Diplomarbeit

Codegenerierung für den digitalen Signalprozessor TI TMS320C5x

Thomas Barschdorf

Achtermannstr. 8
44866 Bochum

Informatik LS12
Universität Dortmund

Betreuer:

Dr. Rainer Leupers

Prof. Peter Marwedel

Inhaltsverzeichnis

Kapitel 1

Einleitung

Elektronische Schaltungen werden zunehmend kleiner. Die Möglichkeiten, Schaltungen einzusetzen, nehmen stetig zu. In sehr vielen Bereichen des täglichen Lebens haben elektronische Schaltungen bereits Einzug gehalten, ohne daß man sie überhaupt bemerkt. Häufig werden diese Schaltungen innerhalb von geschlossenen Systemen eingesetzt. Sie übernehmen dann die Steuerung der Systeme, d.h. sie reagieren auf externe Einflüsse und ändern selbständig den Zustand des Systems. Zum Beispiel wären Mobiltelefone ohne solche *eingebetteten Prozessoren* gar nicht mehr denkbar. Eingebettet bedeutet, daß die Schaltungen fest in das System integriert sind. Es gibt diverse elektronische Bausteine, die für solche Aufgaben eingesetzt werden können. Hier einige Beispiele:

ASIC's[1] , FPGA's[2], DSP's[3], μ-Controller, Standard-CPU's

Die Vorteile von DSP's gegenüber anderen Schaltungs-Typen sind auf der einen Seite die geringen Kosten für die Entwicklung. Man kann auf eine geringe Anzahl von Standardprozessoren zurückgreifen, die nur entsprechend der jeweiligen Aufgabe programmiert werden müssen. Kosten für ein neues Design fallen nicht an, wie z.B. bei ASIC's, die den Anwendungen entsprechend designt und produziert werden müssen. Auf der anderen Seite sind DSP's sehr sparsam, was den Verbrauch der Chipfläche und Energie angeht. Sie erlauben außerdem sehr hohe Geschwindigkeiten. Zum Beispiel FPGA's benötigen, auf Grund ihrer Struktur, sehr große Chipflächen und erlauben im Gegensatz zu DSP's keine kurzen Taktzyklen.

Die hohe Flexibilität von DSP's, die gute Programmierbarkeit sowie die hohe Zuverlässigkeit eröffnen weite Einsatzmöglichkeiten, besonders im Bereich Multimedia, Telekommunikation. Aber ebenso findet man DSP's in digitalen Systemen für Autos,

[1] ASIC: application specific integrated circuit
[2] FPGA: field programmable gate array
[3] DSP: digital signal prozessor

in der Medizintechnik, etc. Digitale Signalprozessoren werden besonders häufig dort eingesetzt, wo es darauf ankommt, Echtzeitbedingungen einzuhalten, in digitalen Filtern zum Beispiel.

Ein großes Defizit der DSP-Entwicklung ist jedoch die noch unzureichende Unterstützung durch effiziente Werkzeuge, etwa Compiler für stark verbreitete Hochsprachen, wie C oder C++. Bestehende Compiler für DSP's nehmen in der Regel keine Rücksicht auf die meist heterogenen Architekturen von DSP's. DSP's verfügen häufig über mehrere funktionale Einheiten, die verschiedene Operationen parallel durchführen können. Die angebotenen Compiler können aber daraus normalerweise keinen Nutzen ziehen, da sie nicht für die Erzeugung von parallelem Code geeignet sind.

Spezielle Eigenschaften von DSP's werden von existierenden Compilern ebenfalls nicht genutzt, z.B. zero-overhead-loops. DSP's sind häufig in der Lage, Codesequenzen ohne zusätzliche Befehle auszuführen, sie benötigen nur den Codebereich, der die Schleife enthält und die Anzahl der Durchläufe. "Normale" Compiler erzeugen aber speziell bei Schleifen und anderen Kontrollstrukturen sehr viel Overhead. In einer Studie des DSPstone-Projekts der TU Aachen [12] wurden C-Compiler für verschiedene DSP's[4] auf deren Codequalität überprüft. Dabei wurde der generierte Code mit manuell erstelltem Assemblercode verglichen. Die ermittelten Ergebnisse zeigen auf, daß der generierte Code um ein Vielfaches mehr Recources verwendet, als der manuell generierte Code. Z.B. die Ausführungszeit der Programme ist bei allen DSP's mindestes 5 Mal länger. Aber auch der Speicherverbrauch und die Programmlänge ist bei dem generierten Code wesentlich höher.

DSP-Programme haben jedoch sehr hohe Anforderungen an die Codequalität, da der verfügbare Speicher bei DSP's sehr begrenzt ist und DSP's häufig in zeitkritischen Anwendungen eingesetzt werden. Um die Fläche auf dem Chip so gering wie möglich zu halten, sind extrem kompakte Programme notwendig. Normalerweise kann der On-Board-Speicher durch externen, und dadurch meist viel langsameren, Speicher erweitert werden, dennoch ist die maximale Größe des Speichers durch die Bus-Architektur des Prozessors eingeschränkt.

Ein großer Teil der Programme für DSP's wird heute noch in Assembler geschrieben, da die Codequalität bestehender Compiler den hohen Anforderungen von DSP's nicht genügt. Ein wesentlicher Nachteil von in Assembler geschriebenen Programmen ist jedoch, daß für jeden neuen DSP-Typ die Programme weitgehend neu geschrieben werden müssen, da sich die Architektur von DSP's bei verschiedenen Typen unterscheidet und somit andere Operationen und andere Register zur Verfügung stehen. Sourcen eines speziellen Typs sind daher nur eingeschränkt für andere Typen zu nutzen.

[4] Analog Device 2101, AT&T 1610, Motorola 56001, NEC 77016 und TI 320C51

Assemblerprogramme sehr schwer zu debuggen, d.h. Fehlersuche ist ein sehr zeitaufwendiger Faktor in der Programmerstellung. Ebenso wächst die der Umfang der eingesetzten Applikationen, so daß bei einer Assemblerprogrammierung die Übersicht schnell verloren gehen kann.

Zur Zeit wird sehr viel an Compilern gearbeitet, die auf Hochsprachen aufsetzen und dennoch auf die spezielle Architektur von DSP's Rücksicht nehmen. Damit kann die Entwicklungszeit von Programmen sehr stark verkürzt und eine entsprechende Codequalität garantiert werden.

Das Thema dieser Diplomarbeit ist die Codeerzeugung für eine Familie von DSP's der Firma Texas Instruments, TMS320C5x, im folgenden nur noch C5x genannt. Sie besteht aus 3 DSP-Typen,

- TMS320C50
- TMS320C51
- TMS320C53

die sich aber im wesentlichen nur in der Größe des zur Verfügung stehenden On-Board-Speichers unterscheiden. Sie können alle mit externem Speicher aufgerüstet werden. Details dazu sind in Kapitel 2 dargestellt.

Ziel dieser Diplomarbeit ist es, bestehende Verfahren zur Codegenerierung (Kapitel 4) an die spezielle Architektur (Kapitel 2) des C5x anzupassen und die Vorteile für die resultierende Codequalität aufzuzeigen. Dazu wird ein Compiler entwickelt, der auf dem C-Frontend LANCE (Kapitel 3.1) aufbaut.

Besonderes Interesse erhält dabei ein Register, das in früheren DSP-Familien von Texas Instruments nicht vorkam, ein schnelles Pufferregister für den Akkumulator. Dieses Register soll auf verschiedene Weisen in den Prozess der Codegenerierung eingefügt werden (Kapitel 5).

Für die Codegenerierung selbst kommt ein weiteres Tool zum Einsatz, OLIVE (Kapitel 3.2). OLIVE ist ein Generator für Codegeneratoren. Mit OLIVE kann die besondere Architektur des C5x sehr gut beschrieben werden und ein Codegenerator erzeugt werden, der bei der Instruktionsauswahl sehr effizient arbeitet (Kapitel 4.1).

Schließlich werden die Verfahren und deren Änderungen anhand verschiedener Beispiele getestet (Kapitel 6) und die Änderungen durch die spezielle Architektur des C5x aufgezeigt.

Kapitel 2

Architektur Modell des C5x

Im folgenden Kapitel wird die Hardwarearchitektur der DSP-Familie C5x beschrieben.

Vergleicht man die Architektur von general-purpose Prozessoren[1] mit der Architektur des C5x, erkennt man einige entscheidende Unterschiede in der Struktur. General-purpose Prozessoren haben in der Regel eine homogene Architektur, d.h. die Operanden einer Operation werden aus einem Registerfile[2] geladen und das Ergebnis wird in dasselbe Registerfile zurück geschrieben. Die Quellen für Operanden können also viele verschiedene Register sein, die allerdings in den Instruktionen mit kodiert werden müssen. Der Preis für diese Flexibilität sind breite Befehlsworte, da in jeder Instruktion einige Bits für diese Informationen benötigt werden. Breite Befehlsworte erfordern aber wiederum breite Datenbusse, wenn die Instruktionen effizient in den Prozessor geladen werden sollen. Breite Datenbusse benötigen jedoch viel Fläche auf dem Chip und verbrauchen ebenfalls viel Energie.

DSP's sind dagegen für wenige, spezielle Aufgaben konzipiert, die sie aber wesentlich effizienter ausführen können, als general-purpose Prozessoren. Man geht deshalb bei DSP's einen anderen Weg. Die Quellen für Operanden und die Register für Resultate der verschiedenen arithmetischen Einheiten werden schon beim Entwurf des DSP festgelegt, d.h. daß für eine Operation die Operanden *immer* aus denselben festgelegten Registern geladen und daß Ergebnisse *immer* in dasselbe festgelegte Register geschrieben werden. Ein Beispiel :

Für eine Multiplikation wird der erste Operand aus dem Register TREG0 geholt,

[1]general-purpose Prozessoren : Prozessoren, die für universelle Aufgaben verwendet werden, z.B. in Workstations. Diese Prozessoren sind normalerweise nicht für bestimmte Aufgaben besonders optimiert, sondern sind so konzipiert, daß sie für die verschiedensten Anwendungen eingesetzt werden können.

[2]Registerfile : Ein "Verbund" aus mehreren Hardwareregistern, die für gleiche Aufgaben verwendet werden, aber einzeln angesteuert werden können.

der Zweite aus dem Speicher. Das Ergebnis wird in das Produktregister PREG geschrieben (s. Kapitel 2.2).

Diese Art der Architektur nennt man heterogen. Die Befehlsworte sind kurz und kompakt, da man keine Quellen oder Ziele der Operationen kodieren muß.

Ein weiterer Unterschied zu general-purpose Prozessoren ist, daß häufig Daten- und Programmspeicher aus Geschwindigkeitsgründen mit auf dem Chip integriert werden. Der OnChip-Speicher besteht aus schnellem statischem Speicher, was den Zugriff erheblich beschleunigt. Der Nachteil hierbei ist jedoch, daß Programm und Daten in den zur Verfügung stehenden Speicher passen müssen, um daraus Nutzen ziehen zu können. Reichen OnChip-Daten- oder OnChip-Programmspeicher nicht aus, so muß auf wesentlich langsameren externen Speicher zurückgegriffen werden. Ein Grund mehr, für extrem kompakten Programmcode zu sorgen.

DSP's werden in zwei Klassen eingeteilt:

- Festkomma DSP (*Fixed Point DSP*) und
- Fließkomma DSP (*Floating Point DSP*)

Die Einteilung erfolgt auf Grund der Daten, mit denen die Prozessoren umgehen können. Fließkomma DSP's sind einfacher in der Handhabung, man muß sich nicht um irgendwelche Skalierungsoperationen kümmern. Andererseits sind Fließkomma DSP's wesentlich aufwendiger, was den benötigten Platzbedarf an Silizium und die Länge der Taktzyklen angeht, da Fließkomma-Operationen um einiges komplexer sind. Dies ist ein Grund, weshalb Festkomma DSPs sehr viel häufiger zum Einsatz kommen. Die Familie C5x besteht aus Festkomma-DSPs.

2.1 Überblick

Zunächst soll ein kurzer Überblick über die wichtigsten Hardware-Komponenten der C5x-Familie gegeben werden, die in den folgenden Abschnitten genauer erläutert werden.

Die DSP-Familie C5x besitzt, wie auch schon frühere Modelle von Texas Instruments, etwa der TMS320C25, eine erweiterte Havard-Architektur, die zur Erhöhung der Rechenleistung getrennte Speicherbusse für Programm und Daten enthält. Die beiden Busse sind jeweils 16-Bit breit und ermöglichen einen maximalen Speicherbereich von $2^{16} = 64$K 16-Bit Worten. Spezielle Instruktionen ermöglichen den Datentransfer zwischen Daten- und Programmspeicher.

Der Kern des C5x ist die CALU (*central arithmetic logic unit*, Kapitel 2.2), die ihrerseits aus folgenden Komponenten besteht:

ALU Die ALU (*arithmetic logical unit* - Arithmetisch-logische Einheit, Kapitel 2.2.1) führt arithmetische Operationen im 2er-Komplement und logische Operationen aus. Als Quelle für Operanden kommen der Datenspeicher, verschiedene Prozessorregister oder im Befehl kodierte Konstanten in Frage. Bei jeder Operation ist allerdings ein Operand immer der Akkumulator. Der Akkumulator ist ein Register, in das das Ergebnis einer ALU-Operation geschrieben wird. Ein weiteres Register der ALU ist der Akkumulator-Puffer.

Multiplizierer Der 16x16-Bit Multiplizierer (Kapitel 2.2.2) führt eine Multiplikation zweier 16-Bit Werte aus und speichert das 32-Bit Resultat im Produktregister (*PREG*). Als weiteres Register gehört das TREG0 zu dem Multiplizierer. Das TREG0 ist ein Register, das einen Operanden der Multiplikation enthält. Wie bei der ALU kann der zweite Operand aus dem Datenspeicher kommen oder im Befehlswort kodiert sein, bei verschiedene Multiply/Accumulate-Befehlen ist es auch möglich, eine Konstante im Befehlswort zu kodieren.

PLU Die PLU (*parallel logic unit* - parallele Logik-Einheit, Kapitel 2.5) kann verschiedene logische Funktionen durchführen, ohne den Inhalt des Akkumulators zu verändern.

Scaling shifter Drei Schieberegister (Kapitel 2.2.3) komplettieren die CALU. Mit diesen Shiftern können Operanden vor und nach Operationen skaliert werden.

Neben der CALU gibt es zwei weitere wesentliche Funktionsblöcke.

Kontrolleinheit Die Kontrolleinheit steuert den Programmablauf und den Betriebsmodus des Prozessors. Dazu gehören die Sprunglogik, der Programmzähler, ein Hardwarestack (Kapitel 2.4.1) und diverse Statusregister (Kapitel 2.4.3).

Hilfsregisterfile Das Hilfsregisterfile (Kapitel 2.3) besteht aus acht Hilfsregistern (*auxiliary register - AR*), die zur indirekten Adressierung und zur Speicherung von Zwischenergebnissen komplexer Berechnungen genutzt werden können. Außerdem gehört eine einfache Arithmetikeinheit dazu, mit der man den Inhalt dieser Register parallel zur Befehlsausführung in der CALU modifizieren kann.

2.2 Central Arithmetic Logic Unit

Die zentrale Arithmetik-Einheit, *CALU*, ist das Kernstück des C5x-Prozessors. In der CALU werden alle arithmetischen und logischen Operationen des C5x ausgeführt. Sie besteht im wesentlichen aus der 32-Bit ALU (Kapitel 2.2.1) und dem

16x16-Bit Multiplizierer (Kapitel 2.2.2). Ebenso gehören aber auch verschiedene Shifter (Kapitel 2.2.3), sowie einige Multiplexer und Kontrollregister zur CALU.

2.2.1 ALU und Akkumulator

Die ALU des C5x bietet zusammen mit dem Akkumulator (*ACCU*) eine breite Auswahl an arithmetischen und logischen Operationen, die zu einem sehr großen Teil in einem Taktzyklus ausgeführt werden können. Das Ergebnis *aller* Operationen der ALU wird an den Akkumulator übertragen, wo weitere Operationen, wie z.B. Scalierung durch einen Shifter (Kapitel 2.2.3) erfolgen können.

Wird der Inhalt des Akkumulators im Verlauf der Programmausführung mehrfach benötigt, gibt es die Möglichkeit, den Wert in einem Pufferregister (*ACCUB*) zu sichern. Der Zugriff auf das Pufferregister kann in der Regel schneller erfolgen, als wenn man den Wert erneut aus dem Speicher laden muß.

Die ALU ist eine Standardkomponente, die mit 32-Bit Daten arbeitet. Wie bei vielen DSP's üblich, bietet auch die ALU des C5x die Möglichkeit, alle Operationen in saturierter Arithmetik[3] durchzuführen. Außer arithmetischen Operationen, wie Addition oder Subtraktion, können in der ALU verschiedene boolesche Operationen, wie UND, ODER, NICHT ausgeführt werden.

Ein Operand für *alle* Operationen der ALU ist der Akkumulator, für den zweiten Operanden gibt es verschiedene Möglichkeiten.

1. Der Datenspeicher
2. Das Produktregister - PREG
3. Der Akkumulatorpuffer - ACCUB
4. Eine Konstante, die im Befehl selbst kodiert ist.

Um den Inhalt des 32-Bit breiten Akkumulators zu speichern, wird dieser in zwei 16-Bit Segmente, ACCU-High und ACCU-Low, aufgeteilt, die mit verschiedenen Instruktionen in den Datenspeicher geschrieben werden können.

Von allen Operationen der ALU werden folgende Statusbits beeinflußt:

Overflow-Bit (OV) Das Overflow-Bit zeigt einen Werte-Überlauf bei den Operationen der ALU an.

[3] Bei saturierter Arithmetik wird im Falle eines Überlaufs nicht das eigentliche Ergebnis, sondern die größte bzw. kleinste darstellbare Zahl im Akkumulator gespeichert.

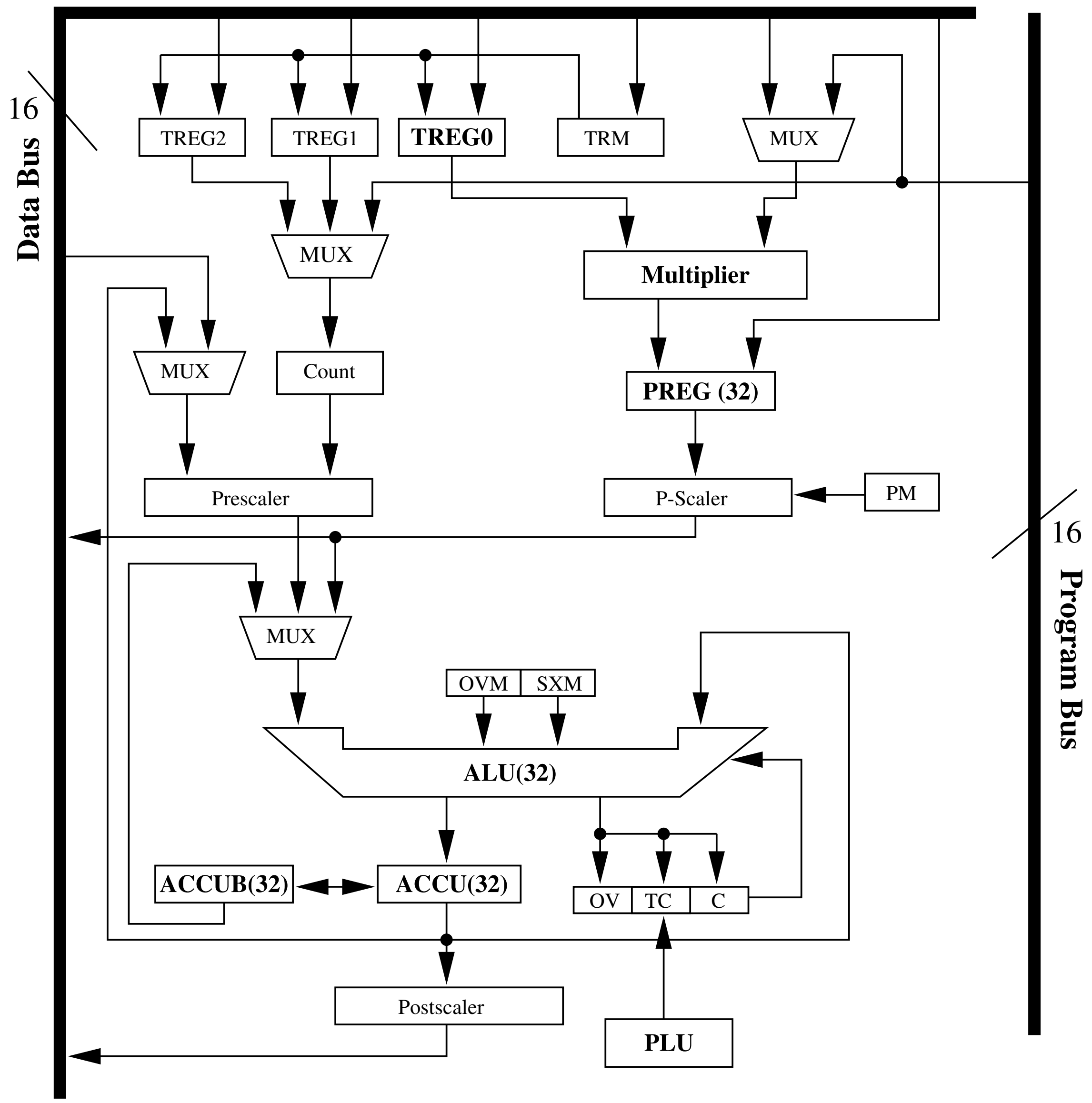

Abbildung 2.1: Blockschaltbild CALU

Carry-Bit (C) Das Carry-Bit zeigt an, ob bei einer Addition ein Übertrag entstanden ist, oder ob bei einer Subtraktion ein Unterlauf entstanden ist.

Test/Control-Bit (TC) Das Test/Control-Bit ist ein Flag, das das Resultat von Bit-Test Operationen enthält.

Diese drei Bits werden auch von Operationen der PLU (Kapitel 2.5) verändert. Auswirkungen haben diese Bits etwa bei bedingten Sprüngen oder bei einer folgenden Addition / Subtraktion.

2.2.2 Multiplizierer

Der C5x besitzt einen 16x16-Bit parallelen Multiplizierer, der ein 32-Bit Produkt in einem Taktzyklus berechnet. Der Multiplizierer besteht aus drei Komponenten:

1. das temporäre Multiplikationsregister TREG0,
2. das Produktregister PREG und
3. der 16x16-Bit Hardware-Multiplizierer.

Bis auf den Befehl MPYU (*multiply unsigned*) arbeiten alle Multiplikations-Befehle mit vorzeichenbehafteten Werten in 2er-Komplement Darstellung.

Jede Multiplikation erwartet den ersten Operanden im temporären Multiplikationsregister TREG0. Mit dem Befehl LT (*load treg0*) wird dieses 16 Bit breite Register mit einem Wert aus dem Speicher geladen. Der zweite Operand für die Multiplikation wird ebenfalls aus dem Datenspeicher geholt oder ist, wie z.B. bei der Addition, direkt im Befehl als Konstante kodiert.

Das Ergebnis der Multiplikation wird im Produktregister (PREG) abgelegt. Von dort kann es entweder in den Speicher geschrieben werden oder es dient als Eingabe für die ALU. Soll der Inhalt des PREG gespeichert werden, wird dieses 32-Bit breite Register, ähnlich dem Akkumulator, in zwei 16-Bit Segmente geteilt, die mit eigenen Befehlen gesichert werden.

2.2.3 Shifter

Die CALU des C5x enthält mehrere Shifter-Bausteine, die hauptsächlich für Skalierungsoperationen vor oder nach der Instruktionsausführung benutzt werden können. Da es sich beim C5x um einen Festkomma-DSP handelt, muß man zusätzlich Skalierungsoperationen benutzen, wenn man mit Fließkommazahlen arbeiten will.

Es gibt 3 Shifter (s. Abbildung 2.1):

Prescaler Der Prescaler sitzt vor dem Eingang der ALU und dient dazu, Daten zu skalieren, bevor sie in die ALU gelangen. Es ist in der Lage, Links-Shifts von bis zu 16 Bit durchzuführen. Die Anzahl der Bits ist als Konstante in der jeweiligen Instruktion kodiert oder wird aus TREG1 geholt.

P-Scaler Der P-Scaler, oder auch Productscaler, befindet sich am Ausgang des Produktregisters und kann, in Abhängigkeit von PM, vier verschiedene Shift-Operationen ausführen:

PM	Operation
00	kein shift
01	links shift um 1 Bit
10	links shift um 4 Bits
11	rechts shift um 6 Bits

Postscaler Den Postscaler findet man am Ausgang des Akkumulators und wird benutzt, wenn der Inhalt des ACCU's in den Speicher geschrieben werden soll. Er kann Daten um bis zu 7 Bits nach links schiften.

Alle verwendeten Shifter sind Barrelshifter, so daß das Ergebnis der Schiebeoperationen nach einem Takt verfügbar ist.

2.3 Hilfsregisterfile

Der C5x verfügt über 8 Hilfsregister (AR - *auxiliary register*), für die indirekte Adressierung, sie können aber auch zur temporären Speicherung von Daten genutzt werden. Diese Register sind zu einem Block zusammengefaßt und das jeweils benötigte Register wird über einen Zeiger, den ARP (*auxiliar register pointer*, s. Abbildung 2.2), indiziert.

Die Register des Hilfsregisterfiles können auf verschiedene Arten mit Daten geladen werden, entweder aus dem Datenspeicher, aus dem Akkumulator oder mit einer beliebigen Konstanten. Außerdem kann der Inhalt des aktuell ausgewählten Registers parallel zur Befehlsausführung modifiziert werden. Dazu besitzt das Hilfsregisterfile eine eigene einfache arithmetische Einheit, ARAU (*auxiliary register arithmetic unit*). Die möglichen Operationen sind das automatische inkrementieren bzw. dekrementieren des im letzten Befehl verwendeten Registers, sowie die Addition bzw. Subtraktion mit dem Inhalt des Indexregisters (*INDX*). Es gibt auch die Möglichkeit, den Inhalt der AR's mit einem eigenen Befehl zu verändern. In diesem Fall

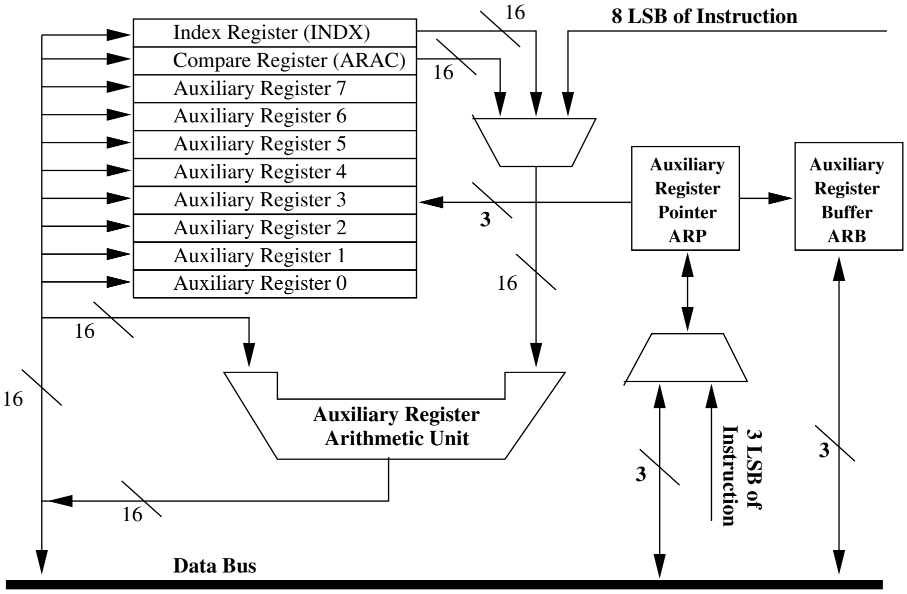

Abbildung 2.2: Blockschaltbild Hilfsregisterfile und ARAU

wird eine Konstante, die im Befehl kodiert ist, zum Inhalt des aktuellen Hilfsregisters addiert (*ADRK*) oder subtrahiert (*SBRK*).

Das Verändern eines Registers im Registerfile dauert 2 Taktzyklen[4], ebenso wie das Laden eines Hilfsregisters mit einem Wert aus dem Speicher, so daß der Inhalt des veränderten Registers erst wieder für den übernächsten Befehl einen gültigen Wert enthält.

Ebenso wie die AR kann auch der ARP parallel zur Ausführung einer Instruktion der ALU verändert werden. Dazu wird in der aktuellen Instruktion angegeben, welches Hilfsregister beim nächsten Befehl benutzt werden soll.

2.4 Systemkontrolle

Der Programmablauf und der Betriebsmodus des C5x wird vom Programmzähler (PC - *programm counter*), einem 8 Bit tiefen Hardwarestack, diversen Statusregistern, Interrupts sowie externen Signalen kontrolliert. Um die Programmabarbeitung zu beschleunigen, besitzt der C5x außerdem eine vierstufige Pipeline.

[4]Befehle, die das veränderte Register vorher verwenden, erhalten auf Grund der Pipeline (Kapitel 2.4.2) einen ungültigen Wert.

2.4.1 Programmzähler, Stack und Sprunglogik

Der Programmzähler ist ein Zeiger für den aktuell zu bearbeitenden Befehl im Programmspeicher. Der PC wird, nachdem ein Befehl aus dem Speicher gelesen wurde, modifiziert und zeigt auf den nachfolgenden Befehl im Programmcode. Der aktuelle Befehl wird in das Befehlsregister (*instruction register - IREG*) geladen.

Der Wert des Programmzählers kann auf verschiedene Arten verändert werden. Solange keine Programmsprünge oder Unterprogrammaufrufe ausgeführt werden, wird der Inhalt des PC um 1 erhöht, um die Adresse des folgenden Befehls zu erhalten. Sollte der aktuelle Befehl jedoch eine 16-Bit Konstante benutzen (Kapitel 2.6.4), muß der PC um 2 erhöht werden, da die Konstante im Programmspeicher direkt hinter dem Befehl steht.

Im Falle eines unbedingten Sprunges wird der PC mit einer Adresse geladen, die direkt auf den Sprungbefehl folgt, das Programm wird dann an der neuen Stelle fortgesetzt, die Pipeline des C5x (Kapitel 2.4.2) wird bei einem Sprung ungültig.

Bei Unterprogrammaufrufen wird der 8 Bit tiefe Hardwarestack benutzt, um die Adresse des auf den Unterprogrammaufruf folgenden Befehls zu speichern. Danach wird der Programmzähler mit der Adresse des Unterprogramms geladen. Nach Abarbeitung des Unterprogramms wird der PC wieder mit der gespeicherten Adresse vom Stack geladen und das Programm an der ursprünglichen Stelle fortgesetzt.

2.4.2 Pipeline

Der C5x besitzt eine vierstufige Befehlspipeline mit den Phasen :

- Instruction fetch - Laden des Befehls ins Befehlsregister
- Instruction decode - Dekodieren des Befehls
- Operand fetch - Laden des benötigten Operanden
- Execute - Ausführung des Befehls

Es befinden sich also immer 4 Befehle in der Pipeline, jeweils in einer anderen Phase der Befehlsausführung. Die Pipeline ist normalerweise für den Programmierer transparent, es gibt allerdings einige Fälle, wo der Programmierer diese Pipeline beachten muß:

- wenn auf Register des DSP-Kern's zugegriffen wird, die in den normalen Speicher eingeblendet werden oder

- wenn Inhalte der Hilfsregister verändert werden (Kapitel 2.3).

In diesen Fällen werden Änderungen an den Daten erst in der Execute-Phase übernommen. D.h. für Befehle, die bereits in der Phase Decode oder Operand-fetch sind, sind die vorher gelesenen Werte nicht mehr gültig. Soll bei aufeinander folgenden Befehlen auf dieselben Register zugegriffen werden, muß man, wenn der Inhalt des Registers verändert wurde, ein oder zwei Befehle einfügen, die nichts tun, NOP (*no operation*).

