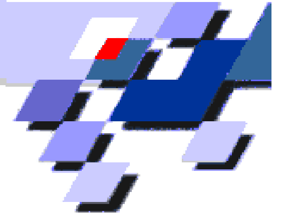
UNIVERSITÄT DORTMUND



Diplomarbeit

Performance-Optimierung in Eingebetteten Systemen durch automatische Ausnutzung von Prozessor-Merkmalen auf Quellcode-Ebene

André Schaefer

Universität Dortmund
Lehrstuhl Informatik XII
Embedded Systems Design

Datum 14.03.2005

Betreuer:
Dr. Heiko Falk
Prof. Dr. Peter Marwedel

André Schaefer, Riegestrasse 7, 45768 Marl
Matrikel Nr. 64896, Universität Dortmund

An dieser Stelle möchte ich mich ganz herzlich bei Herrn Dr. Heiko Falk bedanken. Durch seine außergewöhnlich gute Betreuung und sein Talent, mich voranzutreiben und meine Motivation vielfach neu zu entfachen, ist diese Diplomarbeit entstanden. Weiterhin gilt ein Dankeschön Herrn Jens Wagner für seine Hilfe bei technischen Problemen.

André Schaefer
Marl, 14. 03.2005

Inhaltverzeichnis

1 EINFÜHRUNG 1

1.1 TRADITIONELLE CODE-OPTIMIERUNG 4
1.2 QUELLCODE-OPTIMIERUNG MIT INTRINSICS 8
1.3 VERWANDTE ARBEITEN 11
1.4 ZIELE DER ARBEIT 13
1.5 STRUKTUR 15

2 GRUNDLAGEN UND KONZEPTE 17

2.1 DER TMS C6000 DSP VON TEXAS INSTRUMENTS 17
2.1.1 ARCHITEKTUR 18
2.1.2 INTRINSICS – COMPILER-KNOWN FUNCTIONS 20
2.1.2.1 Zweierkomplement 21
2.1.2.2 Überlauf 21
2.1.2.3 Saturierende Arithmetik 23
2.1.2.4 Multimedia-Befehle - SIMD 24
2.2 SUIF COMPILER SYSTEM 26
2.2.1 FILE EBENE 27
2.2.2 PROZEDUR EBENE 28

3 BITGENAUE DATENFLUSS-ANALYSE 31

3.1 LLIR 32
3.2 LINT 34
3.3 DATENFLUSSGRAPH 35
3.4 DATENFLUSSANALYSE 38

4 PORTIERUNG DER DATENFLUSS-ANALYSE 45

4.1 ANBINDUNG VON SUIF 45
4.2 OPTHELPER-KLASSE 56

5 OPTIMIERUNGEN 59

5.1 SATURIERENDE ARITHMETIK **59**
5.1.1 SATURIERUNG NACH EINER ADDITION 59
5.1.2 SATURIERUNG NACH EINER SUBTRAKTION 63
5.1.3 SATURIERUNG NACH EINER MULTIPLIKATION ZUZÜGLICH LEFT-SHIFT 64
5.1.4 SATURIERUNG NACH EINER LINKSVERSCHIEBUNG 67
5.2 BITBREITENREDUKTION **70**
5.2.1 ALGORITHMUS 70
5.3 MULTIMEDIA-BEFEHLE **77**
5.3.1 VERWENDETE SIMD-BEFEHLE 77
5.3.2 ALGORITHMUS 79

6 ERGEBNISSE **85**

6.1 G.723.1 DUAL SPEECH CODER **85**
6.1.1 SATURIERENDE ARITHMETIK 86
6.1.2 BITBREITEN-REDUKTION 88
6.1.3 MULTIMEDIA-BEFEHLE 90
6.2 GSM **92**
6.2.1 SATURIERENDE ARITHMETIK 93
6.2.2 BITBREITEN-REDUKTION 93
6.2.3 MULTIMEDIA-BEFEHLE 95
6.3 ADPCM – G.721 **96**
6.3.1 BITBREITENREDUKTION - ADPCM 97
6.3.2 BITBREITENREDUKTION – G.721 99

7 ZUSAMMENFASSUNG UND AUSBLICK **101**

7.1 ZUSAMMENFASSUNG **101**
7.2 AUSBLICK **102**

Abbildungsverzeichnis

Abbildung 1-1: Typische Beispiele für Eingebettete Systeme 2

Abbildung 1-2: globale Vernetzung von Dienstleistungen 3

Abbildung 1-3: Schritte eines optimierenden Compilers 6

Abbildung 2-1: Blockschaltbild des TI C62x / C67x 18

Abbildung 2-2: Fixed Point Instruction to functional unit mapping, TMS320 C6000 Technical Brief, Februar 1999, Seiten 2-11 20

Abbildung 2-3: Zweierkomplement einer 4 Bit-Zahl 21

Abbildung 2-4: Übertrag bei einer Addition 22

Abbildung 2-5: SIMD-Struktur 25

Abbildung 2-6: Stand der Zwischendarstellung 26

Abbildung 2-7: SUIF File Ebene 28

Abbildung 2-8: SUIF Procedure Level 30

Abbildung 3-1: Ursprüngliche Struktur der PP32 Implementierung 31

Abbildung 3-2: LLIR Hierarchie 33

Abbildung 3-3: Halbordung der Bit-Information 35

Abbildung 3-4: Datenflussgraph 36

Abbildung 3-5: Port-Nummerierung der DFG-Knoten 37

Abbildung 3-6: Nummerierung der Kanten 37

Abbildung 3-7: Berechnung eines bitweisen UND 39

Abbildung 3-8: Top-Down-Analyse 40

Abbildung 3-9: Bottom-Up Analyse einer Linksverschiebung 41

Abbildung 3-10: Bottom-Up Analyse einer Rechtsverschiebung 41

Abbildung 3-11: Bottom-Up-Analyse 42

Abbildung 3-12: Berechnung des „kleinsten Nenners“ 43

Abbildung 4-1: Zuweisung einer Konstanten 46

Abbildung 4-2: Top-Down und Bottom-Up einer Konvertierung 46

Abbildung 4-3: Vorwärts Übertragungsfunktion der Addition 48

Abbildung 4-4: Top-Down-Analyse einer Addition 49

Abbildung 4-5: Aufwärtsanalyse einer Addition 49

Abbildung 4-6: Übertragungsfunktion einer „AND“-Verknüpfung 50

Abbildung 4-7: Vorwärtsanalyse einer „AND“-Verknüpfung 51

Abbildung 4-8: Aufwärtsanalyse einer „AND“-Verknüpfung 51

Abbildung 4-9: Linksverschiebung um drei Stellen 52

Abbildung 4-10: Aufwärtsanalyse einer Linksverschiebung 53

Abbildung 4-11: Übertragungsfunktion der „NOT“-Instruktion 53

Abbildung 4-12: Funktionstabelle für die Vergleichsoperation „==“ 54

Abbildung 4-13: Vorwärtsanalyse der Vergleichsoperation „<“ 55

Abbildung 5-1: Zeile 1) erkennt ungleiche Vorzeichen 60

Abbildung 5-2: Zeile 1) erkennt gleiche Vorzeichen 61

Abbildung 5-3: Datenflussgraph einer Addition und Saturierung 62

Abbildung 5-4: Multiplikation der zwei kleinsten 16 Bit-Zahlen 64

Abbildung 5-5: Überlauf beim Linksverschieben um eine Stelle 65

Abbildung 5-6: Datenflussgraph der Multiplikation mit Shift und Saturierung 66

Abbildung 5-7: Darstellungen der Hexadezimalzahl 0x3FFFFFFF 67

Abbildung 5-8: Darstellungen der Hexadezimalzahl 0xC0000000 68

Abbildung 5-9: Datenflussgraph der Linksverschiebung 69

Abbildung 5-10: „x“-Werte kennzeichnen überflüssige Bits 71

Abbildung 5-11: Benutzung des _pack2-Intrinsics 77

Abbildung 6-1: G.723.1 als Bestandteil des H.323 Standards 86

Abbildung 6-2: Relative Laufzeiten des G.723.1 Codecs für saturierende Arithmetik 87

Abbildung 6-3: Bitreduktion der G.723.1-Implementierung 88

Abbildung 6-4: Bits in ganzen Bytes und nachfolgende Bits 89

Abbildung 6-5: Anteil in % der optimierten Variablenarten im G.723.1 90

Abbildung 6-6: „Loop Unrolling“ 91

Abbildung 6-7: Laufzeitverbesserung nach SIMD-Optimierung im G.723.1 91

Abbildung 6-8: Laufzeit des GSM-Encoders für saturierende Arithmetik 93

Abbildung 6-9: Bitreduktionen in % der GSM-Implementierung 94

Abbildung 6-10: Anteil in % der optimierten Variablenarten im GSM 95

Abbildung 6-11: Laufzeit nach SIMD-Optimierung des GSM-Verfahrens 96

Abbildung 6-12: Bitbreitenreduktion des ADPCM-Verfahrens 97

Abbildung 6-13: Anteil in % der optimierten Variablenarten im ADPCM 98

Abbildung 6-14: Bitbreitenreduktion des G.721-Verfahrens 99

Abbildung 6-15: Anteil in % der optimierten Variablenarten im G.721 100

1 Einführung

Die menschliche Umgebung ist stets im Umbruch. Man ist darauf bedacht, eine Zukunft zu schaffen, die unsere Lebensräume möglichst komfortabel gestalten lässt. Zeitersparnis und Flexibilität sind daher ein wichtiges Merkmal, das es zu erreichen gilt. Dies wird möglich durch die Einbettung von technischen Systemen in unsere alltägliche Umwelt. Umgebende Technologie, die allerdings in den Hintergrund tritt, obwohl sie allgegenwärtig ist. „Invisible Computing" ist das magische Wort, das den Sachverhalt treffend beschreibt.

In ihrer ursprünglichen Form gelten Computer als allgemeingültige „Werkzeuge". Ihre Anwendungen werden durch den Benutzer bestimmt, und verschiedenste Programme werden auf ihnen entwickelt. Bis in die frühen 90er Jahre wurden Optimierungen bezüglich des Compilerbaus und der Rechnerarchitektur meist nur auf Basis dieser Rechensysteme durchgeführt. Allerdings gewinnen heutzutage Systeme, die als unabhängiger Bestandteil in ein übergeordnetes System eingebettet sind, immer mehr an Bedeutung. Durch eine Vielzahl neuer Technologien ist es nicht mehr möglich, effiziente Applikationen zu entwickeln, die durch die ursprüngliche Form der Computer verwendet werden können. Aufgrund dessen werden Anwendungen benutzt, um auf spezialisierten Systemen zum Einsatz zu kommen. Die meisten Systeme werden einmalig für einen Aufgabenbereich konfiguriert und arbeiten danach unabhängig für das übergeordnete System. Der Benutzer hat meist nur geringen Einfluss auf die Funktionsweise des Gerätes.

Mittlerweile sind Eingebettete Systeme praktisch in allen alltäglichen Umgebungen, die der Mensch benutzt, enthalten. Nicht nur am Arbeitsplatz, sondern in der kompletten Spanne von der Raumfahrt über Flugzeuge, Automobile, Züge, industriellen Automation, Robotik, biomedizinischen Implantaten bis hin zum Audio- und Videobereich, sind Mikroprozessoren Bestandteil von Eingebetteten Systemen. Abbildung 1-1 zeigt typische Beispiele.

Eingebettete Systeme zeichnen sich durch eine Vielzahl von Eigenschaften aus. Zum einen benötigen sie häufig einen hohen Datendurchsatz, da viele Einsatzgebiete in signalverarbeitenden Aufgabenbereichen liegen und externe Quellen gelesen und verarbeitet werden müssen. Somit muss eine Echtzeit-Verarbeitung möglich sein. Da diese Geräte oft ohne externe Stromquelle auskommen müs-

sen, ist die Betrachtung der Leistung und des Energieverbrauchs ein weiteres wichtiges Merkmal.

Abbildung 1-1: Typische Beispiele für Eingebettete Systeme

Kern eines Eingebetteten Systems ist, wie in jedem anderen Computersystem, der Prozessor. Hier bieten sich allerdings unterschiedliche Varianten an, die jeweils Vor- und Nachteile besitzen. Die wohl populärsten und auch leistungsfähigsten Prozessoren sind die Mikroprozessoren. Durch die Leistungsfähigkeit sind diese Prozessoren dementsprechend flexibel, allerdings geht das auf Kosten des Energieverbrauches. Somit ist diese Art der Prozessoren für Eingebettete Systeme meist unbrauchbar und wird bevorzugsweise in fest installierten Rechensystemen benutzt. Die zweite Variante besteht in der Benutzung von Digitalen Signalprozessoren (DSP). Ihre Aufgabengebiete sind viel spezialisierter und beziehen sich meist auf den Audio-, Video-, Graphik- oder Sprachbereich. Die zu verarbeitenden Signale können meist in analoger als auch digitaler Form vorliegen, genau wie die gewünschte Ausgabe nach der Verarbeitung. Ein wesentlicher Aspekt ist, dass mögliche Qualitätsverluste bei der Verarbeitung der Daten reduziert oder gänzlich vermieden werden sollen. Eine besondere Eigenschaft der Digitalen Signalprozessoren ist, dass oft nicht nur Wortbreiten in der Größe einer Zweierpotenz verarbeitet werden können. Ein Beispiel dafür ist ein Grafikprozessor, der für jede Grundfarbe ein Byte, also insgesamt 24 Bits, benötigt, und somit den „True Color Modus“ mit 16 Millionen Farben darstellen kann.

Die dritte Gruppe der Prozessoren bilden die Mikrocontroller. Sie werden in der Regel mit einem technischen Gerät eingesetzt, das durch den Einsatz der Steuer- oder Regelfunktion eines Mikrocontrollers Funktionalität annimmt. Die hervorzuhebenden Eigenschaften der Mikrocontroller sind ihre meist geringe Größe und der geringe Energieverbrauch.

Die genaue Einteilung eines jeweiligen Prozessors in die genannten Gruppen wird mit der Zeit jedoch immer schwieriger, da die Übergänge der einzelnen Gruppen immer unschärfer werden. Vielmehr wird die Integration der einzelnen Gruppen in den Vordergrund gerückt. Die einzelnen Bauteile werden als Module, gemäß der gewünschten Applikation, zusammengefügt, und man spricht von einem „system on a chip". Durch die Verbindung unterschiedlicher Bestandteile und die Möglichkeit, diese zu programmieren und für spezielle Applikationen anzupassen, verringert man zusätzlich das „time-to-market", sprich die Zeit, die vergeht bis man eine Idee realisiert und auf den Markt gebracht hat, sowie die Kosten der Entwicklung. Denn durch die Eigenschaft der Programmierfähigkeit besteht die Möglichkeit der Wiederverwendung der bereits implementierten Software in anderen Bereichen.
Verbunden mit der weltweiten Vernetzung bieten Eingebettete Systeme erhebliche Möglichkeiten. Es wird Realität, Dienstleistungen global und unabhängig von fest installierten Rechnersystemen und vor allem in Echtzeit zur Verfügung zu stellen.

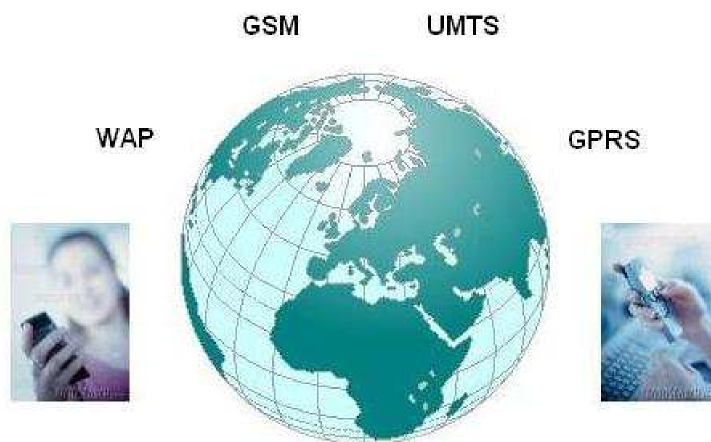


Abbildung 1-2: Globale Vernetzung von Dienstleistungen

Jedoch gibt es eine Vielzahl von unterschiedlichen Prozessoren und somit eine große Anzahl von Möglichkeiten, diese in ein Eingebettetes System zu integrieren. Dies wiederum resultiert in einer sehr starken Heterogenität der unterschiedlichen Systeme, die eine globale Vernetzung erheblich erschwert. Um eine Ausweitung auf die verschiedenen Plattformen weltweit stattfinden zu lassen, sind Standardisierungen der verwendeten Applikationen notwendig, um zu gewähr-

leisten, dass sich die unterschiedlichen Systeme, die jedoch die gleichen Dienste zur Verfügung stellen, unterhalten können. Erst dadurch wird eine globale Vernetzung und Nutzung dieser durch die verschiedenen Dienstleistungen möglich, und man ist in der Lage, die gesamte Bandbreite der Möglichkeiten, die uns mit Eingebetteten Systemen zur Verfügung stehen, auszunutzen.
Allerdings geht durch diese Standardisierung die Möglichkeit verloren, maschinenspezifisch zu entwickeln. Durch die große Vielzahl an Prozessoren ist es unmöglich, eine Referenz-Implementierung zu entwickeln, die auf die speziellen Prozessoren zugeschnitten ist. Diese Arbeit soll Anregungen im Bereich der automatischen Optimierung für spezielle Prozessoren geben. Sie zeigt Beispiele auf, wie es möglich wird, maschinenspezifischen Quellcode automatisch zu erzeugen, um damit eine bessere Code-Qualität zu erreichen.

1.1 Traditionelle Code-Optimierung

Code-Optimierungen haben zur Aufgabe, die Qualität eines erzeugten Maschinenprogramms bezüglich der Laufzeit, der Codegröße, des Speicher- und Energieverbrauchs zu verbessern. Vor allem im Bereich der Eingebetteten Systeme sind diese Eigenschaften von großer Bedeutung, da immer nur eine begrenzte Kapazität der verfügbaren Einheiten verwendet werden kann. Software-Entwicklung für Eingebettete Systeme fand zunächst größtenteils auf der Assembler-Ebene statt. Aufgrund von schlechten Compilern war man gezwungen, Optimierungen direkt am Maschinencode vorzunehmen. Der Trend ging stark zum prozessorbasierten Entwurf. Zwar gab das ein hohes Maß an Flexibilität, da Prozessoren programmierbar sind und bereits bestehende Implementierungen für weitere Projekte wieder verwendet werden konnten. Allerdings waren die zugehörigen Compiler der Prozessoren zu schwach [WZM96]. Dies ist zu begründen durch den maschinenspezifischen Befehlssatz der einzelnen Prozessoren. Zwar sind die typischen DSP-Instruktionen effektiv, aber es besteht kein direkter Bezug zum zugehörigen Quellcode-Programm. Daher wurden die zur Verfügung stehenden Möglichkeiten nur begrenzt durch die Compiler ausgenutzt. Um dieser Diskrepanz entgegenzuwirken, wurden unterschiedliche Strategien angewandt [LM97]. Zum einen war es möglich, einen leistungsstärkeren Prozessor zu verwenden. Allerdings geht dies auf Kosten von erhöhtem Energieverbrauch und Größe der verwendeten Platinen. Daher wurde diese Methode in der Praxis selten angewandt. Zum anderen konnten spezifische DSP-Erweiterungen für die

Programmiersprache benutzt werden. Diese Code-Erweiterungen finden auch in dieser Arbeit ihren Platz und werden genauer in Kapitel 1.2 betrachtet. Die dritte Alternative bestand aus dem Einsatz von Assembler beziehungsweise Inline-Assembler[1]. Darunter litt aber widerrum die Portierbarkeit auf andere Prozessoren aufgrund der unterschiedlichen Befehlssätze. Demnach existierte keine zufrieden stellende Unterstützung der Compiler. In den letzen Jahren wurden allerdings erhebliche Fortschritte in der Entwicklung von Compilern erzielt, so dass der Trend langsam von der Assemblerprogrammierung wich. Optimierungen auf der Quellcode-Ebene wurden im Laufe der Zeit immer notwendiger. Digitale Signalprozessoren werden immer vielseitiger und ihr Funktionsumfang immer gewaltiger. Somit wächst auch die Komplexität der Anwendungen, die auf ihnen ausgeführt werden können. Unterstützt durch hochoptimierende Compiler ist es eine erhebliche Arbeitserleichterung, in einer Hochsprache zu programmieren und zu optimieren. Um die Korrektheit einer Optimierung zu überprüfen, ist es lediglich notwendig, den geänderten Code durch den Compiler übersetzen zu lassen und das lauffähige Programm mit dem originalen zu vergleichen. So ist es effizient möglich, den Source-Code auf verschiedenen Plattformen mit unterschiedlichen Compilern zu testen. Es müssen keinerlei Änderungen am Source-Code vorgenommen werden, um die Implementierungen auf diversen Maschinen ausführen zu können. Im Gegensatz dazu steht, wie oben bereits beschrieben, die Programmierung auf der Assembler-Ebene. Die Vielzahl der Prozessoren unterscheidet sich vehement voneinander und besitzt unterschiedliche Befehlssätze. So ist es nicht möglich, eine Implementierung auf unterschiedlichen Plattformen auszuführen, ohne erhebliche und zeitintensive Veränderungen vorzunehmen. Somit bildet die Entwicklung auf der Quellcode-Ebene die effektivere Art der Programmierung.

Durchgesetzt für die Programmierung von Eingebetteten Systemen hat sich der ANSI-C Standard [KR88]. Er bildet eine Grundlage in der Softwareentwicklung für Digitale Signalprozessoren. Jegliche Compiler der verschiedenen Prozessoren sollen in der Lage sein, diesen Standard zu verstehen und zu verarbeiten. Abbildung 1-3 zeigt die Schritte eines optimierenden Compilers in der üblichen Art der Übersetzung eines Source-Codes.

---

[1] Assemblerprogrammierung eingebettet in den Quellcode einer Hochsprache.

Moderne Compiler benutzen für die Analyse von Quellcode Zwischendarstellungen, die mit hohem, mittlerem oder niedrigem Abstraktionsgrad arbeiten. Zwischendarstellungen repräsentieren die Struktur des betreffenden Programms möglichst unabhängig von Quell- und Zielsprache dar. Das Programm wird in der Quellsprache eingelesen und entsprechend der Struktur des Quellcodes in die Zwischendarstellung überführt. Aktuelle Zwischendarstellungen sind immer mehr graphenbasiert, da man versucht, in der Zwischendarstellung die Daten- und Kotrollabhängigkeiten darzustellen. Dies hat gleichzeitig den positiven Effekt, dass alle für die späteren Optimierungen wichtigen Informationen erhalten bleiben. So werden weitere Analysen und mögliche Optimierungen möglich. Zunächst wird das ANSI-C-Programm in eine Zwischendarstellung mittleren Abstraktionsgrades übersetzt. In dieser Phase finden maschinenunabhängige Optimierungen statt.

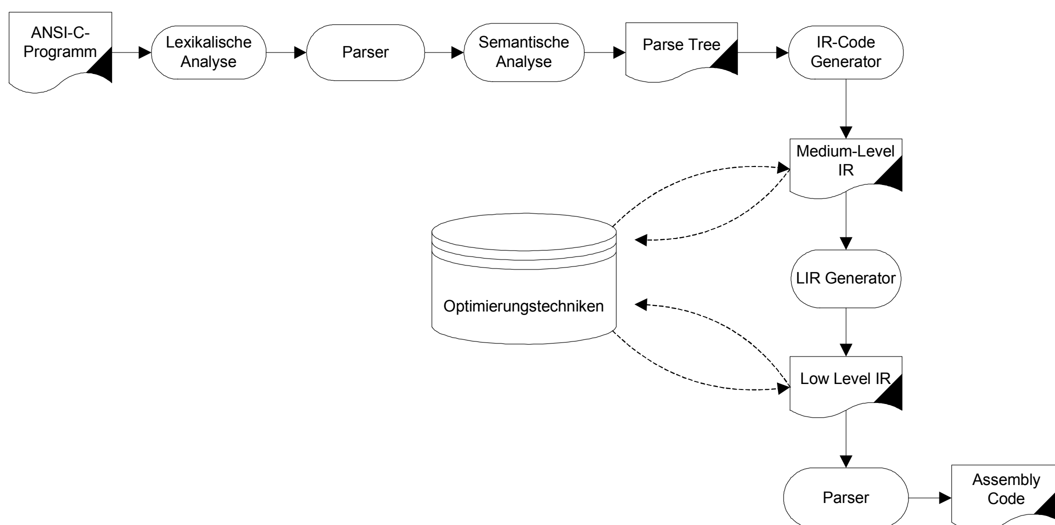


Abbildung 1-3: Schritte eines optimierenden Compilers

Typische Beispiele für maschinenunabhängige Transformationen sind

- **CSE – Common Subexpression Elimination**
  Die Berechnung einer Operation wird als redundant angesehen, wenn sie im Laufe des Quellcodes mehrfach ausgeführt wird, ohne dass sich das Ergebnis ändert. Um solche Redundanzen zu vermeiden, berechnet man das Ergebnis dieser Operation nur einmal und speichert es in einer Vari-

able zwischen. Anstatt diese Operation zu wiederholen, wird nun diese zuvor berechnete Variable benutzt und ersetzt die redundante Operation.

- **Dead Code Elimination**
  Jegliche Art von Berechnung, die im weiteren Verlauf niemals benutzt wird, kann eliminiert werden ohne einen Effekt auf das vorliegende Programm auszuüben. Diese Art von nicht benutztem Code nennt man „Dead Code".

- **Loop-invariant code motion**
  Berechnungen, die innerhalb von Schleifen stattfinden, aber in keiner Abhängigkeit zu anderen Resultaten im Schleifenkörper stehen, heißen „Schleifen-invariant". Daher ist es möglich, diese Berechnungen aus dem Schleifenkörper zu entfernen und sie außerhalb dieser zu berechnen, um Performanceeinbußen umgehen zu können.

- **Constant folding**
  Berechnungen, die garantiert in konstanten Ergebnissen enden, können bereits während des Kompiliervorgangs durchgeführt werden. In dem Fall erstellt der Compiler für jede Berechnung, die in einem festen Wert resultiert, eine Konstante bereits beim Übersetzen des Codes.

Nachfolgend findet eine Übersetzung der vorliegenden Zwischendarstellung in eine abstraktere Art der Darstellung, die lediglich noch assemblerartige Befehle enthält, statt. An dieser Stelle werden nun Optimierungen, die vom jeweiligen Prozessor abhängig sind, durchgeführt. Diese Phase beläuft sich hauptsächlich auf das effiziente Zuordnen des Maschinencodes bezüglich des zu übersetzenden Programms. Diesen Schritt nennt man auch „Code Generation". Normalerweise ist diese Phase in mehrere Schritte eingeteilt, in denen unterschiedliche Optimierungen stattfinden.

- **Code Selection**
  An dieser Stelle der Verarbeitung wird entschieden, welche Assembler Instruktionen für die jeweiligen Code-Fragmente eingesetzt werden. Hierbei ist es das primäre Ziel, Instruktionen zusammenzufügen, die minimale Kosten erzeugen.

- **Data Routing**

  Nachdem die benötigten Operationen bei der „Code Selection“ zu Einheiten zusammengefasst worden sind, findet eine Minimierung von Transportkosten zwischen den einzelnen Einheiten statt.

- **Register Allocation**

  Die Aufgaben der Register-Allokation bestehen darin, zu entscheiden, an welchen Stellen des Speichers die Ergebnisse von Berechnungen während der Programmlaufzeit gespeichert werden sollen. Da die Anzahl der Register eines jeden Prozessors begrenzt ist, muss entschieden werden, welche Werte gleichzeitig in den Registern gehalten werden, um teure Speicherzugriffe zu vermeiden und gleichzeitig die physikalische Anzahl der Register nicht zu überschreiten.

- **Instruction Scheduling**

  In dieser Phase der Optimierung wird entschieden, an welcher Stelle Instruktionen während der Programmlaufzeit ausgeführt werden sollen. Es werden Abhängigkeiten der Operationen zu Hilfe gezogen sowie Prozessor-Merkmale und Einwirkungen wie Pipelining. Das primäre Ziel ist es, die benötigte Ausführzeit des Programms zu minimieren.

Diese Arten von Optimierungen arbeiten auf den unteren Ebenen von Zwischendarstellungen und orientieren sich an der Architektur der Prozessoren. Allerdings sind diese Optimierungsstrategien nur schwer in der Lage, komplexe Strukturen, wie sie in Kapitel 1.2 vorgestellt werden, zu optimieren.

1.2 Quellcode-Optimierung mit Intrinsics

Dieses Kapitel beschäftigt sich mit der Quellcode-Optimierung durch Einsatz von Quellcode-Erweiterungen. Durch die Ausnutzung von „Compiler-known Functions“, den so genannten Intrinsics, werden komplexe Code-Strukturen auf die optimalen Maschinenbefehle des zugrunde liegenden Digitalen Signalprozessors abgebildet.

Definition: Ein *Intrinsic* ist ein vordefiniertes Unterprogramm, das für spezielle Quellcode-Strukturen die optimalen Maschinenbefehle des zugehörigen Prozessors beinhaltet. Die auch *„Compiler-known Functions“* genannten Funktionen werden in Quellcode eingebettet und beim Übersetzen durch den Compiler auf die optimalen Maschinenbefehle abgebildet.

Die Aufgabe eines Compilers ist es, möglichst guten Maschinencode zu produzieren. Allerdings sind die heutzutage verwendeten Compiler nicht in der Lage, alle Eigenschaften eines Quellcode-Programms zu analysieren und zu optimieren. Um eine Kommunikation zwischen den heterogenen Systemen, die sich auf dem Markt befinden, zu gewährleisten, sind Referenz-Implementierungen der jeweiligen Standards notwendig. Durch die große Zahl der unterschiedlichen Prozessoren ist eine maschinenunabhängige Implementierung notwendig, damit jeder verwendete Compiler in der Lage ist, den zugrunde liegenden Quellcode für seinen Prozessor zu übersetzen. Dadurch geht die Möglichkeit verloren, maschinenspezifisch programmierten Code nutzen zu können. Diese Implementierungen können zwar durch die Compiler der jeweiligen Mikrokontroller übersetzt werden, sie sind aber nicht in der Lage, typische DSP-Codestrukturen zu erkennen und auf die optimalen Maschinenbefehle abzubilden. Daher sind immer wieder manuelle Code-Transformationen notwendig, die allerdings Zeit und damit Geld in Anspruch nehmen. Ziel ist es nun, den Quellcode einer Applikation für alle Eingebetteten Prozessoren optimal nutzbar zu machen.

An dieser Stelle kommen die Digitalen Signalprozessoren der Firma Texas Instruments ins Spiel. Mit Hilfe dieser Prozessoren und der dazugehörigen Entwicklungsumgebung soll eine effektive Nutzung der zur Verfügung stehenden Prozessor-Merkmale zum Einsatz kommen. Dieser Prozessor ist durch seine Architektur in der Lage, beispielsweise saturierende Arithmetik sehr effizient berechnen zu können. Außerdem verfügt dieser DSP über Multimedia-Befehle, mit denen es möglich wird, zwei 16 Bit-Befehle zu einem 32-Bit Befehl zu koppeln und gleichzeitig zu bearbeiten.

