UNIVERSITÄT MANNHEIM

Laboratory for Dependable Distributed Systems

Bachelor Thesis

Entwicklung des ULIX-Compilers

Balthasar Biedermann

29. September 2008

Erstkorrektor: Prof. Dr. Felix C. Freiling
Zweitkorrektor: Dipl.-Inform. Thorsten Holz
Betreuer: Prof. Dr. Felix C. Freiling

Zusammenfassung

Komplexe Software, wie in diesem Fall ein Betriebssystem, kann man kaum in reinem Assembler entwickeln. Deshalb soll es für die ULIX-Architektur auch möglich sein, zumindest in der Hochsprache C zu programmieren. Da diese Architektur real nicht existiert und speziell für ULIX entwickelt wurde, gibt es noch keinen Compiler dafür. Um diese Lücke zu schließen, habe ich das GCC-Backend an die ULIX-Architektur angepasst. Dadurch ist es möglich in vielen Hochsprachen für ULIX zu programmieren. Außerdem profitiert der generierte Code vom hohen Reifegrad des GCC.

Danksagungen

Zuerst möchte ich mich bei Prof. Dr. Felix Freiling dafür bedanken, dass er mir diese interessante Bachelor Thesis ermöglicht und mich von der Vorbereitungszeit bis zur fertigen Bachelor Arbeit als Betreuer unterstützt hat. Zudem danke ich Thorsten Holz, dass er sich als Zweitkorrektor zur Verfügung gestellt hat.

Mein Dank geht ebenso an Gunther Haist und Matthias Luft für das Korrekturlesen meiner Arbeit.

Zudem danke ich der Free Software Foundation, insbesondere den GCC Entwicklern und der GCC-Community, dafür, dass sie solch einen ausgereiften und gut dokumentierten Compiler entwickelt haben und meine Fragen geduldig beantwortet haben.

Ich danke meiner Mutter, durch die ich überhaupt erst studieren konnte und die mich zum großen Teil zu dem gemacht hat, was ich heute bin.

Ganz besonders danke ich Melanie Heck, die mich während der anstrengenden Arbeit immer unterstützt hat und die Ausarbeitung korrekturgelesen hat.

Inhaltsverzeichnis

1 Einleitung **6**
- 1.1 Motivation . . . 6
- 1.2 Aufgabenstellung . . . 7
- 1.3 Ausblick auf die Arbeit . . . 8

2 Beschreibung von Ulix und der Ulix-Hardware **9**
- 2.1 ULIX . . . 9
- 2.2 ULIX-Hardware . . . 10

3 Alternativen für den Compiler **11**
- 3.1 Compiler im Eigenbau . . . 11
- 3.2 Verwendung von Werkzeugen . . . 12
- 3.3 Portierung eines bereits existierenden Compilers . . . 13
- 3.4 Entscheidung für GCC . . . 14

4 GCC **16**
- 4.1 Aufbau und Funktionsweise des `gcc` . . . 16
- 4.2 Aufbau und Funktionsweise des `cc1` . . . 17
  - 4.2.1 Frontend . . . 17
  - 4.2.2 Middleend . . . 18
  - 4.2.3 Backend . . . 18
- 4.3 RTL . . . 20
  - 4.3.1 Wie ist RTL aufgebaut? . . . 20
  - 4.3.2 Machine Description . . . 21

5 GCC-Portierung: Design und Implementation **25**
- 5.1 Wie wird das ULIX-Backend integriert? . . . 25
- 5.2 Machine Description . . . 26
  - 5.2.1 Move Instruction Pattern . . . 28
  - 5.2.2 Push Instruction Pattern . . . 29
  - 5.2.3 Pop Instruction Pattern . . . 30
  - 5.2.4 Add Instruction Pattern . . . 31
  - 5.2.5 Subtract Instruction Pattern . . . 31
  - 5.2.6 Multiplication Instruction Pattern . . . 32
  - 5.2.7 Division Instruction Pattern . . . 33
  - 5.2.8 Modulo Instruction Pattern . . . 34
  - 5.2.9 And Instruction Pattern . . . 35

5.2.10 Or Instruction Pattern . . . 35
5.2.11 Xor Instruction Pattern . . . 36
5.2.12 Not Instruction Pattern . . . 37
5.2.13 Shift Instruction Patterns . . . 38
5.2.14 Cast Instruction Pattern . . . 41
5.2.15 Compare Instruction Pattern . . . 45
5.2.16 Conditional Jump Pattern . . . 46
5.2.17 Unconditional Jump Instruction Pattern . . . 49
5.2.18 Indirect Jump Instruction Pattern . . . 50
5.2.19 Subroutine Instruction Pattern . . . 50
5.2.20 Nop Instruction Pattern . . . 55
5.2.21 Dummy Constraints . . . 55
5.3 Target Description Macros . . . 56
5.3.1 Speicher-Layout . . . 57
5.3.2 Alignment . . . 60
5.3.3 Register-Eigenschaften . . . 61
5.3.4 Stack- und Subroutinen-Eigenschaften . . . 64
5.3.5 Adressierung . . . 68
5.3.6 Meta-Assembler Anweisungen . . . 72
5.3.7 Assembler Formatierung . . . 74
5.3.8 Verschiedenes . . . 76
5.3.9 Dummys . . . 78
5.4 Sonstige Implementationen . . . 80
5.4.1 ulix.c . . . 80
5.4.2 Integration von ULIX als Target . . . 83
5.4.3 Patching der GCC . . . 84

6 Evaluation **86**
6.1 ULIX-GCC Build Prozess . . . 86
6.2 Test-Dateien . . . 86
6.2.1 Test der arithmetischen Befehle . . . 87
6.2.2 Test der logischen Befehle . . . 88
6.2.3 Test der Shift-Befehle . . . 89
6.2.4 Test von Cast-Operationen . . . 90
6.2.5 Test von Vergleichs-Operationen . . . 92

7 Zusammenfassung und Ausblick **98**
7.1 Zusammenfassung . . . 98
7.2 Ausblick . . . 99

1 Einleitung

1.1 Motivation

Ein Betriebssystem in heutigen Maßstäben ist komplexe Software, die man nicht komplett in Assembler programmieren möchte. Deshalb nutzt man für gewöhnlich eine Abstraktion der Maschinensprache: die Hochsprache. Durch den Einsatz einer Hochsprache ist man in den meisten Fällen unabhängig von Details der Hardware. Code lässt sich durch komplexere Sprachkonstrukte wie Bedingungen, Schleifen und Unterprogramme leichter entwickeln und warten. Der so entstandene Code ist im Normalfall auch übersichtlicher, wodurch Fehler leichter zu finden sind.

Auf der anderen Seite lässt sich ein vom Compiler generierter Maschinencode nicht mehr komplett kontrollieren. Gerade auf Betriebssystemebene, in der schon einzelne Maschinenanweisungen ins Gewicht fallen können, ist dies allerdings dringend erforderlich um die nötige Performanz zu erreichen. Um die gewünschte Leistungsfähigkeit der Systemsoftware dennoch zu realisieren, gibt es mehrere Möglichkeiten bzw. Bedingungen:

1. Die Programmiersprache sollte möglichst nah an der Hardware sein
2. Die Programmiersprache sollte es ermöglichen direkt Assembler-Instruktionen im Programmcode einzubinden
3. Der generierte Code muss an kritischen Stellen möglicherweise nachträglich manuell optimiert werden

Wenn 1. erfüllt ist, sollte sich der Quellcode nur minimal vom generierten Programmcode unterscheiden. Dadurch hat man einen höheren Grad an Kontrolle über das Ergebnis. Der generierte Code muss deshalb weniger von Hand optimiert werden.

Durch 2. ist es möglich kritische Abschnitte direkt in Assembler zu programmieren. Das manuelle Optimieren entfällt im Idealfall komplett.

Falls 1. und 2. nicht möglich sind, kann man durch manuelle Optimierung des generierten Assemblercodes (3.) die Performance noch weiter verbessern. Dies ist auch zusätzlich möglich. Da aber das nachträgliche Optimieren komplizierter und aufwändiger ist, sollte hierauf nur als letzte Möglichkeit zurückgegriffen werden.

⇒ Die verwendete Programmiersprache muss daher möglichst hardwarenah sein und soll es ermöglichen direkt Assembler-Code einzubinden. Deshalb ist auch heute, zu Zeiten von objektorientierten Programmiersprachen und Garbage-Collection, die Programmiersprache C noch mit die erste Wahl zur Entwicklung von performancekritischer Software, wie in diesem Fall einem Betriebssystem.

Durch die Verwendung einer höheren Programmiersprache treten allerdings neue Probleme auf. Computer verstehen normalerweise keine höheren Programmiersprachen – wie C – sondern erwarten eine architekturspezifische Maschinensprache. Da die Maschinensprache für das Lesen und Verarbeiten durch den Computer optimiert ist, kann sie von Menschen nur sehr schwer gelesen werden. Außerdem unterscheidet sie sich von CPU zu CPU. Um also in C programmieren zu können, benötigt man ein oder mehrere Programme, die C-Code in Maschinencode übersetzen. Diese Aufgabe kann man grob in zwei Aspekte gliedern:

1. Kompilierung
2. Assemblierung

Die Kompilierung übernimmt der Compiler. Er übersetzt C-Code in CPU-spezifischen Assemblercode. Da für jede Architektur eine eigene Assemblersprache definiert ist, muss jeweils auch ein eigener Compiler geschrieben werden. Es ist ebenso möglich einen bereits existierenden Compiler anzupassen. Da die ULIX-Architektur [16] eine neue, virtuelle Architektur ist, existieren hierzu noch keine Compiler.

Die Assemblierung übernimmt der Assembler. Dieser wandelt den für Menschen lesbaren Assembler-Code in Maschinencode um. Das Assembler-Programm sollte nicht mit dem Assembler-Code verwechselt werden, wenn auch beide mit dem Namen Assembler bezeichnet werden.

1.2 Aufgabenstellung

Meine Arbeit wird den Compiler ausführlich behandeln.

Wie bereits erörtert, gibt es für ULIX noch keinen Compiler und daher kann – ohne Modifikation – kein bereits existierender verwendet werden. Meine Aufgabe besteht also darin, einen Compiler zu programmieren, der C-Code in ULIX-Assembler übersetzt. Doch was genau muss der Compiler können und welche Eigenschaften sind nebensächlich?

Um dieser Frage nachzugehen, ist es erst einmal wichtig etwas über die Zielarchitektur zu wissen. Der Compiler soll Assembler-Code für ULIX erzeugen. ULIX wird emuliert, für dieses virtuelle Betriebssystem zu Forschungszwecken keine physische Hardware vorgesehen ist. ULIX ist ein Betriebsystem mit dazugehöriger Architektur, das von Prof. Dr. Felix Freiling am Laboratory for Dependable Distributed Systems der Universität Mannheim entwickelt wird, um Vorgänge und den Aufbau eines Betriebssystems zu erklären. Damit ULIX für Lehrzwecke sinnvolle verwendet werden kann, erfüllt es bestimmte Bedingungen:

1. Für ULIX existiert eine konkrete, wohl definierte Assemblersprache. Nur dadurch kann das Betriebssystem ausgeführt werden.
2. Der ULIX-Assembler abstrahiert von komplizierten Assemblerbefehlen und -regeln, die für einen auf Performanz optimierten Assembler nötig sind.

3. Der Code ist auf gute Lesbarkeit optimiert. Eine performante Ausführung hat keine hohe Priorität.

Auf Details der ULIX-Architektur werde ich in Kapitel 2 noch eingehen.

Die grundlegenden Aufgaben eines Compilers, muss natürlich auch der ULIX-Compiler erfüllen:

- Er muss korrekten Code erzeugen
- Da ULIX unter Linux programmiert wird muss der Compiler mindestens auf Linux lauffähig sein.
- Die Programmiersprache C muss übersetzt werden können, weil ULIX in C und Assembler programmiert wird.

Unwichtig ist, dass der resultierende Code besonders optimiert und performant ist. Stattdessen sollte der Code gut lesbar sein. Der Compiler wird als reiner Cross-Compiler verwendet und muss selbst nicht auf ULIX lauffähig sein. Dies wäre zudem nicht möglich, da ULIX über keine typische Ein- und Ausgabe und kein normales Dateisystem verfügt. Außerdem werden auch keine Standard Libraries benötigt. Auch der Assembler und der Linker werden nicht Bestandteil meiner Thesis sein. Den Assembler und Linker implementiert Nadine Benedum[13].

1.3 Ausblick auf die Arbeit

Die vorliegende Arbeit befasst sich mit der Entwicklung eines Compilers für ULIX und der damit verbundenen Anpassung des GCC-Backends. Sie ist in 7 Kapitel unterteilt.

Kapitel 1 ist die Einleitung. Hier wird die Motivation und die Aufgabenstellung der Arbeit erläutert.

In Kapitel 2 wird das Betriebssystem ULIX und die dazu gehörige Hardware vorgestellt. Durch die Informationen über das Zielsystem wird klar, was vom Compiler erwartet wird.

Die verschiedenen Herangehensweisen an die Entwicklung eines Compilers für ULIX werden in Kapitel 3 erörtert. Außerdem wird erklärt wieso die Entscheidung auf das anpassen des GCC gefallen ist.

Wie der GCC aufgebaut ist und funktioniert bespricht Kaptitel 4. Zuerst werden die Unterprogramme des GCC vorgestellt. Danach wird auf den eigentlichen Compiler eingegangen. Zusätzlich werden noch die nötigen Datenstrukturen besprochen.

Die eigentliche Implementierung des GCC Backends wird in Kapitel 5 vorgenommen. Es werden Machine Descriptions und Target Description Macros definiert. Zusätzlich wird ein Workaround vorgestellt, der das Backend in den GCC integriert.

Kapitel 6 evaluiert die Ergebnisse des ULIX-GCC. Es werden Test-Dateien kompiliert und die Ergebnisse validiert. Zudem wird auf die Grenzen der Evaluation hingewiesen.

In Kapitel 7 wird nocheinmal die gesamte Arbeit reflektiert und es wird ein Ausblick auf die Weiterentwicklung des Projekts durchgeführt.

2 Beschreibung von Ulix und der Ulix-Hardware

Um einen Compiler zu implementieren muss man detailliertes Wissen über das Zielsystem und dessen Assemblersprache haben. In meinem speziellen Fall sind auch Informationen über das zu kompilierende System, das ULIX Betriebssystem, unablässig. Deshalb werde ich in diesem Kapitel genauer auf ULIX und die von Ralf Hund entworfene ULIX-Architektur [16] eingehen. Dadurch werden viele Design-Entscheidungen, die ich während der Entwicklung des Compilers getroffen habe und auf die ich in Kapitel 5 genauer eingehe, klarer.

2.1 Ulix

ULIX ist ein Betriebsystem von Prof. Dr. Felix Freiling, das in der Universitäts-Lehre eingesetzt werden soll. Mit Hilfe von ULIX sollen die Vorgänge, die in einem modernen Betriebsystem ablaufen, so erläutert werden, dass sie an der Universität im Rahmen einer Vorlesung an die Studenten weitergegeben werden können. Die soll sowohl einfach zu verstehen, als auch technisch korrekt sein. ULIX und die dazugehörige Architektur muss also von der Komplexität real existierender Computerarchitekturen abstrahieren und dennoch so konkret sein, dass es auf einem PC ausgeführt bzw. emuliert werden kann und in Beispielen nicht auf Pseudocode zurückgegriffen werden muss. Diese Einfachheit wird einerseits durch die Hardware, auf der Ulix läuft (siehe 2.2) erreicht, andererseits ist ULIX ein sehr schlankes Betriebssystem, das nur die in der Lehre zu Besprechenden Konstrukte implementiert um nicht zu kompliziert zu werden. ULIX muss auch nicht, anders als bei Betriebssystemen im produktiven Einsatz, besonders performant sein und viele Anwendungsmöglichkeiten bieten, sondern es soll möglichst verständlich die Grundfunktionen eines Betriebssystems erklären. Das gesamte ULIX-Projekt baut auf dem Konzept des „Literate Programming“ auf. Dies trägt zusätzlich zum besseren Verständnis bei. Im Zentrum des Literate Programming steht die Prämisse, dass die so entworfenen Programme in erster Linie für den Menschen und nicht für die Maschine geschrieben werden. Dies wird dadurch erreicht, dass die ausführliche Dokumentation und der Quellcode als Einheit in der Form eines Buches geschrieben werden und dann auch genauso wie ein Buch gelesen werden können. Auch meine Bachelorarbeit und der Compiler sind nach dem Prinzip des „Literate Programming“ entworfen. Eine grundlegende Referenz zu ULIX ist „The Design and Implementation of the Ulix Operating System“[15] von Prof. Dr. Felix Freiling.

2.2 Ulix-Hardware

Die ULIX-CPU ist eine rein emulierte Architektur. Dies ermöglicht es viele komplizierte Konstrukte, die in realen CPUs verwendet werden, zu umgehen.

Die ULIX-CPU definiert einen möglichst kleinen und dadurch überschaubaren Befehlssatz aus 27 Instruktionen. Es existieren lediglich 2 Ausführungsmodi, der „system mode“ und der „user mode“. Der Registersatz besteht aus 16 General Purpose Register, dem Stack Pointer (SP), dem Program Counter (PC) und dem Program Status Word (PSW) mit seinen Unterregistern. Alle Register sind 32 Bit lang und werden auf die gleiche Weise verwendet.

Es existieren drei Adressierungsgrößen:

int 32 Bit

short 16 Bit

byte 8 Bit

Jede der drei Größen kann ohne Einschränkung für jeden Operanden verwendet werden. Es gibt vier Adressierungsarten:

- Konstanten
- Absolute Adressen
- Register
- Relative Adressen

Es ist legitim jede Adressierungsart mit jeder Adressierunggröße zu kombinieren.

Alle Details zur ULIX-Architektur sind in [16] zu finden.

3 Alternativen für den Compiler

Es gibt viele Möglichkeiten einen Compiler für eine bestimmte Architektur zu entwickeln. Mein Ziel ist es den Weg zu finden, der es ermöglicht alle Anforderungen komplett und gleichzeitig effektiv zu erfüllen. Hier werde ich die verschiedenen Herangehensweisen erörtern.

3.1 Compiler im Eigenbau

Eine Möglichkeit einen Compiler zu implementieren ist es, ihn von Grund auf selbst zu programmieren. Dies hat den Vorteil, dass jedes Detail genau definiert und angepasst werden kann. Somit sind auch Compiler für sehr spezielle Architekturen und Programmiersprachen möglich. Die ULIX-CPU ist allerdings nicht sehr untypisch oder speziell. Ganz im Gegenteil: die ULIX-CPU ist sogar besonders einfach und hält sich an bewährte Strukturen anderer Architekturen. Ein weiter Vorteil den die Entwicklung „from scratch“ bietet ist, dass man Teile des Compilers, die nicht benötigt oder vielleicht sogar nicht erwünscht sind, einfach nicht implementiert und dass es keine Teile im Code gibt die unklar sind, da man schließlich alles selbst implementiert hat.

Was sind also die Funktionen die ein komplett selbstentwickelter Compiler mindestens bieten muss?

- Frontend
  - Lexikalische Analyse
  - Syntaktische Analyse
  - Semantische Analyse
- Backend
  - Codeerzeugung

Grob kann man sagen, dass das Frontend den Quelltext, der in einer höheren Programmiersprache vorliegt, einliest und analysiert, während das Backend diese Analysen verwendet um Assembler-Code zu erzeugen. Schon allein das Design des Compilers von Grund auf ist ein sehr großer Aufwand und wirft viele Fragen auf:

- Wie kann man die Analysen effektiv und resistent implementieren?
- Wie greifen die verschiedenen Analysen ineinander?
- Wie soll die interne Datenstruktur aussehen?

Ein schlüssiges und gut umzusetzendes Design eines Compilers würde wahrscheinlich schon den Rahmen dieser Bachelorarbeit sprengen. Und die Teile die man nicht implementieren müsste wiegen den Mehraufwand nicht im geringsten wieder auf. Zudem besteht bei einem komplett selbst entwickelten System dieser Größenordnung immer die Gefahr viele, schwerwiegende Fehler zu machen, die auch andere schon gemacht haben. Man müsste also das Rad neu erfinden.

Abschließend lässt sich also sagen, dass der Aufwand den kompletten Compiler selbst zu entwickeln für eine Bachelorarbeit zu groß ist. Außerdem gibt es schon viele Tools die z.B. die lexikalische Analyse übernehmen.

3.2 Verwendung von Werkzeugen

Die nächste Möglichkeit ist also die Verwendung fertiger Tools, die Teilaufgaben des Compilers übernehmen. Mögliche Kandidaten hierfür sind unter anderem die lexikalische Analyse und die syntaktische Analyse.

Programme die eine lexikalische Analyse durchführen nennt man *Lexer* oder *Scanner*. Die Aufgabe des Lexers ist es den Quellcode einzulesen und in Token zu zerlegen. Ein Programm, das diese Aufgabe übernimmt ist z.B. Flex[4].

Für die syntaktische Analyse ist ein Parser zuständig. In der syntaktischen Analyse wird überprüft, ob die Syntax des eingelesenen Programms valide ist. Dafür muss der Parser natürlich die Syntax der Quellsprache kennen. Ein Parser Generator, der u.a. Parser für diesen Zweck generieren kann ist Bison[1].

Beide Programme funktionieren aber auch nicht „out-of-the-box“ sondern müssen erst noch angepasst werden. Gerade bei Bison ist das viel Arbeit. Es ist außerdem sehr wichtig, dass man weiß welche Eingabe ein Werkzeug erwartet und wie die Datenstruktur aussieht die es ausgibt. Denn mit der Ausgabe muss weitergearbeitet werden und es muss möglicherweise in die richtige Form für das nächste Werkzeug gebracht werden.

Gegenüber dem „Compiler im Eigenbau“ spart man sich viel Arbeit und grundlegende Aufgaben, wenn man Tools für die einzelnen Schritte des Compilers verwendet. Wenn die Werkzeuge konfiguriert und an die Bedürfnisse angepasst sind leisten sie gute Arbeit. Dennoch ist die Einarbeitungszeit in die unterschiedlichen Programme sehr hoch und gerade dadurch, dass man selbst für das richtige Zusammenarbeiten sorgen muss kann es leicht zu Fehlern kommen die schwer zu finden sind. Viele Teile des Compilers müssen auch bei der Verwendung von Werkzeugen noch komplett selbst programmiert werden, wie beispielsweise große Teile der Codeerzeugung.

Auch beim Frontend, das nur C lesen können muss und in vielen Compilern schon implementiert ist, muss noch sehr viel angepasst werden. Hier bietet also ein Compiler der schon „von Haus aus“ C lesen kann einen großen Vorteil, da bei diesem nur noch das Backend angepasst werden muss.

3.3 Portierung eines bereits existierenden Compilers

Die dritte Möglichkeit ist also die Portierung eines bereits existierenden Compilers auf die ULIX-Architektur. Dieser Compiler muss bestimmte Mindestbedingungen erfüllen:

1. Erzeugung stabilen Codes, muss also schon entsprechen ausgereift sein.
2. Quellen müssen offen sein (Open-Source). Denn sonst ist es nicht ohne weiteres möglich den Compiler anzupassen.
3. Möglichkeit C als Eingabesprache zu verwenden
4. Gute Dokumentation
   Ohne eine gute Dokumentation wird das Anpassen des Compilers sehr aufwändig, da zuerst der Quellcode des Compilers gelesen und verstanden werden muss bevor eigene Funktionen implementiert werden können. Außerdem zeugt eine gute Dokumentation vom Reifegrad des Compilers.
5. Modularität (mindestens Aufteilung in Frontend und Backend)
   Der Compiler muss eine klare Trennung zwischen Front- und Backend aufweisen, damit nur eine Anpassung des Backends nötig ist. Wenn keine klaren Grenzen vorhanden sind ist es nur schwer möglich die Codeerzeugung zu ändern ohne den gesamten Compiler umzuschreiben.
6. Gute Portierbarkeit
   Im Compiler sollte schon vorgesehen sein, dass er auf verschiedene Architekturen portiert werden kann. Es sollte also eine Art Strategie geben, wie der Compiler anzupassen ist, damit er Code für eine neue Architektur erzeugt. Dadurch wird der Aufwand vertretbar.
7. Lauffähig auf Unix-artigen Systemen
   Da ULIX auf mindestens Linux compiliert wird, muss auch der Compiler auf Unix-artigen Betriebssystemen lauffähig sein. Die ermöglicht es mit hoher Wahrscheinlichkeit auch den Compiler auf Windows mithilfe von Cygwin[2] und auf Mac OS laufen zu lassen.
8. Funktionalität als Cross-Compiler
   Der ULIX-Compiler kann nicht selbst auf ULIX laufen, da ULIX nicht die nötigen Voraussetzungen für einen Compiler erfüllt. Auch muss Ulix zuerst einmal compiliert werden, bevor man darauf Programme laufen lassen könnte.

Zusätzlich zu den Bedingungen die der Compiler unbedingt erfüllen muss gibt es auch noch Funktionen die erwünscht, aber nicht zwingend erforderlich sind.

1. Der Compiler sollte aktuell noch weiterentwickelt werden.
   Bei auftretenden Problemen gibt es so Ansprechpartner zu eventuellen Compilerfehlern. Sollten diese das ULIX-Betriebssystem betreffen, kann man auf einen Bugfix hoffen.

2. Mehrere Eingabesprachen sollten möglich sein.
   Wenn der Compiler neben C noch weitere Eingabesprachen unterstützt wäre das von Vorteil. Die Studenten sollen vielleicht Teile des Betriebssystems austauschen können, um die Auswirkungen auszuprobieren. Wenn sie diese selbst programmierten Teile nicht in Assembler oder C entwickeln wollen, hätten sie so auch noch die Möglichkeit andere Sprachen zu verwenden.