2.4.3 Statusregister

Der Programmablauf und die verschiedenen Betriebsmodi einzelner Hardwarekomponenten des C5x werden durch diverse Status- und Kontrollbits gesteuert, die in 4 besonderen Registern zusammengefaßt sind. Hier befinden sich z.B. das Carry- und Overflow-Bit (Kapitel 2.2.1), der Zeiger für das Hilfsregisterfile ARP, der Zeiger für die aktuelle Speicherseite DP, usw. Ebenso finden sich hier die Informationen, welche Hilfsregister den Ringpuffern (Kapitel2.6.6) zugeordnet sind und ob die Ringpuffer aktiv sind oder nicht.

2.5 Parallele Logik Einheit (PLU)

Die PLU dient dazu, einzelne Bits in diversen Registern oder im Datenspeicher zu testen oder zu verändern, ohne den Inhalt des Akkumulators zu beeinflussen. Die PLU führt eine Read-Modify-Write Operation aus, d.h. der Operand wird vom Speicherbus gelesen, entsprechend der jeweiligen Instruktion verändert oder getestet und wieder in den Speicher zurück geschrieben.

Wie bei Operationen der ALU, werden die Bits OV, C, TC (Kapitel 2.2.1) beeinflußt.

2.6 Adressierungs-Modi

Der C5x besitzt verschiedene Möglichkeiten, Daten in Hauptspeicher zu addressieren. In den folgenden Abschnitten werden diese kurz beschrieben.

2.6.1 Direkte Adressierung

Der gesamte Bereich des Datenspeichers ist in Seiten (*data pages*) zu je 128 16-Bit-Worten organisiert. Durch die Aufteilung einer Adresse in Seitennummer und

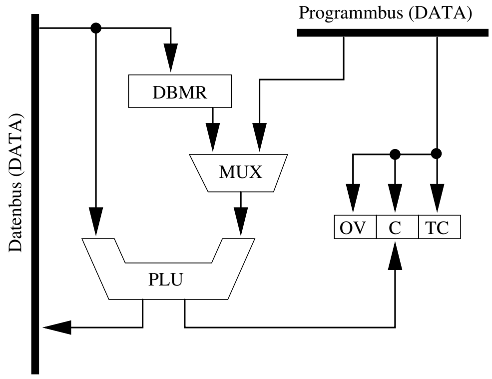

Abbildung 2.3: Blockschaltbild PLU

Offset kann man den kompletten Speicherbereich von 64K 16-Bit Worten mit nur 7 Bit addressieren. Eine komplette Adresse für die direkte Adressierung setzt sich dann aus zwei Teilen zusammen:

1. Der Inhalt des DP (*data memory page pointer*); ein Register, das die Nummer der aktuellen Speicherseite enthält. Der Befehl LDP (*load data page*) lädt das Register mit einem Wert aus dem Speicher.

2. Der Seitenoffset, der im Befehl kodiert ist und durch die letzten 7 Bit des Befehlswortes repräsentiert wird.

2.6.2 Memory-mapped Adressierung

Einen Sonderfall im Datenspeicher nimmt die Seite 0 ein. Auf dieser Seite werden Register des Prozessorkerns, I/O-Ports und Register, die für die Kommunikation mit dem umgebenden System benötigt werden, in den normalen Speicherbereich eingeblendet. Dieses Verfahren bezeichnet man als *memory-mapping*. Es gibt Befehle, die ohne den DP verändern zu müssen, auf diese Adressen, und damit direkt auf verschiedene Prozessorregister, zugreifen können. Im Gegensatz zur direkten Adressierung wird hier nicht der Inhalt des DP zur Vervollständigung der Adresse benötigt. Bei den Befehlen mit dieser Adressierungsart wird der Inhalt des darunter liegenden Datenspeichers nicht verändert.

2.6.3 Indirekte Adressierung

Bei der indirekten Adressierung wird das Hilfsregisterfile (Kapitel 2.3) verwendet, um den Speicherort von Operanden zu bestimmen. Über den Zeiger ARP wird eins der acht Hilfsregister ausgewählt, das die Adresse des Operanden im Datenspeicher enthält.

2.6.4 Unmittelbarer Operand

Für arithmetische Befehle des C5x gibt es die Möglichkeit, den Operanden als Konstante direkt im Befehl mit zu kodieren. Der Wertebereich dieser Konstanten, und damit die mögliche Bit-Breite, hängt von dem entsprechenden Befehl ab. Es gibt Befehle wie *ADD* oder *SUB*, die eine 8-Bit Konstante erlauben. Der Befehl *LDP (load data page pointer)* erlaubt 9 Bit und der Befehl *MPY* (*multiply*) erlaubt in dieser Adressierungsart sogar 13 Bit für die Konstante.

Benötigt man jedoch eine größere Konstante , besteht die Möglichkeit, diese Konstante unmittelbar hinter dem Befehl im Programmspeicher anzugeben. Der gesamte Befehl benötigt dann 2 Speicherworte, es stehen aber alle 16 Bit für die Konstante zur Verfügung.

2.6.5 Adressieren spezieller Register

Es gibt neun Befehle, die ihre Operanden aus zwei speziellen Registern bekommen. Dies sind zum einen Befehle der PLU, die auf das Register *DBMR* (*dynamic bit manipulating register*) zurückgreifen, wenn kein unmittelbarer Operand spezifiziert ist. Zum anderen sind diese Blockverschiebe- und Multiply/Accumulate-Instruktionen, die das *BMAR* (*block move adress register*) als Ziel- oder Quelladresse nutzen.

2.6.6 Ringpuffer

Der C5x besitzt 2 Ringpuffer, die dazu dienen, einen bestimmten Ausschnitt des Speichers ohne großen Aufwand zu durchlaufen. Es reicht dazu, die Start- und Endadresse des Ringpuffers in bestimmten Registern abzulegen. Nach jeder Instruktion wird der aktuelle Zeiger innerhalb dieses Puffers automatisch inkrementiert oder dekrementiert. Der Zeiger für den Ringpuffer wird an einer beliebigen Stelle in das Hilfsregisterfile (Kapitel 2.3) eingeblendet, so daß man mit indirekter Adressierung darauf zugreifen kann.

2.7 Überblick über den Befehlssatz des C5x

Der Befehlssatz des C5x kann in 7 Gruppen eingeteilt werden :

1. Befehle für die ALU und den Akkumulator

Mnemonic	Beschreibung
LACC	Akkumulator laden
SACL	die niederwertigen 16 Bit des ACCU speichern
SACH	die höherwertigen 16 Bit des ACCU speichern
PAC	lade ACCU mit PREG
AND	UND-Verknüpfung, ebenso OR, XOR, NEG, ROR, ROL
ANDB	Operationen wie bei AND Unterschied: zweiter Operand ist der ACCUB)*
ADD	Addition, ebenso SUB, ADDB, SUBB

)* Eine detaillierte Beschreibung der Befehle für den Akkumulator-Puffer findet man in Tabelle 5.1 auf Seite 70.

2. Befehle für das Hilfsregisterfile und den DP

Mnemonic	Beschreibung
LAR	lade Hilfsregister ARn, $n = 0..7$
SAR	speichere Inhalt von Hilfsregister ARn
ADRK	Addiere Konstante zu ARn, ebenso SBRK
MAR	ARP modifizieren
LDP	Lade Data Page pointer (DP)

3. Befehle für die PLU

Mnemonic	Beschreibung
APL	UND-Verknüpfung mit Wert im Speicher, ebenso OPL, XPL
CPL	Vergleich mit Wert im Speicher
SPLK	Speichere 16-Bit Wert im Datenspeicher

4. Befehle für den Multiplizierer, das T-Register und das Produktregister

Mnemonic	Beschreibung
LT	Lade TREG0
MPY	Multipliziere
MAC	Multipliziere und addiere
APAC	Addiere ACCU und PREG
SPL	die niederwertigen 16 Bits des PREG speichern
SPH	die höherwertigen 16 Bits des PREG speichern

5. Sprungbefehle

Mnemonic	Beschreibung
B	Unbedingter Sprung
BCND	Bedingter Sprung
CALL	Unterprogramm aufrufen
RET	Rücksprung vom Unterprogramm

Alle Sprungbefehle können auch verzögert ausgeführt werden, d.h. daß ein Sprung, der normalerweise 4 Zyklen dauert, da die Instruktions-Pipeline ungültig wird und wieder neu gefüllt werden muß, nur zwei Zyklen dauert. Das wird dadurch erreicht, daß die zwei auf den Sprungbefehl folgenden Befehle noch komplett ausgeführt werden. Der Befehl für den verzögerten Sprung wird also um zwei Worte vorgezogen.

6. Ein-/Ausgabe- und Speicherbefehle

Mnemonic	Beschreibung
LMMR	lade memory-mapped Register
SMMR	speichere memory-mapped Register
BLDD	Verschieben von Daten innerhalb des Datenspeichers
BLDP	Verschieben von Daten vom Daten- in den Programmspeicher
BLPD	Verschieben von Daten vom Programm- in den Datenspeicher

7. Befehle zur Programmkontrolle

Mnemonic	Beschreibung
BIT	Bits testen
LST	Statusregister laden
NOP	No Operation
POP	Schiebe Top of Stack in ACCU, ebenso PUSH
POPD	Schiebe Top of Stack in Datenspeicher, ebenso PUSHD
RPT	Wiederhole folgenden Befehl
RPTZ	Wiederhole folgenden Befehlsblock

Eine komplette Liste aller Befehle, incl. der möglichen Adressierungsarten und Ausführungszeiten, findet man in [5], Kapitel 4.

Kapitel 3

Entwicklungsumgebung

3.1 LANCE

Bei der Entwicklung des Codegenerators für den C5x kommt das Tool LANCE [3] zum Einsatz. LANCE steht für "LS12 ANSI C Compilation Environment" und wurde als Hilfsmittel zur Codeerzeugung für DSP's entwickelt. LANCE besteht im wesentlichen aus den folgenden Komponenten:

1. dem ANSI C Frontend incl. diverser Optimierungstools,
2. einer Klassenbibliothek, um auf die vom Frontend erzeugte Zwischendarstellung zugreifen zu können und
3. eine Klassenbibliothek, um den Daten- und Kontrollfluß innerhalb des Programms zu analysieren.

3.1.1 ANSI C Frontend

Das ANSI C Frontend übersetzt ein C-Programm in eine vom verwendeten System unabhängige Zwischendarstellung (*intermediate representation* - *IR*). Es werden zwei Dateien vom Frontend erzeugt. Die erste Datei enthält die aus den C-Befehlen generierten Anweisungen in Form von Drei-Adress-Befehlen. Die zweite Datei, die Symboltabelle, enthält sämtliche Informationen über alle Variablen und Funktionen der IR. Die Struktur der IR wird in Kapitel 3.1.2 genauer erläutert.

Die Übersetzung eines Programms kann über verschiedene Konfigurationseinstellungen beeinflußt werden. Die Einstellungen werden über eine Konfigurationsdatei vorgenommen, die zum einen die zu verwendenden Bitbreiten der unterschiedlichen

Datentypen, die im Programm verwendet werden, enthält, und zum anderen vorgibt, wie mit Kontrollstrukturen im Programm, umzugehen ist, wie z.B. If-Then-Else Statements oder Schleifen.

Weiterhin gehören zum Frontend einige Optimierungstools, die den erzeugten Code verbessern. Zu diesen Tools gehören z.B. Programme, die Sprünge im Programm optimieren oder Programmcode entfernen, der nie ausgeführt wird oder keine Funktion mehr erfüllt. Letzteres kann durch Anwendung anderer Tools entstehen, die redundanten Code nicht explizit entfernen.

Alle Tools, sowie das Frontend selbst, können über die grafische Benutzeroberfläche von LANCE gestartet werden. Änderungen, die von einzelnen Tools vorgenommen werden, können mit dieser Oberfläche direkt verfolgt werden.

3.1.2 Die Intermediate Representation (IR) von LANCE

Die IR besteht aus zwei reinen ASCII-Dateien. Beide Dateien können über Funktionen der mitgelieferten Klassenbibliothek eingelesen werden.

Die erste Datei besteht aus einer Liste aller Anweisungen (Statements) des übersetzten Programms in Drei-Adress-Form. Es gibt zwei Klassen, in die die verschiedenen Anweisungen eingeteilt sind.

1. Primitive Statements :
   - Zuweisung - Assignment
   - Sprung - Jump
   - Marke - Label
   - Bedingte Anweisung - Guarded Statement
   - Rücksprung - Return
2. High-Level Statements:
   - Schleifen - For loop
   - Continue Statement (entspr. C)
   - Break Statement (entspr. C)
   - If - then Statement
   - If - then - else Statement

In wie weit High-Level Statements benutzt werden, hängt von der Konfiguration des Frontend ab. Es kann festgelegt werden, welche High-Level Statements für die IR benutzt werden und welche durch primitive Statements zu ersetzen sind.

Die Klasse der primitiven Statements soll hier noch etwas näher beschrieben werden.

Zuweisung Der am häufigsten vorkommende Typ ist die Zuweisung. Mit der Zuweisung können Daten verschoben oder Operationen ausgeführt werden. Die Zuweisung besteht aus zwei Ausdrücken, dem Ziel und der Quelle. Im Falle einer Operationsausführung ist die auszuführende Operation im Quellausdruck enthalten. Das Ziel gibt an, wo ein Resultat gespeichert werden soll.

Sprung Mit Sprüngen wird der Kontrollfluß im Programm unterbrochen. Ein Sprung hat nur einen Operanden, die Marke, zu der verzweigt wird. Sprünge findet man häufig zusammen mit bedingten Anweisungen, man kann damit z.B. das High-Level Statement *“If-then”* abbilden.

Marke Eine Marke dient zur Kennzeichnung eines Sprungziels.

Bedingte Anweisung Dieses Statement besteht aus zwei Teilen:

1. einer Bedingung und
2. einer Anweisung

Es hängt vom Resultat der Bedingung ab, ob die nachfolgende Anweisung ausgeführt wird oder nicht.

Return Eine Return-Anweisung kennzeichnet das Ende einer Funktion oder des Programms. Bei Funktionen, die einen Wert zurückgeben, besitzt eine Return-Anweisung einen Operanden, den Rückgabewert.

Ebenso wie es verschiedene Statementtypen gibt, existieren in der IR auch verschiedene Typen von Ausdrücken, die innerhalb der Statements verwendet werden:

Symbol Der einfachste Ausdruck ist ein Symbol und steht für eine Variable oder eine Funktion innerhalb des Programms. Alle relevanten Daten zu Symbolen kann man der Symboltabelle entnehmen.

Binäre und unäre Ausdrücke Als nächstes gibt es binäre und unäre Ausdrücke. Binäre Ausdrücke bestehen wiederum aus zwei Unterausdrücken und einem Operator. Als Operator kommen z.B. +, -, UND, ODER, usw. in Frage. Unäre Ausdrücke bestehen dagegen aus einem Ausdruck und dem dazugehörigen unären Operator, z.B. NOT, -.

Konstante Bei Konstanten wird zwischen zwei Typen unterschieden. Zum einen gibt es INTEGER-Konstanten und zum anderen STRING-, FLOAT- und LONG-Konstanten, die unterschiedlich zu behandeln sind. Innerhalb der IR unterscheiden sie sich durch die Art der Darstellung.

Funktionsaufruf Funktionsaufrufe bestehen aus der aufgerufenen Funktion und einer Liste von Argumenten, die an die Funktion übergeben werden.

CAST-Funktion Kommt im Quelltext des Programms an einer Stelle eine explizite CAST-Funktion zum Einsatz, so gibt es in der IR dafür ebenfalls einen CAST-Ausdruck.

Pointer und indizierte Ausdrücke Die beiden letzten Typen sind für Zugriffe über Pointer bzw. für indizierte Ausdrücke wie Zugriffe auf Arrays oder Vektoren.

Die Symboltabelle enthält alle Informationen über verwendete Variablen und Funktionen.

Zu diesen Informationen gehören :

- der Typ einer Variablen: INTEGER, POINTER, ARRAY,usw,
- der Geltungsbereich einer Variablen,
- ist die Variable Parameter einer Funktion,
- handelt es sich um eine temporäre Variable, d.h. sie wurde vom Frontend generiert um Zwischenergebnisse zu speichern,
- den Typ des Rückgabewertes einer Funktion.

3.1.3 C++ Schnittstelle der IR

Um auf die einzelnen Statements oder Ausdrücke der IR zugreifen zu können, wird mit LANCE eine C++-Klassenbibliothek zur Verfügung gestellt, die sämtliche Funktionen enthält, die man für die Analyse der IR benötigt. Diese Klassenbibliothek beinhaltet Methoden, mit denen man grundlegende Datenstrukturen wie z.B. Listen oder Bäume erzeugen und manipulieren kann. Diese Datenstrukturen sind deshalb sehr wichtig, da sie ebenfalls für die interne Darstellung der IR und der Symboltabelle verwendet werden. Bei dem entwickelten Codegenerator CODEGEN werden diese Datenstrukturen häufig benutzt.

3.1.3.1 Funktionen zur Kontroll- und Datenflußanalyse

Es gibt drei besonders wichtige C++-Klassen, die von LANCE bereitgestellt werden und zur Analyse des Kontroll- und Datenflusses innerhalb eines Programms dienen:

1. DataFlowInformation (*DFI*) - Datenflußinformationen
2. ControlFlowGraph (*CFG*) - Kontrollflußgraph
3. BasicBlock (*BB*)

Die Klasse *DFI* liefert Informationen über die Abhängigkeiten der verschiedenen Anweisungen untereinander. Man erhält Informationen, ob und wo Resultate einer Anweisung in anderen Anweisungen verwendet werden. Diese Funktionen stellen eine Basis für den Codegenerator dar, da auf Grund der Abhängigkeiten der Anweisungen die Basisblöcke einer Funktion leicht in Graphen umgesetzt werden können, die bei der Codegenerierung (Kapitel 4) benötigt werden.

Die Klasse *CFG* stellt die Struktur einer Funktion in Form von Basisblöcken dar. Da die Codegenerierung für Basisblöcke zu untersuchen ist, erhält man mit der Klasse *CFG* eine wesentliche Grundlage für die Implementierung des Codegenerators.

Die Methoden der Klasse BasicBlock dienen schießlich dazu, auf die einzelnen Anweisungen innerhalb eines Basisblocks zugreifen zu können.

3.1.4 Beispiel für die IR

Anhand eines kleinen Beispiels soll die Struktur der IR verdeutlicht werden.

Die Funktion *complex_multiply*, aus den DSP-Stone-Benchmarks [12], besteht im wesentlichen aus zwei Anweisungen :

$$c_r = a_r * b_r - a_i * b_i$$
$$c_i = a_r * b_i + a_i * b_r$$

Die von LANCE erzeugte Zwischendarstellung findet man in Abbildung 3.1.

3.2 OLIVE

Als weiteres Tool bei der Entwicklung des Codegenerators für den C5x kommt OLIVE zum Einsatz.

```
function 'main'(19):

@49@ '_t_63'(65) = 'bi'(23) ;
@49@ '_t_65'(67) = 'ai'(21) ;
@49@ '_t_61'(63) = '_t_65'(67) * '_t_63'(65) ;
@49@ '_t_69'(71) = 'br'(22) ;
@49@ '_t_71'(73) = 'ar'(20) ;
@49@ '_t_67'(69) = '_t_71'(73) * '_t_69'(71) ;
@49@ '_t_59'(61) = '_t_67'(69) - '_t_61'(63) ;
@49@ 'cr'(24) = '_t_59'(61) ;

@50@ '_t_79'(81) = 'br'(22) ;
@50@ '_t_81'(83) = 'ai'(21) ;
@50@ '_t_77'(79) = '_t_81'(83) * '_t_79'(81) ;
@50@ '_t_85'(87) = 'bi'(23) ;
@50@ '_t_87'(89) = 'ar'(20) ;
@50@ '_t_83'(85) = '_t_87'(89) * '_t_85'(87) ;
@50@ '_t_75'(77) = '_t_83'(85) + '_t_77'(79) ;
@50@ 'ci'(25) = '_t_75'(77) ;
```

Abbildung 3.1: Beispiel der IR von LANCE

OLIVE ist ein Generator für Codegeneratoren. Um einen Codegenerator für einen beliebigen Zielprozessor zu erhalten, benötigt man eine Beschreibung dieses Zielprozessors. OLIVE benötigt die Beschreibung des Zielprozessors in Form einer Grammatik. Die Ausgabe von OLIVE ist dann der gewünschte Codegenerator. Diese Ausgabe besteht aus einer C-Datei und einer Header-Datei, die man in das eigene Programm einbinden kann.

OLIVE basiert auf anderen Codegenerator-Generatoren, wie TWIG [4] oder iBURG [1], wurde aber an verschiedenen Stellen weiterentwickelt. OLIVE besitzt eine viel größere Flexibilität bei der Beschreibung des Prozessors, für den ein Codegenerator produziert werden soll. Diese Flexibilität drückt sich besonders im Kostenteil der Grammatikregeln aus (Kapitel 3.2.2). Mit OLIVE hat man zum ersten Mal die Möglichkeit mit komplexen Kostenfunktionen zu arbeiten. Man ist nicht mehr, wie etwa bei iBURG, nur auf einfache Integerwerte beschränkt, sondern kann beliebig komplexe Datenstrukturen aufbauen. Wie man später sehen wird, ist dies für die Codegenerierung für DSP's sehr wichtig.

Der Codegenerator muß sehr häufig diese Datenstrukturen miteinander vergleichen, damit man sich immer nur für die kostengünstigste Lösung entscheidet. Aus diesem Grund benötigt der Codegenerator eine Vergleichsfunktion, die den kleineren Kostenwert von zwei möglichen Lösungen ermittelt.

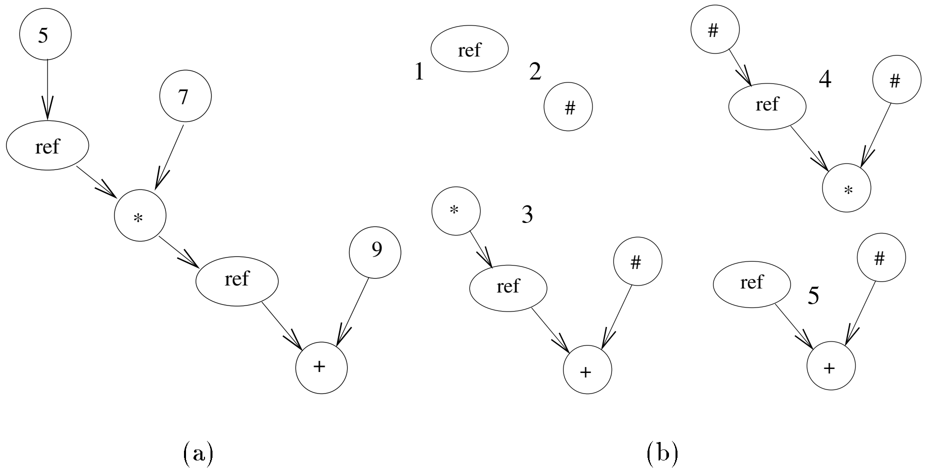

Abbildung 3.2: Datenflußbaum (a) und verschiedene Baummuster (b)

1	Lade Wert vom Speicher in ein Register
2	Lade Konstante in ein Register
3	Benutze ein Produkt zu indirekten Adressierung und addiere dazu einen unmittelbaren Wert
4	Multipliziere unmittelbar mit einer direkten Speicheradresse
5	Addiere unmittelbaren Wert mit einem indirekt adressierten Wert

Tabelle 3.1: Befehle für die Baummuster in Abb. 3.2(b)

3.2.1 Funktionsweise des Codegenerators

Wie ermittelt der Codegenerator aber überhaupt mögliche Lösungen ?

Wie andere Codegenerator-Generatoren (Twig, iBurg) arbeitet Olive ebenfalls mit der Überdeckung von Baummustern (*tree pattern matching*) und dynamischer Programmierung.

Olive akzeptiert als Eingabe die Befehle des Zielprozessors in Form einer Baumgrammatik (Kapitel 3.2.2). Jeder Befehl des Prozessors muß für Olive als Baummuster dargestellt werden. Abbildung 3.2(b) zeigt einen kleinen Ausschnitt der Befehle eines beliebigen Prozessors. Die Bedeutung der 5 Baummuster findet man in Tabelle 3.1.

Mit diesen Befehlen soll der Datenflußbaum (*data flow tree - DFT*) aus Abbildung 3.2(a) überdeckt werden. Es gibt dafür verschiedene Techniken. Ein heuristischer Ansatz für die Überdeckung wurde in [6] vorgestellt. Bei diesem Ansatz wird versucht, von der Wurzel des DFT aus immer die größtmöglichen Baummuster zur

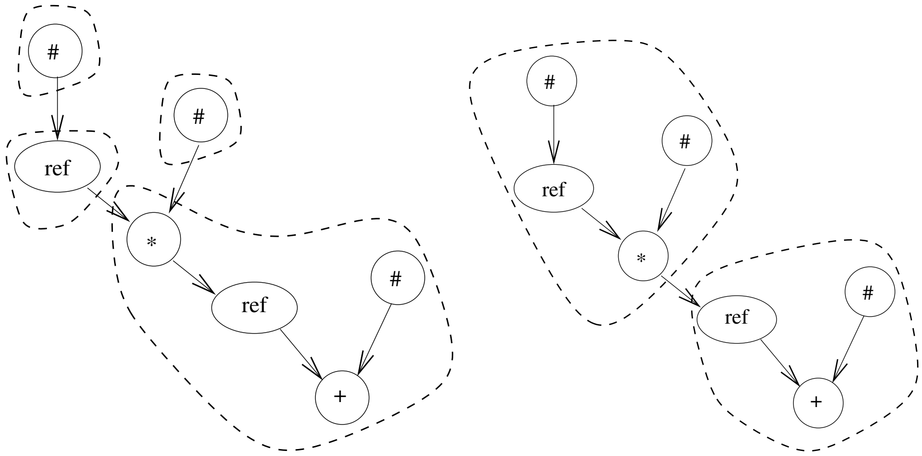

Abbildung 3.3: Überdeckungen für den DFT aus Abb. 3.2 (a)

Überdeckung zu verwenden, um möglichst gute Überdeckungen zu erhalten. Das Ergebnis dieser MMM[1]-Technik für den Beispielbaum sieht man in Abbildung 3.3(a). Diese Lösung ist nicht sehr befriedigend, vergleicht man sie mit der Überdeckung in Abbildung 3.3(b). Lösung (b) wurde mit der Technik erzeugt, die auch im von OLIVE generierten Codegenerator verwendet wird. Es wird *immer* die kostengünstigste Lösung gefunden [4]. Man erkennt in Abbildung 3.3, daß bei der Lösung (a) nur 2 Befehle benutzt werden, statt 4 Befehle in Lösung (b), um den kompletten DFT zu überdecken.

Die Eingabe für den generierten Codegenerator sind Bäume, die aus den Funktionen des zu kompilierenden Programms gebildet werden (Kapitel 4.3). Der Codegenerator ermittelt nun die Menge von Baummustern aus der Prozessorbeschreibung, die den Eingabebaum mit den geringsten Kosten überdeckt. Dies geschieht in einer ersten Phase des Codegenerators, bei der der Baum bottom-up, also von den Blättern zur Wurzel, durchlaufen wird. In dieser Phase können ebenfalls wichtige Daten über die benutzte Menge von Baummustern gesammelt werden. Es ist bei der Codegenerierung für DSP's sehr wichtig zu erfahren, welche Register im Datenpfad des Prozessors in der Lösung verwendet werden. Diese Informationen benötigt man beim Scheduling (Kapitel 4.2).

In der zweiten Phase, die unabhängig von der ersten initiiert werden kann, können die Aktionsteile (Kapitel 3.2.2) der ausgewählten Baummuster ausgeführt werden.

[1]MMM - *maximum munching method*

grammar		**%{ TYPEDEF %} Declaration %% rules**
rule	→	**nonterm : tree [cost] action;**
tree	→	**tree (tree_list)**
	|	**term**
	|	**nonterm**
tree_list	→	**tree_list, child**
	|	**child**
child	→	**tree**
	|	**_**
cost	→	**C-Code**
	|	**C-Expression**
action	→	**C-Code**

Tabelle 3.2: EBNF Darstellung der Baumgrammatik für OLIVE

3.2.2 Beschreibung des Zielprozessors

Um mittels OLIVE einen Codegenerator generieren zu lassen, muß man zunächst den Zielprozessor in einer von OLIVE akzeptierten Weise beschreiben. Diese Beschreibung wird in Form einer Grammatik angegeben, die der EBNF[2] in Tabelle 3.2 genügt. **nonterm** entspricht den Nicht-Terminal-Symbolen der Grammatik, **term** entspricht den Terminal-Symbolen der Grammatik. Das Symbol "_" steht für eine *don't care*[3] Situation.

Wie man in der ersten Zeile der Tabelle erkennen kann, besteht die Prozessorbeschreibung aus drei Teilen :

TypeDef In diesem Teil hat man die Möglichkeit Variablen und Datentypen zu deklarieren, die in den Kosten- und Aktionsteilen der einzelnen Regeln der Grammatik Verwendung finden. Sehr häufig werden z.B. komplexe Kostendaten benötigt und man definiert den Datentyp dafür an dieser Stelle. Der Name des zu definierenden Datentyps für die Kostenfunktionen ist **COST**. Da man die Variablen dieses Datentyps in der Regel nicht mit vorhandenen Operatoren vergleichen kann, muß man hier ebenfalls eine Vergleichsfunktion für die Daten des Typs **COST** definieren. Diese Funktion heißt **COST_LESS**(x,y). Mit **COST_LESS** wird ermittelt, ob die Variable x "echt kleiner" ist als y.

[2] Erweiterte Baccus-Naur-Form

[3] don't care - normalerweise ein Zeichen dafür, daß es keine Rolle spielt, was an dieser Stelle geschieht. In diesem speziellen Fall kann das *"don't care"* dazu benutzt werden, Teilbäume während der Codegenerierung auszulassen.

Declaration Im zweiten Teil der Prozessorbeschreibung werden die Terminal- und Nicht-Terminalsymbole der Grammatik definiert. Terminal-Symbole beginnen mit dem Schlüsselwort `%term` gefolgt von dem Symbol selbst. Zwei Beispiele :

`%term ADD`	mögliches Terminalsymbol für eine Addition
`%term BRANCH`	mögliches Terminalsymbol für einen Sprung

Hinter dem Schlüsselwort `%term` folgt eine Liste von Symbolen, die dann alle als Terminal-Symbole deklariert werden.

Auf ähnliche Art und Weise werden im Deklarations-Teil auch die Nicht - Terminal - Symbole definiert.

Für alle Regeln, die dasselbe Nicht-Terminal-Symbol als Ziel haben, kann im Aktionsteil ein Wert vom Typ `return type` zurückgegeben werden. Der `return type` wird nach dem Schlüsselwort `#declare` angegeben. Gleichzeitig können mit `#declare` für diese Funktionen Argumente deklariert werden.

```
#declare <return type> nonterminal <argumente>
```

rules Im Abschnitt *rules* werden die Regeln der Grammatik definiert. Eine Regel hat folgenden Struktur:

$$
\begin{array}{rcl}
\text{Ziel} & : & \text{Baummuster} \\
 & & \{\ \text{Kostenteil}\ \} \\
 & = & \{\ \text{Aktionsteil}\ \}
\end{array}
$$

Das *Ziel* einer Regel definiert den Ort, an dem das Ergebnis der durch das *Baummuster* dargestellten Operation gespeichert wird. *Ziel* einer Regel ist immer ein Nicht-Terminal der Grammatik. Ein Baummuster ist eine Kombination von Terminal- und Nicht-Terminal-Symbolen und stellt normalerweise einen Befehl des Prozessors dar. Im Kostenteil kann eine beliebig komplexe Berechnung durchgeführt werden, um die Kosten zu berechnen, die entstehen, wenn das Baummuster zur Überdeckung eines Teilbaumes genutzt wird. Je nach Einsatz des Baumparsers, können z.B. die Anzahl der benutzten Instruktionen für dieses Muster oder die Anzahl der Prozessorzyklen berechnet werden, die dieses Muster benötigt. Auf Grund der errechneten Kosten findet im ersten Durchlauf des Codegenerators die kosten-optimale Überdeckung eines Ausdrucksbaumes mit den zur Verfügung stehenden Baummustern statt. Der Kostenteil einer Regel ist optional, d.h. es wird ein Standardwert benutzt, wenn dieser Teil der Regel leer bleibt. Dieser Standardwert wird über eine Konstante **DEFAULT_COST** bestimmt.