Allerdings ist der zugehörige Compiler der Firma Texas Instruments, wie die üblichen Compiler auch, nicht in der Lage, komplexe Quellcode-Strukturen, die durch ein Intrinsic optimiert werden könnten, zu erkennen und in effektiven Maschinencode zu übersetzen. Durch den einfachen Einsatz dieses Compilers ist

es daher nur eingeschränkt möglich, effizienten Maschinencode zu produzieren. Ein solches Code-Konstrukt zeigt das nächste Beispiel:

```
result = var1 + var2 ;
if ( ( ( var1 ^ var2 ) & 0x80000000 ) == 0 )
   if ( ( result ^ var1 ) & 0x80000000 )
       result = var1 < 0 ? 0x80000000: 0x7fff ffff;
```

Dieses Code-Fragment stammt aus der G.723.1-Standard Implementierung [G7231] der Firma Alcatel und findet sich ebenfalls im GSM-Standard [GSM] wieder. Es stellt eine Addition zweier Zahlen dar und eine darauf folgende Saturierung des Ergebnisses. Anhand dieses Beispiels erkennt man die Komplexität solcher Standard-Implementierungen. Es ist demnach notwendig, mehrere if-Statements zu verwenden, um den Überlauf einer Operation zu erkennen. Zusätzlich müssen noch Maximal- und Minimalwerte der Ergebnisvariablen zugewiesen werden. Durch den Einsatz eines Intrinsics und die damit verbundene optimale Abbildung des Quellcodes auf Maschinencode, wäre diese komplexe Struktur durch einen einzigen Zyklus realisierbar.
Der folgende Code zeigt den Aufruf eines Intrinsics des TI C6x, um zwei Zahlen zu addieren und eine Saturierung des Ergebnisses vorzunehmen.

```
result = _sadd ( var1 , var2 ) ;
```

Obwohl diese Möglichkeiten zur Verfügung stehen, wird dennoch sehr ineffizienter Maschinencode durch die Compiler generiert, da sie nicht in der Lage sind, solch typische Strukturen zu erkennen.
Die Ausnutzung von Intrinsics bietet allerdings noch weit mehr Einsatzmöglichkeiten. Eine weitere soll an dieser Stelle zusätzlich vorgestellt werden. Durch eine Bitbreiten-Reduktion und die anschließende Verwendung von Multimedia-Befehlen soll eine weitere Verbesserung der Laufzeit erzielt werden. Nahezu unmöglich scheint es für einen Compiler, unbenutzte und damit überflüssige Bitbreiten von Variablen zu erkennen. Durch eine bitgenaue Analyse des Quellcodes soll dies aber möglich gemacht und genutzt werden. In der zweiten Phase dieser Optimierung erfolgt nun eine Weiterverarbeitung der optimierten sowie der ursprünglich vorhandenen Variablen, denn nun ist es möglich, eine weitere Ei-

genschaft des TI C6x auszunutzen. Durch ein Zusammenfassen zweier Operationen, wie Additionen oder Subtraktionen, wird eine parallele Verarbeitung dieser Instruktionen durch die Nutzung eines Multimedia-Befehles möglich.

```
result1 = a + b;      ==>   tmp_result = _add2 ( a << 16 | c , b <<16 | d );
result2 = c + d;            result1 = tmp_result >> 16;
                            result2 = (short )tmp_result;
```

Es gibt bereits eine Reihe von Ansätzen, die dieses Problem behandeln, allerdings gibt es nur unzufriedenstellende Ergebnisse. Zum einen werden die notwendigen Intrinsics manuell eingefügt, oder es wird mittels eines „Pattern-Matching“ versucht, treffende Datenflüsse zu erkennen und zu optimieren. Dennoch werden Intrinsics aktuell benutzt, und es lassen sich dadurch signifikante Verbesserungen erzielen. Diese Diplomarbeit soll eine Brücke zwischen diesen Problemen bilden. Es soll anschaulich dargestellt werden, dass es möglich wird, eine Performance-Optimierung zu realisieren, die eine automatische Ausnutzung dieser Prozessor-Merkmale vorsieht. Durch eine Quellcode-Transformation werden Intrinsics in bereits bestehende Referenz-Implementierungen eingebettet, wie z.B. dem GSM-Standard [GSM] und dem G.723.1-Standard [G7231] zur Sprachkomprimierung, und ersetzen so implementierte Teile auf der Quellcode-Ebene.

1.3 Verwandte Arbeiten

In diesem Kapitel werden bereits bearbeitete Themen vorgestellt, die mit dem Thema dieser Diplomarbeit verwandt sind. Es wird gezeigt, an welchen Stellen der Arbeiten Schwächen zu finden sind und eine mögliche Optimierung stattfinden kann.

- **G.723.1 Dual-Rate Speech Coder: Multichannel TMS320C62x Implementation [Dil00]**
  Diese Arbeit beschreibt, wie der G.723.1 Dual-Rate Speech Coder auf dem TMS320C62x der Firma Texas Instruments implementiert wurde. Es wurden unter anderem Intrinsics für jede mögliche Art von Berechnung eingesetzt.

Darunter fällt auch die Optimierung von saturierender Arithmetik. Allerdings wurden all diese Änderungen manuell eingepflegt.

- **Optimierung eines 3D Image Rekonstruktions-Algorithmus [ALD02]**
  In dieser Arbeit wurde der TI C67x Prozessor aus der TI C6x Familie benutzt. Neben anderen Optimierungen, wie beispielsweise die Berechnungen mit Floating-Point Zahlen, werden auch hier Intrinsics eingesetzt. Da der TI C67x keine Divisions-Instruktion besitzt, wurde das „reciprocal"-Intrinsic benutzt. Mit Hilfe dieses Intrinsic berechnet man den Kehrwert eines Teilausdrucks und führt anstatt einer Division eine Multiplikation aus.

$$a = \frac{y2 - y1}{x2 - x1} \Rightarrow a = (y2 - y1) * _rcp(x2 - x1)$$

  Durch diese Optimierung wurden 30% der Laufzeit eingespart, jedoch wurden die Optimierungen von Hand eingepflegt.

- **Optimierung durch Multimedia-Instruktionen [PBSB02]**
  In dieser Arbeit kamen die Intrinsics zum ersten Mal automatisiert ins Spiel. Es wurde neben Intrinsics, die eine parallele Bearbeitung von Instruktion vorsehen, auch saturierende Arithmetik untersucht. Allerdings geschah dies durch ein Pattern-Matching. Die nächste Arbeit wird zeigen, dass dies nicht ausreichend ist.

- **Architekturabhängige Quellcodeoptimierung durch Mustererkennung [Jak02]**
  Hier werden unterschiedlichste Optimierungen durchgeführt, und an dieser Arbeit sieht man, dass es grundsätzlich möglich ist, Quellcode-Optimierungen zu nutzen und damit auch sehr gute Ergebnisse zu erzielen. Allerdings zeigt diese Arbeit auch, dass es einfaches Pattern-Matching nicht ausreicht, um allgemeingültigen Code effizient analysieren und optimieren zu können. Dies wird durch die Komplexität des Musters begründet. Je komplexer ein Muster wird, desto enger wird die Vorgabe der Programmstruktur. Das geht soweit, dass man nicht in der Lage ist, allgemeingültige Transformationen zu formulieren, die man auf beliebige, reale Code-Beispiele anwenden kann. Der große Nachteil dieser Arbeit ist, dass man Vorgaben zu machen hat, wie die zu

optimierenden Strukturen aussehen. Probleme gibt es da bei beliebig verschachtelten Schleifen oder bei If-Statements.

- **Erkennen von nicht verwendeten Bitbreiten [BG00]**
  Die Autoren dieser Arbeit stellen einen Algorithmus namens "Bit Value" vor. Dieser Algorithmus ist in der Lage, unbenutzte und konstante Bits in C-Programmen zu entdecken. Durch eine Datenflussanalyse, die der in dieser Arbeit sehr ähnlich ist, wird eine Vorwärts- und Rückwärtsbewegung des Datenflusses simuliert. So werden die Optimierungen „constant-folding", wie bereits beschrieben, und das Erkennen von ungenutzten Bits auf der Bit-Ebene durchgeführt. Das Ergebnis dieser Arbeit besagt, dass für die betrachteten Benchmarks insgesamt 36% der berechneten Bytes nicht verwendet werden und durchschnittlich 26,8% der berechneten Werte lediglich 16 Bits oder weniger benötigen

- **TI C62x Performance Code Optimization [CWK99]**
  In diesem Artikel werden unterschiedliche manuell eingeführte Optimierungen durch Intrinsics durchgeführt. Zum einen werden Multimedia-Befehle benutzt, um parallele Berechnungen von Instruktionen ausführen zu können. Zum anderen werden Intrinsics zur Berechnung von MAC-Operationen ausgenutzt. Durch diese Quellcode-Manipulationen wurden sehr erhebliche Laufzeitverbesserungen erzielt. Der Nachteil dieser Arbeit besteht allerdings darin, dass der gesamte Code von Hand optimiert wurde.

1.4 Ziele der Arbeit

Diese Diplomarbeit ist in dem Bereich der Eingebetteten Systeme angesiedelt. Das Ziel ist es, eine Automatisierung zu implementieren, die typischen DSP-Code auffinden, analysieren und durch Intrinsics ersetzen kann. Ausgehend von einer bereits bestehenden Implementierung einer bitgenauen Datenflussanalyse, die ursprünglich für eine PP32 Netzwerkprozessorarchitektur realisiert wurde [SPR197], soll es möglich gemacht werden, Datenflüsse und Codestrukturen zu erkennen, die durch einen digitalen Signalprozessor verarbeitet und optimiert werden können. Allerdings arbeitet diese Datenflussanalyse auf der Assembler-Ebene durch Ausnutzung von Low Level Intermediate Representations. Da diese

Arbeit eine Optimierung auf Quellcode-Ebene darstellen soll, ist es notwendig, ein weiteres Tool miteinzubringen.
Durch das SUIF Compiler System wird ein Kompilieren und Auftrennen von Source-Code realisiert, und damit ist man in der Lage, die auf Assembler-Ebene arbeitende Datenflussanalyse nutzbar zu machen.
Der durch SUIF kompilierte Code und die damit entstehenden Objekte müssen so in die bestehenden Strukturen der bitgenauen Datenfluss-Analyse portiert werden, dass sie die LLIR-Objekte ersetzen und an deren Stelle agieren. Durch die verwendete Datenflussanalyse wird es möglich, ein einfaches Pattern Matching zu überwinden und komplizierte Strukturen zu finden.
Als weiteres Ziel wird eine Bitbreiten-Reduktion ins Auge gefasst. Ebenfalls durch die bitgenaue Datenflussanalyse realisierbar, soll eine Simulation des vorliegenden Quellcodes durchgeführt werden. Anhand dessen soll erkannt werden, in welchen Bereichen Bitbreiten eingespart werden können. Als weiteren Schritt in dieser Optimierung werden Multimedia-Befehle eingeführt, die eine parallele Verarbeitung von Instruktionen möglich machen sollen.

Zusammenfassend werden folgende Ziele definiert:

1. Nutzbarmachung der bestehenden bitgenauen Datenfluss-Analyse
2. Auffinden von saturierender Arithmetik und ersetzen der Codestrukturen durch Compiler-known Functions
3. Erkennen von nicht verwendeten Bits durch eine Analyse des bestehenden Quellcodes und Reduktion auf wirklich notwendige Bitbreiten.
4. Einsatz von Multimedia-Befehlen, um parallele Verarbeitung zu ermöglichen

1.5 Struktur

Kapitel 2 beschreibt die Grundlagen und die Konzepte, die bei dieser Arbeit benutzt werden. Zunächst wird die Architektur TMS C6x von Texas Instruments sowie dessen Einsatzmöglichkeiten dargestellt, gefolgt von einer detaillierten Beschreibung des SUIF Compiler Systems.

Kapitel 3 beinhaltet die Darstellung der bitgenauen Datenflussanalyse. Zunächst werden die einzelnen Bestandteile des Datenflussgraphen erläutert, gefolgt von einer detaillierten Beschreibung der Datenflussanalyse.

Kapitel 4 stellt die Portierung der bereits bestehenden Implementierung der bitgenauen Datenflussanalyse auf das Compiler System SUIF dar. Durch die Portierung der Datenflussanalyse wird es erst möglich, sie für diese Arbeit nutzbar zu machen, um Quellcode analysieren und optimieren zu können .

Kapitel 5 beinhaltet das eigentliche Kernkapitel. An dieser Stelle werden die implementierten Optimierungen vorgestellt, die durchgeführt worden sind. Zunächst beschreibt es die Optimierung von saturierender Arithmetik, gefolgt von der Bitbreiten-Reduktion und den Multimedia-Befehlen.

Kapitel 6 zeigt die errungenen Ergebnisse auf. Anhand von Benchmarks werden die erreichten Laufzeitverbesserungen aufgeführt und dargestellt.

Kapitel 7 bildet eine Zusammenfassung der vorliegenden Arbeit.

2 Grundlagen und Konzepte

Dieses Kapitel beschäftigt sich mit den Grundlagen, die für die Optimierungen, die diese Arbeit vorstellt, benutzt wurden. Zunächst folgt in Kapitel 2.1 eine Darstellung der verwendeten Entwicklungsumgebung TMS C6000 der Firma Texas Instruments. An dieser Stelle werden die besonderen Eigenschaften der verwendeten DSPs vorgestellt. Ein kurzer Abriss über das Zweierkomplement und mögliche Probleme bei Instruktionen, wie Additionen und Multiplikationen, lassen Lösungsvorschläge für diese Problematiken folgen. Anschließend wird die verwendete Zwischendarstellung der „SUIF Stanford Compiler Group“ vorgestellt. Es werden die Eigenschaften dieses Werkzeugs sowie deren Verwendung ausführlich dargestellt.

2.1 Der TMS C6000 DSP von Texas Instruments

Die TMS C6000 DSP-Plattform wurde für die Ausführung rechenintensiver Verarbeitungsschritte entwickelt, die bei Anwendungen wie digitales Video, medizinische Bildverarbeitung, Maschineninspektion, Sicherheitsüberwachung, Druckern, Kopierern und Scannern erforderlich sind. Sie unterstützt die gesamte TI C6x Familie und liefert ebenfalls Simulatoren für die Benutzung ohne physische Hardware in Form eines Digitalen Signalprozessors.
Die flächendeckende Implementierung von Kabel- und Funknetzwerken führt zu einer steigenden Nachfrage nach der Integration von Video-, Sprach- und Daten-Transportdiensten, die in ein und demselben digitalen Strom übertragen werden. Die C6x DSP bieten eine anspruchsvolle VLIW-Architektur (Very Long Instruction Word), ein flexibles Speicherkonzept und I/O mit hoher Bandbreite, die eine optimale Unterstützung für hochleistungsfähige Video- und Imaging-Anwendungen bieten. Zu den Video-Infrastruktur-Anwendungen zählen Video-Gateways, Video-Server, Multimedia-Router, Mobilfunk-Multimedia-Basisstationen der dritten Generation und andere Systeme, von denen sowohl Privatkunden als auch Firmen und Regierungsorganisationen profitieren können. Endbenutzer-Anwendungen sind Videokonferenzen, Video-on-Demand, digitales Fernsehen, Medien-verarbeitende Anwendungen und andere Systeme für die Übertragung, Aufzeichnung und Wiedergabe von digitalem Video.

2.1.1 Architektur

Im Folgenden werden die Eigenschaften der Zielarchitektur aufgezeigt [SPR197]. Zunächst erfolgt eine Übersicht über die Architektur des TI C62x / C67x, gefolgt von einer Übersicht der zur Verfügung stehenden Instruktionen der jeweiligen funktionellen Einheit.

Die TI C6x Prozessoren bestehen aus drei Teilen: CPU (Kern), Peripherie und Speicher. Abbildung 2-1 zeigt das Blockschaltbild des TI C62x/C67x und gibt einen Überblick über die gegebene Architektur.

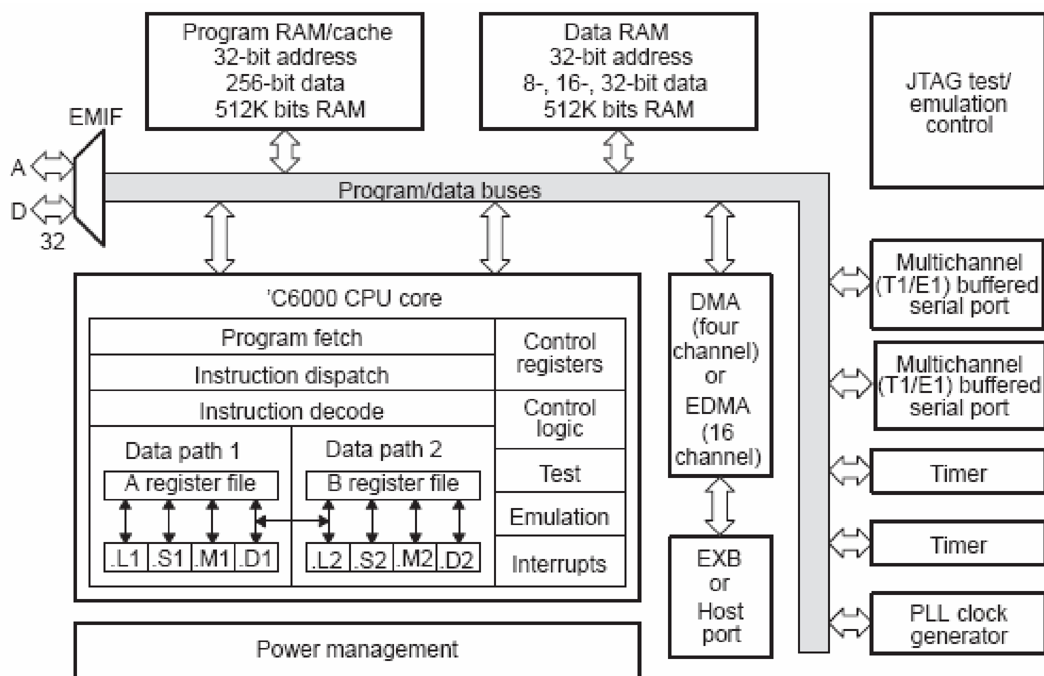


Abbildung 2-1: Blockschaltbild des TI C62x / C67x

Der CPU-Kern besitzt weiterhin

- zwei **Datenpfade** mit jeweils vier funktionellen Einheiten.

  Jeder Datenpfad besitzt ein Register-File mit jeweils 16 32-Bit Registern. Zusätzlich besteht ein Datenpfad aus folgenden funktionellen Einheiten:

  - **.L- Einheit:** 32/40-Bit Arithmetik und Vergleichoperationen
  - **.S-Einheit:** 32-Bit Arithmetik, 32/40-Bit Shifts, 32-Bit Bitfelder-Operationen, 32-Bit Logikoperationen, Sprünge
  - **.M-Einheit:** 16 x 16-Bit Multiplikationsoperationen
  - **.D-Einheit:** 32-Bit Addition, Subtraktion, lineare und zirkuläre Adressenberechnung, Load- und Store-Befehle

- eine **„Program Fetch“-Einheit**, eine **„Instruction Dispatch“-Einheit** und eine **„Instruction Decode“-Einheit**. Diese Blöcke können bis zu acht 32-Bit Instruktionen aus dem Speicher in die funktionelle Einheit innerhalb eines Zyklus laden.
- Kontrollregister
- eine Kontrolllogik
- sowie eine Logik für Tests, Emulationen und Interrupts

Abbildung 2-2 zeigt die zur Verfügung stehenden Instruktionen und die verantwortlichen funktionellen Einheiten. Es sind bis zu acht Instruktionen pro Zyklus ausführbar. Der Compiler muss während der Kompilierzeit die verwendeten Befehle den funktionellen Einheiten zuordnen. Um dies zu realisieren, besitzt jede 32-Bit Instruktion ein Bit, das angibt, ob der nächste auszuführende Befehl noch im gleichen Zyklus abgearbeitet wird.

Der DSP besitzt weiterhin folgende typische Eigenschaften:

- Saturierung
- Multimedia-Befehle (SIMD)
- Modulo-Adressierung
- VLIW-Architektur (Very Long Instruction Word, 256 Bit)
- Normalisierungsoperation
- Integrierte E/A-Leitungen

.L Unit	.M Unit	.S Unit		.D Unit	
ABS	MPY	ADD	SET	ADD	STB (15-bit offset)‡
ADD	MPYU	ADDK	SHL	ADDAB	STH (15-bit offset)‡
ADDU	MPYUS	ADD2	SHR	ADDAH	STW (15-bit offset)‡
AND	MPYSU	AND	SHRU	ADDAW	SUB
CMPEQ	MPYH	B disp	SHRL	LDB	SUBAB
CMPGT	MPYHU	B IRP†	SUB	LDBU	SUBAH
CMPGTU	MPYHUS	B NRP†	SUBU	LDH	SUBAW
CMPLT	MPYHSU	B reg	SUB2	LDHU	ZERO
CMPLTU	MPYHL	CLR	XOR	LDW	
LMBD	MPYHLU	EXT	ZERO	LDB (15-bit offset)‡	
MV	MPYHULS	EXTU		LDBU (15-bit offset)‡	
NEG	MPYHSLU	MV		LDH (15-bit offset)‡	
NORM	MPYLH	MVC†		LDHU (15-bit offset)‡	
NOT	MPYLHU	MVK		LDW (15-bit offset)‡	
OR	MPYLUHS	MVKH		MV	
SADD	MPYLSHU	MVKLH		STB	
SAT	SMPY	NEG		STH	
SSUB	SMPYHL	NOT		STW	
SUB	SMPYLH	OR			
SUBU	SMPYH				
SUBC					
XOR					
ZERO					

† S2 only
‡ D2 only

Abbildung 2-2: Fixed Point Instruction to functional unit mapping, TMS320 C6000 Technical Brief, Februar 1999, Seiten 2-11

2.1.2 Intrinsics – Compiler-known Functions

In diesem Unterkapitel werden die Einsatzmöglichkeiten der in Kapitel 1.2 aufgezeigten Intrinsics dargestellt. Wie bereits erwähnt, sind Intrinsics auf der Quellcode-Ebene in Form von Funktionsaufrufen und damit für den Entwickler sehr leicht anwendbar.

Zahlreiche Intrinsics zählen zum Funktionsumfang. Hierunter fallen unter anderem, für diese Arbeit sehr interessant, saturierende Operationen und Multimedia-Befehle. Im Folgenden werden diese Funktionen genauer erläutert. Zuvor werden jedoch einige der benötigen Begriffe erklärt.

2.1.2.1 Zweierkomplement

Man benutzt das Zweierkomplement, um in der Lage zu sein, sowie positive als auch negative Zahlen darstellen zu können. Man bildet es, indem man von jeder Zahl das Komplement bildet und 1 addiert.

Daraus ergeben sich folgende Darstellungen einer 4-Bit Zahl:

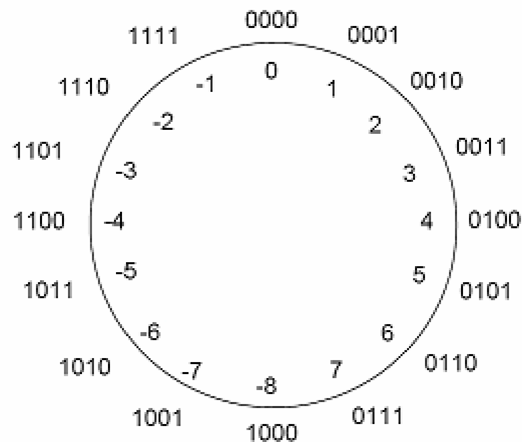


Abbildung 2-3: Zweierkomplement einer 4 Bit-Zahl

Das Zweierkomplement benutzt das hochwertigste Bit als Vorzeichen, wobei eine 0 eine positive Zahl, und eine 1 eine negative Zahl darstellt. Es ist wichtig zu bemerken, dass eine negative Zahl mehr existiert als positive, nämlich die kleinste negative Zahl.

2.1.2.2 Überlauf

Bei den Berechnungen von Additionen, Subtraktionen oder Multiplikationen kann es zu Problemen in Form von Überläufen kommen. Es ist darauf zu achten, dass es nicht ohne weiteres möglich ist, beliebig lange Zahlen zu berechnen. Dies liegt begründet in der Beschaffung der gültigen Wertebereiche, die in der Informatik aufgrund von Speicherkapazitäten begrenzt sind. Als Beispiel soll nun eine 4 Bit große Zahl ausreichen, um zu demonstrieren, wie ein Überlauf zu Stande kommen kann. Die größte mit 4 Bit darstellbare Zahl ist 1111, also dezimal 15. Wird

nun eine Zahl, beispielsweise 1 hinzu addiert, erhalten wir nicht die erwartete Zahl 16, sondern eine 0. Da bekanntlich 1+1 = 0 + Übertrag gilt, bekommt man um genau zu sein nicht 0000, sondern 0000 + Übertrag. Der letzte Übertrag kann aufgrund der begrenzten Bitanzahl nicht mehr gespeichert werden. Der Übertrag wird ersatzlos gestrichen und es kommt zu einem Überlauf. Damit beginnt der Wertebereich wieder von vorne.

Bei der Benutzung des Zweierkomplements zur Darstellung sowohl positiver als auch negativer Zahlen steht ein Bit weniger zur Verfügung, da das höchstwertige Bit zur Darstellung des Vorzeichens benutzt wird. Daher sind in diesem Fall 0111, dezimal 7, und 1000, dezimal -8 die größten Zahlen, die dargestellt werden können. Das Ergebnis von 7+1 ist demnach auch nicht 0000, sondern 1000, da der Übertrag in das höchstwertige Bit übertragen und nicht gestrichen wird. Daher erhalten wir als Ergebnis dieser Addition -8. Zu beachten ist, dass ein Überlauf nur stattfinden kann, falls die Vorzeichen der verwendeten Operanden je nach Instruktion passend sind. Das bedeutet, dass ein Überlauf bei einer Addition nur passieren kann, wenn beide Vorzeichen positiv oder beide negativ sind. Andersherum läuft es bei der Subtraktion. In diesem Fall müssen die verwendeten Vorzeichen unterschiedlich sein. Ist dies nicht der Fall, kommt es nie zu einem Überlauf, da der Wertebereich nicht überschritten werden kann. Weitere Erklärungen und ausführliche Darstellungen folgen in Kapitel 5.1 bei der Analyse der möglichen Saturierungsformen.

Instruktionen werden in der Regel mit größeren Wertebereichen ausgeführt. Dennoch bleibt die Verhaltesweise dieselbe, wie Abbildung 2-4 zeigt.

```
Int a = 15000;                    0011101010011000
Int b = 20000;        ⟹         + 0100111000100000
Int c = a + b;                  ------------------
                                  1000100010111000
```

Abbildung 2-4: Übertrag bei einer Addition

Man erkennt am obigen Beispiel gut, dass es zu einem Überlauf kommt und somit ein Vorzeichenwechsel stattfindet. Um dies zu vermeiden, benutzt man Techniken der saturierenden Arithmetik.

2.1.2.3 Saturierende Arithmetik

Die Aufgabe der saturierenden Arithmetik ist es, einen möglichen Überlauf, der durch eine Operation passieren kann, aufzufangen und das Resultat auf ein gegebenes Maximum oder Minimum zu setzen. Notwendig wird dies in der Praxis beispielsweise bei der Bildverarbeitung. Möchte der Benutzer die Helligkeit eines Bildes erhöhen und wird dabei ein Pixel zu stark erhellt, käme es möglicherweise zu einem Überlauf. In diesem Fall würde der resultierende Wert des Bildpunkts ins Negative überschlagen, und anstatt einer aufhellenden Wirkung gäbe eine es Verdunkelung.

Das nächste Code-Beispiel zeigt eine solche Struktur. Es werden zwei Zahlen a und b mit einer jeweiligen Länge von 32 Bits addiert. Anschließend erfolgt die Überprüfung auf einen Überlauf und die Sättigung der Ergebnisvariablen. Hierbei werden keine Intrinsics für die Saturierung genutzt.

```
int L_add ( int a , int b )
{
        int result;
        result = a + b;
        if (((a ^ b) & 0x80000000 == 0) {
                if (( result ^ a ) & 0x80000000 ) == 0 ) {
                    result = ( a < 0 ) ? 0x80000000: 0x7FFFFFFF;
                }
        }
        return result;
}
```

Ein Überlauf bei einer Addition ist nur möglich, wenn eine Instruktion zwei Zahlen mit gleichem Vorzeichen addieren muss. Daher wird in der ersten Zeile nach der eigentlich Addition anhand einer Bit-Maskierung geprüft, ob das der Fall ist. Darauf folgt wiederum durch eine Bit-Maskierung eine Überprüfung des Ergebnisses dieser Addition. Falls die resultierende Variable ein anderes Vorzeichen besitzt als die beiden verwendeten Operanden, wurde ein Überlauf gefunden, und es erfolgt eine Saturierung des Ergebnisses. Abhängig von dem Vorzeichen der Operanden wird das Ergebnis auf das Maximum oder Minimum des verwendeten Wertebereichs gesetzt.

Diesen komplizierten Code zu einer saturierten Addition kann man durch den Aufruf eines Intrinsics ersetzen.

result = _sadd (a , b);

Durch den einfachen Aufruf einer Funktion, hier „_sadd“, und der Übergabe der erforderlichen Parameter ist es möglich, den gesamten oben dargestellten Code zu vereinfachen.

2.1.2.4 Multimedia-Befehle - SIMD

SIMD bedeutet “Single Instruction, Multiple Data”. Diese Erweiterungen kamen erstmalig 1994 in den Mikroprozessoren MAX2 von HP und in den „VS Erweiterungen“ von SUN zu Tage. Daraufhin fanden wir sie auch in den MMX, MVI und den 3DNow! -Techniken wieder [SIMD05]. Diese Erweiterungen machten es Programmierern möglich, Performance-Steigerungen durch Einsatz von gegebenen Hilfsmitteln der Mikroprozessoren zu erreichen. Allerdings geht der Einsatz dieser Techniken auf die Komplexität der Software über, da es außer der eigentlichen Berechnung der Instruktion auch noch zu weiteren Operationen kommt. Zum einen müssen die jeweiligen Operanden zusammengefasst, und zum anderen nach der Berechnung wieder getrennt werden. Dieses Verhalten beschreibt Abbildung 2-5. Daher wird die Benutzung der SIMDs meist nur in den Gebieten eingesetzt, in denen die höchste Priorität auf der Geschwindigkeit bzw. der Performance der Software liegt. Beschleunigungen von Berechnungen verschieben mögliche Flaschenhälse der Performance in die Hierarchie der Speicherverwaltung der jeweiligen Prozessoren, da zusätzliche Instruktionen notwendig sind.

Traditionelle Berechnungen füllen lediglich ein einzelnes Datenelement in ein Register, ohne Berücksichtigung, ob weiterer Platz in diesem Register übrig bleibt, der für weitere multiple Berechnungen genutzt werden könnte. Ein SIMD-System dagegen füllt mehrere Datenelemente in ein Register und führt ein und dieselbe Berechnung über alle Elemente in der gleichen Zeit aus. Abbildung 2-5 zeigt die Handhabung eines solchen Multimedia-Befehls. Bei zwei gegebenen Instruktionen mit demselben Operator bilden jeweils zwei Operanden der unterschiedlichen Instruktionen die oberen und unteren Bytes der neuen Operanden, die für den Multimedia-Befehl benötigt werden. Das bedeutet, dass die 16 Bits

des ersten Operanden der ersten Instruktion die oberen zwei Bytes und die 16 Bits des ersten Operanden der zweiten Instruktion das dritte und vierte Byte des neuen SIMD-Operanden bilden. Analog dazu werden die oberen und unteren zwei Bytes des zweiten SIMD-Operanden zusammengefasst. Dieses Verfahren ist in Abbildung 2-5 unter 1) zu sehen.