3. Es sollte optional möglich sein Debugging-Informationen in den Assemblercode zu integrieren
   Dies wäre ein durchaus brauchbares Feature, das die Lesbarkeit des Codes, später im Emulator, erhöhen könnte. Dadurch wäre es auch möglich, wie in einem Debugger, den Quellcode verknüpft mit dem Assemblercode, zu debuggen.

4. Lauffähigkeit auf vielen Architekturen und Betriebssystemen
   Wenn der Compiler auf vielen verschiedenen Architekturen und Betriebssystemen lauffähig ist kann er auch von mehr Leuten ohne großen Aufwand verwendet werden. Dies kommt auch wieder den Studenten, die ULIX und den Compiler verwenden zu Gute.

3.4 Entscheidung für GCC

Unter den Open Source Compilern, die die Mindestanforderungen erfüllen habe ich mich für den GCC[7] entschieden. (Obwohl inzwischen GCC für die *GNU Compiler Collection* steht, werde ich trotzdem „der GCC“ schreiben, wie in: der *GNU C Compiler*. *GNU C Compiler* war der ursprüngliche Name des GCC, als nur die Programmiersprache C unterstützt wurde. Da „der GCC“ für mich gewohnter klingt und ich mich meistens auch nur auf den Compiler, nicht die ganze Collection, beziehe, bevorzuge ich diese Bezeichnung.) Ich möchte meine Entscheidung anhand der oben genannten Punkte erläutern.

GCC erzeugt stabilen und zuverlässigen Code. Nicht umsonst ist GCC der Standard-Compiler in den meisten Linuxsystemen, BSD, Mac OS usw. Man kann also von einem sehr hohen Reifegrad ausgehen.

Der GCC wurde unter der GPLv3[8] veröffentlicht und ist somit Open Source. Dadurch habe ich die Möglichkeit den Quellcode zu verwenden und an meine Bedürfnisse anzupassen. Meinen eigenen Quellcode muss ich dann aber auch unter der GPL veröffentlichen.

GCC stand ursprünglich für „GNU C Compiler“ und unterstützt damit C als Eingabesprache. Inzwischen werden aber zusätzlich viele andere Sprachen wie z.B. C++, Objective-C, Fortran, Java und Ada unterstützt. Und genauso wie ich ein eigenes Backend für den GCC schreibe ist es auch möglich ein neues Frontend für den GCC zu programmieren, wodurch dann neue Eingabesprachen unterstützt werden.

Der GCC verfügt mit dem „GCC Internals Manual“[6] über eine 536 Seiten (Stand: Version: 4.3.0 als pdf) umfassende Dokumentation der inneren Abläufe des GCC und deren Konstrukte. Zusätzlich ist auch der meiste Quellcode gut dokumentiert. Die

Informationen, die sich hier nicht finden lassen bekommt man durch die große GCC-Community, z.B. über die GCC-Mailinglisten oder in vielen Foren.

Der GCC ist Modular. Es existieren komplett von einander getrennt bearbeitbare Frontends und Backends. Für einen GCC, der ULIX-Assembler generiert muss also wirklich nur das Backend angepasst werden. Das Parsen und bearbeiten von C unterstützt der GCC also ganz ohne mein Zutun.

Es ist ohne weiteres möglich den GCC auf eine neue Architektur zu portieren. Der GCC wurde extra so entworfen, dass eine Portierung möglichst einfach und ohne tiefen Eingriff ins System möglich ist. Für jede Architektur gibt es mindestens eine Datei mit *Machine Descriptions* und eine Datei, die Architekturspezifische Makros definiert. Zusätzlich dazu muss eigentlich fast nur noch die Architektur im GCC verankert werden. In den schon vorhandenen Dateien des GCC muss außer dieser Verankerung nichts angepasst werden. Die Portierung ist also gut möglich und im GCC Internals Manual[6] auch gut dokumentiert. Dass der GCC für viele Systeme Code erzeugen kann (z.B. i386 und Dialekte, arm, arc, vax, m68k und viele weitere) zeugt schon von seiner guten Portierbarkeit.

Der GCC ist unter vielen Linux- und Unix-artigen Systemen der Standardcompiler. Außerdem ist er auch unter Windows und Mac OS lauffähig. Damit sollte also auch der angepasste ULIX-GCC auf Linux u.a. lauffähig sein.

Der ist GCC sowohl als nativer als auch als Cross-Compiler verwendbar.

Damit wäre gezeigt, dass der GCC alle Mindestanforderungen sehr gut erfüllt und noch darüber hinaus viele nützliche Eigenschaften enthält. Auch die optionalen Voraussetzungen an den Compiler für ULIX deckt der GCC alle ab.

Außer dem GCC wären auch weitere Open Source Compiler in Frage gekommen, die aber nicht alle Bedingungen oder nicht alle optionalen Anforderungen erfüllt hätten. Beispielhaft seien hier *lcc*[14], *Open Watcom C/C++ Compiler*[11] und der pcc[12] genannt.

4 GCC

In diesem Kapitel werde ich mich näher mit dem GCC und seiner Funktionsweise auseinandersetzen.

GCC stand ursprünglich für „GNU C Compiler“. Da heute aber sehr viel mehr Programmiersprachen als C unterstützt werden wurde GCC in „GNU Compiler Collection“ umgedeutet. Wie bereits auf Seite 14 erwähnt werde ich die Bezeichnung „der GCC“, wie der „GNU C Compiler“ verwenden.

Der GCC ist als Teil des GNU Projekts [9] am 22. März 1987 von Richard Stallman veröffentlicht worden. Der GCC ist unter der GPL, der GNU General Public License, veröffentlicht und somit Open Source. Seit seiner Veröffentlichung ist der GCC von Programmierern weltweit sehr stark weiterentwickelt worden.

4.1 Aufbau und Funktionsweise des `gcc`

Die wichtigen Fragen die sich für mich im Rahmen meiner Bachelorarbeit stellen sind: Aus welchen Teilen ist der GCC aufgebaut? Was ist deren Funktion? Welche Teile muss ich für einen Ulix-GCC anpassen und wie? Wie sehen die verwendeten Datenstrukturen aus?

Die Hauptbinary des GCC ist gcc. gcc ist der „Compiler Driver“. Aufgabe der gcc-Binärdatei ist es die einzelnen Unterprogramme des GCC aufzurufen, die nötigen Parameter zu übergeben und die Ausgaben wie gewünscht umzuleiten. Um zu verstehen, wieso überhaupt ein Compiler Driver benötigt wird, muss man erst einmal wissen, dass der GCC viele einzelne Programme verwendet um vom Quellcode zur ausführbaren Datei zu kommen.

Ich werde den Programmablauf anhand einer C-Datei einmal durchspielen:

1. Der **Compiler Driver** `gcc` wird mit dem Pfad zur Quelldatei als Parameter aufgerufen:

   ```
   gcc source.c
   ```

   Jetzt ist es die Aufgabe des gcc-Programms alle nötigen Programme aufzurufen, damit am Ende ein Ausführbares Programm herauskommt.

2. gcc ruft zuerst einmal den **Präprozessor** auf und übergibt diesem den Quelltext. Die Aufgabe des Präprozessors ist es alle Makros und Präprozessoranweisungen zu ersetzen. In diesem schritt werden auch alle `#include`-Anweisungen ersetzt, so dass die Deklarationen in den Headern direkt in die Quelldatei geschrieben werden. Der Präprozessor ersetzt alle Präprozessor-Anweisungen, die üblicherweise mit # beginnen. Die Ausgabe ist also ein Quelltext, der nur noch reines C ohne Makros usw. enthält.

3. Die Ausgabe des Präprozessors wird als Input für den eigentlichen **C-Compiler** `cc1` verwendet. Dies ist der Teil, der für meine Bachelorarbeit interessant ist. Ich werde auf den C-Compiler ab Kapitel 4.2 noch näher eingehen. Die Aufgabe des C-Compilers ist es den Quellcode, der in reinem C vorliegt, in die Assembler-Sprache der Zielarchitektur zu übersetzen. Dafür sind einige Schritte nötig. `cc1` gibt also Assembler-Code als Text aus. Im Gegensatz zu C ist der Assembler architekturspezifisch.

4. Der Assembler-Code wird an das **Assembler**programm übergeben. Dieses wandelt den in Text vorliegenden Assembler-Code in Maschinen-Code für die ULIX Architektur um. Die Ausgabe des Assemblers ist normalerweiße eine Objektdatei in Binärform.

5. Der **Linker** ist das letzte Programm in der Kette das von `gcc` aufgerufen wird. Der Linker bindet die einzelnen Programmmodule, wie beispielsweise Objektdateien und Libraries, zu einem ausführbaren Programm oder einer Library zusammen. In unserem Fall haben wir also mit diesem Schritt eine Ausführbare Datei erhalten.

Da sich meine Bachelor-Arbeit mit dem Compiler beschäftigt ist für mich hauptsächlich der C-Compiler `cc1` interessant. Den Präprozessor werde ich implizit auch verwenden. Da aber der Präprozessor von der Architektur, für die entwickelt wird, unabhängig ist und für C als Eingabesprache einheitlich ist, kann der Präprozessor des GCC unverändert verwendet werden.

Das Assemblieren und Linken wird bei ULIX nicht nach diesem typischen Schema ablaufen. Am Ende wird nicht eine ausführbare Datei entstehen, sondern ein Speicher-Image, das allen Code des Betriebssystems enthält und in den Speicher des emulierten ULIX-Computers gelegt wird. Nähere Informationen zum ULIX-Assembler findet man in der Arbeit von Nadine Benedum[13].

4.2 Aufbau und Funktionsweise des `cc1`

cc1 ist der C-Compiler des GCC. Da es sehr komplex ist ein Programm zu kompilieren, ist auch der Compiler zusätzlich modularisiert. Die wichtigste Einteilung ist die in Frontend und Backend. Das **Frontend** erledigt die sprachspezifischen Aufgaben, ist also für jede Eingabesprache spezifisch. Es verarbeitet den Quelltext. Auf der anderen Seite ist das **Backend** für die architekturspezifischen Aufgaben zuständig. Es generiert den Assemblercode. Für jede CPU für die Code generiert werden kann, muss also ein angepasstes Backend existieren. Zusätzlich zu Frontend und Backend gibt es auch noch Teile des Compilers, die nicht in dies Einteilung passen und zwischen Front- und Backend ausgeführt werden. Dieser Teil wird beim GCC Middleend genannt.

4.2.1 Frontend

Die Aufgabe des Frontends ist es den Quellcode zu parsen, lexikalisch, syntaktisch und semantisch zu analysieren und in ein Format zu bringen, das vom Backend verarbeitet

werden kann. Die Phase in dem das Frontend aktiv ist wird auch **Analysephase** genannt. Normalerweise durchläuft der Compiler 3 Phasen der Analyse. Dies ist aber nicht zwingend erforderlich, wird meist aber eingehalten:

1. **Lexikalische Analyse**
   Während der lexikalischen Analyse wird der Quelltext in zusammengehörige Token aufgeteilt

2. **Syntaktische Analyse**
   Während der syntaktischen Analyse wird geprüft ob die Syntax des Quelltextes valide für die Eingabesprache ist. In dieser Phase wird ein Syntaxbaum generiert. Ist die Syntax nicht korrekt wird ein Syntaxfehler ausgegeben.

3. **Semantische Analyse**
   Während der semantischen Phase werden weitere Bedingungen an den Quellcode überprüft, die über die Syntax hinausgehen. Hier wird z.B. getestet ob richtige Datentypen verwendet werden. Die Knoten des Syntaxbaums werden hier noch mit Attributen versehen.

Durchläuft der Quellcode alle Analysephasen ohne Fehler entsteht somit ein Baum. Dieser Baum muss weiterhin in ein Format gebracht werden, das vom Backend erwartet und weiterverarbeitet werden kann. Diese Phase wird **Gimplification** genannt, da dabei ein GIMPLE-Tree[5] erstellt wird. Dieser GIMPLE-Tree ist komplett Prozessor- und Sprachunabhängig. Bevor der GIMPLE-Tree verwendet wurde war die Baumpräsentation des Quellcodes nicht komplett unabhängig von der verwendeten Sprache und Architektur.

4.2.2 Middleend

Das Middleend ist ein Teil des GCC, das in die Aufteilung in Backend und Frontend nicht hineinpasst. Im Middleend werden Optimierungen im GIMPLE-Tree durchgeführt. Da der GIMPLE-Tree sowohl von der verwendeten Eingabesprache, als auch von der Zielarchitektur unabhängig ist gehören diese Optimierungen also weder zum Front- noch zum Backend. Zusätzlich zur Optimierung im Middleend gibt es auch eine Verfeinerung des Codes die Architekturabhängig ist und deshalb ins Backend gehört.

Optimierungen, die im Middleend verwendet werden sind unter anderem das Entfernen unerreichbarer Codeteile (Dead Code Elimination), Full- und Partial Redundancy Elimination und Sparse conditional constant propagation.

Optimierung hat beim Ulix-Compiler keine Priorität, deshalb wird keine architekturabhängige Optimierung im Backend implementiert. Dadurch lässt sich im generierten Assemblercode sehr gut die Optimierung des Middleends erkennen, wenn man ein und denselben Quellcode mit und ohne Optimierung compiliert.

4.2.3 Backend

Das Backend ist für die Generierung des Assemblercodes aus dem GIMPLE-Tree zuständig. Alle Teile des Compilers die von der Zielarchitektur abhängen sind Teil des

Backends. Daher müssen nur Teile des Backends angepasst werden, um den GCC auf ULIX zu portieren.

Auch das Backend hat wieder verschiedene Phasen die es durchläuft, bis es aus dem GIMPLE-Tree Assemblercode generiert hat. Es ist noch wichtig zu wissen, dass im Backend **Register Transfer Language (RTL)** als Repräsentation des zu kompilierenden Programms verwendet wird. Auf RTL wird in Kapitel 4.3 noch näher eingegangen, da das Verständnis von RTL essentiell für die Entwicklung eines GCC-Backends ist.

Von GIMPLE zu RTL

Um aus dem GIMPLE-Tree die passende RTL-Repräsentation zu generieren benötigt GCC eine für die Ziel-CPU passend **Machine Description**. Jede Zielarchitektur benötigt eine eigene Machine Description. Dadurch wird auch schon klar, dass die erzeugte RTL nicht mehr prozessorunabhängig ist.

Die Machine Description ist in einer Datei mit dem Namen der Architektur und der Endung .md definiert. Für ULIX gibt es also die Datei `ulix.md`. Hierin sind **instruction pattern** definiert. Ein „instruction pattern“ kann einen Namen haben. Dies ist aber nicht für jedes „instruction pattern“ zwingend erforderlich. Im GCC fest einkodiert sind bestimmte Standardnamen, die für die Übersetzung von GIMPLE zu RTL verwendet werden. Beispielsweise gibt es den Standardnamen: `movsi`. Dies steht für „move Single Integer“, also das Kopieren einer Ganzzahl, die genau ein Wort breit ist. Kommt also im C-Programm ein Befehl vor, in dem ein Integer kopiert werden muss, so wird ein entsprechender GIMPLE-Teilbaum erzeugt. Im Backend sucht GCC für diesen Teilbaum das entsprechende „instruction pattern“ mit dem Namen `movsi`. In diesem Pattern steht in was für RTL-Muster dieser Teilbaum übersetzt werden soll. Im GCC Internals Manual[6] sind eine ganze Reihe von Standard-Namen für diese Ersetzung beschrieben. Nur ein Teil der hier beschriebenen „instruction pattern“ müssen zwingend definiert sein. Der Rest sollte nur implementiert werden, wenn die Zielarchitektur über entsprechende Assembler-Instruktionen verfügt. Mit Hilfe dieser Standard-Namen und der dazu gehörigen „instruction pattern“ kann der GCC den gesamten GIMPLE-Tree in RTL umwandeln. Der somit erzeugte RTL-Code ist schon sehr nahe an Assembler-Code. Die RTL Instruktionen sind genauso elementar wie der später zu erzeugende Assembler.

RTL-Optimierung

Nachdem das gesamte Programm in RTL vorliegt kann der RTL-Code architekturspezifisch optimiert werden. Diese Optimierung wird auch Aufgrund der „Machine Description“ vorgenommen. Es ist möglich die verschiedenen Assemblerbefehle zu gewichten, wodurch es dem Compiler möglich wird teure, also langsame Befehle durch besonders günstige, also schnelle Befehle zu ersetzen. Damit der Compiler weiß, wie ein Befehl in mehrere andere Befehle aufgeteilt oder wie mehrere Befehle zu einem anderen Befehl zusammengezogen werden können müssen diese Ersetzungen auch in der „Machine Description“ definiert werden.

Da der für ULIX generierte Code aber nicht besonders optimiert sein muss, habe ich sowohl auf die Gewichtung als auch auf andere Möglichkeiten der architekturabhängigen Optimierung verzichtet.

Reload-Phase

Wie bereits erwähnt ist der aus dem GIMPLE-Tree erzeugte RTL-Code bereits sehr nah am Assembler. Ein wesentlicher Unterschied ist der, dass im RTL nicht nur mit realen Registern („Hard Register“) gearbeitet wird, sondern auch mit virtuellen „Pseudo Registern“. Bisher wird in der RTL-Interpretation einfach davon ausgegangen, dass die CPU beliebig viele Register zur Verfügung stehen hat. Alle Register, die eine höhere Nummer haben als die höchste Registernummer der CPU, sind Pseudo-Register. In der Reload-Phase werden den Pseudo-Registern bei Bedarf reale Register zugewiesen. Das dafür wichtige genauere Wissen über die Eigenschaften der Zeilarchitektur wird in „Target Description Macros“ beschrieben. Diese Makros und die dafür benötigten Funktionen werden in Dateien mit dem Namen der Architektur und der Endung .h für die Makros und .c für die Funktionen, definiert. Für ULIX wurden also die Dateien `ulix.h` und `ulix.c` implementiert. Welche Makros dies genau sind wird in Kapitel 5.3 besprochen.

Von RTL zu Assembler

Nach der Reload-Phase hat man also RTL-Code in dem nur noch „Hard Register“ vorkommen. Jetzt muss nur noch für jeden RTL-Befehl oder jede Gruppe von RTL-Befehlen der passender Assembler-Befehl erzeugt werden. Hierfür werden wieder die „instruction pattern“ in der „Machine Description“ benötigt. In den „instruction pattern“ ist nämlich definiert, was für Assembler-Code aus bestimmtem RTL-Code generiert wird. Damit diese Phase funktioniert, muss für jedes RTL-Muster, dass in den Phasen vorher erzeugt worden ist ein passendes „instruction pattern“ definiert sein, dass den RTL-Ausdruck in Assembler umwandelt.

Wenn alles erfolgreich funktioniert hat wird jetzt das gesamte Programm in Assembler ausgegeben.

4.3 RTL

„Machine Descriptions“ und die darin verwendete RTL, also „Register Transfer Language“ sind die nötigen Datenstrukturen, die für das Entwickeln eines GCC Backends nötig sind. Deshalb ist es essentiell, dass klar ist, wie RTL aussieht, wie es aufgebaut ist und wie es u.a. in den „Machine Descriptions“ verwendet wird.

4.3.1 Wie ist RTL aufgebaut?

RTL ist die Sprache in der das zu kompilierende Programm im Backend des GCC verwendet wird. Es gibt zwei Formen in der RTL vorkommt. Im Compiler wird RTL

als RTL-Baum repräsentiert, da dieser besser verarbeitet werden kann. Da diese Baum-Form aber für den Menschen ohne Hilfsmittel weder gut zu lesen, noch zu schreiben ist, gibt es auch eine Entsprechung in Textform. Diese Form von RTL ist angelehnt an die Syntax von Lisp.

Ich möchte anhand eines Beispiels näher auf die RTL-Syntax eingehen:

21 ⟨*RTL-Beispiel* 21⟩≡

```
(set (reg:SI 1)
     (minus:SI (reg:SI 0)
               (const_int 4)))
```

Der äußerste Befehl ist ein `set`, die Syntax hierfür lautet `(set <Operand1> <Operand2>)`. Dies ist ein Befehl mit Seiteneffekt, d.h. durch den `set`-Befehl werden Werte in Operanden geändert, also Operand1 wird auf den Wert von Operand2 gesetzt.

Der Operand1 ist `(reg:SI 1)`, also das Register mit der Nummer 1 und Größe SI - „Single Integer“, also eine Ganzzahl mit der Breite eines Wortes. Die Syntax von `reg` ist dementsprechend: `(reg:<mode> <register-number>)`, wobei mode die Art und Größe des Registers und register-number die Nummer des Registers bezeichnet.

Operand2 ist `(minus:SI (reg:SI 0) (const_int 4))`. Auch der `minus`-Befehl hat genau wie der `reg`-Befehl einen Modus angegeben. Dadurch wird das Ergebnis des `minus`-Ausdrucks ein „Single Integer“. `minus` und andere arithmetische Befehle haben implizit keine Seiteneffekte in RTL. Das heißt, keiner der Operanden ändert durch den `minus` Befehl, sondern der `minus`-Ausdruck wird nur ausgewertet. Will man einen Seiteneffekt erreichen muss man ihn explizit mit `set` hervorrufen. Die Syntax von `minus` ist `(minus:<mode> <OperandA> <OperandB>)`.

OperandA ist `(reg:SI 0)`, also wieder ein Register.

OperandB ist `(const_int 4)`. const_int bezeichnet einen konstante Ganzzahl, hier die 4.

Alles zusammengefasst, von innen nach außen, geschieht hier also folgendes: Vom Wert des Registers 0 wird der konstante Wert 4 subtrahiert und dann in Register 1 geschrieben.

Es gibt noch sehr viel mehr RTL-Instruktionen, die im GCC Internals Manual[6] nachzulesen sind.

4.3.2 Machine Description

Jetzt, da die Struktur von RTL klar ist stellt sich die Frage, wie der GIMPLE-Tree in RTL umgewandelt wird. Dafür wird das „Machine Description“-File verwendet. Hierin ist definiert, wie bestimmte Konstrukte der Tree-Darstellung in passende RTL-Strukturen übersetzt werden können. Für diesen Übersetzungsschritt werden zwei unterschiedliche Pattern verwendet:

1. Instruction Pattern
2. Expander Definitions

Instruction Pattern

Instruction Pattern, die mit dem Schlüsselwort `define_insn` gekennzeichnet sind bestehen aus 4 erforderlichen Teilen und einem optionalen. Wobei zu beachten ist, dass die erforderlichen Teile auch leer sein dürfen. Die 5 Teil sind:

1. Der **Name** des „Instruction Pattern“ (`insn`). Dieser kann auch leer sein. Um aus einem `insn` RTL zu generieren wird aber ein nicht-leerer Name benötigt. Soll das `insn` schon beim Übersetzen des GIMPLE-Trees zu RTL verwendet werden, so muss das `insn` einen der fest definierten Standard Namen haben. (siehe [6]).

2. Das **RTL-Template** ist ein Vektor aus unvollständigen RTL-Instruktionen. Es zeigt, wie der RTL-Ausdruck aussehen soll. Es ist unvollständig, da die Operanden und teilweise auch noch die Operatoren eingesetzt werden müssen. Um dies zu ermöglichen kommen in dem Template Konstrukte wie `match_operand`, `match_operator` und `match_dup` vor, die bei der RTL-Generierung ersetzt werden.

3. Mit einer zusätzlichen **Bedingung** kann noch weiter eingeschränkt werden, ob ein insn passt oder nicht. Die Bedingung ist ein C-Ausdruck. Falls die Bedingung ein leerer String ist, werden außer dem RTL-Template keine weiteren Bedingungen getestet.

4. Das „Output Template“ gibt an, wie der aus dem RTL-Konstrukt generierte Assembler-Code aussieht. Ein % gefolgt von einer Zahl gibt an, wo die Operanden eingesetzt werden müssen. Als Output Template kann auch ein Stück C-Code angegeben werden, das den Assembler generiert.

5. Ist eine optionale Liste von Attributen.

Wie man sieht kann ein Instruction Pattern mehrere Aufgaben erfüllen. Einerseits ist es dafür zuständig RTL-Code zu erzeugen, entweder in der Umformung von GIMPLE zu RTL oder in Expander Definitions. Dies ist nur möglich wenn das Pattern einen Namen hat. Die zweite Aufgabe ist es, das RTL in der letzten Phase des Backends in Assembler umzuwandeln.

Um den Aufbau eines Instruction Pattern zu verdeutlichen werde ich hier ein Beispiel besprechen:

22 ⟨*Instruktion Pattern* 22⟩≡

```
(define_insn "movsi"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (match_operand:SI 1 "general_operand" ""))]
  ""
  "move   int\\t%0, %1"
)
```

Jedes Instruction Pattern wird durch Klammern eingeschlossen. Direkt nach der öffnenden Klammer steht der Bezeichner `define_insn`, der besagt, dass es sich um ein Instruction Pattern handelt.

Nach dem Bezeichner folgt als String, also in Anführungszeichen eingeschlossen, der Name des `insn`. In diesem Fall trägt das `insn` den Standard Namen `movsi`. Dieser Name wird beim Umwandeln von GIMPLE zu RTL dazu verwendet um einen SI-Wert („Single Integer"), also ein ganzzahligen Wert mit der Länge eines Wortes, in eine Adresse zu schrieben. Kommt also im GIMPLE-Tree solch eine Anweisung vor, so sucht der GCC nach diesem `insn`. Wenn die Operatoren passen und die Bedingung erfüllt ist wird mithilfe des Templates RTL generiert.