Im Aktionteil wird in der Regel der Assemblercode ausgegeben, den das ausgewählte Baummuster repräsentiert, ebenso können auf Grund von Daten, die in der ersten Phase des Codegenerators gesammelter worden sind, Entscheidungen für das Scheduling (Kapitel 4.2) getroffen werden.

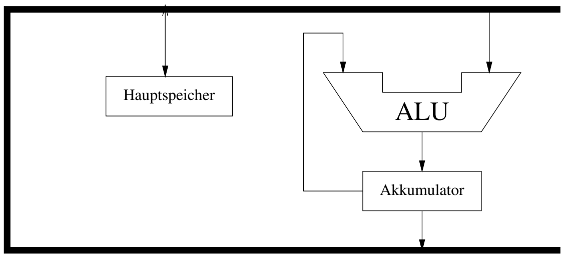

Abbildung 3.4: Sehr einfacher Prozessor

Beispiele der Prozessorbeschreibung für den C5x findet man in Kapitel 4.1, die vollständige Beschreibung, die für das Programm CODEGEN (Kapitel 7.1) verwendet wurde, findet man im Anhang.

3.2.3 Beispiel für die Arbeitsweise von Olive

Anhand eines kleinen Beispiels soll die Arbeitsweise von Olive für eine heterogene Architektur und und die Strktur des generierten C-Codes erläutert werden.

In Abbildung 3.4 ist ein sehr einfacher Prozessor skizziert. Der Prozessor besteht aus einer ALU, die die Operationen Addition, und Absolutwert ausführen kann, wobei die Operanden Konstanten oder Werte aus dem Speicher sein können. Ebenso gibt es zwei Operationen die den Akkumulator mit einem Wert laden, auch hier entweder mit einer Konstanten oder einem Wert aus dem Speicher. Als letzte Operation des Prozessors kann der Inhalt des Akkumulators in den Speicher geschrieben werden.

Diese Struktur führt zu folgender Prozessorbeschreibung (s. Abbildung 3.5) :

Es gibt fünf Terminal-Symbole in der Prozessorbeschreibung, *ADD, ABS, STORE, MEM* und *CONST.* Die ersten drei stehen für die Operationen des Prozessors, Addition, Absolutwert und Speichern des Akkumulators. Das Laden des Akkumulators wird nicht durch ein Terminal-Symbol, sondern durch zwei Regel beschrieben, die dafür sorgen, daß der benötigte Wert in den Akkumulator geladen wird. Die beiden Terminal-Symbole MEM und CONST besitzen in der Regel ein Attribut, das zum einen das Symbol der benötigten Speicherstelle angibt und zum anderen den Wert der verwendeten Konstanten besitzt.

Ebenso gibt es drei Nicht-Terminal-Symbole*, ass, accu* und *mem. ass* ist das Startsymbol, d.h. jeder Ausdrucksbaum, der durch den generierten Baumparser überdeckt werden soll, muß mit einer Regel starten, dessen Ziel *ass* ist. Im Beispiel ist dies nur die Regel *STORE (mem, acc)* . Jeder Baum für das Beispiel muß also mit *STORE ( )* beginnen. *mem* und *acc* beschreiben als Nicht-Terminal-Symbole die Register im Datenpfad des Prozessors, im Beispiel also den Akkumulator und den

Hauptspeicher. Regeln, deren Ziel *mem* ist, geben außerdem einen Wert zurück, normalerweise ein Symbol für eine Speicherstelle.

Aus dieser Beschreibung generiert Olive zwei Dateien. Eine C-Quellcode-Datei und eine C-Header-Datei. In der Header-Datei werden Indizes für die Terminal-Symbole definiert, damit man in dem Programm, von dem der Baumparser aufgerufen wird, dieselben Symbole für die Generierung von Bäumen benutzen kann, wie in der Prozessorbeschreibung. Der Baumparser selbst kennt schließlich nur noch die Indizes.

In der C-Quellcodedatei sind die Funktionen zur Überdeckung eines Baumes bzw. zur Ausführung der Aktionteile aller Regeln in Form von Tabellen mit `switch`-Anweisungen implementiert.

Für die Überdeckung eines Baumes wird die Funktion `burm_label` mit der Wurzel des zu überdeckenden Baumes aufgerufen. Der Baumparser ermittelt nun bottom-up die Kosten für die Überdeckung. Scheitert die Überdeckung, wird ein NULL-Zeiger zurückgegeben, ansonsten wird die kostengünstigste Überdeckung ermittelt und zurückgegeben.

Für jedes Nicht-Terminal-Symbol wird eine Funktion generiert, um die Aktionsteile der Regeln aufzurufen. In unserem Beispiel also die drei Funktionen `mem_action`, `accu_action` und `ass_action`. Da `ass` das Startsymbol der Grammatik in unserem Beispiel ist, kann man den Aktionsteil der Wurzel des überdeckten Baumes mit `ass_action` aufrufen. Im Aktionsteil selbst können dann die Aktionsteile der Kinder aufgerufen werden.

In den Aktionsteilen wird das Scheduling und die Ausgabe des Assemblercodes stattfinden. Da man die Aktionsteile beliebig häufig oft aufrufen kann, können im Aktionsteil verschiedene und voneinander unabhängige Aufgaben durchgeführt werden. Die kann man dadurch erreichen, daß der Aufruf mit verschiedenen Parametern erfolgt.

3.2.3.1 Beispielbaum

Für den folgenden Ausdruck soll für den Prozessor aus Abbildung 3.4 Assemblercode generiert werden : $a = b + c$

a, b und c sind Speicherstellen und werden mit dem Terminal-Symbol MEM definiert, die Operation + wird durch das Terminal -Symbol ADD dargestell und die Zuweisung = entspricht dem Terminal-Symbol STORE.

Daraus ergibt sich der Baum aus Abbildung 3.6(a), der die Eingabe für den Baumparser darstellt. Abbildung 3.6 (b) zeigt den Baum nach der Überdeckung durch den Baumparser. Man erkennt, daß der Baumparser eine Registertransferoperation, *accu* : *mem*, selbstständig eingeführt hat. Dies ist nötig, da der Prozessor,

```
%term CONST MEM ADD ABS STORE

%declare<char*> ass;
%declare<char*> mem     <char* m>;
%declare<char*> accu;
%%
# Speicheroperation

ass: STORE (mem,accu)
{ $cost[0].c = $cost[3].c + 1;
}={
  $action[3](); sprintf(s,"\n\tSACL\t%s", $action[2](NULL));
// Emit_Code(s);
};

# Absolutwert

accu: ABS(accu)
{ $cost[0].c = $cost[2].c + 1;
}={
  $action[2](); sprintf(s,"\n\tABS\t");
 // Emit_Code(s); };

# Addition ADD

accu: ADD(accu,CONST)
{ $cost[0].c = $cost[2].c  + 1;
}={
  $action[2](); sprintf(s,"\n\tADD\t#%s", ATTRIB($3));
// Emit_Code(s); };

accu: ADD(accu,mem)
{ $cost[0].c = $cost[2].c + $cost[3].c + 1;
}={
  $action[2](); sprintf(s,"\n\tADD\t%s", $action[3](NULL));
// Emit_Code(s); };

# Ladeoperationen

accu: CONST
{ $cost[0].c = 1;
}={ sprintf(s,"\n\tLACC\t#%s", ATTRIB($1));
// Emit_Code(s); };

accu: MEM
{ $cost[0].c = $cost[1].c + 1;
}={
  sprintf(s,"\n\tLACC\t%s",ATTRIB($1));
// Emit_Code(s); };
%%
```

Abbildung 3.5: Olive-Beschreibung für den Prozessor aus Abbildung 3.4

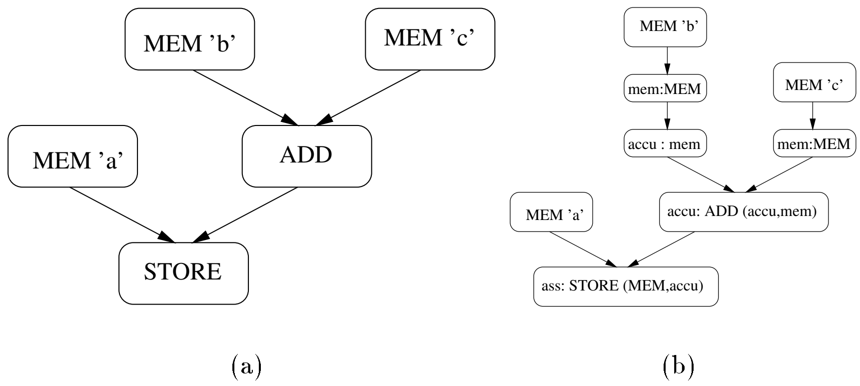

Abbildung 3.6: Eingabe für den Baumparser (a) Überdeckter Baum (b)

auf Grund seiner heterogenen Architektur, die Operanden in bestimmten Registern erwartet. Mit der Prozessorbeschreibung kann die heterogene Architektur des Prozessors modelliert werden.

Nach der Überdeckung wird nun der Aktionsteil der Wurzel ausgeführt. In diesem Aktionsteil werden nun wiederum die Aktionsteile der Teilbäume ausgeführt. In dem Beispiel wird in jedem Aktionsteil nur der Assemblercode für das entsprechende Muster ausgegeben, was zu folgender Sequenz führt :

```
LACC  b
ADD   c
SACL  a
```

Kapitel 4

Grundlagen zur Codegenerierung

Im folgenden Kapitel werden die Methoden vorgestellt und erläutert, die zur Codegenerierung für den C5x eingesetzt werden. Die beiden ersten Methoden (Kapitel 4.1 und 4.2) finden Anwendung bei der Codegenerierung für einen einzelnen Ausdrucksbaum[1]. Hierbei wird durch die Methode aus Kapitel 4.1 eine optimale Instruktionsauswahl für einen Ausdrucksbaum sichergestellt und durch die Methode aus Kapitel 4.2 wird gewährleistet, daß durch eine optimale Wahl der Reihenfolge der Teilbäume des Ausdrucksbaums keine überflüssigen Speicherzugriffe (*Memory Spills*) erzeugt werden.

Diese beiden Methoden funktionieren allerdings nur für Ausdrucksbäume. Wenn man jede Funktion eines Programms als Ausdrucksbaum darstellen könnte, würden die obigen Methoden zur Codegenerierung ausreichen.

In der Regel lassen sich Funktionen mittels Datenflußanalyse durch einen Datenflußgraph beschreiben. Der Datenflußgraph (*DFG - dataflow graph*) ist ein gerichteter azyklischer Graph (*DAG - directed acyclic graph*), der die Struktur der Funktion wiedergibt. Die beiden letzten Methoden (Kapitel 4.3 und 4.4) beschäftigen sich mit der Aufteilung von DFG's in Datenflußbäume (*DFT - dataflow tree*), um die ersten beiden Methoden verwenden zu können.

Der Ansatz von Araujo und Malik (Kapitel 4.3) versucht die Struktur des Zielprozessors auszunutzen, um eine möglichst gute Aufteilung von DFG's in DFT's zu erreichen, während der zweite Ansatz von Leupers (Kapitel 4.4) einen *Simulated-Annealing (SA)* Algorithmus verwendet, um Teilausdrücke, die mehrfach verwendet werden (*CSE - common subexpression*), an günstigen Stellen in der Registerstruktur zwischenzuspeichern.

Die vier hier vorgestellten Methoden verwenden als Referenzprozessor den TMS 320C25, ein Vorgängermodell des in dieser Arbeit zu behandelnden C5x. Im Kapitel

[1] Ausdrucksbaum - Erhält man durch Datenflußanalyse aus dem zu kompilierenden Programm.

5 wird auf die Unterschiede zwischen den beiden Prozessortypen eingegangen, bzw. wie man diese Unterschiede in die Methoden zur Codegenerierung einfließen lassen kann und was sich daraus für Vorteile ergeben.

4.1 Instruktionsauswahl und Registerzuweisung

Wie auch schon im Kapitel 2 erwähnt, besitzt der C25, ebenso wie der C5x, eine heterogene Architektur. Heterogen bedeutet hierbei, daß ein Register, in dem der Operand einer Instruktion erwartet wird, nicht gleichzeitig das Register ist, in welches das Ergebnis der Operation geschrieben wird. Bei homogenen Architekturen, wie man sie in der Regel bei general-purpose Prozessoren antrifft, kann Quelle und Ziel einer Operation das gleiche Register oder Registerfile sein. Quell- und Zielregister sind in diesem Fall unabhängig von der auszuführenden Operation. Man kann daher für homogene Architekturen die Auswahl von Instruktionen unabhängig von der Wahl der Quell- und Zielregister behandeln.

Beim C25 und C5x ist dies durch seine heterogene Architektur anders. Quell- und Zielregister sind fest mit der auszuführenden Operation verbunden.

Zwei Beispiele:

1. Der Multiplikationsbefehl MPY:

   Dieser Befehl erwartet einen Operanden im TREG0 (Kapitel 2), den anderen im Speicher. Das Ergebnis steht nach der Operation in Produktregister (PREG).

   Um also einen Befehl der Form $x = y * z$ auszuführen, muß ein Operand, entweder y oder z ins TREG0 gebracht werden und nach der Multiplikation muß der Inhalt des PREG nach x geschrieben werden.

2. Der Additionsbefehl APAC (addiere PREG zum ACCU).

   Die beiden Operanden werden im Produktregister (PREG) und im Akkumulator erwartet. Das Ergebnis steht wiederum im Akkumulator. Stehen die Summanden aber im Speicher, so müssen sie erst ins PREG bzw. in den ACCU transferiert werden.

Es reicht also nicht aus, eine günstige Instruktion zu wählen und dann die Operanden irgendwie in die festgelegten Register zu transferieren. Vielmehr müssen die Kosten, die bei Registertransferoperationen entstehen, in die Instruktionsauswahl mit einbezogen werden, um wirklich eine kostenoptimale Auswahl an Prozessorinstruktionen für einen gegebenen Ausdruck zu erhalten.

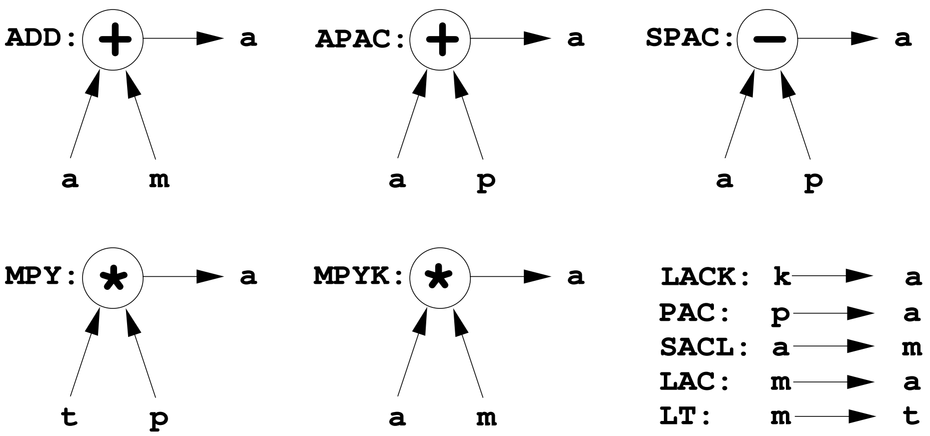

Abbildung 4.1: Baummuster von Instruktionen des C25

Befehl	Operanden	Ziel	Kosten	Drei-Adress-Form	Index
ADD m	a + m	a	1	a ← a + [m]	1
APAC	a + p	a	1	a ← a + p	2
SPAC	a + p	a	1	a ← a - p	3
MPY m	t + m	p	1	p ← t * [m]	4
MPYK k	t + k	p	1	p ←t * k	5
LACK k	k	a	1	a ← k	6
PAC	p	a	1	a ← p	7
SACL m	a	m	1	[m] ← a	8
LAC m	m	a	1	a ← [m]	9
LT m	m	t	1	t ← [m]	10

Tabelle 4.1: Teil des Befehlssatzes des C25

Die Zuweisung der Register und die Auswahl der Instruktionen kann für eine heterogene Architektur deshalb nur zusammen durchgeführt werden.

In Abbildung 4.1 ist ein Teil des Befehlssatzes des C25 als Baummuster dargestellt. Die verwendeten Bezeichner stehen für : a ⇒ Akkumulator, p ⇒ Produktregister (PREG), t ⇒ Multiplikationsregister (TREG0), m ⇒ Datenspeicher (*Memory*), k ⇒ Konstante

Diese Baummuster korrespondieren mit den Drei-Adress-Befehlen einer Zwischendarstellung in Tabelle 4.1. Man erkennt, daß die Register, die bei einer Instruktion benutzt werden, durch die Instruktion eindeutig festgelegt sind. Optimaler Code hängt somit auch davon ab, wie Operanden durch den Datenpfad dahin gebracht werden, wo die Instruktionen sie erwarten.

4.1.1 Problemdefinition

Das Problem der optimalen Befehlsauswahl und Registerzuweisung besteht nun darin, mit den zur Verfügung stehenden Baummustern (s. Abbildung 4.1) einen gegebenen Ausdrucksbaum mit den geringsten Kosten zu überdecken. Man muß dabei beachten, daß nicht nur die Kosten einer Instruktion berücksichtigt werden, sondern auch die Kosten, um die Operanden in die richtigen Register zu transferieren.

4.1.2 Lösung

Zur Lösung dieses Problems wird in [9] ein Generator für Codegeneratoren vorgeschlagen. Ein solcher Generator ist OLIVE *(Kapitel 3.2)*. OLIVE verlangt als Eingabe eine Prozessorbeschreibung, die in Form einer Grammatik vorliegt (Kapitel 3.2.2) und generiert daraus einen Baumparser für den Prozessor. Die Drei-Adress-Form aus Tabelle 4.1 wird dazu in eine Prozessorbeschreibung konvertiert. In Tabelle 4.2 findet man die OLIVE-Beschreibung der Befehle aus Tabelle 4.1, incl. der Assembler-Notation. Diese Tabelle ist jedoch nur insoweit vollständig, daß man die Kommutativität einiger Operationen nicht berücksichtigt hat. Die Addition a + b kann ebenso durch b + a dargestellt werden, d.h. PLUS (a, m) = PLUS (m, a). Für einen gegebenen Ausdrucksbaum kann aber auf Grund der internen Struktur das eine Muster günstiger sein als das andere. Es ist deshalb nötig, für alle kommutativen Operationen, wie +, *, UND oder ODER, beide möglichen Baummuster zu beschreiben.

Der aus dieser Beschreibung generierte Parser erhält als Eingabe den zu überdeckenden Ausdrucksbaum und liefert die kostenoptimale Überdeckung für diesen Baum. Der große Nutzen von OLIVE ist hierbei, daß Registertransferoperationen, die benötigt werden, um die Operanden in die richtigen Register zu bringen, ebenfalls in der OLIVE-Beschreibung modelliert werden können. So wird die Befehlsauswahl und die Registerallokation zusammen durchgeführt. Dies wird dadurch erreicht, daß alle Register im Datenpfad des Prozessors, in denen Werte gespeichert werden können bzw. in denen Operanden erwartet werden, als Nicht-Terminal Symbole der Grammatik auftauchen. Es wird durch entsprechende Grammatikregeln sichergestellt, daß Registertransferoperationen inklusive der entstehenden Kosten mit berücksichtigt werden.

Das Verfahren, das in OLIVE zur Instruktionsauswahl benutzt wird, ist eine Variation eines Verfahrens, das von Aho und Johnson [2] vorgestellt und als optimal bewiesen wurde. Optimal ist es jedenfalls dann, wenn man außer acht läßt, daß durch eine ungünstige Reihenfolge in der Abarbeitung der Teilbäume eines DFT überflüssige Speicherzugriffe, sog. "*Memory spills*", entstehen können. Das Problem und eine Methode zur Vermeidung dieser "*Memory spills*" werden im folgenden Abschnitt erläutert.

Ziel		Baummuster		Befehl
a	:	PLUS (a , m)	{ } = { } ;	(ADD m)
a	:	PLUS (a , p)	{ } = { } ;	(APAC)
a	:	MINUS (a , p)	{ } = { } ;	(SPAC)
p	:	MULT (m , t)	{ } = { } ;	(MPY m)
p	:	MULT (t , CONST)	{ } = { } ;	(MPYK k)
a	:	CONST	{ } = { } ;	(LACK k)
a	:	p	{ } = { } ;	(PAC)
m	:	a	{ } = { } ;	(SACL m)
a	:	m	{ } = { } ;	(LAC m)
t	:	m	{ } = { } ;	(LT m)

Tabelle 4.2: OLIVE-Beschreibung der Befehle aus Tabelle 4.1

4.2 Scheduling

Um optimalen Code zu erzeugen, reicht es nicht aus, eine optimale Instruktionsauswahl für einen Ausdrucksbaum durchzuführen. Trotz optimaler Instruktionsauswahl kann es passieren, daß gerade berechnete Teilergebnisse in einem Register stehen, das von der nächsten Operation wieder überschrieben wird, obwohl es noch an anderer Stelle benötigt wird. Damit diese Teilergebnisse nicht verloren gehen, muß man sie irgendwo retten, was wiederum Kosten verursacht. Diese Speicherzugriffe, die durch einen Registerkonflikt produziert werden, nennt man "*Memory Spills*" und sie sind nicht mit den Speicherzugriffen gleichzusetzen, die während der Instruktionsauswahl durch den Baumparser eingefügt wurden. Die Zugriffe, die durch OLIVE produziert wurden, sind alleine durch die Registerstruktur des Zielprozessors bedingt und lassen sich keinesfalls vermeiden. Es wird also eine Methode benötigt, die die Memory spills so weit wie möglich minimiert oder sogar vollständig vermeidet, denn jeder Zugriff auf den Hauptspeicher kostet Zeit.

Um Memory spills zu vermeiden, muß man eine geeignete Reihenfolge (*schedule*) finden, in der die einzelnen Teilbäume eines Ausdrucksbaums abgearbeitet werden. In [2] wurde von Aho und Johnson für eine breite Klasse von Prozessoren ein Schedule vorgeschlagen, daß auf deren ***Strong Normal Form Theorem*** basiert. Dieses Theorem besagt, daß, für eine homogene Registerarchitektur, jede optimale Reihenfolge innerhalb einer Codesequenz in eine *"Strong normal form (SNF)"* transformiert werden kann. Eine Codesequenz ist in *SNF*, wenn alle Code-Teilsequenzen, die nur durch Speicherzugriffe getrennt werden, durch ein *Strongly Continguous (SC)* schedule gebildet werden. *Strongly Continguous* bedeutet, daß für die Code-Untersequenz gilt, daß an jedem gewählten Baumknoten m mit den Teilbäumen T_1 bzw. T_2 gilt, daß erst Teilbaum T_1, dann Teilbaum T_2 und dann das Muster für m bearbeitet wird. In [7] wurde von Wess eine Heuristik für das Scheduling vorgestellt,

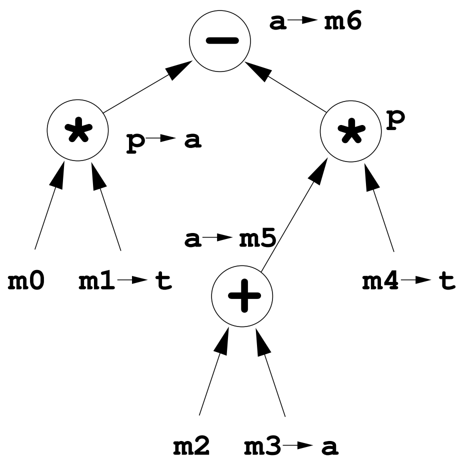

Abbildung 4.2: Überdeckter Ausdrucksbaum

die *SNF* benutzt.

4.2.1 Problembeschreibung

SC-Schedules sind für heterogene Architekturen aber nur sehr eingeschränkt zu gebrauchen. Die Qualität des produzierten Codes hängt sehr stark davon ab, ob die Teilbäume von Rechts nach Links, oder von Links nach Rechts ausgewertet werden. Die Reihenfolge, die für die Wurzel eines Teilbaums optimal ist, kann im ersten Teilbaum schon einen unerwünschten Memory spill produzieren.

Wann können überhaupt Memory spills produziert werden ?

Gegeben sei ein Ausdrucksbaum M, der die Phase der Instruktionsauswahl bereits durchlaufen hat. In M gebe es einen Teilbaum M' mit der Wurzel m' und den beiden Teilbäumen von m', T_1 und T_2. Ein Memory spill wird genau dann produziert, wenn das Ergebnis von Teilbaum T_1 in einem Register steht, das in T_2 ebenfalls verwendet wird. In diesem Fall muß nach der Auswertung von T_1 der Inhalt des Registers gerettet werden, da es sonst durch T_2 überschrieben würde.

Abbildung 4.2 zeigt einen überdeckten Ausdrucksbaum. Die Knoten sind entweder direkt mit den Registern des Prozessors markiert, in denen die Resultate der Operationen stehen, bzw. mit einer, der eigentlichen Operation folgenden, Registertransferoperation, die von OLIVE eingefügt wurde.

Man sieht, daß nach Ausführung des linken Teilbaums das Ergebnis der Multiplikation von m_0 und m_1 nach a kopiert wird, wo es für die Subtraktion gebraucht wird. Der linke Unterbaum des rechten Teilbaums benötigt jedoch ebenfalls das Register a für die Addition von m_2 und m_3.

Left First Schedule			Right First Schedule			Optimal Schedule		
LT	m1	t←[m1]	LT	m4	t ← [m4]	LAC	m3	a ← [m3]
MPY	m0	p←t∗[m0]	LAC	m3	a ← [m3]	ADD	m2	a←a+[m2]
PAC		a←p	ADD	m2	a←a+[m2]	SACL	m5	[m5]←a
SACL	m7	[m7]←a	SACL	m5	[m5]←a	LT	m1	t←[m1]
LAC	m3	a ← [m3]	MPY	m5	p←t∗[m5]	MPY	m0	p←t∗[m0]
ADD	m2	a←a+[m2]	LT	m1	t ←[m1]	PAC		a←p
SACL	m5	[m5]←a	PAC		a ← p	LT	m4	t←[m4]
LT	m4	t←[m4]	SACL	m7	[m7] ← a	MPY	m5	p←t∗[m5]
MPY	m5	p ←t∗[m5]	MPY	m0	p←t∗[m0]	SPAC		a←a−p
LAC	m7	a←[m7]	PAC		a ← p	SACL	m6	[m6]←a
SPAC		a←a−p	LT	m7	t←[m7]			
SACL	m6	[m6]←a	MPYK	1	p←t∗1			
			SPAC		a ← a −p			
			SACL	m6	[m6] ← a			

Tabelle 4.3: Mögliche Schedules für Abb. 4.2

Man kann den kompletten Baum also nicht nach dem SC-Schema *left-first* auswerten, ohne ein Memory spill zu erzeugen. Ebenso ist es nicht möglich, nach dem SC-Schema *right-first* vorzugehen. In diesem Fall stünde das Ergebnis des rechten Teilbaums im Register p, p wiederum wird aber vom linken Teilbaum benutzt, muß also ebenfalls gerettet und später wieder neu geladen werden.

Der richtige Weg ist in diesem Fall also eine Kombination aus beiden Schemata. Das Ergebnis von $m_2 + m_3$ muß auf Grund der folgenden Multiplikation auf jeden Fall gerettet werden, normalerweise also im Speicher abgelegt werden. Führt man danach die Multiplikation von m_0 und m_1 durch und dann die Multiplikation mit m_4, so belegt man zu keiner Zeit ein Register, daß später benutzt wird. Die Codesequenzen, die sich aus den drei beschriebenen Schedules ergeben, kann man in Tabelle 4.3 sehen.

An diesen drei Schedules kann man deutliche Unterschiede in der Codequalität erkennen, allein die Codelänge ist ein gutes Indiz für besseren Code.

Kann man aber für jeden Ausdrucksbaum ein optimales Schedule finden?

Die Antwort darauf wird im folgenden Abschnitt gegeben. Die vorgestellte Lösung wurde von Araujo und Malik in [13] beschrieben. Araujo und Malik entwerfen eine Lösung für eine Klasse von heterogenen Architekturen, die einem [1,∞] Modell entsprechen, d.h. Orte, an denen Daten gespeichert werden können, haben entweder die Kapazität 1, es kann genau ein Wert gespeichert werden, oder sie haben die Kapazität ∞, die Anzahl der Speicherplätze ist nicht begrenzt.

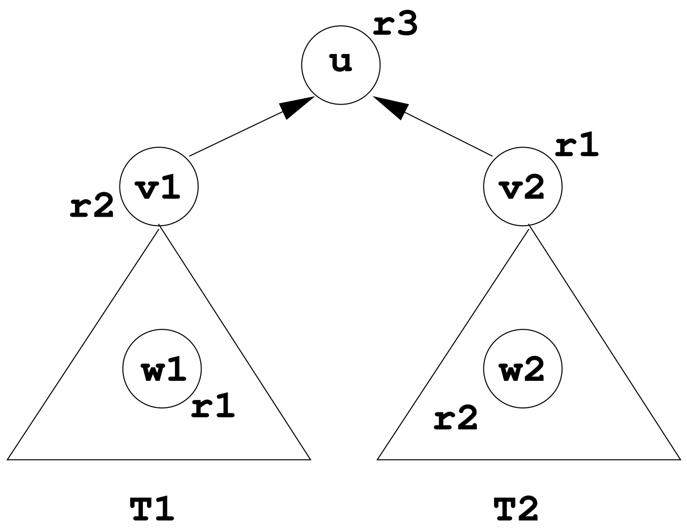

Abbildung 4.3: Allocation Deadlock im Ausdrucksbaum

4.2.2 Problemlösung

Zuerst wird gezeigt, welche Bedingungen eine heterogene Architektur erfüllen muß, um Schedules ohne Memory spills zu ermöglichen (Kap. 4.2.2.1). Danach wird das Konzept des *Register Transfer Graphen* (*RTG*) eingeführt und die Bedeutung für die Codegenerierung aufgezeigt (Kapitel 4.2.2.2). Abschließend wird ein Algorithmus für DSP's mit azyklischem RTG vorgestellt (Kapitel 4.2.2.3), mit dem sich optimale Schedules berechnen lassen. Diese Methode wurde in [13] beschrieben.

In den folgenden Abschnitten werden einige Definitionen benötigt (s. auch Abbildung 4.3):

- T sei ein Ausdrucksbaum mit unären und binären Operationen
- $L : T \to R \cup M$ sei eine Funktion, die die Knoten von T auf die Menge $R \cup M$ abbildet. $R = \{r_i, 1 \leq i \leq N\}$ ist eine Menge, die aus den $|N|$ Registern des Zielprozessors besteht und M ist die Menge aller möglichen Speicherplätze im Hauptspeicher. Beispiel : $L(v) = r$ bedeutet, das Ergebnis der Operation im Knoten v, steht nach der Ausführung in Register r.
- u sei die Wurzel eines Ausdrucksbaumes mit den Kind-Knoten v_1 und v_2. Man nehme an, daß, nach erfolgter Registerallokation (Kapitel 4.1), die Register $L(v_1) = r_1$und $L(v_2) = r_2$ den Knoten v_1 und v_2 zugewiesen sind.
- T_1 und T_2 seien Teilbäume mit den Wurzeln v_1 und v_2.

4.2.2.1 Allocation Deadlock

Definition 1 : Ein *Allocation Deadlock* liegt vor, wenn die folgenden Bedingungen erfüllt sind.

1. $L(v_1) \notin M$ und $L(v_2) \notin M$,

   kein Resultat eines Teilbaums T_1 oder T_2 steht im Speicher.