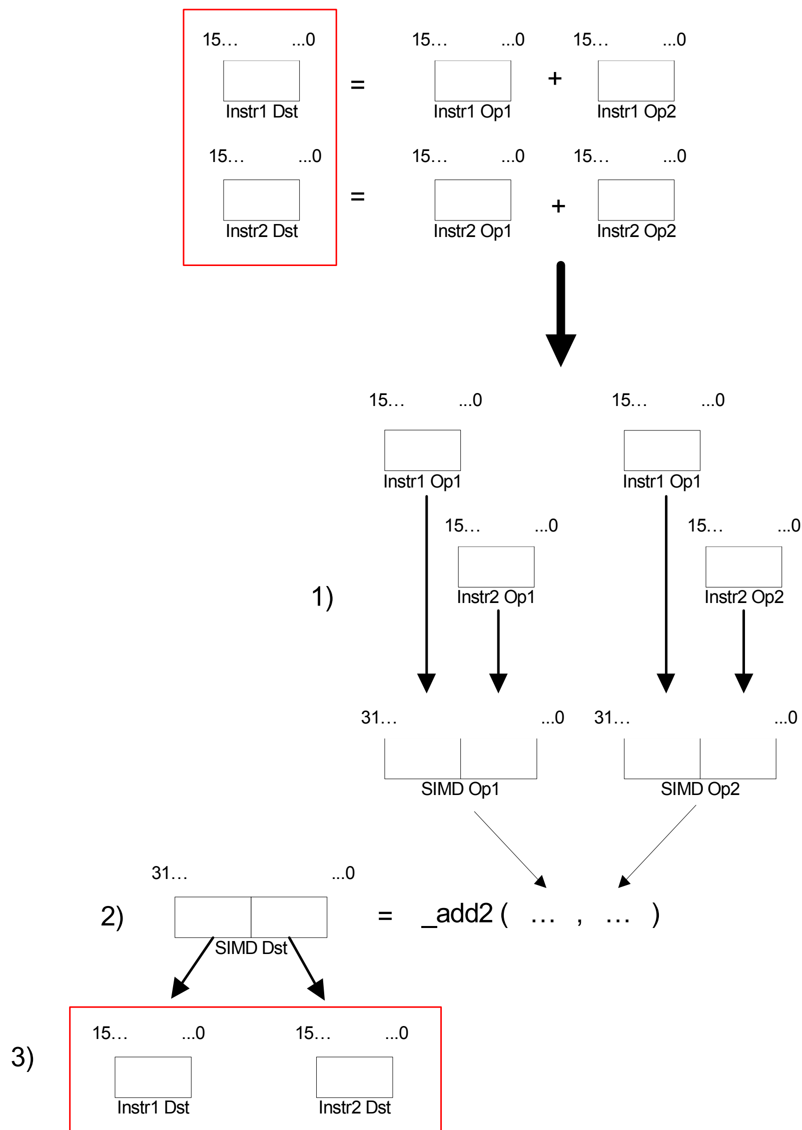


Abbildung 2-5: SIMD-Struktur

Der nächste Schritt, hier unter 2) zu erkennen, behandelt den eigentlichen Aufruf des SIMD-Intrinsics. Die jeweils zwei oberen und unteren Bytes der neu erschaff-

ten Operanden werden getrennt berechnet und in der Variablen „SIMD_Dst“ abgelegt. Anschließend werden die einzelnen Resultate aus der resultierenden SIMD-Variable gelöst und bilden somit die Ergebnisse der ursprünglichen Instruktionen. Zu sehen ist diese Handhabung unter 3) der Abbildung 2-5.

2.2 SUIF Compiler System

In diesem Kapitel wird die zugrunde gelegte Zwischendarstellung, die in dieser Diplomarbeit benutzt wird, vorgestellt. Zwischendarstellungen werden verwendet, um den zu kompilierenden bzw. optimierenden Quellcode in handhabbare Bestandteile aufzuteilen. Durch sie ist es möglich, maschinenunabhängige Optimierungen am Quellcode vorzunehmen. Der Zwischencode stellt eine Datenstruktur dar, die während der Übersetzungszeit generiert und verwendet wird. Nach möglichen Analysen und Optimierungen wird dieser Zwischencode in die Zielsprache überführt. Wie in Abbildung 2-6 zu sehen, besitzt SUIF [SCG94] ein Frontend und ein Backend. Das Frontend ist in der Lage, C und Fortran zu verarbeiten.

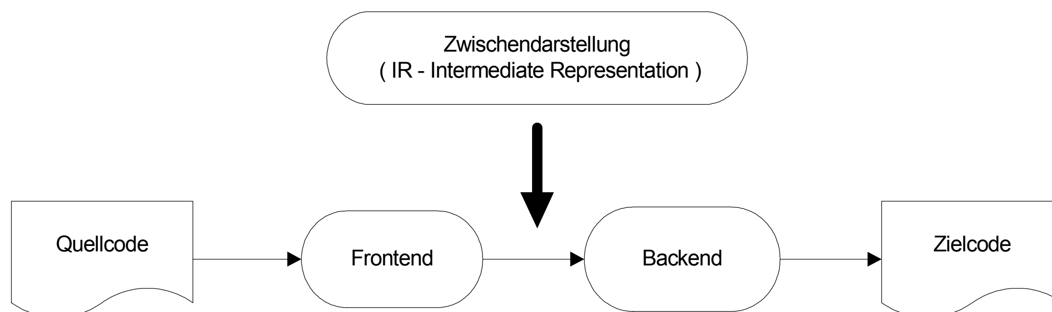


Abbildung 2-6: Stand der Zwischendarstellung

Das besondere am SUIF Compiler-System ist die Möglichkeit, die Zwischendarstellung nach der Bearbeitung wieder in die Programmiersprache des Quellcodes zu überführen. Zum Zeitpunkt dieser Arbeit war lediglich ein anderes System in der Entwicklung, das diese Funktion hätte liefern können [ICD05].

SUIF besitzt zwei Abstraktionsgrade. Durch „High-SUIF“ ist es möglich, einen abstrakten Syntaxbaum zu erstellen, der sämtliche Informationen beinhaltet, um aus dem Zwischencode heraus wieder die Quellsprache zu generieren. Durch

„Low-SUIF“ werden alle Bestandteile dieses Syntaxbaumes in einer assemblerartigen Darstellung und vorzugsweise in Listen gespeichert.
Das SUIF Compiler System ist ein flexibles Framework. Es besteht aus zwei Konzepten. Grundsätzlich besteht es darin, Quellcode in eine Zwischendarstellung (Intermediate Representation – IR) zu transferieren. Zum anderen stellt SUIF bereits bestehendes flexibles Modul-System zur Verfügung. Darin enthalten ist bereits eine Fülle von vordefinierten IR-Knoten, die der Anwender benutzen kann, um Analysen oder Quellcode-Transformationen vorzunehmen. Diese Zwischendarstellung wird im weiteren Verlauf dieser Arbeit benutzt, um gegebenen Quellcode untersuchen und optimieren zu können. Die nächsten Kapitel sind aus der SUIF Dokumentation entnommen und beschreiben die Struktur des SUIF Compiler-Systems detaillierter. Zunächst wird in Kapitel 2.2.1 die File Ebene und anschließend in Kapitel 2.2.2 die Prozedur Ebene dargestellt.

2.2.1 File Ebene

SUIF wurde entworfen, um interprozedurale Analysen und Optimierungen durchzuführen. An der Spitze der Hierarchie eines SUIF Programms steht eine Sammlung von Files, die wiederum eine Liste von Fileset-Einträgen beinhaltet, die durch SUIF kompiliert wurden. Abbildung 2-7 stellt die Struktur der File Ebene dar. Als übergeordnetes Element steht das „Fileset“. Es beinhaltet die Sammlung der „Files“, die durch SUIF bearbeitet werden sollen. Jedes „File“ beinhaltet die Prozeduren, die in dieser Datei integriert sind. Weiterhin werden die Prozeduren in Blöcke aufgeteilt, die zusammengehörige Bestandteile der Prozeduren darstellen. Diese Blöcke werden als Knoten in dem abstrakten Syntaxbaum gespeichert. Sie wiederum können abermals weitere Blöcke des Codes enthalten. Die Weiterhin beinhaltet das Fileset die globale Symboltabelle. Diese Tabelle ist für alle „Files“ sichtbar und verwendbar. So ermöglicht SUIF eine interprozedurale Analyse. So zeigen Referenzen von Symbolen oder Typen auf die gleichen Einträge und es ermöglicht zu erkennen, dass es sich um dieselben Einträge handelt. Weiterhin beinhaltet jedes File seine eigene Symboltabelle, um private Einträge für jedes File darstellen zu können.
Durch die globale Symboltabelle sind tiefere Ebenen der Hierarchie erreichbar. Wie in Abbildung 2-7 zu sehen, werden die Einträge der Symboltabellen mit den Einträgen der einzelnen Prozeduren verknüpft, sobald die Prozeduren als Input durch SUIF kompiliert werden.

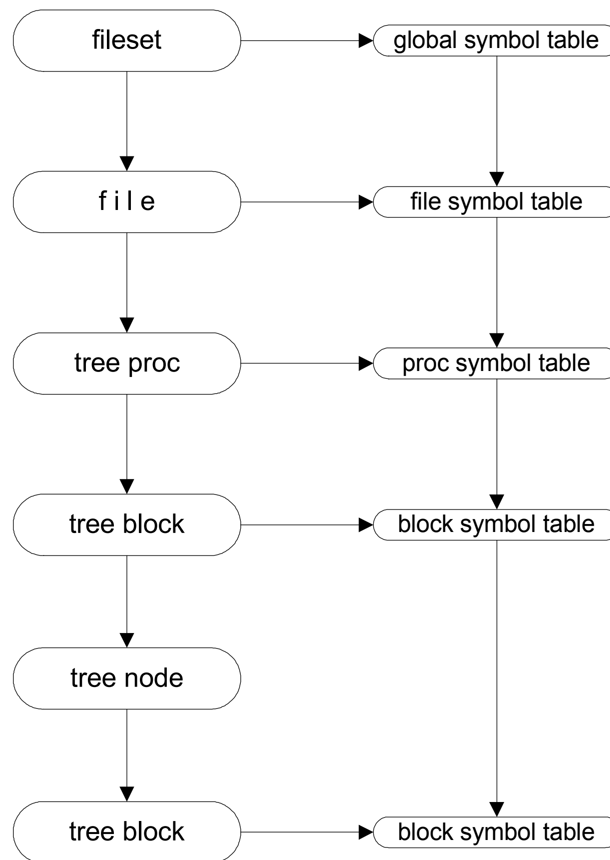


Abbildung 2-7: SUIF File Ebene

2.2.2 Prozedur Ebene

Prozeduren werden, wie bereits erwähnt, durch zwei Arten von Zwischendarstellungen angezeigt. In der ersten Phase der Verarbeitung wird eine „High-SUIF"-Darstellung erzeugt, die sprachenunabhängig einen abstrakten Syntax-Baum erzeugt (AST). In der zweiten Phase des Kompiliervorgangs werden diese Syntax-Bäume zu Listen reduziert und bilden das „Low-SUIF". Beide Darstellungen basieren auf demselben Syntax-Baum und unterscheiden sich nur in der Fülle der Informationen. Aber dennoch stellen beide Darstellungen dieselbe Prozedur dar, und es ist möglich, diese Darstellungen zu mischen. Im Einzelnen bestehen folgende Darstellungen der einzelnen Knoten:

- Die Blätter der ASTs bilden die **Knoten der Instruktionen**. Sie beinhalten einzelne Operationen oder einen weiteren Expression-Tree. In Low-SUIF werden demnach nur noch Listen von Instruktionen festgehalten.

- **„tree-block“-Knoten** beinhalten eingebettete Bereiche im Quellcode. Jeder einzelne Block besitzt eine eigene Symboltabelle, um Deklarationen innerhalb dieser darzustellen, und es ist nicht möglich, auf diese von außerhalb zuzugreifen.

- **„tree-if“-Knoten** stellen typische if-then-else Anweisungen dar. Sie bestehen aus drei Teilen, jeweils eine Liste für den Kopf der Anweisung, sowie für den then-Teil und den else-Teil.

- **„tree-loop“-Knoten** stellen while-do Bedingen dar. Diese Art von Knoten teilen sich in zwei Bestandteile auf. Einerseits in den Schleifenkörper und andererseits in eine Liste, die Konditionen der Schleife prüft.

- **„tree-for“-Knoten** bilden die weit kompliziertere Art der Schleifendarstellung. Neben einer Liste für den Schleifenkörper und den Schleifenkopf müssen Variablen für den Index, die obere und untere Grenze, sowie für die Größe der Schritte festgehalten werden.

Diese Knoten sind wie in Abbildung 2-8 angeordnet. Die Wurzel des abstrakten Syntaxbaumes wird durch das Fileset dargestellt, das sich wiederum in die einzelnen zu bearbeitenden Dateien aufteilt. Ein „Fileset-Entry“ beinhaltet eine Anzahl Prozeduren, wie sie im ursprünglichen Quellcode zu finden sind. Diese Ebene besitzt nun eine Liste, in der die einzelnen Bestandteile des Quellcodes in Blöcke aufgeteilt sind. Die einzelnen Bestandteile werden aus den oben beschriebenen Knoten zusammengesetzt. Abhängig von der Art des Knotens beinhalten sie die erforderlichen Informationen, die durch den Quellcode gegeben sind. Der „tree-if“-Knoten spaltet sich in drei Listen auf, die den Test-Part, den Then-Part und den Else-Part darstellen. Der Knoten der Instruktionen beinhaltet die eigentliche Instruktion. Zusätzlich werden die verwendeten Operanden gespeichert. Operanden können wiederum Instruktionen oder Symbole, sprich Variablen beinhalten. So ist es möglich, verschachtelte Instruktionen darzustellen. Der „tree-loop“-Knoten besteht aus zwei Listen. Zum einen wird der Körper der

Schleife gespeichert, und zum anderen der Test-Part, der angibt unter welchen Bedingungen der Körper ausgeführt wird. Der „tree-for"-Knoten ist der umfangreichste Bestandteil der Prozedur-Ebene. Hier muss neben dem Schleifenkörper zusätzlich der Indep, die obere und untere Grenze, sowie die Schrittweite festgehalten werden.

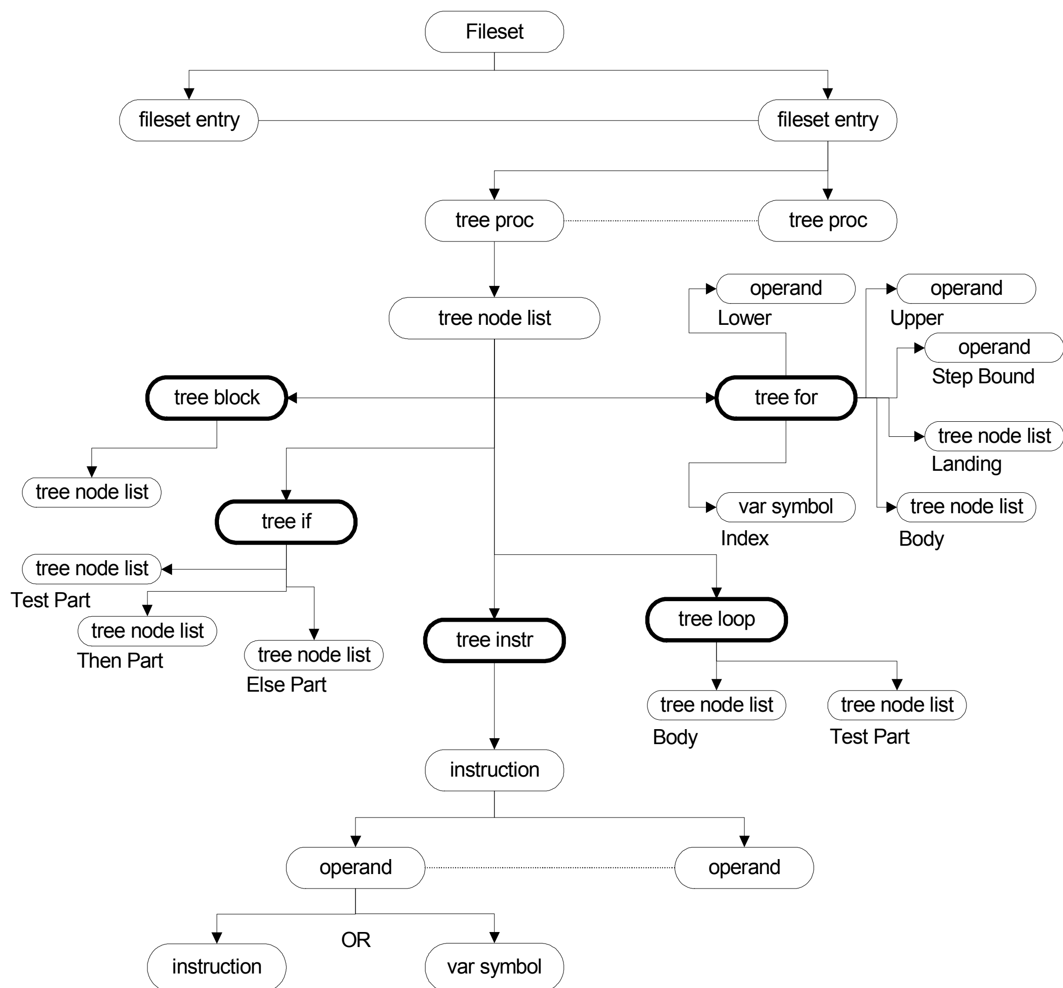


Abbildung 2-8: SUIF Procedure Level

3 Bitgenaue Datenfluss-Analyse

In diesem Kapitel wird die grundlegende Basis für die während der Diplomarbeit erstellen Optimierungen vorgestellt. Ausgehend von einer Implementierung für den PP32 Prozessor, wurde die darin enthaltene Datenfluss-Analyse herausgelöst und grundlegend für den Gebrauch von SUIF umgestaltet, so dass sie für die Zwecke dieser Arbeit benutzt werden konnte. Die Motivation, diese Datenfluss-Analyse zu benutzen, liegt darin, dass durch die bitweise Simulation des zu optimierenden Quellcodes eine genaue Darstellung jedes Datenflusses über die Kanten des Datenflussgraphen berechnet werden kann. So ist es möglich, nicht benutzte Bitbreiten zu erkennen und zu optimieren. Neben dieser Möglichkeit werden auch die anderen Eigenschaften einer Datenflussanalyse benutzt, um Analysen durchzuführen. Die konzeptionelle Entwicklung und implementierungstechnische Umsetzung stellen Wagner und Leupers in [WL01] vor.

Die Struktur der bitgenauen Datenfluss-Analyse wurde modular implementiert, so dass man einen leichten Zugang zu den einzelnen Komponenten finden kann. Folgend wird zunächst die ursprüngliche Struktur der Datenfluss-Analyse dargestellt, die in Abbildung 3-1 zu sehen ist.

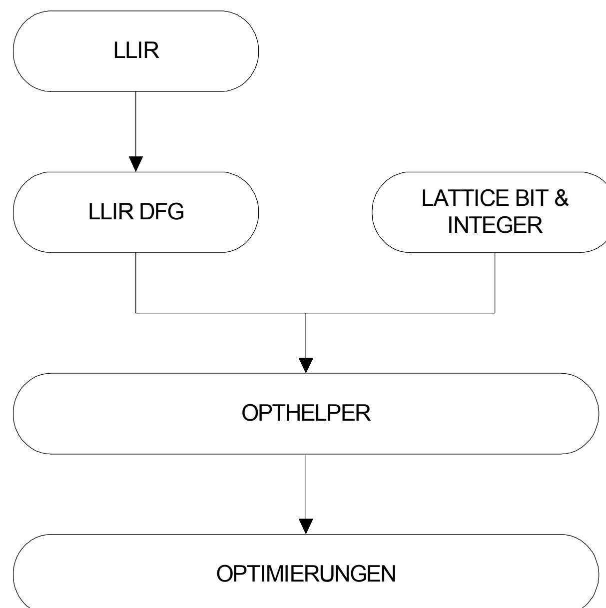


Abbildung 3-1: Ursprüngliche Struktur der PP32 Implementierung

In der Struktur, die in Abbildung 3-1 gezeigt wird, werden nur die Bestandteile aufgeführt, die für diese Diplomarbeit zu verwerten waren. Die dargestellten Pfeile sind so zu verstehen, dass die miteinander verbundenen Module in Laufrichtung aufeinander aufgebaut werden. Die verwendete Zwischendarstellung des ursprünglichen Projektes beruht auf der LLIR[1], die genauer in Kapitel 3.1 aufgezeigt wird. Sie baut auf den Datenflussgraphen auf. Die Klasse der „Lattice Bit & Integer“ beinhaltet die Darstellung von Bitstrings, die später an die Kanten des Datenflussgraphen angehängt werden, um die bitgenauen Datenflüsse zu repräsentieren. Eine genaue Darstellung wird in Kapitel 3.2 aufgegriffen. Den Kern der Implementierung bildet der „Opthelper“. An dieser Stelle wird die Simulation durchgeführt. Diese Klasse beinhaltet ebenso verschiedene Methoden, um Abfragen über verwendete Strukturen zu erstellen. Eine genauere Betrachtung erfolgt in Kapitel 3.4, nachdem zunächst in 3.3 die Struktur des Datenflussgraphen vorgestellt wurde.

3.1 LLIR

Der grundlegende Baustein ist die LLIR. Diese Komponente liest ein Assemblerprogramm ein und erstellt eine Objektstruktur. Nun ist es möglich, verschiedene Arten von Analysen und Änderungen an den einzelnen Objekten vorzunehmen und das resultierende Ergebnis wieder als Assemblerprogramm zurück zu schreiben. Abbildung 3-2 zeigt die Aufteilung dieser Objektstruktur.

Ein Assemblerprogramm ist in der Regel folgendermaßen gegliedert. Ein Programm enthält Funktionen, die sich aus einem oder mehreren Basisblöcken zusammensetzen. Diese Basisblöcke setzen sich aus hintereinander angeordneten Instruktionen zusammen, die wiederum möglicherweise mehrere Operationen beinhalten können. Anders als in der Hochsprache C gibt es in der verwendeten Assembler-Sprache Befehle, die mehr als eine Operation in einer Instruktion beinhalten können. C dagegen besitzt genau eine Operation pro Instruktion und hat daher auch nicht mehr als eine ausgehende Kante für jeden Knoten des Datenflussgraphen.

Operationen werden nun mit einer aus der Syntax festgelegten Anzahl von Parametern bedient. Wie in Abbildung 3-2 zu erkennen ist, wird diese Struktur detailliert durch die Klassenstruktur der LLIR abgebildet. Auf der untersten Ebene

[1] LLIR – Low Level Intermediate Representation

befinden sich die einzelnen Parameter der Operationen, und zusätzlich ist es durch die LLIR möglich, Informationen über zugehörige Registereinträge zu speichern und abzufragen. Diese Elemente werden durch die LLIR_Register und LLIR_Parameter-Klasse abgebildet. Eine Stufe höher befinden sich die LLIR_Operation-Klasse, die die Operationen des Assemblerprogramms beinhalten. Durch die LLIR_Instruction-Klassen werden die Operationen zusammengehalten. Die Klassen LLIR_BB und LLIR_Function repräsentieren demnach die Basisblöcke sowie die einzelnen Funktionen.

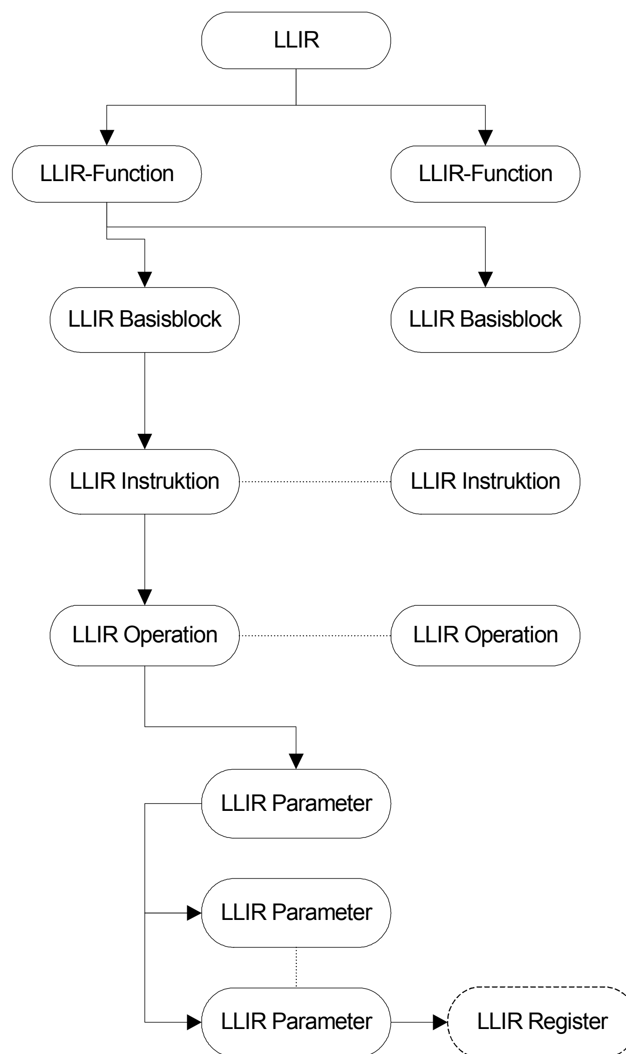


Abbildung 3-2: LLIR-Hierarchie

3.2 LInt

Die LInt-Klasse ist ein weiterer sehr wichtiger Bestandteil der Datenfluss-Analyse. Mit Hilfe dieser Klasse werden die Bitvektoren, die über die Kanten des Datenflussgraphen fließen, gespeichert. Ein Objekt der Klasse LInt besitzt demnach eine Breite gemäß der verwendeten Bitbreiten des zu untersuchenden Programms. Für jede Stelle dieses Vektors wird ein Wert gesetzt, der den Inhalt eines jeden Bits repräsentiert.
Durch Einsatz von Überladungen werden die Operatoren, sowohl arithmetische als auch logische, für die jeweilige Plattform implementiert und somit so einfach zu handhaben wie die Berechnung eines Integers. Um jedes einzelne Bit während der Analyse festhalten zu können, wird eine 6-wertige Logik benutzt.

- **U** – Der Wert des Bits ist unbekannt, und es ist ebenfalls nicht ersichtlich, an welcher Stelle das Bit berechnet wird. Beispiele hierfür sind unter anderem Parametereingaben.

- **L** – Der Wert des Bits ist unbekannt, allerdings konnte im Verlauf der Analyse festgestellt werden, an welcher Stelle das Bit berechnet wird.

- **N** – Dieser Wert ist gleich dem L-Wert, jedoch wurde dieses Bit im Laufe der Verarbeitung invertiert.

- **1** – Der Wert des Bits ist konstant 1

- **0** – Der Wert des Bits ist konstant 0

- **X** – Das Bit kann einen beliebigen Wert erhalten. „Don´t-Care“ wird benutzt, wenn erkannt wird, dass dieses Bit nicht mit in die Berechnung einbezogen wird.

In Abbildung 3-3 wird der Informationsgehalt der 6-wertigen Logik dargestellt. Zu erkennen ist, dass die wertvollste Information der „Don´t-Care“-Wert ist. Anhand dieses Wertes, der in der zweiten Phase der Datenflussanalyse erkannt werden kann, ist es möglich, Optimierungen anhand der tatsächlich benötigten Bitbreite

durchführen zu können. Diese Art der Optimierung wird in Kapitel 5.2 ausführlich erläutert.

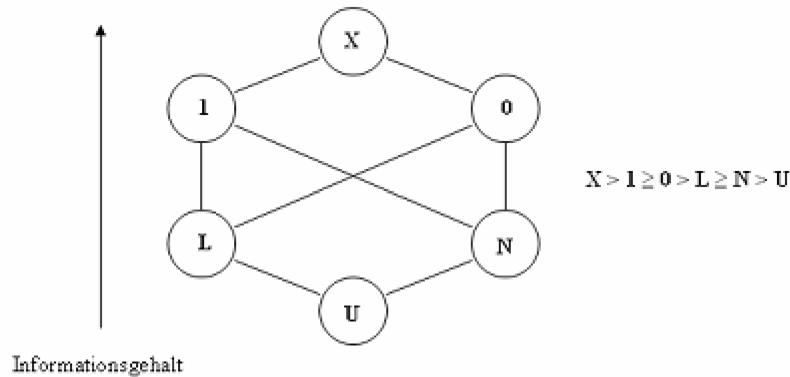


Abbildung 3-3: Halbordung der Bit-Information

3.3 Datenflussgraph

Der Datenflussgraph besteht aus drei Klassen, die die Knoten, Kanten und den zusammenhängenden Graphen darstellen. Der Graph stellt sämtliche Funktionen zur Verfügung, die man braucht, um Manipulationen durchzuführen. So ist es möglich, neue Knoten oder Kanten einzufügen, Objekte zu löschen oder Anfragen an die Existenz von Objekten zu stellen.

Definition: Der Datenflussgraph ist ein gerichteter Graph G=(V,E). Jeder Knoten v Є V repräsentiert entweder eine Instruktion oder einen Parameter in einer Prozedur. Jede Kante $e=(v^1, v^2)$ Є E repräsentiert einen Datenfluss in der Prozedur, d.h. v^2 benutzt Daten, die von v^1 zur Verfügung gestellt werden. Die letzte Kante in einem Pfad, die einen Zyklus bildet, heißt Back-Kante.

Abbildung 3-4 zeigt einen solchen Datenflussgraphen. Allerdings werden hier schon die Informationen angezeigt, die zusätzlich durch den Graphen gespeichert werden.

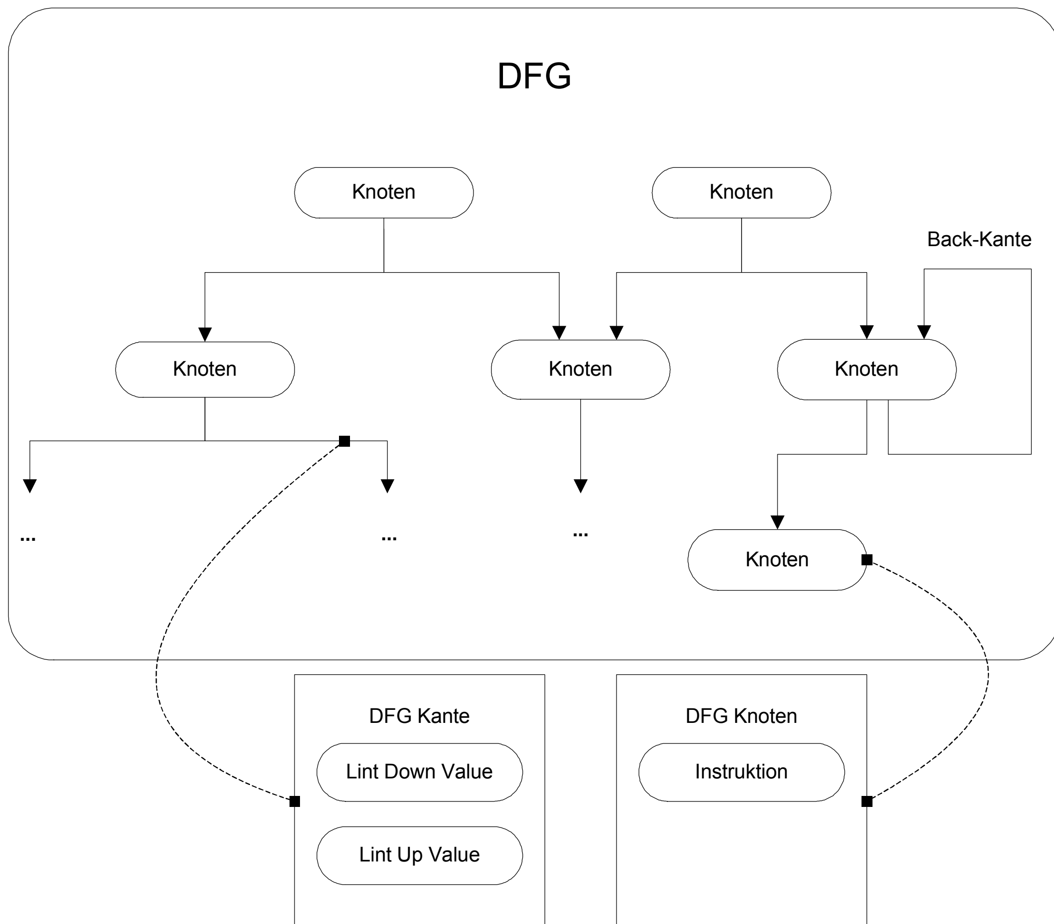


Abbildung 3-4: Datenflussgraph

Wie in Abbildung 3-4 dargestellt, besteht ein Datenflussgraph aus Knoten und Kanten. Knoten werden unterschieden zwischen Quellen, Senken sowie inneren Knoten. Quellen haben die Eigenschaft, dass sie nur mit Kanten verbunden sind, die aus ihnen heraus fließen. Im Gegensatz dazu stehen die Senken, die nur eingehende Kanten besitzen. Die restlichen Knoten, die mit eingehenden sowie ausgehenden Kanten verbunden sind, heißen innere Knoten. Quellen können sowohl Instruktionen als auch Variablen beinhalten, die nicht definiert, sondern nur benutzt werden. Senken und innere Knoten beinhalten immer Instruktionen. Eine Kante speichert zwei von der Datenflussanalyse erstellte Informationen. Dabei handelt es sich um zwei LInt-Vektoren, deren Berechnung im folgenden Kapitel 3.4 näher erklärt wird.

Die einzelnen Gruppen der Kanten sind durchnummeriert. Eingehende Kanten bekommen positive Nummern, die den Index des Parameters der jeweiligen Instruktion darstellen. Ausgehende Kanten werden ebenfalls pro Parameter nummeriert, allerdings besitzen sie negative Nummerierungen (siehe Abbildung 3-5).

