Diplomarbeit

Entwicklung eines generischen Codegenerators für RISC-Architekturen

Jörg Kamphausen

Diplomarbeit
am Lehrstuhl 12
des Fachbereichs Informatik
der Universität Dortmund

7. Januar 2004

Betreuer:

Dr. Markus Lorenz
Prof. Dr. Peter Marwedel

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Der zunehmende alltägliche Einsatz von mobilen Systemen hat zu neuen Anforderungen an die verwendete Hardware geführt. Erst wurden Systeme mit ausreichender Performance und geringem Energieverbrauch gefordert, wie z.B. ASICs (Application Specific Integrated Circuits), später zusätzlich flexiblere Systeme mit programmierbaren Komponenten, wie z.B. RISC- oder DSP-Prozessoren (DSPs = Digitale Signal Prozessoren). Durch die Ausweitung der Anwendungsbereiche in den letzten Jahren kam noch eine weitere Anforderung hinzu, die universelle Verwendbarkeit. So können moderne RISC-Prozessoren (RISC = Reduced Instruction Set Computer) in Handheld PCs, im KFZ-Bereich, in der Telekommunikation und auch im Multimedia-Bereich eingesetzt werden.

Mit diesen Prozessoren kommen neue Anforderungen auf die Softwareentwicklung zu. Um trotz der Komplexität heutiger Betriebssysteme und Anwendungsprogramme akzeptable Entwicklungszeiten zu erreichen, ist die Softwareentwicklung praktisch nur mit Programmier-Hochsprachen möglich. Neben der komfortableren Entwicklung durch komplexe Datenstrukturen, wie z.B. bei der Objekt-Orientierten Programmierung, besteht der Hauptvorteil in der großen Portabilität. Die verschiedenen RISC-Plattformen, wie z.B. ARM [Adv], LEON [Gai], MIPS [Mip], Motorola Dragonball [Mot], Intel XScale [Int], ermöglichen keine Portierung von Assembler- oder Binärcode untereinander. Bei Verwendung einer Programmier-Hochsprache wie z.B. C/C++ oder Java ist zur Portierung lediglich ein *Compiler* für den jeweiligen Prozessor nötig. „Ein Compiler ist ein Programm, das ein in einer bestimmten Sprache - der Quellsprache - geschriebenes Programm liest und es in ein äquivalentes Programm einer anderen Sprache - der Zielsprache - übersetzt“[Tei97].

Für die Übersetzung der Hochsprachen-Quellprogramme in Maschinencode sind leistungsfähige Compiler nötig, deren Erstellung ein nicht unbedeutender Kostenfaktor bei der (Weiter-) Entwicklung von RISC-Architekturen ist. Ein Teil des Compilers ist der *Codegenerator*, der das Quellprogramm aus der prozessorunabhängigen, compiler-internen Darstellung in das lauffähige Maschinenprogramm überführt. Häufig wird für einen neuen Prozessor ein komplett neuer Codegenerator entwickelt, was jedoch trotz der Gemeinsamkeiten von RISC-Plattformen mit einem großen Overhead verbunden ist. Eine mögliche Lösung zur Reduzierung dieses Aufwands sind sogenannte *retargierbare Compiler*, d.h. Compiler die mehrere Plattformen unterstützen und leicht an weitere adaptierbar sind.

1.1 Motivation

Aufgrund der unterschiedlichen RISC-Architekturen gibt es einen Bedarf an generischen Codegeneratoren bzw. an einzelnen Compiler-Techniken, die für verschiedene Prozessoren eingesetzt werden können. Zur maschinenunabhängigen Betrachtung des Quellprogramms dient die High-Level-Zwischendarstellung (IR = Intermediate Representation), die keine architekturspezifischen Details enthält. Werden maschinenspezifische Informationen hinzugefügt, spricht man von der Low-Level-Zwischendarstellung (LIR = Low-Level Intermediate Representation). Sie ist die Basis für architekturspezifische Optimierungen und für die Synthese des Zielprogramms, der Codegenerierung. Herkömmliche Compiler verwenden keine standardisierte LIR, so dass die einzelnen Compiler-Techniken für unterschiedliche Prozessoren jeweils neu entwickelt oder angepaßt werden müssen. Eine universelle LIR würde dagegen deren Wiederverwertbarkeit ermöglichen.

Am Lehrstuhl XII des FB Informatik der Universität Dortmund wurde die Compiler-Zwischendarstellung GeLIR (Generic Low-Level Intermediate Representation) entwickelt [GeL]. Sie vereint die IR und LIR, indem sie einerseits das Quellprogramm maschinenunabhängig, aber auch die alternativen Maschinenprogramme darstellen kann. Die prozessorspezifischen Merkmale werden davon getrennt abgelegt, so dass sie ohne Veränderung der Compiler-Techniken ausgetauscht werden können. GeLIR ist als Grundlage für die Entwicklung retargierbarer Compiler geeignet. Wie in Abb. 1.1 zu sehen ist, ermöglicht GeLIR retargierbare Compiler für eine ganze Architekturklasse, wobei die einzelnen Architekturspezifikationen von der generischen Codegenerierung getrennt sind.

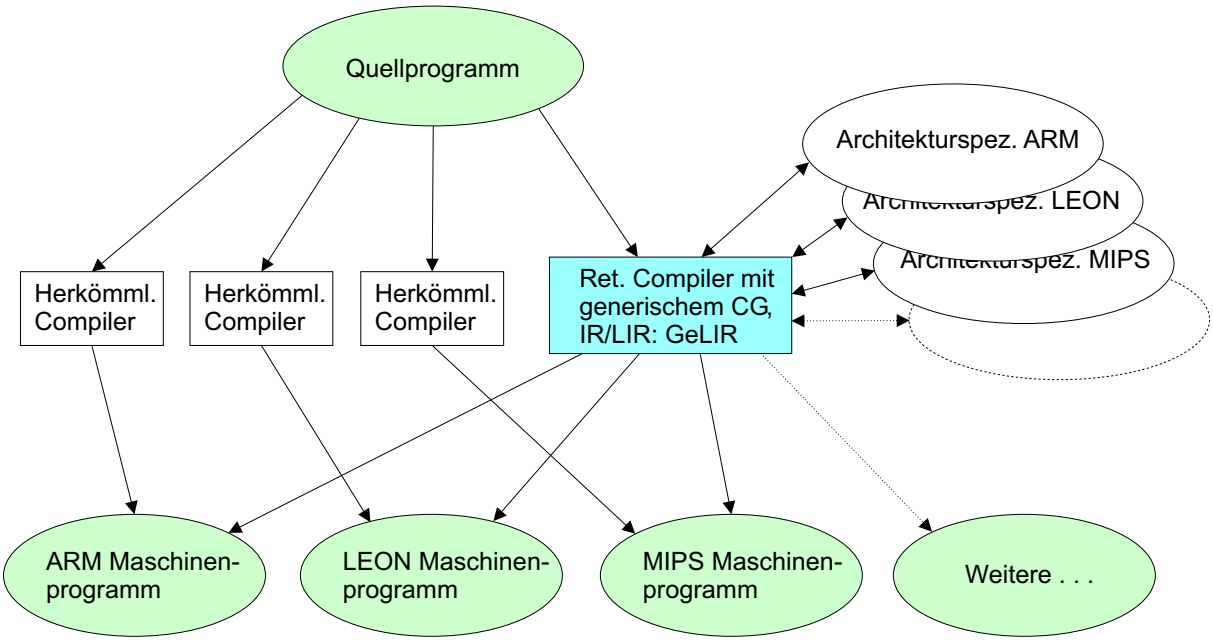

Abb. 1.1: Verwendung eines retargierbaren Compilers mit mehreren Architekturspezifikationen

1.2 Zielsetzung

Die Zielsetzung dieser Arbeit besteht in der Entwicklung eines generischen Compilers für RISC-Architekturen. Dieser enthält einen Codegenerator, der die notwendigen Hardware-Details eine Prozessors der Architekturspezifikation entnimmt. Der Codegenerator muss zumindest die Codegenerierungs-Phasen *Codeselektion* (CS) und *Registerallokation* (RA) enthalten. Bei der CS wird für jede abstrakte Anweisung der maschinenunabhängigen IR eine prozessorspezifische Operation ausgewählt, bei der RA werden die Variablen an konkrete Register gebunden.

Im einzelnen ergeben sich daraus folgende Ziele für diese Diplomarbeit:

1. Das Hauptziel ist die Entwicklung eines generischen Codegenerators für RISC-Architekturen

   RISC-Architekturen zeichnen sich durch ähnliche Eigenschaften aus. Sie sind Load-/Store-Architekturen und haben einen homogenen Registersatz, d.h. Speicherzugriffe sind nur über die Load-/Store-Befehle möglich und die Register des Prozessors sind in ihrer Verwendung gleichwertig. Aufgrund der Gemeinsamkeiten ist die Codegenerierung durch generische Compiler-Techniken möglich, die einzelnen Details für den jeweiligen Prozessor werden aus der Architekturbeschreibung entnommen. Die Basis der generischen Codegenerierung bildet die Compiler-Zwischendarstellung GeLIR, die als IR und LIR des Codegenerators dient. Um das Quellprogramm in ein äquivalentes Maschinenprogramm der jeweiligen RISC-Architektur zu überführen, müssen die dafür notwendigen Codegenerierungs-Techniken implementiert werden. Erforderlich sind die Codeselektion und die Registerallokation (RA), wobei die RA aus einem globalen (Basisblock-übergreifenden) und einem lokalen (Basisblock-internen) Verfahren besteht. Die Instruktionsanordnung ist dagegen nicht zwingend notwendig, da bereits durch das Einlesen des Quellpogramms in die GeLIR-Programmdarstellung eine gültige Ausführungsreihenfolge der Anweisungen vorliegt. Sie kann aber leicht in den Codegenerator integriert werden. Der Schwerpunkt bei der Entwicklung der einzelnen Compiler-Techniken liegt vor allem auf der Wiederverwertbarkeit für andere Compiler. Eine gute Codequalität ist dabei zweitrangig, aber dennoch wünschenswert. Sie kann nachträglich z.B. durch das Einfügen zusätzlicher Optimierungen oder den Austausch bereits vorhandener Techniken noch verbessert werden.

   Die Demonstration der generischen Codegenerierung erfolgt an zwei verschiedenen RISC-Maschinen: der ARM7- [Adv] und der LEON-Architektur [Gai], für die jeweils eine Architekturspezifikation erstellt wird. Diese beiden Prozessoren repräsentieren aufgrund unterschiedlicher Eigenschaften auch viele andere RISC-Architekturen. Die Ausgabe des Zielprogramms erfolgt in ausführbaren Assemblercode der jeweiligen Maschine.

   GeLIR beschränkt die Wiederverwendbarkeit von generischen Compiler-Techniken nicht auf Prozessoren derselben Architekturklasse. Der generische Codegenerator ist zwar an RISC-Architekturen angepasst, allerdings sind die einzelnen Techniken maschinenunabhängig realisiert. Dazu gehört u.a. die globale RA, die auch in Compilern für andere Architekturklassen verwendet werden kann. Sie wird z.B. von

einem Compiler für den M5-DSP benutzt, einer in Entwicklung befindlichen Nachfolgearchitektur des M3-DSP [FWD+98, WFL+99]. Bei der Implementierung des generischen Codegenerators wurden Verfahren von vier auf GeLIR basierenden Arbeiten benutzt:

- *Performance- und energieeffiziente Compilierung für digitale SIMD-Signalprozessoren mittels genetischer Algorithmen* [Lor03]:
  In der Arbeit wurde ein genetischer Codegenerator für den M3-DSP entwickelt. Für diese Architektur wird bei einer getrennten Durchführung der Codegenerierungs-Phasen potentiell ineffizienter Code erzeugt, so dass sie simultan gelöst werden. Einige in der Arbeit beschriebene Techniken dienen ebenfalls als Grundlage des in dieser Arbeit entwickelten generischen Codegenerators.
- *XML-basierte generische Zwischendarstellung für Compiler* [Fie01]:
  Um bei der Entwicklung auf GeLIR basierender Techniken Zwischenzustände zu reproduzieren, ist ein standardisiertes Speicherformat nötig. Die Hauptanforderung daran ist eine gute Strukturierbarkeit, damit der gespeicherte Zustand leicht wieder einzulesen ist, was z.B. XML (eXtensible Markup Language) erfüllt. Die daraus resultierende Zwischendarstellung wird als *XeLIR* bezeichnet. Die Bearbeitung kann mittels eines Texteditors oder durch Verwendung einer API (Application Programming Interface) erfolgen. Das XeLIR-Austauschformat eignet sich u.a. für Peephole-Optimierungen und Assemblercode-Ausgabe mit Mustern (Pattern), welche Vorschriften zur Instruktionsmanipulation bzw. zur Codeausgabe enthalten. Auf diese Weise erfolgt die Assemblercode-Ausgabe des in dieser Diplomarbeit entwickelten Compilers.
- *Schleifenoptimierungen zur Ausnutzung paralleler Rechenwerke von Prozessoren der M3-DSP-Plattform* [Hor01b]:
  Da die Prozessoren der M3-DSP-Plattform für die Verarbeitung großer Datenmengen in Schleifen entworfen wurden, besteht bei Schleifenoptimierungen ein großes Potential zur Verbesserung der Codequalität. In der Arbeit wurden dazu zwei Optimierungen entwickelt: die *Vektorisierung* von Speicherzugriffen und Operationen, und die Unterstützung von *Zero Overhead Hardware Loops*. Die Grundlage dafür bilden mehrere Analysemethoden, z.B. Abhängigkeitsanalysen und die Erkennung von Schleifen. Einige davon werden in dieser Diplomarbeit zur Verbesserung der Codequalität wiederverwendet.
- *Generische Low-Level Optimierungen für RISC-Architekturen* [Hor01a]:
  In [Hor01a] wurden einige Standard-Optimierungen für den ARM7-Prozessor implementiert, wie z.B. *Redundant Load/Store Elimination* oder *Constant Propagation/ Folding*. Es handelt sich um maschinenunabhängige Optimierungen, die auch maschinenspezifisch auf dem generierten Assemblercode durchgeführt werden können. Um sie generisch zu halten, erfolgte keine Bindung an Details der Architekturbeschreibung, was die Verwendung ihm Rahmen dieser Diplomarbeit ermöglicht.

In Abb. 1.2 wird der Aufbau des in dieser Arbeit entwickelten Compilers gezeigt. Das C-Quellprogramm wird eingelesen und in die GeLIR-Darstellung überführt. Auf dieser werden einige Optimierungen und die Codegenerierung durchgeführt.

Der Codegenerator enthält eine Codeselektion und eine Registerallokation. Da es sich um generische Techniken handelt, werden sämtliche prozessorspezifischen Details aus der Architekturdarstellung entnommen. Die GeLIR-Datenstruktur kann jederzeit während der Durchführung einer Optimierung oder der Codegenerierung im XeLIR-Format gespeichert werden. Nach der Codegenerierung erfolgt die Ausgabe des Maschinenprogramms als Assemblercode.

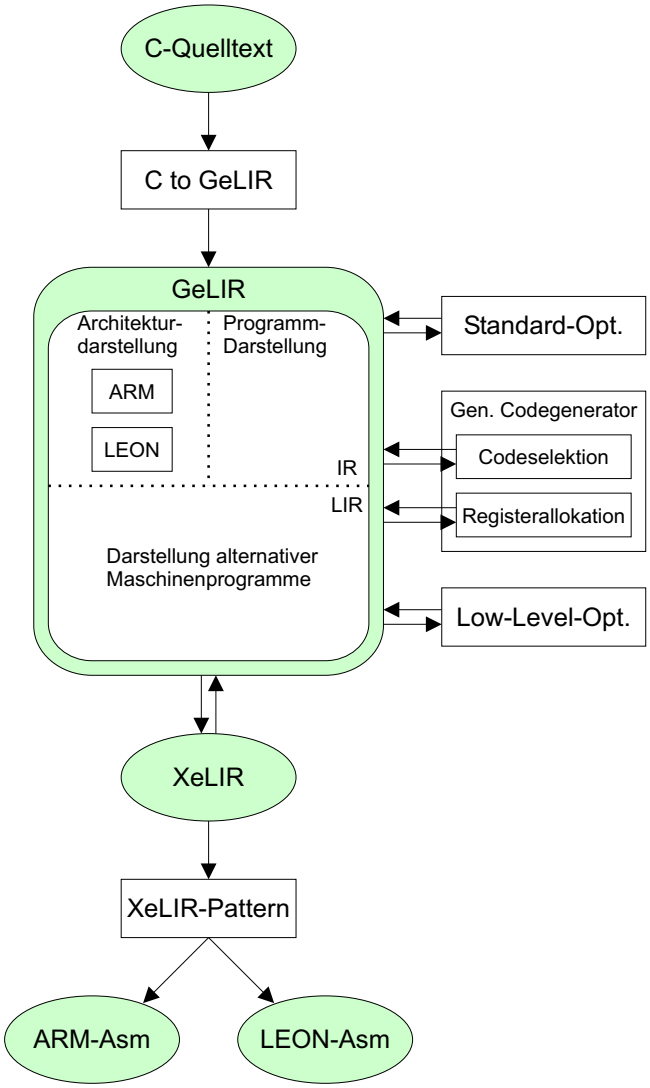

Abb. 1.2: Aufbau des Compilers

2. Erweiterung der Architekturbeschreibung von GeLIR

   Mit dem ARM7-Prozessor [Hor01a], dem M3-DSP [Lor03] und dem M5-DSP gibt es bereits mehrere GeLIR-Architekturbeschreibungen. Bisher wurden diese teilweise mit maschinenspezifischen Compiler-Techniken verwendet, so dass in der Architekturspezifikation nicht zwangsweise alle für die generische Codegenerierung erforderlichen Informationen enthalten sein müssen. Dies betrifft vor allem die Programmier-Konventionen eines Prozessors, z.B. die Verwendung der Register zur Übergabe von Funktions-Parametern.

3. Vergleich mit bestehenden Compilern

   Zur Beurteilung der erzielten Codequalität des Codegenerators wird ein Vergleich mit bereits vorhandenen Compilern durchgeführt. Dabei handelt es sich um den ENCC (Energy Aware C Compiler) des LS XII [Enc], der Code für den ARM7- und LEON-Prozessor generiert, und um jeweils einen weiteren Compiler. Die Betrachtung des ENCC ist besonders interessant, weil das gleiche Front-End wie bei dem Compiler dieser Diplomarbeit verwendet wird. Das Quellprogramm wird von beiden Compilern der LANCE-IR [Lan] entnommen, wodurch der Aufbau des Zielprogramms und der generierte Code eine gewisse Ähnlichkeit haben. Der Codegenerator und die einzelnen entwickelten Techniken lassen sich so besser vergleichen als mit anderen Compilern. Beim ENCC handelt es sich auch um einen retargierbaren Compiler, allerdings basiert er nicht auf generischen Techniken, so dass die Adaption an weitere RISC-Architekturen einen erheblich größeren Aufwand erfordert, als bei dem Compiler dieser Diplomarbeit.

1.3 Übersicht

Im nachfolgenden Kapitel werden zunächst die für die Entwicklung des Codegenerators nötigen Grundlagen zusammengefasst. Diese bestehen aus drei Teilen, einer Beschreibung des ARM7- und des LEON-Prozessors sowie des Aufbaus vom Compilern und der GeLIR-Zwischendarstellung.

Der Aufbau des generischen Codegenerators wird danach detailliert in Kapitel 3 beschrieben. Der Schwerpunkt liegt dabei auf den für diese Arbeit entwickelten Techniken, der Codeselektion und der Registerallokation. Dazu werden bereits existierende Techniken vorgestellt und die implementierten ausführlich erklärt.

Die Qualität des erzeugten Codes wird durch die Betrachtung von Benchmark-Ergebnissen analysiert und beurteilt. In Kapitel 4 erfolgt dazu für den ARM7- und den LEON-Prozessor ein Vergleich mit jeweils zwei Compilern.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen geschaffen, die für das Verständnis der Aufgabe, den Aufbau und die Funktionsweise des entwickelten Codegenerators erforderlich sind. Zuerst werden in Abschnitt 2.1 RISC-Architekturen betrachtet, für die der Codegenerator entwickelt wurde. In Abschnitt 2.2 wird der Aufbau und die Funktion von modernen Compilern beschrieben um die einzelnen Aufgaben des Codegenerators abzugrenzen. In Abschnitt 2.3 wird dann GeLIR beschrieben, die Compiler-Zwischendarstellung, auf der der Codegenerator basiert.

2.1 RISC-Architekturen

RISC-Architekturen (RISC = Reduced Instruction Set Computer) haben im Gegensatz zu CISC-Architekturen (CISC = Complex Instruction Set Computer) einen reduzierten Befehlssatz und sind für einen höheren Instruktionsdurchsatz entworfen [Mar03]. Es handelt sich dabei um Load/Store-Architekturen, die einen CPI-Wert (CPI = Cycles Per Instruction) nahe eins haben. Speicherzugriffe erfolgen nur über Load/Store-Befehle, arithmetisch-logische Befehle arbeiten dagegen nur auf den universellen Registern (GPRs = General Purpose Register). Die Instruktionen eines RISC-Prozessors haben eine einheitliche Befehlswortlänge. Programm-Optimierungen finden i.d.R. nur durch den Compiler statt.

Die entwickelten Codegenerierungs-Techniken sind allgemein für RISC-Architekturen ausgelegt. Zur Demonstration der Anwendbarkeit der entwickelten Techniken werden in dieser Arbeit zwei konkrete RISC-Architekturen betrachtet. Dies sind:

- der ARM7TDMI-Prozessor
- der LEON-Prozessor

Ursprünglich stand zur weiteren Auswahl noch die MIPS-Architektur [Mip]. Für die Entscheidung ausschlaggebend war, dass der ARM- und der LEON-Prozessor aufgrund ihrer unterschiedlichen Registerorganisation auch viele weitere RISC-Architekturen repräsentieren. Ein weiterer Grund war die Existenz des ENCC (Energy Aware C Compiler) [Enc] des LS XII, der Code für die ARM7- und die LEON-Prozessoren erstellt. Aufgrund einiger

Gemeinsamkeiten mit dem in dieser Diplomarbeit entwickelten Compiler lassen sich die erzeugten Maschinenprogramme besonders gut vergleichen (siehe Kapitel 4).

Bei der Beschreibung dieser RISC-Architekturen liegt der Schwerpunkt auf den für die Softwareentwicklung wichtigen Aspekten. Wichtig sind dabei neben dem Befehlssatz und der Registerorganisation vor allem die Möglichkeiten bzw. Standards für die Realisierung von Funktionsaufrufen. Einzelne Hardware-Details sind nur aufgenommen soweit sie für das Verständnis nachfolgender Kapitel nötig sind.

2.1.1 ARM7TDMI

Der ARM7TDMI-Prozessor gehört zu der ARM7-Familie von Advanced RISC Machine Ltd. [Adv]. Er ist die Basis-Ausführung von insgesamt vier verschiedenen 32-Bit Low-Power-RISC-Prozessoren. Im Gegensatz zu den erweiterten Versionen hat er nur einen Integerkern, ohne Erweiterungen wie z.B. Java-Unterstützung oder einer speziellen Speicherverwaltungseinheit (MMU = Memory Managemanent Unit). Verwendung findet er z.B. in mobilen Kommunikationssystemen und PDAs.

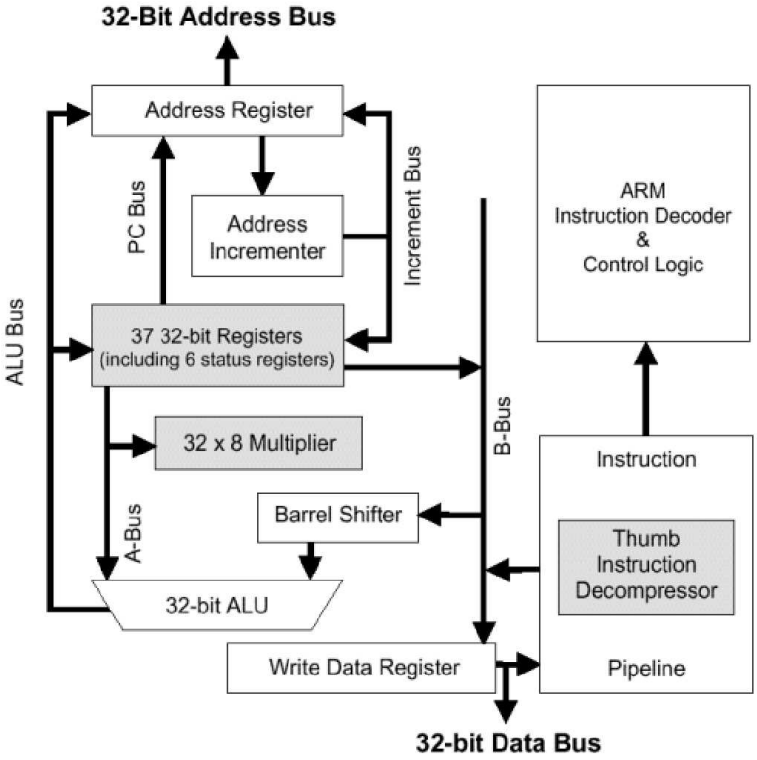

Abb. 2.1: Aufbau des ARM7-Prozessors [Adv99]

Details zur ARM7-Architektur, die über die nötigen Grundlagen zum Bau eines Codegenerators hinausgehen, können [Adv99] und [Adv95b] entnommen werden. Abb. 2.1 gibt einen Überlick über den Aufbau eines ARM7-Prozesors. Die wichtigsten Merkmale des ARM7 sind:

- drei 32-Bit-Funktionseinheiten:

  - ALU (Arithmetic Logical Uni)
  - Barrel-Shifter
  - Multiplizierereinheit

- zwei verschiedene Befehlssätze
  - 32-Bit-ARM-Instruktionssatz
  - 16-Bit-THUMB-Instruktionssatz

- 37 32-Bit-Register
- gemeinsamer Daten- und Programmspeicher mit 32-Bit-Adressbus
- 3-stufige Instruktionspipeline

Befehlssatz

Der ARM7 hat zwei verschiedene Betriebsmodi, den ARM und den THUMB-Modus, zwischen denen im laufenden Betrieb durch spezielle Befehle gewechselt werden kann. Im ARM-Modus stehen alle 80 32-Bit-Instruktionen zur Verfügung. Der THUMB-Modus ist der Low-Power-Modus des ARM7, in dem die Befehle nur durch 16-Bit große Wörter kodiert werden. Dadurch ergibt sich wesentlich kompakterer Code, der durchschnittlich nur 65% der Codegröße im Vergleich zum ARM-Modus benötigt. Durch die THUMB-Logik im Prozessor werden die 16-Bit-Befehle zur Laufzeit in ihre äquivalenten 32-Bit-Befehle umgewandelt. Die 16-Bit breite Codierung beschränkt den Befehlssatz auf die 36 wichtigsten Befehle, die aber auf die 32-Bit-Register zugreifen. Dabei ist die Register- und Speicher-Adressierung, sowie der Wertebereich von Immediates im Vergleich zum ARM-Modus eingeschränkt [Adv95a].

Da die Entwicklung von Low-Power-Compilern ein Forschungsschwerpunkt des LS XII ist, wird in dieser Arbeit nur der THUMB-Modus betrachtet. Auf diesen Modus bezieht sich nachfolgend die Bezeichnung *ARM*. Die Befehle des THUMB-Modus lassen sich in drei Gruppen einteilen:

1. Registertransfer und ALU-Befehle

   Diese Befehle werden nur auf den Registern des ARM7-Prozessors ausgeführt. Dazu gehören alle Anweisungen für arithmetische & logische Operationen sowie Move-Befehle. Durch die Einschränkungen im THUMB-Modus steht kein Divisions-Befehl zur Verfügung und es sind auch keine Fließkomma-Operationen möglich.

2. Branch-Befehle

   Bedingt durch die 16-Bit Befehlswortgröße sind Sprünge, die als eine einzige Instruktion kodiert sind, begrenzt. Die Sprungweite beträgt bei bedingten Sprüngen maximal 256 Byte, bei unbedingten maximal zwei KBytes. Funktionsaufrufe werden durch zwei Befehlswörter kodiert und ermöglichen einen Sprung über vier MBytes.

3. Load/Store, Stack-Befehle

   Speicherzugriffe erfolgen ausschließlich über diese Befehle. Die Adressierung erfolgt dabei entweder absolut, Befehlszähler-relativ oder Stack-Pointer-relativ. Im Gegensatz zu den Registern, die alle 32-Bit breit sind, sind für den Speicher fünf verschiedene Datentypen vorgesehen:

   - Byte (8-Bit), signed
   - Byte (8-Bit), unsigned
   - Halbwort (16-Bit), signed
   - Halbwort (16-Bit), unsigned
   - Wort (32-Bit)

   Diese werden durch entsprechende Load/Store-Befehle unterstützt.

Registerorganisation

Der ARM7-Prozessor hat insgesamt 37 Register, deren Verwendung vom Betriebs-Modus abhängt. Für die Codegenerierung von Anwendungsprogrammen müssen davon folgende berücksichtigt werden (siehe auch Abb. 2.2):

- ARM-Modus
  - R0-R12 frei verwendbar, GPRs
  - R13 Stack Pointer (SP)
  - R14 Link Register (LR)
  - R15 Befehlszähler (PC = Program Counter)
  - CPSR (Current Program Status Register), enthält Flags für die Entscheidung bei Sprungbefehlen
  - SPSR (Saved Prozessor Status Register)
- THUMB-Modus
  - R0-R7 Low-Registers, frei verwendbar, GPRs
  - R8-R12 High-Registers, nur eingeschränkte Verwendung möglich
  - R13 Stack Pointer (SP)
  - R14 Link Register (LR)
  - R15 Befehlszähler (PC)
  - CPSR
  - SPSR

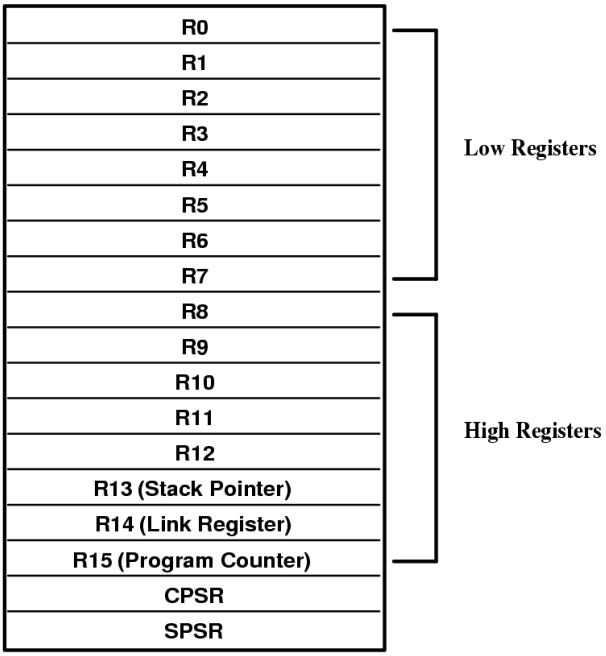

Abb. 2.2: Register des ARM7 im THUMB-Modus [Adv99]

Konventionen für Funktionsaufrufe

Im Hinblick auf die Erzeugung lauffähigen Maschinencodes in der Codegenerierung müssen einige Konventionen für die Verwendung der Register bei Funktionsaufrufen beachtet werden [Adv99]. Dies gilt vor allem für die Benutzung von externen Funktionen, z.B. aus Bibliotheken.

Parameterübergabe: Einer Funktion können beim Aufruf mehrere Werte (Funktions-Parameter) übergeben werden. Laut Konvention des ARM7 werden die ersten vier Parameter bei einem Funktionsaufruf in den ersten vier Low-Registern (r0-r3) übergeben, weitere Parameter müssen auf dem Stack abgelegt werden. Der Rückgabewert einer Funktion wird in Register r0 übergeben.

Register-Sicherung: Um das Überschreiben noch benötigter Werte in Registern durch die aufgerufene Funktion zu verhindern, ist deren Sicherung erforderlich, i.d.R. durch Kopieren auf den Stack.