2. $L(v_1) \neq L(v_2)$,

   die Resultate beider Teilbäume stehen nicht im selben Register und

3. es gibt zwei Knoten $w_1$und w_2 mit $w_1 \in T_1$und $w_2 \in T_2$, wobei $L(w_1) = L(v_2)$ und $L(w_2) = L(v_1)$,

   das Register, in dem das Resultat von Teilbaum T_1 steht, wird irgendwo innerhalb von T_2 verwendet und ebenso anders herum.

Die obige Definition wird in Abbildung 4.3 verdeutlicht. Die Register r_1 und r_2, in denen die Resultate der Teilbäume stehen, werden im jeweils anderen Teilbaum benutzt.

Die Definition führt zu folgendem Theorem.

Theorem 1 : Sei T ein Ausdrucksbaum. Wenn T kein Schedule ohne Memory spills besitzt, so gibt es mindestens einen Teilbaum mit einem Allocation deadlock.

Beweis : Wir nehmen an, alle Knoten u in T sind der Gestalt, daß kein Teilbaum T_u einen Allocation deadlock enthält, T selber aber dennoch kein gültiges Schedule besitzt.

Nach Definition 1 liegt *kein* Allocation deadlock vor wenn :

1. $L(v_1) = M$ oder $L(v_2) = M$

   Mindestens ein Resultat eines Teilbaums liegt im Speicher. In diesem Fall wird der entsprechende Teilbaum zuerst ausgewertet und es gibt ein Schedule ohne Memory spills.

2. $L(v_1) = L(v_2)$

   Beide Operanden von u liegen im selben Register. Dieser Fall kann nicht eintreten, da keine binäre Operation beide Operanden aus demselben Register beziehen kann.

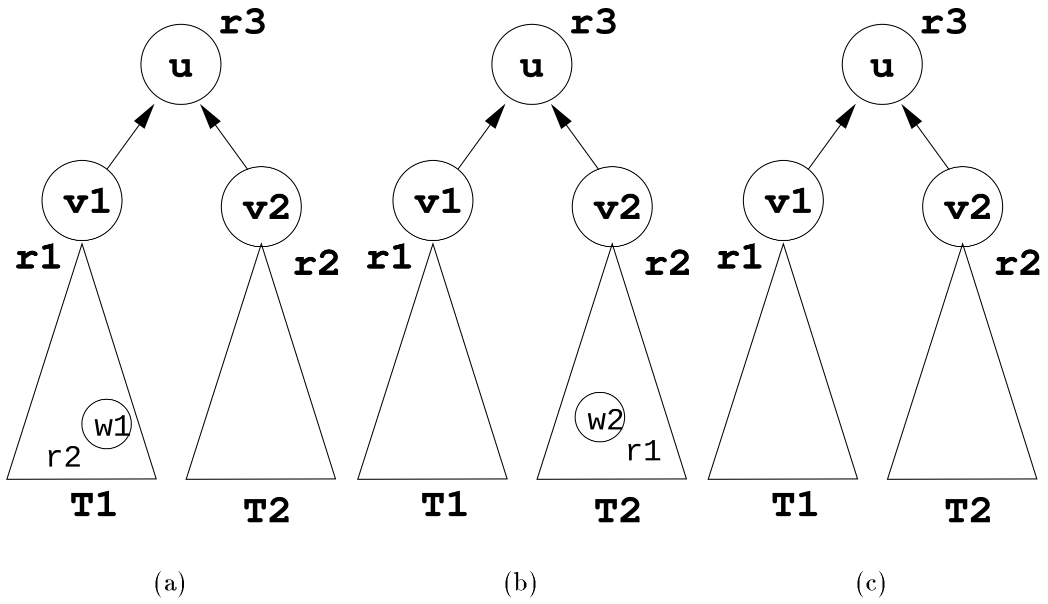

Abbildung 4.4: Bäume ohne Allocation Deadlock

3. $L(v_1) \neq L(v_2)$, $L(w_1) = L(v_2)$, es gibt aber keinen Knoten w_2 mit $L(w_2) = L(v_1)$

   Das Register $L(v_1) = r_1$, das das Resultat des Teilbaumes T_1 enthält, wird im Teilbaum T_2 nicht benutzt (Abbildung 4.4a). In diesem Fall wird Teilbaum T_1 vor T_2 ausgewertet, da durch T_2 das Resultat in r_1 nicht überschrieben wird. Anschließend wird der Knoten u behandelt. Es existiert ein Schedule ohne Memory spills

4. $L(v_1) \neq L(v_2)$, $L(w_2) = L(v_1)$, es gibt aber keinen Knoten w_1 mit $L(w_1) = L(v_2)$

   Das Register $L(v_2) = r_2$, das das Resultat des Teilbaumes T_2 enthält, wird im Teilbaum T_1 nicht benutzt (Abbildung 4.4b). Dieser Fall ist symetrisch zu Fall 3 und entsprechend gibt es auch hier ein Schedule ohne Memory spills.

5. $L(v_1) \neq L(v_2)$ und es gibt keine Knoten w_1 oder w_2

   In keinem Teilbaum wird ein Register benutzt, daß das Resultat des anderen Teilbaums enthält. (Abbildung 4.4c). In diesem Fall können die Teilbäume in beliebiger Reihenfolge bearbeitet werden, jedes beliebige Schedule ist gültig.

Für *alle möglichen* Bäume ohne Allocation deadlock wurde gezeigt, daß ein Schedule ohne Memory spills existiert und damit ist die Annahme, daß es mindestes einen Teilbaum mit einem Allocation deadlock geben muß, widerlegt.

Die Aussage von Theorem 1 läßt sich umformen und man erhält folgende wichtige Aussage :

Korrolar 1 : Wenn ein Ausdrucksbaum *keinen* Allocation deadlock enthält, so *existiert ein* Schedule ohne Memory spills.

Beweis : Durch Umformung von Theorem 1.

Diese Schedule läßt sich mit Hilfe des Beweises zu Theorem 1 sogar berechnen.

4.2.2.2 Das RTG Modell und Theorem

Definition 2 :Der *Register Transfer Graph* (RTG) ist ein gerichteter, markierter Graph, bei dem jeder Knoten ein Register aus dem Datenpfad des Zielprozessors repräsentiert, in dem Daten gespeichert werden können. Eine Kante von r_i nach r_j wird mit den Befehlen markiert, die ihre Operanden aus r_i holen und das Ergebnis in r_j speichern.

Die Knoten des RTG können hier sowohl für Registerfiles als auch für einzelne Register stehen. Register, die genau einen Wert speichern können, werden durch einen Kreis dargestellt, Registerfiles dagegen werden mit einem doppelten Kreis gekennzeichnet. Der Speicher selbst wird im RTG nicht beschrieben, vielmehr stehen abgehende oder eingehende Kanten für Speichertransfers.

Ein RTG heißt azyklisch, wenn der RTG keine Zyklen enthält. Eine Kante, die direkt wieder auf denselben Knoten zurückführt, zählt hier nicht als Zyklus, eine solche Kante wird im folgenden Eigenzyklus genannt.

Abbildung 4.5 zeigt den vollständigen RTG des C25. Die Knoten sind mit den Registern des Prozessors gekennzeichnet : Akkumulator a, Produktregister p und Multiplikationsregister t. Die Kanten des RTG sind mit Befehlen des Prozessors markiert, wobei die Nummern mit den Indizes aus Tabelle 4.1 korrespondieren. Die Markierung ist natürlich nicht vollständig, da nur die Befehle aus Tabelle 4.1 verwendet wurden.

Man erkennt in Abbildung 4.5, daß der RTG des C25 azyklisch ist, denn von keinem Knoten des RTG kann man denselben Knoten wieder erreichen, es sein denn, man geht über den Speicher. Viele andere DSP's haben ebenfalls azyklische RTG's, weshalb für diese Klasse von Prozessoren im folgenden eine Lösung des Schedulingproblems entwickelt wird.

Theorem 2 : Ist der RTG einer Architektur azyklisch, so gibt es für jeden Ausdrucksbaum ein Schedule ohne Memory spills.

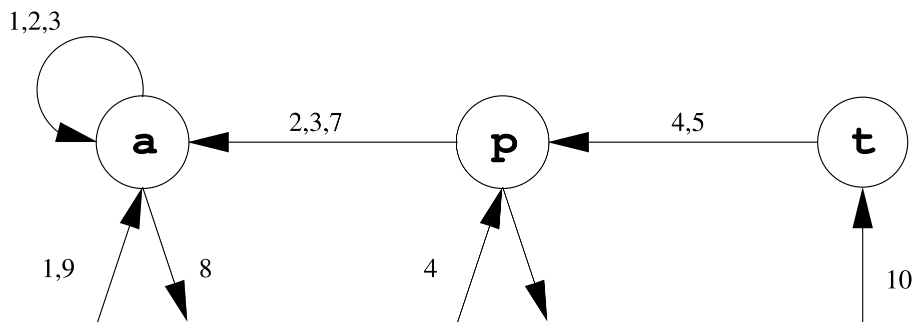

Abbildung 4.5: RTG des TMS320C25

Beweis : Sei T ein Ausdrucksbaum mit der Wurzel u. $v_1$und v_2 seien Kinder von T mit $L(u) = r_3$, $L(v_1) = r_1$ und $L(v_2) = r_2$. T_1 und T_2 seien Teilbäume von T mit den Wurzeln v_1 und v_2. P_k seien Teilbäume von T mit der Wurzel p_k. Die Ergebnisse der Operationen aller p_k werden im Speicher abgelegt, also $L(p_k) \in M$ für alle k. Definiere Q_i $(i = 1, 2)$ als die Teilbäume, die übrigbleiben, wenn alle P_k aus den Teilbäumen T_i entfernt werden, $Q_i = T_i/P_k$. Es wird gezeigt, daß immer eine Reihenfolge für P_k, Q_i und u gefunden werden kann, mit der keine Memory spills entstehen.

Man muß zwei Fälle betrachten:

1. T enthält keinen Allocation deadlock.

   Nach Korrolar 1 (S. 46) existiert ein Schedule ohne Memory spills.

2. T enthält einen Allocation deadlock.

   Man nehme an, der Allocation deadlock wird durch die Register r_1 und r_2 verursacht. r_1 und r_2 sind die Register, die die Ergebnisse der Teilbäume T_1 und T_2 enthalten. Nehmen wir nun an, der RTG enthält einen Pfad von r_2 nach r_1. Dann muß *jeder* Knoten in T_2, der dem Register r_1 zugewiesen ist, auf dem Weg zur Wurzel des Teilbaums T_2, seinem Vorfahr v_2, dem das Register r_2 zugeordnet ist, irgendwann einen Knoten durchlaufen, der dem Speicher zugewiesen ist. Dies kommt dadurch, daß der azyklische RTG den Pfad von r_2 nach r_1 enthält, jeder Pfad von r_1 nach r_2 muß also durch den Speicher führen. Wenn nun also die Teilbäume P_k in T_2 zuerst bearbeitet werden, so ist der Deadlock aufgelöst ohne Memory spills einzuführen, denn Q_2 kann keine Knoten mehr besitzen, die das Register r_1 überschreiben würden. Alle Teilbäume P_k, in denen das Register r_1 benutzt wird und die in T_2 liegen, wurden bearbeitet und die Resultate liegen im Speicher. Nun haben wir nur noch die Teilbäume T_1 und Q_2 und den Knoten u, die keinen Deadlock mehr enthalten und nach Korrolar 1 existiert damit ein Schedule ohne Memory spills.

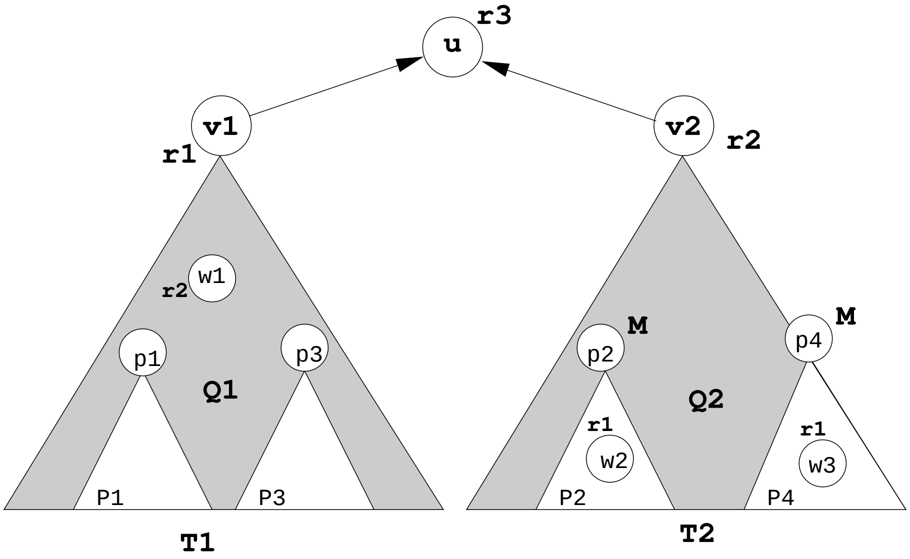

Abbildung 4.6: Das RTG Theorem

Der Beweis soll anhand des Beispiels in Abbildung 4.6 verdeutlicht werden.

Der gezeigte Ausdrucksbaum T besitzt die Wurzel u und die Teilbäume T_1 mit Wurzel v_1 und T_2 mit Wurzel v_2. $L(v_1) = r_1$ und $L(v_2) = r_2$. P_k, $k \in \{1, 2, 3, 4\}$ sind Teilbäume, deren Resultate in den Speicher geschrieben werden. Q_1 ist der Teilbaum, den man erhält, wenn man P_1 und P_3 aus T_1 entfernt, ebenso ist Q_2 der Teilbaum, den man erhält, wenn man P_2 und P_4 aus T_2 entfernt. Q_1 und Q_2 sind in Abbildung 4.6 schraffiert. Wenn nun zunächst die Teilbäume P_2 und P_4 bearbeitet werden, existiert in Q_2 kein Knoten mehr, der das Ergebnis seiner Operation in r_1 ablegt. Danach kann man den kompletten Teilbaum T_1 abarbeiten, es wird kein relevantes Ergebnis überschrieben, da ja alles, was bisher berechnet wurde, im Speicher liegt. Das Ergebnis von T_1 liegt schließlich in r_1. Es kann nun ohne Probleme Q_2 und zum Schluß die Wurzel von T bearbeitet werden.

Auf dem Beweis zu Theorem 2 läßt sich ein Algorithmus aufbauen, der für einen Prozessor mit azyklischem RTG für jeden Ausdrucksbaum ein optimales Schedule berechnet.

4.2.2.3 Algorithmus für optimales Scheduling

Der hier vorgestellte Algorithmus (Abbildung 4.7) basiert auf dem Beweis von Theorem 2 (S. 46). Die Eingabe für den Algorithmus ist ein Ausdrucksbaum T, der die Phase der Instruktionsauswahl und der Registerallokation bereits durchlaufen hat.

Der Algorithmus produziert für diesen Ausdrucksbaum eine Codesequenz ohne Memory spills.

Im ersten Durchlauf werden mit der Prozedur GETUSAGE für jeden Knoten v des Baumes T zwei Mengen berechnet :

- Memory-Set und
- Register-Set

Die Menge Memory-Set besteht für alle Knoten u aus den Knoten des Teilbaums T' mit u als Wurzel, die dem Speicher zugeordnet wurden. Für unser Beispiel aus Abbildung 4.6 besteht die Menge Memory-Set($Wurzel(T)$) aus den Knoten p_i, $i = 1, 2, 3, 4$.

Die Menge Register-Set besteht für jeden Knoten aus allen Registern, die im Teilbaum T' mit u als Wurzel verwendet werden. Wird ein Knoten mit einem Speicherzugriff entdeckt, so fängt von dieser Stelle an die Menge Register-Set wieder mit der leeren Menge an. Alles was vor diesem Knoten war, also in dem Teilbaum der mit diesem Speicherzugriff endet, spielt für den Weg bis zur Wurzel keine Rolle mehr. Der Teilbaum mit dem Speicherzugriff wird nämlich bearbeitet, bevor die Menge Register-Set des verbleibenden Teilbaums für das Scheduling benötigt wird. Für das Beispiel aus Abbildung 4.6 bedeutet dies, daß die Menge Register-Set(v_1) alle verwendeten Register aus Teilbaum Q_1 und die Menge Register-Set(v_2) alle verwendeten Register aus Teilbaum Q_2 enthält.

Im zweiten Durchlauf wird die Prozedur `OptSchedule` gestartet. Diese Prozedur führt zwei Funktionen aus. Zuerst werden alle Knoten, die in der Menge Memory-Set enthalten sind, rekursiv bearbeitet. Es ist ja theoretisch möglich, daß innerhalb eines Teilbaums P_i, dessen Ergebnis in den Speicher geschrieben wird, wiederum einen Allocation deadlock enthält, etwa in Teilbaum P_i'. Der Allocation deadlock in P_i' muß natürlich zuerst aufgelöst werden, bevor der Deadlock in P_i aufgelöst werden kann. Ist ein Teilbaum gefunden, für den die Menge Memory-Set leer ist, wird die Funktion `FreeSchedule` aufgerufen. In der Funktion `FreeSchedule` wird nun das richtige Schedule für einen Teilbaum ohne Allocation Deadlock errechnet. Der wichtige Punkt in dieser Funktion ist die korrekte Reihenfolge der Teilbäume. Dafür sorgt die Zeile `v`$_1$ `= unique(children(u))`. Die Funktion `unique` gibt die Kinder in der Reihenfolge zurück, daß kein Registerkonflikt entsteht. Um das zu erreichen, werden die Mengen Register-Set der Kinder von u ausgewertet. Sind schließlich alle Knoten aus Memory-Set bearbeitet, können die verbliebenen Restbäume, im Beispiel Q_1 und Q_2, ebenfalls mit der Funktion `FreeSchedule` bearbeitet werden und anschließend die Wurzel von T selbst.

Theorem 3 : Der Algorithmus `OptSchedule` liefert ein optimales Schedule.

Beweis : Der Beweis ist trivial, da mit dem Algorithmus OptSchedule der Beweis zu Theorem 2 implementiert wurde.

Der Algorithmus `OptSchedule` wurde, wie alle noch folgenden Algorithmen, in einer Pseudo-Sprache geschrieben, die leichter zu verstehen ist, da programmiersprachliche Besonderheiten nicht berücksichtigt werden müssen.

Für das Beispiel aus Abbildung 4.6 ergibt der Algorithmus OptSchedule folgendes Schedule ohne Memory Spills : P_1 , P_2, P_3, P_4, Q_1, Q_2 , u.

Die in diesem Abschnitt getroffenen Aussagen beziehen sich vielfach darauf, daß zwischen zwei Registern, die als Ziel oder Quelle an einer Operation beteiligt sind, genau ein Pfad im RTG existiert. Dies impliziert allerdings, daß die Register, die im RTG des C25 modelliert sind, jeweils genau einen Wert speichern können. Der Speicher dagegen wird als ∞ angenommen, so daß jede Speicheroperation verschiedene, unterscheidbare Zellen im Hauptspeicher anspricht.

4.3 Araujo und Malik's Heuristik für DAG's

Programme oder Funktionen sind normalerweise durch Datenflußanalyse nicht direkt in Bäume umzusetzen, sondern sie lassen sich durch gerichtete, azyklische Graphen (DAG - directed acyclic graph) darstellen (Abbildung 4.11). Das Problem der Überdeckung eines kompletten DAG wurde als NP-vollständig bewiesen [8]. Um aber dennoch die Methoden der vorangegangenen Abschnitte nutzen zu können, muß der DAG in einzelne Bäume zerlegt werden. Dies erreicht man durch Aufbrechen von Kanten im DAG. Welche Kanten kann man aber aufbrechen, ohne daß sich die Qualität des erzeugten Codes dadurch verschlechtert? Das Kapitel 4.3 und das Kapitel 4.4 geben zwei unterschiedliche Antworten darauf.

4.3.1 Problembeschreibung

Das Aufbrechen einer Kante eines DAG's impliziert, daß das Ergebnis des Knotens, von dem die aufzubrechende Kante ausgeht, gespeichert und später wieder neu eingelesen werden muß. Das Abspeichern des Zwischenergebnisses geschieht in der Regel im Hauptspeicher, kann aber ebenso in jedem geeigneten Register im Datenpfads geschehen. In der Heuristik von Araujo und Malik kommt das im vorigen Abschnitt eingeführte Konzept des RTG zum Einsatz. Mit Hilfe der RTG-Informationen kann man Kanten erkennen, die auf jeden Fall einen Speicherzugriff erzeugen.

Zunächst soll das Problem, das beim Aufspalten von DAG's entstehen kann, am Beispiel in Abbildung 4.8 noch einmal genauer erläutert werden.

```
GetUsage(u)
    begin
      memory-set(u) = ∅;
      register-set(u) = ∅;
      if match(u) is not memory
        register-set(u) = match(u);
      foreach v in children(u)
        GetUsage(v);
        if match(v) is memory
          memory-set(u) = memory-set(u) ∪ {v};
        else
          memory-set(u) = memory-set(u) ∪ memory-set(v);
          register-set(u) = register-set(u) ∪ register-set(v);
        endif
      endfor
    end

    OptSchedule(u)
    begin
      foreach p in memory-set(u)
        OptSchedule(p);
      foreach v in children(u)
        FreeSchedule(v);
      emit(u);
    end

    FreeSchedule(u)
    begin
      if match(u) is memory
        return;
      if u is not a leaf
        v1 = unique(children(u));
        foreach w in children(v1)
          FreeSchedule(w);
      endif;
      emit_code(u);
    end
```

Abbildung 4.7: Algorithmus OptSchedule

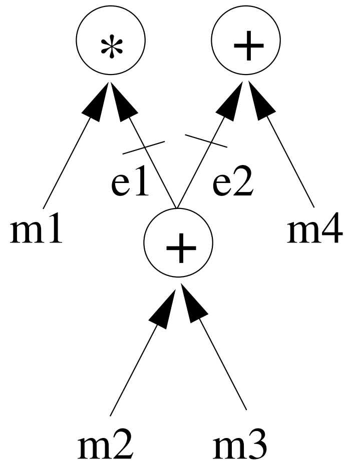

Abbildung 4.8: DAG mit zwei Kanten zum Aufspalten

Der gezeigte DAG besitzt einen Knoten mit zwei abgehenden Kanten, e_1 und e_2. Diese beiden Kanten stellen die beiden Möglichkeiten dar, den Graph in Bäume zu zerlegen. Die Qualität des resultierenden Codes für diesen Graph ist jedoch davon abhängig, welche der beiden Kanten tatsächlich aufgebrochen wird. Die beiden Codesequenzen, die man durch das Brechen der Kanten erhält, sind in Tabelle 4.4 aufgeführt.

Fall 1 : Kante $e_1$wird aufgebrochen.

Das Ergebnis der Addition von m_2 und m_3 muß temporär gespeichert werden, da es keine Möglichkeit im Befehlssatz des C25 gibt, den Akkumulator als Operand für die folgende Multiplikation zu verwenden (s. Abbildung 4.5). Der Weg der Daten führt also in jedem Fall durch den Speicher, in diesem Fall m_5. Die zweite Addition mit m_4 kann danach direkt ausgeführt werden. Zum Schluß folgt die Multiplikation von m_1 und m_5.

Fall 2 : Kante e_2 wird gebrochen.

Das Ergebnis der Addition von m_2 und m_3 wird genau wie in Fall 1 in m_5 zwischengespeichert, da es für die Multiplikation im Speicher stehen muß. Nach der Multiplikation folgt dann die zweite Addition. Da diese aber jetzt einen selbstständigen Teilbaum bildet, kann der Baumparser nicht wissen, daß das Ergebnis der ersten Addition ja eigentlich noch im Akkumulator steht. Es wird eine Lade-Operation, LAC m_5, eingefügt, die das Resultat wieder in den Akkumulator bringt. Erst dann kann die Addition mit m_4 durchgeführt werden.

Kante e_1			Kante e_2		
LAC	m2	a←[m2]	LAC	m2	a←[m2]
ADD	m3	a ←a+[m3]	ADD	m3	a ←a+[m3]
SACL	m5	[m5]←a	SACL	m5	[m5]←a
ADD	m4	a←a+ [m4]	LT	m1	t←[m1]
LT	m1	t←[m1]	MPY	m5	p←t*[m5]
MPY	m5	p←t*[m5]	LAC	m5	a←[m5]
			ADD	m4	a←a+ [m4]

Tabelle 4.4: Codesequenzen nach Spalten einer Kante

4.3.2 Problemlösung

Die in [9] vorgestellte Heuristik (Kapitel 4.3.3.1) zur Lösung des Problems besteht aus vier Phasen :

1. teilweise Zuordnung der Register,
2. hinzufügen von Architekturinformationen um die Kanten des DAG zu kennzeichnen,
3. anhand der Klassifizierung der Kanten die günstigsten Kanten aufbrechen und
4. festlegen der Reihenfolge der entstandenen Teilbäume.

4.3.2.1 Teilweise Zuordnung der Register

Bei heterogenen Architekturen werden die Register, in denen Resultate von Operationen gespeichert werden, durch die Operation selbst festgelegt. In der ersten Phase des Algorithmus für die Aufteilung von DAG's werden daher alle Knoten des DAG mit den Informationen über das Zielregister der auszuführenden Operation markiert. Es wird keine komplette Registerzuordnung durchgeführt, wie in Kapitel 4.1 beschrieben. Es werden nur die Zielregister sämtlicher Operationen, die in den Knoten des DAG kodiert sind, ermittelt, Transferoperationen zwischen den Registern bleiben unberücksichtigt. Nehmen wir als Beispiel die Operationen *ADD* und *MPY*. Die Operation *ADD* legt ihr Ergebnis *immer* im Akkumulator ab und die Operation *MPY* legt das Resultat *immer* im Produktregister ab.

Die Markierungen der Knoten werden benutzt, um die Kanten des DAG in verschiedene Klassen einzuteilen.

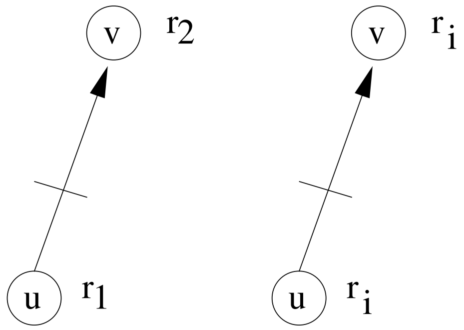

Abbildung 4.9: Natural Edges

4.3.2.2 Natural Edges

Wie bereits in Abbildung 4.8 durch Kante e_1 gezeigt, gibt es Kanten im DAG, die auf jeden Fall einen Speicherzugriff erzwingen.

Seien r_1 und r_2 ein Registerpaar im Datenpfad eines Prozessors mit azyklischen RTG. Weiterhin sei $L : D \rightarrow R \cup M$ die Funktion, die die Knoten von einem DAG D anhand der in den Knoten enthaltenen Operation auf die Menge $R \cup M$ abbildet, R ist die Menge aller Prozessorregister und M eine Menge von Speicherplätzen. Also z.B. $L(ADD) = Akkumulator$ oder $L(MPY) = Produktregister$.

Definition : *Natural Edges* sind alle die Kanten (u, v) eines DAG mit $L(u) = r_1$ und $L(v) = r_2$, für die es aber keinen entsprechenden Pfad von r_1 nach r_2 im RTG des Prozessors gibt. Ebenso sind alle Kanten, die von einem Register r wieder zu sich selbst zurückführen, *Natural Edges*, wenn r im RTG keinen Eigenzyklus (s. Abbildung 4.9) besitzt.

Beispiel : Auf Grund des azyklischen RTG ist beim C25, wie auch beim C5x, jede Kante, die von a nach p führt und jede Kante, die von p nach p führt, eine Natural Edge. Ein Aufbrechen einer solchen Kante würde also keine zusätzlichen Speicheroperation bedeuten, da die Architektur des Prozessor diese Speicheroperation erzwingt und diese während der Instruktionsauswahl auf jeden Fall eingefügt wird.

Als Ergebnis kann man feststellen, daß die Qualität der resultierenden Codes durch das Aufbrechen von Natural Edges nicht verschlechtert wird. Natural Edges werden, wie in Abbildung 4.9, im folgenden mit einem einfachen Querstrich markiert.

4.3.2.3 Pseudo-Natural Edges

In DAG's gibt es häufig Konstellationen, in denen die Interaktion zweier Kanten im DAG dafür verantwortlich ist, daß eine von beiden Kanten durch den Speicher laufen muß. Das heißt, daß für ein Kantenpaar sicher ist, daß eine der Kanten eine Speicheroperation erzeugt.

Definition : *Pseudo-Natural Edges* sind die Kanten eines Kantenpaares, die folgende Bedingungen erfüllen :

1. beide Kanten enden in demselben Knoten des DAG,
2. die Startknoten beider Kanten werden durch die Abbildung L auf das gleiche Register r abgebildet.

   Fall 1 : Auch der Endknoten beider Kanten wird auf dasselbe Register wie die Startknoten abgebildet. Nur wenn das Register r im RTG des Prozessors einen Eigenzyklus enthält, kann eine der beiden Kanten durch diesen Eigenzyklus von r abgedeckt werden, der Pfad für die andere Kante muß über den Speicher führen. Enthält der RTG für r keinen Eigenzyklus, sind beide Kanten per Definition Natural Edges.

   Fall 2 : Der Endknoten beider Kanten wird nicht auf dasselbe Register abgebildet, wie die Startknoten und es existiert im RTG genau ein Pfad, mit dem die Kanten überdeckt werden können. Die zweite Kante kann nur von einem Pfad der RTG überdeckt werden, der durch den Speicher führt.

Beispiel : Für den C25 gibt es die drei folgenden Konstellationen mit Pseudo-Natural Edges.

1. $r_i = AKKUMULATOR$ in Abbildung 4.10 links.
2. $r_i = AKKUMULATOR$ und $r_j = PRODUKTREGISTER$ in Abbildung 4.10 rechts.

Pseudo-Natural Edges sind ebenfalls potentielle Kandidaten zum Aufspalten des DAG. Es gibt aber immer zwei Kanten, an denen man den DAG aufspalten kann. Die falsche Wahl kann zu schlechterem Code führen. In der Phase der Codegenerierung, in der der DAG aufgeteilt wird, stehen aber keine weiteren Informationen zur Verfügung, mit denen man die optimale Kante erkennen kann.

Als Beispiel kann man sich an dieser Stelle Abbildung 4.11 ansehen. Alle Knoten sind mit den Zielregistern ihrer Operation markiert. Natural Edges sind mit einem

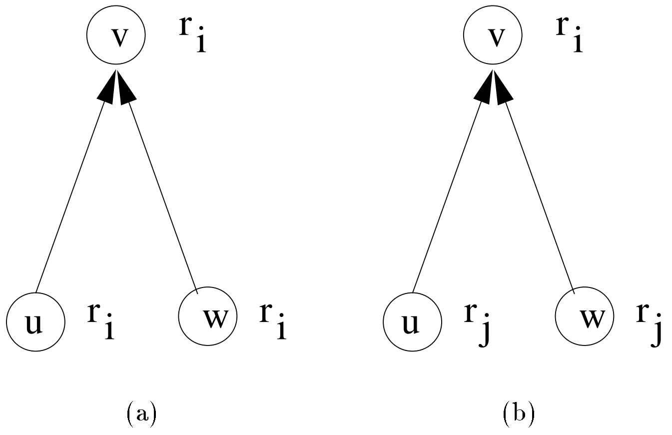

Abbildung 4.10: Pseudo Natural Edges

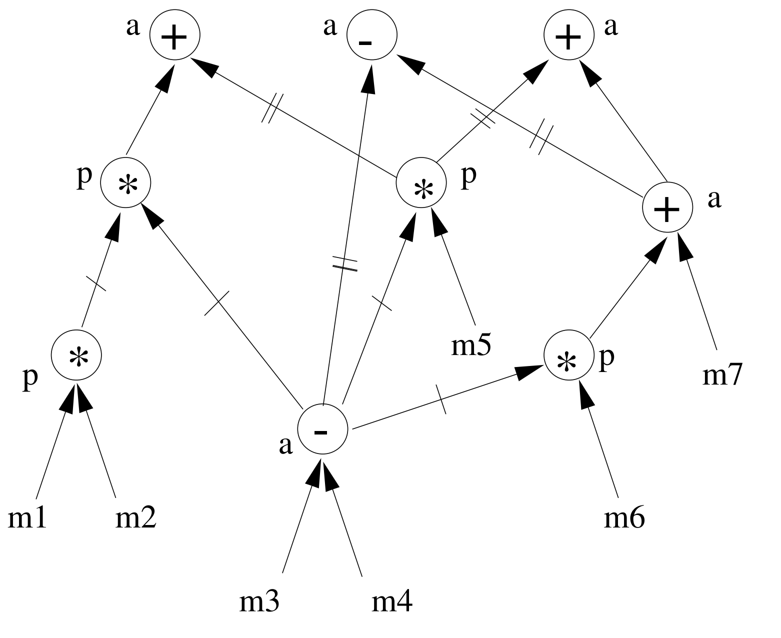

Abbildung 4.11: Ausdrucks DAG nach teilweiser Registerallokation und Klassifizierung der Kanten.

einfachen Querstrich, Pseudo-Natural Edges sind mit einem doppelten Querstrich kenntlich gemacht.

