): Dabei werden S_1 und S_2 im Gleichschritt ausgeführt, wodurch ein Makroschritt Mikroschritte sowohl aus S_1 als auch aus S_2 enthält.
- $S_1|||S_2$: asynchron parallele Ausführung von S_1 und S_2
- $S_1|S_2$: parallele Ausführung von S_1 und S_2, wobei nur im Kontrollfluss von S_1 oder von S_2 ein Schritt zur nächsten *pause*-Anweisung unternommen wird

Disjunktiv aktive Anweisungen der Form $S_1||S_2$, $S_1|||S_2$ oder $S_1|S_2$ sind aktiv, solange S_1 oder S_2 aktiv ist, wohingegen konjunktiv aktive Ausdrücke der Form $S_1\&\&S_2$, $S_1\&\&\&S_2$ oder $S_1\&S_2$ nur aktiv sind, solange S_1 und S_2 aktiv sind.

[7] bedingte Aktionen/Guarded Action: Tupel aus boolescher Bedingung und Aktion

Semantische Probleme

Bei synchronen Sprachen können semantische Probleme auftreten, die vom Compiler behandelt werden müssen [Sch09].

Schizophrene Anweisungen Schizophrene Anweisungen haben innerhalb eines Makroschrittes mehrere Instanzen einer Variablen. Dies kommt nur vor, wenn sich die Anweisung in einer Schleife befindet. Mehrere Instanzen einer Variablen treten insbesondere in verschachtelten Schleifen auf, wenn alle Schleifen zum selben Zeitpunkt abgebrochen und neu gestartet werden. In jedem Makroschritt wird der Wert einer Variablen einmalig bestimmt. Allerdings werden die Geltungsbereiche der Variablen in den Mikroschritten betreten und verlassen, sodass diese Geltungsbereiche in einer schizophrenen Anweisung beachtet werden müssen. Dies wird als Reinkarnation einer lokalen Variablen bezeichnet. Für diese Reinkarnationen werden Kopien der lokal deklarierten Variablen entsprechend der Anzahl an möglichen Eintritten in ihren Wirkungsbereich angelegt [Sch09].

```
module Schizophrenic(event !x0,!x1,!
    x2,!x3) {
  loop {
    event x;
    if(x) x1 = true; else x0 = true;
    assert(!x);
    pause;
    x = true;
    if(x) x3 = true; else x2 = true;
    assert(x);
  }
}
```

Abbildung 3.2: Schizophrene Deklaration

Bedingung	→	**Aktion**		
True	→	x	=	True
x	→	x3	=	True
!x	→	x2	=	True
x@0	→	x1	=	True
!x@0	→	x0	=	True

Abbildung 3.3: resultierende bedingte Aktionen für den Schleifenzustand

Abbildung 3.2 zeigt ein Quartzprogramm mit schizophrenen Anweisungen. Durch die Schleife befinden sich zwei durch ihren Geltungsbereich unterschiedliche Variablen mit dem Namen x innerhalb eines Makroschrittes. Der Compiler „qrz2aif“ legt deshalb weitere Variablen an. Abbildung 3.3 zeigt die bedingten Aktionen dieses Makroschrittes. Nach Anwendung der EFSM-Transformation befinden sich diese Aktionen innerhalb eines Zustandes des resultierenden Automaten.

Probleme der Kausalität Kausalitätszyklen entstehen, wenn eine Aktion durch ihre Ausführung im selben Schritt ihre Vorbedingung ändert. Daher kann es vorkommen, dass eine Aktion im Widerspruch zu ihrer Vorbedingung steht (siehe Programmfragment 3.4). Diese Probleme der Kausalität müssen von Compilern für synchrone Sprachen erkannt werden [Sch09].

Datentypen

Quartz hat nur statische Typen.

```
module P (event o){
if(!o) o = true;
}
```

Abbildung 3.4: Quartz-Programmfragment mit Kausalitätszyklus[Sch09]

Definition 1 (Quartz-Datentypen). *[Sch09]*
Ausdrücke in Quartz haben einen der folgenden Datentypen, wobei n ein statischer Ausdruck vom Typ nat ist und $n > 0$ gilt. Atomare Datentypen:

- *bool: boolean-Werte true und false*
- *bv[n]: Bitvektor mit n Bits*
- *nat $< n >$: Integer ohne Vorzeichen*
- *int $< n >$: Integer mit Vorzeichen*
- *bv: unbeschränkter Bitvektor*
- *nat: unbeschränkter Integer ohne Vorzeichen*
- *int: unbeschränkter Integer mit Vorzeichen*

Zusammengesetzte Datentypen:

- *$array(\alpha, n)$: Array des Typs α der Länge n*
- *$\alpha * \beta$: Tupel, zusammengesetzt aus den Datentypen α und β*

Eine kompakte Zusammenfassung aller Ausdrücke und Datentypen in Quartz bietet die Quartz Reference Card (siehe [Emb]).

Informationsfluss von Variablen In Quartz muss für jede Variable bei ihrer Deklaration neben einem Datentypen der Informationsfluss und ein Speichertyp angegeben werden. Durch Angabe des Informationsflusses werden Variablen als *input*, *inout*, *output* oder *local* klassifiziert. Als input deklarierte Variablen können nur gelesen werden, *output*-Variablen können nur geschrieben werden, *inout* und *local*-Variablen können sowohl gelesen als auch geschrieben werden, wobei auf *local*-Variablen nur innerhalb ihres Geltungsbereiches zugegriffen werden kann [Sch09].

Speichertypen von Variablen Der Speichertyp einer Variablen muss entweder als *event* oder als *memorized* deklariert werden. Wird der Wert einer Variablen in einem Makroschritt nicht durch einen Mikroschritt bestimmt, findet eine „reaction to absence“[8] statt. Als *event* deklarierte Variablen werden in einem solchen Fall auf ihren Standardwert gesetzt, *memori-*

[8] reaction to absence, Reaktion auf Abwesenheit

zed-Variablen behalten ihren Wert aus dem vorherigen Makroschritt. *Event* entspricht somit einem Signal und *memorized* einem gepufferten Wert [Sch09].

3.4 Averest und das Averest Intermediate Format

Averest Bei der Implementierung werden zahlreiche Funktionen aus Averest verwendet. Averest ist ein Framework zur Spezifikation, Verifikation und Implementierung reaktiver Systeme. Es wird von der AG Eingebettete Systeme an der TU Kaiserslautern entwickelt. Das Framework ist weitestgehend in F# geschrieben. [Emb]

Averest Intermediate Format Beim Entwurf eingebetteter Systeme ist es üblich, dass der Prozess der Kompilierung und Synthese unterteilt wird. Da Zielplattformen häufig wechseln, sind Zwischenformate erstrebenswert. Im AIF werden synchrone bedingte Aktionen[9] verwendet, um das Verhalten eines Systems zu beschreiben. Synchrone bedingte Aktionen sind mit Guarded Commands[10] vergleichbar, verhalten sich aber dem synchronen Berechnungsmodell[11] entsprechend.
Sowohl Kontrollfluss als auch Datenfluss werden durch eine Menge von bedingten Aktionen der Form $\langle \gamma \Rightarrow \mathcal{C} \rangle$ beschrieben. Dabei ist γ ein boolescher Ausdruck und $\mathcal{C}$ eine Aktion, die einer Aktion der Quellsprache entspricht. Die Aktion $\mathcal{C}$ wird ausgeführt, wenn die Bedingung γ erfüllt ist. Bedingte Aktionen bieten einen guten Kompromiss zwischen dem Entfernen von Komplexität durch Abstraktion vom Quellcode und der Unabhängigkeit von Zielsystemen. [BS11]

3.5 Erweiterter endlicher Zustandsautomat (EFSM)

Definition 2 (EFSM). *[BBS11]*
Ein erweiterter endlicher Zustandsautomat ist ein Tupel (S,s_0,T,D), wobei:

- *S: Zustandsmenge*
- *$s_0 \epsilon S$: Startzustand*
- *$T \subseteq (S \times C \times S)$: endliche Menge von Übergangsrelationen, wobei C die Menge der Bedingungen für die Übergänge ist*
- *D: Funktion $S \rightarrow \mathcal{D}$, die jedem Zustand $s \epsilon S$ eine Menge von bedingten Aktionen $D(s) \subseteq \mathcal{D}$ zuordnet*