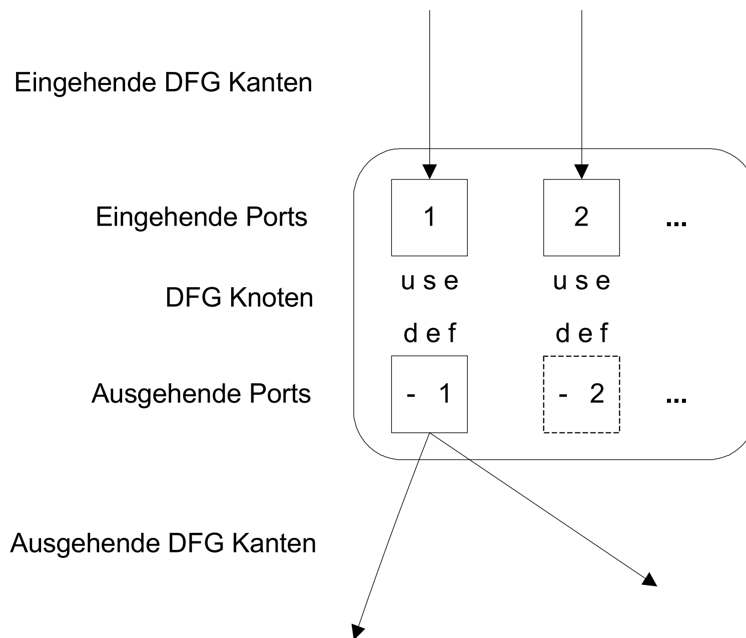


Abbildung 3-5: Port-Nummerierung der DFG-Knoten

Zu beachten ist hier, dass die LLIR in der Lage ist, mehrere Operationen pro Instruktion abzubilden. Der Unterschied zwischen Assemblerprogrammen dargestellt durch die LLIR und Quellcode dargestellt durch SUIF ist, dass Instruktionen in SUIF nur jeweils eine Operation pro Instruktion besitzen und somit nur jeweils einen ausgehenden Parameter besitzen, der die Ergebnisvariable einer Instruktion in C darstellt. Abbildung 3-6 zeigt ein einfaches Beispiel. Es wird eine einfache Instruktion dargestellt, in der zwei Variablen addiert werden sollen. Demnach bekommt die erste Kante, die die Variable „b“ darstellt, die Nummer 1 und die zweite Kante, die c darstellt, die Nummer 2 zugewiesen. Die resultierende Kante bekommt den Wert -1, da es eine ausgehende Kante ist.

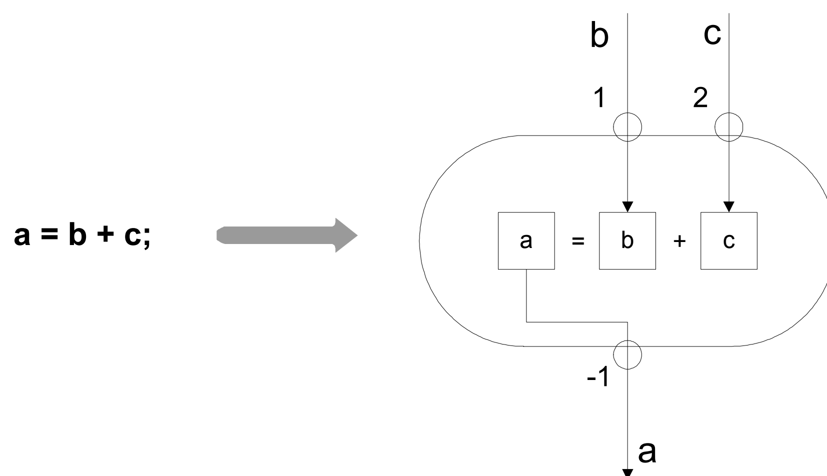


Abbildung 3-6: Nummerierung der Kanten

Zusätzlich zu den Informationen, welche Nummer eine jeweilige Kante trägt, und mit welchen Knoten sie verbunden ist, enthält sie einen LInt-Wert. Durch diese Klasse wird in einer folgenden Analyse möglichst exakt ermittelt, welche Datenpakete über die einzelnen Kanten laufen. Und genau in diesem Punkt liegt die Besonderheit dieser Datenflussanalyse. Durch eine bitgenaue Analyse nach dem Erstellen des Datenflussgraphen wird versucht, eine möglichst genaue Berechnung jedes einzelnen Bits vorzunehmen. Durch die Verwendung der LInt-Klasse ist es außerdem möglich, auf jedes beliebige Bit zuzugreifen, es abzufragen oder aber auch zu manipulieren.

3.4 Datenflussanalyse

Die Datenflussanalyse beinhaltet zwei Phasen, die „Top-Down-Analyse" und die entsprechende „Bottom-Up-Analyse". Im ersten Fall findet eine Analyse entsprechend der Laufrichtung der Kanten statt. Es wird bei den Quellen gestartet, und jede Instruktion eines jeden Knotens wird simuliert. Die Ergebnisse werden anhand zweier LInt-Vektoren zur zugehörigen ausgehenden Kante gespeichert. Weiterführend wird der Zielknoten jeder Kante betrachtet und übernimmt zur weiteren Berechnung die vom Vorgänger berechneten LInt-Werte. Somit breitet sich der Vorgang von den Quellen zu den Senken aus.

Das folgende Beispiel zeigt die Vorgehensweise der Analyse. Dafür betrachten wir folgenden C-Code.

```
unsigned char bsp (unsigned char c, unsigned char a)
{
        unsigned char d;
        d = ( c + a ) & 0x33;

        return (( d >> 4 ) + ( d << 2 ));
}
```

Der Algorithmus sucht zunächst nach den im Code vorkommenden Instruktionen und fügt sie in den Graphen ein. Im zweiten Schritt erfolgt die Zuweisung der Kanten gemäß Definitionen und Verwendungen der Daten im Quellcode. An-

schließend erfolgt die Top-Down-Analyse, indem die eingehenden Parameter zunächst mit u – unbekannt deklariert werden. Nun wandert die Verarbeitung von den Quellen zu den Senken und simuliert die Berechnung der einzelnen Bits anhand der jeweiligen Instruktion, wie das Beispiel in Abbildung 3-7 zeigt.

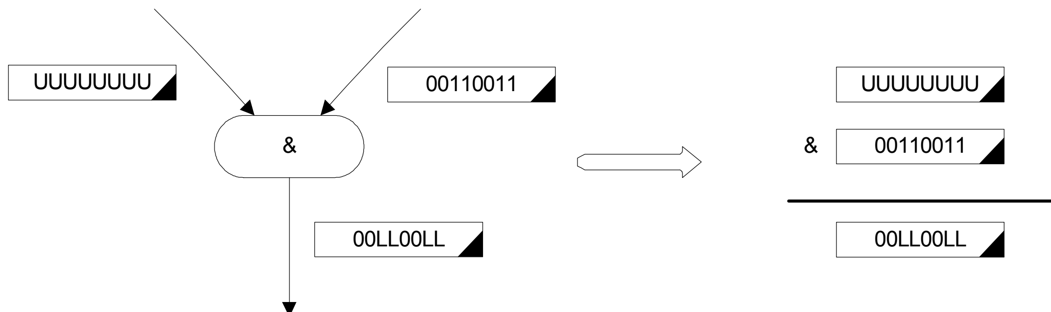


Abbildung 3-7: Berechnung eines bitweisen UND

Um den resultierenden LInt-Wert im obigen Beispiel zu berechnen, werden die einzelnen Bits der jeweiligen „Down-Werte“ mit & verknüpft. Dabei erhält man als Ergebnis eine „0“, falls wenigstens ein Bit eines LInt-Vektors „0“ ist, da es demnach keinen Unterschied macht, welchen Wert das zweite Bit annimmt oder annehmen könnte. Man erhält ein „L“, sobald ein Bit den Wert „unbekannt“ trägt, aber das zweite Bit eine „1“ darstellt. Anhand dieser Bit-Kombination kann man nicht sagen, welchen Wert das zu errechnende Bit tragen wird. Allerdings besagt das „L“, dass es bekannt ist, an welcher Stelle dieses Bit berechnet wird. Ein Ergebnis-Bit erhält den Wert „1“ nur dann, wenn beide zu berechnenden Bits eine „1“ tragen. Das Ergebnis wird in den „Down-Wert“ der ausgehenden Kante eingetragen, und die Analyse fährt mit dem Zielknoten dieser Kante fort. Die übertragenen LInt-Werte werden in die folgenden Berechnungen miteinbezogen, so dass eine komplette Simulation der Instruktionen von den Quellen bis zu den Senken erfolgt. Resultierend ergibt sich der Graph in Abbildung 3-8. Die eingehenden Parameter werden mit einem Bit-Vektor, der nur unbekannte Bits enthält, initialisiert.

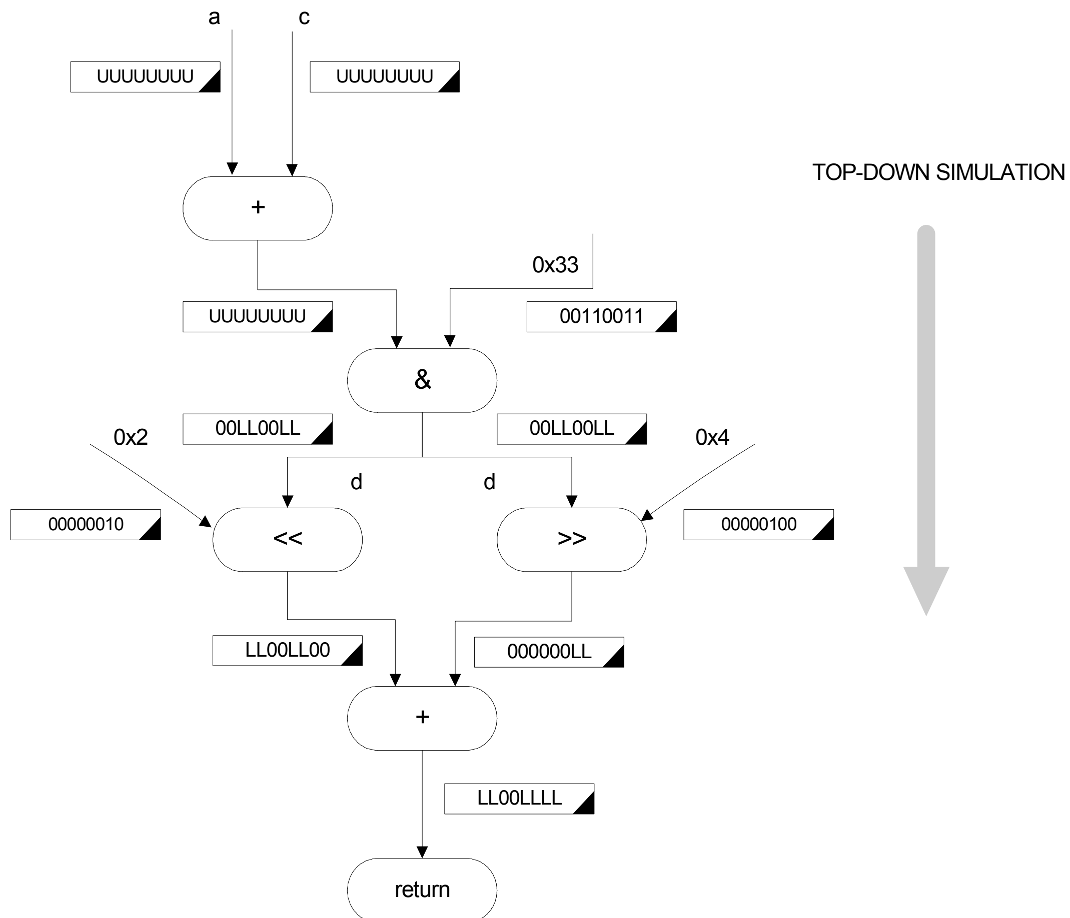


Abbildung 3-8: Top-Down Analyse

Die zweite Phase der Analyse läuft entgegengesetzt. Zunächst werden die bei der ersten Phase der Analyse berechneten Werte in den Kanten als „Up-Value" gespeichert, und man beginnt den Lauf bei den Senken. Die Analyse betrachtet demnach alle eingehenden Kanten und breitet sich so zu den Quellen aus. In dieser Phase werden allerdings keine neuen Werte berechnet. Es findet lediglich eine Überprüfung statt, welche Teile der Berechnung notwendig sind, um die gefundenen Ergebnisse berechnen zu können. Somit werden die berechneten Ergebnisse nur um die „Don´t-Care"-Werte ergänzt.
Abbildung 3-9 zeigt, wie hier vorgegangen wird. Durch eine Linksverschiebung der Bits um zwei Stellen werden die zwei hochwertigsten Bits aus dem Gültigkeitsbereich verschoben und sind somit nicht weiter relevant. Bei der umgekehrten Berechnung, werden diese beiden Bits als x – „Don´t-Care" eingefügt, da sie in der regulären Berechnung nicht benötigt werden.

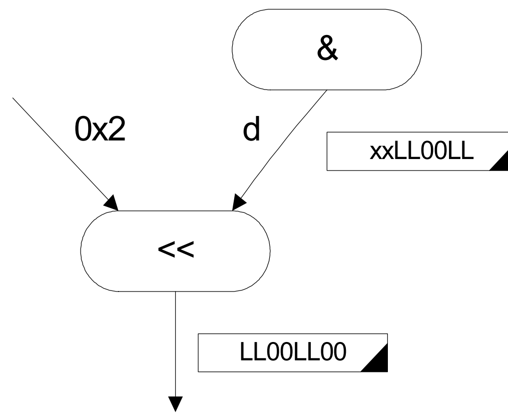


Abbildung 3-9: Bottom-Up Analyse einer Linksverschiebung

Anders verhält sich die Berechnung bei Benutzung eines Rechsshifts. Hier werden „Don´t Care"-Werte von rechts in den Vektor eingefügt. Abbildung 3-10 zeigt diese Handhabung.

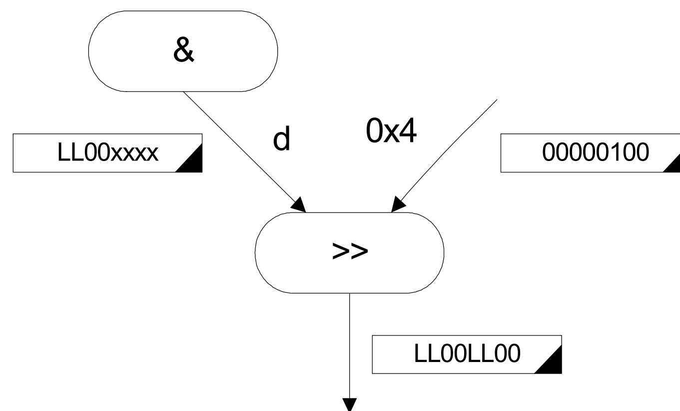


Abbildung 3-10: Bottom-Up Analyse einer Rechtsverschiebung

Resultierend ergibt sich der Graph in Abbildung 3-11. Zu beachten sind „Up-Werte" der eingehenden Parameter. Obwohl sich noch zwei „Don´t Care"-Werte an den Stellen 2 und 3 in den LInt-Vektoren befinden, werden diese durch die Addition zu unbekannten Bits transformiert, die oberen zwei „Don´t Care"-Werte allerdings nicht. Die Ursache liegt an der Position der Bits. Da die höherwertigen Bits keine Manipulation, beispielsweise durch einen Übertrag bei einer Addition, bewirken können, spielen diese Werte keine Rolle und können vernachlässigt werden. Die Bits an Stelle 2 und 3 müssen allerdings betrachtet werden, da nicht

zu ersehen ist, ob diese Stellen durch die darunter liegenden Werte, beispielsweise durch ein Carry-Bit, verändert werden könnten.

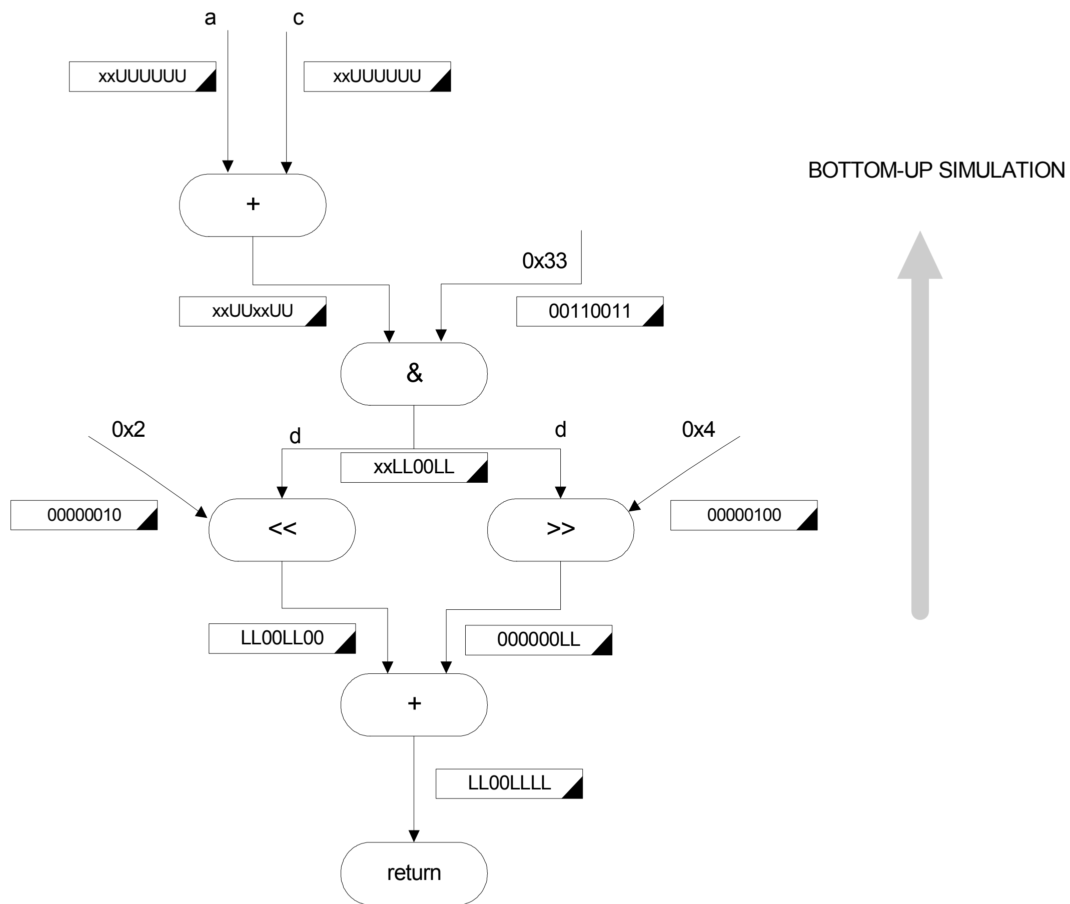


Abbildung 3-11: Bottom-Up Analyse

Im Falle, dass auf zwei verschiedenen Wegen unterschiedliche Up-Werte zu Stande kommen, muss der Wert ausgewählt werden, der für beide ursprüngliche Berechnungen die Bits beinhaltet, um beide Berechnungen möglich zu machen. Es kommt zu einer Berechnung eines „kleinsten Nenners“, wie sie in Abbildung 3-12 zu sehen ist.

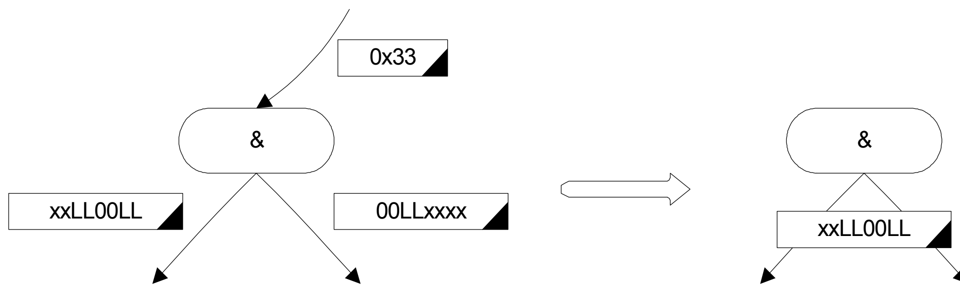


Abbildung 3-12: Berechnung des "kleinsten Nenners“

An dieser Stelle der Analyse muss entschieden werden, welche Bits notwendig sind, um die ursprüngliche Funktion beizubehalten. Somit ergibt sich der gezeigte LInt-Vektor, der in diesem Beispiel mit Hilfe des LInt-Wertes 0x33 ausgewählt wurde. Würde nicht diese Kombination von Bits ausgewählt werden, wäre die ursprüngliche Funktionalität nicht weiter gegeben. Durch den einfließenden Wert von 0x33 in diesen Knoten wird festgelegt, welche Bits notwendig bleiben. Würde man den anderen LInt-Wert auswählen, gäbe es ein Problem beim Bestimmen der unteren vier Bits. Durch die Angabe der „x“-Bits wäre es somit egal, welchen Wert diese Bits annehmen können. Damit hätte man allerdings ein Problem, denn die Analyse würde nicht erkennen, dass die unteren zwei Bits unbekannt bleiben müssen, da das Ergebnis durch den Wert 0x33 offen bleibt.

4 Portierung der Datenfluss-Analyse

Die erste Aufgabe dieser Diplomarbeit besteht in der Portierung und Anwendung der im vorherigen Kapitel vorgestellten bitgenauen Datenflussanalyse. Sie dient als grundlegendes Werkzeug für die in dieser Arbeit erstellten Optimierungen. Grundlegend wichtig ist die Anbindung des Compiler-Systems SUIF, um den erstellten Datenflussgraphen analysieren zu können. Kapitel 4.1 gibt einen Überblick über die von SUIF benutzten Instruktionen und zeigt, wie sie in die Analyse eingebunden worden sind. Kapitel 4.2 zeigt die Klasse „Opthelper", durch die die Datenflussanalyse gesteuert wird.

4.1 Anbindung von SUIF

Dieses Unterkapitel zeigt die von SUIF zur Verfügung gestellten Operationen, die zur besseren Übersicht in Gruppen zusammengefasst werden. Diese Informationen stammen aus der Dokumentation „The SUIF Library" [TSL94]. Zu jeder angegebenen Gruppe wird ein anschauliches Beispiel angegeben, um die Funktionalität der portierten Datenflussanalyse darzustellen. Aus Platzgründen wird darauf verzichtet, die vollständige Spezifikation für jede SUIF-Operation anzugeben. Selbstverständlich sind sämtliche SUIF-Operationen in die bitgenaue Datenflussanalyse vollständig integriert worden.

- **Zuweisung von Konstanten**
  Durch die SUIF-Instruktion „ldc – load constant" werden Zuweisungen von Konstanten unterstützt. Diese Instruktionen bilden immer Quellen in der Datenflussanalyse, und somit existiert keine Aufwärtsanalyse bei dieser Instruktion. Dennoch ist diese Operation ein wichtiger Bestandteil der Analyse, da durch sie Variablen initialisiert werden. Damit bilden sie neben den Knoten, die die Variablen der Parameterübergabe beinhalten, die Startpunkte der Top-Down-Analyse. Beim Aufbau des Datenflussgraphen bzw. beim Erstellen der Analyse werden die zugehörigen LInt-Vektoren durch den zu übertragenen konstanten Wert gebildet. Abbildung 4-1 zeigt einen Blattknoten, der eine „ldc"-Instruktion enthält und den erstellten LInt-Vektor an die ausgehenden Kanten verteilt.

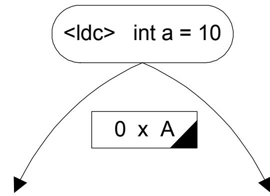


Abbildung 4-1: Zuweisung einer Konstanten

- **Konvertierung**

  Die „cvt“-Instruktion ist für die Konvertierung von Datentypen zuständig. Der Datentyp des übergebenen Operanden wird demnach in den gewünschten Datentypen konvertiert und in der Ergebnisvariablen gespeichert. Diese Operation typisiert Integer Datentypen mit verschiedenen Bitbreiten. Dabei sind ebenfalls Datentypen mit oder ohne Vorzeichen möglich. Nicht unterstützt werden Arrays, Structs und Unions. Der folgende Quellcode zeigt eine mögliche Struktur:

```
int a;
return (short)a;
```

  Die Datenflussanalyse erkennt in der zweiten Phase, dass die oberen 16 Bits der Variablen a nicht benötigt werden und kennzeichnet sie mit der Wertigkeit „x-Don´t Care“. Abbildung 4-2 zeigt den Graphen nach der zweiten Phase.

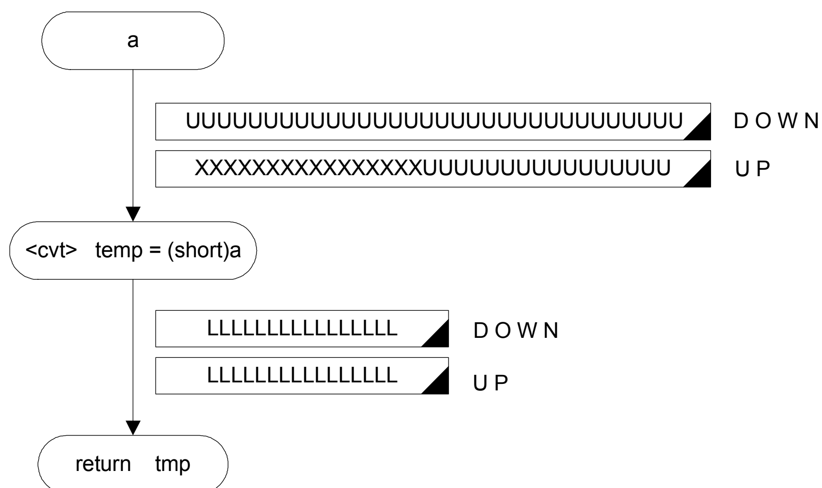


Abbildung 4-2: Top-Down und Bottom-Up einer Konvertierung

Die SUIF-Operation „cvt“ fügt eine temporäre Variable ein und legt das Ergebnis der konvertierten Variablen a darin ab. Anschließend wird dieser temporäre Wert durch die „return“-Instruktion zurückgegeben. Die LInt-Vektoren der Top-Down-Analyse beinhalten nur die Wertigkeit „U“, da die Variable a als Parameter mit unbekanntem Inhalt in die Prozedur übergeben wird. Bei der Aufwärtsanalyse erfolgt eine rücklaufende Berechnung. Zunächst werden die LInt-Vektoren der Abwärtsanalyse in die LInt-Vektoren der Aufwärtsanalyse kopiert. Die zweite Phase beginnt an der Senke und damit an der „return“-Instruktion. An dieser Stelle wird keine Änderung des LInt-Wertes der eingehenden Kante vorgenommen, und die Analyse wandert weiter zur „cvt“-Instruktion. An dieser Stelle wird nun der LInt-Vektor rückwärtig berechnet. Aus einer Variablen mit dem Datentypen short wird eine Variable mit dem Datentypen integer. Da es sich dabei aber um eine rückläufige Konvertierung von 16 Bit nach 32 Bit handelt, werden die führenden 16 Bits mit der Wertigkeit „x-Don´t Care“ versehen.

- **Carry Bit Operationen Addition, Subtraktion, Multiplikation**

  Diese Instruktionen gehören zu den „Three operand instructions“. Sie erhalten zwei Operanden, die durch den jeweiligen Operator berechnet werden. Das folgende Code-Fragment zeigt ein einfaches Beispiel einer Addition:

```
char a = 10;
char b = 20;
char c = a + b;
```

Zunächst erfolgen Zuweisungen von konstanten Werten in die Variablen a und b. Im folgenden Datenflussgraphen, der den Fluss dieser Codezeilen zeigt, bilden sie die Quellen. Anschließend erfolgt die Addition, und der resultierende Wert wird in der Ergebnisvariablen c gespeichert. Danach wird die Variable zur weiteren Berechnung freigegeben. Viele Instruktionen können berechnet werden, indem eine Tabelle zu Hilfe gezogen wird. In diesem Fall erfolgt die Ermittlung Bit für Bit. Abbildung 4-3 zeigt die Tabelle der Übertragungsfunktion für die Addition. Die beiden ersten Spalten zeigen die Wertigkeit der Bits der jeweiligen Variablen, die es zu addieren

gilt. Die mit „carry“ überschriebene Spalte gibt die möglichen Überträge „0“, “1“ und „u“ von der Addition der vorherigen Bit-Position an. An dieser Stelle macht es allerdings keinen Sinn, die Bit-Wertigkeiten „L“ und „N“ zu berücksichtigen, da es sich nur um ein temporäres Bit handelt, das für die Berechnung der nachfolgenden Bit-Position benötigt wird. Die in Tabelle 4-3 angegebenen Werte entsprechen somit der Addition a+b+carry. Das höherwertige Ergebnis-Bit dieser Addition stellt das Carry-Bit an die nächsthöhere Position dar. Zu beachten ist, dass das Carry-Bit niemals den Wert „x- Don´t Care“ annehmen kann.

a	b	carry		
		0	1	u
0	0	00	01	0L
0	1	01	10	UL
0	L	0L	UL	UL
0	N	0N	UL	UL
0	U	0L	UL	UL
1	0	01	10	1U
1	1	10	11	1U
1	L	UL	1L	UL
1	N	UL	1N	UL
1	U	UU	1U	UU

a	b	carry		
		0	1	u
L	0	0L	UL	UL
L	1	UL	1L	UL
L	L	UL	UL	UL
L	N	UL	UL	UL
L	U	UL	UL	UL
N	0	0L	UL	UL
N	1	UL	1L	UL
N	L	UL	UL	UL
N	N	UL	UL	UL
N	U	UL	UL	UL
U	0	0L	UL	UL
U	1	UL	1L	UL
U	L	UL	UL	UL
U	N	UL	UL	UL
U	U	UL	UL	UL

Abbildung 4-3: Vorwärts Übertragungsfunktionen der Addition

Die Addition wird nun anhand dieser Tabelle ausgeführt, und es ergibt sich der in Abbildung 4-4 resultierende Datenflussgraph inklusive der Top-Down-Analyse.

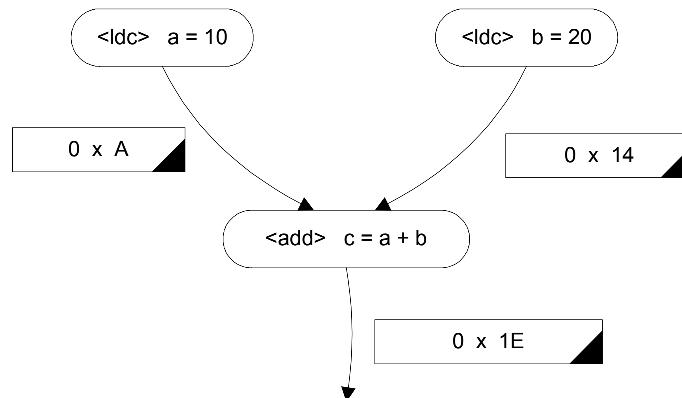


Abbildung 4-4: Top-Down-Analyse einer Addition

Bei der Aufwärtsanalyse von Carry Bit Operationen werden die Bit-Wertigkeiten „x-Don´t Care“ von den führenden zusammenhängenden Bits übernommen, die durch die ausgehende Kante zurückfließen. Abbildung 4-5 zeigt den Datenflussgraphen nach der Aufwärtsanalyse und die Übernahme der „x-Don´t-Care“-Bits. Die „x“-Wertigkeiten, die durch die ausgehende Kante in den Knoten der „add“-Instruktion zurückfließen, sind lediglich zur besseren Darstellung von aufwärts fließenden LInt-Vektoren der Addition angegeben. Vier führende „x“-Wertigkeiten sind beispielsweise durch eine rückwärtige Linksverschiebung um vier Stellen möglich.

