Entwurf eines Compilerbackends für einen konfigurierbaren Vektorprozessor

Hannes Klas
hannes.klas@googlemail.com
Hochschule Augsburg

29. Januar 2010

Erstellungserklärung

Hiermit erkläre ich, dass die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet wurden.

Zusammenfassung

Die Diplomarbeit “Entwurf eines Compilerbackends für einen konfigurierbaren Vektorprozessor” beschäftigt sich mit der Portierung des C-Compilers der GNU Compiler Collection (GCC) auf den konfigurierbaren Vektorprozessor HiCoVec.

Hierfür werden zunächst Grundlagen von Compilern im Allgemeinen, der Architektur des HiCoVec sowie des GCC erklärt.

Anschließend wird detailliert erklärt, wie der portierte Compiler funktioniert, sowie welche Anpassungen am Prozessor durchgeführt wurden.

Das Ergebnis ist ein eingeschränkt lauffähiger C-Compiler, dessen Kompilate auf einem Simulator (HiCoSim) verifiziert wurden. Weiterhin wird erklärt, wie der GCC zu portieren ist, und was dabei beachtet werden muss. Die Verifikation lief mit einer Sammlung C-Programme ab, deren Ergebnis und Laufzeit mit einem aktuellen PC verglichen wurde.

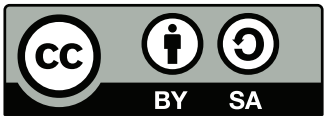
Abstract

The thesis “draft of a compiler backend for a configurable vector processor” deals with the port of the C Compiler of the GNU Compiler Collection (GCC) to the configurable vector processor HiCoVec.

For making this topic easier to understand, this document explains compiler basics, the architecture of HiCoVec and the architecture of GCC first.

The ported compiler is explained in detail, as well as which changes were made to the processor.

The result is a working C compiler with limited features. The programs it built were verified on a simulator (HiCoSim). We’ll explain how to port the GCC to a new processor, as well as the difficulties encountered while doing this. The verification was done with a collection of C programs, which were run on the simulator and a current PC. The results and performance between these targets were compared.



Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Der HiCoVec ist nach Wissen des Autors der derzeit einzige konfigurierbare Vektorprozessor, der als Open-Source verfügbar ist. Er ist ein leistungsfähiger, lauffähiger Prozessor, der primär auf FPGAs eingesetzt werden soll. Die Entwicklung des Prozessors und benötigter Entwicklungswerkzeuge ist dabei ein aktives Projekt, das an der Hochschule Augsburg durchgeführt wird.

Ein Compiler für die Sprache C wird dabei als Kernstück der Entwicklungswerkzeuge betrachtet und wird benötigt, um komplexere Programme überhaupt schreiben zu können.

Da bereits einige gute Open-Source-C-Compiler existieren (Open Watcom, pcc, gcc) ist es hierbei sinnvoller, einen der existierenden Compiler zu portieren, anstatt einen neuen Compiler zu entwickeln. Hierbei kann davon profitiert werden, dass die Compiler selbst weit verbreitet und getestet sind. Zusätzlich werden fortgeschrittene Optimierungen unterstützt, die bei einem eigenen Compiler nur mit sehr hohem Aufwand umsetzbar wären.

In dieser Arbeit wurde der GCC gewählt, da er einen sehr hohen Verbreitungsgrad besitzt, und sehr weit entwickelte Möglichkeiten zur Optimierung bietet. Zusätzlich ist er einer der am meisten portierten Compiler überhaupt, weswegen davon ausgegangen werden kann, dass er in Bezug auf die Prozessorarchitektur sehr flexibel anpassbar ist.

1.2 Ziel dieser Arbeit

Ziel der Arbeit ist es, ein initiales Backend des GNU C Compilers für den Prozessor HiCoVec zu portieren. Der Schwerpunkt liegt komplett auf der Umsetzung von C-Konstrukten für die Skalareinheit. Die Vektoreinheit wird nur als Zwischenspeicher verwendet und sonst nicht beachtet.

Der Funktionsumfang wird hierbei auf eine sinnvolle Untermenge der normalen GCC-Features beschränkt, um die Implementierung möglichst einfach zu halten. Hierbei werden Funktionen unterstützt, Strukturen, Arrays, Ganzzahlarithmetik sowie alle Vergleiche und Schleifen.

Die Vektoreinheit kann hierbei verwendet werden, indem vom Programmierer in seinem C-Programm Assemblercode eingebettet wird. Viele Algorithmen, die gut vektorisierbar sind, können so komfortabel umgesetzt werden.

1.3 Aufbau dieser Arbeit

Zunächst werden im Kapitel 2 Begriffe und grundlegende Zusammenhänge erklärt. Dazu gehören der grobe Aufbau des Prozessors sowie ein Überblick über den Aufbau des GCC.

Anschließend werden im Kapitel 3 der Entwurf und die Implementierung des Backends erklärt und dabei aufgetretene Probleme erläutert.

Im Kapitel 4 wird eine Auswahl der getesteten C-Programme erklärt, sowie der generierte Assemblercode ausschnittsweise betrachtet, um Einzelheiten des Backends zu beschreiben. Eine Bewertung des Compilers und eine Auswertung der Performance der generierten Programme befindet sich in Kapitel 5.

Im Kapitel 6 werden die Ergebnisse diskutiert sowie mögliche Fortführungen vorgeschlagen. Hierzu gehören auch Änderungsvorschläge für den HiCoVec selbst, mit denen ein besserer Compiler möglich wäre.

Längere Code-Abschnitte werden im Anhang untergebracht, um den Lesefluss innerhalb der Arbeit nicht zu stören.

2 Grundlagen

2.1 Begriffe

2.1.1 Compiler

Ein Compiler ist ein Programm, welches ein in einer bestimmten Sprache geschriebenes Programm in eine andere Sprache übersetzt. Hierbei wird in der Literatur zwischen verschiedenen Typen unterschieden. So nennt man einen Übersetzer Assemblercode nach Binärcode üblicherweise Assembler, einen Übersetzer in eine Hochsprache oft Translator und einen Übersetzer in Assemblercode Compiler. In dieser Arbeit wird nur ein Übersetzer nach Assemblercode betrachtet.

Ein Compiler arbeitet hierbei meist in zwei konzeptionellen Stufen.

Parsen

Beim Parsen wird der Eingangscode eingelesen und in einen abstrakten Syntaxbaum (AST) abgebildet. Dies ist eine abstrakte Darstellung, auf der sich maschinell wesentlich besser arbeiten lässt als auf dem Text selbst.

Hierbei gibt es üblicherweise Knotentypen für jede direkt ausführbare arithmetische Operation, Addressen, Variablen, Zuweisungen und Funktionsaufrufe. Der Syntaxbaum wird von einem Parser aufgebaut, und später von anderen Teilen des Compilers weiterverarbeitet (vgl. [GBJL00], Seite 9ff).

Auf dieser Ebene werden schon einfache Optimierungen vorgenommen, wie die Auflösung von konstanten Rechenoperationen.

Ein vereinfachter AST befindet sich als kurzes Beispiel für den folgenden C-Code befindet sich in der Abbildung 2.1. Dieser AST ist insofern vereinfacht, dass er keine zusätzlichen Annotationen wie die Typen der Variablen enthält, und dient daher nur zum groben Verständnis.

```
1    x = y + 2 * f(z);
```

Codegeneration

Sobald ein fertiger AST vorliegt, findet die Codegenerierung statt. Hierfür wird der Syntaxbaum traversiert und für jeden Knoten Assemblercode erzeugt. Dabei muss für jeden verwendeten Knotentyp vorgefertigter Assemblercode vorhanden sein. In dieser Stufe können auch noch Optimierungen durchgeführt werden, die in einem lokalen Kontext arbeiten. Ein Beispiel hierfür ist das Entfernen von redundanten Load-Anweisungen.

Falls ein Typ auf der Maschine nicht direkt darstellbar ist, muss dieser entweder emuliert werden, oder er darf nicht vom Compiler generiert werden. Ein typisches Beispiel hierfür sind die Branch-Fälle. Dieser Vorgang wird im Kapitel 3.4.1 weiter ausgeführt.

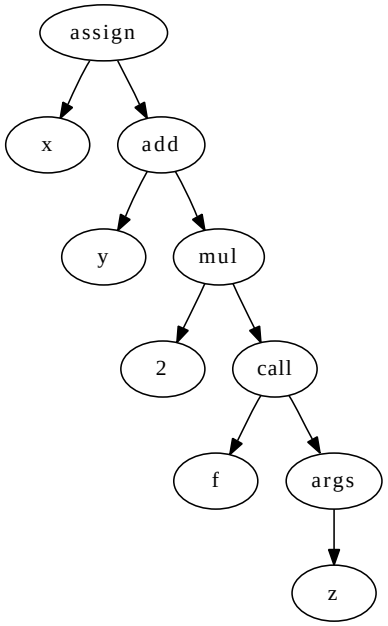


Abbildung 2.1: Zuweisung - C-Code

2.1.2 Konfigurierbare Prozessoren

Unter einem konfigurierbaren Prozessor versteht man einen Mikroprozessor, dessen Funktionalität konfigurierbar ist [IL06]. Dies bedeutet zum Teil die Existenz ganzer Einheiten (FPU, MMU), zum Teil die Parametrisierung (Shiftbreite, Anzahl der Register). Beim HiCoVec ist dies derzeit die Breite der Vektoren, die Anzahl der Vektorregister, die Verwendbarkeit eines Multiplikationsbefehls sowie die mögliche Weite von Shift-Befehlen.

Andere Prozessoren unterstützen die Konfiguration einer Pipeline, einer FPU oder einer MMU. In der Regel lassen sich auch zusätzliche Rechenbefehle implementieren, wodurch eine bessere Anpassung an die Problemstellung möglich ist.

Die Konfigurierbarkeit wird in dieser Arbeit nur in sofern beachtet, dass mindestens ein Vektorregister vorhanden sein muss.

2.1.3 Application Binary Interface (ABI)

Ein Application Binary Interface (ABI) ist eine Beschreibung der Schnittstelle, mit der Subroutinen aufgerufen werden können. Sie beschreibt dabei in erster Linie, wie Parameter zwischen Routinen übergeben werden können.

Dazu gehört zunächst der Aufbau des Stacks im Speicher. Es muss definiert werden, durch welche Register der Stack addressiert wird (Stackpointer und Base/Framepointer[1]), in welche Richtung er wächst, und welche Plätze reserviert bleiben. Die reservierten Plätze können für das Speichern des Return-Pointers und des alten Basepointers verwendet werden.

[1]Im entwickelten Compiler sind Basepointer und Framepointer identisch und werden synonym verwendet

2.2 Der HiCoVec-Prozessor

Der HiCoVec-Prozessor wurde in einer vorhergehenden Diplomarbeit [Man07] ausgearbeitet. Er ist dabei ein komplett freier konfigurierbarer Prozessor, der an der Hochschule Augsburg entstanden ist. Er liegt in Quellen als VHDL-Code vor, ausserdem existiert ein in Java geschriebener Simulator namens HiCoSim. Das vollständige Projekt mit allen Teilprojekten befindet sich unter [Kie10].

Die Architektur des HiCoVec besteht konzeptionell aus 2 Einheiten, der Skalar- und der Vektor-Einheit. Eine Instruktion für den HiCoVec besteht dabei aus 32 Bit, wobei die oberen 12 Bit den Befehl der Skalar- und die unteren 20 den Befehl der Vektoreinheit darstellen.

Die Skalareinheit ist eine einfache herkömmliche CPU. Sie ist eine Akkumulatormaschine mit 3 Registern (A, X und Y) sowie Flags für Carry und Zero.

Die Vektoreinheit hingegen ist mit der SSE-Einheit von x86-Prozessoren vergleichbar. Sie hat eigene Vektorregister, auf denen sie parallel operiert. Hierbei ist die Breite der Vektorregister (also die Anzahl der Worte pro Register) und die Anzahl der Vektorregister zur Synthesezeit einstellbar. Die Befehle umfassen 20 Bit und sind ebenfalls auf 3 Operanden kodiert. Dabei werden bei allen arithmetischen Befehlen zwei komplette Vektorregister miteinander verrechnet und in ein drittes abgespeichert. Dies funktioniert, indem die Recheneinheiten N mal repliziert werden.

Zusätzlich bestehen kooperative Befehle zwischen den beiden Einheiten. Diese werden für mehrere Zwecke verwendet. Zunächst können Daten zwischen beiden Einheiten transferiert werden. Dies wird zum Beispiel angewendet, wenn die Skalareinheit die Vektoreinheit mit Daten befüllt, die dann parallel von dieser abgearbeitet werden.

Ein weiterer Einsatzzweck für kooperative Befehle stellt der Speicherzugriff von der Vektoreinheit dar. Hier berechnet die Skalareinheit die effektiven Addressen, während die Vektoreinheit die Daten liest oder schreibt.

2.2.1 Befehle

Im folgenden werden nur die in dieser Arbeit benötigten Befehle erklärt. Dies bedeutet, dass die meisten Vektorbefehle weggelassen werden. Diese können in [Man07] nachgelesen werden. Die Skalarbefehle und deren Einfluss auf die Flags können der Tabelle in Abbildung 2.2 entnommen werden.

Grundsätzlich gilt, dass alle Immediate-Werte als 16-Bit Signed Integer interpretiert werden. Dies bezieht sich auch auf Addressen bei den Befehlen LD und ST, die als Addresse intern den normalen ADD-Befehl durchführen. Dadurch sind relativ zu einem Register die Addressen $\pm$ 32k möglich. Bei absoluten Addressen wird als erstes Offset 0 verwendet, sodass effektiv die Addressen 0-32767 und die Addressen 4294967295-4294934527 zur Verfügung stehen.

In dieser Tabelle wird folgende Konvention verwendet:

axy0 Register A, X, Y oder der Immediate-Wert 0

axyi Register A, X, Y oder ein Immediate-Wert zwischen -32768 und 32767

axy Register A, X oder Y

arg0 Zielregister

arg1 Quellregister 1

arg2 Quellregister 2

c Carryflag vor der Operation

z Zeroflag vor der Operation

res Ergebnis der Operation

loaded value Geladener Wert bei LD

pc Program Counter – Register mit aktueller Position im Programm

2.3 Aufbau der GNU Compiler Collection

Die GNU Compiler Collection besteht aus Front-Ends, dem sogenannten Mid-End und mehreren Back-Ends. Alle davon kommunizieren über die Registertransfersprache RTL. Diese Sprache kann dabei als abstrakter, deklarativer Lisp-Dialekt verstanden werden. Auf diese Sprache wird später noch genauer eingegangen werden. Gesteuert werden diese ganzen Programmteile von einem sogenannten Driver, der den Kompilierungsprozess startet und die passenden Programmteile auswählt. Die Sprache RTL wird auch als RTX bezeichnet, beide Begriffe werden hier synonym verwendet.

Das Front-End ist dabei für eine Programmiersprache zuständig. Es parst eine Datei in einer Sprache (zum Beispiel C, Ada oder Fortran) und übersetzt es in die RTL.

Das Middle-End kann als der Kern verstanden werden. Hier wird die RTL analysiert und optimiert. Dieser Ansatz mit explizitem Middle-End hat den Vorteil, dass alle unterstützten Prozessoren und Programmiersprachen von zusätzlichen Optimierungsmethoden sofort profitieren, sobald die neue Methode im Middle-End implementiert ist.

Das Back-End baut abschließend die RTL um in maschinenspezifischen Code. Dabei wird sehr viel Arbeit innerhalb des Backends von der Compilerinfrastruktur abgenommen. Man muss nur eine verhältnismäßig kleine Anzahl an Eigenschaften und Befehlen der Maschine vorgeben. Es muss zum Beispiel nur die Anzahl der Register vorgeben werden, die GCC kümmert sich um deren Verteilung. Dabei können auch viele Zugriffe bei der Optimierung entfernt werden, falls der benötigte Wert noch von einer anderen Berechnung zur Verfügung steht.