Laut Konvention darf die aufgerufene Funktion die Inhalte der Register r0-r3 überschreiben, d.h. falls diese Register später noch verwendete Werte enthalten, muss die aufrufende Funktion diese sichern. Die Inhalte der restlichen Register (Ausnahme: Befehlszähler) müssen nach Beendigung der Funktion den gleichen Inhalt haben wie zu Beginn, d.h. die aufgerufene Funktion ist für deren Sicherung verantwortlich.

Verwendung der High-Register

Der Zugriff auf High-Register ist nur über drei verschiedene Instruktionen möglich:

- Die MOV-Instruktion überträgt Werte von Low- in High-Register bzw. umgekehrt von High- in Low-Register.

- Die ADD-Instruktion kann den Wert eines Low-Registers zu dem eines High-Registers addieren bzw. umgekehrt den Wert eines High-Registers zu dem eines Low-Registers.

- Die CMP-Instruktion kann die Werte aus Low- mit denen aus High-Registern vergleichen.

Die High-Register sind deshalb nur sehr eingeschränkt verwendbar und können bei der Codegenerierung nicht gleichwertig mit den GPRs betrachtet werden.

2.1.2 Leon

Der 32-Bit-RISC-LEON-Prozessor wurde von Jiri Gaisler bei der ESA (European Space Agency) entwickelt. Das Ziel war die Verwendung bei Weltraummissionen, weshalb auch eine „fault-tolerant“-Version entstand, der LEON-FT. Die Weiterentwicklungen erfolgen von der Firma Geisler Research [Gai]. Das Prozessormodell des LEON liegt in VHDL (Very High Speed Integrated Circuit Hardware Description Language) vor und unterliegt der GPL[1] und LGPL[2]. Der Kern kann frei genutzt und modifiziert werden, solange die Änderungen veröffentlicht werden. Seit der LEON-1 Version 2.3 ist der Prozessor SPARC V8 (Scaleable Processor ARChitecture Version 8) konform.

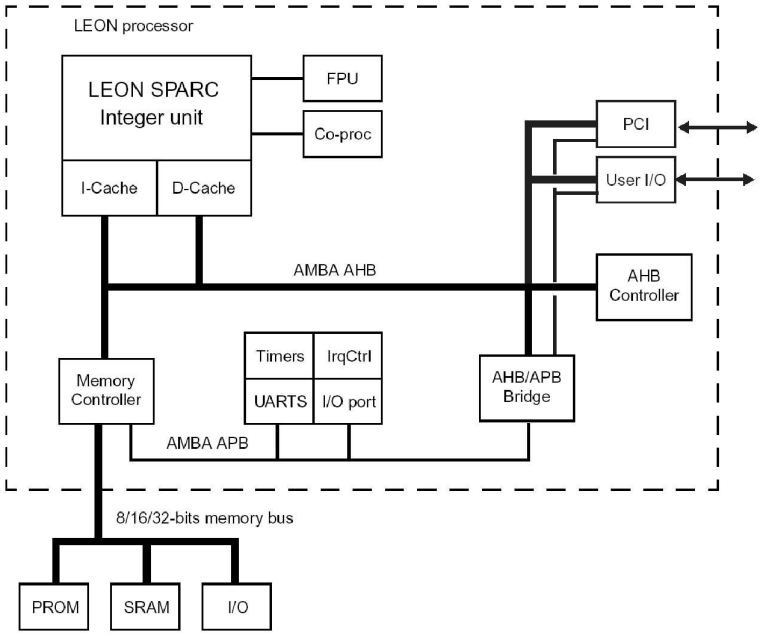

Abb. 2.3: LEON-Blockschaltbild [Gai01]

[1]GNU Public License, http://www.gnu.org/copyleft/gpl.html
[2]Lesser GNU Public License, http://www.gnu.org/copyleft/gpl.html

Details zum LEON-Prozessor, die über die nötigen Grundlagen zum Bau eines Codegenerators hinausgehen, können der Beschreibung des SPARC V8-Standards [Spa99] entnommen werden. Der Aufbau des LEON-Prozessors ist in Abb. 2.3 zu sehen. Die wichtigsten Merkmale des Prozessors sind:

- drei 32-Bit-Funktionseinheiten
  - Integer-Einheit
  - Fließkomma-Einheit
  - Koprozessor
- 32-Bit Befehlssatz
- Registerfenstertechnologie
- getrennte Caches für Instruktionen und Daten
- 5-stufige Instruktionspipeline

32-Bit-Befehlssatz

Beim LEON sind alle Instruktionen 32-Bit groß und an 32-Bit Adressen im Speicher ausgerichtet. Laut SPARC V8-Standard werden die Befehle in sechs Kategorien eingeteilt:

- Load/Store

  Dies sind die einzigen Befehle die auf den Speicher zugreifen. Sie haben zwei Operanden, entweder zwei Register oder ein Register und einen Immediate-Wert, durch deren Addition die Adresse des Speicherzugriffs ermittelt wird. Die Load/Store-Befehle unterstützen sechs verschiedene Datentypen im Speicher:

  - Byte (8-Bit), signed
  - Byte (8-Bit), unsigned
  - Halbwort (16-Bit), signed
  - Halbwort (16-Bit), unsigned
  - Wort (32-Bit)
  - Doppel-Wort (64-Bit)

  Bei einem 64-Bit Zugriff wird statt eines einzelnen Registers ein Register-Paar angesprochen.

- Arithmetisch/logisch

  Diese Befehle führen arithmetische und logische Operationen auf den 32-Bit-Registern aus.

- Kontrollfluss

  Bedingte Sprungbefehle sind auf eine Sprungweite von 8 MBytes begrenzt. Der CALL-Befehl unterliegt dagegen keiner Beschränkung und ermöglicht Funktionsaufrufe im gesamten 32-Bit-Adressraum. Die Instruktionen werden in einer Pipeline abgearbeitet, die aus fünf Stufen besteht:

  1. Instruction Fetch: Die Instruktion wird aus dem Instruktions-Cache oder falls dort nicht vorhanden, durch den Memory Controller in die Pipeline geholt.
  2. Decode: Die Instruktion wird dekodiert. Dabei werden die Operanden der Instruktion ausgelesen und Call/Sprung-Adressen erzeugt.
  3. Execute: ALU-Operationen werden ausgeführt, für Speicheroperationen und Sprungbefehle erfolgt die Adressberechnung.
  4. Memory: Falls nötig wird auf den Datencache oder den Speicher zugegriffen.
  5. Write: Das Ergebnis einer in der Execute-Stufe ausgeführten Operation wird, falls nötig, in ein Register geschrieben.

  Wenn eine Sprungadresse berechnet wird, ist der nachfolgende Befehl bereits in der Pipeline und wird ausgeführt. Da dies unabhängig davon geschieht, ob der Sprung erfolgt oder nicht, spricht man von einem *Branch Delay Slot* der Länge eins. Dies muss während Codegenerierung, vor allem bei bedingten Sprüngen, beachtet werden. Eine einfache Lösung des Problems ist das Einfügen des NOP-Befehls (NOP = No Operation) direkt nach allen Sprunganweisungen, da dieser den Prozessorstatus nicht verändert.

- Status-Register-Zugriff

  Diese Befehle dienen dem Lesen und Schreiben verschiedener Zustands- und implementierungsabhängiger Prozessor-Register.

- Fließkomma-Befehle

  Diese Befehle führen Berechnungen in der Fließkomma-Einheit durch.

- Koprozessor-Befehle

  Diese Befehle führen Berechnungen im Koprozessor durch.

Registerorganisation

Der SPARC V8 Standard erlaubt eine Implementierung von 40 bis 520 gleichwertigen, frei verwendbaren 32-Bit Registern. Acht Register sind global verwendbar, auf die anderen wird per Registerfenster-Technologie zugegriffen. Bei der Betrachtung eines einzelnen Registerfensters stehen insgesamt 32 Register zur Verfügung:

- 8 *Global*-Register r[0]-r[7], dienen der globalen Verwendung
- 8 *Out*-Register r[8]-r[15], dienen zum Werteaustausch mit aufgerufenen Funktionen
- 8 *Local*-Register r[16]-r[23], dienen der lokalen Verwendung

- 8 *In*-Register r[24]-r[31], dienen zum Werteaustausch mit der aufrufenden Funktion

Ein Register-*Set* besteht aus acht *In*-Registern und acht *Local*-Registern und wird durch das CWP-Register (CWP = Current Window Pointer) adressiert. Der in dieser Arbeit betrachtete LEON-Prozessor hat die default-Anzahl von 16 Register-*Sets*, d.h. die Gesamtzahl der Register beträgt:

8 *Global*- + 16 * 16 *In/Local*-Register = 264 universelle Register

Ein Register-*Fenster* besteht aus den 16 Registern des aktuell betrachteten *Sets* (= window(CWP)) und den acht *In*-Registern des nächsten *Sets* (= window(CWP-1)), die als *Out*-Register adressiert werden, siehe Abb. 2.4.

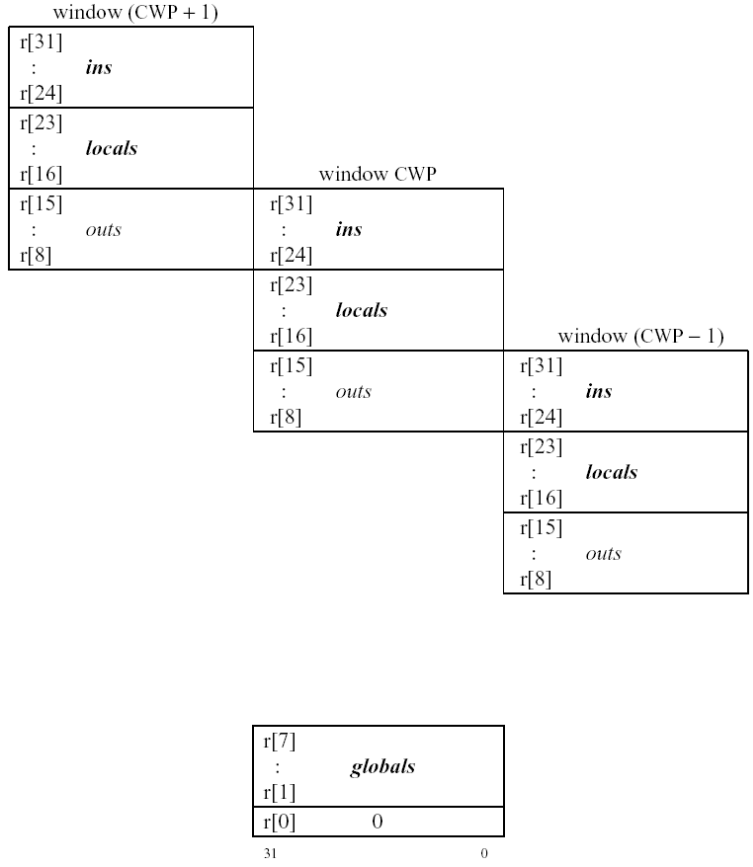

Abb. 2.4: Registerfenster-Organisation des SPARC V8- Standards [Spa99]

Einige Register haben spezielle Funktionen:

- r[0]: Enthält immer den Wert 0. Ein in dies Register geschriebener Wert wird verworfen.
- r[15]: Der CALL-Befehl schreibt seine eigene Adresse für den Rücksprung in dieses Register.
- r[17], r[18]: Falls ein Trap[3] auftritt, wird der CWP um eins erhöht und in diesen beiden Registern werden die aktuellen Programmzähler abgelegt.

[3]eine Programmunterbrechung, auslösbar durch ausgeführte Instruktionen oder externe Ereignisse

Konventionen für Funktionsaufrufe

Wie beim ARM7 gibt es auch beim SPARC V8 Standards zur Verwendung der Register bei Funktionsaufrufen [Spa99].

Parameterübergabe: Die Parameterübergabe für Funktionen wird durch die Registerfenster-Technologie sehr vereinfacht. Die zu übergebenden Parameter werden von der aufrufenden Funktion in den ersten sechs *Out*-Registern (r[8]-r[13]) abgelegt. In der aufgerufenen Funktion wird als erstes das Registerfenster durch den Save-Befehl um ein Set verschoben, so dass die Parameter in den ersten sechs *In*-Registern (r[24]-r[29]) liegen und acht unbelegte *Local*-Register zur Verfügung stehen. Bei der Rückgabe des Funktionswerts wird genau umgekehrt vorgegangen: Der zu übergebende Wert wird im ersten *In*-Register (r[24]) abgelegt und ist nach Zurückschieben des Registerfensters durch den Restore-Befehl und Beendigung der Funktion im ersten *Out*-Register (r[8]) auslesbar, siehe auch Abb. 2.4.

Register-Sicherung: Die Inhalte der acht *Global*-Register müssen, falls sie später benötigte Werte enthalten, von der aufrufenden Funktion gesichert werden, da für diese Register keine Vorschrift existiert. Die Werte in den *Out*-Registern müssen ebenfalls gesichert werden.

2.2 Compiler

Die Aufgabe des in dieser Diplomarbeit entwickelten Compilers ist die Übersetzung des Quellprogramms von der Programmier-Hochsprache ANSI-C in den Assemblercode der jeweils gewählten RISC-Zielarchitektur. In diesem Abschnitt wird zuerst die Compiler-Zwischendarstellung (IR = Intermediate Representation) beschrieben und dann der Aufbau eines Compilers.

2.2.1 Compiler-Zwischendarstellungen

Zur Compiler-internen Programmdarstellung werden verschiedene graphische Darstellungen verwendet. Die für diese Arbeit wichtigsten sind der Kontrollfluss- (CFG), der Datenfluss- (DFG) und der Kontrolldatenflussgraph (CDFG). Im folgenden wird die Notation aus [Lor03] und [Bas01] benutzt.

Definition 2.1 (Kontrollflussgraph) *Ein Kontrollflussgraph (CFG) ist ein gerichteter Graph G = (V,E), dessen Knoten $v \in V$ entsprechend des potentiell möglichen Kontrollflusses über Kanten $e_{i,j} = (v_i, v_j) \in E \subseteq V \times V$ miteinander verbunden werden. Die Knoten selbst enthalten sequentiell auszuführende Anweisungen und können aufgrund von Verzweigungen des Kontrollflusses mehrere Nachfolger haben.*

Um den CFG zu vereinfachen wird nachfolgend davon ausgegangen, dass die Knoten jeweils die maximale Anzahl von sequentiell ausführbaren Anweisungen enthalten. Sie werden dann auch *Basisblöcke* genannt.

Definition 2.2 (Basisblock) *Ein Basisblock (BB) stellt eine maximale Sequenz von Anweisungen dar, bei der sich der Kontrollfluss nur nach der letzten Anweisung der Sequenz aufteilen und nur bei der ersten Anweisung der Sequenz wieder zusammenfließen kann.*

Die Ausführungsreihenfolge der Basisblöcke hängt von dem Verlauf des Kontrollflusses ab. Die Abfolge der einzelnen Anweisungen innerhalb eines Basisblocks ist dagegen gleichbleibend. Nach dem Aufbau der Zwischendarstellung spricht man bei diesen einzelnen Anweisungen von *abstrakten Maschinenoperationen (AMOs)*.

Definition 2.3 (Abstrakte Maschinenoperation) *Eine abstrakte Maschinenoperation (AMO) stellt eine maschinenunabhängige, elementare Anweisung der Zwischendarstellung dar.*

Die am häufigsten verwendete Darstellungsform für AMOs ist der *Drei-Adress-Befehl*.

Definition 2.4 (Drei-Adress-Befehl) *Ein Drei-Adress-Befehl hat die Form $x = y\ op\ z$, wobei x, y und z Namen, Konstanten oder vom Compiler generierte temporäre Werte sind und op ein binärer arithmetischer oder logischer Operator ist. Die Anzahl und Semantik der Operatoren ist unabhängig vom zugrunde gelegten Quellprogramm fest vorgegeben. In diesem Befehl wird x definiert und y und z werden verwendet.*

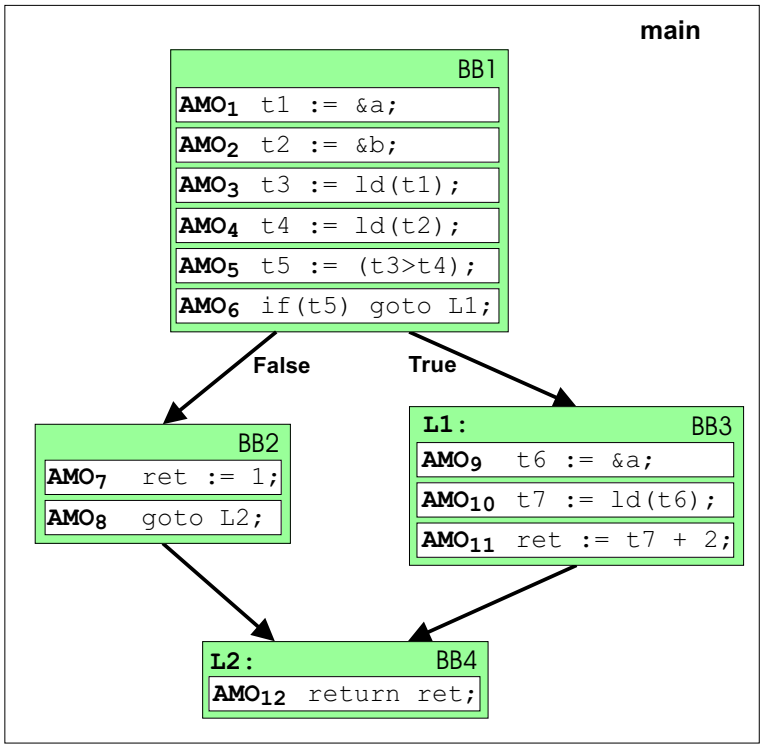

Abb. 2.5: Darstellung von AMOs durch Drei-Adress-Befehle in einem CFG aus Basisblöcken.

Beispiel: Das folgende Programm-Beispiel wird in Abb. 2.5 graphisch dargestellt. Der Kontrollfluss wird als CFG veranschaulicht und die Sequenz der AMOs durch Drei-Adress-Befehle repräsentiert.

```
int a,b;

int main()
{
  int ret;

  if (a > b)
    ret = a + 2;
  else
    ret = 1;

  return ret;
}
```

Mit den bisher beschriebenen Darstellungsformen lassen sich keine architekturspezifischen Merkmale darstellen. Da die Überführung dieser Darstellung in eine maschinenabhängige Form das Ziel der Codegenerierung ist, müssen auch prozessorspezifische Informationen abgelegt werden. Dazu gehören u.a. dessen Ressourcen.

Definition 2.5 (Sequentielle Ressourcen) *Sequentielle Ressourcen eines Prozessors sind lesbare bzw. schreibbare Ressourcen wie Registerbänke, Speicherbänke oder Ein- und Ausgabeports, deren Inhalte mehr als einen Instruktionszyklus erhalten bleiben.*

Definition 2.6 (Flüchtige Ressourcen) *Flüchtige Ressourcen sind lesbare bzw. schreibbare Ressourcen, deren Inhalt nur innerhalb eines Instruktionszyklus gültig ist. Dies sind z.B. Signalleitungen zwischen Funktionseinheiten oder Registern zum Zwischenspeichern eines Resultats einer Funktionseinheit, das noch im gleichen Instruktionszyklus von einer anderen Funktionseinheit gelesen wird.*

Während der Codegenerierung wird jeder AMO eine maschinenspezifische Operation zugewiesen. Dabei wird in dieser Arbeit zwischen zwei verschiedenen Arten von *Maschinenoperationen* unterschieden:

Definition 2.7 (Maschinenoperation) *Eine Maschinenoperation (MO) ist eine elementare Operation auf einem Prozessor, wobei die Operanden aus sequentiellen Ressourcen gelesen werden und das Resultat in eine sequentielle Ressource geschrieben wird. Eine Maschinenoperation ist an weitere Ressourcen gebunden, wie z.B. an Funktionseinheiten, auf denen die Operation ausgeführt wird.*

Definition 2.8 (Komplexe Maschinenoperation) *Eine komplexe Maschinenoperation setzt sich aus mindestens zwei MOs zusammen, die jeweils mindestens eine flüchtige Ressource benutzen.*

Eine weitere Aufgabe der Codegenerierung ist die Zusammenfassung von parallel ausführbaren MOs, zu *Maschineninstruktionen (MIs)*.

Definition 2.9 (Maschineninstruktion) *Eine Maschineninstruktion (MI) repräsentiert eine Menge von parallel auszuführenden Maschinenoperationen auf einem Prozessor.*

Für die Codegenerierung ist die sequentielle Darstellung der MOs innerhalb eines Basisblocks ungeeignet. Die drei verschiedenen Datenabhängigkeiten der MOs untereinander sind daraus nicht direkt ersichtlich, so dass eine graphische Darstellung vorteilhafter ist. Bei den Abhängigkeiten werden folgende unterschieden:

Definition 2.10 (Datenflussabhängigkeit) *Wenn AMO_i vor AMO_j ausgeführt wird und AMO_i eine Variable defniniert, die AMO_j verwendet, dann liegt eine Datenflussabhängigkeit zwischen diesen beiden AMOs vor.*

Definition 2.11 (Ausgabeabhängigkeit) *Wenn AMO_i vor AMO_j ausgeführt wird und beide AMOs dieselbe Variable definieren, dann liegt eine Ausgabeabhängigkeit zwischen diesen beiden AMOs vor.*

Definition 2.12 (Antiabhängigkeit) *Wenn AMO_i vor AMO_j ausgeführt wird und AMO_i eine Variable als Argument verwendet, die AMO_j defniniert, dann liegt eine Antiabhängigkeit zwischen diesen beiden AMOs vor.*

Die daraus resultierenden graphischen Darstellungen werden Datenabhängigkeitsgraphen genannt. Der für diese Arbeit wichtigste ist der *Datenflussgraph*.

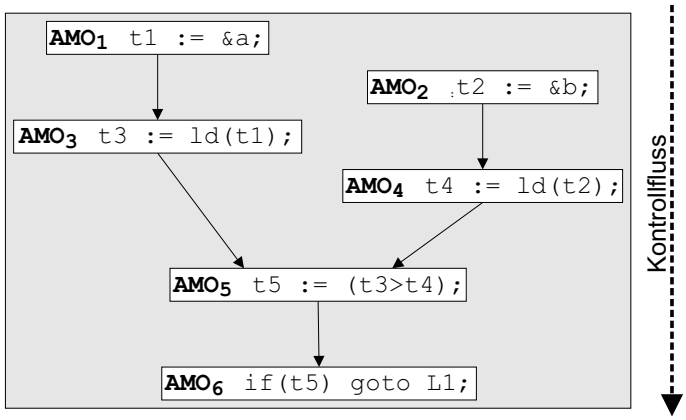

Abb. 2.6: Darstellung von AMOs eines Basisblocks als DFG.

Definition 2.13 (Datenflussgraph) *Ein Datenflussgraph (DFG) ist ein gerichteter Graph G = (V,E), in dem zwei Knoten $v_i, v_j \in V$, zwischen denen eine Datenflussabhängigkeit besteht, über eine Kante $e_{i,j} = (v_i, v_j) \in E \subseteq V \times V$ verbunden werden.*

In dieser Arbeit wird dabei zwischen dem *lokalen Datenfluss*, d.h. den Datenflussabhängigkeiten innerhalb eines Basisblocks, und dem *globalen* Datenfluss unterschieden, den Datenflussabhängigkeiten zwischen mehreren Basisblöcken. In Abb. 2.6 ist der DFG eines Basisblocks dargestellt. Werden die MOs/AMOs eines Basisblocks als DFG und die Basisblocks einer Funktion als CFG dargestellt, spricht man von einem Kontroll-Datenflussgraph (CDFG).

2.2.2 Compiler-Aufbau

Der Übersetzungsvorgang besteht aus drei Phasen, dem Front-, Middle-, und Back-End, wobei erstere die Analyse-Phase ist und letztere die Synthese-Phase. In diesem Abschnitt wird der Aufbau dieser Phasen betrachtet, wobei der Schwerpunkt auf dem Back-End liegt, das u.a. den Codegenerator enthält. In Abb. 2.7 wird darüber eine kurze Übersicht gegeben. Ausführliche Beschreibungen der einzelnen Phasen können z.B. [Muc97] entnommen werden.

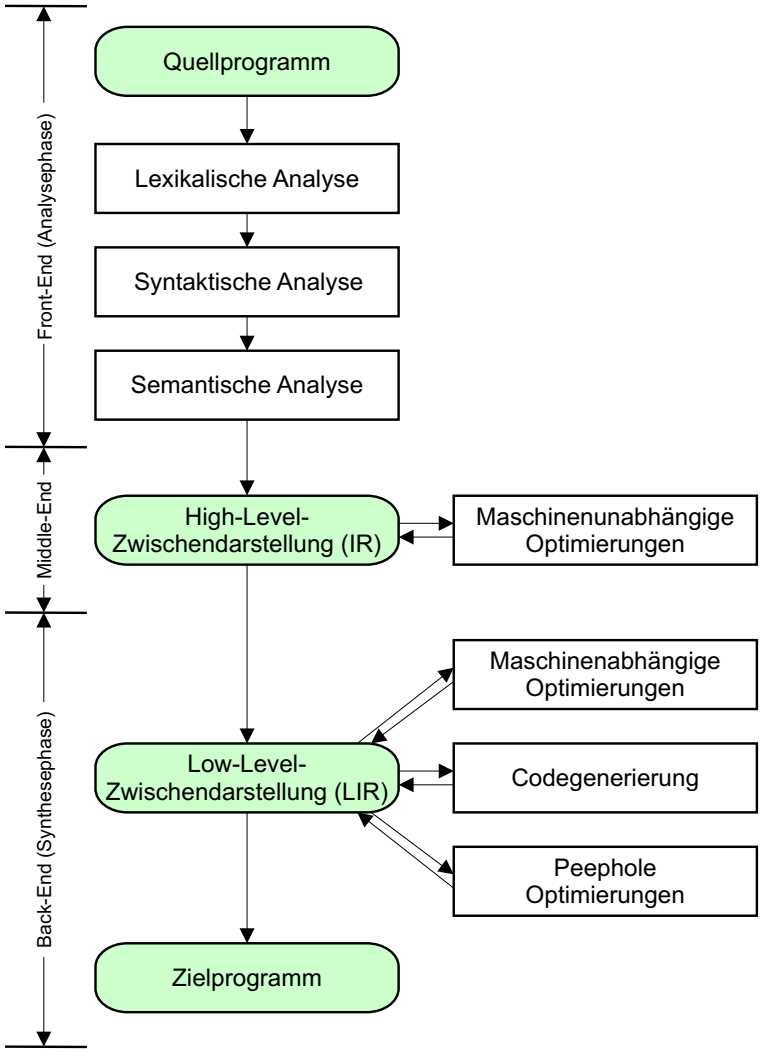

Abb. 2.7: Die einzelnen Phasen des Compilierungs-Prozesses

Front-End

Die Aufgabe des Front-Ends ist das Einlesen des Quellprogramms und der Aufbau der internen Zwischendarstellung (IR = Intermediate Representation) sowie der Symboltabelle, die anschließend von dem Middle- und Back-End benutzt werden. Das Front-End lässt sich wiederum in drei weitere Phasen aufteilen, die nacheinander durchlaufen werden.

Lexikalische Analyse (Scanning) Der Scanner liest das Quellprogramm als Zeichenfolge ein und zerlegt es in eine Folge lexikalischer Einheiten, sog. Symbole (tokens). Dies sind:

- Schlüsselwörter
- Satzzeichen
- Bezeichner
- Operatoren

Dabei wird die Symboltabelle aufgebaut, in der alle im Quellprogramm deklarierten Bezeichner sowie deren Attribute enthalten sind.

Syntaktische Analyse (Parsing) Der Parser überprüft, ob die beim Scanning erkannten Symbole der (kontextfreien) Grammatik der jeweiligen Quellsprache entsprechen. Dabei werden die Symbole zu Sätzen zusammengefasst und es wird ein Syntaxbaum aus Bezeichnern, Zeichen und Konstanten aufgebaut. Abb. 2.8 zeigt ein Beispiel: Die Markierung R eines Knoten symbolisiert, dass der Wert in einem Register abgelegt ist, die Markierung K bedeutet, dass es sich bei dem Wert um eine Konstante handelt.

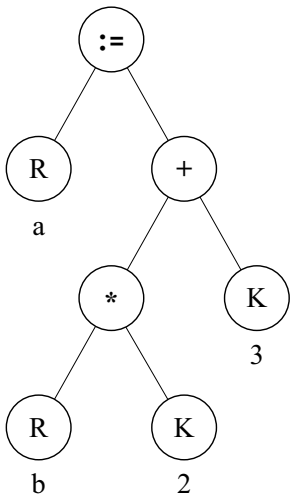

Abb. 2.8: Syntaxbaum für `a := b * 2 + 3`

Semantische Analyse In dieser Phase wird das Quellprogramm auf semantische Fehler überprüft. Dies geschieht durch eine Untersuchung des Syntaxbaums. Typische Überprüfungen sind:

- statische Typüberprüfung
- Überprüfungen des Kontrollflusses
- Überprüfungen der Eindeutigkeit von Deklarationen

Die erzeugte Zwischendarstellung hat mehrere Vorteile gegenüber dem Quellcode in einer Hochsprache. Komplizierte Ausdrücke und geschachtelte Kontrollflussanweisungen sind aufgelöst. Sie werden durch AMOs dargestellt und ermöglichen erst so verschiedene Optimierungen und die Codegenerierung. Durch das Einfügen von temporären Variablennamen sind die Abhängigkeiten zwischen den einzelnen Anweisungen analysierbar und ermöglichen z.B. deren Umordnung (siehe Abschnitt 2.2.2). Da die IR durch die Linearisierung des Syntaxbaums entsteht ist sie gleichzeitig ein gültiger Ablaufplan für die einzelnen Befehle.

Middle-End

Nach der Analyse, dem Front-End, liegt das Programm als IR vor. Dies ist eine Art Zwischenprogramm für eine abstrakte Maschine.

Die Aufgabe des Middle-Ends ist die Durchführung von High-Level-Optimierungen auf der IR, welche von der Sprache des Quellprogramms und von der Zielmaschine unabhängig sind. Mögliche Optimierungen sind:

- Constant Folding: Sind alle Argumente einer AMO Konstanten, dann kann diese durch das konstante Ergebnis der Operation ersetzt werden.
- Constant Propagation: Ersetzung von konstanten Variablenwerten durch Konstanten an ihrer Verwendungsstelle.
- Copy Propagation: Ersetzung von Kopien von Variablen-Werten durch die Benutzung der ursprünglichen Variable.
- Dead Code Elimination: Eliminierung von Operationen die nicht ausgeführt oder deren Ergebnisse nicht benötigt werden.
- Function-Inlining: Ersetzen von Funktionsaufrufen durch die Befehlssequenz der aufzurufenden Funktion, um Overhead einzusparen.

Back-End

In dieser Phase des Übersetzungsprozesses wird die interne Compiler-Zwischendarstellung vom Codegenerator in ein äquivalentes Maschinenprogramm überführt. Anschließend werden maschinenspezifische Peephole-Optimierungen durchgeführt und es erfolgt die Ausgabe des Maschinenprogramms in Assemblercode.

Die Codegenerierung besteht aus drei Phasen, der Codeselektion, der Instruktionsanordnung und der Registerallokation. Bei RISC-Maschinen lassen sie sich getrennt durchführen, anders als bei irregulären Architekturen, wie z.B. dem M3-DSP, der für optimale Ergebnisse eine phasengekoppelte Codegenerierung erfordert [Lor03]. Nachfolgend werden die einzelnen Phasen in der Reihenfolge vorgestellt, wie sie von einem Compiler für RISC-Architekturen am sinnvollsten durchlaufen werden.

