Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Entwurf und Implementierung eines SPARC Backend

Studienarbeit von

Hannes A. Rapp

an der Fakultät für Informatik

Gutachter:	Prof. Dr.-Ing. Gregor Snelting
Betreuender Mitarbeiter:	Dipl.-Inform. Matthias Braun

Bearbeitungszeit: 1. Mai 2010 – 4. August 2010

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

SPARC, *Scalable Processor Architecture*, ist eine von Sun Microsystems[sun] seit 1985 entwickelte Prozessorarchitektur. Durch die Tatsache, dass die Spezifikation und Hardwarebeschreibung der Architektur unter einer Open-Source Lizenz[ope] verfügbar sind, gewinnt SPARC in den letzten Jahren, vor allem im naturwissenschaftlichen Bereich, sowie in der Luft- und Raumfahrttechnik, zunehmend an Bedeutung. Durch die, im Vergleich zu den bisher verfügbaren Architekturen in LIBFIRM[fir], hohe Anzahl an verfügbaren Registern und Features wie delayed branches (Sprungbefehle, die mehr als einen Zyklus benötigen), bietet ein SPARC Backend die Möglichkeit um z.B. neue Algorithmen für Registerallokation zu untersuchen und mit bestehenden zu vergleichen.

1.2 Ziel

Das Ziel dieser Arbeit ist die Implementierung eines SPARC Backends für die LIBFIRM[fir] Compilersuite, welches die grundlegenden Funktionen, wie z.B. Steuerfluss und Integerarithmetik, unterstützt. Zur Evaluierung wurde der erzeugte Assembler-Code kompiliert und anschließend auf dem freien Prozessoremulator QEMU[qem] gestartet. Ein erster wichtiger Meilenstein ist der 164.gzip[164] Test aus der SPEC CPU2000[spe] Benchmark Suite.

1.3 Struktur der Arbeit

Zuerst werden Grundlagen über SSA-Form[RWZ88], Firm[Lin02] allgemein und die einzelnen Backendphasen von LIBFIRM beschrieben und erklärt. Anschließend erfolgt eine Beschreibung der SPARC Architektur und deren Besonderheiten. Auf Basis dieser Informationen wird dann detailiert auf die einzlenen Implementierungen der SPARC spezifischen Funktionen eingegangen, bevor sich das letzte Kapitel zukünftigen Verbesserungen und einem Ausblick für Weiterentwicklungen widmet.

2 Grundlagen

2.1 SSA - Static Single Assignment

Die SSA-Form ist eine Programmrepräsentation. Sie wurde von Rosen, Wegman und Zadeck[RWZ88] 1988 vorgeschlagen und wird heute in modernen Compilern wie der *GNU Compiler Collection*[gcc], *llvm*[llv] und LIBFIRM eingesetzt.

Die grundlegende Idee ist, dass jeder Variablen im Programm **statisch** nur einmal ein Wert zugewiesen wird. Damit ist die Definition einer Variablen eindeutig bestimmt und erleichtert unter anderem einige Optimierungs- und Analysetechniken erheblich, die bisher mit aufwändigen Datenflussanalysen bewerkstelligt werden mussten. Listing 2.1 zeigt einen Codeauszug, wie man ihn häufig in Quellcodes findet. Dieser ist nicht in SSA Form, da mehrfach eine Zuweisung an Variable x erfolgt.

```
1  int f(int x)
2  {
3    int i = 42;
4    x = x + i;
5
6    if (x > 4) {
7      x = i / 2;
8      x = x++;
9    } else {
10     x = 0;
11   }
12
13   return x;
```

Listing 2.1: nicht in SSA-Form

Man erreicht die SSA-Form, indem man für im Quelltext auftretende Zuweisungen jeweils einen neuen (meist werden die Variablen durchnummeriert), eindeutigen Bezeichner einführt. Diese Zuweisungen können zwar während eines Programmablaufs mehrfach durchlaufen werden, sind aber statisch gesehen eindeutig, da es sich jeweils um dieselbe Instruktion handelt.

```
1  int f(int x_0)
2  {
3    int i_0 = 42;
4    x_1 = x_0 + i;
5
6    if (x_1 > 4) {
7      x_2 = i_0 / 2;
8      x_3 = x_2++;
9    } else {
10     x_4 = 0;
11   }
12
```

```
13    return φ(x_3,x_4);
```

Listing 2.2: in SSA-Form

Listing 2.2 zeigt, wie man das zuvor angeführte Beispiel in SSA-Form formulieren würde.
Häufig kommt es vor, dass der Steuerfluss zweier Grundblöcke zusammenläuft und es dann zu einer (statisch) nicht mehr eindeutigen Zuweisung kommen würde. Ursache dafür ist, dass in den Vorgängerblöcken verschiedene Definitionen der ursprünglich zugrundeliegenden Variablen vorliegen können. Welche Definition nun tatsächlich erreicht wird, hängt vom konkreten Steuerfluss ab. Dieses Problem wird durch das Einführen einer speziellen ϕ-Funktion gelöst. Diese ist so definiert, dass sie je nach vorliegendem Steuerfluss die entsprechende korrekte Definition auswählt. Im in Listing 2.2 gezeigten Beispiel, das sich in SSA-Form befindet, wird beim Verlassen der Funktion entweder `x_3` oder `x_4` zurückgeliefert, abhängig davon, ob zuvor der `if` oder `else` Zweig ausgeführt wurde. Da es sich bei diesem Hilfskonstrukt offensichtlich nicht um eine reale Instruktion, die von Prozessoren implementiert wird, handelt, muss sie vor der Codegenerierung entsprechend behandelt und aufgelöst werden.
Ein effizienter Algorithmus zur Konstruktion einer SSA Darstellung wird z.b. von Cytron et. ali.[CFR+91] vorgestellt.

2.2 Firm

Firm[Lin02] ist eine am Lehrstuhl für Programmierparadigmen der Universität Karlsruhe entwickelte, graphbasierte SSA Zwischendarstellung von Programmen. LIBFIRM[fir] ist eine Implementierung dieser Darstellung in C. Im Unterschied zu anderen SSA-basierten Compilern, behält LIBFIRM bis zuletzt diese graphbasierte SSA-Darstellung bei. Alle durchlaufenen Phasen arbeiten auf diesem Graphen und erst mit der Codeausgabe wird diese Darstellung verlassen. Im folgenden werden nur auf die für diese Arbeit relevanten Details von Firm eingegangen.

2.2.1 Firm Methodengraph

Der Methodengraph repräsentiert den Programmcode einer einzelnen Funktion der Hochsprache in Firm-Darstellung. Es ist ein Steuerflussgraph, bestehend aus den einzelnen Grundblöcken und speziell ausgezeichneten Start- und Endblöcken. Die Grundblöcke enthalten jeweils die einzelnen Instruktionsknoten, die mit Abhängigkeitskanten verbunden sind. Diese Abhängigkeitskanten implizieren eine Halbordnung[hal] auf der Menge der Instruktionen. Das bedeutet, dass außer den Abhängigkeiten keine konkrete Reihenfolge der einzelnen Befehle mehr gegeben ist. Die ursprüngliche Reihenfolge, die sich aus dem Programmcode ergab, ist nichtmehr von Bedeutung. Diese Darstellung ist folglich sehr abstrakt und bietet dadurch die Möglichkeit für viele sprach- und architekturneutrale Optimierungen.

In Listing 2.3 ist ein, in Anlehnung an das bereits in Listing 2.1 gezeigte Beispiel, kleines Programm gegeben. Abbildung 2.1 zeigt den zugehörigen, Methodengraphen der `main` Funktion.

```
1  static int globalVar = 1;
2  extern int printf(const char *str, ...);
3
4  int __attribute__((noinline)) f(int x)
5  {
6    int i = 42;
7    i += 2;
8    return i;
9  }
10
11
12 int main(int argc, char **argv)
13 {
14   globalVar = 42;
15   globalVar = rand();
16
17   if (globalVar > 8) {
18     globalVar = f(8);
19     printf("if branch\n");
20   } else {
21     printf("else branch\n");
22   }
23 }
```

Listing 2.3: einfaches Beispielprogramm

2.3 Phasen eines LIBFIRM Backends

Um von der abstrakten Darstellung letztendlich zur Codeausgabe kommen zu können, wird der Graph in verschiedenen Phasen immer näher an die Struktur der Zielarchitektur angepasst. Abbildung 2.1 zeigt diesen abstrakten Ausgangsgraphen der `main` Funktion aus Listing 2.3. Neben den nachfolgend beschriebenen Phasen der Codeauswahl, Befehlsanordnung, Registerallokation und Codeausgabe existieren noch weitere Phasen zur Berechnung von relativen Stackadressen und Verifizierung von Bedingungen bestimmter Knoten, auf die im Rahmen dieser dieser Arbeit allerdings nicht näher eingangen wird.