2.3.1 Buildprozess

Der Buildprozess des GCC verläuft in mehreren Etappen, die als Stages bezeichnet werden. Der erste Durchlauf erzeugt dabei eine unoptimierte Binary namens xgcc, mit der die libgcc (vgl. Abschnitt 2.3.9) kompiliert werden soll. Hierbei entsteht ein minimaler C-Compiler, mit

Operation	Carryflag	Zeroflag	Erklärung
LD axy0, [axy0 + axyi]	-	loaded value = 0	OP0 = [OP1 + OP2]
ST [axy0 + axyi], A	-	-	[OP1 + OP2] = A
ADD axy0, axy0, axyi	res > 0xffffffff	res = 0	OP0 = OP1 + OP2
ADC axy0, axy0, axyi	res > 0xffffffff	res = 0	OP0 = OP1 + OP2 + Carry
INC axy0, axy0	res > 0xffffffff	arg2 = 0xffffffff	OP0 = OP1 + 1
SUB axy0, axy0, axyi	arg3 <= arg2	res = 0	OP0 = OP1 - OP2
SBC axy0, axy0, axyi	c=0: SUB; c=1: 1	res = 0	OP0 = OP1 - OP2 + Carry
DEC axy0, axy0	arg2 != 0	arg2 = 1	OP0 = OP1 - 1
AND axy0, axy0, axy0	0	res = 0	OP0 = OP1 & OP2
OR axy0, axy0, axy0	0	res = 0	OP0 = OP1 | OP2
XOR axy0, axy0, axy0	0	res = 0	OP0 = OP1 ^ OP2
LSL axy0, axy0	arg2 & 0x80000000	res = 0	OP0 = OP1 « 1
LSR axy0, axy0	arg2 & 1	res = 0	OP0 = OP1 » 1
ROL axy0, axy0	arg2 & 0x80000000	res = 0	OP0 = OP1 « 1 | Carry
ROR axy0, axy0	arg2 & 1	res = 0	OP0 = (Carry « 31) | OP1 » 1
MUL axy0, axy0, axyi	0	res = 0	OP0 = (OP1 * OP2) & 0xffff
JMP [axy0 + axyi]	-	-	goto OP0 + OP1
JAL axy0, [axy0 + axyi]	-	-	goto OP1 + OP2, OP0 = PC + 1
JZ [axy0 + axyi]	-	-	goto OP0 + OP1 if Zero == 1
JNZ [axy0 + axyi]	-	-	goto OP0 + OP1 if Zero == 0
JC [axy0 + axyi]	-	-	goto OP0 + OP1 if Carry == 1
JNC [axy0 + axyi]	-	-	goto OP0 + OP1 if Carry == 0
CLZ	-	0	Zero = 0
SEZ	-	1	Zero = 1
CLC	0	-	Carry = 0
SEC	1	-	Carry = 1
HALT	-	-	Stop
NOP	-	-	Do Nothing
VNOP	-	-	Do Many Nothing
MOV axy, r0-15(axy0)	-	-	OP0 = OP1(OP2)
MOV r0-15(axy0), axy	-	-	OP0(OP1) = OP2

Abbildung 2.2: Befehlssatz (eingeschränkt)

dem der GCC sich selbst neu baut, und dabei auch die anderen aktivierten Frontends erzeugt. Dieser Aufbau ist unter [Rau03] auf Folie 39 noch genauer beschrieben. Da der gcc-hicovec derzeit noch nicht in der Lage ist, sich selbst zu kompilieren, entfällt dieser Schritt.

Auf die nötigen Anpassungen des Buildprozesses wird in Abschnitt 3.1 noch genauer eingegangen.

2.3.2 Backend

Das Backend besteht aus einer aufwändigen Maschinenbeschreibung in mehreren Dateien. Im folgenden Text wird unter `TARGET` die Architektur verstanden, für die kompiliert werden soll, hier also *hicovec*. Die wichtigsten sind dabei die Machine Description (`TARGET.md`) und die C-Dateien (`TARGET.c` und `TARGET.h`). Diese werden vom GCC immer erwartet. Weitere Dateien können hinzugefügt werden und müssen dann von Hand eingebunden werden. Komplexere Backends verwenden dies zum Beispiel, um Prädikate für Operanden auszugliedern. So definiert das Backend des x86 die Prädikate `fp_register_operand` (i387-Register) und `any_fp_register_operand` (i387- oder SSE-Register). Diese Prädikate wären schwer zu finden, wenn sie in der Hauptdatei stehen würden.

Die grundlegenden Eigenschaften werden als C-Makros in der Datei `TARGET.h` definiert. Hierzu gehören alle Eigenschaften der Maschine, die sich nicht direkt auf mögliche Befehle beziehen, wie die Anzahl und Verwendung der Register oder dem Speicheraufbau. Wenn die Makros zu komplex werden, empfiehlt es sich, in Funktiosaufrufe zu expandieren, die in `TARGET.c` abgelegt werden.

In der Datei `TARGET.md` werden Patterns definiert, mit denen die Maschine auf Befehlsebene beschrieben wird. Hierbei müssen einige Standardpatterns (vgl. [Sta92], Kapitel *Standard Names*) immer umgesetzt werden, da sie in der RTL-Generierung schon benötigt werden. Diese Patterns sind in Kapitel 2.3.7 aufgelistet. Beispiele dafür befinden sich in Kapitel 3.4.1.

2.3.3 Interne Sprachen

Der GCC verwendet während des Kompilierungsvorgangs mehrere Sprachen. Ein Frontend kann die Sprache GENERIC oder GIMPLE erzeugen, was unter [Sta92] in den Kapiteln 8 und 8.2 genauer beschrieben wird. Im Fall von GENERIC muss ein zusätzlicher Durchlauf stattfinden, um GIMPLE zu erzeugen. Alle High-Level-Optimierungen verwenden dabei GIMPLE, was unter [Sta92] im Kapitel 8.4 nachgelesen werden kann. Weitere Informationen über die Sprache selbst finden sich unter [Sta92] in den Kapiteln 11 und 12.

Aus dem fertig verarbeiteten GIMPLE wird dann in mehreren Stufen RTX erzeugt. Hierbei werden die in Abschnitt 2.3.7 beschriebenen Patterns in der Generation der RTX verwendet und müssen daher definiert sein.

2.3.4 Befehlsmodi

Zu jedem Knotentyp in den verwendeten Sprachen wird ein sogenannter Modus zugeordnet. Darunter versteht man hier den Datentyp, auf den diese Operation gilt. Diese können in [Sta92] im Kapitel “Machine Modes” genauer nachgelesen werden. Die Wichtigsten davon sind:

SI Single Integer - Native Breite des CPU-Worts (hier: 32Bit)

SF Single Float - 32Bit Float

CC Condition Code - Flag Register

VOID Unspezifizierter Modus. Wird zum Beispiel für Konstanten verwendet, die in beliebige Integer passen.

BLK Block - Zusammengesetzter Datentyp, der in kein anderes Format passt.

HI Half Integer - Halbes Wort (16Bit)

QI Quarter Integer - 8Bit

DI Double Integer - 64Bit

DF Double Float - 64Bit

2.3.5 RTX Kurzreferenz

RTX lässt sich vergleichbar wie Lisp lesen, ist allerdings im Gegensatz dazu rein deklarativ. Die wichtigste Direktive ist `define_insn`, mit der neue Instruktionen angelegt werden können. Hierauf wird in den Beispielen in 3.4.1 noch genauer eingegangen. Zusätzlich kann die Beschreibung der Patterns in der Arbeit [Nil00] gelesen werden.

Ein Pattern ist hierbei ein Muster von RTX-Instruktionen, die erkannt (gematcht) werden können. Wenn ein Pattern gematcht wird, wird Code generiert. Wie dies genau funktioniert kommt auf den verwendeten Definitionstyp an.

Die Definition `define_insn` ist am intuitivsten zu verstehen – Wenn das definierte Pattern gematcht werden kann, wird der zugehörige Assemblercode direkt in die Ausgabe eingebettet.

Mit `define_expand` kann eine Expansion definiert werden. Hierbei ist es möglich, eine komplexe Instruktion durch mehrere Primitive auszudrücken. Ist die Generierung erfolgreich kann dies durch das Präprozessormakro DONE signalisiert werden. Wenn sie generell nicht möglich ist kann FAIL verwendet werden - in diesem Fall versucht der GCC, den umliegenden Code weiter zu expandieren bis die Generierung erfolgreich läuft. Falls gar nichts signalisiert wird, oder das Expansionspattern komplett leer ist, wird das Pattern nur zur RTX-Generierung verwendet. Es muss daher ein Insn-Pattern vorliegen, das den Code später korrekt expandiert.

Zusätzlich gibt es noch `define_split`. Hiermit können Instruktionspatterns definiert werden, die in mehrere Teile getrennt werden müssen. Dies kann zum Beispiel bei Architekturen mit Branch Delay Slot der Fall sein. Hier kann der Code direkt vor dem Branch im Delay Slot untergebracht werden, solange der Branch nicht vom Code abhängt. Da im HiCoVec keine derartigen Instruktionen vorhanden sind wurden Splits nicht verwendet.

Ein vereinfachtes Pattern wird in Beispiel 2.3 gezeigt. Dieses Beispiel ist insofern vereinfacht, dass nur Registeroperanden erlaubt sind. Der Compiler würde also nur noch mit Registern rechnen, und Konstanten immer vorher in ein Register laden.

Zunächst wird in Zeile 1 mit `define_insn` ein neues Pattern namens `addsi3` definiert. Diese Definition matcht auf das Pattern in den Zeilen 2-5 (Argument 2 für `define_insn`). Das Pattern ist wie folgt zu lesen:

Setze (match0) auf den Wert von (match1) + (match2).

(match0) ist dabei der erste (beginnend bei 0) Wert, der von dem Pattern gematcht wird. Er wird im Modus SI (Single Integer) angesprochen und ist ein Register (Prädikat: "register_operand"). Zusätzlich erhält er das Constraint, dass er ein Register ist ("r") und mit einem sinnvollen Wert überschrieben wird ("="). Die Constraints können zusätzlich noch in Abschnitt 4.3 nachgeschlagen werden. Die Operanden (match1) und (match2) lassen sich genauso lesen, nur ohne das Zuweisungsconstraint. Das bedeutet, dass aus diesen Registern nur gelesen wird.

Aus diesem Beispiel ist erkennbar, dass sich die Bedeutung von Prädikaten und Constraints zum Teil überlappen. Hierbei kann man die Faustregel verwenden, dass beim RTL-Aufbau primär die Prädikate genutzt werden, bei der Codegeneration selbst die Constraints. Man sollte *immer* beide angeben, auch wenn diese redundant erscheinen. Fehlen zum Beispiel die Constraints, läuft die RTL-Generierung komplett durch, der Compiler scheitert aber dann bei der Codegenerierung, da er kein passendes Pattern findet.

In Zeile 6 kann zusätzlicher C-Code angegeben werden, der entscheidet, ob dieses Pattern überhaupt zutrifft. Diese Möglichkeit wird primär verwendet, wenn bei unterschiedlichen Prozessorversionen Features hinzugefügt werden.

In Zeile 7 befindet sich die Codegeneration. Hierbei bietet `define_insn` mehrere Möglichkeiten. Zunächst kann der Text wie im Beispiel direkt in den Assemblercode ausgegeben werden. Dann gibt es die Methode der Fallunterscheidung, wenn das Pattern auf unterschiedliche Argumentconstraints gleichzeitig matchen kann. Dies wird später bei den MOV-Patterns benötigt. Abschließend besteht noch die Möglichkeit, hier direkt C-Code einzubinden, der den Code dann generiert. Hiermit wird die höchste Mächtigkeit erreicht, da in diesem Code der vollständige Instruktionsstrom des GCC analysiert werden kann, und in Abhängigkeit davon möglichst effizienter Code generiert werden kann. Dies wird später bei den Sprungbefehlen verwendet.

In Zeile 8 wird zusätzlich noch ein Attribut für den Condition Code gesetzt (vgl. 2.3.6). Dieses Attribut besagt, dass der Befehl sowohl die Flags Carry als auch Zero setzen kann. Das Attribut wird dann später vom entsprechenden Code ausgewertet.

2.3.6 Condition Codes

Der GCC bietet eine mächtige Einrichtung, um bedingte Sprünge zu verwalten, die Condition Codes. Diese können in [Sta92] in Kapitel "Target Macros → Condition Code" nachgeschlagen werden.

Grundsätzlich gibt es hier zwei Möglichkeiten, die unterstützt werden:

- Ein Prozessor kann ein Statusregister haben, das von Befehlen gesetzt wird und in Sprungbefehlen implizit verwendet wird.
- Ein Prozessor kann Befehle haben, um bestimmte Eigenschaften in normale Register zu schreiben, die von den Sprungbefehlen explizit verwendet werden.

```
1 (define_insn "addsi3"
2  [(set (match_operand:SI 0 "register_operand" "=r")
3    (plus:SI
4     (match_operand:SI 1 "register_operand" "r")
5     (match_operand:SI 2 "register_operand" "r")))]
6  ""
7  "add %0, %1, %2"
8  [(set_attr "cc" "set_cz")])
```

Abbildung 2.3: Addition - Vereinfacht

Beide Möglichkeiten werden vom GCC unterstützt und können unterschiedlich angesprochen werden. Beim HiCoVec wird das Statusregister verwendet.

Das verwendete Register wird im Code immer als `cc0` bezeichnet.

2.3.7 Benötigte Patterns

Die hier beschriebenen Patterns werden für einen minimal lauffähigen GCC-Port auf jeden Fall benötigt. Die restlichen werden primär für zusätzliche Optimierungen gebraucht, oder können in die libgcc (vgl. 2.3.9) als Funktion ausgelagert werden.

Diese Patterns sind dabei Erfahrungswerte aus den getesteten C-Programmen und sind daher nicht zwansläufig vollständig. Mit dieser Liste hat der GCC die getesteten Programme schon erfolgreich umgesetzt, alle weiteren bewirken nur besseren Code oder bessere Verständlichkeit des Backends.

addsi3 Addieren der Operanden

subsi3 Subtrahieren der Operanden

cmpsi ODER (tstsi UND setCC) Setzt die Flags durch einen Vergleich

andsi3 Logische Verknüpfung mit UND

iorsi3 Logische Verknüpfung mit ODER

xorsi3 Logische Verknüpfung mit XODER

movsi Kopieren von Operand 1 nach Operand `0`

nop No Operation

beq Sprung falls gleich

bne Sprung falls ungleich

bgt Sprung falls größer

bge Sprung falls größer oder gleich

blt Sprung falls kleiner

ble Sprung falls kleiner oder gleich

jump Absoluter Sprung

indirect_jump Indirekter Sprung

call Absoluter Funktionsaufruf, Rückgabewert wird ignoriert

call_value Absoluter Funktionsaufruf

epilogue ODER return Zurückspringen aus einer Funktion

Zur Referenz sollte unter [Sta92] das Kapitel “Standard Names” herangezogen werden.

Besondere Beachtung verdient das Pattern `movsi`, dass für alle möglichen Kombinationen von Operanden immer auflösbar sein muss. Beispielsweise muss ein indiziertes Kopieren des Stackpointers in den Speicher und zurück möglich sein, da der Compiler dies bei einem Funktionsaufruf selbstständig generiert.

2.3.8 Registerallokation

Ein Teil des Kompilierungsvorgangs ist die Registerallokation. Als Referenz sei neben diesem Abschnitt unter [Sta92] das Kapitel 8.5 “RTL Passes” als Lektüre empfohlen. Auf die verwendete Registerdefinition wird in Abschnitt 3.3.4 noch genauer eingegangen.

Der GCC verwendet beim Kompilieren zunächst ein vereinfachtes Schema für die Registerallokation, bei der davon ausgegangen wird, dass unendlich viele Pseudoregister zur Verfügung stehen. Die meisten Optimierungen finden bereits mit dieser Repräsentation statt.

Vor der Codegeneration läuft der Registerallokator und versucht, die Pseudoregister auf die echten Register abzubilden. Diese echten Register werden als harte Register bezeichnet. Je mehr Register dem Allokator zur Verfügung stehen, desto besser wird das Ergebnis. Wenn die Register ausgehen, müssen die übrigen Register temporär in den Stack ausgelagert werden. Diesen Vorgang bezeichnet man als Spill.

Wenn Register ausgelagert werden müssen, durchläuft der GCC den Code und sucht nach Instruktionen, die durch den Spill invalidiert werden. Dies ist der Fall, wenn die Instruktion davon abhängt, dass ein Wert in einem spezifischem Register liegt.

Vor diesen Instruktionen werden dann zusätzliche Instruktionen eingefügt, die das Register rechtzeitig neu laden. Dies bezeichnet man als Reload.

Unter Umständen kann es passieren, dass der GCC nicht in der Lage ist, ein Register auszulagern. Das passiert zum Beispiel, wenn zum Auslagern ein zusätzliches Register benötigt wird, wie für einen registerindirekten Store.

Alle Spezialregister (Statusregister, Stackpointer, Basepointer) sind von der Registerallokation ausgeschlossen. Diese müssen eigens definiert werden, wie es in Abschnitt 3.3.4 beschrieben wird.

2.3.9 libgcc

Die libgcc ist eine Bibliothek, die statisch in jedes Kompilat eingebaut wird. Sie besteht konzeptionell aus zwei Stufen, libgcc1 und libgcc2. Dabei enthält die libgcc1 alle nicht direkt umsetzbaren Funktionen, die Teil der Sprache C selbst sind. Die libgcc2 ist Bestandteil des GCC selbst, die libgcc1 muss beim Portieren geschrieben werden, falls sie benötigt wird. So muss keine Funktion `_mulsi3` definiert werden, wenn dieser als Instruktion vorliegt und deswegen ein eigenes Instruktionspattern bekommt.