Codeselektion (CS): Die Aufgabe der Codeselektion ist die Abbildung der AMOs der IR auf die Maschinenoperationen des Zielprozessors, sowie die Zuordnung der MOs zu den Maschineninstruktionen. Dabei wird je nach Kostenmodell z.B. die Befehlssequenz mit dem niedrigsten Energieverbrauch, der größten Ausführungsgeschwindigkeit oder mit dem kompaktesten Code ausgewählt. Sobald in der IR maschinenspezifische Informationen abgelegt werden, z.B. Prozessor-Ressourcen oder die Menge der MOs, die eine AMO auf der Zielmaschine umsetzen können, spricht man von der LIR (Low-Level Intermediate Representation).

Bei der Codeselektion für RISC-Prozessoren wird von einer unbegrenzt großen Anzahl symbolischer Register ausgegangen, d.h. es findet noch keine Zuordnung von Variablen zu Registern oder Speicherplätzen statt. Dies ist die Aufgabe der dritten Codegenerierungsphase, der Registerallokation.

Instruktionsanordnung (IA): Durch die Zwischencodegenerierung im Front-End wird bereits eine Reihenfolge der AMOs/MOs bestimmt. Diese hängt nur von der Linearisierung des Syntaxbaums ab und ist nicht architekturspezifisch, so dass weiteres Optimierungspotential besteht. Dabei müssen die drei verschiedenen Datenabhängigkeiten der Anweisungen beachtet werden (siehe Abschnitt 2.2.1).

Die Ziele der Umordnung einzelner Instruktionen sind:

- Abbau des *Registerdrucks*, d.h. Minimierung der Anzahl gleichzeitig zur Programmausführung benötiger Werte in den Registern, damit bei der Registerallokation möglichst wenige Werte im Speicher abgelegt werden. Dies ist bei RISC-Architekturen aufgrund hoher Kosten für den Speicherzugriff das Hauptziel der Instruktionsanordnung.
- Energieoptimierung, z.B. Reduzierung der Switching Power und damit des Energieverbrauchs beim Laden und Entladen von Lastkapazitäten im Prozessor.
- Codekompaktierung, um Maschinenoperationen besser zu Maschineninstruktionen zusammenzufassen, zwecks Parallelverarbeitung
- bei superskalaren Pipelinearchitekturen: optimale Auslastung der Funktionseinheiten des Prozessors, Berücksichtigung von Caches und Instruktion-Prefetching

Registerallokation (RA): Die Aufgabe der Registerallokation ist die Abbildung von Variablen und Ausdrücken der LIR auf physikalische Register und Speicherplätze. Der Zugriff auf den Speicher bei RISC-Architekturen verursacht, bedingt durch die zusätzlichen Load/Store-Befehle, größere Kosten als die Verwendung von Registern. Das Ablegen einer Variablen im Speicher wird als *Zwischenspeichern* oder *Spilling* bezeichnet.

Es lassen sich zwei Teilprobleme der Registerallokation unterscheiden:

1. Registervergabe: Es erfolgt eine Entscheidung, welche Variablen in Registern abgelegt werden. Dabei sind die Kosten von Zwischenspeicherungen zu minimieren.
2. Registerbindung: Für jede Variable wird ein physikalisches Register bzw. eine Speicheradresse ausgewählt.

Die Grundlage der Registerallokation ist die Bestimmung der Lebenszeitspannen aller Variablen. Diese erfolgt durch die Analyse der Kanten des DFGs.

Peephole-Optimierungen: Diese werden nach Abschluss der drei Codegenerierungsphasen durchgeführt. Es handelt sich dabei um lokal begrenzte, maschinenabhängige Optimierungen die nur einen kleinen Ausschnitt des Programms betrachten. Sie können auf der LIR oder dem Assemblercode angewendet werden. Gängige Peephole-Optimierungen sind:

- Redundant Store Elimination (RSE)
- Redundant Load Elimination (RLE)
- Dead-Code-Elimination
- Kontrollflussoptimierungen, z.B. Zusammenfassung von zwei aufeinanderfolgenden Sprüngen zu einem einzelnen
- Algebraische Vereinfachungen/Transformationen
- Berücksichtigung von maschinenspezifischen Eigenschaften, z.B. Inkrementfunktionen

Das Optimierungspotential von Peephole-Optimierungen ist im Vergleich zu vor der Codegenerierung durchgeführten Optimierungen geringer, da die Registerbindung bereits stattgefunden hat. Die Datenabhängigkeiten zwischen den MOs beziehen sich auf die in begrenzter Anzahl vorhandenen physikalischen Register. Die High-Level-Optimierungen im Middle-End und die Low-Level-Optimierungen vor der Codegenerierung betrachten dagegen die in unbegrenzter Anzahl vorhandenen symbolischen Register.

Assemblercode-Ausgabe Im letzten Schritt der Codegenerierung wird das in der LIR vorliegende Maschinenprogramm als Assemblercode ausgegeben. Dazu gehören neben den Maschinenbefehlen und Sprungmarkierungen vor allem die Variablendeklarationen und assemblerspezifische Anweisungen, z.B. für die Einteilung der Code- und Datensegmente.

2.3 Compiler-Zwischendarstellung GeLIR

GeLIR (Generic Low Level Intermediate Representation) [GeL] ist ein universelles Compiler-Austauschformat. Es dient als Zwischendarstellung für das Middle- und Back-End, so dass das Quellprogramm auf beiden Abstraktionsebenen dargestellt werden kann: maschinenunspezifisch als IR und maschinenspezifisch als LIR. Getrennt von der Darstellung des Quellprogramms werden die prozessorspezifischen Architekturmerkmale verwaltet, wie z.B. die einzelnen Ressourcen und der Instruktionssatz.

GeLIR ist eine Weiterentwicklung von CoLIR [Bas01] am LS XII und wurde ursprünglich für die Codegenerierung irregulärer Architekturen entworfen, z.B. für den M3-DSP. Das Ziel war eine einheitliche Zwischendarstellung zur Wiederverwendung einmal entwickelter Compiler-Techniken auf beiden Abstraktions-Ebenen.

Detailliertere Beschreibungen von GeLIR können [Lor03] und [Fie01] entnommen werden.

2.3.1 Programmdarstellung

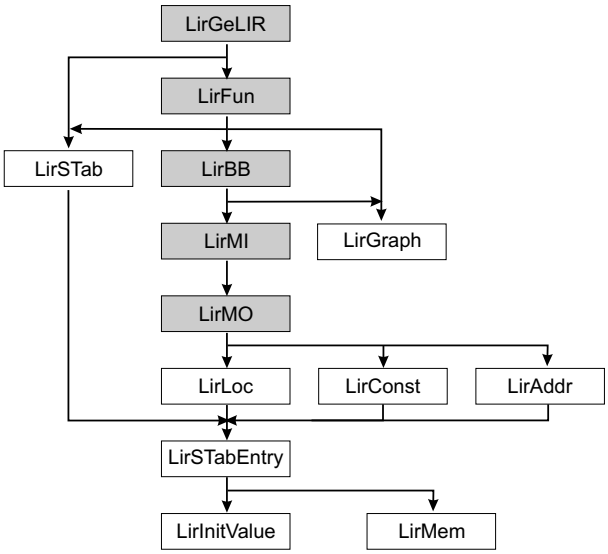

Abb. 2.9: Übersicht über die Klassen der GeLIR-Programmdarstellung

Die Hierarchie der GeLIR-Programmdarstellung ist in Abb. 2.9 dargestellt. Die Pfeile repräsentieren ‚benutzt '-Beziehungen zwischen den GeLIR-Klassen. Die zentralen Klassen sind:

- *LirGeLIR:* Diese Klasse verwaltet die gesamte Programmdarstellung und verweist auf die globale Symboltabelle (*LirSTab*) sowie auf alle Funktionen des Programms.
- *LirFun:* Die Objekte dieser Klasse repräsentieren die Funktionen des Programms. Neben der lokalen Symboltabelle (*LirSTab*) enthalten sie Informationen über den CFG einer Funktion und Verweise auf die enthaltenen Basisblöcke.
- *LirBB:* Jeder Basisblock ist ein Knoten im CFG einer Funktion. Ein LirBB-Objekt enthält Informationen über die dort verwendeten Variablen. Alle im Basisblock definierten Variablen stehen in der V_{Out}-Liste. Die V_{In}-Liste enthält alle Variablen, die im Basisblock verwendet werden und nicht zuvor in diesem definiert wurden. Die Variablen dieser beiden Listen, die am globalen Datenfluss beteiligt sind, werden nachfolgend mit V_{GlobIn} und $V_{GlobOut}$ bezeichnet. Sie sind die Basis für Datenflussanalysen während der RA.
- *LirMI:* Diese Objekte stellen eine MI dar, die nach der Codegenerierung eine oder mehrere MOs enthalten. Die Werteübergabe zwischen MOs einer MI erfolgt nur über flüchtige Ressourcen (siehe Definition 2.6).
- *LirMO:* Eine Maschinenoperation (MO) repräsentiert genau eine elementare Anweisung. Dies können Konstanten, Adressen, arithmetische & logische Operationen, Load-/Store-Anweisungen, Sprünge, Funktionsaufrufe/-rücksprünge und Datentypumwandlungen sein. Zwei Arten von MOs haben eine besondere Bedeutung: *CopyMOs* repräsentieren (potentiell) vorhandene Datentransfer-Befehle (siehe Abschnitt

2.3.3), *komlexe MOs* dienen zur Modellierung von Maschinenoperationen die auf dem Prozessor während eines einzelnen Taktzyklus ausgeführt werden (siehe Abschnitt 2.3.4).

2.3.2 Architekturdarstellung

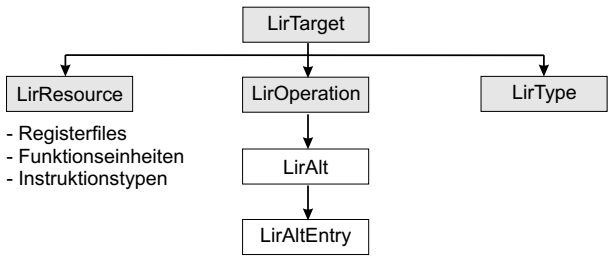

Abb. 2.10: Übersicht über die Klassen der GeLIR-Architekturdarstellung

Die Beschreibung der Zielarchitektur ist in einer eigenen Datenstruktur abgelegt. Diese stellt die Ressourcen, die Datentypen und die Operationen des zugrunde liegenden Prozessors dar (siehe Abb. 2.10). Die entsprechenden Klassen sind:

- *LirResource:* Mit dieser Klasse werden Registerfiles, Funktionseinheiten und Instruktionstypen dargestellt. Instruktionstypen dienen zur Modellierung von parallelen Ausführungsmöglichkeiten.
- *LirOperation:* LirOperation-Objekte beschreiben jeweils eine Operation, wobei zwischen AMOs und MOs unterschieden wird. Die gültigen Ressource-Kombinationen einer MO werden als *alternative MOs* oder als *Alternativen* bezeichnet und sind jeweils in einem *LirAltEntry*-Objekt abgelegt. Die Zuordnung einer alternativen MO zu einer AMO wird als *Überdeckung* bezeichnet.

LirAltEntry 1:

```
Op    = {ADD}
FU    = {ALU}
IT    = {it_none}
Def   = {Lo, Hi}
Arg1 = {Lo, Hi}
Arg2 = {Lo, Hi}

Exec-Time = 1
Covered-Arg-Pos = 1


Bsp: Lo = ADD(Lo,Hi)
     Hi = ADD(Hi,Hi)
```

LirAltEntry 2:

```
Op    = {ADD}
FU    = {ALU}
IT    = {it_none}
Def   = {Lo}
Arg1 = {Lo}
Arg2 = {Lo, cnst_uint7}

Exec-Time = 1
Covered-Arg-Pos = 1


Bsp: Lo = ADD(Lo,127)
```

LirAltEntry 3:

```
Op    = {ADD}
FU    = {ALU}
IT    = {it_none}
Def   = {Lo}
Arg1 = {Lo}
Arg2 = {Lo, cnst_uint3}

Exec-Time = 1


Bsp: Lo = ADD(Lo,7)
```

Abb. 2.11: Darstellung der ADD-Instruktion des ARM durch *LirAltEntry*-Objekte

In Abb. 2.11 sind die LirAltEntry-Objekte zur Modellierung der ADD-Instruktion des ARM-Prozessors dargestellt. Die Ressourcen-Mengen haben folgende Bedeutung:

  - *Op:* Enthält die Menge von Maschinenoperationen, die eine Addition auf der Zielmaschine durchführen können.
  - *FU:* Gibt die Funktionseinheiten an, auf der die MOs ausgeführt werden können.
  - *IT:* Der Instruktionstyp *it_none* gibt an, dass die MO nicht parallelisierbar ist.
  - *Def:* Diese Menge enthält die Registerfile-Ressourcen, in die der Ergebnis-Wert der MO geschrieben werden kann.
  - *Arg1/2:* Die Registerfile-Ressourcen, aus denen die Argumente der MO gelesen werden können, sind in dieser Menge enthalten.

  Das Attribut *ExecTime* gibt die Ausführungszeit der beschriebenen MOs auf dem Prozessor an und *Covered-Arg-Pos = 1* bedeutet, dass die Definition der MO in das Registerfile des ersten Arguments geschrieben wird.

- *LirType:* LirType-Objekte beschreiben die verwendeten Typen. Es können die Standard-Typen von C, aber auch komplexe Typen wie z.B. Pointer, Arrays oder Structs dargestellt werden.

Es lassen sich mit der Architekturdarstellung viele Details des Prozessors beschreiben. Besonders deutlich wird dies bei der Spezifikation der Registerfiles. So kann z.B. die Anzahl der einzelnen Elemente, die Größe der Elemente und der aufzunehmende Datentyp beschrieben werden. Es werden mit Registerfiles die Register, der Hauptspeicher und flüchtige Ressourcen des Prozessors dargestellt. Allerdings sind für bestimmte Codegenerierungs-Techniken die Hardware-Details eines Prozessors nicht ausreichend. Zur Durchführung der Registerallokation werden darüber hinaus noch Informationen über die Programmier-Konventionen benötigt, z.B. wie die Verwendung von Registern zur Übergabe von Funktions-Parametern erfolgt.

2.3.3 Copy-MO

Dabei handelt es sich um eine abstrakte Maschinenoperationen, die einen vorhandenen oder potentiell vorhandenen Datentransfer-Befehl darstellt. Sie ist die einzige AMO, die u.a. mit der *Copy-Alternative* überdeckt wird. Diese repräsentiert eine Maschinenoperation ohne Funktion und hat die Ausführungzeit null. Weitere alternative MOs sind z.B. die Alternativen von Load-/Store- und Move-Befehlen. Bei der Codeselektion wird für jede abstrakte Anweisung der IR eine alternative MO ausgewählt. Da die Copy-Alternative keine Operation auf dem Zielprozessor repräsentiert, werden anschließend alle damit überdeckten MOs entfernt.

Die Verwendung von Copy-MOs wird für die Codeselektion in Abschnitt 3.5 und für die Registeralloktion in Abschnitt 3.7 beschrieben.

2.3.4 Komplexe MO

Mit komplexen Maschinenoperationen lassen sich z.B. Immediate-Werte als Operations-Argumente oder Load-/Store-Befehle mit Adressaddition innerhalb eines Taktzyklus modellieren. In Abb. 2.12 wird dies an dem Beispiel der Addition mit einem Immediate-Wert

gezeigt. MO_1 definiert eine flüchtige Ressource, MO_3 verwendet diese, was durch einen gestrichelten Pfeil im Datenflussgraphen dargestellt ist. Werden komplexe MOs erstellt, z.B. nach Durchführung der Codeselektion, wird MO_1 zur SubMO von MO_3.

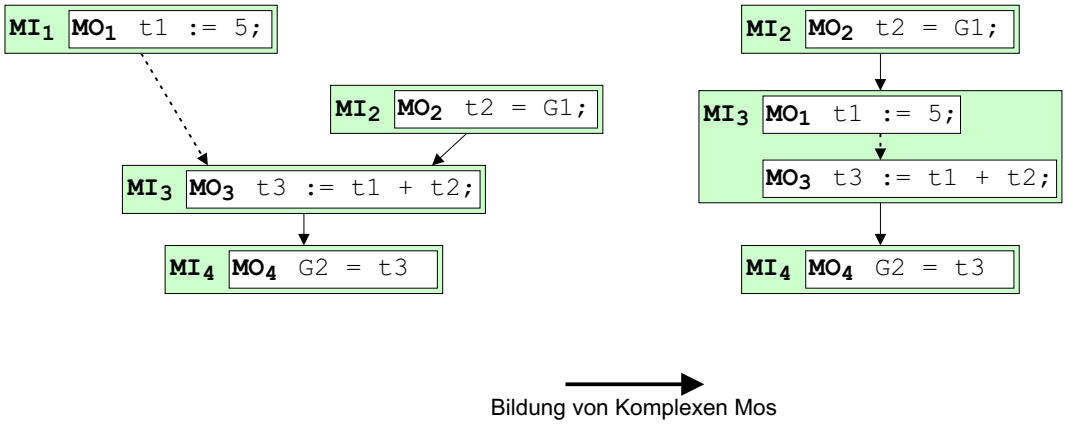

Abb. 2.12: Zusammenfassung von zwei MOs zu einer komplexen MO

Zusammen mit Copy-MOs können so z.B. gleichzeitig mehrere Möglichkeiten der Addition eines Registerinhaltes mit einer Konstanten dargestellt werden. Dies wird in Abschnitt 3.5.1 detailliert beschrieben.

Kapitel 3

Implementierung des Codegenerators

Die Codegenerierung ist die letzte Phase des Übersetzungsprozesses. Die Eingabe ist die maschinenunabhängige Zwischendarstellung des Quellprogramms, die Ausgabe ist ein semantisch äquivalentes Maschinenprogramm in Assemblersprache. In diesem Kapitel wird der Aufbau des gesamten Compilers erläutert, wobei vor allem die implementierten Techniken des generischen Codegenerators betrachtet werden.

3.1 Verwandte Arbeiten

Es bestehen bereits verschiedene retargierbare Compiler, die für RISC-Prozessoren verwendbar sind. Zwei verbreitete sind der GCC und der LCC.

- **GCC:**

  Dieser Compiler [Gcc] kann frei verwendet, modifiziert und weiterverteilt werden. Es existieren Front-Ends für mehrere Programmiersprachen, z.B. C++, Java, Fortran. Back-Ends gibt es für eine große Anzahl von RISC- und CISC-Prozessoren, z.B. für x86, M68000, PowerPC, Sparc, MIPS und auch ARM7 (ARM-Modus!). Der GCC-Compiler ist sehr flexibel, allerdings auch sehr komplex in seiner Struktur. Da der Code in ca. 20 Phasen generiert wird, wäre eine Anpassung im Rahmen dieser Diplomarbeit zu aufwendig. Außerdem ist z.B. die Bewertung des Energieverbrauchs nicht möglich, wodurch die spätere Erweiterbarkeit des generischen Codegenerators eingeschränkt würde [Ste02].

- **LCC:**

  Der LCC (Little C Compiler) ist ein weiterer häufig in der Forschung verwendeter Compiler. Im Gegensatz zum GCC ist dessen Implementierung detailliert dokumentiert, in Form eines Buches [FH95]. Mitgelieferte Back-Ends gibt es für die Prozessoren ALPHA, SPARC, MIPS und x86 [Lcc]. Die Qualität des generierten Codes ist schlechter als beim GCC, da nur wenige Optimierungen vorhanden sind.

- **ENCC:**

  Der ENCC (Energy Aware C Compiler) [Enc] ist ein Low-Power-Forschungscompiler des LS XII. Er entnimmt das Quellprogramm der LANCE-IR [Lan] (siehe auch Abschnitt 3.3). Im Gegensatz zu dem generischen Compiler dieser Diplomarbeit ist der ENCC fest an die LANCE-IR gekoppelt. Es existieren Back-Ends für die ARM- und LEON-Architektur, die allerdings nicht auf einer generischen Codegenerierung basieren. Der Aufbau und die für den ARM verwendeten Compiler-Techniken werden detailliert in [Ste02] beschrieben.

In Bezug auf die Ziele dieser Diplomarbeit haben die verfügbaren retargierbaren Compiler einen großen Nachteil. Es wurde bei der Entwicklung davon ausgegangen, dass unterschiedliche Ziel-Prozessoren auch verschiedene Codegeneratoren benötigen. Der alleinige Austausch der architekturabhängigen Spezifikationen ohne sonstige Veränderung der Codegeneratoren ist dabei nicht vorgesehen. Für jede der beiden betrachteten RISC-Architekturen müsste ein seperates Back-End implementiert werden.

Die für einen generischen Codegenerator nötige Compiler-Infrastruktur muss eine Trennung zwischen Programmdarstellung, Architekturdarstellung und Codegenerierung ermöglichen. Außerdem ist eine standardisierte Zwischendarstellung nötig, die maschinenspezifische Details aufnehmen kann, aber selbst nicht an eine bestimmte Architektur gebunden ist. Unter diesen Voraussetzungen kann ein Codegenerator implementiert werden, der keine Detailkenntnisse über den jeweiligen Zielprozessor besitzt. Außerdem ist bei einer solchen Programmdarstellung die Wiederverwertbarkeit einmal implementierter Codegenerierungs-Techniken oder Optimierungen möglich, sogar zwischen unterschiedlichen Prozessor-Klassen. Die GeLIR-Zwischendarstellung entspricht diesen Anforderungen.

3.2 Architektur-Voraussetzungen

Bei der Entwicklung der generischen Techniken dieses Codegenerators wurden vor allem die Merkmale der ARM- und der SPARC-V8-Architektur betrachtet, ohne jedoch zu starke Einschränkungen vorzunehmen. Dadurch ergeben sich eine Reihe von Voraussetzungen an die Zielprozessoren:

- Homogener Registersatz: Es wird davon ausgegangen, dass das Ergebnis jeder Instruktion in einem dieser universellen Register abgelegt werden kann, von Store-Befehlen abgesehen.

- Load-/Store-Architektur: Speicherzugriffe erfolgen nur über dafür vorgesehene Load-/ Store-Befehle. Der direkte Zugriff, z.B. durch arithmetisch-logische Befehle, wird nicht unterstützt.

- Load-/Store-Befehle mit Adresskodierung als Immediate-Wert: Für das während der Registerallokation (RA) eingefügte Zwischenspeichern sind Load-/ Store-Befehle erforderlich, deren Zugriffsadresse im Befehlswort kodiert wird, wie z.B. überlicherweise bei SP-relativer (SP = Stack Pointer) Adressierung.

- Keine parallele Programmausführung: Es erfolgt keine Berücksichtigung evtl. vorhandener Parallelisierungsmöglichkeiten. Dies stellt aber keine grundsätzliche Einschränkung dar, da entsprechende Techniken zur Parallelisierung einfach dem Compiler hinzugefügt werden können.

3.3 Aufbau des Compilers

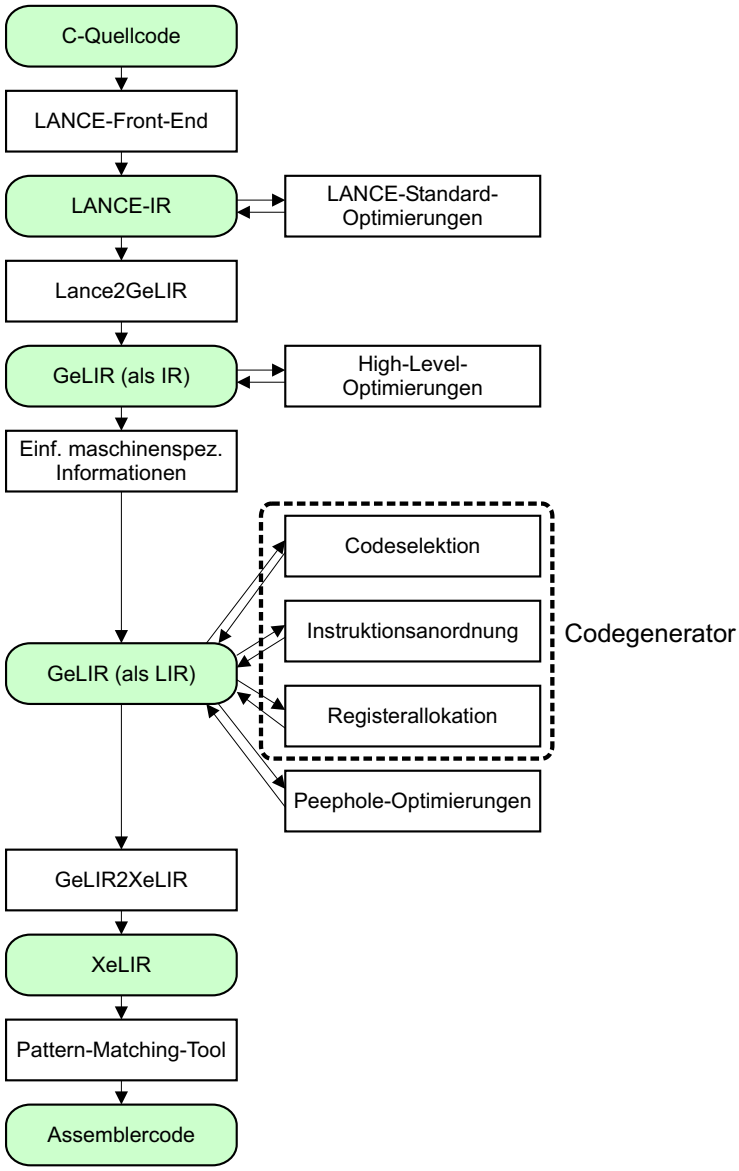

Abb. 3.1: Die einzelnen Schritte des realisierten Compilers

In diesem Abschnitt wird ein Gesamtüberblick über den erstellten Compiler gegeben. Der Ablauf des gesamten Compilierungs-Vorgangs ist in Abb. 3.1 dargestellt. Die im Rahmen dieser Arbeit implementierten Techniken werden in den nachfolgenden Abschnitten detailliert beschrieben.

Der *C-Quellcode* wird von dem *LANCE-Front-End* eingelesen und analysiert. *LANCE* [Lan] ist eine an der Universität Dortmund entwickelte IR. Die Darstellung erfolgt als Drei-Adress-Code und kann als C-Syntax zur Validierung mit einem herkömmlichen Compiler herausgeschrieben werden. Neben dem Front-End gibt es mehrere maschinenunabhängige Standard-Optimierungen die eine gegebene IR lesen und modifiziert zurückschreiben, wie z.B. Jump Optimization, Common Subexpression Elimination oder Dead Code Elimination.

Mit der *LANCE2GeLIR*-Methode von GeLIR wird die GeLIR-Programmdarstellung aus der LANCE-IR aufgebaut, wobei die vorläufige Reihenfolge der Anweisungen übernommen wird. Auf dieser noch maschinenunabhängigen GeLIR-Programmdarstellung werden einige High-Level-Optimierungen durchgeführt, die in Abschnitt 3.4 näher beschrieben werden. Anschließend erfolgt die *Einfügung maschinenspezifischer Informationen*, wie z.B. Registerfile-Ressourcen und alternativen MOs, wodurch die Programmdarstellung von der IR in die LIR überführt wird.

Sind die maschinenspezifischen Informationen in der Programmdarstellung abgelegt, wird die Codegenerierung durchgeführt. Diese besteht aus insgesamt drei Phasen, die in den nachfolgenden Abschnitten ausführlicher beschrieben werden. Die erste ist die *Codeselektion* (siehe Abschnitt 3.5), in der für jede abstrakte Anweisung der IR eine prozessorspezifische Maschinenoperation ausgewählt wird. Die zweite Phase ist die *Instruktionsanordnung* (siehe Abschnitt 3.6), in der die Abfolge der Instruktionen unter Betrachtung der Datenabhängigkeiten optimiert wird, und als letztes folgt die *Registerallokation* (siehe Abschnitt 3.7), in der die symbolischen Register an physikalische Register und Speicherplätze gebunden werden.

Für den THUMB-Modus des ARM7 [Hor01a] werden auf der GeLIR-Zwischendarstellung nach der Codegenerierung sechs bereits vorhandene Low-Level-Optimierungen durchgeführt. Dabei handelt es sich um folgende Standard-Optimierungen:

- Constant Folding
- Constant Propagation
- Copy Propagation
- Redundant Store Elimination (RSE)
- Redundant Load Elimination (RLE)
- Dead Code Elimination

Die Durchführung der Optimierungen ist unabhängig von der exakten Modellierung des ARM-Prozessors. Sie sind mit der ARM-Architekturdarstellung dieser Arbeit verwendbar, wobei allerdings Einschränkungen bestehen. Die Voraussetzung für die Ausführung der Optimierungen ist eine erfolgte Codeselektion.

Um den *Assemblercode* auszugeben, wird zunächst die Programm- und die Architekturdarstellung in die XeLIR-Datenstruktur [Fie01] exportiert. Aus dieser Darstellung wird der Assemblercode mittels eines *Pattern-Matching-Tools* herausgeschrieben (siehe Abschnitt 3.8).

3.4 High-Level-Optimierungen

Für das Middle-End wurde neben einigen Standard-Optimierungen wie Dead Code Elimination und Copy Propagation eine algebraische Transformation implementiert. Diese Transformation ersetzt Mul-Befehle mit einer Konstanten $k = 2^n$ als Faktor durch einen Shift-Left-Befehl mit n als Argument bzw. Div-Befehle durch den entsprechenden Shift-Right-Befehl. Ein häufig auftretender Fall ist die Multiplikation mit der Zahl vier zur Berechnung des 32-Bit-Offset innerhalb eines Arrays. Nachfolgend ist die algebraische Formation einer Multiplikation mit vier für den ARM-Prozessor dargestellt, bei der sich der mit vier zu multiplizierende Wert im Register *r0* befindet.

```
MOV r1, 4
MUL r0, r1   =>   SLS r0, 2
```

Vor allem für den ARM-Prozessor erzeugt diese Optimierung qualitativ besseren Code. Die Gründe dafür sind:

- Mögliche Übersetzung von Quellprogrammen mit einfachen Divisions-Operationen, denn der THUMB-Befehlssatz des ARM7-Prozessors stellt keinen Div-Befehl zur Verfügung. Eine Alternative wäre der Wechsel in den ARM-Modus.
- Einsparung eines verwendeten Registers, da die Shift-Befehle im Gegensatz zum Mul-Befehl auch Immediate-Werte als Argument ermöglichen.
- Schnellerer Code, da ein Mul-Befehl 2-9 Taktzyklen benötigt, die Shift-Befehle dagegen nur 2 Taktzyklen

Der Mul-Befehl des LEON-Prozessors unterstützt Immediate-Werte als Argument. Er benötigt immer 4 Taktzyklen, ein Shift-Befehl mit Immediate-Wert als Argument dagegen nur 1 Taktzyklus.

3.5 Codeselektion

In diesem Abschnitt wird der implementierte Codeselektions-Algorithmus erklärt. Dazu werden die beiden zugrundeliegenden Codeselektionstechniken vorgestellt: Das weitverbreitete *Tree-Pattern-Matching* und die Codeselektion mit Hilfe der *Constraintpropagierung*. Aus diesen beiden Verfahren wurde ein einfacher Algorithmus für die Codeselektion für RISC-Prozessoren implementiert.

3.5.1 Bestehende Techniken

Die Aufgabe der Codeselektion besteht aus der Abbildung der AMOs auf den Instruktionssatz der Zielmaschine. Es wird zwischen graphbasierten und baumbasierten Verfahren unterschieden. Der Vorgang wird als *Überdeckung* bezeichnet. Werden die AMOs eines Basisblocks als DFG dargestellt, kann dieser in Bäume aufgeteilt werden, indem z.B. anhand der Common Sub Expressions die Graphen in Bäume zerlegt werden.

Tree-Pattern-Matching

Dies ist ein weitverbreitetes Verfahrung zur Überdeckung von Bäumen. Es führt in Laufzeit O(n) eine optimale Codeselektion auf Datenflussbäumen durch, wobei sequentieller Code ausgegeben wird. Der Datenflussbaum wird dabei durch die Anwendung von *Baumtransformationsregeln* auf einen einzigen Knoten reduziert, wobei jede Anwendung einer Regel die Auswahl einer alternativen MO zur Folge hat. Eine ausführlichere Beschreibung dieses Verfahrens kann [Tei97] und [Bas95] entnommen werden.