Nach dem Namen folgt das RTL-Template. Es ist in eckige Klammern eingeschlossen. Es enthält unvollständigen RTL-Code. In diesem Beispiel ist es eine `set` Instruktion und die Operanden sind `match_operand`-Konstrukte. Ein `match_operand` ist folgendermaßen aufgebaut: `(match_operand:<mode> <number> <predicate> <constraint>)` **mode** gibt die Art und Größe des Operanden an, hier SI, also „Single Integer". **number** ist die Nummer, die dem Operanden zugewiesen wird. Diese Nummer wird benötigt um später mit dem Operanden zu arbeiten. Das **predicate** ist eine Bedingung, die außer dem mode erfüllt werden muss, damit der Operand passt. Wird das predicate nicht erfüllt, so passt das gesamte Template nicht. Das **constraint** gibt an, ob das Operand möglicherweise in ein anderes register geladen werden muss oder welche Speicheradresse ideal ist. Der Unterschied zwischen predicate und constraint ist, dass dass predicate entscheidet, ob ein Operand passt. Das constraint spielt für diese Entscheidung keine Rolle. Das constraint spielt in vielen Entscheidungen erst dann eine Rolle wenn der Operand passt. In Ulix können alle Register und alle Speicheradressen gleich verwendet werden. Es besteht auch kein Vorteil, wenn man beispielsweise ein Register, statt einer Speicheradresse verwendet. Das ist der Grund wieso für Ulix keine constraints verwendet werden. Als predicate kommt meistens `general_operand` zum Einsatz, das jeden gültigen Operanden zulässt. In diesem Beispiel wird beim Operand mit der Nummer 0 das predicate `nonimmediate_operand` verwendet. Dies kontrolliert, dass der Operand kein immediate operand, also kein Operand mit festem Zahlenwert, wie z.B. #42, vorkommt. Die Semantik von set besagt, dass der Wert des Operanden 1 in Operand 0 geschrieben wird. Daher muss Operand 0 eine Adresse oder ein Register sein, und keine Immediate, da in ein Immediate keine Wert geschrieben werden kann.

Die Bedingung des `insn` ist ein leerer String. Es werden also außer dem passenden Template keine weiteren Bedingungen gestellt.

Das Output-Template lautet `move int\\t%0, %1`. %0 wird durch den Operand 0 ersetzt und %1 durch den Operand 1 ersetzt. Angenommen der passende RTL-Code lautet: `(set (reg:SI 4) (reg:SI 9)`, dann ist der Operand 0 also das Register mit der Nummer 4 und der Operand 1 das Register mit der Nummer 9. Dadurch wird %0 durch r4 und %1 durch r9 ersetzt. Der resultierende Assembler-Code lautet also: `move int r4, r9`.

Expander Definitions

Zusätzlich zu den Instruction Pattern gibt es auch noch Expander Definitions, die durch das Schlüsselwort `define_expand` gekennzeichnet sind. Eine Expander Definition besteht aus vier Operatoren:

1. Der **Name** ist, im Gegensatz zu den Instruction Pattern, zwingend erforderlich, da ein Expander nur zur Erzeugung von RTL verwendet werden kann.
2. Das **RTL-Template** ist genauso aufgebaut wie beim `insn`. Es gibt vor, wie der erzeugte RTL-Code aussieht.
3. Die **Bedingung** werden auch hier weitere Anforderungen gestellt, damit die Definition passt.
4. Die **Preparation Statements** enthalten C-Ausdrücke, die ausgeführt werden, bevor das RTL-Konstrukt erstellt wird. Meistens werden hier temporäre Register vorbereitet. Es können aber auch direkt RTL-Ausdrücke generiert werden.

Da für Ulix kaum Expander Definitions verwendet wurden und die Syntax analog zu den Instruction Pattern verläuft werde ich hier kein Beispiel besprechen.

5 GCC-Portierung: Design und Implementation

Das theoretische Wissen aus den vorhergehenden Kapiteln muss nun praktisch angewendet werden um das Backend des GCC auf die Ulix-Architektur zu portieren.

5.1 Wie wird das Ulix-Backend integriert?

Zuerst einmal muss genau geklärt werden welche Dateien alle verändert oder erstellt werden müssen um ein eigenes Backend für den GCC zu entwickeln.

Für jedes Backend existiert ein eigenes Unterverzeichnis mit dem Namen der Architektur in dem Ordner `gcc/config/`. Im Fall von Ulix wird also das Verzeichnis `gcc/config/ulix/` benötigt. In diesem Ordner werden mindestens vier Dateien benötigt:

ulix.h Beschreibt mit Hilfe von Makros die Eigenschaften der Ulix-Architektur. Es wird beispielsweise die Byte-Reihenfolge, die Länge eines Wortes und der Aufbau des Stacks festgelegt.

ulix.md Definiert wie der GIMPLE-Baum in RTL und später RTL wieder in Assembler umgewandelt wird. Auch die prozessorabhängige Peephole-Optimierung, auf die bei Ulix verzichtet wurde, wird hier beschrieben.

ulix.c Enthält Unterprogramme, die in `ulix.h` oder `ulix.md` benötigt werden. Zudem werden auch noch Target Hooks, die weitere Architektur-Details beschreiben, hier implementiert.

t-ulix Enthält Variablen und zusätzliche Regeln für make. Es wird die Makefile-Syntax verwendet. Hier können u.a. zusätzliche Machine Description-Files festgelegt werden.

Für Ulix genügt es eine leere Datei `t-ulix` anzulegen, da die standardmäßig verwendeten Makefiles von GCC genügen.

Damit Ulix überhaupt als Target erkannt wird müssen zusätzlich noch die Dateien `config.sub`, `configure.ac` und `config.gcc` angepasst und Ulix als Zielarchitektur eingetragen werden.

5.2 Machine Description

Nachdem in Kapitel 4.3.2 die Theorie hinter den Machine Descriptions beschrieben worden ist wird jetzt die konkrete Implementierung beschreiben. Alle Machine Descriptions liegen in der Datei `ulix.md`, die folgenden Aufbau hat.

26a ⟨*ulix.md* 26a⟩≡
⟨*Lizenz von ulix.md* 26b⟩
⟨*Definition der Register Nummern* 27a⟩
⟨*Instruction Pattern/Expander Definitions* 27b⟩

Der GCC ist unter der GPL[8] veröffentlicht. Dementsprechend müssen auch alle meine Quelltexte unter der Lizenz der GPL veröffentlicht werden. Am Anfang von `ulix.md` steht der Hinweis auf die Lizenz unter der die Datei veröffentlicht ist. Jede Zeile die mit ; beginnt ist ein Kommentar und hat deshalb keinen Einfluss auf das Programm. Ein ; kann natürlich auch an einer anderen Stelle einer Zeile vorkommen. Dann ist alles nach dem ; bis zum Ende der Zeile ein Kommentar.

26b ⟨*Lizenz von ulix.md* 26b⟩≡ (26a)

```
;;- Machine description for Ulix for GNU compiler
;;  Copyright 1991, 1993, 1994, 1995, 1996, 1996, 1997, 1998, 1999, 2000,
;;  2001, 2002, 2003, 2004, 2005, 2006, 2007 Free Software Foundation, Inc.
;;  Ulix-Description by Balthasar Biedermann (balthasar@biedermann.es).

;; This file is part of GCC.

;; GCC is free software; you can redistribute it and/or modify it
;; under the terms of the GNU General Public License as published
;; by the Free Software Foundation; either version 3, or (at your
;; option) any later version.

;; GCC is distributed in the hope that it will be useful, but WITHOUT
;; ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
;; or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public
;; License for more details.

;; You should have received a copy of the GNU General Public License
;; along with GCC; see the file COPYING3.  If not see
;; <http://www.gnu.org/licenses/>.

;;- See file "rtl.def" for documentation on define_insn, match_*, et. al.
```

Vor den Instruction Pattern werden zuerst einige Konstanten festgelegt, mit denen später gearbeitet werden kann. Die wichtigste Zuordnung ist die zwischen der Register-Nummer und einem sinnvollen Namen für das Register. Durch diese Zuweisung ist es möglich mit dem Namen, anstatt den Zahlen zu arbeiten. Dies ist vorallem deshalb sinnvoll, da die Ulix-Architektur noch nicht fertig entwickelt war, als ich mit der Arbeit am GCC begonnen habe. Dadurch muss die Änderung von Registernummern

nur an einer einzigen Stelle vorgenommen werden. Der Quelltext wird dadurch auch einfacher zu verstehen sein.

27a ⟨*Definition der Register Nummern* 27a⟩≡ (26a)

```
;;-------------------------------------
;; Constants

;; Register numbers
(define_constants
  [(SP_REGNUM       16)         ; Stack pointer
   (USP_REGNUM      17)         ; User-Stack pointer
   (PC_REGNUM       18)         ; Program counter
   (PSW_REGNUM      19)         ; Program Status Word
   (MODE_REGNUM     20)         ; Execution Mode: 1 = System-Mode, 0 = User-Mode
   (ZERO_REGNUM     21)         ; Zero-Flag
   (C1_REGNUM       22)         ; Wichtig bei Vergleichen
   (C2_REGNUM       23)         ; Wichtig bei Vergleichen
   (IRR_REGNUM      26)         ; Interrupt Request Register
   (IER_REGNUM      27)         ; Interrupt Enable Register
  ]
)
```

Da die nötigen Konstanten definiert sind kann in den Instruction Pattern und den Expander Definitions damit gearbeitet werden.

Es gibt verschiedene Gruppen von `insns`, die definiert werden. Außerdem wird in `ulix.md` noch constraints definiert, die ebenfalls in den Instruction Pattern verwendet werden können.

27b ⟨*Instruction Pattern/Expander Definitions* 27b⟩≡ (26a)

```
;;-------------------------------------
;; Insn patterns
;;

⟨move instructions 28⟩
⟨push instructions 29⟩
⟨pop instructions 30⟩
⟨add instructions 31a⟩
⟨subtract instructions 31b⟩
⟨multiplicate instructions 32⟩
⟨divide instructions 33⟩
⟨Modulo instructions 34⟩
⟨and instructions 35⟩
⟨or instructions 36a⟩
⟨xor instructions 36b⟩
⟨shift instructions 38a⟩
⟨not instructions 37⟩
⟨Operations for up- and down-casts 41⟩
⟨Compare instructions 46⟩
⟨Conditional Jump instructions 47⟩
```

⟨*Unconditional Jump instruction* 49⟩
⟨*subroutine instructions* 50b⟩
⟨*nop instruction* 55b⟩
⟨*indirect jump instruction* 50a⟩
⟨*dummy constraint* 55c⟩

5.2.1 Move Instruction Pattern

Der move-Befehl ist ein essentieller Befehl in fast jedem Assemblercode. Denn auf irgendeine Art und Weise muss es möglich sein Daten von einer Adresse in eine andere zu kopieren. Deshalb muss der move-Befehl mindestens von einer viertel Wortbreite bis zur Wortbreite definiert sein, da es sonst keine Möglichkeit gibt Daten von einer Adresse zu einer anderen zu kopieren. Da in ULIX Integer mit der Breite eines Wortes (32 Bit) die größte elementare Einheit sind, muss kein größerer move-Befehl implementiert sein. Benötigt werden also move-Instruktionen für folgende Modi: QI („Quarter-Integer“, also Integer mit einem Byte Länge), HI („Half Integer“, also Integer mit 2 Byte Länge) und SI („Single Integer“, Integer mit Wortlänge also 4 Byte). Der Standard Pattern Name für den move-Befehl lautet `mov`*m*, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht.

Es muss also jeweils ein `insn` für `movqi`, `movhi` und `movsi` definiert werden.

28 ⟨*move instructions* 28⟩≡ (27b)

```
;; move instructions

(define_insn "movqi"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (match_operand:QI 1 "general_operand" ""))]
  ""
  "move   byte\\t%0, %1"
)

(define_insn "movhi"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (match_operand:HI 1 "general_operand" ""))]
  ""
  "move   short\\t%0, %1"
)

(define_insn "movsi"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (match_operand:SI 1 "general_operand" ""))]
  ""
  "move   int\\t%0, %1"
)
```

Der move-Befehl entspricht genau einem set-Befehl in RTL, wobei der erste Operand das Ziel und der zweite Operand die Quelle des Kopiervorgangs ist. Interessant sind

beim move-Befehl auch noch die Constraints. Für den Ziel-Operanden wird `nonimmediate_operand` als constraint verwendet. Wie der Name schon andeutet erlaubt `nonimmediate_operand` alle Operanden die nicht immediate, also keine konstanten Zahlenwerte, sind. Als Ziel eines Kopiervorgangs darf natürlich nur eine Speicherstelle (Register oder Memory) genutzt werden. Für den zweiten Operanden genügt allerdings der allgemeinere constraint `general_operand`, da jeder gültige Operand, auch ein immediate, kopiert werden darf. Aus diesen ersten drei Instruction Pattern geht auch hervor, dass der Modus QI genau einem `byte`, der Modus HI einem `short` und der Modus SI einem `int` in ULIX-Assembler entspricht.

5.2.2 Push Instruction Pattern

In ULIX existiert eine Push-Instruktion, dementsprechend sollte diese auch im GCC abgebildet werden. Der Stack-Pointer in ULIX zeigt immer auf die nächste freie Stelle auf dem Stack. Da der Stack nach unten wächst funktioniert ein push mit dem Postdekrement des Stack-Pointers. Die push-Instruktion entspricht einem elementaren move-Befehl, wobei das Ziel die Speicheradresse ist, auf die der Stack-Pointer zeigt, deshalb wird auch hier ein RTL-set verwendet. Der Standard Pattern Name für den push-Befehl lautet `push`*m*`1`, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 1 steht in dem Namen, weil genau einen Operand erwartet wird.

29 ⟨*push instructions* 29⟩≡ (27b)

```
;; push instructions

(define_insn "pushqi1"
  [(set (mem:QI (post_dec (reg:SI SP_REGNUM)))
        (match_operand:QI 0 "general_operand" ""))]
  ""
  "push   byte\\t%0"
)

(define_insn "pushhi1"
  [(set (mem:HI (post_dec (reg:SI SP_REGNUM)))
        (match_operand:HI 0 "general_operand" ""))]
  ""
  "push   short\\t%0"
)

(define_insn "pushsi1"
  [(set (mem:SI (post_dec (reg:SI SP_REGNUM)))
        (match_operand:SI 0 "general_operand" ""))]
  ""
  "push   int\\t%0"
)
```

Ein push wird mit nur einem Operanden aufgerufen. Beim set ist der erste Operand beim push immer das Postdekrement des Stack-Pointers. Dies wird in RTL durch

eine Schachtelung erreicht. Zuerst einmal wird mit `(reg:<mode> SP_REGNUM)` auf das Stack-Pointer-Register zugegriffen. SP_REGNUM ist eine Konstante, die auf Seite 27 definiert wurde. Davon wird mit `post_dec` das Postdekrement gebildet. Da dieser Wert ein Pointer ist, der auf den Hauptspeicher verweist, muss er noch dereferenziert werden. Dies passiert mit dem RTL-Konstrukt `mem`, das für eine Hauptspeicher-Adresse steht. In ULIX darf jeder Wert direkt gepusht werden, auch immediates und Werte, die schon im Speicher liegen. Deshalb wird als constraint `general_operand` verwendet.

5.2.3 Pop Instruction Pattern

Der pop-Befehl ist das Inverse zum push. Demnach wird für die pop-Instruktion das Präinkrement der Stack-Pointers benötigt. Für den pop-Befehl gibt es keinen Standard Pattern Namen, also wird er beim Übersetzen von GIMPLE zu RTL nicht berücksichtigt. Beim Epilog von Subroutinen generiert das ULIX-Backend allerdings manuell pop-RTLs. Deshalb wird kein leerer Name verwendet.

30 ⟨*pop instructions* 30⟩≡ (27b)

```
;; pop instructions

(define_insn "popqi1"
  [(set (mem:QI (pre_inc (reg:SI SP_REGNUM)))
        (match_operand:QI 0 "general_operand" ""))]
  ""
  "pop    byte\\t%0"
)

(define_insn "pophi1"
  [(set (mem:HI (pre_inc (reg:SI SP_REGNUM)))
        (match_operand:HI 0 "general_operand" ""))]
  ""
  "pop    short\\t%0"
)

(define_insn "popsi1"
  [(set (mem:SI (pre_inc (reg:SI SP_REGNUM)))
        (match_operand:SI 0 "general_operand" ""))]
  ""
  "pop    int\\t%0"
)
```

Das Vorgehen beim Prädekrement und die Begründung dafür ist analog zur push-Instruktion (siehe Kapitel 5.2.2). Das Prädekrement ist beim pop allerdings das Ziel und nicht die Quelle.

5.2.4 Add Instruction Pattern

Der add-Befehl ist eine normale arithmetische Addition. Der Standard Pattern Name für den add-Befehl lautet `add`*m*`3`, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 3 steht für die Anzahl der Operanden, die erwartet wird. Der zweite und dritte Operand werden addiert und in den ersten Operand geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

31a ⟨*add instructions* 31a⟩≡ (27b)

```
;; add instructions

(define_insn "addqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (plus:QI (match_operand:QI 1 "general_operand" "")
                 (match_operand:QI 2 "general_operand" "")))]
  ""
  "add    byte\\t%0, %1, %2"
)

(define_insn "addhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (plus:HI (match_operand:HI 1 "general_operand" "")
                 (match_operand:HI 2 "general_operand" "")))]
  ""
  "add    short\\t%0, %1, %2"
)

(define_insn "addsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (plus:SI (match_operand:SI 1 "general_operand" "")
                 (match_operand:SI 2 "general_operand" "")))]
  ""
  "add    int\\t%0, %1, %2"
)
```

Der RTL-Ausdruck `plus` addiert zwei Zahlen ohne Seiteneffekte. Daher muss das Ergebnis noch implizit mit `set` in den ersten Operanden geschrieben werden.

5.2.5 Subtract Instruction Pattern

Der sub-Befehl ist eine normale arithmetische Subtraktion. Der Standard Pattern Name für den sub-Befehl lautet `sub`*m*`3`, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 3 steht für die Anzahl der Operanden, die erwartet wird. Der dritte Operand wird vom zweiten subtrahiert und das Ergebnis wird in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

31b ⟨*subtract instructions* 31b⟩≡ (27b)

```
;; subtract instructions
```

```
(define_insn "subqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (minus:QI (match_operand:QI 1 "general_operand" "")
                  (match_operand:QI 2 "general_operand" "")))]
  ""
  "sub    byte\\t%0, %1, %2"
)

(define_insn "subhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (minus:HI (match_operand:HI 1 "general_operand" "")
                  (match_operand:HI 2 "general_operand" "")))]
  ""
  "sub    short\\t%0, %1, %2"
)

(define_insn "subsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (minus:SI (match_operand:SI 1 "general_operand" "")
                  (match_operand:SI 2 "general_operand" "")))]
  ""
  "sub    int\\t%0, %1, %2"
)
```

Der RTL-Ausdruck `minus` subtrahiert zwei Zahlen ohne Seiteneffekte. Deshalb muss das Ergebnis noch implizit mit `set` in den ersten Operanden geschrieben werden.

5.2.6 Multiplication Instruction Pattern

Der mul-Befehl ist eine normale arithmetische Multiplikation. Der Standard Pattern Name für den mul-Befehl lautet `mul`*m*`3`, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 3 steht für die Anzahl der Operanden, die erwartet wird. Der zweite und dritte Operand werden multipliziert und das Ergebnis wird in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

32 ⟨*multiplicate instructions* 32⟩≡ (27b)

```
;; multiplicate instructions

(define_insn "mulqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (mult:QI (match_operand:QI 1 "general_operand" "")
                 (match_operand:QI 2 "general_operand" "")))]
  ""
  "mul    byte\\t%0, %1, %2"
)

(define_insn "mulhi3"
```

```
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (mult:HI (match_operand:HI 1 "general_operand" "")
                 (match_operand:HI 2 "general_operand" "")))]
  ""
  "mul    short\\t%0, %1, %2"
)

(define_insn "mulsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (mult:SI (match_operand:SI 1 "general_operand" "")
                 (match_operand:SI 2 "general_operand" "")))]
  ""
  "mul    int\\t%0, %1, %2"
)
```

Der RTL-Ausdruck `mult` multipliziert zwei Zahlen ohne Seiteneffekte. Deshalb muss das Ergebnis noch implizit mit `set` in den ersten Operanden geschrieben werden.

5.2.7 Division Instruction Pattern

Der div-Befehl ist eine normale arithmetische Division. Der Standard Pattern Name für den div-Befehl lautet `div`m`3`, wobei m der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 3 steht für die Anzahl der Operanden, die erwartet wird. Der zweite Operand wird durch den dritten dividiert und das Ergebnis wird in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

33 ⟨*divide instructions* 33⟩≡ (27b)

```
  ;; divide instructions

  (define_insn "divqi3"
    [(set (match_operand:QI 0 "nonimmediate_operand" "")
          (div:QI (match_operand:QI 1 "general_operand" "")
                  (match_operand:QI 2 "general_operand" "")))]
    ""
    "div    byte\\t%0, %1, %2"
  )

  (define_insn "divhi3"
    [(set (match_operand:HI 0 "nonimmediate_operand" "")
          (div:HI (match_operand:HI 1 "general_operand" "")
                  (match_operand:HI 2 "general_operand" "")))]
    ""
    "div    short\\t%0, %1, %2"
  )

  (define_insn "divsi3"
    [(set (match_operand:SI 0 "nonimmediate_operand" "")
          (div:SI (match_operand:SI 1 "general_operand" "")
```

```
                (match_operand:SI 2 "general_operand" "")))]
  ""
  "div    int\\t%0, %1, %2"
)
```

Der RTL-Ausdruck `div` multipliziert zwei Zahlen ohne Seiteneffekte. Deshalb muss das Ergebnis noch implizit mit `set` in den ersten Operanden geschrieben werden.

5.2.8 Modulo Instruction Pattern

Der mod-Befehl berechnet das Modulo, also den arithmetischen Rest aus der Division zweier ganzer Zahlen. Der Standard Pattern Name für den mod-Befehl lautet `mod`*m*`3`, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 3 steht für die Anzahl der Operanden, die erwartet wird. Es wird das Modulo des zweiten vom dritten Operanden berechnet und das Ergebnis wird in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

34 ⟨*Modulo instructions* 34⟩≡ (27b)

```
;; Modulo instructions

(define_insn "modqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (mod:QI (match_operand:QI 1 "general_operand" "")
                (match_operand:QI 2 "general_operand" "")))]
  ""
  "mod    byte\\t%0, %1, %2"
)

(define_insn "modhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (mod:HI (match_operand:HI 1 "general_operand" "")
                (match_operand:HI 2 "general_operand" "")))]
  ""
  "mod    short\\t%0, %1, %2"
)

(define_insn "modsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (mod:SI (match_operand:SI 1 "general_operand" "")
                (match_operand:SI 2 "general_operand" "")))]
  ""
  "mod    int\\t%0, %1, %2"
)
```

Der RTL-Ausdruck `mod` berechnet das Modulo zweier Zahlen ohne Seiteneffekte. Deshalb muss das Ergebnis noch implizit mit `set` in den ersten Operanden geschrieben werden.

5.2.9 And Instruction Pattern

Die and-Instruktion berechnet das bitweise, logische Und über zwei Zahlenwerte. Der Standard Pattern Name für den and-Befehl lautet `and`*m*`3`, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 3 steht für die Anzahl der Operanden, die erwartet wird. Es wird das bitweise, logische Und des zweiten und dritten Operanden berechnet und das Ergebnis wird in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

35 ⟨*and instructions* 35⟩≡ (27b)

```
;; and instructions

(define_insn "andqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (and:QI (match_operand:QI 1 "general_operand" "")
                (match_operand:QI 2 "general_operand" "")))]
  ""
  "and    byte\\t%0, %1, %2"
)

(define_insn "andhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (and:HI (match_operand:HI 1 "general_operand" "")
                (match_operand:HI 2 "general_operand" "")))]
  ""
  "and    short\\t%0, %1, %2"
)

(define_insn "andsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (and:SI (match_operand:SI 1 "general_operand" "")
                (match_operand:SI 2 "general_operand" "")))]
  ""
  "and    int\\t%0, %1, %2"
)
```

Der RTL-Ausdruck `and` berechnet das bitweise, logische Und zweier Zahlen ohne Seiteneffekte. Deshalb muss das Ergebnis noch implizit mit `set` in den ersten Operanden geschrieben werden.

5.2.10 Or Instruction Pattern

Die or-Instruktion berechnet das bitweise, inklusive Oder über zwei Zahlenwerte. Der Standard Pattern Name für den or-Befehl lautet `or`*m*`3`, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 3 steht für die Anzahl der Operanden, die erwartet wird. Es wird das bitweise, logische Oder des zweiten und dritten Operanden berechnet und das Ergebnis wird in den ersten Operanden geschrieben. Deshalb darf

der erste Operand auch kein immediate sein.

36a ⟨*or instructions* 36a⟩≡ (27b)

```
;; or instructions

(define_insn "iorqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (ior:QI (match_operand:QI 1 "general_operand" "")
                (match_operand:QI 2 "general_operand" "")))]
  ""
  "or     byte\\t%0, %1, %2"
)

(define_insn "iorhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (ior:HI (match_operand:HI 1 "general_operand" "")
                (match_operand:HI 2 "general_operand" "")))]
  ""
  "or     short\\t%0, %1, %2"
)

(define_insn "iorsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (ior:SI (match_operand:SI 1 "general_operand" "")
                (match_operand:SI 2 "general_operand" "")))]
  ""
  "or     int\\t%0, %1, %2"
)
```

Der RTL-Ausdruck `ior` berechnet das bitweise, inklusive Oder zweier Zahlen ohne Seiteneffekte. Deshalb muss das Ergebnis noch implizit mit `set` in den ersten Operanden geschrieben werden.