Im Fall des HiCoVecs muss die libgcc1 Routinen für beliebig breite Shifts, Multiplikation, Division und Modulo bereitstellen. Für Division und Modulo liegt eine effiziente, in C geschriebene Implementierung im GCC selbst vor, die hier verwendet wurde.

In der libgcc2 wird spezifischer Code ausgeführt, beispielsweise die Konstruktoren für globale Objekte in C++. Da weder C++ noch Prozesse zur Verfügung stehen, und für diese Funktionalität der gas (GNU Assembler) und gld (GNU Linker) benötigt werden, wurde die libgcc2 nicht näher betrachtet.

3 Implementierung

In diesem Kapitel wird die komplette Umsetzung des Backends auf den HiCoVec beschrieben. Dazu gehören alle Anpassungen am GCC selbst (nur das Buildsystem) sowie die Umsetzung der Funktion, Makros und der Maschinenbeschreibung, mit denen der Prozessor und das gewählte ABI beschrieben werden.

3.1 Anpassung des Buildsystems

Um den GCC überhaupt kompilieren zu können, muss zunächst im Buildsystem ein neues Ziel eingetragen werden. Das Buildsystem der GCC ist ein normales Autoconf-Projekt (vgl. [Fou10]), das aus mehreren Unterprojekten besteht. Die Dateien des Backends selbst befinden sich im Baum des GCC innerhalb des Verzeichnisses `gcc/config/hicovec`. Die Existenz dieses Backends muss dann nur noch in die vorhandenen autoconf-Dateien eingetragen werden, bevor der Build-Prozess funktioniert:

In der Datei `config.sub` musste in der Liste der unterstützten CPU-Typen etwa ab Zeile 300 eingetragen werden:

```
1 case $basic_machine in
2 ...
3         | hicovec \
4 ...
```

Zusätzlich wurde in der Datei `configure.ac` in der Konfiguration über Möglichkeiten des Buildprozesses eingetragen, dass nur die Sprache C unterstützt wird, und keine GCC-internen Bibliotheken mitgebaut werden soll:

```
1 case "${target}" in
2 ...
3   hicovec-*)
4     noconfigdirs="$noconfigdirs target-libffi target-boehm-gc gdb libgloss"
5     unsupported_languages="$unsupported_languages fortran java c++"
6     ;;
7 ...
```

Abschließend musste nur noch die Existenz des Backends selbst eingetragen werden. Dies geschieht in der Datei `gcc/config.gcc`:

```
1 case target in
2   ...
3     hicovec-none)
4           tmake_file="hicovec/t-hicovec"
5   ...
```

Hierbei bedeutet die Zuweisung der Variable `tmake_file`, dass ein zusätzliches Makefile im Backend verwendet wird. Hierin wird die Abhängigkeit beschrieben, dass die Floating-Point-Bibliothek mitgebaut werden soll. Da die Unterstützung hierfür noch nicht abgeschlossen ist, kann dieses Makefile als unvollständig betrachtet werden und wird daher hier nicht aufgeführt.

Um die nötigen Makefiles zu generieren, muss folgender Befehl ausgeführt werden:

```
1   configure hicovec
```

Der Compiler wird dann mit folgendem Kommande kompiliert:

```
1   make
```

3.2 Application Binary Interface (ABI)

3.2.1 Stackaufbau

Da nur 3 Register vorhanden sind, können nicht 2 davon statisch reserviert werden um den Stack- und Framepointer zu definieren. Daher wird die erste Vektorzeile hierfür mitverwendet. Da der HiCoVec indizierte Addressierung aus allen Skalarregistern unterstützt, wird der Framepointer statisch in Y abgelegt. Dieser wird häufiger benötigt als der Stackpointer, da jeder Zugriff auf eine lokale Variable über den Framepointer erfolgt.

Es wäre prinzipiell auch möglich, die meisten Zugriffe über den Stackpointer durchzuführen, allerdings wird in Spezialfällen trotzdem ein Framepointer benötigt. Da das Problem hier die Anzahl der Register und nicht die Anzahl der benötigten Instruktionen ist, wurde dieser Ansatz ebenfalls verworfen.[1]

SP Stackpointer, Register R0(0), zeigt auf die letzte lokal verwendete Addresse

BP Basepointer, Register Y, statisch pro Funktionsaufruf, zeigt auf die Basis des Stacks und wird indiziert verwendet um Variablen zu lesen. Framepointer und Basepointer werden hier synonym verwendet.

Aufgrund der Änderung, auch negative Immediate-Werte zu erlauben, wird der Stack klassisch aufgebaut, wachsend von oben nach unten. Beim Funktionsaufruf werden die Argumente rückwärts auf den Stack gelegt, da dies später bei variadischen Funktionen wichtig ist. Hierdurch befindet sich das erste Argument immer an der gleichen Stelle und kann vom Code erkannt werden. Dies wird unter anderem bei der Standardfunktion `printf` gebraucht. Die

[1] Die MIPS-Portierung verwendet diesen Ansatz und kann zum Vergleich herangezogen werden.

Register zeigt auf	Addresse	Beschreibung
	0x8000	undefiniert
	0x7FFC	RETP main
BP main =>	0x7FF8	BP __start
SP main =>	0x7FF4	Lokale 1 main
	0x7FF0	Argument y add2i
	0x7FEC	Argument x add2i
	0x7FE8	RETP add2i
BP add2i =>	0x7FE4	BP main
SP add2i =>	0x7FE0	Lokale 1 add2i

Abbildung 3.1: Stackaufbau

Register A und X werden bei einem Funktionscall nicht zurückgesetzt, der Callee muss den Basepointer zurücksetzen. Das Aufräumen des Stackpointers übernimmt der Caller.

Hierdurch kann der Compiler die Register A und X für Berechnungen verwenden. Lokale Variablen befinden sich fast immer im Speicher und werden in diese Register kopiert. Der Compiler schreibt das Ergebnis einer Rechnung direkt nach A, sodass die Variable leichter weiterverarbeitet werden kann.

Der hierfür benötigte Prolog und Epilog wird in Abschnitt 3.2.3 genauer erklärt.

3.2.2 Beispiel anhand einer einfachen Funktion

In diesem Beispiel wird eine einfache Funktion *add2i* implementiert, die zwei Zahlen addiert. Der dargestellte Maschinencode ist dabei vollständig vom Compiler generiert.

Dabei zeigt Abbildung 3.1 den Aufbau des Stack nach Aufruf der Funktion add2i aus Abbildung 3.2. Diese Funktion wird in Abbildung 3.3 vom Compiler assembliert gezeigt. RETP steht hierbei für den Rücksprungpointer einer Funktion, die Argumente werden von 1 aufwärts von links nummeriert.

Der Funktionsprolog und -epilog werden in Abschnitt 3.2.3 erklärt.

Hierbei wird in Abbildung 3.3 in den Zeilen 23-25 vom Compiler eine zusätzliche lokale Variable angelegt, die den Wert zwischendurch enthält. Da deren Lebzeit innerhalb der Funktion bleibt, wird der Stackpointer nicht verändert.

In Abbildung 3.4 wird zunächst die Funktion *__main* aufgerufen. Diese wird normalerweise von der libgcc2 zur Verfügung gestellt und enthält generierte Initialisierungsfunktionen, beispielsweise für das Initialisieren eines nicht konstanten Strings. Da derzeit die libgcc2 nicht unterstützt wird, wurde eine Dummy-Funktion verwendet.

3.2.3 Funktionsprolog und Funktionsepilog

Jede Funktion benötigt einen Prolog, in dem das Stackframe angelegt wird, sowie einen Epilog, in dem dieses wieder abgebaut wird.

```
1       int add2i(int x, int y){
2         return x + y;
3       }
4       int main(void){
5         return add2i(3, 5);
6       }
```

Abbildung 3.2: Beispiel für Funktionsaufbau - C

Der Prolog befindet sich in Abbildung 3.5. Er wird dabei in leicht veränderter Form in jede Funktion statisch eingebaut. Zum Zeitpunkt, an dem der Prolog angesprungen wird, befindet sich die Rücksprungaddresse im Register A. Das Register X ist unbenutzt, das Register Y enthält noch den alten Basepointer. Der Stackpointer ist vom Aufrufenden so angepasst worden, dass er auf das zuletzt hinzugefügte Element zeigt.

Zunächst wird in Zeile 1 der Stackpointer nach X geladen und in Zeile 2 die Rücksprungaddresse gesichert. Dann wird der alte Basepointer in den Zeilen 3 und 4 ebenfalls im Stack abgelegt. Der Basepointer wird dann in Zeile 5 auf die neue Basisaddresse gesetzt, 8 Byte unter dem Stackende, also auf das erste Element unterhalb der gespeicherten Werte.

Dann wird in Zeile 6 Platz für alle lokalen Variablen angelegt. Diese Größe wird dabei vom GCC statisch einkompiliert. Sie enthält auf jeden Fall 8 Byte für die Rücksprungaddresse und den alten Basepointer, sowie zusätzlich 4 Byte für jede lokale Variable.

Dieser geänderte Stackpointer wird dann nur noch in Zeile 7 zurückgeschrieben, bevor der Rest der Funktion abgearbeitet wird.

Der zugehörige Epilog befindet sich in Abbildung 3.6. Da dieser keine funktionsspezifischen Teile hat, ist er als Subroutine implementiert, die am Ende jeder Funktion angesprungen wird.

Sobald der Epilog betreten wird, befindet sich in Register A der Rückgabewert der Funktion. Register X ist unbenutzt, Register Y enthält noch den Basepointer der zurückkehrenden Funktion. Zunächst wird in Zeile 1 und 2 der Stackpointer auf den alten Wert zurückgesetzt, indem 8 auf den Basepointer addiert werden. Dann wird in Zeile 3 im Register X temporär die Rücksprungaddresse abgelegt. Anschließend wird in Zeile 4 der alte Basepointer wieder geladen und in Zeile 5 die Rücksprungaddresse angesprungen. Der ursprünglich Aufrufende muss abschließend noch den Stack aufräumen, also den Platz wieder freigeben, an dem die Argumente der aufgerufenen Funktion liegen. Hierfür generiert der GCC selbstständig Code, sodass dieser Teil nicht beachtet werden muss.

```
1  .org 0x00
2          .start
3  __start:
4          or X, 0, 0x4000    ;stack starts at 0x8000
5          add X, X, X        ;stack start at 0x8000
6          or Y, X, X         ;base = top
7          mov r0(0), X       ;store sp
8          jal A, [0 + main]  ;call main
9          halt               ;main returned. stop
10
11 __fun_epilogue:            ;always the same, can be a subroutine
12         add X, Y, 8        ;set sp = bp + 8
13         mov r0(0), X       ;store it
14         ld X, [Y + 4]      ;retp => X
15         ld Y, [0 + Y]      ;old bp => bp
16         jmp [0 + X]        ;jmp retp
17
18 __main:
19         jmp [0 + A]
20
21 add2i:
22         mov X, r0(0)
23         st [X + -4], A     ;store retp
24         or A, Y, Y
25         st [X + -8], A     ;store old bp
26         sub Y, X, 8        ;bp = sp
27         sub X, X, 12       ;8 for retp and old bp, 4 allocated for local vars
28         mov r0(0), X       ;write sp back
29
30         ld X, [Y + 8]      ;get arg 1
31         or A, X, X
32         st [Y + -8], A     ;store in stack
33         ld A, [Y + 12]     ;load arg 2
34         ld X, [Y + -8]     ;load arg 1
35         add X, X, A        ;add it
36         or A, X, X         ;store as return value
37
38         jmp [0 + __fun_epilogue]
39
40         .end
```

Abbildung 3.3: Beispiel für Funktionsaufbau - Assemblercode - Teil 1

```
1  main:
2          mov X, r0(0)
3          st [X + -4], A  ;store retp
4          or A, Y, Y
5          st [X + -8], A  ;store old bp
6          sub Y, X, 8     ;bp = sp
7          sub X, X, 12    ;8 for retp and old bp, 4 allocated for local vars
8          mov r0(0), X    ;write sp back
9
10         jal A, [0 + __main]     ;call init code
11         mov A, r0(0)            ;push arguments for call
12         add A, A, -8            ;adjust sp => -4
13         mov r0(0), A            ;store it
14         mov X, r0(0)
15         or A, X, X
16         st [Y + -8], A          ;store old SP in stack
17         or A, 0, 3              ;load 3
18         ld X, [Y + -8]          ;load SP
19         st [0 + X], A           ;push 3
20         or A, 0, 5              ;load 5
21         ld X, [Y + -8]
22         st [X + 4], A           ;push it
23         jal A, [0 + add2i]      ;call add2i
24         mov X, r0(0)            ;get old SP back
25         add X, X, 8
26         mov r0(0), X            ;done
27
28         jmp [0 + __fun_epilogue]
```

Abbildung 3.4: Beispiel für Funktionsaufbau - Assemblercode - Teil 2

```
1          mov X, r0(0)
2          st [X + -4], A  ;store retp
3          or A, Y, Y
4          st [X + -8], A  ;store old bp
5          sub Y, X, 8     ;bp = sp
6          sub X, X, 32    ;8 for retp and old bp, 24 allocated for local vars
7          mov r0(0), X    ;write sp back
```

Abbildung 3.5: Funktionsprolog

```
1          add X, Y, 8
2          mov r0(0), X
3          ld X, [Y + 4]    ;pop retp
4          ld Y, [0 + Y]    ;pop ebp
5          jmp [0 + X]
```

Abbildung 3.6: Funktionsepilog

```
1  #define STACK_GROWS_DOWNWARD 1
2  #define FRAME_GROWS_DOWNWARD 1
3  #undef ARGS_GROW_DOWNWARD       /* Our args grow upwards.
4                                     This ain't Kansas anymore */
5  #define STACK_POINTER_OFFSET 0
6  #define STARTING_FRAME_OFFSET -4 /* 0 is part of our frame.
7                                      First Var is at -4 */
8  #define FIRST_PARM_OFFSET(FNDECL)  8 /* First parameter is at +8 from BP */
9  #define FRAME_POINTER_REQUIRED 1 /* Don't try to optimize it away */
10 #define STACK_POINTER_REGNUM     SP_REG_NUMBER
11 #define FRAME_POINTER_REGNUM     BP_REG_NUMBER /* Use BP for locals */
12 #define ARG_POINTER_REGNUM       BP_REG_NUMBER /* Use BP for Arguments */
13 #define PUSH_ARGS 0              /* We don't have push. don't try to use it */
```

Abbildung 3.7: Stackaufbau - Makros

3.3 Entwurf der Beschreibungsmakros (hicovec.h und hicovec.c)

3.3.1 Stackaufbau und Calling Convention

Hier werden die für die in Abschnitt 3.2 beschriebene ABI benötigten Makros aufgelistet. Sie sind dabei eine direkte Umsetzung des Entwurfs und enthalten die passenden Offsets in den Stack. Sie befinden sich in Abbildung 3.7.

3.3.2 Datentypen

Hier werden die verwendeten Datentypen beschrieben. Der Code dafür befindet sich in der Datei `hicovec.h` im Abschnitt "LAYOUT OF SOURCE LANGUAGE DATA TYPES".

Alle ganzzahligen Datentypen werden als 32Bit-Worte behandelt. Dies vereinfacht die Machine Description, da größtenteils nur der SIMode behandelt werden muss (vgl. 2.3.4). Dies betrifft auch Pointer, die somit in jeder ganzzahligen Variablen untergebracht werden können.

Da in der aktuellen Befehlskodierung keine Modi vorgesehen sind um Transfers unterhalb der Wortbreite vorzunehmen, bleiben die MOV-Patterns für QI und HI (8 bzw 16 Bit) undefiniert.

Hierbei tritt unerwartetes Verhalten auf, wenn sich der C-Code auf das Überlaufverhalten von `char` oder `short`-Variablen verlässt. Da diese immer 32 Bit lang sind, laufen diese auch erst bei den vollen 32 Bit über. Code der dieses Verhalten benötigt muss danach explizit mit dem benötigten Bereich verUNDen.

Falls Code den Datentyp `long long` verwendet, kann dies ebenfalls zu unerwartetem Verhalten führen, da dieser ebenfalls nur aus 32Bit besteht.

Floats (`single` und `double`) können prinzipiell verwendet werden, allerdings können diese noch nicht als Konstanten ausgegeben werden, da noch keine Printfunktion für diese implementiert ist. Zusätzlich fehlt die Soft-Float-Bibliothek aus dem GCC, für die auf jeden Fall noch 8- und 64-Bit Transfers benötigt werden.