Die Regeln zur Baumüberdeckung haben die Form:

$$E \leftarrow T \ \{A\}$$

Passt das Muster T auf einen Teilbaum des Datenflussbaums, kann dieser durch den Knoten E ersetzt und die Aktion A ausgeführt werden. T ist ein Baum, der die Anwendbarkeitsbedingung für die Regel beschreibt. Ein Muster *passt* wenn es mit dem betrachteten Teilbaum hinsichtlich Knoten, Kinderzahl und Ordnung der Teilbäume übereinstimmt. Eine Regel repräsentiert eine MO auf dem Zielprozessor, wird sie angewendet findet durch die Aktion A die Code-Ausgabe bzw. Zuordnung der AMO zur MO statt.

Eine Erklärung erfolgt anhand eines kleinen Beispiels, der Anweisung `a := 5`, wobei davon ausgegangen wird, dass es sich bei `a` um eine globale Variable handelt und diese daher im Speicher abgelegt werden muss. In Abb. 3.2 ist der Datenflussbaum dieser Zuweisung dargestellt. Das Terminal + repräsentiert die Addition von zwei Integer-Werten, das Terminal := das Speichern des Wertes des rechten Kindes an der durch das linke Kind gegebenen absoluten Adresse. Die Terminale K repräsentieren jeweils eine bestimmte Konstante, das Terminal R ein Register. Der Bezeichner *&a* ist eine Konstante und steht für die SP-relative (SP = Stack Pointer) Adresse der definierten Variable.

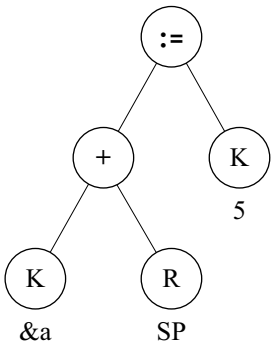

Abb. 3.2: Ein Datenflussbaum für die Anweisung `a := 5`

Die Menge der Regeln ist eine mögliche Beschreibung des Instruktionssatzes des Zielprozessors und wird deshalb auch Maschinengrammatik genannt. Das Problem der Codeselektion kann so auch als Parse-Problem für kontextfreie Grammatiken aufgefasst werden [GG78]. Die Anwendbarkeit der Regeln auf die Teilbäume wird durch Mustererkennung überprüft. Passt das Muster, wird der entsprechende Teilbaum durch den Knoten E ausgetauscht und die mit der Regel verknüpfte Aktion ausgeführt. Ist der Datenflussbaum auf einen einzigen Knoten reduziert oder passt kein Muster mehr, terminiert das Verfahren. Für den Fall, dass auf einen Teilbaum mehrere Regeln passen, ist ein weiteres

Unterscheidungskriterium nötig. Dazu werden den Regeln Kosten zugewiesen, z.B. die Ausführungszeit der entsprechenden Instruktion in Taktzyklen.

Abb. 3.3 zeigt passende Regeln zur Baumüberdeckung des Beispiels `a := 5`, die THUMB-Befehle des ARM7 repräsentieren. Die angegebenen Kosten entsprechen den Taktzyklen der jeweiligen Instruktion. Der Store-Befehl des ARM (Regel (2)) berechnet die Adresse des Speicherzugriffs durch die Addition von zwei Registern (Regeln (3)), oder durch die Addition eines Registers mit einer Konstanten (Regeln (4)). Die Berechnung der Adresse durch die Addition eines Registers mit der Konstanten `c = 0` wird extra betrachtet (Regel (1)), da diese nicht im Datenflussbaum dargestellt wird. Die Regeln (3) & (4) entsprechen MOs des Zielprozessors, aber keinen Maschineninstruktionen. Sie erzeugen keine Code-Ausgabe, sondern dienen zur Modellierung der Adressberechung bei der Ausführung einer Store-Instruktion. Die Nichtterminale stellen die Ressourcenklassen der Zielmaschine dar: k steht für eine Konstante, r für ein Register, *'+'* für eine flüchtigen Ressource (siehe Definition 2.6 auf Seite 22) und m für einen Speicherplatz. Die Anzahl der von einer Store-Instruktion benötigten Taktzyklen ist unabhängig von der Adressberechnung, es muss also die Überdeckung mit Regel (1) die gleichen Kosten ergeben wie die Überdeckung mit den Regeln (3) & (2) bzw. (4) & (2).

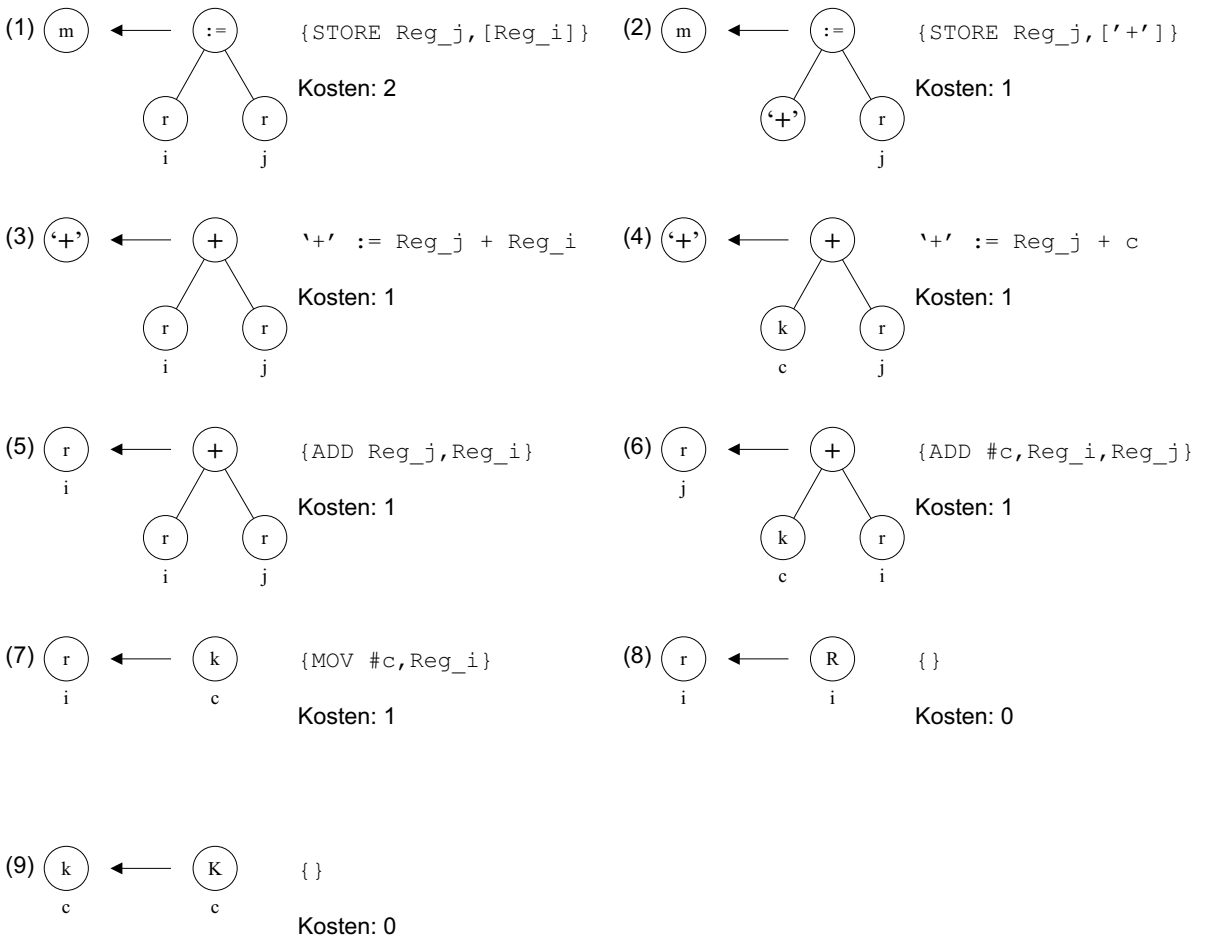

Abb. 3.3: Die Baumtransformationsregeln für das Beispiel aus Abb. 3.2

Es gibt für den Datenflussbaum aus Abb. 3.2 vier mögliche Überdeckungen mit den Regeln

aus Abb. 3.3. Die Regeln (8) & (9) werden nachfolgend nicht aufgeführt, da sie keine Code-Ausgabe erzeugen und unabhängig von der übrigen Überdeckung des Baums angewendet werden. Mit einer anschließend durchgeführten Registerallokation ergeben sich folgende möglichen Pseudo-Assemblercode-Ausgaben:

```
1. Möglichkeit:            Gesamtkosten : 5

(7)     MOV   &a, Reg_1                 : 1
(5)     ADD   Reg_1, Reg_SP             : 1
(7)     MOV   #5, Reg_2                 : 1
(1)     STORE Reg_2, [Reg_1]            : 2


2. Möglichkeit:            Gesamtkosten : 4

(6)     ADD   &a, Reg_SP, Reg_1         : 1
(7)     MOV   #5, Reg_2                 : 1
(1)     STORE Reg_2, [Reg_1]            : 2


3. Möglichkeit:            Gesamtkosten : 4

(7)     MOV   &a, Reg_1                 : 1
(7)     MOV   #5, Reg_2                 : 1
(3)(2)  STORE Reg_2, [Reg_1 + Reg_SP]   : 1+1


4. Möglichkeit:            Gesamtkosten : 3

(7)     MOV   #5, Reg_2                 : 1
(4)(2)  STORE Reg_2, [Reg_SP + &a]      : 1+1
```

Um bei der Überdeckung das globale Kostenminimum für den Datenflussbaum zu erreichen, werden die Regeln erst angewendet, wenn die Kosten für alle Unterbäume eines Knotens bekannt sind. Daraus ergibt sich ein zweistufiges Vorgehen. Im ersten Schritt werden durch *Tree-Pattern-Matching* für jeden Knoten alle passenden Regeln und die dazugehörigen Kosten des gesamten Teilbaums ermittelt. Im zweiten Schritt wird nach dem Grundprinzip der *Dynamischen Programmierung* die kostenminimale Überdeckung ausgewählt. Der folgende Algorithmus wird Tree-Pattern-Matching-Algorithmus (TPM) genannt.

TPM-Algorithmus:

1. Markierungs-Phase

   Diese Phase arbeitet *bottom-up*, d.h. von den Blättern des Datenflussbaums hin zum Wurzel-Knoten. Jeder Knoten wird mit allen passenden Regeln überdeckt. Dabei

werden die Ersetzungs-Nichtterminale der Regeln und die dazugehörigen minimalen Kosten des Teilbaums vermerkt. Für jeden Knoten werden folgende Schritte ausgeführt:

(a) Überdeckung mit den passenden Regeln:

Durch Mustererkennung werden alle Regeln gefunden, die auf den aktuell betrachteten Teilbaum passen, d.h. die hinsichtlich des Terminals, der Nichtterminale, der Anzahl der Kinder und der Ordnung der Unterbäume übereinstimmen. In dem Beispiel sind dies:

- Terminal R : Regel (8), erreichbare Nichtterminale: r
- Terminal K : Regeln (9),(7), erreichbare Nichtterminale: *k, r*
- Terminal $+$: Regeln (3),(4),(5),(6), erreichbare Nichtterminale: *r, '+'*
- Terminal $:=$: Regeln (1),(2), erreichbares Nichtterminal: m

(b) Kostenberechnung:

In diesem Schritt werden für die erreichbaren Nichtterminale eines Knotens die minimalen Kosten des Teilbaums berechnet. Dazu werden für jede passende Regel die Kosten durch Addition der Regel-Kosten mit den entsprechenden Kosten der Unterbäume berechnet. Dies ist möglich, da aufgrund des bottom-up-Verfahrens die Kinderknoten bereits betrachtet wurden. Im Beispiel ergeben sich folgende Kostenberechnungen, wobei die Kosten c in der Reihenfolge *c(Regel) + c(linkes Kind) + c(rechtes Kind)* addiert werden:

- Terminal K
  - *c(k)* $= 0$
  - *c(r)* $=$ *k(Regel (7))* $= 1$
- Terminal $+$
  - *c(r)* $= \min($*k(Regel (5))* $+ 1 + 0$, *k(Regel (6))* $) + 0 + 0$ $)$
    $= \min(1 + 1 + 0, 1 + 0 + 0) = 1$
    merke Regel (6)
  - *c('+')* $= \min($*k(Regel (3))* $+ 1 + 0$, *k(Regel (4)* $) + 0 + 0$ $)$
    $= \min(1 + 1 + 0, 1 + 0 + 0) = 1$
    merke Regel (4)
- Terminal $:=$
  - *c(m)* $= \min($*k(Regel (1))* $+ 1 + 1$, *k(Regel (2)* $) + 1 + 1$ $)$
    $= \min(2 + 1 + 1, 1 + 1 + 1) = 3$
    merke Regel (2)

2. Reduzierungs-Phase

In der vorherigen Phase wurden für jedes erreichbare Nichtterminal eines Knotens die minimalen Kosten des Teilbaums berechnet. An der Wurzel stehen so die minimalen Kosten für den gesamten Datenflussbaum.

Am Wurzelknoten wird das Nichtterminal mit den geringsten Kosten ausgewählt und die dafür vermerkte Regel betrachtet. Bei den Kinderknoten werden die Mengen der Nichtterminale auf das jeweils von der betrachteten Regel überdeckte Element

reduziert. In top-down-Reihenfolge erfolgt so für alle weitere Knoten die Reduzierung der möglichen Nichtterminale. Sind die Unterbäume eines Knotens abgearbeitet, wird die vermerkte Regel angewendet.

Der Tree-Pattern-Matching-Algorithmus stellt bei der Überdeckung eines Baums die *Knotenkonsistenz* und die *Kantenkonsistenz* her. Die von einer Maschinenoperation verwendeten und definierten Ressourcen werden nur durch die Auswahl einer Regel, die eine Maschinenoperation repräsentiert, bestimmt. Ungültige Kombinationen können bei der Überdeckung nicht auftreten, so dass die Knotenkonsistenz gewahrt wird. In der Markierungsphase erfolgt eine Überdeckung des Knotens mit Regeln, die u.a. hinsichtlich der Nichtterminale der Kinderknoten übereinstimmen. Die Kantenkonsistenz zum Eltern-Knoten wird erst durch dessen Überdeckung sicher gestellt.

Die Codeselektion durch den TPM-Algorithmus benötigt die Laufzeit $O(n)$, da in beiden Phasen jeder Knoten genau einmal betrachtet wird.

Codeselektion durch Constraintpropagierung

Dabei handelt es um ein Hilfsmittel für die Codeselektion auf Graphen und Bäumen. Jede MO erhält durch die Überdeckung mit Alternativen eine bestimmte Kombination von Registerfile-Ressourcen. Allerdings ist dabei eine sinnvolle Einschränkung auf die Kombinationen nötig, die ein alternatives Maschinenprogramm darstellen. Dies ist die Aufgabe der Constraintpropagierung.

Eine Codeselektion auf Basis der Constraintpropagierung besteht aus 3 Schritten.

1. Erzeugung der abstrakten Darstellung

   Es wird die GeLIR-Zwischendarstellung aufgebaut, z.B. aus der Lance-IR. Die prozessorspezifischen Informationen werden davon getrennt in der Architekturdarstellung abgelegt, die Programmdarstellung erfolgt maschinenunabhängig als IR.

2. Initiale Überdeckung

   Jede AMO wird mit allen alternativen MOs überdeckt, die die Operation auf dem Zielprozessor ausführen. Die Argumente und die Definition der AMO werden mit den Registerfile-Ressourcen der alternativen MOs überdeckt. Durch diesen Vorgang wird die AMO zur maschinenspezifischen MO. GeLIR stellt nun alle alternativen Maschinenprogramme dar, auch die aufgrund ihrer Registerfile-Ressourcen ungültigen Programme.

3. Einschränkung bis zur eindeutigen Programmdarstellung

   Die Constraintpropagierung beschränkt die Mengen der Registerfile-Ressourcen (Kantenkonsistenz) und der Alternativen (Knotenkonsistenz) jeder einzelnen MO auf die gültigen Kombinationen. Der erste Aufruf dieser Methode nach der initialen Überdeckung erzeugt alle alternativen Maschinenprogramme des Quellprogramms. Im weiteren Verlauf der Codeselektion wird nach jeder Entfernung einzelner Registerfiles oder alternativer MOs die Constraintpropagierung durchgeführt. Dadurch wird schrittweise die Anzahl der alternativen Maschinenprogramme verringert, bis nur

noch ein einziges, gültiges Maschinenprogramm dargestellt wird. Dies ist der Fall, wenn die Mengen aller Registerfile-Ressourcen und alternativen MOs bis auf jeweils ein Element eingeschränkt sind, und eine Zuordnung der Maschinenoperationen zu den Maschineninstruktionen erfolgt ist. Dabei sind die Ressourcen-Kombinationen, welche durch die alternativen MOs definiert werden, und die Datenabhängigkeiten zwischen den MOs einzuhalten.

Die Aufgabe der Codeselektion lässt sich so auch als schrittweise Einschränkung der Registerfile-Ressourcen und Alternativen beschreiben, die terminiert, wenn ein gültiges Maschinenprogramm vorliegt. Eine detaillierte Beschreibung der Constraintpropagierung kann [Lor03] entnommen werden, wo diese zur Realisierung eines phasengekoppelten Codegenerators für den M3-DSP verwendet wird.

Eine Codeselektion ohne Kopplung an andere Codegenerierungsphasen lässt sich sehr einfach realisieren: in jedem einzelnen Schritt der Einschränkung wird für eine MO die beste Alternative ausgewählt.

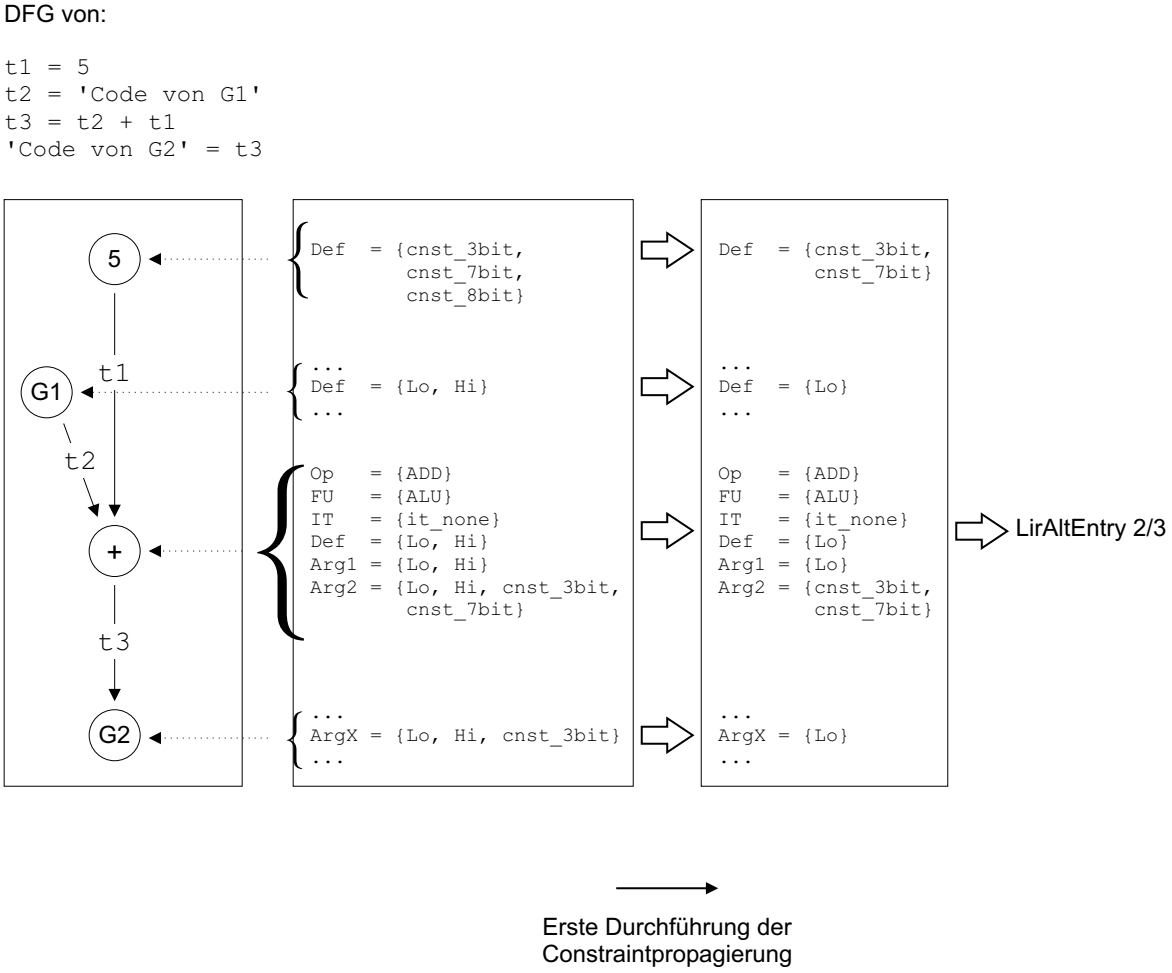

Abb. 3.4: Erste Durchführung der Constraintpropagierung nach der initialen Überdeckung am Beispiel einer Add-MO für den ARM

In Abb. 3.4 wird das Resultat der ersten Durchführung der Constraintpropagierung nach

der initialen Überdeckung dargestellt. Zuerst ergeben sich die Ressourcen der Additions-MO aus der Vereinigung der Ressourcen-Mengen aller in Abb. 2.11 (siehe Seite 30) dargestellten Alternativen. Nach der Durchführung der Constraintpropagierung ergeben sich die Ressourcen-Mengen durch die Vereinigung der *LirAltEntry*-Objekte (2) und (3). Eine Konstanten-MO enthält nur Ressourcen für die Definition des konstanten Wertes, da sie allein der Modellierung dient und keiner Operation auf dem Zielprozessor entspricht. Deshalb wurde Alternative (1), die nur die Addition von Registerinhalten darstellt, ausgeschlossen.

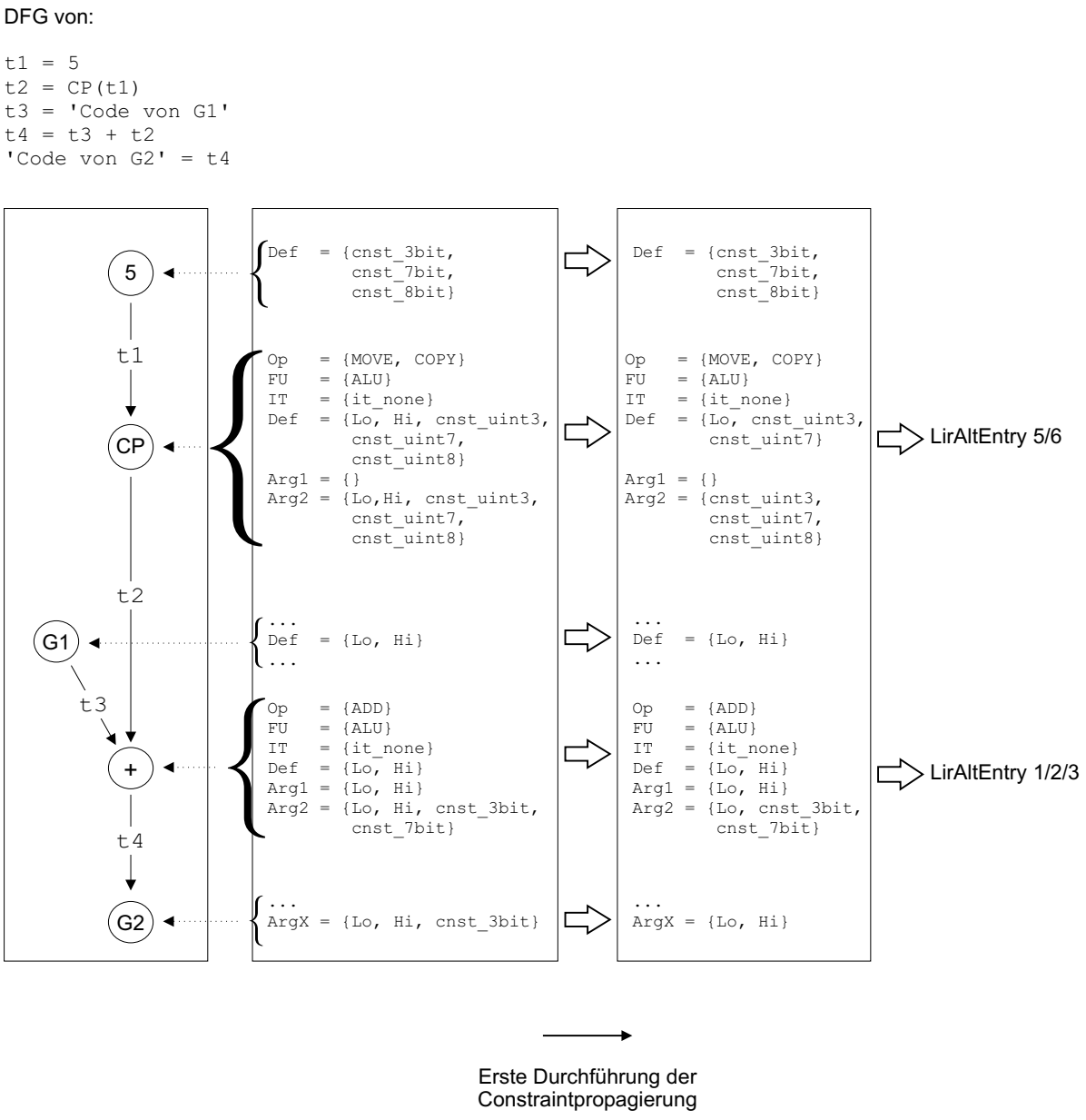

Abb. 3.5: Codeselektion am Beispiel einer Copy-MO für den ARM

In Abb. 3.5 wird die Überdeckung einer Copy-MO (siehe Abschnitt 2.3.3) veranschaulicht. Diese repräsentiert einen Datentransfer-Befehl und wird zusätzlich immer mit der Copy-Alternative überdeckt (LirAltEntry (6), siehe Abb. 3.6), die eine Maschinenoperation ohne

LirAltEntry 4:

```
Op   = {MOVE}
FU   = {ALU}
IT   = {it_none}
Def  = {Lo, Hi}
Arg1 = {}
Arg2 = {Lo, Hi}

Exec-Time = 1

Bsp: Lo = MOVE(Hi)
```

LirAltEntry 5:

```
Op   = {MOVE}
FU   = {ALU}
IT   = {it_none}
Def  = {Lo}
Arg1 = {}
Arg2 = {cnst_uint8}

Exec-Time = 1

Bsp: Lo = MOVE(255)
```

LirAltEntry 6:

```
Op   = {COPY}
FU   = {ALU}
IT   = {it_none}
Def  = {Lo, Hi, cnst_uint3,
        cnst_uint7, cnst_uint8}
Arg1 = {}
Arg2 = {Lo, Hi, cnst_uint3,
        cnst_uint7, cnst_uint8}

Exec-Time = 0

Bsp: ---
```

Abb. 3.6: *LirAltEntry*-Objekte zur Überdeckung einer Copy-MO

Funktion auf dem Zielprozessor darstellt und Kosten null hat (siehe auch Abschnitt 2.3.3 auf Seite 31). In das Beispiel aus Abb. 3.4 wurde eine Copy-MO eingefügt, um den Werte-Bereich der Konstanten durch einen evtl. eingefügten Move-Befehl auf einen 8 Bit breiten Integer zu erhöhen. Die Alternativen dafür bestehen aus einer Move-Instruktion und einer Copy-Alternative, siehe Abb. 3.6. Mit Copy-Alternativen überdeckte Copy-MOs werden nach der Codeselektion wieder entfernt. Würde in diesem Beispiel eine Konstante $127 < k \leq 255$ verwendet, stünde die Copy-Alternative nach der ersten Constraintpropagierung nicht zur Auswahl, weil sie nicht mit einem 3 oder 5 Bit breiten Integer dargestellt werden könnte.

Die Constraintpropagierung benötigt für einen Datenflussgraph mit n Knoten im Worst Case die Zeit $O(n^3 * |\text{alternative MOs}|)$. Die durchschnittliche Laufzeit ist allerdings viel geringer, da nach der Entfernung einzelner Ressourcen oder alternativer MOs während der Codeselektion nicht der gesamte Graph erneut betrachtet werden muss.

3.5.2 Implementierte Technik

Für diese Arbeit ist das einfache, additive Kostenmodell des TPM-Algorithmus ausreichend. Die konstanten Kosten einer Maschinenoperation entsprechen der Anzahl der benötigten Taktzyklen auf dem Zielprozessor, wobei angenommen wird, dass diese während der Ausführung von weiteren Faktoren, wie z.B. von der Instruktionssequenz oder vom Instruktions- bzw. Daten-Cache unabhängig sind. Der Einfluss von Cache-Hits und Cache-Misses beim LEON wird deshalb nicht berücksichtigt.

Es basieren mehrere Codeselektions-Tools auf dem TPM-Algorithmus, z.B. *iburg* [FHP92], *Twig* und *Olive* [Tji93]. Olive wurde im Rahmen dieser Diplomarbeit näher betrachtet und an einem kurzen Code-Fragment getestet. Das Ziel dabei war abzuwägen, wie groß der Aufwand für die Erstellung einer Schnittstelle zu GeLIR sein würde. Es ergab sich eine Liste von Vor- und Nachteilen für die Benutzung von Olive:

Vorteile der Verwendung von Olive:

- Bewährtes Tool

  Olive ist eine Erweiterung der Codeselektions-Tools Twig und iburg, die bereits für viele Forschungsarbeiten erfolgreich benutzt wurden.

- flexible Baumdarstellung/Schnittstelle

  Die interne Darstellung des Datenflussbaums ist nicht vorgegeben. Es existiert eine Schnittstelle in Form einiger Funktionen zur Abfrage der Kosten und der Baumstruktur.

- flexible Kosten-Berechnung

  Im Gegensatz zu dem Vorgänger iburg findet die Kostenberechnung nicht mehr durch Addition von Konstanten statt, die mit den einzelnen Regeln verknüpft sind. Stattdessen ist für jede Regel eine flexible Kostenberechnung durch Einfügung von C-Code möglich.

Nachteile der Verwendung von Olive:

- Architekturdarstellung

  Olive benötigt eine eigene Grammatik zur Darstellung des Instruktionssatzes. Sie besteht im wesentlichen aus einer Liste von Regeln, mit der die einzelnen Instruktionen sowie die dazugehörigen Kostenberechnungs- und Aktions-Vorschriften spezifiziert werden. Die GeLIR-Architekturdarstellung wird dagegen aus Objekten aufgebaut, die den Instruktionssatz und die gesamten Ressourcen des Prozessors beschreiben. Bei der gemeinsamen Verwendung mit GeLIR wären somit zwei komplette Architekturspezifikationen pro betrachteter Maschine erforderlich, die absolut gleichwertig sein müssten, was entweder die Konvertierung der GeLIR-Darstellung in eine Olive-Grammatik oder die gleichzeitige Pflege beider Architekturbeschreibungen bedeuten würde.

- Programmiersprache

  Olive erstellt den Quellcode des Codeselektors inkl. der kompletten Maschinengrammatik in C. GeLIR ist dagegen komplett objektorientiert in C++ implementiert. Es müsste eine Schnittstelle erstellt werden, um von dem C-Code der Codeselektion aus folgende Zugriffe zu ermöglichen:

  - Abfrage der Baum/Graph-Struktur der GeLIR-Programmdarstellung
  - evtl. Abfrage der Kosten, falls diese nicht statisch in der Olive-Grammatik abgelegt werden
  - (Rück-)Konvertierung der ausgewählten Regeln zu alternativen MOs
  - Auswahl der MOs in der GeLIR-Programmdarstellung
  - Einfügen zusätzlicher Datentransfer-Befehle in Form von Copy-MOs

- Programm-Darstellung

  GeLIR verwendet eine graphbasierte Darstellung der AMOs. Für die Codeselektion mit Olive müsste der DFG eines Basisblocks z.B. an den Common Sub Expressions (CSEs) in Bäume aufgeteilt werden, indem die Verwendungen der CSEs durch eine Variable ersetzt werden und der Teilgraph zur Berechnung der CSE in einem getrennten Durchlauf betrachtet wird. Dies stellt keine Einschränkung dar, hätte aber einen größeren Aufwand bei der Implementierung der Schnittstelle zwischen Olive und GeLIR zur Folge.

