Instruktionsanordnung und Registerallokation auf der Basis ganzzahliger linearer Programmierung für den digitalen Signalprozessor ADSP-2106x

Diplomarbeit

Lehrstuhl für Übersetzerbau und Programmiersprachen
FB14 Informatik
Universität des Saarlandes

Daniel Kästner
kaestner@cs.uni-sb.de

9. September 1997

Hiermit erkläre ich an Eides statt, daß ich für die Anfertigung dieser Arbeit keine anderen als die angegebenen Quellen verwendet habe. Ich versichere, sie noch keinem anderen Prüfungsamt vorgelegt zu haben.

Saarbrücken, im September 1997

Daniel Kästner

Danksagung

Herrn Prof. Dr. R. Wilhelm danke ich für die Vergabe des interessanten und anspruchsvollen Themas sowie für seine Hilfsbereitschaft und Unterstützung. Bei Herrn Prof. Dr. W. Paul möchte ich mich für seine Bereitschaft bedanken, diese Arbeit zu begutachten. Danken möchte ich weiterhin H. Rieder für die Bereitstellung der Prozessordokumentation, der Entwicklungsumgebung und vieler Testprogramme sowie für die informativen und anregenden Gespräche. Herrn Prof. Dr. K. Mehlhorn danke ich für die Erlaubnis der Benutzung von Hard- und Softwareressourcen des Max-Planck-Institutes für Informatik. Mein besonderer Dank gilt weiterhin C. Ferdinand für viele wertvolle Hinweise, die intensive Betreuung und das Korrekturlesen dieser Arbeit. Danken möchte ich auch meinem Freund und Kollegen M. Langenbach für die zahlreichen anregenden und ergiebigen Diskussionen und die enge Zusammenarbeit während der Entwicklung und Implementierung der gemeinsamen Programmteile.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Ein Übersetzer erhält als Eingabe ein Programm, transformiert dieses in der Regel in eine Zwischendarstellung und führt anschließend die Codeerzeugung aus, die ein semantisch äquivalentes Programm in der Zielsprache, dem Instruktionssatz der Zielmaschine, ausgibt. Die Aufgabe der Codeerzeugung umfaßt drei Teilaufgaben: Codeselektion, Instruktionsanordnung und Registerallokation. Da alle diese Teilaufgaben komplexe Probleme darstellen, führen viele Compiler die Codeerzeugung in drei weitgehend unabhängigen Phasen durch, wobei die einzelnen Teilprobleme in der Regel durch Einsatz heuristischer Verfahren gelöst werden.

Aufgrund der Verbreitung von RISC-Prozessoren, bei denen meist nur eine mögliche Operationsimplementierung zur Verfügung steht und die Verwaltung der funktionalen Einheiten und der Pipelines dem Compiler überlassen wird, konzentriert sich der Schwerpunkt des Übersetzungsprozesses weitgehend auf die Teilaufgaben der Instruktionsanordnung und der Registerallokation. Da sich diese gegenseitig stark beeinflussen, verstärken sich die Forschungsbemühungen, die beiden Phasen möglichst eng miteinander zu koppeln.

Modellierungen mit Hilfe ganzzahliger linearer Programmierung bieten die Möglichkeit, Instruktionsanordnung und Aspekte der Registerallokation in einer einheitlichen Problembeschreibung zu integrieren und gemeinsam zu lösen. Darüber hinaus ist es im Prinzip möglich, eine optimale Lösung des Instruktionsanordnungs- und Registerallokationsproblems zu ermitteln – allerdings unter Inkaufnahme hoher Rechenzeiten. Die klassischen heuristischen Verfahren haben trotz ihrer Schnelligkeit den Nachteil, daß nur approximative, suboptimale Lösungen ermittelt werden können und keinerlei Information über die Qualität der gefundenen Lösung verfügbar ist.

Im Rahmen dieser Diplomarbeit soll die Realisierbarkeit einer optimalen Instruktionsanordnung unter Einbeziehung von Aspekten der Registerallokation auf der Basis ganzzahliger linearer Programmierung für einen digitalen Signalprozessor untersucht werden.

Dazu werden folgende Untersuchungen durchgeführt:

- Zwei verschiedene ILP-Modellierungsansätze aus dem Gebiet der Architektursynthese werden vorgestellt und an eine konkrete Zielarchitektur, den digitalen Signalprozessor ADSP-2106x, angepaßt.
- Die Erweiterbarkeit beider Modellierungen wird im Hinblick auf die Anforderungen einer globalen Instruktionsanordnung mit verschränkter Registerzuteilung überprüft.
- Verschiedene Heuristiken zur Ermittlung einer Näherungslösung auf der Basis der ganzzahligen linearen Programmierung werden vorgestellt.
- Anhand von Testprogrammen werden beide Modellierungen bezüglich ihres Laufzeitverhaltens miteinander verglichen, die Heuristiken im Hinblick auf Laufzeit und

Qualität des generierten Codes untersucht und schließlich mit konventionellen, graphbasierten Verfahren zur Instruktionsanordnung verglichen. Diese werden im Rahmen von [Lan97] im Detail analysiert.

Die vorliegende Arbeit besitzt folgenden Aufbau: In Kapitel 2 wird die zugrundeliegende Architektur, der digitale Signalprozessor ADSP-2106x, vorgestellt und ein Überblick über die wesentlichen Hardwarecharakteristika gegeben. Im Rahmen von [Lan97] werden graphbasierte Algorithmen zur Instruktionsanordnung und Registerallokation untersucht und für denselben Prozessor getestet; daher ist dieses Kapitel in beiden Arbeiten enthalten. Kapitel 3 ist der Beschreibung der Instruktionsanordnungs- und Registerallokationsproblematik gewidmet. Die zentralen Begriffe und Datenstrukturen werden definiert und verschiedene Berechnungsansätze skizziert. In Kapitel 4 werden die mathematischen Grundlagen, die dem ILP-Ansatz zugrundeliegen, präsentiert und die wichtigste Berechnungsmethode, der Branch-and-Bound-Algorithmus, vorgestellt. Die beiden untersuchten Modellierungen werden in Kapitel 5 detailliert beschrieben und in Kapitel 6 an die Architektureigenschaften des ADSP-2106x, sowie an die Anforderungen globaler Instruktionsanordnung und Registerallokation angepaßt. In Kapitel 7 werden verschiedene Approximationen präsentiert, die bei Verzicht auf die Ermittlung einer beweisbar optimalen Lösung schneller und auch bei größeren Eingabeprogrammen zu guten Ergebnissen führen. Die Implementierung wird in Kapitel 8 beschrieben; im Anschluß daran werden alle vorgestellten Modellierungsvarianten zur Lösung verschiedener Testprogramme eingesetzt und die Ergebnisse dieser Untersuchungen präsentiert. Der Vergleich mit konventionellen, graphbasierten Ergebnissen beruht dabei auf den in [Lan97] ermittelten Testresultaten. In Kapitel 9 werden die wichtigsten Ergebnisse überblicksartig zusammengefaßt und ein Resümee gezogen. Der Anhang enthält Einzelheiten über den Instruktionssatz des ADSP-2106x; anschließend werden einige verwendete Algorithmen skizziert. Für alle untersuchten Testprogramme wird die Eingabe in Assemblercode angegeben und einer optimierten Ausgabecodesequenz gegenübergestellt. Schließlich werden im Glossar Begriffe, die ohne explizite Definition verwendet wurden, erklärt.

Kapitel 2

Architekturüberblick

2.1 Charakteristika digitaler Signalprozessoren

Während der letzten zehn Jahre haben sich digitale Signalprozessoren als die bevorzugten Prozessoren zur Implementierung eingebetteter Anwendungen im Konsumgütermassenmarkt etabliert. Eingebettete Systeme werden zum Beispiel für einfache Steuerungsfunktionen in Haushaltsgeräten, wie etwa in Waschmaschinen oder Mikrowellenherden eingesetzt, aber auch für komplexe Aufgaben, wie Sendung und Empfang von Sprachsignalen in digitalen, portablen Funktelefonen (Handys) [Cas95]. Der Einsatz auf dem Massenmarkt ergibt einerseits einen Zwang zu niedrigen Preisen; eingebettete Systeme stellen andererseits meist aber hohe Leistungsanforderungen. Daher sind diese oft so konfiguriert, daß sie nur eine spezielle, anwendungsspezifische Menge von Aufgaben erfüllen können.

Die Preise von digitalen Signalprozessoren werden niedrig gehalten durch Minimierung der Chipfläche, geringen Stromverbrauch und weitestmöglichen Verzicht auf den Einsatz von externem Speicher. Um dennoch den Leistungsanforderungen gerecht zu werden, verwenden viele DSPs spezielle Hardware-Features wie Unterstützung von Schleifen mit geringem Overhead, akkumulatorbasierte Datenpfade, Instruktionen zur parallelen Ausführung unabhängiger Operationen und mehrere Speicherbänke, die parallele Datenzugriffe erlauben [Lee88]. Um dieses Leistungspotential auszuschöpfen, ist eine effektive Nutzung solcher speziellen Merkmale erforderlich. In der Vergangenheit konnte dies am besten durch Programmierung in Assembler erreicht werden. Mit wachsender Funktionalität und immer höherentwickelten Algorithmen wird es zunehmend wünschenswert, zur Programmierung eine weitverbreitetete, standardisierte und portable Hochsprache zu verwenden. Allerdings sind existierende Hochsprachen-Compiler häufig nicht in der Lage, Code zu generieren, der die Leistungsanforderungen eingebetteter Anwendungen erfüllt [ZSMS95].

2.2 Übersicht über den ADSP-2106x

Der ADSP-2106x SHARC (*Super Harvard Architecture Computer*) ist ein 32 Bit breiter digitaler Signalprozessor (DSP) für multimediale Anwendungen, wie Sprach-, Ton- und Bildverarbeitung. Seine Instruktionszykluszeit beträgt 25 ns, die Taktfrequenz somit 40 MHz, und er erreicht eine Instruktionsrate von 40 MIPS.

Der ADSP-2106x wurde 1995 als Weiterentwicklung des ADSP-21000 auf dem Markt eingeführt. Im Vergleich zu seinem Vorgängermodell liegen die Erweiterungen insbesondere in einem auf dem Chip integrierten dual-ported SRAM, sowie in der Unterstützung integrierter I/O-Peripheriegeräte durch einen speziellen I/O-Bus. Ein Überblick über die Architektur kann aus Abbildung 2.1 gewonnen werden, die ein detailliertes Blockdiagramm des ADSP-2106x zeigt.

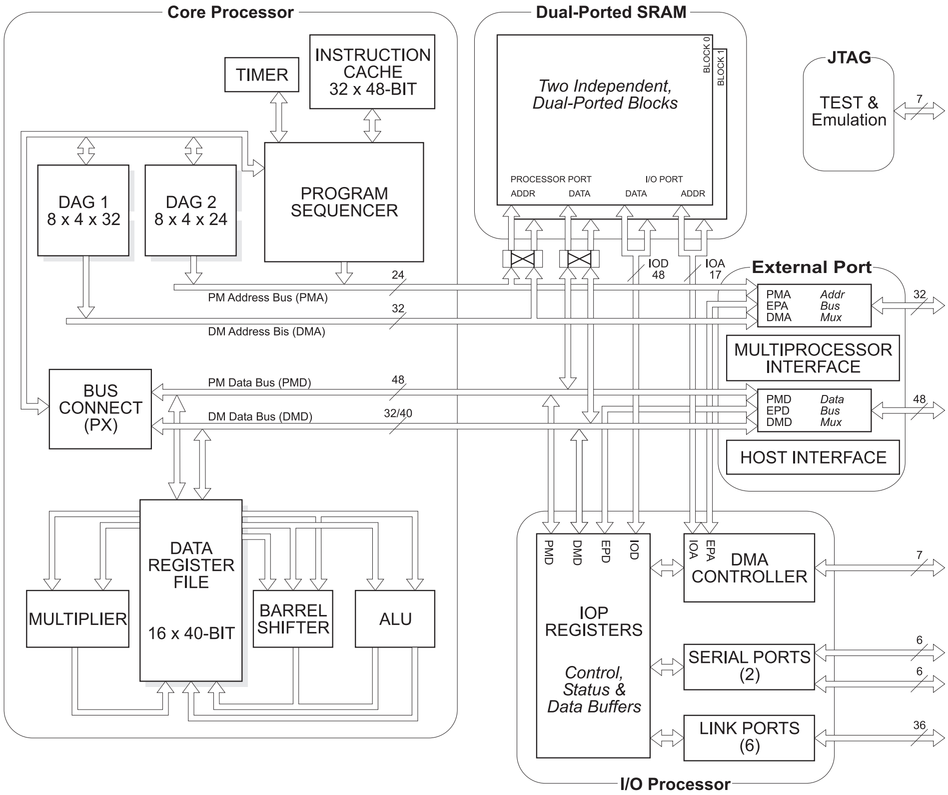


Abbildung 2.1: Blockschaltbild des ADSP-2106x.

Der Kernprozessor (*core processor*) des ADSP-2106x besteht aus dem Datenregisterfile, drei Berechnungseinheiten, einer Kontrolleinheit, zwei Adreßgeneratoren, dem Zeitgeber (*Timer*) und dem Instruktionscache. Diese Komponenten werden in den nachfolgenden Abschnitten näher beschrieben. Auf dem Chip liegen drei Busse: der PM-Bus (*Program Memory*), der DM-Bus (*Data Memory*) und der I/O-Bus. Der PM-Bus wird zum Zugriff auf Instruktionen und auf Daten verwendet. Der Prozessor kann in einem Zyklus auf zwei Datenoperanden zugreifen, einen über den DM-Bus und einen über den PM-Bus, eine Instruktion laden (aus dem Cache) und einen DMA-Zugriff ausführen.

In den einheitlichen Adreßraum des ADSP-2106x ist ein vier Gigabyte umfassender externer Adreßraum integriert. Die auf dem Chip liegenden Busse werden am externen Port gemultiplext, um einen externen Systembus mit einem 32-Bit Adreßbus und einem 48-Bit Datenbus zu bilden. Während der interne Speicher in zwei Blöcke aufgeteilt ist, ist der externe Speicherbereich in vier Bänke unterteilt.

Die Hostschnittstelle erlaubt einfache Verbindungen zu 16 und 32 Bit breiten Standart-Mikroprozessor-Bussen; Zugriffe erfolgen über den externen Port und werden auf den vereinigten Adreßraum abgebildet. Dazu stehen vier DMA-Kanäle zur Verfügung. Um eine effiziente Ausführung von Hostkommandos zu ermöglichen, werden Vektorinterrupts[1] unterstützt.

Alle Instruktionen haben eine feste Länge von 48 Bit. Neben Addition, Subtraktion und Multiplikation sind noch weitere arithmetische Operationen festverdrahtet implementiert. Hierzu zählen z.Bsp. kombinierte Addition und Subtraktion, kombinierte Multiplikation und Addition/Subtraktion, Minimum- und Maximumbildung, Shifts und Rotationen.

In einem Zyklus können gleichzeitig mehrere Aktionen ausgeführt werden:

- Übertragung zweier Operanden zwischen Speicher und Registerfile
- Übergabe zweier Operanden an die ALU
- Übergabe zweier Operanden an den Multiplizierer
- Empfang zweier Resultate von ALU und Multiplizierer

Datenübertragungen und arithmetische Operationen können somit in derselben Instruktion parallel ausgeführt werden.

Als Zahlendarstellungen werden das 32-Bit IEEE Gleitkomma-Format, eine 32 Bit breite ganzzahlige Zahldarstellung (2er-Komplement, sowie vorzeichenlose Darstellung) und

[1] Vektorinterrupts werden in Multiprozessorsystemen für Anweisungen zwischen verschiedenen Prozessoren verwendet (vgl. [Ana95c]).

ein 40 Bit breites erweitertes IEEE Gleitkomma-Format (32 Bit Mantisse, 8 Bit Exponent) verwendet.

Der ADSP-2106x ist eine Load/Store-Architektur. Zulässige Adressierungsmodi sind direktes und indirektes Adressieren; dabei sind sowohl *pre-* als auch *post increment* des Adreßregisters möglich. Zudem wird die Adressierung innerhalb zirkulärer Datenpuffer auf Hardwareebene unterstützt. Zirkuläre Puffer werden häufig bei der digitalen Signalverarbeitung eingesetzt, z.Bsp. in digitalen Filtern und Fouriertransformationen.

Die Ausführung von Verzweigungen kann sowohl verzögert (*delayed*) als auch unverzögert (*non-delayed*) erfolgen. Zählerbasierte Schleifen können ohne zusätzliche Instruktionen zum Verzweigen, zum Bedingungstest oder zur Zähleraktualisierung realisiert werden (vgl. Kapitel 2.10.3).

In den folgenden Abschnitten werden die im Rahmen dieser Arbeit wichtigen Hardwarecharakteristika detaillierter beschrieben.

2.3 Die Berechnungseinheiten

Es gibt drei unabhängige Berechnungseinheiten: eine ALU, einen Multiplizierer mit Festkomma-Akkumulator und einen Shifter. Die ALU führt arithmetische und logische Ope-

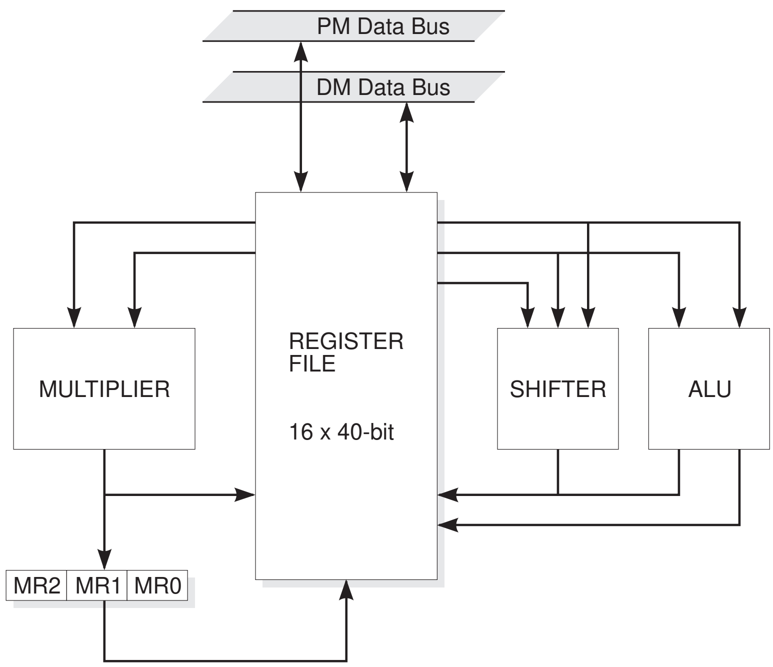


Abbildung 2.2: Die Berechnungseinheiten.

rationen sowohl für Fest- als auch für Gleitkommazahlen in den oben genannten Formaten aus. Der Multiplizierer berechnet Gleitkomma- und Festkommamultiplikationen ebenso wie kombinierte Multiplikationen und Additionen/Subtraktionen für Festkommazahlen. Der Shifter schließlich führt logische und arithmetische Shifts, sowie spezielle Bitmanipulationen auf 32-Bit Operanden aus. Alle Instruktionen werden innerhalb eines Zyklus ausgeführt. ALU, Multiplizierer und Shifter sind parallel miteinander verbunden, so daß die Ausgabe einer beliebigen Einheit als Eingabe jeder beliebigen Berechnungseinheit im nächsten Zyklus verwendet werden kann. In einer Multifunktionsinstruktion führen ALU und Multiplizierer gleichzeitig unabhängige Instruktionen aus. Am Ende jeder Operation werden verschiedene Bits in Kontroll- und Statusregistern sowie weitere Flags gesetzt, die als Grundlage für nachfolgende bedingte Anweisungen dienen können.

2.3.1 Die arithmetisch-logische Einheit

Die ALU arbeitet auf einem oder zwei Operanden, die Datenregister des Registerfiles sein müssen. Üblicherweise wird ein Ergebnis zurückgeliefert, bei kombinierter Addition und Subtraktion jedoch zwei und bei Vergleichsoperationen gar keines. Statt dessen führen Vergleiche zur Aktualisierung bestimmter Statusbits. Eine Übersicht über alle Instruktionen der ALU und der mit ihnen verbundenen Aktualisierung von Statusbits ist in Anhang A.2.1 gegeben. Die von der ALU produzierten Ergebnisse werden ins Registerfile zurückgeschrieben. Eingabeoperanden werden innerhalb der ersten Zyklenhälfte aus dem Registerfile gelesen und Ergebnisse in der zweiten Hälfte dorthin zurückgeschrieben. Somit kann die ALU dasselbe Register innerhalb eines Zyklus auslesen und zurückschreiben. Dies gilt ebenfalls für Multiplizierer und Shifter.

2.3.2 Der Multiplizierer

Der Multiplizierer benötigt zwei Operanden, die Register des Registerfiles sein müssen. Festkommaoperationen können Ergebnisse in einem von zwei lokalen Ergebnisregistern (*Multiplier Result Register, MR-Register*) akkumulieren oder zurück ins Registerfile schreiben. In den MR-Registern gespeicherte Ergebnisse können mittels spezieller Operationen gerundet oder saturiert[2] werden. Die Ergebnisse von Gleitkommaoperationen müssen immer direkt ins Registerfile zurückgeschrieben werden. Jedes der beiden MR-Register hat eine Länge von 80 Bit; ihr Format ist identisch. Sie sind in drei Felder MR0, MR1 und MR2 unterteilt, wobei MR0 und MR1 32 Bit lang sind und MR2 16 Bit lang ist. Jedes dieser Felder kann gesondert aus dem Registerfile eingelesen oder dorthin zurückgeschrieben werden. Die beiden MR-Register werden als MRF (*Foreground*) und MRB (*Background*)

[2]Bei positivem Überlauf von Festkommazahlen wird die größte ganze Zahl (0x7FFF FFFF) zurückgeliefert; bei negativem Überlauf die größte negative Zahl (0x8000 0000).

bezeichnet; welches von beiden aktiv ist, wird von einem Bit eines Modusregisters bestimmt. Anders als bei anderen Registern, für die alternative Registermengen existieren, kann auf beide MR-Register zur selben Zeit zugegriffen werden. In Anhang A.2.2 sind die Multipliziererinstruktionen mit den zugehörigen Statusbitveränderungen aufgeführt.

2.3.3 Der Shifter

Der Shifter arbeitet mit 32-Bit Festkommaoperanden. Er kann einen, zwei oder drei Operanden haben: den X-Input, der stets ein 32-Bit Festkommawert sein muß und durch die Instruktion nicht verändert wird, den Y-Input, der Shiftlänge, Bitfeldlängen oder Bitpositionen beschreibt und den Z-Input, der ebenfalls ein 32-Bit Festkommawert sein muß, nach der Instruktion jedoch aktualisiert wird (z.B. in `Rz = Rz or lshift Rx by Ry`). Der Shifter schreibt ein Ergebnis in das Registerfile zurück. Eine Liste der Instruktionen des Shifters zusammen mit den Aktualisierungen der Statusbits ist in Anhang A.2.3 zu finden.

2.3.4 Multifunktionsinstruktionen

Multifunktionsinstruktionen kombinieren parallele Ausführung von Multiplizierer und ALU oder parallele Funktionen innerhalb der ALU. Eine Übersicht über alle Multifunktionsinstruktionen ist in Anhang A.3 gegeben. Beide Operationen werden auf dieselbe Weise ausgeführt, wie bei der exklusiven Ausführung. Flags und Statusbits werden ebenfalls in gleicher Weise verändert, jedoch mit der Einschränkung, daß bei der kombinierten Addition/Subtraktion in der ALU die Flags der beiden Operationen disjunktiv verknüpft werden. Jeder der vier Eingabeoperanden für Berechnungen, die sowohl ALU, als auch Multiplizierer verwenden, ist auf eine Menge von vier bestimmten Registern des Registerfiles beschränkt, wie in Abbildung 2.3 zu sehen ist.

2.4 Die Register des ADSP-2106x

Fast alle Register im Kernprozessor (*core processor*) sind Universalregister. Es stehen Instruktionen zur Verfügung, um Daten zwischen zwei beliebigen Universalregistern oder einem Universalregister und dem Speicher zu übertragen. Dies gilt für Kontroll- und Statusregister ebenso wie für die Datenregister des Registerfiles. Die Datenregister des Registerfiles, die Register der Adreßgeneratoren (siehe auch Kapitel 2.6), sowie die Akkumulatorregister des Multiplizierers (MR-Register) besitzen Alternativregister, deren Verwendung einen schnellen Kontextwechsel ermöglicht. Kontrollbits in einem Modus-Kontrollregister bestimmen zu jedem Zeitpunkt, welche Registermenge aktiv ist. Die beiden Alternativmen-

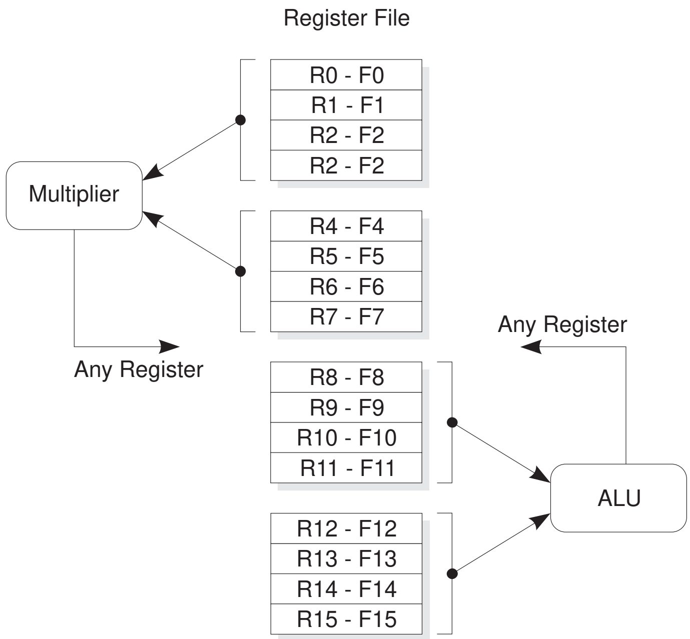


Abbildung 2.3: Eingaberegister bei Multifunktionsinstruktionen.

gen des 10 Port-Registerfiles bestehen aus zwei Mengen von jeweils 16 40-Bit Registern. Jede Hälfte des Registerfiles (R0 - R7 und R8 - R15) kann ihre alternative Registermenge unabhängig von der anderen Hälfte aktivieren. So können Daten zwischen Kontexten geteilt werden, indem die gemeinsamen Daten in eine Hälfte des Registerfiles geschrieben werden und die alternative Registermenge der anderen Hälfte aktiviert wird.

Es gibt keine speziellen Gleitkommaregister; vielmehr können die Datenregister des Registerfiles sowohl für Fest- als auch für Gleitkommaoperationen genutzt werden. Wird dasselbe Datenregister des Registerfiles innerhalb einer Instruktion sowohl als Quelle eines Operanden als auch als Ziel eines Ergebnisses oder Memory-Fetches spezifiziert, so wird es in der ersten Hälfte des Zyklus ausgelesen und in der zweiten Hälfte beschrieben. Somit wird der alte Inhalt als Eingabe verwendet, bevor das Register mit den neuen Daten aktualisiert wird (siehe auch Kapitel 2.3.1).

Finden im selben Zyklus zwei Schreibzugriffe auf dasselbe Register statt, so wird nur ein Schreibvorgang tatsächlich ausgeführt. Dies wird von einer Priorität gesteuert, die von der Quelle der Daten, die geschrieben werden sollen, abhängt. Im folgenden werden nun die möglichen Quellen nach absteigender Priorität aufgelistet:

- Datenspeicher (*Data Memory*) oder Universalregister
- Programmspeicher (*Program Memory*)
- ALU

- Multiplizierer
- Shifter

Neben dem Registerfile und den Kontroll- und Statusregistern gibt es noch vier weitere Typen von Registern, die den Datenadreßgeneratoren zugeordnet sind: Indexregister (I), Modify-Register (M), Basisregister (B) und Längenregister (L). Die Adreßgeneratoren (DAG1 und DAG2) enthalten acht Register von jedem dieser Typen, die bei DAG1 32 Bit und bei DAG2 24 Bit breit sind. Ein Indexregister fungiert als Zeiger auf den Speicher und ein M-Register enthält den Wert, um den dieser Zeiger inkrementiert oder dekrementiert werden soll. B- und L-Register werden nur für zirkuläre Datenpuffer verwendet, wobei das B-Register die Basisadresse des Puffers enthält. Das korrespondierende L-Register beinhaltet dessen Länge, also die Anzahl der im Puffer enthaltenen Speicherstellen. Hierdurch wird die Adressierung innerhalb eines zirkulären Datenpuffers unterstützt.

2.5 Interner Speicher

Der ADSP-21060 enthält vier MBit SRAM, bestehend aus zwei 2 MBit-Blöcken; der ADSP-21062 enthält zwei MBit SRAM, bestehend aus zwei 1 MBit-Blöcken. Jeder Block kann für verschiedene Kombinationen von Code- und Datenspeicherung konfiguriert werden. Die Blöcke besitzen jeweils zwei Ports, um unabhängige Zugriffe vom Kernprozessor einerseits und I/O-Prozessor oder DMA-Controller andererseits innerhalb eines Zyklus zu ermöglichen. In jedem Speicherblock können Kombinationen von Code und Daten gespeichert werden. Die Zugriffe sind am effizientesten, wenn ein Block Daten speichert und den DM-Bus für die Übertragung verwendet, während der andere Block Instruktionen und Daten speichert und den PM-Bus zur Übertragung gebraucht.

Auf den Speicher können Zugriffe auf 32-Bit Wörter für Gleitkommazahlen mit einfacher Genauigkeit erfolgen, auf 48-Bit Wörter, die entweder Instruktionen oder 40-Bit Gleitkommazahlen mit erweiterter Genauigkeit enthalten und auf 16-Bit Halbwörter, die vor allem für ganzzahlige Datenwerte eingesetzt werden.