2.3.1 Codeauswahl

Die Codeauswahl erzeugt aus den architekturneutralen Instruktionsknoten die entsprechenden konkreten Instruktionsknoten der jeweiligen Architektur. Dabei können oft mehrere dieser neutralen Knoten zu einer einzigen Instruktion auf der Architektur zusammengefasst werden. Abbildung 2.2 zeigt den FIRM-Graphen der `main` Funktion nach der Codeauswahl. Aus Gründen der Übersichtlichkeit, ist nur der erste Grundblock des gesamten Graphen abgebildet. Er beinhaltet den SPARC-Prolog (s. Unterabschnitt 3.4.3) und das Laden der (symbolischen) Konstanten

2.3.2 Befehlsanordnung

Bei der Befehlsanordnung werden in dem Graphen zusätzliche Kanten erzeugt, welche eine gültige Ausführungsreihenfolge der einzelnen Befehle neu festlegt, die bei der Transformation in die abstrakte graphbasierte (s. Unterabschnitt 2.2.1) Darstellung aufgegeben wurde. Im Allgemeinen existieren mehrere gültige Ausführungsreihenfolgen, wobei *gültig* bedeutet, dass die Semantik des ursprünglichen Programms unverändert bleibt. Abbildung 2.3 zeigt wiederum den ersten Grundblock des gesamten Graphen mit den neuen Kanten der Befehlsanordnung in der Farbe Magenta.

2.3.3 Registerallokation

Hier werden den Instruktionsknoten der Architektur die benötigten Register zugewiesen. Dies ist ein sehr komplexer Vorgang, da dieser Instruktionen teilweise über spezielle Bedingungen zur Vergabe der Register verfügen können, die beachtet werden müssen. Außerdem muss an einigen Stellen Auslagerungs-Code erzeugt werden, um Register frei zu machen und Werte zu sichern.

2.3.4 Codeausgabe

Diese Phase durchläuft den endgültigen Graphen entlang der während der Befehlsanordnung neu erzeugten Kanten und gibt den Assemblercode für die einzelnen Knoten aus.