Das Beispiel aus Abbildung 4.11 stammt aus [9] und wird im folgenden häufiger verwendet, um verschiedene Sachverhalte zu verdeutlichen.

4.3.3 Algorithmus zum Aufspalten von DAG's

Beim Aufspalten des DAG's werden Abhängigkeiten zwischen einzelnen Teilbäumen eingeführt , sog. *Read after Write* (RAW) Abhängigkeiten. *Read after Write* bedeutet, daß das Ergebnis eines Knotens oder Teilbaums erst gelesen werden darf, wenn es auch tatsächlich berechnet worden ist. Die Abhängigkeiten entstehen dadurch, daß man auch nach dem Aufspalten des DAG bestehende Datenabhängigkeiten des Orginal-DAG auch in den Teilbäumen einhalten muß, um eine korrekte Ausführung des Programms zu gewährleisten. Ein Problem entsteht dann, wenn durch ungeschicktes Aufbrechen von Kanten, zyklische Abhängigkeiten zwischen zwei Teilbäumen entstehen, und zwar so, daß beide Teilbäume auf das Ergebnis des jeweils anderen Teilbaums warten. Da aber die Bäume nur nacheinander bearbeitet werden können, führt dies zu einem nicht auflösbaren Deadlock. Am Beispiel in Abbildung 4.12 wird das Problem erläutert.

Der dargestellte DAG (Abbildung 4.12 links) besitzt zwei Knoten mit zwei abgehenden Kanten. Jeweils eine von diesen abgehenden Kanten muß aufgebrochen werden, wenn man Bäume erhalten will. Besonderes Augenmerk legen wir hierbei auf die beiden wieder zusammenlaufenden Pfade (*reconvergent path*) von u nach v. Betrachten wir die linke Konstellation. Hier wurden die Kanten (u, T_2) und (T_2, v) aufgebrochen und das Ergebnis sind die Bäume T_3 und T_4. Die gepunkteten Pfeile stellen die aufgebrochenen Kanten dar. Es gibt jetzt eine Kante von T_3 nach T_4 und eine Kante von T_4 nach T_3. Durch diese zyklische Abhängigkeit gibt es keine Möglichkeit für ein gültiges Schedule der beiden Teilbäume.

Eine weitere Möglichkeit besteht darin, daß statt der Kante (T_2, v) die Kante (T_2, w) aufgebrochen wird. In diesem Fall wird jedoch eine RAW-Abhängigkeit, (u, T_2), in den Teilbaum T_3 verlegt. Der Baumparser, der mit Olive generiert wird, kann jedoch mit diesen Abhängigkeiten nicht umgehen. Eine Möglichkeit besteht darin, den Baumparser dahin gehend zu verändern, daß diese Abhängigkeiteten erfüllt werden können, dies ist eine sehr schwierige Aufgabe, für die nach [9] keine effiziente Lösung existiert .

Beim Aufspalten des DAG muß also auf diese zyklische Abhängigkeit geachtet werden. Aus den beiden beschriebenen Fällen kann man erkennen, daß man alle Pfade, die wieder zusammenlaufen, aufbrechen muß, um auf keinen Fall zyklische Abhängigkeiten zu erhalten.

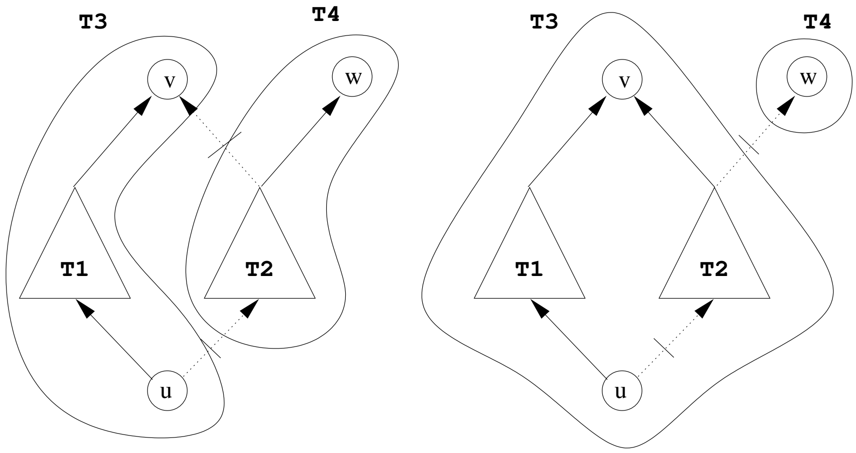

Abbildung 4.12: Zyklische Abhängigkeiten

```
begin
foreach node in DAG
    mark_node(node);
  foreach edge in DAG
    mark_edge(edge);
   break_natural_edges;
   break_reconvergent_path;
  foreach node in DAG
    if fan_out(node) > 1
      break_all_edges_but_one(node);
end
```

Abbildung 4.13: Algorithmus Dismantle

4.3.3.1 Algorithmus Dismantle

Aus den Erkenntnissen der vorigen Abschnitte läßt sich ein Algorithmus zum Aufspalten von DAG's herleiten.

Der vorgestellte Algorithmus Dismantle (Abbildung 4.13) arbeitet folgendermaßen. Zuerst werden mit `mark_node` alle Knoten und mit `mark_edge` alle Kanten des DAG markiert (Kapitel 4.3.2.1, 4.3.2.2 und 4.3.2.3). Alle Natural Edges werden mit `break_natural_edges` aufgebrochen, da dies keine neuen Kosten verursacht. Danach werden im DAG alle Pfade gesucht, die wieder zusammenlaufen. Wenn in diesen Pfaden Pseudo-Natural Edges enthalten sind, so werden die Pfade an diesen Stellen durch `break_reconvergent_path` aufgebrochen, gibt es keine Pseudo-Natural Edges mehr, so muß eine "normale" Kante aufgebrochen werden. Dies ist immer die erste Kante des rekonvergenten Pfades, da durch diese Kante der Fan-Out

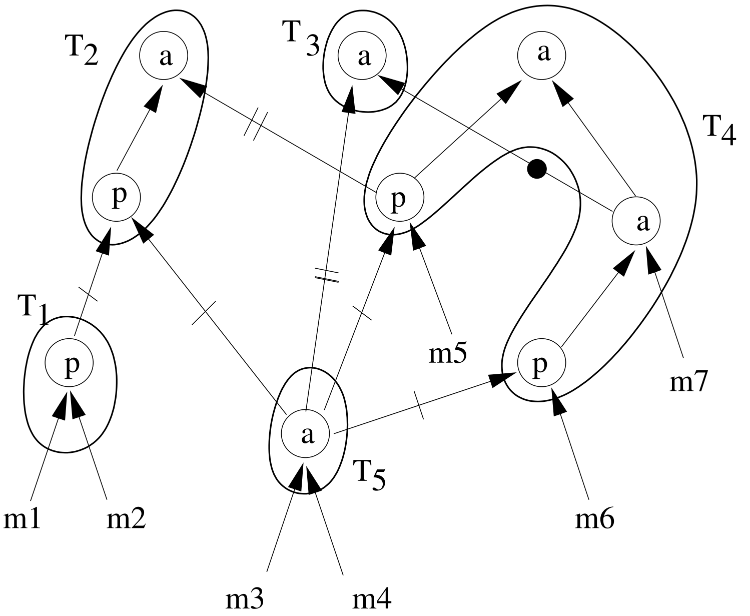

Abbildung 4.14: Ausdrucks DAG nach erfolgter Aufspaltung

des Startknotens reduziert wird. Sollten an diesem Punkt noch Knoten mit mehr als einer abgehenden Kante existieren, so müssen alle Kanten dieses Knotens, bis auf eine, ebenfalls aufgebrochen werden. Dies geschieht mit `break_all_edges_but_one`. Die entstandenen Bäume können nun in der Reihenfolge bestehender Abhängigkeiten mit den zuvor beschriebenen Techniken zur Codegenerierung bearbeitet werden. In Abbildung 4.14 erkennt man das Ergebnis des Algorithmus Dismantle für den Ausdrucksbaum aus Abbildung 4.11. Die aufgebrochenen Natural- und Pseudo-Natural Edges sind markiert. Außerdem ist eine Kante mit einem Kreis markiert. Diese wurde aufgebrochen, da der Startknoten immer noch einen Fan-Out = 2 hatte. Man hätte an dieser Stelle auch die Geschwisterkante nehmen können, doch es handelt sich bei der tatsächlich aufgebrochenen um eine Pseudo-Natural Edge (s. Abbildung 4.11), und diese erhalten in jeder Situation den Vorzug.

4.3.4 Bewertung

Der Algorithmus von Araujo und Malik führt zunächt eine Klassifizierung der Kanten des DAG durch, die sich nach der Prozessorarchitektur richtet. Auf Grund dieser Klassifizierung wird der DAG so aufgeteilt, daß möglichst keine Extrakosten durch die Aufteilung entstehen. Wenn dies für den ganzen DAG möglich ist, erhält man eine gute, wenn nicht sogar optimale, Aufteilung.

Durch den Versuch, möglichst wenig Kanten aufzubrechen, die hohe Kosten verursachen, muß der Algorithmus mit einem weiteren Problem kämpfen; den rekonvergenten Pfaden. Diese Pfade müssen erkannt und aufgebrochen werden, da sonst unauflösbare Abhängigkeiten entstehen.

Für das Aufbrechen der Kanten werden immer Strukturinformationen des Zielprozessors verwendet, was in der Regel zu einer Aufteilung führt, die minimale Extrakosten produziert. Erst wenn diese Informationen nicht mehr weiterhelfen, müssen Kanten zufällig aufgebrochen werden, was die Lösung verschlechtern kann.

4.4 Simulated Annealing für DAG's

In [10] wird von Leupers ein völlig anderer Ansatz für die Aufspaltung von DAG's verfolgt. Im Gegensatz zur Methode von Araujo und Malik werden die Knoten und Kanten des DAG nicht klassifiziert und anhand dieser Klassifizierung möglichst geschickt aufgeteilt. Bei Leupers wird der DAG an den Knoten aufgeteilt, die einen mehrfach verwendeten Teilausdruck (*common subexpressions* - CSE) enthalten. Dies sind alle Knoten mit Fan-Out größer als 1. Für diese CSE's gibt es im Prinzip zwei Möglichkeiten :

1. mehrfach berechnen oder
2. zwischenspeichern.

Ohne näher darauf einzugehen ist klar, daß eine Mehrfachberechnung der CSE nicht in Frage kommt, da dies in der Regel wesentlich aufwendiger und kostspieliger ist, als das Resultat zwischenzuspeichern. Anders als in klassischen Ansätzen der Codegenerierung [11], und wie auch bei Araujo und Malik [9] angewendet, wird in der Methode von Leupers nicht zwingenderweise der Hauptspeicher als Ort für Zwischenergebnisse verwendet. Vielmehr wird mit einem probabilistischen Algorithmus versucht, auch die Register des Zielprozessors als Zwischenspeicher zu verwenden.

4.4.1 Problembeschreibung

Im Datenpfad des C25 gibt es vier Möglichkeiten, einen Wert zu speichern. Zum einen ist dies der Hauptspeicher und zum anderen sind dies die drei Register des Prozessors: Akkumulator, Multiplikationsregister und Produktregister (s. auch Abbildung 2.1). Die Daten in den Registern bleiben immer solange gültig, bis durch eine andere Operation das Register überschrieben wird. Dies macht sich Leupers bei seiner Methode zu nutze.

Gegeben sei ein DAG D mit k CSE's $C = \{c_1, ..., c_k\}$ und eine Abbildung $K : C \rightarrow \{\text{MEM, ACCU, PREG, TREG0}\}$, die allen k CSE Register zuweist, in denen sie zwischengespeichert werden sollen. $K(c_i) = L$, $L \in$ {MEM, ACCU, PREG, TREG0} bedeutet, daß das Ergebnis der CSE c_i in dem Register L gespeichert ist, und daß jedesmal, wenn c_i verwendet wird, der Wert aus dem Register L gelesen wird. An dieser Stelle wird auch zur Vereinheitlichung für den Speicher der Begriff Register verwendet. Dieses "Register" ist jedoch nicht, wie die übrigen, auf einen Wert begrenzt, sondern kann viele Werte enthalten und unterscheiden.

Man muß beachten, daß nicht alle Registerbelegungen auch tatsächlich gültig sind. Es ist durchaus möglich, daß das Register, in dem die CSE c_i gespeichert werden soll, überschrieben wird, bevor alle Teilbäume, die c_i verwenden, bearbeitet wurden. Wenn ein solcher Fall auftritt, kann das gewählte Register nicht für die CSE c_i verwendet werden. Man muß ein anderes Register oder den Speicher wählen. Es gibt jedoch keinen Grund, den Speicher generell zu bevorzugen, es gibt eher Gründe dafür, Register zu verwenden :

1. Das Lesen einer CSE von einem Register kann zu geringerer Codegröße und höherer Performance führen, da Speicherzugriffe vermieden werden.

2. Wenn externer Speicher, anstelle von On-Chip Speicher, verwendet wird, sind Speicherzugriffe sehr viel langsamer als Registerzugriffe.

3. Speicherzugriffe verbrauchen mehr Energie, da mehr Kommunikation auf den Daten- und Adressbus nötig ist.

4. Es kann sein, daß die Speicheradressen für die CSE's mit weiteren Instruktionen berechnet werden müssen.

Das Ziel ist es, unter allen möglichen und gültigen Registerbelegungen, diejenige Belegung herauszufinden, die die geringsten Kosten bei der Instruktionsauswahl verursacht.

Betrachten wir das Beispiel aus Abbildung 4.15. Um für diesen DAG Code generieren zu können, wird der DAG an den Kanten mit den Punkten aufgebrochen. Es entstehen drei Bäume, für die der Code getrennt erzeugt werden kann. Die aufgebrochenen Kanten stellen allerdings Abhängigkeiten zwischen den Bäumen dar, auf die bei der Reihenfolge der Codegenerierung geachtet werden muß.

In Tabelle 4.5 sieht man zwei unterschiedliche Codesequenzen für den DAG aus Abbildung 4.15. Bei der linken Sequenz wurde zum Speichern der CSE der Hauptspeicher gewählt. Das Ergebnis der CSE wird also in *temp* gespeichert und muß für die beiden folgenden Bäume wieder neu geladen werden. Bei der rechten Sequenz wurde als Zwischenspeicher das Produktregister gewählt, in dem das Resultat nach der Multiplikation $m_1 * m_2$ ja sowieso liegt. Bei beiden folgenden Additionen kann

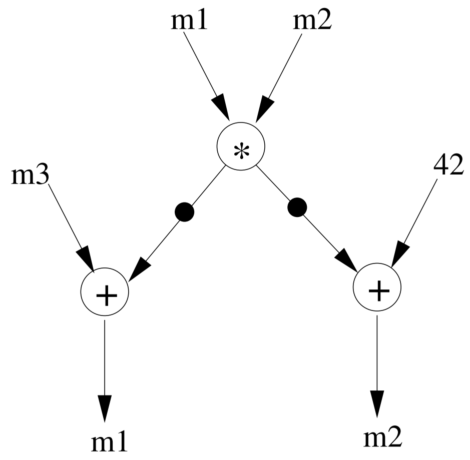

Abbildung 4.15: Beispiel DAG

Speicher		Produktregister	
LT	m1	LT	m1
MPY	m2	MPY	m2
SPL	temp	PAC	
LAC	temp	ADD	m3
ADD	m3	SACL	m1
SACL	m1	PAC	
LAC	temp	ADDK	42
ADDK	42	SACL	m2
SACL	m2		

Tabelle 4.5: Alternative Codesequenzen für den DAG aus Abbildung 4.15

man die CSE direkt aus dem Produktregister lesen, da der Wert nicht überschrieben wird und so die ganze Zeit gültig ist. Man erkennt, daß die rechte Codesquenz nicht nur eine Instruktion weniger besitzt, sondern auch daß anstelle von drei Speicherzugriffen (SPL und LAC) nur zwei Registertransferoperationen (PAC) benötigt werden.

Weiterhin ist es sehr wichtig, die gegenseitige Abhängigkeit zwischen der Registerzuweisung für CSE's und der Reihenfolge der Teilbäume zu beachten. Dazu betrachte man das Beispiel in Abbildung 4.16. Durch die Aufteilung des DAG erhält man die drei Bäume T_1, T_2und T_3. Die gegebenen Abhängigkeiten besagen, daß T_1 vor T_2 und T_3 bearbeitet werden muß. Die Reihenfolge für T_2und T_3 ist jedoch nicht festgelegt. Nehmen wir nun an, daß für die CSE das Produktregister gewählt wurde. Wenn zuerst T_2 bearbeitet wird, wird der Wert von T_1 im Produktregister überschrieben, da auch in T_2 eine Multiplikation erfolgt. Für T_3 ist der Wert der CSE nicht mehr gültig. Wird jedoch zuerst T_3 bearbeitet, so bleibt für T_2 der Wert der CSE gültig,

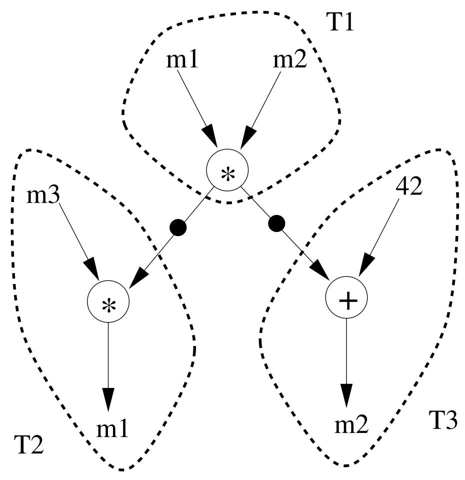

Abbildung 4.16: Beispiel DAG 2

da die Addition in T_3 das Produktregister nicht überschreibt.

Man muß deshalb für eine bestimmte Registerbelegung verschiedene Schedules der einzelnen Bäume untersuchen. Die im folgenden Abschnitt beschriebene Methode versucht diesem gerecht zu werden.

4.4.2 Algorithmus mit SA

Aufgrund der großen Zahl von möglichen Registerbelegungen und einer Vielzahl von möglichen Schedules der Teilbäume wird bei Leupers ein Algorithmus auf Basis von *Simulated Annealing*[2] (*SA*) verwendet. SA ist eine probabilistische Optimierungsstrategie, die häufig bei komplexen Optimierungsproblemen eingesetzt wird. Mit SA kann man lokale Optima überspringen, um ein globales Optimum zu erhalten. Dies wird dadurch erreicht, daß SA zuläßt, daß eine Verschlechterung der aktuell gefundenen Lösung unter gewissen Umständen in Kauf genommen wird, um durch Modifikation der "schlechteren" Lösung eine global bessere Lösung zu finden.

Der Algorithmus CSE-Registerzuweisung arbeitet folgendermaßen.

Zuerst wird der DAG G in Bäume aufgeteilt, `DECOMPOSE`. Dazu werden alle Kanten von Knoten mit Fan-Out > 2 gespalten. Anstelle dieser Kanten werden spezielle CSE-Schreib- und CSE-Lese-Knoten eingesetzt. Der entstandene Graph ist G'. Während der Optimierung steht die aktuelle CSE-Belegung in CSE_alloc. Alle CSE-Register werden vor dem Start der Optimierung mit einem Platz im Hauptspeicher initialisiert und die Kosten für diese Start-Belegung ermittelt, `INITIAL_COST`. Für die Ermittlung der Kosten wird hier bereits der Baumparser OLIVE eingesetzt.

[2] Simulated annealing - simuliertes Abkühlen

```
INPUT DAG G mit k CSE's;
OUTPUT Codesequenz für G;
VAR CSE_alloc : array[1..k] of enum {MEM, ACCU, PREG, TREG0};
     G': DAG;
     T : Ausdrucksbaum;
    float temp;
    integer count, cost, best, delta;
     schedule, best_schedule: Liste von Ausdrucksbäumen;
begin
  G'= DECOMPOSE(G);
  CSE_alloc[1..k] = MEM;
  best= INITIAL_COST(G');
  temp = 50;
  while temp > 0.1 do
    for count = 1 to 10 do
      DO_MODIFICATION(G', CSE_alloc);
      schedule = TOPOLOGICAL_SORT(G');
      cost = 0;
      for all trees T in schedule do
        cost += COVER_COST(T) + ADDR_COST(T);
        if REGISTER_CONFLICT(T) then cost = ∞;
      end for
      delta = cost - best;
      if delta < 0 or RANDOM(0,1) < exp(-delta/temp) then
        best = cost;
        best_schedule = schedule;
      else UNDO_MODIFICATION(G', CSE_alloc);
      end if
    end for
    temp = 0.9 * temp;
  end while
  for all trees T in best_schedule do
    EMIT_ASSEMBLY_CODE(T);
  end for
end algorithm
```

Abbildung 4.17: Algorithmus CSE-Registerzuweisung

Die Optimierung wird mit einer bestimmten Temperatur[3] (*hier 50*) gestartet. Diese Temperatur bestimmt, zusammen mit dem Abkühlungsfaktor[4] (*hier 0,9*), die Häufigkeit des Optimierungsdurchlaufs. Der Algorithmus endet, wenn eine bestimmte Temperatur erreicht wird (*hier 0,1*).

Nun wird der Baum G' verändert, `DO_MODIFICATION`. Mit einer Wahrscheinlichkeit von 50% wird nun entweder die aktuelle CSE-Belegung per Zufall geändert oder es werden Kanten in G' hinzugefügt, die die Reihenfolge der Teilbäume beinflussen. Kanten, die an dieser Stelle eingefügt werden, dürfen auf keinen Fall den Abhängigkeiten auf G wiedersprechen. Dies ist genau dann der Fall, wenn durch eine eingefügte Kante ein Zyklus in G' entstehen würde.

Für die neue CSE-Belegung wird eine gültige Reihenfolge der Teilbäume ermittelt, `TOPOLOGICAL_SORT`, und für diese Reihenfolge werden die Kosten `cost` ermittelt. Auch hier werden mit dem Baumparser die tatsächlichen Kosten errechnet. Wenn es, wie bereits oben beschrieben, zu einem Registerkonflikt kommt, werden die Kosten der Lösung auf Unendlich gesetzt, so daß diese ungültige Lösung auf jeden Fall ignoriert wird.

Wenn die neue Lösung eine Verbesserung darstellt, so wird sie übernommen, aber auch wenn die Lösung schlechter ist, kann sie übernommen werden. Dies geschieht, wenn der Ausdruck `RANDOM(0,1) < exp(-delta/temp)` wahr wird. Dieser Ausdruck besagt, daß je höher die Temperatur und je geringer die Verschlechterung der aktuell besten Lösung ist, umso höher ist die Wahrscheinlichkeit, daß die schlechtere Lösung verwendet wird. Wird die neue Lösung nicht verwendet, so müssen die vorgenommenen Änderungen wieder rückgängig gemacht werden, `UNDO_MODIFICATION`.

Nachdem die "beste" Lösung gefunden wurde, wird noch der Code für G ausgegeben, `EMIT_ASSEMBLY_CODE`.

Es ist zu beachten, daß dieser Algorithmus nur ein probabilistischer Algorithmus ist und nicht immer die optimale Lösung liefert.

4.4.3 Beispiel für den SA-Algorithmus

In Abbildung 4.18 erkennt man die Aufteilung für den DAG aus Abbildung 4.11, wie sie von dem SA-Algorithmus (Abbildung 4.17) vorgenommen wird. Alle Kanten, die von einem CSE-Knoten ausgehen, werden aufgebrochen. CSE-Knoten in Abbildung 4.18 sind die Wurzeln der Teilbäume T_2, T_4 und T_5. Die Resultate der CSE-Knoten werden mit dem Algorithmus nun auf die möglichen Speicherort im Datenpfad des Prozessors oder im Hauptspeicher verteilt, so daß für die nachfolgende Überdeckung mit Instruktionen die geringsten Kosten entstehen.

[3] Temperatur : ein Parameter der SA-Optimierung
[4] Abkühlungsfaktor : ein weiterer SA Parameter

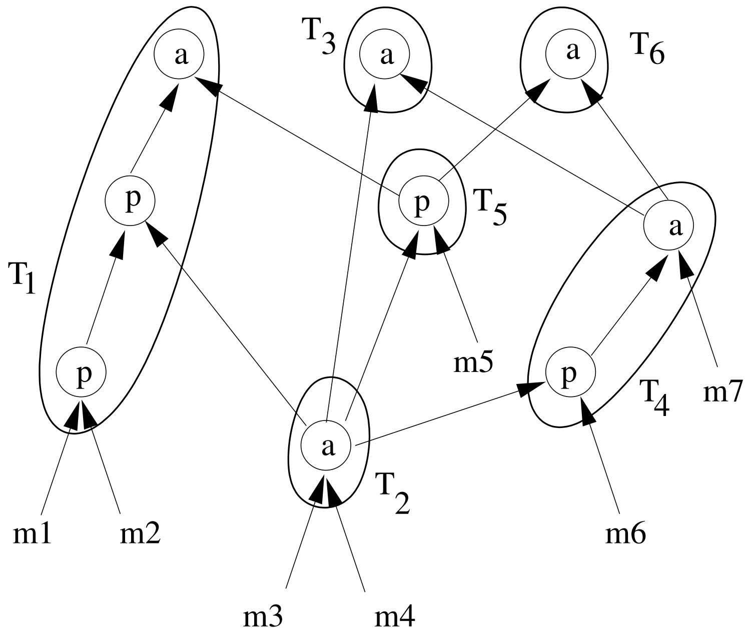

Abbildung 4.18: Aufteilung des DAG aus Abbildung 4.8 mit dem SA-Algorithmus

Für das Beispiel in Abbildung 4.18 ist nur die folgende Lösung möglich :

CSE von T_2 : Für das Resultat in der Wurzel von T_2 bleibt nur der Hauptspeicher als Möglichkeit der Speicherung übrig. Alle anderen in Frage kommenden Register werden bei jeder Reihenfolge der übrigen Teilbäume überschrieben.

CSE von T_5 : Auch das Ergebnis in der Wurzel von T_5 kann nur im Speicher gerettet werden, da in T_1 das Produktregister überschrieben wird.

CSE von T_4 : Für die CSE in T_4 gibt es allerdings die Möglichkeit, das Zwischenergebnis in einem Register zu speichern, wenn die Bäume T_1 und T_5 vor T_4 bearbeitet werden. Wenn dies der Fall ist, kann die CSE im Produktregister gespeichert werden. Der Weg dorthin führt zwar auch über den Hauptspeicher, aber das Ergebnis muß später nicht von beiden nachfolgenden Bäumen aus dem Speicher gelesen werden. Diese Zugriffe werden durch Registertransferoperationen ersetzt.

4.4.4 Bewertung

Die Aufteilung des DAG geschieht bei Leupers unabhängig von der Struktur des Zielprozessors. Dies ist von Vorteil, wenn man dieselbe Methode für verschiedene

Zielarchitekturen verwenden möchte. Durch die Aufteilung des DAG an allen CSE Knoten werden in der Regel mehr Teilbäume entstehen, als in der Aufteilung von Araujo und Malik. Bei Araujo und Malik werden nicht alle Kanten an CSE Knoten aufgebrochen, eine Kante bleibt in der Regel bestehen, wodurch größere Bäume für den Baumparser entstehen. Dies ist für den Baumparser von in Vorteil, da dieser optimal arbeitet.

Für den SA-Algorithmus bedeuten aber mehr Bäume auch mehr Möglichkeiten, die Reihenfolge der Bäume zu beeinflussen, was die Wahrscheinlichkeit eines Registerkonflikts reduziert. Dadurch daß mehr kleine Bäume bei der Aufteilung entstehen, werden in jedem Baum auch weniger Register benutzt.

Es gibt jedoch ein Problem, das die Qualtität der Lösung negativ beeinflussen kann. Der Kern des SA-Algorithmus ist die Funktion, die berechnet, ob eine gewählte Registerbelegung gültig ist. Unter bestimmten Umständen ist es möglich, daß man eine gültige Belegung nicht als gültig erkennt. Der Grund ist in der Verwendung des Baumparsers zu suchen. Ähnlich wie in Kapitel 4.2 werden für die Erkennung eines Konflikts Informationen über die in den einzelnen Bäumen benutzten Register benötigt. Wenn ein Baum ein Register verwendet, in dem eine noch nicht vollständig verwendete CSE gespeichert ist, ist die gerade gewählte Registerbelegung ungültig.

Was passiert aber, wenn in einem Baum eine CSE zum letzten Mal verwendet wird, in einem anderen Teilbaum desselben Baumes aber das Register, in dem die CSE gespeichert ist, überschrieben wird?

Diese Situation ist in Abbildung 4.19 dargestellt. Irgendwo in T_2 wird die CSE in Register R zum letzten mal benutzt. Im zweiten Teilbaum T_1 wird das Register R ebenfalls verwendet. Wenn beim Scheduling der Teilbaum T_2 zuerst geschedult wird, wäre die Belegung tatsächlich gültig, da die CSE in R nicht mehr benötigt wird. Wird erst T_1 geschedult, ist die Belegung ungültig, da die CSE in R überschrieben wird. In der Phase der Codegenerierung, in der der Registerkonflikt geprüft wird, liegen aber nur Informationen darüber vor, welche Register verwendet werden und nicht, wann diese Register vewendet werden. Daher bleibt nur die Möglichkeit, eine eigentlich gültige Lösung als ungültig zu erklären.

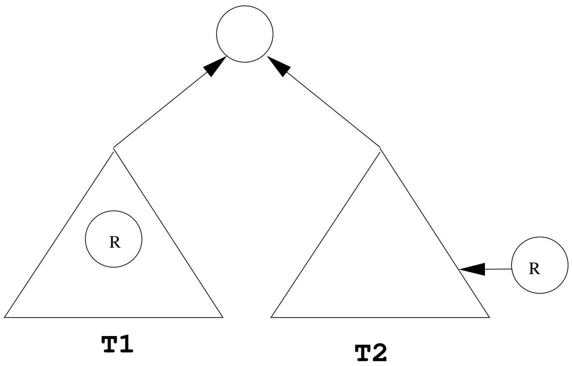

Abbildung 4.19: Probleme beim Erkennen von Registerkonflikten

Kapitel 5

Die Integration des AccuBuffers

Die wesentliche Neuerung in der Prozessorstruktur des TMS320C5x gegenüber dem Vorgängermodel TMS320C25, auf den sich alle in Kapitel 4 vorgestellten Methoden beziehen, ist die Integration eines weiteren Prozessorregisters, des Akkumulatorpuffers (*accumulator buffer* - ACCUB). Der ACCUB ist nur mit dem Akkumulator und der ALU verbunden , d.h. Daten können vom Akkumulator in den ACCUB kopiert und wieder ausgelesen werden. Ebenso kann der ACCUB als Eingang für die ALU genutzt werden, so daß Operanden für arithmetische Operationen auch aus dem ACCUB kommen können. In Abbildung 5.1 ist dieser Unterschied noch einmal grafisch dargestellt.

Für die in Kapitel 4 beschriebenen Methoden ergeben sich durch die erweiterte Struktur des C5x verschiedene Änderungen, die in den folgenden Abschnitten beschrieben werden.