2.6 Kontrolleinheit und Adreßgeneratoren

Die Kontrolleinheit (*program sequencer*) liefert Instruktionsadressen an den Programmspeicher (*program memory*), kontrolliert Schleifeniterationen und berechnet bedingte Ausdrücke. Mit Hilfe eines internen Schleifenzählers und eines Schleifenkellers können Schleifen ohne explizite Sprunginstruktionen oder Instruktionen zum Dekrementieren des Zählers sowie zum Testen ausgeführt werden. Es gibt eine Instruktionspipeline, die aus *fetch-*, *decode-* und *execute*-Zyklen besteht (vgl. Kapitel 2.8).

Durch die beiden Adreßgeneratoren DAG1 und DAG2 werden gleichzeitige Lese- und Schreibzugriffe auf zwei Operanden ermöglicht. DAG1 liefert 32-Bit Adressen für den Datenspeicher, DAG2 liefert 24-Bit Adressen für den Programmspeicher. Zudem implementieren die Adreßgeneratoren zirkuläre Datenpuffer auf Hardwareebene.

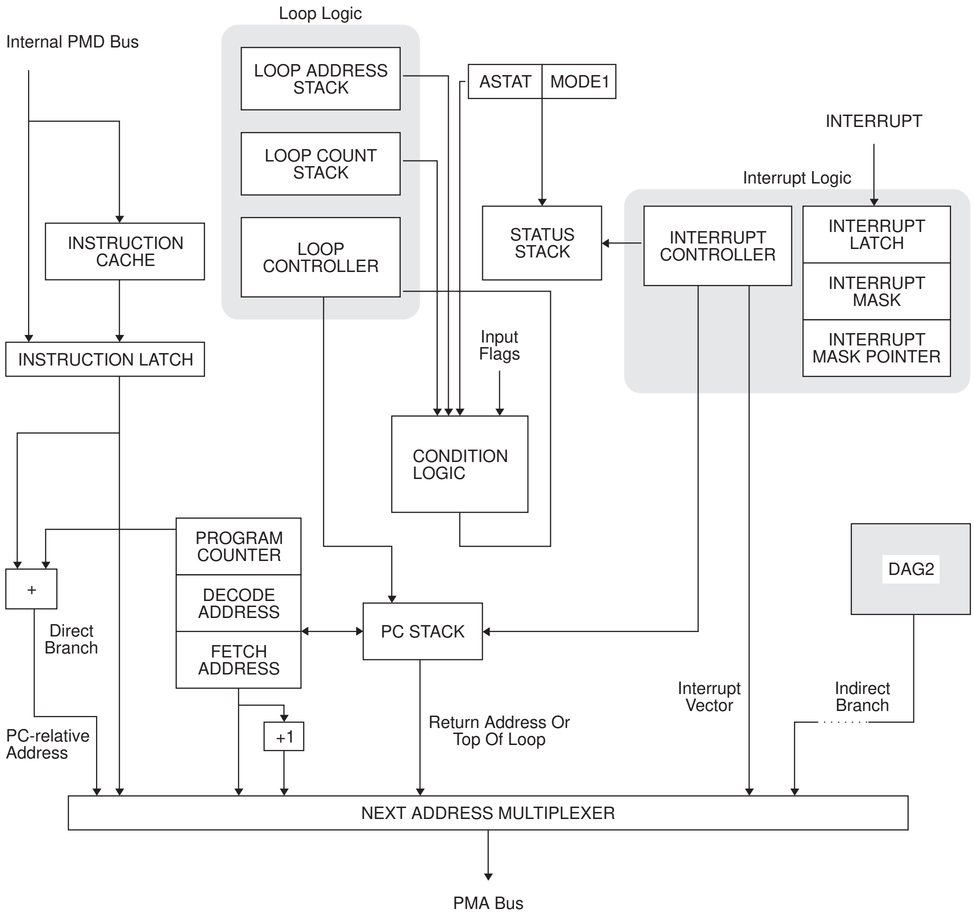


Abbildung 2.4: Die Kontrolleinheit des ADSP-2106x.

Detailliertere Angaben zur Kontrolleinheit können Abbildung 2.4 entnommen werden. Das *fetch address*-Register, das *decode address*-Register und der PC (*Program Counter*) enthalten die Adressen der Instruktionen, die gerade abgerufen, decodiert bzw. ausgeführt werden. Der PC ist mit dem PC-Keller (*PC-Stack*) gekoppelt, der 30 Einträge der Länge 24 Bit aufnehmen kann und verwendet wird, um Rücksprungadressen und Schleifeneinsprungadressen zu speichern. Der Interrupt-Controller führt alle Funktionen aus, die mit der Interrupt-Verarbeitung verbunden sind.

Die Kontrolleinheit wertet bedingte Anweisungen und Schleifenabbruchbedingungen aus, indem sie die Informationen aus den Statusregistern verwendet. Schleifenadreßkeller (*Loop-address Stack*) und Schleifenzählerkeller (*Loop-counter Stack*) unterstützen geschachtelte Schleifen. Beide sind sechs Ebenen tief, wobei auf jeder Ebene ein 32-Bit Wort gespeichert werden kann, das aus einer 24-Bit Schleifenendadresse, einem 5-Bit Terminierungscode und einer zwei Bit langen Codierung des Schleifentyps besteht; ein Bit wird nicht verwendet. Der Statuskeller (*Status Stack*) speichert den Inhalt von Statusregistern zur Implementierung geschachtelter Interrupt-Routinen. Alle Register der Kontrolleinheit sind Universalregister, so daß Datenübertragungen zwischen ihnen und dem Datenspeicher oder anderen Universalregistern möglich sind.

Die Register ebenso wie die obersten Kellereinträge sind lesbar, alle Register außer dem *fetch address*-Register, dem *decode address*-Register und dem PC sind auch beschreibbar. Aufgrund des Befehlsfließbandes werden Schreibzugriffe zu einigen dieser Register nicht direkt im nächsten Zyklus wirksam. Ebenso werden manche Register nicht sofort in dem auf einen Schreibzugriff folgenden Zyklus aktualisiert; das Auslesen eines solchen Registers liefert erst nach einem zusätzlichen Zyklus den neuen Wert.

2.7 Instruktionscache

Die Kontrolleinheit enthält einen Instruktionscache, der 2-Wege Set-assoziativ ist und 32 Instruktionen aufnehmen kann; er verwendet eine LRU-Strategie. Der Cache ist selektiv: Nur Instruktionen, deren Abrufen (*Fetch*) mit dem Laden von Daten aus dem Programmspeicher (PM) in Konflikt stehen, werden in den Cache geladen.

Erfordert die Instruktion an Adresse n einen Datenzugriff aus dem PM, so gibt es unter der Voraussetzung linearen Programmflusses (vgl. Abbildung 2.5) einen Konflikt mit dem Abrufen (*Fetch*) der Instruktion an Adresse $n + 2$. Im Instruktionscache wird dann diese Instruktion gespeichert, nicht die Instruktion, die den Datenzugriff aus dem PM benötigt. Ist eine benötigte Adresse im Cache, liefert der Cache die Instruktion, während der PM-Datenzugriff ausgeführt wird. Ist die benötigte Instruktion nicht im Cache, so findet das Abrufen der Instruktion aus dem Speicher in dem Zyklus statt, der auf den PM-Datenzugriff folgt, so daß ein Zyklus "verschwendet" wird. Diese Instruktion wird in den Cache geladen, wenn der Cache aktiviert ist; bei der nächsten Ausführung der Instruktion, die den PM-Datenzugriff benötigt, ist sie dann verfügbar.

2.8 Instruktionszyklus

Der ADSP-2106x führt Instruktionen innerhalb von drei Taktzyklen aus:

- Im *Fetch*-Zyklus wird die Instruktion aus dem Programmspeicher PM oder dem Instruktionscache abgerufen.
- Im *Decode*-Zyklus wird die Instruktion decodiert.
- Im *Execute*-Zyklus wird die Instruktion ausgeführt, das heißt, die im Rahmen dieser Instruktion auszuführenden Operationen werden vollendet.

Die Zyklen sind gepipelined. Während eine Instruktion abgerufen wird, wird bei linearem Programmfluß die Instruktion, die im vorigen Zyklus geholt wurde, decodiert und die, die vor zwei Zyklen geholt wurde, wird ausgeführt. Somit beträgt der Durchsatz eine Instruktion pro Zyklus; jeder nicht-lineare Programmfluß kann den Instruktionsdurchsatz jedoch reduzieren. Nicht-lineare Operationen sind:

- Datenzugriffe aus dem PM, die mit dem Abrufen (*Fetch*) von Instruktionen kollidieren
- Sprünge
- Unterprogrammaufrufe und -rücksprünge
- Interrupts und Interruptrücksprünge
- Schleifen

2.9 Befehlsvorrat

Die Befehle des ADSP-2106x können in vier Kategorien aufgeteilt werden:

- ***Compute and Move or Modify*-Befehle**. Dies sind Instruktionen, die eine Berechnung spezifizieren, die parallel zu einem oder zwei Datentransportbefehlen oder der Veränderung eines Indexregisters erfolgen.
- **Kontrollflußinstruktionen**, die bestimmte Typen von Verzweigungen, Unterprogrammaufrufe, Rücksprünge und Schleifen spezifizieren. Einige dieser Instruktionen können auch eine Berechnung und/oder einen Datentransport bezeichnen.
- **Unmittelbare Datentransportinstruktionen** (*immediate data move*), die *immediate*-Felder der Instruktion als Operanden zum Adressieren nehmen
- **gemischte Instruktionen**, wie zum Beispiel `nop` und `idle`.

Es gibt 23 Instruktionsformate, von denen einige mehr als eine syntaktische Form besitzen; viele Instruktionen können bedingt sein. Im Anhang A.1 findet sich eine Auflistung aller Instruktionsformate. Bemerkenswert ist hierbei, daß man innerhalb des Instruktionssatzes gewisse Mikrooperationen[3] unterscheiden kann. Diese können eine eigenständige Instruktion darstellen, aber auch mit anderen Mikrooperationen zu einer größeren Instruktion kombiniert werden.

Beispiel 2.1 Die Instruktionen des folgenden Codeausschnittes können nacheinander ausgeführt werden:

```
1) r0   =  dm(i4, m4);
2) r1   =  pm(i12, m12);
3) r2   =  r3 * r5;
4) r16  =  r9 + r13;
```

Dabei wird in Instruktion 1 das Register `r0` mit dem Inhalt der Speicherzelle beschrieben, die durch das Register `i4` adressiert wird. Nach dem Speicherzugriff wird `i4` um den in `m4` enthaltenen Wert inkrementiert. In Instruktion 2 wird in ähnlicher Weise ein Datenzugriff auf den Programmspeicher PM ausgeführt. Die Instruktionen 3 und 4 berechnen eine Multiplikation bzw. eine Addition, deren Operandenregister zu einem früheren Zeitpunkt geladen wurden. All diese Instruktionen können auch zu einer Multifunktionsinstruktion zusammengefaßt und gemeinsam innerhalb eines Zyklus ausgeführt werden.

```
r0 = dm(i4, m4), r1 = pm(i12, m12), r2 = r3 * r5, r16 = r9 + r13;
```

□

In der Assemblerdatei ist dies dadurch gekennzeichnet, daß zu einer Instruktion zusammengefaßte Mikrooperationen durch Kommata getrennt werden, wohingegen Instruktionen, die in verschiedenen Zyklen ausgeführt werden, mittels Strichpunkten abgegrenzt sind. Desweiteren wird im folgenden die Konvention eingeführt, daß alle zu einer Instruktion zusammengefaßten Mikrooperationen in einer Zeile des Maschinenprogrammes aufgeführt werden. Instruktionen, die verschiedenen Zyklen zugeordnet sind, stehen in verschiedenen Zeilen des Assemblerprogrammes. Somit repräsentiert jede Zeile des Maschinenprogrammes eine gegebenenfalls aus verschiedenen Mikrooperationen zusammengesetze Instruktion, die innerhalb eines Taktzyklus ausgeführt wird.

[3]Dieser Begriff darf jedoch nicht mit den Mikrooperationen verwechselt werden, die im Zusammenhang mikroprogrammierter Kontrolleinheiten verwendet werden; beim ADSP-2106x ist die Kontrollogik fest verdrahtet.

2.10 Steuerung des Kontrollflusses

Eine Übersicht über alle möglichen Verläufe des Kontrollflusses ist in Abbildung 2.5 gegeben. In den nachfolgenden Abschnitten werden diese Verläufe detaillierter beschrieben.

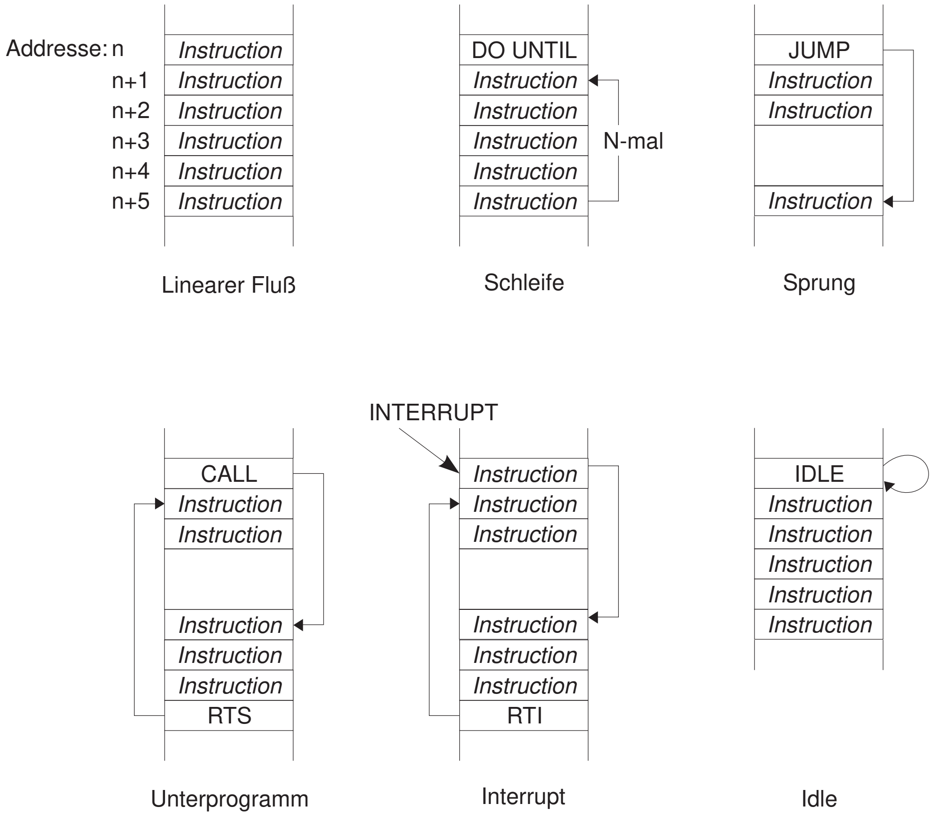


Abbildung 2.5: Mögliche Kontrollflußverläufe.

2.10.1 Ausführung von bedingten Anweisungen

Die Kontrolleinheit wertet Bedingungen aus, um festzustellen, ob eine bedingte Anweisung auszuführen ist und wann eine Schleife beendet werden soll. Die Bedingungen basieren auf Informationen aus dem arithmetischen Statusregister (`ASTAT`), dem Modus-Kontrollregister 1 (`MODE1`), weiteren Flags und dem Schleifenzähler.

Die meisten Bedingungen können von der Kontrolleinheit auf wahr und falsch überprüft werden, also auf gleich Null und ungleich Null; insgesamt gibt es 32 Bedingungs- und Terminierungscodes (vgl. Anhang A.4). So wird zum Beispiel die `lce`-Bedingung (*loop* *counter expired*) häufig in einer `do until`-Instruktion verwendet. Da die `lce`-Bedingung den Wert des Schleifenzählers testet (`curlcntr`), sollte eine `not lce`-Bedingung nicht auf ein Beschreiben des Schleifenzählers erfolgen; sonst wird nämlich der alte Wert des Zählers verwendet.

2.10.2 Sprunganweisungen

Es gibt vier Arten von Sprunganweisungen:

- Prozeduraufruf (*call*)
- Sprung (*jump*)
- Prozedurrücksprung (*rts* – *return from subroutine*)
- Interruptrücksprung (*rti* – *return from interrupt*)

Sowohl Sprünge als auch Prozeduraufrufe führen zu Abweichungen vom linearen Programmfluß. Ein wichtiger Unterschied zwischen Sprüngen und Prozeduraufrufen liegt darin, daß ein Prozeduraufruf im Gegensatz zu Sprüngen eine Rücksprungadresse auf den PC-Stack legt. Ein Interruptrücksprung veranlaßt den Prozessor, zu der Adresse im obersten Eintrag des PC-Kellers zu springen. Die Art der Sprunganweisung kann durch mehrere Parameter spezifiziert sein:

- Sprünge, Prozeduraufrufe und Rücksprünge können bedingt sein.
- Sprünge und Prozeduraufrufe können direkt, indirekt oder relativ zum PC erfolgen. Direkte Sprunganweisungen springen zu der (24-Bit-)Adresse, die im *immediate*-Feld der Instruktion angegeben ist. Ein indirekter Sprung führt zu einer Adresse, die vom Adreßgenerator DAG2 geliefert wird. PC-relative Sprünge addieren eine in der Instruktion spezifizierte *immediate*-Konstante zum aktuellen Wert des PC, um die Adresse des Sprungziels zu berechnen.
- Alle Sprunganweisungen können verzögert (*delayed*) oder unverzögert (*non-delayed*) ausgeführt werden. In einem verzögerten Sprung werden die zwei Instruktionen, die unmittelbar auf die Sprunganweisung folgen, ausgeführt. Bei einem unverzögerten Sprung unterdrückt die Kontrolleinheit die Ausführung dieser beiden Instruktionen; statt dessen werden `nop`'s ausgeführt. Soll ein Sprung verzögert ausgeführt werden, wird der eigentlichen Instruktion ein Modifikator `(DB)` nachgestellt. Zur Veranschaulichung sind die Verläufe verzögerter und unverzögerter Sprünge sowie Prozeduraufrufe in den Abbildungen 2.6 und 2.7 dargestellt.

Unverzögerte Sprünge oder Unterprogrammaufrufe

Zyklen $\longrightarrow$

Execute	n	nop	nop	j
Decode	n+1 → nop	n+2 → nop	j	j+1
Fetch	n+2	j	j+1	j+2
	n+1 unterdrückt	n+2 unterdrückt; für UP-Aufruf: n+1 auf PC-Stack abgelegt		

Unverzögerter Rücksprung

Zyklen $\longrightarrow$

Execute	n	nop	nop	r
Decode	n+1 → nop	n+2 → nop	r	r+1
Fetch	n+2	r	r+1	r+2
	n+1 unterdrückt	n+2 unterdrückt; r vom PC-Stack entfernt		

n = Verzweigungsinstruktion
j = Instruktion an Sprung- oder Unterprogrammaufruf-Adresse
r = Instruktion an Rücksprungadresse

Abbildung 2.6: Unverzögerte Sprunganweisungen.

Verzögerte Sprünge oder Unterprogrammaufrufe

Zyklen $\longrightarrow$

Execute	n	n+1	n+2	j
Decode	n+1	n+2	j	j+1
Fetch	n+2	j	j+1	j+2

für UP-Aufruf:
n+3 auf
PC-Stack
abgelegt

Verzögerter Rücksprung

Zyklen $\longrightarrow$

Execute	n	n+1	n+2	r
Decode	n+1	n+2	r	r+1
Fetch	n+2	r	r+1	r+2

r vom PC-Stack
entfernt

n = Verzweigungsinstruktion
j = Instruktion an Sprung- oder Unterprogrammaufruf-Adresse
r = Instruktion an Rücksprungadresse

Abbildung 2.7: Verzögerte Sprunganweisungen.

An den beiden unmittelbar auf eine Verzweigung folgenden Stellen darf keine der folgenden Instruktionen stehen:

- andere Sprünge, Prozeduraufrufe oder Rücksprünge
- *push*- und *pop*-Operationen auf dem PC-Keller
- Schreibzugriffe auf den PC-Keller oder den PC-Kellerzeiger
- `do until`-Instruktion
- `idle`-Instruktion

2.10.3 Schleifen (`do until`)

Die `do until`-Instruktion ermöglicht die effiziente Ausführung von Schleifen, ohne daß zusätzliche Instruktionen zum Verzweigen, Testen von Bedingungen oder Dekrementieren von Zählern erforderlich sind. Wird eine `do until`-Instruktion ausgeführt, legt die Kontrolleinheit die Adresse der letzten Schleifeninstruktion und die Schleifenabbruchbedingung auf den Schleifenadreßkeller (*loop address stack*). Die Adresse der ersten Instruktion der Schleife, die also direkt auf `do until` folgt, wird auf den PC-Keller gelegt.

Wegen des dreistufigen Befehlsfließbandes (*fetch, decode, execute*) muß die Terminierungsbedingung bereits vor dem Schleifenende ausgewertet werden; sie wird getestet, wenn die Instruktion, die zwei Stellen vor der letzten Instruktion der Schleife steht, ausgeführt wird (also an Stelle $e-2$, wobei e die Adresse der letzten Schleifeninstruktion ist). Bei der Verarbeitung von Schleifen sind mehrere Einschränkungen zu beachten:

- Geschachtelte Schleifen können nicht mit derselben Instruktion beendet werden.
- Die drei letzten Instruktionen einer Schleife dürfen keine Verzweigungsinstruktionen sein.
- In der drittletzten Instruktion einer zählerbasierten Schleife darf kein Schreibzugriff auf den Zähler ausgeführt werden.
- Abhängig von der Schleifenlänge liefert die Kontrolleinheit eine Sonderbehandlung, um Leerlaufzeiten (`nop`'s) zu vermeiden.
- Bei geschachtelten Schleifen muß die letzte Instruktion der äußeren Schleife mindestens zwei Adressen nach der letzten Instruktion der inneren Schleife stehen.

Wegen der Instruktionspipeline müssen bei der Beendung von nicht-zählerbasierten Schleifen drei Sonderfälle betrachtet werden:

- In einer aus drei Instruktionen bestehenden Schleife wird die Bedingung getestet, wenn die erste Instruktion ausgeführt wird. Ist die Abbruchbedingung wahr, erfolgt dennoch ein kompletter Durchlauf.

- In einer aus zwei Instruktionen bestehenden Schleife wird die Bedingung während der Ausführung der letzten Instruktion getestet. Wird die Bedingung während der ersten Instruktion wahr, erfolgt noch ein Durchlauf, wird sie während der zweiten erfüllt, so erfolgen noch zwei Durchläufe.

- In einer Schleife, die aus nur einer Instruktion besteht, wird die Terminierungsbedingung in jedem Zyklus getestet. Wird die Bedingung wahr, so wird die Schleife noch dreimal ausgeführt.

Kapitel 3

Grundlagen der Codeerzeugung

Ein Code-Erzeuger erhält als Eingabe eine Zwischendarstellung des Quellprogrammes, als Ausgabe erzeugt er ein äquivalentes Zielprogramm. Ziel der Codeerzeugung ist es, eine Folge von Maschineninstruktionen zu generieren, die unter Wahrung der Semantik des Eingabeprogrammes das Leistungspotential der zugrundeliegenden Maschine möglichst gut ausschöpft. Eine optimale Lösung kann in aller Regel nicht ermittelt werden, denn die Komplexität des Problems steht dem Erreichen der Optimalität im Wege: Die Codeerzeugung umfaßt drei Teilprobleme, nämlich Codeselektion, Registerallokation und Instruktionsanordnung; Registerallokation und Instruktionsanordnung stellen $\mathcal{NP}$-schwere Probleme dar.

1. Ziel der Codeselektion ist die Auswahl möglichst "guter" Befehlsfolgen für die Ausdrücke und Anweisungen des Quellprogrammes. Die meisten Rechner bieten mehrere Befehlsfolgen als Übersetzung einer Quellsprachenanweisung an. Es gilt, eine bezüglich Ausführungszeit, Speicherplatzbedarf und/oder Befehlsfolgenlänge möglichst gute Befehlsfolge auszuwählen. Bei RISC-Prozessoren ist dieses Problem jedoch aufgrund der einfacheren Befehlssatzstruktur entschärft [WM97, Bra91].

2. Die Registerallokation hat zum Ziel, die Werte von Variablen und Ausdrücken der Zwischendarstellung auf physikalische Register abzubilden, um die Anzahl der Speicherzugriffe während der Programmausführung zu minimieren. Sie besteht selbst wiederum aus zwei Teilaufgaben:

   - Zunächst muß festgelegt werden, welche Werte in den Registern der Maschine geladen werden sollen. Im allgemeinen übersteigt die Anzahl der gleichzeitig lebenden Werte die Anzahl der Register. Um den Datentransfer von und zum Speicher zu minimieren, muß der Allokator Entscheidungen darüber treffen, welche Werte in Registern gehalten werden sollen. Im Kontext verteilter Registermengen, wie sie auch beim ADSP-2106x gegeben sind, ist es eine Aufgabe steigender Wichtigkeit, Werte auf bestimmte Registermengen abzubilden.
   - Nach der Allokation müssen die physikalischen Register bestimmt werden, in denen die Werte liegen sollen. Das ist Aufgabe der Registerzuteilung ([Bas95]).

3. Unter Instruktionsanordnung schließlich versteht man die Umordnung einer Instruktionssequenz, um effektiven Gebrauch von den Parallelverarbeitungskapazitäten der Maschine und dem Befehlsfließband zu machen. Dies bezieht auch das Kombinieren parallel ausführbarer Operationen zu einer Maschineninstruktion bei VLIW-artigen Architekturen ein.

Diese Teilprobleme, oder Phasen, sind nicht unabhängig voneinander, sondern beeinflussen sich gegenseitig. Es kann vorkommen, daß es für ein Quellprogramm p zwei verschiedene korrekte Übersetzungen p_1 und p_2 gibt, wobei p_1 nach dem verwendeten Kostenmaß billiger als p_2 ist. Wurde die Codeselektion ohne Berücksichtigung von Anforderungen der Registerallokation durchgeführt, so wird als Ausgabe der Codeselektion das Zielprogramm p_1 generiert. Wurden jedoch innerhalb von p_1 mehr virtuelle Register eingesetzt, als physikalische Register zur Verfügung stehen, müssen jetzt Zwischenspeicherungen und Ladeoperationen eingeführt werden, die das resultierende Programm eventuell teurer als p_2 machen (vgl. Kapitel 3.3). Umgekehrt kann der Codeselektor nach einer vorherigen Registerallokation eventuell eine billigste Befehlssequenz gar nicht auswählen, weil ihm dazu die Register fehlen [WM97].

Bei RISC-Prozessoren ist aufgrund des einfacheren Befehlssatzes die Codeselektion einfacher zu lösen, als bei CISCs, so daß die Interaktion zwischen Registerallokation und Codeselektion an Bedeutung verliert. Statt dessen tritt die Interaktion zwischen Registerallokation und Instruktionsanordnung in den Vordergrund, da der Scheduler Register benötigt, um die Ausführung unabhängiger Operationen zu überlappen. Im Rahmen der Instruktionsanordnung werden Register verwendet, um Parallelismus auf Instruktionsebene auszunutzen; im Kontext der Registerallokation werden sie benötigt, um Speicherzugriffe zu reduzieren. Da die Ziele dieser beiden Phasen oft in Konflikt miteinander stehen, erlegt die zuerst ausgeführte Teilaufgabe der zweiten Beschränkungen auf, die zu ineffizientem Code führen können. Man spricht hier vom Phasenanordnungsproblem. Wird zum Beispiel Registerallokation zuerst ausgeführt, kann sie das Umordnungspotential des Schedulers beschneiden, indem dasselbe physikalische Register unabhängigen Zwischenergebnissen zugewiesen wird. Dies verhindert eine Überlappung der entsprechenden Operationen durch den Scheduler. Geht die Instruktionsanordnung der Registerallokation voran, kann die Anzahl der gleichzeitig lebenden Werte so stark erhöht werden, daß viele dieser Werte im Hauptspeicher zwischengespeichert werden müssen.

3.1 Programmdarstellungen

Als Ausgangsbasis werden in den oben genannten Teilaufgaben verschiedene Programmdarstellungen verwendet, die im folgenden vorgestellt werden. Nachfolgend wird dabei im wesentlichen die Terminologie verwendet, wie sie in [WM97] bzw. [Bas95] definiert ist.

Der Kontrollflußgraph einer Prozedur gibt an, welche Instruktionen möglicherweise nacheinander ausgeführt werden. Ob das bei der Ausführung des Programmes tatsächlich geschieht, hängt eventuell von Bedingungen ab, die zur Übersetzungszeit im allgemeinen nicht ausgewertet werden können.