5.2.11 Xor Instruction Pattern

Die xor-Instruktion berechnet das bitweise, exklusive Oder über zwei Zahlenwerte. Der Standard Pattern Name für den xor-Befehl lautet `xor`*m*`3`, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 3 steht für die Anzahl der Operanden, die erwartet wird. Es wird das bitweise, logische Oder des zweiten und dritten Operanden berechnet und das Ergebnis wird in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

36b ⟨*xor instructions* 36b⟩≡ (27b)

```
;; xor instructions

(define_insn "xorqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (xor:QI (match_operand:QI 1 "general_operand" "")
```

```
                (match_operand:QI 2 "general_operand" "")))]
  ""
  "xor    byte\\t%0, %1, %2"
)

(define_insn "xorhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (xor:HI (match_operand:HI 1 "general_operand" "")
                (match_operand:HI 2 "general_operand" "")))]
  ""
  "xor    short\\t%0, %1, %2"
)

(define_insn "xorsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (xor:SI (match_operand:SI 1 "general_operand" "")
                (match_operand:SI 2 "general_operand" "")))]
  ""
  "xor    int\\t%0, %1, %2"
)
```

Der RTL-Ausdruck `xor` berechnet das bitweise, exklusive Oder zweier Zahlen ohne Seiteneffekte. Deshalb muss das Ergebnis noch implizit mit `set` in den ersten Operanden geschrieben werden.

5.2.12 Not Instruction Pattern

Die not-Instruktion berechnet das bitweise Not, also das Einerkomplement über eine Zahl. Der Standard Pattern Name für den not-Befehl lautet `one_cmpl`m`2`, wobei m der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 2 steht für die Anzahl der Operanden, die erwartet wird. Es wird das Einerkomplement des zweiten Operanden berechnet und das Ergebnis wird in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

37 ⟨*not instructions* 37⟩≡ (27b)

```
;; not instructions

(define_insn "one_cmplqi2"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (not:QI (match_operand:QI 1 "general_operand" "")))]
  ""
  "not    byte\\t%0, %1"
)

(define_insn "one_cmplhi2"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (not:HI (match_operand:HI 1 "general_operand" "")))]
  ""
```

```
  "not    short\\t%0, %1"
)

(define_insn "one_cmplsi2"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (not:SI (match_operand:SI 1 "general_operand" "")))]
  ""
  "not    int\\t%0, %1"
)
```

Der RTL-Ausdruck `not` berechnet das Einerkomplement einer Zahl ohne Seiteneffekte. Deshalb muss das Ergebnis noch implizit mit `set` in den ersten Operanden geschrieben werden.

5.2.13 Shift Instruction Patterns

In der ULIX-Architektur gibt es drei unterschiedliche shift-Befehle:

1. Links-Shift: `ls`
2. Logischer Rechts-Shift: `lsr`
3. Arithmetischer Rechts-Shift: `asr`

38a ⟨*shift instructions* 38a⟩≡ (27b)

```
;; shift instructions
⟨left shift 38b⟩
⟨arithmetic right shift 39⟩
⟨logical right shift 40⟩
```

Left-Shift Instruction Pattern

Der Links-Shift interpretiert eine Zahl als Folge von Bits und verschiebt die Bits um eine vorgegebene Anzahl Stellen nach links. Für jede Stelle geht ein Bit am Anfang des Bitfeldes verloren und am Ende wird eine Null angefügt. Der Standard Pattern Name für den Links-Shift lautet `ashlm3`, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 3 steht für die Anzahl der Operanden, die erwartet wird. Der zweite Operand wird um die Anzahl im dritten Operanden nach links verschoben. Das Ergebnis wird in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

38b ⟨*left shift* 38b⟩≡ (38a)

```
;; left shift

(define_insn "ashlqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
```

```
        (ashift:QI (match_operand:QI 1 "general_operand" "")
                (match_operand:QI 2 "general_operand" "")))]
  ""
  "sl     byte\\t%0, %1, %2"
)

(define_insn "ashlhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (ashift:HI (match_operand:HI 1 "general_operand" "")
                (match_operand:HI 2 "general_operand" "")))]
  ""
  "sl     short\\t%0, %1, %2"
)

(define_insn "ashlsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (ashift:SI (match_operand:SI 1 "general_operand" "")
                (match_operand:SI 2 "general_operand" "")))]
  ""
  "sl     int\\t%0, %1, %2"
)
```

Der RTL-Ausdruck `ashift` verschiebt seinen ersten Operanden um die Anzahl, die in seinem zweiten Operanden steht nach links. Dies passiert ohne Seiteneffekte. Deshalb muss das Ergebnis noch implizit mit `set` in sein Ziel geschrieben werden.

Arithmetic Right-Shift Instruction Pattern

Der arithmetische Rechts-Shift interpretiert eine Zahl als Folge von Bits und verschiebt die Bits um eine vorgegebene Anzahl Stellen nach rechts. Dabei wird darauf geachtet, welchen Wert das erste Bit, das Vorzeichen-Bit, hat. War das erste Bit eine 1, so wird für jede Stelle, die geshiftet wird eine 1 angefügt. Bei einer 0 werden Nullen angefügt. Dadurch bleibt das Vorzeichen der ursprünglichen Zahl erhalten. Für jede Stelle die geshiftet wird geht eine Stelle auf der rechten Seite verloren. Der Standard Pattern Name für den arithmetischen Rechts-Shift lautet `ashr`*m*`3`, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 3 steht für die Anzahl der Operanden, die erwartet wird. Der zweite Operand wird um die Anzahl im dritten Operanden nach rechts geshiftet. Das Ergebnis wird in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

39 ⟨*arithmetic right shift* 39⟩≡ (38a)

```
  ;; arithmetic right shift

  (define_insn "ashrqi3"
    [(set (match_operand:QI 0 "nonimmediate_operand" "")
          (ashiftrt:QI (match_operand:QI 1 "general_operand" "")
                  (match_operand:QI 2 "general_operand" "")))]
    ""
```

```
  "asr    byte\\t%0, %1, %2"
)

(define_insn "ashrhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (ashiftrt:HI (match_operand:HI 1 "general_operand" "")
                (match_operand:HI 2 "general_operand" "")))]
  ""
  "asr    short\\t%0, %1, %2"
)

(define_insn "ashrsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (ashiftrt:SI (match_operand:SI 1 "general_operand" "")
                (match_operand:SI 2 "general_operand" "")))]
  ""
  "asr    int\\t%0, %1, %2"
)
```

Der RTL-Ausdruck `ashift` verschiebt seinen ersten Operanden um die Anzahl, die in seinem zweiten Operanden steht arithmetisch nach rechts. Dies passiert ohne Seiteneffekte. Deshalb muss das Ergebnis noch implizit mit `set` in sein Ziel geschrieben werden.

Logical Right-Shift Instruction Pattern

Der logische Rechts-Shift interpretiert eine Zahl als Folge von Bits und verschiebt die Bits um eine vorgegebene Anzahl Stellen nach rechts. Für jede Stelle geht ein Bit am Ende des Bitfeldes verloren und am Anfang wird eine Null angefügt. Dadurch kann sich das Vorzeichen der ursprünglichen Zahl ändern. Für jede Stelle die geshiftet wird geht eine Stelle auf der rechten Seite verloren. Der Standard Pattern Name für den arithmetischen Rechts-Shift lautet `ashr`*m*`3`, wobei *m* der Abkürzung des Modus in Kleinbuchstaben entspricht. Die 3 steht für die Anzahl der Operanden, die erwartet wird. Der zweite Operand wird um die Anzahl im dritten Operanden nach rechts geshiftet. Das Ergebnis wird in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

40 ⟨*logical right shift* 40⟩≡ (38a)

```
(define_insn "lshrqi3"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (lshiftrt:QI (match_operand:QI 1 "general_operand" "")
                (match_operand:QI 2 "general_operand" "")))]
  ""
  "lsr    byte\\t%0, %1, %2"
)

(define_insn "lshrhi3"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
```

```
        (lshiftrt:HI (match_operand:HI 1 "general_operand" "")
                (match_operand:HI 2 "general_operand" "")))]
  ""
  "lsr    short\\t%0, %1, %2"
)

(define_insn "lshrsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (lshiftrt:SI (match_operand:SI 1 "general_operand" "")
                (match_operand:SI 2 "general_operand" "")))]
  ""
  "lsr    int\\t%0, %1, %2"
)
```

5.2.14 Cast Instruction Pattern

Es muss möglich sein einen großen Datentyp in einen kleineren zu konvertieren und umgekehrt. Dafür werden Casts benötigt. Hier wird wieder die Einfachheit der ULIX-Architektur offensichtlich. Denn für den Up-cast, also das umwandeln eines kleinen Datentyps in einen großen existiert ein spezieller Assembler-Befehl. In realen CPUs ist dies unüblich und es wird sich meistens mit arithmetischen Shifts beholfen. Diese Methode ist allerdings nicht besonders schön zu lesen und für einen Assembler-Laien kaum zu verstehen. Das ist der Grund wieso ein expliziter upcast-Befehl existiert. Der upcast-Befehl ist sign-extended, es wird also das Vorzeichen beibehalten. Dies ist für vorzeichenbehaftete Zahlen wichtig. Für vorzeichenlose Zahlen wäre dieses verhalten allerdings nicht erwünscht, da hier das erste Bit nicht das Vorzeichen-Bit ist, sondern das Most Significant Bit. Würden also bei einem upcast vor die Zahl lauter Einsen angefügt, so würde der Wert der Zahl größer werden. Bei einem Cast soll der Wert aber, wenn möglich, gleich bleiben. Im Vergleich zum upcast kann dies bei einem downcast nicht immer erreicht werden.

Die Cast-Befehle sind folgendermaßen gruppiert:

41 ⟨*Operations for up- and down-casts* 41⟩≡ (27b)

```
;;
;; Operations for up- and down-casts
;;
```

⟨*truncate instructions* 42a⟩
⟨*sign-extend instructions* 43⟩
⟨*zero-extend instructions* 44⟩

Down-Cast Instruction Pattern

Die truncate instructions sind für den Down-Cast zuständig. Wie der Name schon sagt, wird der überschüssige Teil einfach abgeschnitten. Daher ist es auch nicht immer möglich, dass der Zahlenwert bei einem Down-Cast gleich bleibt. Ist der Zahlenwert für

den Zieldatentyp zu groß, funktioniert der Down-Cast nicht erwartungsgemäß. Deshalb führt C implizit keinen Down-Casts durch.

Der Standard Pattern Name für den trunc-Befehl lautet `trunc`*mn*`2`, wobei *m* dem Modus des Eingabewertes und *n* dem Modus in den gecastet wird, entspricht. Die 2 steht für die Anzahl der Operanden, die erwartet wird. Der zweite Operand, der im Modus *m* vorliegt, wird in den Modus *n* gecastet und dann in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

42a ⟨*truncate instructions* 42a⟩≡ (41)

```
;; truncate operations from si

(define_insn "truncsiqi2"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (truncate:QI (match_operand:SI 1 "general_operand" "")))]
  ""
  "move   byte\\t%0, %1"
)

(define_insn "truncsihi2"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (truncate:HI (match_operand:SI 1 "general_operand" "")))]
  ""
  "move   short\\t%0, %1"
)

;; truncate operation from hi

(define_insn "trunchiqi2"
  [(set (match_operand:QI 0 "nonimmediate_operand" "")
        (truncate:QI (match_operand:HI 1 "general_operand" "")))]
  ""
  "move   short\\t%0, %1"
)
```

Der Down-Cast lässt sich bei ULIX sehr leicht durchführen, da auf jede Adresse, Speicher oder Register, in allen Größen zugegriffen werden kann.

Zum Beispiel: Im Register r0 steht ein int-Wert mit 4 Byte. Dieser Wert soll in ein 2 Byte short gecastet werden. Die Zieladresse ist die Speicherstelle auf die das Register r2 verweist. Dafür genügt es einen move-Befehl der Länge short durchzuführen:

42b ⟨*Downcast-Assembler Beispiel* 42b⟩≡

```
move   short    [r1 + 0], r0
```

Dieser Befehl liest den Wert im Register r0 als short ein, auch wenn in r0 eigentlich ein int-Wert steht. Das Ergebnis wird nach `[r1 + 0]`, also in die Speicheradresse, die in r1 steht, geschrieben. Mehr muss für den Down-Cast nicht getan werden.

Sign-Extend Up-Cast Instruction Pattern

Der vorzeichenbehaftete Up-Cast in ULIX funktioniert ganz einfach über den Assembler-Befehl `upcast`.

Der Standard Pattern Name für den extend-Befehl lautet `extend`mn`2`, wobei m dem Modus des Eingabewertes und n dem Modus in den gecastet wird, entspricht. Die 2 steht für die Anzahl der Operanden, die erwartet wird. Der zweite Operand, der im Modus m vorliegt, wird in den Modus n gecastet und dann in den ersten Operanden geschrieben. Deshalb darf der erste Operand auch kein immediate sein.

43 ⟨*sign-extend instructions* 43⟩≡ (41)

```
;; sign-extend operations

;; sign-extend from qi

(define_insn "extendqihi2"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (sign_extend:HI (match_operand:QI 1 "general_operand" "")))]
  ""
  "upcast short\\t%0, byte %1"
)

(define_insn "extendqisi2"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (sign_extend:SI (match_operand:QI 1 "general_operand" "")))]
  ""
  "upcast int\\t%0, byte %1"
)

;; sign-extend from hi

(define_insn "extendhisi2"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (sign_extend:SI (match_operand:HI 1 "general_operand" "")))]
  ""
  "upcast int\\t%0, short %1"
)
```

Zero-Extend Up-Cast Instruction Pattern

Der Upcast für nicht-vorzeichenbehaftete Zahlen ist nicht ganz so einfach. Um das Prinzip zu erklären werde ich ein kleines Beispiel besprechen:

Angenommen man hat den Wert 42 als vorzeichenlosen byte-Wert, also mit einem Byte Breite gespeichert. Dann sieht die binäre Form folgendermaßen aus: `00101010`. Diese Zahl soll in einen 2 Byte breiten short-Wert gecastet werden.

Die erste Idee ist es diesen Wert einfach in die short-Adresse zu kopieren. Es ist bei ULIX auch ohne Weiteres möglich einen byte-Wert in eine short-Adresse zu kopieren.

Der Wert wird wie erwünscht in das Least Significant Byte kopiert. Das Problem ist nur, dass wir nicht wissen, welcher Wert im Most Significant Byte ist, da der move-Befehl mit byte-Breite das höhere Byte komplett unberührt lässt. Angenommen in der Zieladresse stand vorher der Wert 23232. Binär sieht dieser Wert so aus: 01011010 11000000. Durch den Move-Befehl passiert also folgendes:

```
         00101010 = 42
                            wird kopiert nach
01011010 11000000 = 23232
--------------------------
01011010 00101010 = 23082
```

Am Ende steht in der Speicherstelle der Wert 23082, obwohl 42 darin stehen sollte. Dieses Problem wird behoben, indem zuerst der Wert 0 als short und erst danach die 42 in die Ziel-Speicherstelle geschrieben wird. Dadurch ergibt sich folgendes Beispiel:
Zuerst wird die Null ins Ziel kopiert.

```
00000000 00000000 = 0
                            wird kopiert nach
01011010 11000000 = 23232
--------------------------
00000000 00000000 = 0
```

Danach kann die 42 ganz normal ins Ziel kopiert werden.

```
         00101010 = 42
                            wird kopiert nach
00000000 00000000 = 0
--------------------------
00000000 00101010 = 42
```

44 ⟨*zero-extend instructions* 44⟩≡ (41)

```
;; zero-extend instructions

;; zero-extend from qi

(define_insn "zero_extendqihi2"
  [(set (match_operand:HI 0 "nonimmediate_operand" "")
        (zero_extend:HI (match_operand:QI 1 "general_operand" "")))]
  ""
  "move   short\\t%0, #0\;move   byte\\t%0, %1"
)

(define_insn "zero_extendqisi2"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (zero_extend:SI (match_operand:QI 1 "general_operand" "")))]
```

```
  ""
  "move   int\\t%0, #0\;move   byte\\t%0, %1"
)

;; zero-extend from hi

(define_insn "zero_extendhisi2"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (zero_extend:SI (match_operand:HI 1 "general_operand" "")))]
  ""
  "move   int\\t%0, #0\;move   short\\t%0, %1"
)
```

Was hier auffällt ist, dass im Output Template zwei Assembler-Befehle stehen, die durch ein ; getrennt sind. Dies bedeutet, dass zwei Assembler-Befehle generiert werden. Für einen optimierten Compiler wäre es in diesem Fall besser gewesen, statt dem `define_insn` ein `define_expand` zu verwenden. Diese Expander Definition hätte dann zwei RTL-Pattern generieren können, die genau das selbe machen, wie die zwei Assembler-Befehle. Der Vorteil daran wäre, dass der GCC die RTL-Muster noch optimieren könnte, während die Assembler-Befehle genau so ausgegeben werden. Ich habe mich aber bewusst für diese Variante entschieden, da man so den Zero-Extended Up-Cast leichter erkennen kann und dadurch der Assembler-Code besser zu lesen ist.

5.2.15 Compare Instruction Pattern

Vergleiche kommen in höheren Programmiersprachen wie C häufig vor, beispielsweise bei if-Abfragen, bei der Abbruchbedingung von Schleifen und so weiter.

Um solch einen Vergleich auf ULIX-Assembler abzubilden werden źzwei Gruppen von Befehlen benötigt:

1. Compare-Befehle
2. Conditional Branch-Befehle

Egal welcher Vergleich zwischen zwei Werten angestellt wird, der erste Befehl ist immer eine Compare-Instruktion. Aufgabe der Compare-Instruktion ist es die beiden Werte zu vergleichen und die Flags im Program Status Word entsprechend zu setzen. Der zweite Befehl ist ein Conditional Branch, also ein bedingter Sprung. Die Bedingung auf die geprüft wird hängt von dem Vergleich ab, der durchgeführt wird. Dem bedingten Sprung hat als Operand ein Label als Sprungziel. Ist die Bedingung erfüllt, so wird zum Label gesprungen. Im anderen Fall wird ganz normal die nächste Instruktion ausgeführt. Der Programm-Ablauf verzweigt sich also, abhängig davon, ob die Bedingung erfüllt ist.

Welche Flags durch den Compare-Befehl gesetzt werden und wie der Conditional Branch-Befehl die Flags verwertet ist in „The Ulix CPU“[16] genau erklärt.

Der Standard Pattern Name für die Compare Instruktion ist `cmpm`, wobei m der Abkürzung des Modus in Kleinbuchstaben entspricht.

46 ⟨*Compare instructions* 46⟩≡ (27b)

```
;; Compare instructions

(define_insn "cmpqi"
  [(set (reg:QI PSW_REGNUM) (compare:QI (match_operand:QI 0 "general_operand" "")
                        (match_operand:QI 1 "general_operand" "")))]
  ""
  "cmp    char\\t%0, %1"
)

(define_insn "cmphi"
  [(set (reg:HI PSW_REGNUM) (compare:HI (match_operand:HI 0 "general_operand" "")
                        (match_operand:HI 1 "general_operand" "")))]
  ""
  "cmp    byte\\t%0, %1"
)

(define_insn "cmpsi"
  [(set (reg:SI PSW_REGNUM) (compare:SI (match_operand:SI 0 "general_operand" "")
                        (match_operand:SI 1 "general_operand" "")))]
  ""
  "cmp    int\\t%0, %1"
)
```

Der Compare-Befehl existiert für jede Operanden-Größe einmal. Beide Operanden müssen gleich groß sein, damit sie verglichen werden können. Ist dies nicht der Fall, so müssen die Operanden auf die selbe Größe gecastet werden.

5.2.16 Conditional Jump Pattern

Im ULIX-Assembler gibt es zehn Conditional Branch-Instruktionen, die für den GCC von Bedeutung sind. Sie unterscheiden sich alle in der Sprungbedingung. Mögliche bedingte Sprünge sind:

jeq springt, wenn die verglichenen Werte *gleich* sind.

jne springt, wenn die verglichenen Werte *nicht gleich* sind.

jsgt springt, wenn der erste Wert größer als der zweite Wert ist. Die Werte sind vorzeichenbehaftet.

jslt springt, wenn der erste Wert kleiner als der zweite Wert ist. Die Werte sind vorzeichenbehaftet.

jsge springt, wenn der erste Wert größer gleich dem zweite Wert ist. Die Werte sind vorzeichenbehaftet.

jsle springt, wenn der erste Wert kleiner gleich dem zweite Wert ist. Die Werte sind vorzeichenbehaftet.

jgt springt, wenn der erste Wert größer als der zweite Wert ist. Die Werte sind vorzeichenlos.

jlt springt, wenn der erste Wert kleiner als der zweite Wert ist. Die Werte sind vorzeichenlos.

jge springt, wenn der erste Wert größer gleich dem zweite Wert ist. Die Werte sind vorzeichenlos.

jle springt, wenn der erste Wert kleiner gleich dem zweite Wert ist. Die Werte sind vorzeichenlos.

Zusätzlich zu diesen Conditional Branches gibt es noch acht bedingte Sprünge in ULIX, die die einzelnen Flags prüfen (siehe [16]). Für den GCC sind diese Branches aber uninteressant.

Der grundsätzliche Ablauf eines Vergleichs ist in Kapitel 5.2.15 besprochen.

Der Standard Pattern Name eines Conditional Branches ist `b`*`cond`*, wobei `cond` die Bedingung ist, die geprüft wird. Für `cond` wird der RTL-Name der Bedingung verwendet.

47 ⟨*Conditional Jump instructions* 47⟩≡ (27b)

```
;; Conditional Jumps

;; Conditional Jumps

(define_insn "beq"
  [(set (pc)
        (if_then_else (eq (reg:SI PSW_REGNUM) (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc))
  )]
  ""
  "jeq       \\t%l0"
)

(define_insn "bne"
  [(set (pc)
        (if_then_else (ne (reg:SI PSW_REGNUM) (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "jne       \\t%l0"
)

(define_insn "bgt"
  [(set (pc)
```

```
        (if_then_else (gt (reg:SI PSW_REGNUM) (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc))
  )]
  ""
  "jsg       \\t%l0"
)

(define_insn "blt"
  [(set (pc)
        (if_then_else (lt (reg:SI PSW_REGNUM) (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "jsl       \\t%l0"
)

(define_insn "bgtu"
  [(set (pc)
        (if_then_else (gtu (reg:SI PSW_REGNUM) (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "jg        \\t%l0"
)

(define_insn "bltu"
  [(set (pc)
        (if_then_else (ltu (reg:SI PSW_REGNUM) (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "jl        \\t%l0"
)

(define_insn "bge"
  [(set (pc)
        (if_then_else (ge (reg:SI PSW_REGNUM) (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "jsge      \\t%l0"
)

(define_insn "ble"
  [(set (pc)
        (if_then_else (le (reg:SI PSW_REGNUM) (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
```

```
  ""
  "jsle      \\t%l0"
)

(define_insn "bgeu"
  [(set (pc)
        (if_then_else (geu (reg:SI PSW_REGNUM) (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "jge      \\t%l0"
)

(define_insn "bleu"
  [(set (pc)
        (if_then_else (leu (reg:SI PSW_REGNUM) (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "jle      \\t%l0"
)
```

Das RTL-Template setzt den pc, also den Program Counter. Der RTL-Ausdruck `(pc)` verweist aus diesen. Der `if_then_else`-Ausdruck hat drei Operanden. Der erste ist die Bedingung, die geprüft wird. Ist die Bedingung erfüllt wird der zweite, sonst der dritte Operand zurückgegeben. In der Bedingung wird das Program Status Word immer mit dem konstanten Wert Null verglichen, wobei der Vergleich der Bedingung entspricht, die geprüft wird. Hierbei ist zu beachten, dass die ULIX-CPU intern anders verfährt und das hier verwendete Vorgehen nicht zum Erfolg führen würde. Da die bedingten Sprünge in `define_insn` definiert worden sind, wird der resultierende RTL-Code einfach in den zugehörigen Assembler umgewandelt. Dadurch kommt es zu keinen Problemen, auch wenn der GCC dadurch nicht so gut optimieren kann.

5.2.17 Unconditional Jump Instruction Pattern

Zusätzlich zu den bedingten Sprüngen gibt es auch noch den unbedingten Sprung. Dieser Sprung wird immer ausgeführt, ohne dass eine Bedingung geprüft wird. Der Standard Pattern Name des Unconditional Jumps ist `jump`.

49 ⟨*Unconditional Jump instruction* 49⟩≡ (27b)

```
;; Unconditional Jump instruction

(define_insn "jump"
  [(set (pc)
        (label_ref (match_operand 0 "" "")))]
  ""
  "jmp       \\t%l0"
)
```

Der Unconditional Jump ist immer direkt.

5.2.18 Indirect Jump Instruction Pattern

Der indirekte Sprung funktioniert bei ULIX genauso, wie ein direkter, da der `jmp`-Befehl auch indirekte Adressierung erlaubt. Der Aufbau ist also analog zum Unconditional Jump. Der Standard Pattern Name des Indirect Jumps ist `indirect_jump`.