3.3.3 Addressierungsmodi

Hier werden die Einträge aus der Datei `hicovec.h` genauer erklärt, die sich die möglichen Addressierungsmodi beschreiben. Diese befinden sich in der Datei im Abschnitt “ADDRESSING MODES”.

Zunächst wird eingestellt, dass jede valide konstante Integer auch eine valide Addresse ist. Zusätzlich wird konfiguriert, dass keine Form von Prä- oder Postinkrement in der Addressierung möglich ist. Prä- und Postinkrement bedeutet, dass bei einer Addressierung die verwendete Addresse gleichzeitig mitverändert werden kann. Prä bedeutet vor, Post nach dem Speicherzugriff. Inkrement bedeutet eine Erhöhung, Dekrement eine Verminderung der Addresse. Dies ist beim HiCoVec durchgehend nicht möglich.

```
1 #define CONSTANT_ADDRESS_P(X) CONSTANT_P(X)
2 #define HAVE_POST_INCREMENT 0
3 #define HAVE_POST_DECREMENT 0
4 #define HAVE_POST_MODIFY_DISP 0
5 #define HAVE_POST_MODIFY_REG 0
```

Für den weiteren Code ist es wichtig, das Makro `REG_OK_STRICT` zu verstehen. Dieses wird vom GCC selbst gesetzt, bevor die TARGET.h lokal inkludiert wird. Falls das Flag gesetzt ist (`ifdef REG_OK_STRICT`), muss jedes in einer Addresse verwendete Register geprüft werden, ob es ein valides Register ist. Falls nicht, reicht die Feststellung, ob es überhaupt ein Register ist.

Dies ist wichtig, da in den ersten Läufen die Registerallokation noch nicht stattgefunden hat und der Compiler noch dabei ist, RTL zu generieren, um herauszufinden, ob der Code hier überhaupt umgesetzt werden kann. Später wird der Addressvalidierungscode benutzt um festzustellen, ob ein weiterer Registerallokationsdurchlauf durchgeführt werden muss.

Hiermit werden zunächst die Register überprüft, ob sie ein valides Basis- oder Indexregister darstellen. Dies ist einfach:

```
1 #define REGNO_OK_FOR_BASE_P(REGNO) ((REGNO) < 3)
2 #define REGNO_OK_FOR_INDEX_P(REGNO) REGNO_OK_FOR_BASE_P(REGNO)
3
4 #ifdef REG_OK_STRICT
5 #define REG_OK_FOR_BASE_P(X) REGNO_OK_FOR_BASE_P (REGNO (X))
```

```
6  #define REG_OK_FOR_INDEX_P(X)     REGNO_OK_FOR_INDEX_P (REGNO (X))
7  #else
8  #define REG_OK_FOR_BASE_P(X)      1
9  #define REG_OK_FOR_INDEX_P(X)     1
10 #endif
```

Interessanter ist hierbei das Makro:

```
1    GO_IF_LEGITIMATE_ADDRESS(MODE, X, LABEL)
```

Dieses Makro soll auswerten, ob das RTL-Konstrukt X im Modus MODE eine valide Addresse darstellt. Wenn ja, soll ein `goto LABEL` ausgeführt werden. Diese Auswertung ist komplex, sodass hier in einen Funktionsaufruf in der Datei `hicovec.c` expandiert wird:

```
1  #define GO_IF_LEGITIMATE_ADDRESS(MODE, X, LABEL)                \
2  {                                                               \
3    if (hicovec_legitimate_address_p (MODE, X, 1))
4        goto LABEL;                                               \
5  }
```

Diese Funktion muss hier voll wiedergegeben werden, da ihr Verständnis essentiell für das Verständnis des Kompiliervorgangs des GCC ist. Sie befindet sich in der Abbildung 3.8.

Eine Addresse kann hierbei absolut (integer, label, symbol), indirekt (Register) oder relativ (Register + Register, Register + Integer) sein. Gleichzeitig muss geprüft werden, ob die Inhalte der Addresse valide sind, falls der Parameter `strict` gesetzt ist.

Hierfür werden 3 Flags eingesetzt - `base_valid`, `index_valid` und `all_valid`. Dabei wird davon ausgegangen, dass ein Label immer aufgelöst werden kann. Dies ist später nicht mehr der Fall, wenn der Code zu groß wird, gilt aber für jetzt.

Im switch von Zeile 8-35 wird dabei die Addresse zerlegt. Falls eine Addresse nur eine Konstante ist, kann direkt 1 zurückgegeben werden. Wenn die Addresse ein Register ist, wird es etwas interessanter. Es muss geprüft werden, ob es ein valides Register (A, X, Y) ist, allerdings nur, wenn `strict` gesetzt ist. Diese Ueberprüfung findet in den Zeilen 36-45 statt.

Für den Fall, dass die Addresse relativ ist, muss geprüft werden, ob die einzelnen Teile valide sind. Hierbei darf die Basis nur ein Register oder die Zahl `0` sein, der Index ein Register oder eine 16Bit Konstante. Diese Überprüfung findet in den Zeilen 28-31 statt.

3.3.4 Definition der Register

Für die Verteilung der Register mussten dem GCC zunächst die Anzahl und Klassen der Register mitgeteilt werden. Hierfür wurden 4 Register angenommen: A, X, Y und SP. SP ist dabei der Stackpointer, Y der Basepointer. SP befindet sich in r0(0).

Damit haben Y und SP Sonderstellung und sind wie folgt zu behandeln:

```
1  #define FIRST_PSEUDO_REGISTER 4
2  #define FIXED_REGISTERS \
3    {0, 0, 1, 1}
4  #define CALL_USED_REGISTERS \
5    {1, 1, 1, 1}
```

```
1   /* Returns 1 if the address can be
2   understood. Includes things like [A + -20] */
3  int hicovec_legitimate_address_p (int unused, rtx addr, int strict) {
4   rtx base = NULL_RTX, index = NULL_RTX;
5   int base_valid = 0;
6   int index_valid = 0;
7   int all_valid = 0;
8   switch (GET_CODE (addr)) {
9     case CONST_INT:
10        all_valid = 1;
11      break;
12    case SYMBOL_REF: case LABEL_REF:
13      all_valid = 1;
14      break;
15 /* Use the Register as index as the assembler might be able to squeeze a vecto
16    case REG:
17      index = addr;
18      base = const0_rtx;
19      base_valid = 1;
20      index_valid = 1;
21      break;
22    case PLUS:
23      base = XEXP(addr, 0);
24      index = XEXP(addr, 1);
25      if (REG_P(base) || (GET_CODE(base) == CONST_INT && INTVAL(base) == 0))
26        base_valid = 1;
27      if (REG_P(index) || (GET_CODE(index) == CONST_INT))
28        index_valid = 1;
29      break;
30    default:
31      return FALSE;    }
32   if (strict && base_valid && index_valid) {
33      if (!CONSTANT_ADDRESS_P (base))
34        if(REG_P(base))
35          if (!REGNO_OK_FOR_BASE_P (REGNO (base)))
36            return FALSE;
37      if (!CONSTANT_ADDRESS_P (index))
38        if(REG_P(index))
39          if (!REGNO_OK_FOR_INDEX_P (REGNO (index)))
40            return FALSE;    }
41   if (all_valid || (base_valid && index_valid))
42     return TRUE;
43  return FALSE;
44 }
```

Abbildung 3.8: Legitimate Address - C-Code

Als Registerklassen wurden zusätzlich die Klassen `ARITH_REGS` und `SP_REGS` eingeführt:

```
1 enum reg_class
2 {
3   NO_REGS,
4   GENERAL_REGS,
5   ARITH_REGS,
6   SP_REGS,
7   ALL_REGS,
8   LIM_REG_CLASSES
9 };
```

Dabei enthält `GENERAL_REGS` nur das Register A, `ARITH_REGS` die Register A, X und Y, `SP_REGS` das Register SP. Gleichzeitig besteht folgende Zuordnung zu den Registernummern:

Register	Nummer
A	0
X	1
Y	2
SP	3

Diese Einteilung wird vom GCC als Bitmaske benötigt, um zu einer gegebenen Registernummer schnell die Klasse finden zu können:

```
1 #define REG_CLASS_CONTENTS                                  \
2   {                                                         \
3     {0x00000000}, /* NO_REGS                        */      \
4     {0x00000001}, /* GENERAL_REGS                   */      \
5     {0x00000007}, /* ARITH_REGS                     */      \
6     {0x00000008}, /* SP_REGS                        */      \
7     {0x0000000F}  /* ALL_REGS                       */      \
8   }
```

Da nur das Register A in den Speicher schreiben kann, muss dem GCC mitgeteilt werden, wie man aus einem anderen Register rausschreiben kann. Hierfür besteht noch die Funktion `hicovec_secondary_reload` in der Datei `hicovec.c`. Diese wird vor jedem Speicherzugriff aufgerufen um dem GCC zu signalisieren, wenn ein zusätzliches Register gebraucht wird. Sie befindet sich noch in Abbildung 3.9.

Hier wird nur geprüft, ob der Zugriff ein Speicherzugriff ist, das Register nicht A, und der Zugriff schreibend erfolgt. Falls dies alles zutrifft wird ein weiteres Allzweckregister gebraucht, was hier nur A ist. Der GCC spillt dann das Register A, um es für diesen Zugriff verwenden zu können.

Zusätzlich wurde das Makro `SMALL_REGISTER_CLASSES` definiert, das dem GCC mitteilt, dass sehr wenige Register vorhanden sind. In diesem Fall wird der Codegenerator versuchen, schon in der RTL-Generation die Lebenszeit harter Register zu minimieren. Dadurch werden in der Regel weniger Fehler produziert, allerdings konnte dies mit den getesteten Programmen nicht nachvollzogen werden.

```
1  /* Returns the additional needed register class (if any) for a mov */
2  enum reg_class hicovec_secondary_reload (bool in_p, rtx x,
3  enum reg_class reload_class, enum machine_mode mode,
4  secondary_reload_info *SRI)
5  {
6    if (MEM_P (x) && (reload_class != GENERAL_REGS) && ! in_p )
7      return GENERAL_REGS;
8      /* For supporting Floating Point, bigger constants than
9      16 bit are needed. Support might be added here */
10   return NO_REGS;
11 }
```

Abbildung 3.9: Secondary Reload

Die Programme, die vorher nicht mit aktiviertem Optimizer gebaut werden konnten, konnten dies auch danach nicht – Die anderen Programme liefen langsamer, da mehr Reloads benötigt werden.

Das Makro bleibt trotzdem aktiv, da es eigentlich benötigt wird und eventuell später als Basis für eine sauberere Implementierung verwendet werden kann.

3.3.5 Definition der Condition Codes

Die Condition Codes sind in der impliziten Variante kodiert worden (vgl. 2.3.6). Hierfür wurde nur die Funktion `hicovec_notice_update_cc` definiert, die die eigentliche Logik enthält, und die im Makro `NOTICE_UPDATE_CC` aufgerufen wird. Diese Funktion befindet sich in Abbildung 3.10.

Sinngemäß wird dieses Makro vom GCC für jede Instruktion aufgerufen. Sie soll als Seiteneffekt in der globalen Variablen `cc_status` die verwendbaren Flags und deren Abhängigkeiten angeben. Diese Abhängigkeiten beziehen sich dabei auf Werte direkt aus dem Programm.

Hierbei bekommt jeder Befehl ein Attribut, das beschreibt, wie dieser die Flags verändert. Die Funktion liest diese nur noch aus und setzt den Status entsprechend. Die Attribute sind:

none Keine Auswirkungen, die Flags sind genau wie vorher (Sprünge, Store)

set_z Zero verwendbar (Bit-Operationen, Load)

set_cz Carry und Zero verwendbar (Add, Sub, Shift)

compare Alles verwendbar (Vergleich)

clobber Werte unbrauchbar

Wenn der GCC einen bedingten Sprung generieren soll versucht er dabei, diesen in Abhängigkeit von einer schon vorhandenen Instruktion zu setzen. Das bedeutet, dass zum Beispiel

bei einer Schleife die gegen `0` läuft die Flags der Subtraktion selbst schon verwendet werden können. Nur wenn dies nicht möglich ist, generiert er einen neuen Vergleich. Der GCC ist hierbei selbstständig in der Lage, die globale Variable `cc_status` zu kopieren und zurückzusetzen, wenn dies nötig ist.

3.4 Machine Description

Bei der Implementierung der Machine Description wurden zunächst die Standardnamen aus der GCC-Dokumentation durchgearbeitet und möglichst viele davon definiert. Hierbei waren die arithmetischen Befehle verhältnismäßig trivial, mit Ausnahme der Shifts, da die Standardpatterns hier einen variabel breiten Shift erwarten.

Als Implementierung für `cmpsi` wurde direkt ein `sub` ohne Zielregister definiert, wodurch die Flags korrekt gesetzt werden.

Die Implementierung der Sprünge erwies sich ebenfalls als relativ schwierig, da der HiCoVec nur die Flags Zero und Carry besitzt. Damit lassen sich die Relationen $\leq$ und $>$ nicht direkt ausdrücken. In vielen Fällen kann der Compiler hierbei einfach den Vergleich umdrehen und den Sprung durch einen der vorhandenen ersetzen. Dies geht jedoch nicht, wenn ein Wert mit einer Konstanten verglichen wird, da diese nicht als Operand 2 in einem SUB kodiert werden kann.

Als besonders problematisch haben sich die Move-Patterns herausgestellt. Zwischen den Skalarregistern sind diese leicht, allerdings war es schwer, die Verwendung der Vektorregister als Skalarregister in RTL zu formulieren. Hierfür wurde eine zusätzliche Registerklasse eingeführt, die spezielle MOV-Patterns bekommen und sonst wie normale Register behandelt werden. Da sie nicht als Allzweckregister markiert sind, werden sie auch nicht vom Registerallokator mitverwendet.

Hierbei gab es wiederum ein Problem, dass das Pattern für einen Reload (Zwischenspeichern eines Registerwerts auf dem Stack, Durchführen einer Operation, Wiederherstellen des Registers) nicht terminierte und der xgcc deswegen in eine endlose Rekursion lief, in der er ständig versuchte, ein Register mit sich selbst neu zu laden.

3.4.1 Beispiele für benötigte Patterns

In diesem Abschnitt werden Beispiele für die zu definierenden Patterns aufgeführt. Als Konvention wird hier in der generierten Assemblercode noch in Kommentaren eine andere Syntax für die Assemblercode dargestellt. Dies war hilfreich beim Debuggen und wurde deswegen im Code belassen.

Arithmetische Befehle

In Beispiel 3.11 wird der Additionsbefehl des HiCoVec implementiert, indem eine neue Instruktion namens “addsi3” definiert wird.

Hierbei steht SI immer für Single Integer, also im Fall des HiCoVec einer 32bit Signed Integer. Die 3 am Ende der arithmetischen Befehle steht für 3 erwartete Argumente. Der Name selbst ist zumindest bei den arithmetischen Befehlen selbsterklärend.

```
1  void hicovec_notice_update_cc (rtx exp, rtx insn)
2  {
3    rtx set;
4    switch (get_attr_cc (insn)){
5    case CC_NONE:
6      break;
7    case CC_SET_Z:
8        set = single_set (insn);
9        CC_STATUS_INIT;
10       if (set)
11         {
12           cc_status.flags |= CC_NO_OVERFLOW;
13           cc_status.value1 = SET_DEST (set);
14         }
15     break;
16   case CC_SET_CZ:
17       set = single_set (insn);
18       CC_STATUS_INIT;
19       if (set)
20         {
21           cc_status.value1 = SET_DEST (set);
22         }
23       break;
24     break;
25   case CC_COMPARE:
26       set = single_set (insn);
27       CC_STATUS_INIT;
28       if (set)
29         cc_status.value1 = SET_SRC (set);
30       break;
31     break;
32   case CC_CLOBBER:
33     CC_STATUS_INIT;
34     break;
35   }
36 }
```

Abbildung 3.10: Condition Code Update

```
1       (define_insn "addsi3"
2         [(set (match_operand:SI 0 "register_operand" "=r")
3               (plus:SI
4                  (match_operand:SI 1 "hicovec_reg_or_0" "r0")
5                  (match_operand:SI 2 "nonmemory_operand" "ri")))]
6        ""
7        "add %0, %1, %2")
```

Abbildung 3.11: Beispiel: Additionsbefehl

a	Arithmetisches Register (A, X, Y)
i	Immediate Zahl (signed 16Bit)
m	Speicher (beliebig addressiert, also auch indirekt)
r	General Purpose Register (A)
R	Stackpointer (r0(0))

Abbildung 3.12: Constraints - Machine Description

Die Zeile Assemblercode wird dabei direkt in die Zieldatei ausgegeben. Es werden nur noch Ersetzungen der mit % markierten Operanden vorgenommen.