Definition 3.1 (Kontrollflußgraph) *Der **Kontrollflußgraph** einer Prozedur ist ein knotenmarkierter, kantengeordneter, gerichteter Graph $G_{cf} = (N, E_{cf}, n_{start}, n_{stop})$. Zu jeder primitiven Anweisung p der Prozedur gibt es einen Knoten $n_p \in N$, der mit dieser Anweisung markiert ist. Für zusammengesetzte Anweisungen ergeben sich die in Abbildung 3.1 dargestellten Teilgraphen. Kanten, die zu expliziten Sprüngen gehören, führen vom Knoten des Sprunges zu dem Sprungziel. n_{start} sei der eindeutig bestimmte Eintritts-*

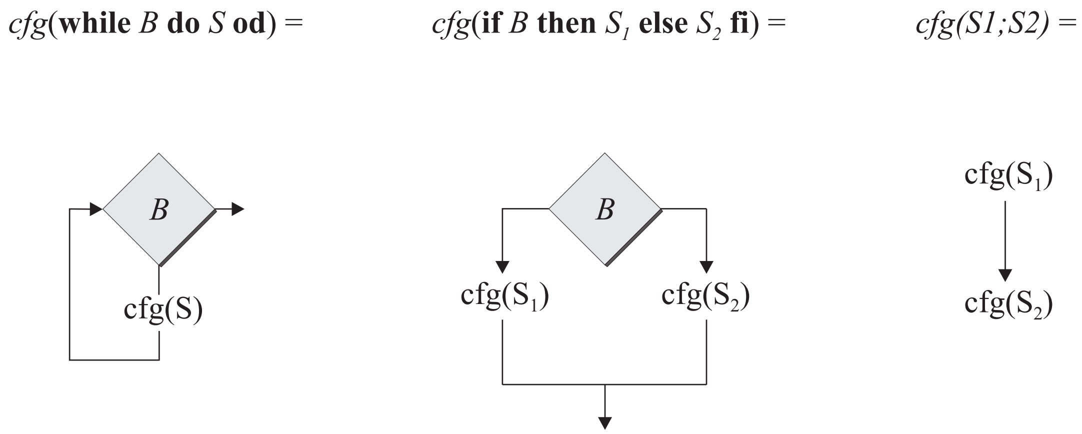


Abbildung 3.1: Kontrollflußgraph für zusammengesetzte Anweisungen. Die Einsetzung eines Teilgraphen $cfg = (N', E', n'_{start}, n'_{stop})$ geschieht folgendermaßen: Alle eingehenden Kanten führen zu n'_{start}, alle ausgehenden Kanten zum eindeutig aus dem Kontext bestimmten Nachfolgeknoten.

knoten der Prozedur; er gehört zur ersten auszuführenden Anweisung der Prozedur. Ebenso sei n_{stop} ein Endknoten, der auf jedem Weg durch den Kontrollflußgraphen erreicht wird.

Beispiel 3.1 Gegeben sei folgende Codesequenz:

```
   r4 = dm(i4, m4);
   f2 = dm(i5, m5);
   comp(r2, r4);
   if gt jump T;
F: r10 = r2;
   jump END;
T: r10 = r4;
   jump END;
END:
```

Der zugehörige Kontrollflußgraph ist in Abbildung 3.2 dargestellt.

□

Definition 3.2 (Basisblock) *Ein **Basisblock** in einem Kontrollflußgraphen ist ein maximal langer Pfad, der höchstens am Anfang eine Verschmelzung und höchstens am Ende eine Verzweigung hat. Dabei bezeichnet man Knoten mit mehr als einem Vorgänger als **Verschmelzungen** und Knoten mit mehr als einem Nachfolger als **Verzweigungen**.*

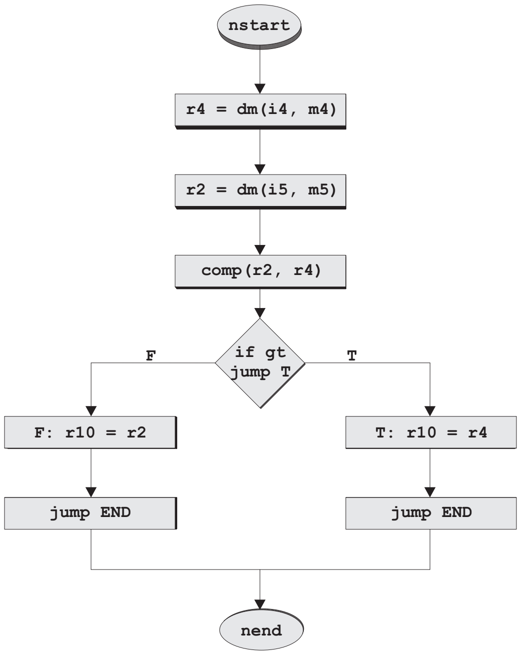


Abbildung 3.2: Der Kontrollflußgraph zu Beispiel 3.1.

Für jeden Basisblock gilt: wird seine erste Anweisung ausgeführt, dann werden bei fehlerfreier Verarbeitung (keine Laufzeitfehler, exceptions, u.ä.) auch alle anderen Anweisungen ausgeführt.

Definition 3.3 (Basisblockgraph) *Der* ***Basisblockgraph*** G_B *eines Kontrollflußgraphen* G_{cf} *entsteht aus* G_{cf}*, indem Basisblöcke zu einem Knoten zusammengefaßt werden. Kanten in* G_{cf}*, die zum ersten Knoten eines Basisblockes führen, führen in* G_B *in den Knoten des Basisblockes hinein; Kanten, die in* G_{cf} *aus dem letzten Knoten eines Basisblockes herausführen, führen in* G_B *aus dem Knoten des Basisblockes heraus.*

Beispiel 3.2 Der Basisblockgraph des Kontrollflußgraphen aus Beispiel 3.2 ist in Abbildung 3.3 dargestellt. Innerhalb der Knoten des Basisblockgraphen sind dabei die zugehörigen Instruktionen aufgeführt.

□

Die Befehle einer Befehlsfolge eines Basisblockes können unter Beachtung gewisser Einschränkungen umgeordnet werden, wobei sich der Gesamteffekt der Befehlsfolge nicht ändern darf. Diese Einschränkungen sind bestimmt durch die Datenabhängigkeiten zwischen den Befehlen. Abhängigkeiten bestehen in der Reihenfolge lesender bzw. schreibender

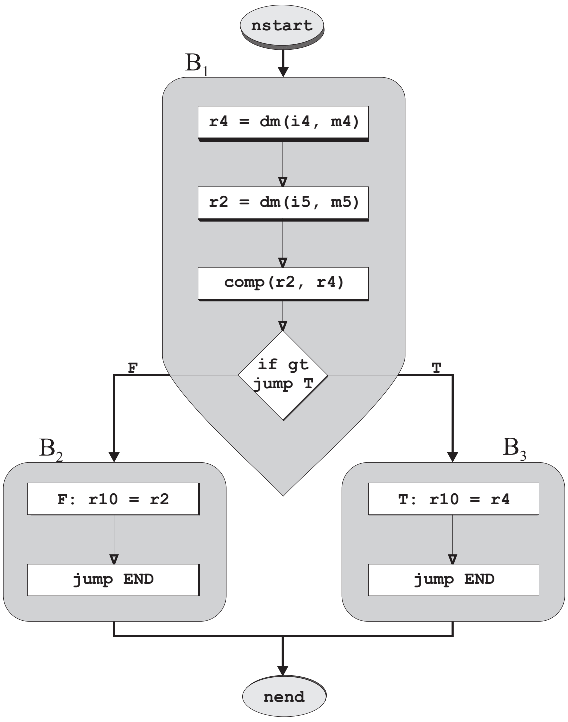


Abbildung 3.3: Der Basisblockgraph zu Beispiel 3.2.

Zugriffe auf bestimmte Komponenten des Maschinenzustandes, etwa Registerinhalte oder Inhalte von Speicherzellen. Schreibende Zugriffe werden als **Setzungen**, lesende Zugriffe als **Benutzungen** bezeichnet. Innerhalb eines Maschinenprogrammes kann es verschiedene Arten von Setzungen geben:

- Modifikation von Registerinhalten durch Lade-Befehle, durch Resultatablage nach Operationen und durch automatische In- bzw. Dekrementierung von Adreßregistern,
- Setzen von Überlauf-, Unterlauf-, oder Übertragsbits und Vergleichsergebnissen im Bedingungscode,
- Abspeichern von Werten in Speicherzellen und
- Modifikation eines Kellerzeigers durch Kellern oder Entkellern von Werten und durch Befehle zur Prozedurorganisation.

Von Benutzungen spricht man in den folgenden Fällen:

- Benutzung von Registerinhalten in Operationen oder zur Adressierung,
- Speichern von Registerinhalten,

- Laden des Inhaltes von Speicherzellen und
- Abfragen von Teilen des Bedingungscodes.

Die Datenabhängigkeiten eines Basisblockes werden mit Hilfe des nun zu definierenden Datenabhängigkeitsgraphen dargestellt.

Definition 3.4 (Datenabhängigkeitsgraph eines Basisblocks) *Es sei ein Basisblock eines Maschinenprogrammes gegeben. Sein* ***Datenabhängigkeitgraph*** *ist ein markierter, azyklischer, gerichteter Graph $G_D = (V_D, E_D)$, dessen Knoten mit den Befehlen des Basisblockes markiert sind. Es führt eine Kante vom Knoten eines Befehls a zum Knoten eines anderen Befehls b, wenn a vor b ausgeführt werden muß, d.h. wenn a in der Befehlsfolge vor b steht und wenn*

- *a eine Ressource setzt, b sie benutzt und der Weg von a nach b setzungsfrei ist (Setzungs- Benutzungsabhängigkeit, true dependence)*
- *a eine Ressource benutzt, b sie setzt und der Weg von a nach b setzungsfrei ist (Benutzungs- Setzungsabhängigkeit, anti dependence)*
- *a und b die gleiche Ressource setzen und der Weg von a nach b setzungs- und benutzungsfrei ist (Setzungs-Setzungsabhängigkeit, output dependence).*

Bezeichnet E_D^{true} die Menge der Setzungs-Benutzungsabhängigkeiten, E_D^{anti} die Menge der Benutzungs-Setzungsabhängigkeiten und E_D^{output} die Menge der Setzungs-Setzungsabhängigkeiten, dann kann die Kantenmenge des Abhängigkeitsgraphen geschrieben werden als

$$E_D \;=\; E_D^{true} \;\cup\; E_D^{anti} \;\cup\; E_D^{output}.$$

Beispiel 3.3 Es sei folgende Codesequenz gegeben:

```
1: r0 = dm(i0, m0);
2: r1 = pm(i8, m8);
3: r2 = r1 + r0;
4: r0 = r3 + r4;
5: r0 = r5 + r6;
```

Diese Befehlsfolge führt zu dem in Abbildung 3.4 dargestellten Datenabhängigkeitsgraphen. Die Instruktionen 1 und 2 definieren die Register, die in Instruktion 3 als Operanden verwendet werden. Daher besteht zwischen den Instruktionen 1 und 3 sowie zwischen 2

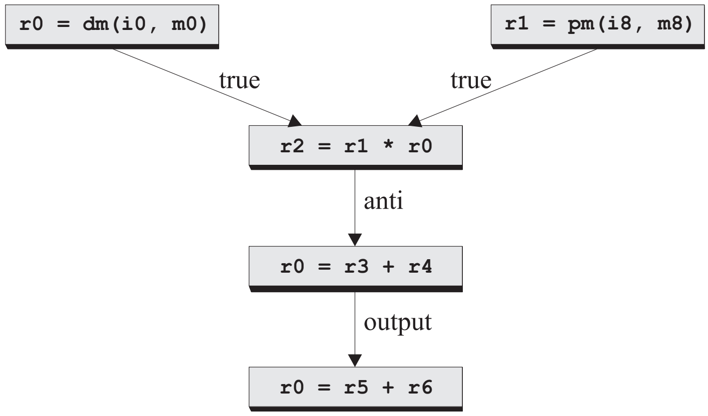


Abbildung 3.4: Der Datenabhängigkeitsgraph zu Beispiel 3.3.

und 3 eine Setzung-Benutzungsabhängigkeit. Instruktion 3 führt einen lesenden Zugriff, die nachfolgende Instruktion einen schreibenden Zugriff auf Register `r0` aus. Somit besteht zwischen 3 und 4 eine Benutzungs-Setzungsabhängigkeit. Da zwischen den Setzungen der Instruktionen 1 und 4 eine Benutzung des Registers `r0` erfolgt, besteht in diesem Fall keine Setzungs-Setzungsabhängigkeit. Die Instruktionen 4 und 5 beschreiben jedoch ohne dazwischenliegende Setzungen oder Benutzungen dasselbe Register; also besteht zwischen diesen Instruktionen eine Setzungs-Setzungsabhängigkeit.

□

Der mögliche Parallelismus innerhalb von Basisblöcken ist in aller Regel sehr beschränkt; deshalb müssen Befehlsanordnungen über die Grenzen von Basisblöcken hinaus betrachtet werden. Zu diesem Zwecke ist die Definition des Abhängigkeitsgraphen zu erweitern und wir erhalten den **verallgemeinerten Abhängigkeitsgraphen** G_D. Änderungen ergeben sich bei bedingten Anweisungen und bei Schleifen. Bei einer bedingten Anweisung gibt es, abhängig vom Wert des Bedingungsausdruckes, zwei mögliche Wege. Der verallgemeinerte Abhängigkeitsgraph enthält alle Abhängigkeiten zwischen Benutzungen und Setzungen auf beiden Wegen. Bei einer Schleife müssen ebenfalls zwei Wege betrachtet werden, die abhängig vom Wert der Schleifenabbruchbedingung eingeschlagen werden können, nämlich ein (weiterer) Schleifendurchlauf oder die Fortsetzung des Programmes mit der ersten Instruktion nach Schleifenende.

Der Datenabhängigkeitsgraph erweist sich jedoch im Hinblick auf globale Analysen als unzureichend. Dies kann man sich an Abbildung 3.5 veranschaulichen. Allein durch Datenabhängigkeiten kann die Verschiebung der Instruktion `r5 = 0` aus Basisblock B_3 in den Basisblock B_2, also in den *else*-Teil hinein, nicht ausgeschlossen werden, da zu dieser Instruktion keine Datenabhängigkeiten existieren. Eine solche Verschiebung hätte jedoch zur Folge, daß die Ausführung von `r5 = 0` abhängig vom Wert der Bedingung der if-Anweisung

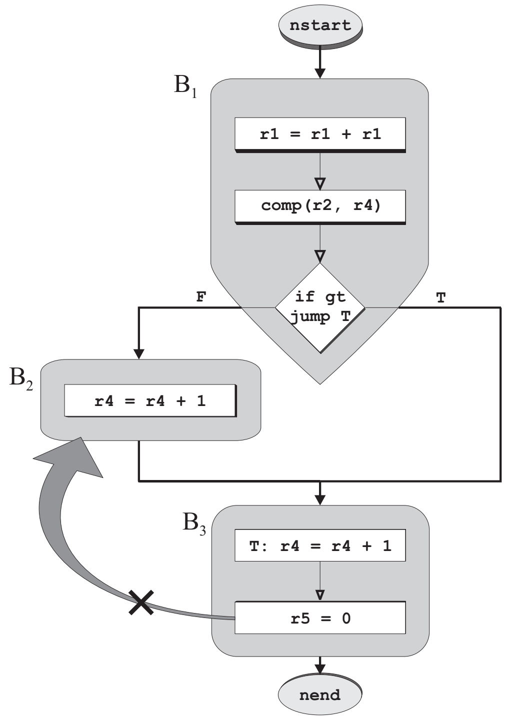


Abbildung 3.5: Problematik globaler Transformationen.

wäre. Die resultierende Codesequenz hätte also eine andere Semantik als das Eingabeprogramm. Bei solchen Abhängkeiten handelt es sich nicht um Datenabhängigkeiten, sondern um Abhängigkeiten, die die Kontrollstruktur des Eingabeprogrammes abbilden. Diese werden mittels eines weiteren Graphen modelliert, des **Kontrollabhängigkeitsgraphen**, der nun im folgenden in Anlehnung an Bashford [Bas95] definiert wird.

Definition 3.5 (Dominator) *Es sei ein Kontrollflußgraph* $G_{cf} = (N, E_{cf}, n_{start}, n_{stop})$ *gegeben. Ein Knoten* $x \in N$ ***dominiert*** *einen Knoten* $y \in N$, $x \; \Delta_d \; y$, *genau dann, wenn jeder Pfad vom Eintrittsknoten* n_{start} *der Prozedur zu* y *den Knoten* x *enthält. Jeder Knoten dominiert sich selbst.*

Definition 3.6 (unmittelbarer Dominator) *Die Menge der Dominatoren eines Knotens* x *bildet eine Kette.* x *ist genau dann* ***unmittelbarer Dominator*** *von* y*, wenn*

- $x \; \Delta_d \; y$
- $\nexists z : \; x \; \Delta_d \; z \; \wedge \; z \; \Delta_d \; y \; \wedge \; z \neq x.$

Definition 3.7 *Der* ***Dominatorbaum*** *eines Kontrollflußgraphen ist ein Baum, der die Knoten des Kontrollflußgraphen enthält. Seine Wurzel ist der Eintrittsknoten* n_{start} *der*

Prozedur. Es existiert eine Kante zwischen x und y genau dann, wenn x unmittelbarer Dominator von y ist.

Auf analoge Weise werden die Postdominanz und verwandte Begriffe definiert. Ein Knoten x **postdominiert** einen Knoten y, $x \ \Delta_p \ y$, genau dann, wenn jeder Pfad von y zu n_{stop} den Knoten x enthält. Ein Knoten x postdominiert sich nie selbst. Die reflexive Hülle der Postdominanz wird mit $\overline{\Delta_p}$ bezeichnet. Der erste Knoten in der Postdominatorkette eines Knotens x wird als **unmittelbarer Postdominator** von x bezeichnet. Die Menge der Postdominatoren eines Knotens $x \neq n_{stop}$ ist nichtleer, so daß jeder Knoten außer n_{stop} einen eindeutig bestimmten unmittelbaren Postdominator besitzt. Der **Postdominatorbaum** ist ein Baum mit Wurzel n_{stop} und einer Kante von x nach y genau dann, wenn x unmittelbarer Postdominator von y ist. Der Postdominatorbaum wird als Dominatorbaum über dem umgekehrten Kontrollflußgraphen berechnet; ein Algorithmus zur Berechnung des Dominatorbaumes ist in Anhang B.1 dargestellt ([ASU88]).

Definition 3.8 *Gegeben sei ein Kontrollflußgraph $G_{cf} = (N, E_{cf}, n_{start}, n_{stop})$. Ein Knoten $x \in N$ hat genau dann **Kontrollabhängigkeit** über $y \in N$, $x \ \delta_c^a \ y$, wenn gilt:*

1. *$(x, a) \ \in \ E_{cf}$*
2. *y postdominiert nicht x, $\neg(y \ \Delta_p \ x)$*
3. *Es existiert ein Pfad $p = x, a, \ldots, y$, so daß für alle $z \in p$ mit $z \neq x$, $z \neq y$ gilt: $y \ \Delta_p \ z$.*

Gilt $x \ \delta_c^a \ y$, dann sagen wir auch, daß Knoten y kontrollabhängig von x ist.

Definition 3.9 *Der **Kontrollabhängigkeitsgraph** G_{cd} des Kontrollflußgraphen $G_{cf} = (N, E_{cf}, n_{start}, n_{stop})$ ist ein gerichteter Graph $G_{cd} = (N, E_{cd})$ mit Kantenmarkierung, so daß $(x, y, tf) \in E_{cd} \ \Leftrightarrow x \ \delta_c^a \ y$ und $(x, y) \in E_{cf}$ und die Kante $(x, y) \in E_{cf}$ ist mit T bzw. F markiert.*

Ist der Postdominatorbaum gegeben, können die Kontrollabhängigkeiten durch Betrachtung der Kanten des Kontrollflußgraphen bestimmt werden. Der in der Implementierung verwendete Algorithmus ist in Anhang B.2 beschrieben ([FOW87]).

Beispiel 3.4 Der Kontrollabhängigkeitsgraph des in Abbildung 3.6 skizzierten Kontrollflußgraphen ist in Abbildung 3.7 dargestellt. Knoten 1 hat Kontrollabhängigkeit über Knoten 2 und 3, da diese nur ausgeführt werden, wenn der bedingte Ausdruck in Knoten 1 den Wert *wahr* annimmt. Knoten 3 ist nicht kontrollabhängig von 2, weil er Knoten 2 postdominiert. Knoten 9 ist Postdominator jeden Knotens außer n_{stop} und sich selbst. Daher ist er von keinem Knoten kontrollabhängig.

□

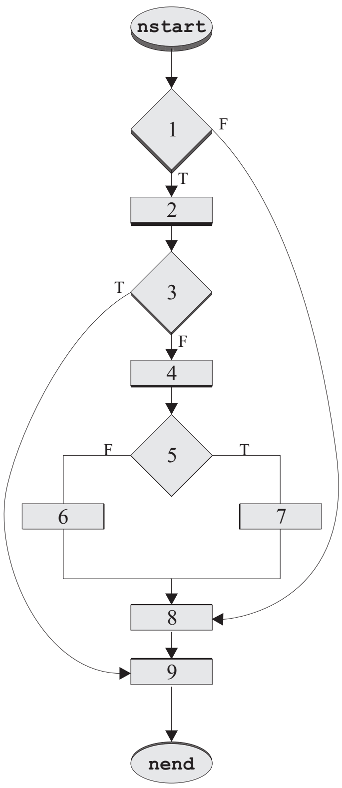


Abbildung 3.6: Der Kontrollflußgraph zu Beispiel 3.4.

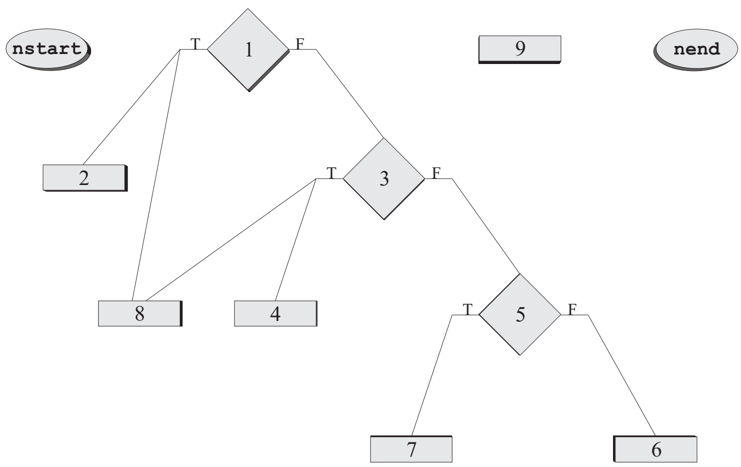


Abbildung 3.7: Der Kontrollabhängigkeitsgraph zu Beispiel 3.4

3.2 Instruktionsanordnung

Durch den Abhängigkeitsgraph wird eine partielle Ordnung auf den Instruktionen des Eingabegraphen definiert. Wir definieren eine Vorrangrelation $\prec \subset V_D \times V_D$ mit

$$i \prec j \;\Leftrightarrow i \underset{G_D}{\overset{*}{\rightarrow}} j$$

Es gilt $i \prec j$ genau dann, wenn Operation j direkt oder indirekt von Operation i abhängt.

Hieraus ergibt sich zunächst das Problem, die Instruktionen so anzuordnen, daß keine derartigen Vorrangbedingungen verletzt werden. Vernachlässigen wir die Benutzungs-Setzungsabhängigkeiten, so entspricht dies dem klassischen Problem des *Precedence Constrained Scheduling*, der Ablaufplanung unter Berücksichtigung von Vorrangbedingungen [GJ79]:

Gegeben sei eine Menge M von Aufgaben der Länge 1, eine partielle Ordnung $\prec$ auf M, ferner m Maschinen und eine Obergrenze T für die Planlänge. Gesucht ist ein Ablaufplan (Schedule) $\sigma \;:\; M \quad\rightarrow\quad \{1,\ldots,T\}$, so daß für alle $i \in \{1,\ldots,T\}$ mit $|\{t \in M : \sigma(t) = i\}| \leq m$ gilt[1]:

$$t \prec t' \qquad \Rightarrow \qquad \sigma(t) < \sigma(t')$$

Diese Problem ist $\mathcal{NP}$-vollständig, außer wenn nur zwei Prozessoren existieren ([Law76]) oder jede Aufgabe i bezüglich der Relation $\prec$ höchstens einen unmittelbaren Nachfolger besitzt ([GJ79]).

Übertragen auf das Instruktionsanordnungsproblem entsprechen die Aufgaben Instruktionen und die Maschinen repräsentieren parallel arbeitende funktionale Einheiten des betrachteten Prozessors. Eine globale Instruktionsanordnung muß zusätzlich dafür sorgen, daß auch die Kontrollabhängigkeiten beim Umordnungsprozeß aufrechterhalten werden.

Um das Scheduling-Problem zu lösen, werden sowohl heuristische Schedulingalgorithmen, die approximative, suboptimale Lösungen finden, als auch exakte Algorithmen, die die optimale Lösung ermitteln können, verwendet. Ein breites Spektrum an heuristischen Algorithmen wird eingesetzt, unter anderem List-Scheduling ([Bas95]) oder Trace-Scheduling ([WM97, Fis81]).

List-Scheduling beginnt mit einer leeren Liste von Instruktionen. Eine Mikrooperation wird in die letzte Instruktion der Liste eingefügt, wenn sie die folgenden drei Bedingungen erfüllt:

[1] Besteht zwischen t und t' als einzige Abhängigkeit eine Benutzungs-Setzungsabhängigkeit, dann muß (bei der zugrundeliegenden Architektur) gelten: $\sigma(t) \leq \sigma(t')$.

1. Sie ist bereit, d.h. alle Vorgänger im Datenabhängigkeitsgraph wurden bereits angeordnet.

2. Sie hat die höchste Priorität innerhalb der Menge aller bereiten Mikrooperationen. Diese Priorität wird aufgrund von Gewichtungsheuristiken berechnet.

3. Die Einfügung in die letzte Instruktion ist zulässig, d.h. alle bereits enthaltenen Mikrooperationen können parallel zu der aktuell betrachteten ausgeführt werden.

Kann keine der bereiten Mikrooperationen in die letzte Instruktion eingefügt werden, wird eine neue Instruktion an die Liste angehängt. List-Scheduling ist die gebräuchlichste Technik für lokale Instruktionsanordnung; die Komplexität beträgt $\mathcal{O}(n^2)$ ([LDSM80, Gas89]).

Ein Verfahren zur globalen Instruktionsanordnung ist das **Anordnen von Pfaden (Trace-Scheduling)**. Die Grundidee dieses Algorithmus ist es, die Instruktionen aufeinanderfolgender Basisblöcke gemeinsam anzuordnen, um so die mögliche Parallelität zu erhöhen. Dabei werden möglichst solche Basisblöcke gemeinsam behandelt, die während der Programmausführung häufig unmittelbar nacheinander ausgeführt werden. Dazu benötigt der Codeerzeuger Informationen darüber, wie oft die Basisblöcke einer Prozedur durchlaufen werden. Diese kann er durch Messungen oder heuristische Schätzungen erhalten. Das Verfahren zerlegt den Kontrollflußgraph einer Prozedur in disjunkte Teilpfade. Es betrachtet zunächst den am häufigsten durchlaufenen Basisblock. Dann wird entschieden, ob und welche vorangehenden bzw. nachfolgenden Basisblöcke gemeinsam mit diesem angeordnet werden sollen. Der gewonnene Teilpfad wird nun so geschickt wie möglich angeordnet. Dabei können eventuell Instruktionen aus den ursprünglichen Basisblöcken über Verschmelzungs- und Verzweigungspunkte verschoben werden. In Basisblöcken, die in diesen Teilpfad einmünden oder von ihm ausgehen, macht dies Kompensationscode erforderlich. Von den noch nicht angeordneten Basisblöcken wird wiederum der am häufigsten durchlaufenene ausgewählt und das Verfahren wiederholt, bis alle Basisblöcke der Prozedur angeordnet sind.

Es wird deutlich, daß bei diesen Verfahren an vielen Stellen Entscheidungen aufgrund von Heuristiken getroffen werden und oft Vereinfachungen in Kauf genommen werden. Für eine detaillierte Analyse solcher heuristischer Methoden sei auf [Lan97] verwiesen. Die meisten exakten Algorithmen hingegen verwenden ganzzahlige lineare Programmierung (ILP – Integer Linear Programming), um die *optimalen* Lösungen zu berechnen. ILP-basierte Scheduler liefern – bis zu einer bestimmten Problemgröße (vgl. Kapitel 8.2) – bessere Schedules als heuristische Algorithmen [CWM94].

3.3 Registerallokation

Die folgende Beschreibung der Registerallokationsproblematik wurde im wesentlichen aus [WM97] übernommen. Eingabe für die Aufgabe der Registerallokation ist im allgemeinen ein Zwischensprachenprogramm, in dem jeder Operation und jeder modifizierten Variablen ein symbolisches Register zugeteilt wurde. Die im Prinzip unbeschränkt vielen symbolischen Register sind den beschränkt vielen Registern der realen Maschine zuzuordnen. Dabei kann niemals ein reales Register zwei verschiedenen symbolischen Registern zugeteilt werden, wenn diese beiden gleichzeitig “lebendig” sind und nicht sicher denselben Wert enthalten. Die nachfolgenden Definitionen basieren auf der Darstellung des Eingabeprogrammes in Form eines Kontrollflußgraphen.

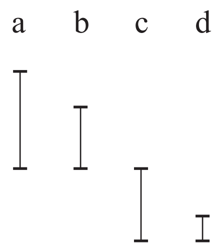
Definition 3.10 (lebendig, Lebensspanne) *Ein symbolisches Register r ist* ***lebendig*** *an einem Programmpunkt p, wenn es auf einem Programmpfad vom Eintrittsknoten der Prozedur nach p eine Setzung von r gibt und einen Pfad von p zu einer Benutzung von r, auf dem r nicht gesetzt wird. Die* ***Lebensspanne*** *eines symbolischen Registers r ist die Menge der Programmpunkte, an denen r lebendig ist.*

Definition 3.11 (Kollision, Registerkollisionsgraph) *Zwei Lebensspannen von symbolischen Registern* ***kollidieren****, wenn eins von ihnen in der Lebensspanne des anderen gesetzt wird. Der* ***Registerkollisionsgraph*** *ist ein ungerichteter Graph. Seine Knoten sind Lebensspannen von symbolischen Registern, und es besteht eine Kante zwischen je zwei kollidierenden Lebensspannen.*

Abbildung 3.8 zeigt ein Beispielprogramm und gibt die Lebenszeiten aller darin definierten Variablen an. Der zugehörige Registerkollisionsgraph ist in Abbildung 3.9 wiedergegeben.

```
a = 2;
b = a+2;
c = a+b;
d = 2*b;
a = c+d;
```

(a) Codesequenz



(b) Lebenszeiten

Abbildung 3.8: Beispielprogramm mit Lebenszeiten aller Variablen.