50a ⟨*indirect jump instruction* 50a⟩≡ (27b)

```
;; indirect jump instruction

(define_insn "indirect_jump"
  [(set (pc)
        (match_operand:SI 0 "" ""))]
  ""
  "jmp       \\t%0"
)
```

5.2.19 Subroutine Instruction Pattern

Um Unterprozeduren aufrufen zu können werden folgende Teile benötigt:

50b ⟨*subroutine instructions* 50b⟩≡ (27b)

```
;; Subroutine instructions
⟨call ohne Rückgabewert 50c⟩
⟨call mit Rückgabewert 51c⟩
⟨return 55a⟩
⟨prologue und epilogue 53a⟩
```

Unterprozeduraufrufe mit Rückgabewert unterscheiden sich von denen ohne Rückgabewert und werden deshalb getrennt implementiert. Der Prologue wird beim Eintritt, der Epilogue beim Abschließen einer Subroutine ausgeführt.

Subroutinen ohne Rückgabewert

Für die Unterprozeduren ohne Rückgabewert existiert eine Instruction Pattern und eine Expander Definition.

50c ⟨*call ohne Rückgabewert* 50c⟩≡ (50b)

```
⟨call ohne Rückgabewert - expand 51a⟩
⟨call ohne Rückgabewert - insn 51b⟩
```

Die Expander Definition mit dem Standard Pattern Name `call` wird dann ausgeführt, wenn eine Unterprozedur aufgerufen wird.

51a ⟨*call ohne Rückgabewert - expand* 51a⟩≡ (50c)

```
(define_expand "call"
  [(match_operand 0 "" "")]
  ""
{
  ulix_expand_call (NULL_RTX, operands[0]);
  DONE;
}
)
```

Alles was diese Expander Definition macht ist die in `ulix.c` implementierte Funktion `ulix_expand_call (NULL_RTX, operands[0])` aufzurufen. `ulix_expand_call` generiert alle nötigen RTLs für den Unterprozeduraufruf (siehe Seite 52).

51b ⟨*call ohne Rückgabewert - insn* 51b⟩≡ (50c)

```
(define_insn "*call_internal"
  [(call (match_operand:SI 0 "" "")
         (const_int 0))]
  ""
  "jsr       \\t%0"
)
```

Das Instruction Pattern mit dem Namen `*call_internal` ist dafür zuständig, den von `call` generierten RTL-Code in Assembler umzuwandeln. Der Name Beginnt mit einem *. Dies bedeutet, dass der Name nur fürs Debugging verwendet werden kann. Es ist also nicht möglich dieses Pattern explizit aufzurufen. Dies ist auch nicht nötig, da es nur dafür da ist bereits generierten RTL-Code in Assembler umzuwandeln. Es gibt ein `jsr`-Assembler Befehl aus.

Subroutinen mit Rückgabewert

Für die Unterprozeduren mit Rückgabewert existiert eine Instruction Pattern und eine Expander Definition.

51c ⟨*call mit Rückgabewert* 51c⟩≡ (50b)

```
⟨call mit Rückgabewert - expand 51d⟩
⟨call mit Rückgabewert - insn 52b⟩
```

Die Expander Definition mit dem Standard Pattern Name `call_value` wird dann ausgeführt, wenn eine Unterprozedur aufgerufen wird, die einen Rückgabewert hat.

51d ⟨*call mit Rückgabewert - expand* 51d⟩≡ (51c)

```
(define_expand "call_value"
  [(match_operand 0 "" "")
   (match_operand 1 "" "")]
```

```
  ""
{
  ulix_expand_call (operands[0], operands[1]);
  DONE;
}
)
```

Alles was diese Expander Definition macht ist die in `ulix.c` implementierte Funktion `ulix_expand_call (operands[0], operands[1])` aufzurufen. `ulix_expand_call` generiert alle nötigen RTLs für den Unterprozeduraufruf.

52a ⟨*ulix-expand-call* 52a⟩≡ (82c)

```
void ulix_expand_call(rtx retval, rtx mem){
        if (!retval)
                emit_call_insn (gen_rtx_CALL (VOIDmode, mem, const0_rtx));
        else
                emit_call_insn (gen_rtx_SET (GET_MODE (retval), retval,
                        gen_rtx_CALL (VOIDmode, mem, const0_rtx)));
}
```

Zuerst testet `ulix_expand_call`, ob der erste Parameter ein leeres `NULL_RTX` ist. Trifft dies zu, so muss RTL für eine Funktion ohne Rückgabewert erzeugt werden. Im anderen Fall ist der erste Parameter die Adresse in die der Rückgabewert geschrieben werden muss.

Mit `emit_call_insn (gen_rtx_CALL (VOIDmode, mem, const0_rtx));` wird ein RTL-`call` erzeugt, der als erster Parameter die Adresse der Unterprozedur hat. Dieses RTL-Konstrukt wird in der Assemblierungs-Phase dann von `*call_internal` in ein `jsr`-Befehl umgewandelt.

Mit `emit_call_insn (gen_rtx_SET (GET_MODE (retval), retval, gen_rtx_CALL (VOIDmode, mem, const0_rtx)));` wird ein RTL-`call` erzeugt, der als erster Parameter die Adresse der Unterprozedur hat. Dieses RTL-Konstrukt wird in der Assemblierungs-Phase dann von `*call_value_internal` in ein `jsr`-Befehl umgewandelt. Der Rückgabewert der Funktion wird durch das RTL-`set` in das Ziel geschrieben.

52b ⟨*call mit Rückgabewert - insn* 52b⟩≡ (51c)

```
(define_insn "*call_value_internal"
  [(set (match_operand:SI 0 "" "")
        (call (match_operand:SI 1 "" "")
              (const_int 0)))]
  ""
  "jsr       \\t%1"
  )
```

Das Instruction Pattern mit dem Namen `*call_value_internal` ist dafür zuständig, den von `call_value` generierten RTL-Code in Assembler umzuwandeln. Der Name Beginnt mit einem *. Dies bedeutet, dass der Name nur fürs Debugging verwendet werden kann. Es ist also nicht möglich dieses Pattern explizit aufzurufen. Dies ist

auch nicht nötig, da es nur dafür da ist bereits generierten RTL-Code in Assembler umzuwandeln. Es gibt ein `jsr`-Assembler Befehl aus.

Prologue und Epilogue

Der Prologue wird zu Beginn der Subroutine ausgeführt. Er ist dafür zuständig den alten Frame Pointer und die Register zu sichern. Argument Pointer so gesetzt, damit mit dessen Hilfe auf Parameter, die der Subroutine übergeben wurden, zugegriffen werden kann.

53a ⟨*prologue und epilogue* 53a⟩≡ (50b) 54a▷

```
(define_expand "prologue"
  [(const_int 0)]
  ""
  "output_function_prologue (); DONE;"
)
```

Die gesamte Funktionalität des Prologues ist in `ulix.c` in der Funktion `output_function_prologue` implementiert.

53b ⟨*output-function-prologue* 53b⟩≡ (82c)

```
void output_function_prologue (void)
{
        //Rette den alten FP auf den Stack
        emit_insn(gen_pushsi1(gen_rtx_REG (SImode, FRAME_POINTER_REGNUM)));
        /* Setze den neuen FP auf die Speicherstelle in der der alte FP
           eben abgelegt wurde. Dies wird dadurch erreicht, indem
           der SP + die Länge eines Wortes in den FP geschrieben wird.
           Damit zeigt der FP genau eine Speicherstelle über den SP*/
        emit_insn (gen_addsi3 (gen_rtx_REG (SImode, FRAME_POINTER_REGNUM),
          gen_rtx_REG (SImode, SP_REGNUM),
          gen_rtx_CONST_INT(SImode, UNITS_PER_WORD)));
        /* Jetzt wird der SP so angepasst, dass sich die lokalen Variablen
           und der Stack nicht in die Quere kommen. Mit Hilfe von
           get_frame_size () bekommt man die Anzahl Bytes der lokalen
           Variablen. Also muss nur noch der SP um diese Anzahl verringert
           werden. Dadurch entsteht im Stack-Frame ein freier Bereich in den
           die lokalen Variablen geschrieben werden.*/
        emit_insn (gen_subsi3 (gen_rtx_REG (SImode, SP_REGNUM),
          gen_rtx_REG (SImode, SP_REGNUM),
          gen_rtx_CONST_INT(SImode, get_frame_size ())));
        /* Rette alle Register, die in dieser Funktion verwendet werden,
           auf den Stack. Dadurch haben diese Register nach dem Durchlaufen
           der Funktion wieder den selben Wert wie vorher. Die aufrufende
           Funktion kann also alle Register weiterverwenden, als ob keine
           Unterprozedur aufgerufen worden wäre.*/
        int regno;
        for (regno = 0; regno < FIRST_PSEUDO_REGISTER; regno++)
                if (df_regs_ever_live_p(regno) &&
```

```
                ! call_used_regs [regno])emit_insn
                (gen_pushsi1(gen_rtx_REG (SImode, regno)));
        /* Setze den ARG_POINTER auf das nächste Wort hinter dem Framepointer.
           Von hier aus werden dann die Argumente(Aufruf-Parameter)
           adressiert.*/
        emit_insn (gen_subsi3(gen_rtx_REG (SImode, ARG_POINTER_REGNUM),
          gen_rtx_REG (SImode, FRAME_POINTER_REGNUM),
          gen_rtx_CONST_INT(SImode, UNITS_PER_WORD)));
}
```

Mit `emit_insn` wird immer ein in den Machine Description definiertes Instruction Pattern generiert. Die Funktion `gen_name`, bei der `name` dem Namen eines Instruction Pattern ist, bestimmt, welches Pattern erzeugt wird. Beispielsweise `gen_pushsi1` erzeugt einen `pushsi1`, also ein push, mit dem Modus SI.

Der Epilogue wird am Ende einer Subroutine ausgeführt. Er ist dafür zuständig den alten Frame Pointer und die Register, die im Prologue gesichert wurden, wiederherzustellen. Dann wird der Stack Pointer so gesetzt, dass das `rts` auch zur richtigen Adresse zurückspringt. Zuletzt generiert der Epilog noch das `return`-RTL, das in der Assemblierungs-Phase zu einem `rts` umgewandelt wird.

54a ⟨*prologue und epilogue* 53a⟩+≡ (50b) ◁53a

```
(define_expand "epilogue"
  [(const_int 0)]
  ""
  "output_function_epilogue (); DONE;"
)
```

Die gesamte Funktionalität des Prologues ist in `ulix.c` in der Funktion `output_function_epilogue` implementiert.

54b ⟨*output-function-epilogue* 54b⟩≡ (82c)

```
void output_function_epilogue (void)
{
        /* Stelle alle Register, die in dieser Funktion verwendet wurden,
           wieder her. Diese liegen auf dem Stack. Dadurch haben diese Register
           nach dem Durchlaufen der Funktion wieder den selben Wert wie vorher.
           Die aufrufende Funktion kann also alle Register weiterverwenden,
           als ob keine Unterprozedur aufgerufen worden wäre.*/
        int regno;
        for (regno = FIRST_PSEUDO_REGISTER - 1; regno >= 0; regno-)
                if (df_regs_ever_live_p(regno) &&
                  ! call_used_regs [regno])
                  emit_insn(gen_popsi1(gen_rtx_REG (SImode, regno)));
        /* Setze den SP auf den FP. Dadurch ist direkt über dem SP die
           Rücksprungadresse, die beim JSR auf den Stack gelegt wurde.*/
        emit_insn(gen_movsi(gen_rtx_REG (SImode, SP_REGNUM),
          gen_rtx_REG (SImode, FRAME_POINTER_REGNUM)));
```

```
        /* Der FP wird auf den Wert gesetzt, den er hatte, bevor diese Funktion
           aufgerufen wurde. Also kann in der aufrufenden Funktion wieder damit
           gearbeitet werden, als wäre nichts passiert. */
        emit_insn(gen_movsi(gen_rtx_REG (SImode, FRAME_POINTER_REGNUM),
          gen_rtx_MEM (SImode, gen_rtx_REG (SImode, FRAME_POINTER_REGNUM))));
        /* Zuletzt wird das return-RTX, also ein rts ausgeführt. Damit springt
           der PC wieder in die aufrufende Funktion.*/
        emit_jump_insn (gen_return ());
}
```

Return Instruction Pattern

Die return-Instruktion generiert den `rts` Assembler-Befehl. Dieses Instruction Pattern wird explizit im Epilogue aufgerufen.

55a ⟨*return* 55a⟩≡ (50b)

```
(define_insn "return"
  [(return)]
  ""
  "rts"
)
```

5.2.20 Nop Instruction Pattern

Bei ULIX existiert eine nop-Instruktion, also eine Instruktion, die nichts tut. Der Standard Pattern Name der nop Instruktion ist `nop`.

55b ⟨*nop instruction* 55b⟩≡ (27b)

```
;; The nop instruction

(define_insn "nop"
  [(const_int 0)]
  ""
  "nop"
)
```

5.2.21 Dummy Constraints

Es ist möglich für den GCC eigene Constraints zu definieren, die dann für die `match_operand`-Instruktionen in den RTL-Templates verwendet werden können. Der GCC erwartet, dass Constraints definiert werden. Da für ULIX aber keine benutzerdefinierten Constraints benötigt werden, habe ich 2 Dummy-Constraints implementiert, die aber nicht verwendet werden.

55c ⟨*dummy constraint* 55c⟩≡ (27b)

```
;; Dummy Constraints
```

```
(define_register_constraint "f" "NO_REGS"
 "Legacy FPA registers @code{f0}-@code{f7}.")

(define_constraint "I"
 "Blubb."
 (and (match_code "const_int")
      (match_test "1 >= 0")))
```

5.3 Target Description Macros

Neben den Machine Descriptions, die die Verbindung zwischen GIMPLE-Elementen und Assembler-Befehlen herstellen sind die Target Description Macros der zweite, große Teil, der für die Erstellung eines GCC Backends notwendig ist. Mit Hilfe dieser Makros werden Eigenschaften der Zielarchitektur abgebildet, die vom GCC verwendet werden. Beispiele dafür sind der Stackaufbau, das Format von Assemblerbefehlen und deren Operanden und die Byte- und Bit-Reihenfolge.

Die Target Description Macros sind in der Datei `ulix.h` enthalten. `config/ulix/ulix.h`

56a ⟨*ulix.h* 56a⟩≡

```
⟨Lizenz von ulix.h 56b⟩
#ifndef GCC_ULIX_H
#define GCC_ULIX_H

⟨Target CPU builtins 57a⟩
⟨Endianness 57b⟩
⟨Eigenschaften elementarer Einheiten 58a⟩
⟨Alignment 60e⟩
⟨Register-Eigenschaften 61b⟩
⟨Stack- und Subroutinen-Eigenschaften 64a⟩
⟨Adressierung 68c⟩
⟨Meta-Assembler Anweisungen 72c⟩
⟨Assembler Formatierung 74a⟩
⟨Verschiedenes 77a⟩
⟨Dummys 78f⟩
#endif /* ! GCC_ULIX_H */
```

Am Anfang von `ulix.h` steht wieder der Hinweis, dass der GCC und somit auch das ULIX Backend unter der GPL[8] veröffentlicht sind.

56b ⟨*Lizenz von ulix.h* 56b⟩≡ (56a)

```
/* Definitions of target machine for GNU compiler, for Ulix.
   Copyright (C) 1991, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000,
   2001, 2002, 2003, 2004, 2005, 2006, 2007 Free Software Foundation, Inc.
   Ulix-Description by Balthasar Biedermann (balthasar@biedermann.es).

   This file is part of GCC.

   GCC is free software; you can redistribute it and/or modify it
```

```
under the terms of the GNU General Public License as published
by the Free Software Foundation; either version 3, or (at your
option) any later version.

GCC is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public
License for more details.

You should have received a copy of the GNU General Public License
along with GCC; see the file COPYING3.  If not see
<http://www.gnu.org/licenses/>.  */
```

`TARGET_CPU_CPP_BUILTINS` definiert eingebaute Präprozessormakros und Assertions für die Zielarchitektur.

57a ⟨*Target CPU builtins* 57a⟩≡ (56a)

```
/* Target CPU builtins.  */
#define TARGET_CPU_CPP_BUILTINS()              \
  do                                           \
    {                                          \
      builtin_define ("__ulix__");             \
      builtin_assert ("cpu=ulix");             \
      builtin_assert ("machine=ulix");         \
    }                                          \
  while (0)
```

5.3.1 Speicher-Layout

Diese Makros definieren die Byte- und Bit-Reihenfolge der ULIX-Architektur. ULIX verwendet Big Endian.

57b ⟨*Endianness* 57b⟩≡ (56a)

```
/* Define this to be 1 if most significant bit is lowest numbered
   in instructions that operate on numbered bit-fields.  */
#define BITS_BIG_ENDIAN  1

/* Define this if most significant byte of a word is the lowest numbered.
 */
#define BYTES_BIG_ENDIAN  1

/* Define this if most significant word of a multiword number is the lowest
   numbered.
 */
#define WORDS_BIG_ENDIAN  1
```

In ULIX ist ein Wort 32 Bit, also 4 Byte. Mit `UNIT` sind Bytes gemeint.

58a ⟨*Eigenschaften elementarer Einheiten* 58a⟩≡ (56a) 58b▷

```
/* Define the number of bytes building a word
 */
#define UNITS_PER_WORD  4
```

Die größte Ganzzahl ist 32 Bit lang.

58b ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁58a 58c▷

```
/* Size in bits of the largest integer machine mode that should actually be used
 */
#define MAX_FIXED_MODE_SIZE 32
```

Jetzt wird den C-Datentypen die Länge in Bits zugeordnet.
Ein `int` ist 32 Bit lang.

58c ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁58b 58d▷

```
/* Size in Bits of int
 */
#define INT_TYPE_SIZE 32
```

Ein `short` ist 16 Bit lang.

58d ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁58c 58e▷

```
/* Size in Bits of short
 */
#define SHORT_TYPE_SIZE 16
```

Ein `long` ist 32 Bit lang.

58e ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁58d 58f▷

```
/* Size in Bits of long
 */
#define LONG_TYPE_SIZE 32
```

Ein `long long` ist 32 Bit lang.

58f ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁58e 59a▷

```
/* Size in Bits of long long
 */
#define LONG_LONG_TYPE_SIZE 32
```

Ein `char` ist 8 Bit lang.

59a ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁58f 59b▷

```
/* Size in Bits of char
 */
#define CHAR_TYPE_SIZE 8
```

Ein `char` ist standardmäßig vorzeichenlos. Möchte man ein vorzeichenbehafteten `char` so muss man dies explizit durch `signed char` angeben.

59b ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁59a 59c▷

```
/* in default char is unsigned
 */
#define DEFAULT_SIGNED_CHAR 0
```

Der Datentyp `size_t` entspricht einem `unsigned int`.

59c ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁59b 59d▷

```
/* Type of size_t
 */
#define SIZE_TYPE "unsigned int"
```

Die Differenz zweier Pointer hat den Datentyp `int`.

59d ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁59c 59e▷

```
/* The Type of the difference of two pointers.
 */
#define PTRDIFF_TYPE "int"
```

Der Datentyp für Wide Character ist 32 Bit lang.

59e ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁59d 59f▷

```
/* Size in bits of wide characters
 */
#define WCHAR_TYPE_SIZE 32
```

Der erste vorzeichenbehaftete, ganzzahlige Datentyp, der die maximale Breite hat ist ein `int`.

59f ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁59e 60a▷

```
/* The "first" signed integer-class with max size
 */
#define INTMAX_TYPE "int"
```

Der erste vorzeichenlose, ganzzahlige Datentyp, der die maximale Breite hat ist ein `unsigned int`.

60a ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁59f 60b▷

```
/* The "first" unsigned integer-class with max size
 */
#define UINTMAX_TYPE "unsigned int"
```

Ein Pointer hat in Ulix 32 Bit, entspricht also einem Single Integer

60b ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁60a 60c▷

```
/* The Mode of a Pointer in Ulix is the same as an int*/
#define Pmode SImode
```

Der Datentyp von Referenzen auf eine Funktion entspricht dem eines Pointers.

60c ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁60b 60d▷

```
/* The machine mode used for memory references to functions being called*/
#define FUNCTION_MODE  Pmode
```

Die Elemente einer Sprung-Tabelle sind Pointer.

60d ⟨*Eigenschaften elementarer Einheiten* 58a⟩+≡ (56a) ◁60c

```
/* This is the machine mode that elements of a jump-table should have.
 */
#define CASE_VECTOR_MODE Pmode
```

5.3.2 Alignment

Jetzt wird die Ausrichtung von Daten im Speicher festgelegt. In Ulix müssen Daten nicht besonders ausgerichtet sein, d.h. jeder Datentyp kann an jeder ganzzahligen Adresse liegen. Die Ausrichtung ist demnach Byte-genau.

60e ⟨*Alignment* 60e⟩≡ (56a) 61a▷

```
/* Alignment required for function parameters in bits
 */
#define PARM_BOUNDARY   8

/* Alignment required for the stack in bits
 */
#define STACK_BOUNDARY  8

/* Alignment required for a function entry point, in bits.
 */
```

```
#define FUNCTION_BOUNDARY  8

/* The biggest alignment that any data type can require
 */
#define BIGGEST_ALIGNMENT  8

/* Alignment in bits to be given to a structure bit-field
   that follows an empty field
 */
#define EMPTY_FIELD_BOUNDARY 8
```

Ist eine Adresse nicht auf ein Byte genau ist, gibt es einen Fehler.

61a ⟨*Alignment* 60e⟩+≡ (56a) ◁60e

```
/* Nonzero if move instructions will actually fail to work
   when given unaligned data.
 */
#define STRICT_ALIGNMENT 1
```

5.3.3 Register-Eigenschaften

Jetzt wird die Anzahl und die Eigenschaften der Register festgelegt.

ULIX hat 26 Register, demnach ist das Register mit der Nummer 26 das erste Pseudo Register.

61b ⟨*Register-Eigenschaften* 61b⟩≡ (56a) 61c▷

```
/* Number of real Registers/Number of the first pseudo register
 */
#define FIRST_PSEUDO_REGISTER  26
```

In dieser Liste wird angegeben, welche Register eine feste Bedeutung haben und deshalb nicht zum Vorhalten von Daten genutzt werden können. Die Register, die zum Vorhalten verwendet werden können sind die Register 0 - 15, außer dem Register 3, weil dieses register in dieser Implementierung als Return Register verwendet wird. Diese Register bekommen den Wert 0, die Register mit besonderer Bedeutung den Wert 1.

61c ⟨*Register-Eigenschaften* 61b⟩+≡ (56a) ◁61b 62a▷

```
/* This shows which register has a fixed use-case, i.e. PC, and
   could not be used for allocating normal Data. A 0 stands for
   a general usable register, a 1 for a fixed register.
```

```
 */
#define FIXED_REGISTERS \
{                       \
  0,0,0,1,0,0,0,0,      \
  0,0,0,0,0,0,0,0,      \
  1,1,1,1,1,1,1,1,      \
  1,1                   \
}
```

In dieser Liste bekommen die Register eine Null, die in einer Subroutine verwendet werden können und deshalb dann auch gespeichert und nach der Subroutine wiederhergestellt werden müssen. Alle Register, die in der `FIXED_REGISTERS` Liste schon eine 1 hatten müssen jetzt auch wieder eine bekommen. Zusätzlich kommt jetzt nur noch das Register 0 hinzu, weil dieses Register als Frame Pointer verwendet wird und von Funktions Pro- und Epilogue manuell gesichert wird.

62a ⟨*Register-Eigenschaften* 61b⟩+≡ (56a) ◁61c 62b▷

```
/* This shows the register which could be used in a call or if it
   must be saved before entering the call and restored after.
   A 0 must be saved/restored, a 1 not. Every fix register must be a 1 too.
 */
#define CALL_USED_REGISTERS \
{                           \
  1,0,0,1,0,0,0,0,          \
  0,0,0,0,0,0,0,0,          \
  1,1,1,1,1,1,1,1,          \
  1,1                       \
}
```

Hier wird festgelegt, dass jeder Wert in ein Register gespeichert werden kann. Da der größte Modus Single Integer ist und jedes Register diesen Modus fassen kann, wird immer genau ein Register benötigt.

62b ⟨*Register-Eigenschaften* 61b⟩+≡ (56a) ◁62a 63a▷

```
/* Tests how many registers, starting with REGNO is needed to store a value
   of mode MODE. */
#define HARD_REGNO_NREGS(REGNO, MODE)  1

/* nonzero if it is permissible to store a value of mode MODE in hard register
   number REGNO. Because every register could store everything it is always 1
 */
#define HARD_REGNO_MODE_OK(REGNO, MODE) 1
```

Bei ULIX hat kann jede Instruktion auf jedem Register durchgeführt werden. Deshalb werden keine besonderen Register-Klassen benötigt. Zwingend erforderlich sind die Klassen `NO_REGS`, der zu der kein Register gehört und `ALL_REGS` zur der alle Register gehören. `LIM_REG_CLASSES` ist nur dafür zuständig dem GCC die Anzahl der Register

verfügbar zu machen. Zusätzlich wird noch `GENERAL_REGS` benötigt, das ich manuell auf `ALL_REGS` setze.