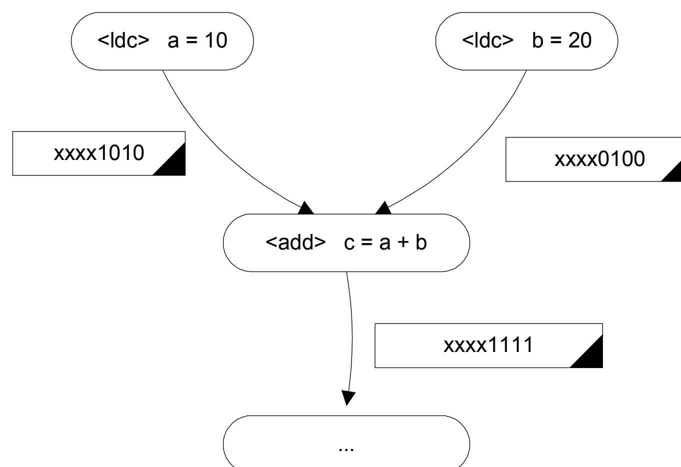


Abbildung 4-5: Aufwärtsanalyse einer Addition

- **Bitweise Verknüpfungen AND, OR und XOR**
  Diese Instruktionen sind für die bitweisen Berechnungen von Variablen zuständig. Die zwei verwendeten Operanden werden mit dem Operator verknüpft und bitweise bearbeitet. Der folgende Quellcode zeigt ein einfaches Beispiel einer „AND“-Verknüpfung:

```
short a;
short b = a & 0x33;
```

Die Deklaration der Variablen a mit dem Datentypen „short“ sowie die Zuweisung der Konstanten „0x33“ bilden die Quellen des Datenflussgraphen. Anschließend erfolgt die bitweise Verknüpfung mit der „AND“-Operation. Abbildung 4-6 zeigt die zugehörige Tabelle für die Vorwärtsfunktion.

a	b				
	0	1	U	L	N
0	0	0	0	0	0
1	0	1	L	L	L
U	0	L	L	L	L
L	0	L	L	L	L
N	0	L	L	L	L

Abbildung 4-6: Übertragungsfunktion einer „AND“-Verknüpfung

Anhand der Übertragungstabelle ist zu erkennen, dass das Ergebnis eines Bit-Vergleiches nur dann 1 ist, falls beide zu vergleichenden Bits die Wertigkeit 1 besitzen. Unbekannte Wertigkeiten führen dazu, dass das Ergebnis ebenfalls unbekannt bleibt. Falls mit einer 0 verglichen wird, ist das Ergebnisbit stets 0. Durch den Vergleich mit der binären Zahl „00110011“, die den Hexadezimalwert 0x33 darstellt, werden vier Stellen durch die Wertigkeit 0 gekennzeichnet. Diese Stellen sind daher in der Ergebnisvariablen ebenfalls 0. Die restlichen Bits bleiben unbekannt. Abbildung 4-7 zeigt den Datenflussgraph nach der Vorwärtsanalyse.

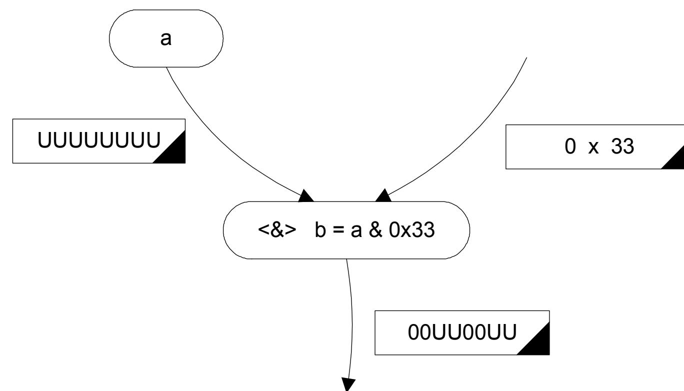


Abbildung 4-7: Vorwärtsanalyse einer "AND"-Verknüpfung

Die Aufwärtsanalyse einer Bit-Verknüpfung benutzt, um den zurückfließenden LInt-Wert der eingehenden Kanten zu berechnen, nicht nur den Wert, der durch die ausgehende Kante des Knotens zurückfließt, sondern auch die eingehenden Flüsse dieses Knotens. Abbildung 4-8 zeigt den resultierenden Datenflussgraphen nach der Aufwärtsanalyse.

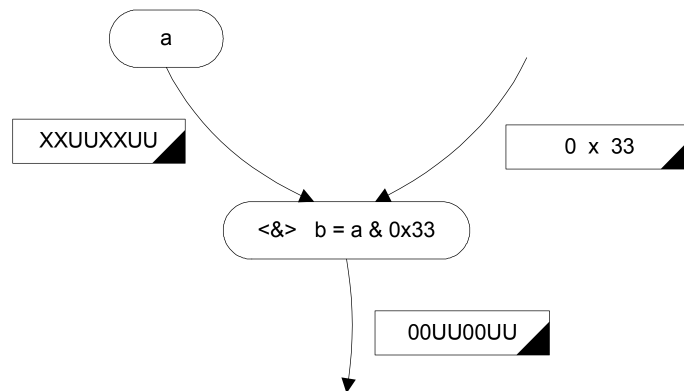


Abbildung 4-8: Aufwärtsanalyse einer „AND“-Verknüpfung

Durch das Wissen, dass die durch 0 gekennzeichneten Stellen des LInt-Vektors, die durch den zweiten Operanden 0x33 in den Knoten fließen, in der Ergebnisvariablen ebenfalls eine 0 forcieren, ist die Analyse in der Lage, diese Stellen des anderen Operanden durch die Bit-Wertigkeit „x-Don´t Care“ zu versehen, da es keine Rolle spielt, welchen Wert diese Bits in der Berechnung annehmen.

- **Shift-Operationen**

  SUIF stellt drei verschiedene Arten von Shift-Operatoren zur Verfügung. Zum einem die „asr“-Instruktion, die ein Shiften nach rechts unterstützt. Die Besonderheit bei dieser Operation ist das Beachten des Vorzeichens der zu schiebenden Variablen. Die Instruktionen „lsl“ und „lsr“ lassen den Inhalt einer Variablen nach links und rechts shiften, beachten dabei jedoch das Vorzeichen nicht. Alle drei Operationen benötigen eine Angabe der Anzahl von Shifts mit dem Datentypen „unsigned int“. Die erste angegebene Variable wird um diese Anzahl Shifts verschoben und in der Ergebnisvariablen gespeichert. Die nächsten Zeilen zeigen ein Beispiel für einen Linksshift:

  short a = 10;
  short b = a << 3;

  Zunächst wird wieder eine Konstante geladen, die die Quelle im folgenden Datenflussgraphen bildet. Anschließend erfolgt die Linksverschiebung der Variablen a um drei Stellen. Das Ergebnis wird in der Variablen b gespeichert. Abbildung 4-9 zeigt den resultierenden Top-Down-Durchlauf der obigen Codestruktur.

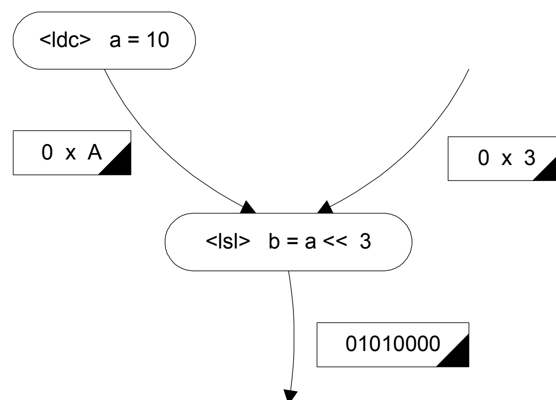


Abbildung 4-9: Linksverschiebung um drei Stellen

Beim Shiften um eine konstante Länge werden lediglich Bits mit der Wertigkeit „0“ an den LInt-Vektor angehängt. Bei der Verwendung einer Variablen, die eine unbekannte Anzahl von Shifts enthält, wird der komplette LInt-Vektor mit den Bit-Wertigkeiten „U“ gefüllt. Die Aufwärtsanalyse lässt die zu verschiebende Variable in die gegenseitige Richtung laufen. Dabei werden aller-

dings Bits mit der Wertigkeit „X“ verknüpft. Abbildung 4-10 zeigt den Datenflussgraphen der Aufwärtsanalyse.

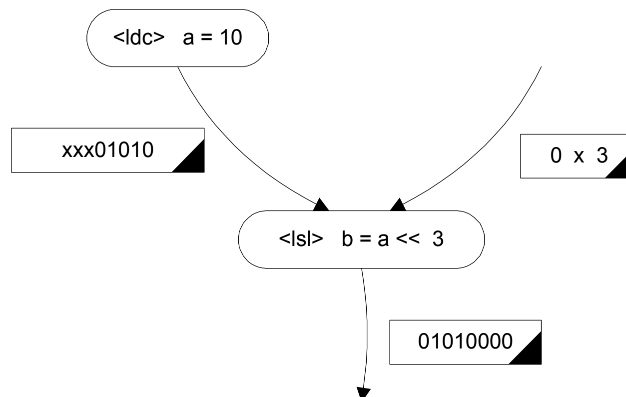


Abbildung 4-10: Aufwärtsanalyse einer Linksverschiebung

- **Negation und NOT**

  Die SUIF-Instruktion „neg“ ändert das Vorzeichen eines Wertes in einer Variablen. Dagegen ist die „NOT“-Operation eine bitweise Instruktion, die das Komplement jedes einzelnen Bits bildet. Beide Instruktionen benutzen nur einen Operanden, der nach der Berechnung in die Ergebnisvariable kopiert wird. Abbildung 4-11 zeigt die Funktionstabelle der „NOT“-Instruktion. Potentiell überflüssige Bits ändern ihren Wert nicht und bleiben mit „x“ gekennzeichnet. Die konstanten Werte 1 und 0 bilden das Komplement der Konstanten. Die Wertigkeiten „L“ und „U“ werden zur dafür vorgesehenen Wertigkeit „N“.

a	X	1	0	L	N	U
not a	X	0	1	N	L	N

Abbildung 4-11: Übertragungsfunktion der „NOT“-Instruktion

Bei der Aufwärtsanalyse der beiden Operationen werden die potentiell überflüssigen Bits, die mit „X“ gekennzeichnet sind, auf den Wert der eingehenden Kante des Knotens übertragen.

- **Vergleichsoperationen**

  Durch das SUIF-System werden vier verschiedene Vergleichsoperatoren zur Verfügung gestellt. Dabei handelt es sich zum einen um die Prüfung auf Gleichheit und Ungleichheit zweier Variablen. Zum anderen stehen zwei Operatoren zur Verfügung, die testen, ob eine Variable einen kleineren oder größeren Inhalt als eine zu vergleichende Variable besitzt. Bei einer erfolgreichen Überprüfung erhält die Ergebnisvariable den Wert des ersten Operanden, ansonsten den Wert 0. Die Datenflussanalyse testet die Bits von der höchstwertigsten Position abwärts bis zur Position 0. Bei der Überprüfung auf Gleichheit wird jedes einzelne Bit getestet und die Anzahl der ungleichen und unbekannten Bits ermittelt. Abbildung 4-12 zeigt, bei welchen Bit-Kombinationen ein ungleiches Bit gezählt wird.

a	b					
	0	1	L	N	U	X
0	0	1	1	1	1	1
1	1	0	1	1	1	1
L	1	1	1	1	1	1
N	1	1	1	1	1	1
U	1	1	1	1	1	1
X	1	1	1	1	1	0

Abbildung 4-12: Funktionstabelle für die Vergleichsoperation „==“

Analog dazu funktioniert der Test auf Ungleichheit. Falls dabei ein Bit mit einer unbekannten Wertigkeit entdeckt wird, ist das Ergebnis ebenfalls unbekannt. Bei der Überprüfung auf unterschiedliche Größen der Inhalte führt die Analyse den Vorgang so lange aus, bis zwei unterschiedliche Bit-Wertigkeiten gefunden werden. Auch in diesem Fall wird der erste Operand in die Ergebnisvariable kopiert, falls eine erfolgreiche Überprüfung stattfindet. Bei einer erfolglosen Prüfung erhält das Resultat die Konstante 0. Falls eines der Bits einen unbekannten Wert enthält, erhält die Ergebnisvariable die Wertigkeit „U“ für unbekannt. SUIF übersetzt einen großen Teil von verwendeten For-Schleifen in Do-While-Schleifenkonstrukte, bei denen dann die Schleifenvariable analysiert werden kann. Das folgende Code-Fragment zeigt ein mögliches Konstrukt:

```
int i = 0;
do {
   …
   i++;
} while ( i < 10 );
```

Die Aufwärtsanalyse wird nur durch die Operatoren „kleiner“ und „kleiner gleich“ umgesetzt. Der zugehörige Datenflussgraph nach der Aufwärts-analyse ist in Abbildung 4-13 zu sehen.

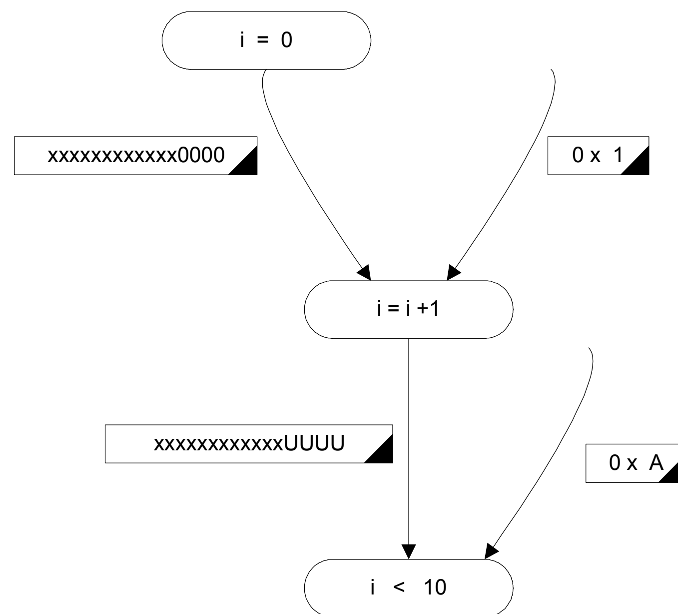


Abbildung 4-13: Aufwärtsanalyse der Vergleichsoperation "kleiner"

Bei dem Vergleich mit einer Konstanten erkennt die Analyse, welcher Wertebereich für die zu vergleichende Variable notwendig ist. Die so erkannten unbenutzen Bits werden durch die Bit-Wertigkeit „x-Don´t Care“ ausgefüllt. Durch dieses Verfahren kann unter anderem die Anzahl von Schleifendurchläufen analysiert werden. Sobald erkannt wird, dass eine Zählvariable einen konstanten Wert nicht überschreitet, wird der rückläufige LInt-Vektor angepasst.

4.2 Opthelper-Klasse

Um die Simulation des Quellcodes durchführen zu können, kommt ein weiterer Bestandteil hinzu, die Opthelper-Klasse. Mit dieser Klasse werden Methoden zur Verfügung gestellt, um prozessorspezifische Merkmale abfragen und die einzelnen Instruktionen abhängig von der benutzen Zwischendarstellung und dem verwendeten Prozessor simulieren zu können. Dabei wird die Simulation in zwei Funktionen getrennt. Die eine absolviert die natürliche Abfolge der Instruktionen, wobei die andere Funktion die Aufwärtsbewegung der Analyse simuliert. Demnach ist es modular möglich, die Simulationen zu steuern.

```
void Sim ( const instruction *instr, map<int, LInt>& Operators,
                                    map<int, LInt>& Results );
void Simup ( const instruction *instr, map<int, LInt>&Operators,
                                    map<LInt>& ChangedOperators ) ;
```

Diese beiden Funktionen bilden das Kernstück der Opthelper-Klasse. Sie beinhalten die Funktionalität, um den Datenflussgraphen analysieren zu können. Mit Hilfe der im vorherigen Abschnitt präsentierten Übertragungsfunktionen, die das bitgenaue Verhalten der jeweiligen SUIF-Befehle simulieren, werden die LInt-Vektoren erstellt. Die erste Funktion „Sim“ führt die erste Phase der Analyse durch. Durch sie werden demnach die Instruktionen in der „Top-Down“-Analyse bearbeitet. Diese Funktion enthält drei Parameterübergaben. Zum einen handelt es sich um die zu untersuchende Instruktion. Die beiden anderen Übergaben sind Referenzen von „maps“, die LInt-Vektoren beinhalten. Die LInt-Werte sind nach der Port-Nummerierung, wie sie in Kapitel 3.3 dargestellt wurde, angeordnet. Sie spiegeln die Reihenfolge der Operanden in der zu simulierenden Instruktion wider. Das Resultat wird mit einer „-1“ gekennzeichnet und stellt die Ergebnisvariable der Instruktion dar.

Die zweite Funktion übernimmt die Simulation der zweiten Phase. In der „Bottom-Up“-Analyse werden die zu untersuchenden Instruktionen an dieser Stelle simuliert. Auch bei dieser Funktion existieren drei Parameter, die übergeben werden. Es handelt sich dabei wiederum um die zu simulierende Instruktion sowie um „maps“ von LInt-Werten. Der Parameter Operators beinhaltet die bisher errechneten LInt-Werte, die während der vorherigen Stufe erstellt worden sind. Falls an dieser Stelle noch keine Berechnung erfolgte, ist der Aufwärtswert gleich dem

Wert, der zuvor bei der „Top-Down“-Analyse bestimmt worden ist. Im dritten Parameter werden die Ergebnisse der Aufwärtsanalyse der jeweiligen Instruktion gespeichert. Zu beachten ist, dass dieser Wert nur um die Angabe von „x – Don´t Care“ ergänzt wird. Jegliche andere Änderung an diesen LInt-Vektoren ist unzulässig.

5 Optimierungen

Dieses Kapitel beschreibt die in dieser Diplomarbeit bearbeiteten Optimierungen. Zunächst findet in Kapitel 5.1 die Behandlung von saturierender Arithmetik statt. Es wird detailliert gezeigt, welche verschiedenen Code-Strukturen eine Saturierung enthalten, und mit welcher Weise man diese erkennen und durch Intrinsics optimieren kann. Im folgenden Kapitel 5.2 wird die durchgeführte Bitreduktion dargestellt. Auch hier wird dargestellt, wie man mögliche Optimierungen erkennen und ausnutzen kann. In Kapitel 5.3 kommen schließlich Multimedia-Befehle zum Einsatz, um eine parallele Berechnung von Instruktionen auszunutzen.

5.1 Saturierende Arithmetik

Wie bereits in Kapitel 2.1.2.3 erwähnt, spielt die saturierende Arithmetik in dieser Arbeit eine wichtige Rolle. Die erste Optimierung, die hier vorgestellt werden soll, ist das Auffinden und Ersetzen solcher Code-Strukturen, um eine Performance-Steigerung zu erzielen. Dazu stellt dieses Kapitel zunächst die möglichen Arten von Saturierung vor, um anschließend eine Analyse folgen zu lassen. Dadurch wird gezeigt, wie die Technik der saturierenden Arithmetik funktioniert, und in welcher Art und Weise ein Erkennen dieser komplexen Strukturen möglich ist. Zu diesem Zweck werden Teile des G.723.1- und GSM-Codes gezeigt und durch dieses Kapitel analysiert.

5.1.1 Saturierung nach einer Addition

Wie funktioniert nun saturierende Arithmetik? Wie bereits in den Kapiteln 2.1.2.2 und 2.1.2.3 beschrieben, ist die Saturierung eine Methode, um einen möglichen Überlauf zu prüfen und zu verhindern. Wir betrachten zunächst eine Saturierung nach einer Addition, die sich von der Saturierung nach einer Subtraktion in zwei Dingen unterscheidet.

```
        L_var_out = L_var1 + L_var2;
1)      If (((L_var1 ^ L_var2) & 0x80000000) == 0L)
2)          If ((L_var_out ^ L_var1) & 0x80000000)
3)              L_var_out = (L_var1 < 0L)? 0x80000000: 0x7FFFFFFF;
```

Betrachten wir zunächst Zeile 1) nach der eigentlichen Additions-Instruktion. An dieser Stelle werden die beiden Vorzeichen der Operanden, die zur eigentlichen Addition gehören, geprüft. Wie bereits in Kapitel 2.1.2.2 erläutert, kann es nur zu einem Überlauf kommen, falls beide verwendeten Operanden je nach Instruktion das passende Vorzeichen besitzen. In diesem Fall der Addition müssen beide Operanden das gleiche Vorzeichen besitzen, um einen Überlauf erst möglich zu machen. Diese Überprüfung wird mit Hilfe von Vergleichen auf der Bit-Ebene gelöst, wie die folgenden Beispiele in Abbildung 5-1 und Abbildung 5-2 erläutern. Da der Ausschnitt des Quellcodes mit 32 Bit-Zahlen arbeitet, müssen passend große Zahlen gewählt werden, um einen Überlauf zu demonstrieren.

```
int L_var1 = 2000000000;
int L_var2 = -1000000000;

                                  01110111001101011001010000000000
L_var1 ^ L_var2     ==>     XOR   11000100011001010011011000000000
                                  --------------------------------
                            ➡     10110011010100001010001000000000

                                  10110011010100001010001000000000
& 0x80000000        ==>      &    10000000000000000000000000000000
                                  --------------------------------
                            ➡     10000000000000000000000000000000
```

Abbildung 5-1: Zeile 1) erkennt ungleiche Vorzeichen

Wie in Abbildung 5-1 zu sehen ist, wird bei dieser Kombination von Wertzuweisungen festgestellt, dass das Ergebnis von

(L_var1 ^ L_var2) & 0x80000000

ungleich „0“ ist, falls die zu überprüfenden Operanden unterschiedliche Vorzeichen besitzen. Somit bleibt festzuhalten, dass beide Vorzeichen unterschiedlich sind. Das hat zur Folge, dass kein Überlauf zu Stande kommen kann und die Überprüfung abgebrochen wird.

Falls andererseits diese Überprüfung von Zeile 1) erfolgreich stattfindet, sprich das Ergebnis wird erfolgreich mit „0“ verglichen, wie in Abbildung 5-2 zu erkennen ist, muss nun getestet werden, ob das Ergebnis der zu überprüfenden Instruktion ein anderes Vorzeichen als die verwendeten Operanden besitzt.

```
int L_var1 = 2000000000;
int L_var2 = 1000000000;
                                       01110111001101011001010000000000
L_var1 ^ L_var2    ==>    XOR          00111011100110101100101000000000
                                       --------------------------------
                          ->           01001100101011110101111000000000

                                       01001100101011110101111000000000
& 0x80000000       ==>    &            10000000000000000000000000000000
                                       --------------------------------
                          ->           00000000000000000000000000000000
```

Abbildung 5-2: Zeile 1) erkennt gleiche Vorzeichen

Diese Prüfung findet analog zur vorherigen statt mit dem Unterschied, dass nicht darauf geprüft wird, ob gleiche, sondern unterschiedliche Vorzeichen bei den Operanden und dem Ergebnis vorkommen. Daher ändert sich das Statement in Zeile 2) lediglich dahin gehend, dass nicht auf Gleichheit mit der Konstanten 0, sondern auf Ungleichheit getestet wird. Falls auch Zeile 2) erfolgreich durchgeführt wird, wurde ein Überlauf gefunden.

Letztendlich muss entschieden werden, welcher Wert der Variablen des Ergebnisses zugewiesen wird. Diese Entscheidung wird in Zeile 3) getroffen, indem ein Test durchgeführt wird, der prüft, ob die Operanden der Instruktion positiv oder negativ sind. An dieser Stelle reicht es natürlich, einen beliebigen Operanden (hier: den ersten) zu testen, da bereits festgestellt wurde, dass beide das gleiche Vorzeichen besitzen. Dementsprechend bekommt die Variable, die das Ergebnis der ursprünglich zu überprüfenden Instruktion repräsentiert, den Wert der kleinsten negativen oder der größten positiven darstellbaren Zahl. Abbildung 5-3 zeigt den resultierenden Datenflussgraphen der Addition und anschließenden Saturierung in einer vereinfachten Form. Bei der Erzeugung der Zwischendarstellung

eines Quellcodes typisiert das SUIF Compiler System die zu berechnenden Variablen in verschiedene Datentypen mit wechselnden Vorzeichen. Vorzugsweise wird allerdings der Datentyp „unsigned integer" für eine Vielzahl der SUIF-Operationen verwendet. Nach den eigentlichen Berechnungen konvertiert SUIF die Variablen in ihre ursprünglichen Formen zurück. Durch die Implementierung von SUIF kann es daher zu großen unübersichtlichen Graphen kommen, da auch mehrere Konvertierungen hintereinander auftauchen können. Um die hier gezeigten Graphen übersichtlich zu halten, werden Teile dieser Typisierung weggelassen. Deutlich zu erkennen bleibt allerdings der benötigte Fluss, um eine Saturierung der Ergebnisvariablen der zu prüfenden Addition zu erkennen.

Abbildung 5-3: Datenflussgraph einer Addition und Saturierung

Zunächst sucht der für diese Arbeit entworfene Algorithmus nach einer Addition mit zwei 32 Bit-Zahlen. Anschließend muss geprüft werden, ob nach dieser Addition ein Test auf die Gleichheit der Vorzeichen der verwendeten Operanden erfolgt. Im Anschluss erfolgt eine Suche nach der Überprüfung auf die Unterschied-

lichkeit der Vorzeichen eines Operanden und dem resultierenden Wert der Addition. Zu guter letzt muss verifiziert werden, ob eine Saturierung erfolgt. Im Datenflussgraphen ist zu erkennen, dass die Ergebnisvariable mit den Maximalwerten definiert und zur weiteren Verarbeitung benutzt werden kann. An dieser Stelle ist die „return“-Instruktion lediglich ein Beispiel für den weiteren Verlauf. Es können natürlich auch sämtliche anderen Instruktion folgen. Findet der Algorithmus alle notwendigen Schritte, erfolgt die Optimierung. Die gesamte Saturierungsüberprüfung und die zu prüfende Instruktion werden durch den Aufruf eines Intrinsics ersetzt. An dieser Stelle wird folgender Aufruf eingefügt:

"L_var_out = _sadd (L_var1 , L_var2);“

5.1.2 Saturierung nach einer Subtraktion

Wie bereits im vorherigen Kapitel erwähnt, gibt es zwischen der Saturierung nach einer Addition und der Saturierung nach einer Subtraktion zwei Unterschiede. Zu beachten ist, wie in Kapitel 2.1.2.2. erwähnt, dass die Vorzeichen der Operanden einer Subtraktion unterschiedlich sein müssen, damit ein Überlauf zu Stande kommen kann. Anders als bei der Addition wird somit bei der ersten Überprüfung in Zeile 1) nicht darauf geprüft, ob das Ergebnis einer 0 entspricht, sondern ob es ungleich 0 ist.

```
        L_var_out = L_var1 - L_var2;
1)      If (((L_var1 ^ L_var2) & 0x80000000 != 0L) {
2)          If ((L_var_out ^ L_var1) & 0x80000000) {
3)              L_var_out = (L_var1 < 0L)? 0x80000000: 0x7FFFFFFF;
```

Wiederum wird nun, analog zur Saturierung nach einer Addition, in Zeile 2) geprüft, ob das Ergebnis ein anderes Vorzeichen als der erste Operand besitzt. Darin besteht der zweite Unterschied gegenüber der Saturierung in Kapitel 5.1.1, denn an dieser Stelle ist es nicht möglich, einen beliebigen Operanden zu wählen. Genau wie in Zeile 3) muss bei der Saturierung nach einer Subtraktion der erste Operand für die Vergleiche benutzt werden. Ansonsten würde das Ergebnis verfälscht werden, da das Ergebnis genau das entgegengesetzte Vorzeichen erhalten würde. Der Datenflussgraph der Subtraktion mit anschließender Saturie-

rung ist analog zum vorherigen Datenflussgraphen mit dem Unterschied, dass der Vergleich der beiden Vorzeichen der Operanden auf ungleich 0 geprüft wird. Daher wird auf die Darstellung an dieser Stelle verzichtet. Die Vorgehensweise, um eine Saturierung nach einer Subtraktion zu finden, läuft analog zu der in Kapitel 5.1.1 vorgestellten. Allerdings ist nicht zu vergessen, dass auf die beiden beschriebenen Unterschiede geachtet wird. Wird eine Saturierung erkannt, erfolgt der Ersatz der gesamten Code-Struktur durch:

“L_var_out = _ssub (L_var1 , L_var2);”

5.1.3 Saturierung nach einer Multiplikation zuzüglich Left-Shift

Die Saturierung nach einer Multiplikation zuzüglich eines Left-Shifts zeigt eine gänzlich andere Code-Struktur. Das Ergebnis einer Multiplikation zweier 16 Bit-Zahlen resultiert in einem 32-Bit Wert. Hier ist es aber nicht von Belang, ob die Vorzeichen der zu multiplizierenden Operanden gleich sind. Ein Überlauf kommt bei einer Multiplikation mit einem darauf folgenden Shift nach links nur bei einer sehr speziellen Kombination der Operanden zu Stande. Und zwar nur dann, wenn die beiden kleinsten negativen Zahlen, sprich (-32768) miteinander multipliziert werden. Das Ergebnis einer solchen Multiplikation resultiert in der größten mit 32 Bit darstellbaren Zahl, die um 1 nach links verschoben werden kann, ohne die Anzahl der Bits zu überschreiten, wie es Abbildung 5-4 zeigt.

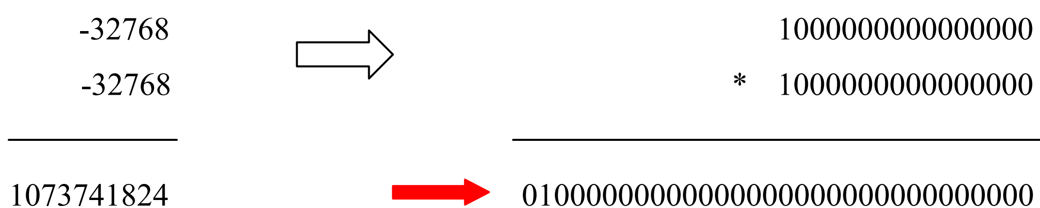


Abbildung 5-4: Multiplikation der zwei kleinsten 16 Bit-Zahlen

Bei der nun aber anstehenden Multiplikation mit 2 bzw. dem Verschieben des Ergebnisses um ein Bit nach links, das in Abbildung 5-5 dargestellt wird, ändert sich das höchstwertigste Bit von 0 auf 1. Damit hätte man aus einer Multiplikation von zwei negativen Zahlen ein negatives Ergebnis und erhält damit einen Überlauf.

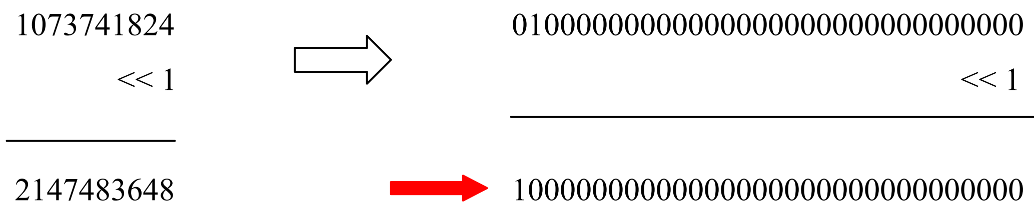


Abbildung 5-5: Überlauf beim Linksverschieben um eine Stelle

Das Ergebnis ist demnach nicht mit 32 Bit korrekt darstellbar, und somit wird es auf 2147483647 korrigiert.

Die Code-Strukturen, die diesen Fall bearbeiten, sehen wie folgt aus:

```
        L_var_out = var1 * var2;
1)      If (L_var_out != 0x40000000) {
2)              L_var_out *= 2;
        } else {
3)              L_var_out = 0x7FFFFFFF;
        }
```

In Zeile 1) wird zunächst darauf geprüft, ob das Ergebnis der Multiplikation den größtmöglichen Wert annimmt, denn Hexadezimal 0x40000000 entspricht dezimal einem Wert von 1073741824. Falls das nicht der Fall ist, wird das Verschieben um eine Stelle nach links in Zeile 2) durchgeführt, da kein Überlauf passieren kann. Andernfalls erhält die Ergebnisvariable der ursprünglichen Instruktion den maximalen positiven Wert von 2147483647, der mit 32 Bit darstellbar ist. Abbildung 5-6 zeigt den resultierenden Datenflussgraphen einer Multiplikation mit anschließendem Linksshift und Saturierung.