Die Hauptgründe gegen die Verwendung von Olive waren die benötigte doppelte Architekturdarstellung und der Aufwand für die Implementierung der Schnittstelle zwischen den GeLIR-Klassen in C++ und dem generierten Codeselektor in C. Eine eigene Implementierung der Olive-Funktionalität auf der GeLIR-Programmdarstellung schien die bessere Lösung. Dazu wurde der TPM-Algorithmus um eine Knotenmarkierung ergänzt, die eine doppelte Betrachtung von Knoten des DFGs ausschließt. Für Architekturen mit homogenen Registersätzen, wie bei RISC-Prozessoren, führt dieser Algorithmus eine global optimale Codeselektion auf Graphen durch [Ert99]. Nachfolgend wird der für die Codeselektion auf Graphen modifizierte Tree-Pattern-Matching-Algorithmus mit mit TPM_{DFG} bezeichnet.

Die GeLIR-Zwischendarstellung unterstützt ein schrittweises Vorgehen bei der Codeselektion durch die Constraintpropagierung (siehe Abschnitt 3.5.1). Für deren Verwendung sind einige Überlegungen hinsichtlich der Anwendung des TPM_{DFG}-Algorithmus für RISC-Architekturen nötig:

- Knoten-Betrachtungsreihenfolge in der Markierungs-Phase

  Die bottom-up Betrachtungsreihenfolge der Knoten in der Markierungs-Phase ist nicht umkehrbar. Für die Kostenberechnung des Teilgraphen, dessen Wurzel der aktuell betrachtete Knoten bildet, sind die Kosten zur Verwendung der Nichtterminale der Kinder-Knoten erforderlich. D.h. diese müssen bereits betrachtet worden sein.

- Knotenbetrachtungsreihenfolge in der Reduzierungs-Phase

  In der Reduzierungs-Phase erfolgt im allgemeinen Fall die Betrachtung der Knoten von der Wurzel zu den Blättern. Es wird davon ausgegangen, dass die Überdeckung eines Teilgraphen abhängig von der Überdeckung des Eltern-Knotens ist. Können Teilgraphen dagegen unabhängig von dem Eltern-Knoten überdeckt werden, z.B. weil eine ein-elementige Menge von Nichtterminalen in der Reduzierungs-Phase betrachtet wird, so muss diese Reihenfolge nicht eingehalten werden.

- Optimale lokale Überdeckung

  Falls eine optimale *lokale Überdeckung* möglich ist, d.h. bei der ersten Betrachtung eines Knotens in der Markierungs-Phase entschieden werden kann, welche Regel in der Reduzierungs-Phase ausgewählt wird, dann kann der TPM_{DFG}-Algorithmus in einer einzigen Phase durchgeführt werden. Dafür müssen die drei Ressourcenklassen von RISC-Architekturen näher betrachtet werden:

  1. Universelle Register

     Bei RISC-Architekturen ist dies die überwiegend verwendete Registerfile-Ressource. Nur bei Zugriff auf den Speicher, Verwendung von Immediate-Werten als Argument oder bei komplexen MOs werden weitere Ressourcen benötigt.

  2. Flüchtige Ressourcen

     Diese Ressourcen werden verwendet um komplexe MOs zu modellieren (siehe Abschnitt 2.3.4 auf Seite 31. Da sich diese aus mindestens zwei Maschinenoperationen zusammensetzen, müssen die Kosten der Instruktion auf die einzelnen Regeln der alternativen MOs verteilt werden.

3. Speicherressourcen

   In diese Ressource wird nur von Store-Befehlen geschrieben und aus ihr wird nur von Load-Befehlen gelesen.

Wenn eine Maschinenoperation nur eine einzige Registerfile-Ressource definieren kann, führt eine lokal optimale Überdeckung des Teilgraphen offensichtlich auch zu einer global optimalen Überdeckung, da sie keinen Einfluss auf die Überdeckung des Eltern-Knotens hat. Andernfalls müssen die Kosten von komplexen MOs so auf die einzelnen Maschinenoperationen verteilt sein, dass alle möglichen Überdeckungen des Eltern-Knotens gleiche Kosten haben. In dem Beispiel `a := 5` aus Abb. 3.2 auf Seite 38 ist dies bei dem Knoten mit dem Terminal + nicht der Fall. Die Regeln (3)-(6) (siehe Abb. 3.3 auf Seite 39) sind von den Kosten her anscheinend gleichwertig. Die Auswahl der Regel (5) oder (6) verursacht aber höhere Kosten bei der Überdeckung des Eltern-Knotens. Werden die Kosten wie in Abb. 3.7 verteilt, kann am Knoten + bereits eine global optimale Teilüberdeckung ausgewählt werden. So kann der TPM$_{DFG}$-Algorithmus für RISC-Architekturen auf einen einzigen Graph-Durchlauf reduziert werden.

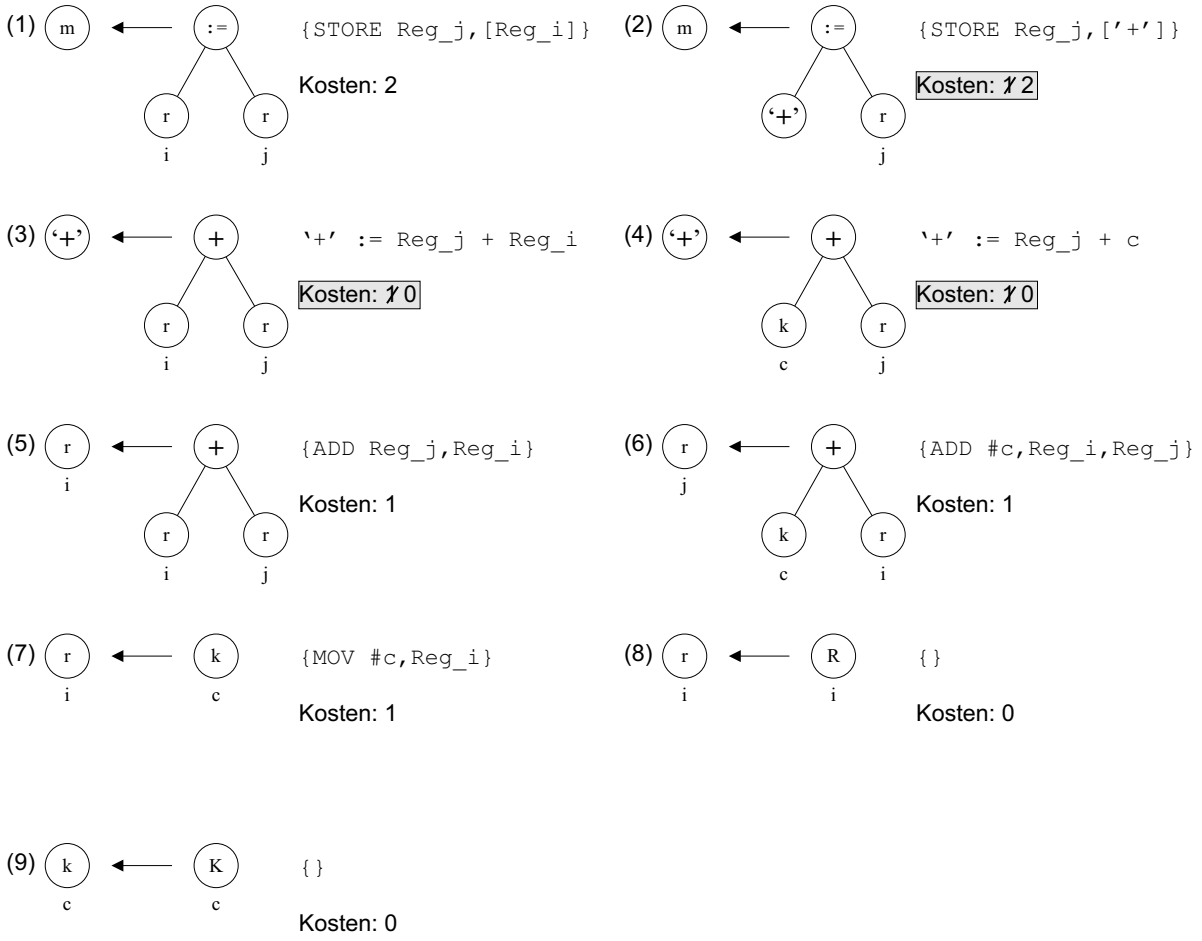

Abb. 3.7: Angepasste Baumtransformationsregeln für das Beispiel aus Abb. 3.2

Die Durchführung der eigentlichen Codeselektion erfordert mehrere Vorbereitungsschritte:

- Einfügen potentieller Datentransfer-Befehle

  Diese werden im DFG zwischen MOs, die Konstanten bzw. Speicheradressen repräsentieren, und deren Eltern-Knoten jeweils in Form einer Copy-MO eingefügt. Sie dienen der Modellierung unterschiedlicher Befehle zum Laden von Konstanten. Für den ARM-Prozessor sind nachfolgend die für die Codeselektion relevanten Alternativen der Copy-MO aufgeführt:

  - Copy-Alternative: zur Verwendung der Konstanten als Immediate-Wert
  - Move-Befehle: zum Laden von kleinen Konstanten mit $0 \leq k < 256$
  - PC-relativer Load-Befehl: zum Laden von Adressen und großen Konstanten mit $0 \leq k < 2^{32}$

- Initiale Überdeckung

  Dies ist die Überdeckung der AMOs mit allen Alternativen, die die jeweilige Operation auf dem Zielprozessor ausführen. Dadurch werden alle alternativen Maschinenprogramme erzeugt.

- Erster Aufruf der Constraintpropagierung

  Dadurch wird die Darstellung auf die gültigen Ressourcen-Kombinationen begrenzt. Jede MO wird jetzt nur noch von den Alternativen überdeckt, die jeweils in mindestens einem alternativen Maschinenprogramm vorkommen.

Nach diesen Vorbereitungen wird der TPM_{DFG}-Algorithmus durchgeführt. Dabei handelt es sich um die auf einen einzigen Graph-Durchlauf reduzierte Version. Diese wurde unter Verwendung der Constraintpropagierung von GeLIR implementiert, so dass als einzige Aufgabe die Auswahl der jeweils günstigsten Alternative für eine AMO bleibt. Die MOs eines Datenflussgraphen werden von den Blättern hin zu den Wurzel betrachtet, die Codeselektion erfolgt für jede AMO in drei Schritten:

1. Bestimmung der alternativen MO mit den geringsten Kosten.
2. Entfernung der alternativen MOs, bis auf die im 1. Schritt ausgewählte Alternative.
3. Aufruf der Constraintpropagierung, damit die Alternativen und die Registerfile-Ressourcen aller MOs des DFGs auf die nun gültigen Kombinationen eingeschränkt werden.

Zur Nachbereitung der Codeselektion werden als erstes die überflüssigen Datentransfers, d.h. die mit Copy-Alternativen überdeckten Copy-MOs, entfernt. Anschließend werden MOs, die flüchtige Ressourcen benutzen, zu komplexen MOs (siehe Abschnitt 2.3.4) zusammengefasst.

3.6 Instruktionsanordnung

Im Rahmen dieser Arbeit wurde keine Instruktionsanordnung (IA) implementiert. Sie kann aber zu einem späteren Zeitpunkt einfach in den Compiler integriert werden. Es wird während der gesamten Codegenerierung die von der LANCE-IR übernommene Anweisungsabfolge betrachtet. Das Hauptziel einer IA für RISC-Architekturen ist die Minimierung gleichzeitig zu verwendender Register (Registerdruck), damit in der nachträglich durchzuführenden Registerallokation weniger Werte in den Speicher ausgelagert werden müssen. Ein einfaches Verfahren, wie es z.B. auch beim ENCC verwendet wird, basiert auf der Unterscheidung von drei Arten von Befehlen:

1. Anweisungen, die den Registerdruck erhöhen: Dies sind z.B. das Laden einer Konstanten in ein Register oder das Laden aus einem SP-relativ adressierten Speicherplatz. In beiden Fällen wird nach der Ausführung der Instruktion ein Register mehr benötigt als vorher. Diese Anweisungen sind in der Ausführungsreihenfolge nach hinten zu verschieben.

2. Anweisungen, die den Registerdruck verringern: Dies sind z.B. Store-Befehle oder arithmetisch-logische Operationen mit zwei Registern als Argumente. Diese Anweisungen sind nach vorne zu verschieben.

3. Anweisungen, die den Registerdruck nicht verändern: Sie können beliebig verschoben werden, um eine bessere Anordnung der beiden anderen Arten von Anweisungen (siehe (1.) und (2.)) zu ermöglichen.

Die Verschiebung der Anweisungen erfolgt jeweils soweit dies aufgrund der Datenabhängigkeiten zulässig ist.

3.7 Registerallokation

Die Aufgabe der Registerallokation ist die Abbildung der Variablen auf physikalische Register und Speicherplätze. Da die Verwendung der Speichers aufgrund der erforderlichen Load-/Store-Befehle höhere Kosten verursacht als die Benutzung von Registern, ist dabei das Zwischenspeichern von Registerinhalten im Speicher zu minimieren. Bei den Variablen wird zwischen *globalen* und *lokalen* Variablen unterschieden:

- *Lokale Variablen* sind nur während der Ausführung der Funktion, in der sie definiert werden, gültig. Um bei verschachtelten Funktionsaufrufen die Gültigkeit der lokalen Variablen zu gewährleisten, kann jeder Funktion ein Bereich im Speicher (Stack-Frame) zugewiesen werden, welcher über den SP (Stack Pointer) referenziert wird. Nach dem Funktionsaufruf wird durch eine Verringerung des SP-Registers der Stack-Frame erzeugt, vor dem Rücksprung durch Wiederherstellung des ursprünglichen Inhalts des SP-Registers wieder entfernt.

- *Globale Variablen* existieren in genau einer Instanz und sind von allen Funktionen aus zugreifbar. Sie werden im Datensegment des Speichers abgelegt und über absolute Speicheradressen angesprochen. Bei RISC-Prozessoren erfolgt der Zugriff in zwei Schritten: Zuerst wird die Adresse geladen, dann wird auf die globale Variable per Load-/Store-Befehl zugegriffen.

3.7.1 Bestehende Techniken

Es wird zwischen globalen und lokalen RA-Verfahren unterschieden. Sie unterscheiden sich durch die betrachteten Programmausschnitte: Die globalen Verfahren führen die RA für alle Basisblöcke einer Funktion in einem Schritt durch, lokale für jeden Basisblock separat. In diesem Abschnitt wird jeweils ein RA-Verfahren vorgestellt, der Graph-Färbe- und der Leftedge-Algorithmus. Eine detailliertere Beschreibung der beiden Verfahren kann [Tei97] entnommen werden.

Graph-Färbe-Algorithmen

Graph-Färbe-Algorithmen sind die zur Zeit verbreitesten Registerallokations-Verfahren mit gekoppelter globaler und lokaler RA, und finden in den meisten modernen Compilern Verwendung. Sie ermöglichen die Durchführung einer optimalen Registerallokation, erfordern aber einen großen Implementierungsaufwand. Unter Annahme, dass die Codeselektion von einer unbeschränkten Anzahl symbolischer Register ausgeht, besteht das Problem der Registerbindung in der Bindung der symbolischen an die physikalischen Register, wobei Zwischenspeicherungen im Speicher zu minimieren sind. Dieses Problem lässt sich auch als Färbungsproblem beschreiben. Die Grundlage für solche Verfahren ist der *Registerkonfliktgraph.*

Definition 3.1 (Registerkonfliktgraph) *Ein Registerkonfliktgraph G = (V,E) bezeichnet einen ungerichteten Graphen, in dem die Knotenmenge V die Menge der symbolischen Register darstellt. Die Kantenmenge* $E = \{\{v_i, v_j\} : v_i \neq v_j, v_i, v_j \in V, i \neq j\}$ *drückt die Konfliktrelation der Register aus:* $v_i \neq v_j$ *bedeutet, dass sich die Lebenszeitspannen von* v_i *und* v_j *überschneiden.*

Das Problem der Registervergabe und Registerbindung kann als k-Färbungsproblem des Konfliktgraphen aufgefasst werden, wobei k die Anzahl der physikalischen Register beträgt. Es wird versucht, den Konfliktgraphen mit k Farben einzufärben, so dass benachbarte Knoten unterschiedliche Farben bekommen. Das zugehörige Entscheidungsproblem ist für allgemeine Graphen NP-vollständig. Durch eine Heuristik können Knoten entfernt werden, bei denen eine Farbzuweisung offensichtlich möglich ist. Dies sind die Knoten $v_i \in V : deg(v_i) \lessdot k$, wobei $deg(v_i)$ den Grad (engl: Degree) des Knotens v_i angibt. Wenn der Graph k-färbbar ist, lassen sich dadurch nacheinander alle Knoten des Konfliktgraphen entfernen. Ist eine Graphfärbung nicht möglich, d.h. es bleiben nach wiederholter Anwendung der Heuristik Knoten $v_i \in V : deg(v_i) \geq k$ übrig, müssen Registerinhalte zwischengespeichert werden. Nach dem Einfügen von Spill-Code in den Programmcode ist eine Anpassung des Konfliktgraphen nötig.

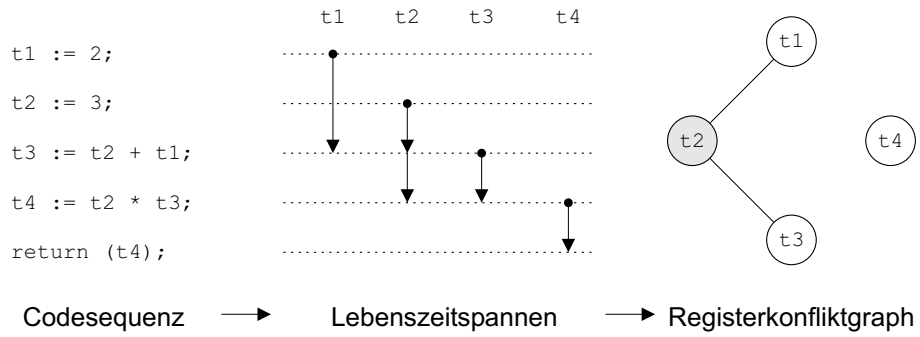

Abb. 3.8: Erstellung des Registerkonfliktgraphen aus der Codesequenz

Beispiel: Abb. 3.8 zeigt für ein Beispielprogramm den Registerkonfliktgraphen. Jede MO entspricht einem Zeitpunkt während der Programmausführung (horizontale, gestrichelte Linie). Falls sich die Lebenszeitspannen zweier Variablen zeitlich überschneiden, existiert im Registerkonfliktgraphen eine Kante zwischen den entsprechenden Knoten. Für die Färbung des Graphen werden zwei Farben benötigt.

Leftedge-Algorithmus

Beim Leftedge-Algorithmus handelt es sich um ein einfaches, lokales RA-Verfahren. Wie beim Graph-Färbe-Algorithmus wird vorausgesetzt, dass bei der Codeselektion von einer unbeschränkten Anzahl symbolischer Register ausgegangen wurde. Die Grundlage des Leftedge-Algorithmus ist der *Intervallgraph.*

Definition 3.2 (Intervallgraph) *Ein ungerichteter Graph $G = (V, E)$ heißt Intervallgraph, falls man jedem Knoten $v_i \in V$ ein Intervall $[l_i, r_i)$ zuordnen kann, mit $l_i, r_i \in \mathbb{Z}$ und $l_i < r_i$, so dass die Kante $\{v_i, v_j\}$ genau dann existiert, wenn sich die Intervalle $[l_i, r_i)$ und $[l_j, r_j)$ überlappen.*

Ein Intervallgraph ist auch ein Registerkonfliktgraph. Die Knoten repräsentieren die Variablen, die Intervalle entsprechen den Lebenszeitspannen. Die Darstellung von Intervallen erlaubt allerdings nur die Berücksichtigung von sequentiell ausführbaren Anweisungen. Registerkonflikte können nicht über Kontrollflussverzweigungen hinaus betrachtet werden, weshalb eine auf Intervallgraphen beruhende Registerallokation jeweils nur einen einzelnen Basisblock betrachtet.

Sei I die Menge der zu betrachtenden Intervalle mit $I_i = [l_i, r_i)\ \forall i = 1, \ldots, \mid V \mid$ und die Funktion $c : V \rightarrow \mathbb{N}$ die Graphfärbung, die jedem Knoten $v \in V$ eine Farbe $c(v)$ zuweist.

LEFTEDGE(I)
{
    Sortiere Elemente von I in Liste L in aufsteigender Reihenfolge von l_i;
    $j := 0$;
    WHILE ($L \neq \{\}$)
    {
        $S := \{\}$;

```
        r := 0; /* init. größte rechte Grenze von Elementen in S */
        WHILE (∃ Element I_i in L : l_i ≥ r)
        {
            i :=Element in L mit kleinstem l_i und l_i ≥ r;
            S := S ∪ {I_i};
            r := r_i;
            L := L \ {I_i};
        }
        j := j + 1;
        FOR ALL (I_i ∈ S)
            c(I_i) := j;
    }
    RETURN (c(I));
}
```

Zuerst werden die Intervalle aufsteigend nach deren linken Intervallgrenzen sortiert, daher die Bezeichnung *Leftedge*. Entsprechend dieser Intervall-Reihenfolge werden die korrespondierenden Knoten gefärbt. Dabei wird jeweils der kleinste Index gewählt, der nicht bereits überlappenden Intervallen zugewiesen ist.

Der Leftedge-Algorithmus führt eine optimale Registerallokation auf sequentiellem Code ohne Kontrollfluss-Verzweigungen durch, d.h. für einen einzelnen Basisblock. Dabei wird die Zeit O(n log n) mit $n = |V|$ benötigt. Neben der geringen Laufzeit ist ein weiterer Vorteil dieses RA-Verfahrens der im Vergleich zu Graph-Färbe-Verfahren geringe Implementierungsaufwand.

Da bei dem Beispiel in Abb. 3.8 kein bedingter Sprung vorkommt und alle Lebenszeitspannen der Variablen Intervalle sind, kann darauf der Leftedge-Algorithmus durchgeführt werden. Die Knoten werden dann in der Reihenfolge *t1, t2, t3, t4* markiert.

3.7.2 Implementierte Technik

Bei der Implementierung der Registerallokations-Techniken waren Wiederverwendbarkeit und eine einfache Realisierung die Hauptkriterien. Deshalb wurde eine Trennung der globalen und lokalen RA vorgenommen.

Es werden eine globale und eine lokale RA getrennt voneinander durchgeführt. Für beide Verfahren wurde jeweils ein einfacher, effektiver Algorithmus implementiert. Die Vorteile der Trennung sind das mögliche schrittweise Vorgehen bei der Implementierung und die später bessere Wiederverwertbarkeit in anderen Codegeneratoren. Die globale RA betrachtet keine Anweisungen der IR/LIR, sondern nur den globalen Datenfluss einer Funktion, d.h. die Variablen aus V_{GlobIn} und $V_{GlobOut}$ der Basisblöcke. Als weitere Eingabe werden die zu vergebenden physikalischen Register benötigt, die in der GeLIR-Architekturdarstellung abgelegt sind. Die globale RA analysiert die Datenabhängigkeiten der Variablen des globalen Datenflusses und bindet die Variablen aus V_{GlobIn} und $V_{GlobOut}$ an Register bzw. Speicherplätze. Dabei wird für jeden Basisblock eine Menge der *reservierten Register* erstellt, die die Register und Speicherplätze enthält, die während der Ausführung des Basisblocks nicht überschrieben werden dürfen. Die lokale RA wird dann anschließend unter Berücksichtigung der gebundenen Variablen aus V_{GlobIn} bzw. $V_{GlobOut}$

und der reservierten Register durchgeführt. Diese globale RA ist flexibel einsetzbar. Da sie keine Anweisungen in den Basisblöcken betrachtet ist sie maschinenunspezifisch und kann vor oder nach der Codeselektion durchgeführt werden.

Die getrennte Durchführung der RA hat auch einen Nachteil. Die Qualität der Registervergabe ist potentiell schlechter als Verfahren mit gekoppelter globaler und lokaler RA. Die Variablen des globalen Datenflusses werden, vor den lokal in Basisblöcken verwendeten, statisch vergeben. Werden dabei zu viele Variablen an (universelle) Register vergeben, werden bei der lokalen RA vermehrt Zwischenspeicherungen erzwungen. Werden dagegen zu wenige an universelle Register vergeben, entstehen unnötige Kosten für den Speicherzugriff. Diese Problematik tritt bei gekoppelten Verfahren, z.B. beim Graph-Färbe-Algorithmen, nicht auf, da sie die Registervergabe global optimieren.

Die beiden RA-Techniken wurden unter zwei Hauptgesichtspunkten entwickelt:

1. Einfache Implementierung

   Die Forschung auf dem Gebiet der Registerallokation betrachtet fast ausschließlich Verfahren mit gekoppelter globaler und lokaler RA. Das Wichtigste ist das Graph-Färbe-Verfahren, was aber einen erheblich größeren Implementierungsaufwand erfordert als ein einfacher Algorithmus. Deshalb wurde im Rahmen dieser Diplomarbeit ein RA-Algorithmus entwickelt, der auf Standard-Graph-Algorithmen beruht.

2. Generische Durchführung

   In der globalen und lokalen RA sollten sowenig Informationen über die RISC-Architektur enthalten sein, um sie in anderen Compilern wiederverwenden zu könne. Die lokale RA setzt bestimmte Archtektureigenschaften voraus, wie z.B. eine Load-/Store-Architektur und einen homogenen Registersatz. Für die globale RA gibt es keine derartigen Einschränkungen. Wie beispielhaft für den M5-DSP gezeigt, kann dieses Verfahren auch in Compilern für andere Klassen von Prozessoren verwendet werden.

Globale RA

Zuerst wird die globale RA betrachtet. Deren Aufgaben sind:

1. Globale Registervergabe/-bindung:

   In allen Basisblöcken einer Funktion müssen die Variablen V_{GlobIn} und $V_{GlobOut}$ entweder an ein Register oder einen Speicherplatz gebunden werden.

2. Register/Speicherplatz-Reservierung:

   Die an Variablen des globalen Datenflusses gebundenen Register bzw. Speicherplätze dürfen während ihrer Lebenszeitspanne bei der lokalen RA nicht neu definiert werden, außer die Inhalte werden zuvor im Speicher gesichert und danach wieder geladen. Dazu wird bei der globalen RA für jeden Basisblock die Menge der reservierten Register bestimmt, die alle zu dem Zeitpunkt aktiven Variablen enthält.

Der implementierte Algorithmus der globalen RA benötigt folgende Eingaben:

- Kontrollflussgraph (CFG)

  Der CFG einer Funktion kann von einem *LirFun*-Objekt abgefragt werden, wobei die Knoten dieses Graphen *LirBB*-Objekte sind.

- Ziel-Register/Speicherplätze

  Es wird eine Menge von verfügbaren Registern bzw. Speicherplätzen benötigt, an die die Variablen bei der globalen RA gebunden werden können. In Frage kommen die universellen Register, über Move-Befehle ansprechbare Register, wie z.B. die High-Register des ARM, und SP-relativ adressierbare Speicherplätze. Absolut adressierbare Speicherplätze sind ausgeschlossen weil sie bei rekursiven Funktionen evtl. gesichert werden müssen und in diesem Fall einen weiteren Befehl zum Laden der absoluten Adresse erfordern würden.

  Es wurde eine Erweiterung der GeLIR-Architekturdarstellung vorgenommen, um die für die Registerallokation verwendbaren Registerfile-Elemente abzulegen. Die Erweiterung wird in Abschnitt 3.9 näher beschrieben.

Sei der gerichtete Graph $G = (V, E)$ der CFG der Funktion, wobei V die Menge der enthaltenen Basisblöcke und E die Menge der möglichen Kontrollflusswege darstellt. Es gibt eine gerichtete Kante $e = (v_i, v_j) \in E$ genau dann, wenn BB_j in der Ausführungsreihenfolge direkt auf BB_i folgt. Dies ist der Fall wenn a) die letzte Anweisung in BB_i ein bedingter oder unbedingter Sprung, mit BB_j als Sprungziel, ist, oder b) BB_j direkt auf BB_i folgt und BB_i keinen unbedingten Sprung enthält. G ist ein zusammenhängender und evtl. zyklischer Graph. Der Algorithmus für die globale RA wird für jede Variable des globalen Datenflusses var_n getrennt durchgeführt, nachfolgend wird ein einzelner Durchlauf betrachtet.

1. Der Graph $G_M = (V, E_M)$ mit $E_M = E \setminus \{ (v_i, v_j) \in E \mid$ in BB_j wird var_n definiert, aber nicht verwendet$\}$ wird erstellt. Aus G werden also die Kanten entfernt, über die der Wert der betrachteten Variable var_n nicht übertragen werden kann, da in BB_j dieser Wert überschrieben wird. Somit werden auf dem CFG alle Pfade zerstört, auf denen var_n neudefiniert wird. Der Graph G_M ist im Gegensatz zu G nicht mehr unbedingt zusammenhängend.

2. Es wird V_S, die Menge der SZKs (Starke Zusammenhangskomponente) von G_M berechnet [WHFS02]. Daraus wird der Graph $G_S = (V_S, E_S)$ erstellt, wobei $E_S = \{(w_k, w_l) \subseteq (V_S \times V_S) \mid$ es gibt einen BB in w_l, der in der Ausführungsreihenfolge direkt auf einen BB aus w_k folgt$\}$ ist. G_S ist azyklisch. Es sei $V_{Start} = \{x \in V_S \mid var_n$ wird in x definiert, aber nicht verwendet$\}$.

3. Für alle SZKs $v \in V_S$ wird in diesem Schritt der Wert von $reserve(v)$ bestimmt. $reserve(v)$ hat den Wert $TRUE$, falls die Variable var_n in der SZK v oder in einer im Kontrollfluss folgenden SZK verwendet wird. Dazu wird folgender DFS-Algorithmus auf G_S durchgeführt.

   $DFS(G_S)$ für $G_S = (V_S, E_S)$
   $\{$
   $\quad i := 0;$

FOR ALL $x \in V_S : num(x) := 0, reserve(x) := false$;
FOR ALL $x \in V_{Start} : DFS(x, 0)$
}
$DFS(v, u)$
{
$i := i + 1; num(v) := i$;
FOR ALL $w \in Adj(v)$ mit $w \neq u$:
{
if $num(w) = 0$ then $DFS(w, u)$;
$reserve(v) := reserve(v)\ OR\ reserve(w)$;
}
$reserve(v) := reserve(v)\ OR$ {in v gibt es mindestens einen BB, in dem var_n verwendet oder definiert wird};
}

4. Sei $V_R = \{x \in V_S \mid reserve(x) = true\}$.

5. Sei *Reg* die Menge aller für die globale RA verfügbaren Register und Speicherplätze, absteigend nach ihrer Priorität sortiert:
FOR ALL $V_k \in V_R$:
FOR ALL $BB_i \in V_k$:
Entferne alle in BB_i verwendeten Register/Speicherplätze aus *Reg*
Sei *Register r* das erste Register aus *Reg*.