Eine globale Registerallokation muß, ebenso wie globale Instruktionsanordnung, die Kontrollstruktur der zu analysierenden Prozdedur berücksichtigen. Hierzu werden verschiedene Techniken eingesetzt, wie z. Bsp. Packalgorithmen ([Ben94]), probabilistische

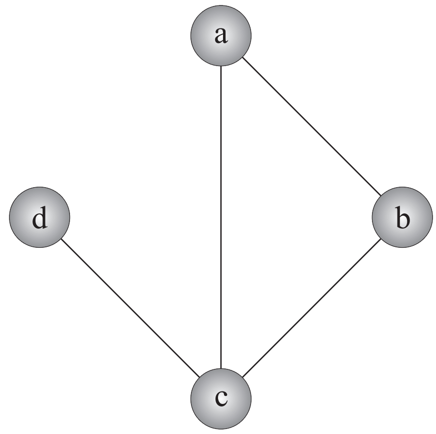


Abbildung 3.9: Der Kollisionsgraph zur Codesequenz aus Abbildung 3.8.

Registerallokation ([PF92]), eine Kombination lokaler und globaler Allokation und Graphfärbung, bei der sowohl Registerallokation als auch -zuteilung berechnet werden (vgl. Seite 40).

Beim Graphfärbeansatz wird das Registerallokationsproblem übersetzt in das Problem, den Registerkollisionsgraph mit k Farben zu färben, wobei k die Anzahl der zur Verfügung stehenden realen Register ist. Die Farben entsprechen also realen Registern. Direkt verbundenen Knoten darf niemals die gleiche Farbe zugeteilt werden. Für $k > 2$ ist das Problem, für beliebige Graphen zu entscheiden, ob sie k-färbbar sind, $\mathcal{NP}$-vollständig. Deshalb werden heuristische Methoden eingesetzt, um eine Färbung zu bestimmen.

Die Grundüberlegung für den Graphfärbealgorithmus ist die folgende: Enthält der Graph G einen Knoten n mit Grad kleiner als k, so kann n mit Sicherheit eine Farbe zugeordnet werden, die von den Farben aller Nachbarn verschieden ist. Der Knoten n wird aus G entfernt, wodurch sich ein neuer Graph G' ergibt, der einen Knoten und einige Kanten weniger enthält. Das Problem wurde rekursiv auf ein kleineres Problem zurückgeführt.

Kann keine k-Färbung gefunden werden – was nicht bedeutet, daß der Graph nicht k-färbbar ist –, müssen einige Lebenszeiten zwischengespeichert werden (*Spilling*). Sei x ein Knoten, dessen Nachbarn die Farben $\{1, \ldots, k\}$ haben, so daß keine Farbe mehr für x übrig ist. Dann gibt es drei Ansätze zur Lösung dieses Problems [Bas95]:

1. x wird im Hauptspeicher zwischengespeichert; alle Zugriffe auf x werden auf den Hauptspeicher abgebildet. Diese Methode kann für Lade-/Speicherarchitekturen wie den zugrundeliegenden ADSP-2106x jedoch nicht angewendet werden.

2. Die Einführung von Zwischenspeicherungen (*Spillcode*) speichert eine Variable nach jeder Definition und lädt sie vor jeder Verwendung.

3. Die Lebenszeit wird in mehrere, disjunkte Lebenszeiten aufgespalten. Damit wird *Spillcode* nur an bestimmten Stellen einer Lebenszeit eingefügt und an bestimmten Stellen wieder geladen. Jede neue Lebenszeit stellt einen neuen Knoten im Registerkollisionsgraph dar, wobei es gewöhnlich weniger Kollisionen gibt, als mit der ursprünglichen Lebenszeit [CH90, Bas95].

Das Einfügen von Zwischenspeicherungen (*Spillcode*) oder das Aufspalten einer Lebensspanne führen zu einem neuen Kollisionsgraphen. Dabei kann der zweite Ansatz als Extremfall von Ansatz 3 angesehen werden. Das Aufspalten von Lebensspannen, d.h. die Bestimmung der aufzuspaltenden Lebensspannen und der Stellen der Aufspaltung, ist jedoch sehr schwierig. Die Bestimmung von optimalen Lösungen ist für beide Probleme $\mathcal{NP}$-schwer ([GJ79]).

Die Probleme der Aufspaltung von Lebensspannen bzw. Einfügung von Zwischenspeicherungen sind in eine ILP-Beschreibung nicht vollständig zu integrieren, da jedes aufgestellte lineare Programm sich immer auf eine fest vorgegebene Instruktionssequenz bezieht. Allerdings kann bei einer der untersuchten Modellierungen (vgl. Kapitel 6.7) im Falle, daß die vorhandenen Register nicht ausreichen, eine Empfehlung für eine Aufspaltung einer Lebensspanne gegeben werden. Aufgrund dieser kann eine umgebende Allokationsfunktion die vorgeschlagene Aufspaltung durchführen und danach die Erzeugung eines neuen ILP's für die geänderte Befehlsfolge anstoßen. Die ermittelten Lösungen sind also optimal bezüglich des Resultates der Codeselektion und bezüglich der zuvor durchgeführten Verteilung von Variablen auf Register bzw. Hauptspeicher; Aufspaltungen von Lebensspannen und Einfügungen von Zwischenspeicherungen werden nicht erfaßt.

Kapitel 4

Ganzzahlige lineare Programmierung

4.1 Allgemeiner Überblick

Die Anwendungsgebiete ganzzahliger linearer Programmierung (*Integer Linear Programming, ILP*) sind sehr vielfältig. ILP-Methoden werden eingesetzt zur Lösung natur- und wirtschaftswissenschaftlicher, ebenso wie politischer Problemstellungen ([NW88, NKT89]). Typische Anwendungen betreffen, allgemein gesprochen, die Verwaltung und den effizienten Einsatz knapper Ressourcen zur Steigerung der Produktivität. Hierbei sind zum Beispiel Produktionsplanung, Maschinenfolgeplanung ([Brü95]), Portfolio-Analyse([DK96]), aber auch VLSI-Design und Design automatisierter Produktionssysteme zu nennen. Neuere wissenschaftliche Anwendungen befassen sich mit Problemen der Molekularbiologie, Hochenergiephysik und Röntgenkristallographie. Ein politisches Anwendungsgebiet ist die Aufteilung eines Landes in Wahlbezirke. Das Ermitteln der optimalen Lösung eines ganzzahligen linearen Programmes ist $\mathcal{NP}$-schwer ([GJ79]); dennoch können viele große Instanzen solcher Probleme gelöst werden. Neuere Forschungen haben zu einem Verständnis von Eigenschaften geführt, die einige ILP-Formulierungen leichter lösbar machen. Solche Formulierungen bezeichnet man als strukturiert ([CWM94]), weil ihre Nebenbedingungen eine spezielle Struktur haben. Die Verwendung strukturierter Formulierungen hat zu bedeutenden Erfolgen bei dem ILP-Ansatz zur Lösung $\mathcal{NP}$-vollständiger Probleme, wie z.B. dem Problem des Handlungsreisenden ([LLKS85]) oder der (0-1)-Programmierung geführt ([CJP83]). Um eine effiziente Lösung des Instruktionsanordnungsproblems zu erreichen, ist es daher sehr wichtig, eine strukturierte Formulierung und keinen ad-hoc-Ansatz zu wählen ([CWM94]).

4.2 Mathematische Grundlagen

In diesem Kapitel werden zunächst einige mathematische Grundbegriffe eingeführt, die für das Verständnis der nachfolgenden Abschnitte unerläßlich sind. Anschließend werden die wichtigsten Konzepte der (reinen) linearen und der ganzzahligen linearen Programmierung vorgestellt.

Definition 4.1 *Sei gegeben eine Menge $S \subseteq \mathbb{R}^n$. Ein Punkt $x \in \mathbb{R}^n$ heißt* ***Konvexkombination*** *von Punkten aus S, wenn es eine endliche Menge von Punkten $\{x^i\}_{i=1}^t$ in S und ein $\lambda \in \mathbb{R}^t$ gibt mit $\sum_{i=1}^t \lambda_i = 1$, $x = \sum_{i=1}^t \lambda_i x^i$ und $\lambda_i \geq 0$ für alle $i = 1, \ldots, t$. Gilt $\lambda_i > 0$ für alle $i = 1, \ldots, t$, so heißt x* ***echte Konvexkombination*** *der Punkte aus S. Die* ***konvexe Hülle*** *von S, $conv(S)$, ist die Menge aller Punkte, die Konvexkombinationen von Punkten in S sind.*

Definition 4.2 *Eine Menge von Punkten $x^1, \ldots, x^k \in \mathbb{R}^n$ heißt* **linear unabhängig**, *wenn für die einzige Lösung von $\sum_{i=1}^k \lambda_i x^i = 0$ gilt: $\lambda_i = 0$ für alle $i = 1, \ldots, k$.*

Definition 4.3 *Eine Menge von Punkten* $x^1, \dots, x^k \in \mathbb{R}^n$ *heißt* **affin unabhängig**, *wenn für die einzige Lösung von* $\sum_{i=1}^k \alpha_i x^i = 0$, $\sum_{i=1}^k \alpha_i = 0$ *gilt:* $\alpha_i = 0$ *für alle* $i = 1, \dots, k$.

Definition 4.4 *Ein* ***Polyeder*** $P \subseteq \mathbb{R}^n$ *ist eine Menge von Punkten, die eine endliche Anzahl linearer Ungleichungen erfüllt;* $P = \{x \in \mathbb{R}^n | Ax \leq b\}$, *wobei* (A, b) *eine* $m \times (n+1)$*-Matrix ist. Ein Polyeder heißt* ***rational***, *wenn es eine* $m \times (n+1)$*-Matrix* (A', b') *mit rationalen Koeffizienten gibt, so daß gilt* $P = \{x \in \mathbb{R}^n | A'x \leq b'\}$.

Definition 4.5 *Ein Punkt* $x \in P$ *heißt* ***Ecke des Polyeders*** P, *wenn sich* x *nicht als echte Konvexkombination von zwei verschiedenen Punkten aus* P *darstellen läßt.*

Definition 4.6 *Ein Polyeder* $P \subseteq \mathbb{R}^n$ *heißt* ***beschränkt***, *wenn es ein* $\omega \in \mathbb{R}_+$ *gibt, so daß* $P \subseteq \{x \in \mathbb{R}^n | -\omega \leq x_j \leq \omega$ *für* $j = 1, \dots, n\}$. *Ein beschränktes Polyeder heißt* ***Polytop***.

Definition 4.7 *Ein Polyeder* P *besitzt* **Dimension** k, $dim(P) = k$, *wenn die maximale Anzahl affin unabhängiger Punkte in* P $k+1$ *beträgt.*

Ist ein Polyeder $P = \{x \in \mathbb{R}^n | Ax \leq b\}$ gegeben, stellt sich die Frage, welche der Gleichungen $a^i x \leq b_i$ notwendig zur Beschreibung von P sind und welche weggelassen werden können.

Definition 4.8 *Eine Ungleichung* $\pi x \leq \pi_0$ *heißt* ***gültig*** *für* P, *wenn sie von allen Punkten in* P *erfüllt wird.*

Definition 4.9 *Ist* $\pi x \leq \pi_0$ *eine gültige Ungleichung für* P *und* $F = \{x \in P | \pi x = \pi_0\}$, *so wird* F *ein* ***Seite*** *von* P *genannt und wir sagen, daß* $\pi x \leq \pi_0$ *die Menge* F *repräsentiert. Eine Seite* F *heißt* ***eigentliche Seite***, *wenn* $F \neq \emptyset$ *und* $F \neq P$ *gilt. Eine Seite* F *von* P *heißt* ***Facette*** *von* P, *wenn gilt:* $dim(F) = dim(P) - 1$.

In [NW88] wurde gezeigt, daß die Charakterisierung von Facetten zur Beschreibung von P notwendig und hinreichend sind. Insbesondere ist für jede Facette F von P eine der Ungleichungen, die F repräsentieren, zur Beschreibung von P notwendig. Jede Ungleichung $a^r x \leq b_r$, die eine Seite von P der Dimension $d \leq dim(P) - 1$ repräsentiert, ist zur Beschreibung von P irrelevant. Zu jedem Polyeder P gibt es ein endliches System linearer Ungleichungen, die eine minimale Repräsentation von P darstellen ([NW88]). Bei der Beschreibung eines Polyeders sollte grundsätzlich versucht werden, eine solche minimale Repräsentation zu finden.

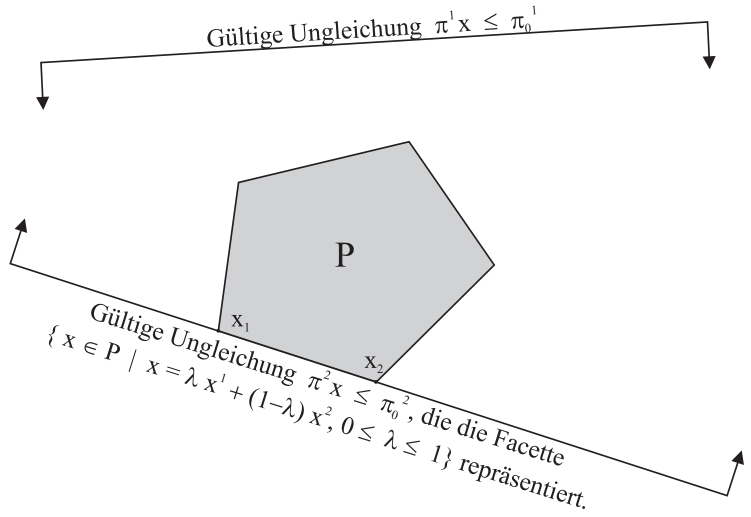


Abbildung 4.1: Gültige Ungleichungen und Facetten.

4.2.1 Theorie der linearen Programmierung

Ein gutes Verständnis von Theorie und Algorithmen der linearen Programmierung ist unerläßlich zum Verständnis ganzzahliger Programmierung. Algorithmen der linearen Programmierung werden häufig innerhalb von Algorithmen der ganzzahligen linearen Programmierung verwendet, um Schranken für den optimalen Zielfunktionswert zu erhalten. Eine tiefere Verbindung zwischen linearer und ganzzahliger linearer Programmierung besteht darin, daß es zu jedem ganzzahligen Problem ein Problem der linearen Programmierung gibt, das dieselbe Lösung wie das ILP besitzt.

Im folgenden wird kurz auf die verwendete Terminologie eingegangen und einige zentrale Sätze der **Dualitätstheorie** werden aufgeführt. Auf Dualitätsaussagen beruhen notwendige und hinreichende Kriterien für die Optimalität einer Lösung sowie Berechnungsmethoden zur Ermittlung unterer Schranken für den optimalen Zielfunktionswert eines linearen Programmes.

Beim Problem der linearen Programmierung ist eine lineare Zielfunktion gegeben, die unter Berücksichtigung einer endlichen Anzahl linearer Nebenbedingungen minimiert (oder maximiert) werden soll. Ein lineares Programm besteht aus folgender Optimierungsaufgabe:

$$\begin{array}{rcl} min \quad z_{LP} & = & c^T x \\ Ax & \geq & b \\ x & \in & \mathbb{R}^n_+ \end{array} \tag{4.1}$$

mit

$$c \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}.$$

Die Menge $S = \{x \in \mathbb{R}^n_+ \mid Ax \geq b\}$ heißt **Zulässigkeitsbereich**, ein $x \in S$ heißt **zulässige Lösung**. Der Zulässigkeitsbereich S eines linearen Programmes ist ein konvexes Polyeder. Die obige Definition des Zulässigkeitsbereiches unterscheidet sich von der Definition 4.4 in der Richtung des Ungleichungszeichens. Da jede $\geq$-Bedingung jedoch durch Multiplikation mit -1 in eine äquivalente $\leq$-Bedingung transformiert werden kann, ist dieser Unterschied nicht von Bedeutung. Eine Instanz des Problems heißt zulässig, wenn $S \neq \emptyset$. Die Funktion $z_{LP} = c^T x$ wird als **Zielfunktion** bezeichnet, ein zulässiger Punkt $x \in S$, der z_{LP} minimiert, als **optimale Lösung**. Ist x^* eine optimale Lösung, so wird $z_{LP}(x^*) = c^T x^*$ als **optimaler Zielfunktionswert** bezeichnet. Eine Instanz von (4.1) heißt **unbeschränkt**, wenn es für alle $\omega \in \mathbb{R}$ ein $x \in S$ gibt mit $c^T x < \omega$. Die Lösung einer Instanz von (4.1) zu bestimmen, bedeutet, eine optimale Lösung zu ermitteln, oder zu zeigen, daß sie unbeschränkt oder unzulässig ist. Sind alle auftretenden Größen rational, was eine natürliche Annahme bei auf Computern gelösten Problemen darstellt, besitzt jede zulässige Instanz eines linearen Programmes mindestens eine optimale Lösung oder ist unbeschränkt.

Ist ein lineares Programm gegeben, dann stellt sich die Frage, wie die Optimalität einer Lösung bewiesen werden kann, und ob eine untere Schranke für den optimalen Zielfunktionswert ermittelt werden kann. In diesem Zusammenhang wird das Problem 4.1 als **primales Problem** bezeichnet. Das sogenannte **duale Problem** zu (4.1), kurz das **Duale**, wird wie folgt definiert:

$$\begin{array}{rrcl} max & z'_{LP} & = & y^T b \\ & y^T A & \leq & c \\ & y & \in & \mathbb{R}^m_+ \end{array} \tag{4.2}$$

Der Vektor der Zielfunktionskoeffizienten des Dualen entspricht also dem Begrenzungsvektor des primalen Problems und umgekehrt; die Koeffizientenmatrix des Dualen ist die transponierte Koeffizientenmatrix des Primalen. Jedes lineare Programm läßt sich auf diese Weise dualisieren. Das Duale des Dualen (4.2) ist wiederum das Primale (4.1), so daß es keine Rolle spielt, welches der beiden Probleme als Duales und welches als Primales bezeichnet wird.

Beispiel 4.1 Es sei das folgende primale Problem gegeben:

$$
\begin{array}{lrcrcrcl}
min & 36x_1 & + & 72x_2 & + & 24x_3 & & \\
 & x_1 & + & 2x_2 & + & x_3 & \geq & 6 \\
 & 2x_1 & + & 3x_2 & & & \geq & 3 \\
 & 4x_1 & - & x_2 & + & x_3 & \geq & -9 \\
 & -x_1 & + & x_2 & + & x_3 & \geq & 15 \\
 & & & & & x_1, x_2, x_3 & \geq & 0
\end{array}
$$

Dann lautet das zugehörige Duale:

$$
\begin{array}{lrcrcrcrcl}
max & 6y_1 & + & 3y_2 & - & 9y_3 & + & 15y_4 & & \\
 & y_1 & + & 2y_2 & + & 4y_3 & - & y_4 & \leq & 36 \\
 & 2y_1 & + & 3y_2 & - & y_3 & + & y_4 & \leq & 72 \\
 & y_1 & - & & + & y_3 & + & y_4 & \leq & 24 \\
 & & & & & & & y_1, y_2, y_3, y_4 & \geq & 0
\end{array}
$$

$\square$

Satz 4.1 (schwache Dualität) *Ist $\overline{x}$ eine zulässige Lösung des primalen Problems*

$$
\begin{array}{lrcl}
min & c^T x & & \\
 & Ax & \geq & b \\
 & x & \geq & 0
\end{array}
$$

und $\overline{y}$ zulässige Lösung des dualen Problems

$$
\begin{array}{lrcl}
max & y^T b & & \\
 & y^T A & \leq & c \\
 & y & \geq & 0,
\end{array}
$$

dann gilt

$$\overline{y}^T b \leq c^T \overline{x}.$$

Satz 4.2 (starke Dualität) *Seien x^* und y^* zulässige primale bzw. duale Lösungen. Es gilt $c^T x^* = y^{*T} b$ genau dann, wenn x^* optimale Lösung des Primalen und y^* optimale Lösung des Dualen ist.*

Folgerung:

Für ein Paar dualer linearer Programme P und D gelten folgende Aussagen:

1. Ein lineares Programm besitzt genau dann eine optimale Lösung, wenn sowohl P als auch D (mindestens) eine zulässige Lösung besitzen. Die optimalen Zielfunktionswerte von P und D stimmen überein.
2. Ist P nicht nach unten beschränkt, ist D unlösbar.
3. Ist D nicht nach oben beschränkt, ist P unlösbar.

Diese Sätze finden z.B. in dem in Kapitel 4.3 vorgestellten Branch-and-Bound-Algorithmus Anwendung.

Zur Lösung von linearen Programmen wurden verschiedene Algorithmen entwickelt; die bekanntesten sind der **Simplex Algorithmus**, die **Ellipsoid-Methode** und der **projektive Skalierungsalgorithmus** von Karmarker. Primaler und Dualer Simplexalgorithmus sind Bestandteil vieler Softwaresysteme zur linearen und ganzzahligen linearen Programmierung. Das gilt auch für CPLEX, das im Rahmen dieser Arbeit zur Implementierung eingesetzt wird. Eine intuitive Beschreibung der Vorgehensweise des Simplex-Algorithmus findet sich im Anhang. Die Laufzeit beträgt typischerweise $\mathcal{O}(m)$, wobei m die Anzahl der Nebenbedingungen angibt ([Chv83]). Dennoch kann die Laufzeit des Simplexalgorithmus für bestimmte Eingaben exponentiell in der Eingabegröße wachsen (z.B. für den Klee-Minty-Würfel, siehe dazu [Chv83], [Sei96]). Der Ellipsoid-Algorithmus, sowie der projektive Skalierungsalgorithmus besitzen hingegen polynomielle Laufzeit; dennoch ist die Ellipsoid-Methode aufgrund zu hoher Laufzeiten praktisch nicht einsetzbar. Auf eine genauere Beschreibung dieser Algorithmen wird an dieser Stelle verzichtet; detaillierte Behandlungen können [NW88], [PS82], [Chv83], [Sei96] oder [Sch93] entnommen werden.

4.2.2 Theorie der ganzzahligen linearen Programmierung

Unter einem **gemischt ganzzahligen linearen Programm** (*Mixed Integer Linear Program*, MILP) versteht man ein Optimierungsproblem der Form

$$\min \{c^T x + h^T y \mid Ax + Gy \geq b;\ x \in \mathbb{Z}^n,\ y \in \mathbb{R}^n\} \tag{4.3}$$

mit

$$c \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}$$

Müssen alle auftretenden Variablen ganzzahlig sein, so ergibt sich als Spezialfall ein **ganzzahliges lineares Programm** (*Integer Linear Program*, ILP):

$$\begin{array}{rrcl} \min & z_{IP} & = & c^T x \\ & x & \in & P_F \cap \mathbb{Z}^n \end{array} \tag{4.4}$$

mit

$$P_F = \{x \mid Ax \geq b,\ x \in \mathbb{R}^n_+\},\ c \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}$$

Für die weiteren Betrachtungen sei angenommen, daß $A \in \mathbb{Z}^{m\times n}$ und $b \in \mathbb{Z}^m$ gilt. Dann kann die optimale Lösung eines linearen Programmes ermittelt werden, indem das folgende Problem gelöst wird ([NW88]):

$$\begin{aligned} \min \quad z_{IP} &= c^T x \\ x &\in P_I \end{aligned} \tag{4.5}$$

mit

$$P_I = conv(\{x \mid x \in P_F \ \cap \mathbb{Z}\}).$$

Im folgenden wird noch eine weitere Definition benötigt:

Definition 4.10 (Relaxation) *Es sei ein Optimierungsproblem Q mit Zulässigkeitsbereich $X(Q)$ gegeben. Ein Optimierungsproblem Q^R heißt eine **Relaxation** von Q, wenn für den Zulässigkeitsbereich $X(Q^R)$ gilt:*

$$X(Q) \subset X(Q^R).$$

Eine Darstellung von P_F und P_I (Gleichungen 4.4, 4.5) ist für den zweidimensionalen Fall in Abbildung 4.2 gegeben.

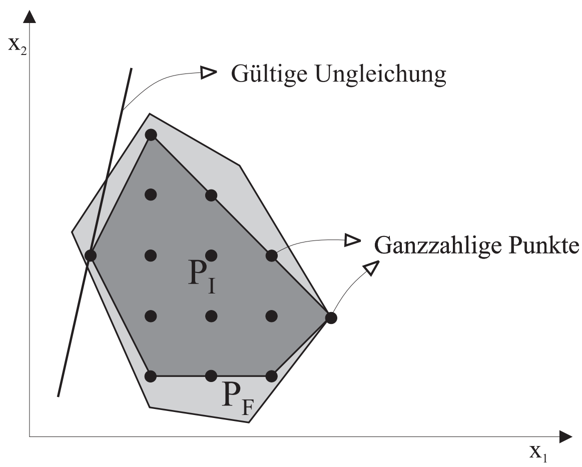


Abbildung 4.2: Zulässigkeitsbereiche.

Die ganzzahligen Punkte innerhalb von P_F stellen die zulässigen Lösungen des ganzzahligen linearen Programmes dar; abhängig von der Zielfunktion entspricht mindestens einer von ihnen einer optimalen Lösung. Der Zulässigkeitsbereich von (4.4) besteht nur aus den ganzzahligen Punkten, wohingegen der Zulässigkeitsbereich von (4.5), P_I, aus der konvexen Hülle dieser Punkte besteht.

Da P_F nur mit Hilfe von Gleichungen und Ungleichungen beschrieben wird, kann jede lineare Funktion über P_F in polynomieller Zeit durch Einsatz von Algorithmen der linearen Programmierung optimiert werden. Diese Erkenntnis kann in der Regel jedoch nicht unmittelbar bei der Lösung eines ILP's umgesetzt werden. Meist ist keine Repräsentation von P_I als lineares Gleichungssystem bekannt; zudem ist die Anzahl der Ungleichungen, die erforderlich sind, um die konvexe Hülle zu beschreiben, im allgemeinen extrem groß ([NKT89]). Vor diesem Hintergrund wird zunächst versucht, ein verwandtes Problem zu lösen, nämlich die **LP-Relaxation** des ganzzahligen linearen Programmes, die wie folgt lautet:

$$\begin{array}{rrcl} \min & z_R & = & c^T x \\ & x & \in & P_F \end{array} \tag{4.6}$$

mit

$$P_F = \{x \mid Ax \geq b,\ x \in \mathbb{R}^n_+\},\ c \in \mathbb{R}^n,\ b \in \mathbb{Z}^m,\ A \in \mathbb{Z}^{m \times n}$$

Da $P_I \subseteq P_F$, folgt aus (4.5) und (4.6), daß $z_R \leq z_{IP}$. Gilt $P_F = P_I$, dann heißt das Polyeder P_F ganzzahlig und in diesem Falle gilt $z_R = z_{IP}$. Die optimale Lösung kann also in Polynomialzeit ermittelt werden, indem die LP-Relaxation gelöst wird. Daher sollte man beim Formulieren eines ganzzahligen linearen Programmes versuchen, eine Menge von Nebenbedingungen zu finden, die P_F als ganzzahliges Polyeder beschreiben. Es konnte bewiesen werden, daß es für jedes beschränkte System rationaler Ungleichungen ein ganzzahliges Polyeder gibt ([GE93], [PS82] 318-322). Leider ist für die meisten Probleme nicht bekannt, wie diese zusätzlichen linearen Ungleichungen zu formulieren sind – und es könnte exponentiell viele von ihnen geben ([NKT89]).

Im allgemeinen gilt $P_I \subset P_F$ und die LP-Relaxation liefert eine untere Schranke für die Zielfunktion. Die Effizienz vieler Algorithmen zur Lösung von ganzzahligen linearen Programmen hängt von der Güte dieser Schranke ab. Je besser P_F den Zulässigkeitsbereich P_I approximiert, desto schärfer ist die Schranke, so daß es für eine effiziente Lösung einer ILP-Formulierung extrem wichtig ist, daß P_F nahe an P_I liegt (vgl. Kapitel 4.3).

Für kombinatorische Probleme in anderen Bereichen wurden Fortschritte bei der Lösung von ILP-Formulierungen auf verschiedenen Wegen erzielt. Zunächst wurde durch formale Analysen der Struktur der Nebenbedingungen eine straffe Beschreibung von P_F entwickelt, die P_I näher approximiert. Die Problemformulierung konnte zudem durch neue gültige Ungleichungen, die aufgrund der Ganzzahligkeit von Variablen entstehen, weiter gestrafft werden. Damit konnten bessere Schranken für die Zielfunktion gefunden werden und die Lösungseffizienz der ILP-Formulierungen gesteigert werden, da straffere Modelle eine kleinere Menge an zulässigen Lösungen besitzen, die durchsucht werden müssen. Schließlich hat das vertiefte Verständnis der Nebenbedingungen auch zu besseren Relaxations- und Branchingstrategien geführt. Die in dieser Arbeit untersuchten Modellierungsansätze versuchen, durch Einsatz solcher Techniken auch eine effiziente Lösung des verschränkten Instruktionsanordnungs- und Registerallokationsproblems zu realisieren.

4.3 Das Branch-And-Bound Verfahren

Mit Hilfe des Branch-and-Bound-Verfahrens ([NW88]) kann die optimale Lösung eines ganzzahligen linearen Programmes ermittelt werden. Durch gesteuerte Aufzählung sowie durch Berechnung von Relaxationen oder der Lösung dualer linearer Programme wird der Zulässigkeitsbereich S des zu berechnenden Optimierungsproblems IP in eine Menge von disjunkten Teilmengen $\{S^i \mid i = 1, \ldots, k\}$ aufgespalten und das Problem über jeder dieser Teilmengen berechnet.