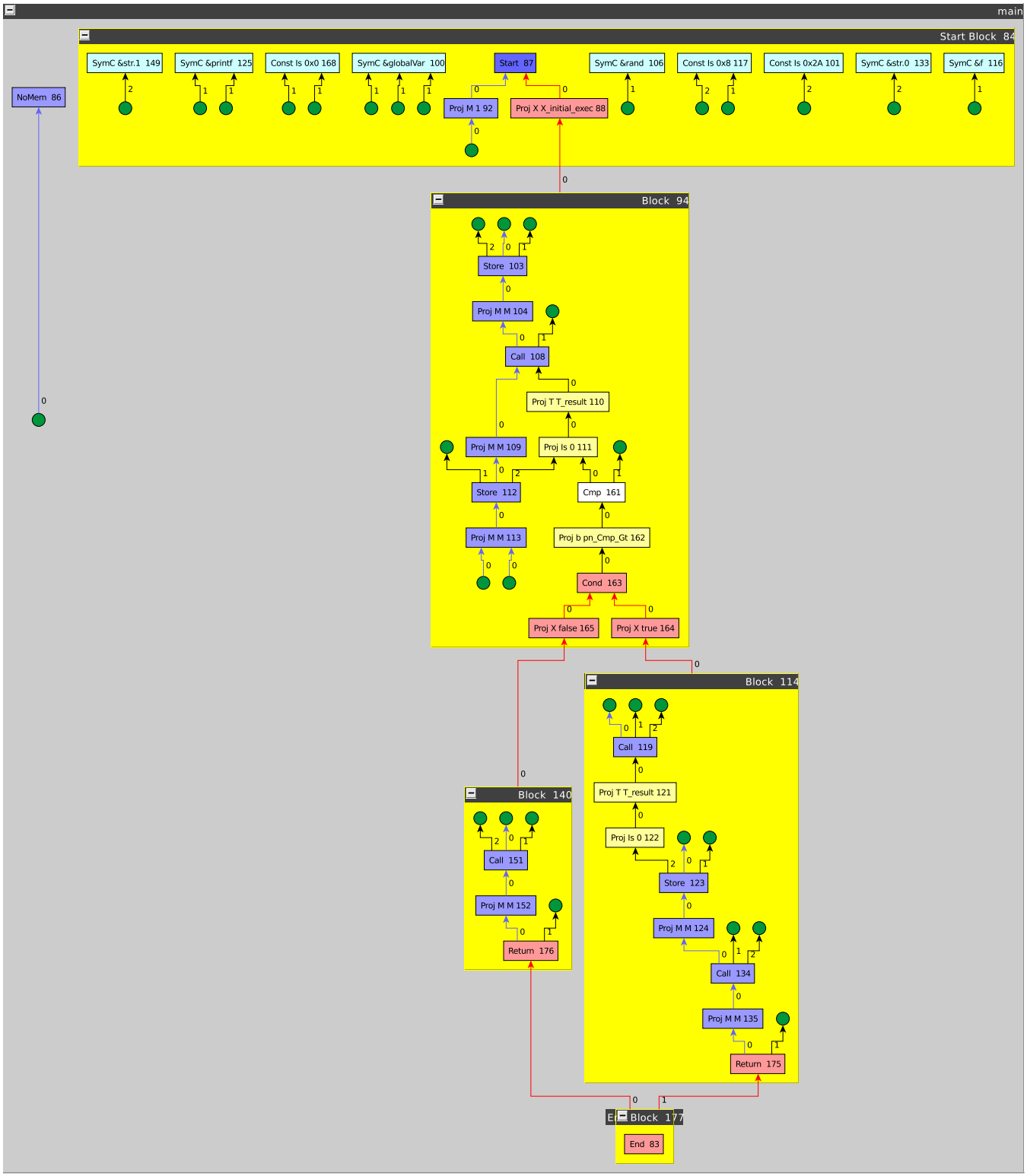

Abbildung 2.1: abstrakter Methodengraph der main-Funktion

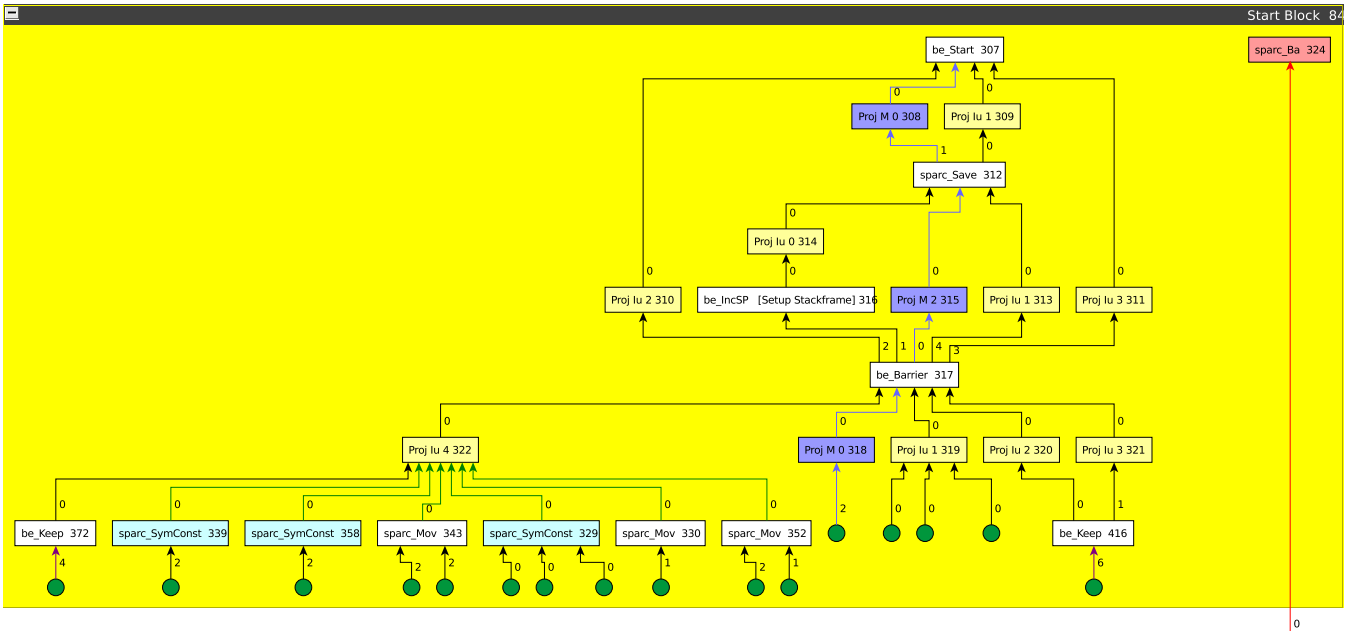

Abbildung 2.2: Prolog der main-Funktion nach Codeauswahl

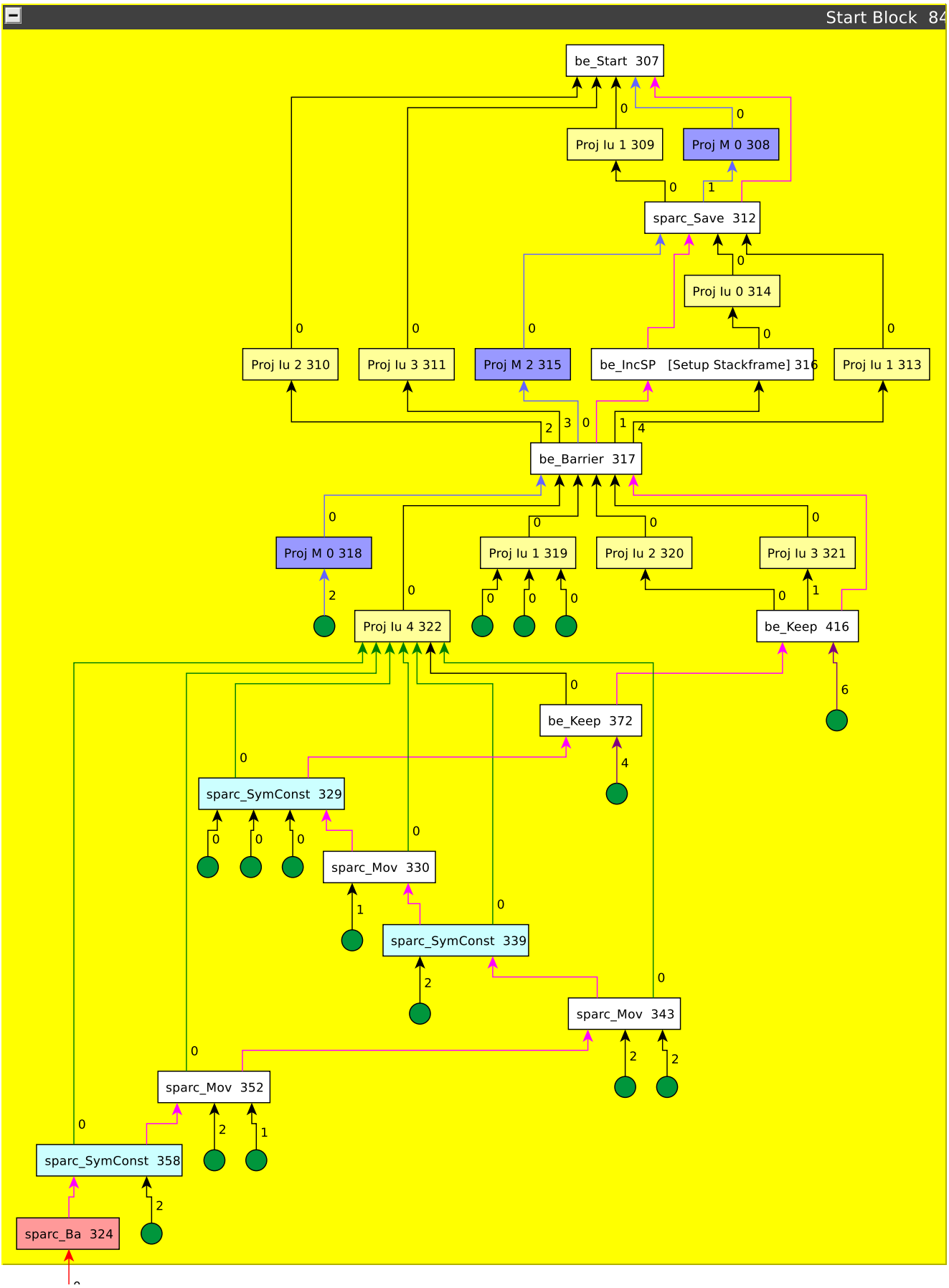

Abbildung 2.3: Prolog der main-Funktion nach Befehlsanordnung

3 SPARC V8 Architektur

3.1 Überblick, RISC

SPARC V8 ist eine 32bit, load-store RISC (*reduced instruction set computer*) Prozessorarchitektur. Das bedeutet, dass sie die folgenden zwei wesentlichen Eigenschaften besitzt:

Load-Store Architektur Es greifen ausschließlich spezielle Lade- und Speicherinstruktionen auf den Speicher zu. Alle übrigen Instruktionen arbeiten auf Registern.

umfangreicher Registersatz Aufgrund der Tatsache, dass nur einfache Instruktionen zur Verfügung stehen, steigt die Anzahl der erzeugten Zwischenergebnisse. Um diese nicht jedesmal in den Hauptspeicher zurückschreiben zu müssen, stehen, im Vergleich zur CISC-Architektur (*complex instruction set computer*), deutlich mehr Register zur Verfügung.

Weitere Charakteristika der SPARC Architektur sind eine feste Instruktionsbreite von 4 Bytes, eine streng strukturierte Aufrufkonvention, die auf Registerfenstern basiert, und ein Pipeline-Mechanismus mit verzögerten Instruktionen zur Steuerung des Kontrollflusses.
Zur Abgrenzung des Begriffs Funktion zwischen Hochsprache und Assemblerebene, bezeichnet der Begriff *Routine* im folgenden eine "'Funktion"' auf Assembler Ebene, *Subroutine* eine darin aufgerufene "'Unterfunktion"'.

3.2 Register

Die SPARC Architektur bietet 32 General Purpose Register (im Weiteren als GP Register bezeichnet) für 32bit Integer Operationen, sowie 32 Floating-Point Register und einige Status-Register. Tatsächlich verfügt SPARC über wesentlich mehr GP Register, typischerweise sind es 128, maximal 512. Allerdings stehen einer Routine durch das Registerfenster während der Ausführung nur 32 davon zur Verfügung.

3.3 Das Registerfenster

Als Registerfenster wird der Mechanismus bezeichnet, der bei Eintritt in eine Routine, ausgenommen optimierte Blatt-Routinen, den Zugriff auf einen Satz von 24 Registern steuert. Diese 24 Register sind eine Teilmenge der 32 insgesamt zur Verfügung stehenden GP Register und stehen jeder aufgerufenen Routine separat zur Verfügung. Sie sind lokal pro Routine verfügbar und in etwa vergleichbar mit lokalen Variablen in Funktionen oder Methoden einer Hochsprache. Zusätzlich stehen 8 Register global zur Verfügung, deren Verwendung sich alle Routinen teilen. Ebenso wird die Verwendung der 32 Floating-Point Register global von allen Routinen geteilt. Aus der, je nach Prozessorimplementierung, verfügbaren tatsächlichen Anzahl zwischen 128 und 512 GP Registern, ergibt sich eine Anzahl von Registerfenstern zwischen minimal 2 und maximal 32.

Der *Current Window Pointer* (im folgenden `CWP` genannt) zeigt jweils auf einen aktuellen Satz dieser 24 Register, welche aufgeteilt sind in jeweils acht Input-, Output- und lokale Register. Außer den zusätzlich verfügbaren 8 globalen GP Registern, sind alle übrigen GP Register für die aktuelle Routine nicht sichtbar.

Diese insgesamt 32 GP Register sind wie folgt eingeteilt:

%g0 - %g7 Acht global für alle Routinen verfügbare Register, die ihrerseits wiederum eine spezielle Sematik besitzen:

- **%g0** liefert immer den Wert 0. Das Schreiben eines Wertes in dieses Register hat keinen Effekt.
- **%g1** temporäre / globale Variable 1
- **%g2** globale Variable 2
- **%g3** globale Variable 3
- **%g4** globale Variable 4
- **%g5** reserviert von SPARC ABI
- **%g6** reserviert von SPARC ABI
- **%g7** reserviert von SPARC ABI

%i0 - %i7 Acht Input-Register des Register-Fensters mit folgender Semantik:

- **%i0 - %i5** die ersten 6 Argumente, die an die Routine übergeben wurden
- **%i6 / %fp** Framepointer
- **%i7** Return-Adresse - 8

%l0 - %l7 Acht frei verfügbare lokale Register des Register-Fensters

%o0 - %o7 Acht Output-Register des Register-Fensters, mit folgender Semantik:

- **%o0 - %o5** Argumente 1-6 für eine aufzurufende Subroutine
- **%o6 / %sp** Stackpointer

%o7 Adresse der `call` Instruktion

Die 32 Floating-Point Register sind mit %f[0-31] ansprechbar. Das %y Register enhält die höherwertigen Bits bei Division und Multiplikation.

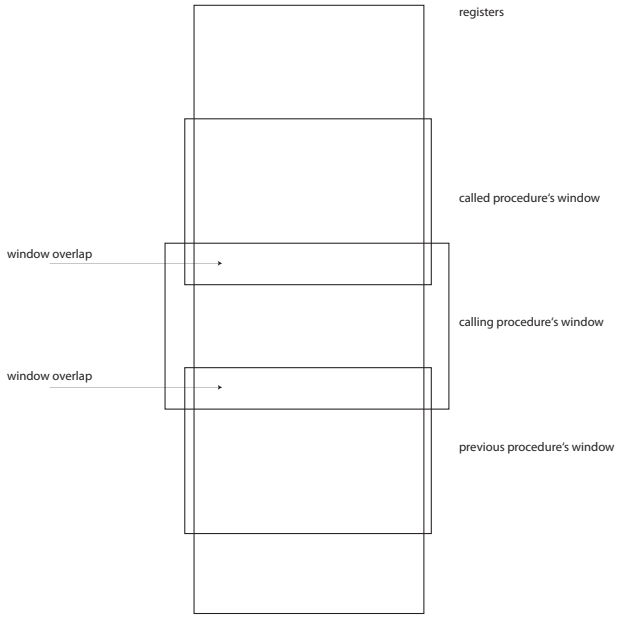

Abbildung 3.1: überlappende Registerfenster[spaa]

Der `CWP` wird beim Prolog (s. Unterabschnitt 3.4.3) einer Routine, der die `save` Instruktion beinhaltet, dekrementiert und zeigt danach auf ein neues Fenster von 24 Registern. Tatsächlich wird das Registerfenster allerdings lediglich um 16 Register verschoben, da sich 8 Register des alten und neuen Fensters überlappen (s. Abbildung 3.1).

Der Grund ist, dass die 8 Output-Register %o[0-7] der aufrufenden Routine, auf die 8 Input-Register %i[0-7] der aufgerufenen Routine gemappt werden. Damit liegen die ausgehenden Argumente des Aufrufers automatisch in den eingehenden Registern der aufgerufenen Routine. Ist die Routine beendet, wird in deren Epilog (s. Unterabschnitt 3.4.4), der die Instruktion `restore` enthält, der `CWP` inkrementiert, so dass das Registerfenster bei Rückkehr wieder die ursprünglichen 24 Register der aufrufenden Routine enthält. Abbildung 3.2 zeigt nochmals schematisch die Funktionsweise des `CWP` und Abbildung 3.3 den Zusammenhang der Input-, Output-Registernamen bei Wechsel des Registerfensters.