Um schnellen sequentiellen Code aus synchronen Programmen zu generieren, kann die Repräsentation des Programms als erweiterter endlicher Zustandsautomat[12] verwendet werden. In Averest existiert eine EFSM-Implementierung, die aus den bedingten Aktionen für Kontroll- und Datenfluss einen Zustandsgraphen erstellt. Dabei repräsentiert jeder Zustand s

[9] synchrone bedingte Aktionen, synchronous Guarded Actions
[10] Guarded Command: Anweisung, die ausgeführt wird, falls eine Vorbedingung erfüllt ist
[11] Berechnungsmodell, MoC, Model of Computation
[12] EFSM: Extended Finite State Machine, erweiterter endlicher Zustandsautomat

eine Teilmenge der Programmlabel, $Labels(s) \subseteq \mathcal{L}$. Ein Zustand enthält dann alle Aktionen, die für den Makroschritt nötig sind, der diesem Zustand entspricht. Die Zustandsübergänge der EFSM werden aus den Bedingungen der bedingten Aktionen des Kontrollflusses erzeugt. Wird ausgehend von dieser EFSM für das Zielsystem Quellcode erzeugt, so müssen für jeden Makroschritt nur noch die benötigten Aktionen betrachtet werden.
Im Unterschied zu einem Kontrollflussgraphen, bei dem die Zustände Anweisungen enthalten, die sequentiell ausgeführt werden können, enthält die EFSM Aktionen, die innerhalb eines Makroschrittes nebenläufig ausgeführt werden müssen. Zustandsübergänge der EFSM beenden einen Makroschritt.[BBS11]

Ein Nachteil der Codegenerierung aus der Repräsentation eines synchronen Programms als Zustandsautomat ist die Größe des Zielcodes. Die Anzahl der *pause*-Anweisungen eines Programms der Länge n liegt in $\mathcal{O}(n)$. Daher befindet sich die Anzahl der Zustände eines zu einem solchen Programm zugehörigen Zustandsautomaten in $\mathcal{O}(2^n)$ [Sch09].

3.6 Static Single-Assignment Form

Bei der Analyse des Datenflusses bauen Compiler häufig Def-Use-Ketten[13] für Variablen auf. Die static single-assignment Form (SSA-Form) ist eine Verbesserung der Def-Use-Ketten, da jede Variable nur einmal statisch definiert wird. Die Definition kann jedoch dynamisch öfter ausgeführt werden. Folgende Vorteile entstehen durch die Verwendung der SSA-Form:

- Datenflussanalyse und Optimierungsalgorithmen können vereinfacht werden, da jede Variable nur einmal definiert wird.
- Die Repräsentation von Def-Use-Ketten benötigt viel Speicherplatz (quadratisch abhängig von der Größe des Programms), wohingegen der Speicherplatz, den ein in die SSA-Form transformiertes Programm benötigt, nur linear von der Größe des Ursprungsprogramms abhängt.
- Def-Anweisungen sind im Kontrollflussgraphen Dominatoren[14] der Use-Anweisungen, dadurch wird z.B. die Konstruktion von Interferenzgraphen vereinfacht, die z.B. bei der Registerzuteilung verwendet werden.
- Verwendungen einer Variablen an mehreren Programmstellen ohne Bezug gibt es in der SSA-Form nicht mehr, gibt es mehrere Schleifen, die dieselbe Schleifenvariable haben, muss für diese Variable bei der Codeerzeugung nicht dasselbe Register verwendet werden.

Durch die Verwendung der SSA-Form können Optimierungen des Compilers schneller ausgeführt werden. [App02]

3.7 Scheduling

Während ein Sequenzgraph die Abhängigkeiten zwischen Operationen beschreibt, werden durch einen Schedule die genauen Startzeitpunkte der Operationen und somit deren Ne-

[13] Def-Use-Kette: Liste von Pointern auf Vorkommen von Definition und Verwendung einer Variablen
[14] Dominator: geht jeder Pfad im Kontrollflussgraphen vom Startknoten zum Zielknoten durch einen Knoten d, so ist dieser ein Dominator des Zielknotens

benläufigkeit bestimmt. Bei Implementierungen in Hardware wird die Anzahl an nebenläufigen Operationen pro Zeitpunkt durch Hardwareressourcen beschränkt. Ist nur eine Ressource vorhanden, müssen die Operationen sequentiell ausgeführt werden.
Sind Ressourcenkonflikte durch Serialisierung von Operationen gelöst, die dieselbe Ressource benötigen, können Scheudlingalgorithmen angewendet werden, die keine Beschränkungen durch Ressourcen berücksichtigen [Dem94].

ASAP Scheduling ASAP[15] Scheduling berücksichtigt keine Ressourcenbeschränkungen. Bei diesem Schedulingalgorithmus wird für den Startzeitpunkt einer Operation jeweils der frühestmögliche Zeitpunkt ausgewählt [Dem94].
Es ist außerdem bekannt, dass ASAP Scheduling ein exakter, datenflussorientierter Algorithmus ist, dessen Laufzeitkomplexität in $\mathcal{O}(|E|)$ liegt, wobei E die Menge der Kanten des Graphen der Abhängigkeiten ist.

3.8 Lebendigkeit von Variablen

In der Zwischendarstellung eines Programms existieren viele temporäre Variablen, ein Prozessor, auf dem der Zielcode ausgeführt wird, hat jedoch nur wenige Register. Für die Registerzuteilung ist daher von Interesse, welche Variablen zum selben Zeitpunkt in den Registern gehalten werden müssen. Dazu werden Uses und Defs betrachtet [App02].

Definition 3 (Uses und Defs)**.** *[App02]*
Eine Zuweisung zu einer (temporären) Variablen definiert diese (Def). Vorkommen einer Variablen auf der rechten Seite einer Zuweisung bedeuten eine Verwendung dieser Variablen (Use).

Eine Variable gilt dann als lebendig, wenn sie einen Wert enthält, der zu einem späteren Zeitpunkt im Programm benötigt wird. Zur Analyse der Lebendigkeit einer Variablen werden die Anweisungen eines Kontrollflussgraphen untersucht [App02].

Definition 4 (Lebendigkeit)**.** *[App02]*
Eine Variable ist an einer Kante des Kontrollflussgraphen lebendig, wenn es von dieser Kante aus einen Weg zu einem Use der Variablen gibt, der nicht durch ein Def der Variablen geht.

3.9 Registerzuteilung und Graphfärben

Die Aufgabe der Registerzuteilung ist es, die Variablen eines Programms Registern zuzuteilen. Das Problem der Registerzuteilung ist NP-vollständig[16] und kann auf das Problem der Graphfärbung reduziert werden [Cha81],[App02]. Dieses Problem besteht darin, dass den Knoten eines Graphen Farben zugeordnet werden sollen. Dabei müssen zwei Knoten, die durch eine

[15] ASAP: as soon as possible; deutsch: so früh wie möglich
[16] NP-vollständig: nichtdeterministisch in Polynomialzeit lösbare Probleme, geratene Lösung kann in Polynomialzeit überprüft werden

Kante verbunden sind, unterschiedliche Farben zugeordnet werden. Die Anzahl K an Farben ist dabei fest und für die Registerzuteilung entsprechen die Farben den Registern. Einen Graph, für den eine Färbung mit K Farben existiert, nennt man K-färbbar. Das Problem der Graphfärbung ist zwar NP-vollständig, es existiert allerdings ein Algorithmus, der in linearer Zeit ein heuristisches Ergebnis liefert [App02]:

Erzeugen des Interferenzgraphen Durch die Analyse der Lebendigkeit der Variablen kann ein Interferenzgraph gewonnen werden. Die Knoten dieses Graphen sind die Variablen des Programms. Zwei Knoten sind durch eine Kante verbunden, wenn ihre Variablen nicht demselben Register zugeordnet werden können. Der häufigste Grund dafür ist die Lebendigkeit zweier Variablen zum gleichen Zeitpunkt. Kanten im Interferenzgraphen können allerdings auch durch Eigenschaften des Prozessors entstehen, z.B. wenn eine Anweisung Ergebnisse nur in bestimmte Registertypen schreiben kann [App02].