MOV-Befehle

Im Beispiel 3.13 befindet sich der komplette Code, der derzeit für MOV-Patterns verwendet wird. Im Pattern wird der Iterator SIF benutzt, der in Zeile 1 des Beispiels definiert wird.

In Zeile 2 wird der mode als String in den Patternnamen eingefügt. Das komplette Pattern wird dabei zwei mal durchlaufen, einmal für den Mode SI, einmal für SF. Dadurch wird definiert, dass der Code sowohl für 32-Bit Integer als auch für 32-Bit Floats verwendet werden soll. Der endgültige Name ist dabei movsi bzw. movsf.

Das zutreffende Pattern ist die Zuweisung (set “nonimmediate_operand” “general_operand”). Das bedeutet sinngemäß, dass jeder zuweisbare Platz sich von überall zuweisen lässt. Die Constraints dahinter werden in Abbildung 3.12 erklärt.

Dabei muss dieser Code so gelesen werden, dass jede Spalte in den Operanden (durch Komma getrennt) einer Zeile im generierten Code entspricht. Die folgenden Beispiele sind dabei so zu lesen, dass jeweils eine Zeile in der Fallunterscheidung einer Spalte in den Operanden entspricht. Wegen besserer Lesbarkeit wurde das “=”-Zeichen zwischen die beiden Constraints geschrieben. In Reihenfolge sind dies:

a = i Ein Register wird aus einem Immediate-Wert geladen. In HiCoVec-Assemblercode lässt sich dies durch Verodern der Konstante mit 0 umsetzen.

a = a Zuweisung eines Registers aus einem anderen Register. Wurde hier durch Veroderung des Registers mit sich selbst umgesetzt, es kann aber auch eine Verundung mit sich selbst oder

```
1  (define_mode_iterator SIF [SI SF])
2  (define_insn "mov<mode>"
3    [(set (match_operand:SIF 0 "nonimmediate_operand" "=a, a, a, m, a, R")
4          (match_operand:SIF 1 "general_operand" "i, a, m, r, R, a"))]
5    ""
6    "@
7          or %0, 0, %1     ; a = i
8          or %0, %1, %1    ; a = a
9          ld %0, %a1       ; a = m
10         st %a0, %1       ; m = r
11         mov %0, r0(0)    ; a = R
12         mov r0(0), %1    ; R = a
13   [(set_attr "cc" "set_z, set_z, clobber, none, none, none")])
```

Abbildung 3.13: Beispiel: MOV-Pattern

Veroderung mit `0` verwendet werden.

a = m Zuweisung aus dem Speicher. Da im HiCoVec ein LD in jedes Register stattfinden kann wurde hier das constraint a verwendet.

m = r Zuweisung in den Speicher. Ein ST kann nur aus A durchgeführt werden, daher zählt dieses als einziges General Purpose Register.

a = R Kopieren des Stackpointers in ein Register. Wird zusammen mit R = a für den Funktionsaufruf gebraucht.

R = a Zuweisung eines Registers in den Stackpointer.

Wenn das MOV-Pattern noch komplexer wird ist es sinnvoll, hier nur ein leeres `define_expand` zu verwenden und die Fälle dann getrennt zu behandeln, wie es bei den Sprungbefehlen schon der Fall ist. Dann kann auch mit präziseren Constraints gearbeitet werden.

Sprungbefehle

In Abbildung 3.14 ist repräsentativ für alle Branches der bge (Branch Greater Equal) aufgeführt. Die anderen Patterns sehen genauso aus, nur mit jeweils einem anderen Vergleich.

Dieser Expand wird dabei nur als Zusicherung für den Compiler verwendet, dass das Pattern bge verwendet werden kann und daher in der RTL-Generierung verwendet werden darf.

Die Expansion in Maschinencode findet später mit dem Code in Abbildung 3.16 statt. Hier ist zu beachten, dass die generierte Assemblercode von der eigenen C-Funktion `ret_cond_branch` generiert wird, die die nötigen Umbauten durchführt.

```
1 (define_expand "bge"
2   [(set (pc)
3         (if_then_else (ge (cc0) (const_int 0))
4                       (label_ref (match_operand 0 "" ""))
5                       (pc)))
6 ]
7   ""
8   "")
```

Abbildung 3.14: Beispiel: Branchpattern

Dieser Code befindet sich in Abbildung 3.17. Die C-Funktion soll dabei einen C-String zurückliefern, der den vollständigen Assemblercode für dieses Pattern enthält. Der generierte Code ist dabei in den meisten Fällen einfach, mit Ausnahme von GT (Greater than). Sinngemäß muss hier überprüft werden, dass Carry gesetzt ist und Zero nicht. An dieser Stelle können keine Labels generiert werden, was die Sprünge komplizierter macht.[2]

Zunächst wird die aktuelle Position im Programm in Zeile 18 in ein Register geladen, das vorher kopiert wurde um wieder hergestellt werden zu können. Dann wird relativ dazu ein JC vorwärts im Code durchgeführt, dessen Ziel effektiv ein JZ ist, der aus dem Block rausspringt. Falls Carry nicht gesetzt ist, oder Zero gesetzt ist, wird der Block verlassen und das Register A wieder hergestellt.

Im Then-Fall läuft der Code weiter auf das Neuladen des Registers A mit dem ursprünglichen Wert, gefolgt von einem unbedingten Sprung an das Ziel des Branchpatterns. Da dies sowohl im Text als auch im Code schwer nachzuvollziehen ist, befindet sich in der Abbildung 3.18 noch eine Visualisierung der Branches als Graph.

An diesem Graphen können die Zustände abstrakt nachvollzogen werden. Im Graphen stellt “fallthrough” den else-Fall dar, “target” den then-Fall.

Zunächst wird unterschieden, welcher der Vergleichsfälle hier geprüft werden soll. Diese nötigen Kombinationen der Flags können der Abbildung 3.15 entnommen werden. Aus der Tabelle ist klar ersichtlich, dass für alle Fälle ausser $>$ und $<=$ ein einzelner Sprungbefehl verwendet werden kann.

Zunächst wird der Fall $<=$ untersucht. Dieser ist einfach darzustellen, da er nur eine ODER-Verknüpfung zweier Sprünge ist. Daher können diese beiden Sprünge direkt hintereinander geschrieben werden, mit dem Then-Fall als Ziel. Dies ist möglich, da ein Sprungbefehl die Flags nicht verändert.

Schwieriger ist der Fall $>$. Hier muss geprüft werden, dass die Bedingungen $>=$ und $!=$ gleichzeitig gelten, also eine UND-Verknüpfung. Dies wird realisiert, indem der erste Sprung im Erfolgsfall auf den zweiten Sprung springt (jc $\rightarrow$ jz), und dieser dann ans Ziel. Da derzeit kein relativer Sprung möglich ist und hier zur Zeit der Programmentwicklung die Funktion `gen_label_rtx` noch nicht bekannt war, findet dieser Sprung registerindirekt statt.

[2] Hierfür kann die Funktion `gen_label_rtx` verwendet werden. Diese Information wurde zu spät in der Diplomarbeit gefunden um verwendet werden zu können.

Fall	Zustand
==	Z
!=	NZ
>=	C
>	C && NZ
<	NC
<=	Z || NC

Abbildung 3.15: Tabelle der Flags für Branches

```
1  (define_insn "branch"
2    [(set (pc)
3          (if_then_else (match_operator 1 "comparison_operator"
4                          [(cc0)
5                           (const_int 0)])
6                        (label_ref (match_operand 0 "" ""))
7                        (pc)))
8
9     ]
10    ""
11    "*
12     return ret_cond_branch (operands[1], 0);"
13    [(set_attr "cc" "clobber")])
```

Abbildung 3.16: Beispiel: Branch INSN

Hierfür wird zu Beginn der Fallunterscheidung das Register A in eine globale Variable aus der libgcc gespeichert, sodass A als Basis des relativen Sprungs verwendet werden kann. Das Register muss natürlich innerhalb der Fallunterscheidung wieder geladen werden. Dies findet am Ende beider Fälle statt. Es ist nicht möglich, diesen Reload vorzuziehen, da der Load-Befehl selbst wieder die Flags ändert. Nach dem else-Reload läuft der Code ganz normal weiter, nach dem then-Reload findet direkt danach ein Sprung auf target statt, wodurch die Unterscheidung abgeschlossen ist.

3.5 Implementierung der libgcc

Um mit dem Compiler vernünftig arbeiten zu können, wird eine libgcc benötigt, die alle nicht direkt verfügbaren Befehle implementiert.

Die hier benötigte Liste ist:

mulsi3 Multiplikation 32*32=>32Bit

ashlsi3 Arithmetischer Shift nach links

```
1  const char * ret_cond_branch (rtx x, int reverse){
2    RTX_CODE cond = reverse ? reverse_condition (GET_CODE (x)) : GET_CODE (x);
3    switch (cond){
4    case GE:
5    case GEU:
6        return "jc %a0";
7    case LT:
8    case LTU:
9        return "jnc %a0";
10   case LE:
11   case LEU:
12     return "jz %a0\n\tjnc %a0"; /* Check if equal, then Less Than */
13   case GT:
14   case GTU:                     /* Look at the picture. It's complicated.
15                                    Basically the same pattern as the AVR,
16                                    but with faked relative branches*/
17     return "st [0 + __retp_libc], A\n"
18       "\tjal A, [0 + __ret_pc]\n"
19       "\tjc [0 + __skip_2]\n"
20       "\tjal A, [0 + __skip_3]\n"
21       "\tjz [0 + __skip_5]\n"
22       "\tld A, [0 + __retp_libc]\n"
23       "\tjmp %a0\n"
24       "\tld A, [0 + __retp_libc]\n";
25   default:
26     sorry("Branch Pattern broken");
27   }
28 }
```

Abbildung 3.17: Beispiel: Branch C-Code

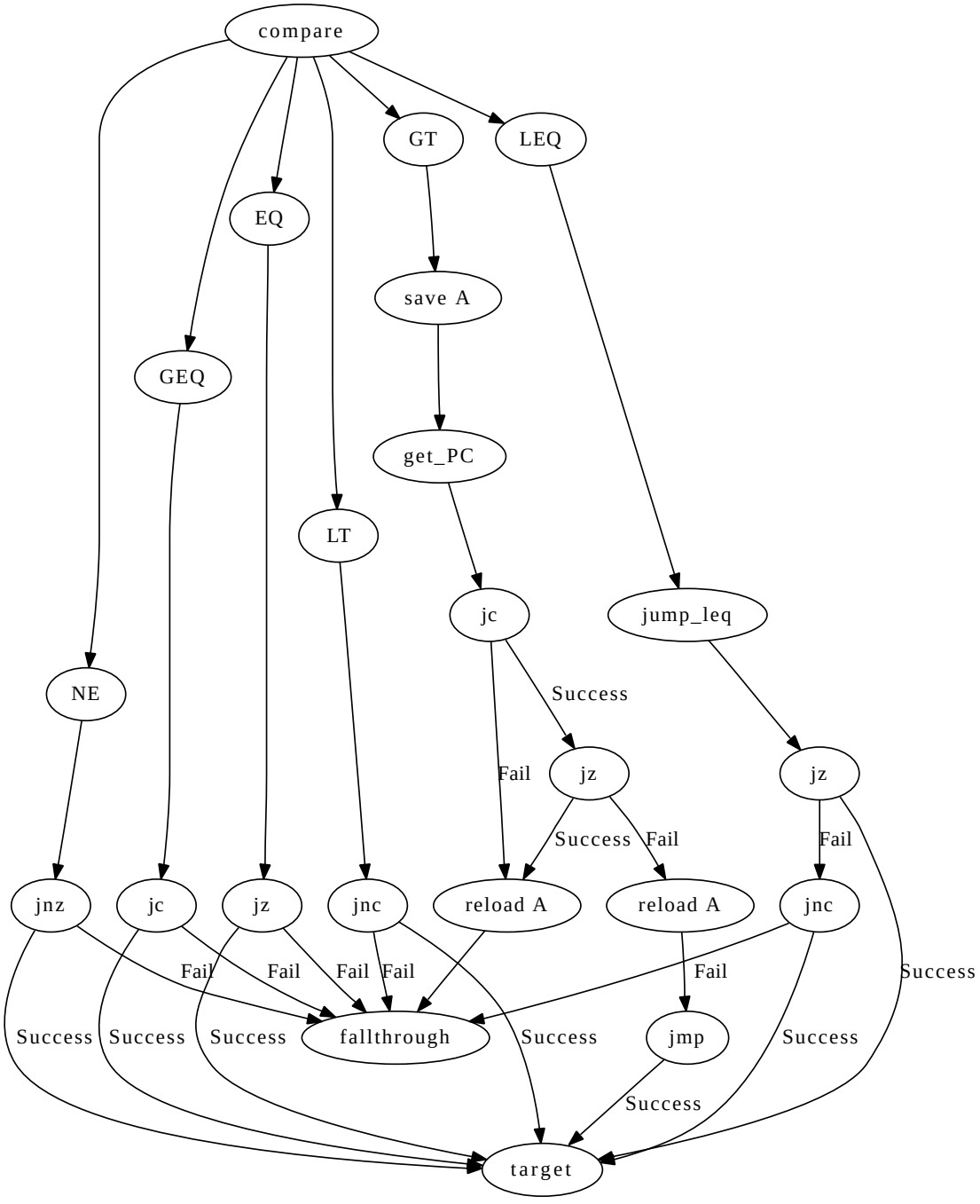


Abbildung 3.18: Graph für Branches

ashrsi3 Arithmetischer Shift nach rechts

lshlsi3 Logischer Shift nach links

lshrsi3 Logischer Shift nach rechts

divsi3 Signed Division

udivsi3 Unsigned Division

modsi3 Signed Modulo

umodsi3 Unsigned Modulo

Hierbei wurde für die Divisions- und Modulo-Funktionen die Datei `udivmodsi.c` aus dem GCC verwendet, die mit dem erstellten C-Compiler kompiliert wurde. Sie wurde dabei einmalig kompiliert, und die Labels im Kompilat von Hand ausgetauscht. Dies war nötig, da der scotchas[3] keine lokalen Labels beherrscht. Dieser Assemblercode wird im fertigen Kompilat direkt vor das eigentliche Programm gehängt und springt dieses beim Starten an.

Alle weiteren Funktionen wurden von Hand assembliert und befinden sich in der libgcc.s.

3.5.1 Multiplikation

Die Multiplikation wird in 3.19 dargestellt. Hierbei wird in den Zeilen 2-4 die alter Registerwerte für die Rücksprungaddresse und den Basepointer in globalen Variablen abgelegt, um mit allen Registern arbeiten zu können.

Dann werden in den Zeilen 5-8 die Register A und X mit den Operanden geladen sowie das Register Y auf `0` gesetzt. Y wird hier als Akkumulator verwendet.

In den Zeilen 9-16 befindet sich die Schleife, die den Multiplikationsalgorithmus umsetzt. Hierbei werden die beiden Operanden gegeneinander geshiftet, und Operand 1 bei ungeradem Operand `0` in Register Y akkumuliert.

Sobald Operand `0` den Wert `0` enthält, ist der Algorithmus durchgelaufen und das Ergebnis befindet sich auf 32 Bit abgeschnitten in Register Y. Abschließend werden in den Zeilen 17-20 noch die Originalwerte der Register hergestellt, mit Ausnahme des Rückgabewertes in Register A.

3.5.2 Shifts

In Abbildung 3.20 wird die verwendete Implementierung des Logischen Shift nach links dargestellt. Die anderen Shifts sind analog gestaltet und werden daher hier nicht dargestellt.

Die Shiftroutinen haben sich in den getesteten Programmen als Bottleneck herausgestellt und wurden daher optimiert. Hierfür wurde die Schleife auf die maximal sinnvolle Shiftbreite von 32 entrollt, und danach in diese Schleife gesprungen.

[3] ScotchAS ist ein einfacher Assembler für den HiCoVec, der in einer vorhergehenden Projektarbeit entstanden ist.

```
1  __mulsi3:
2          st [0 + __retp_libc], A
3          or A, Y, Y
4          st [0 + __bp_libc], A
5          mov X, r0(0)
6          ld A, [X + 4]
7          ld X, [X + 0]
8          xor Y, Y, Y
9  __mul_loop:
10         and 0, A, 1
11         jz [0 + __mul_skip]
12         add Y, Y, X
13 __mul_skip:
14         lsl X, X
15         lsr A, A
16         jnz [0 + __mul_loop]
17         or A, Y, Y
18         ld Y, [0 + __bp_libc]
19         ld X, [0 + __retp_libc]
20         jmp [0 + X]
```

Abbildung 3.19: Multiplikation der libgcc

Zunächst werden die Register wie bei der Multiplikation gespeichert und mit den Operanden geladen. Dann wird in den Zeilen 8-10 das Offset berechnet, wie viele Shifts übersprungen werden können.