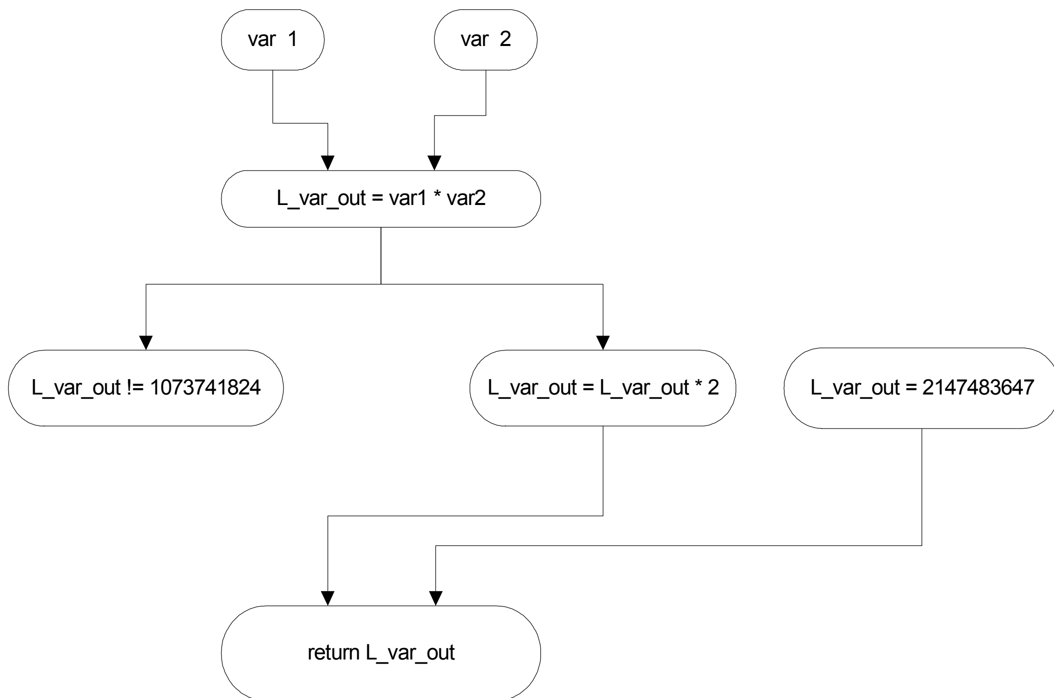


Abbildung 5-6: Datenflussgraph der Multiplikation mit Shift und Saturierung

Um diese Art der Saturierungsüberprüfung finden und optimieren zu können, geht man wie folgt vor. Zunächst sucht der Algorithmus nach einer passenden Multiplikation mit zwei 16 Bit-Operanden. Anschließend erfolgt eine Kontrolle, ob das Ergebnis mit der größtmöglichen positiven Zahl, die nach einer Multiplikation erfolgen kann, verglichen wird. Diese Zahl entspricht, wie oben beschrieben, dem Ergebnis der Multiplikation von -32768 mit sich selbst. Falls diese Art der Überprüfung gefunden wird, erfolgt anschließend die Suche nach einer Multiplikation mit der Konstanten 2 bzw. einer Linksverschiebung der Ergebnisvariablen um eine Stelle. Trifft auch dieses Ereignis ein, muss die resultierende Saturierung erfolgen, die ebenfalls im Datenflussgraphen dargestellt wird. Demnach erhält die Ergebnisvariable den maximalen positiven Wert 0x7FFFFFFF. Wird eine solche Code-Struktur erkannt, wird sie komplett durch den Aufruf des entsprechenden Intrinsics ersetzt, in diesem Fall durch:

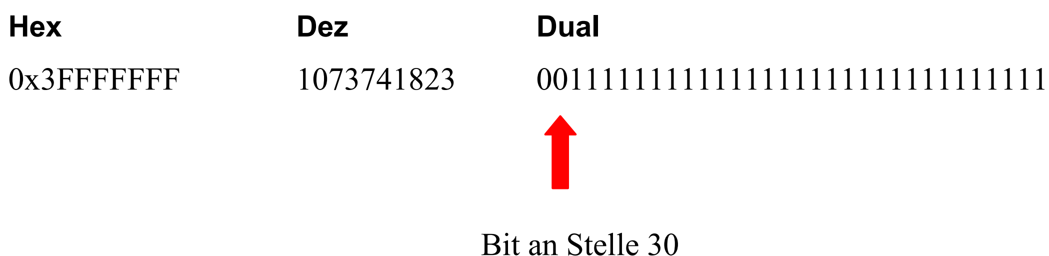
”L_var_out = _smpy (var1 , var2);”

5.1.4 Saturierung nach einer Linksverschiebung

Die Überprüfung auf einen Überlauf nach einer Linksverschiebung ist wohl der komplizierteste Fall der saturierenden Arithmetik, der in dieser Arbeit behandelt wird. Die Code-Struktur einer solchen Saturierung ist im folgenden dargestellt:

```
        int L_var1;  short var2;
        …
1)      for (; var2 > 0; var2--) {
2)         if (L_var1 > 0x3FFFFFFF) {
3)               L_var_out = 0x7FFFFFFF; break;
           }
           else
4)              if (L_var1 < 0xc0000000) {
5)                    L_var_out = 0x80000000; break;
           }
6)         L_var1 *= 2;
        }
```

Bei der Saturierung einer Linksverschiebung wird nach jedem einzelnen Shift überprüft, ob bereits ein Überlauf stattgefunden hat. Daher wird an dieser Stelle eine Schleife verwendet, die maximal die vorgesehene Anzahl von Shifts ausführt. In Zeile 2) wird nun überprüft, ob eine positive Zahl vorliegt, die bearbeitet werden soll. Ist das der Fall, darf der Wert dieser Variablen nicht in einen negativen Bereich eintreten. Mit der Überprüfung, ob der Wert der Variablen die Grenze von 0x3FFFFFFF bereits überschritten hat, wird dies gewährleistet. Abbildung 5-7 stellt die verschiedenen Darstellungsarten dieses Wertes dar.



Hex	Dez	Dual
0x3FFFFFFF	1073741823	00111111111111111111111111111111

Abbildung 5-7: Darstellungen der Hexadezimalzahl 0x3FFFFFFF

Falls der nun zu testende Wert größer wäre als 0x3FFFFFFF, würde das Bit an der Stelle 30 bereits eine 1 beinhalten. Käme es nun zu einer weiteren Linksverschiebung dieses Wertes, würde sich das höchstwertigste Bit von 0 nach 1 ändern. Damit hätte man einen Überlauf, der an dieser Stelle vermieden wird. Falls das der Fall ist, wird der Ergebnisvariablen der maximale positive Wert zugewiesen. Daraufhin bricht die Schleife ab und die Saturierung ist zu Ende.
Der andere Fall setzt voraus, dass der zu schiebende Wert negativ ist. Daher wird in Zeile 4) geprüft, ob der zu bearbeitende Wert bereits die Grenze von 0xC0000000 unterschritten hat. Abbildung 5-8 zeigt widerrum die unterschiedlichen Darstellungsarten dieses Wertes.

Hex	Dez	Dual
0xC0000000	-1073741824	11000000000000000000000000000000

Bit an Stelle 30

Abbildung 5-8: Darstellungen der Hexadezimalzahl 0xC0000000

Zu betrachten ist wieder das Bit an der Stelle 30. Zeile 4) überprüft an dieser Stelle, ob die Variable bereits den Wert 0xC0000000 unterschritten hat. Wäre das der Fall, würde an der Stelle 30 das Bit den Wert 0 besitzen. Käme es nun zu einer weiteren Linksverschiebung des Wertes, würde das höchstwertigste Bit den Wert 0 annehmen, da keine 1 mehr an der vorherigen Stelle steht, um das höchstwertigste Bit zu ersetzen. Damit würde dieses Bit von 1 auf 0 wechseln und sich somit das Vorzeichen von negativ auf positiv ändern. Ist es der Fall, dass der Wert der Variablen L_var1 die Grenze von 0xC0000000 unterschritten hat, wird diese in Zeile 5) auf die größte negative Zahl von 0x80000000 gesetzt. Im Anschluss bricht die Schleife ab, und die Saturierung ist beendet.
Falls beide Überprüfungen allerdings nicht erfolgreich sind und damit gilt

$$\mathbf{-1073741824 > L_var1 > 1073741823}$$

wird die Variable um eine Stelle nach links geschoben. Falls die maximale Anzahl der Linksverschiebungen noch nicht erreicht ist, erfolgt nun ein weiterer Schleifendurchlauf.

Abbildung 5-9 zeigt den resultierenden Datenflussgraphen in vereinfachter Form ohne Darstellung der For-Schleife.

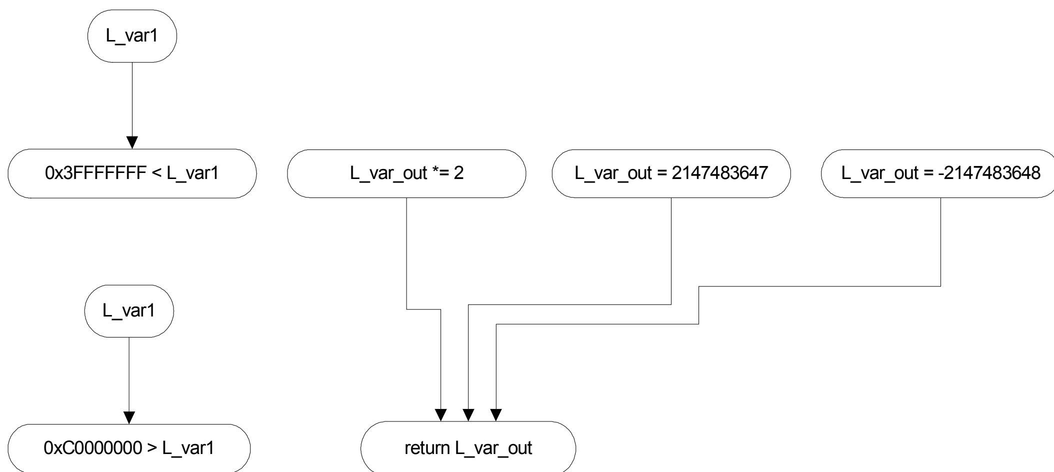


Abbildung 5-9: Datenflussgraph der Linksverschiebung

Wie in den vorherigen Optimierungsroutinen auch, findet an dieser Stelle eine Überprüfung der notwendigen Instruktionen, die für solch eine Saturierung verwendet werden, statt. Zunächst erfolgt die Suche nach einer geeigneten Linksverschiebung bzw. einer Multiplikation mit der Konstanten 2. Ist diese Suche erfolgreich, findet eine Kontrolle der nachfolgenden Instruktionen statt. Anhand des Datenflussgraphen wird die zu verschiebende Variable daraufhin getestet, ob sie größer als 0x3FFFFFFF oder kleiner als 0xC0000000 ist. Ist dies der Fall, erfolgt eine Suche nach der möglichen Saturierung, sprich nach der Übertragung der Maximalwerte an die Ergebnisvariable. Zusätzlich muss kontrolliert werden, ob diese gesamte Struktur in eine Schleife eingebettet ist, die die Anzahl der Linksverschiebungen simuliert. Werden alle Anforderungen erfüllt, kann die Optimierung stattfinden. Die gesamte Struktur wird demnach durch den Aufruf des entsprechenden Intrinsics ersetzt. In diesem Fall durch:

“L_var_out = _sshl (L_var1 , var2);“

5.2 Bitbreitenreduktion

Die zweite Optimierung, die diese Diplomarbeit vorsieht, ist eine Bitbreitenreduktion. Durch die in Kapitel 3.3 beschriebene Datenflussanalyse ist eine Simulation des zu optimierenden Quellcodes möglich. Wie bereits dargestellt, besteht diese Datenflussanalyse aus zwei Teilen. Zum einen findet ein „Top-Down-Durchlauf statt, der den Datenfluss von den Quellen zu den Senken repräsentiert. Die zweite Phase allerdings besteht aus einem Durchlauf von den Senken zurück zu den Quellen. In dieser Phase ist es möglich, nicht benutze Bits zu erkennen und zu optimieren.
Eine Bitbreitenreduktion hat das Ziel, Datentypen von Variablen zu reduzieren. Die Aufgabe einer Bitbreitenreduktion ist es daher, nicht benötigte Kapazitäten von Bits aufzudecken und zu optimieren.

5.2.1 Algorithmus

Bits gelten als potentiell überflüssig, wenn sie in einer Berechnung keinen Einfluss auf das Ergebnis nehmen. Diese nicht benutzten Bits werden durch die Aufwärtsanalyse durch die Bit-Wertigkeit „x-Don´t Care“ gekennzeichnet. Voraussetzung für eine folgende Optimierung ist außerdem, dass diese potenziell überflüssigen Bits an den höchstwertigen Stellen einer Variablen zusammenhängend gefunden werden. Weiterhin ist dies nicht die einzige Möglichkeit, überflüssige Bits zu erkennen. Variablen, die ihren gesamten Wertebereich laut ihrer Deklaration nicht ausschöpfen, können ebenfalls Potential zur Optimierung beinhalten. So ist es beispielsweise denkbar, dass eine 32 Bit-Variable, die einen konstanten Wert nicht überschreitet, auf einen Datentypen reduziert werden kann, der weniger Bits, aber dennoch den benötigten Wertebereich zur Verfügung stellt. In diesem Fall wären das je nach Bitbreitenbedarf die Datentypen „short“ mit 16 Bits und „char“ mit 8 Bits. Da die verschiedenen Datentypen in zwei Varianten, einmal mit Vorzeichen und einmal ohne, deklariert werden können, ist eine genaue Untersuchung der verwendeten Variablen notwendig. Der in dieser Arbeit entworfene Algorithmus benutzt eine Reihe von Regeln, die das Erkennen von unbenutzten Bits möglich macht. Die folgende Auflistung zeigt diese Art der verwendeten Regeln:

Eine Variable enthält Potential zu einer Optimierung gemäß einer Bitbreitenreduktion, falls

- eine Menge höchstwertiger Bits mit dem Wert „x-Don´t Care“ gekennzeichnet worden ist. Dabei spielt es keine Rolle, ob eine Variable mit oder ohne Vorzeichen vorliegt. Eine denkbar mögliche Optimierung zeigt Abbildung 5-10.

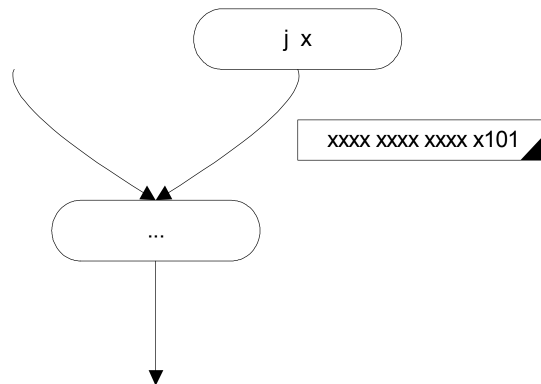


Abbildung 5-10: „x“-Werte kennzeichnen überflüssige Bits

Die vorliegende Variable ist als „short“ deklariert und besitzt daher eine Bitbreite von 16 Bits. An den führenden 13 Bits, die mit „x“ gekennzeichnet werden konnten, erkennt der Algorithmus nun die Überflüssigkeit der verwendeten Bits. An dieser Stelle wäre eine Optimierung der Variablen von „short“ auf „char“ mit 8 Bits möglich, da die führenden 8 Bits nicht benötigt werden.

- eine Menge höchstwertiger Bits den konstanten Wert „0“ beinhaltet. Auch in diesem Fall spielt es keine Rolle, ob eine Variable mit oder ohne Vorzeichen deklariert worden ist. Führende Bits mit der Wertigkeit „0“ spiegeln ein nicht Ausnutzen der zur Verfügung stehenden Bitbreiten dar. Bei einer möglichen Optimierung durch eine Bitbreitenreduktion ist allerdings zu beachten, dass das erste folgende Bit nach den überflüssigen Bits, die reduziert werden, die gleiche Wertigkeit besitzt wie die führenden Bits. Ansonsten käme es bei einer Reduktion der Bitbreite zu einem Vorzeichenwechsel. Das folgende Beispiel zeigt diese Situation:

int a = 20000; // a entspricht 0000000000000000100111000100000

Die Variable a entspricht nach dieser Deklaration und Zuweisung der Konstanten 20000 der auf der rechten Seite dargestellten Binärzahl. Zu sehen ist an diesem Beispiel, dass die 16 höchstwertigen Bits die Wertigkeit „0“ besitzen, was eine Bitbreitenreduktion ermöglichen würde, da die zugewiesene Konstante nicht mehr als 16 Bits benötigt. Eine tatsächliche Optimierung ist in diesem Fall allerdings nicht möglich, da der resultierende Wert „100111000100000“ eine „1“ an führender Stelle besitzt. Wie in Kapitel 2.1.2.1 erläutert, stellt diese Binärzahl im Zweierkomplement eine negative Zahl dar. Da Variable a aber einen positiven Wert beinhaltet, ist eine Bitbreitenreduktion nicht möglich. Anders wäre es bei der Deklaration einer „unsigned int“-Variablen. Dieser Datentyp sieht kein Vorzeichen vor, und der untere Wertebereich endet bei „0“. In diesem Fall wäre eine Optimierung durch eine Bitbreitenreduktion möglich und ergäbe folgendes Ergebnis:

unsigned int a = 20000; ⟹ **unsigned short a = 20000;**

- eine Menge höchstwertiger Bits den konstanten Wert „1“ beinhaltet und als Variable mit Vorzeichen deklariert worden ist. Wie in Kapitel 2.1.2.1 gezeigt wurde, stellt die Konstante „1“ an der höchstwertigen Stelle einer Variablen einen negativen Wert dar. Durch führende Bits mit der Wertigkeit „1“ werden somit analog zum vorherigen Punkt Bits gekennzeichnet, die nicht verwendet werden. Das folgende Beispiel zeigt solch eine Variable:

int a = -10000; // a entspricht 11111111111111111101100011110000

Die Konstante „-10000“ könnte demnach in einer Variablen mit einer Bitbreite von 16 Bits ausreichend dargestellt werden. In diesem Fall wäre eine Optimierung von „int“ mit 32 Bits nach „short“ mit 16 Bits möglich. Zu beachten ist auch in dieser Situation, dass das erste nachfolgende Bit hinter den als überflüssig erkannten und optimierten Bits die gleiche Wertigkeit besitzen muss. Würde eine Situation analog zur vorangehenden Regel vorliegen, wäre eine Optimierung nicht möglich.

int a = 20000; // a entspricht 11111111111111111011000111011111

Durch die Reduktion von 16 Bits würde das führende Bit zu einer „0“ werden. Durch dieses Kennzeichen würde das Vorzeichen wechseln und einen falschen Wert darstellen.
Falls jedoch eine mögliche Optimierung erkannt wird, ist folgende Handhabe möglich:

int a = -10000; ⟹ **short a = -10000;**

Dieses Verfahren ist allerdings nur möglich, falls eine Variable mit Vorzeichen deklariert worden ist. Im anderen Fall stellt ein Bit der Wertigkeit „1“ keine negative, sondern stets eine positive Zahlen dar. Durch eine Reduktion würde der Inhalt der Variablen lediglich verkleinert und damit das Ergebnis erheblich verfälscht werden.

Der in dieser Arbeit implementierte Algorithmus verwendet diese Regeln, um mögliche Optimierungen zu erkennen. Der folgende Pseudocode zeigt die Struktur des gesamten Algorithmus:

```
while ( !List_of_local_variables.is_empty() ) {

   var_sym *analyze_variable = List_of_local_variables.step();
   int min_bits_unused = MAX_32; // Init min_bits_unused with maximum

   for ( all nodes V  defining analyze_variable &&
         all source-nodes V containing analyze_variable ) {

      for( all edges E going out of node V ) {
        LInt Up_LInt = E->GetUpValue();
        min_bits_unused=min(Reduction_Rules(Up_LInt),min_bits_unused)
      }
   }
   if ( 0 < min_bits_unused < MAX_32 )
      Optimize_analyzed_variable ( analyze_variable, min_bits_unused );
}
```

Der Algorithmus iteriert über der Menge aller in der betrachteten Prozedur verwendeten Variablen. Im nächsten Schritt sucht dieses Verfahren alle Knoten V aus dem Datenflussgraphen, die Instruktionen beinhalten, die die betrachtete Variable definieren. Außerdem müssen die Knoten betrachtet werden, die lediglich die zu untersuchende Variable beinhalten. Nicht jede Variable wird in einer Prozedur definiert. Eine Vielzahl der verwendeten Variablen wird als Parameter in eine Funktion übergeben und lediglich in ihr benutzt. Diese Variablen werden durch Quellen im Datenflussgraphen dargestellt und ebenso betrachtet wie alle anderen verwendeten Variablen, die im Laufe der Prozedur definiert, sprich einen Wert zugewiesen bekommen. Dieser Algorithmus arbeitet nur innerhalb einer Prozedur und ist nicht in der Lage, Optimierungen über diese Grenzen hinweg zu bearbeiten. Diese Einschränkung hat zur Folge, dass Variablen, die potenziell überflüssige Bits enthalten und optimiert werden könnten, aber durch eine Parameterübergabe in die betrachtete Prozedur übergeben werden, durch diesen Algorithmus nicht verbessert werden können. Ebenso ist es nicht möglich, die Bitbreiten der Variablen zu reduzieren, die als Ergebnis aus einer Funktion zurückgegeben werden. Weiterhin werden mögliche Bitbreitenreduktionen bei Variablen, die innerhalb der untersuchten Prozedur als Parametereingabe für einen weiteren Funktionsaufruf benutzt werden, nicht unterstützt. Um diese Optimierung möglich zu machen, wäre eine Betrachtung der zusammenhängenden Funktionen nötig, da diese Variablen auch im weiteren Verlauf keine größeren Wertebereiche benutzen dürften.

Der weitere Ablauf des Algorithmus sieht nun eine Betrachtung aller ausgehenden Kanten E eines jeden Knotens V vor, die im vorherigen Schritt ausgewählt werden. Dabei wird die kleinste Anzahl von potenziell überflüssigen Bits ausgewählt. Dies geschieht durch die Funktion „Reduction_Rules“, die die oben genannten Regeln zur Bestimmung von potentiell überflüssigen Bits benutzt. Als Übergabeparameter dient der entsprechende LInt-Vektor, der den rückläufigen Datenfluss der betrachteten Kante widerspiegelt. Ergibt sich nun aus der ermittelten kleinsten Anzahl von nicht verwendeten Bits die Möglichkeit zu einer Optimierung, geschieht dies anschließend durch die im Pseudocode dargestellte Funktion „Optimize_analyzed_variable“. In dieser Funktion werden auch die oben genannten Einschränkungen zur Optimierung der Bitbreite einer Variablen geprüft. Zusätzlich zu den führenden potentiell überflüssigen Bits einer Variablen werden weitere Informationen über den Wert, den diese Variable annehmen kann, gesammelt. Es ist für statistische Zwecke weiterhin erforderlich, sämtliche nicht

benutzten Bits zu betrachten, um die durch diese Arbeit erstellten Ergebnisse mit den Resultaten in [BG00] zu vergleichen. Kapitel 6 greift dieses Thema erneut auf und zeigt den Vergleich zwischen den Resultaten dieser Arbeit und den Ergebnissen von Budiu und Goldstein.
Die gesammelten Informationen über die werden in vier Gruppen aufgeteilt:

- **Bits saved by declaration**
  Diese Gruppe stellt die Anzahl der Bits dar, die bereits durch die Deklaration eingespart wurden. Falls eine Variable beispielsweise mit dem Datentypen „short“ mit 16 Bits deklariert wird, hält der Algorithmus bereits 16 gesparte Bits in dieser Gruppe fest. Um den an dieser Stelle berechneten Wert prozentual auswerten zu können, werden alle untersuchten Variablen unabhängig ihrer tatsächlichen Bitbreite mit 32 Bits gezählt.

- **Whole Bytes**
  In dieser Gruppe werden die führenden Bits, die ganze Bytes darstellen, gezählt. Werden beispielsweise 18 führende Bits als potenziell überflüssig erkannt, werden die ersten 16 Bits in dieser Gruppe festgehalten.

- **Contiguous Bits**
  Anhand dieses Wertes erkennt man die folgenden potentiell überflüssigen Bits, die hinter den Bits gefunden werden, die ganze Bytes bilden. Um das vorherige Beispiel aufzugreifen, erhält man bei 18 potentiell überflüssigen Bits zwei folgende Bits, die in dieser Gruppe gezählt werden. Die an dieser Stelle gespeicherten Bits tragen nicht zu einer Optimierung bei. Diese Information wird lediglich zu Zwecken der Statistik gespeichert.

- **All other unused Bits**
  Zu dieser Gruppe zählen alle restlichen Bits, die in dem Wert einer Variablen gefunden werden. Dabei spielt es keine Rolle, an welcher Stelle diese Bits auftauchen. Ein Beispiel dazu wäre der folgende Wert:

00101011010101xx

In dieser Anreihung von Bits werden zwei potentiell überflüssige Bits am Ende gefunden. Diese Bits führen natürlich nicht zu einer Optimierung, sondern werden wiederum lediglich zu Statistikzwecken festgehalten.

Nachdem nun die kleinste Anzahl an führenden Bits aus den Werten, den die betrachtete Variable annehmen kann, ermittelt worden ist, erfolgt, falls möglich, die Optimierung. Dabei wird der passende Datentyp ermittelt, damit die untersuchte Variable mit dem tatsächlich notwendigen Wertebereich arbeiten kann.

Die folgende Aufzählung zeigt, wie in dieser Phase gearbeitet wird:

int **unsigned int**	*16 überflüssige Bits*	⟹	**short** **unsigned short**
	24 überflüssige Bits	⟹	**char** **unsigned char**
short **unsigned short**	*8 überflüssige Bits*	⟹	**char** **unsigned char**
char **unsigned char**		⟹	**kleinster Datentyp,** **es erfolgt keine** **Optimierung**

5.3 Multimedia-Befehle

Die dritte Optimierung, die durch diese Diplomarbeit behandelt wird, ist der Einsatz von Multimedia-Befehlen. Die Handhabung wurde bereits in Kapitel 2.1.2.4 erläutert. Kapitel 5.3.1 zeigt zunächst die verwendeten Multimedia-Befehle sowie Hilfsfunktionen, die notwendig sind, um die verwendeten Operanden vorzubereiten bzw. nach der Berechnung in die vorgesehenen Ergebnisvariablen zu übertragen. Kapitel 5.3.2 zeigt, wie der für diese Aufgabe entworfene Algorithmus arbeitet, und welche Voraussetzungen vorhanden sein müssen, um entsprechende Quellcode-Transformationen durchführen zu können. Anhand von aufgestellten Regeln, die der Algorithmus benutzt, werden mögliche Einsätze von Multimedia-Befehlen dargestellt.

5.3.1 Verwendete SIMD-Befehle

Der Compiler stellt folgende Intrinsics, die für diese Optimierung nützlich sind, zur Verfügung:

- **int _pack2 (short , short)**
  Dieses Intrinsic ist ein nützliches Hilfsmittel, um die Operanden der zu optimierenden Funktionen vorzubereiten. Wie bereits in Kapitel 2.1.2.4 gezeigt wurde, ist es für diese Art von Intrinsic notwendig, die zu berechnenden Operanden zusammenzufassen. Diese Funktion erhält man durch das „_pack2“-Intrinsic. Als Übergabe für diese Funktion dienen zwei 16 Bit-Variablen, die zu einer 32 Bit-Variablen zusammengefasst werden. Abbildung 5.11 zeigt die Vorgehensweise dieses Intrinsics.

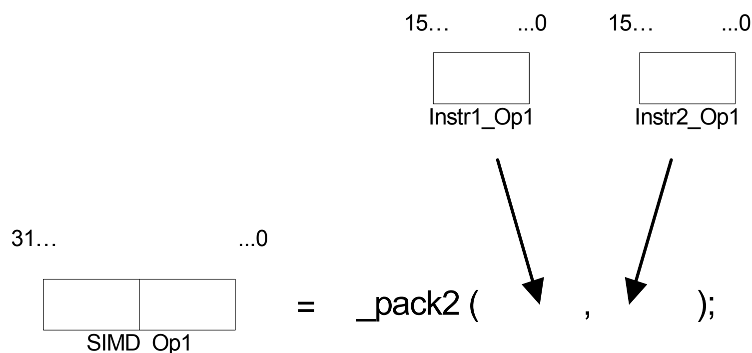


Abbildung 5-11: Benutzung des _pack2-Intrinsics

Die jeweiligen ersten bzw. zweiten Operanden der zu optimierenden Instruktionen dienen als Eingabe für diese Funktion. Die Ergebnisvariable SIMD_Op1 beinhaltet nun die beiden Operanden „Instr1_Op1“ und „Instr2_ Op1“. Die erste der beiden Operanden wurde in den 16 hochwertigen Bits der Variable SIMD_Dst abgelegt, wobei der zweite Operand in den 16 unteren Bits von „SIMD_Op1“ Platz findet.

- **int _add2 (int , int)**
  Dieses Intrinsic steht zur Berechnung zweier Additionen mit 16 Bitzahlen zur Verfügung. Das Intrinsic erhält aber nun die beiden vorher durch den „_pack2“-Befehl zusammengestellten 32 Bit-Operanden. Durch dieses Intrinsic findet nun eine Berechnung der oberen 16 Bits des ersten mit den oberen 16 Bits des zweiten Operanden, sowie der unteren 16 Bits der ersten und unteren 16 Bits des zweiten Operanden statt. Das besondere an diesem Intrinsic ist, dass beide Teilberechnungen parallel verarbeitet werden.

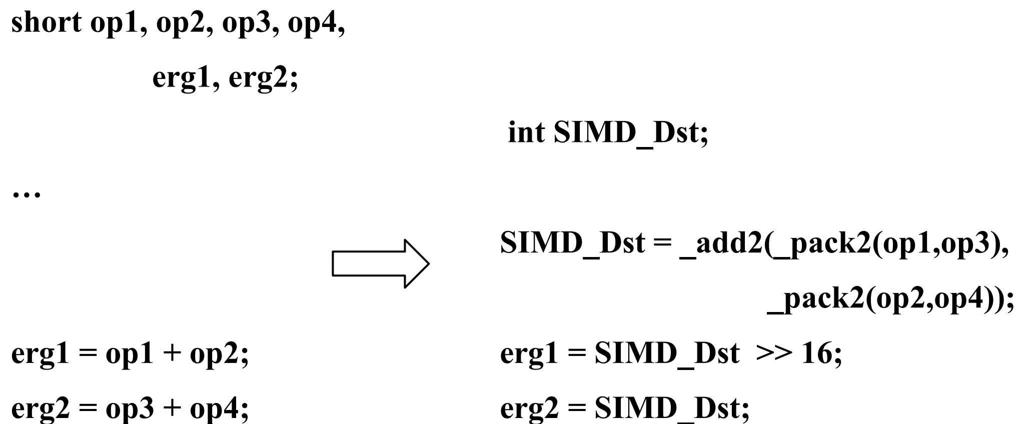
- **int _sub2 (int , int)**
  Analog dazu funktioniert das „_sub2“-Intrinsic. Auch hier werden die durch den „pack2“-Befehl erstellten 32 Bit-Operanden übergeben, und es findet eine parallele Berechnung der Instruktionen statt. Allerdings handelt es sich in diesem Fall um zwei Subtraktionen.

Nach der Berechnung der eigentlichen Addition bzw. Subtraktion ist es nun erforderlich, die eigentlich resultierenden Ergebnisse den Ergebnisvariablen der ursprünglichen Instruktionen zuzuweisen. Auch dafür steht ein Intrinsic zur Verfügung. Allerdings haben Tests herausgestellt, dass dies nicht die beste Alternative bezüglich der benötigten Laufzeit ist. Daher werden die Ergebnisse auf konventionelle Art den Ergebnisvariablen zugewiesen.

```
erg1 = SIMD_Dst >> 16;
erg2 = SIMD_Dst;
```

„erg1“ und „erg2“ stellen die beiden 16 Bit-Ergebnisvariablen der ursprünglichen Instruktionen dar. Durch das Verschieben der 16 oberen Bits der Ergebnisvariablen „SIMD_Dst“, die das Resultat eines SIMD-Befehls darstellen soll, werden die

oberen 16 Bits der Ergebnisvariablen erg1 zugewiesen. Durch die einfache Definition der „erg2“-Variablen mit der SIMD-Ergebnisvariablen werden ihr aufgrund des implizierten Type-Casts die unteren 16 Bits zugewiesen. Zu beachten ist, dass das Zuweisen der Ergebnisvariablen an den gleichen Stellen im Quellcode durchgeführt wird, an denen die ursprünglichen Instruktionen platziert waren. Die gesamte Codestruktur, die zwei Instruktionen koppelt und durch Multimedia-Befehle bearbeiten lässt, sieht wie folgt aus:



```
short op1, op2, op3, op4,
      erg1, erg2;
                                   int SIMD_Dst;
...
                        =>         SIMD_Dst = _add2(_pack2(op1,op3),
                                                    _pack2(op2,op4));
erg1 = op1 + op2;                  erg1 = SIMD_Dst  >> 16;
erg2 = op3 + op4;                  erg2 = SIMD_Dst;
```

5.3.2 Algorithmus

Um eine Optimierung durch Multimedia-Befehle vornehmen zu können, müssen bestimmte Voraussetzungen erfüllt und bestimmte Regeln beachtet werden. Dieses Unterkapitel beschäftigt sich mit der Handhabung und dem Einsatz von SIMD-Befehlen unter Beachtung dieser Einschränkungen.