Es ist möglich den Prolog (s. Unterabschnitt 3.4.3) und Epilog (s. Unterabschnitt 3.4.4) von Blatt-Routinen, das sind Routinen die keine `call` Instruktion enthalten, so zu optimieren, dass sie ohne Veränderung des `CWP` und folglich ohne neues Registerfenster auskommen. Das entspricht einer Übersetzung ohne Framepointer.

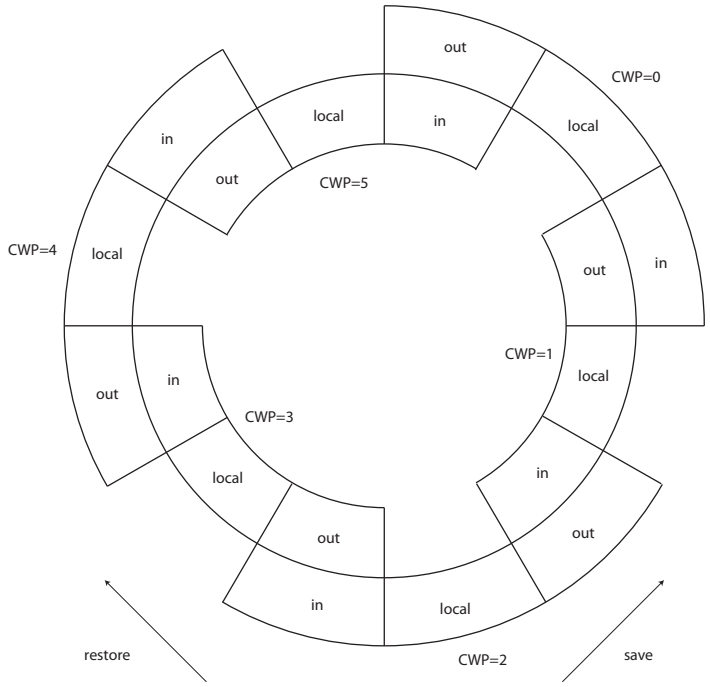

Abbildung 3.2: schematisch funktionsweise des CWP und Registerfensters[spaa]

Je nach Rekursionstiefe von mittels `call` aufgerufenen Subroutinen, kann es vorkommen, dass der `CWP` überläuft. Das bedeutet, alle verfügbaren Registerfenster mit je 24 Registern sind belegt durch die zuvor aufgerufenen Routinen. In diesem Fall wird ein *CWP Overflow Interrupt* ausgelöst und in die ensprechende Funktion des Betriebssystems zur Interruptbehandlung gesprungen, welche die 24 Register des nächsten (belegten) Registerfensters auf dem Stack (s. Unterabschnitt 3.4.1) der zugehörigen Routine sichert. Dieser Vorgang wird vom Betriebssystem auch bei der Behandlung von *Context-Switch Interrupts* wie z.B. blockierendes I/O, Scheduling etc., für alle Registerfenster durchgeführt. So gesicherte Register werden bei Rückkehr in die betreffende Routine wiederhergestellt. Die Floating-Point Register sind von diesem Registerfenster Mechanismus ausgeschlossen und werden global von allen Routinen verwendet.

3.4 ABI: Application Binary Interface

Nachfolgend wird das *Application Binary Interface*[San] der SPARC Architektur beschrieben. Eine ABI ist eine Schnittstelle auf maschienenebene zwischen einem Programm und Betriebssystem oder auch verschiedenen Programmbestandteilen untereinander. Diese Schnittstelle beschreibt unter

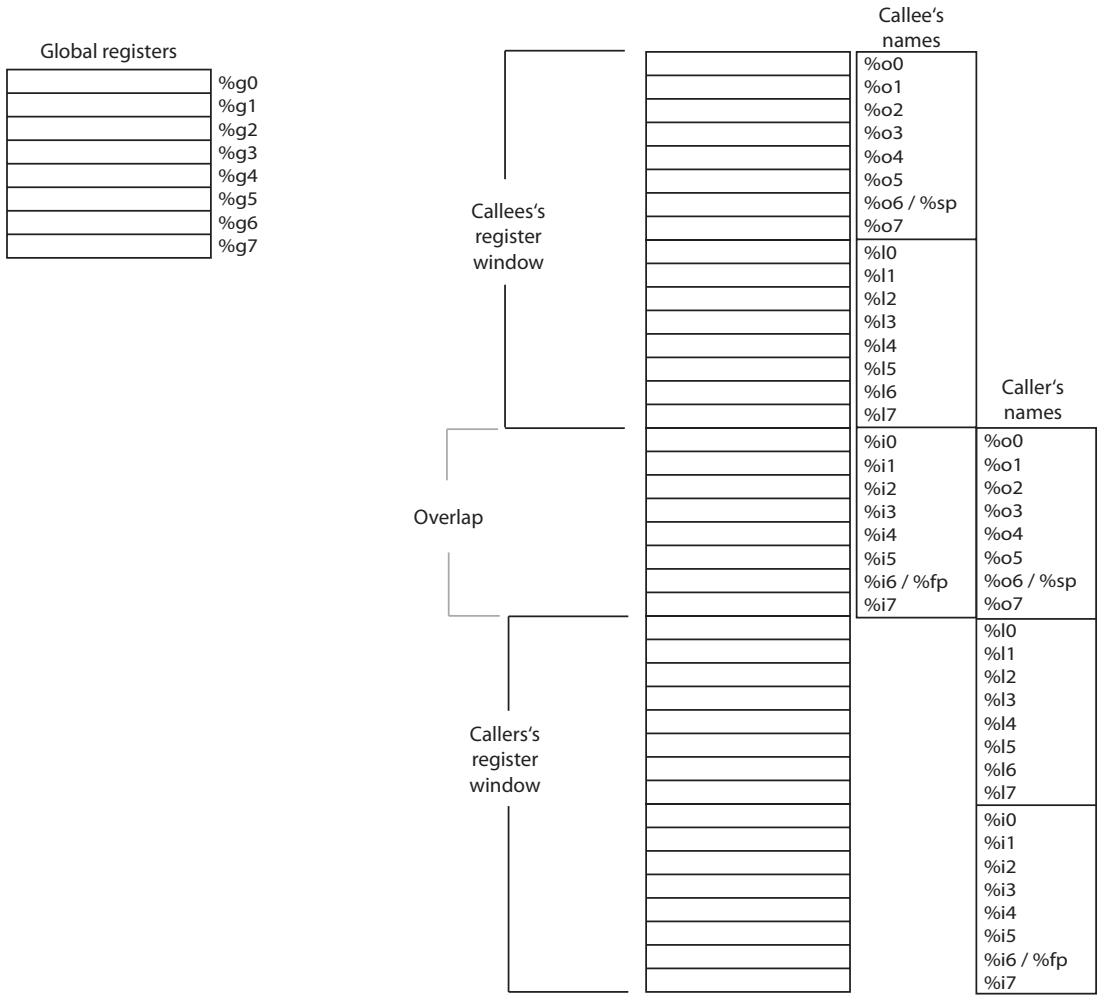

Abbildung 3.3: Mapping der Input- und Output-Registernamen beim Wechsel des Registerfensters[spaa]

anderem Aufrufkonventionen und wie die Übergabe von Parametern zu erfolgen hat. Außerdem legt sie Richtung und Aufbau des Stacks fest und welche Aufgaben jeweils beim Betreten und Verlassen einer Routine zu erledigen sind.

3.4.1 Stackaufbau

Der SPARC Stack wächst, wie bei den meisten Architekturen, nach unten, das bedeutet, der Stackpointer zeigt stehts auf die niedrigste verfügbare Adresse. Die ersten 64 Bytes eines Stackframes müssen immer verfügbar sein, um Inhalte der Register %l[0-7] und %i[0-7] bei auftreten eines CWP

Overflow Interrupts dort automatisch sichern zu können. Die Allozierung dieser initialen Stackgröße wird normalerweise direkt mit dem Aufruf der `save` Instruktion im Prolog einer Routine erledigt. Die nächsten 4 Bytes werden reserviert allerdings nur gefüllt, falls der Rückgabetyp der Routine eine Struktur ist und enthält deren Stackadresse. Danach folgt Platz zur Speicherung der Adressen der Eingangsargumente 1-6, falls diese benötigt werden, sowie weiterer optionaler Platz für alle über den sechsten Parameter hinausgehende Eingangsargumente, die über den Stack übergeben werden müssen. Der variable Platz von hier an bis zum Framepointer, dieser entspricht dem Stackpointer des vorherigen Stackframes, wird zur Speicherung der lokalen, temporären Stack-Variablen verwendet. Eine schematische Abbildung des Stackaufbaus[spab] ist im Anhang Abschnitt A.1 zu finden. Bei Rückkehr einer Routine wird der aktuelle Stackframe entfernt und der Programmfluss springt zurück an die im Register %i7 gespeicherte Adresse (Return-Adresse - 8). Durch die sich überlappenden Register %o[0-7] und %i[0-7] und der Tatsache, dass %o6 dem aktuellen Stackpointer entspricht und %i6 dem Stackpointer des vorherigen Frames (also der aktuelle Framepointer), ergibt sich der neue Stackpointer automatisch durch das verschieben des Register-Fensters aus dem aktuellen Framepointer bzw. %fp/%i6 => %sp/%o6. Umgekehrt ergibt sich beim Aufruf einer Subroutine dadurch automatisch der Framepointer aus dem aktuellen Stackpointer bzw. Register %sp/%o6 => %fp/%i6.