Vereinfachen des Graphen Gibt es im Graphen einen Knoten mit weniger als K Kanten zu unterschiedlichen Nachbarn, wird dieser aus dem Graphen entfernt und auf einen Stapel gelegt. Wenn der resultierende Graph K-färbbar ist, dann ist es auch der ursprüngliche, da dem entfernten Knoten eine Farbe zugewiesen werden kann. Diese Vereinfachung wird solange ausgeführt, bis es keinen Knoten mehr gibt, der weniger als K Kanten hat [App02].

Auslagern von Variablen Kann ein Graph nicht weiter vereinfacht werden, besteht er nur noch aus Knoten mit mindestens K Nachbarn. Einer dieser Knoten wird auch auf den Stapel gelegt, allerdings markiert, da nicht sichergestellt werden kann, dass eine Färbung gefunden wird, bei der der Knoten einer Farbe zugeordnet werden kann. Gibt es keine solche Färbung, wird die Variable dieses Knotens nicht durch ein Register, sondern im Speicher repräsentiert. Nachdem der Knoten entfernt wurde, wird mit dem Vereinfachen des Graphen fortgefahren [App02].

Auswählen von Farben Beginnend mit einem leeren Graphen wird der ursprüngliche Graph rekonstruiert. Knoten werden vom oberen Ende des angelegten Stapels genommen und in den Graphen eingefügt. Für einen unmarkierten Knoten kann immer eine Farbe gefunden werden, für einen markierten Knoten muss überprüft werden, ob seine Nachbarn im Graphen mit weniger als K Farben gefärbt sind. Ist das nicht der Fall, wird die entsprechende Variable im Speicher repräsentiert. Bei der Codeerzeugung muss dann Code für das Sichern von Registerinhalten und das Laden der Variablen angelegt werden [App02].

3.10 Die Abacus-Prozessorarchitektur

Die Abacus-Prozessorarchitektur ist eine RISC[17]-Prozessorarchitektur der AG Eingebettete Systeme der Technischen Universität Kaiserslautern, die für Lehr- und Forschungszwecke eingesetzt wird. Abacus ist eine 16-Bit-Architektur[18]. Ein Abacus-Prozessor hat sieben frei verwendbare Register, ein Nullregister und ein zusätzliches Register zur Speicherung des

[17] RISC: Reduced Instruction Set Computer
[18] Registerbreite beträgt 16 Bit, Befehlswörter 16 Bit

Überlaufs, welches die obere Hälfte des Ergebnisses einer ALU[19]-Operation enthält. Die ersten sechs Bit eines Befehls enthalten den Opcode[20]. Zur Adressierung der acht Register werden bei der Codierung eines Befehls drei Bit verwendet. Es gibt folgende Befehlstypen:

Befehlstyp	6 Bit	10 Bit
R-Typ:	Opcode,	3 Register
I-Typ:	Opcode,	2 Register, 4-Bit Direktoperand
S-Typ:	Opcode,	1 Register, 7-Bit Direktoperand
J-Typ:	Opcode,	10-Bit Direktoperand

Dabei sind die R- und I-Typen Arithmetik- und Logikbefehle, die S-Typen Speicherzugriffsbefehle und die J-Typen Kontrollstrukturbefehle. Die Arithmetikbefehle für Addition, Subtraktion, Multiplikation und Division sind alle in vier Varianten verfügbar: jeweils als R- und I-Typ und jeweils für vorzeichenlose Zahlen und Zahlen mit Vorzeichen. Die Kontrollstrukturbefehle (Sprungbefehle) sind alle relativ zum PC[21], es gibt also keine absoluten Sprünge.[ES14]

[19] arithmetic logic unit, deutsch: arithmetisch-logische Einheit
[20] operation code, Code eines Maschinenbefehls
[21] Program Counter, Befehlszähler, aktuelle Position im Programm

4 Implementierung

4.1 Programmiersprache

Wegen der Eignung für den Übersetzerbau wurde für die Implementierung eine funktionale Programmiersprache gewählt. Da das Averest-Framework in .NET[1] geschrieben wurde, fiel die Wahl auf F#. Dadurch lässt sich leicht auf die Funktionen des Frameworks Averest zugreifen.

4.2 Übersetzung in das Zwischenformat AIF

Da sich das Quellprogramm mit Hilfe des Compilers „qrz2aif“, der sich im Averest-Framework befindet, in das Zwischenformat AIF übersetzen lässt, ist die Analysephase (siehe 3.1.1) des Compilers bereits abgeschlossen. Das Programm liegt dann in einer Datenstruktur vor, auf die von einem F#-Programm aus zugegriffen werden kann.

Eliminieren zusammengesetzter Datentypen In der vorliegenden Datenstruktur können zusammengesetzte Datentypen wie z.B. Tupel oder Arrays vorkommen, die aus mehreren einfachen Datentypen bestehen. Um diese zusammengesetzten Datentypen zu eliminieren, wird die Transformation „EliminateCompound“[Emb] eingesetzt. Diese ersetzt jedes Element eines zusammengesetzten Datentyps durch eine Variable einfachen Datentyps, wobei die Indizes der Variablen im zusammengesetzten Datentyp an den Variablennamen angehängt werden.

Hinzufügen der Reaktion auf Abwesenheit Im AIF ist die Reaktion auf Abwesenheit implizit, sie wird getrennt von den bedingten Aktionen gespeichert [BS11]. Das Averest-Framework bietet allerdings Funktionen, um die Reaktion auf Abwesenheit in Form von bedingten Aktionen hinzuzufügen. In der in dieser Arbeit vorgestellten Implementierung wird die Reaktion auf Abwesenheit für Variablen von Speichertyp *event* hinzugefügt. Daher muss bei der Codegenerierung nicht mehr zwischen den Speichertypen unterschieden werden.

4.3 Laden von Konstanten und Speichern von Registerinhalten

Beim Laden einer Konstanten muss beachtet werden, dass diese nicht immer in einen einzelnen Befehl passt. Da die Registerbreite 16 Bit beträgt, der Befehl *movu* allerdings nur sieben Bit lange vorzeichenlose Zahlen in ein Register laden kann, müssen Konstanten, deren Binärdarstellung länger als sieben Bit ist, schrittweise geladen werden. Dazu werden die oberen sieben Bit der Konstanten in das Register geladen und verschoben. Da es im Abacus-Befehlssatz

[1] .NET: Framework von Microsoft

keinen Befehl speziell zum Verschieben eines Registerinhalts gibt, wird dazu ein Multiplikationsbefehl verwendet. Anschließend werden die nächsten Bits auf das Register addiert. Das Verschieben des Registerinhalts und die Addition weiterer Bits wird solange wiederholt, bis die Konstante vollständig im Register steht. In Tabelle 4.1 ist zum Vergleich des resultierenden Codes der Assemblercode für das Laden einer Konstanten mit sieben Bit und einer größeren Konstanten angegeben.

movu r, 127	129 1000 0001
	btv2 1 0000
	btv1 001
		movu r, btv2
		muliu r, r, 8
		addiu r, r, btv1

Tabelle 4.1: *(a)* Durch 7 Bit darstellbare Konstante
(b) Aufteilen der Binärdarstellung
(c) Konstante mit längerer Binärdarstellung

Auch wenn Registerinhalte zurück in den Speicher geschrieben werden, muss beachtet werden, dass nicht alle Konstanten in den Befehl *st* passen. Dieser Speicherbefehl enthält das Register, dessen Inhalt in den Speicher geschrieben werden soll, und außerdem ein Register und eine Konstante, die verrechnet die Speicheradresse ergeben. Passt die Binärdarstellung einer Adresse nicht in den Befehl, muss diese Adresse zunächst in ein Register geladen werden.

st r, r0, 15	Adresse in r1 laden st r, r1, 0

Tabelle 4.2: *(a)* Durch 4 Bit darstellbare Konstante
(b) Laden einer Adresse, die nicht durch 4 Bit darstellbar ist