Die folgende Auflistung spiegelt die Regeln wider, die erfüllt sein müssen, um zwei Instruktionen durch SIMD-Befehle optimieren zu können:

- **Die zu berechnenden Operanden müssen überprüft werden.**
  Die oben genannten Intrinsics berechnen 16 Bit-Operanden mit Vorzeichen. Daher ist es notwendig, die Datentypen dieser Operanden zu überprüfen. Erlaubt sind demnach Operanden, die eine Bitbreite von 16 Bits nicht überschreiten. Behandelt werden demnach Variablen mit dem Datentypen „short“ mit 16 Bits und „char“ mit 8 Bits. Ein möglicher Überlauf, der bei 8 Bit-Operanden zu Stande kommen kann, wird bei der Zuweisung der Ergebnisvariablen der zu optimierenden Instruktionen durch einen implizierten „Type-Cast“ korrigiert. Zu bemerken ist außerdem, dass

die Ergebnisvariablen der zu berechnenden Instruktionen eine abweichende Bitbreite benutzen dürfen. Es wäre denkbar, dass eine Operation zwei 8 Bit-Zahlen addiert und das Ergebnis in einer 16 Bit-Variablen speichert.

- **Die resultierenden Variablen der zu untersuchenden Instruktionen müssen unabhängig voneinander sein.**
  Unabhängig bedeutet in diesem Fall, dass eine Ergebnisvariable nicht aus der anderen hervorzugehen darf. Eine Struktur wie im nächsten Beispiel wäre demnach unzulässig:

```
erg1 = op1 + op2;     // Instruktion 1
erg2 = erg1 + op3;    // Instruktion 2
```

  Falls eine der Ergebnisvariablen durch die Berechnung der anderen Ergebnisvariablen resultiert, muss gewährleistet sein, dass der neue Wert der Variablen bereits vollständig ermittelt worden ist. So ist es nicht möglich, eine parallele Berechnung der beiden Instruktionen vorzunehmen.
  Zu überprüfen ist diese Konstellation durch den gegebenen Datenfluss. Es ist lediglich zu überprüfen, ob ein Weg von Instruktion 1 zu Instruktion 2 und umgekehrt im Datenflussgraph existiert.

- **Die Positionen der zu untersuchenden Instruktionen muss analysiert werden.**
  Eine weitere Voraussetzung ist die korrekte Position der zu optimierenden Instruktionen im Quellcode. Genauer betrachtet werden müssen demnach die Blöcke des Quellcodes. Unter anderem heißt das, dass keine Optimierung über die Grenzen von Schleifen-Körpern erlaubt ist. Die folgenden Zeilen zeigen ein Beispiel für eine Schleife:

```
 erg1 =  op1 + op2; // Instruktion 1
for ( int i = 0; i < 10; i++) {
     erg2 = op3 + op4; // Instruktion 2
}
```

Dieses Code-Konstrukt darf augenscheinlich nicht optimiert werden. Durch den Einsatz eines Multimedia-Befehls an dieser Stelle würde die gesamte Logik dieser Struktur verloren gehen. Eine weitere problematische Struktur ist die Logik eines If-Statements. Das folgende Code-Fragment zeigt eine mögliche Struktur:

```
if ( … ) {
   erg1 = op1 + op2;  // Instruktion 1
} else {
   erg2 = op3 + op4; // Instruktion 2
}
```

Generell ist es möglich, eine Ersetzung von Instruktionen, die sich in verschiedenen „Scopes" eines If-Statements befinden, durch SIMD-Intrinsics durchzuführen. Der Multimedia-Befehl, der beide Instruktionen berechnet, muss bereits ausgeführt sein, bevor die Logik des If-Statements verarbeitet wird, damit die ursprünglichen Instruktionen mit dem Inhalt der Ergebnisvariablen des SIMD-Intrinsics gefüllt werden können. Allerdings bleibt es fraglich, ob es sich dabei tatsächlich um eine Optimierung handelt, denn bei der obigen Struktur würde eine Addition ausgeführt, aber nicht verwendet werden. Denkbar wäre auch eine Struktur mit mehrfach verschachtelten If-Statements, die das Problem noch erschweren würden. Durch die gegebene Komplexität wird an dieser Stelle verzichtet, tiefer auf dieses Problem einzugehen, da der Umfang die Kapazität dieser Diplomarbeit sprengen würde. Daher werden in dieser Arbeit nur Optimierungen von Instruktionen betrachtet, die sich in den gleichen Bereichen, sprich den gleichen „Scopes" befinden.

- **Die Verwendung der Operanden in den zu untersuchenden Instruktionen muss genauer überprüft werden.**

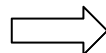
  Es bleibt zu überprüfen, ob die verwendeten Operanden zwischen den zu untersuchenden Instruktionen neu definiert werden. Da die ursprünglichen Ergebnisvariablen an den gleichen Stellen mit dem Ergebnis des SIMD-Intrinsics gefüllt werden, ist es demnach kein Problem, falls die zu berechnenden Operanden der ersten Funktion neue Inhalte zugewiesen be-

kommen. Allerdings müssen neue Definitionen der in der zweiten Operation verwendeten Operanden berücksichtigt werden. Die nächsten Zeilen zeigen ein solches Konstrukt:

```
erg1 = op1 + op2;  // Instruktion 1

op3 = 5;

erg2 = op3 + op4;  // Instruktion 2
```

Bei dieser Kombination von Wertzuweisungen ist eine genauere Betrachtung der Position des einzusetzenden SIMD-Intrinsics notwendig. Wie in den obigen Fällen ist es daher nicht möglich, den Multimedia-Befehl vor die erste zu optimierende Operation zu setzen. Durch die Änderung des Inhalts der Variablen „op3“ ist der Algorithmus gezwungen, das SIMD-Intrinsic erst nach der Definition von „op3“ in den Quellcode einzubetten. Das Ergebnis dieser Quellcode-Transformation zeigt das nächste Code-Konstrukt:

```
erg1 = op1 + op2;

op3 = 5;                    op3 = 5;

erg2 = op3 + op4;           SIMD_Dst = _add2(_pack2(op1,op3),
                                             _pack2());
                             erg1 = op1 + op2;
                             erg2 = op3 + op4;
```



Durch den Standort des neuen SIMD-Intrinsics werden die Ergebnisvariablen „erg1“ und „erg2“ erst nach der letzten Definition der zu addierenden Operanden der zweiten Instruktion berechnet. Das hat wiederum zur Folge, dass die Ergebnisvariable des Multimedia-Befehls erst an dieser Stelle verfügbar ist. Um eine korrekte Abfolge der Instruktionen zu gewährleisten, müssen demnach die verwendeten Operanden der ersten Operation sowie deren Ergebnisvariable zwischen der ursprünglichen ersten Instruktion und dem potentiell neuen Multimedia-Befehls untersucht

werden. Sobald eine der zu berechnenden Variablen ebenfalls neu definiert wird, ist eine Optimierung dieser beiden Instruktionen nicht möglich. Durch den neuen Inhalt dieser Variablen würde das Ergebnis verfälscht werden, da der Inhalt der Variablen erneut überschrieben werden würde. Ebenso ist eine Benutzung der Ergebnisvariablen der ersten Instruktion nicht vor der Berechnung des SIMD-Intrinsics möglich. Das bedeutet, dass auch in diesem Fall keine Optimierung stattfinden darf, falls die Ergebnisvariable der ersten Instruktion vor der letzten Definition eines Operanden der zweiten Instruktion verwendet wird.

Folgende Zeilen zeigen ein denkbares Code-Fragment, das nicht durch ein SIMD-Intrinsic optimiert werden könnte:

```
erg1 = op1 + op2;          // Instruktion 1

erg1_use = erg1 - 5;       // Instruktion, die „erg1“ benutzt
erg1 = 5                   // Instruktion, die „erg1“ neu definiert
op1 = 7;                   // Instruktion, die „op1“ oder „op2“ neu definiert

op3 = 5;                   // Definition einer Variablen der zweiten Instruktion

erg2= op3 + op4;           // Instruktion 2
```

Durch die Definition der Variablen „op3“ wird es notwendig, das SIMD-Intrinsic erst nach dieser Instruktion einzusetzen. Durch ein Benutzen der Variablen „erg1“ oder eine Definition einer der drei verwendeten Variablen der ersten Instruktion wird eine Optimierung durch einen Multimedia-Befehl verhindert.

Der für diese Optimierung entworfene Algorithmus verwendet die oben genannten Regeln, um den genauen Standort im Quellcode für den Einsatz eines Multimedia-Befehls festzulegen, bzw. um zu erkennen, dass die Kombination zweier betrachteter Instruktionen nicht optimiert werden kann. Der folgende Pseudocode zeigt die Struktur des verwendeten Algorithmus:

```
for ( all add- or sub-instructions I_1 ) {
   if ( check operands of instruction I_1 ) {
         for ( all following instructions I_2 matching with instruction I_1 ) {
                  if ( check operands of instruction I_2 ) {
                           if ( check_SIMD_Rules ( I_1, I_2 ) )
                                    Optimize_instr ( I_1, I_2 );
                  }
         }
   }
}
```

Der Algorithmus iteriert über der Menge der Instruktionen innerhalb einer Prozedur. Zunächst besteht die Aufgabe darin, eine Instruktion zu finden, die einen passenden Operator vorweisen kann. Die verwendeten Intrinsics wurden im vorherigen Kapitel erläutert. Um sicherzustellen, dass eine gültige Instruktion gefunden worden ist, werden die verwendeten Operanden geprüft. Die Ergebnisvariable sowie die verwendeten Operanden dürfen eine Bitbreite von 16 nicht überschreiten. Das bedeutet, dass in diesem Fall „short“-Variablen mit 16 Bits sowie „char“-Variablen mit 8 Bits betrachtet werden können. Anschließend erfolgt die Suche nach einer zweiten passenden Instruktion. Durch die Überprüfung der genannten Regeln wird nun entschieden, ob der Einsatz für einen Multimedia-Befehl zulässig ist. Falls diese Überprüfung erfolgreich stattfindet, wird die Quellcode-Transformation durchgeführt. Das passende Intrinsic wird mit den notwendigen Instruktionen an die richtige Stelle in den Quellcode eingesetzt. Die beiden optimierten Instruktionen werden aus dem Quellcode entfernt.

6 Ergebnisse

Dieses Kapitel behandelt die in dieser Diplomarbeit durch die implementierten Optimierungen erzielten Ergebnisse. In den nachfolgenden Kapiteln werden die betrachteten Benchmarks vorgestellt sowie die Resultate, die durch die beschriebenen Optimierungen erreicht wurden. Behandelt wurden die Referenz-Implementierungen des G.723.1 Dual Speech Coder, des GSM, des G.721 und des ADPCM-Kodierers bzw. Dekodierers.
Um Daten über globale Netzwerke transportieren zu können, müssen die erforderlichen Daten zunächst nach einem definierten Verfahren kodiert bzw. digitalisiert werden. Das angewendete Verfahren wird als Codec bezeichnet. Liegen die Daten bereits in digitalisierter Form vor, besteht die Möglichkeit einer zusätzlichen Datenkomprimierung, um die erforderliche Bandbreite zur Übertragung möglichst gering zu halten. Bei der Betrachtung einer guten Sprachqualität und einer geringen Bandbreite spielt auch die Komplexität eines Codec eine wichtige Rolle. Sie spiegelt den benötigten Zeitbedarf bis zur Wiedergabe der Sprachdaten wider.
Dieses Kapitel zeigt zunächst die untersuchten Codecs. Anschließend werden die Ergebnisse der Optimierungen aller drei in dieser Arbeit durchgeführten Techniken vorgestellt.

6.1 G.723.1 Dual Speech Coder

Diese Software ist ein Bestandteil des H.323 Standard und dient der Audio-Kompression, -Kodierung und –Dekodierung und bildet eine Referenz in der Internet-Telephonie. Der G.723.1 Standard bildet einen Bestandteil des H.323 Terminals. Verbunden mit der Audio-Ausrüstung ist dieser Baustein zuständig für die Audiokompression. Zum Einsatz kommt dieses Terminal bei bekannter und kommerzieller Software wie Netmeeting von Microsoft oder generell beim VoIP-Protokoll.
Abbildung 6-1 zeigt den Standort des Audio Codecs. Eingebettet in das H.323 Terminal gibt der G.723.1 seine verarbeiteten Daten weiter bis in das Local Area Network und damit in das damit verbundene Internet. Der entgegengesetzte Weg beginnt mit der Aufnahme von Audiodaten, die durch das Local Area Network in

das H.323 Terminal gelangen. Dekodiert durch den Audio Codec, wandern die Daten direkt zum Audio Equipment und damit zum Benutzer.
Der G.723.1 Codec ist als zeitkritisch zu betrachten. Somit ist es nicht nur Aufgabe, die Korrektheit und die Portierbarkeit der Codec-Software als Priorität zu sehen, sondern vor allem auch die Geschwindigkeit der Algorithmen, um die geforderten Daten zu bearbeiten.

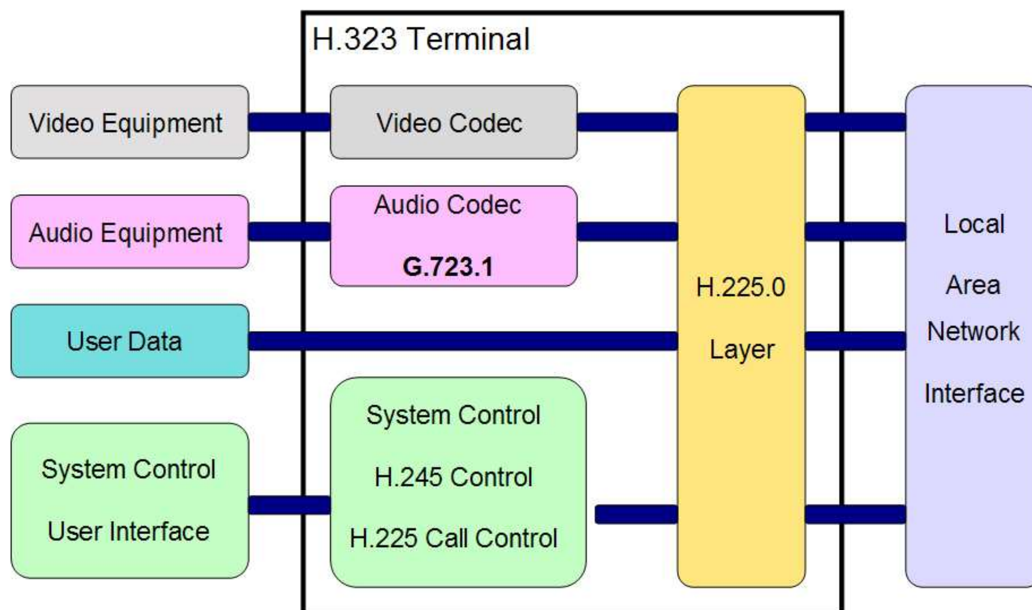


Abbildung 6-1: G.723.1 als Bestandteil des H.323 Standards

6.1.1 Saturierende Arithmetik

Dieses Unterkapitel spiegelt die Ergebnisse der ersten Optimierung dieser Arbeit wider. In den Quellcodes der G.723.1 Referenzimplementierung wurden die in Kapitel 5.1 dargestellten Strukturen zur Saturierung gefunden. Durch den Einsatz der zugehörigen Intrinsics konnte eine durchschnittliche Laufzeitverbesserung von 39% erzielt werden.

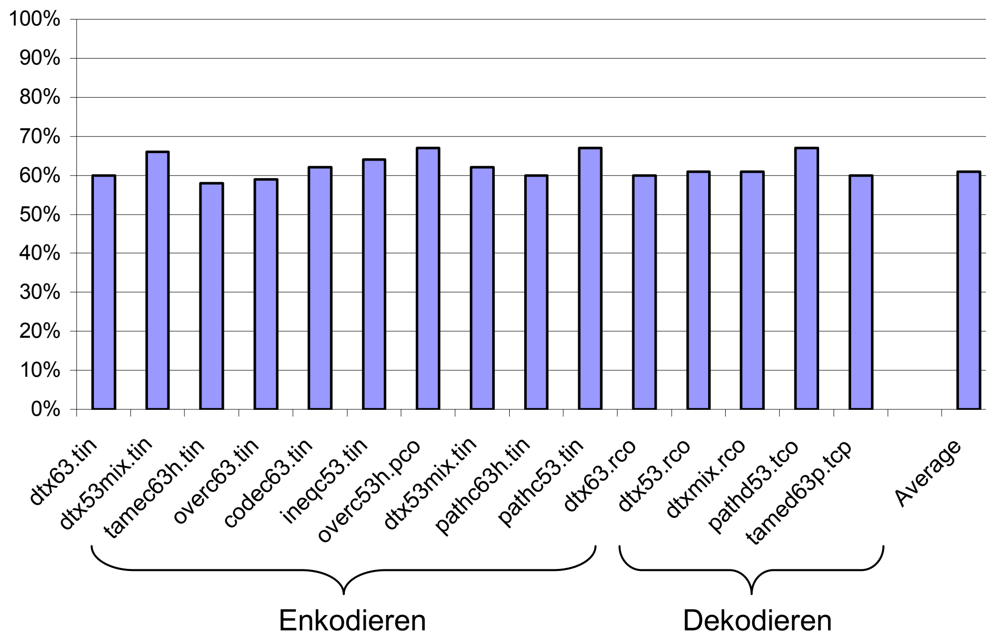


Abbildung 6-2: Relative Laufzeiten des G.723.1 Codecs für saturierende Arithmetik

Abbildung 6-2 zeigt 15 Durchläufe mit unterschiedlichen Testdaten, die sequentiell aus der angegebenen Datei gelesen und durch die G.723.1-Implementierung verarbeitet wurden. Die horizontale Achse des Säulendiagramms zeigt die unterschiedlichen Dateien, deren Inhalt durch den G.723.1 Dual Speech Coder bearbeitet worden sind. Bei den ersten zehn Einträgen handelt es sich um Audio-Dateien, die durch den Kodierer komprimiert worden sind. Die weiteren fünf Einträge sind Dekodierungen von Audio-Files. Der letzte Eintrag in dieser Reihe spiegelt den Durchschnitt aller Testdurchläufe wider. Die vertikale Achse des Säulendiagramms zeigt die prozentuale Laufzeit eines jeden Durchlaufs im Vergleich zur originalen Laufzeit vor der Optimierung. Die Balken geben demnach die verbleibende Laufzeit in Prozent an, die mit der Ausnutzung von Saturierungs-Intrinsics, wie sie in Kapitel 5.2 dargestellt wurde, erreicht werden konnte. Die maximale relative Laufzeit eines Tests beträgt 67%, die niedrigste 58% der ursprünglichen benötigten Zyklen. Damit ist eine minimale Laufzeitverbesserung von 33% (overc53.h und pathc53.tin) und eine maximale Verbesserung von 42% (tamec63.h) erreicht worden. Durch diese sehr hohen Gewinne erkennt man die Mächtigkeit von „Compiler-known Functions“, und damit wurde gezeigt, wie viel Potential für Optimierungen außerhalb optimierender Compiler noch zur Verfügung steht.

6.1.2 Bitbreiten-Reduktion

Die in diesem Unterkapitel dargestellten Ergebnisse zeigen die Resultate der in dieser Arbeit implementierten Bitreduktion, die auf die G.723.1 Referenzimplementierung angewendet worden ist. Die Darstellung der Ergebnisse ist an die Messverfahren angepasst worden, wie sie in „Bit Value Interference: Detecting and Exploiting Narrow Bitwidth Computations“ von Budiu und Goldstein [BG00] verwendet und in Kapitel 5.2 erläutert worden sind. Das Verfahren, das Budiu und Goldstein anwenden, untersucht zusätzlich zu den tatsächlich unbenutzten Bits die von der Deklaration in den Quellen eingesparten Bitbreiten. So erfolgt eine automatische Einsparung von 16 Bits, sobald eine Variable durch den Datentypen „short“ mit 16 Bits statt „integer“ mit 32 Bits deklariert worden ist.

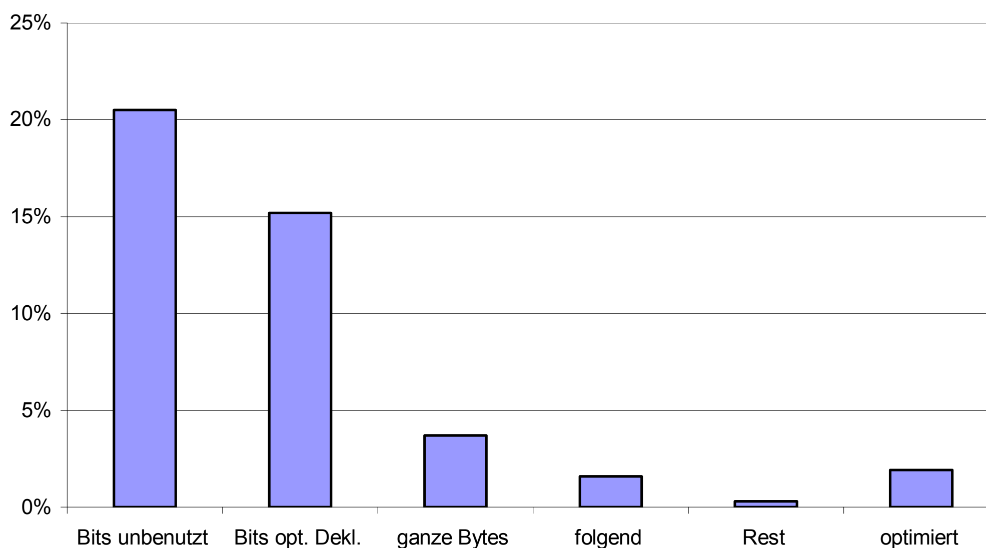


Abbildung 6-3: Bitreduktion der G.723.1-Implementierung

Durch die damit ansteigende Anzahl der Bits, die insgesamt in der Implementierung verwendet worden sind, reduzieren sich die restlichen prozentualen Ergebnisse dieses Verfahrens. So ist zu bemerken, dass die an dieser Stelle potentiell überflüssigen Bits, die in zusammenhängenden Bytes gefunden worden sind, stark erhöhen würden, falls nur die tatsächlich benutzen Bitbreiten berechnet werden würden. Abbildung 6-3 zeigt die Ergebnisse im Detail. Die horizontale Achse zeigt die Einteilung der Bits, die in Kapitel 5.2.1 dargestellt wurde. Die vertikale Achse des Säulendiagramms zeigt die Anzahl der Bits in Relation zur Gesamtanzahl der Bits, die in den untersuchten Prozeduren verwendet worden ist. Dabei zeigt der erste Wert die insgesamt unbenutzten Bits, die in der G.723.1-

Referenzimplementierung gefunden worden sind. Sie belaufen sich auf 20,5%. Der zweite Balken zeigt den prozentualen Anteil der Bits, die bereits durch die Deklaration eingespart worden sind. Er beläuft sich auf 15,2%. Der anschließende Balken zeigt die Bits, die in ganzen Bytes gefunden worden sind und beläuft sich auf 3,4%. So werden beispielsweise 2 Bytes, sprich 16 Bits als unbenutzt erkannt, wenn durch die Bitreduktion 16 unbenutzte Bits an den höchsten Stellen einer Variablen zusammenhängend entdeckt worden sind. Der vierte Balken stellt die darauf folgenden Bits dar. Abbildung 6-4 zeigt ein Beispiel dieses Messverfahrens:

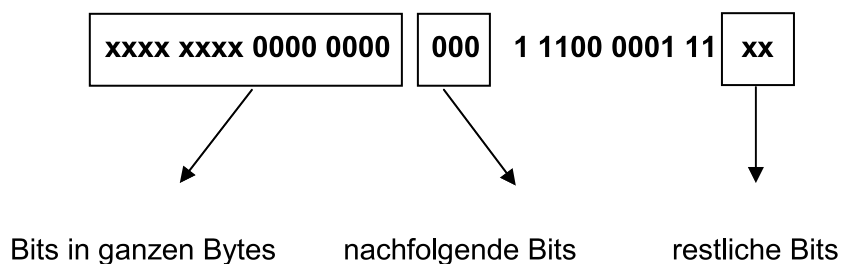


Abbildung 6-4: Bits in ganzen Bytes und nachfolgende Bits

Der fünfte Messwert zeigt den prozentualen Anteil der restlichen Bits, die weiterhin gefunden worden sind. Diese restlichen Bits zeichnen sich dadurch aus, dass sie durch ein „x – Don´t Care“-Bit gekennzeichnet sind. In diesem Fall beläuft sich der Wert auf 0,3%. Optimiert wurden in den Quellen lediglich 1,9% der verwendeten Bitbreiten. Dieser Wert fällt kleiner aus als die in ganzen Bytes gefundenen Bits. Das ist zu begründen durch die Benutzung der gefundenen Bits. Die in dieser Arbeit erstellte Optimierung basiert darauf, dass nur einzelne Prozeduren betrachtet werden. So ist dieses „Werkzeug“ nicht in der Lage, Variablen, die als Parametereingaben in die zu untersuchende Prozedur übergeben werden, zu optimieren. Weiterhin ist es nicht möglich, Variablen zu optimieren, die zwar unbenutzte Bits enthalten, die aber als Parameter für weitere Funktionsaufrufe in dieser Prozedur verwendet werden. Um die Bitbreiten dieser Variablen optimieren zu können, müsste man die benutze Funktion betrachten, die durch die untersuchte Funktion aufgerufen wird. Diese Art der Optimierung wurde in dieser Arbeit nicht betrachtet. Um eine Übersicht von den tatsächlich optimierten Variablen zu erhalten, hilft Abbildung 6-5. Zu sehen ist der prozentuale Anteil der Variablen-Arten.

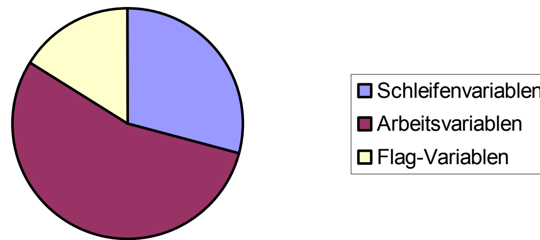


Abbildung 6-5: Anteil in % der optimierten Variablenarten im G.723.1

Wie in Abbildung 6-5 zu erkennen ist, bestehen die meisten Optimierungen bezüglich der Bitreduktion aus typischen Arbeitsvariablen, die einen prozentualen Anteil von 55% ausmachen. Darunter fallen auch Variablen, die temporär benutzt werden. Der zweitgrößte Anteil wird von den Schleifenvariablen eingenommen. Hier ist ein prozentualer Anteil von 29% zu verzeichnen. Die Flag-Variablen, die nur einen konstanten Wert zugewiesen bekommen, halten einen Anteil von 16% der optimierten Variablen.

6.1.3 Multimedia-Befehle

An dieser Stelle werden die Ergebnisse der von dieser Arbeit erstellten dritten Optimierung dargestellt. Die Quellen der G.723.1-Implementierung wurden auf eine mögliche parallele Verarbeitung von Instruktionen, wie sie in Kapitel 5-3 aufgezeigt wurde, untersucht. Durch ein Verbinden zweier Instruktionen und Berechnung dieser durch einen Multimedia-Befehl soll eine beschleunigte Laufzeit erreicht werden. Es ist zu bemerken, dass die zweite Optimierung dieser Arbeit, die Bitreduktion, als Vorbereitung auf den Einsatz von Multimedia-Befehlen gezielt hat. Durch das Verbinden dieser Optimierungen wurde allerdings nicht der gewünschte Zweck erzielt. Es konnten vor der Bitreduktion wie danach lediglich 13 Strukturen im Quellcode gefunden werden, die durch SIMDs optimiert werden konnten. Dieser Umstand ist damit zu begründen, dass SIMD-Instruktionen auf die parallele Bearbeitung großer Datenmengen, wie sie typischerweise in Multimedia-Applikationen vorkommen, ausgerichtet sind. Zur Vorbereitung des Einsatzes von SIMD-Befehlen wird häufig Loop Unrolling eingesetzt. Abbildung 6-6 zeigt ein typisches Beispiel für Loop Unrolling.

```
for (i=0; i<N; i++)              for (i=0; i<N; i+=2)
{                                {
   A[i] = B[i] + C[i];      =>      A[i]   = B[i] + C[i];
                                    A[i+1] = B[i+1] + C[i+1]
}                                }
```

Abbildung 6-6: „Loop Unrolling“

Hiernach können beide Array-Additionen der abgerollten Schleife parallel ausgeführt werden. Typischerweise werden große Datenmengen in Feldern gespeichert. Die in dieser Arbeit präsentierten Optimierungstechniken enthalten jedoch keinerlei Array-Datenflussanalysen, so dass sowohl Bitbreiten-Reduktion als auch die Optimierung von Multimedia-Befehlen ausschließlich auf skalaren Variablen zum Zuge kommt. Dieser 'zweckentfremdete' Einsatz von SIMD-Befehlen verbunden mit dem Umstand, dass die Bitreduktion effektiv nur 1,9% aller Bits durch Optimierung einsparen kann (vgl. Abschnitt 6.1.2), führt zu dem Effekt, dass die Bitreduktion keinen Einfluss auf die SIMD-Optimierung hat. Dennoch sind teilweise beträchtliche Laufzeitverbesserungen zu verzeichnen. Abbildung 6-7 zeigt die erreichten Resultate.

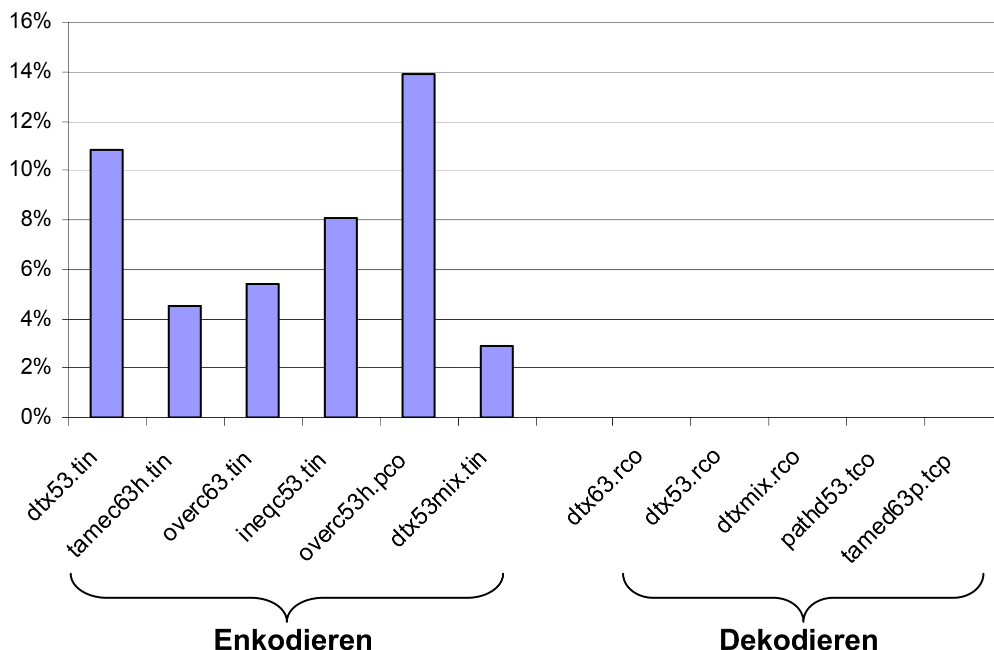


Abbildung 6-7: Laufzeitverbesserung nach SIMD-Optimierung im G.723.1

Die horizontale Achse beschreibt wiederum die verschiedenen Audio-Dateien, die durch das Verfahren bearbeitet worden sind. Die vertikale Achse allerdings zeigt in dieser Abbildung die nach dem Einsatz von SIMD-Intrinsics gemessene Laufzeitverbesserung in Prozent. Zu erkennen ist, dass nur bei sechs der getesteten Durchläufe Performance-Steigerungen zwischen 2,9% und 13,9% erzielt werden konnten. Das ist damit zu begründen, dass nur in den Routinen, die der Kodierer benutzt, Code-Strukturen gefunden wurden, die durch Multimedia-Befehle optimiert werden konnten. Die ersten sechs Balken spiegeln somit Enkodiervorgänge des G.723.1, die weiteren fünf Balken Dekodiervorgänge wieder. Zu beachten ist, dass trotz geringer Einsatzmöglichkeiten von Multimedia-Befehlen eine Laufzeitverbesserung bis zu 13,9% erreicht werden konnte.