63a ⟨*Register-Eigenschaften* 61b⟩+≡ (56a) ◁62b 63b▷

```
/* The different register classes.
   There are no special classes in the ulix-cpu, so there are only
   the needed classes.
 */
enum reg_class
{
  NO_REGS,
  ALL_REGS,
  LIM_REG_CLASSES
};
/* Siehe oben
 */
#define REG_CLASS_NAMES  \
{                        \
  "NO_REGS",             \
  "ALL_REGS",            \
}
/* Number of register classes
 */
#define N_REG_CLASSES  (int) LIM_REG_CLASSES
/* Every register is a general register.
 */
#define GENERAL_REGS  ALL_REGS

/* The Class ALL_REGS contains every register. NO_REGS of cause no register.
 */
#define REG_CLASS_CONTENTS  {{0}, {0x3ffffff}}

/* This macro should return the class of REGNO, here always ALL_REGS
 */
#define REGNO_REG_CLASS(REGNO)  ALL_REGS
```

Jedes Hard Register kann sowohl als Basis- und als Index-Register verwendet werden.

63b ⟨*Register-Eigenschaften* 61b⟩+≡ (56a) ◁63a

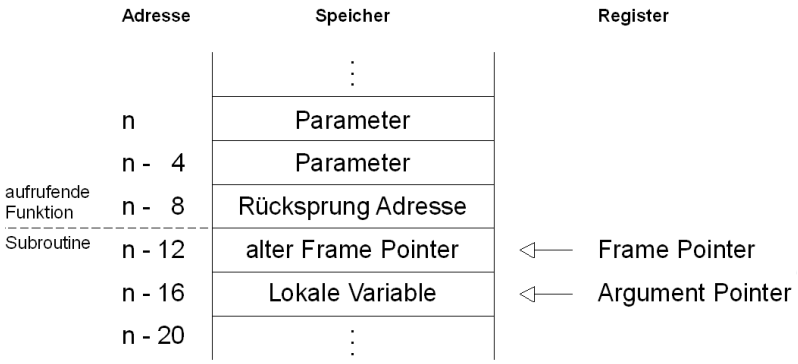
```
/* A valid base register must belong to this class.
 */
#define BASE_REG_CLASS ALL_REGS

/* The class, usable as index register.
 */
#define INDEX_REG_CLASS  ALL_REGS

/* These assume that REGNO is a hard or pseudo reg number.
```

```
   They give nonzero only if REGNO is a hard reg of the suitable class
   or a pseudo reg currently allocated to a suitable hard reg.
   Since they use reg_renumber, they are safe only once reg_renumber
   has been allocated, which happens in local-alloc.c.  */
#define REGNO_OK_FOR_INDEX_P(regno)     \
  ((regno) < FIRST_PSEUDO_REGISTER || reg_renumber[regno] >= 0)
#define REGNO_OK_FOR_BASE_P(regno)      \
  ((regno) < FIRST_PSEUDO_REGISTER || reg_renumber[regno] >= 0)
```

5.3.4 Stack- und Subroutinen-Eigenschaften

Jetzt werden die Eigenschaften, die mit dem Stack und den Subroutinen zu tun haben, festgelegt.

Für das Verständnis, wieso die Makros so definiert wurden hilft ein Schaubild des Stacks.



Der Stack wächst nach unten.

64a ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩≡ (56a) 64b▷

```
/* Stack grows downwards, from high adresses to low => decrease at push;
   increase at pop
 */
#define STACK_GROWS_DOWNWARD  1
```

Bei einem push wird das Postdekrement verwendet, da der Frame Pointer auf die nächste freie Stelle des Stacks zeigt.

64b ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁64a 65a▷

```
/* For a push you need a post decrement because the FP points to the
   next free memory location
 */
#define STACK_PUSH_CODE  POST_DEC
```

Auch im Stack-Frame wachsen die Daten nach unten, d.h. die lokalen Variablen haben eine niedrigere Adresse als der Framepointer.

65a ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁64b 65b▷

```
/* The Frame grows downwards
   The local variables have a lower adress than the frame-pointer
 */
#define FRAME_GROWS_DOWNWARD  1
```

Die erste lokale Variable kommt direkt nach dem Frame Pointer.

65b ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁65a 65c▷

```
/* Offset from the frame pointer to the first local variable slot
   to be allocated.*/
#define STARTING_FRAME_OFFSET  0
```

Die Offset zwischen dem ersten Parameter der der Funktion übergeben worden ist und dem Argument Pointer ist 12 Byte.

65c ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁65b 65d▷

```
/* Offset of first parameter from the argument pointer register value. */
#define FIRST_PARM_OFFSET(FNDECL) 12
```

Das Register 3 wird als Return Register verwendet, d.h. der Rückgabewert einer Funktion liegt immer im Register 3. Damit kann dann für `RETURN_ADDR_RTX` das RTL-Konstrukt erzeugt werden, in dem der Rückgabewert liegt.

65d ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁65c 66a▷

```
#define RETURN_REGNUM  3

/* A C expression whose value is RTL representing the value of the return
   address for the frame COUNT steps up from the current frame.  */
#define RETURN_ADDR_RTX(COUNT, FRAME)                                   \
  ((COUNT == 0)                                                         \
  ? get_hard_reg_initial_val (Pmode, RETURN_REGNUM)                     \
  : (rtx) 0)

/* The only valid return Register is the register with the number RETURN_REGNUM
 */
#define FUNCTION_VALUE_REGNO_P(N) ((N) == RETURN_REGNUM)

/* Define DEFAULT_PCC_STRUCT_RETURN to 1 if all structure and union return
   values must be in memory. */
#define DEFAULT_PCC_STRUCT_RETURN 0
```

Das Makro `FUNCTION_VALUE` soll ein RTX, also ein RTL Ausdruck ausgeben, in dem der Rückgabewert einer Funktion liegt. Übergeben wird dem Makro der Typ des Rückgabewerts und der Funktion.

66a ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁65d 66d▷
```
#define FUNCTION_VALUE(VALTYPE, FUNC) \
  function_value (VALTYPE, FUNC);
```

Das Makro verwendet die Funktion `function_value` aus der Datei `ulix.c` um das RTX zu generieren.

66b ⟨*function-value* 66b⟩≡ (82c)
```
rtx
function_value(const_tree type, const_tree func ATTRIBUTE_UNUSED)
{
  enum machine_mode mode;
  mode = TYPE_MODE (type);
  return ulix_function_value(mode);
}
```

Die Funktion fragt den Modus des Rückgabewerts ab und übergibt diesen an die Funktion `ulix_function_pointer`, die den RTX generiert.

66c ⟨*ulix-function-value* 66c⟩≡ (82c)
```
rtx
ulix_function_value(enum machine_mode mode){
  return gen_rtx_REG (mode, RETURN_REGNUM);
}
```

`ulix_function_value` übernimmt einen Modus und gibt ein RTX zurück, das über den selben Modus verfügt und auf das Return Register verweist.

Die Register-Nummer des Stack Pointers ist in `ulix.md` in die Konstante `SP_REGNUM` geschrieben werden. Dies wird jetzt auch als Makro festgelegt. Dadurch kann mit dem RTL-Ausdruck `(pc)` auf den Stack Pointer zugegriffen werden.

66d ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁66a 66e▷
```
/* Defines the number of the stack pointer
 */
#define STACK_POINTER_REGNUM    SP_REGNUM
```

In Ulix ist kein bestimmtes Register als Frame Pointer oder Argument Pointer vorgesehen. Deshalb definiere ich hier, das das Register 0 als Frame Pointer und das Register 1 als Argument Pointer verwendet wird. Außerdem lege ich fest, dass ein Argument Pointer benötigt wird.

66e ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁66d 67a▷
```
/* Frame Pointer
```

```
   Da es bei Ulix keinen festen FP gibt verwende ich hier 0
*/
#define FRAME_POINTER_REGNUM    0

/* Argument Pointer Register
   Da es keinen festen AP gibt verwende ich hier 1
*/
#define ARG_POINTER_REGNUM      1

/* Der Frame Pointer ist notwendig.
 */
#define FRAME_POINTER_REQUIRED  1
```

Alle Parameter, mit denen eine Subroutine aufgerufen wird, werden auf den Stack gepusht.

67a ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁66e 67b▷

```
/* Push insns will be used to pass outgoing arguments.
   The Ulix-CPU has a push-instruction, so this is 1*/
#define PUSH_ARGS  1

/* The number of bytes actually pushed onto the stack when
   an instruction attempts to push BYTES bytes. There is no
   rounding. Also odd-numbered Byte-Values could be pushed.*/
#define PUSH_ROUNDING(BYTES) (BYTES)

/* Define where to put the arguments to a function.
   Value is zero to push the argument on the stack,
   or a hard register in which to store the argument.

   MODE is the argument's machine mode.
   TYPE is the data type of the argument (as a tree).
    This is null for libcalls where that information may
    not be available.
   CUM is a variable of type CUMULATIVE_ARGS which gives info about
    the preceding args and about the function being called.
   NAMED is nonzero if this argument is a named parameter
    (otherwise it is an extra parameter matching an ellipsis).

   On the Ulix all args are pushed.  */
#define FUNCTION_ARG(CUM, MODE, TYPE, NAMED) 0
```

Nachdem von einer Unterprozedur zurückgesprungen wurde müssen keine Argumente vom Stack gepopt werden.

67b ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁67a 68a▷

```
/* Value is the number of byte of arguments automatically
   popped when returning from a subroutine call.
   FUNDECL is the declaration node of the function (as a tree),
```

```
   FUNTYPE is the data type of the function (as a tree),
   or for a library call it is an identifier node for the subroutine name.
   SIZE is the number of bytes of arguments passed on the stack.
 */
#define RETURN_POPS_ARGS(FUNDECL, FUNTYPE, SIZE)  0

/* 1 if N is a possible register number for function argument passing.
   On Ulix, no registers are used in this way.  */
#define FUNCTION_ARG_REGNO_P(N) 0
```

Jedes Register, das in einer Subroutine verwendet wird muss vorher auf den Stack gesichert werden.

68a ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁67b 68b▷

```
/* Always save/resore every register before/after each function
 */
#define CALLER_SAVE_PROFITABLE(refs, calls)  1
```

Hier wird ein RTX ausgegeben, das auf den Rückgabewert einer Library Funktion verweist. Momentan werden im ULIX-Betriebssystem keine Libraries verwendet, deshalb ist diese Funktion nicht wichtig. Da Rückgabewerte aber immer über das Return Register zurückgegeben werden, kann ich mit der Funktion `ulix_function_value` (siehe Seite 66) ohne großen Aufwand das RTX generieren.

68b ⟨*Stack- und Subroutinen-Eigenschaften* 64a⟩+≡ (56a) ◁68a

```
/* Define how to find the value returned by a library function
   assuming the value has mode MODE.  */
#define LIBCALL_VALUE(MODE)  \
  ulix_function_value (MODE)
```

5.3.5 Adressierung

Hier wird festgelegt welche Art der Adressierung valide ist.

In ULIX existiert keine Art von expliziter de- oder inkrementierender Adressierung. Bei keiner Adressierung treten Seiteneffekte auf. Nur `push` und `pop` verwenden implizit diese Adressierungsart.

68c ⟨*Adressierung* 68c⟩≡ (56a) 69a▷

```
/* Ulix do not support any kind of de-/incrementing addressing modes
 */
#define HAVE_PRE_INCREMENT  0

#define HAVE_PRE_DECREMENT  0

#define HAVE_POST_DECREMENT  0

#define HAVE_POST_DECREMENT  0
```

```
/* Ulix do not supports pre- or post-address side-effect generation
   involving constants other than the size of the memory operand */
#define HAVE_PRE_MODIFY_DISP  0

#define HAVE_POST_MODIFY_DISP  0
```

Jede valide konstante Ganzzahl kann auch als Adresse in ULIX verwendet werden.

69a ⟨*Adressierung* 68c⟩+≡ (56a) ◁68c 69b▷

```
/* Every valid constant is a valid constant address
 */
#define CONSTANT_ADDRESS_P(x)  CONSTANT_P(x)
```

In jeder Adressierung kann maximal ein Register vorkommen.

69b ⟨*Adressierung* 68c⟩+≡ (56a) ◁69a 69c▷

```
/* maximum number of registers that can appear in a valid memory address.
 */
#define MAX_REGS_PER_ADDRESS  1
```

Von `REG_OK_FOR_INDEX_P`, `REG_OK_FOR_BASE_P` und `GO_IF_LEGITIMATE_ADDRESS` existiert eine strikte und eine nicht strikte Variante. Die strikte Variante wird in der Reload-Phase verwendet (siehe Kaptitel 4.2.3 auf Seite 20). Diese strikten Makros werden also dazu verwendet für noch nicht allozierte Pseudo Register ein Hard Register zu finden in das sie vor der Verwendung geschrieben werden. In den strikten Varianten dürfen also nur Hard Register valide sein, während in den nicht strikten Varianten auch Pseudo Register valide sind.

`REG_OK_FOR_INDEX_P(X)` und `REG_OK_FOR_BASE_P(X)` geben an, ob das Register, dass in X beschrieben wird, ein gültiges Index- oder Basisregister ist.

`GO_IF_LEGITIMATE_ADDRESS(MODE, X, ADDR)` muss kontrollieren ob das RTX X mit dem Modus MODE eine gültige Speicher-Adresse beschreibt. Wenn dies zutrifft soll zum Label ADDR gesprungen werden.

69c ⟨*Adressierung* 68c⟩+≡ (56a) ◁69b 72a▷

```
#ifndef REG_OK_STRICT

/* Nonzero if X is a hard reg that can be used as an index
   or if it is a pseudo reg.  */
#define REG_OK_FOR_INDEX_P(X) 1

/* Nonzero if X is a hard reg that can be used as a base reg
   or if it is a pseudo reg.  */
#define REG_OK_FOR_BASE_P(X) 1

/* GO_IF_LEGITIMATE_ADDRESS recognizes an RTL expression
   that is a valid memory address for an instruction.  */
#define GO_IF_LEGITIMATE_ADDRESS(MODE, X, ADDR) \
```

```
  { if (legitimate_address_p ((MODE), (X), 0)) goto ADDR; }

#else //REG_OK_STRICT

/* Nonzero if X is a hard reg that can be used as an index.  */
#define REG_OK_FOR_INDEX_P(X) REGNO_OK_FOR_INDEX_P (REGNO (X))

/* Nonzero if X is a hard reg that can be used as a base reg.  */
#define REG_OK_FOR_BASE_P(X) REGNO_OK_FOR_BASE_P (REGNO (X))

/* GO_IF_LEGITIMATE_ADDRESS recognizes an RTL expression
   that is a valid memory address for an instruction.  */
#define GO_IF_LEGITIMATE_ADDRESS(MODE, X, ADDR) \
  { if (legitimate_address_p ((MODE), (X), 1)) goto ADDR; }

#endif //REG_OK_STRICT
```

Die Implementierung von `REG_OK_FOR_INDEX_P` und `REG_OK_FOR_BASE_P` sind trivial. Im nicht strikten Fall sind alle Register valide, im strikten Fall nur die Hard Register. Die auf Seite 63 definierten Funktionen `REGNO_OK_FOR_INDEX_P` und `REGNO_OK_FOR_BASE_P` testen ein Register, ob es ein Hard Register ist und wird deshalb hier verwendet.

Für `GO_IF_LEGITIMATE_ADDRESS` ist hier nur der Rahmen implementiert. Es wird die Funktion `legitimate_address_p` aus `ulix.c` aufgerufen, die prüft, ob eine Adresse valide ist. Trifft dies zu, so wird mit `goto ADDR` zum Label `ADDR` gesprungen.

Interessant ist hauptsächlich die Funktion `legitimate_address_p`. Hier wird festgelegt welche Adressierungsmodi in ULIX erlaubt sind. Dies sind die **absolute Adressierung**, **direkte Register-Adressierung** und **relative Adressierung**. Zusätzlich gibt es noch **Konstanten**, da diese aber keine Speicher-Adressierung darstellen, werden sie hier nicht berücksichtigt.

70 ⟨*legitimate-address-p* 70⟩≡ (82c)

```
/* legitimate_address_p returns 1 if it recognizes an RTL expression "x"
   that is a valid memory address for an instruction.
   The MODE argument is the machine mode for the MEM expression
   that wants to use this address.  */
int
legitimate_address_p (enum machine_mode mode ATTRIBUTE_UNUSED,
  rtx x, int strict){
  // Check for [#<address>]
  if(check_memory_immediate(x)) return 1;
  // Check for [<register> + offset]
  if(check_register_indirect(x, strict)) return 1;
  // Check for [r<register-number> + 0]
  if(check_register_direct(x, strict)) return 1;
  return 0;
}
```

`legitimate_address_p` gibt eine 1 zurück, wenn die Adresse valide ist, sonst eine 0. Um dies zu überprüfen werden Unterfunktionen verwendet, die jeweils eine Adressierungsart überprüfen: `check_memory_immediate` prüft auf absolute Adressierung, `check_register_indirect` auf indirekte Adressierung und `check_register_direct` prüft ob ein ein Register die Adresse vorhält.

71a ⟨*check-memory-immediate* 71a⟩≡ (82c)
```
static int check_memory_immediate(rtx x){
  return CONSTANT_ADDRESS_P(x);
}
```

Es wird überprüft ob das RTX X eine konstante Adresse ist. Dafür wird das auf Seite 69 definierte Makro `CONSTANT_ADDRESS_P` verwendet.

71b ⟨*check-register-indirect* 71b⟩≡ (82c)
```
static int check_register_indirect(rtx x, int strict){
  rtx reg;
  rtx cons;
  rtx temp;
  if (GET_CODE (x) != PLUS) return 0;
  reg = XEXP (x, 0);
  cons = XEXP (x, 1);
  if(!BASE_REGISTER_P(reg, strict)){
    temp = reg;
    reg = cons;
    cons = temp;
    if(!BASE_REGISTER_P(reg, strict)) return 0;
  }
  if(!CONSTANT_P(cons)) return 0;
  return 1;
}
```

Die Funktion `check_register_indirect` prüft ob das RTX x relative Adressierung verwendet. Relative Adressierung folgt in RTL diesem Schema: `(plus (reg ...) (const_int ...))`. Es wird also ein Register mit einer Konstante addiert. Die äußerste Verknüpfung muss also eine Plus RTX sein. Dies wird mit `GET_CODE (x) != PLUS` überprüft. Mit `XEXP` kann man die Operanden eines RTX bekommen. Es werden also die beiden Operanden des Plus RTX den Variablen `reg` und `cons` zugewiesen. Falls eine dieser RTX ein Register und das andere eine Konstante ist, handelt es sich um relative Adressierung.

71c ⟨*check-register-direct* 71c⟩≡ (82c)
```
static int check_register_direct(rtx x, int strict){
  if(REG_P (x)){
    if(!strict || REG_OK_FOR_BASE_P(X)) return 1;
  }
  return 0;
}
```

Mit `check_register_direct` wird getestet, ob es sich um direkte Adressierung mit Hilfe eines Registers handelt. Dazu wird geprüft ob das RTX x ein Register ist. Wenn es sich um die strikte Variante handelt wird noch zusätzlich geprüft ob x eine Hard Register ist.

In ULIX ist jede Konstante eine legitime Konstante.

72a ⟨*Adressierung* 68c⟩+≡ (56a) ◁69c 72b▷

```
#define LEGITIMATE_CONSTANT_P  CONSTANT_P
```

Gibt an, ob das RTX X ein valides Basis Register ist. Als Parameter wird zusätzlich angegeben, ob diese Prüfung strikt durchgeführt werden soll.

72b ⟨*Adressierung* 68c⟩+≡ (56a) ◁72a

```
/* Nonzero if X is a hard reg that can be used as a base reg
   or, if not strict, if it is a pseudo reg.  */
#define BASE_REGISTER_P(X, STRICT) \
(REG_P (X) && (!(STRICT) || REGNO_OK_FOR_BASE_P (REGNO (X))))
```

5.3.6 Meta-Assembler Anweisungen

Meta-Assembler Anweisungen sind Teile des Assemblers die keine Assembler-Befehle der Architektur sind, aber für das Assembler-Programm wichtig sind. Hier wird definiert, wie verschiedene Segmente des Assemblers gekennzeichnet sind oder woran man einen Kommentar im Assembler erkennen kann.

Ein Text-Segment enthält nur lesbare Daten. Alle Assembler-Befehle sind im Text-Segment. Das Text-Segment wird mit `.text` gekennzeichnet.

72c ⟨*Meta-Assembler Anweisungen* 72c⟩≡ (56a) 72d▷

```
/* Output before read-only data.  */
#define TEXT_SECTION_ASM_OP "\t.text"
```

Ein Daten-Segment enthält lese- und schreibbare Daten die bereits zur Compile-Zeit initialisiert sind. Das Daten-Segment wird mit `.data` gekennzeichnet.

72d ⟨*Meta-Assembler Anweisungen* 72c⟩+≡ (56a) ◁72c 72e▷

```
/* Output before writable (initialized) data.  */
#define DATA_SECTION_ASM_OP "\t.data"
```

Ein Bss-Segment enthält lese- und schreibbare Daten die zur Compile-Zeit noch nicht initialisiert sind. Das Bss-Segment wird mit `.bss` gekennzeichnet. Bei meinen Tests ist bisher nie ein Bss-Segment erstellt worden. Statt dessen wurden uninitialisierte Daten immer als Global Common Symbol definiert.

72e ⟨*Meta-Assembler Anweisungen* 72c⟩+≡ (56a) ◁72d 73a▷

```
/* Output before writable (uninitialized) data.  */
#define BSS_SECTION_ASM_OP "\t.bss"
```

Mit `.globl` wird ein Label global zugreifbar gemacht.

73a ⟨*Meta-Assembler Anweisungen* 72c⟩+≡ (56a) ◁72e 73b▷

```
/* Globalizing directive for a label.  */
#define GLOBAL_ASM_OP ".globl "
```

Kommentare beginnen im Assembler mit `;`

73b ⟨*Meta-Assembler Anweisungen* 72c⟩+≡ (56a) ◁73a 73c▷

```
/* String containing the assembler's comment-starter.  */
#define ASM_COMMENT_START ";"
```

Ein Global Common Symbol ist ein globales Label, das auf zur Compile-Zeit uninitialisierten Speicher zeigt. Es wird mit `.comm` *<name>*`,`*<size>* definiert. *name* ist der Name des Symbols und `size` die Größe in Bytes.

73c ⟨*Meta-Assembler Anweisungen* 72c⟩+≡ (56a) ◁73b 73d▷

```
/* This says how to output an assembler line
   to define a global common symbol. */
#define ASM_OUTPUT_COMMON(FILE, NAME, SIZE, ROUNDED)  \
( fputs (".comm ", (FILE)),                           \
  assemble_name ((FILE), (NAME)),                     \
  fprintf ((FILE), ",%u\n", (int)(ROUNDED)))
```

Ein Local Common Symbol ist ein lokales Label, das auf zur Compile-Zeit uninitialisierten Speicher zeigt. Es wird mit `.lcomm` *<name>*`,`*<size>* definiert. *name* ist der Name des Symbols und `size` die Größe in Bytes.

73d ⟨*Meta-Assembler Anweisungen* 72c⟩+≡ (56a) ◁73c 73e▷

```
/* This says how to output an assembler line
   to define a local common symbol. */
#define ASM_OUTPUT_LOCAL(FILE, NAME, SIZE, ROUNDED)  \
( fputs (".lcomm ", (FILE)),                         \
  assemble_name ((FILE), (NAME)),                    \
  fprintf ((FILE), ",%u\n", (int)(ROUNDED)))
```

Ein internes Label besteht aus dem Label-Präfix `L` und der Nummer des Labels. Hat das Label beispielsweise die Nummer 4, so heißt das Label `L4`. interne Label werden z.B. für Schleifen verwendet.

73e ⟨*Meta-Assembler Anweisungen* 72c⟩+≡ (56a) ◁73d

```
#define ASM_GENERATE_INTERNAL_LABEL(STRING, PREFIX, NUM)  \
  sprintf (STRING, "*%s%u", PREFIX, (unsigned int)(NUM))
```

5.3.7 Assembler Formatierung

In den Machine Descriptions ist zwar schon definiert, was für Assembler-Befehle generiert werden soll, doch wie die Operanden ausgegeben werden sollen wird erst hier implementiert.

Um die richtigen Register-Namen auszugeben, müssen sie dem GCC mit dem Makro `REGISTER_NAMES` bekannt gemacht werden. Das Makro definiert eine Liste der Register-Namen in der richtigen Reihenfolge. Durch dieses Makro ist es möglich das GCC-interne Array `reg_names` zuzugreifen um den Registernamen zu erhalten. `reg_names[16]` gibt z.B. `sp` aus.

74a ⟨*Assembler Formatierung* 74a⟩≡ (56a) 74b▷

```
/* This is a list of the register-names */
#define REGISTER_NAMES                                          \
{                                                               \
  "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7",               \
  "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15",         \
  "sp", "usp", "pc", "psw", "m", "z", "c1", "c2",               \
  "irr", "ier"                                                  \
}
```

Das Makro `PRINT_OPERAND` wird immer dann ausgeführt, wenn ein Operand ausgegeben werden soll. Die Operanden liegen bereits in einer validen Adressierung vor.