Definition 4.11 *Es sei eine Menge S gegeben. Die Mengen $\{S^i\}_{i=1}^k$ heißen **Zerlegung** von S, wenn gilt $\bigcup_{i=1}^k S^i = S$. Eine Zerlegung heißt **Partition**, wenn $S^i \cap S^j = \emptyset$ für $i, j = 1, \ldots, k,\ i \neq j$.*

Satz 4.3 *Sei $\{S^i\}_{i=1}^k$ eine Zerlegung von S und $z_{IP}^i = \min\{c^T x \mid x \in S^i\}$. Dann gilt für $z_{IP} = \min\{c^T x \mid x \in S\}$*

$$z_{IP} = \min_{i=1,\ldots,k} z_{IP}^i.$$

Dieser Satz beschreibt den Ausgangspunkt für einen **Teile und Herrsche**-Algorithmus. Ein Problem wird in mehrere kleine Teile zerlegt, die dann mit dem gleichen Algorithmus rekursiv gelöst werden ([Meh88]). Danach wird die Lösung des Problems aus den Lösungen der Teilprobleme zusammengesetzt.

Im Extremfall kann dies zu einer vollständigen Aufzählung aller Elemente von S führen – dieser Weg ist nur für Probleme mit einer sehr kleinen Anzahl von Elementen begehbar. Damit Hoffnung besteht, mittels dieses enumerativen Ansatzes zu einer Lösung zu gelangen, muß die Anzahl der betrachteten Partitionen möglichst gering gehalten werden. Um den Berechnungsaufwand für die einzelnen Teilprobleme einzugrenzen, können Relaxationen oder die dualen Teilprobleme gelöst werden, d.h. es braucht nicht auch bei der Betrachtung jedes Knotens die optimale Lösung des entsprechenden ganzzahligen Teilproblems berechnet zu werden. Sei RP^i eine Relaxation von IP^i und DP^i das Duale von IP^i. Der nachfolgende Satz liefert Kriterien dafür, wann Äste des Aufzählungsbaumes abgeschnitten werden können. Das Vorgehen kann man sich an dem in Abbildung (4.3) dargestellten Baum veranschaulichen, bei dem die Kinder eines Knotens Partitionen des Elternproblems darstellen; einige Zweige konnten abgeschnitten werden und brauchen daher im weiteren Verlauf des Algorithmus nicht betrachtet zu werden. Der nachfolgende Satz liefert Kriterien dafür, wann einzelne Äste des Aufzählungsbaumes abgeschnitten werden können.

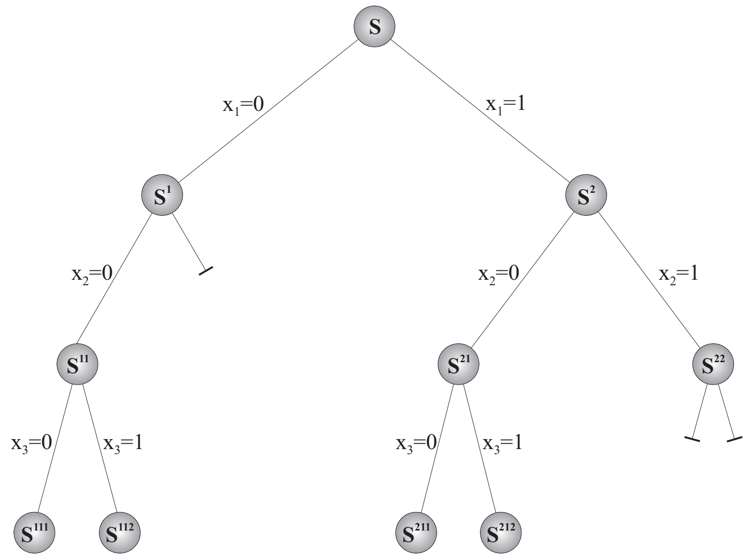


Abbildung 4.3: Exemplarischer Branch-and-Bound-Baum.

Satz 4.4 *Der Aufzählungsbaum kann am Knoten für S^i beschnitten werden, wenn eine der folgenden Bedingungen erfüllt ist:*

1. *RP^i ist unzulässig.*
2. *Eine optimale Lösung x_R^i von RP^i erfüllt $x_R^i \in S^i$ und $z_R^i = c^T x_R^i$.*
3. *$z_R^i \geq \overline{z_{IP}}$, wobei $\overline{z_{IP}}$ der Wert irgendeiner zulässigen Lösung von IP ist.*
4. *Das Problem DP^i ist nicht nach oben beschränkt.*
5. *DP^i besitzt eine zulässige Lösung z_{DP}^i mit $z_{DP}^i \geq \overline{z_{IP}}$.*

Erläuterung:

zu 1.) Existiert keine zulässige Lösung der Relaxation, dann existiert auch keine ganzzahlige zulässige Lösung, denn der Zulässigkeitsbereich der Relaxation stellt eine Obermenge des Zulässigkeitsbereiches des ganzzahligen linearen Programmes dar.

zu 2.) Wurde eine zulässige Lösung gefunden, so braucht das Problem nicht weiter zerlegt zu werden.

zu 3.) Der Zielfunktionswert z_R^i von RP^i stellt immer eine untere Schranke für den Zielfunktionswert von IP^i dar. Da bereits eine zulässige Lösung von IP gefunden wurde, deren Zielfunktionswert $\overline{z_{IP}}$ kleiner oder gleich z_R^i ist, kann IP^i keine bessere zulässige Lösung liefern. Somit braucht IP^i nicht weiter zerlegt zu werden; der Teilbaum mit Wurzel IP^i kann also abgeschnitten werden.

zu 4.) Ist das Duale nicht nach oben beschränkt, dann ist das zugehörige primale Problem unlösbar. Es existiert somit keine zulässige Lösung, insbesondere keine ganzzahlige zulässige Lösung.

zu 5.) Aufgrund des Satzes der schwachen Dualität stellt jede zulässige Lösung des dualen eine untere Schranke für alle zulässigen Lösungen des primalen Problems dar. Die bisher gefundene Lösung kann also durch eine Lösung von IP^i nicht verbessert werden.

Der allgemeine Branch-and-Bound-Algorithmus ist auf Seite 67 in Pseudocode angegeben (Algorithmus 4.1). Dabei sei $L = \{IP^i\}$ eine Menge von ganzzahligen linearen Programmen der Form $z_{IP}^i = \min\{c^T x \mid x \in S^i\}$ mit $S^i \subset S$. Mit jedem Problem aus L ist eine untere Schranke $\underline{z^i} \leq z_{IP}^i$ verbunden.

Zur Realisierung der Zerlegung werden verschiedene Verfahren eingesetzt. Als ein Beispiel wird nun das **Dakin-Verfahren** vorgestellt ([Din92]), bei dem nach einer vorher festgelegten Variable x_n separiert wird. Wir erhalten die beiden Teilprobleme IP^{i_1} und IP^{i_2}, indem zum einen $x_n \leq \lfloor x_n^R \rfloor$ (Teilproblem IP^{i_1}) und zum anderen $x_n \geq \lceil x_n^R \rceil$ (Teilproblem IP^{i_2}) als neue Ungleichung zu IP^i hinzugefügt wird. Ein Vorteil dieser Methode ist, daß nur einfache Nebenbedingungen, die untere und obere Schranken angeben, zur Relaxation hinzugefügt werden. Daher ist es nur nötig, sich diese Schranken zu merken; der erforderliche Berechnungsaufwand zur Lösung der Relaxationen vergrößert sich im Gegensatz zu einigen anderen Verfahren, wie z.B. der Land-Doig Methode ([Din92]), mit zunehmender Tiefe des Branch-and-Bound-Baumes nicht.

Satz 4.5 *Besitzt Knoten t des Aufzählungsbaumes den Zulässigkeitsbereich S^t und gilt $\min\{c^T x \mid x \in S_R^t\} < \overline{z_{IP}}$, dann kann Knoten t nicht abgeschnitten werden.*

Dieser Satz zeigt, daß unabhängig von der konkreten Entwicklung des Baumes die Schranken, d.h. die Güte der Relaxation, der bestimmende Faktor für die Effizienz eines Branch-and-Bound-Algorithmus sind. Daher ist es wichtig, daß der Zulässigkeitsbereich

$L = \{IP\}$; $S^0 = S$; $\underline{z^0} = -\infty$; $\overline{z_{IP}} = \infty$;

Solange $L \neq \emptyset$

/* Knotenselektion und Relaxation */

Wähle ein Problem $IP^i \in L$ und entferne es aus L;

Löse die Relaxation RP^i;

Sei z_R^i der optimale Zielfunktionswert der Relaxation und x_R^i eine optimale Lösung, falls eine solche existiert.

/* Beschneidung */

Wenn $z_R^i \geq \overline{z_{IP}}$ **Dann**

führe nächste Schleifeniteration durch. Wird die Relaxation mit Hilfe eines dualen Algorithmus gelöst, ist dieser Schritt anwendbar, sobald der duale Zielfunktionswert $\overline{z_{IP}}$ erreicht oder überschritten hat.

Ende Wenn

Wenn $x_R^i \in S^i$ **Dann**

Wenn $c^T x_R^i < \overline{z_{IP}}$ **Dann**

Setze $\overline{z_{IP}} = c^T x_R^i$;

$x^0 = x_R^i$;

Ende Wenn

Entferne aus L alle Probleme mit $\underline{z^i} \geq \overline{z_{IP}}$;

Wenn $c^T x_R^i = \overline{z_{IP}}$ **Dann**

führe nächste Schleifeniteration durch.

Ende Wenn

Ende Wenn

/* Zerlegung */

Sei $\{S^{ij}\}_{j=1}^k$ eine Zerlegung von S^i.

Füge zu L die Probleme $\{IP^{ij}\}_{j=1}^k$ hinzu mit $\underline{z^{ij}} = z_R^i$ für $j = 1, \ldots, k$. Als untere Schranke wird für jedes der neuen Teilprobleme der Zielfunktionswert der Relaxation ihres Elternknotens, also der aktuell betrachteten Relaxation eingesetzt.

Ende Solange

Wenn $\overline{z_{IP}} < \infty$ **Dann**

Die Lösung x^0, die zu $\overline{z_{IP}} = c^T x^0$ geführt hat, ist optimal.

$x^* = x^0$; $z^* = \overline{z_{IP}}$;

Sonst

Es existiert keine zulässige Lösung.

$S = \emptyset$;

Ende Wenn

Algorithmus 4.1: Branch-and-Bound-Algorithmus.

der Relaxation möglichst nahe den Zulässigkeitsbereich des ganzzahligen Problems approximiert, wie bereits in Kapitel 4.2.2 angesprochen wurde. Weitere wichtige Entscheidungen betreffen die Auswahl eines Knotens für das nächste zu betrachtende Teilproblem und die Auswahl der Variable, nach der separiert werden soll. Bisher konnten noch keine zuverlässigen Methoden zur Bestimmung einer geeigneten Separationsvariable entwickelt werden ([NKT89]). Daher ist es üblich, spezielle vom Anwender zu vergebende **Verzweigungsprioritäten** zu verwenden. So sollte z.B. einer Binärvariable, die angibt, ob ein Projekt durchgeführt werden soll oder nicht, eine höhere Priorität zugeordnet werden, als Entscheidungsvariablen zu Detailproblemen innerhalb des Projekts.

Bei der **Knotenselektion** ist eine Liste L der aktiven Teilprobleme, also die Menge der noch nicht beschnittenen Knoten des Entscheidungsbaumes, gegeben, und es stellt sich die Frage, welcher Knoten als nächstes untersucht werden soll. Hierbei kann unterschieden werden zwischen a priori-Regeln, die im Voraus den Aufbau des Baumes vorgeben und Adaptivregeln, die Informationen über den aktuellen Status der aktiven Knoten verwenden. Die wichtigsten a priori-Regeln sind Tiefensuche mit Backtracking sowie Breitensuche. Bei der Tiefensuche wird, wenn der aktuelle Knoten nicht beschnitten wird, als nächstes eines der beiden Kinder besucht. Backtracking bedeutet, daß, wenn ein Knoten beschnitten wird, wir auf dem Pfad von diesem Knoten zur Wurzel so lange nach oben laufen, bis der erste Knoten erreicht wird, der ein noch nicht besuchtes Kind hat. Bei der Breitensuche werden zunächst alle Knoten einer bestimmten Ebene betrachtet, bevor irgendwelche Knoten der nächsten Ebene besucht werden. Aufgrund des hohen Speicherplatzbedarfs zur Verwaltung der Liste mit aktiven Knoten ist diese Strategie für große Probleme oft nicht praktikabel, zumal experimentelle Ergebnisse zeigen, daß zulässige Lösungen wahrscheinlich tiefer im Baum liegen und nicht in Wurzelnähe ([NW88]). Verbreitete Adaptivregeln sind die Regel der besten Schranke (*Best Bound*) oder die Regel der besten Schätzung (*Best Estimate*). Bei *Best Bound* wird der Knoten mit der kleinsten unteren Schranke, d.h. mit dem kleinsten Zielfunktionswert der Relaxation gewählt. Bei *Best Estimate* wird der Knoten gewählt, der am wahrscheinlichsten eine optimale Lösung enthält. Ist $\hat{z}^i \geq \underline{z}^i$ eine Schätzung für z^i_{IP}, dann wird das $i \in L$ gewählt, das $\hat{z}^i$ minimiert.

Branch-and-Bound-Methoden können auch mit anderen Verfahren zum Schrankenberechnung, wie z.B. mit Cutting-Plane Algorithmen und dem Lagrange-Dualen kombiniert werden. Eine genauere Darstellung dieser Verfahren würde jedoch den Rahmen dieser Arbeit sprengen; für tiefergehende Informationen sei auf [NW88], [NKT89] oder [PS82] verwiesen.

Kapitel 5

ILP-Modellierungen

Die zwei im folgenden vorgestellten ILP-Formulierungen beruhen auf Modellierungen, die im Rahmen der Architektursynthese entwickelt wurden. Sie versuchen, durch Umsetzung der im vorigen Kapitel vorgestellten Prinzipien eine geschickte Problembeschreibung zu konstruieren, die auch für größere Probleminstanzen die Ermittlung einer optimalen Lösung ermöglicht. Das Ziel der Architektursynthese besteht darin, für eine gegebene Codesequenz entweder bezüglich einer bestimmten Kostengrenze die schnellste Architektur, oder bezüglich einer vorgegebenen Leistungsanforderung die kostengünstigste Architektur zu realisieren ([GE92]). Diese Problemstellung hat einen engen Bezug zum integrierten Instruktionsanordnungs- und Registerallokationsproblem. Zur korrekten Synthese ist das Instruktionsanordnungsproblem zu berücksichtigen; die Registerallokation[1] ist als Teil einer allgemeineren Ressourcenallokation ebenfalls Gegenstand der Darstellung. Der Unterschied liegt hauptsächlich darin, daß zur Lösung des Instruktionsanordnungsproblems innerhalb eines Übersetzers die Hardware fest vorgegeben ist und die Eingabe so zu transformieren ist, daß diese Ressourcen unter Bewahrung der Semantik des Eingabeprogrammes optimal ausgenutzt werden. Ressourcenallokation und Hardwarekostenminimierung treten somit in den Hintergrund.

Sind keine Ressourcenschranken zu berücksichtigen, so kann lokale Instruktionsanordnung durchgeführt werden, indem auf den Abhängigkeitsgraphen die sogenannte **Methode des kritischen Pfades** (*Critical Path Method*) (CPM) angewendet wird. Dieser Algorithmus berechnet für jeden Knoten des Abhängigkeitsgraphen, also jede Instruktion des Eingabeprogrammes, den frühestmöglichen und den spätestmöglichen Ausführungszeitpunkt. Der frühestmögliche Zeitpunkt wird dabei als *asap*-Kontrollschritt (*as soon as possible*), der spätestmögliche als *alap*-Kontrollschritt (*as late as possible*) bezeichnet. Die Bezeichnung Kontrollschritt wird als Synonym für Taktzyklus verwendet, was bedeutet, daß zwei Instruktionen, deren Ausführungszeit auf denselben Kontrollschritt festgelegt wurde, innerhalb desselben Taktzyklus der zugrundeliegenden Maschine ausgeführt werden. Die *asap*-Werte werden ermittelt, indem für alle Instruktionen die längsten Wege berechnet werden, auf denen sie von einer Quelle des Abhängigkeitsgraphen aus erreicht werden können. Die *alap*-Werte erhält man, indem die längsten Wege von den Senken des Abhängigkeitsgraphen zu allen Knoten berechnet werden und die so ermittelten Werte von einer oberen Schranke für die Ausführungszeit subtrahiert werden. Als obere Schranke wird hierbei die Anzahl der Instruktionen eingesetzt, da der schlimmste Fall die sequentielle Ausführung ist und bei der zugrundeliegenden Architektur jede Instruktion in einem Taktzyklus ausgeführt werden kann. Zur Berechnung der längsten Wege wird der in Anhang B.4 dargestellte **Bellman-Ford Algorithmus** verwendet; damit kann die CPM-Analyse in $\mathcal{O}(n \cdot e)$ Zeit ausgeführt werden, wobei n die Anzahl der Knoten und e die Anzahl der Kanten des Graphen bezeichnet. Die asap- und alap-Werte sind für beide vorgestellten Modellierungen wichtig, da sie es ermöglichen, jeder Instruktion ein Intervall von gültigen Kontrollschritten zuzuordnen,

[1] Registerallokation ist hier nur als Registerzuteilung bzw. Registermengenzuteilung zu verstehen. Die Entscheidung, welche Werte in Register gespeichert werden und ob Zwischenspeicherungen oder Aufteilungen von Lebensspannen erforderlich sind, kann nicht in eine einheitliche ILP-Darstellung integriert werden (vgl. Kapitel 6.7).

was den Zulässigkeitsbereich der zu konstruierenden ILPs verkleinert.

Jede Operation des Eingabeprogrammes kann auf einem bestimmten Ressourcentyp ausgeführt werden. Um die Zuordnung von Instruktionen zu Hardwareressourcen zu beschreiben, wird nachfolgend der sogenannte Ressourcengraph verwendet, der in Anlehnung an [Zha96] definiert ist. Dieser Graph braucht nicht berechnet zu werden, sondern ist statisch durch den Instruktionssatz der Zielmaschine definiert. In der Implementierung wird er nicht verwendet; er dient nur als Hilfsmittel zur Beschreibung der Nebenbedingungen in den vorgestellten ILP-Formulierungen.

Definition 5.1 (Ressourcengraph) *Der* ***Ressourcengraph*** $G_R = (V_R, E_R)$ *ist ein bipartiter gerichteter Graph. Seine Knotenmenge* $V_R = V_D \cup V_K$ *setzt sich zusammen aus den Knoten des Abhängigkeitsgraphen* V_D *und den verschiedenen Ressourcentypen, die durch die Knotenmenge* V_K *repräsentiert werden. Seine Kantenmenge* $E_R \subset V_D \times V_K$ *beschreibt eine mögliche Zuweisung, wobei* $(j, k) \in E_R$ *bedeutet, daß Instruktion* $j \in V_D$ *auf Ressourcen vom Typ* k *ausgeführt werden kann.*

Alle im Rahmen der vorgestellten Modellierungen verwendeten Variablen werden als nichtnegativ vorausgesetzt; diese Nichtnegativitätsbedingungen werden jedoch nicht explizit aufgeführt.

5.1 SILP-Modellierung

Die in diesem Abschnitt beschriebene ILP-Formulierung wurde in [Zha96] unter der Bezeichnung **SILP** (*Scheduling and Allocating with Integer Linear Programming*) vorgestellt. Zunächst werden die im Rahmen der SILP-Modellierung wichtigen Größen und Bezeichnungen aufgelistet, um einen Überblick über die verwendete Terminologie zu geben:

- Die Variable t_i gibt die relative Position einer Mikrooperation i innerhalb der Instruktionen der optimierten Codesequenz an; die t_i-Werte müssen ganzzahlig sein. Bei linearem Programmfluß ist die Zuordnung einer Mikrooperation zur c-ten Instruktion des Maschinenprogrammes gleichbedeutend mit der Zuordnung des Startzeitpunktes der Ausführung dieser Mikrooperation zu Taktzyklus (Kontrollschritt) c (vgl. Kapitel 2.9). Bei nicht-linearem Programmfluß (vgl. Kapitel 2.10) können auch Programmteile, die sich gegenseitig ausschließen, hintereinander in ein Maschinenprogramm angeordnet werden. Die Interpretation von t_i als Startzeitpunkt der Mikrooperation i ist in diesem Fall problematisch. Dennoch ist das Minimieren der gesamten "Ausführungszeit", also des maximalen t_i-Wertes auch bei nichtlinearem Programmfluß korrekt. Jede Instruktion des Ausgabeprogrammes benötigt genau einen Taktzyklus zur Ausführung; die Anzahl der Instruktionen gibt somit die Anzahl der Taktzyklen wieder, die zur Ausführung aller Instruktionen erforderlich sind. Ist es gelungen, mehrere Mikrooperationen zu einer VLIW-artigen Instruktion zu zusammenzufassen, so wurde dadurch die Ausführungszeit reduziert – auch wenn sich diese Optimierung nur auf einem einzigen Kontrollpfad auswirkt.

- Q_j^k beschreibt die Ausführungszeit der Instruktion $j \in V_D$ bei Ausführung auf Ressourcentyp k.
- w_j beschreibt die tatsächliche Ausführungszeit der Instruktion $j \in V_D$; der Unterschied zu Q_j^k liegt darin, daß hierbei die Möglichkeit der Ausführung auf verschiedenen Ressourcentypen berücksichtigt wird.
- L_{ij}^k repräsentiert die Latenz der funktionalen Einheit $k \in V_K$, also das minimale Zeitintervall zwischen zwei aufeinanderfolgenden Benutzungen i und j.
- Die Anzahl der zur Verfügung stehenden Ressourcen vom Typ $k \in V_K$ beträgt R_k.
- τ_j beschreibt die Lebenszeit einer Variablen, die in Operation j erzeugt wird.

5.1.1 Basisformulierung

Als Ausgangspunkt der ILP-Erstellung wird der sogenannte Ressourcenflußgraph G_F verwendet. Er beschreibt die Programmausführung als einen Fluß der vorhandenen Hardwareressourcen durch die Instruktionen des Programmes; für jeden Ressourcentyp entsteht dabei ein gesondertes Flußnetzwerk. Ein Ressourcentyp $k \in V_K$ wird durch zwei Knoten k_Q und k_S in G_F repräsentiert, wobei k_Q als Quelle, k_S als Senke in dem zu definierenden Flußnetzwerk fungiert. Von k_Q aus wird der ersten Instruktion, die auf Ressourcentyp k ausgeführt werden soll, eine Ressource k_r dieses Typs übergeben; diese Instruktion reicht k_r nach vollständiger Ausführung an die nächste Instruktion weiter, die denselben Ressourcentyp verwendet. Die letzte Instruktion, die eine bestimmte Instanz eines Ressourcentyps k verwendet, gibt diese an den Knoten k_S zurück. Die Anzahl der gleichzeitig "in Umlauf befindlichen" Ressourcen (also Instanzen eines Ressourcentyps) darf die Anzahl der vorhandenen Ressourcen dieses Typs nie überschreiten.

Definition 5.2 (Ressourcenflußgraph) *Der **Ressourcenflußgraph** G_F ist ein gerichteter Graph $G_F = (V_F, E_F)$ mit*

$$V_F = \bigcup_{k \in V_K} V_F^k \quad \textit{und} \quad E_F = \bigcup_{k \in V_K} E_F^k$$

Dabei ist

$$V_F^k = V_D^k \cup \{k_Q, k_S\} = \{u \in V_D \;|(u,k) \in E_R\} \cup \{k_Q, k_S\}$$

und

$$\begin{aligned} E_F^k &= \{(i,j)\ |i,j \in V_D^k\ \wedge j \not\prec i\ \wedge i \neq j\} \\ &\cup\ \{(k_Q,j)\ |(k,j) \in E_R\ \} \\ &\cup\ \{(j,k_S)\ |(k,j) \in E_R\ \} \end{aligned}$$

Jeder Kante $(i,j) \in E_F^k$ wird eine Flußvariable $x_{ij}^k \in \{0,1\}$ zugeordnet. Eine Hardwarekomponente k wird genau dann über die Kante (i,j) von Knoten i nach Knoten j befördert, wenn $x_{ij}^k = 1$ gilt.

V_D^k ist die Menge aller Knoten des Abhängigkeitsgraphen, die zu Instruktionen gehören, welche auf Ressourcentyp k ausgeführt werden können. Jede Kante $(\mu,\nu) \in E_F^k$ bezeichnet einen möglichen Fluß der Ressourcen des Typs $k \in V_K$ von μ nach ν. Sind μ und ν beides Knoten des Abhängigkeitsgraphen, dann wird eine Kante (μ,ν) nur in die Menge E_F^k aufgenommen, wenn sie nicht gegen eine Vorrangrelation $\nu \prec \mu$ verstößt. Außerdem brauchen Operationsknoten, die von anderen Operationen im Sinne der Vorrangrelation abhängig sind, nicht durch eine eingehende Kante mit den Ressourcenknoten verbunden zu werden, wenn nur eine Instanz dieser Ressource existiert. Dasselbe gilt für Operationen, die niemals als letzte einen Ressourcentyp verwenden können. Die Differenzierung zwischen Quellen- und Senkenknoten eines Ressourcentyps (k_Q bzw. k_S) braucht bei der Benennung der Entscheidungsvariablen nicht berücksichtigt zu werden. Die Variable x_{kj}^k ist eindeutig der Kante (k_Q,j) zugeordnet, während x_{ik}^k der Kante (i,k_S) entspricht. Der Grund hierfür liegt einfach darin, daß die Quelle niemals als Zielknoten, die Senke niemals als Startknoten einer Kante auftreten kann, so daß die Bezeichnung eindeutig ist.

Beispiel 5.1 Alle vorgestellten Nebenbedingungen werden konkret für die folgende Codesequenz angegeben, um das Verständnis der Formulierungen zu erleichtern.

```
1)   r1 = dm(i0, m0);
2)   r2 = pm(i8, m8);
3)   r3 = r1 * r2;
4)   MRF = MRF + r1 * r2;
5)   r4 = dm(i1, m1);
6)   r5 = pm(i9, m9);
7)   r6 = r4 + r5;
8)   r7 = min(r4, r5);
```

Es seien vier Ressourcenknoten gegeben: MU zur Modellierung von Multipliziererinstruktionen, S zur Modellierung von ALU-Instruktionen sowie DM bzw. PM für Zugriffe auf den Daten- bzw. Programmspeicher. Dann ergibt sich für obige Codesequenz der in Abbildung 5.1 dargestellte Ressourcenflußgraph.

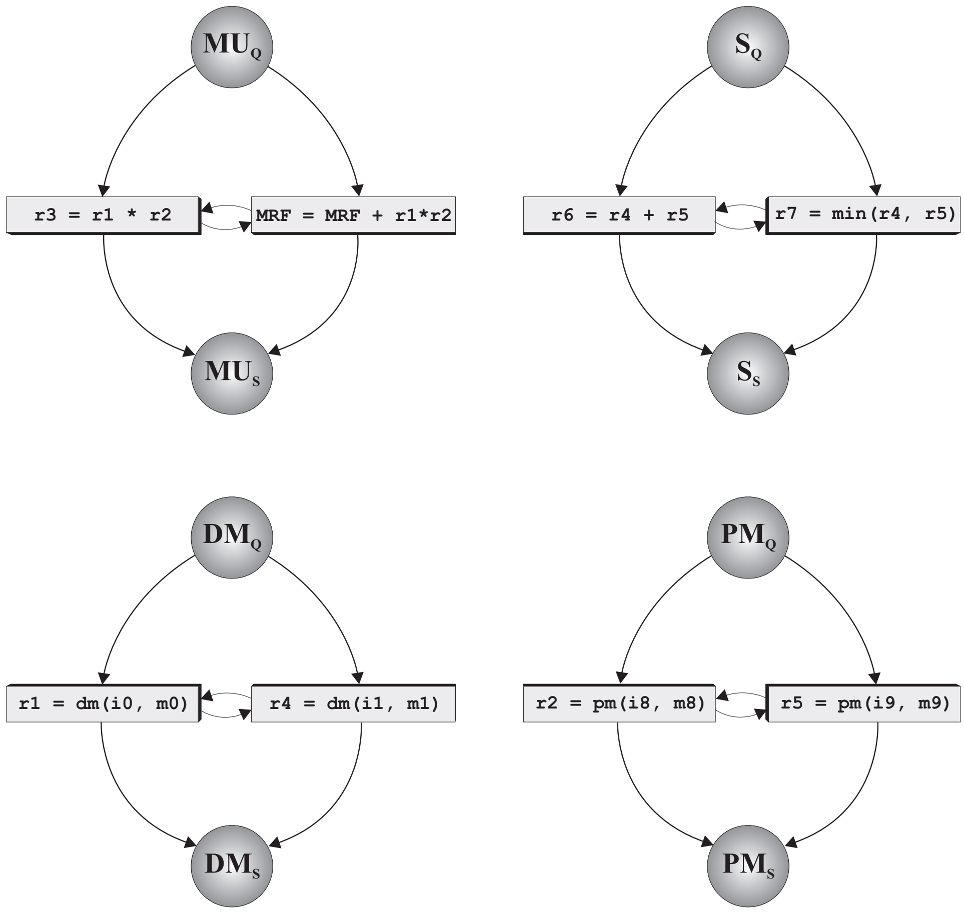


Abbildung 5.1: Der Ressourcenflußgraph zu Beispiel 5.1.

□

Der Ressourcenflußgraph kann algebraisch mit Hilfe von Flußgleichungen dargestellt werden.

$$\Phi_j^k - \Psi_j^k = 0 \qquad \forall\, j \in V_D,\ \forall\, k \in V_k :\ (j,k) \in E_R \tag{5.1}$$

$$\sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \Phi_j^k = 1 \qquad \forall\, j \in V_D \tag{5.2}$$

Dabei stellt Φ_j^k den Fluß dar, der in einen Knoten $j \in V_D$ eintritt und Ψ_j^k den Fluß, der den Knoten j verläßt. Ihre exakte Definition lautet

$$\Phi_j^k = \sum_{(i,j) \in E_F^k} x_{ij}^k; \qquad \Psi_j^k = \sum_{(j,i) \in E_F^k} x_{ji}^k \tag{5.3}$$

Die Ungleichung 5.2 garantiert dabei, daß jede Operation auf genau einer Hardwarekomponente ausgeführt wird.