4.4 Implementierung mit globalem Scheduling

4.4.1 Aufgaben des Compilers

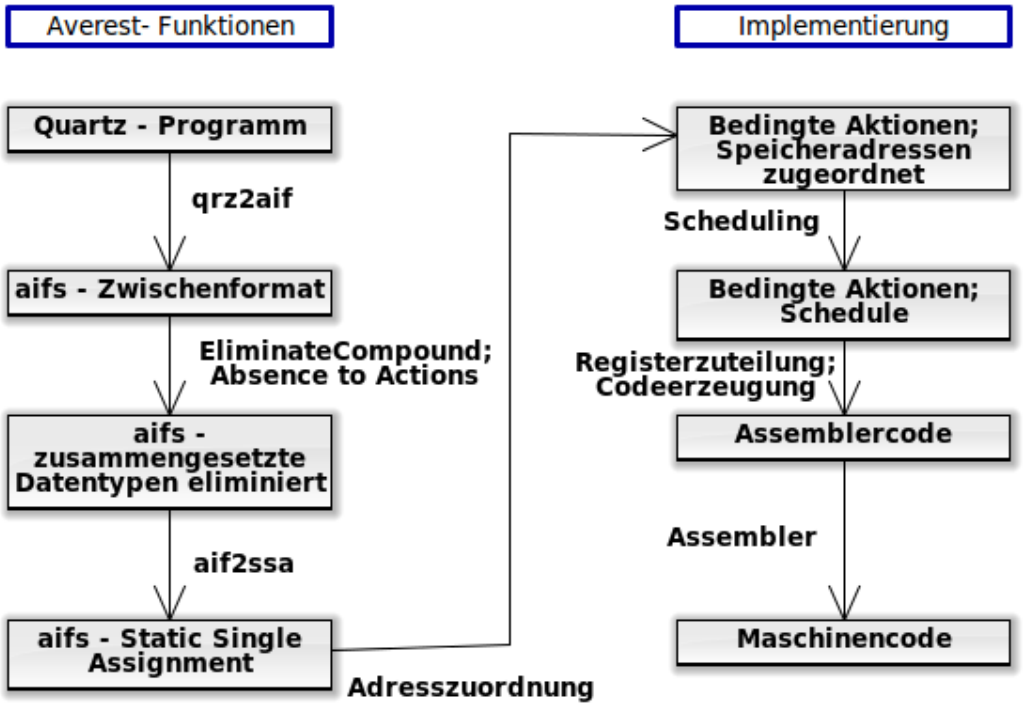

Abbildung 4.1: Implementierung des Compilers

Mit dem Programm „aif2ssa“ wird der Zwischencode[2] in die SSA-Form[3] weiter transformiert. Für Variablen des Speichertyps event müssen Aktionen eingefügt werden, die diese auf ihren Standardwert zurücksetzen, da zugewiesene event-Variablen für die weitere Verwendung zwischengespeichert werden. Nach Anwendung dieser Funktionen müssen den Variablen Speicheradressen zugeordnet werden, Datenabhängigkeiten müssen aufgelöst werden und aus dem Zwischencode muss Assemblercode erzeugt werden, welcher in Maschinencode übersetzt werden muss, um auf dem Zielsystem ausgeführt werden zu können (siehe Abbildung 4.1). Dazu läuft der Zielcode in einer Endlosschleife, in jeder Iteration wird somit ein Makroschritt abgearbeitet.

4.4.2 Adresszuordnung

Durch die Transformation „EliminateCompound “[Emb] wurde die Adresszuordnung vereinfacht, da es auf diese Weise nur noch einfache Datentypen gibt, die direkt Speicheradressen zugeordnet werden können. Einem Element der Liste der Variablen wird die jeweils nächste

[2]Zwischencode: Code im Zwischenformat
[3]SSA-Form: Static Single Assignment Form

Speicheradresse zugewiesen. Die Variablennamen werden zusammen mit den jeweils zugehörigen Speicheradressen in der F#-Datenstruktur *Dictionary* abgespeichert, weil diese Datenstruktur einen schnellen Zugriff ermöglicht [Mic].

4.4.3 Scheduling

Um Datenabhängigkeiten aufzulösen, wird zunächst ein Datenabhängigkeitsgraph aufgestellt, der die echten[4] Datenabhängigkeiten beschreibt. Anschließend werden die einzelnen bedingten Aktionen unter Berücksichtigung der Abhängigkeiten umsortiert. Das Ergebnis des Schedulings ist eine Liste von bedingten Aktionen, deren Abhängigkeiten aufgelöst sind und die daher für die Codeerzeugung verwendet werden kann.

Globales Scheduling

Das Scheduling wird naiv implementiert. Dazu werden Datenabhängigkeiten zwischen Aktionen global betrachtet. Mit der Methode MkActionGraph aus Averest wird ein Graph erzeugt, mit dem überprüft werden kann, ob eine Datenabhängigkeit zwischen zwei Aktionen besteht.

Naiver Algorithmus zum Auflösen der Datenabhängigkeiten:

```
let rec SolveDependencies (readyList:GrdAction list)
                          (todoList:GrdAction list)
                          (actionGraph:ActionGraph): GrdAction list =
  match todoList with
    | ga::restlist ->
      if (HasForwDep ga todoList actionGraph)
      then SolveDependencies readyList (restlist@[ga]) actionGraph
      else SolveDependencies (readyList @ [ga]) restlist (
          updatedActionGraph actionGraph ga)
    | _ -> readyList
```

Vor dem Aufruf muss mit MkActionGraph der Datenabhängigkeitsgraph erzeugt werden. Dieser Schedulingalgorithmus wird auf die direkten Zuweisungen angewendet. Die verzögerten Zuweisungen werden gesondert behandelt und an den Schedule angehängt. Beim Aufruf wird eine leere Liste für die bereits sortierten Aktionen (readyList) und die Liste aller Aktionen für die zu sortierenden Aktionen (todoList) übergeben. Der Algorithmus nimmt das erste Element der todoList und überprüft, ob es von einem anderen Element der todoList abhängt, also ob eine RAW-Datenabhängigkeit besteht. Besteht keine Abhängigkeit, wird das Element beim rekursiven Aufruf des Algorithmus an die readyList angehangen. Besteht eine Datenabhängigkeit, wird es an das Ende der todoList geschoben. Für einen nicht-zyklischen Graphen werden die echten Datenabhängigkeiten dadurch aufgelöst.
Bei dieser globalen Betrachtung der Datenabhängigkeiten muss der Programmierer allerdings in seinen Möglichkeiten beschränkt werden, da ansonsten Zyklen im Graphen der Abhängigkeiten auftreten können und kein Schedule gefunden werden kann. Programme, bei denen

[4] echte Datenabhängigkeit: RAW, read-after-write

solche Zyklen auftreten würden, müssen umgeschrieben werden. Werden Variablen vom Programmierer nur einmal zugewiesen, tritt dieses Problem nicht mehr auf.

Laufzeitkomplexität Im Worst Case[5] ist nur das jeweils letzte Element der Liste der noch zu sortierenden Aktionen (todoList) nicht von einer anderen Aktion abhängig. Daraus ergibt sich eine Laufzeitkomplexität von $\mathcal{O}(n^2)$.

4.4.4 Registerzuteilung

Zunächst wird die Codeerzeugung mit einer naiven Registerzuteilung implementiert, bei der jeweils die Variablen einer bedingten Aktion in die vorhandenen Register geladen werden und nach dem Codeblock der Aktion wieder in den Speicher zurückgeschrieben werden. Nach der Transformation in SSA-Form besteht eine bedingte Aktion nur noch aus einer Bedingung, die eine Variable oder eine Konstante enthalten kann, und einer Aktion, die höchstens drei Variablen enthält. In dieser Implementierung wird zunächst, falls die Vorbedingung aus einer Variablen besteht, der Wert dieser Variablen aus dem Speicher in ein Register geladen. Dieses wird mit dem Nullregister verglichen, dessen Inhalt zu dem boolean-Wert false äquivalent ist. Ist der geladene Wert ebenfalls false, wird der generierte Code für die Aktion übersprungen. Zur Ausführung der Aktion werden alle benötigten Variablen in die Register des Prozessors geladen und die Berechnung durchgeführt. Zuletzt wird die veränderte Variable in den Speicher zurückgeschrieben. Durch die SSA-Form enthält die Bedingung höchstens eine Variable, die Aktion höchsten drei, von denen nur eine zurückgeschrieben wird. Diese Implementierung kommt also mit drei Registern aus. Da der Prozessor (siehe 3.10) 7 GPR[6] besitzt, sind bei dieser Implementierung immer genügend Register vorhanden, in die Daten hineingeladen werden können.