3.4.2 Aufrufkonvention

Die Aufrufkonvention (*Calling Convention*) legt fest, wie eine Routine aufzurufen ist und die Übergabe von Argumenten an diese zu erfolgen hat. Aufgrund der RISC Architektur ist diese bei SPARC sehr einfach. Die ersten 6 an eine Routine zu übergebenden Argumente werden in den Registern %o[0-5] abgelegt. Die aufgerufene Subroutine kann diese ersten sechs Argumente dann aus ihren entsprechenden Eingangsregistern %i[0-5] erreichen. Alle weiteren Argumente, die über den sechsten Parameter hinausgehen, werden über den Stack übergeben.

3.4.3 Prolog

Der Prolog ist der Code, der bei jedem Aufruf einer Routine vor dem eigentlichen Code ausgeführt wird. Bei SPARC besteht dieser Prolog aus dem dekrementieren des `CWP` und dem Allozieren des initalen Stacks. Beide Aufgaben werden durch die `save` Instruktion erledigt, die den `CWP` dekrementiert und eine Addition auf den Stackpointer durchführt:

```
1  myfunc:
2    save %sp, -92, %sp    /* SPARC Prolog */
```

Listing 3.1: SPARC Prolog einer Routine

3.4.4 Epilog

Das Pendant zum Prolog einer Routine, ist der Epilog. Dieser Code wird vor jedem Verlassen einer Routine nach dem eigentlichen Code ausgeführt. Bei SPARC müssen im Epilog der `CWP` inkrementiert und der Kontrollfluss geändert werden, um mit der Ausführung des Programms an der ursprünglichen Stelle vor dem Aufruf der Routine fortzufahren. Das Zurückkehren des Kontrollflusses an die Aufrufstelle wird mit der `ret` Instruktion durchgeführt. Da es sich bei dieser Instruktion um einen *Delayed Branch* (s. Unterabschnitt 3.5.3) handelt, kann der Delay-Slot für die `restore` Instruktion verwendet werden, die den `CWP` inkrementiert:

```
1  ret     /* Zurückkehren zum Aufrufer */
2  restore   /* Delay-Slot der ret Instruktion, CWP inkrementieren */
```

Listing 3.2: SPARC Epilog einer Routine

3.5 Instruktionsformate

Aufgrund der RISC Architektur von SPARC existieren im Vergleich zu anderen Architekturen wie z.B. x86 nur wenige unterschiedliche Adressierungs- und Befehlsformate.

3.5.1 Formate

Die SPARC Architektur ist big-endian und besitzt 3 unterschiedliche Befehlsformate, deren Breite alle 32 Bit betragen. Format 1 ist speziell für `call` und `branch` Instruktionen und nimmt eine 30 Bit breite, zum *Program Counter* relative Adresse (word-aligned und sign-extended) als Argument. Format 2 ist für Condition-Code abhängige `branch` Instruktionen wie z.B. `Bicc` (*Branch on integer condition code*) und für die spezielle `sethi` Instruktion. Je nach Variante, nimmt dieses Format entweder ein Register und eine 22 Bit breite Konstante, für die oberen 22 Bit eines Registers (`sethi`), oder eine 22 Bit breite, relative Adresse als Argument. Befehlsformat 3 ist das am häufigsten auftretende und gibt es ebenfalls in 3 Variationen. Dieses Format ist für alle arithmetische, logische Shifts und Speicherbefehle zuständig. Es nimmt entweder 2 Register als Argumente, oder ein Register und einen 13 Bit breiten, sign-extended Immediate Wert.

3.5.2 32 Bit Adressen

Adressen und Werte, die breiter als 13 Bit sind, können nicht direkt, sondern müssen getrennt in ihren 22 Bit breiten Hi- und 10 Bit breiten Low Wert geladen werden. Dafür steht die spezielle `sethi` (*set high order 22 bits*) Instruktion zur Verfügung, mit der die oberen 22 einer 32 Bit breiten Adresse oder Wertes in ein Register geladen werden können, wobei die unteren 10 Bits auf Null

gesetzt werden. Anschließend können die unteren 10 Bits in dieses Register geladen werden z.B. über eine `or` Instruktion oder einer Addition eines Immediate Wertes.

3.5.3 Delayed Branches

Grundsätzlich sind alle Instruktionen (ausgenommen von *Ticc*), die den Kontrollfluss d.h. den *Program Counter* ändern, mit einer Verzögerung von einer Instruktion behaftet. Das bedeutet, dass die darauf folgende Instruktion ausgeführt wird, bevor der Kontrollfluss geändert wird. Diese Instruktion wird *Delay-Slot* genannt und entspricht im Normalfall der nächsten Anweisung im Assemblercode (genaugenommen handelt es sich um die Instruktion im nPC - *next program counter*). Dieses Verhalten kann bei manchen `branch` Instruktionen (conditional branches, außer branch always) dahingehend beeinflusst werden, dass der Delay-Slot nur dann ausgeführt wird, wenn der Kontrollfluss auch wirklich geändert wurde. Dies erfolgt durch das setzten des `a` Annul-Bits im Befehlsformat und auf Assemblerebene durch Anhängen des Suffix `, a` an das betreffende Branch Mnemonic z.B. `bg,a`.
Diese Funktion ist in der aktuellen Implementierung des Backends nicht berücksichtigt. Stattdessen sollte der Delay-Slot durch Optimierung (s. Unterabschnitt 5.4.1) mit einer Instruktion belegt werden.

4 Implementierung

4.1 Backendstruktur

Es folgt die detailierte Beschreibung der Codeauswahl- und Codeausgabephase (s. Abschnitt 2.3) des Backends. Um wiederkehrende Arbeiten zu erleichtern, verfügt LIBFIRM über eine teilweise automatisierte Infrastruktur zur Erstellung des Grundgerüsts. Die dazu notwendigen Konfigurationen, erfolgen in der .spec Datei des jeweiligen Backends. Das System generiert anhand der definierten Registerklassen, Schablonen für Codeausgabe und Instruktionen den entsprechenden Konstruktions- und Ausgabe-Code der SPARC-Knoten in FIRM, Registerkonstanten, Codeausgabefunktionen etc. In der SPARC Assemblersyntax steht das Ziel- oder Ergebnisregister stehts auf der rechten Seite: `.add %l0, %i4, %o0` enthält das Additionsergebnis in `%o0`. Im folgenden wird diese Konvention auch bei Beschreibung von FIRM-Knoten verwendet. `b = a + b` als FIRM-Knoten ist: `Add a, b, b`.

4.1.1 Registerklassen

Um dem Registerallokator bekannt zu machen, über welche und wieviele Register die Architektur verfügt und welchen Bedingungen diese ggf. unterliegen, müssen die Register in Registerklassen eingeteilt werden. Außerdem muss für jedes Register konfiguriert werden, ob es sich um ein Callee- oder Caller-Save Register handelt. Das Backend besitzt 3 Registerklassen:

gp Diese Klasse enthält alle General Purpose Register, also %g[0-7], %i[0-7], %l[0-7], %o[0-7]. Wie in Abschnitt 3.2 beschrieben, sind die globalen Register %g0 und %g[5-7] reserviert und können vom Allokator nicht zugeteilt werden. Ebenso verhält es sich mit den speziellen Registern für Stack- und Framepointer %o7 / %sp und %i6 / %fp. Register %o7 wird ebenfalls von der Zuteilung ausgeschlossen, da hier die Rücksprungadresse gespeichert ist. Die Caller- und Callee-Save Einstellungen sind wie folgt:

- **callee-save** existieren keine, da die Register %l[0-7] und %i[0-5] duch das Registerfenster für jede Routine neu zur Verfügung stehen
- **caller-save** %o[0-5], %g[1-4]

fp Diese Klasse enthält alle Floating-Point Register %f[0-31]. Da die Floating-Point Register alle global verfügbar sind, müssen sie alle als caller-save konfiguriert werden.

flags Diese Klasse enthält spezielle Flag- und Statusregister, bisher ist dies nur das %y Register. Register dieser Klasse werden vom Allokator nicht zugeteilt.