Mit Einführung des ACCUB wurde der Befehlssatz des C5x um die entsprechenden Befehle erweitert. Zu diesen Befehlen gehören zuerst einmal reine Registertransferoperationen, mit denen der Inhalt des Akkumulators (ACCU) im ACCUB gespeichert werden kann, *store ACCU in ACCUB (SACB)* , und ebenso auch aus dem ACCUB wieder in den ACCU geladen werden kann, *load ACCU from ACCUB (LACB).* Zusätzlich ist noch ein Vertauschen beider Register möglich.

Die Gruppe der arithmetischen Befehle des C5x wurde ebenfalls ergänzt. Bei den arithmetischen Befehlen wurde der Befehlssatz um *alle* binären Operationen der ALU erweitert, d.h. daß für alle binären arithmetischen Operationen der zweite Operand nun auch aus dem ACCUB gelesen werden kann. Eine Liste der neuen Befehle findet man in Tabelle 5.1.

Außer der Einführung des ACCUB gibt es noch eine Reihe von kleinen Änderungen, die jedoch für die Aufgabe der reinen Codeerzeugung auf Basisblockebene keine Rolle spielen. Z.B. die Erweiterung der Funktionalität der Hilfsregistereinheit (Kapitel

Mnemonic	Beschreibung
LACB	Lade ACCU vom ACCUB
SACB	Speichere ACCU in ACCUB
EXAR	Vertausche ACCU und ACCUB
ADDB	Addiere ACCU und ACCUB
SUBB	Subtrahiere ACCUB von ACCU
ANDB	UND-Verknüpfung von ACCU und ACCUB
ORB	ODER-Verknüpfung von ACCU und ACCUB
XORB	Exclusiv ODER-Verknüpfung von ACCU und ACCUB

Tabelle 5.1: Befehle des C5x für den ACCUB

2.3). Das Hilfsregisterfile wird für die Erzeugung von Code für Basisblöcke nicht unmittelbar verwendet, es ist besonders wichtig, wenn man sich mit der Zuweisung von Variablen an die zur Verfügung stehenden Hilfsregister beschäftigt oder wenn man den erzeugten Code in einer weiteren Phase der Codeerzeugung weiter kompaktieren will. Die Codekompaktierung ist allerdings nicht Thema dieser Arbeit.

Es sei noch angemerkt, daß der Assemblercode des C25 weitgehend aufwärts kompatibel zum C5x ist, d.h. daß Programme, die für den C25 geschrieben wurden, mit sehr geringen Änderungen, auch auf dem C5x verwendet werden können. Diese Änderungen beziehen sich häufig nur auf geänderte Mnemonics der beiden Assemblersprachen oder auf die Zusammenführung verschiedener Befehle zu einem einzigen Befehl. Z.B. gibt es beim C25 gesonderte Befehle für den Umgang mit Konstanten. Ladeoperationen und arithmetische Operationen gibt es einmal für Operanden aus dem Speicher (*ADD memory*) und einmal für Konstanten (*ADDK #konstante*). Diese Befehle fallen beim C5x zum Befehl *ADD* zusammen. Eine komplette Liste aller Befehle zur Umsetzung von C25 in C5x Code gibt es in [5], Kapitel 4.

5.1 Einbau des ACCUB in die OLIVE-Grammatik

Durch die neuen Befehle des C5x, die den ACCUB nutzen, ergeben sich natürlich Anpassungen der OLIVE-Beschreibung. Der ACCUB wird in der Grammatik, wie alle übrigen Prozessorregister auch, durch ein Nicht-Terminal-Symbol dargestellt (s. Kapitel 3.2).

Entscheidend für die Auswahl der Baummuster, die von OLIVE für die Überdeckung von Bäumen ausgewählt werden, ist jetzt die Kostenfunktion im Kostenteil der Regeln. Sämtliche Regeln aus Tabelle 5.2 gibt es ebenfalls für Baummuster, die den zweiten Operanden aus dem Hauptspeicher erhalten, und nicht aus dem ACCUB.

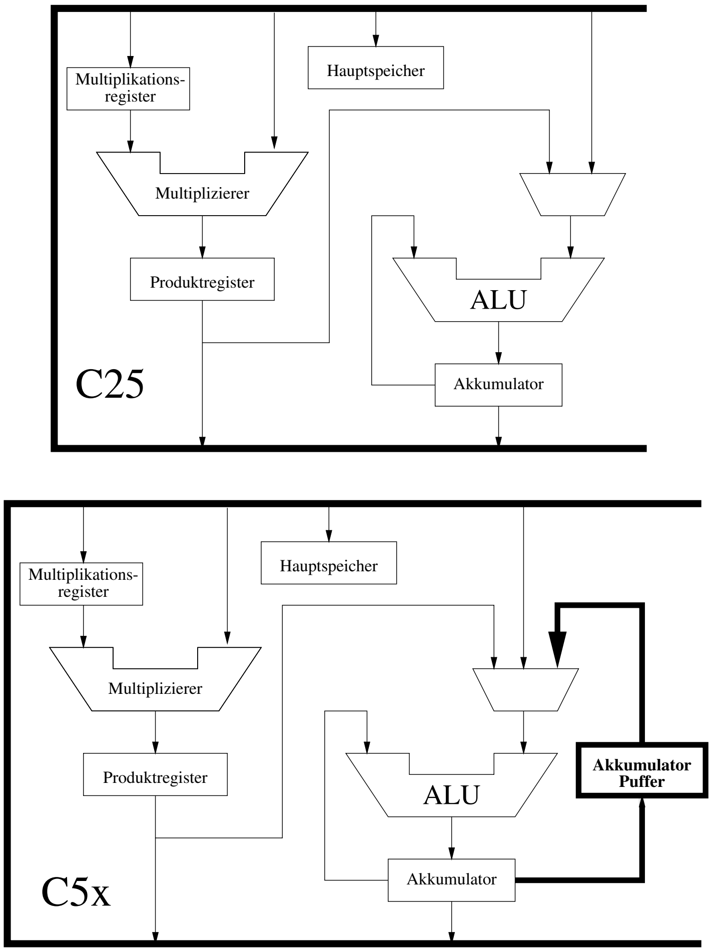

Abbildung 5.1: Die Unterschiede in der Registerstruktur des C25 und C5x

Ziel		Baummuster		Befehl
accu	:	accub	{ } = { } ;	LACB
accub	:	accu	{ } = { } ;	SACB
accu	:	PLUS (accu , accub)	{ } = { } ;	ADDB
accu	:	MINUS (accu , accub)	{ } = { } ;	SUBB
accu	:	UND (accu , accub)	{ } = { } ;	ANDB
accu	:	ODER (accu , accub)	{ } = { } ;	ORB
accu	:	XOR (accu , accub)	{ } = { } ;	XORB

Tabelle 5.2: OLIVE-Beschreibung aller Befehle aus Tabelle 5.1

Ziel		Baummuster		Befehl
accu	:	memory	{ } = { } ;	LACC
memory	:	accu	{ } = { } ;	SACL
accu	:	PLUS (accu , memory)	{ } = { } ;	ADD
accu	:	MINUS (accu , memory)	{ } = { } ;	SUB
accu	:	UND (accu , memory)	{ } = { } ;	AND
accu	:	ODER (accu , memory)	{ } = { } ;	OR
accu	:	XOR (accu , memory)	{ } = { } ;	XOR

Tabelle 5.3: Arithmetische Befehle aus Tabelle 5.2 mit einem Operanden aus dem Speicher

Die entsprechenden Baummuster findet man in Tabelle 5.3. Die Entscheidung von OLIVE, welches Muster gewählt wird, wird ausschließlich anhand der Kosten für die Überdeckung eines Baumes getroffen.

Im folgenden nehmen wird für alle verwendeten Muster die Kosten 1 an, genau wie in Tabelle 4.1.

Für OLIVE gibt es keine Unterschiede zwischen den einzelnen Nicht - Terminal - Symbolen, d.h. es werden alle Möglichkeiten der Überdeckung ausprobiert und die günstigste, auf Grund der Kostenfunktion, wird verwendet. Es wird bei gleichen Überdeckungskosten kein Nicht-Terminal-Symbol bevorzugt.

Nehmen wir einmal den folgenden Fall an :

Das Ergebnis im Akkumulator muß zwischengespeichert werden, weil es sonst überschrieben würde. Später wird der Inhalt des Akkumulators noch als Operand einer Addition benötigt. Es gibt zwei Möglichkeiten, diesen Fall mit den oben beschriebenen Baummustern zu überdecken.

Variante 1	Variante 2
memory : accu	*accub* : accu
accu : (beliebig)	accu : (beliebig)
accu : PLUS (accu, *memory*)	accu : PLUS (accu, *accub*)

Die mittlere Zeile soll in beiden Varianten für eine beliebige Folge von Operationen stehen, die an einer Stelle den Inhalt des Akkumulators überschreiben, so daß der Inhalt des Akkumulators gerettet werden muß, um ihn später noch verwenden zu können. OLIVE soll nun aus beiden Varianten die günstigere auswählen. OLIVE entscheidet ausschießlich anhand der Kostenfunktion.

Welche Variante ist aber die "günstigere"? Mit den angenommenen Kosten 1 für alle Muster sind beide Varianten gleich teuer. Betrachtet man aber die obere und untere Zeile beider Varianten, stellt man fest, daß in Variante 1 zwei Speicheroperationen verwendet werden - erst wird der Inhalt des Akkumulators in den Hauptspeicher geschrieben, und später, für die Addition, wieder daraus gelesen. Die rechte Variante benötigt dagegen zwei Registertransferoperationen - kopieren des Akkumulators in den ACCUB und lesen des Operanden für die Addition aus dem ACCUB. Diese Registertransferoperationen werden in der Regel schneller ausgeführt, abhängig von der Art des verwendeten Hauptspeichers (Kapitel 2). Außerdem ist es möglich, daß die Adresse im Speicher, an der das Ergebnis gespeichert wurde, wieder neu berechnet werden muß, was auch wieder weitere Kosten verursachen kann. Das bedeutet, daß die Variante 1 auf jeden Fall teurer werden muß, als die Variante 2. Speicheroperationen sind also mit höheren Kosten zu veranschlagen als Registeroperationen. In Kapitel 7 sieht man, wie die Kostenfunktion in der Prozessorbeschreibung des C5x für den Codegenerator realisiert wird.

5.1.1 Beispiel für ACCUB in der Grammatik

Nehmen wir als Beispiel die Funktion :

$$e = (a - b) + (c - d)$$

Zunächst werden die Ausdrücke (a - b) und (c - d) ausgewertet, beide Ausdrücke sind Summanden der anschließenden Addition und liegen nach der Berechnung im Akkumulator. Die folgende Addition kann aber nicht beide Operanden gleichzeitig aus dem Akkumlator erhalten. Ein Zwischenergebnis muß also gespeichert werden. Die Transferoperationen, die benötigt werden, um ein Teilergebnis zu sichern und später wieder als Operand bereit zu stellen, werden während der Instruktionsauswahl des mit OLIVE generierten Baumparsers eingefügt. Der Baumparser gewährleistet also, daß alle Operanden dort stehen, wo die Befehle sie benötigen.

Um den obigen Ausdruck zu überdecken, kommen wir mit den Mustern aus den Tabellen 5.2 bzw. 5.3 aus. Mit diesen Mustern sind zwei Überdeckungen möglich (Tabelle 5.4).

In Überdeckung 1, mit dem ACCUB in der Grammatik, erkennt man zwei Speicheroperationen weniger als in der Überdeckung 2, die ohne ACCUB auskommt. Wenn Speicheroperationen höhere Kosten haben, als Registertransferoperationen, wird vom Baumparser Überdeckung 1 gewählt.

Die Tabelle 5.5 zeigt die resultierenden Codesequenzen, einmal ohne ACCUB in der Grammatik, einmal mit ACCUB in der Grammatik.

<table>
<tr><th colspan="2">Überdeckung 1</th></tr>
<tr><td colspan="2">accu : PLUS(accu , accub)</td></tr>
<tr><td>accu : MINUS(accu , memory)
accu : memory</td><td>accub : accu
accu : MINUS(accu , memory)
accu : memory</td></tr>
<tr><th colspan="2">Überdeckung 2</th></tr>
<tr><td colspan="2">accu : PLUS (accu , memory)</td></tr>
<tr><td>accu : MINUS(accu , memory)
accu : memory</td><td>memory : accu
accu : MINUS (accu , memory)
a : memory</td></tr>
</table>

Tabelle 5.4: Mögliche Überdeckungen durch den Baumparser. Das Nicht-Terminal-Zeichen memory ist nicht weiter aufgelöst, steht aber jedesmal für einen Speicherzugriff.

LACC	a	LACC	a
SUB	b	SUB	b
SACL	temp	SACB	
LACC	c	LACC	c
SUB	d	SUB	d
ADD	temp	ADDB	
SACL	e	SACL	e

Tabelle 5.5: Codesequenzen; links Grammatik ohne ACCUB , rechts Grammatik mit ACCUB

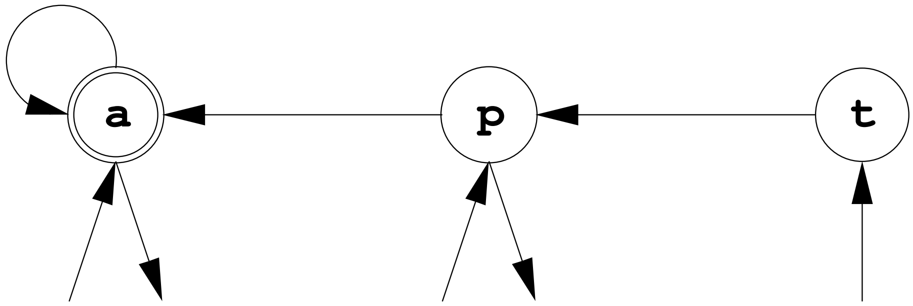

Abbildung 5.2: Registertransfergraph des C5x

5.1.2 Bewertung

Es bleibt festzuhalten, daß durch die Integration des ACCUB in die Grammatik Vorteile für den resultierenden Code entstehen, da durch den ACCUB Speicherzugriffe eingespart und durch Registertransferoperationen ersetzt werden können. Registertransferoperationen verbrauchen aber weniger kostbare Resourcen als Speicherzugriffe. Es ist durch eine geeignete Wahl der Kostenfunktion in der Grammatik möglich, die Verwendung von Baummustern, die den ACCUB verwenden, zu forcieren.

5.2 Änderungen beim Scheduling

In Kapitel 4.2 wurde gezeigt, daß das Scheduling eine sehr wichtige Rolle bei der Codegenerierung spielt, wenn es darauf ankommt, optimalen Code ohne Memory spills zu generieren. Das Verfahren, das in Kapitel 4.2 vorgestellt wurde, macht sich die Registerstruktur des Zielprozessors zu nutze. Wie man jedoch in Abbildung 5.1 erkennt, hat sich diese Struktur mit dem C5x geändert. Der Akkumulator kann nicht mehr als ein einzelnes Register betrachtet werden, sondern wird durch den ACCUB zu einem Registerfile, in dem genau zwei Werte gespeichert werden können. Den neuen RTG kann man in Abbildung 5.2 sehen. Die innere Struktur des entstandenen Registerfiles ist in Abbildung 5.3 dargestellt. Man sieht, daß innerhalb des Registerfiles die Daten nur zwischen ACCU und ACCUB transferiert werden können, der ACCUB hat keinerlei Verbindungen zu anderen Registern oder dem Speicher. D.h. Daten können nur vom ACCU in den ACCUB gelangen.

Was bedeuten diese Änderungen am RTG für das Scheduling ?

Das Konzept des RTG wurde für die Berechnung eines optimalen Schedules benutzt. Der RTG des C5x ist immer noch azyklisch, gemäß Definition 2 (S. 46). Allerdings gilt die Annahme nicht mehr, daß alle Register die Kapazität 1 haben, und daß deshalb zwischen zwei Registern, die eine Kante im RTG verbindet, genau *ein* Pfad

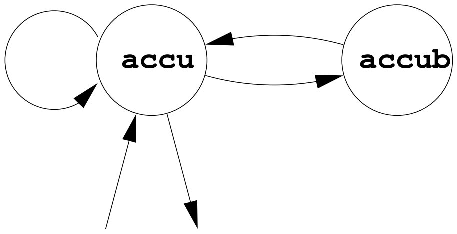

Abbildung 5.3: Registerfile mit ACCU und ACCUB

existiert. Der C5x entspricht damit nicht mehr dem [1,∞] Model, das eine weitere Voraussetzung für die Berechnung eines optimalen Schedules war. Nehmen wir als Beispiel die Kante von PREG (p) nach ACCU (a) im RTG des C5x. Die Kapazität des ACCU ist durch den ACCUB jetzt 2 und es gibt zwei Möglichkeiten einen Pfad von p nach a zu finden. Diese beiden Pfade sind :

$$preg \rightarrow accu \quad (1)$$
$$preg \rightarrow accu \rightarrow accub \rightarrow accu \quad (2)$$

Von besonderer Bedeutung ist hierbei, daß der erste Pfad auch gleichzeitig ein Teil des zweiten Pfades ist. Wenn es also eine Situation gibt, in der die Operanden einer Operation beide Pfade durchlaufen, um für die Operation verwendet werden zu können, ist die Reihenfolge, in der die Pfade benutzt werden müssen, schon vorgegeben.

Die Voraussetzung - nach Kapitel 4.2 - für ein Schedule ohne Memory spills, ist also nicht mehr gegeben. Das bedeutet aber, daß es einen Registerkonflikt geben kann, der sich nicht - wie in Kapitel 4.2 gezeigt - auflösen läßt.

Gibt es eine Situation, die einen Allocation deadlock verursachen kann ?

In Kapitel 4.2 wurde für den C25 gezeigt, daß *jede* Situation, die einen Allocation deadlock zwischen zwei beliebigen Registern besitzt, aufgelöst werden kann. Und zwar so, daß keine unnötigen Kosten durch Speicheroperationen entstehen. Der Grund dafür ist, daß zwischen zwei Registern, die einen Allocation deadlock auslösen, zwei Pfade im RTG existieren, von denen ein Pfad *immer* durch den Speicher läuft und sich die durch diesen Pfad erzeugten Speicherzugriffe keinesfalls vermeiden lassen. Beim C5x ist es nicht der Fall, daß zwischen *allen* Registerpaaren des Prozessors immer ein Pfad im RTG durch den Speicher läuft. Zwischen dem ACCU und dem ACCUB gibt es zwei Pfade, $accu \rightarrow accub$ und $accub \rightarrow accu$, von denen aber keiner durch den Speicher führt. Zwischen dem ACCU und dem ACCUB kann es zu einem Allocation deadlock kommen.

Kann man diesen Allocation deadlock erkennen und beheben ?

In Abbildung 5.4 ist die Konstellation gezeigt, die einen Allocation deadlock auslöst.

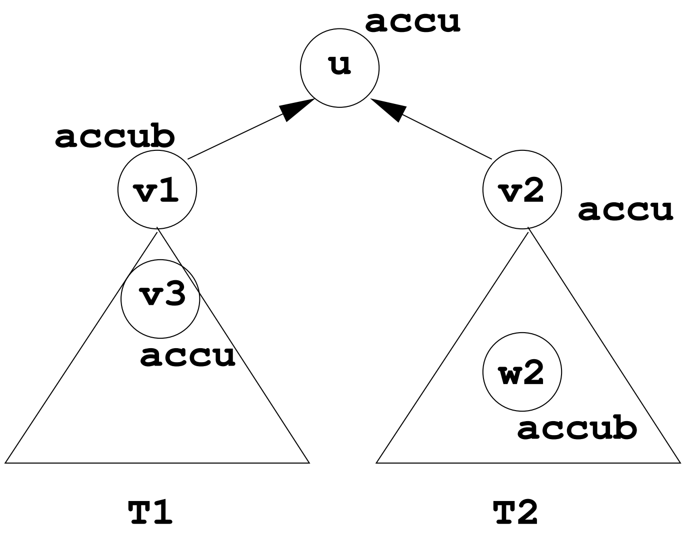

Abbildung 5.4: Allocation deadlock durch ACCU und ACCUB

Alle drei Bedingungen für einen Allocation deadlock aus Definition 1 (S. 44) sind erfüllt.

1. Die Resultate beider Teilbäume liegen in einem Register des Prozesors und nicht im Speicher, $L(v_1) = accub$ und $L(v_2) = accu$,
2. beide Teilergebnisse stehen nicht im selben Register, $accu \neq accub$, und
3. das Register, in dem das Resultat eines Teilbaums liegt, wird im jeweils anderen Teilbaum verwendet, $v_3 \in T_1$ und $L(v_3) = accu$, $w_2 \in T_2$ und $L(w_2) = accub$.

Für den Baum aus Abbildung 5.4 ist es also nötig, ein Teilergebnis im Speicher zwischenzuspeichern.

T sei ein Ausdrucksbaum, der aus den beiden Teilbäumen T_1 und T_2 besteht. Nachdem T mit Instruktionen des Prozessors überdeckt wurde, liegt das Resultat von v_1, der Wurzel von T_1, im ACCUB, das Resultat von v_2, der Wurzel von T_2, im ACCU. Um aber das Resultat des Teilbaums T_1 überhaupt in den ACCUB zu bekommen, muß direkt vor v_1 ein Knoten existieren, dessen Ergebnis im ACCU liegt. Anders kann der ACCUB nicht erreicht werden, jeder Pfad in den ACCUB führt unmittelbar über den ACCU. Im Beispiel ist dies der Knoten v_3. Dieser Knoten ist also der letzte Knoten in T_1, bevor v_1 erreicht wird. Für den Knoten v_2, der ebenfalls an dem Deadlock beteiligt ist, kann man die Position nicht genau definieren. Da auch andere Pfade, als der vom ACCUB, in den ACCU führen, müssen die Knoten w_2 und v_2 keine direkte Verbindung haben. Der Knoten w_2 liegt irgendwo in T_2. Der

ACCU, in dem das Ergebnis von T_2 liegt, wird also in T_1 auf jeden Fall verwendet, immer direkt vor v_1. Um einen Deadlock festzustellen, muß also nur geprüft werden, ob in dem zweiten Teilbaum der ACCUB ebenfalls verwendet wird.

Welches Teilergebnis soll man aber zwischenspeichern, um keine unnötigen Kosten zu produzieren ?

Betrachten wir den Fall, daß das Resultat von T_2 gespeichert wird. Die Reihenfolge sei also :

$$T_2 \rightarrow \text{Register retten} \rightarrow T_1 \rightarrow u$$

Bevor der Knoten u bearbeitet wird, steht das Ergebnis von T_2 im Speicher, das von T_1 im ACCUB. Es ist also eine weitere Transferoperation nötig, damit der Befehl in der Wurzel von T überhaupt ausgeführt werden kann, denn es existiert kein Befehl für den C5x, der einen Operanden aus dem ACCUB und den anderen Operanden aus dem Speicher holen kann. Die letzte Instruktion in T_1, die Transferoperation aus dem ACCU in den ACCUB, ist also eigentlich überflüssig, denn sie bedingt eine weitere Transferoperation.

Der zweite Fall birgt das gleiche Problem in sich. Wenn man die Teilbäume in der Reihenfolge

$$T_1 \rightarrow \text{Register retten} \rightarrow T_2 \rightarrow u$$

abarbeitet, muß man nach T_1 den Inhalt des ACCUB retten, da er durch T_2 überschrieben wird. Um den ACCUB zu retten, muß der Inhalt wieder in den ACCU zurückgeschrieben werden. Diese zusätzliche Registertransferoperation ist nötig, da es keine Möglichkeit gibt, den Inhalt des ACCUB direkt in den Speicher zu schreiben. Auch hier entstehen, wie im ersten Fall, zwei Transferoperationen, die sich gegenseitig aufheben.

Beide möglichen Schedules für den Baum in Abbildung 5.4 haben zwei Dinge gemeinsam. Es werden zwei Speicheroperationen benötigt, um den Inhalt eines Registers, das am Allocation deadlock beteiligt ist, zu retten und es werden zwei überflüssige Registertransferoperationen benötigt, die den Inhalt des ACCUB erst in den ACCU schreiben und später wieder in den ACCU einlesen.

Die beiden Speicheroperationen lassen sich nicht vermeiden, da ein Register auf jeden Fall gerettet werden muß. Lassen sich aber die - eigentlich überflüssigen - Registertransferoperationen vermeiden ?

In Abbildung 5.5 ist der Baum aus Abbildung 5.4 etwas verändert dargestellt. Der Baum T_1 (gepunktet) wird aufgeteilt in den Knoten v_1 und den Baum T_1'. T_1' enthält alle Knoten von T_1 ohne v_1. Die ein- und ausgehenden Kanten von v_1 stehen für die

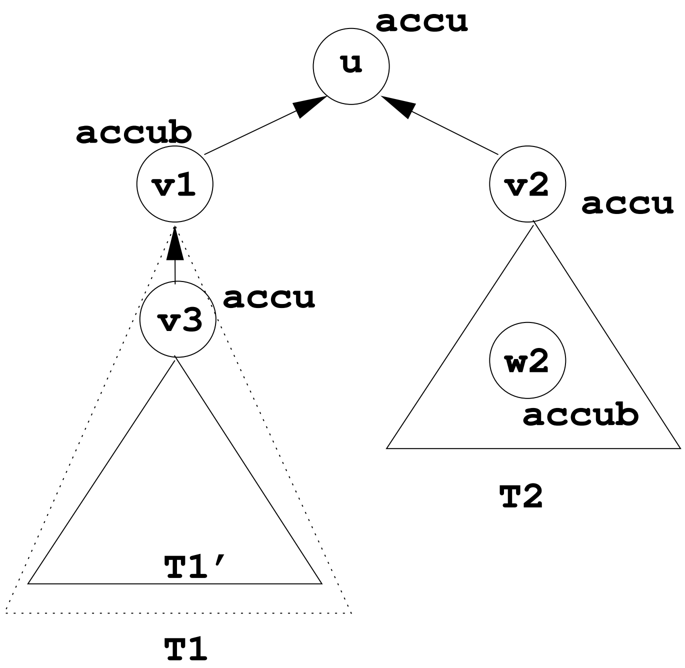

Abbildung 5.5: Der Baum aus Abbildung 5.4 etwas anders betrachtet.

oben beschriebenen (überflüssigen) Transferoperationen in den ACCUB, bzw. aus dem ACCUB heraus. Die beiden Registertransferoperationen heben sich gegenseitig auf, so daß der Baum ohne den Knoten v_1 dasselbe Resultat liefern muß.

In Abbildung 5.6 wird der Baum dargestellt, den man erhält, wenn man den Knoten v_1 aus Teilbaum T_1 entfernt. Übrig bleiben die Teilbäume T_1' und T_2. Diese Situation ist bereits aus Kapitel 4.2 Abbildung 4.4 bekannt. Der entstandene Baum hat, gemäß Definition 1 (S. 44), keinen Allocation deadlock mehr und kann mit der bekannten Methode so geschedult werden, daß kein Memory spill entsteht.

Kann man den Baum so verändern, daß der Knoten v_1 herausfällt ?

Die Lösung ist eine dynamische Kostenfunktion, die in der Grammatik bereits sicherstellt, daß ein Allocation deadlock gar nicht erst entsteht.

In der Phase der Instruktionsauswahl wird der zu überdeckende Baum bottom-up, also von den Blättern zur Wurzel, durchlaufen, und an jedem Knoten werden alle in Frage kommenden Baummuster der Grammatik zur Überdeckung des Knotens herangezogen. Jedes Baummuster produziert aber andere Kosten und es wird am Ende das Muster verwendet, das, in Bezug auf den gesamten Baum, für diesen Knoten die günstigste Alternative darstellt. Während der Überdeckung werden bereits weitere Informationen über die bereits überdeckten Bäume gesammelt, die beim Scheduling eingesetzt werden. Da die Bäume bottom-up durchlaufen werden, hat man an jedem Knoten bereits Informationen über die darunter liegenden Teilbäume gesam-

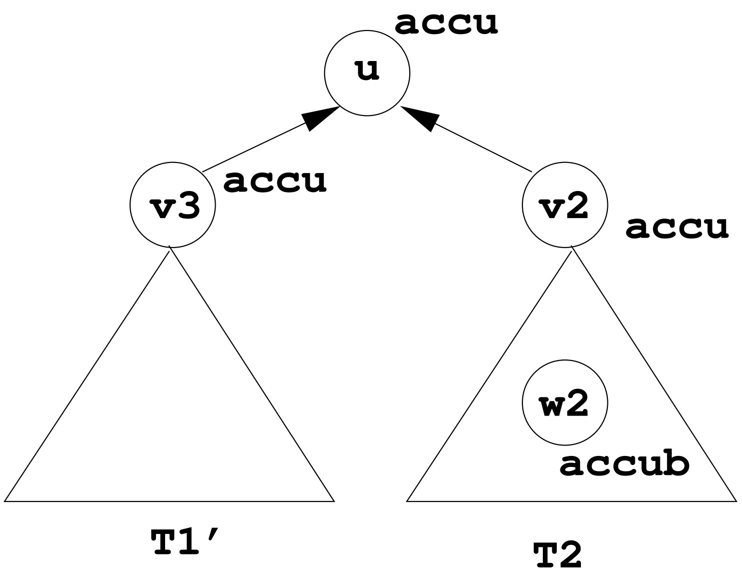

Abbildung 5.6: Der Baum aus Abbildung 5.4 nach Entfernung des Knoten v_1.

melt. Zu diesen Informationen gehören auch Angaben über die in den Teilbäumen verwendeten Registern.

Wie oben beschrieben, kann es nur zwischen dem ACCU und dem ACCUB zu einem Allocation deadlock kommen. Wenn man nun während der Instruktionsauswahl an einen Knoten kommt, für den ein Muster in Frage kommt, das den ACCUB benutzt, kann man die Informationen der darunter liegenden Teilbäume verwenden, um einen Allocation deadlock zu vermeiden.

Nehmen wir an, es würde das Muster *PLUS (accu , accub)*[1] (s. Tabelle 5.2) in Frage kommen und zu einem Allocation Deadlock führen. Nach den Erkenntnissen aus diesem Abschnitt muß der Teilbaum, der dem ACCU zugewiesen wurde, irgendwo auch den ACCUB benutzten. An der Stelle, an der das Muster *PLUS (accu , accub)* verwendet werden soll, liegt die Information, ob in dem entsprechenden Teilbaum der ACCUB schon verwendet wurde, bereits vor. Trifft dies zu, d.h. durch das zu wählende Muster wird ein Deadlock produziert, werden die Kosten für dieses Muster auf ∞ gesetzt, ansonsten werden die normalen Kosten[2] verwendet. Die Kosten ∞ bedeutet, daß das entsprechende Muster nicht verwendet wird. Eine Verwendung kann deshalb sicher ausgeschlossen werden, da der Baum auch mit den Mustern aus Tabelle 5.3 überdeckt werden kann, und diese günstiger sind als ∞. An dem Knoten, an dem auch das Muster *PLUS (accu, accub)* verwendet werden kann, kann ebenso auch das Muster *PLUS (accu, memory)* verwendet werden, um eine Überdeckung zu erreichen. Der Teilbaum, der vorher dem ACCUB zugewiesen

[1] jedes andere Muster aus Abbildung 5.2 könnte zur Erläuterung genauso verwendet werden.
[2] siehe Kostenfunktion der Prozessorbeschreibung

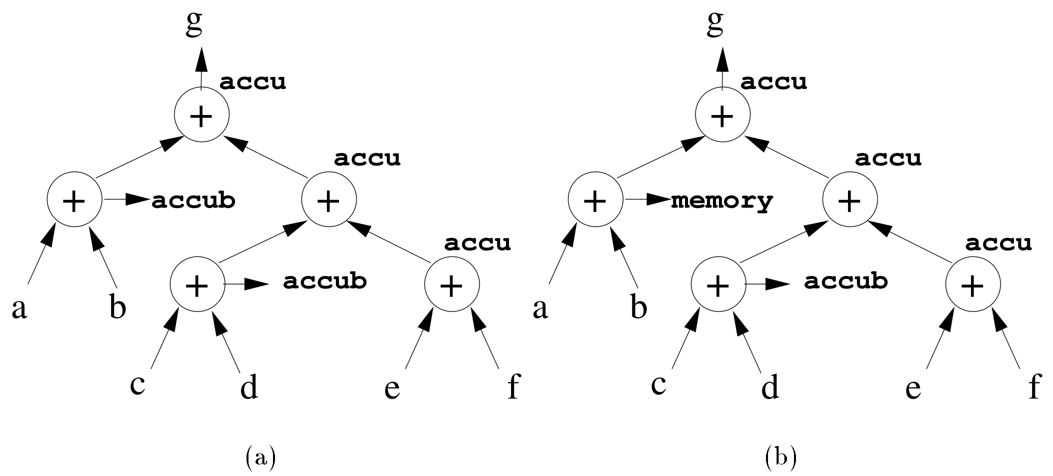

Abbildung 5.7: Zwei Überdeckungen für den DFT der Funktion $g = (a + b) + ((c + d) + (e + f))$

wurde, wird jetzt natürlich dem Speicher (*memory*) zugewiesen, was wiederum zwei Speicheroperationen erfordert, eine zum Speichern des ACCU und eine zweite, um den Operanden aus dem Speicher zu laden. In diesem Abschnitt wurde aber schon gezeigt, daß auch die Verwendung des Musters *PLUS (accu, accub)* zwei Speicheroperationen erzwingen würde. Zusätzlich würde eine Überdeckung mit dem Muster *PLUS (accu, accub)* zwei Registertransferoperationen erfordern, die die Verwendung dieses Musters sogar teurer gestalten würde.