Bedingung	→	**Aktion**
b	→	x = y + z

Abbildung 4.2: Beispiel: bedingte Aktionen in SSA-Form

```
movu r1,localsMem.[b]
ld r1,r1,0
bez r1,ActionBlock_length+1
```

```
movu r1,localsMem.[y]
ld r1,r1,0
movu r2,localsMem.[z]
ld r2,r2,0
add r1,r1,r2
st r1,r0,lhsAdress
```

Abbildung 4.3: Generierter Assemblercode

4.5 Implementierung anhand des Kontrollflussgraphen

Da ein Kontrollflussgraph passend für das Verhalten eines sequentiellen Prozessors ist [Edw03], kann der Zielcode durch die Berücksichtigung des Kontrollflusses effizienter werden. Bei der

[5] Worst Case: ungünstigster anzunehmender Fall
[6] GPR: General Purpouse Register, frei verwendbare Register

vorläufigen Implementierung wurden nur Datenabhängigkeiten betrachtet. Das hatte zu Folge, dass bei der Ausführung des Programms die Bedingung jeder einzelnen bedingten Aktion ausgewertet werden musste.

4.5.1 Aufgaben des Compilers

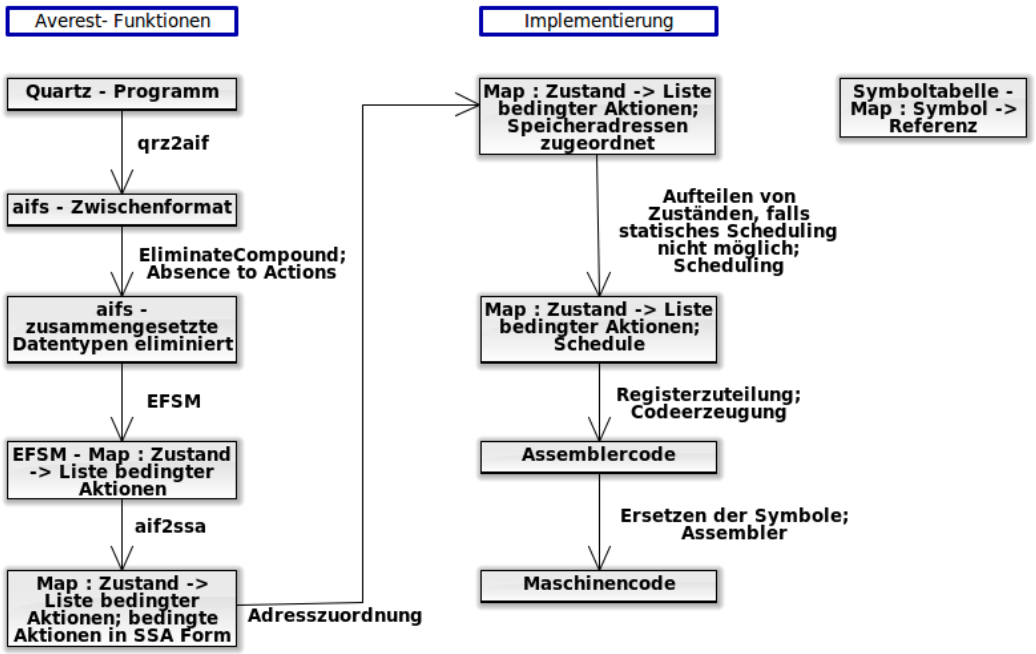

Abbildung 4.4: Implementierung des Compilers

Durch Anwenden der „EFSM“[7]-Transformation des Averest Frameworks wird ein Zustandsautomat erzeugt, der den Kontrollfluss des Quartzprogramms beschreibt. Bei der Erstellung dieses Graphen werden die Makroschritte betrachtet und somit bestimmt, welche Aktionen sich zusammen in einem Zustand befinden. Innerhalb eines Zustandes befindet sich eine ungeordnete Menge von Aktionen. Auch der Kontrollfluss des Graphen wird durch bedingte Aktionen repräsentiert. Auf die Menge aller Aktionen in den Zuständen des Graphen und des Kontrollflusses wird die Transformation „aif2ssa“ angewendet. Diese ersetzt alle Aktionen durch äquivalente Aktionen in SSA-Form. Dazu werden weitere bedingte Aktionen angelegt, deren Aktionen Teilausdrücke der ursprünglichen Aktionen darstellen. Die neu angelegten Aktionen enthalten eine Zuweisung eines Ausdrucks zu einer Variablen vom Speichertyp event. Nach Anwendung der SSA-Transformation müssen diese Aktionen in alle Zustände kopiert werden, in denen sie entweder in den bedingten Aktionen des Datenflusses oder in den bedingten Aktionen, die die Zustandsübergänge beschreiben, vorkommen.

[7] EFSM: Extended finite state machine, erweiterter endlicher Zustandsautomat

4.5.2 Adresszuordnung

Die Adresszuordnung erfolgt wie in Abschnitt 4.4.2. Es gibt wieder nur einfache Datentypen, sodass jede Variable einer Speicheradresse zugeordnet werden kann. Dabei müssen allerdings nur verwendeten Variablen Speicheradressen zugeordnet werden. Durch die EFSM entfallen Variablen für den Kontrollfluss.

4.5.3 ASAP-Scheduling und Kontrollflussgraph

Das Scheduling wird jeweils auf die Menge der Aktionen eines Makroschrittes angewendet. Dabei werden zunächst nur Aktionen mit direkten Zuweisungen betrachtet. Die Methode MkActionGraph aus Averest wird verwendet, um den Graphen zu erzeugen, der die Datenabhängigkeiten der Aktionen beschreibt. Zum Auflösen der Datenabhängigkeiten wird ASAP-Scheduling verwendet. Dabei wird die Menge aller Aktionen bestimmt, die keine Abhängigkeit besitzen. Sie kann zur Generierung von sequentiellem Code beliebig geordnet werden. Die bestimmten Aktionen werden anschließend aus dem Datenabhängigkeitsgraphen gelöscht und der Algorithmus wird für die übrigen Aktionen wiederholt. Auf den entstandenen Mengen entsteht so eine Ordnung.

```
// starten mit rdyList = []
let rec SolveASAP (rdyList:GrdAction list) (todoList:GrdAction) list(
    actionGraph:ActionGraph): (GrdAction list * GrdAction list) =
  if List.length todoList > 0
    then
      let new_todo,new_rdy = List.partition has_Dependency todoList
      let new_actionGraph = updatedActionGraph actionGraph new_rdy
      if List.length rdyList = List.length new_rdy
      then
        rdyList,todoList // in diesem Fall: Zyklus in der todoList
      else
        SlvASAP (rdyList@new_rdy) new_todo new_actionGraph
    else rdyList,[]
```

Da den Aktionen des Datenflusses eines Zustandes das synchrone Berechnungsmodell zugrunde liegt, kann für sie nicht in jedem Fall eine Anordnung gefunden werden, bei der alle Datenabhängigkeiten aufgelöst sind. Es können Zyklen auftreten. Der Schedulingalgorithmus liefert in diesem Fall neben den geordneten Aktionen eine Menge noch zu ordnender Aktionen.

```
{
pause;
if (b) {c:=d;}
else {d:=c;}
pause;
}
```

Abbildung 4.5: Quartz-Programmfragment

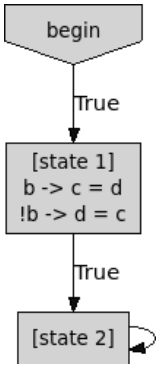

Abbildung 4.6: resultierende EFSM

Das Programmfragment in Abbildung 4.5 und die zugehörigen Aktionen in Abbildung 4.6 sind ein Beispiel für einen Makroschritt, für den kein statischer Schedule existiert. In diesem Fall müssen die übrig gebliebenen bedingten Aktionen auf Zugehörigkeit zu einem Zyklus untersucht werden. Gibt es Aktionen innerhalb eines Zyklus, die anhand ihrer Vorbedingungen unterschiedlichen Zuständen zugeordnet werden können, müssen diese Zustände erzeugt werden. Im Beispielprogramm kann die Bedingung b erst zur Laufzeit ausgewertet werden, weswegen neue Zustände erzeugt werden. Die Übergänge zu diesen Zuständen werden in diesem Fall durch die Bedingung b bestimmt.