Anschließend wird die Addresse direkt vor den Shifts über eine Subroutine in Register A geladen (Zeile 11). Die beiden Addressen werden dann addiert angesprungen, sodass sich die Ausführung mitten in den Shifts befindet.

Hier wird dann nur noch N mal das Register Y geshiftet, und dann anschließend wie bei der Multiplikation zurückgegeben.

3.6 Änderungen am Prozessor

In diesem Abschnitt werden die durchgeführten Änderungen am Prozessor beschrieben, die zum Erstellen des Compilers nötig waren. Die Änderungen wurden dabei im Prozessor selbst und im HiCoSim durchgeführt, allerdings nur im HiCoSim getestet. Daher kann davon ausgegangen werden, dass diese auf dem echten Prozessor noch nicht funktionieren.

3.6.1 Signed Immediates

Nach Absprache mit dem betreuenden Professor wurde beschlossen, dass die Immediate-Werte in den Befehlen sinnvoller als vorzeichenbehaftete Zahlen behandelt werden können.

```
1  __lshlsi3:
2          st [0 + __retp_libc], A
3          or A, Y, Y
4          st [0 + __bp_libc], A    ; Store boring stuff
5          mov X, r0(0)
6          ld Y, [0 + X]            ; Y >> X
7          ld X, [X + 4]
8          and X, X, 31             ; Cap on 31 Shifts
9          or A, 0, 33              ; 33 - (0..31) => (33..2)
10         sub X, A, X
11         jal A, [0 + __ret_pc]    ; A points Here
12         jmp [A + X]              ; Jump into the shifts
13         lsl Y, Y                 ; Lots of Shifting
14         ;;; SNIP.
15         ;;; 31 LSL all in all. do the math
16         lsl Y, Y
17         or A, Y, Y               ; Done Shifting. Return Value
18         ld Y, [0 + __bp_libc]
19         ld X, [0 + __retp_libc]
20         jmp [0 + X]
```

Abbildung 3.20: Shift der libgcc

Somit kann auf dem Stack in beide Richtungen addressiert werden, was die Logik für lokale Variablen und Parameter von Funktionen deutlich vereinfacht.

3.6.2 Transferbefehle zwischen Skalar- und Vektorregistern

Wegen akutem Registermangel musste der bestehende MOV-Befehl so verändert werden, dass ein Transfer in beide Richtungen zu den Vektorregistern ohne Indexregister möglich ist. Für den Compiler wird derzeit nur der Spezialfall gebraucht, dass Transfers zwischen einem Skalarregister und dem Vektorregister r0(0) stattfinden können.

Dieser Fall ließ sich einfach kodieren - beim bestehenden MOV-Befehl wird im Opcode statt dem Offset t das Offset s verwendet um den Index zu bestimmen. Hierdurch ist auch die Hardwareänderung einfach, da das Registerfile schon die 0 ausgibt. Die Änderung bleibt im Assemblercodecode vollständig kompatibel.

Da im HiCoVec auch der MOVA von diesem Wert angesteuert wird, wird dieser ebenfalls umkodiert.

4 Getestete C-Programme

Bei den hier gezeigten Programmen wird im Assemblercode die libgcc und der Initialisierungscode aus Platzgründen weggelassen. Diese befinden sich natürlich trotzdem im generierten Code.

4.1 Hallo Welt

Hier wird ein Programm erklärt, dass "Hallo-Welt" auf dem VGA-Modul des HiCoSim ausgibt. Der C-Code befindet sich in Abbildung 4.1, der Assemblercode in Abbildung 4.3, ein Screenshot in Abbildung 4.2. Dieser Code wurde mit Optimierungen kompiliert, um einige Features hervorzuheben.

Hierbei wird in der Variable i ein Pointer auf den VGA-Speicher gehalten, in p ein Pointer auf das aktuelle Zeichen des Strings. Diese werden in der Schleife in den Zeilen 7-10 parallel hochgezählt, bis das Ende des Strings erreicht ist. Dabei wird das aktuelle Zeichen mit dem Farbwert kombiniert und anschließend in den Speicher geschrieben.

Die Umsetzung in Assemblercode wird nun in Abbildung 4.3 nachvollzogen. Zunächst wird in den Zeilen 6-12 der Stackframe von `main` aufgebaut. in den Zeilen 15-23 befindet sich die Initialisierung der lokalen Variablen. Hierbei ist hervorzuheben, dass der Wert der Variable c (`0xf500`) zu groß ist für eine Kodierung als Immediate. Der GCC hat dies erkannt und hierfür zusätzlich ein globales Symbol mit diesem Wert als Konstanten eingefügt (Zeile 1, LLC1). Zusätzlich hat der Compiler die Initialisierung von p aus der Schleife herausgezogen und in die Zeilen 15-17 gelegt.

Nach der Initialisierung springt er den Kopf der Schleife an. Diese wurde hier vom Optimizer ans Schleifenende verschoben, da dadurch das Sprungmuster besser wird. Auf diese Art werden in der Schleife nur Sprünge nach hinten durchgeführt, die damit in einem Branch Predictor leichter vorherzusagen wären. Dieser ist zwar im HiCoVec derzeit nicht vorhanden, könnte aber hinzugefügt werden.

Im Kopf (Zeilen 41-46) wird hier der Wert $*p$ geladen und in eine vom Compiler eingefügte Stack-Variable kopiert. Anschließend wird der Wert auf `0` geprüft (Schleifenende) und der Schleifenkörper angesprungen.

In diesem (Zeilen 26-39) wird zunächst das effektive Zeichen berechnet. Dies liegt an der Eigenart des VGA-Moduls, dass jedes Zeichen seinen eigenen Farbwert enthalten muss. Diese Wert wird in den Zeilen 26-30 berechnet und auf dem Stack abgelegt. Hierbei ist zu beachten, dass der Compiler die unnötige Zeile 30 nicht wegoptimiert hat, was aber nur ein unwichtiges Artefakt darstellt.

Dann wird in den Zeilen 31-32 die aktuelle VGA-Addresse geladen und das Zeichen in das Modul geschrieben. In den Zeilen 33-35 wird die Addresse dann hochgezählt und wieder im Stack abgelegt. In den Zeilen 36-39 passiert das gleiche für den Pointer in den String.

```
1  char *hallo = "HALLO WELT";
2
3  int main(void){
4    int c = 0xf500;
5    int *i = 0x4000;
6    char *p;
7    for (p = hallo; *p; p++){
8      *i = c | (*p);
9      i++;
10   }
11
12   return 0;
13 }
```

Abbildung 4.1: Hallo Welt - C-Code

Sobald der Wert `0` (Stringterminator) erreicht wird, fällt der Schleifenkopf in Zeile 46 durch und das Programm wird beendet.

4.2 Verkettete Listen

Hier wird ein einfaches Beispiel gezeigt, in dem eine verkettete Liste implementiert wird. Das Beispiel befindet sich in Abbildung 4.4.

Zunächst wird hier in den Zeilen 1-4 eine Struktur `cons` definiert, die aus zwei Elementen besteht. Eine `cons` wird synonym auch als Zelle bezeichnet. Konzeptionell zeigt hierbei `car` immer auf das in der Liste enthaltene Element, `cdr` immer auf die nächste `cons`.

Angelegt wird eine Zelle hierbei nur über die Funktion `cons`, die eine neue Zelle allokiert und diese mit den Parametern der Funktion befüllt. Anschließend wird die Zelle als Pointer zurückgegeben. Hierbei wird die Funktion `new_cons` als Hilfsfunktion verwendet, da die Funktion `malloc` nicht zur Verfügung steht. Diese verwaltet dabei ein globales Array von 100 Zellen und gibt jeweils ein neues Element daraus zurück. Zusätzlich gibt es ein Element `NIL`, dass das Listenende symbolisiert.

In der Anwendung wird dabei in den Zeilen 29-31 eine Liste aufgebaut, die die Zahlen von 1 bis einschließlich 10 enthält. Anschließend befindet sich der Pointer auf das erste Element in der Variable c.

In den Zeilen 33-35 wird dann die Fakultätsfunktion über die Liste gebildet. Hierfür wird eine Akkumulatorvariable a eingefügt, die mit 1 initialisiert wird, da 1 der Identitätswert der Multiplikation ist. Es wird über die Liste iteriert, und jedes Element auf die Variable a aufmultipliziert. Schleifenende ist, sobald die Iterationsvariable `NIL` erreicht. Hierbei ist zu beachten, dass der GCC hier automatisch Aufrufe für die Multiplikationsfunktion einfügt, die sich in der libgcc befindet.

Abschließend wird der berechnete Wert mit dem korrekten Wert verglichen und das Er-

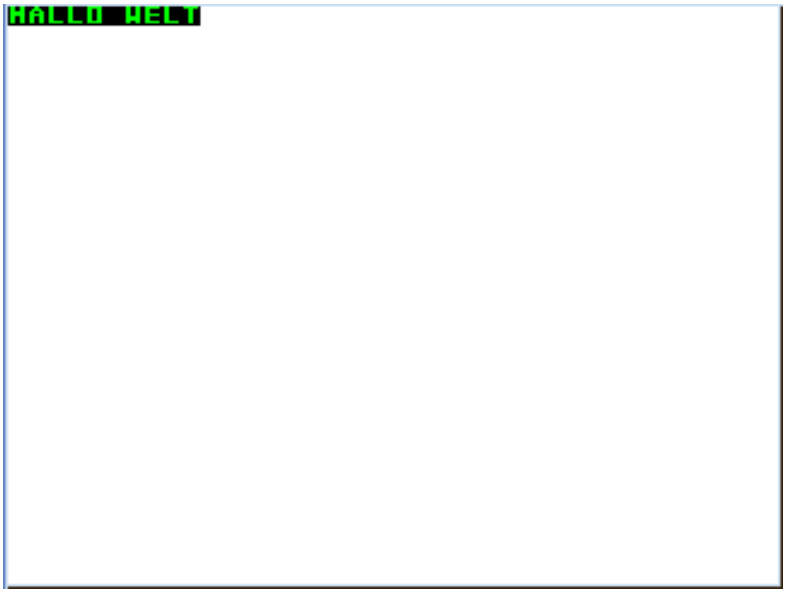


Abbildung 4.2: Hallo Welt - Screenshot

gebnis zurückgegeben. Bei korrekter Ausführung muss das Programm also 1 zurückgeben.

Die Umsetzung in Assemblercode läuft hier recht direkt. Es wird zu viel Code erzeugt um ihn hier wiederzugeben, und es treten keine neuen Besonderheiten auf.

4.3 Verwendung von Assemblercode im C-Programm (Inline Assembly)

In diesem Abschnitt wird anhand eines einfachen Beispiels erklärt, wie sich C-Code mit Assemblercode kombinieren lässt. Das Beispiel befindet sich in Abbildung 4.6. Hierbei können die Constraints aus Abbildung 4.5 verwendet werden.

Die grundsätzliche Syntax ist dabei wie folgt:

```
1  asm("code %0, %1, %2"
2      : Geschriebene Register
3      : Gelesene Register
4      : Geclobberte Register)
```

Als Kurzbeispiel:

```
1  int x = 3;
2  int y = 5;
3  asm("add %0, %0, %1"
4      : "=a"(x)
5      : "a"(y));
```

```
1  LLC1:    .dc 62720
2  main:
3           mov X, r0(0)
4           st [X + -4], A  ;store retp
5           or A, Y, Y
6           st [X + -8], A  ;store old bp
7           sub Y, X, 8     ;bp = sp
8           sub X, X, 28    ;8 for retp and old bp, 20 allocated for local vars
9           mov r0(0), X    ;write sp back
10          jal A, [0 + __main]     ;Initialization - Dummy
11          ld X, [0 + hallo]       ; p = hallo...
12          or A, X, X
13          st [Y + -16], A         ; ... to stack
14          or X, 0, 16384          ; VGA Char Address..
15          or A, X, X
16          st [Y + -8], A          ; ... to Stack
17          ld X, [0 + LLC1]        ; Color f5 (green on black)..
18          or A, X, X
19          st [Y + -12], A         ; ...to Stack
20          jmp [0 + LL2]           ; -> Loop Check
21 LL3:     ld X, [Y + -24]         ;*p
22          ld A, [Y + -12]         ;c
23          or X, X, A              ;c = c | (*p) => load color
24          or A, X, X
25          st [Y + -20], A         ;store colored char in stack
26          ld X, [Y + -8]          ;Current VGA Address
27          st [0 + X], A           ;Write Char
28          add X, X, 1             ;Inc VGA Address
29          or A, X, X
30          st [Y + -8], A          ;Store Address in Stack
31          ld X, [Y + -16]
32          add X, X, 1             ;Inc String Pointer
33          or A, X, X
34          st [Y + -16], A         ;Store it
35 LL2:     ld A, [Y + -16]         ;A = p
36          ld X, [0 + A]           ;X = *p
37          or A, X, X              ;A = X
38          st [Y + -24], A         ;to stack
39          sub 0, X, 0             ;*p != 0?
40          jnz [0 + LL3]           ;yep => loop back
41          jmp [0 + __fun_epilogue] ;Finished
42 LLC2:    .dc 0x48 .dc 0x41 .dc 0x4c .dc 0x4c .dc 0x4f
43          .dc 0x20 .dc 0x57 .dc 0x45 .dc 0x4c .dc 0x54 .dc 0x0
44 hallo:   .dc LLC2
```

Abbildung 4.3: Hallo Welt - Assemblercode

```
1  struct cons {
2    struct cons *car;              /* Left Element */
3    struct cons *cdr;              /* Right Element */
4  };
5
6  struct cons CONSES[100];         /* Object Pool */
7  struct cons NIL = {0, 0};        /* End of List */
8  static int alloc_offset = 0;     /* Offset in Pool */
9
10 /* Get a new Cell */
11 struct cons* new_cons(){
12   return &(CONSES[alloc_offset++]);
13 }
14
15 /* New Cell with (Car . Cdr) */
16 struct cons *cons(struct cons *car, struct cons *cdr){
17   struct cons *c = new_cons();
18   c->car = car;
19   c->cdr = cdr;
20   return c;
21 }
22
23 int main(void){
24   struct cons *c = &NIL;
25   struct cons *iter;
26   int i;
27   int a = 1;
28   /* Build a List (1..10) */
29   for (i = 1; i <= 10; i++){
30     c = cons((struct cons *) i, c);
31   }
32   /* Reduce it with * aka factorial */
33   for (iter = c; iter != &NIL; iter = iter->cdr){
34     a *= (int) iter->car;
35   }
36   return a == 3628800;
37 }
```

Abbildung 4.4: Verkettete Listen - C-Code

“a”	Register A oder Register X
“r”	Register A
“=”	Wird zugewiesen
“+”	Wird gelesen und geschrieben
“ ~ ”	Wird mit nicht verwendbarem Wert überschrieben (clobber)

Abbildung 4.5: Constraints – Bedeutungen

Dabei stehen die Strings “%0-2” für den entsprechenden Parameter, gezählt von links. Da das Register Y statisch belegt wird, können hier nur 2 Register verwendet werden. Die Constraints “r” und “a” stehen für Register. Hierbei ist es bemerkenswert, dass diese vom GCC selbst allokiert und mit passenden Werten geladen werden. In Abbildung 4.6 wird in Zeile 11 die konkrete Expansion dargestellt. Der GCC setzt hier einen Reload für die Variable x direkt vor die Zeile Assemblercode.Der Compiler weiss hier nichts von den Vektorregistern. Aus diesem Grund müssen diese vom Anwender verwaltet werden.

Hierbei ist grundsätzlich zu beachten, dass ein Block Assemblercode aus Sicht des GCC nur aus einem opaken String und Constraints besteht. Die Constraints werden dabei vom GCC verwendet um Abhängigkeiten zwischen dem Assemblercode und dem umgebenden Code aufzustellen, und den ganzen Code dann in Abhängigkeit davon zu optimieren.

Beim Mischen von C und Assemblercode ist es dabei grundsätzlich sinnvoll, wenn möglich den Kontrollfluss in C zu belassen und nur einzelne Probleme in Assemblercode auszudrücken. Dadurch kann der GCC noch einige Optimierungen durchführen, wie das Entrollen von Schleifen (unroll loops) oder die lokale Expansion von Funktionen (inline function).

4.3.1 Beispiel mit Optimierung

In diesem Abschnitt wird die Effektivität des Optimizers demonstriert. Hier wurde ein kurzes Beispiel gewählt, dass mit aktiviertem Optimizer noch funktioniert. Für allgemeine Nutzbarkeit muss hier noch gearbeitet werden.