Beispiel 5.2 Für das in Beispiel 5.1 angegebene Eingabeprogramm sind zur Modellierung der Flußerhaltung folgende Nebenbedingungen erforderlich:

$$\begin{aligned}
x_{51}^{DM} + x_{DM1}^{DM} - x_{15}^{DM} - x_{1DM}^{DM} &= 0 \\
x_{62}^{PM} + x_{PM2}^{PM} - x_{26}^{PM} - x_{2PM}^{PM} &= 0 \\
x_{43}^{MU} + x_{MU3}^{MU} - x_{34}^{MU} - x_{3MU}^{MU} &= 0 \\
x_{34}^{MU} + x_{MU4}^{MU} - x_{43}^{MU} - x_{4MU}^{MU} &= 0 \\
x_{15}^{DM} + x_{DM5}^{DM} - x_{51}^{DM} - x_{5DM}^{DM} &= 0 \\
x_{26}^{PM} + x_{PM6}^{PM} - x_{62}^{PM} - x_{6PM}^{PM} &= 0 \\
x_{87}^{S} + x_{S7}^{S} - x_{78}^{S} - x_{7S}^{S} &= 0 \\
x_{78}^{S} + x_{S8}^{S} - x_{87}^{S} - x_{8S}^{S} &= 0
\end{aligned}$$

Die Nebenbedingungen zur Realisierung der Ausführungsbeschränkungen lauten wie folgt:

$$\begin{aligned}
x_{51}^{DM} + x_{DM1}^{DM} &= 1 \\
x_{62}^{PM} + x_{PM2}^{PM} &= 1 \\
x_{43}^{MU} + x_{MU3}^{MU} &= 1 \\
x_{34}^{MU} + x_{MU4}^{MU} &= 1 \\
x_{15}^{DM} + x_{DM5}^{DM} &= 1 \\
x_{26}^{PM} + x_{PM6}^{PM} &= 1 \\
x_{87}^{S} + x_{S7}^{S} &= 1 \\
x_{78}^{S} + x_{S8}^{S} &= 1
\end{aligned}$$

□

Wird eine Operation von einer Ressource des Typs k ausgeführt, dann muß eine der Instanzen des Typs k diese Operation über eine eingehende Flußkante erreichen. Die Ausführungszeit w_j einer Operation j kann also berechnet werden durch

$$w_j = \sum_{k \in V_K} \sum_{(i,j) \in E_F^k} Q_j^k \cdot x_{ij}^k \qquad \forall j \in V_D \tag{5.4}$$

Wenn eine Hardwarekomponente des Typs k eine Operation j ausführt, können neue Operationen erst von dieser Komponente ausgeführt werden, wenn sie bereit ist, neue Daten zu empfangen. Die *busy-time* der Komponente für eine Operation j ist dann definiert als

$$z_j = \sum_{k \in V_K} \sum_{(i,j) \in E_F^k} L_{ij}^k \cdot x_{ij}^k \qquad \forall j \in V_D \tag{5.5}$$

Die Anzahl der Operationen, die gleichzeitig auf Ressourcen vom Typ k ausgeführt werden, darf nie die Anzahl der verfügbaren Instanzen dieses Ressourcentyps übersteigen. Dies kann im zugrundeliegenden Modell durch Beschränkung des Ausgangsgrades jedes Ressourcenknotens $k \in V_k$ erreicht werden.

$$\sum_{(k,j) \in E_F^k} x_{kj}^k \leq R_k \qquad \forall k \in V_K \tag{5.6}$$

Beispiel 5.3 Für die in Beispiel 5.1 angegebene Codesequenz sind somit weitere Nebenbedingungen erforderlich:

$$\begin{aligned} x_{DM1}^{DM} + x_{DM5}^{DM} &\leq 1 \\ x_{PM2}^{PM} + x_{PM6}^{PM} &\leq 1 \\ x_{MU3}^{MU} + x_{MU4}^{MU} &\leq 1 \\ x_{S7}^{S} + x_{S8}^{S} &\leq 1 \end{aligned}$$

□

Desweiteren müssen die durch den Abhängigkeitsgraphen gegebenen Vorrangbeziehungen modelliert werden. Die zu erstellenden Ungleichungen hängen dabei vom Typ der Abhängigkeiten ab. Besteht zwischen den Instruktionen i und j eine Setzungs-Setzungs- oder eine Setzungs- Benutzungsabhängigkeit, ist die Ungleichung

$$t_j - t_i \geq w_i \tag{5.7}$$

zu erzeugen. Liegt hingegen eine Benutzungs-Setzungs-Abhängigkeit vor, so genügt es, zu gewährleisten, daß niemals Instruktion j vor i ausgeführt wird.

$$t_j - t_i \geq 0 \tag{5.8}$$

Beispiel 5.4 Wir betrachten wieder unser Beispielprogramm

```
1)   r1 = dm(i0, m0);
2)   r2 = pm(i8, m8);
3)   r3 = r1 * r2;
4)   MRF = MRF + r1 * r2;
5)   r4 = dm(i1, m1);
6)   r5 = pm(i9, m9);
7)   r6 = r4 + r5;
8)   r7 = min(r4, r5);
```

Gehen wir davon aus, daß die Ausführung jeder Instruktion genau einen Taktzyklus benötigt, ergeben sich die folgenden Nebenbedingungen zur Darstellung der Datenabhängigkeiten:

$$\begin{aligned}
t_3 - t_1 &\geq 1 \\
t_4 - t_1 &\geq 1 \\
t_3 - t_2 &\geq 1 \\
t_4 - t_2 &\geq 1 \\
t_7 - t_5 &\geq 1 \\
t_8 - t_5 &\geq 1 \\
t_7 - t_6 &\geq 1 \\
t_8 - t_6 &\geq 1
\end{aligned}$$

□

Das lineare Programm zur Lösung des Problems der Instruktionsanordnung unter Berücksichtigung von Ressourcenbeschränkungen kann allgemein folgendermaßen formuliert werden:

- Zielfunktion

$$\min \quad M_{steps} \tag{5.9}$$

- Nebenbedingungen

  1. Zeitschranken
     Bei keiner Instruktion darf die Startzeit das zu berechnende Maximum M_{steps} an Kontrollschritten überschreiten.

$$t_j \leq M_{steps} \qquad \forall\ j \in V_D \tag{5.10}$$

  2. Vorrangbedingungen
     Ist Instruktion j von Instruktion i abhängig, dann darf j erst nach der Ausführung von i ausgeführt werden.

$$\begin{array}{rcll} t_j - t_i & \geq & w_i & \forall\ (i,j) \in E_D^{output} \ \cup\ E_D^{true} \\ t_j - t_i & \geq & 0 & \forall\ (i,j) \in E_D^{anti} \end{array} \tag{5.11}$$

  3. Flußerhaltung
     Der einen Knoten betretende positive Nettofluß muß denselben Wert haben wie der positive Nettofluß, der den Knoten verläßt.

$$\Phi_j^k - \Psi_j^k \ = \ 0 \qquad \forall\ j \in V_D,\ \forall\ k \in V_k :\ (j,k) \in E_R \tag{5.12}$$

  4. Ausführungsbeschränkung
     Jede Operation darf genau einmal auf genau einer Hardwarekomponente ausgeführt werden.

$$\sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \Phi_j^k \ = \ 1 \qquad \forall\ j \in V_D \tag{5.13}$$

  5. Ressourcenschranken
     Die Anzahl der vorhandenen Instanzen aller Ressourcentypen darf nicht überschritten werden.

$$\sum_{(k,j) \in E_F^k} x_{kj}^k \leq R_k \qquad \forall k \in V_K \tag{5.14}$$

  6. Serialisierungsbedingungen
     Sind Operationen i und j demselben Ressourcentyp k zugeordnet, dann muß j auf die Ausführung von i warten, wenn tatsächlich eine Komponente des Typs k über die Kante $(i,j) \in E_F^k$ befördert wird, also wenn $x_{ij}^k = 1$.

$$t_j - t_i \geq z_i + (\sum_{\substack{k \in V_K: \\ (i,j) \in E_F^k}} x_{ij}^k - 1) \cdot \alpha_{ij} \qquad \forall (i,j) \in E_F^k \tag{5.15}$$

     Dabei sind die Werte α_{ij} ausreichend groß zu wählen (s. unten).

Je besser das Zulässigkeitspolytop der Relaxation P_F den Zulässigkeitsbereich des ganzzahligen Problems P_I approximiert, desto effizienter kann das ganzzahlige lineare Programm gelöst werden. Aufgrund der Struktur der Serialisierungsbedingungen entsteht jedoch eine Abweichung zwischen P_F und P_I, die die Lösungseffizienz stark beeinträchtigt. Es gilt nun, durch geschickte Wahl des Parameters α_{ij} diese Abweichung möglichst gering zu halten. Das straffste Lösungspolytop wird unter Betrachtung einer Serialisierungsbedingung der Form $t_j - t_i \geq z_i + (x_{ij} - 1)\alpha_{ij}$ mit folgendem α-Wert erreicht: $\alpha_{ij} = z_i - asap(j) + alap(i)$. Der Beweis hierzu findet sich in [Zha96].

Die Serialisierungsbedingungen haben neben ihrer Sequentialisierungswirkung noch eine weitere Funktion: durch sie wird garantiert, daß die Lösung des ILP's die gewünschten Flüsse im jeweiligen Flußnetzwerk impliziert. Existiert z.B. genau eine Instanz eines Ressourcentyps k ($R_K = 1$) und werden auf dieser Ressource sechs Instruktionen ausgeführt, dann sollte daraus ein Fluß entstehen, wie in Abbildung 5.2 dargestellt. Die Formulierung

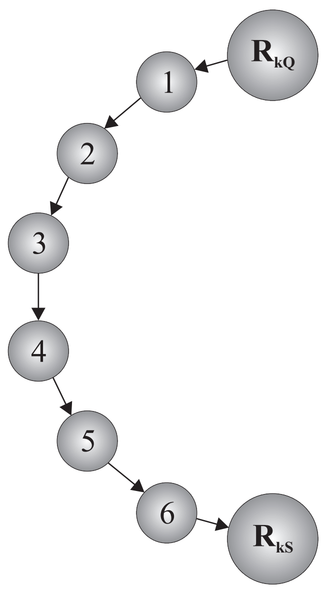


Abbildung 5.2: Erwünschter Flußgraph.

des ILP ohne Serialisierungsbedingungen würde aber auch die in Abbildung 5.3 dargestellte "Lösung" zulassen. Bei Hinzunahme der Serialisierungsbedingungen wird durch Abbildung 5.3 jedoch folgende Ordnung zwischen den Instruktionen impliziert: $t_4 < t_5 < t_6 < t_4$. Dies stellt also keine zulässige Lösung dar. In einer zulässigen Lösung verlaufen alle Flüsse korrekt zwischen den Ressourcenknoten. Durch diese Flüsse entsteht kein Widerspruch, da den Ressourcenknoten keine Startzeitpunkte von Instruktionen zugeordnet werden. Die Berechnungsabläufe im Eingabeprogramm werden durch die Serialisierungsbeziehungen somit korrekt modelliert.

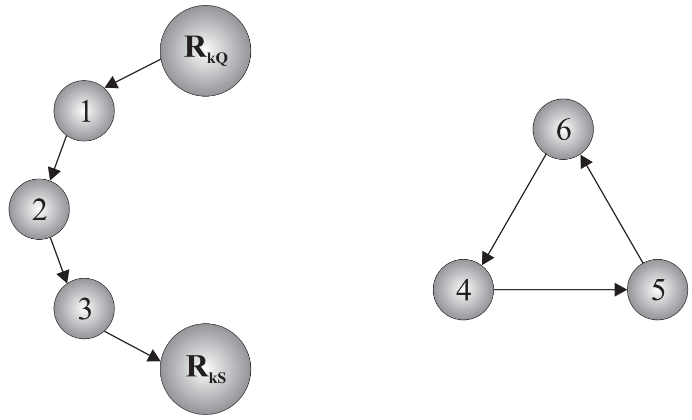


Abbildung 5.3: Fehlerhafter Flußgraph.

5.1.2 Integration der Registerallokation

In diesem Abschnitt wird die oben beschriebene ILP-Formulierung um Aspekte der Registerallokation erweitert. Die Lebenszeit einer Variable, die von Operation j produziert wird, wird im folgenden als τ_j bezeichnet. Bei der Lösung des Registerallokationsproblems werden traditionellerweise zwei verschiedene Darstellungen verwendet. Bei Verwendung des Kompatibilitätsgraphen wird jedes τ_j auf einen Knoten abgebildet; Kanten sind zwischen allen Paaren von Knoten einzufügen, deren Lebenszeiten disjunkt sind. Der Kollisionsgraph verwendet dieselbe Knotendefinition wie der Kompatibilitätsgraph; seine Kanten verlaufen jedoch zwischen allen Paaren von Operationen, deren Lebenszeiten sich überlappen (vgl. Kapitel 3.3).

Im Unterschied dazu wird beim zugrundeliegenden Ansatz das Registerallokationsproblem als Registerverteilungsproblem formuliert, indem wieder das Konzept des Ressourcenflußgraphen aufgegriffen wird.

Definition 5.3 (Registerflußgraph) *Der Registerflußgraph $G_F^g = (V_F^g, E_F^g)$ ist ein gerichteter Graph mit Knotenmenge $V_F^g = V_g \cup G$ und einer Menge von gerichteten Kanten E_F^g. Die Menge G enthält einen Ressourcenknoten g, der die vorhandene Registermenge darstellt, $G = \{g\}$. Ein Knoten $j \in V_g$ repräsentiert eine Operation, die einen Schreibzugriff auf ein Register ausführt und damit eine Variable erzeugt, der die Lebenszeit τ_j zugeordnet wird. Jede Kante $(i, j) \in E_F^g$ symbolisiert nun einen möglichen Fluß eines Registers von i nach j und ist einer Flußvariable $x_{ij}^g \in \{0, 1\}$ zugeordnet. Dann wird dasselbe Register genau dann zur Speicherung der in Knoten i und j erzeugten Variablen verwendet, wenn $x_{ij}^g = 1$.*

Lebenszeiten von Variablen werden stets durch Setzungs-Benutzungs-Abhängigkeiten definiert. Wird in einer Instruktion i ein Register beschrieben, das in den Instruktionen

$j_1, \ldots, j_m$ verwendet wird, so reicht die Lebensspanne des von i beschriebenen Registers bis zum Zeitpunkt der letzten Verwendung. Um dies zu modellieren, wird die Formulierung der Setzungs-Benutzungs-Abhängigkeiten verändert. Mittels einer Hilfsvariable $b_{ij} \geq 0$ wird der Abstand zwischen der definierenden Instruktion i und der verwendenden Instruktion j gemessen. Um dabei die Vorrangrelation nicht zu verletzen, wird für alle Setzungs-Benutzungs-Abhängigkeiten folgende Gleichung erzeugt:

$$t_j - t_i - b_{ij} = w_i$$

Für die Lebenszeit des Registers, das von Instruktion i beschrieben wird, gilt dann

$$\tau_i \geq b_{ij} + w_i \qquad \forall\ (i,j) \in E_D^{true}$$

Eine Instruktion j darf nur dann dasselbe Register beschreiben wie eine vorangehende Instruktion i, wenn j zu einem Zeitpunkt ausgeführt wird, zu dem die Lebenszeit von i, τ_i, bereits beendet ist. Mit anderen Worten: Hat die Variable, die von Instruktion i produziert wird, die Lebenszeit τ_i, und soll die Ausgabe von Instruktion j in dasselbe Register geschrieben werden, wie die Ausgabe von Instruktion i (gilt also $x_{ij}^g = 1$), dann muß gelten $t_j - t_i \geq \tau_i$. Dieser Sachverhalt läßt sich mittels der folgenden Registerserialisierungsbedingung formulieren:

$$t_j - t_i \geq w_i - w_j + \tau_i + (x_{ij}^g - 1) \cdot 2T \tag{5.16}$$

Dabei ist T die Anzahl der Instruktionen des Eingabeprogrammes, die sicherlich eine obere Schranke für die maximal mögliche Lebenszeit darstellt. Verwenden t_i und t_j nicht dasselbe Register, so gilt $x_{ij}^g = 0$, und Gleichung 5.16 ergibt sich zu $t_j - t_i \geq \beta$ mit $\beta \leq -T$, was stets erfüllbar ist.

Beispiel 5.5 Es seien die beiden folgenden Mikrooperationen gegeben:

```
1)          r1 = 5;
2)          r1 = r1 + 4;
```

Werden diesen Instruktionen die Zeitpunkte $t_1 = 1$ und $t_2 = 2$ als Startzeitpunkte zugeordnet, so können beide Instruktionen dasselbe Zielregister verwenden. Die Lebenszeit des in Instruktion 1 beschriebenen Registers wird korrekt zu $\tau_1 = 1$ berechnet. Durch die Belegung $x_{12}^g = 1$ wird also die Lösbarkeit des Problems nicht beeinträchtigt.

$$\begin{array}{rclcrcl} t_2 - t_1 - b_{12} & = & 1 & \Leftrightarrow & 2 - 1 - b_{12} & = & 1 \Rightarrow b_{12} = 0 \\ \tau_1 & \geq & b_{12} + w_1 & \Leftrightarrow & \tau_1 & \geq & 1 \\ t_2 - t_1 & \geq & \tau_1 + (x_{12}^g - 1) \cdot 2T & \Leftrightarrow & t_2 - t_1 & \geq & 1 \end{array}$$

□

Schließlich müssen zur Modellierung des Registerflußgraphen, ebenso wie bei den übrigen Ressourcenflußgraphen Flußerhaltungsbedingungen, Ressourcenschranken und Ausführungsbeschränkungen formuliert werden. Damit ergibt sich für das lineare Programm zur simultanen Lösung des Problems der Instruktionsanordnung und der Registerallokation im allgemeinen Fall die folgende Gestalt:

$$\min \quad M_{steps} \tag{5.17}$$

unter Berücksichtigung folgender Nebenbedingungen:

$$\begin{aligned}
t_j &\leq M_{steps} && \forall\ j \in V_D && (5.18)\\
t_j - t_i - b_{ij} &= w_i && \forall\ (i,j) \in E_D^{true} && (5.19)\\
t_j - t_i &\geq w_i && \forall\ (i,j) \in E_D^{output} && (5.20)\\
t_j - t_i &\geq 0 && \forall\ (i,j) \in E_D^{anti} && (5.21)\\
\Phi_j^k - \Psi_j^k &= 0 && \forall\ j \in V_D,\ \forall\ k \in V_k:\ (j,k) \in E_R && (5.22)\\
\sum_{\substack{k \in V_K:\\ (j,k) \in E_R}} \Phi_j^k &= 1 && \forall\ j \in V_D && (5.23)\\
\sum_{(k,j) \in E_F^k} x_{kj}^k &\leq R_k && \forall\ k \in V_K && (5.24)\\
t_j - t_i &\geq z_i + \Big(\sum_{\substack{k \in V_K:\\ (i,j) \in E_F^k}} x_{ij}^k - 1\Big) \cdot \alpha_{ij} && \forall (i,j) \in E_F^k && (5.25)\\
\tau_i &\geq b_{ij} + w_i && \forall\ j \in V_D, \forall\ (i,j) \in E_D^{true} && (5.26)\\
\Psi_g^g &\leq R_g && && (5.27)\\
\Phi_j^g &= 1 && \forall\ j \in V_g && (5.28)\\
\Phi_j^g - \Psi_j^g &= 0 && \forall\ j \in V_F^g && (5.29)\\
t_j - t_i &\geq w_i - w_j + \tau_i + (x_{ij}^g - 1) \cdot 2T && \forall (i,j) \in E_F^g && (5.30)
\end{aligned}$$

Wie sich später zeigen wird, ist diese Formulierung noch gewissen Beschränkungen unterworfen. Voraussetzung für das korrekte Funktionieren ist, daß der Abstand zwischen zwei Instruktionen im Maschinenprogramm tatsächlich einer möglichen Lebensspanne entspricht. Für einen linearen Programmfluß ist diese Voraussetzung erfüllt, in Gegenwart von bedingten Anweisungen gilt sie jedoch nicht mehr. Ziel der Instruktionsanordnung ist die möglichst effiziente Anordnung der Instruktionen auf allen Kontrollpfaden, d.h. es müssen auch Operationen aus sich gegenseitig ausschließenden Kontrollwegen im Ausgabecode eingeordnet werden. Werden zum Beispiel die Instruktionen des *then*-Teiles einer bedingten Anweisung vor denen des *else*-Teiles plaziert und wird im *then*-Teil ein Register beschrieben, das auch im *else*-Teil verwendet wird, so ist hierdurch **keine** Lebenszeit definiert. Sollen also Eingabeprogramme mit Schleifen oder bedingten Anweisungen betrachtet werden, sind weitere Nebenbedingungen erforderlich. Eine genauere Analyse dieses Sachverhaltes findet sich in Kapitel 6.4 und 6.5.

5.1.3 Komplexitätsanalyse

Um ein Maß für die Komplexität der Modellierungen zu erhalten, werden die Anzahl der Nebenbedingungen und die Anzahl der Binärvariablen untersucht (vgl [NW88]).

Die Anzahl der Nebenbedingungen für die Zeitschranken (5.10) beträgt $|V_D|$; zur Darstellung der Vorrangbedingungen (5.11) sind $|E_D| = \mathcal{O}(|V_D|^2)$ Nebenbedingungen erforderlich. Dabei stimmt $|V_D|$, die Anzahl der Knoten des Abhängigkeitsgraphen, mit der Anzahl der Instruktionen des Eingabeprogrammes überein. Die Modellierung der Flußerhaltung (5.12) erfordert $|V_D| \cdot |V_k| = \mathcal{O}(|V_D|)$ Nebenbedingungen, die Ausführungsbeschränkungen (5.13) werden durch $|V_D|$ Nebenbedingungen erfaßt. Um zu gewährleisten, daß die Anzahl der vorhandenen Instanzen aller Ressourcentypen nicht überschritten werden (5.14), ist für jeden Ressourcentyp eine Nebenbedingung zu formulieren; die Anzahl dieser Nebenbedingungen ist also konstant. Die Anzahl der aufzustellenden Serialisierungsbedingungen beträgt

$$N_s = |\bigcup_k \{(i,j)^k | i,j \in V_D^k \wedge j \not< i \wedge i \neq j\}| = \mathcal{O}(|V_D|^2)$$

Beim ADSP-2106x ist für alle Instruktionen i die Ausführungszeit w_i mindestens so groß, wie die Latenz z_i der ausführenden funktionalen Einheit. Daher brauchen keine Serialisierungsbedingungen zwischen zwei Instruktionen i und j formuliert zu werden, wenn aufgrund von Datenabhängigkeiten j stets nach i ausgeführt werden muß. Dies läßt sich mit Hilfe der transitiven Hülle des Datenabhängigkeitsgraphen (vgl. Anhang B.5) feststellen. Die Anzahl der erforderlichen Serialisierungsbedingungen wird in der Praxis also deutlich kleiner als $|V_D|^2$ sein.

Insgesamt sind $\mathcal{O}(|V_D|^2)$ Nebenbedingungen erforderlich; diese Schranke gilt offensichtlich auch bei Berücksichtigung der Registerallokation.

Eine obere Schranke für die Anzahl der Variablen ist gegeben durch $\mathcal{O}(|V_D|^2)$; es werden $|V_D|$ Variablen zur Modellierung der Startzeitpunkte, sowie $\mathcal{O}(|V_D|^2)$ Flußvariablen benötigt. Es ist jedoch nicht erforderlich, die Variablen t_j als ganzzahlig zu deklarieren und nicht alle Flußvariablen müssen als binär spezifiziert werden. Die Koeffizientenmatrizen der Gleichungssysteme aus (5.22) und (5.23) bzw. aus (5.28) und (5.29) sind total unimodular[2] ([Zha96]). Die Lösung eines linearen Programmes, dessen Koeffizientenmatrix total unimodular ist, führt stets zu einem ganzzahligen Optimum ([PS82], [NW88]). Es ist daher nur erforderlich, diejenigen Flußvariablen als binär zu spezifizieren, die innerhalb der Serialisierungsbedingungen verwendet werden. Der Beweis hierzu findet sich in [Zha96]. Dasselbe gilt auch für die Kanten des Registerflußgraphen; auch hier brauchen nur die in den Registerserialisierungsbedingungen verwendeten Variablen als Binärvariablen deklariert zu werden.

[2]Eine quadratische, ganzzahlige Matrix B heißt unimodular (UM), wenn für ihre Determinante gilt: $det(b) = \pm 1$. Eine ganzzahlige Matrix M heißt total unimodular (TUM), wenn jede quadratische, reguläre Teilmatrix von M unimodular ist.

Die Anzahl der zu erzeugenden Flußvariablen ist nach oben durch $\mathcal{O}(|V_D|^2)$ beschränkt; dies ist auch eine obere Schranke für die Anzahl der als binär zu spezifizierenden Flußvariablen. Die tatsächliche Anzahl der Binärvariablen wird meist deutlich geringer sein; dies hängt jedoch von der Struktur des Eingabeprogrammes ab.

5.1.4 Effizienzsteigerung

Ermittlung überflüssiger Flußvariablen und Serialisierungsbedingungen

Aufgrund der zuvor durchgeführten Bestimmung der frühest- und spätestmöglichen Beginnzeitpunkte aller Instruktionen kann die Anzahl der Kanten des Ressourcenflußgraphen und damit der Flußvariablen, sowie der Serialisierungsbedingungen reduziert werden.

- Eine Flußvariable x_{ij} braucht nur erzeugt zu werden, wenn $z_i \leq alap(j) - asap(i)$ gilt. Gilt $z_i > alap(j) - asap(i) \Leftrightarrow z_i + asap(i) > alap(j)$, so wird Instruktion i stets nach Instruktion j ausgeführt und kann somit niemals eine Ressource an j weiterreichen.
- Eine Serialisierungsbedingung $t_j - t_i \geq z_i + (x_{ij} - 1)\alpha_{ij}$ braucht nur formuliert zu werden, wenn $z_i \geq asap(j) - alap(i) \Leftrightarrow z_i + alap(i) \geq asap(j)$. Andernfalls wird Instruktion j ohnehin nach Instruktion i ausgeführt, so daß die Sequentialisierung von i und j nicht explizit erzwungen werden muß.

Erzeugung gültiger linearer Ungleichungen

Durch Hinzufügen der in diesem Abschnitt vorgestellten Ungleichungen wird die Effizienz der ILP-Formulierung gesteigert. Diese Verbesserung resultiert daher, daß der Zulässigkeitsbereich P_F der Relaxation näher an das Lösungspolytop des ganzzahligen Problems angenähert wird (vgl. Kapitel 4; vgl. [Zha96]).

- Es muß stets gelten
$$x_{ij}^k + x_{ji}^k \leq 1 \qquad \forall\ (i,j)^k\ \in E_F^k$$
Bei einem Verstoß gegen diese Ungleichung müßte $x_{ij}^k = x_{ji}^k = 1$ gelten. Dann wäre kein korrekter Flußgraph mehr möglich, da für alle Instruktionen sowohl der eintretende, als auch der herausfließende Fluß den Wert 1 haben muß. Durch die Kanten x_{ij}^k und x_{ji}^k entstünde somit ein eigenständiger Fluß, der von keinem Ressourcenknoten aus erreicht werden kann. Dieser Sachverhalt wird in Abbildung 5.4 veranschaulicht. Von den anderen Variablen, die einer ein- oder ausgehenden Flußkante entsprechen, kann keine den Wert 1 annehmen.

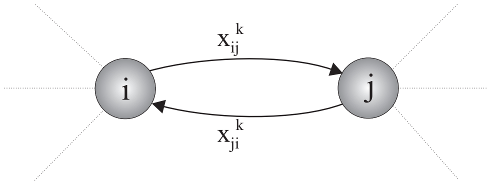


Abbildung 5.4: Gültige Ungleichung: $x_{ij}^k + x_{ji}^k \leq 1$.

- Desweiteren muß stets gelten

$$x_{jm}^k + x_{nj}^k \leq 1 \qquad \forall\ (m,n) \in E_D\ \wedge\ (j,m),(n,j) \in E_F^k$$

Dies läßt sich anhand von Abbildung 5.5 veranschaulichen. Gilt sowohl $x_{jm}^k = 1$ als auch $x_{nj}^k = 1$, bedeutet das, daß Instruktion n vor Instruktion m ausgeführt werden muß. Somit entsteht ein Widerspruch zu der Datenabhängigkeit $(m,n) \in E_D$, die in der Abbildung durch die fett gedruckte Kante dargestellt ist.

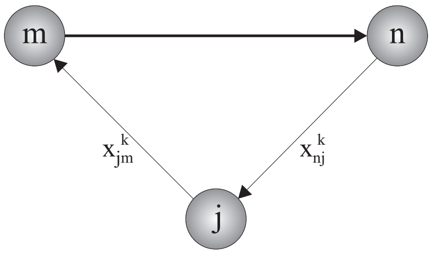


Abbildung 5.5: Gültige Ungleichung: $x_{jm}^k + x_{nj}^k \leq 1$.

5.2 OASIC-Modellierung

Ein zweiter Modellierungsansatz, genannt **OASIC** (*Optimal Architectural Synthesis with Interface Constraints*) ([GE92], [GE93]), soll nun der SILP-Modellierung gegenübergestellt werden. Auch bei diesem Ansatz werden Erkenntnisse der Polyeder-Theorie verwendet, um Bedingungen zu formulieren, die die Größe des Lösungsraumes reduzieren und damit die Lösungseffizienz verbessern.