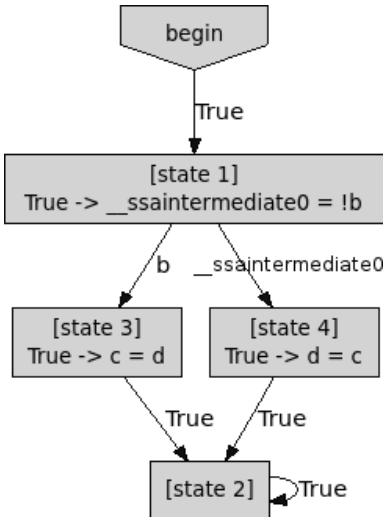

Abbildung 4.7: Kontrollflussgraph nach Aufteilung der Zustände

Alle Aktionen, für die noch kein Schedule gefunden wurde, werden in die entstandenen Zustände kopiert, wobei die Bedingung für den Übergang in einen entstandenen Zustand in allen Aktionen des Zustandes ausgewertet wird. Die Übergänge des ursprünglichen Zustandes werden von diesem entfernt und als Übergänge der Folgezustände verwendet. Durch rekursive Anwendung mit jeweiliger Durchführung des ASAP-Schedulings wird die EFSM in einen Kontrollflussgraphen überführt (siehe 4.7). Aus diesem lässt sich dann Maschinencode für einen sequentiellen Prozessor generieren.

4.5.4 Registerzuteilung durch Graphfärben

Für die Implementierung der Registerzuteilung wird der in Abschnitt 3.9 beschriebene Algorithmus verwendet. Dabei wird jeweils ein Zustand des Kontrollflussgraphen betrachtet. Da die bedingten Aktionen in die SSA-Form transformiert worden sind, ist die Konstruktion von Interferenzgraphen vereinfacht (siehe 3.6). Zur Konstruktion des Interferenzgraphen wird zunächst die Lebendigkeit jeder einzelnen Variablen bestimmt. Anders als in der Definition der Lebendigkeit von Variablen werden hierbei statt der Kanten die Knoten des Kontrollflussgraphen betrachtet. Erst bei der Codeerzeugung wird für einen Programmpunkt unterschieden, ob Variablen ab diesem oder bis zu diesem lebendig sind. Neben dem Punkt im Programm, an dem eine Variable geschrieben wird oder zum ersten Mal vorkommt, werden alle Punkte ermittelt, an denen sie gelesen wird. Bis zu dem letzten dieser Punkte ist die Variable dann lebendig. Für jeden Punkt im Programm wird eine Liste angelegt, die alle zu diesem Zeitpunkt lebendigen Variablen enthält. Aus diesen Listen kann dann der Interferenzgraph erzeugt werden. Abbildungen 4.8 zeigt die Lebendigkeit von Variablen und Abbildung 4.9 den zugehörigen Interferenzgraphen.

Bedingung	→	**Aktion**	**Lebendige Variablen**
True	→		
__lvar000	=	x<x_old	__lvar000, x, x_old
True	→		__lvar000, x
__ssai1	=	a/x	__lvar000, x, a, __ssai1
True	→		__lvar000, x, __ssai1
__ssai0	=	!__lvar000	__lvar000, x, __ssai1, __ssai0
True	→		__lvar000, x, __ssai1, __ssai0
__ssai2	=	x+__ssai1	__lvar000, x, __ssai1, __ssai0, __ssai2
__ssai0	→		__lvar000, x, __ssai0, __ssai2
rdy	=	True	__lvar000, x, __ssai2, rdy
True	→		__lvar000, x, __ssai2
__ssai3	=	__ssai2/2	__lvar000, x, __ssai2, __ssai3
__lvar000	→		__lvar000, x, __ssai3
Next(x_old)	=	x	__lvar000, x, __ssai3, x_old
__lvar000	→		__lvar000, __ssai3
Next(x)	=	__ssai3	__ssai3, x

Abbildung 4.8: Bedingte Aktionen eines Zustandes der EFSM eines Beispielprogramms

Nach Konstruktion des Interferenzgraphen beginnt die Phase der Vereinfachung. Die Abacus-Prozessorarchitektur besitzt sieben frei verwendbare Register und ein Nullregister. Bei dieser Implementierung des Graphfärbens werden sechs der sieben Register zum Bestimmen einer Färbung verwendet, da ein Register für das Zurückschreiben von Variablen in den Speicher benötigt wird. Dies ist beispielsweise nötig, wenn eine Speicheradresse nicht in den Befehl *st* passt und erst in ein Register geladen werden muss.
In der Phase der Vereinfachung werden die Knoten des Graphen durchlaufen und es werden

Knoten	**Nachbarn**
__lvar000	$\{x, x\_old, a, \_\_ssai1, \_\_ssai0, \_\_ssai2, rdy, \_\_ssai3\}$
x	$\{\_\_lvar000, x\_old, a, \_\_ssai1, \_\_ssai0, \_\_ssai2, rdy, \_\_ssai3\}$
x_old	$\{\_\_lvar000, x, \_\_ssai3\}$
a	$\{\_\_lvar000, x, \_\_ssai1\}$
__ssai1	$\{\_\_lvar000, x, a, \_\_ssai0, \_\_ssai2\}$
__ssai0	$\{\_\_lvar000, x, \_\_ssai1, \_\_ssai2\}$
__ssai2	$\{\_\_lvar000, x, \_\_ssai1, \_\_ssai0, rdy, \_\_ssai3\}$
rdy	$\{\_\_lvar000, x, \_\_ssai2\}$
__ssai3	$\{\_\_lvar000, x, \_\_ssai2, x\_old\}$

Abbildung 4.9: Interferenzgraph

alle Knoten auf einem Stapel abgespeichert und aus dem Graphen gelöscht, die weniger als sechs Kanten zu andern Knoten haben. Gibt es solche Knoten nicht mehr, muss ein Knoten mit sechs oder mehr Kanten aus dem Graphen entfernt werden. Auch dieser wird auf dem Stapel abgelegt, allerdings wird er für die nächste Phase markiert. Wurden alle Knoten aus dem Graphen gelöscht, wird dieser rekonstruiert, wobei jedem eingefügten Knoten eine Farbe (ein Register) zugewiesen wird, falls dies möglich ist, was für markierte Knoten nicht garantiert werden kann. Ist der Interferenzgraph nicht sechs-färbbar, muss während der Codeerzeugung Code zur Auslagerung von Variablen in den Speicher eingefügt werden.

Register	**zugeordnete Variablen**
r2	$\{\_\_ssai0, x\_old, a, rdy\}$
r3	$\{\_\_ssai1, \_\_ssai3\}$
r4	$\{x\}$
r5	$\{\_\_lvar000\}$
r6	$\{\_\_ssai2\}$
r7	$\{\}$

Abbildung 4.10: Färbung des Graphen

Adresse	**Variable**
1	a
2	x
3	rdy
5	__lvar000
6	x_old
7	__ssai0
8	__ssai1
9	__ssai2
10	__ssai3

Abbildung 4.11: Speicherzuordnung

Abbildung 4.10 zeigt die Färbung des Graphen aus Abbildung 4.9. Für dieses Beispiel wurde eine Färbung gefunden. Variablen können daher solange wie nötig in den Registern gehalten werden. Auf Abbildung 4.11 ist die Speicherzuordnung zu sehen, welche bei der Codeerzeugung (siehe Abbildung 4.12) verwendet wird.
Um für eine Vorbedingung oder eine Aktion Assemblercode zu erzeugen, muss zunächst die Lebendigkeit von Variablen des jeweils aktuellen Programmpunktes mit der Lebendigkeit von Variablen dess vorherigen Programmpunktes verglichen werden. Variablen, die am aktuellen Programmpunkt nicht mehr lebendig sind, müssen in den Speicher zurückgeschrieben werden. Der Wert einer Variablen, die neu hinzugekommen ist, muss aus dem Speicher in ein Register geladen werden, sofern diese gelesen wird. Wenn eine Variable neu hinzu kommt, allerdings nur geschrieben wird, muss sie nicht aus dem Speicher geladen werden und das durch das

Graphfärben bestimmte Register wird als Zielregister des zu schreibenden Wertes verwendet.