6. FOR ALL $V_k \in V_R$:
FOR ALL $BB_i \in V_k$:
IF $var_n \in \{V_{GlobIn} \cup V_{GlobOut}\}$
binde var_n in BB_i an *Register r*
ELSE
füge var_n in die Liste der reservierten Variablen von BB_i ein

In Abb. 3.9 wird das Vorgehen bei der globalen RA an einem Beispiel dargestellt. In den Graphen von G und G_M ist eingetragen, in welchen Basisblöcken die Variable x verwendet bzw. definiert wird. Bei der Erzeugung von G_M werden aus G die Kanten zu den Basisblöcken (1) und (5) nicht übernommen, da dort die Variable x definiert aber nicht verwendet wird. Dies sind die einzigen Kanten, über die ein Datenfluss bzgl. x auszuschließen ist. Die grauen Felder zeigen die SZKs von G_M, wobei die SZKs die die Basisblöcke (1) und (5) enthalten in die Menge der Start-SZKs aufgenommen werden. Auf dem Graph G_S wird dann die eigentliche Registerallokation durchgeführt.

Die Betrachtungsreihenfolge der Variablen hat dabei großen Einfluss auf die Codequalität. Denn die zuerst bearbeiteten Variablen werden an Register gebunden, später betrachtete werden dagegen evtl. an Speicherplätze auf dem Stack gebunden. Deshalb werden zuerst werden die Funktions-Parameter betrachtet, da diese gemäß Programmier-Konvention in vorgegebenen Registern übergeben werden. Danach werden die übrigen Variablen betrachtet, wobei diesen unterschiedliche Prioritäten zugewiesen werden. Durch die Verwendung mehrerer Algorithmen zur Datenflussanalyse aus [Hor01b] ist eine einfache Klassifikation der Basisblöcke im CFG möglich. Damit wird für alle Basisblöcke und somit auch für

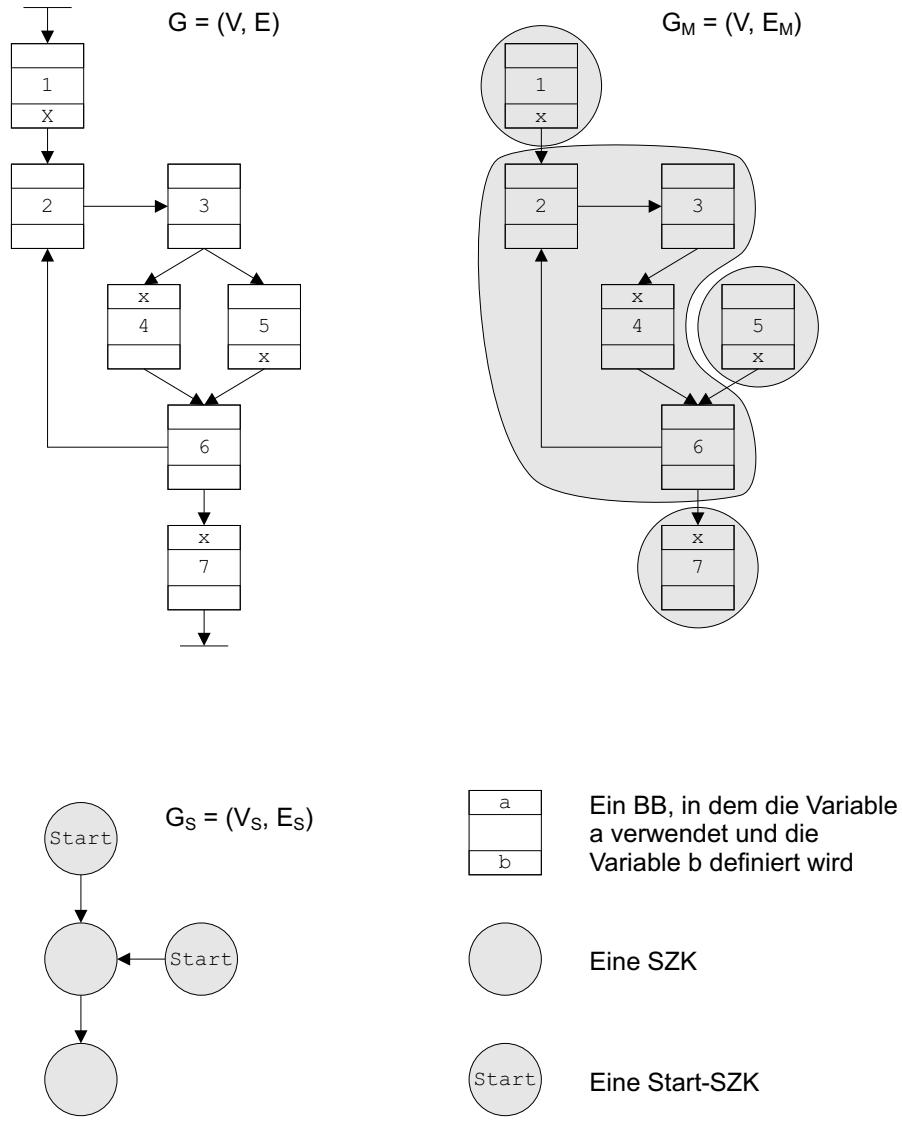

Abb. 3.9: Beispiel für die globale Registerallokation

alle Variablen festgestellt, ob sie in einer inneren, äußeren oder in keiner Schleife liegen. Die Betrachtung der Variablen des globalen Datenflusses einer Funktion erfolgt in dieser Reihenfolge:

1. Funktions-Parameter
2. Variablen, die u.a. in inneren Schleifen verwendet werden
3. Variablen, die u.a. in äußeren Schleifen verwendet werden
4. übrigen Variablen

Die Funktions-Parameter werden gemäß den Programmier-Konventionen in bestimmten Registern und Speicherplätzen auf dem Stack erwartet. Damit diese nicht bereits zuvor an andere Variablen gebunden werden, erfolgt die Betrachtung der weiteren Variablen

anschließend. Variablen, die in Schleifen verwendet oder definiert werden, sollten nach Möglichkeit an Register gebunden werden, damit der Code für den Speicherzugriff nicht mit jedem Schleifendurchlauf ausgeführt wird. Aus diesem Grund werden sie vor den übrigen, nicht in Schleifen benutzten Variablen, an Register gebunden.

Der Algorithmus für die globale RA wurde auch erfolgreich bei der Codegenerierung für den M5-DSP (Nachfolgearchitektur des M3-DSP) verwendet und steht als eigenständige GeLIR-Klasse allgemein zur Verfügung. In Kapitel 4 sind einige Resultate des M5-Codegenerators dargestellt.

Lokale Registerallokation

Die lokale RA wurde durch den Leftedge-Algorithmus realisiert. Er benötigt folgende Eingaben:

1. DFG des zu betrachtenden Basisblocks
2. $V_{GlobIn} + V_{GlobOut}$

   Diese Variablen sind bereits an Register bzw. Speicherplätze gebunden. Zur Verwendung müssen evtl. Load-/Store- oder Move-Befehle eingefügt werden.
3. Menge der universellen Register

   An diese Register werden die lokalen Variablen gebunden.
4. Menge der verfügbaren Register & Speicherplätze für Spilling

   Falls nicht genügend universelle Register verfügbar sind, XXX werden Werte ...welche?XXX zwischengespeichert. Durch den Spill-Code werden sie aus den Registern gelesen und bei Verwendung wieder zurückgeschrieben.
5. Menge der reservierten Register

   Diese Register und Speicherplätze enthalten Variablen des globalen Datenflusses. Werden sie bei der lokalen RA benötigt, sind sie durch Spill-Code zwischenzuspeichern.

Die Voraussetzung für die lokale RA ist, dass nur eine Maschinenprogramm-Alternative vorliegen darf, d.h. die CS muss so durchgeführt worden sein, dass für die Argumente und die Definition einer MO jeweils nur noch eine Registerfile-Ressource zur Auswahl steht.

Die lokale RA erzeugt folgende Ausgabe:

1. Load/Store-Bindung an Register und Speicher
2. restliche MOs: Bindung der Argumente und der Definition (falls nötig) an jeweils ein Register
3. Spill-Code, der aus CP-MO-Paaren besteht. Diese werden durch die Befehls-Kombinationen Store-Load, aber auch Mov-Mov (ARM) überdeckt.

Die lokale RA wird nach der globalen ausgeführt. Sie betrachtet jeweils nur einen Basisblock, also eine sequentielle Abfolge von Befehlen. Da einige Variablen bereits durch die globale RA gebunden sind, musste der Leftedge-Algorithmus etwas modifiziert werden. Die dabei zu berücksichtigen Variablen sind zum einen die Variablen des globalen Datenflusses, V_{GlobIn} und $V_{GlobOut}$ die in der globalen RA gebunden wurden, und zum anderen die Operanden einiger Maschinenbefehle, wie z.B. Call- und Return-Instruktionen, die in bestimmten Registern erwartet werden. Vor der Durchführung des modifizierten Leftedge-Algorithmus werden CP-MOs eingefügt, um für an gleiche Register zu bindende Variablen die Überschneidung der Lebenszeitintervalle zu vermeiden. Die CP-MOs stellen wieder potentielle Datentransfers dar und werden wie folgt eingefügt:

- V_{GlobIn} und $V_{GlobOut}$

  Es werden jeweils zwei CP-MOs eingefügt. Diese werden entsprechend der Ressource der Variable überdeckt: ist die Variable an ein Register gebunden, werden die beiden CP-MOs mit zwei Mov-Befehlen überdeckt, ist sie an einen Speicherplatz gebunden, wird eine CP-MO mit einem Mov-Befehl und die andere mit einem Load bzw. Store-Befehl überdeckt.

  Bei Variablen aus V_{GlobIn} wird eine CP-MO am Anfang des Basisblocks eingefügt, d.h. vor der ersten ursprünglichen MO, und eine vor der ersten MO, die die betrachtete Variable als Operanden hat. Bei Variablen aus $V_{GlobOut}$ wird eine nach der letzten Anweisung, die die Variable definiert, eingefügt und eine nach der letzten Nicht-Sprung-Anweisung des Basisblocks. Falls die letzte Nicht-Sprung-Anweisung des Basisblocks ein Compare-Befehl ist, wird sie vor diesen eingefügt. Denn Move- und Load/Store-Befehle einiger RISC-Architekturen schreiben in das Status-Register, das das Ergebnis des Compare-Befehls enthält.

- Anweisungen

  Die Parameter für Funktions-Aufrufe und der Rückgabewert bei Funktions-Rücksprüngen werden in bestimmten Registern erwartet, gemäß Programmier-Konvention oder Architekturaufbau (siehe Abschnitt 2.1). Für jedes Argument einer solchen MO wird eine CP-MO eingefügt, ebenso für die Definition (falls vorhanden). Die Überdeckung erfolgt mit Move-Befehlen.

Bei der lokalen RA wird die Anzahl der später benötigten CP-MOs minimiert. Nach der lokalen RA können CP-MOs mit identischen Quell- und Zielregistern entfernt werden, weshalb entsprechende Register-Bindungen bevorzugt werden. Insgesamt ergibt sich bei der lokalen RA der folgende Ablauf:

1. Einfügen von CP-MOs für V_{GlobIn} und $V_{GlobOut}$
2. Einfügen von CP-MOs für MOs, die Operanden in festen Registern erfordern
3. Leftedge mit folgenden Modifikationen
   - Berücksichtigung bereits gebundener Variablen
   - Minimierung von Move-Befehlen

- Einfügen von Spill-Code

4. Entfernung aller CP-MOs mit Quell-Register = Ziel-Register

5. Einfügen von Code zur Sicherung von Registern am Funktions-Anfang, am Funktions-Ende, vor/nach Funktionsaufruf und Reservierung bzw. Freigabe von Speicherplatz auf dem Stack. Dies geschieht durch prozessorspezifische Methoden um Hardware-Details zu berücksichtigen, z.B. beim ARM der Rücksprung und das Wiederherstellen mehrerer gesicherter Register mittels eines einzigen POP-Befehls.

3.8 Assemblercode-Ausgabe

Die Grundlage für die Assemblercode-Ausgabe ist die XeLIR-Datenstruktur von GeLIR [Fie01]. Dies ist die GeLIR-Programm- und Architekturdarstellung im XML-Format. Sie ermöglicht das Abspeichern und Wiederherstellen des gesamten Zustandes der GeLIR-Datenstruktur. Die XeLIR-Darstellung selbst kann z.B. für Peephole-Optimierungen und die Assemblerausgabe durch Pattern verwendet werden. Die Ausgabe des Assemblercodes erfolgt in vier Schritten:

1. Zuerst erfolgt der Aufruf der GeLIR2XeLIR-Methode der XeLIR-Bibliothek, die die gesamte GeLIR-Programm- und Architekturdarstellung in das XML-Format konvertiert und in eine Datei schreibt.

2. Die Sequenz der Maschineninstruktionen und Sprungmarkierungen wird mit dem XML-Pattern-Tool aus der XeLIR-Darstellung und einer Pattern-Definition erstellt. Für jede MI existiert ein Muster, das eine individuelle Ausgabe der Instruktion ermöglicht. In Abb. 3.10 ist das Muster für den Load-Befehl des ARM dargestellt, der der Regel (2) & (3) aus Abb. 3.3 (siehe Seite 39) entspricht.

   Die `<Print ....>`-Elemente enthalten die Ausgabevorschrift. Nach der Erzeugung der Pattern aus der XeLIR-Programmdarstellung sind diese leer und müssen durch die Assembler-Befehle der jeweiligen Instruktion ersetzt werden. Zur Ausgabe der Assembler-Befehle durchläuft das Pattern-Matching-Tool sequentiell alle MIs und Sprungmarkierungen der XeLIR-Programmdarstellung und sucht das jeweils passende Muster, dessen Ausgabevorschriften dann ausgeführt werden. In Abb. 3.11 ist dieses Vorgehen graphisch dargestellt.

3. Die Programmdarstellung enthält nur die Sequenz der Maschineninstruktionen und Sprungmarkierungen. Der Assembler benötigt weitere Angaben, wie z.B. die Variablendeklarationen und Speichersegmentangaben. Aus der GeLIR-Programmdarstellung werden zwei maschinen- und assemblerabhängige Dateien generiert, die diese zusätzlichen Informationen enthalten. Sie werden vor bzw. nach der Instruktionssequenz eingefügt. In Abb. 3.12 ist dies am Beispiel einer Assemblercode-Datei für den ARM-Prozessor dargestellt. In Datei (a) wird das Codesegment deklariert und in Datei (b) ist die Instruktions-Sequenz des Maschinenprogramms enthalten. Datei (c) enthält eine Tabelle mit Konstanten und den Adressen globaler Variablen, sowie das Datensegment mit den globalen Variablen. Die Assemblercode-Datei ergibt sich durch Aneinanderfügen der Dateien (a), (b) und (c).

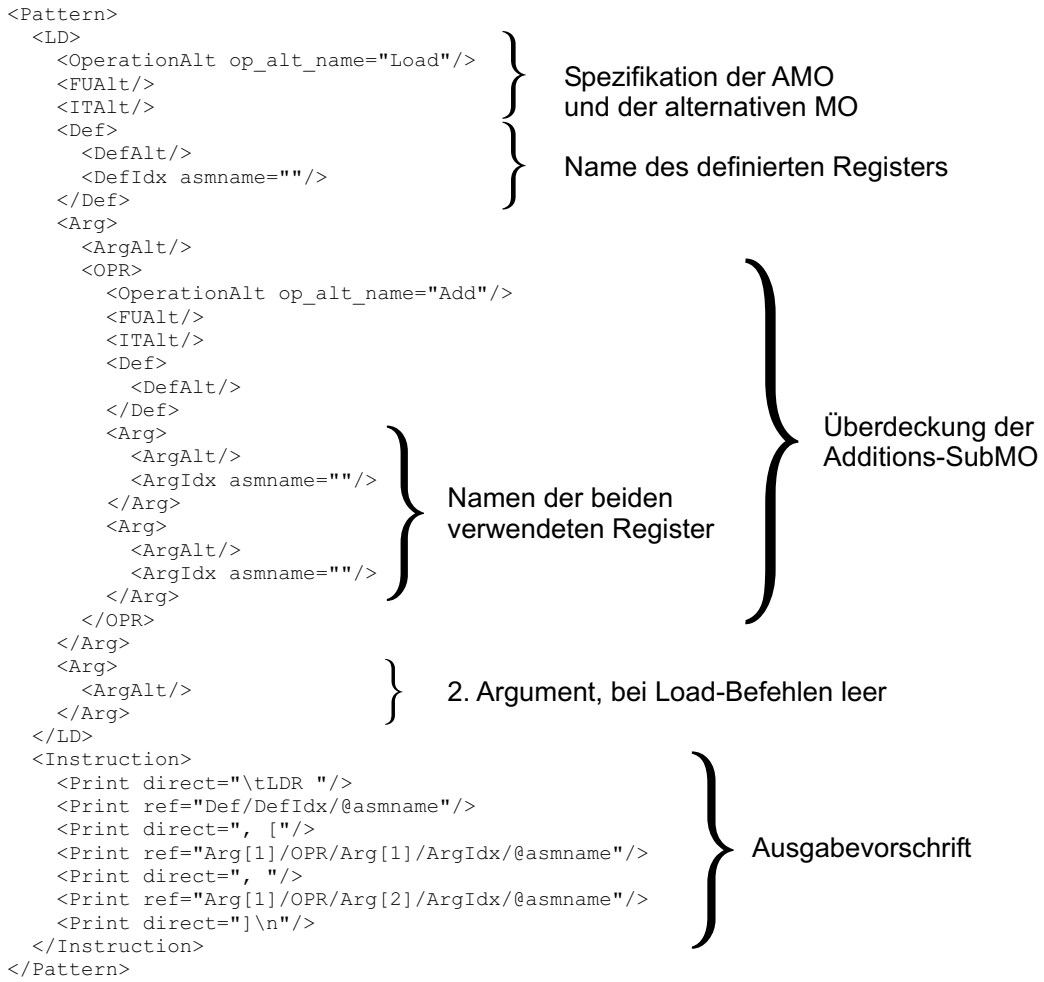

Abb. 3.10: Überblick über ein verwendetes Muster

4. Im letzten Schritt werden die Dateien in der Reihenfolge (1.), (2.) und (3.)

   im 2. Schritt und die im 3. Schritt erzeugten Dateien zu einer einzigen Assemblercode-Datei zusammenkopiert.

Die Vorteile dieses Assemblercode-Ausgabe-Verfahrens ist die einfache Implementierung und die Flexibilität durch die Ausgabevorschriften. Eine Ausgabe direkt aus GeLIR würde durch C++-Code für jede einzelne MI erfolgen, um die Register und Werte der einzelnen Operanden abzufragen. Bei den Ausgabevorschriften der Pattern erfolgt der Zugriff auf die Operanden dagegen durch einfache String-Anweisungen. Dadurch lassen sich die Pattern zur Ausgabe aller Instruktionen eines Prozessors in kurzer Zeit erstellen.

Ein Nachteil ist, dass der Assemblercode als eine lange Instruktions-Sequenz ausgegeben wird. Beim THUMB-Befehlssatz des ARM7 werden große Konstanten ($k > 255$), und damit auch die absoluten Speicheradressen von globalen Variablen, über den PC-relative (PC = Program Counter) Zugriffe geladen. Bei kleinen Programmen reicht es, die

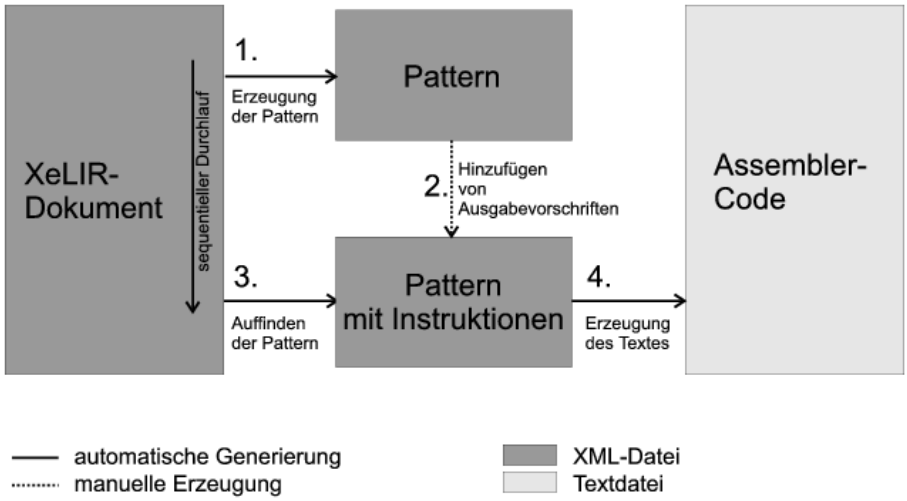

Abb. 3.11: Schreiben des Assemblercodes [Fie01]

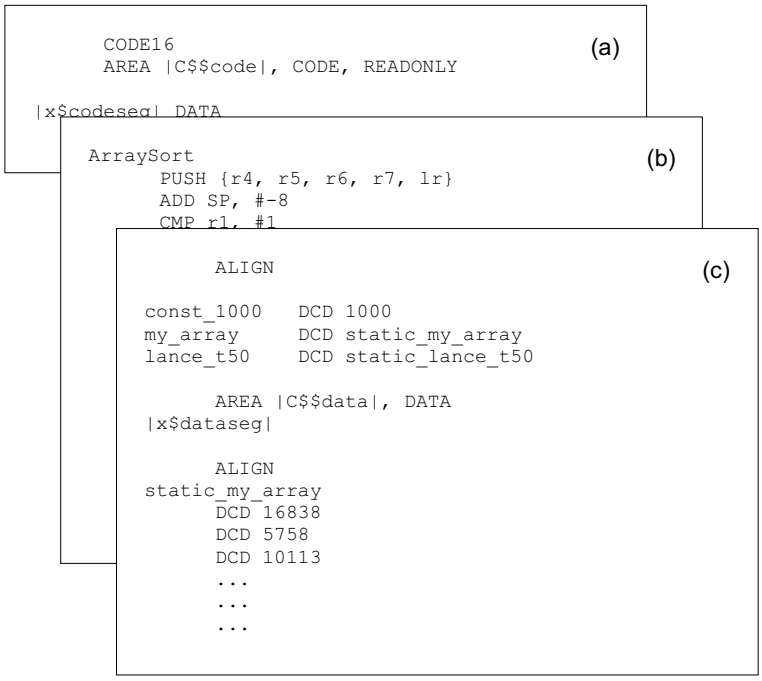

Abb. 3.12: Aufbau einer Assemblercode-Datei

Tabelle mit den großen Konstanten und den Adressen der globalen Variablen nach der Instruktions-Sequenz einzufügen. Da PC-relative Zugriffe aber auf eine Entfernung von 1 KByte begrenzt sind, müssen bei größeren Programmen evtl. Teile der Tabelle innerhalb der Instruktions-Sequenz, z.B. zwischen zwei Funktionen, eingefügt werden. Bei Verwendung dieses Assemblercode-Ausgabe-Verfahrens ist dies bislang noch nicht ohne weiteres möglich. Durch eine Erweiterung könnte z.B. die Ausgabe auf bestimmte Teil-Sequenzen beschränkt werden.

3.9 Erweiterung der Architekturdarstellung

In der GeLIR-Architekturdarstellung waren bisher nicht alle für die Entwicklung generischer Compiler-Techniken notwendigen Informationen enthalten.

Die Klasse *LirTarget* wurde erweitert, um Details über die Verwendung einzelner Register aufzunehmen. Dabei handelt es sich um die Programmier-Konventionen zur der Verwendung der Register. Für die verschiedenen Arten von Variablen werden jeweils die daran zu bindenden Registerfile-Elemente verwaltet. Die Unterscheidung der Variablen erfolgt in:

1. Variablen des globalen Datenflusses
2. Variablen des lokalen Datenflusses
3. Variablen, die zum Zwischenspeichern von Registerinhalten während der lokalen RA erzeugt werden
4. Funktions-Parameter, die von der aufrufenden Funktion übergeben werden
5. Funktions-Parameter, die von der aufgerufenen Funktion erhalten werden
6. Funktions-Wert, der von der aufgerufenen Funktion zurückgegeben wird
7. Funktions-Wert, den die aufrufende Funktion zurückbekommt

Ein Registerfile-Element darf durchaus führ mehrere Arten von Variablen verwendet werden. So können z.B. die Registerfile-Elemente, die beim ARM- und beim LEON-Prozessor die Speicherplätze auf dem Stack modellieren, an die Variablen aus (1.), (3.), (4.) und (5.) gebunden werden. Auf Grundlage dieser Mengen von Registerfile-Elementen sind die globale und lokale RA generisch durchgeführbar.

3.10 Adaption an weitere Architekturen

Die Einfache Adaption an weitere Architekturen ist der Hauptgrund für die Verwendung retargierbarer Compiler. Durch die Benutzung eines generischen Codegenerators erfolgt eine klare Trennung zwischen dem eigentlichen Compiler und der Architekturspezifikation.

Der größte Aufwand bei der Anpassung an eine weitere RISC-Architektur, wie z.B. der MIPS-Architektur, besteht in der Erstellung der GeLIR-Architekturdarstellung. Diese besteht aus der Spezifikation der Prozessor-Ressourcen, wie z.B. Registerfiles oder Funktionseinheiten, und aus der Beschreibung des Instruktionssatzes. Die einzelnen Befehle des Zielprozessors werden durch alternative MOs dargestellt.

Für die Ausgabe der Instruktionssequenz des Assemlercodes ist die Angabe von Ausgabevorschriften erforderlich, wie in Abschnitt 3.8 beschrieben. Im Vergleich zum Aufbau der GeLIR-Architekturdarstellung wird dafür jedoch nur ein sehr geringer Aufwand benötigt.

Es sind allerdings nicht alle Details der Codegenerierung generisch durchführbar. So erfolgt z.B. im Assemblercode die assemblerspezifische Deklaration von Variablen und Speichersegmenten. Auch einige spezielle Befehle, wie z.B. der Save-/Restore-Befehl des LEON,

der das Registerfenster verschiebt und den Stack-Frame auf- bzw abbaut, werden von den generischen Techniken nicht berücksichtigt, so dass dafür architekturspezifischer Code erzeugt werden muss.

Kapitel 4

Ergebnisse

In diesem Kapitel wird der Assemblercode des generischen Codegenerators (CG) mit dem von anderen Compilern erzeugten Code verglichen und noch vorhandenes Optimierungspotential aufgezeigt. Zuerst wird die Testumgebung beschrieben, in der die Ergebnisse generiert wurden. Anschließend werden die vom generischen Codegenerator erstellten Maschinenprogramme getrennt für den LEON und den ARM analysiert. Für den M5-DSP wird ein Überblick über die Ergebnisse von DSP-Benchmarks gegeben, allerdings erfolgt keine Codeanalyse.

Das Hauptziel dieser Arbeit ist die Realisierung generischer Compiler-Techniken, eine gute Codequalität ist dabei nur zweitrangig. Der Vergleich erfolgt deshalb vor allem im Hinblick auf mögliche Erweiterungen des generischen Codegenerators.

4.1 Testumgebung

Als Grundlage dieses Kapitels wurden zahlreiche Maschinenprogramme für mehrere Benchmarks generiert. Durch die Simulation dieser Maschinenprogramme für die ARM- und LEON-Prozessoren wurden verschiedene Vergleichswerte ermittelt, die Ausführungszeit, die Anzahl ausgeführter Instruktionen und der Energieverbrauch in Joule. Zunächst werden die dabei benutzten Programme vorgestellt.

4.1.1 Verwendete Compiler und Tools

Für beide RISC-Architekturen wird der generierte Code des generischen Codegenerators mit dem von je zwei weiteren Compilern verglichen. Für den ARM sind dies der ENCC und der TCC, für den LEON der ENCC und RTEMS-GCC. Zur Generierung der Benchmark-Ergebnisse für den ARM-Prozessor wurden die folgenden Tools verwendet:

- **ENCC:** Version 0.5, auch als ARM12CC bezeichnet

  Der ENCC [Enc] ist ein retargierbarer Low-Power-Forschungscompiler des LS XII, der wie der Compiler dieser Diplomarbeit das LANCE-Front-End und die Standard-Optimierungen auf der LANCE-IR benutzt (siehe auch Abschnitt 3.3). Für die

Codeselektion wird der TPM-Algorithmus (TPM = Tree-Pattern-Matching) verwendet und für die RA ein heuristisches Graph-Färbe-Verfahren [AG98]. Back-Ends gibt es für die ARM- und LEON-Architektur. Der Aufbau und die verwendeten Compiler-Techniken werden detailliert in [Ste02] beschrieben.

- **TCC:** Norcroft Thumb C Version 1.20 (ARM Ltd SDT2.50)

  Dies ist ein kommerzieller Compiler für den THUMB-Modus des ARM-Prozessors. Er ist in dem ARM-SDT (SDT = Software Development Toolkit) enthalten.

- **ARM-SD:** ARM Source-level Debugger Version 4.60 (ARM Ltd SDT2.50)

  Der Debugger des ARM-SDT erstellt bei der Simulation eines Testprogramms eine Trace-Datei, in der alle ausgeführten Instruktionen aufgeführt werden. Diese wird durch das Tool EnProfiler analysiert.

- **EnProfiler:**

  Zusammen mit dem ENCC wurde das Tool EnProfiler [Enc] entwickelt, das aus der vom ARM-SD erzeugten Trace-Datei u.a. die Anzahl der ausgeführten Instruktionen und Taktzyklen, die Anzahl der Speicherzugriffe und den Energieverbrauch ermittelt.

Für den LEON-Prozessor wurden folgende Tools verwendet:

- **ENCC:** Version 0.5, auch als LEON12CC bezeichnet

  Dies ist der ENCC-Compiler mit Back-End für den LEON-Prozessor.

- **RTEMS-GCC:** GNU C/C++ Compiler Version 2.95.2

  Dieser ist Bestandteil des von Gaisler Research [Gai] für den ERC32- und den LEON-Prozessor zur Verfügung gestellten LECCS (LEON/ERC32 Cross Compiler System), welches neben Compiler, Linker, Assembler etc. auch den Echtzeit-Kern RTEMS enthält. Es handelt sich dabei um einen an die LEON-Architektur angepassten GCC-Compiler (siehe auch Abschnitt 3.1).

- **TSIM:** TSIM/LEON SPARC Simulator, Version 1.1.3 (Evaluation Version)

  Der von Gaisler Research [Gai] zur Verfügung gestellte Simulator gibt u.a. die Anzahl der für einen Programmdurchlauf benötigten Instruktionen und Taktzyklen aus. Eine ausführliche Beschreibung befindet sich in [Sch02].

Beim Vergleich der verschiedenen Compiler und Techniken wurde der ENCC als Referenz-Compiler gewählt, weil sich dieser besonders gut zur Bewertung der generischen Codegenerierung eignet. Wie beim Compiler dieser Diplomarbeit wird das Quellprogramm zuerst von dem LANCE-Front-End analysiert, siehe auch Abb. 4.1. Anschließend werden die LANCE-Standard-Optimierungen ausgeführt, so dass die beiden Compiler das gleiche Quellprogramm in der LANCE-IR als Basis für die Codegenerierung erhalten. Allerdings wird beim ENCC keine generische Zwischendarstellung, wie z.B. GeLIR, verwendet. Deshalb ist die Adaption an weitere Prozessoren und der Austausch einzelner Compiler-Techniken mit einem erheblich größeren Aufwand verbunden. Die Unterschiede

zwischen den generierten Maschinenprogrammen resultieren nur aus der unterschiedlichen Codegenerierung, weshalb sich einzelne Compiler-Techniken gut vergleichen lassen. Zwischen dem Assemblercode beider Compiler besteht eine gewisse Ähnlichkeit. Sprungmarkierungen und temporäre Variablen tragen die gleichen, vom LANCE-Front-End generierten, Bezeichner und es sind viele gleiche Abfolgen von Instruktionen vorhanden. Da für den generischen Codegenerator keine Instruktionsanordnung (IA) implementiert wurde, wird bei dem Vergleich der Benchmark-Ergebnisse der ENCC mit aktivierter IA und mit deaktivierter IA betrachtet. Der ENCC verwendet für die Codeselektion den TPM-Algorithmus, der unter den gegebenen Bedingungen optimale Ergebnisse liefert, und für die Registerallokation ein heuristisches Graph-Färbe-Verfahren, das sehr gute, aber keine optimalen Ergebnisse liefert.