4.1.2 Schablonen für Codeausgabe

Um die Codeausgabe zu vereinfachen, wurden für häufig auftretende Fälle folgende, eigene Schablonen definiert:

RxI das x-te Register oder Immediate einer Operation

Sx das x-te Quellregister einer Operation

Dx das x-te Zielregister einer Operation

C Immediate

LM Integer load-Mode je nach FIRM Modus, z.B. `ldub` (load unsigned byte), `ldsh` (load signed Halfword)

SM Integer store Mode je nach FIRM Modus z.B. `stb` (store byte), `st` (store word)

O Address-Offset z.B. `[%fp+72]`

Diese Schablonen können innerhalb einer `emit` Option der .spec Datei verwendet werden oder direkt im Code aufgerufen werden.
Für generierte SPARC-Knoten, die meist einen einfachen Assembler-Befehl erzeugen, wurde die Codeausgabe direkt bei der Konfiguration dieses Knotens definiert. Bei manchen Knoten muss je nach Attributwerten oder Kontext eine Fallunterscheidung getroffen werden, wie z.B. bei bedingten Sprüngen nach Vergleichsoperationen. Hier erfolgt die Fallunterscheidung und Codeausgabe in einer separat geschriebenen Ausgabefunktion für den jeweiligen Knoten.

```
1 Sub => {
2   irn_flags => "R",
3   comment   => "construct Sub: Sub(a, b) = a - b",
4   mode    => $mode_gp,
5   reg_req   => { in => [ "gp", "gp" ], out => [ "gp" ] },
6   emit      => '. sub %S1, %R2I, %D1',
7   constructors => \%binop_operand_constructors,
8 }
```

Listing 4.1: Definition des Sub-Knoten in der .spec Datei

`reg_req` legt fest, welche Registerklassen für die eingehenden Operanden und das erzeugte Ergebnis benötigt werden. `mode` definiert den FIRM Modus des Knotens z.B. `float, int, reference, signed, data` etc. `emit` definiert den zu erzeugenden Ausgabecode unter Verwendung von Ausgabeschablonen für die Registernamen und `constructors` erzeugt Konstruktor-Code für den SPARC Sub-Knoten (getrennt jeweils für *Register - Register* und *Register - Immediate* Variante)

4.1.3 Attribute

Manche Knoten benötigen bei der Codeauswahl- oder Codeausgabephase zusätzliche Metainformationen, wie z.B. ob es sich um einen Load/Store Knoten handelt, Stackpointer-Offsets etc. Um diese Informationen zu speichern, besitzt jeder Knoten einen Pointer zu einer Attributstruktur, welche die entsprechenden Informationen enthält. Folgende Attribute wurden für die entsprechenden Knoten definiert:

sparc_attr_t Basisattribut für alle weiteren SPARC Attribute, enhält Integer Immediate-Wert und ein Flag, ob es sich um eine Lade- oder Speicherinstruktion handelt

sparc_load_store_attr_t enthält Modus (Word, Halfword etc.), Vorzeichen, Offset und ob es sich um ein Frame-Entity handelt

sparc_symconst_attr_t enthält das Entity selbst und dessen Stack-Offset

sparc_cmp_attr_t enthält Flags für Permutation und signed oder unsigned Modus eines Vergleichs

sparc_jmp_cond_attr_t enthält die Proj-Nummer des Vergleichs-Modus (größer, kleiner, gleich etc.)

sparc_jmp_switch_attr_t Anzahl der Zweige / Projs und die Proj-Nummer des `default` Zweigs

4.2 Codeauswahl

Die Codeauswahl- oder Transformationsphase, erzeugt aus architekturneutralen Knoten des Ausgangsgraphen die endgültigen Instruktionsknoten der SPARC Architektur.

4.2.1 generierte Instruktionen

Es wird hier anhand der Load Instruktion exemplarisch gezeigt, wie die Beschreibung eines solchen Knotens in der .spec Datei aussieht und verwendet wird:

```
1  Load => {
2    op_flags  => "L|F",
3    comment   => "construct Load: Load(ptr, mem) = LD ptr -> reg",
4    state     => "exc_pinned",
5    ins       => [ "ptr", "mem" ],
6    outs      => [ "res", "M" ],
7    reg_req   => { in => [ "gp", "none" ], out => [ "gp", "none" ] },
8    attr_type => "sparc_load_store_attr_t",
9    attr      => "ir_mode *ls_mode, ir_entity *entity, int entity_sign, long offset, bool is_frame_entity",
10   emit      => '. ld%LM [%S1%O], %D1'
11 }
```

Listing 4.2: Definition des SPARC Load-Knotens in der .spec Datei

Die `attr_type` Option legt dabei fest, dass dieser Knoten ein Attribut vom Typ `sparc_load_store_attr_t` besitzen wird. Der aus dieser Beschreibung generierte Konstruktor-Code ist in Listing 4.3 zu sehen. Er enhält alle Parameter um den Knoten und seine Attributwerte

zu initialisieren. In Listing 4.4 ist zu sehen, wie dieser Konstruktor-Code verwendet wird, um einen neuen SPARC Load-Knoten im Code zu erzeugen.

```
ir_node *new_bd_sparc_Load(dbg_info *dbgi, ir_node *block, ir_node *ptr, ir_node *mem, ir_mode *ls_mode, ir_entity *entity, int
    entity_sign, long offset, bool is_frame_entity);
```

Listing 4.3: erzeugter Konstruktor-Code

```
ir_node  *new_load = NULL;
new_load = new_bd_sparc_Load(dbgi, block, new_ptr, new_mem, mode, NULL, 0, 0, false);
```

Listing 4.4: erzeugen eines neuen SPARC Load-Knotens

4.2.2 generische Backend-Instruktionen

Manche Knoten des architekturneutralen Graphen sind auf den Architekturen 1:1 verfügbar und unterscheiden sich nur im konkret zu erzeugenden Code, wie z.B. IncSP zur Verwaltung des Stackpointers. Diese Knoten müssen in der Phase der Codeauswahl nicht beachtet werden oder nur durch Attribute mit Metainformationen angereichert werden, um später in der Codeausgabephase korrekt ausgegeben werden zu können. Listing 4.5 zeigt das Beispiel des `IncSP` Knotens, der nicht transformiert wird und nur einen Funktion zur Codeausgabe besitzt:

```
/**
 * Emits code for stack space management
 * @param node the IncSP node
 */
static void emit_be_IncSP(const ir_node *irn)
{
  int offs = -be_get_IncSP_offset(irn);

  if (offs == 0)
      return;

  /* SPARC stack grows downwards */
  if (offs < 0) {
    be_emit_cstring("\tsub ");
    offs = -offs;
  } else {
    be_emit_cstring("\tadd ");
  }

  sparc_emit_source_register(irn, 0);
  be_emit_irprintf(", %d", offs);
  be_emit_cstring(", ");
  sparc_emit_dest_register(irn, 0);
  be_emit_finish_line_gas(irn);
}
```

Listing 4.5: Funktion zur Codeausgabe des IncSP Knotens

Neben IncSP ist dies der Fall bei folgenden weiteren Knoten:

Copy kopiert den Inhalt eines Registers in ein zweites

CopyKeep dito

MemPerm permutiert Speicher ohne Verwendung eines zusätzlichen Registers

Perm permutiert zwei Register

Knoten	SPARC Knoten	Erläuterung
Add	Add	Addition zweier Register, oder Register und Immediate Wert
Sub	Sub	Subtraktion zweier Register, oder Register und Immediate Wert
Minus	Minus	Negativer Wert: `Minus(a) = -a`
Mul	Mul	die 32 niedrigwertigsten Bits einer Multiplikation (Reg x Reg oder Reg x Imm)
Mulh	Mulh	die 32 höherwertigen Bits einer Multiplikation (Reg x Reg oder Reg x Imm)
Div	Div	Ergebnis einer Division (Reg / Reg oder Reg / Imm)

Abs	Abs	Betragsfunktion `abs(a) = b` wird transformiert in folgendes Muster:
		`Mov a, b`
		`ShiftRA b, 31, b`
		`Xor a, b, b`
		`Sub a, b, b`

Shl	ShiftLL	logische Linksverschiebung (Reg, Reg oder Reg, Imm)
Shr	ShiftLR	logische Rechtsverschiebung (Reg, Reg oder Reg, Imm)
Shrs	ShiftRA	arithmetische Rechtsverschiebung, Miteinbezug des Vorzeichens (Reg, Reg oder Reg, Imm)
Not	Not	NOT Verknüpfung `Not(a) = !a` (Reg oder Imm)
And	And	AND Verknüpfung `And(a,b) = a & b` (Reg, Reg oder Reg, Imm)
Or	Or	OR Verknüpfung `Or(a,b) = a | b` (Reg, Reg oder Reg, Imm)
Xor	Xor	XOR Verknüpfung `Xor(a,b) = a` $\oplus$ `b` (Reg, Reg oder Reg, Imm)
Load	Load	Laden von einer Adresse in ein Register
Store	Store	Speichern eines Registers an einer Adresse
AddSP	AddSP	Stack vergrößern, d.h. Stackpointer erniedrigen: `sub %sp, 42, %sp`
SubSP	SubSP	Stack verkleinern, d.h. Stackpointer erhöhen: `add %sp, 42, %sp`
Copy	-	Register kopieren, Zuweisung einer Registerklasse, keine Transformation
Call	-	Aufruf einer Routine, Flags setzen, keine Transformation
FrameAddr	FrameAddr	Zugriff auf eine (relative) Adresse des Stackframes
Cond	SwitchJmp/Branch	entweder ein einfacher Bedingter Sprung oder switch/case Sprünge

Tabelle 4.1: Übersicht der Transformation in SPARC Knoten

Knoten	SPARC Knoten	Erläuterung
Cmp	Cmp	Vergleich von Registern
SymConst	SymConst	symbolische Konstanten. Details in Unterabschnitt 4.3.4
Phi	-	SSA-Phi Knoten, abhängig vom Modus zuweisen einer Registerklasse, keine Transformation
Proj	Proj	Transformation nud Zuordnung der alten Proj-Knoten auf die neu erzeugten SPARC-Knoten

Conv	Conv	Konvertierung zwischen verschiedenen FIRM-Modi. Für Floating-Point nach Integer, sowie Floating-Point untereinander (Single, Double, Quad) stehen spezielle Instruktionen zur Verfügung (FpDToFpS, FpSToFpD, FpSToInt, FpDToInt, IntToFpS, IntToFpD etc.).
		Zur Umwandlung von Integern existieren 2 Umwandlungsfunktionen. `shift_width` ist die Differenz der Bitzahl des Quell- und Zielmodus.
		sign-extension:
		`ShiftLL a, shift_width, a`
		`ShiftRA a, shift_width, a`
		zero-extension (8bit):
		`And a, 0xFF, a`
		zero-extension (16bit):
		`ShiftLL a, 16, a`
		`ShiftRA a, 16, a`

Jmp	Jmp	einfacher Sprung, ohne Bedingung
Const	-	Konstanten werden je nach Größe in kleine Teilgraphen transformiert. Dies wird detailiert in Unterabschnitt 4.3.3 beschrieben.