5.2.1 Beispiel für Scheduling mit ACCUB

In Abbildung 5.7 sind zwei Möglichkeiten der Überdeckung des Ausdrucksbaums der Funktion $g = (a + b) + ((c + d) + (e + f))$ dargestellt.

In Abbildung 5.7(a) wurde die Wurzel des Baumes mit dem Muster *PLUS(accu,accub)* überdeckt und der linke Teilbaum ist dementsprechend dem ACCUB zugewiesen worden. In Abbildung 5.7(b) wurde die Wurzel des Baumes dagegen mit dem Muster *PLUS(accu,memory)* überdeckt und der linke Teilbaum ist dementsprechend dem Speicher zugewiesen worden. Wenn man keine dynamische Kostenfunktion für das Muster *PLUS(accu,accub)* verwendet, sind die Kosten für die linke Überdeckung sicherlich geringer, da statt zweier Speicheroperationen in Baum (b), in Baum (a) nur zwei Registertransferoperationen nötig sind. Nach Kapitel 5.1 werden aber Speicheroperationen mit höheren Kosten bewertet als Registertransferoperationen. Da der rechte Teilbaum in beiden Fällen mit denselben Mustern, und damit auch mit denselben Kosten, überdeckt wurde, ist Lösung (a) die günstigere. Leider führt Lösung (a) aber zu einem Allocation deadlock. Um diesen aufzulösen, werden, wie oben beschrieben, zwei Speicheroperationen benötigt. Damit werden also dieselben Kosten produziert, wie in Lösung (b). Es spricht also nichts dagegen, gleich Lösung (b) zu benutzen, da das Scheduling mit dem aus Kapitel 4.2 bekannten Algorithmus durchgeführt werden kann.

Im rechten Teilbaum gibt es noch einmal eine Situation, wie an der Wurzel des Baumes. Der Additionsknoten kann auch von beiden Mustern überdeckt werden. Da es aber in dem Rest des rechten Teilbaums nicht mehr zu einem Allocation deadlock kommt, wird durch die dynamische Kostenfunktion an dieser Stelle das Muster gewählt, das den ACCUB verwendet.

In der Tabelle 5.6 erkennt man die Unterschiede für den resultierenden Code, einmal ohne dynamische Kostenfunktion (links) und einmal mit dynamischer Kostenfunktion (mitte). Außerdem ist zum Vergleich eine Codesequenz angegeben, die den ACCUB gar nicht verwendet, wie sie also für den C25 erzeugt wird (rechts).

In der linken Sequenz wird der linke Teilbaum komplett abgearbeitet, d.h. bis zur Transferoperation, die das Ergebnis der Addition in den ACCUB kopiert. Da dieses Ergebnis aber gerettet werden muß, wird eine zweite Transferoperation eingefügt, weil der ACCUB nicht direkt in den Speicher geschrieben werden kann. Der rechte Teilbaum kann danach ohne Registerkonflikte bearbeitet werden. Schließlich wird auch das gerettete Zwischenergebnis addiert und das Endergebnis in g gespeichert.

In der mittleren Sequenz wird die dynamische Kostenfunktion verwendet, und so die Verwendung des ACCUB im linken Teilbaum ausgeschlossen. Das Ergebnis des linken Teilbaums wird direkt nach der Addition in den Speicher geschrieben und die Registertransferoperationen aus der linken Sequenz sind nicht vorhanden.

In der rechten Sequenz erkennt man, daß der Prozessor ohne den ACCUB noch zwei weitere Speicheroperationen erzwingen würde. Der Inhalt des ACCU muß zweimal gespeichert und wieder neu geladen werden.

5.2.2 Bewertung

In Tabelle 5.6 erkennt man, daß es besser ist, den ACCUB gar nicht zu verwenden, wenn dadurch ein Allocation deadlock entstehen würde. Es ist teurer, den Deadlock später aufzulösen, als ihn gar nicht erst entstehen zu lassen. Die Aufgabe der dynamischen Kostenfunktion ist es, Stellen, an denen ein Allocation deadlock entstehen kann, zu erkennen und den Deadlock zu vermeiden, indem die Verwendung des ACCUB nicht zugelassen wird. Es wird dadurch erreicht, daß der Algorithmus `OptSchedule` (S. 51 aus Kapitel 4.2 ohne Änderungen verwendet werden kann. Diese Algorithmus wurde in Kapitel 4.2 als optimal bewiesen. Das Scheduling ist durch die dynamische Kostenfunktion also auch nach Integration des ACCUB in die Grammatik optimal.

LACC	a	LACC	a	LACC	a
ADD	b	ADD	b	ADD	b
SACB		SACL	temp	SACL	temp1
LACB		LACC	c	LACC	c
SACL	temp	ADD	d	ADD	d
LACC	c	SACB		SACL	temp2
ADD	d	LACC	e	LACC	e
SACB		ADD	f	ADD	f
LACC	e	ADDB		ADD	temp2
ADD	f	ADD	temp	ADD	temp1
ADDB		SACL	g	SACL	g
ADD	temp				
SACL	g				

Tabelle 5.6: Unterschiede der möglichen Schedules; links ohne dynamische Kostenfunktion, mitte mit dynamischer Kostenfunktion, rechts ohne ACCUB

5.3 Ergänzung der Heuristik von Araujo und Malik

Die Heuristik für die Aufteilung eines DAG, die von Araujo und Malik vorgeschlagen wird (Kapitel 4.3), ist direkt aus der Prozessorstruktur abgeleitet. Die Markierung der Knoten, und damit auch die Markierung der Kanten des DAG erfolgt ausschließlich auf Grund der Strukturinformationen des Prozessors. Diese Struktur hat sich aber mit dem C5x geändert (s. Abbildung 5.1). Damit ergeben sich auch Änderungen für die Aufteilung des DAG.

In Kapitel 4.3.2.2 wurden die Natural Edges definiert. Zur Erinnerung :

Natural Edges eines DAG sind die Kanten w, die von einem Knoten v, der auf das Register r_1 abgebildet wird, zu einem Knoten u, der auf das Register r_2 abgebildet wird, führen, es aber keinen Pfad im RTG des Prozessors von r_1 nach r_2 gibt. Jeder Pfad, mit dem die Kante w überdeckt werden kann, läuft also durch den Speicher.

Die veränderte Struktur des C5x hat für Natural Edges keine Auswirkungen, d.h. jede Kante eines DAG, die für den C25 eine Natural Edge ist, ist auch für den C5x eine Natural Edge.

Begründung : Es gibt beim C25 und beim C5x nur zwei Register, in denen die Ergebnisse einer Operation abgelegt werden können, Produktregister und Akkumulator. Die Tatsache, daß der Akkumulator beim C5x die Kapazität 2 hat, ändert nichts daran, daß ein Datum, das im Akkumulator gespeichert ist und als Operand für eine Multiplikation gebraucht wird, auf jeden Fall durch den Speicher laufen muß. Es gibt keinen Weg im RTG des C5x, vom Akkumulator zum Produktregister, der nicht durch den Speicher führt (s. Abbildung

5.2). Das gleiche gilt, wenn ein Wert, der im Produktregister steht, bei einer weiteren Multiplikation als Operand verwendet wird. Auch hier für gibt es keinen Weg im RTG des C5x, der nicht durch den Speicher führt, da das Produktregister auch im C5x keinen Eigenzyklus besitzt.

Anders sieht es allerdings für die in Kapitel 4.3.2.3 definierten Pseudo-Natural Edges aus. Zur Erinnerung :

Pseudo-Natural Edges sind beide Kanten eines Kantenpaares, für die, auf Grund der Prozessorstruktur, *genau eine* Kante durch den Speicher führen muß. Die beiden Konstellationen, die Pseudo-Natural Edges für den C25 bedeuten, findet man in Kapitel 4.3.2.3 (Beispiel auf Seite 55).

Konstellation 1 kommt dadurch zustande, daß es im RTG des C25 genau einen Pfad gibt, der vom Akkumulator wieder zum Akkumulator führt, nämlich der Eigenzyklus des Akkumulators. Dies gilt für den C5x nicht mehr, denn es gibt einen zweiten Pfad, über den Akkumulator-Puffer. Damit ist die Voraussetzung für Pseudo-Natural Edges für Konstellation 1 nicht mehr gegeben, denn es führt keine der beiden Kanten unter diesen Umständen durch den Speicher.

Die Voraussetzung für die Konstellation 2 ist ähnlich, mit dem Unterschied, daß beide Kanten vom Produktregister zum Akkumulator laufen. Auch hier muß, da es nur eine Kante im RTG des C25 gibt, eine Kante auf jeden Fall über den Speicher führen. Auch diese Voraussetzung ist durch den Akkumulator-Puffer im C5x nicht mehr erfüllt. Es existiert ein zweiter Pfad vom Produktregister zum Akkumulator, nämlich der über den ACCUB (s. auch Kapitel 5.2).

Beide Voraussetzungen für Pseudo-Natural Edges sind nicht mehr gegeben, d.h. nach der Definition von Araujo und Malik gibt es für den C5x keine Pseudo-Natural Edges mehr.

Für den Algorithmus `Dismantle` (Abbildung 4.13) ergeben sich entsprechende Änderungen.

Pseudo-Natural Edges werden an zwei Stellen des Algorithmus benutzt :

1. um rekonvergente Pfade des DAG an günstigen Stellen aufzuspalten und
2. um Kanten zu spalten, deren Startknoten einen Fan-Out größer als 1 haben.

Zu Punkt 1 : Pseudo-Natural Edges können nicht mehr dazu benutzt werden, eine Unterscheidung der Kanten von rekonvergenten Pfaden herbeizuführen. Wenn es nach der Aufspaltung aller Natural-Edges noch rekonvergente Pfade gibt, sind alle Kanten dieser Pfade für den C5x “normale” Kanten, also Kanten, für die es Entsprechungen im RTG des Prozessors gibt. Deshalb ist die Chance groß, daß alle Kanten eines rekonvergenten Pfades dieselben zusätzlichen Kosten verursachen, wenn sie aufgebrochen werden und das bis dahin errechnete Ergebnis zwischengespeichert werden muß. Es wird auch in dem Algorithmus `OptSchedule` von Araujo und Malik nicht gewährleistet, daß *immer* die optimale Kante aufgebrochen wird und für den Fall, daß keine Pseudo-Natural Edge in einem rekonvergenten Pfad existiert, wird auch dort die erste Kante des Pfades aufgebrochen. Die Chance, eine "gute"[3] Kante zu spalten, ist dennoch sehr hoch, da die Knoten, an denen rekonvergente Pfade starten, immer einen Fan-Out größer als 1 haben, sonst könnten sie nicht wieder zusammenlaufen. Knoten, die einen Fan-Out größer als 1 haben, werden im letzten Schritt des Algorithmus `OptSchedule` nochmal getrennt behandelt und viele dieser Kanten werden auch aufgebrochen.

Punkt 2 : Im letzten Schritt des Algorithmus OptSchedule werden Kanten gespalten, die bei einem Knoten zu einem Fan-Out größer als 1 führen. Dies geschieht beim Original-Algorithmus in der Reihenfolge : erst Pseudo-Natural Edges, dann "normale" Kanten. Da es Pseudo-Natural Edges aber nicht mehr gibt, kann diese Unterscheidung unterbleiben. In diesem Schritt werden nicht alle Kanten an dem Knoten mit Fan-Out größer 1 gebrochen. Eine Kante bleibt erhalten (Kapitel 4.3). Da man aber an keiner Stelle eine Pseudo-Natural Edge spalten kann, hat man an allen Knoten, die in Frage kommen, die Möglichkeit, willkürlich eine Kante zu behalten und alle anderen zu brechen. Das kann dazu führen, daß man bei der Aufteilung eines DAG für den C5x häufiger die Möglichkeit der Wahl einer Kante hat, als bei der Aufteilung desselben DAG für den C25. Damit erhält man mehrere verschiedene Varianten für die Aufteilung.

5.3.1 Beispiel für die Integration des ACCUB in die Heuristik

Dieses Beispiel bezieht sich auf den DAG aus Abbildung 4.11. In diesem DAG hat man an zwei Knoten - in den Beispielen fett gezeichnet - die Möglichkeit, je eine von zwei Kanten zu wählen. Für den C25 hat man diese Auswahl nicht, da jeweils eine der Kanten eine Pseudo-Natural Edge ist und diese beim Aufspalten bevorzugt werden. Diese Wahlmöglichkeit führt zu den vier verschiedenen Aufteilungen in Abbildung 5.8. Die Lösung (a) entspricht der Aufteilung, die der Algorithmus auch für den C25 liefert, sie ergibt einen Kostenwert[4] von 66 für den kompletten DAG, ebenso wie Lösung (d). Die beiden anderen Lösungen, (b) und (c), führen sogar zu minimal höheren Kosten von 68.

[3] gut im Sinne der Kostenfunktion, also eine Kante, die nicht unnötig hohe Kosten für die Überdeckung mit Prozessorinstruktionen verursacht

[4] nach dem Kostenmaß, das für den Baumparser definiert wurde. Kapitel 7.

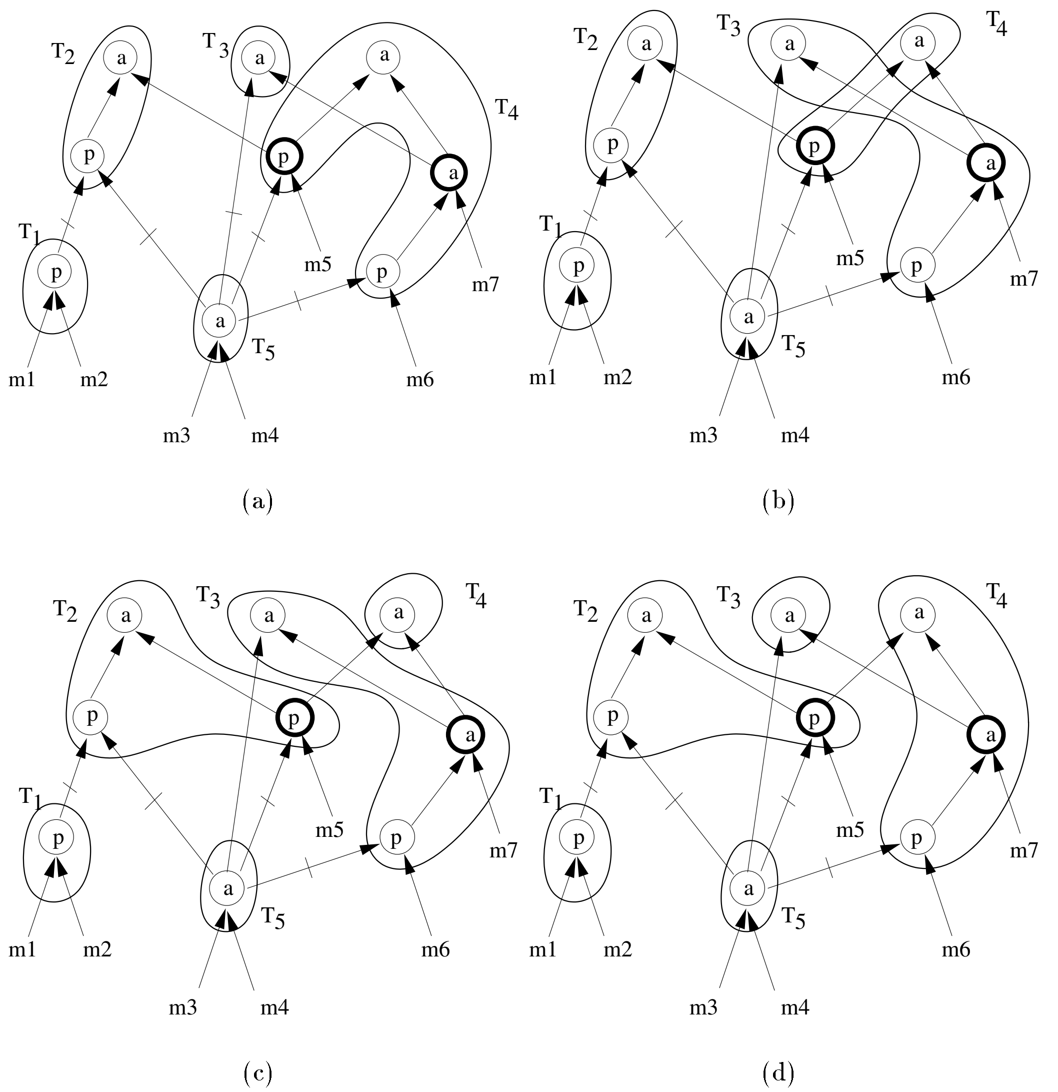

Abbildung 5.8: Unterschiedliche Aufteilungen des DAG aus Abbildung 4.11 nach Wegfall der Pseudo-Natural Edges.

5.3.2 Bewertung

Das obige Beispiel zeigt, daß die Heuristik zur Aufteilung des DAG auch ohne Pseudo-Natural Edges gute Resultate liefert. Die Möglichkeit an verschiedenen Stellen beliebig Kanten auszuwählen und aufzuspalten, führt zwar zu mehreren verschiedenen Aufteilungsvarianten, doch in der Praxis wird ein DAG mit vielen Wahlmöglichkeiten nicht sehr oft vorkommen. In erster Linie deshalb, weil nur Kanten an Knoten mit einem größeren Fan-Out als 1 diese Wahlmöglichkeiten überhaupt bieten, eben diese Kanten werden aber auch bei der Spaltung von rekonvergenten Pfaden eines DAG ausschließlich aufgebrochen. Auf Grund der sehr guten Laufzeit des Algorithmus, $O(n)$ [13], ist es jedoch möglich, den Algorithmus mehrfach zu durchlaufen, die zu brechenden Kanten zufällig zu wählen und die beste Lösung zu verwenden.

5.4 CSE-Registerzuweisung mit ACCUB

Die geänderte Struktur des C5x hat für die Aufteilung des DAG beim SA-Algorithmus aus Kapitel 4.4 keine wesentlichen Auswirkungen, da die Aufteilung unabhängig von der Prozessorstruktur ist. Das neu hinzugekommene Register, ACCUB, stellt allerdings eine weitere Möglichkeit dar, eine *common subexpression* in einem Register zu speichern.

Der Algorithmus CSE-Registerzuweisung (S. 64) wurde dennoch verändert. Und zwar in der Funktion `Decompose`. Die Funktion `Decompose` hat die Aufgabe, den zu überdeckenden DAG in Bäume zu zerlegen. Bisher wurde ein DAG ausschließlich an den CSE-Knoten aufgebrochen. Nach Araujo und Malik [13] [9] ist es aber auf Grund des azyklischen RTG des C5x möglich, Natural Edges eines DAG aufzubrechen, ohne zusätzliche Kosten zu verursachen. Dieses wird auch in die Funktion `Decompose` des SA-Algorithmus mit aufgenommen, da keine zusätzlichen Kosten für die Überdeckung entstehen und sich somit die Qualität der Lösung nicht verschlechtern kann.

Die Funktion `Decompose_neu` ist in Abbildung 5.9 dargestellt. Die Funktionen Markiere_Knoten und Markiere_Kanten entsprechen den Funktionen aus Kapitel 4.3 und werden dazu benötigt, Natural Edges des DAG zu erkennen. Nachdem die Natural Edges identifiziert wurden, wird der DAG, wie vorher auch, an allen CSE-Knoten aufgespalten. Sind danach noch Natural Edges vorhanden, werden diese ebenfalls alle aufgebrochen.

```
Input DAG D;
Output Liste list_of_trees;

begin
  MARKIERE_KNOTEN(D)
  MARKIERE_KANTEN(D)
  SPALTE_GRAPH_AN_CSE(D)
  SPALTE_GRAPH_AN_NATURAL_EDGES(D)
end
```

Abbildung 5.9: Funktion DECOMPOSE_NEU

5.4.1 Beispiel

In Abbildung 5.10 sieht man die neue Aufteilung des DAG aus Abbildung 4.18. Diese Aufteilung wird dadurch erreicht, daß der Baum T_1 aus Abbildung 4.18, hier gepunktet dargestellt, an einer Natural Edge zusätzlich aufgebrochen wird. Es entstehen die Teilbäume T_1 und T_7. Zusätzliche Kosten werden dadurch nicht verursacht (Kapitel 4.3).

Wenn man nun den CSE-Knoten in T_2 betrachtet, erkennt man, daß drei von vier Kanten, die von T_2 wegführen, in einer Multiplikation enden, nämlich in den p-Knoten in T_4, T_5 und T_7. Man sieht, daß es für diese CSE eine sehr gute Wahl ist, das Multiplikationsregister TREG0 zu benutzen, da die folgenden Multiplikationen den Wert der CSE direkt aus einem Register lesen können. Außerdem wird in keinem der drei Bäume, T_4, T_5 und T_7, das TREG0 durch eine weitere Operation überschrieben, so daß der Wert im TREG0 solange gültig bleibt, bis alle Knoten die CSE ausgelesen haben.

Wenn man aber andererseits die ursprüngliche Aufteilung verwendet, erkennt man, daß in T_1(gepunktet) eine weitere Multiplikation stattfindet. Diese Multiplikation macht eine Belegung des TREG0 ungültig, egal in welcher Reihenfolge die Teilbäume T_4, T_5 und T_7 abgearbeitet werden. Es bleibt bei dieser Aufteilung nur die Möglichkeit, die CSE von T_2 im Speicher abzulegen. Sie muß aber später viermal wieder eingelesen werden, zusammen sind also fünf Speicherzugriffe nötig. Nach der neuen Aufteilung sind aber nur zwei Speicherzugriffe nötig; Akkumulator speichern und Multiplikationsregister laden. Die übrigen Zugriffe erfolgen direkt auf das Register. Dieser Unterschied drückt sich deutlich in den Kosten[5] für die anschließende Überdeckung mit Instruktionen aus :

[5] Es ist die Kostenfunktion gemeint, die für den Baumparser definiert wird, der im Codegenerator eingesetzt wird(s. Kapitel 7)

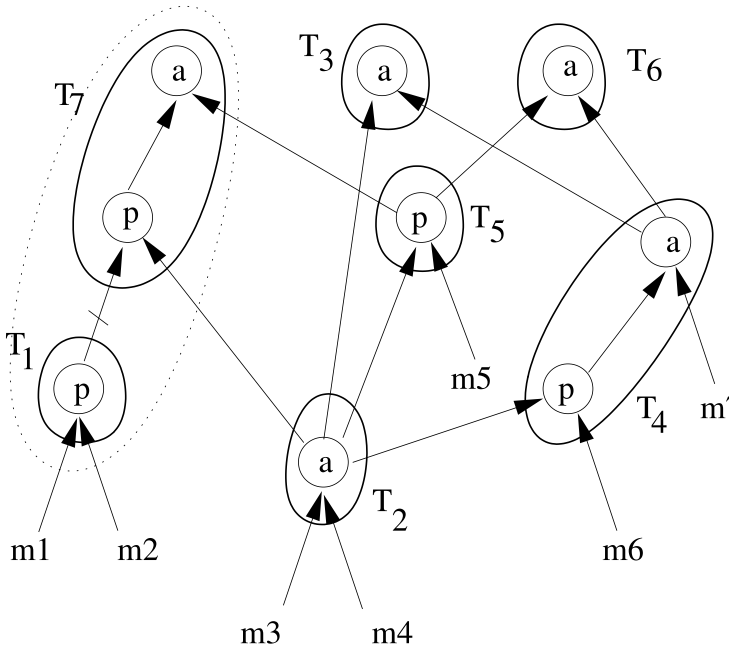

Abbildung 5.10: Neue Aufteilung für den DAG aus Abbildung 4.18.

Kosten für die alte Aufteilung :	65
Kosten für die neue Aufteilung :	56

Die beiden Werte wurden mit dem Baumparser ermittelt, die Belegung der beiden anderen CSE ist dabei :

Register für die CSE von T_4 :	ACCUB
Register für die CSE von T_5 :	MEMORY

5.4.2 Bewertung

Durch die zusätzliche Möglichkeit der Aufteilung des DAG wird die Anzahl der einzelnen Teilbäume weiter erhöht. Dadurch, daß die verbleibenden Teilbäume kleiner sind als bei der alten Aufteilung, reduziert sich die Wahrscheinlichkeit eines Registerkonflikts (Kapitel 4.4). Da in kleineren Teilbäumen meist weniger Register verwendet werden als in großen und damit Register, die als Speicherort einer CSE gewählt wurden, seltener überschrieben werden, gibt es häufiger gültige Registerbelegungen. Im Beispiel[6] in Abbildung 5.10 sieht man, daß überhaupt erst durch das

[6] An dieser Stelle sei noch einmal darauf hingewiesen, daß das verwendete Beispiel nicht kontruiert wurde, um den Algorithmus positiv darzustellen. Wie in Kapitel 4.3 erwähnt, stammt es aus einem Artikel [9], der mit der Registerzuweiung für CSE überhaupt nichts zu tun hat.

Aufbrechen der vorhandenen Natural Edge die Möglichkeit geschaffen wurde, die CSE von T_2 in einem Register zu speichern.

Durch den ACCUB als zusätzlichen Speicherort für CSE's steigen die Chancen für eine gültige Registerbelegung der CSE's ebenfalls.

Kapitel 6

Experimente

Die Experimente in diesem Abschnitt wurden mit dem entwickelten Codegenerator CODEGEN (Kapitel 7.1) auf einem System mit 128 MB physischem und 128 MB virtuellem Speicher, mit dem Betriebssystem LINUX, durchgeführt. In CODEGEN wurden alle in Kapitel 4 und 5 vorgestellten Methoden und Algorithmen implementiert, um die Änderungen, die durch die Integration das ACCUB in die Methoden zur Codegenerierung entstehen, an verschiedenen Benchmarks zu untersuchen.

6.1 Ermittlung der SA-Parameter

Der Algorithmus CSE-Registerzuweisung aus Abbildung 4.17 basiert auf der probabilistischen Optimierungsstrategie Simulated-annealing. Bei dieser Strategie werden an mehreren Stellen Parameter verwendet, die für die Qualität der gefundenen Lösung ausschlaggebend sind. Zu diesen Parametern gehören :

- Starttemperatur des Algorithmus
- Endtemperatur des Algorithmus
- Durchläufe der inneren Schleife
- Abkühlungsfaktor

Um für die Durchführung der Experimente die besten Parameter herauszufinden, wurden mit einem Benchmark[1] verschiedene Parameter untersucht. Dabei wurden die Parameter Starttemperatur, Abkühlungsfaktor und Durchläufe der inneren

[1] Der verwendete Benchmark hat etwa die Struktur des Beispiels aus 4.11. Der zu überdeckende DAG besteht aus 7 Teilbäumen und besitzt 3 CSE's.

Schleife variiert. Die Endtemperatur ist für alle Parameter-Tripel gleich, 0,1. Für jedes Parameter-Tripel wurden 10 Durchläufe gestartet, da der Algorithmus, wie in Kapitel 4.4 beschrieben, nicht bei jedem Durchlauf dieselbe Lösung findet. Die Resultate der Durchläufe sind in Tabelle 6.1 aufgelistet. In der ersten Zeile findet man die verwendeteten Parameter; T = Starttemperatur, C = Abkühlungsfaktor (cooling), L = Wiederholungen der inneren Schleife (loops). Am Ende wurde der Mittelwert über alle Durchläufe gebildet und es ist ebenfalls aufgeführt, wie häufig die optimale Lösung[2] gefunden wurde. Weiterhin wurde für vier Parameter-Tripel der Speicherverbrauch des Algorithmus angegeben, in Megabyte.

Der Tabelle 6.1 kann man entnehmen, daß die Parameter für den Algorithmus die gefundenen Lösungen nicht sehr stark beeinflussen. Bei fast allen Parameter-Tripeln wurde mindestens einmal die optimale Lösung gefunden. Auf Grund des sehr hohen Speicherverbrauchs des Algorithmus wurde, für die Durchführung der Experimente, das Parameter-Tripel gewählt, das insgesamt die geringsten Durchläufe produziert.

Starttemperatur	30
Abkühlungsfaktor	0,8
innere Schleife	5

Mit diesem Parameter-Tripel wurde der zweitbeste Durchschnitt erreicht und zweimal die optimale Lösung gefunden.

6.2 Ergebnisse der Experimente

Die Ergebnisse der Experimente mit CODEGEN sind in zwei Tabellen (6.2 und 6.3) aufgelistet. Die verwendeten Benchmarks stammen aus verschiedenen Algorithmen-Paketen:

1 Entstanden aus dem Beispiel in Abbildung 4.11

2-8 DSP-Stone-Benchmarks

9,10 MPEG-2

11 JPEG

12-17 Eigene Benchmarks

[2] Die optimale Lösung wurde "per Hand" ermittelt.

Durchlauf	**30 0.8 5**	50 0.8 5	30 0.8 10	50 0.8 10	30 0.9 5	50 0.9 5	30 0.9 10	50 0.9 10	T C L
1	**80**	80	86	83	83	86	80	80	
2	**80**	80	80	83	86	80	80	80	
3	**83**	80	83	68	80	68	80	68	
4	**80**	80	86	83	80	86	86	68	
5	**68**	80	68	83	80	80	80	83	
6	**80**	86	80	83	86	80	69	80	
7	**68**	83	80	80	80	83	69	80	
8	**80**	86	83	80	80	83	83	83	
9	**86**	83	83	80	68	83	86	80	
10	**83**	80	83	80	80	80	80	83	
Mittelwert	**78,8**	81,8	81,2	80,3	80,3	80,9	79,3	78,5	
Optimum	**2**	0	1	1	1	1	2	2	
Speicher	**30**			63	58			130	

Tabelle 6.1: Ermittlung der SA-Parameter für die Experimente

Alle Benchmarks wurden mit LANCE (Kapitel 3.1) in eine Zwischendarstellung (IR) übersetzt und mit Hilfe der diversen Tools von LANCE optimiert. Die IR wurde nachträglich teilweise manuell verändert, so daß sich die Codegenerierung nur auf existierende Basisblöcke bezieht. Kontrollstrukturen oder Unterprogrammsprünge wurden komplett entfernt.

Bei den Benchmarks aus MPEG-2, JPEG wurden kleine Teile größeren Programmen entnommen. Der entnommene C-Code ist jeweils das innere einer Schleife, in der größere Berechnungen vorkommen, so daß man große Basisblöcke für die Codegenerierung gewinnt.