74b ⟨*Assembler Formatierung* 74a⟩+≡ (56a) ◁74a 75a▷

```
/* This Makro tells the compiler how an operand is printed
   in the different valid addressing-modes */
#define PRINT_OPERAND  print_operand
```

Das Makro verweist auf die Funktion `print_operand`, die in `ulix.c` definiert ist:

74c ⟨*print-operand* 74c⟩≡ (82c)

```
void print_operand(FILE *stream, rtx x, int code  ATTRIBUTE_UNUSED){
        rtx addr;
        rtx reg;
        rtx off;
        if (x == 0)
        {
          output_operand_lossage ("missing operand");
          return;
        }
        if ( CONSTANT_P (x) ){
                fprintf (stream, "#%d", XINT (x, 0));
        }
        else switch (GET_CODE (x)){
                case REG:
                        fprintf (stream, "%s", reg_names[REGNO (x)]);
                        break;
                case MEM:
```

```
                addr = XEXP (x, 0);
                /* call 'print_operand_address'  */
                output_address (addr);
                break;;
        case PLUS:
        case MINUS:
                output_address (x);
                break;;
        default: //ERROR
                //SIC, soll ein Fehler ergeben
                gcc_assert (0 == 1);
                break;
    }
}
```

Die Funktion `print_operand` erwartet, dass ihr der Operand als RTX x übergeben wird. Zuerst wird überprüft ob der Operand eine Konstante ist. in diesem Fall wird der Wert in x als Konstante in der Form `#<value>` ausgegeben. Ist x keine Konstante, so wird in einem switch-Statement getestet, ob es ein Register, eine Speicheradresse oder um ein PLUS- oder MINUS-RTX handelt. Ist es ein Register, so wird der Registername ausgegeben. Falls es sich um ein PLUS- oder MINUS-RTX handelt, so muss eine indirekte Speicher-Adressierung vorliegen. Deshalb wird genauso vorgegangen, wie wenn x eine Speicheradresse ist. Es wird die GCC-Funktion `output_address` aufgerufen, die wiederum `PRINT_OPERAND_ADDRESS` aufruft.

Trifft keiner der Fälle zu, so ist ein Fehler bei der Validierung aufgetreten und es wird ein Error ausgegeben.

75a ⟨*Assembler Formatierung* 74a⟩+≡ (56a) ◁74b

```
/* This Makro defines how a memory adress is printed in assembler */
#define PRINT_OPERAND_ADDRESS  print_operand_address
```

`PRINT_OPERAND_ADDRESS` verweist auf die Funktion `print_operand_address`, die in `ulix.c` implementiert ist.

75b ⟨*print-operand-address* 75b⟩≡ (82c)

```
void print_operand_address (FILE * file, rtx addr){
        rtx reg;
        rtx off;
        rtx temp;
        switch(GET_CODE (addr)){
                case REG:
                        fprintf(file, "[%s + 0]", reg_names[REGNO (addr)]);
                        break;
                case PLUS:
                        reg = XEXP (addr, 0);
                        off = XEXP (addr, 1);
                        if(GET_CODE(reg) != REG){
                                temp = off;
```

```
                        off = reg;
                        reg = temp;
                }
                fprintf (file, "[%s + %d]", reg_names[REGNO (reg)], XINT(off, 0));
                break;
        case MINUS:
                reg = XEXP (addr, 0);
                off = XEXP (addr, 1);
                if(GET_CODE(reg) != REG){
                        temp = off;
                        off = reg;
                        reg = temp;
                }
                fprintf (file, "[%s - %d]", reg_names[REGNO (reg)], XINT(off, 0));
                break;
        case SYMBOL_REF:
                fprintf (file, "%s", XSTR(addr, 0));
                break;
        default:
                if ( CONSTANT_P (addr) ){
                        fprintf (file, "[#%d]", XINT (addr, 0));
                        break;
                }else{
                        //ERROR
                        //SIC, soll ein Fehler ergeben
                        gcc_assert (0 == 1);
                        break;
                }
    }
}
```

Die Funktion `print_operand_address` erwartet, dass das ihr übergebene RTX `addr` eine Speicheradresse beinhaltet. Ist `addr` ein Register, so handelt es sich um indirekte Adressierung der Form `[<Register-Name> + #0]`. Es wird also auf eine Spezialform der relativen Adressierung abgebildet, bei der der Index Null ist. Handelt es sich bei `addr` um eine PLUS- oder MINUS-RTX, so wurde ebenfalls relative Adressierung verwendet. Falls `addr` ein Symbol ist, so wird einfach der Name des Labels ausgegeben.

Trifft keiner der Fälle zu, wird überprüft, ob `addr` eine Konstante ist. In diesem Fall ist es eine absolute Adressierung in der Form `[#<address>]`. Andernfalls ist ein Fehler bei der Validierung aufgetreten und es wird ein Error ausgegeben.

5.3.8 Verschiedenes

Es gibt noch einige Makros, die sich nicht so gut in eine Gruppe zusammenfassen lassen.

Jeder Wert ist in ULIX auch mit kleinerem Modus verwendbar. Zum Beispiel kann

ein `int` Wert ohne weiteres als `short` zu einem anderen Wert addiert werden. Zu beachten ist nur, dass dabei Genauigkeit verloren gehen kann.

77a ⟨*Verschiedenes* 77a⟩≡ (56a) 77b▷
```
/* nonzero if a value of mode mode1 is accessible in mode mode2 without copying.
 */
#define MODES_TIEABLE_P(mode1, mode2) 1

/* You could always work on an integer as if it would be a smaller integer.
   It is the normal way of truncation in Ulix.  */
#define TRULY_NOOP_TRUNCATION(outprec, inprec)  1
```

Das Standard-Verhalten ist hier erwünscht, da es sowieso nur eine Register-Klasse gibt.

77b ⟨*Verschiedenes* 77a⟩+≡ (56a) ◁77a 77c▷
```
/* Given an rtx X being reloaded into a reg required to be
   in class CLASS, return the class of reg to actually use.
   In general this is just CLASS; but on some machines
   in some cases it is preferable to use a more restrictive class.  */
#define PREFERRED_RELOAD_CLASS(X,CLASS)  (CLASS)
```

Jeder Wert passt in Ulix in jedes Register, da der größte Wert ein Single Integer mit 32 Bit ist und alle Register genau die selbe Größe haben.

77c ⟨*Verschiedenes* 77a⟩+≡ (56a) ◁77b 77d▷
```
/* Tests how many registers of class CLASS is needed to store a value of mode MODE.
 */
#define CLASS_MAX_NREGS(CLASS, MODE)  1
```

Alle Vergleiche können in Ulix ohne Probleme umgekehrt werden. Soll z.B. in einem if-Statement nach Gleichheit geprüft werden, funktioniert es auch nach Ungleichheit zu prüfen. Nur die Sprungziele müssen dann entsprechend vertauscht werde.

77d ⟨*Verschiedenes* 77a⟩+≡ (56a) ◁77c 77e▷
```
/* Return one if it is always safe to reverse a comparison whose mode is MODE.
   In Ulix it is always safe to reverse a comparision.  */
#define REVERSIBLE_CC_MODE(MODE) 1
```

In Ulix ist es ohne weiteres möglich bei eine Branch von einem Ende des Speichers zum anderen zu springen.

77e ⟨*Verschiedenes* 77a⟩+≡ (56a) ◁77d 78a▷
```
/* Indicate whether or not your architecture has conditional branches
   that can span all of memory. In Ulix it is allowed to make a conditional jump
   all over the memory.*/
#define HAS_LONG_COND_BRANCH 1
```

78a ⟨*Verschiedenes* 77a⟩+≡ (56a) ◁77e 78b▷
```
/* Indicate whether or not your architecture has unconditional branches
   that can span all of memory. In Ulix it is allowed to make a
   unconditional jump all over the memory.*/
#define HAS_LONG_UNCOND_BRANCH 1
```

Die größte Speichermenge die in ULIX in einer einzigen Instruktion kopiert werden kann sind 4 Byte, und zwar mit einem `move int`.

78b ⟨*Verschiedenes* 77a⟩+≡ (56a) ◁78a 78c▷
```
/* The maximum number of bytes that a single instruction can move quickly
   between memory and registers or between two memory locations.
   In Ulix this is a word/32bit/4byte */
#define MOVE_MAX 4
```

Da ULIX emuliert wird ist ein Speicherzugriff nicht schneller, wenn man auf Werte zugreift, die kleiner als ein Wort sind.

78c ⟨*Verschiedenes* 77a⟩+≡ (56a) ◁78b 78d▷
```
#define SLOW_BYTE_ACCESS  1
```

Hiermit wird verhindert, dass in jedem Binary in der main-Methode automatisch __main__ aufgerufen wird.

78d ⟨*Verschiedenes* 77a⟩+≡ (56a) ◁78c 78e▷
```
#define HAS_INIT_SECTION
```

78e ⟨*Verschiedenes* 77a⟩+≡ (56a) ◁78d
```
#undef  INIT_SECTION_ASM_OP
```

5.3.9 Dummys

Es gibt beim GCC einige Makros, die für ULIX nicht benötigt werden, aber trotzdem definiert werden müssen, damit der GCC kompiliert werden kann. Deshalb habe ich dafür Dummys erstellt.

78f ⟨*Dummys* 78f⟩≡ (56a)
```
/* Nested Functions werden nicht benötigt. GCC verlangt aber nach diesem Makro
 */
#define INITIALIZE_TRAMPOLINE(TRAMP, FNADDR, CXT)                       \
{                                                                       \
}

#define TRAMPOLINE_SIZE 16

/* Output an element of a dispatch table.
```

```
   In Ulix the addresses in a dispatch table are absolute*/
#define ASM_OUTPUT_ADDR_VEC_ELT(STREAM, VALUE)                  \
  do                                                            \
    {                                                           \
      asm_fprintf (STREAM, "\t.word\t%LL%d\n", VALUE);          \
    }                                                           \
  while (0)

#define ASM_OUTPUT_ALIGN(FILE,LOG)      \
    if ((LOG)!=0) fprintf ((FILE), "\t.align %d\n", 1<<(LOG))

/* This is how to output an assembler line
   that says to advance the location counter by SIZE bytes. */
#define ASM_OUTPUT_SKIP(FILE,SIZE)  \
  fprintf (FILE, "\t.space "HOST_WIDE_INT_PRINT_UNSIGNED"\n", (SIZE))

/* String constant for text to be output before each asm statement
   or group of consecutive ones */
#define ASM_APP_ON  "#APP"

/* String constant for text to be output after each asm statement
   or group of consecutive ones */
#define ASM_APP_OFF "#NO_APP"

/* Go to LABEL if ADDR (a legitimate address expression)
   has an effect that depends on the machine mode it is used for. */
#define GO_IF_MODE_DEPENDENT_ADDRESS(ADDR, LABEL)                       \
{                                                                       \
  if (   GET_CODE (ADDR) == PRE_DEC || GET_CODE (ADDR) == POST_DEC      \
      || GET_CODE (ADDR) == PRE_INC || GET_CODE (ADDR) == POST_INC)     \
    goto LABEL;                                                         \
}

/* Store in the variable DEPTH the initial difference between the
   frame pointer reg contents and the stack pointer reg contents,
   as of the start of the function body.  This depends on the layout
   of the fixed parts of the stack frame and on how registers are saved.
*/
#define INITIAL_FRAME_POINTER_OFFSET(DEPTH) (DEPTH) = 0;

/* Define a data type for recording info about an argument list
   during the scan of that argument list.  This data type should
   hold all necessary information about the function itself
   and about the args processed so far, enough to enable macros
   such as FUNCTION_ARG to determine where the next arg should go.

   In Ulix all arguments are passed on the stack, so there is no need
   to store anything in CUMULATIVE_ARGS; however, the data structure
   must exist and should not be empty, so I use int.  */
```

```
#define CUMULATIVE_ARGS int

/* Initialize a variable CUM of type CUMULATIVE_ARGS
   for a call to a function whose data type is FNTYPE.
   For a library call, FNTYPE is 0.

   On the Ulix-CPU, all parameter are pushed on the stack,
   so CUMULATIVE_ARGS are not needed.  */
#define INIT_CUMULATIVE_ARGS(CUM, FNTYPE, LIBNAME, INDIRECT, N_NAMED_ARGS) \
 ((CUM) = 0)

/* Update the data in CUM to advance over an argument
   of mode MODE and data type TYPE.
   (TYPE is null for libcalls where that information may not be available.)  */
#define FUNCTION_ARG_ADVANCE(CUM, MODE, TYPE, NAMED)    \
  ((CUM) += ((MODE) != BLKmode                          \
             ? (GET_MODE_SIZE (MODE))           \
             : (int_size_in_bytes (TYPE))))

/* Profiling wird nicht benötigt, GCC verlangt aber nach diesem Makro */
#define FUNCTION_PROFILER(STREAM, LABELNO)                              \
{                                                                       \
}
```

Es existieren noch sehr viel mehr Target Description Macros die alle im GCC Internals Manual[?] beschrieben sind. Doch werden für ULIX nicht alle benötigt oder ihr Standardverhalten ist bereits zutreffend.

5.4 Sonstige Implementationen

5.4.1 ulix.c

Zusätzlich zu der Datei `ulix.h` existiert auch noch die Datei `ulix.c` in der viele Methoden, die in `ulix.h` verwendet werden ausprogrammiert sind. Im Kapitel 5.3 wurden die verschiedenen Methoden auch schon besprochen.

Jetzt soll nur noch der Aufbau von `ulix.c` dokumentiert werden.

80a ⟨*ulix.c* 80a⟩≡
- ⟨*Lizenz von ulix.c* 80b⟩
- ⟨*Includes von ulix.c* 81⟩
- ⟨*Initialisierung von Datenstrukturen* 82a⟩
- ⟨*Funktions Prototypen* 82b⟩
- ⟨*Funktionen* 82c⟩

Auch `ulix.c` enthält wieder einen Verweis auf die GPL[8] unter der sie veröffentlicht ist.

80b ⟨*Lizenz von ulix.c* 80b⟩≡ (80a)

```
/* Output Routine for GCC for Ulix.
   Copyright (C) 1991, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001,
   2002, 2003, 2004, 2005, 2006, 2007 Free Software Foundation, Inc.
   Ulix-Description by Balthasar Biedermann (balthasar@biedermann.es).

   This file is part of GCC.

   GCC is free software; you can redistribute it and/or modify it
   under the terms of the GNU General Public License as published
   by the Free Software Foundation; either version 3, or (at your
   option) any later version.

   GCC is distributed in the hope that it will be useful, but WITHOUT
   ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
   or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public
   License for more details.

   You should have received a copy of the GNU General Public License
   along with GCC; see the file COPYING3.  If not see
   <http://www.gnu.org/licenses/>.  */
```

In `ulix.c` müssen einige Header eingebunden werden, die in den Funktionen verwendet werden.

81 ⟨*Includes von ulix.c* 81⟩≡ (80a)

```
#include "config.h"
#include "system.h"
#include "coretypes.h"
#include "tm.h"
#include "rtl.h"
#include "tree.h"
#include "regs.h"
#include "hard-reg-set.h"
#include "real.h"
#include "insn-config.h"
#include "conditions.h"
#include "function.h"
#include "output.h"
#include "insn-attr.h"
#include "recog.h"
#include "expr.h"
#include "optabs.h"
#include "flags.h"
#include "debug.h"
#include "toplev.h"
#include "tm_p.h"
#include "target.h"
#include "target-def.h"
#include "basic-block.h"
```

```
#include "ggc.h"
#include "langhooks.h"
```

Das struct `targetm` wird von vielen verwendeten Funktionen, die der GCC zur Verfügung stellt verwendet weswegen dies zwingend benötigt wird. Hier wird es initialisiert.

82a ⟨*Initialisierung von Datenstrukturen* 82a⟩≡ (80a)

```
struct gcc_target targetm = TARGET_INITIALIZER;
```

Jede in `ulix.c` implementierte Funktion hat einen Funktions Prototyp.

82b ⟨*Funktions Prototypen* 82b⟩≡ (80a)

```
rtx ulix_function_value(enum machine_mode mode);
rtx function_value(const_tree type, const_tree func);
static int check_memory_immediate(rtx x);
static int check_register_indirect(rtx x, int strict);
static int check_register_direct(rtx x, int strict);
int legitimate_address_p (enum machine_mode mode, rtx x, int strict);
void print_operand(FILE *stream, rtx x, int code);
void print_operand_address (FILE * file, rtx addr);
void ulix_expand_call(rtx retval, rtx mem);
void output_function_prologue(void);
void output_function_epilogue(void);
static rtx emit_set_insn (rtx, rtx);
rtx arm_compare_op0, arm_compare_op1;
```

Alle Funktionen im Überblick.

82c ⟨*Funktionen* 82c⟩≡ (80a)

⟨*ulix-function-value* 66c⟩

⟨*function-value* 66b⟩

⟨*output-function-prologue* 53b⟩

⟨*output-function-epilogue* 54b⟩

⟨*check-memory-immediate* 71a⟩

⟨*check-register-indirect* 71b⟩

⟨*check-register-direct* 71c⟩

⟨*legitimate-address-p* 70⟩

⟨*ulix-expand-call* 52a⟩

⟨*print-operand* 74c⟩

⟨*print-operand-address* 75b⟩

5.4.2 Integration von Ulix als Target

Damit ULIX als mögliches Target, also Zielarchitektur, erkannt wird, müssen noch vier Dateien angepasst werden.

config.sub

In die Datei `config.sub` muss in den Abschnitt, der mit `case $basic_machine` in beginnt noch ULIX als neue Basic Machine eingetragen werden. Dies geschieht durch das Einfügen folgender Zeilen:

```
        ulix)
                basic_machine=ulix-unknown
                ;;
```

Die Zielarchitektur lautet damit `ulix-unknown`.

configure.ac

In die Datei `configure.ac` muss ULIX eingetragen werden, damit beim Aufruf von `configure` mit dem Parameter `-target=ulix` oder einem ähnlichen Aufruf, klar ist, was zu tun ist.

In den Abschnitt, der mit `case $targetin` beginnt muss ULIX als mögliches Target eingetragen werden:

```
  ulix-*-*)
    noconfigdirs="$noconfigdirs"
    libgloss_dir=ulix
    ;;
```

Durch diese Zeilen wird jedes Target, dass mit `ulix` beginnt erkannt.

gcc/config.gcc

In der Datei `gcc/config.gcc` muss in den Abschnitt, der mit `case $target in` beginnt, folgendes hinzugefügt werden:

```
ulix-*-*)
        cpu_type=ulix
        ;;
```

5.4.3 Patching der GCC

Um ULIX schnell und einfach in den GCC zu integrieren habe ich mit dem Unix Program `diff` einen Patch generiert und ein Skript geschrieben, das die GCC Dateien patcht und den ULIX Ordner nach `gcc/config/` kopiert.

Der Patch, der die Dateien `config.sub`, `configure.ac` und `gcc/config.gcc` mithilfe des Unix Programms `patch` automatisch anpasst sieht folgendermaßen aus:

84 ⟨*ulix-gcc.diff* 84⟩≡

```
diff -urp gcc-4.3.0/config.sub ../svn/ulix/ulix-compiler/gcc-4.3.0/config.sub
-- gcc-4.3.0/config.sub        2008-01-23 03:37:40.000000000 +0100
+++ ../svn/ulix/ulix-compiler/gcc-4.3.0/config.sub      2008-06-22 18:47:00.000000000 +0200
@ -1074,6 +1074,9 @@ case $basic_machine in
                basic_machine=a29k-amd
                os=-udi
                ;;
+       ulix)
+               basic_machine=ulix-unknown
+               ;;
        ultra3)
                basic_machine=a29k-nyu
                os=-sym1
diff -urp gcc-4.3.0/configure.ac ../svn/ulix/ulix-compiler/gcc-4.3.0/configure.ac
-- gcc-4.3.0/configure.ac      2008-02-02 04:29:30.000000000 +0100
+++ ../svn/ulix/ulix-compiler/gcc-4.3.0/configure.ac    2008-06-22 18:51:34.000000000 +0200
@ -856,6 +856,10 @@ case "${target}" in
   spu-*-*)
     skipdirs="target-libssp"
     ;;
+  ulix-*-*)
+    noconfigdirs="$noconfigdirs"
+    libgloss_dir=ulix
+    ;;
   v810-*-*)
     noconfigdirs="$noconfigdirs bfd binutils gas gcc gdb ld target-libstdc++-v3 opcodes target-libgl
     ;;
diff -urp gcc-4.3.0/gcc/config.gcc ../svn/ulix/ulix-compiler/gcc-4.3.0/gcc/config.gcc
-- gcc-4.3.0/gcc/config.gcc    2008-01-29 17:28:10.000000000 +0100
+++ ../svn/ulix/ulix-compiler/gcc-4.3.0/gcc/config.gcc  2008-06-24 17:17:47.000000000 +0200
@ -2514,6 +2514,9 @@ strongarm-*-kaos*)
        md_file=arm/arm.md
        extra_modes=arm/arm-modes.def
        ;;
+ulix-*-*)
+       cpu_type=ulix
+       ;;
 v850e1-*-*)
        target_cpu_default="TARGET_CPU_v850e1"
        tm_file="dbxelf.h elfos.h svr4.h v850/v850.h"
```

Die Datei führt die oben genannten Änderungen durch. Dafür werden die ungepatchten Sources des GCC in der Version 4.3.0 benötigt.

Das Skript, dass automatisch alle Änderungen am GCC einpflegt erwartet im selben Verzeichnis den Unterordner `ulix`, der die Dateien `ulix.md`, `ulix.c`, `ulix.h` und `t-ulix` enthält.

85 ⟨*patch-gcc-for-ulix.sh* 85⟩≡

```
#!/bin/sh

if [ $# != 1 ]
then
        echo "Error: Wrong number of arguments!"
        echo "Usage:"
        echo "$0 <path to gcc 4.3.0 directory>"
else
        gcc_dir=‘readlink -f $1‘
        script_path=‘dirname $0‘
        script_path=‘readlink -f $script_path‘
        echo "copying ulix/ to $gcc_dir/gcc/config/"
        cp -a $script_path/ulix $gcc_dir/gcc/config/
        cd $gcc_dir
        patch -p1 < $script_path/ulix-gcc.diff
fi
```

Das Skript erwartet als Parameter den Pfad zum ungepatchten GCC. Zuerst kopiert es den `ulix` Ordner ins GCC Unterverzeichnis `gcc/config/`. Danach benutzt er den Patch um die GCC-Dateien, die verändert werden müssen, anzupassen.

Damit ist es einfach und schnell möglich den GCC für ULIX fit zu machen.

6 Evaluation

6.1 Ulix-GCC Build Prozess

Zuerst müssen die GCC-Sources der Version 4.3.0 mit dem in Kapitel 5.4.3 besprochenen Verfahren für Ulix gepatcht werden. Dann muss die GCC-Sources mit Ulix als Target kompiliert werden. Dafür verwendete ich folgenden Aufruf:

```
./configure --target=ulix --enable-languages=c
make all-gcc install-gcc
```

Der `make`-Befehl muss mehrere Male aufgerufen werden, da zuerst die benötigten Libraries gebaut werden. Wenn man den Ulix-GCC nicht ins Standardverzeichnis installieren will, so muss man beim `./configure`-Befehl noch `--prefix=<Pfad>` angeben. Der hier verwendete Pfad ist dann das Wurzelverzeichnis für die Ulix-GCC Installation. Während der Phase, als das Ulix-Backend noch entwickelt wurde habe ich beim `make` Befehl noch die Parameter `BOOT_CFLAGS='-O0 -g3' CFLAGS=g3 -O0"` angehängt. Dadurch ist der Compiler bei Fehlern leichter zu debuggen. Außerdem kann mit `-j<Anzahl Jobs>` die Anzahl der parallelen Jobs angegeben werden. Bei einem Mehrkernprozessor oder bei Verwendung eines Compile-Clusters kann man damit die Dauer des Kompiliervorgangs immens verkürzen. Ich habe beispielsweise ein mit `icecc`[10] aufgebautes Compile Cluster bestehend aus einem Core2Duo und ein bis zwei Single Core Notebooks verwendet. Dadurch konnte die doch ziemlich lange Zeit für den Kompiliervorgang etwas verringert werden. Da ich für die Entwicklung des Backends sehr oft einen Clean Build des GCC vornehmen musste hat ccache[3] diese Vorgänge zusätzlich beschleunigt.

Der GCC Compiler Driver liegt nach dem erfolgreichen Kompiliervorgang im Verzeichnis `/bin` und trägt den Namen `ulix-gcc`. Dadurch kann auch ohne weiteres der normale GCC und der Ulix-GCC parallel verwendet werden. Der eigentliche C Compiler `cc1` liegt in `/libexec/gcc/ulix/4.3.0`. Gerade zum Debuggen sollte man statt dem Compiler Driver lieber direkt `cc1` verwenden.

6.2 Test-Dateien

Nachdem der Ulix-GCC erfolgreich kompiliert wurde habe ich zur Evaluation Testdateien in C geschrieben, die elementare Bestandteile des Ulix-Backends testen. Da das Assembler-Programm für Ulix, zu der Zeit in der ich den Compiler entwickle, noch nicht fertig gestellt ist, kann ich die generierten Assembler-Programme noch nicht ausführen. Die einzige Möglichkeit die Ergebnisse zu prüfen ist den Assembler-Text manuell durchzulesen und auf mögliche Fehler zu prüfen. Eine weitere Schwierigkeit

besteht darin, dass für ULIX keine Standard-Library besteht. Die meisten Quelltexte, die zur Verfügung stehen benötigen aber zumindest die Standard Libraries und können deshalb nicht kompiliert werden. Die einzige Möglichkeit zur Validierung des Compilers besteht also darin kleine C-Programme zu schreiben, die keine Libraries verwenden. Sind die Programme zu groß oder komplex ist eine manuelle Überprüfung des Assemblercodes nur mit erheblichem Aufwand zu bewältigen. Im Rahmen dieser Bachelor-Thesis ist es also nicht möglich festzustellen ob komplexe Programme, zu denen das ULIX-Betriebssystem gehört, erfolgreich und richtig kompiliert werden. Die Tests müssen sich also auf die Grundfunktionen beschränken.