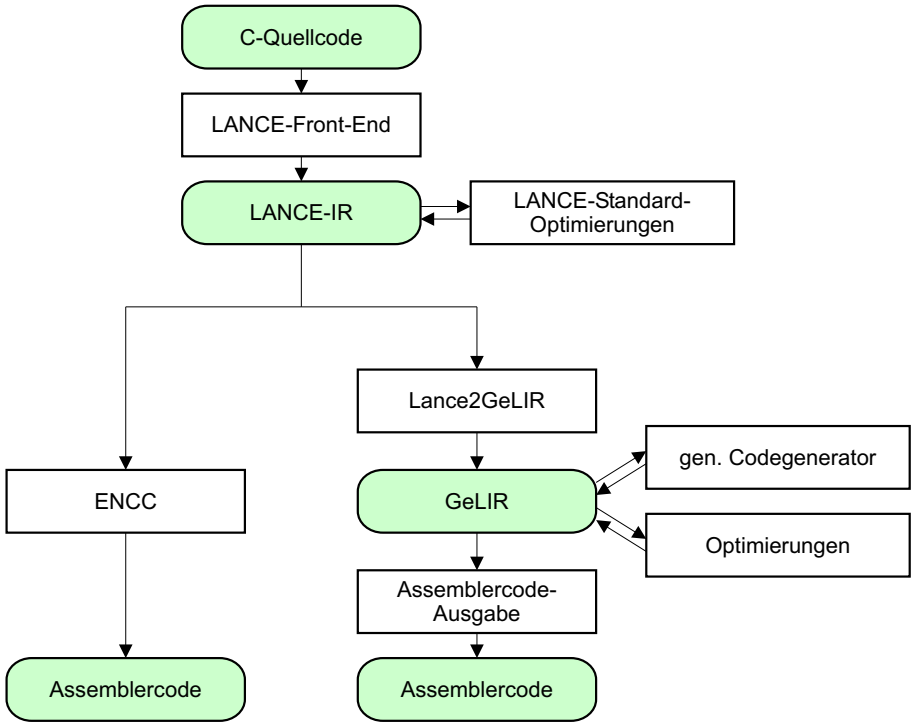

Abb. 4.1: Übersicht über den ENCC und den generischen Codegenerator

Der von den Compilern TCC und RTEMS-GCC generierte Assemblercode lässt sich dagegen nicht zum Vergleich auf Ebene einzelner Basisblöcke oder Instruktionen heranziehen, da z.B. für For-Schleifen oder die Auswertung von booleschen Ausdrücken völlig anderer Code erstellt wird. Dies liegt daran, dass die beiden Compiler ein eigenes Front-End und andere High-Level-Optimierungen verwenden, was einen großen Einfluß auf die Codequalität hat. Die Qualität der Codegeneratoren kann deshalb nicht gut verglichen werden.

4.1.2 Benchmarks

Zum Vergleich der von den verschiedenen Compilern generierten Maschinenprogramme wird die Simulation von sechs Sortieralgorithmen betrachtet. Diese wurden u.a. bereits in [Ste02] und [Hor01a] als Benchmarks verwendet. Im einzelnen handelt es sich um folgende Sortieralgorithmen:

- Bubble-Sort
- Heap-Sort
- Insertion-Sort
- Multi-Sort
- Quick-Sort
- Selection-Sort

4.1.3 Vergleichskriterien

Bei der Simulation der generierten Maschinenprogramme sind verschiedene Größen messbar. Die wichtigsten davon sind:

1. **Ausführungszeit:** Diese wird anhand der ausgeführten Taktzyklen gemessen, die mit dem EnProfiler für den ARM und mit TSIM für den LEON ermittelt wird.

2. **Anzahl ausgeführter Instruktionen:** Diese werden vom EnProfiler bzw. TSIM ermittelt. Da die einzelnen Instruktionen unterschiedlich viele Taktzyklen zur Ausführung benötigen, ist diese Größe allein betrachtet nicht so aussagekräftig wie z.B. die Ausführungszeit oder der Energieverbrauch. Zusammen mit der Anzahl der ausgeführten Taktzyklen ist der CPI-Wert (CPI = Cycles Per Instruction) bestimmbar.

3. **Codegröße in Befehlswörtern:** Diese werden aus dem Assemblercode ermittelt. Globale Variablen im Datensegment, wie z.B. die Inhalte der zu sortierenden Arrays oder die Strings von Fehlerausgaben, werden dabei nicht berücksichtigt.

4. **Energieverbrauch in Joule:** Dieser wird nur für den ARM-Prozessor vom EnProfiler ermittelt. Für den LEON-Prozessor sind weitere Tools erforderlich, z.B. ModelSim [MSi] oder LISA Processor Design Plattform [Lis], die beide detailliert in [Sch02] beschrieben werden.

Die Ausführungszeit ist das Hauptvergleichskriterium, weil dies die einzige berücksichtigte Messgröße der implementierten Techniken ist. Da das Gebiet der Low-Power-Compiler ein Forschungsschwerpunkt des LS XII ist, werden auch der Energieverbrauch und die Codegröße kurz betrachtet. Allerdings erfolgen keine speziellen Optimierungen dafür, wie z.B. Minimierung der Switching Power oder Verwendung eines Onchip-Speichers, die aber noch integriert werden könnten. Die Grundlagen dieser beiden Energieeinsparungs-Techniken werden in [Ste02] beschrieben.

4.2 Ergebnisse LEON

Für den Vergleich der Benchmark-Ergebnisse des LEON-Prozessors werden die Ausführungszeit, die Anzahl ausgeführter Instruktionen und die Codegröße betrachtet. Die Darstellung der Ergebnisse erfolgt in Relation zu denen des ENCC mit deaktivierter IA. In Anhang A.1 sind die absoluten Messwerte jeweils tabellarisch dargestellt.

Die Ermittlung des Energieverbrauchs ist mit TSIM nicht möglich. Auf die Verwendung eines weiteren Simulators wurde verzichtet, da aufgrund der geringen Unterschiede bei den bereits durchgeführten Vergleichen keine gravierenden Unterschiede zwischen ENCC und dem generischen Codegenerator zu erwarten sind.

4.2.1 Ausführungszeit

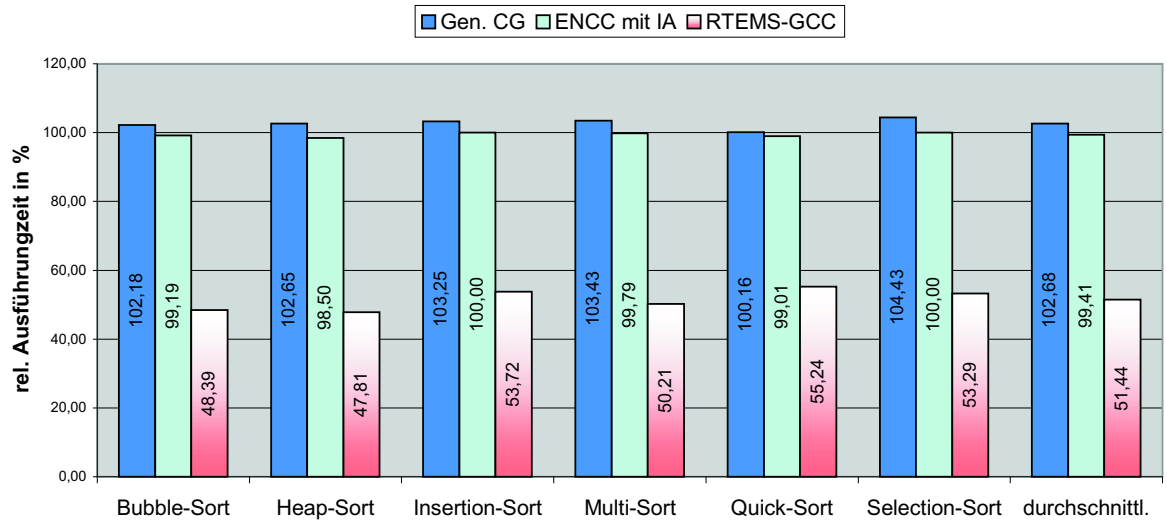

Abb. 4.2: Ausführungszeiten beim LEON (ENCC ohne IA $\widehat{=}$ 100%)

Wie in Abb. 4.2 zu sehen ist, erzeugt der RTEMS-GCC bei allen Benchmarks den mit Abstand schnellsten Code. Die durchschnittlichen Ausführungszeiten der übersetzten Benchmarks betragen im Vergleich zum ENCC (ohne IA):

Gen. CG : 102,68%
ENCC mit IA: 99,41%
RTEMS-GCC: 51,44%

Ein Vergleich der Compiler-Techniken zwischen dem RTEMS-GCC und dem generischen Codegenerator ist aufgrund der unterschiedlichen Struktur der erzeugten Maschinenprogramme nicht möglich. Allerdings sind in dem vom RTEMS-GCC erstellten Code vergleichsweise wenige NOP-Befehle (NOP = No Operation) vorhanden. Der generische Codegenerator und der ENCC fügen hinter jeder Sprunganweisung einen NOP-Befehl ein, da diese Anweisung aufgrund des Branch Delay Slots (siehe Abschnitt 2.1.2) auch dann ausgeführt wird, wenn ein Sprung stattfindet. Wie in Tabelle 4.1 aufgeführt, sind in den vom generischen Codegenerator übersetzten Benchmarks durchschnittlich 17,5% der

Codegröße und 20,8% der Anzahl ausgeführter Instruktionen auf die NOP-Befehle zurückzuführen. So wird 18,14% der Ausführungszeit für die ungenutzten Branch Delay Slots verbraucht, was ein großes Optimierungspotential darstellt. Der vom ENCC generierte Code enthält in gleicher Größenordnung NOP-Befehle.

	Anz. NOPs im Code	% von Codegröße	Anz. NOPs ausgeführt	% von ausgef. Instruktionen	% von Taktzyklen
Bubble-Sort	19	16,96	27.757	18,91	16,57
Heap-Sort	54	17,70	100.200	16,66	14,52
Insertion-Sort	26	17,22	15.897	21,76	18,79
Multi-Sort	89	17,45	60.372	20,64	18,24
Quick-Sort	47	19,83	6.370	22,37	18,70
Selection-Sort	16	15,84	25.738	24,45	22,04
durchschnittl.		17,50		20,80	18,14

Tabelle 4.1: Anteil von NOPs im LEON-Code des generischen CG

	Anz. NOPs im Code	% von Codegröße	Anz. NOPs ausgeführt	% von ausgef. Instruktionen	% von Taktzyklen
Bubble-Sort	3	3,45	5.006	8,53	6,31
Heap-Sort	1	0,58	1.001	0,39	0,31
Insertion-Sort	2	1,71	101	0,33	0,23
Multi-Sort	7	2,12	5.204	4,38	3,24
Quick-Sort	1	0,72	201	1,41	1,07
Selection-Sort	3	3,41	101	0,23	0,17
durchschnittl.		2,00		2,55	1,89

Tabelle 4.2: Anteil von NOPs im Code des RTEMS-GCC

Wie in Tabelle 4.2 zu sehen ist, generiert der RTEMS-GCC Code mit einer viel geringeren Anzahl von NOP-Befehlen. Für sie wird lediglich 1,89% der Ausführungszeit verbraucht. Die Ursache der vergleichsweise kurzen Ausführungszeiten liegt also teilweise in der besseren Ausnutzung der Branch Delay Slots. Diese könnten beim generischen Codegenerator durch eine Optimierung ebenfalls besser genutzt werden.

Die Durchführung der IA beim ENCC steigert die Ausführungszeit der generierten Benchmarks um 0,59%. Bei den betrachteten Sortieralgorithmen ist aufgrund der großen Anzahl universeller Register des LEON das Zwischenspeichern von Registerinhalten nicht erforderlich, unabhängig davon, ob die IA durchgeführt wird, so dass nur geringfügige Änderungen auftreten. Beim Heap- und Multi-Sort-Benchmark enthalten die bei deaktivierter IA compilierten Programme einige Move-Befehle mehr, weshalb mit der Durchführung der IA z.B. beim Heap-Sort-Benchmark die Anzahl der erforderlichen Taktzyklen jeweils um ca. 1,5% verringert werden kann. Für die betrachteten Benchmarks hat die IA bei der Codegenerierung für den LEON-Prozessor nur eine geringe Auswirkung.

Beim Quick-Sort-Benchmark bewirkt die Durchführung der IA eine Verringerung der Ausführungszeit um 1% bei gleicher Anzahl ausgeführter Instruktionen. Dies geschieht

durch die Änderung der Reihenfolge einer Sequenz aus Load- und Store-Befehlen, wodurch zwei aufeinander folgende Speicherzugriffe auf die gleiche Adresse stattfinden. Der zweite Speicherzugriff erfordert aufgrund des Daten-Caches des LEON weniger Taktzyklen. In Abb. 4.3 wird dies an der entsprechenden Stelle im Code dargestellt.

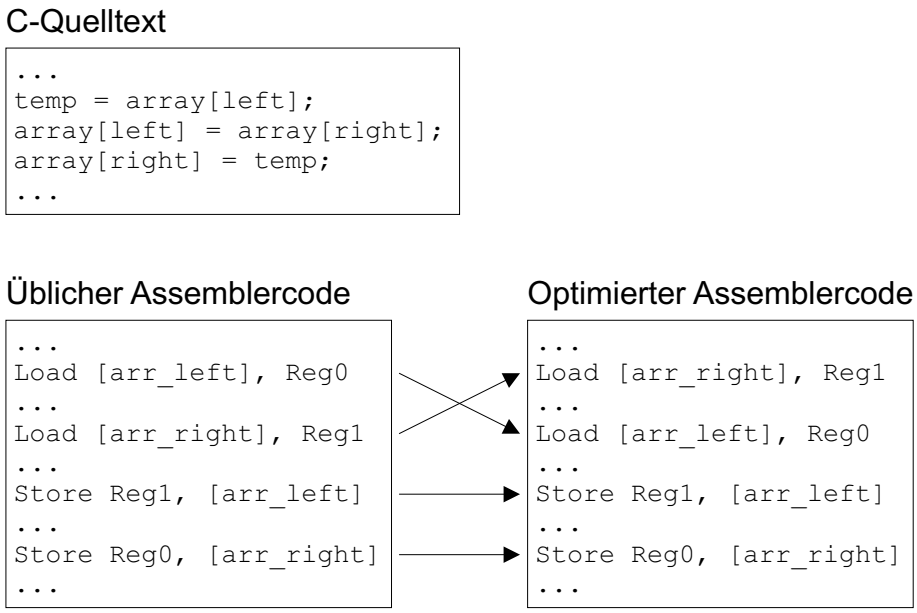

Abb. 4.3: Optimierung von Speicherzugriffen zur besseren Cache-Nutzung

Die Codeselektion vom generischen Codegenerator und vom ENCC liefern identische Ergebnisse, d.h. es werden die gleichen Befehle des LEON-Prozessors ausgewählt. Unterschiede im erzeugten Code bestehen nur in der unterschiedlichen Registerbindung und in einzelnen Move-Befehlen. Dabei handelt es sich um die vor der lokalen Registerallokation eingefügten Datentransfers (siehe Abschnitt 3.7.2). Nach der Definition einer Variablen des globalen Datenflusses wird eine Copy-MO eingefügt, um evtl. erforderliche Store- oder Move-Befehle zu modellieren. Da der modifizierte Leftedge-Algorithmus jeweils nur die aktuelle Instruktion und die Menge der belegten Register betrachtet, wird die Registerbindung nicht vorausschauend durchgeführt. Deshalb kommt es vor, dass am Ende eines Basisblocks ein oder mehrere Registerinhalte umgeladen werden müssen. Durch eine Erweiterung der Leftedge-Algorithmus könnte dieser Effekt zumindest teilweise beseitigt werden.

4.2.2 Anzahl ausgeführter Instruktionen

In Abb. 4.4 ist die Anzahl der ausgeführten Instruktionen dargestellt. Im Vergleich zum ENCC (ohne IA) beträgt sie durchschnittlich:

Gen. CG : 103,07%
ENCC mit IA: 99,59%
RTEMS-GCC: 43,83%

Die Größenordnungen sind ähnlich wie bei der Ausführungszeit. Für die vom RTEMS-GCC generierten Maschinenprogramme beträgt der CPI-Wert durchschnittlich 1,35. Bei

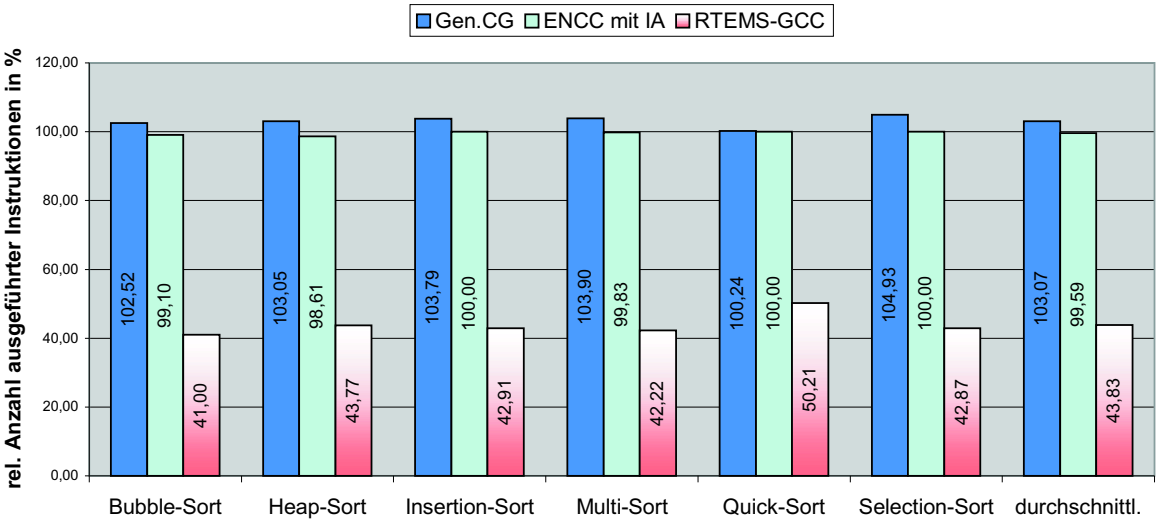

Abb. 4.4: Anzahl ausgeführter Instruktionen beim LEON (ENCC ohne IA $\widehat{=}$ 100%)

dem vom ENCC (mit IA) und vom generischen Codegenerator erzeugten Code beträgt der CPI-Wert 1,15, was auf die größere Anzahl ausgeführter NOP-Befehle zurückzuführen ist. Diese erfordern jeweils nur einen Taktzyklus und können so in großer Anzahl den CPI-Wert auf nahe eins verringern.

4.2.3 Codegröße

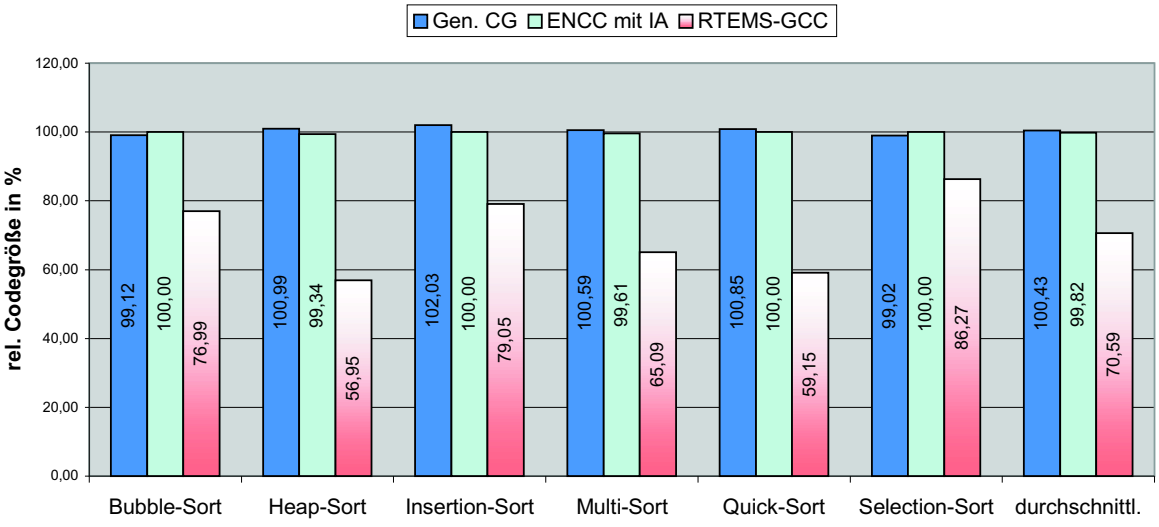

Abb. 4.5: Codegröße beim LEON (ENCC ohne IA $\widehat{=}$ 100%)

In Abb. 4.5 ist die Codegröße der generierten Maschinenprogramme zu sehen. Im Vergleich zum ENCC (ohne IA) beträgt sie durchschnittlich jeweils:

Gen. CG : 104,03%
ENCC mit IA: 99,82%
RTEMS-GCC: 70,59%

Der RTEMS-GCC erzeugt bzgl. der Codegröße durchschnittlich wieder mit Abstand den besten Code. Allerdings beruht auch hier ein großer Teil des Unterschieds auf der besseren Nutzung der Branch Delay Delay Slots. Würden bei der Bestimmung der Codegröße die NOP-Befehle nicht berücksichtigt, ergäbe sich z.B. beim Selection-Sort-Benchmark für alle drei Compiler fast der gleiche Wert.

4.2.4 Zusammenfassung

Der RTEMS-GCC liefert für alle Benchmarks die besten Ergebnisse, lässt sich aber nur eingeschränkt zu einer Bewertung des generischen Codegenerators heranziehen, da die generierten Maschinenprogramme große Unterschiede aufweisen. Der Vergleich mit dem ENCC (ohne IA) ergab, dass die Codeselektion des generischen Codegenerators optimale Ergebnisse liefert. Im Durchschnitt ist die Ausführungszeit der generierten Maschinenprogramme um nur 2,68% länger als beim ENCC. Die Gründe dafür liegen in der lokalen Registerallokation, die nicht vorausschauend durchgeführt wird. Durch eine Erweiterung könnte dieser Effekt allerdings größtenteils beseitigt werden.

Eine deutliche Steigerung der Codequalität wäre durch die bessere Nutzung der Branch Delay Slots (BDS) möglich. Eine einfache Peephole-Optimierung wäre, bei unbedingten Sprüngen die vorherige Instruktion in den BDS zu verschieben. Da bei bedingten Sprüngen eine Datenflussabhängigkeit zu dem Compare-Befehl besteht, müssen in diesem Fall jedoch bei der Verschiebung einer Instruktion in den BDS alle Datenabhängigkeiten eingehalten werden, so dass dies nicht immer möglich sein dürfte.

4.3 Ergebnisse ARM

Für den Vergleich der Benchmark-Ergebnisse des ARM-Prozessors werden die Ausführungszeit, die Anzahl ausgeführter Instruktionen, die verbrauchte Energie und die Codegröße betrachtet. Für den generischen Codegenerator erfolgt bei der Darstellung der Messwerte eine Unterscheidung, ob die High-Register wie beim ENCC zum Zwischenspeichern von Registerinhalten verwendet werden (*Gen. CG, Hi-Regs*), oder ob nur der Stack dafür benutzt wird (*Gen. CG, Stack*).

Die Darstellung der Ergebnisse erfolgt ebenfalls in Relation zu denen des ENCC mit deaktivierter IA. In Anhang A.2 sind die absoluten Messwerte jeweils tabellarisch dargestellt.

4.3.1 Ausführungszeit

In Abb. 4.6 sind die Ausführungszeiten der übersetzten Benchmarks veranschaulicht. Die durchschnittlichen Ausführungszeiten betragen im Vergleich zum ENCC (ohne IA):

Gen. CG, Stack : 124,00%
Gen. CG, Hi-Regs: 108,45%
ENCC mit IA : 99,20%
TCC : 67,93%

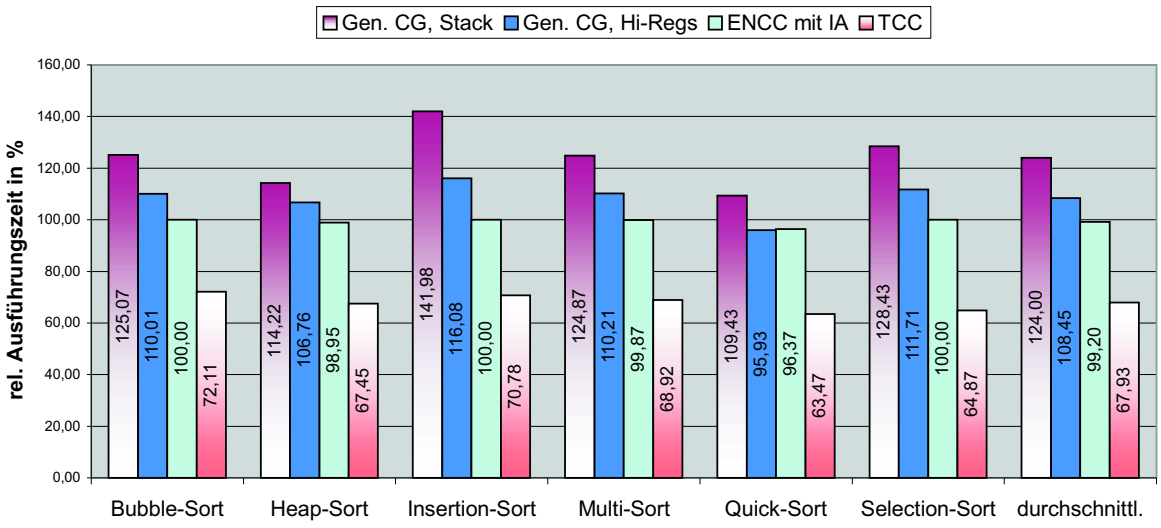

Abb. 4.6: Ausführungszeiten beim ARM (ENCC ohne IA $\widehat{=}$ 100%)

Der TCC erzeugt den schnellsten Code. Allerdings eignet er sich nur bedingt zur Bewertung des generischen Codegenerators, da ein Vergleich der generierten Maschinenprogramme aufgrund der recht unterschiedlichen Codestruktur nur eingeschränkt durchführbar ist.

Der generische Codegenerator fügt in alle Maschinenprogramme für den ARM-Prozessor Spillcode ein, der ENCC nur beim Heap-, Multi-, und Quick-Sort-Benchmark.

Bei der Übersetzung des Bubble-, Insertion- und Selection-Sort-Benchmarks hat die IA keine Auswirkungen. Beim Heap- und Multi-Sort-Benchmark werden dadurch einige Move-Befehle entfernt, was z.B. beim Heap-Sort zu einer Reduzierung der Ausführungszeit um 1,05% führt. Beim Quick-Sort-Benchmark wird durch die IA die Anzahl der Zwischenspeicherungen verringert, was eine Verbesserung der Ausführungszeit um 3,63% bewirkt.

Wie schon beim Vergleich für den LEON-Prozessor betrachtet, sind auch in dem vom generischen Codegenerator für den ARM erzeugten Assemblercode Move-Befehle enthalten, die während der lokalen Registerallokation (RA) eingefügt werden. Beim Code für den ARM kommen noch Zwischenspeicherungen hinzu.

Aufgrund der getrennten lokalen und globalen RA findet keine global optimale Registervergabe statt. Da die globale RA vor der lokalen durchgeführt wird, haben die Variablen des globalen Datenflusses eine höhere Priorität. So kommt es z.B. vor, dass eine selten verwendete Variable an ein Register gebunden und in mehreren Basisblöcken von der lokalen RA zwischengespeichert wird. Werden bei der globalen RA zuviele Variablen an Register gebunden, so gibt es bei der lokalen RA häufiger Zwischenspeicherungen. Werden zu wenige Variablen an Register gebunden, sind nach der lokalen RA noch unbelegte Register vorhanden. Verfahren, die die lokale und globale RA in einem Schritt durchführen, betrachten die Variablen des lokalen und globalen Datenflusses synchron. Daher tritt dieser Effekt dort nicht auf. Durch eine nachträgliche Kopplung der beiden Verfahren könnte die Codequalität allerdings noch gesteigert werden.

Auf die beim Zwischenspeichern verwendeten High-Register wird durch Move-Befehle zugegriffen. Diese kosten jeweils einen Taktzyklus und sind im Vergleich zu Speicherzugriffen

recht preiswert (Load: 3, Store: 2 Taktzyklen) [Adv95a]. Werden die High-Register von der Zwischenspeicherung ausgeschlossen, steigt die durchschnittliche Ausführungszeit der erzeugten Maschinenprogramme von 108,24% auf 124%, relativ zum ENCC (ohne IA). Im Vergleich zur Verwendung der High-Registern kostet jeder Lesezugriff auf eine im Speicher zwischengespeicherte Variable die dreifache Anzahl an Taktzyklen und jeder Schreibzugriff die doppelte Anzahl.

4.3.2 Anzahl ausgeführter Instruktionen

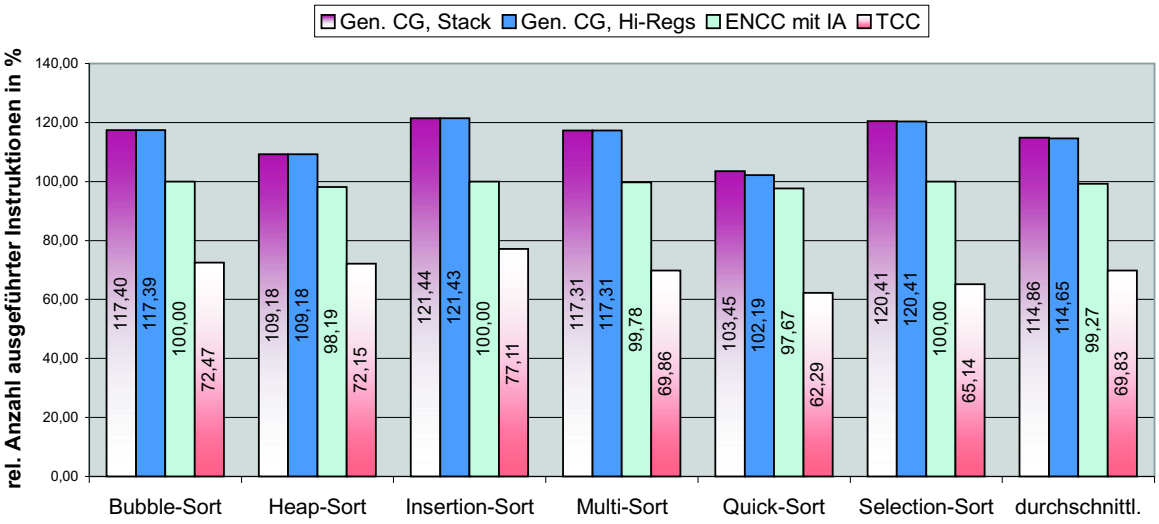

Abb. 4.7: Anzahl ausgeführter Instruktionen beim ARM (ENCC ohne IA $\widehat{=}$ 100%)

Die Anzahl der ausgeführten Instruktionen ist in Abb. 4.7 dargestellt. Die durchschnittliche Anzahl der Instruktionen beträgt im Vergleich zum ENCC (ohne IA):

Gen. CG, Stack	:	114,86%
Gen. CG, Hi-Regs	:	114,65%
ENCC mit IA	:	99,27%
TCC	:	69,83%

Wird beim generischen Codegenerator ausschließlich der Stack zum Zwischenspeichern benutzt, steigt bei den generierten Benchmarks die durchschnittliche Anzahl der ausgeführten Instruktionen leicht. Das liegt an dem geringeren Overhead der Funktionen, da die High-Register bei den betrachteten Benchmarks nur selten gesichert werden müssen, wohingegen bei Benutzung des Stacks der Auf- und Abbau des Stack-Frames unverzichtbar ist.