Im Vergleich zur einfachen Registerzuteilung aus Abschnitt 4.4.4 ist der Code wegen des Graphfärbens wesentlich kürzer. Der erzeugte Code in 4.12 ist bei einfacher Registerzuteilung 110% länger und hat 107% Speicherzugriffsbefehle mehr.

Bedingung	→	**Aktion**	**Erzeugter Assemblercode**
True	→		

__lvar000	=	x<x_old	`movu r4 2 // x laden`
			`ld r4 r4 0`
			`movu r2 6 // x_old laden`
			`ld r2 r2 0`
			`sltu r5 r4 r2`

True	→		`st r2 r0 6`

__ssai1	=	a/x	`movu r2 1 // a laden`
			`ld r2 r2 0`
			`divu r3 r2 r4`

True	→		`st r2 r0 1`

__ssai0	=	!__lvar000	`addi r1 r5 1`
			`mov r2 1`
			`and r2 r2 r1`

True	→		
__ssai2	=	x+__ssai1	`addu r6 r4 r3`
__ssai0	→		`st r3 r0 8` `bez r2 3`
rdy	=	True	`st r2 r0 7` `addi r2 r0 1`
True	→		`st r2 r0 3`
__ssai3	=	__ssai2/2	`divu r3 r6 r5`
__lvar000	→		`st r6 r0 9` `bez r5 2`
Next(x_old)	=	x	`add r2 r4 r0`

__lvar000	→		`st r4 r0 2`
			`st r2 r0 6`
			`bez r5 3`

Next(x)	=	__ssai3	`st r5 r0 5` `add r4 r3 r0`
			`st r4 r0 2` `st r3 r0 10`

Abbildung 4.12: Aus den bedingten Aktionen erzeugter Code

4.6 Codeerzeugung

4.6.1 Datenstruktur Assembler

Für die Codeerzeugung wird eine Datenstruktur angelegt, aus der einzelne Assemblerbefehle als Objekte angelegt werden können.

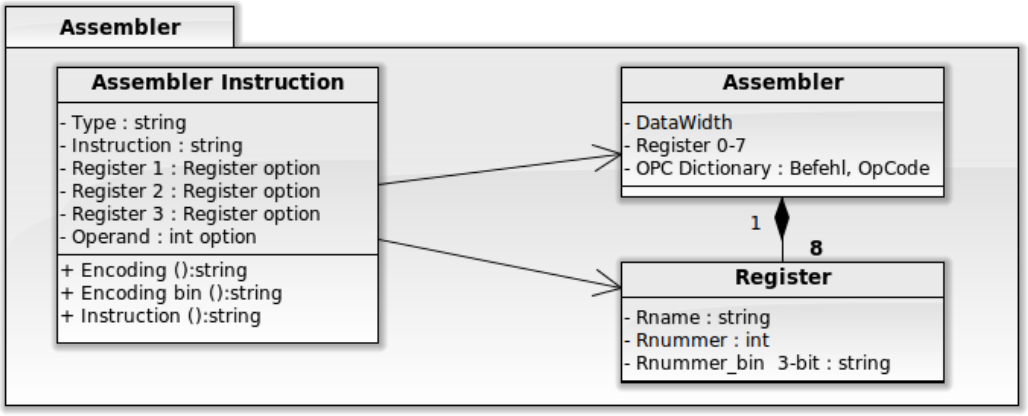

Abbildung 4.13: Datenstruktur Assembler

Wird ein Assemblerbefehl angelegt, speichert diese Datenstruktur den Befehl, dessen Typen, Register und ggf. den Operanden. Aus der Datenstruktur können dann die Binärdarstellung und die Darstellung als Assemblerbefehl ausgelesen werden.

4.6.2 Codeerzeugung aus einer bedingten Aktion

Für jeden einzelnen Zustand der EFSM wird eine Liste von Assemblerbefehlen erzeugt. Diese besteht jeweils aus dem Assemblercode für Vorbedingungen und Aktionen und dem Code, der für die Zustandsübergänge nötig ist.
Bei der Codeerzeugung wird davon ausgegangen, dass Datentypen eine Größe von 16 Bit nicht überschreiten. Es wird im Averest-Framework künftig Methoden geben, die größere Datentypen auflösen. Dies muss vor der Übersetzung aus dem AIF-Zwischenformat in Maschinencode geschehen.

Codeerzeugung aus einer Vorbedingung

Für die Vorbedingung einer bedingten Aktion muss Assemblercode erzeugt werden, der die eigentliche Aktion überspringt, falls die boolesche Bedingung nicht erfüllt ist (für eine bedingte Aktion der Form $b \rightarrow \mathit{Aktion}$ siehe Abbildung 4.14).

`movu rb,localsMem.[b]`	Adresse ins Register laden
`ld rb,rb,0`	Variable ins Register laden
`bez rb,Aktionsblock_length+1`	Verzweigung, Aktion ggf. überspringen
Aktionsblock	

Abbildung 4.14: Generierter Assemblercode

Codeerzeugung aus einer Aktion

Jede Aktion weist einer linksseitigen Variable einen Ausdruck zu. Dabei muss für die Codeerzeugung zwischen den atomaren Datentypen aus Kapitel 3.3 unterschieden werden. Je nach Datentyp werden unterschiedliche Befehle des Prozessors benötigt. Es wird Code für die Auswertung des jeweiligen Ausdrucks angelegt. Das Ergebnis dieser Auswertung wird dann in das Register geschrieben, welches der linksseitigen Variable zugeordnet ist. Dabei muss beachtet werden, dass die Register, aus denen Daten gelesen werden, nicht überschrieben werden dürfen. Das Laden von Werten in Register und das Speichern von Registerinhalten geschieht jeweils vor und nach der Codeerzeugung für eine Aktion. Wird ein Register für Zwischenergebnisse benötigt, kann das Register benutzt werden, welches nicht für die Färbung des Graphen verwendet wird.

Boolean und Bitvektoren Für Ausdrücke des Typs Boolean und für Bitvektoren sind vor allem die Assemblerbefehle *and*, *or*, *nand* und *nor* relevant, da einige Ausdrücke direkt auf diese Befehle abgebildet werden können. Für alle weiteren Ausdrücke müssen weitere Befehle verwendet werden. Abbildung 4.15 zeigt als Beispiel den erzeugten Code für einen Anweisung, bei der überprüft werden muss, ob zwei boolesche Variablen äquivalent sind.

rl = ra ⇔ rb	
`sub r1,ra,rb`	r1 = ra - rb
`bnz r1,3`	Ergebnis mit 0 vergleichen
`addi rl,r0,1`	r1 = r2 → True
`j 2`	nicht auszuführenden Code überspringen
`addi rl,r0,0`	r1 ≠ r2 → False

Abbildung 4.15: Generierter Assemblercode

Nat und Int Für Ausdrücke des Typs Nat sind vor allem Assemblerbefehle wie z.B. *addu* oder *mulu* relevant, welche Berechnungen mit vorzeichenlosen Zahlen ausführen. Ausdrücke vom Typ Int verwenden hauptsächlich Assemblerbefehle wie z.B. *add* oder *mul*, welche mit Vorzeichenzahlen rechnen. Ein Abacus-Prozessor verwendet für Vorzeichenzahlen deren Darstellung als Zweierkomplement [ES14]. Daher müssen Konstanten vom Typ Int, die in Register geladen werden soll, in die Darstellung als Zweierkomplement konvertiert werden.