5.2.1 Basisformulierung

Zunächst wird eine Übersicht über die verwendete Terminologie gegeben, die sich in einigen wichtigen Punkten von der SILP-Begrifflichkeit unterscheidet.

- Die zentralen Entscheidungsvariablen werden bei dieser Formulierung mit x_{jn}^k bezeichnet. Dabei bedeutet x_{jn}^k, daß Mikrooperation j in Instruktion $n \geq 1$ angeordnet wird und auf Ressourcentyp k ausgeführt wird. Auch hier kann bei sequentiellen Programmen die relative Position der Instruktion als Synonym zum Taktzyklus der Ausführung der Instruktion angesehen werden (vgl. Kapitel 5.1).

- t_j bezeichnet wieder die relative Position einer Mikrooperation j innerhalb der Instruktionen der optimierten Codesequenz bzw. den Startzeitpunkt einer Mikrooperation j (vgl. Kapitel 5.1). Diese Variable wird jedoch nur aus Gründen der Übersichtlichkeit eingeführt; in den erstellten linearen Programmen wird sie nicht verwendet, sondern mittels folgender Beziehung dargestellt:

$$t_j = \sum_{k:(j,k) \in E_R} \sum_{n \in N(j)} n \cdot x_{jn}^k$$

- $N(j) = \{asap(j), asap(j)+1, \ldots, alap(j)\}$ stellt für jede Operation j die Menge der Kontrollschritte dar, in denen eine Ausführung von j möglich ist. $asap(j)$ bezeichnet wieder den frühestmöglichen Zeitpunkt, $alap(j)$ den spätestmöglichen.

Wie bereits im Kapitel 4 angesprochen wurde, existiert für jedes beschränkte System linearer Ungleichungen ein ganzzahliges Polytop, das dieselben ganzzahligen Punkte enthält. Wird dieses als Zulässigkeitsbereich eines Optimierungsproblems verwendet, so kann eine ganzzahlige optimale Lösung durch Einsatz von Methoden der linearen Programmierung in polynomieller Zeit ermittelt werden ([NW88], [PS82]). Die Nebenbedingungen, die erforderlich sind, um die ganzzahligen Ecken eines Polytops zu bestimmen, werden ganzzahlige Facetten genannt.

Ziel der OASIC-Formulierung ist es, das ganzzahlige lineare Programm so zu formulieren, daß es in einen *Node Packing*-Graph transformiert werden kann, für den einige ganzzahlige Facetten bekannt sind. Das resultierende Polytop stimmt in der Regel nicht mit dem ganzzahligen Polytop überein, es wird jedoch durch die Berücksichtigung der zusätzlichen Facetten eine bessere Approximation an das ganzzahlige Polytop erreicht.

Gegeben sie ein ungerichteter Graph $G_N = (V_N, E_N)$ und ein Kostenvektor c. Dann ist das gewichtete *Node Packing*-Problem wie folgt definiert:

$$\begin{array}{lrcll} max & \sum_u c_u x_u & & & \\ & x_u + x_v & \leq & 1 & \forall\ (u,v) \in E_N \\ & x_u & \in & \{0,1\} & \forall\ u \in V_N \end{array} \tag{5.31}$$

Die Nebenbedingungen haben also die Gestalt $Ax \leq b$, wobei A eine $(0,1)$-Matrix ist, bei der jede Zeile zwei Einsen enthält und b ein Einsvektor ist.

Zunächst wird eine vorläufige Formulierung präsentiert, mit der das Instruktionsanordnungsproblem gelöst werden soll; diese Formulierung wird dann in das *Node Packing*-Problem transformiert. Es muß gewährleistet werden, daß jede Instruktion genau einem Startzeitpunkt und genau einer funktionalen Einheit zugewiesen wird.

$$\sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{n \in N(j)} x_{jn}^k = 1 \qquad \forall\ j \in V_D$$

Beispiel 5.6 Auch diese Nebenbedingungen werden in der zu unserem Beispielprogramm gehörenden Ausprägung angegeben. Zunächst sei wieder die betrachtete Codesequenz aufgeführt:

```
1)   r1 = dm(i0, m0);
2)   r2 = pm(i8, m8);
3)   r3 = r1 * r2;
4)   MRF = MRF + r1 * r2;
5)   r4 = dm(i1, m1);
6)   r5 = pm(i9, m9);
7)   r6 = r4 + r5;
8)   r7 = min(r4, r5);
```

Die asap/alap-Analyse ergibt für Instruktionen i = 1,2,5 und 6 als frühestmöglichen Ausführungszeitpunkt $asap(i) = 1$ und als spätestmöglichen $alap(i) = 7$. Für die übrigen Instruktionen j lauten diese Werte $asap(j) = 2$ und $alap(j) = 8$. Die erforderlichen Nebenbedingungen lauten dann wie folgt:

$$\begin{aligned}
x_{11}^{DM} + x_{12}^{DM} + x_{13}^{DM} + x_{14}^{DM} + x_{15}^{DM} + x_{16}^{DM} + x_{17}^{DM} &= 1 \\
x_{21}^{PM} + x_{22}^{PM} + x_{23}^{PM} + x_{24}^{PM} + x_{25}^{PM} + x_{26}^{PM} + x_{27}^{PM} &= 1 \\
x_{32}^{MU} + x_{33}^{MU} + x_{34}^{MU} + x_{35}^{MU} + x_{36}^{MU} + x_{37}^{MU} + x_{38}^{MU} &= 1 \\
x_{42}^{MU} + x_{43}^{MU} + x_{44}^{MU} + x_{45}^{MU} + x_{46}^{MU} + x_{47}^{MU} + x_{48}^{MU} &= 1 \\
x_{51}^{DM} + x_{52}^{DM} + x_{53}^{DM} + x_{54}^{DM} + x_{55}^{DM} + x_{56}^{DM} + x_{57}^{DM} &= 1 \\
x_{61}^{PM} + x_{62}^{PM} + x_{63}^{PM} + x_{64}^{PM} + x_{65}^{PM} + x_{66}^{PM} + x_{67}^{PM} &= 1 \\
x_{72}^{S} + x_{73}^{S} + x_{74}^{S} + x_{75}^{S} + x_{76}^{S} + x_{77}^{S} + x_{78}^{S} &= 1 \\
x_{82}^{S} + x_{83}^{S} + x_{84}^{S} + x_{85}^{S} + x_{86}^{S} + x_{87}^{S} + x_{88}^{S} &= 1
\end{aligned}$$

□

Durch die folgende Ungleichung wird sichergestellt, daß einem Ressourcentyp nicht mehr Instruktionen im selben Kontrollschritt zugeordnet werden können, als Instanzen dieses Ressourcentyps vorhanden sind:

$$\sum_{j \in V_D : (j,k) \in E_R} \sum_{n \in N'(j,k,n')} x_{jn}^k \leq R_k \qquad \forall\ k \in V_K\ \wedge\ 0 \leq n' \leq M_{steps}$$
$$\text{mit } N'(j,k,n') = \{n \in N(j) : n = n' + p,\ 0 \leq p \leq Q_j^k - 1\}$$

Beispiel 5.7 Für die Codesequenz aus Beispiel 5.6 ergeben sich hierdurch die folgenden Nebenbedingungen; aus Gründen der Übersichtlichkeit wurden diese als Konjunktion dargestellt. Im linearen Programm entfallen die Konjunktionsoperatoren, und jede Ungleichung fungiert als eigenständige Nebenbedingung.

$$\begin{array}{rclcrclcrclc}
x_{21}^{PM} + x_{61}^{PM} & \leq & 1 & \wedge & x_{11}^{DM} + x_{51}^{DM} & \leq & 1 & \wedge & x_{22}^{PM} + x_{62}^{PM} & \leq & 1 & \wedge \\
x_{12}^{DM} + x_{52}^{DM} & \leq & 1 & \wedge & x_{72}^{S} + x_{82}^{S} & \leq & 1 & \wedge & x_{32}^{MU} + x_{42}^{MU} & \leq & 1 & \wedge \\
x_{23}^{PM} + x_{63}^{PM} & \leq & 1 & \wedge & x_{13}^{DM} + x_{53}^{DM} & \leq & 1 & \wedge & x_{73}^{S} + x_{83}^{S} & \leq & 1 & \wedge \\
x_{33}^{MU} + x_{43}^{MU} & \leq & 1 & \wedge & x_{24}^{PM} + x_{64}^{PM} & \leq & 1 & \wedge & x_{14}^{DM} + x_{54}^{DM} & \leq & 1 & \wedge \\
x_{74}^{S} + x_{84}^{S} & \leq & 1 & \wedge & x_{34}^{MU} + x_{44}^{MU} & \leq & 1 & \wedge & x_{25}^{PM} + x_{65}^{PM} & \leq & 1 & \wedge \\
x_{15}^{DM} + x_{55}^{DM} & \leq & 1 & \wedge & x_{75}^{S} + x_{85}^{S} & \leq & 1 & \wedge & x_{35}^{MU} + x_{45}^{MU} & \leq & 1 & \wedge \\
x_{26}^{PM} + x_{66}^{PM} & \leq & 1 & \wedge & x_{16}^{DM} + x_{56}^{DM} & \leq & 1 & \wedge & x_{76}^{S} + x_{86}^{S} & \leq & 1 & \wedge \\
x_{36}^{MU} + x_{46}^{MU} & \leq & 1 & \wedge & x_{27}^{PM} + x_{67}^{PM} & \leq & 1 & \wedge & x_{17}^{DM} + x_{57}^{DM} & \leq & 1 & \wedge \\
x_{77}^{S} + x_{87}^{S} & \leq & 1 & \wedge & x_{37}^{MU} + x_{47}^{MU} & \leq & 1 & \wedge & x_{78}^{S} + x_{88}^{S} & \leq & 1 & \wedge \\
x_{38}^{MU} + x_{48}^{MU} & \leq & 1 & & & & & & & & &
\end{array}$$

□

Schließlich darf keine Vorrangbedingung verletzt werden; ist also eine Instruktion j von einer Instruktion i abhängig, darf i nicht nach j ausgeführt werden. Diese Bedingung wird zunächst mit Hilfe der Ungleichung (5.32) umgesetzt.

$$\sum_{\substack{k \in V_K: \\ (i,k) \in E_R}} x_{in_i}^k + \sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} x_{jn_j}^k \ \leq\ 1 \qquad \forall (i,j) \in E_D \quad \text{mit} \quad n_j \leq n_i + Q_i^k - 1; \tag{5.32}$$
$$n_i \in N(i),\ \ n_j \in N(j)\ \wedge\ x_{in_i}^k \in \{0,1\}$$

Werden die Integralitätsbedingungen für die x_{in_i} zu $0 \leq x_{in_i} \leq 1$ relaxiert, erhalten wir ein lineares Ungleichungssystem der Form $Ax \leq b$, wobei A eine $(0,1)$-Matrix und b ein Einsvektor ist. Aus einem solchen Ungleichungssystem wird durch das nachfolgend beschriebene Vorgehen ein *Node Packing*-Graph konstruiert, aus dem ganzzahlige Facetten extrahiert werden können. Zunächst wird jede Variable x_{in}^k auf einen Knoten $u \in V_N$ abgebildet. Die Kanten E_N des Graphen werden wie folgt definiert: Für jede Zeile von A (jede Nebenbedingung), definieren die Spalten (Variablen) mit Koeffizient 1 einen vollständigen Teilgraphen[3] in G. Dieser Graph stellt nun eine Instanz des *Node Packing*-Problems dar. Mit Hilfe von Cliquen können ganzzahlige Facetten dieses Problems ermittelt werden ([GE92], Kapitel 4). Es gilt

$$\sum_{u \in K} x_u \leq 1 \qquad \text{für alle Cliquen } K \tag{5.33}$$

Innerhalb der vorliegenden Modellierung wurden solche Clique-Facetten extrahiert, verallgemeinert und in ein relaxiertes LP-Modell eingebaut. Dieses Vorgehen wird an einem Beispiel verdeutlicht.

Beispiel 5.8 Seien gegeben zwei Instruktionen i und j mit Ausführungszeiten $w_i = w_j = 1$, zwischen denen eine Abhängigkeit der Form $i \prec j$ besteht. Die Mengen der zulässigen Kontrollschritte seien wie folgt definiert: $N(i) = \{1, \ldots, 4\}; N(j) = \{2, \ldots, 5\}$. Die Kanten

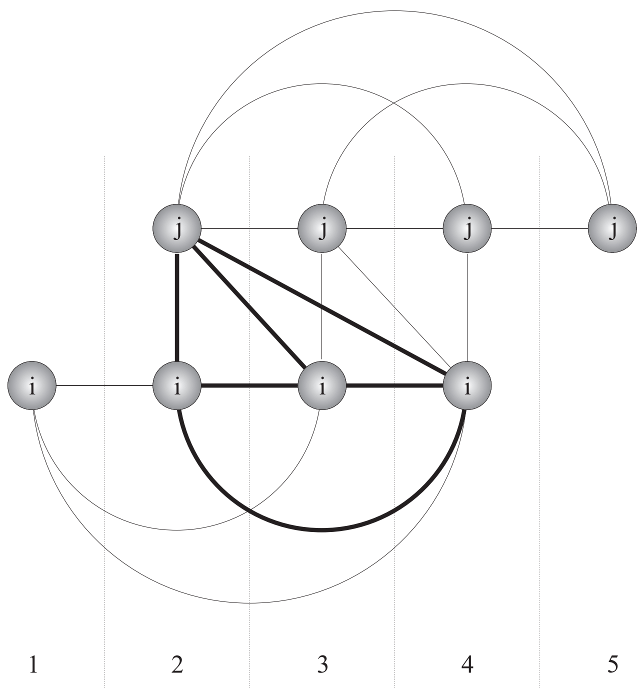


Abbildung 5.6: Node Packing.

[3] Ein vollständiger Teilgraph (Clique) C eines Graphen G ist ein Graph $C = (V_C, E_C)$, so daß für alle $u, v \in V_C$ gilt: $(u, v) \in E_C$.

des Graphen werden entsprechend der Ungleichung (5.32) gezogen, d.h. alle unzulässigen Anordnungen von i und j bilden Kanten im Graph. Eine Clique-Facette für Kontrollschritt 2 ist durch fett gedruckte Kanten in Abbildung 5.6 hervorgehoben.

□

Unter Berücksichtigung dieser Clique-Facette ergibt sich dann folgende Formulierung der Vorrangbedingungen:

$$\sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{\substack{n_j \le n \\ n_j \in N(j)}} x^k_{jn_j} + \sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{\substack{n_i \ge n - Q^k_i + 1 \\ n_i \in N(i)}} x^k_{in_i} \le 1 \qquad \forall \quad (i,j) \in E_D$$
$$\wedge \quad n \in (\{n + Q^k_i - 1 \mid n \in N(i)\} \cap N(j))$$

Beispiel 5.9 Aus Platzgründen werden in diesem Beispiel nur die Nebenbedingungen aufgelistet, die sich aus der Datenabhängigkeit zwischen den Instruktionen `r1 = dm(i0, m0)` und `r3 = r1 * r2` des Eingabeprogrammes aus Beispiel 5.6 ergeben.

$$\begin{array}{rcl}
x^{MU}_{32} + x^{DM}_{12} + x^{DM}_{13} + x^{DM}_{14} + x^{DM}_{15} + x^{DM}_{16} + x^{DM}_{17} & \le & 1 \\
x^{MU}_{32} + x^{MU}_{33} + x^{DM}_{13} + x^{DM}_{14} + x^{DM}_{15} + x^{DM}_{16} + x^{DM}_{17} & \le & 1 \\
x^{MU}_{32} + x^{MU}_{33} + x^{MU}_{34} + x^{DM}_{14} + x^{DM}_{15} + x^{DM}_{16} + x^{DM}_{17} & \le & 1 \\
x^{MU}_{32} + x^{MU}_{33} + x^{MU}_{34} + x^{MU}_{35} + x^{DM}_{15} + x^{DM}_{16} + x^{DM}_{17} & \le & 1 \\
x^{MU}_{32} + x^{MU}_{33} + x^{MU}_{34} + x^{MU}_{35} + x^{MU}_{36} + x^{DM}_{16} + x^{DM}_{17} & \le & 1 \\
x^{MU}_{32} + x^{MU}_{33} + x^{MU}_{34} + x^{MU}_{35} + x^{MU}_{36} + x^{MU}_{37} + x^{DM}_{17} & \le & 1
\end{array}$$

□

Das Polytop des ganzzahligen Problems wird bei Verwendung dieser Formulierung besser durch das relaxierte Polytop approximiert, als bei Einsatz von Ungleichung (5.32). In Spezialfällen kann das relaxierte Polyeder mit dem ganzzahligen übereinstimmen; im allgemeinen ist das *Node Packing*-Polytop jedoch nicht-ganzzahlig ([GE92], [GE93]).

Damit ergibt sich zur Lösung des Instruktionsanordnungsproblems die folgende ILP-Formulierung:

- Zielfunktion

$$\min \ M_{steps} \tag{5.34}$$

- Nebenbedingungen

1. Zeitschranken
   Bei keiner Instruktion darf die Startzeit das zu berechnende Maximum M_{steps} an Kontrollschritten überschreiten.

$$t_j \leq M_{steps} \qquad \forall\, j \in V_D \tag{5.35}$$

2. Vorrangbedingungen
   Ist Instruktion j von Instruktion i abhängig, dann darf j erst nach Ausführung von i ausgeführt werden.

$$\begin{aligned}
\sum_{\substack{k\in V_K:\\(j,k)\in E_R}} \sum_{\substack{n_j\leq n\\n_j\in N(j)}} x^k_{jn_j} + \sum_{\substack{k\in V_K:\\(j,k)\in E_R}} \sum_{\substack{n_i\geq n-Q^k_i+1\\n_i\in N(i)}} x^k_{in_i} \;&\leq\; 1 \qquad \forall\,(i,j)\in E_D^{true} \cup E_D^{output},\\
n \in (\{n+Q^k_i-1 \mid n \;&\in\; N(i)\} \cap N(j))\\
\sum_{\substack{k\in V_K:\\(j,k)\in E_R}} \sum_{\substack{n_j< n\\n_j\in N(j)}} x^k_{jn_j} + \sum_{\substack{k\in V_K:\\(j,k)\in E_R}} \sum_{\substack{n_i\geq n-Q^k_i+1\\n_i\in N(i)}} x^k_{in_i} \;&\leq\; 1 \qquad \forall\,(i,j)\in E_D^{anti},\\
n \in (\{n+Q^k_i-1 \mid n \;&\in\; N(i)\} \cap N(j))
\end{aligned} \tag{5.36}$$

   Der Unterschied zwischen diesen beiden Ungleichungen liegt darin, daß bei einer Benutzungs-Setzungsabhängigkeit die gleichzeitige Ausführung zugelassen werden muß. Daher darf n_j bei der Darstellung der Benutzungs-Setzungsabhängigkeiten in der ersten Summation nur Werte $n_j < n$ annehmen.

3. Ausführungsbeschränkung
   Die Ausführung jeder Operation muß in genau einem Kontrollschritt beginnen und wird von genau einem Ressourcentyp durchgeführt.

$$\sum_{\substack{k\in V_K:\\(j,k)\in E_R}} \sum_{n\in N(j)} x^k_{jn} \;=\; 1 \qquad \forall\, j \in V_D \tag{5.37}$$

4. Ressourcenschranken
   In einem Kontrollschritt dürfen nicht mehr als R_k Operationen auf Ressourcentyp k ausgeführt werden, d.h. die Anzahl der vorhandenen Instanzen von k darf nicht überschritten werden.

$$\begin{aligned}
\sum_{j\in V_D:(j,k)\in E_R} \sum_{n\in N'(j,k,n')} x^k_{jn} \leq R_k \qquad \forall k\in V_K \wedge 0\leq n'\leq M_{steps}\\
\text{mit } N'(j,k,n') = \{n \in N(j) : n = n'+p,\; 0\leq p\leq Q^k_j-1\}
\end{aligned} \tag{5.38}$$

5.2.2 Integration der Registerallokation

Um das Problem der Registerallokation bezüglich einer homogenen Registermenge zu berücksichtigen, muß obige Formulierung lediglich um einige Nebenbedingungen erweitert werden. Es soll gewährleistet werden, daß in jedem Kontrollschritt nicht mehr als R Register verwendet werden, so daß sich die Lebenszeiten von höchstens R Variablen überlappen.

Eine Variable i wird an einem Programmpunkt definiert, wenn ihr ein Wert zugewiesen wird. Eine Variable wird verwendet, wenn sie in einem Programmpunkt referenziert wird. Betrachtet man eine fest vorgegebene Instruktionsanordnung, so kann die Lebenszeit einer Variable durch eine lebenszeitdefinierende Kante $i \rightarrow j$ zwischen der Instruktion i, die die Variable produziert und der Instruktion j, die die Variable als letzte verwendet, repräsentiert werden. Jedes symbolische Register kann aber innerhalb mehrerer Instruktionen verwendet werden. Die Konsequenz ist, daß keine eindeutigen lebenszeitdefinierenden Kanten gefunden werden können, denn bei der simultanen Betrachtung von Instruktionsanordnung und Registerallokation ist die Anordnung der Instruktionen nicht fixiert. Naiv betrachtet muß also eine lebenszeitdefinierende Kante zwischen einer Definition und jeder Verwendung gezogen werden. Die Anzahl der Kanten kann jedoch durch Transitivitätsanalyse und mit Hilfe der eingangs erwähnten *asap-alap*-Analyse reduziert werden.

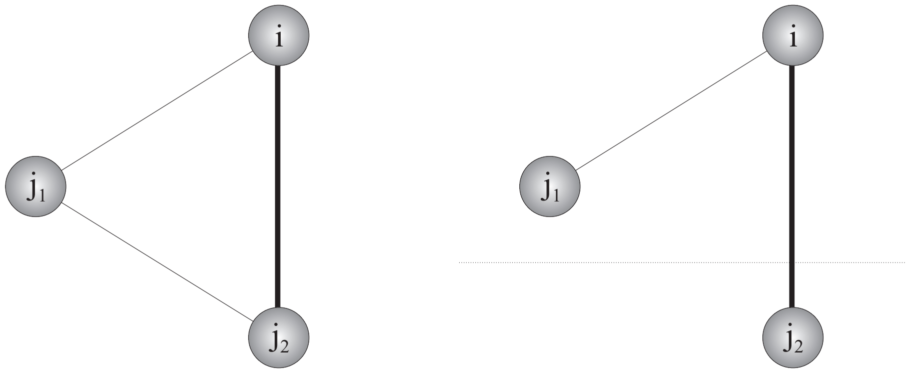
Beispiel 5.10 In Abbildung 5.7 repräsentieren die Kanten Abhängigkeiten zwischen einer setzenden Instruktion i und zwei verwendenden Instruktionen j_1 und j_2. Aufgrund der Abhängigkeit der Instruktion j_2 von j_1 braucht in (a) nur die Kante $i \rightarrow j_2$ als lebenszeitdefinierend betrachtet zu werden. In Abbildung 5.7(b) wird der Fall gezeigt, daß aufgrund der asap/alap-Analyse Kanten vernachlässigt werden können. Gilt $asap(j_2) \geq alap(j_1)$, braucht nur die Kante $i \rightarrow j_2$ als lebenszeitdefinierend betrachtet zu werden.

$\square$

Zur Formulierung der Nebenbedingungen für die Registerallokation müssen zunächst einige neue Begrifflichkeiten definiert werden.

Definition 5.4 *Eine Kante* $i \prec j$ **kreuzt** *Kontrollschritt* n*, wenn* $N(i) \cap \{0, 1, \ldots, n - (w_i - 1)\} \neq \emptyset$ *und* $N(j) \cap \{n+1, n+2, \ldots, T\} \neq \emptyset$*.*

Die Anzahl der Kanten mit Ausgangsknoten i, die Kontrollschritt n kreuzen, wird mit $e_n(i)$ bezeichnet; formal: $e_n(i) = |\{i \prec j_k \mid k = 1, \ldots, l;\ (i, j_k) \text{ kreuzt Kontrollschritt } n\}|$. Desweiteren wird mit $M(n)$ die Menge aller maximalen Mengen $M'(n)$ von Kanten bezeichnet, die Kontrollschritt n kreuzen und paarweise verschiedene Startknoten haben.



(a) (b)

Abbildung 5.7: Zu berücksichtigende lebenszeitdefinierende Kanten; (a) nach Transitivitätsanalyse und (b) nach asap/alap-Analyse.

$$\begin{aligned}\sum_{j_a \prec j_b \in M'(n)} \Big(\sum_{k \in V_K} \sum_{\substack{n_1 \leq n \\ n_1 \in N(j_a)}} x^k_{j_a n_1} + \sum_{k \in V_K} \sum_{\substack{n_2 > n \\ n_2 \in N(j_b)}} x^k_{j_b n_2} \; - \\ \sum_{k \in V_K} \sum_{\substack{n_3 \leq n \\ n_3 \in N(j_b)}} x^k_{j_b n_3} - \sum_{k \in V_K} \sum_{\substack{n_4 > n \\ n_4 \in N(j_a)}} x^k_{j_a n_4} \Big) \;\leq\; 2 \cdot R \\ \forall\, n \;\wedge\; \forall\, M'(n) \in M(n)\end{aligned} \tag{5.39}$$

Kreuzen also e Kanten Kontrollschritt n und haben unter diesen $e_n(i)$ den Ausgangspunkt i, während der Rest der Kanten paarweise verschiedene Ausgangspunkte hat, muß Ungleichung (5.39) genau e-mal für Kontrollschritt n erzeugt werden. Im allgemeinen Fall ergibt sich die Anzahl der für Kontrollschritt n zu erzeugenden Nebenbedingungen zu $\prod_i e_n(i)$. Innerhalb dieser Nebenbedingungen wird zweimal die Anzahl der Kreuzungskanten für jeden Kontrollschritt berechnet. Hierbei werden die zu betrachtenden Variablen in vier Gruppen aufgeteilt und abhängig von ihrer Gruppenzugehörigkeit entweder mit positivem oder negativem Vorzeichen in der Ungleichung berücksichtigt. Dieses Vorgehen wird nun an einem Beispiel genauer erläutert.

Beispiel 5.11 Seien gegeben zwei Instruktionen i und j mit $w_i = w_j = 1$ und $N(i) = \{1, \ldots, 4\}$, $N(j) = \{2, \ldots, 5\}$. Dann wird für Kontrollschritt n die folgende Gleichung erzeugt:

$$\sum_{1 \leq n_1 \leq 2} x^k_{in_1} + \sum_{3 \leq n_2 \leq 5} x^k_{jn_2} + \sum_{2 \leq n_3 \leq 2} x^k_{in_3} - \sum_{3 \leq n_4 \leq 4} x^k_{in_4} \;\leq\; 2R$$

$$\Leftrightarrow \qquad x_{i1}^k + x_{i2}^k - x_{i3}^k - x_{i4}^k - x_{j2}^k + x_{j3}^k + x_{j4}^k + x_{j5}^k \;\leq\; 2R$$

Graphisch veranschaulicht erhalten wir damit folgende Anordnung:

	$+$	$-$
1	x_{i1}^k	
2	x_{i2}^k	x_{j2}^k
3	x_{i3}^k	x_{j3}^k
4	x_{i4}^k	x_{j4}^k
5		x_{j5}^k
	$+$	$-$

Hierbei ist zu beachten, daß $x_{ir}^k = 1$ und $x_{js}^k = 1$ für genau ein $r \in \{1, \ldots, 4\}$ und genau ein $s \in \{2, \ldots, 5\}$ gelten muß. Existiert also eine Kreuzungskante, dann ergibt sich für die linke Seite als Wert 2, existiert keine Kreuzungskante, so ergibt sich der Wert 0. Aufgrund der Anordnung der $+$ und $-$-Felder kann niemals die linke Seite der Ungleichung (5.39) einen negativen Wert ergeben, da sonst gegen eine Vorrangbedingung verstoßen würde.

$\square$

5.2.3 Berechnungsverfahren zur Ermittlung der optimalen Lösung

Zunächst wird die LP-Relaxation des Problems gelöst, d.h. die Integralitätsbedingung wird für alle Variablen aufgehoben. Der Zielfunktionswert dieses Problems stellt eine untere Schranke für den optimalen Zielfunktionswert des ganzzahligen linearen Programmes dar. Die Kenntnis einer solchen unteren Schranke ermöglicht es, eine Abschätzung für die Güte einer approximativen Lösung anzugeben, selbst wenn die optimale Lösung aus Komplexitätsgründen nicht berechnet werden kann.

Besitzen alle Variablen dieser Lösung ganzzahlige Werte, stellt sie auch eine optimale Lösung des ganzzahligen Problems dar; ansonsten werden alle nicht-ganzzahligen Variablen als binär spezifiziert und das Problem erneut gelöst. Dieses Verfahren wird iterativ so lange wiederholt, bis alle Variablen ganzzahlige Werte besitzen. Bei jedem Schritt dieses Iterationsverfahrens repräsentiert der Zielfunktionswert eine zunehmend exaktere untere Schranke für die optimale Lösung des Problems.

5.2.4 Komplexitätsanalyse

Innerhalb der OASIC-Modellierung wird einer Instruktion bezüglich jeden möglichen Ausführungszeitpunktes und jedes verwendeten Ressourcen- bzw. Registermengenknotens eine eigene Variable x_{ij}^{km} zugeordnet.

Mit $u_x = \max\{alap(j) - asap(j) \mid j \in V_D\}$ läßt sich die Anzahl der verwendeten Variablen nach oben abschätzen durch $u_x \cdot |V_D| \cdot |V_R|$. Es gilt $u_x \leq |V_D|$, $|V_R| = 1$ für alle funktionalen Einheiten und $|V_R| \leq 4$ für alle Registermengenknoten. Also werden höchstens $\mathcal{O}(u_x \cdot |V_D| \cdot |V_R|) = \mathcal{O}(|V_D|^2)$ Variablen verwendet.