Tabelle 4.2: Übersicht der Transformation in SPARC Knoten (Fortsetzung)

Return gibt den Code des Epilog (s. Unterabschnitt 3.4.4)

4.2.3 Phi- und Proj-Knoten

In FIRM existieren Knoten, die ein Tupel als Wert erzeugen können. Die einzelnen Tupel-Elemente werden durch sogenannte `Proj`-Knoten aus den Knoten heraus verfügbar gemacht. Da diese einzelnen Elemente bei den Knoten der Zielarchitektur ggf. unterschiedlich sind, müssen diese Proj-Knoten bei der Transformation separat behandelt und den neu erzeugten SPARC Knoten richtig zugeordnet werden.
Listing 4.6 zeigt beispielhalft, wie die 2 Proj-Knoten (Speicher und Ergebnis) für einen Load-Knoten dem neu erzeugten SPARC-Knoten zugeordnet werden.

```
1  ir_node  *load     = get_Proj_pred(node);
2  ir_node  *new_load = be_transform_node(load);
3  dbg_info *dbgi     = get_irn_dbg_info(node);
```

```
4   long     proj      = get_Proj_proj(node);
5
6   /* renumber the proj */
7   switch (get_sparc_irn_opcode(new_load)) {
8     case iro_sparc_Load:
9       /* handle all gp loads equal: they have the same proj numbers. */
10      if (proj == pn_Load_res) {
11        return new_rd_Proj(dbgi, new_load, mode_Iu, pn_sparc_Load_res);
12      } else if (proj == pn_Load_M) {
13        return new_rd_Proj(dbgi, new_load, mode_M, pn_sparc_Load_M);
14      }
15    break;
16    default:
17      panic("Unsupported Proj from Load");
18  }
19
20    return be_duplicate_node(node);
```

Listing 4.6: Zuordnung der Projs eines Load-Knotens auf neuen SPARC Load-Knoten

Außerdem gibt es aufgrund der SSA-Form und der Tatsache, dass FIRM diese Darstellung bis zuletzt beibehält, Phi-Knoten, für die keine Entsprechung als reale Instruktion existiert. Da sich die Modi der Phi-Knoten ändern können, muss dem Phi-Knoten in der Transformationsphase eine dem Modus entsprechende Registerklasse zugewiesen werden.

4.3 SPARC spezifische Instruktionen

Es wird nun auf SPARC die spezifischen Implementierungsdetails eingegangen.

4.3.1 Implementierung der Aufrufkonvention

In Unterabschnitt 3.4.2 wurde die Aufrufkonvention von SPARC beschrieben sowie der Zusammenhang der Input- und Outputregister. Aus Sicht des Registerallokators bedeutet dies, dass sich die Register zwischen Aufrufer und aufgerufener Routine unterscheiden und z.B. Parameter 1 nach einem `call` nicht über dasselbe Register erreichbar ist, in dem er zuvor abgelegt worden war. Um diese Logik abzubilden, musste der Code zur Verwaltung der Aufrufkonventionen entsprechend erweitert werden. Für die zur Parameterübergabe zuständigen Register wurde ein Kontextflag eingeführt, über das gesteuert werden kann, ob es sich um ein Register aus Callee-, Caller-Sicht oder beides handelt. Damit lassen sich für Caller bzw. Callee unterschiedliche Register zur Parameterübergabe definieren. Analog dazu verhält es sich mit dem Rückgabewert, der nach der Rückkehr in Register %o0 erwartet, allerdings von der aufgerufenen Routine in %i0 abgelegt wird.

```
1   for (i = 0; i < n; i++) {
2     /* reg = get reg for param i;          */
3     /* be_abi_call_param_reg(abi, i, reg); */
4
5     /* pass outgoing params 0-5 via registers, remaining via stack */
6     /* on sparc we need to set the ABI context since register names of parameters change to i0-i5 if we are the callee */
7     if (i < 6) {
8       be_abi_call_param_reg(abi, i, sparc_get_RegParamOut_reg(i), ABI_CONTEXT_CALLER);
9       be_abi_call_param_reg(abi, i, sparc_get_RegParamIn_reg(i), ABI_CONTEXT_CALLEE);
10    } else {
```

```
11        tp   = get_method_param_type(method_type, i);
12        mode = get_type_mode(tp);
13        be_abi_call_param_stack(abi, i, mode, 4, 0, 0, ABI_CONTEXT_BOTH); /*< stack args have no special context >*/
14      }
15    }
16
17    /* set return value register: return value is in i0 resp. f0 */
18    if (get_method_n_ress(method_type) > 0) {
19      tp   = get_method_res_type(method_type, 0);
20      mode = get_type_mode(tp);
21
22      be_abi_call_res_reg(abi, 0,
23        mode_is_float(mode) ? &sparc_fp_regs[REG_F0] : &sparc_gp_regs[REG_I0], ABI_CONTEXT_CALLEE); /*< return has no special
            context >*/
24
25      be_abi_call_res_reg(abi, 0,
26            mode_is_float(mode) ? &sparc_fp_regs[REG_F0] : &sparc_gp_regs[REG_O0], ABI_CONTEXT_CALLER); /*< return has no special
              context >*/
27    }
```

Listing 4.7: Calling-Convention Register Scopes

4.3.2 start Knoten

Da die ABI (s. Unterabschnitt 3.4.3) fordert, dass nach dem betreten einer Routine unmittelbar die `save` Instruktion aufgerufen werden muss, um das Registerfenster zu verschieben, folgt dem `start` Knoten unmittelbar der entsprechende SPARC `save` Knoten. Gleichzeitig wird der Stack mit der von der ABI geforderten Größe initialisiert und der Stackpointer entsprechend geändert. Dieser initiale auf dem Stack reservierte Platz muss bei der Codeausgabe von relativen Stackadressen wieder korrigiert werden, indem ein entsprechendes festes Offset auf die relative Adresse addiert wird.

4.3.3 Const Knoten

Da architekturbedingt nur Konstanten bis zu einer breite von 13 Bit direkt geladen werden können und darüber hinaus ein Laden getrennt in Hi- und Low-Werten erfolgen muss, erzeugt die Transformationsphase aus Const-Knoten einen entsprechenden Teilgraphen um diese Logik abzubilden (s. Listing 4.8). Für Const-Knoten, die einen Wert zwischen -4096 und 4096 besitzen, kann ein einfacher SPARC Mov-Knoten mit dem entsprechenden Immediate Wert erzeugt werden: `Mov 42, %l0`. Außerdem wird bei allen Knoten von arithmetischen und logischen Operationen für die Operanden überprüft, ob sich diese in Immediate Werte kodieren lassen und dann ggf. dieselbe Logik angewendet. Ist der Wert des Const-Knotens außerhalb dieses Bereichs, werden ein HiImm- und ein LoImm-Knoten erzeugt, um getrennt die Hi- und Low-Werte der Konstanten zu laden. Der schließlich aus diesem Knotenmuster erzeugte Assemblercode ist in Listing 4.9 zu sehen.

```
1 /**
2  * Creates a possible DAG for a constant.
3  */
4 static ir_node *create_const_graph_value(dbg_info *dbgi, ir_node *block, long value)
5 {
6   ir_node *result;
```

```
7
8   // we need to load hi & lo separately
9   if (value < -4096 || value > 4095) {
10    ir_node *hi = new_bd_sparc_HiImm(dbgi, block, (int) value);
11    result = new_bd_sparc_LoImm(dbgi, block, hi, value);
12    be_dep_on_frame(hi);
13  } else {
14    result = new_bd_sparc_Mov_imm(dbgi, block, (int) value);
15    be_dep_on_frame(result);
16  }
17
18  return result;
19 }
```

Listing 4.8: Aufbau eines Teilgraphen für Const-Knoten

```
1 sethi %hi(11880), %l0
2 or %l0, %lo(11880), %l0
```

Listing 4.9: erzeugter Assemblercode für HiImm- und LoImm-Knoten

4.3.4 SymConst Knoten

Symbolische Konstanten werden von Adressen geladen. Bei SPARC müssen 32 Bit Adressen getrennt in Hi- und Low-Wert geladen werden. Dafür wird ein SPARC SymConst Knoten erzeugt, der das entsprechende Firm-Entity enthält. In der Codeausgabephase kann dann mit Hilfe dieser Information, die Konstante mit `sethi` und `or` korrekt von ihrer Adresse geladen werden. Dabei stehen die zwei Assemblerdirektiven `%hi` und `%lo` zur Verfügung, um den jeweiligen Hi und Low Teil eines Wertes zu extrahieren.

```
1 sethi %hi(.Lstr.0), %l0
2 or %l0, %lo(.Lstr.0), %l0
3 ...
4 .Lstr.0:
5   .string "some dummy text\n"
```

Listing 4.10: erzeugter Code zum Laden einer Symbolischen Konstante

5 Fazit

5.1 Probleme

Probleme gab es insbesondere bei den Tests der erzeugten Programme auf dem freien Prozessoremulator QEMU[qem]. So war es z.B. unmöglich, negative Integer-Werte auszugeben über die standardmäßige libc-Funktion `printf`. Der Emulator stürzt aufgrund eines auftretenden Segmentation Fault ab. Später stellte sich herraus, dass eine selbstkompilierte SPARC libc dieses Problem behebt.