Der GCC versucht, wenn man ihn nur mit dem zu kompilierenden Programm aufruft, ein ausführbares Programm zu erstellen. Da aber noch kein ULIX-Assembler Programm existiert, wird dies fehlschlagen. Man muss den GCC also dazu veranlassen nach dem eigentlichen Kompiliervorgang zu stoppen und den Assembler-Code auszugeben. Dies geschieht indem man folgenden Befehl ausführt:

```
./ulix-gcc -S Datei.c
```

Der Parameter `-S` sorgt dafür, dass der GCC nur Assembler-Code produziert. Die Assembler-Datei lautet hier `Datei.s`. Alle Tests wurden ohne Optimierung kompiliert.

6.2.1 Test der arithmetischen Befehle

Zuerst werde ich die arithmetischen Befehle testen und überprüfen, ob der ULIX-GCC die richtigen ULIX-Befehle generiert. Dafür verwende ich ein kleines C-Programm, das zwei Integer-Werte **addiert**, **subtrahiert**, **dividiert**, **multipliziert** und ihr **Modulo** berechnet.

87a ⟨*arithmetic.c* 87a⟩≡

```
int main(void){
        int a = 23;
        int b = 42;
        int c = a + b;
        c = a - b;
        c = a / b;
        c = a * b;
        c = a % b;
}
```

Der Assembler-Code sieht folgendermaßen aus:

87b ⟨*arithmetic.s* 87b⟩≡

```
        .text
.globl main
main:
        push   int      r0
        add    int      r0, sp, #4
        sub    int      sp, sp, #28
        sub    int      r1, r0, #4
```

```
move    int     [r0 + -12], #23
move    int     [r0 + -8], #42
add     int     [r0 + -4], [r0 + -12], [r0 + -8]
sub     int     [r0 + -4], [r0 + -12], [r0 + -8]
div     int     [r0 + -20], [r0 + -12], [r0 + -8]
move    int     [r0 + -4], [r0 + -20]Zuerst werde
mul     int     [r0 + -4], [r0 + -12], [r0 + -8]
move    int     [r0 + -24], [r0 + -12]
mod     int     [r0 + -28], [r0 + -24], [r0 + -8]
move    int     [r0 + -4], [r0 + -28]
move    int     sp, r0
move    int     r0, [r0 + 0]
rts
```

Für dieses Programm wird nur ein Text-Segment benötigt, das mit `.text` eingeleitet wird. Das Label `main` global gemacht und das Label definiert.Die main-Funktion muss global zugreifbar sein, sonst würde das Programm nicht starten können.

Danach beginnt das eigentliche Program. Da die `main`-Funktion auch eine Subroutine ist wird also zuerst das Funktions-Prologue (siehe Kaptitel 5.2.19 auf Seite 53) generiert. Es wird also der alte Frame-Pointer auf dem Stack gesichert (`push int r0`) und dann das Frame-Pointer Register `r0` ausgerichtet (`add int r0, sp, #4`). Der Stack Pointer wird so verändert, dass sich der Stack und die lokalen Variablen nicht in die Quere kommen (`sub int sp, sp, #28`) und danach wird der Argument Pointer, `r1` ausgerichtet (`sub int r1, r0, #4`). Damit ist der Prologue der Funktion fertig. In diesem Fall müssen keine Register gesichert werden, da die Funktion keine besonderen verwendet.

Mit `move int [r0 + -12], #23` beginnt der erste Befehl, der in der C-Datei in der main-Funktion programmiert wurde. Es werden die beiden Werte in den Speicher geschrieben. Danach werden jeweils die arithmetischen Befehle `add`, `sub`, `div`, `mul` und `mod` ausgeführt. Es passiert also genau das, was von dem Programm erwartet wurde. Der GCC generiert allerdings noch einige nicht erforderlichen move-Befehle. Mit eingeschalteter Optimierung wären diese Befehle wegoptimiert worden.

`move int [r0 + -4], [r0 + -28]` ist der letzte Befehl der in der C-Datei programmiert wurde. Danach folgt der Epilogue der Funktion (siehe Kaptitel 5.2.19 auf Seite 54). Dieser korrigiert den Stack Pointer so, dass das `rts` richtig ausgeführt werden kann (`move int sp, r0`) und stellt den alten Frame Pointer wieder her (`move int r0, [r0 + 0]`). Eigentlich existiert kein alter Frame Pointer, da die main-Funktion die erste Funktion ist, die ausgeführt wird. Der GCC behandelt die main-Funktion aber genauso wie jede andere auch. Zuletzt wird noch der `rts` Befehl ausgeführt.

6.2.2 Test der logischen Befehle

Jetzt werden die logischen Befehle getestet. Das dafür verwendete C-Programm, initialisiert zwei Integer-Werte, die es dann mit dem **logischen Und**, dem **logischen**

inklusiven Oder und dem **logischen exklusiven Oder** verknüpft. Zuletzt wird noch das **logische Nicht**, also das bitweise Inverse, des ersten Integers gebildet.

89a ⟨*logic.c* 89a⟩≡
```
int main(void){
        int a = 23;
        int b = 42;
        int c = a & b;
        c = a | b;
        c = a ^ b;
        c = ~a;
}
```

Der erzeugte Assembler-Code sieht folgendermaßen aus:

89b ⟨*logic.s* 89b⟩≡
```
        .text
.globl main
main:
        push   int      r0
        add    int      r0, sp, #4
        sub    int      sp, sp, #16
        sub    int      r1, r0, #4
        move   int      [r0 + -12], #23
        move   int      [r0 + -8], #42
        and    int      [r0 + -4], [r0 + -12], [r0 + -8]
        or     int      [r0 + -4], [r0 + -12], [r0 + -8]
        xor    int      [r0 + -4], [r0 + -12], [r0 + -8]
        not    int      [r0 + -4], [r0 + -12]
        move   int      sp, r0
        move   int      r0, [r0 + 0]
        rts
```

Es werden erst wieder die Werte in den Speicher geschrieben. Danach wird der `and`, `or`, `xor` und der `not`-Befehl ausgeführt. Es funktioniert also alles wie gewünscht.

6.2.3 Test der Shift-Befehle

Jetzt werden die Shift Befehle getestet. Das dafür verwendete C-Programm initialisiert einen Integer-Werte, den es zuerst um fünf nach rechts und dann um vier nach links shiftet.

89c ⟨*shift.c* 89c⟩≡
```
int main(void){
        int i = 42;
        int j = i » 5;
        j = i « 4;
}
```

Der erzeugte Assembler-Code sieht folgendermaßen aus:

90a ⟨*shift.s* 90a⟩≡

```
        .text
.globl main
main:
        push   int      r0
        add    int      r0, sp, #4
        sub    int      sp, sp, #12
        sub    int      r1, r0, #4
        move   int      [r0 + -8], #42
        asr    int      [r0 + -4], [r0 + -8], #5
        sl     int      [r0 + -4], [r0 + -8], #4
        move   int      sp, r0
        move   int      r0, [r0 + 0]
        rts
```

Als erstes wird der Wert in den Speicher geschrieben. Jetzt wird der `asr` und dann der `sl`-Befehl ausgeführt. Da der Integer-Wert vorzeichenbehaftet war, wird der arithmetische Rechts-Shift verwendet. Sonst wäre der logische Rechts-Shift verwendet worden.

6.2.4 Test von Cast-Operationen

Als nächstes wird getestet ob Up- und Down-Casts richtig durchgeführt werden. Bei den Up-Casts muss zwischen einem vorzeichenbehafteten und einem vorzeichenlosen Up-Cast unterschieden werden.

Signed Cast

Der vorzeichenbehaftete Cast wird mit `signed char` bis `signed int` durchgeführt. Zuerst wird ein `signed char` initialisiert. Danach wird er einem `signed short` zugewiesen. Dabei wird implizit ein Up-Cast durchgeführt. Genauso kommt es zu einem Up-Cast, wenn der `signed short` dem `signed int` und wenn der `signed char` dem `signed int` zugewiesen wird. Danach werden noch alle Werte explizit einem Down-Cast unterzogen.

90b ⟨*castSigned.c* 90b⟩≡

```
int main(void){
        signed char c = 4;
        signed short s = c;
        signed int i = s;
        i = c;
        s = (signed short) i;
        c = (signed char) s;
        c = (signed char) i;
}
```

Der erzeugte Assembler-Code sieht folgendermaßen aus:

91a ⟨*castSigned.s* 91a⟩≡

```
        .text
.globl main
main:
        push   int      r0
        add    int      r0, sp, #4
        sub    int      sp, sp, #21
        sub    int      r1, r0, #4
        move   byte     [r0 + -7], #4
        upcast short    [r0 + -6], byte [r0 + -7]
        upcast int      [r0 + -4], short [r0 + -6]
        upcast int      [r0 + -4], byte [r0 + -7]
        move   int      [r0 + -15], [r0 + -4]
        move   short    [r0 + -6], [r0 + -13]
        move   short    [r0 + -17], [r0 + -6]
        move   byte     [r0 + -7], [r0 + -16]
        move   int      [r0 + -21], [r0 + -4]
        move   byte     [r0 + -7], [r0 + -18]
        move   int      sp, r0
        move   int      r0, [r0 + 0]
        rts
```

Der `byte`-Wert wird in des Speicher geschrieben. Jetzt wird ein `upcast short <Wert>, byte <Wert>` durchgeführt. Dann ein `upcast int <Wert>, short <Wert>` und ein `upcast int <Wert>, byte <Wert>`. Das sind also genau die Up-Casts, die implizit im C-Program vorkommen. Die Down-Casts sind normale move-Befehle, die als Operanden-Größe die Ziel-Größe des Casts haben. Alle Casts funktionieren wie erwartet.

Unsigned Cast

Diesmal werden vorzeichenlose Ganzzahlen verwendet. Dabei ist zu erwarten, dass der Down-Cast genauso funktioniert, während sich der Up-Cast unterscheidet.

91b ⟨*castUnsigned.c* 91b⟩≡

```
int main(void){
        unsigned char c = 4;
        unsigned short s = c;
        unsigned int i = s;
        i = c;
        s = (unsigned short) i;
        c = (unsigned char) s;
        c = (unsigned char) i;
}
```

Der erzeugte Assembler-Code sieht folgendermaßen aus:

92 ⟨*castUnsigned.s* 92⟩≡

```
        .text
.globl main
main:
        push   int      r0
        add    int      r0, sp, #4
        sub    int      sp, sp, #21
        sub    int      r1, r0, #4
        move   byte     [r0 + -7], #4
        move   short    [r0 + -6], #0
        move   byte     [r0 + -6], [r0 + -7]
        move   int      [r0 + -4], #0
        move   short    [r0 + -4], [r0 + -6]
        move   int      [r0 + -4], #0
        move   byte     [r0 + -4], [r0 + -7]
        move   int      [r0 + -15], [r0 + -4]
        move   short    [r0 + -6], [r0 + -13]
        move   short    [r0 + -17], [r0 + -6]
        move   byte     [r0 + -7], [r0 + -16]
        move   int      [r0 + -21], [r0 + -4]
        move   byte     [r0 + -7], [r0 + -18]
        move   int      sp, r0
        move   int      r0, [r0 + 0]
        rts
```

Die Up-Casts sind hier jeweils zwei aufeinander folgende Befehle. Beispielhaft erläutere ich den Cast von `char` nach `short`.

```
        move   byte     [r0 + -7], #4
        move   short    [r0 + -6], #0
        move   byte     [r0 + -6], [r0 + -7]
```

Zuerst wird der Wert 4 als `byte` in den Speicher geschrieben. Danach wird die Null als `short`, also in der Ziel-Größe, in die Ziel-Speicherstelle geschrieben. Danach wird ganz normal der im Speicher befindliche `byte`-Wert in die Ziel-Speicherstelle kopiert werden.

Der Down-Cast funktioniert analog zum vorzeichenbehafteten Down-Cast.

6.2.5 Test von Vergleichs-Operationen

Die Vergleichs-Operationen werden hier anhand von `if` Statements geprüft. Dabei ist zwischen vorzeichenbehafteten und vorzeichenlosen Vergleichen zu unterscheiden.

Vergleiche von vorzeichenbehafteten Werten

In diesem Beispiel-Programm werden zwei Integer `i` und `j` initialisiert. Dann wird nacheinander getestet ob `i` **gleich** `j` ist, ob `i` **nicht gleich** `j` ist, ob `i` **kleiner als** `j`

ist, ob `i` **größer als** `j` ist, ob `i` **größer gleich** `j` ist und ob `i` **kleiner gleich** `j` ist.

93a ⟨*compareSigned.c* 93a⟩≡

```
int main(void){
        int i = 5;
        int j = 6;
        if(i == j) j++;
        else i-;
        if(i != j) j++;
        else i-;
        if(i < j) j++;
        else i-;
        if(i > j) j++;
        else i-;
        if(i >= j) j++;
        else i-;
        if(i <= j) j++;
        else i++;
}
```

Der erzeugte Assembler-Code sieht folgendermaßen aus:

93b ⟨*compareSigned.s* 93b⟩≡

```
        .text
.globl main
main:
        push   int      r0
        add    int      r0, sp, #4
        sub    int      sp, sp, #12
        sub    int      r1, r0, #4
        move   int      [r0 + -8], #5
        move   int      [r0 + -4], #6
        cmp    int      [r0 + -8], [r0 + -4]
        jne             L2
        add    int      [r0 + -4], [r0 + -4], #1
        jmp             L3
L2:
        add    int      [r0 + -8], [r0 + -8], #-1
L3:
        cmp    int      [r0 + -8], [r0 + -4]
        jeq             L4
        add    int      [r0 + -4], [r0 + -4], #1
        jmp             L5
L4:
        add    int      [r0 + -8], [r0 + -8], #-1
L5:
        cmp    int      [r0 + -8], [r0 + -4]
        jsge            L6
        add    int      [r0 + -4], [r0 + -4], #1
```

```
        jmp             L7
L6:
        add     int     [r0 + -8], [r0 + -8], #-1
L7:
        cmp     int     [r0 + -8], [r0 + -4]
        jsle            L8
        add     int     [r0 + -4], [r0 + -4], #1
        jmp             L9
L8:
        add     int     [r0 + -8], [r0 + -8], #-1
L9:
        cmp     int     [r0 + -8], [r0 + -4]
        jsl             L10
        add     int     [r0 + -4], [r0 + -4], #1
        jmp             L11
L10:
        add     int     [r0 + -8], [r0 + -8], #-1
L11:
        cmp     int     [r0 + -8], [r0 + -4]
        jsg             L12
        add     int     [r0 + -4], [r0 + -4], #1
        jmp             L15
L12:
        add     int     [r0 + -8], [r0 + -8], #1
L15:
        move    int     sp, r0
        move    int     r0, [r0 + 0]
        rts
```

Ich werde die Vergleich beispielhaft an der **Prüfung auf Gleichheit** analysieren:

```
        move    int     [r0 + -8], #5
        move    int     [r0 + -4], #6
        cmp     int     [r0 + -8], [r0 + -4]
        jne             L2
        add     int     [r0 + -4], [r0 + -4], #1
        jmp             L3
L2:
        add     int     [r0 + -8], [r0 + -8], #-1
L3:
```

Zuerst werden beide Werte in den Speicher geschrieben. Danach werden sie mit dem `cmp`-Befehl verglichen und die Flags des Program Status Words werden entsprechend gesetzt. jetzt kommt ein `jne`, also ein Conditional Branch, der dann ausgeführt wird, wenn die beiden verglichenen werte *nicht gleich* sind. Dies ist zuerst einmal verwirrend, da doch auf Gleichheit getestet werden soll. Doch der GCC weiß, dass er alle Vergleiche auch ohne Probleme invertieren kann (vergleiche Kapitel 5.3.8, Seite 77). Also schauen wir uns nochmal den entscheidenden Code-Abschnitt des C-Programms an:

```
        if(i == j) j++;
        else i--;
```

Sind beide Werte gleich sind, so soll j inkrementiert werden, ansonsten soll i dekrementiert werden.

Wenn beide Werte nicht gleich sind, so wird zum Label `L2` gesprungen. Der Befehl nach `L2` ist ein `add`-Befehl, jedoch mit der negativen Konstanten `#-1`. i wird also dekrementiert. Dies ist genau das Verhalten, das erwünscht ist.

Sind beide Werte gleich, so wird nicht gesprungen, sondern es wird der nächste Befehl ausgeführt. Hier wird ein `add` mit der Konstanten `#1` ausgeführt, es wird also wie erwartet j inkrementiert. Danach wird mit `jmp L3` unbedingt nach L3 gesprungen, wodurch das Dekrement übersprungen wird. Der Assembler verhält sich also wie erwünscht.

Die nachfolgenden Vergleich laufen analog ab. Auch hier werden die invertierten Vergleiche verwendet: `jeq`, `jsge`, `jsle`, `jsl` und `jsg`. Alle Vergleiche sind signed, also die vorzeichenbehaftete Variante der Conditional Branches.

Vergleiche von vorzeichenlosen Werten

Das Program, das die vorzeichenlosen Vergleiche testet ist analog zum eben besprochenen aufgebaut. Nur werden eben `unsigned int` verwendet.

95a ⟨*compareUnsigned.c* 95a⟩≡

```
int main(void){
        unsigned int i = 5;
        unsigned int j = 6;
        if(i == j) j++;
        else i-;
        if(i != j) j++;
        else i-;
        if(i < j) j++;
        else i-;
        if(i > j) j++;
        else i-;
        if(i >= j) j++;
        else i-;
        if(i <= j) j++;
        else i++;
}
```

Zu erwarten ist, dass der Code genauso aussieht, wie beim vorzeichenbehafteten Vergleich, nur das die vorzeichenlosen Conditional Branches verwendet werden. Der erzeugte Assembler-Code sieht folgendermaßen aus:

95b ⟨*compareUnsigned.s* 95b⟩≡

```
        .text
.globl main
main:
```

```
        push    int     r0
        add     int     r0, sp, #4
        sub     int     sp, sp, #12
        sub     int     r1, r0, #4
        move    int     [r0 + -8], #5
        move    int     [r0 + -4], #6
        cmp     int     [r0 + -8], [r0 + -4]
        jne             L2
        add     int     [r0 + -4], [r0 + -4], #1
        jmp             L3
L2:
        add     int     [r0 + -8], [r0 + -8], #-1
L3:
        cmp     int     [r0 + -8], [r0 + -4]
        jeq             L4
        add     int     [r0 + -4], [r0 + -4], #1
        jmp             L5
L4:
        add     int     [r0 + -8], [r0 + -8], #-1
L5:
        cmp     int     [r0 + -8], [r0 + -4]
        jge             L6
        add     int     [r0 + -4], [r0 + -4], #1
        jmp             L7
L6:
        add     int     [r0 + -8], [r0 + -8], #-1
L7:
        cmp     int     [r0 + -8], [r0 + -4]
        jle             L8
        add     int     [r0 + -4], [r0 + -4], #1
        jmp             L9
L8:
        add     int     [r0 + -8], [r0 + -8], #-1
L9:
        cmp     int     [r0 + -8], [r0 + -4]
        jl              L10
        add     int     [r0 + -4], [r0 + -4], #1
        jmp             L11
L10:
        add     int     [r0 + -8], [r0 + -8], #-1
L11:
        cmp     int     [r0 + -8], [r0 + -4]
        jg              L12
        add     int     [r0 + -4], [r0 + -4], #1
        jmp             L15
L12:
        add     int     [r0 + -8], [r0 + -8], #1
L15:
        move    int     sp, r0
```

```
move    int       r0, [r0 + 0]
rts
```

Der Code sieht wie erwartet aus. Die Vergleiche auf Gleichheit und Ungleichheit sind die selben wie im vorhergehenden beispiel, da sie sowohl für signed- als auch unsigned-Werte gleich sind. Die restlichen Conditional Branches sind die vorzeichenlose Varianten: `jge`, `jle`, `jl` und `jg`.

Schleifen funktionieren ebenso mit Hilfe von Vergleichen und sollten also äquivalent funktionieren.

7 Zusammenfassung und Ausblick

Das letzte Kapitel fasst noch einmal alles was erreicht wurde zusammen und zeigt ebenso auf, was nicht möglich war. Zudem wird noch ein Ausblick auf die mögliche Weiterentwicklung und Verbesserung des ULIX-Compilers gegeben.

7.1 Zusammenfassung

Die wichtigste Aufgabe war es, einen lauffähigen Compiler für ULIX zu implementieren. Dies ist mit der Anpassung des GCC Backends geglückt. Alle Test-Dateien werden kompiliert und erzeugen validen Code. Soweit es momentan möglich ist, wurde die Qualität des Assembler-Codes evaluiert. Dabei ist aber darauf hinzuweisen, dass es noch nicht möglich ist, größere Tests durchzuführen, da dafür noch der Assembler für ULIX fehlt. Deshalb kann abschließend noch nicht davon ausgegangen werden, dass der ULIX-GCC fehlerfrei ist, auch wenn bisher kein Fehler nachgewiesen werden kann. Der erzeugte Code erreicht auch nicht annähernd die Qualität des i386 Assemblers. Da der Code aber nicht besonders performant sein muss, sollte die Qualität den Anforderungen genügen. Es ist sogar, auch wenn es nicht gefordert war, möglich den Code zu optimieren. Dabei wird zwar keine auf die Architektur abgestimmte Optimierung durchgeführt, aber auch die architekturunabhängige Optimierung, die der GCC bietet ist schon beachtlich. Nach meiner Erfahrung wird der Code beim Optimierungslevel 1 (Parameter `-O1`) sogar leichter zu verstehen, da nicht benötigte `move` Befehle wegoptimiert werden.

Zusammenfassend lässt sich sagen, dass alle Mindestanforderungen erfüllt werden. Sogar die optionalen Anforderungen werden teilweise erfüllt oder können mit überschaubarem Aufwand eingepflegt werden: Da der GCC viele Eingabesprachen unterstützt, sollte es möglich sein auch andere Sprachen als C zu kompilieren. Gerade C++ sollte ohne großen Aufwand unterstützt werden. Ich konnte diese Funktionalität aber noch nicht testen. Da es der GCC auch unterstützt Debugging-Informationen einzupflegen wäre es also auch möglich im ULIX-Emulator den C-Code anzuzeigen und wie in einem Debugger befehlsweise durchzugehen. Doch die dafür nötige Anpassung müsste noch eingepflegt werden. Aber die Grundsteine dafür sind gelegt. Der GCC sollte sich auf vielen unterschiedlichen Architekturen verwenden lassen. Das neue Backend hat keine tiefgreifenden Veränderungen im Quellcode hervorgerufen. Dadurch sollte auch der ULIX-GCC auf allen Plattformen funktionieren, auf denen auch der normale GCC lauffähig ist.

7.2 Ausblick

Der ULIX-GCC wird sich noch vielen Bewährungsproben stellen müssen. Die erste Aufgabe, die nach der Fertigstellung dieser Arbeit noch auf den ULIX-GCC zukommen wird, ist es die Kompatibilität mit dem ULIX-Assembler herzustellen. Am elegantesten wäre es natürlich, wenn der Assembler direkt vom Compiler Driver `gcc` angesprochen werden würde und somit nicht mehr explizit aufgerufen werden muss. Ob und wie dies möglich ist wird nach der Fertigstellung des Assembler-Programms noch entschieden werden.

Wenn der Compiler und das Assembler-Programm dann fehlerfrei zusammenarbeiten wird das ULIX-Betriebssystem kompiliert werden. Dies ist das erste Programm, das über ein paar Zeilen C-Code hinausgeht und mit dem ULIX-GCC kompiliert wird. Es ist also davon auszugehen, dass dabei noch nicht entdeckte Fehler auftreten werden.

Da der GCC stetig weiter entwickelt wird ergibt sich daraus die Möglichkeit auch den ULIX-Compiler von den Verbesserungen des GCC profitieren zu lassen. Dadurch wird es in Zukunft vielleicht nötig werden das Backend an Veränderungen im GCC-Quelltext anzupassen. Wie schnell diese Änderung vonstatten geht sieht man schon an der GCC Version, die für den ULIX-GCC verwendet wurde. Dies ist die Version 4.3.0. Sie war aktuell, als ich mit dem Projekt ULIX-Compiler begonnen habe. Im Moment liegt der GCC in Version 4.3.2 vor.

Literaturverzeichnis

[1] Bison - GNU parser generator. Internet: `http://www.gnu.org/software/bison/`, 2007.

[2] Cygwin. Internet: `http://www.cygwin.com/`, 2007.

[3] ccache - compiler cache. Internet: `http://ccache.samba.org/`, 2008.

[4] The flex project. Internet: `http://flex.sourceforge.net/`, 2008.

[5] GCC internals - GIMPLE. Internet: `http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html`, 2008.

[6] GCC internals manual. Internet: `http://gcc.gnu.org/onlinedocs/gccint/`, 2008. Version: 4.3.0.

[7] GCC, the GNU compiler collection. Internet: `http://gcc.gnu.org/`, 2008.

[8] The GNU general public license. Internet: `http://www.gnu.org/licenses/`, 2008.

[9] The GNU project. Internet: `http://www.gnu.org/`, 2008.

[10] Icecream compile cluster. Internet: `http://en.opensuse.org/Icecream`, 2008.

[11] Open watcom C/C++ compiler. Internet: `http://www.openwatcom.org`, 2008.

[12] pcc - portable c compiler. Internet: `http://pcc.ludd.ltu.se/`, 2008.

[13] Nadine Benedum. Der ULIX assembler. Bachelorarbeit, Universität Mannheim, 2008.

[14] Christopher Fraser and David Hansen. *A Retargetable C Compiler: Design and Implementation.* Addison Wesley Pub Co Inc, 1995.

[15] Felix C. Freiling. *The Design and Implementation of the ULIX Operating System.* 2008.

[16] Ralf Hund. The ULIX cpu. Seminararbeit, Universität Mannheim, August 2008.