6.2 GSM

GSM bedeutet „Global System for Mobile Communication“. Dieses System wurde 1991 eingeführt, als die europäische Konferenz der Post- und Fernmeldeverwaltungen (CEPT[1]) die Normierung des europäischen Funktelefonsystems im 900 Mhz-Bereich verfolgt hat. Bereits Ende 1997 stand dieses System in über 100 Ländern zur Verfügung. Zu diesem Zeitpunkt bildete GSM den Standard in Europa und Asien. GSM bietet eine Reihe von Diensten an, die sich in Teledienste und Trägerdienste spalten. Unter den Telediensten verstehen wir beispielsweise die Telefonie und den Kurzmitteilungsdienst (SMS). Diese Art der Dienstleistung unterstützt jegliche Art von Diensten, die durch das verwendete Schichtenmodell zur Verfügung gestellt wird. Zu nennen ist an dieser Stelle das OSI-Referenzmodell der ISO, das durch GSM benutzt wird. Dieses Schichtenmodell besteht aus sieben Schichten, die unterschiedliche Dienste zur Verfügung stellen. Man kann transportorientierte und anwendungsorientierte Schichten unterscheiden. Schichten 1 bis 4 gehören zu den transportorientierten und Schichten 5 bis 7 zu den anwendungsorientierten Elementen. Trägerdienste sind allein für den Transport von Daten zuständig. Der Anwender kann diesen Dienst für eine Kommunikationsübertragung zu seinem Gesprächspartner nutzen. Der GSM-Standard kann Daten mit einer maximalen Übertragungsgeschwindigkeit von 9600 Bit/s übertragen. Dies ist für Text völlig ausreichend. Um jedoch multime-

[1]CEPT - Conférence Européenne des Administration des Postes

diale Inhalte zu versenden, bedarf es einer wesentlich höheren Übertragungsrate.

6.2.1 Saturierende Arithmetik

Auch in den Quellen der GSM-Referenzimplementierung finden sich Strukturen der saturierenden Arithmetik wieder. Abbildung 6-8 zeigt die Performance-Steigerung, nachdem diese Strukturen durch Saturierungs-Intrinsics optimiert worden sind. Mit einer Laufzeitverbesserung von 48% befindet sich diese Verbesserung an der Spitze der in dieser Arbeit vorgestellten Optimierungen. In den Quellen wurden Strukturen für die Addition, Subtraktion, Multiplikation zuzüglich Linksverschiebung und für das allgemeine Linksverschieben von 32 Bit-Zahlen optimiert.

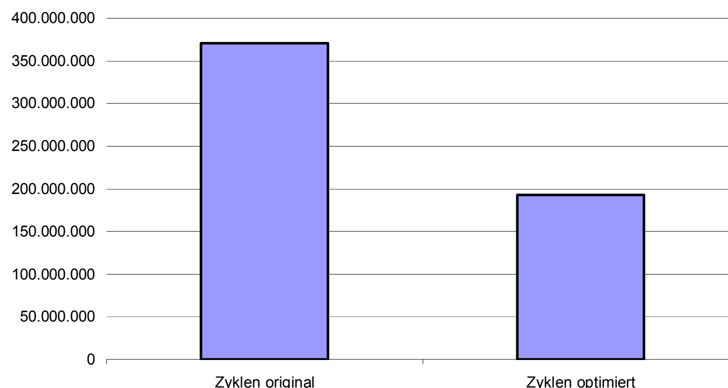


Abbildung 6-8: Laufzeit des GSM-Encoders für saturierende Arithmetik

Die horizontale Achse zeigt die beiden Durchläufe, die für den GSM bearbeitet worden sind. Zum einen sieht man die Laufzeit des unveränderten Quellcodes der Referenzimplementierung, welche sich auf 371.029.013 Zyklen beläuft. Nach dem Einsatz der Saturierungs-Intrinsics wurde eine gesamte Laufzeit von 192.935.086 Zyklen gemessen, wie in dem rechten Balken von Abbildung 6-8 dargestellt. Aus diesen Messwerten ergibt sich die oben genannte Verbesserung von 48%.

6.2.2 Bitbreiten-Reduktion

Dieses Kapitel spiegelt die Ergebnisse wider, die durch die Anwendung der Bitbreitenreduktion auf die GSM-Referenzimplementierung erreicht werden konn-

ten. Auch an dieser Stelle findet sich das gleiche Verfahren wieder, wie es bereits in Kapitel 5-2 angewendet worden ist. Abbildung 6-9 zeigt die jeweiligen Ergebnisse aufgeteilt in sechs Balken.

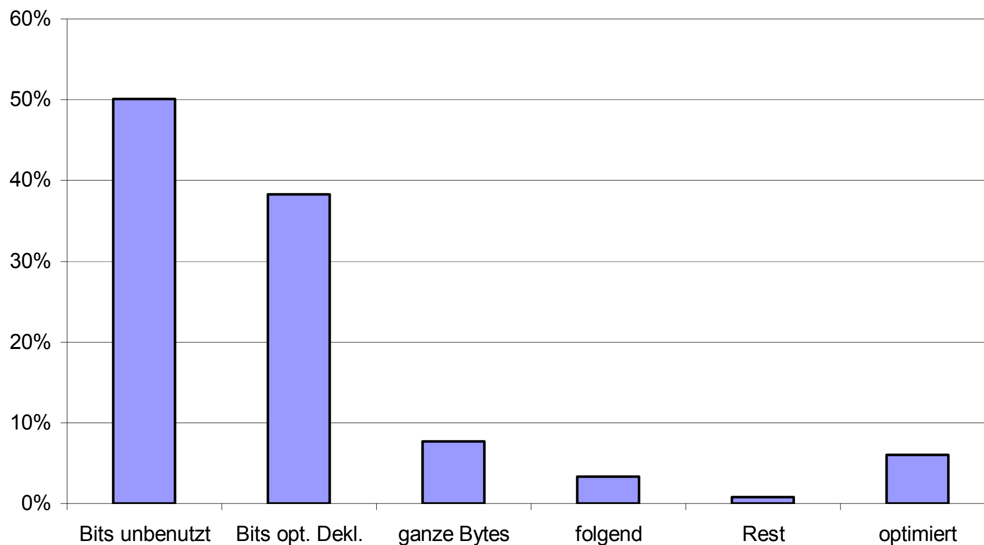


Abbildung 6-9: Bitreduktionen in % der GSM-Implementierung

Die Struktur von Abbildung 6-3 ist identisch mit der von Abbildung 6-9, so dass hier auf die Erläuterung der einzelnen Rubriken des Diagramms verzichtet wird. Das Potential zur Bit-Reduktion ist im GSM-Benchmark wesentlich höher als in der G.723.1 Software. So ist zu beobachten, dass 7,7% der benutzten Bits überflüssig und in ganzen Bytes angeordnet sind. Im Gegensatz zum G.723.1 kommen hier die Restriktionen bzgl. Funktionsaufrufen nicht so sehr zum Tragen, so dass von diesen 7,7% ganzer überflüssiger Bytes auch nahezu 78% tatsächlich optimiert werden konnten. In der Kategorie der auf ganze Bytes folgenden überflüssigen Bits, welche selbstverständlich nicht optimiert werden können, wurden 3,3% aller Bits gefunden. Die fünfte Reihe zeigt die Anzahl der Bits, die in den restlichen Inhalten der Variablen gefunden wurden und bildet 0,8%. Festzuhalten ist allerdings, dass ein großer Teil der erkannten Bits, die ganze Bytes bilden, optimiert werden konnten. Insgesamt wurden 6% aller im GSM-Benchmark vorkommenden Bits auch tatsächlich im Source-Code reduziert, wie durch den letzten Balken von Abbildung 6-9 dargestellt. Um auch an dieser Stelle einen Überblick über die optimierten Variablenarten geben zu können, zeigt Abbildung 6-10 die Analyse bezüglich der Bitreduktion, die auf die GSM Referenzimplementierung angewendet worden ist.

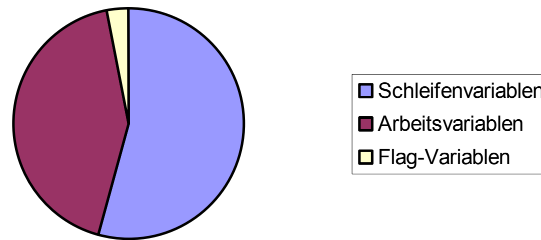


Abbildung 6-10: Anteil in % der optimierten Variablenarten im GSM

In Abbildung 6-10 sieht man, dass die in dieser Arbeit angewendete Bitreduktion hauptsächlich Laufvariablen in Schleifen optimieren konnte. Sie nehmen einen Anteil von 54% ein. Der zweitgrößte Anteil von 43% gibt den Anteil der optimierten Arbeitsvariablen an. Den kleinsten Anteil haben die Flag-Variablen mit 3% zu verzeichnen.

Im Vergleich zu den Ergebnissen, die Budiu und Goldstein in [BG00] darstellen, ist zu beobachten, dass in dieser Arbeit ca. 17% mehr an potentiell überflüssigen Bits gefunden worden sind. Es ist allerdings zu bemerken, dass der prozentuale Anteil der durch die Deklarationen eingesparten Bits bei den Quellen, die diese Arbeit benutzt, erheblich höher liegen als in den Quellen, die in der Arbeit von Budiu und Goldstein verwendet werden. So stehen 38,3% in dieser Arbeit den ca. 13% in [BG00] gegenüber. Dies lässt darauf schließen, dass unterschiedlichen Quellcodes verwendet worden sind.

6.2.3 Multimedia-Befehle

Auch bei diesem Benchmark wurde die Optimierung bezüglich der parallelen Verarbeitung durch Multimedia-Befehle durchgeführt. Insgesamt konnten 42 verschiedene Code-Strukturen aufgefunden und optimiert werden. Zu bemerken ist allerdings wiederum, dass nach der durchgeführten Bitreduktion keine weiteren Code-Fragmente der GSM-Implementierung durch SIMDs optimiert werden konnten. Durch den Einsatz der 42 eingefügten Multimedia-Befehle wurde jedoch eine Laufzeitverbesserung von 16,5% erzielt. Abbildung 6-11 zeigt die Optimierungen in einer graphischen Darstellung.

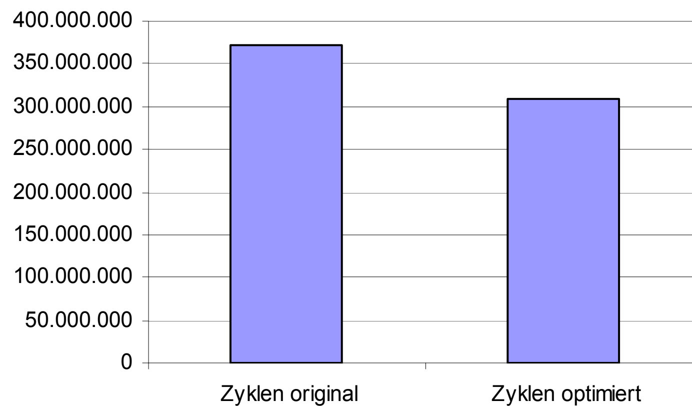


Abbildung 6-11: Laufzeit nach SIMD-Optimierung des GSM-Verfahrens

Ursprünglich wurde eine Laufzeit des GSM-Encoders von 371.029.013 Zyklen gemessen. Nach dem Einsetzen der gefundenen SIMD-Intrinsics sank die Laufzeit auf 309.809.225 Zyklen.

6.3 ADPCM – G.721

Das ADPCM-Verfahren ist eine Pulsecodemanipulation mit einem Vorhersagemechanismus. Die Pulsecodemanipulation ist ein Standard der ITU[1] [ITU05] und ist für Sprachdigitalisierung zuständig. Bei dieser Art der Digitalisierung werden zeitdiskrete analoge Signale durch eine Quantisierung in zeit- und wertdiskrete Werte umgewandelt. Die Quantisierung bildet einen Teil der Digitalisierung von analogen Signalen, die in regelmäßigen Abständen abgetastet werden. Zu jedem dieser Abtastpunkte wird der zugehörige Spannungswert in ein digitales Signal umgewandelt. In diesem Fall wird das Signal 8000-mal in der Sekunde abgetastet und in 8-Bit-Werte umgewandelt. Die resultierende Übertragungsgeschwindigkeit beträgt 64 kbit/s. Zwei Verfahren wurden von der ITU definiert: das µ-Law- und das A-Law-Verfahren. Das A-Law-Verfahren ist ein Verfahren für die Dynamikkompression von Audiosignalen. Die Dynamikkompression dient zur Verbesserung des Signal-Rausch-Verhältnisses. Dieses Verfahren wird hauptsächlich in Europa eingesetzt. In den USA und Japan dagegen wird meist das µ-LAW-Verfahren verwendet. Dieses Verfahren arbeitet ähnlich dem A-Law-Verfahren. Es benutzt allerdings andere Quantisierungsstufen. Die Pulsecodemanipulation

[1] ITU – International Telecommunication Union

versucht, die mögliche Signalform zu ermitteln und bildet eine Differenz mit dem tatsächlichen Signal. Da die Differenz dementsprechend kleiner ist, kann diese Differenz mit einer kleineren Bitzahl kodiert werden. Das vorhergesagte Signal wird kontinuierlich an das tatsächliche Signal angepasst. Dieses Verfahren ist verlustbehaftet, allerdings werden die Daten von Musik und Sprache auf 50% reduziert. 1986 wurde das ADPCM-Verfahren als ITU-Standard G.721 spezifiziert. Aufgrund gänzlich verschiedener Quellcodes werden hier die Benchmarks zwischen ADPCM und G.721 unterschieden. Sowohl ADPCM als auch G.721 sind fester Bestandteil der Mediabench Benchmark-Suite [LPM97]. Die in diesem Kapitel optimierten Quellcodes wurden Mediabench entnommen.

Da beide Benchmarks keine Strukturen zur saturierenden Arithmetik enthalten, und auch keine SIMD-Befehle eingesetzt werden konnten, um parallele Berechnungen zu ermöglichen, beläuft sich das Ergebnis lediglich auf der Darstellung der reduzierten Bitbreiten.

6.3.1 Bitbreitenreduktion - ADPCM

Zunächst behandelt dieses Kapitel die Bitbreitenreduktion des ADPCM-Verfahrens. Abbildung 6-12 zeigt die erreichten Resultate im Detail.

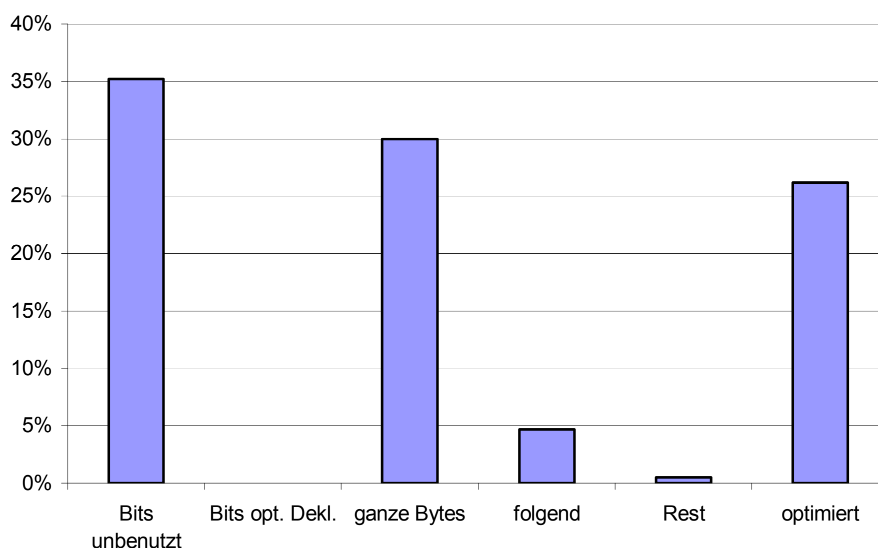


Abbildung 6-12: Bitbreitenreduktion des ADPCM-Verfahrens

Die Struktur von Abbildung 6-12 ist identisch mit den vorherigen Abbildungen 6-3 und 6-9, so dass hier ebenfalls auf die Erläuterung des Diagrammaufbaus ver-

zichtet wird. Zu beobachten ist bei diesem Benchmark, dass in der Implementierung dieses Verfahrens für die untersuchten Variablen stets der Datentyp „integer“ benutzt wurde. Daher wurden durch die Deklaration keinerlei Bits eingespart. Das hat zur Folge, dass ein großer prozentualer Anteil der gefundenen potentiell überflüssigen Bits auch optimiert werden konnten. 35,2% der gesamten Bits wurden als potentiell überflüssig eingestuft, wobei 30% der gesamten Bits in ganzen Bytes gefunden wurden. Bei dieser Implementierung ist zu erkennen, dass durch die fehlende Einsparung von Bits durch deren Deklaration von möglichst kleinen Datentypen, der prozentuale Anteil der optimierten Bits hoch ist. 26,2% der gesamten Bits konnten demnach optimiert werden. Die auf ganze Bytes folgenden Bits verzeichnen bei diesem Benchmark einen prozentualen Anteil von 4,7%, wobei die restlich gefundenen Bits einen Anteil von 0,5% ausmachen. Zur besseren Übersicht zeigt Abbildung 6-13 den prozentualen Anteil der optimierten Variablenarten.

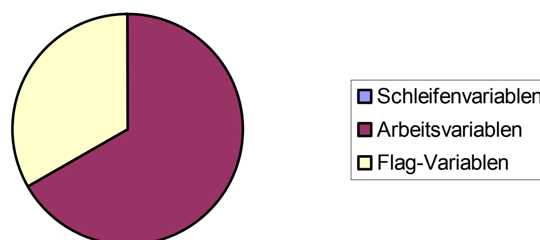


Abbildung 6-13: Anteil in % der optimierten Variablenarten im ADPCM

Wie zu erkennen ist, wurden in dem Quellcode dieses Verfahrens keinerlei Schleifenvariablen optimiert. Den größten Anteil der optimierten Variablenarten nehmen demnach die verwendeten Arbeitsvariablen mit 67% ein. Die restlichen 33% geben den Anteil der optimierten Flag-Variablen an.

Im Vergleich mit den Ergebnissen, die Budiu und Goldstein in [BG00] erreicht haben, ist zu beobachten, dass in dieser Arbeit 5,2% mehr potentiell überflüssige Bits in den Quellen des ADPCM gefunden worden sind. Allerdings ist zu bemerken, dass in den hier untersuchten Quellcodes alle untersuchten Variablen mit dem Datentyp „integer“ deklariert worden sind. Demnach ergeben sich, anders als in den Ergebnissen, die Budiu und Goldstein darstellen, keinerlei unbenutzte Bits, die bereits durch die Deklaration der Variablen eingespart worden sind.

Daraus ist zu schließen, dass die untersuchten Quellcodes ebenfalls nicht identisch sind.

6.3.2 Bitbreitenreduktion – G.721

Beim letzten Benchmark handelt es sich um das G.721-Verfahren. Wie schon in den vorherigen Kapiteln wurde die in dieser Arbeit implementierte Bitbreitenreduktion auf die Quellen dieses Verfahrens angewendet. Abbildung 6-14 zeigt die detaillierten Resultate.

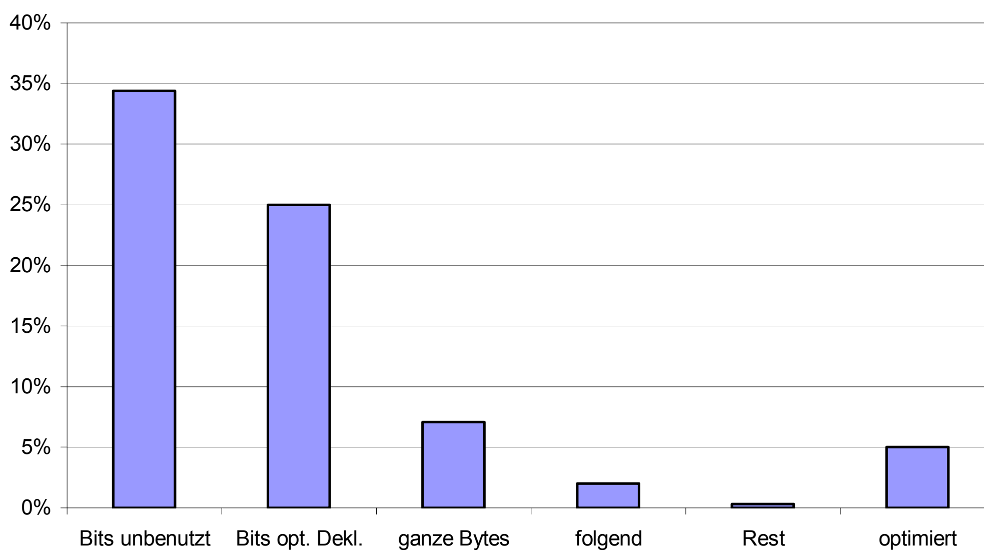


Abbildung 6-14: Bitbreitenreduktion des G.721-Verfahrens

Auch an dieser Stelle wird auf eine ausführliche Beschreibung der Diagrammstruktur verzichtet. Sie ist identisch mit den vorherigen Abbildungen 6-3, 6-9 und 6-12. Anders als bei der ADPCM-Implementierung wurden bei diesem Verfahren bereits durch die Deklaration der verwendeten Variablen 25% der zur Verfügung stehenden Bits eingespart. Wie zu erkennen ist, konnten 70% der potentiell überflüssigen Bits, die ganze Bytes darstellen, auch optimiert werden. Die auf ganze Bytes folgenden Bits tragen hier einen prozentualen Wert von 2%, wobei die restlichen Bits einen geringen Anteil von 0,3% bilden. Um auch an dieser Stelle einen besseren Überblick über die optimierten Variablenarten zu erhalten, zeigt Abbildung 6-15 deren prozentualen Anteil.

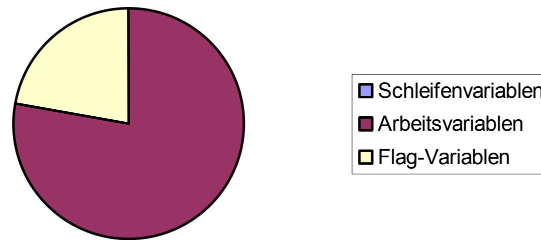


Abbildung 6-15: Anteil in % der optimierten Variablenarten im G.721

Auch bei diesem Verfahren nehmen die Arbeitsvariablen den größten prozentualen Anteil mit 77% ein. Die restlichen 23% geben den Anteil von optimierten Flag-Variablen an. Es wurden keine Schleifenvariablen optimiert.

Im Vergleich zu den Ergebnissen, die Budiu und Goldstein in [BG00] mit dem G.721 erreichen konnten, fällt wiederum auf, dass in den Quellen, die in dieser Arbeit untersucht worden sind, wesentlich mehr Bits durch die Deklaration der Variablen eingespart wurden. Zu den 25% hier entdeckter unbenutzter Bits, die durch Deklarationen eingespart worden sind, stehen ca. 8,5% in den Quellen, die Budiu und Goldstein benutzen. Weiterhin ist zu sagen, dass die hier erstellten Ergebnisse ca. 6% weniger potentiell überflüssige Bits zeigen als in den Arbeiten von Budiu und Goldstein. Auffällig ist auch der prozentuale Anteil der unbenutzen Bits, die in ganzen Bytes gefunden wurden. Budiu und Goldstein geben bei diesem Ergebnis einen prozentualen Wert von ca. 26% an. In dieser Arbeit wurden allerdings nur 7,1% gefunden.

7 Zusammenfassung und Ausblick

Dieses abschließende Kapitel gibt einen zusammenfassenden Überblick der in dieser Arbeit behandelten Thematiken. Kapitel 7.1 fasst noch einmal die behandelten Optimierungen und deren Ergebnisse zusammen. Kapitel 7.2 gibt einen Ausblick und zeigt weitere Optimierungsstrategien, die die Resultate dieser Arbeit noch verbessern könnten.

7.1 Zusammenfassung

Diese Diplomarbeit ist im Bereich der Eingebetteten Systeme angeordnet. Der Inhalt dieser Arbeit behandelt Quellcode-Optimierungen, die durch automatische Quellcode-Transformationen vorgenommen werden. Ein besonderes Augenmerk liegt auf dem Einsatz von Intrinsics, durch die eine erhebliche Performance-Steigerung erzielt wird. Weiterhin wird eine Optimierung zur Reduktion von Bitbreiten behandelt. Im Einzelnen wurden folgende Optimierungen für die Prozessorreihe C6x von Texas Instruments betrachtet:

- Optimierung von saturierender Arithmetik und Ersetzung dieser Strukturen durch Intrinsics.

- Bitbreitenreduktion mit Hilfe einer bitgenauen Datenflussanalyse.

- Einsatz von Multimedia-Befehlen, um parallele Berechnungen von Additionen und Subtraktionen durchführen zu können.

Durch die Optimierung von saturierender Arithmetik konnten Laufzeiteinsparungen von 33% bis 48% erzielt werden. Anhand dieser Zahlen sieht man die Mächtigkeit dieser „Compiler-known Functions“. Damit wurde bewiesen, dass die heutzutage eingesetzten Compiler nicht in der Lage sind, die entsprechenden Code-Strukturen zu erkennen und zu optimieren. Bisher war es immer noch nötig, dem Compiler unter die Arme zu greifen, und diese Optimierungen manuell in den Quellcode einzupflegen.

Weiterhin wurden durch den Einsatz einer bitgenauen Datenflussanalyse die tatsächlich notwendigen Bitbreiten von Variablen untersucht. Dabei wurden zwischen 20,5% und 50,1% der Bitbreiten als überflüssig erkannt.
Die dritte Optimierung beschäftigt sich mit der möglichen parallelen Bearbeitung von Additionen und Subtraktionen. Durch den Einsatz von Multimedia-Befehlen wurden Laufzeiteinsparungen bis zu 16,5% erreicht.
Die Performance-Steigerung der GSM-Referenzimplementierung durch Einsatz aller Optimierungstechniken beläuft sich auf 49%. Damit wurde die benötigte Laufzeit nahezu halbiert.

7.2 Ausblick

Die in dieser Diplomarbeit erstellten Optimierungen sind nicht immer voll ausgeschöpft worden. So beschränken sich die durchgeführten Quellcode-Transformationen auf einzelne Prozeduren. Diese Grenzen wurden nicht überschritten, und damit wird das mögliche Optimierungspotential nicht komplett ausgenutzt. Dieser Nachteil trifft bei der Bitbreitenreduktion zu. Es werden zwar potentiell überflüssige Bits erkannt, allerdings kann nur ein Bruchteil dieser Bits auch tatsächlich optimiert werden. Sobald eine Variable durch eine Parameterübergabe in oder aus der betrachteten Prozedur tritt, ist eine Optimierung nicht möglich, ohne die zusammenhängenden Funktionen ebenfalls zu überprüfen. Um den Anteil der tatsächlich optimierten überflüssigen Bits zu erhöhen, ist die gezielte Anwendung von „Function Inlining“ zur Vorbereitung der Bitreduktion denkbar. Die Bitbreitenreduktion behandelt lediglich Integer, Short- und Char-Variablen. Eine Unterstützung von Arrays und Pointern wurde in dieser Arbeit nicht miteinbezogen. Die Integration einer bitgenauen Array-Datenflussanalyse könnte zu noch größerem Einsparungspotential führen.
Weiterhin wurden aufgrund der Komplexität nicht alle Intrinsics betrachtet. Es stehen beispielsweise neben den Multimedia-Befehlen zur parallelen Berechnung von zwei 16 Bit-Additionen oder -Subtraktionen auch Befehle zur Verfügung, die vier 8 Bit-Operationen parallel ausführen können. Außerdem wurden keine vorläufigen Optimierungen durchgeführt, die den Einsatz von SIMD-Befehlen unterstützen. So wäre es beispielsweise denkbar, dass zuvor ein Abrollen der Schleifen durchgeführt wird, um eine größere Anzahl von parallelen Bearbeitungen einsetzen zu können. Wie in Kapitel 5 beschrieben, wurde ebenfalls kein Algorithmus entwickelt, der die Ersetzung von Instruktionen in verschiedenen „Scopes“ berechnet.

Literaturverzeichnis

[Fal04] Heiko Falk. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software.* Dissertation, Universität Dortmund, Dortmund, Juni 2004

[WZM96] M. Willems, V. Zivonjovic und H. Meyr. DSP-Compiler: Produktqualität für kontrolldominierte Anwendungen, In *DSP Deutschland ´96*, Seiten 49-56, 1996

[LM97] Rainer Leupers und Peter Marwedel. Optimierende Compiler für DSPs: Was ist verfügbar?, In *DSP Deutschland ´97*, 1997

[KR88] Brian W. Kernighan und Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1988

[GSM] Jörg Eberspächer, Hans-Jörg Vögel und Christian Bettstetter. *GSM, Global System for Mobile Communication.* Januar 2001

[G7231] *ITU-T Recommendation G.723.1 – Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbit/s*, März 1996

[Dil00] Thomas J. Dillon, Jr. Texas Instruments Application Report: G.723.1 Dual-Rate Speech Coder: Multichannel TMS320C62x Implementation, In *Electronic Engineering Times*, Februar 2000

[ALD02] Tom Vander Aa, Rudy Lauwereins und Geert Deconinck. Optimizing a 3D Image Reconstruction Algorithm: Analyzing the Capabilities of a Modern Compiler, In *IEEE Workshop on Signal Processing Systems Design and Implementation*, Seiten 246-251, San Diego, Oktober 2002

[PBSB02] Pokam, Bihan, Simonnet und Bodin: *SWARP: A Retargetable Preprocessor for Multimedia Instructions*, IRISA Campus Universitaire de Beaulieu, Rennes Cedex, Frankreich, 2002

[Jak02] Jacek Jakubowski. Architekturunabhängige Quellcodeoptimierung durch Mustererkennung, Diplomarbeit. Universität Dortmund, Dortmund, April 2002

[BG00] Mihai Budiu und Seth Copen Goldstein. *Bit Value Interference: Detecting and Exploiting Narrow Bitwidth Computations*, CMU-CS-00-141, School of Computer Science, Carnegie Mellon University, Pittsburgh, Oktober 2000

[CWK99] Martin Coors, Oliver Wahlen, Holger Keding, Olaf Lüthje und Heinrich Meyr, TI C62x Performance Code Optimization. In *DSP Deutschland ´99*, Seiten. 155-164, 1999

[SPR197] *TMS320C6000 Technical Brief.* Texas Instruments Inc., Literature Number SPRU197D, Februar 1999

[WL01] Jens Wagner and Rainer Leupers. Compiler Design for a Network Processor. In *IEEE Transactions on CAD*, 2001

[SIMD05] Simdtech Homepage. http://www.simdtech.org, 2005

[ICD05] Informatik Centrum Dortmund – ICD Homepage. http://www.icd.de, 2005

[SCG94] The Stanford Compiler Group, SUIF Compiler System, http://suif.stanford.edu, 1994

[ITU05] ITU Homepage. http://www.itu.int\home, 2005

[TSL94] *The SUIF Library – A Set of core routines for manipulating SUIF data structures*. Stanford Compiler Group, 1994.

[LPM97] C.Lee, M. Potkonjak und W.H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, Seiten 330-335, 1997