Das Beispiel ruft zwei Funktionen hintereinander auf. Die Innerste soll dabei ihren ersten Parameter um amount Stellen nach rechts shiften. Als Ergebnis soll dabei (128 » 5) = 4 zurückgegeben werden. Der Shift wurde dabei in inline Assembly umgesetzt, da das Entrollen eines C-Shifts noch nicht implementiert ist. Der C-Code befindet sich dabei in Abbildung 4.7.

Der unoptimierte Assemblercode befindet sich in den Abbildungen 4.8 und 4.9. Er ist dabei wie im Textbuch umgesetzt – Jede Variable befindet sich im Stack, jeder Funktionsaufruf wird ausgeführt, die Schleife wird komplett durchlaufen. Er ist damit eine direkte Transliteration des C-Codes.

Wesentlich interessanter sieht es dabei in Abbildung 4.10 aus. Hier wurde der GCC mit der Optimierungsstufe `-Os` (small) betrieben. Bei `-O1` bis `-O3` wird allerdings der gleiche Code generiert.

Der GCC hat dabei über die ab der Optimierungseinstellung `-O1` implizite Optimierung `-finline-small-functions` beschlossen, dass die Funktionen `foo` und `_shr` klein genug sind, um inline expandiert zu werden.

```
1  int input1[8];
2  int input2[8];
3  int results[8];
4
5  void load_vector_reg1(int *arr){
6    int i;
7    int x;
8    for(i = 0; i < 8; i++){
9      x = arr[i];
10     asm("mov r1(%0), %1" : : "a"(i), "a"(x));
11 /* expands to:  mov r1(A), X */
12   }
13 }
14
15 void load_vector_reg2(int *arr){
16   int i;
17   int x;
18   for(i = 0; i < 8; i++){
19     x = arr[i];
20     asm("mov r2(%0), %1" : : "a"(i), "a"(x));
21   }
22 }
23
24 int main(void){
25   int i;
26   for(i = 0; i < 8; i++){
27     input1[i] = i;
28     input2[i] = 8 - i;
29   }
30   load_vector_reg1(input1);
31   load_vector_reg2(input2);
32   asm("vsub.dw r1, r1, r2");
33   asm("vst [0 + %0], r1" : : "a"(results));
34   return 0;
35 }
```

Abbildung 4.6: C-Programm mit Vektorassemblercode

```
1  static int _shr(int n, int amount){
2    int x = n;
3    int i = amount;
4    while(i--){
5      asm("lsr %0, %0": "+a" (x));
6    }
7    return x;
8  }
9
10 static int foo(){
11   return _shr(128, 5);
12 }
13
14 int main(void){
15   return foo();
16 }
```

Abbildung 4.7: Optimizer in Aktion - C-Datei

Zusätzlich hat er die Konstanten verfolgt und festgestellt, dass die Schleife in den Zeilen 4-6 nur von der Konstanten 5 abhängt. Diese Schleife wurde deswegen komplett entrollt, sodass nur noch die 5 Assemblerinstruktionen für die Shifts übrig geblieben sind.

Da die Funktionen `_shr` und `foo` als `static` deklariert sind, wurden sie gar nicht in den generierten Code aufgenommen. Würde dieser Qualifier nicht dabei stehen werden sie trotzdem generiert, da der Compiler davon ausgeht, dass eventuell anderer Code gegen diese Funktion linken will.

Die komplette Optimierung wurde dabei auf GIMPLE-Ebene durchgeführt. Der Compiler hat hierbei einen asm-Knoten angelegt, und über die Constraints dessen Abhängigkeiten verfolgt.

Die einzige maschinenspezifische Optimierung die durchgeführt wurde war das Ablegen der zu Shiftenden Variable in Register A, sodass dieses direkt zurückgegeben werden kann (Assemblerzeile 11).

Ohne Optimierung hat das Programm 151 Instruktionen ausgeführt, mit Optimierung nur 28. Hierbei muss beachtet werden, dass konstant 22 Instruktionen für die Initialisierung des C-Programms benötigt werden und daher bei beiden abgezogen werden müssen. Das Endergebnis beträgt damit 129 Instruktionen gegenüber 6, wobei die 6 hier sogar als optimal für den HiCoVec betrachtet werden können.

4.4 Sieb des Eratosthenes

Um die Umsetzung von algorithmischem Code zu testen, wurde eine leicht modifizierte Variante des Sieb des Eratosthenes programmiert. Diese befindet sich in Abbildung 4.11. Der

```
1  _shr:
2          mov X, r0(0)
3          st [X + -4], A  ;store retp
4          or A, Y, Y
5          st [X + -8], A  ;store old bp
6          sub Y, X, 8     ;bp = sp
7          sub X, X, 20    ;8 for retp and old bp, 12 allocated for local vars
8          mov r0(0), X    ;write sp back
9
10         ld A, [Y + 8]
11         st [Y + -12], A
12         ld A, [Y + 12]
13         st [Y + -8], A
14         jmp [0 + LL2]
15 LL4:
16         ld A, [Y + -12]
17         lsr A, A
18         st [Y + -12], A
19 LL2:
20         ld A, [Y + -8]
21         sub 0, A, 0
22         jz [0 + LL3]
23         or A, 0, 1
24 LL3:
25         st [Y + -16], A
26         ld X, [Y + -8]
27         add X, X, -1
28         or A, X, X
29         st [Y + -8], A
30         ld A, [Y + -16]
31         sub 0, A, 0
32         jnz [0 + LL4]
33         ld A, [Y + -12]
34
35         jmp [0 + __fun_epilogue]
```

Abbildung 4.8: Optimizer in Aktion - Unoptimierter Assemblercode (O0) - Teil 1

```
1
2  foo:
3          mov X, r0(0)
4          st [X + -4], A  ;store retp
5          or A, Y, Y
6          st [X + -8], A  ;store old bp
7          sub Y, X, 8     ;bp = sp
8          sub X, X, 12    ;8 for retp and old bp, 4 allocated for local vars
9          mov r0(0), X    ;write sp back
10
11         mov A, r0(0)
12         add A, A, -8
13         mov r0(0), A
14         mov X, r0(0)
15         or A, X, X
16         st [Y + -8], A
17         or A, 0, 128
18         ld X, [Y + -8]
19         st [0 + X], A
20         or A, 0, 5
21         ld X, [Y + -8]
22         st [X + 4], A
23         jal A, [0 + _shr]
24         mov X, r0(0)
25         add X, X, 8
26         mov r0(0), X
27
28         jmp [0 + __fun_epilogue]
29
30 ;       .globl  main
31 main:
32         mov X, r0(0)
33         st [X + -4], A  ;store retp
34         or A, Y, Y
35         st [X + -8], A  ;store old bp
36         sub Y, X, 8     ;bp = sp
37         sub X, X, 8     ;8 for retp and old bp, 0 allocated for local vars
38         mov r0(0), X    ;write sp back
39
40         jal A, [0 + __main]
41         jal A, [0 + foo]
42
43         jmp [0 + __fun_epilogue]
```

Abbildung 4.9: Optimizer in Aktion - Unoptimierter Assemblercode (O0) - Teil 2

```
1  main:
2          mov X, r0(0)
3          st [X + -4], A  ;store retp
4          or A, Y, Y
5          st [X + -8], A  ;store old bp
6          sub Y, X, 8     ;bp = sp
7          sub X, X, 8     ;8 for retp and old bp, 0 allocated for local vars
8          mov r0(0), X    ;write sp back
9
10         jal A, [0 + __main]
11         or A, 0, 128    ; A = 128
12 ;APP
13         lsr A, A                ; inline assembly
14         lsr A, A
15         lsr A, A
16         lsr A, A
17         lsr A, A
18 ;NO_APP
19
20         jmp [0 + __fun_epilogue]
```

Abbildung 4.10: Optimizer in Aktion - Optimierter Assemblercode (Os)

generierte Assemblercode wird in Ausschnitten wiedergegeben.

Die grundlegende Idee bei diesem Algorithmus ist, Primzahlen zu finden, indem nicht-Primzahlen markiert werden. Das Beispiel ist dabei so aufgebaut, dass alle Primzahlen im Bereich 1-1000 gefunden werden. Hierfür wird ein Array der Größe 1000 angelegt, und mit den Zahlen 3-1000 befüllt (Zeilen 12 und 13). Die Zahlen 1 und 2 können als Spezialfall ignoriert werden. Die 2 wird gleich im der nächsten Zeile behandelt und dem Ergebnisarray hinzugefügt. Die Betrachtung ist dabei so, dass alle nicht-Primzahlen mit einer 0 überschrieben werden.

Dann kommt eine Schleife, die den Algorithmus enthält. Das Verlassen der Schleife (und des Programms) geschieht dabei in Zeile 19, sobald die Zähl-Variable größer als der valide Wertebereich ist.

In der Schleife in den Zeilen 18-23 wird die nächste Primzahl gesucht. Dabei ist nach dem Prinzip des Algorithmus die nächste Zahl im Array, die ungleich 0 ist, prim. Hierbei wird die 3 als Initialwert in Zeile 8 direkt gesetzt. Die Suche läuft dabei in 2er-Schritten, da alle Primzahlen außer 2 ungerade sind. Falls eine Zahl ungleich 0 gefunden wurde, wird diese der Variable $recent$ zugewiesen, die die zuletzt gefundene Primzahl darstellt.

Die gefundene Primzahl wird dann direkt in das Ergebnisarray geschrieben (Zeile 24). Anschließend werden alle vielfachen der Primzahl mit 0 überschrieben, da diese per Definition nicht prim sind. Dies geschieht in den Zeilen 26 und 27. Hierbei ist zu beachten, dass die Schleife bei größeren Primzahlen immer schneller wird, da das Inkrement der Zählervariable entsprechend größer ist.

Der Code benötigt auf dem HiCoVec 180k Instruktionen, auf dem x86 25k. Beide Kompilate laufen dabei ohne Optimierung, da der Optimizer bei diesem Code auf dem HiCoVec nicht funktioniert. Dies ist ein respektables Ergebnis, da auf dem x86 hier alle Befehle direkt als Instruktionen verbaut werden können, beim HiCoVec aber viele Subroutinen verwendet werden.

```
1  int arr[1000];
2  int results[200];
3
4  int main(void){
5    int i;
6    int j;
7    int k;
8    int recent = 3;                    /* Next starting point in search */
9    int offset = 0;
10   /* Initialize Array - Don't care about 0..2 as
11      they are handled elsewhere. */
12   for(i = 3; i < 1000; i++)
13     arr[i] = i;
14   results[offset++] = 2;
15
16   while(1){
17     /* Search for next Prime, skipping even numbers */
18     for (j = recent;; j += 2){
19       if (j >= 1000)              /* Ran past the end of the array */
20         return 0;
21       if ((recent = arr[j]))      /* Remember next starting point */
22         break;
23     }
24     results[offset++] = recent;   /* Found a Prime. Store it */
25     /* Zero out the multiples of the found Prime */
26     for(k = recent; k < 1000; k += recent)
27       arr[k] = 0;
28   }
29
30   /* results now contains the primes below 1000.
31      Printing is left as an exercise  */
32   return 0;
33 }
```

Abbildung 4.11: Vereinfachtes Sieb des Eratosthenes - C-Quellcode

```
1   ld A, [Y + -20] ; A <= j - Line 1 x86
2   mov X, r0(0)    ; arr is an integer array
3   add X, X, -8    ; index has to be shifted by 2
4   mov r0(0), X    ; shifting is a subroutine....
5   mov X, r0(0)    ; call goes on...
6   st [0 + X], A
7   or A, 0, 2
8   mov X, r0(0)
9   st [X + 4], A
10  jal A, [0 + __ashlsi3]
11  mov X, r0(0)
12  add X, X, 8
13  mov r0(0), X    ; end of call
14  st [Y + -28], A ; now calculate the address
15  ld X, [0 + LLC0] ; A <= arr
16  or A, X, X      ; back to the stack. no more registers
17  st [Y + -32], A
18  ld X, [Y + -28] ; base
19  ld A, [Y + -32] ; offset
20  add X, X, A     ; final address
21  or A, X, X
22  st [Y + -28], A
23  ld A, [Y + -28] ; redundant reload, optimizer is not turned on
24  ld A, [0 + A]   ; finally load arr[j] - Line 2 x86
25  st [Y + -12], A ; and store it into l - Line 3 x86
```

Abbildung 4.12: Auszug Arrayzugriff HiCoVec

```
1   movl    -24(%ebp), %eax      ; eax <= j
2   movl    arr(,%eax,4), %eax   ; eax <= arr[eax]
3   movl    %eax, -16(%ebp)      ; l <= eax
```

Abbildung 4.13: Auszug Arrayzugriff X86

5 Ergebnisse

In diesem Kapitel wird die Verwendbarkeit sowie die Einschränkungen des erstellten Compilers beschrieben.

5.1 Verwendbarkeit

In der Arbeit ist ein verwendbarer Port des GCC entstanden, der Code für den HiCoVec produzieren kann. Dabei kann er als Prototyp betrachtet werden, der als Basis für eine Weiterentwicklung verwendet wird, bei der dann weitere Features des GCC unterstützt werden.

Die Performance des generierten Codes ist erstaunlich gut (vgl. Auswertung aus dem Kapitel 4). Es wird erwartet, dass diese bei Weiterentwicklung noch deutlich steigen wird, da der Compiler derzeit noch einige Codepatterns umständlich umsetzt.

Während der Entwicklung wurden zum Testen des Compilers etwa 50 einfache C-Programme geschrieben, die unterschiedliche Funktionalitäten verwenden. Diese wurden mit dem erstellten Compiler übersetzt und deren Ergebnisse und Performance verglichen mit Kompilaten, die auf einem Intel Core 2 Duo mit dem GCC 4.3.4 gebaut wurden. Zusätzlich wurden als externe Bibliothek die longdiv-Routinen aus dem GCC erfolgreich kompiliert und eingebunden. Diese Routinen implementieren Division und Modulo in portablem C und sind Teil des GCC. Sie befinden sich in der Datei `gcc/config/udivmodsi.c`. Sie werden automatisch eingebunden, wenn der entsprechende Operator verwendet wird.

5.2 Bewertung der Effizienz des generierten Codes

5.2.1 Statistische Auswertung der getesteten Programme

In Abbildung 5.1 befindet sich eine Auflistung der mit dem HiCoSim gezählten Befehle der einzelnen Programme. Diese soll darauf untersucht werden, ob durch zusätzliche Befehle im HiCoVec oder Patterns im GCC stärker optimiert werden kann. Alle Programme wurden dabei ohne Optimizer kompiliert.

Hierbei fällt bei Eratosthenes auf, dass fast alle Funktionsaufrufe auf die arithmetischen Shifts stattfinden, da die Pointer hier für das Array skaliert werden müssen. Hierfür werden für jeden Funktionsaufruf im Schnitt 6 MOVs benötigt, sowie mehrere LD und ST für Spills der Funktionsparameter. Dies lässt sich nahezu komplett eliminieren, wenn im GCC ein Pattern für konstante Shifts eingeführt wird. Dies heißt, dass hier von den Funktionsaufrufen nur noch die Shifts selbst ausgeführt würden. Es würden dabei pro Funktionsaufruf 6 MOV, 4 ST, 4 LD, 1 ADD und 1 ST wegfallen. Dies entspricht etwa 112000 Befehle, die wegfallen würden. (16 Instruktionen auf 6912 Shifts, davon 8 Speicherzugriffe). Dies ist bei einer Gesamtzahl von derzeit 178000 Instruktionen ein beeindruckender Wert.

Befehl	Fibonacci	Eratosthenes	Listen	Division
OR	109486	23994	180	39
ADD	98552	13658	128	16
SUB	65701	7582	76	35
LSL	0	6912	59	0
LSR	0	0	29	8
AND	0	3456	39	0
LD	197176	42308	431	94
ST	175223	31277	329	59
JMP	116227	14661	95	10
JZ	6765	503	13	2
JNZ	0	0	29	4
JC	17710	0	0	0
JNC	20	3284	9	1
JAL	47966	6914	52	5
MOV	175140	24196	254	29

Abbildung 5.1: Befehlsstatistiken einiger getesteter Programme (HiCoSim)

Diese Erweiterung gilt dabei für jeden indizierten Zugriff auf Array-Elemente und ist damit auf so gut wie jedes C-Programm übertragbar.

Beim Fibonacci-Programm (Code wird hier nicht aufgelistet, da trivial) finden wie erwartet sehr viele Funktionsaufrufe statt. Hierbei fällt auf, dass pro Funktionsaufruf im Schnitt 3.5 MOV-Befehle benötigt werden. Diese MOV-Befehle würden komplett entfallen, wenn der Stackpointer ein richtiges Register wäre. Dies entspricht in diesem Programm etwa 175k von 1M Instruktionen. Zusätzlich wird dadurch bei jedem Call mit variablen Parametern ein Spill erzeugt, der ein weiteres Load/Store-Paar generiert. Um den Stackpointer als eigenes Register zu führen, müssten allerdings sowohl der Prozessor als auch der GCC geändert werden.