6.2.1 Ergebnisse nach der Methode von Araujo und Malik

In Tabelle 6.2 sind die Überdeckungskosten und die Anzahl der Speicherzugriffe für alle Benchmarks aufgelistet, die bei der Verwendung der DAG-Teilungsmethode von Araujo und Malik (Kapitel 4.3) entstehen. In Spalte 1 stehen die Resultate, die für den Algorithmus ohne Verwendung des ACCUB ermittelt wurden, in Spalte 2 stehen die Resultate mit Benutzung des ACCUB. Der Wert in der jeweils linken Spalte steht für die Kosten, die der Baumparser in CODEGEN für die Überdeckung berechnet hat. In der rechten Spalte ist die Anzahl der Speicherzugriffe eingetragen, die in der jeweiligen Codesequenz enthalten sind. In der vorletzten Zeile wurden die Kosten und Speicherzugriffe aller Benchmarks addiert, um die prozentuale Veränderung

Algorithmus Araujo						
Nr.	Benchmark	Knoten	1		2	
1	araujo	9	66	21	66	21
2	biquad_one_section	22	57	16	57	16
3	biquad_N_section	39	66	16	66	16
4	lms	9	35	13	33	11
5	fir	26	36	14	36	14
6	fft	30	104	44	109	40
7	n_real_updates	9	15	8	15	8
8	n_complex_updates	39	87	34	87	34
9	IDCT_col	37	134	37	128	13
10	IDCT_row	22	87	25	87	21
11	j_int_DCT	45	91	37	86	29
12	Bench1	5	37	12	37	12
13	Bench2	15	93	31	85	23
14	Bench3	11	52	17	50	15
15	Bench4	9	62	19	62	19
16	Bench5	6	39	13	37	11
17	Bench6	24	163	48	157	42
Gesamt Kosten / Speicherzugriffe			1224	405	1198	360
Durchschnittliche Verbesserung in %					2,1%	11,1%

Tabelle 6.2: Experimentelle Ergebnisse mit dem Algorithmus von Araujo und Malik

durch die Integration des ACCUB (letzte Zeile) zu erhalten.

Die erzeugten Codesequenzen sind bei Dr. Rainer Leupers, LS12 Uni Dortmund, einzusehen, ebenso findet man dort die verwendeten Benchmarks und die erzeugte Zwischendarstellung von LANCE.

Die verschiedenen Codesequenzen für einen Benchmark werden durch die Endung im Dateinamen identifiziert. Für Spalte 1 besitzen die Dateien die Endung `.AR` (*Araujo*), für Spalte 2 wurde die Endung `.AR_mGRAM` (*Araujo mit ACCUB in der Grammatik*) benutzt.

Beispiel: Die generierte Codesequenz für den Benchmark lms und den Code mit ACCUB enthält z.B. die Datei `lms.c.asm.AR_mGRAM`.

6.2.2 Ergebnisse CSE-Registerzuweisung

Für die Aufteilung des Graphen nach der Methode der CSE-Registerzuweisung (Kapitel 4.4) wurden dieselben Benchmarks verwendet wie im Abschnitt zuvor. Die

Nummerierung in Tabelle 6.3 entspricht der Nummerierung aus Tabelle 6.2.

Für die Methode der Aufteilung von DAG's wurden allerdings vier verschiedene Versuchsreihen durchgeführt, da es zwei unterschiedliche Möglichkeiten gibt, den ACCUB in den Algorithmus zu integrieren :

1. Als weiteres Register, in dem eine CSE gespeichert werden kann oder
2. durch Integration des ACCUB in die Grammatik.

Die beiden Möglichkeiten wurden einzeln und in der Kombination getestet, was zu folgenden vier Varianten führt :

Spalte 4 : In dieser Spalte stehen die Resultate für den Algorithmus ohne den ACCUB überhaupt zu verwenden, wie er also in [10] für den C25 vorgestellt wurde. Einzige Änderung gegenüber dem Originalalgorithmus ist die in Kapitel 5.4 beschriebene veränderte Aufteilung des DAG.

Spalte 5 : Der ACCUB wurde in die Grammatik integriert.

Spalte 6 : Der ACCUB wird nur als zusätzliche Speichermöglichkeit verwendet.

Spalte 7 : Die Resultate erhält man, wenn man beide Möglichkeiten kombiniert. Der ACCUB wird sowohl in die Grammatik integriert, als auch für die Speicherung einer CSE verwendet.

Spalte 4 - 7 sind, wie auch in Tabelle 6.2 die Spalten 1 und 2, zweigeteilt. In der linken Spalte stehen jeweils die ermittelten Überdeckungskosten und in der rechten Spalte steht die Anzahl der verwendeten Speicherzugriffe. Außerdem wurde eine Spalte für die Anzahl der *common subexpressions* der Benchmarks aufgenommen (*CSE*) und in Spalte 3 wurde - der Vollständigkeit halber - der Kostenwert eingetragen, den man für die Überdeckung erhält, wenn alle CSE's im Speicher stehen würden (INITIAL_COST im Algorithmus CSE-Registerzuweisung).

Die beiden letzten Zeilen enthalten die Summen der Kosten bzw. der Speicherzugriffe und die prozentuale Veränderung gegenüber der Variante ohne Verwendung des ACCUB (Spalte 4).

Die Endungen für die vier Varianten der CSE - Registerzuweisung sind :

- original Algorithmus - .SA (simulated annealing)
- ACCUB in der Grammatik - .SA_mGRAM
- ACCUB als CSE-Speicher - .SA_mSA
- Kombination beider Möglichkeiten - .SA_mGRAM_mSA

Algorithmus CSE										
Nr.	CSE	3	4		5		6		7	
1	3	73	62	20	62	20	56	17	56	17
2	1	61	61	17	61	16	59	17	59	16
3	3	77	72	25	72	25	72	25	72	25
4	1	34	33	13	32	11	33	13	32	11
5	1	36	36	14	36	14	36	14	36	14
6	6	91	88	34	78	31	81	30	62	24
7	0	15	15	8	15	8	15	8	15	8
8	3	67	65	25	65	25	65	25	65	25
9	2	136	136	36	128	29	130	34	128	31
10	2	92	92	27	86	23	87	24	86	21
11	4	115	111	34	111	34	92	28	92	28
12	2	40	38	12	38	12	28	7	28	7
13	0	93	93	31	85	23	93	31	85	23
14	2	59	59	19	54	16	54	17	52	15
15	3	69	66	20	66	20	62	19	62	19
16	1	43	41	13	39	11	37	11	39	11
17	4	163	144	43	138	38	144	43	138	38
		1264	1212	391	1166	356	1144	363	1107	340
		-4,3%			3,8%	9,0%	5,6%	7,2%	8,7%	13,0%

Tabelle 6.3: Experimentelle Ergebnisse mit dem SA-Algorithmus in verschiedenen Variationen

6.3 Auswertung der Ergebnisse

In den 4 Charts, Abbildung 6.1 bis 6.4, sind die Ergebnisse der Tabellen 6.2 und 6.3 graphisch dargestellt. Die Beschriftung der X-Achsen aller 4 Charts entspricht den Spaltennummern der beiden Tabellen.

In Abbildung 6.1 ist der durchschnittliche Kostenwert aller Benchmarks aufgetragen. Man kann erkennen, daß durch den Einsatz des ACCUB (Spalte 2, 5, 6 und 7) die vom Baumparser ermittelten Kosten im Schnitt gesunken sind, unabhängig davon, wie der ACCUB verwendet wurde. Für die beiden Algorithmen nach der Methode von Araujo und Malik (Spalte 1 und 2) ergibt sich eine Reduktion der durchschnittlichen Kosten von 72, für den Algorithmus ohne ACCUB, auf 70, wenn der ACCUB in die Grammatik des Baumparsers integriert wird. Prozentual bedeutet dies eine Verbesserung um 2,1% (Abbildung 6.2).

Für die vier Varianten, in denen die Methode von Leupers untersucht wurde, zeigt sich dieselbe Tendenz. Im Vergleich mit dem ursprünglichen Algorithmus (Spalte 4), werden bei allen drei Varianten (Spalte 5, 6 und 7) geringere Kosten durch den Baumparser ermittelt. Durch die alleinige Integration des ACCUB in die Grammatik sinken die durchschnittlichen Kosten von 71 auf 69, bei der Verwendung des ACCUB als Möglichkeit eine CSE zu speichern, sinken die Kosten sogar auf 67. Die höchste Reduktion der Kosten erhält man allerdings durch die Kombination beider Integrationsmöglichkeiten. Der Kostenwert fällt hierbei von 71 auf 65, was einer prozentualen Senkung um über 8,7% entspricht. Bei dem getrennten Einsatz beider Varianten sind es 3,8% bzw 5,6%.

Der Kostenwert, den der Baumparser ermittelt, ist allein kein ausreichendes Kriterium, um die Verbesserung, die durch Integration des ACCUB erreicht wird, zu bewerten. Die Kostenfunktion, anhand derer die Kosten berechnet werden, ist teilweise auch eine Frage der Definition.

Die Verbesserungen, die durch den ACCUB erzielt werden, erreicht man in erster Linie durch Ersetzung von Speicheroperationen durch Registertransferoperationen. Wenn man nun die Kosten für einen Speicherzugriff in der Grammatik sehr hoch ansetzt, werden die Differenzen der Kostenwerte größer und damit steigt die prozentuale Verbesserung der Lösungen.

Da aber die Unterschiede hauptsächlich auf Ersetzung von Speicheroperationen beruhen, ist die Anzahl von Speicheroperationen in den resultierenden Codesequenzen ein gutes Kriterium für die Qualität der erzeugten Codes. Ein anderes, gutes Kriterium für die Codequalität ist die Codelänge. Wie zu Beginn der Arbeit erwähnt, ist die Größe des verfügbaren Speichers begrenzt. Bei den meisten Testreihen konnte jedoch durch den ACCUB keine nennenswerte Reduzierung der Codelänge erreicht werden. Das Maximum der Einsparung lag bei ca. 2%. Deshalb wird die Codelänge hier nicht als Kriterium angeführt.

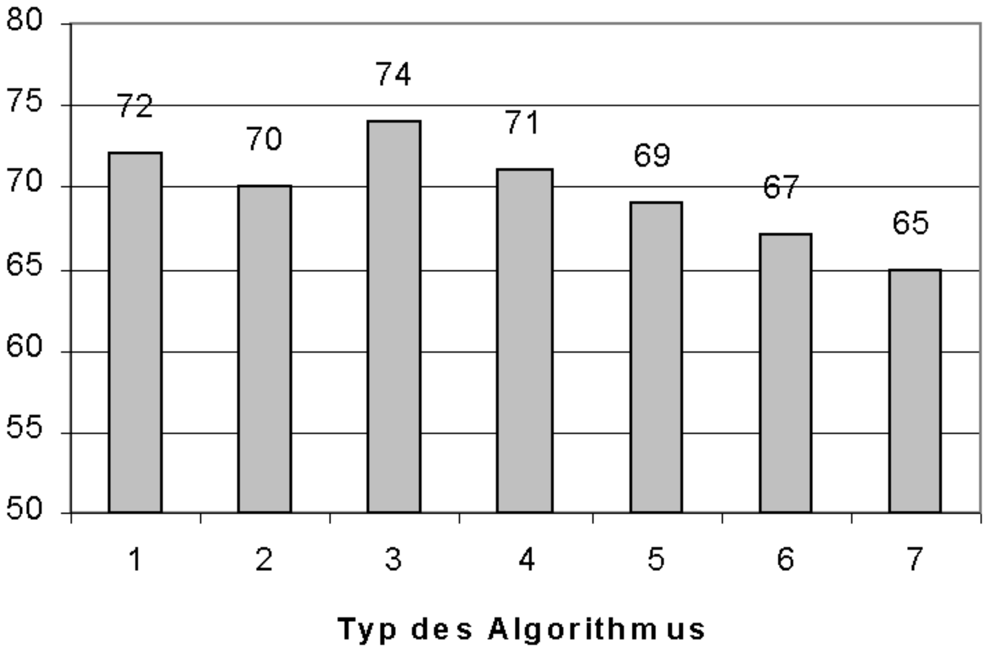

Abbildung 6.1: Durchschnittliche Kosten pro Benchmark

Die Resultate, die in Bezug auf die Anzahl der Speicheroperationen erzielt wurden, sind in den Abbildungen 6.3 und 6.4 dargestellt.

Für den Algorithmus von Araujo und Malik wurde durch den ACCUB die Anzahl der Speicherzugriffe von durchschnittlich 24 auf 11 gesenkt (Abbildung 6.3), was einer Verbesserung um über 11% entspricht (Abbildung 6.4). Mit dem Simulated-annealing Algorithmus wurde eine Verbesserung von durchschnittlich 23 auf 21 Speicherzugriffe erreicht, wenn die beiden oben erwähnten Variationsmöglichkeiten unabhängig voneinander eingesetzt wurden.

Die prozentuale Verbesserung für den Einsatz des ACCUB in die Grammatik beträgt 9,0% (s. Abbildung 6.4), bei der Verwendung des ACCUB als Speicherplatz für CSE's 7,2 %. In der Kombination beider Möglichkeiten wurde sogar eine Reduktion auf durchschnittlich 20 Operationen erreicht, was einem Gewinn von 13,0% entspricht.

Diese Werte sind ein eindeutiges Indiz für eine Verbesserung der Codequälität durch den ACCUB.

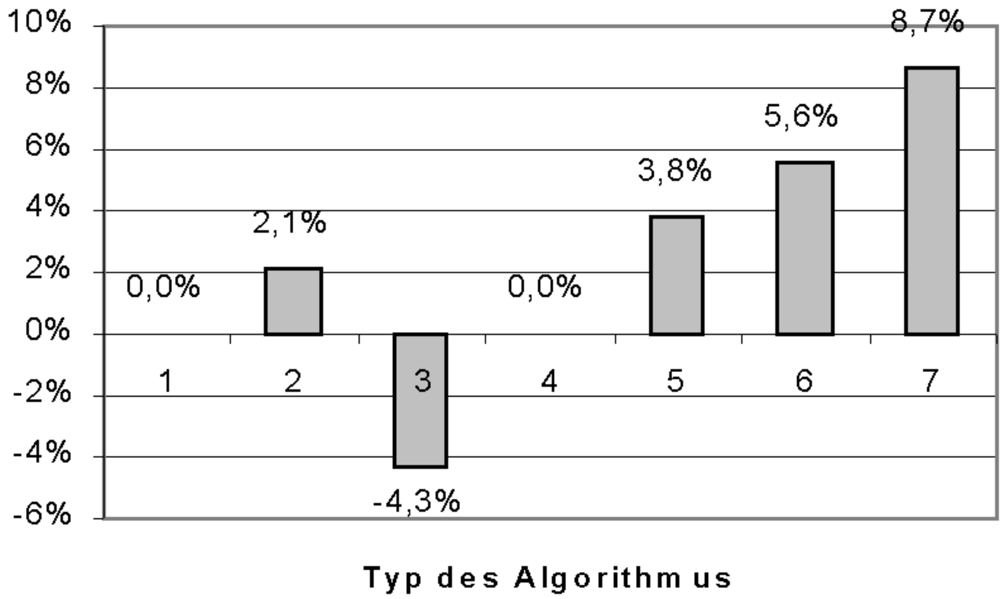

Abbildung 6.2: Prozentuale Reduzierung der Kosten durch Integration des ACCUB

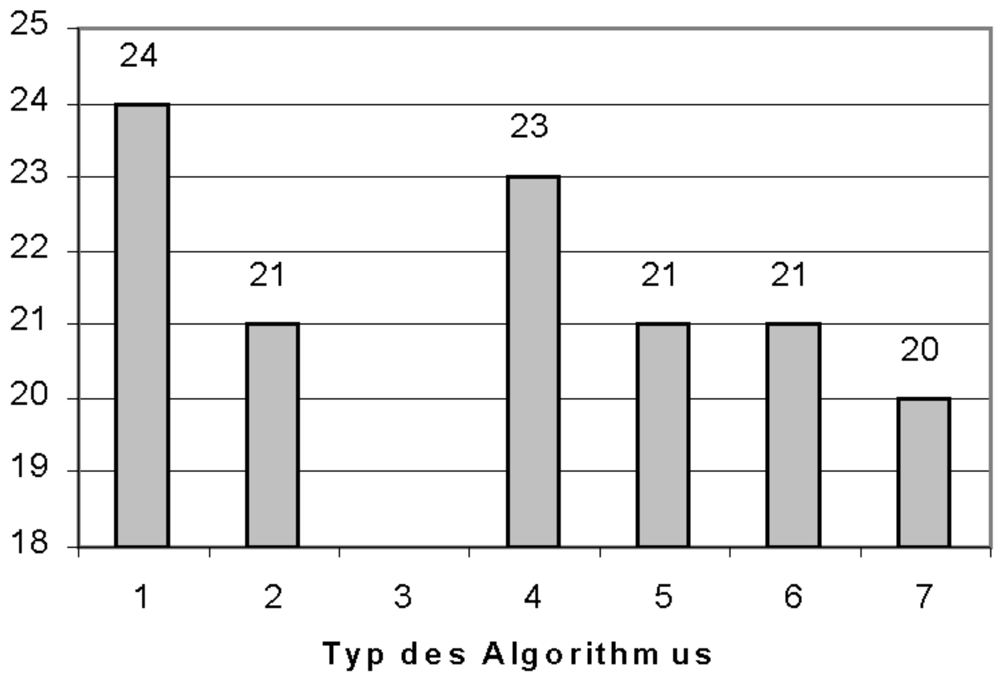

Abbildung 6.3: Durchschnittliche Speicherzugriffe pro Benchmark

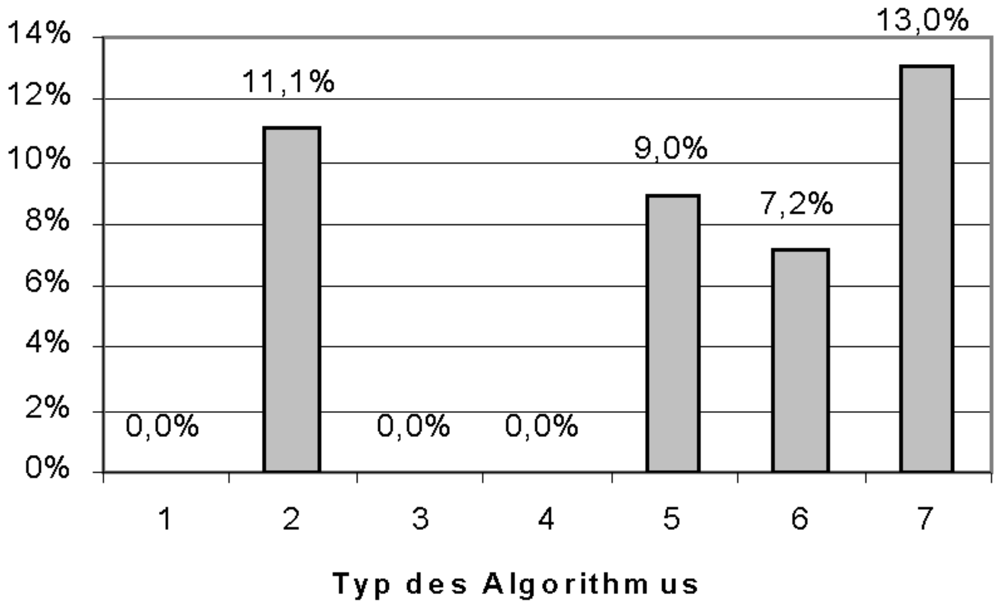

Abbildung 6.4: Reduzierung der Speicherzugriffe durch Integration des ACCUB

Kapitel 7

Implementierung

7.1 Programm CODEGEN

Das Programm CODEGEN wurde während dieser Arbeit entwickelt, um die verschiedenen Algorithmen zu implementieren. CODEGEN wurde zur einfacheren Handhabung in das C-Frontend LANCE integriert und sollte auch nur von dort aufgerufen werden. Für das Programm wird eine Konfigurationsdatei benötigt, mit deren Hilfe man die verschiedenen Funktionen von CODEGEN aktivieren kann. In LANCE erscheinen die Konfigurationsmöglichkeiten als eigenes Menü (Abbildung 7.1), in dem alle Variablen und Schalter verändert werden können.

Die Variablen haben dabei folgende Bedeutung :

USE_ACCUB Mit diesem Schalter kann man auswählen, ob der ACCUB in der Grammatik verwendet werden soll oder nicht.

USE_ARAU0_OR_SA1 Dieser Schalter wählt den verwendeten Algorithmus.

USE_ACCUB_SA Mit diesem Schalter wird bestimmt, ob der ACCUB beim SA-Algorithmus zur Speicherung der CSE gewählt werden kann.

SA_START_TEMP Hier wählt man die Anfangstemperatur für den SA-Algorithmus.

SA_END_TEMP Bestimmt die Endtemperatur des Algorithmus. Da in der Konfigurationsdatei nur INTEGER-Werte möglich sind, hier aber ein FLOAT-Wert benötigt wird, wird der tatsächlich eingesetzte Wert durch 10 geteilt, so daß die Starttemperatur 0,1 ist, wenn im Menü der Wert 1 eingetragen ist.

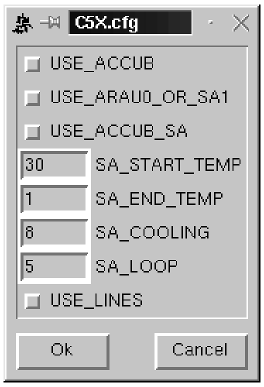

Abbildung 7.1: Konfigurationsmenü von CODEGEN

SA_COOLING Abkühlungsfaktor des SA-Algorithmus. Auch hier wird der Wert der Konfigurationsdatei durch 10 geteilt, um den tatsächlichen Faktor zu erhalten.

SA_LOOP Steht für die Anzahl der Durchläufe der inneren Schleife.

USE_LINES Mit diesem Schalter kann bestimmt werden, ob der ausgegebene Assemblercode mit oder ohne Zeilennummern generiert wird. Die Zeilennummern spielen bei der Anzeige des Codes in LANCE eine Rolle.

CODEGEN wurde mit C++ unter Linux entwickelt, funktioniert aber ebenso unter Solaris.

7.2 OLIVE-Beschreibung des C5x

In diesem Kapitel werden einige Punkte der Prozessorbeschreibung erläutert. Die komplette Beschreibung, die als Eingabe für Olive verwendet wurde, findet man im Anhang A.

7.2.1 Kostenteil

Im Kostenteil jeder Regel der Grammatik werden die Kosten berechnet, die entstehen, wenn das Baummuster dieser Regel zur Überdeckung des Eingabebaumes verwendet wird. Außerdem werden verschiedene Daten über die Knoten des zu überdeckenden Baumes gesammelt, die für das Scheduling von Bedeutung sind (s. Abbildung 7.2).

```
accu: ADD(accu,preg)
{ $cost[0].c = $cost[2].c + $cost[3].c + 1;
  $cost[0].reg = $cost[2].reg->Union($cost[3].reg);
  $cost[0].reg_comp = $cost[2].reg_comp->Union($cost[3].reg_comp);
  $cost[0].reg->Insert(AREG);
  $cost[0].reg_comp->Insert(AREG);
  $cost[0].memlist = $cost[2].memlist->Copy();
  $cost[0].memlist->Append($cost[3].memlist);
}
=
{ if (do_mems)
  { if ($getcost[3].memlist->Length() > 0)
    { $action[3](); $getcost[3].memlist->Delete((void*)done_tmpmem);
    }
    else
      if ($getcost[2].memlist->Length() > 0)
      { $action[2](); $getcost[2].memlist->Delete((void*)done_tmpmem);
      }
  }
  if (emit_code)
  { $action[2]();
    $action[3]();
    sprintf(s,"\n\tAPAC");
    Add_Code(strdup(s));
  }
};
```

Abbildung 7.2: Beispiel einer Regel der Prozessorbeschreibung

cost[0].c Diese INTEGER-Variable enthält die Kosten des darunterliegenden Teilbaums, cost[2].c + cost[3].c, zuzüglich der Kosten für das Muster selbst. Diese sind bei allen Mustern 1. Die einzige Ausnahme bildet ein Speicherzugriff. Immer, wenn ein Wert in den Speicher geschrieben oder aus dem Speicher gelesen wird, steigen die Kosten um 3. Damit wird, wie in Kapitel 5.1 gefordert, jeder Speicherzugriff teurer als ein Registertransfer und immer, wenn ein Registertransfer möglich ist, wird diese Lösung günstiger sein.

cost.reg Diese Variable steht für eine Menge. Diese Menge enthält alle Register, die in den unterliegenden Teilbäumen für die Überdeckung des Baumes benutzt wurden (s. Register-Set im Algorithmus `OptSchedule` S. 51). Verwendet wird diese Menge fürs Scheduling (Kapitel 4.2).

cost.memlist In dieser Liste werden alle Knoten des zu überdeckenden Baumes gespeichert, die zu einem Speicherzugriff führen. Diese Liste wird ebenfalls beim Scheduling der Bäume benutzt.

cost.reg_comp Diese Menge hat fast dieselbe Funktion, wie die Menge *cost.reg*. Der Unterschied ist, daß in *cost.reg_comp* alle Register enthalten sind, die in den darunterliegenden Teilbäumen verwendet wurden. Diese Menge wird benötigt, um für den SA-Algorithmus einen Registerkonflikt zu erkennen.

```
accu: ADD(accu,accub)
{ if ($cost[2].reg->Contains(ABREG))
    $cost[0].c = 1000000;
  else
    $cost[0].c = $cost[2].c + $cost[3].c + 1;
```

Abbildung 7.3: Einsatz der dynamischen Kostenfunktion

7.2.1.1 Dynamische Kostenfunktion für den ACCUB

In Kapitel 5.2 wurde erläutert, wie man mit einer dynamischen Kostenfunktion einen Allocation deadlock schon in der Grammatik vermeiden kann. In Abbildung 7.3 ist ein Stück des Kostenteils einer Regel gezeigt, der die dynamische Kostenfunktion verwendet.

Wenn in dem Teilbaum, für den das Nicht-Terminal *accu* in *ADD(accu,accub)* steht, der ACCUB bereits verwendet wird, werden die Kosten mit 1000000 angesetzt. Wird der ACCUB nicht verwendet, sind die Kosten wie üblich 1.

Die dynamische Kostenfunktion findet man in allen Regeln, die das Nicht-Terminal-Symbol *accub* im Muster enthalten.

7.2.2 Aktionsteil

Der Aktionsteil aller Regeln ist zweigeteilt. In Kapitel 4.2 wurde diese Zweiteilung erläutert. Im ersten Teil, (*do_mems*), werden alle Teilbäume bearbeitet, die in einem Speicherzugriff enden. Dieser Teil entspricht der Funktion `OptSchedule` aus dem Algorithmus von S. 51.

Der zweite Teil, (*emit_code*), entspricht der Prozedur `FreeSchedule`. Hier werden die verbliebenen Teilbäume ohne Deadlock bearbeitet.

Kapitel 8

Zusammenfassung

Die Ergebnisse der Experimente aus Kapitel 6 haben deutlich gezeigt, daß durch den ACCUB Vorteile für die Codegenerierung entstehen. In erster Linie können durch den ACCUB Speicheroperationen durch Registertransferoperationen ersetzt werden. Dies bedeutet, daß weniger Energie verbraucht wird, um das entsprechende Programm abzuarbeiten. Wenn man heute z.B. Werbung für Mobiltelefone sieht, ist die maximale Gesprächsdauer und die Stand-By-Zeit der Geräte immer noch ein Kriterium, das den Kunden von einem bestimmten Gerät überzeugen soll. Vorteile im Prozentbereich können über den Erfolg eines Produktes entscheiden.

Das Thema dieser Arbeit ist die Codegenerierung auf Basisblockebene. D.h. verschiedene Änderungen an der Architektur des C5x konnten gar nicht mit einbezogen werden. Z.B. die Erweiterung der ARAU. Diese Änderungen machen sich erst in anderen Phasen der Codegenerierung bemerkbar.

In Kapitel 2 wurde die Architektur des C5x beschrieben. Die heterogene Struktur des Prozessors, sowie alle funktionalen Einheiten wurden dargestellt.

In Kapitel 4 wurden verschiedene Methoden erläutert, die bei der Codegenerierung für eine heterogene Architektur eine wesentliche Rolle spielen. Die 4 vorgestellten Methoden wurden anhand eines Vorgängermodells des C5x, dem TMS 320C25, erläutern. Zwei der Methoden beschäftigten sich mit der optimalen Instruktionsauswahl für einen Ausdrucksbaum. Die beiden anderen wiederum hatten eine günstige Aufteilung eines Ausdrucksgraphen als Ziel, da Programme oder Funktinen, für die Code erzeugt werden soll, nur sehr selten direkt als Baum dargestellt werden können. Es wurde ebenfalls das RTG-Kriterium eingeführt, das bei zwei Methoden eine wichtige Rolle spielt.

Auf Grund der veränderten Struktur des C5x, gegenüber dem C25, mußten die verwendeten Methoden an verschiedenen Punkten verändert werden. In Kapitel 5 wurden diese Änderungen erläutert und die Vor- und Nachteile für die Codegenerierung aufgezeigt.

In Kapitel 6 wurden die Änderungen anhand verschiedener Benchmark untersucht.

Der Akkumulator war sicherlich ein Nadelöhr in der Architektur der Vorgängermodelle des C5x. Ein Großteil aller Daten läuft durch den ACCU, da er Ziel jeder Operation der ALU ist. Das bedeutet, daß häufig Zwischenergebnisse aus dem ACCU gesichert werden müssen, damit sie nicht überschrieben werden. Bevor der ACCUB in die Architektur integriert wurde, blieb nur der Hauptspeicher als Möglichkeit zum Zwischenspeichern, dies kann nun aber im ACCUB erfolgen, was an vielen Stellen Vorteile für den resultierenden Code bedeutet.

In dieser Arbeit wurde rein sequentieller Code erzeugt. In den Experimenten zeigte sich aber, daß alleine durch Integration des ACCUB in die Codegenerierung keine deutliche Reduzierung der Codelänge erreicht werden konnte. Wie es aber für DSP's üblich ist, gibt es beim C5x die Möglichkeit, verschiedene Arbeiten parallel durchzuführen. Während einer ALU-Operation kann z.B. gleichzeitig ein Register in der ARAU verändert werden oder Bedingungen für Sprünge können in der PLU berechnet werden, ohne den Inhalt des Akkumulators zu verändern, usw. All dies sind Möglichkeiten, den automatisch generierten Code zu parallelisieren und dadurch zu kompaktieren. Für weitere Forschung in diesem Bereich sind reichlich Themen gegeben.

Literaturverzeichnis

[1] FRASER, HANSON UND PROBESTING 1993: Engineering a simple, efficient code generator. *Journal of the ACM 22, 12, S. 248 - 262*

[2] AHO UND JOHNSON 1976: Optimal code generation for expression trees. *Journal of the ACM 23, 3, S. 488 - 501*

[3] LEUPERS, DINGEL 1999: LANCE User's Guide. *Homepage von Rainer Leupers http:\\ls-12.www.informatik.de\~leupers*

[4] AHO, GANAPATHI UND TJIANg 1989: Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems 11,4, S. 491-516*

[5] TEXAS INSTRUMENTS 1993: TMS320C5X User's Guide

[6] CATTELL 1978: Formalisation and automativ derivation of code generators. *PhD thesis, Carnegie-Mellon University, Pittsburg*

[7] WESS 1990: On the optimal code generation for signal flow computation. *Proc. Int. Conf. Circuits and Systems, Volume 1 (1990), S. 444-447*

[8] GARAY UND JOHNSON 1979: Computers and Intractability. *W.H. Freemann and Company, New York*

[9] ARAUJO UND MALIK 1998: Code generation for Fixed-point DSPs. *ACM TODEAS Vol. 3, Februar 1998*

[10] LEUPERS 1999, angemeldet zur Veröffentlichung

[11] AHO, SETHI UND ULLMAN 1986: Compilers - Principles, Technics and Tools. *Addison-Wesley*

[12] ZIVOJNOVIC UND ANDERE 1994: DSP-Stone - A DSP-oriented benchmark methodology. *Int. Conf. on Signal Processing Applications and Technology (ICSPAT) 1994*

[13] Araujo und Malik 1995: Optimal code generation for embedded memory non-homogeneous register architectures. *8th Internation Symposium on System Synthesis, September 1995, S. 36-41*

Abbildungsverzeichnis

Tabellenverzeichnis

Index