4.3.3 Codegröße

In Abb. 4.8 ist die Codegröße dargestellt. Im Vergleich zum ENCC (ohne IA) beträgt sie durchschnittlich:

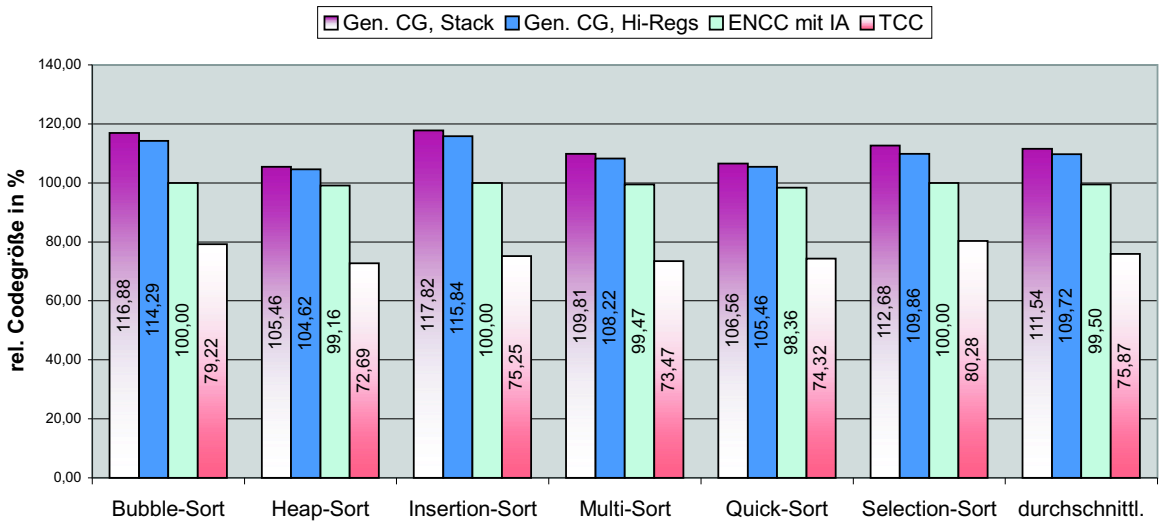

Abb. 4.8: Codegröße beim ARM (ENCC ohne IA 100%)

Gen. CG, Stack : 111,54%
Gen. CG, Hi-Regs: 109,72%
ENCC mit IA : 99,50%
TCC : 75,87%

Hier ergibt sich ein ähnliches Bild, wie bei den vorherigen Benchmarks. Die beiden leicht von einander abweichenden Werte für den generischen Codegenerator sind die Folge des unterschiedlich großen Funktions-Overheads, wie in Abschnitt 4.3.2 beschrieben.

4.3.4 Energieverbrauch

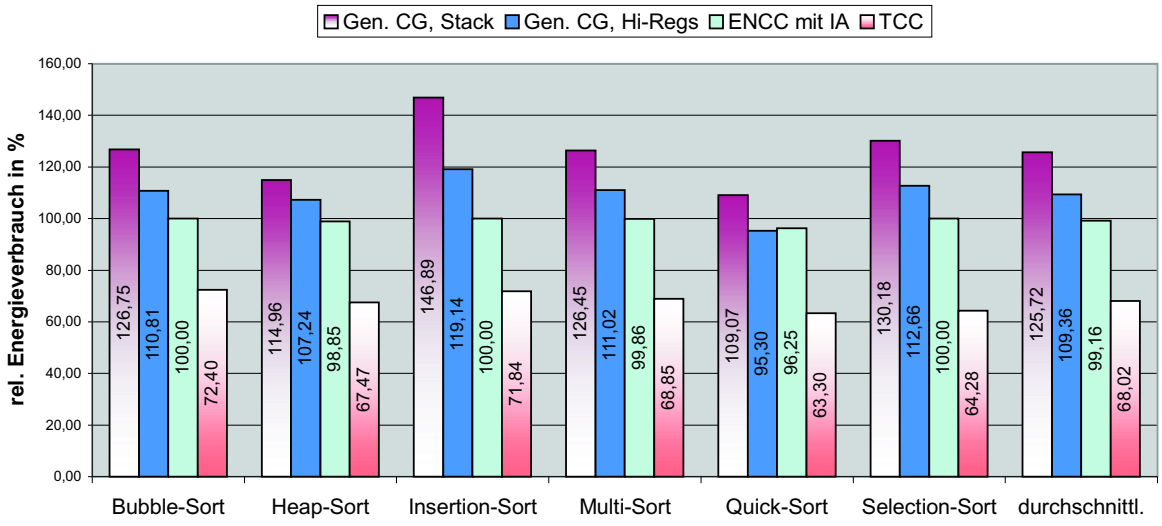

Abb. 4.9: Vergleich des Energieverbrauchs beim ARM (ENCC ohne IA $\widehat{=}$ 100%)

In Abb. 4.9 ist der Energieverbrauch dargestellt. Im Vergleich zum ENCC (ohne IA) beträgt dieser durchschnittlich:

Gen. CG, Stack	: 125,72%
Gen. CG, Hi-Regs	: 109,36%
ENCC mit IA	: 99,16%
TCC	: 68,02%

Der direkte Zusammenhang mit der Ausführungszeit (siehe Abb. 4.6) ist offensichtlich.

Bei der Codeselektion für den ARM-Prozessor gibt es kaum Unterschiede. Es tritt nur ein Fall auf, bei dem bestimmte Konstanten energiesparender geladen werden:

```
generischer Codegenerator:
  ldr r0, [const_1000]  Taktzyklen: 3

ENCC:
  mov r0, #250          Taktzyklen: 1
  lsl r0, #2            Taktzyklen: 2
```

Bei beiden Möglichkeiten wird die gleiche Anzahl von Taktzyklen benötigt und, unter Berücksichtigung des Ladevorgangs der Instruktionen, auch die gleiche Anzahl von Speicherzugriffen durchgeführt. Die zweite Variante verbraucht aber 1% weniger Energie, als die erste [Ste02]. Dieser Spezialfall könnte beim generischen Codegenerator z.B. durch die Implementierung einer einfachen Peephole-Optimierung berücksichtigt werden.

4.3.5 Zusammenfassung

Die Registerallokation des generischen Codegenerators basiert auf Standard-Graph-Algorithmen, wobei die lokale und globale RA ohne Kopplung durchgeführt werden. Im Vergleich zum ENCC mit deaktivierter IA benötigen die vom generischen Codegenerator erzeugten Maschinenprogramme durchschnittlich eine nur 8,45% längere Ausführungszeit, was in Anbetracht der beiden einfachen RA-Verfahren und deren getrennten Durchführung ein sehr positives Ergebnis ist. Durch die Erweiterung der lokalen RA oder durch die Kopplung der beiden Verfahren kann die Codequalität noch gesteigert werden.

Bei den betrachteten Benchmarks führt die Instruktionsanordnung des ENCC zu einer Verringerung der Ausführungszeit um 0,8%, hat also nur sehr geringe Auswirkungen.

4.4 Verwendung der globalen RA für den M5-DSP

Die globale Registerallokation steht als eigenständige GeLIR-Klasse allgemein zur Verfügung und kann ohne Adaption mit anderen Architekturklassen verwendet werden. Sie wird auch von einem Compiler für den M5-DSP benutzt. Bei dieser Architektur handelt es sich um eine aktuell in Entwicklung befindliche Nachfolgearchitektur des M3-DSP [FWD+98, WFL+99]. Neben den bei DSPs üblichen Spezialregistern hat der M5-DSP aber auch eine

Reihe von homogenen Registerfiles, in denen Werte zwischengespeichert werden können. Vor der Benutzung der globalen RA aus dieser Arbeit führte der M5-DSP-Compiler nur eine lokale RA durch und die Variablen des globalen Datenflusses wurden im Speicher abgelegt. Durch die globale RA können diese auch an die homogenen Register gebunden werden, wodurch die Codequalität deutlich gesteigert wird.

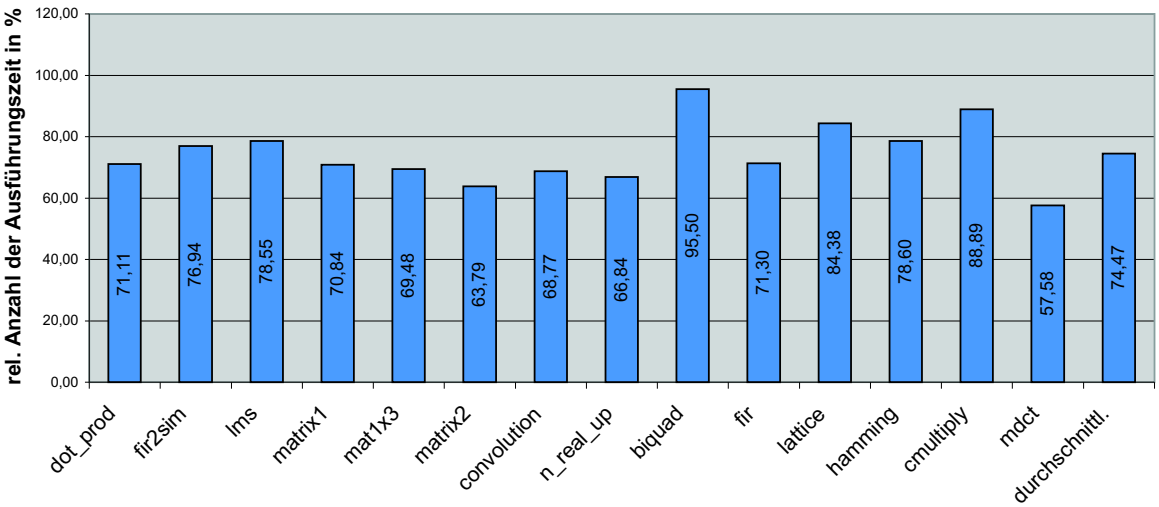

Abb. 4.10: Ausführungszeiten beim M5-DSP (ohne globale RA $\widehat{=}$ 100%)

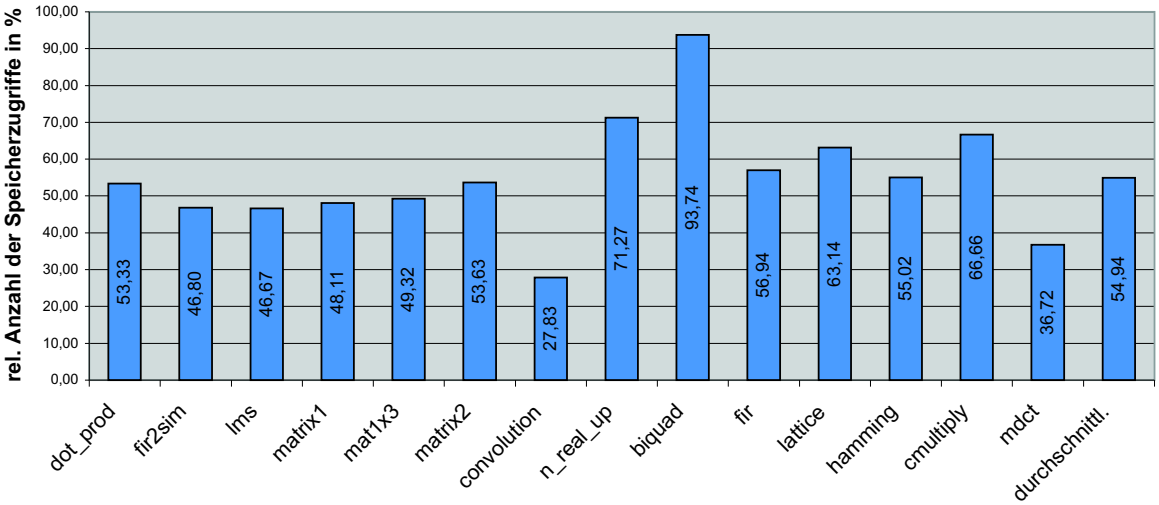

Abb. 4.11: Anzahl Speicherzugriffe beim M5-DSP (ohne globale RA $\widehat{=}$ 100%)

Für die Bewertung der Auswirkungen der globalen RA werden mehrere DSP-Benchmarks betrachtet, die u.a. auch in [Lor03] benutzt wurden. Im Durchschnitt wird dabei die Ausführungszeit auf 74,47% (siehe Abb. 4.10) und die Anzahl der Speicherzugriffe auf 54,94% gesenkt (siehe Abb. 4.11). In Anhang A.3 sind die Ergebnisse detailliert aufgeführt.

4.5 Bewertung

Bei der Realisierung des generischen Codegenerators wurden einfache Techniken ausgewählt und diese im Hinblick auf eine Wiederverwendung in anderen Compilern implementiert. Obwohl die erzeugte Codequalität nur zweitrangig war, werden bei den Benchmark-Vergleichen gute Ergebnisse erzielt. Die Codeselektion liefert für die beiden RISC-Prozessoren ARM und LEON jeweils optimale Resultate. Die Registerallokation kann dagegen noch etwas verbessert werden, z.B. durch Erweiterung der lokalen RA oder durch Kopplung der lokalen und globalen RA. Die Vorteile der entwickelten Registerallokation werden bei der Wiederverwendung in einem Compiler für den M5-DSP deutlich. Die globale RA ermöglicht die Nutzung der homogenen Registerfiles der M5-DSP-Architektur, wodurch die Ausführungszeit der Maschinenprogramme um durchschnittlich 25,53% verringert wird.

Bei den Vergleichen wurde auch das Potential einzelner Optimierungen deutlich. Bei den betrachteten Benchmarks sind die Auswirkungen der Instruktionsanordnung beim ENCC gering. Dies liegt möglicherweise an der geringen Größe der Benchmarks. Eine deutliche Steigerung der Codequalität hätte dagegen die bessere Nutzung der Branch Delay Slots beim LEON-Prozessor zur Folge, da bei den vom generischen Codegenerator erzeugten Maschinenprogrammen durchschnittlich 18,14% der Ausführungszeit auf NOP-Befehle entfällt. Beim RTEMS-GCC sind es dagegen nur 1,89%.

Der RTEMS-GCC für den LEON und der kommerzielle TCC für den ARM-Prozessor liefern in allen Messreihen mit Abstand die besten Ergebnisse. Durch die Unterschiede in der Struktur der erzeugten Maschinenprogramme können die Codegeneratoren nicht so gut mit dem in dieser Arbeit entwickelten generischen Codegenerator verglichen werden.

Kapitel 5

Zusammenfassung & Ausblick

In diesem Kapitel werden die durchgeführten Arbeiten zusammengefasst und es wird ein Ausblick über mögliche Erweiterungen des generischen Codegenerators gegeben.

5.1 Zusammenfassung

Das Hauptziel dieser Arbeit war die Entwicklung eines generischen Codegenerators für RISC-Architekturen (RISC = Reduced Instruction Set Computer). RISC-Prozessoren haben mehrere gemeinsame Eigenschaften. Die zwei wichtigsten sind die Load-/Store-Architektur, bei der Speicherzugriffe nur über die dafür vorgesehenen Load- und Store-Befehle erfolgen, und der homogene Registersatz, bei dem die Register des Prozessors in ihrer Verwendbarkeit gleichwertig sind. Diese beiden Eigenschaften ermöglichen die Entwicklung von einfachen, generischen Compiler-Techniken. Diese sind maschinenunspezifisch realisiert und beziehen die erforderlichen Prozessor-Details aus einer von der Codegenerierung getrennten Architekturdarstellung. Als Basis für einen generischen Codegenerator wird ein Austauschformat benötigt, das einerseits die getrennte Darstellung mehrerer Architekturen ermöglicht, andererseits eine Programmdarstellung auf High-Level-Ebene (IR = Intermediate Representation) und zugleich generisch auf Low-Level-Ebene (LIR = Low Level Intermediate Represantation) bietet. Die Compilerzwischendarstellung GeLIR (Generic Low-Level Intermediate Representation) erfüllt diese Anforderungen und bildet die Basis des in dieser Diplomarbeit entwickelten generischen Codegenerators. Sie ermöglicht sogar die Wiederverwendung der implementierten Compiler-Techniken für andere Architekturklassen, so dass z.B. die globale Registerallokation dieser Arbeit von einem M5-DSP-Compiler benutzt wird.

Für einen funktionsfähigen Codegenerator mussten mindestens zwei Phasen realisiert werden, die Codeselektion (CS) und die Registerallokation (RA). Um den Aufwand in vertretbarem Rahmen zu halten wurden jeweils einfache Techniken implementiert. Für die CS wurde der TPM-Algorithmus (TPM = Tree Pattern Matching) so modifiziert, dass er auf Graphen angewendet werden kann und bestehende GeLIR-Funktionalitäten benutzt. Abgesehen von nachträglich optimierbaren Spezialfällen wie z.B. das Laden bestimmter Konstanten, liefert die CS optimale Ergebnisse. Die RA basiert auf einfachen Graph-Algorithmen und wurde als zweistufiges Verfahren realisiert, bei dem die globale (Basisblock-übergreifende) und lokale (Basisblock-interne) RA ohne Kopplung durch-

geführt werden. Der größte Vorteil dieser getrennten Durchführung ist, dass die globale RA auch für andere Architekturklassen benutzbar ist, wie beispielhaft für den M5-DSP gezeigt wurde. Weitere Vorteile sind, im Vergleich zu Graph-Färbe-Verfahren, die einfache Implementierung und die kurze Laufzeit.

Der generische Codegenerator erzeugt Maschinenprogramme für den ARM7-Prozessor im THUMB-Modus und für den LEON-Prozessor. Für den ARM-Prozessor wird bei den übersetzten Benchmarks durchschnittlich eine nur 8,45% längere Ausführungszeit benötigt, verglichen mit dem ENCC bei deaktivierter Instruktionsanordnung. Beim LEON-Prozessor wird sogar eine nur um 2,68% längere Ausführungszeit erreicht, was auf die große Anzahl universeller Register zurückführbar ist. Auf dem M5-DSP wird durch die Benutzung der globalen RA eine beträchtliche Verbesserung der Codequalität erzielt, die Ausführungszeit der betrachteten Benchmarks wurde um durchschnittlich 25,53% gesenkt.

Bei der Realisierung des generischen Codegenerators ging es hauptsächlich um die Demonstration des Konzepts eines retargierbaren Compilers auf Basis von GeLIR. Die Vorteile eines solchen Compilers liegen in der einfachen Erweiterbarkeit. Vor allem die Adaption an weitere RISC-Architekturen ist mit einem geringen Aufwand verbunden. So müssen z.B. für die Erweiterung des generischen Codegenerators für die MIPS-Architektur lediglich die Architekturspezifikation und die Ausgabevorschriften für die Assemblercode-Ausgabe erstellt werden.

Aufgrund der generischen Programmdarstellung können Compiler-Techniken anderer Arbeiten leicht wiederverwendet werden. Im generischen Codegenerator kommen so z.B. eine Schleifen-Analyse, Low-Level-Optimierungen für den ARM und die Assemblercode-Ausgabe aus der XeLIR-Darstellung zum Einsatz.

5.2 Ausblick

Die Codequalität war bei der Entwicklung des Codegenerators nur zweitrangig. Deshalb gibt es hier noch Optimierungspotential, was z.B. durch High-Level-Optimierungen für Schleifen und für die Auswertung boolescher Ausdrücke oder durch die Kopplung der lokalen und globalen Registerallokation ausgeschöpft werden könnte.

Die in dieser Arbeit entwickelten Techniken können auch in Compilern für andere Prozessoren eingesetzt werden. Am Beispiel der globalen RA wurde gezeigt, dass der Austausch sogar zwischen unterschiedlichen Klassen von Prozessoren möglich ist. Denkbar wären z.B. auch die Wiederverwendung der implementierten High-Level-Optimierungen oder der Codeselektion.

Anhang A

Benchmark-Ergebnisse

A.1 LEON

A.1.1 Ausführungszeit in Taktzyklen

	generischer CG		ENCC mit IA		RTEMS-GCC		ENCC ohne
	absolut	%	absolut	%	absolut	%	IA ($\widehat{=}$ 100%)
Bubble-Sort	167.541	102,18	162.646	99,19	79.344	48,39	163.970
Heap-Sort	690.104	102,65	662.200	98,50	321.405	47,81	672.295
Insertion-Sort	84.609	103,25	81.942	100,00	44.022	53,72	81.942
Multi-Sort	330.970	103,43	319.315	99,79	160.669	50,21	319.993
Quick-Sort	34.056	100,16	33.662	99,01	18.781	55,24	34.000
Selection-Sort	116.795	104,43	111.842	100,00	59.602	53,29	111.842
durchschnittl.		102,68		99,41		51,44	

A.1.2 Anzahl ausgeführter Instruktionen

	generischer CG		ENCC mit IA		RTEMS-GCC		ENCC ohne
	absolut	%	absolut	%	absolut	%	IA ($\widehat{=}$ 100%)
Bubble-Sort	146.811	102,52	141.906	99,10	58.706	41,00	143.197
Heap-Sort	601.608	103,05	575.684	98,61	255.528	43,77	583.802
Insertion-Sort	73.046	103,79	70.381	100,00	30.204	42,91	70.381
Multi-Sort	292.550	103,90	281.096	99,83	118.895	42,22	281.578
Quick-Sort	28.479	100,24	28.412	100,00	14.266	50,21	28.412
Selection-Sort	105.280	104,93	100.330	100,00	43.016	42,87	100.330
durchschnittl.		103,07		99,59		43,83	

A.1.3 Codegröße in Anzahl der Befehlswörter

	generischer CG		ENCC mit IA		RTEMS-GCC		ENCC ohne
	absolut	%	absolut	%	absolut	%	IA ($\widehat{=}$ 100%)
Bubble-Sort	112	99,12	113	100,00	87	76,99	113
Heap-Sort	305	100,99	300	99,34	172	56,95	302
Insertion-Sort	151	102,03	148	100,00	117	79,05	148
Multi-Sort	510	100,59	505	99,61	330	65,09	507
Quick-Sort	237	100,85	235	100,00	139	59,15	235
Selection-Sort	101	99,02	102	100,00	88	86,27	102
durchschnittl.		100,43		99,82		70,59	

A.2 ARM

A.2.1 Ausführungszeit in Taktzyklen

	generischer CG		ENCC mit IA		TCC		ENCC ohne
	absolut	%	absolut	%	absolut	%	IA ($\widehat{=}$ 100%)
Bubble-Sort	400.652	110,01	364.202	100,00	262.622	72,11	364.202
Heap-Sort	1.653.123	106,76	1.532.170	98,95	1.044.445	67,45	1.548.406
Insertion-Sort	198.913	116,08	171.361	100,00	121.285	70,78	171.361
Multi-Sort	810.816	110,21	734.753	99,87	507.089	68,92	735.717
Quick-Sort	89.326	95,93	89.739	96,37	59.103	63,47	93.119
Selection-Sort	297.808	111,71	266.582	100,00	172.934	64,87	266.582
durchschnittl.		108,45		99,20		67,93	

Bei ausschließlicher Nutzung des Stacks zum Zwischenspeichern von Registerinhalten werden folgende Ergebnisse erzielt:

	generischer CG	
	absolut	%
Bubble-Sort	455.495	125,07
Heap-Sort	1.768.632	114,22
Insertion-Sort	243.305	141,98
Multi-Sort	918.680	124,87
Quick-Sort	101.902	109,43
Selection-Sort	342.362	128,43
durchschnittl.		124,00

A.2.2 Anzahl ausgeführter Instruktionen

	generischer CG		ENCC mit IA		TCC		ENCC ohne
	absolut	%	absolut	%	absolut	%	IA (≙ 100%)
Bubble-Sort	118.553	117,39	100.987	100,00	73.186	72,47	100.987
Heap-Sort	489.437	109,18	440.163	98,19	323.414	72,15	448.281
Insertion-Sort	58.664	121,43	48.310	100,00	37.253	77,11	48.310
Multi-Sort	239.797	117,31	203.973	99,78	142.799	69,86	204.419
Quick-Sort	22.227	102,19	21.244	97,67	13.548	62,29	21.751
Selection-Sort	88.211	120,41	73.260	100,00	47.720	65,14	73.260
durchschnittl.		114,65		99,27		69,83	

Bei ausschließlicher Nutzung des Stacks zum Zwischenspeichern von Registerinhalten werden folgende Ergebnisse erzielt:

	generischer CG	
	absolut	%
Bubble-Sort	118.555	117,40
Heap-Sort	489.439	109,18
Insertion-Sort	58.666	121,44
Multi-Sort	239.803	117,31
Quick-Sort	22.501	103,45
Selection-Sort	88.213	120,41
durchschnittl.		114,86

A.2.3 Codegröße in Anzahl der Befehlswörter

	generischer CG		ENCC mit IA		TCC		ENCC ohne
	absolut	%	absolut	%	absolut	%	IA (≙ 100%)
Bubble-Sort	88	114,29	77	100,00	61	79,22	77
Heap-Sort	249	104,62	236	99,16	173	72,69	238
Insertion-Sort	117	115,84	101	100,00	76	75,25	101
Multi-Sort	408	108,22	375	99,47	277	73,47	377
Quick-Sort	193	105,46	180	98,36	136	74,32	183
Selection-Sort	78	109,86	71	100,00	57	80,28	71
durchschnittl.		109,72		99,50		75,87	

A.2.4 Energieverbrauch in Joule

	generischer CG		ENCC mit IA		TCC		ENCC o.
	in μJ	%	in μJ	%	in μJ	%	IA in μJ
Bubble-Sort	6.069.786	110,81	5.477.751	100,00	3.965.824	72,40	5.477.751
Heap-Sort	24.966.194	107,24	23.013.629	98,85	15.707.471	67,47	23.280.549
Insertion-Sort	3.000.501	119,14	2.518.526	100,00	1.809.208	71,84	2.518.526
Multi-Sort	12.274.025	111,02	11.040.269	99,86	7.612.593	68,85	11.056.117
Quick-Sort	1.352.556	95,30	1.366.132	96,25	898.444	63,30	1.419.313
Selection-Sort	4.508.583	112,66	4.002.073	100,00	2.572.356	64,28	4.002.073
durchsch.		109,36		99,16		68,02	

A.3 M5-DSP

A.3.1 Ausführungszeit in Taktzyklen

	mit globaler RA		ohne globale RA
	absolut	%	($\widehat{=}$ 100%)
dot_prod	8	53,33	15
fir2sim	9.728	46,80	20.785
lms	112	46,67	240
matrix1	4.200	48,11	8.730
mat1x3	36	49,32	73
matrix2	4.200	53,63	7.831
convolution	32	27,83	115
n_real_up	320	71,27	449
biquad	240.600	93,74	256.680
fir	65.764	56,94	115.489
lattice	198.659	63,14	314.632
hamming	3.842	55,02	6.983
cmultiply	8.192	66,66	12.289
mdct	4.680	36,72	12.744
durchschnittl.		54,94	

A.3.2 Anzahl Speicherzugriffe

	mit globaler RA		ohne globale RA
	absolut	%	($\widehat{=}$ 100%)
dot_prod	32	71,11	45
fir2sim	65.144	76,94	84.665
lms	531	78,55	676
matrix1	21.478	70,84	30.319
mat1x3	173	69,48	249
matrix2	19.978	63,79	31.319
convolution	185	68,77	269
n_real_up	776	66,84	1.161
biquad	681.867	95,50	713.987
fir	247.024	71,30	346.473
lattice	823.274	84,38	975.729
hamming	15.872	78,60	20.193
cmultiply	24.584	88,89	27.657
mdct	19.808	57,58	34.398
durchschnittl.		74,47	

Literaturverzeichnis

[Adv] Advanced RISC Machines Ltd (ARM). http://www.arm.com/.

[Adv95a] Advanced RISC Machines Ltd (ARM). *An Introduction to Thumb*, März 1995. Version 2.0.

[Adv95b] Advanced RISC Machines Ltd (ARM). *ARM7TDMI Data Sheet*, 1995.

[Adv99] Advanced RISC Machines Ltd (ARM). *ARM7TDMI Technical Reference Manual*, März 1999.

[AG98] A. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[Bas95] Steven Bashford. Code Generation Techniques for Irregular Architektures. Technical Report 596, University of Dortmund, Lehrstuhl Informatik XII, November 1995.

[Bas01] Steven Bashford. *Constraintbasierte Codegenerierung für eingebettete Prozessoren*. PhD thesis, Universität Dortmund, Lehrstuhl Informatik XII, 2001.

[Enc] ENCC. http://ls12-www.cs.uni-dortmund.de/research/encc/.

[Ert99] M. Anton Ertl. Optimal Code Selection in DAGs. *Principles of Programming Languages*, 1999.

[FH95] C.W. Fraser and D.R. Hanson. *A Retargetable C Compiler: Design and Implementation*. The Benjamin/Cummings Publishing Company, Inc., 1995.

[FHP92] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems, 1(3)*, September 1992.

[Fie01] Markus Fiesel. XML-basierte generische Zwischendarstellung für Compiler. Master's thesis, Universität Dortmund, Lehrstuhl Informatik XII, Dezember 2001.

[FWD+98] G. Fettweis, M. Weiss, W. Drescher, U. Walther, F. Engel, and S. Kobayashi. Breaking new grounds over 3000 MOPS: A broadband mobile multimedia modem DSP. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT)*, pages 1547–1551, Toronto, Canada, September 1998.

[Gai] Gaisler Research. http://www.gaisler.com/.

[Gai01] Gaisler Research. *The LEON Processor User's Manual, Version 2.3.3*, Mai 2001.

[Gcc] GCC Compiler. http://gcc.gnu.org/.

[GeL] GeLIR. http://ls12-www.cs.uni-dortmund.de/research/gelir/.

[GG78] R. S. Glanville and S.L. Graham. A new method for compiler code generation. *Proc. 5th ACM symp. on Principles of Programming Languages*, pages 231–240, 1978.

[Hor01a] Lars Hornbach. Generische Low-Level Optimierungen für RISC-Architekturen. Master's thesis, Universität Dortmund, Lehrstuhl Informatik XII, November 2001.

[Hor01b] Martin Horst. Schleifenoptimierungen zur Ausnutzung paralleler Rechenwerke von Prozessoren der M3-DSP Plattform. Master's thesis, Universität Dortmund, Lehrstuhl Informatik XII, November 2001.

[Int] Intel XScale. http::/www.intel.com/design/intelxscale/.

[Lan] LANCE retargetable Compiler. http://www.icd.de/es/lance/lance.html.

[Lcc] LCC Compiler. http://www.cs.princeton.edu/software/lcc/.

[Lis] Institute for Integrated Signal Processing Systems (ISS). http://www.iss.rwth-aachen.de/lisa/.

[Lor03] Markus Lorenz. *Performance- und energieeffiziente Compilierung für digitale SIMD-Signalprozessoren mittels genetischer Algorithmen.* PhD thesis, Universität Dortmund, Lehrstuhl Informatik XII, 2003.

[Mar03] P. Marwedel. Skript zur Vorlesung „Rechnerarchitektur/Rechensysteme". Universität Dortmund, Lehrstuhl Informatik XII, April 2003.

[Mip] Mips. http://www.mips.com/.

[Mot] Motorola Dragonball. http://e-www.motorola.com/.

[MSi] Model Technology. http://www.model.com/products/se.asp.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[Sch02] Sergej Schwenk. Entwicklung eines Energiemodells für den LEON Prozessor. Master's thesis, Universität Dortmund, Lehrstuhl Informatik XII, März 2002.

[Spa99] SPARC International, Inc. *The SPARC Architecture Manual*, 1999.

[Ste02] Stefan Steinke. *Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik.* PhD thesis, Universität Dortmund, Lehrstuhl Informatik XII, 2002.

[Tei97] Jürgen Teich. *Digitale Hardware-/Software-Systeme*. Springer-Verlag, 1997.

[Tji93] S. Tjiang. An Olive Twig. Technical report, Synopsys Inc., 1993.

[WFL+99] M.H. Weiss, G. P. Fettweis, M. Lorenz, R. Leupers, and P. Marwedel. Toolumgebung für plattformbasierte DSPs der nächsten Generation. In *Proceedings of DSP Deutschland*, pages 175–184, Munich, Germany, September 1999.

[WHFS02] Ingo Wegener, Thomas Hoffmeister, Paul Fischer, and Detlef Sieling. Skript zur Vorlesung „Effiziente Algorithmen“, 2002.