5.3 Einschränkungen

In diesem Kapitel werden die Einschränkungen des Compilers beschrieben, und wie diese behoben werden könnten.

5.3.1 Datentypen

Die Verwendung von Floats ist nicht möglich. Es sind zusätzliche MOV-Patterns nötig, damit sich die Soft-Float-Bibliothek bauen lässt, konkret für die Modi DF (double float) und HI (half integer). Diese könnten aufwändig emuliert werden, indem die übrigen Bits ausmaskiert werden. Alternativ wären zusätzliche Load-Befehle möglich.

Da alle ganzzahligen Datentypen in 32 Bit abgelegt werden, wird auch zusätzlicher Speicher bei `chars` und `shorts` gebraucht, was sich besonders bei Strings bemerkbar macht. Falls der Programmierer hier nur mit 8 bzw. 16 Bit Genauigkeit rechnen will, muss er diese selbst nach

der Rechnung maskieren.

5.3.2 Register Spills

Der Compiler beendet sich mit einer Fehlermeldung, wenn die Register komplett ausgehen. Dies passiert besonders leicht wenn der Optimizer angeschaltet ist, da der Compiler hier versucht, alte Rechnungsergebnisse in Registern zu halten.

Wenn nun ein Wert bei einer Berechnung schon in einem Register liegt, versucht der Optimizer, es hier zu halten, da der Transfer Register nach Register billiger ist als der Transfer Speicher nach Register.

Der Effekt wird dadurch verstärkt, dass im Backend noch keine Kosten für die einzelnen Befehle implementiert sind, sodass der GCC seine Default-Einstellungen benutzt. In diesen ist insbesondere ein Load mit einer Immediate-Konstanten teurer als ein Register-Transfer.

Der Fehler kann grundsätzlich nur durch mehr Register behoben werden. Es ist allerdings möglich, mit unterschiedlichen Einstellungen des Optimizers zu arbeiten, oder diesen gar nicht zu aktivieren. Mit deaktiviertem Optimizer produziert der Compiler hier bei allen getesteten Programmen lauffähigen Code.

5.3.3 Inline-Assemblercode

Bei Inline-Assemblercode können in einem einzigen Assemblerblock nur 2 Register verwendet werden. Daher sollten diese Blöcke so kurz wie möglich gehalten werden. Also statt

```
1   int x, y, z;
2   asm ("lsr %0, %1\n\t"
3        "add %2, %1, %0" : "+a"(x), "=a"(y) : "a"(z));
```

muss dies aufgesplittet werden in:

```
1   int x, y, z;
2   asm("lsr %0, %1" : "=a"(x) : "a"(z));
3   y = x + z;
```

Es ist allgemein sinnvoll, so viel Code wie möglich in C zu schreiben und Assembly immer nur in einzelnen Befehlen zu verwenden.

5.3.4 Falsche Verarbeitung der Flags im Backend

Es scheint sich noch ein Fehler in der Verarbeitung des Condition Codes zu befinden (vgl. Abschnitt 3.3.5). So führte zum Beispiel eine Zuweisung in einer Schleifenkondition manchmal zu Problemen:

```
1   while((i = i->next))
2     ...
```

Hier scheint die Auflösung nicht korrekt zu funktionieren, dass hier noch eine Überprüfung stattfinden muss.

Ein wahrscheinlich verwandtes Problem wurde in der Arbeit behoben. Der Load aus dem MOV-Pattern (vgl. Abbildung 3.13) bekam zunächst das Attribut `set_z`, da er das Zero-Flag setzen kann. Der Compiler hat allerdings trotzdem versucht, das Carry-Flag zu verwenden, was auf eine inkorrekte Definition der Funktion `hicovec_notice_update_cc` schließen lässt. Als Workaround wurde daher das Attribut auf `clobber` gesetzt, sodass der Compiler hier einen zusätzlichen Vergleich anstellen muss.

6 Fazit und Ausblick

6.1 Fazit

Im Lauf der Arbeit ist ein verwendbarer C-Compiler enstanden, der sich gut als Ausgangspunkt für eine vollständige Toolchain eignet. Die Anpassungen am Prozessor waren denkbar einfach, die Portierung des GCC war nach einigen Startschwierigkeiten relativ geradlinig.

Der Compiler produziert im Rahmen der durchgeführten Tests durchgängig lauffähigen Code. Selbst mit den oben genannten Einschränkungen sollte der Compiler leichter verwendbar sein als der vorhandene Assembler. Durch die Verwendung von Inline Assemblercode kann sich hier zumindest der Treibercode für Vektor-Algorithmen in C schreiben lassen, was eine Vereinfachung darstellt.

6.2 Ausblick

In diesem Abschnitt werden kurz mögliche Fortführung des Projekts am Compiler, der Toolchain sowie dem Prozessor selbst umrissen.

6.2.1 Mögliche Weiterentwicklungen des Compilers

Shift-Patterns

Es ist sehr sinnvoll, dedizierte Shift-Patterns zu entwickeln. Die meisten Shifts in C-Programmen sind zur Compile-Zeit konstant und können daher vom Compiler inline geschrieben werden. Dies ist zumindest für die logischen Shifts leicht in HiCoVec-Assemblercode umzusetzen.

Durch diese Änderung sollte die Berechnung von Addressen wesentlich besser funktionieren, da kein Funktionsaufruf mehr stattfinden muss, der neue Spills erzeugt.

Zusätzliche Multiplikationsroutine

Es besteht die Möglichkeit, in der libgcc1 eine zweite Multiplikationroutine zu implementieren, die den Multiplikationsbefehl der Skalareinheit mitbenutzt. Dies wird aufwändig, da dieser nur die jeweils unteren 16 Bit der Eingaberegister verarbeiten und daher die Werte vorher gesplittet, getrennt verrechnet, und abschließend neu zusammengesetzt werden müssten.

Diese Routine kann dann optional statt der Bestehenden dazugelinkt werden, wenn der Prozessor für Multiplikation konfiguriert wurde.

Verwendbare Optimierung

Im Compiler müssen noch einige Bugs gefunden und entfernt werden, damit der Optimizer sinnvoll verwendet werden kann. Hierzu gehört insbesondere der Fehler mit den Flags aus Abschnitt 5.3.4.

Ein sehr wichtiger Schritt ist hierbei auch die Definition einer Kostenfunktion, mit der der GCC die Effizienz des generierten Codes bewerten kann. Diese Möglichkeit wird derzeit noch gar nicht verwendet, sodass der Optimizer in einigen Fällen schlechteren Code produziert als komplett unoptimierter Code.

Selbst dann ist der Optimizer noch nicht sinnvoll verwendbar, da er mehr Register zum Spillen braucht. Hierzu ist es sinnvoll, die Vektoreinheit als Auslagerungsspeicher einzubinden, was allerdings zusätzlichen Aufwand erfordert.

Vektorisierung

Ein späterer Schritt ist, die Vektorisierung des GCC zu verwenden. Hierfür muss die komplette Vektoreinheit noch dem GCC als eigene Register bekannt gemacht werden. Hierzu ist die Einführung eines für die Vektoreinheit passenden Vektormodus nötig, sowie die Beschreibung der Register und Befehle neben denen der Skalareinheit.

Dieser Schritt ist sehr aufwändig und daher wahrscheinlich nicht in naher Zukunft zu realisieren.

Stack in Vektorregistern

Es ist prinzipiell auch möglich, den Stack in den Vektorregistern abzulegen, um so selten wie möglich den normalen Stack im Arbeitsspeicher verwenden zu müssen. Hierbei kann man sich an den Registerfenstern des SPARC orientieren.

Für diesen Ansatz wurde schon ein Modell ausgearbeitet, dass sich aber als unpassend für die Architektur des GCC herausgestellt hat. Das Problem war, dass der Framepointer sinngemäß ein Pointer in den Registerstack ist, aber der GCC erwartet, dass dieser in den Speicher geht. Prinzipiell ist dieses Modell zwar umsetzbar, aber es war mit dem gegenwärtigen Wissensstand zu aufwändig.

Ein zusätzliches Problem stellt noch das Laufzeitverhalten dar. Wenn die Vektorregister überlaufen, muss der Stack transparent für den Programmcode in den Speicher kopiert werden, sodass immer genug Platz zur Verfügung steht. Hierbei besteht das Problem darin, dass sich hier leicht ein Fall konstruieren lässt, in dem der Stack oft hin- und her kopiert werden muss. Dieses Verhalten im Compiler zu verhindern ist schwer, sodass eventuell der User mithelfen muss.

Hierzu kann eine Funktion implementiert werden, die den Stack vor kritischen Stellen manuell kopiert.

6.2.2 Portierung der binutils

Ein wichtiger Schritt ist auf jeden Fall die Portierung der binutils, insbesondere des GNU Assemblers gas. Dadurch könnte direkt das ELF-Format verwendet werden, sodass Objektda-

teien gelinkt werden können.

Sobald der Assembler und Linker funktionieren, kann die bestehende libgcc integriert werden, sodass ein normaler GCC-Cross-Compiler gebaut werden kann. Zusätzlich können Projekte aus mehreren Dateien bestehen, sowie bestehende Bibliotheken verwendet werden.

Hierfür muss im GCC der Driver implementiert werden. Dieser Schritt wurde mangels Testbarkeit in dieser Arbeit nicht durchgeführt. Solange die binutils komplett verwendet werden und deren normaler Buildprozess genutzt wird, kann dieser Schritt übersprungen werden.

Danach müssen die ausgelassenen Direktiven im GCC angepasst werden. Diese befinden sich alle in der Datei 'hicovec.h' in der Sektion "ASSEMBLER FORMAT". Details lassen sich hier aus der GCC-Dokumentation entnehmen.

6.2.3 Mögliche Optimierungen durch Änderungen am HiCoVec

In diesem Abschnitt werden mögliche Änderungen am Prozessor umrissen. Die Vorschläge hier sind rein subjektiv und sind nur aus Sicht des Compilerbauers, nicht des Hardwareentwicklers. Sie sollten daher vor der Umsetzung sorgfältig durchdacht werden, da sie die Hardware deutlich verkomplizieren können.

Statisch indizierter MOV-Befehl

Es könnte ein MOV-Befehl eingeführt werden, dessen Offset statisch kodiert wird. Dadurch könnten die Vektorregister als expliziter Cache verwendet werden. Zusätzlich können Spills zunächst in die Vektorregister stattfinden, was sowohl schneller ist, als auch in diesem Fall keine zusätzlichen Spills verursacht.

Hierzu könnten die aktuellen don't care-Bits[1] beim MOV-Befehl als statischer Index verwendet werden, wenn das Indexregister `0` ist. Dies kostet allerdings zusätzliche Logik in der Hardware, eventuell kann hier eine bessere Lösung gefunden werden.

Relative Sprünge

Einige Patterns könnten viel leichter ausgedrückt werden, wenn Sprünge relativ zum Program Counter möglich wären. Dies lässt sich am leichtesten realisieren, wenn bei Sprungbefehlen statt dem Register Y der PC geladen wird (bei gleicher Kodierung). Da Y bei der aktuellen ABI den Frame Pointer darstellt, ist kein praktischer Nutzen daraus zu ziehen, diesen als Sprungregister zu verwenden.

Relative Sprünge sind derzeit zwar emulierbar, indem eine Routine aufgerufen wird, die direkt zurückspringt, allerdings wird dafür ein zusätzliches Register benötigt. Ausserdem wird die Berechnung der Offsets insbesondere von Hand schwerer.

Diese Änderung würde ausserdem Code in den nicht direkt addressierbaren Speicherregionen überhaupt erst verwendbar machen. Das bedeutet beim HiCoVec, dass Code nur bis zur Addresse `0x7ffc` sinnvoll nutzbar ist. Hier muss allerdings der Compiler angepasst werden, dass zwischen nahen- und fernen calls unterschieden wird, vergleichbar zum Modell des 286.

[1] don't care-Bits sind Bits im Opcode, bei denen es nach Spezifikation keinen Unterschied macht, ob diese gesetzt werden oder nicht. Hierdurch kann Code aufwärtskompatibel geschrieben werden, wenn neue Befehle die alten don't care-Bits verwenden.

Alternativ kann jeder Call relativ ablaufen, was einen zusätzlichen Load pro Call bewirken würde.

Mehr Register

Da nur wenige Register vorhanden sind, musste der Stackpointer in ein Vektorregister ausgelagert werden. Dadurch muss bei jeder Änderung des SP dieser in ein Register kopiert, geändert, und zurückgeschrieben werden. Das Anlegen von Funktionsargumenten beim Aufruf passiert immer relativ zum Stackpointer, sodass dieser in einem Register gehalten muss. Da A zum Schreiben benötigt wird, kann dies nur im Register X geschehen. Dadurch werden aber zwangsläufig die Werte gespillt, die sich vorher in diesen Registern befanden, sodass beim Funktionsaufruf etwa doppelt so viele Zugriffe in den Stack stattfinden wie grundsätzlich nötig.

Hierzu kann zum Beispiel eine 2-Operand-Kodierung eingesetzt werden, die ähnlich funktioniert wie die des M68K oder des x86. Hiermit wären bei gleicher Befehlsbreite und -Kodierung 7 Register kodierbar, wovon 2 für BP und SP verbraucht werden. Dadurch können 5 Register vom Compiler verwendet werden, wodurch dieser wesentlich besseren Code produziert. Diese Kodierung ist in der Machine Description auch leicht zu beschreiben.

Store-Befehle aus jedem Register

Wenn der Befehl ST so kodiert wäre, dass die beiden don't-care-Bits für dd genutzt würden, um das Quell-Register anzusteuern, wäre ein ST aus jedem Register möglich.

Im Compiler bedeutet dies, dass die Zahl der Reloads deutlich vermindert werden würde, da jetzt oft ein Austausch der Register X und A stattfinden muss.

Zusätzliche Sprung JGT

Die Branchpatterns wären sehr einfach zu formulieren, wenn es eine spezielle Instruktion für den Fall $>=$ geben würde. Dieser muss gleichzeitig die Flags Carry und Nicht-Zero prüfen. Derzeit sind in den Sprung-Opcodes noch 10 Patterns frei, sodass dieser leicht umzusetzen ist.

Glossar

Basepointer Hier: Synonym für Framepointer.

Framepointer Pointer auf die Basis des aktuellen Frames.

RISC Reduced Instruction Set Computer. Eine Prozessorarchitektur, bei der die Anzahl und Mächtigkeit einzelner Befehle reduziert ist. Dies führt oft dazu, dass der Prozessor einfacher aufgebaut und schneller getaktet werden kann.

Rücksprungpointer Nach einem Funktionsaufruf die Addresse, an die die aufgerufene Funktion zurückspringen soll.

Stack Ein Stack (Stapel) ist eine einfache Datenstruktur, auf die man Werte legen und wieder herunternehmen kann. Ein Stack wird in der Sprache C für lokale Variablen verwendet.

Stackframe Speicherbereich für den lokalen Zustand einer Funktion. Enthält lokale Variablen, den alten Basepointer sowie den Rücksprungpointer

Stackpointer Pointer auf das zuletzt hinzugefügte Element eines Stacks.

VHDL Very High Speed Integrated Circuit Hardware Description Language. Eine Hardwarebeschreibungssprache, für die viele Werkzeuge zur Verfügung stehen.

VLIW Very Large Instruction Word. Eine Prozessorarchitektur, bei der eine Instruktion aus mehreren Instruktionen besteht

Literaturverzeichnis

[Fou10] Free Software Foundation. Autoconf - GNU Project. `http://www.gnu.org/software/autoconf/`, 2010.

[GBJL00] D. Grune, H. Bal, C. Jacobs, and K. Langendoen. *Modern Compiler Design*. Wiley, 1st edition, October 2000.

[IL06] Paolo Ienne and Rainer Leupers. *Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon)*. Morgan Kaufmann, 1st edition, July 2006.

[Kie10] Prof. Dr.-Ing. Gundolf Kiefer. HiCoVec Projektseite. `http://www.hs-augsburg.de/~kiefer/hicovec/index.html`, 2010.

[Man07] Harald Manske. Entwicklung eines konfigurierbaren Vektorprozessors. Diplomarbeit, Hochschule Augsburg, 2007.

[Nil00] Hans-Peter Nilsson. Porting GCC for Dunces. `http://ftp.axis.se/pub/users/hp/pgccfd/pgccfd.pdf`, 2000.

[Rau03] J. Prof. Dr. Andreas Rausch. *Softwarearchitektur verteilter Systeme (Vorlesungsskript)*. Technische Universität Kaiserslautern, 2003.

[Sta92] Richard M. Stallman. GCC Internals. `http://gcc.gnu.org/onlinedocs/gccint/`, 1992.