4.6.3 Programmverzweigungen

Die Abacus-Prozessorarchitektur besitzt die Sprungbefehle *j* und *jmp*. Während *jmp* ein Register und einen 7-Bit Direktoperanden erhält, erhält *j* einen 10-Bit Direktoperanden. Beide Befehle setzen Sprünge relativ zum Befehlszähler um. Soll in einem Programm weiter gesprungen werden, müssen dazu Codeblöcke eingefügt werden:

Symbol	Befehl	Register und Operanden
	j	Ziel'
	...	
	j	2
Ziel'	j	Ziel
	...	
Ziel		

Abbildung 4.16: Codeblöcke für Sprungbefehle

4.6.4 Ersetzen von Symbolen

Bei der Implementierung mit globalem Scheduling wird nur Code für einzelne bedingte Aktionen erzeugt, die in einer Endlosschleife laufen. Nur für diese Endlosschleife müssen Codeblöcke für den Rücksprung eingefügt werden. Diese werden ausschließlich zwischen zwei Codeblöcken eingefügt, die jeweils aus einer bedingten Aktion entstanden sind, sodass keine Symbole eingeführt werden müssen.
Für die Implementierung anhand des Kontrollflussgraphen ist es notwendig, den Assemblerbefehl um zwei Variablen zu erweitern: es wird ein Integer für die Zeilennummer benötigt und eine Integer-Option für das Ziel von Sprungbefehlen. Die Symboltabelle referenziert in ihren Einträgen die Zeilennummern der Assemblerbefehle. Ziele von Sprungbefehlen sind Symbole, die als Integer in der Symboltabelle gespeichert werden. Wurden die Assemblercodestücke für alle Zustände des Kontrollflussgraphen erzeugt, werden diese zusammengesetzt. Anschließend werden die Zeilennummern der Assemblerbefehle angepasst, damit diese durchgehend nummeriert sind. Dann können anhand der Symbole die relativen Sprünge neu berechnet werden.

4.7 Assembler

Im letzten Schritt der Synthesephase wird die aus der Codeerzeugung entstandene Liste von Assemblerbefehlen in eine Liste aus Maschinencodebefehlen überführt. Dazu wird die in der Datenstruktur der Assemblerbefehle angelegte Binärdarstellung verwendet. Die Maschinencodebefehle werden in eine Binärdatei geschrieben, die dann von einem Abacus-Prozessor oder einem Simulator ausgeführt werden kann.

5 Ergebnisse und mögliche weitere Arbeiten

Der resultierende Codegenerator

Für diese Arbeit wurde ein Codegenerator implementiert, der ausgehend von dem Zwischenformat AIF Assemblercode für die Abacus-Prozessorarchitektur erzeugt. Probleme der Kausalität werden dabei erkannt. Wird ein Programm, das solche Probleme beinhaltet oder welches nicht korrekt ist, übersetzt, wird eine Fehlermeldung ausgegeben.
Für die einzelnen Zustände des Kontrollflussgraphen erzeugt der Compiler den gewünschten Code. Die Implementierung, bei welcher der Code der Zustände des Kontrollflussgraphen mit Hilfe einer Symboltabelle zusammengesetzt wird, wurde allerdings nicht fertiggestellt, da Sprünge, die ihr Ziel nicht erreichen können, Probleme bereitet haben. Daher existiert für das Zusammensetzen der Codeblöcke nur eine einfache Implementierung, welche das Problem zu kurzer Sprünge nicht löst. Bei dieser Implementierung wird die Liste der Codeblöcke in einer Endlosschleife durchlaufen und immer nur der Codeblock des aktuellen Zustandes ausgeführt.

Anmerkungen zur Implementierung

Bei der Codeerzeugung ist während der Implementierung das Problem aufgetreten, dass der erzeugte Code in Register geschrieben hat, deren Inhalt noch benötigt wurde. Wird beispielsweise für eine Aktion, bei der *rl* das Register für den linksseitigen Ausdruck ist und *ra* und *rb* die Register für den rechtsseitigen Ausdruck sind, Code erzeugt, so muss darauf geachtet werden, dass die Registerinhalte von *ra* und *rb* nicht verändert werden. Außerdem darf erst in *rl* geschrieben werden, wenn *ra* und *rb* nicht mehr gelesen werden müssen, da *rl* und z.B. *ra* dasselbe Register bezeichnen können.
Ein weiteres Problem ist das Ersetzen der Symbole vor der Übersetzung in Maschinencode. Da der Befehlssatz eines Abacus-Prozessors nur Sprünge relativ zum aktuellen Befehlszähler umsetzen kann und diese in ihrer Reichweite stark beschränkt sind, müssen, wie in Abschnitt 4.6.3 erläutert, Codeblöcke eingefügt werden, um zu weiter entfernten Stellen im Programm zu springen. Es kann vorkommen, dass mehrere Sprünge nicht direkt zu ihrem Ziel springen können. In diesem Fall kann das Einfügen eines Blocks für das Weiterspringen eines Sprungs dazu führen, dass das Ziel eines anderen Sprungs nicht mehr erreicht werden kann. Tritt dieses Problem auch andersherum auf, kann der Compiler in eine Endlosschleife geraten. Vor allem bei asynchron parallelen Programmen, deren Kontrollflussgraph sehr viele Zustände besitzt, besteht dieses Problem.

Mögliche weitere Arbeiten

In der vorgestellten Implementierung wurde das Graphfärben nur jeweils auf die bedingten Aktionen eines Zustandes des Kontrollflussgraphen angewendet. Es besteht jedoch die Möglichkeit, den Kontrollflussgraphen weiter zu analysieren und so mehr Informationen über

die Lebendigkeit von Variablen zu gewinnen [App02]. Während in der vorgestellten Implementierung bei einem Zustandsübergang alle Registerinhalte in den Speicher geschrieben werden müssen, könnten die Variablen so noch länger in den Registern gehalten werden.
Die Sprache Quartz besitzt inzwischen auch den Datentyp *real* (siehe Quartz Language Reference Card [Emb]). Für diesen Datentyp wurden Konvertierungen nach der Norm IEEE 754 [IEE85] implementiert. Bei der Codeerzeugung wurde jedoch nicht für alle Ausdrücke Code erzeugt.

Literaturverzeichnis

[Aho08] Alfred V. Aho. *Compiler*. Pearson Studium, München [u.a.], 2., aktualisierte aufl., german language ed. edition, 2008.

[App02] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge Univ. Press, Cambridge [u.a.], 2. ed. edition, 2002.

[BBS11] Y. Bai, J. Brandt, and K. Schneider. Data-flow analysis of extended finite state machines. In B. Caillaud, J. Carmona, and K. Hiraishi, editors, *Application of Concurrency to System Design (ACSD)*, pages 163–172, Newcastle Upon Tyne, England, UK, 2011. IEEE Computer Society.

[BS11] J. Brandt and K. Schneider. Separate translation of synchronous programs to guarded actions. Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, March 2011.

[Cha81] Gregory J. Chaitin. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[Dem94] Giovanni Demicheli. *Synthesis and optimization of digital circuits*. McGraw-Hill, New York, NY [u.a.], 1994.

[DTLS04] Jean-Marc Daveau, Thomas Thery, Thierry Lepley, and Miguel Santana. A retargetable register allocation framework for embedded processors. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 202–210, New York, NY, USA, 2004. ACM.

[Edw03] Stephen A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Trans. Des. Autom. Electron. Syst.*, 8(2):141–187, April 2003.

[Emb] University of Kaiserslautern Embedded Systems Group. The averest system. `http://www.averest.org/`. Accessed: 2013.12.28 `http://www.webcitation.org/6MD7dPdb0`, `http://www.webcitation.org/6MD7k3vQ3`, `http://www.webcitation.org/6MD7qBMrl`, `http://www.webcitation.org/6MD7toMP3`.

[ES14] ES. The abacus processor architecture. noch unveröffentlicht, 2014.

[IEE85] IEEE. Ieee standard for binary floating-point arithmetic, 1985. IEEE754.

[Mic] Microsoft. Fsharp dictionary. `http:http://msdn.microsoft.com/de-de/library/xfhwa508%28v=vs.85%29.aspx`. Accessed: 2013.12.28 `http://www.webcitation.org/6MD700sv1`.

[Sch09] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[SRH04] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm

for graph-coloring register allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 277–288, New York, NY, USA, 2004. ACM.

[WM97] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau*. Springer, Berlin [u.a.], 2., überarb. und erw. aufl. edition, 1997.