5.2 unimplementierte Funktionen

5.2.1 Floating-Point Operationen

Der umfangreichste Teil ist die fehlende Unterstützung der Floating-Point Operationen. Das betrifft neben den einzelnen arithmetischen Instruktionen auch solche zum Laden und Speichern der Werte und die Konvertierung in andere Datentypen (z.B. Float $=>$ Integer). Ebenso fehlt Unterstützung für Routinen mit Floating-Point Argumenten, da sich hier die Aufrufkonvention im Vergleich zu den Integer/GP Registern unterscheidet. SPARC bietet neben Single und Double Precision Genauigkeit für Floating-Point Werte auch die Möglichkeit für Quad Precision Genauigkeit, bei der die Werte auf mehrere Register aufgesplittet werden. Auch hierfür existieren wiederum spezielle arithmetische Instruktionen. Sollen diese unterschiedlichen Genauigkeiten implementiert werden, muss auch eine entsprechende Logik im Registerallokator hinterlegt werden, um die korrekte Aufteilung auf mehrere Register zu gewährleisten. Z.b. müssen Double Precision Werte in einem geraden und dem darauf folgenden ungeraden Register gespeichert werden. Dies wird von den derzeitig verfügbaren Registerallokatoren noch nicht unterstützt.

5.2.2 optimierte Instruktionen

Neben den Standardinstruktionen für Addition, Subtraktion etc. existieren noch einige Instruktionen für häufig auftretende Spezialfälle. Durch den Einsatz dieser Instruktionen könnten bestimmte Muster aus mehreren Instruktionen durch eine einzelne ersetzt werden.

swap tausche Register mit Speicher

tst Teste Register auf 0 und setzte Condition-Code Bit entsprechend

xnor exklusives NOR

andn AND NOT

5.3 SPEC 164.gzip

Der Meilenstein, den 164.gzip Test der SPEC CPU2000[spe] Benchmark Suite zu kompilieren, wurde erreicht. Allerdings muss aufgrund der fehlenden Floating-Point Unterstützung eine kleine Modifikation am Quelltext vorgenommen werden, da ansonsten die Kompillierung fehlschlägt. Diese Modifikation ist für das Testergebnis jedoch unrelevant, da der entsprechende Code ohnehin nie ausgeführt wird. Die in Listing 5.1 gezeigte Funktion kann vollständig entfernt oder auskommentiert werden.

```
1  long int seedi;
2  double ran()
3  /* See "Random Number Generators: Good Ones Are Hard To Find", */
4  /*     Park & Miller, CACM 31#10 October 1988 pages 1192-1201. */
5  /*****************************************************/
6  /* THIS IMPLEMENTATION REQUIRES AT LEAST 32 BIT INTEGERS ! */
7  /*****************************************************/
8  #define _A_MULTIPLIER  16807L
9  #define _M_MODULUS     2147483647L /* (2**31)-1 */
10 #define _Q_QUOTIENT    127773L     /* 2147483647 / 16807 */
11 #define _R_REMAINDER   2836L       /* 2147483647 % 16807 */
12 {
13   long lo;
14   long hi;
15   long test;
16
17   hi = seedi / _Q_QUOTIENT;
18   lo = seedi % _Q_QUOTIENT;
19   test = _A_MULTIPLIER * lo - _R_REMAINDER * hi;
20   if (test > 0) {
21     seedi = test;
22   } else {
23     seedi = test + _M_MODULUS;
24   }
25   return ( (float) seedi / _M_MODULUS);
26 }
```

Listing 5.1: 164.gzip: spec.c Zeile 52 - 77

5.4 Future Work und Optimierungen

Abgesehen von den nicht implementierten Funktionen aus Abschnitt 5.2 besteht an einigen Stellen die Möglichkeit weiterer Optimierungen, die sich teilweise auch stark auf die Performance des erzeugten Codes auswirken dürften.

5.4.1 Delay-Slot Optimierung

Das (effiziente) ausnützen von Delay-Slots bei Instruktionen, die den Kontrollfluss ändern, bietet mitunter das größte Optimierungspotential. Diese Optimierung kann vor der bzw. während der Emitter-Phase eingreifen und entscheiden ob und welche vorhergehenden Instruktionen in den Delay-Slot verschoben werden können. Damit können die bisherigen `nop` Instruktionen eingespart und dadurch Ausführungsgeschwindigkeit gewonnen werden.

5.4.2 optimierte Blatt-Funktionen

Dies ist eine Optimierung, die auch im Manual[Spac] der SPARC V8 Architektur vorgeschlagen und beschrieben wird. Ziel ist es, die Instruktionen zur Verwaltung des Registerfensters (s. Abschnitt 3.3 und Unterabschnitt 3.4.3, Unterabschnitt 3.4.4) einzusparen und damit auch das Verschieben des Fensters zu vermeiden. Diese Optimierung kann allerdings nur bei sogenannten Blatt-Funktionen angewendet werden. Das sind solche, die im Call-Graph eines Programms ein Blatt bilden, d.h. keine `call` oder `jumpl` Instruktionen enthalten. Wie diese Optimierungen genau durchzuführen sind und was dabei zu beachten ist, ist im SPARC V8 Manual[Spac] im Anhang D.5 beschrieben.

5.4.3 Berechnung von Adress-Offsets

Vergleicht man den erzeugen Assemblercode mit der Ausgabe des SPARC GCC[gcc], so fällt auf, dass die Berechnung von Offset-Adressen wesentlich kompakter erfolgt und unter Umständen deutlich weniger Instruktionen benötigt, im Vergleich zur aktuellen Implementierung dieses Backends. Listing 5.2 zeigt das Laden von einer Adresse, die sich über ein Offset des Framepointers ergibt, in der aktuellen Implementierung und Listing 5.3 die kompakte Variante des GCC.

```
1 add %fp, 32, %l0
2 ld [%l0], %l1
```

Listing 5.2: Laden einer Adresse: aktuelle FIRM Variante

```
1 ld [%fp+32], %l1
```

Listing 5.3: Laden einer Adresse: kompakte gcc Variante

5.4.4 Sprungtabelle

Unabhängig von ihrer Größe werden switch/case Statements bisher als semantisch äquivalente if/elseif Konstrukte ausgegeben. Hier könnte eine Optimierung durch das implementieren einer Sprungtabelle erreicht werden.

5.4.5 weitere Optimierungen

IncSP/FrameAddr unterstützen derzeit keine Offsets größer als 1024

Const(0) Knoten hierfür könnte das %g0 Register benützt werden

IncSP/AddSP bei IncSP und AddSP Knoten wird derzeit jeweils eine `add` Instruktion mit der entsprechenden Größe erzeugt. Diese könnten duch eine Peephole Optimierung zu einer kummulierten Addition zusammengefasst werden. Ideal wäre es, diese kummulierte Addition direkt bei der `save` Instruktion zu erledigen.

64bit Intrinsics

A Anhang

A.1 SPARC Stack Layout

SPARC Architecture Stack Structure

Literaturverzeichnis

[164] *SPEC CPU 2000 164.gzip*. http://www.spec.org/cpu2000/CINT2000/164.gzip/docs/164.gzip.html,

[CFR+91] CYTRON, Ron ; FERRANTE, Jeanne ; ROSEN, Barry K. ; WEGMAN, Mark N. ; ZADECK, F. K.: Efficiently computing static single assignment form and the control dependence graph. In: *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS* 13 (1991), S. 451–490

[fir] *LibFirm Website*. http://www.libfirm.org,

[gcc] *The GNU Compiler Collection*. http://gcc.gnu.org,

[hal] *Halbordnung auf Wikipedia*. http://de.wikipedia.org/wiki/Ordnungsrelation,

[Lin02] LINDENMAIER, Götz: libFIRM – A Library for Compiler Optimization Research Implementing FIRM. Version: September 2002. `http://www.info.uni-karlsruhe.de/papers/Lind_02-firm_tutorial.ps`. Universität Karlsruhe, Fakultät für Informatik (2002-5). – Forschungsbericht. – Online–Ressource. – 75 S

[llv] *The LLVM Compiler Infrastructure*. http://www.llvm.org,

[ope] *OpenSPARC website*. http://www.opensparc.net/,

[qem] *QEMU Opensource processor emulator*. http://www.qemu.org,

[RWZ88] ROSEN, Barry K. ; WEGMAN, Mark N. ; ZADECK, F. K.: Global Value Numbers and Redundant Computations. In: *POPL*, 1988, S. 12–27

[San] The Santa Cruz Operation, Inc.: *SYSTEM V APPLICATION BINARY INTERFACE - SPARC Processor Supplement*. Third Edition

[spaa] *SPARC Architektur Grafiken, Computer Science department, University of New Mexico*. http://www.cs.unm.edu/ maccabe/classes/341/labman/node11.html,

[spab] *the SPARC stack layout, Computer Science department, Princeton University*. http://www.cs.princeton.edu/courses/archive/fall02/cs217/precepthandouts/1119/sparcstack.pdf

[Spac] Sparc International Inc.: *The SPARC Architecture Manual*. Version 8, Rev. SAV080SI9308

[spe] *SPEC CPU 2000 Benchmark*. http://www.spec.org/cpu2000/,

[sun] *SUN Microsystems*. http://www.sun.com,