Aufgrund des in Abschnitt 5.2.3 beschriebenen iterativen Berechnungsverfahrens hängt die genaue Anzahl der als binär zu spezifizierenden Variablen vom Berechnungsverlauf bzw. vom Eingabeprogramm ab. Im schlimmsten Falle müssen alle Variablen als binär deklariert werden; eine obere Schranke für die Anzahl der relevanten Binärvariablen ist also ebenso wie bei der SILP-Modellierung $\mathcal{O}(|V_D|)^2$.

Zu untersuchen ist weiterhin die Anzahl der erforderlichen Nebenbedingungen. Zur Darstellung der Zeitschranken (5.35) sind $|V_D|$ Nebenbedingungen erforderlich, für die Vorrangbedingungen (5.36) $\mathcal{O}(u_r \cdot |E_D|)$. Dabei ist u_r wie folgt definiert:

$$u_r = \max\{alap(j) \mid j \in V_D\} - \min\{asap(i) \mid i \in V_D\} + 1$$

Es gilt $\mathcal{O}(u_r \cdot |E_D|) = \mathcal{O}(|V_D| \cdot |E_D|) = \mathcal{O}(|V_D|^3)$. Zur Modellierung der Ausführungsbeschränkungen (5.37) benötigen wir $|V_D|$ Nebenbedingungen, für die Ressourcenschranken (5.38) schließlich $|V_K| \cdot |V_D| = \mathcal{O}(|V_D|)$. Ohne Berücksichtigung der Registerallokation läßt sich die Anzahl der Nebenbedingungen also durch $\mathcal{O}(|V_D|^3)$ abschätzen.

Zur Erzeugung der Registerkreuzungsbeschränkungen ergeben sich für jeden Kontrollschritt n genau $\Pi_i e_n(i)$ Nebenbedingungen. Hier läßt sich nur eine sehr ungenaue obere Schranke angeben, da die Anzahl der erforderlichen Nebenbedingungen stark vom jeweiligen Eingabeprogramm abhängt. Existieren zu jedem definierenden Vorkommen zwei angewandte Vorkommen, so daß sich die zugehörigen lebenszeitdefinierenden Kanten für einen Kontrollschritt n kreuzen, beträgt die Anzahl der Kreuzungskanten $\Pi_i 2 = 2^i$. [GE93] zufolge ist die Anzahl der erforderlichen Registerkreuzungsbedingungen für die meisten Algorithmen gering; dennoch kann diese exponentiell in der Anzahl der Instruktionen wachsen.

Kapitel 6

Modellierungserweiterungen

6.1 Berücksichtigung der Globalität

Wie schon in Kapitel 3.1 angesprochen wurde, ist zur Bewahrung der Semantik des Eingabeprogrammes bei globalen Analysen die alleinige Betrachtung von Datenabhängigkeiten nicht ausreichend. Ein verbreiteter Ansatz ist es, am Anfang und Ende jeden Basisblockes virtuelle Knoten einzuführen (Ellis, 1986; [GE92]). Durch eigens erzeugte Datenabhängigkeiten zu diesen virtuellen Knoten wird verhindert, daß Instruktionen aus dem Basisblock entfernt werden, dem sie im Eingabeprogramm zugeordnet sind. Die Verschiebung von Instruktionen über Basisblockgrenzen hinweg ist damit ausgeschlossen. Im Rahmen dieser Arbeit sollen solche globale Änderungen jedoch zugelassen werden; hierzu wird für jede Anweisung überprüft, in welchen Basisblöcken sie stehen darf, und die entstehende Disjunktion wird in das lineare Programm aufgenommen. Zur korrekten Modellierung sind Binärvariablen, analog zur Formulierung der Parallelisierungsverbote, erforderlich. Das folgende Vorgehen wurde in Anlehnung an [DK96] entwickelt (vgl. hierzu auch [NW88]).

Es sei ein Basisblock B_k gegeben. Diesem Basisblock werden ein Beginnzeitpunkt t_k^A und ein Endzeitpunkt t_k^E zugeordnet. Eine Instruktion j liegt genau dann im Basisblock B_k, wenn gilt $t_k^A \leq t_j \leq t_k^E$. Dann können wir die Menge X_k aller Instruktionen, die im Basisblock B_k liegen folgendermaßen definieren:

$$X_k = \{j \in V_D \mid t_k^A - t_j \leq 0 \ \wedge \ t_j - t_k^E \leq 0\}.$$

Um nun den Sachverhalt, daß eine Instruktion j zu genau einem von l möglichen Basisblöcken $B_1, \ldots, B_l$ gehören muß, durch ein simultanes Ungleichungssystem unter Einschluß von Binärvariablen $y_1, \ldots, y_l$ darstellen zu können, benötigen wir zunächst eine obere Schranke T, so daß $t_j \leq T \quad \forall\, j \in V_D$ gilt. Ersetzt man eine Nebenbedingung $t_j - t_\alpha \leq 0$ durch $t_j - t_\alpha \leq T$, dann wird sie redundant und schränkt den Zulässigkeitsbereich nicht mehr ein. Für die Binärvariablen wird somit vereinbart: $y_j = 0,$ falls $j \in B_k$ und $y_j = 1,$ falls $j \notin B_k$. Als Wert für die obere Schranke wählen wir $T = 2I + 1$, wobei $I = |V_D|$ die Anzahl der Instruktionen des Eingabeprogrammes angibt.

Es werden die folgenden Nebenbedingungen generiert, die garantieren, daß Instruktion j in genau einer der Mengen $X_1, \ldots, X_l$ liegt:

$$\begin{aligned} t_k^A - t_j - T y_j^k \ &\leq \ 0 \quad \forall\, k = 1, \ldots, l \qquad (6.1) \\ t_j - t_k^E - T y_j^k \ &\leq \ 0 \quad \forall\, k = 1, \ldots, l \\ \sum_{k=1}^{l} y_j^k &= l - 1 \\ y_j^k &\in \{0, 1\} \quad \forall\, k = 1, \ldots, l \end{aligned}$$

Stehen genau zwei Mengen X_1 und X_2 zur Wahl ($l = 2$) so braucht nur eine Binärvariable eingeführt zu werden. Dann sind folgende Nebenbedingungen erforderlich:

$$\begin{aligned} t_1^A - t_j - Ty_j &\leq 0 \\ t_j - t_1^E - Ty_j &\leq 0 \\ t_1^A - t_j - T(1-y_j) &\leq 0 \\ t_j - t_1^E - T(1-y_j) &\leq 0 \\ y_j \in \{0,1\} \end{aligned} \tag{6.2}$$

Beispiel 6.1 Kann eine Instruktion j in den drei Basisblöcken B_1, B_2 und B_3 liegen, werden folgende Nebenbedingungen erzeugt:

$$\begin{array}{ccccccc} t_1^A & - & t_j & - & Ty_j^1 & \leq & 0 \\ t_j & - & t_1^E & - & Ty_j^1 & \leq & 0 \\ t_2^A & - & t_j & - & Ty_j^2 & \leq & 0 \\ t_j & - & t_2^E & - & Ty_j^2 & \leq & 0 \\ t_3^A & - & t_j & - & Ty_j^3 & \leq & 0 \\ t_j & - & t_3^E & - & Ty_j^3 & \leq & 0 \\ y_j^1 & + & y_j^2 & + & y_j^3 & = & 2 \end{array}$$

Gilt in einer zulässigen Lösung $y_j^2 = 0$, so muß Instruktion j im Basisblock B_2 liegen. Die Nebenbedingungen ergeben sich dann zu:

$$\begin{array}{ccccc} t_1^A & - & t_j & \leq & T \\ t_j & - & t_1^E & \leq & T \\ & & t_2^A & \leq & t_j \\ & & t_j & \leq & t_2^E \\ t_3^A & - & t_j & \leq & T \\ t_j & - & t_3^E & \leq & T \\ y_j^1 & + & y_j^3 & = & 2 \end{array}$$

□

Die Menge der Basisblöcke, denen eine Instruktion zugeordnet werden kann, läßt sich anhand des Kontrollabhängigkeitsgraphen ermitteln. Besitzen Instruktionen verschiedener

Basisblöcke exakt dieselben Vorgänger im Kontrollflußgraph und stimmen die Kantentypen, durch die sie mit ihren Vorgängern verbunden sind, überein, so können sie in jedem dieser Basisblöcke stehen.

Beispiel 6.2 Es werde folgende Codesequenz betrachtet:

```
1)          r15 = r0 * r3;
2)          r14 = r1 * r4;
3)          comp(r14, r15);
4)          if gt jump _G;
5)          r14 = r15 - r14;
6)          jump _M;
7)      _G: r14 = r14 - r15;
8)      _M: r13 = r8 + r12;
9)          r10 = r7 + r11;
```

Der zugehörige Kontrollabhängigkeitsgraph ist in Abbildung 6.1 dargestellt. Dann ergibt sich, daß Instruktionen 1,2,8 und 9 denselben Kontrollbedingungen unterliegen und daher auch gemeinsam in einem Basisblock angeordnet werden können. Instruktion 4 stellt eine Verzweigung dar; eine Verschiebung aus dem aktuellen Basisblock ist daher nicht sinnvoll (vgl. dazu den nächsten Abschnitt). Da Instruktion 3 die Statusbits setzt, die in Instruktion 4 ausgewertet werden, muß sie unmittelbar vor dieser ausgeführt werden, kann also auch nicht in einen anderen Basisblock verschoben werden.

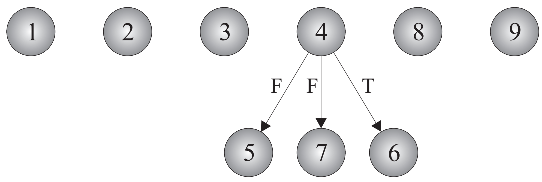


Abbildung 6.1: Der Kontrollabhängigkeitsgraph zu Beispiel 6.2.

□

Damit die Zuordnung von Instruktionen zu Basisblöcken wohldefiniert ist, dürfen weder Verzweigungen noch Schleifenanweisungen aus ihrem aktuellen Basisblock entfernt werden, da durch diese Instruktionen die Basisblockstruktur definiert ist. Wird ein Basisblock B_k von einer Schleifenanweisung i_1 eingeleitet und von einer Verzweigung i_2 abgeschlossen, bedeutet die Zuordnung einer Instruktion j zu B_k, daß j zwischen i_1 und i_2 angeordnet wird. Eine Umordnung der Basisblöcke wird ausgeschlossen, d.h. die im Eingabeprogramm vorliegende Anordnung der Basisblöcke wird auch für das Ausgabeprogramm als verbindlich angesehen. Hierdurch wird jedoch nur die Abfolge der Kontrollflußinstruktionen (vgl Kapitel 2.10) fixiert; alle anderen Instruktionen können zwischen Basisblöcken, die denselben Kontrollbedingungen unterliegen, verschoben werden. Die Abfolge der Basisblöcke wird im linearen Programm festgelegt, indem jeweils für zwei aufeinanderfolgende Basisblöcke B_{k1} und B_{k2} folgende Ungleichungen erzeugt werden:

$$\begin{aligned} t^A_{k1} - t^E_{k1} &\leq 0 \\ t^A_{k2} - t^E_{k2} &\leq 0 \\ t^E_{k1} &< t^A_{k2} \end{aligned} \tag{6.3}$$

Es werden also nicht alle möglichen Umordnungen der Instruktionen zugelassen. Die Qualität des erzeugten Codes hängt jedoch vom Grad der Parallelisierung der Instruktionen ab; die Reihenfolge der Basisblöcke hat darauf keinen bestimmenden Einfluß. Demzufolge ist diese Einschränkung in aller Regel unbedeutend. Dies läßt sich an Beispiel 6.3 veranschaulichen.

Beispiel 6.3 Es sei folgende Codesequenz gegeben:

```
    r15 = r0 * r3;
    r14 = r1 * r4;
    comp(r14, r15);
    if gt jump _G;
    jump _M;
_G: r14 = r14 - r15;
_M: r13 = r8 + r12;
    r10 = r7 + r11;
```

Dann ist folgende Umordnung zulässig:

```
    r15 = r0 * r3, r13 = r8 + r12;
    r14 = r1 * r4, r10 = r7 + r11;
    comp(r14, r15);
    if gt jump _G;
    jump _M;
_G: r14 = r14 - r15;
_M:
```

□

Diese Vorgehensweise ermöglicht die Ermittlung einer global optimalen Lösung des Instruktionsanordnungsproblems unter Berücksichtigung bedingter Anweisungen oder von Schleifen. Zu beachten ist lediglich, daß von der im Kapitel 5.1 vorgestellten intuitiven Interpretation der SILP-Flußvariablen etwas abstrahiert werden muß. Gilt für eine Variable $x_{ij}^k = 1$, so wurde dies damit beschrieben, daß Instruktion i eine Instanz der Hardwareressource k an Instruktion j weiterreicht. Gehören nun i und j zu sich gegenseitig ausschließenden Basisblöcken, so findet in der realen Programmausführung eine solche Übergabe nicht statt. Dann muß die Aussage $x_{ij}^k = 1$ so interpretiert werden, daß Instruktion i und j dieselbe Instanz des Ressourcentyps k verwenden, und daß keine Instruktion, die im Ausgabecode zwischen i und j angeordnet ist, auf dieser Instanz ausgeführt wird. Die Korrektheit der Modellierung wird hiervon nicht berührt, lediglich die Beschreibung der mathematischen Aussage $x_{ij}^k = 1$ muß angepaßt werden.

6.2 Anpassung von SILP an den ADSP-2106x

Wie aus Kapitel 5.1.1 ersichtlich wird, ist die Zuordnung von Instruktionen zu Hardwareressourcen für die Modellierung von großer Bedeutung. Hierdurch werden parallel arbeitende Hardwarekomponenten spezifiziert und somit die Parallelverarbeitungskapazität des Prozessors im erstellten linearen Programm abgebildet. Um die Rechenvorgänge des ADSP-2106x korrekt zu modellieren, benötigen wir folgende Ressourcenknoten:

- DM: modelliert Zugriffe auf den Datenspeicher
- PM: modelliert Datenzugriffe auf den Programmspeicher
- S (Standard): Durch diesen Ressourcenknoten wird eine Abstraktion von den technischen Gegebenheiten des Prozessors vollzogen, indem ihm ALU- und Shifterinstruktionen sowie die gemischten Instruktionen (vgl. 2.9) zugeordnet werden. Zwar stellen ALU und Shifter eigenständige funktionale Einheiten dar, dennoch können sie niemals parallel arbeiten. Da gerade die Möglichkeit der parallelen Ausführung ein Grund für die Einführung verschiedener Ressourcenknoten darstellt, wäre die Zuteilung zu unterschiedlichen Knoten nicht sinnvoll.
- MU: modelliert Multiplizierer-Instruktionen
- C: modelliert Kontrollflußinstruktionen

Zwischen Instruktionen, die demselben Ressourcentyp zugeordnet sind, wird eine Parallelausführung durch die Serialisierungsbedingungen ausgeschlossen. Instruktionen hingegen, die verschiedenen Ressourcentypen zugeordnet sind, können grundsätzlich parallel ausgeführt werden.

6.2.1 Vereinfachungen

Da beim ADSP-2106x jede Instruktion auf genau einem Ressourcentyp ausgeführt werden kann, entfällt das Problem der Ressourcenselektion (vgl hierzu auch Kapitel 6.6). Dadurch wird die Formulierung vereinfacht, denn die Ungleichungstypen 5.12 und 5.13 können nun wie folgt zusammengefaßt werden:

$$\Phi_j^k = 1 \qquad \forall\, j \in V_D,\ \forall\, k \in V_k:\ (j,k) \in E_R \tag{6.4}$$

$$\Psi_j^k = 1 \qquad \forall\, j \in V_D,\ \forall\, k \in V_k:\ (j,k) \in E_R \tag{6.5}$$

Beispiel 6.4 Es sei wieder die folgende Beispielcodesequenz gegeben:

```
1)   r1 = dm(i0, m0);
2)   r2 = pm(i8, m8);
3)   r3 = r1 * r2;
4)   MRF = MRF + r1 * r2;
5)   r4 = dm(i1, m1);
6)   r5 = pm(i9, m9);
7)   r6 = r4 + r5;
8)   r7 = min(r4, r5);
```

Es folgen die für dieses Eingabeprogramm zu generierenden Nebenbedingungen vom Typ (6.5), die aus Gründen der Übersichtlichkeit als Konjunktion dargestellt sind.

$$
\begin{array}{rclcrclcrclc}
x_{51}^{DM} + x_{DM1}^{DM} & = & 1 & \wedge & x_{62}^{PM} + x_{PM2}^{PM} & = & 1 & \wedge & x_{43}^{MU} + x_{MU3}^{MU} & = & 1 & \wedge \\
x_{34}^{MU} + x_{MU4}^{MU} & = & 1 & \wedge & x_{15}^{DM} + x_{DM5}^{DM} & = & 1 & \wedge & x_{26}^{PM} + x_{PM6}^{PM} & = & 1 & \wedge \\
x_{87}^{S} + x_{S7}^{S} & = & 1 & \wedge & x_{78}^{S} + x_{S8}^{S} & = & 1 & \wedge & x_{15}^{DM} + x_{1DM}^{DM} & = & 1 & \wedge \\
x_{26}^{PM} + x_{2PM}^{PM} & = & 1 & \wedge & x_{34}^{MU} + x_{3MU}^{MU} & = & 1 & \wedge & x_{43}^{MU} + x_{4MU}^{MU} & = & 1 & \wedge \\
x_{51}^{DM} + x_{5DM}^{DM} & = & 1 & \wedge & x_{62}^{PM} + x_{6PM}^{PM} & = & 1 & \wedge & x_{78}^{S} + x_{7S}^{S} & = & 1 & \wedge \\
x_{87}^{S} + x_{8S}^{S} & = & 1 & & & & & & & & &
\end{array}
$$

□

6.2.2 Berücksichtigung der eingeschränkten Parallelität

Problematik

Eine Annahme, die der in SILP verwendeten ILP-Formulierung implizit zugrundeliegt, besteht in der Zugehörigkeit aller im Ressourcenflußgraph eingeführten Ressourcenknoten zu parallelen funktionalen Einheiten. Durch Zuordnung zweier Instruktionen zu verschiedenen Ressourcenknoten wird also implizit die Möglichkeit gleichzeitiger Ausführung vorausgesetzt. Bei der zugrundeliegenden Architektur ist eine solche Parallelität jedoch nur eingeschränkt realisiert. Lediglich bestimmte ALU- und Multipliziererinstruktionen sind gleichzeitig ausführbar; parallele Speicherzugriffe sind ebenfalls nicht für alle Lade- und Speicheroperationen möglich. Zudem hängt die Parallelisierbarkeit auch von der Registerzuteilung ab (vgl. Kapitel 2.3.4). Daher muß die Formulierung des ILP's im Vergleich zu der obigen Darstellung erweitert werden, um Einschränkungen der Parallelität erzwingen zu können. Eine Möglichkeit bestünde darin, den Ressourcenflußgraph durch Kollisionskanten zwischen jedem Paar von Operationen zu erweitern, die nicht gleichzeitig ausgeführt werden dürfen. Falls also zwei Instruktionen i und j nicht gleichzeitig ausgeführt werden können, muß eine zu $t_i \neq t_j$ äquivalente Nebenbedingung in das lineare Programm aufgenommen werden.

Da jedoch keine gegebene Anordnung zwischen i und j vorausgesetzt werden kann, kann diese Ungleichung **nicht** auf $t_i - t_j > 0$ oder $t_i - t_j < 0$ reduziert werden; es besteht – naiv betrachtet – nur die Möglichkeit der Formulierung in der Form $|t_i - t_j| > 0$ bzw. $(t_i - t_j)^2 > 0$. Diese Nebenbedingungen sind jedoch nicht mehr linear und können daher in dieser Form nicht verwendet werden.

Lösungsansatz: virtueller Synchronisationsbus

Das Problem der eingeschränkten Parallelausführung kann ebenfalls als Flußproblem dargestellt werden. Zu den Ressourcenknoten wird ein virtueller Synchronisationsbus B hinzugenommen, mit dessen Hilfe dann Serialisierungsbedingungen für Operationen auf verschiedenen funktionalen Einheiten formuliert werden können. Dieser Bus entspricht keinem in der Hardware tatsächlich vorhandenen Bus, da sich die Kombination zu Multifunktionsinstruktionen nicht auf die Nutzung der verschiedenen Systembusse reduzieren läßt.

Dazu müssen die Nebenbedingungen 5.13 und 5.12 wie folgt abgeändert werden:

$$\begin{aligned} \sum_{\substack{k \in V_K - \{B\}: \\ (j,k) \in E_R}} \Phi_j^k &= 1 \qquad \forall\, j \in V_D && (6.6) \\ \Phi_j^B &= 1 \qquad \forall\, j \in V_D^B && (6.7) \\ \Psi_j^B &= 1 \qquad \forall\, j \in V_D^B && (6.8) \end{aligned}$$

Jede Operation darf also auch weiterhin nur auf einem realen Ressourcentyp ausgeführt werden, kann aber gleichzeitig auch die Synchronisationsressource B beanspruchen. Das Gesetz der Flußerhaltung behält auch für B seine Gültigkeit.

Der Schlüssel zum Verständnis dieses Vorgehens liegt in der Definition der z_i-Werte (Gleichung 5.5). Die Werte L_{ij}^k werden im voraus für alle Paare von Instruktionen i und j berechnet; für Operationen, die gleichzeitig ausgeführt werden können, wird der L_{ij}^B-Wert einfach auf Null gesetzt. Somit müssen diese Operationen nicht serialisiert werden; die parallele Ausführung von i und j ist möglich.

Die Anzahl an neuen Serialisierungsbedingungen ist weiterhin dadurch beschränkt, daß der Serialisierungsbus nur bei Operationen verwendet wird, die auf verschiedenen Ressourcentypen ausgeführt werden. Bei den übrigen Instruktionspaaren verhindern bereits die Serialisierungsbedingungen bezüglich anderer Ressourcentypen eine Parallelausführung.

$$L_{ij}^B = 0 \qquad \forall\, i,j \in V_D \;\text{ mit }\; \exists\, k \neq B: \; (i,j)^k \in E_F^k \tag{6.9}$$

Die neuen Serialisierungsbedingungen erfordern eine Anpassung der Definition des Ressourcenflußgraphen, denn es müssen jetzt auch Flußkanten zwischen Operationen verlaufen, die verschiedene Ressourcentypen erfordern.

$$\begin{aligned} V_D^B &= \{i \mid i \in V_D \ \wedge \ \exists\, j \in V_D: \ L_{ij}^B = 1\} \\ E_F^B &= \{(i,j)^B \mid i,j \in V_D^B, \ i \not\prec j, \ j \not\prec i, \ i \neq j, res(i) \neq res(j)\} \end{aligned}$$

Damit kann die Nebenbedingung 5.15 ohne Veränderung übernommen werden, sie erfaßt nun aber auch die neu hinzugekommenen Fälle.

Es muß eine weitere Hardwareressource in die Betrachtung integriert werden, nämlich der virtuelle Synchronisationsbus B, so daß nun folgende Ressourcenknoten verwendet werden: *DM, PM, MU, S, C, B*.

Damit sind bei diesem Ansatz für jedes Paar nicht parallelisierbarer Operationen verschiedener Ressourcentypen zwei Binärvariablen erforderlich, die den Flußkanten zwischen den beiden Operationsknoten entsprechen. Zudem muß jeder Instruktionsknoten, der nur eingeschränkt parallelisierbar ist, ebenfalls über zwei Kanten mit dem Ressourcenknoten B verbunden werden. Dies erhöht die Anzahl der Binärvariablen höchstens um $2 \cdot |V_D|$. Bezeichnen wir nun mit dem Parameter P die Anzahl der Paare nicht parallelisierbarer Operationen verschiedener Ressourcentypen, beträgt die Anzahl zusätzlicher Binärvariablen $\mathcal{O}(|V_D|+2P)$. Da im schlimmsten Fall Parallelisierungsverbote zwischen allen Instruktionen zu formulieren sind, gilt $P = \mathcal{O}(|V_D|^2)$. Die asymptotische obere Schranke für die Anzahl der Binärvariablen erhöht sich damit im Vergleich zur Basisformulierung nicht; sie beträgt auch mit dieser Erweiterung $\mathcal{O}(|V_D|^2)$.

Zur Formulierung der Flußerhaltung sind $2 \cdot |V_D^B|$ Nebenbedingungen erforderlich, die Anzahl neuer Serialisierungsbedingungen entspricht $2 \cdot P$, so daß insgesamt $\mathcal{O}(2|V_D|+2P) = \mathcal{O}(|V_D| + P) = \mathcal{O}(|V_D|^2)$ Nebenbedingungen hinzugefügt werden müssen.

Dieses Verfahren harmonisiert zwar mit dem Flußgedanken, der dem Modellierungsansatz zugrundeliegt, dennoch motiviert der Anstieg der Binärvariablenzahl die Suche nach einem effizienteren Verfahren.

Lösungsansatz: Parallelisierungsverbote

Bei diesem zweiten Verfahren wird wieder auf die Ausgangsidee zurückgegriffen, nämlich der Formulierung von Nebenbedingungen, die explizit die parallele Ausführung nicht parallelisierbarer Instruktionen verhindern. Sind zwei Operationen i und j gegeben, für die $t_i \neq t_j$ gelten muß, so werden mit Hilfe von Binärvariablen Nebenbedingungen formuliert, die die Disjunktion $(t_i > t_j) \vee (t_i < t_j)$ abbilden. Hierzu sind folgende Ungleichungen erforderlich:

$$
\begin{aligned}
t_i - t_j &> -v_{ij}T && (6.10)\\
t_i - t_j &< (1 - v_{ij})T && (6.11)\\
v_{ij} &\in \{0, 1\} && (6.12)\\
T &= 2I + 1 && (6.13)
\end{aligned}
$$

Dabei bezeichnet $I = |V_D|$ die Anzahl der Knoten des Abhängigkeitsgraphen, die eine obere Schranke für die maximale Laufzeit darstellt. Der Grund hierfür ist, daß jede Instruktion genau einen Zyklus zur Ausführung benötigt und der schlimmste Fall die sequentielle Ausführung darstellt.

Beweis: Im folgenden soll die Korrektheit dieser Nebenbedingungen gezeigt werden.

1. Weder $t_i > t_j$ noch $t_i < t_j$ dürfen durch diese Bedingungen ausgeschlossen werden.

   (a) $v_{ij} = 1$: $\quad t_i - t_j > -T$
   $\quad t_i - t_j < 0$

   Da T eine obere Schranke für die maximale Ausführungszeit darstellt, ist die Ungleichung $t_i - t_j > -T$ immer erfüllt; erhält also v_{ij} im Laufe der Berechnung den Wert $v_{ij} = 1$, so muß $t_i < t_j$ gelten.

   (b) $v_{ij} = 0$: $\quad t_i - t_j > 0$
   $\quad t_i - t_j < \mathrm{T}$

   Analog zu Fall 1a ist die Ungleichung $t_i - t_j < T$ immer erfüllt; erhält also v_{ij} im Laufe der Berechnung den Wert $v_{ij} = 0$, so muß $t_i > t_j$ gelten.

2. Die Hinzufügung dieser Nebenbedingungen muß stets $t_i \neq t_j$ gewährleisten. Der Beweis hierzu wird indirekt geführt.

   Annahme: $\quad t_i = t_j$

$$
\begin{array}{lllll}
0 = & t_i - t_j & > & -v_{ij}T & \Rightarrow v_{ij} > 0 \\
 & & & & \Leftrightarrow v_{ij} = 1 \\
0 = & t_i - t_j & < & (1 - v_{ij})T & \Rightarrow 1 - v_{ij} > 0 \\
 & & & & \Leftrightarrow v_{ij} < 1 \\
 & & & & \Leftrightarrow v_{ij} = 0
\end{array}
$$

$$\Rightarrow v_{ij} = 1 \ \wedge \ v_{ij} = 0 \quad \Rightarrow \quad \text{Widerspruch!}$$

Also wird durch diese Formulierung die gleichzeitige Ausführung der Instruktionen i und j ausgeschlossen, ohne daß eine feste Ordnung zwischen den Startzeiten t_i und t_j impliziert wird. $\square$

Auf diese Weise ergeben sich im allgemeinen Fall für jedes Paar nicht parallelisierbarer Instruktionen verschiedener Ressourcentypen zwei zusätzliche Nebenbedingungen und eine zusätzliche Binärvariable. Mit dem in 6.2.2 eingeführten Parameter P ergibt sich hier, daß die Anzahl zusätzlicher Binärvariablen $\mathcal{O}(P) = \mathcal{O}(|V_D|^2)$ und die Anzahl zusätzlicher Nebenbedingungen $\mathcal{O}(2P) = \mathcal{O}(P) = \mathcal{O}(|V_D|^2)$ beträgt. Es brauchen jedoch nicht für alle nicht-parallelisierbaren Instruktionen explizite Parallelisierungsverbote formuliert zu werden. Ist Instruktion j von Instruktion i transitiv abhängig und besteht zwischen ihnen nicht nur eine Benutzungs-Setzungs-Abhängigkeit, so können sie in keinem zulässigen Lösungsschedule parallel ausgeführt werden. Wird bei der Globalitätsanalyse (vgl. Kapitel 6.1) festgestellt, daß i und j niemals im selben Basisblock stehen können, ist ein explizites Verbot ebenfalls überflüssig. Somit ist die Anzahl der notwendigen Parallelisierungsverbote im allgemeinen deutlich geringer als im schlimmsten Fall (vgl. Kapitel 8.2).

Beispiel 6.5 Gegeben sei erneut das folgende Eingabeprogramm:

```
1)   r1 = dm(i0, m0);
2)   r2 = pm(i8, m8);
3)   r3 = r1 * r2;
4)   MRF = MRF + r1 * r2;
5)   r4 = dm(i1, m1);
6)   r5 = pm(i9, m9);
7)   r6 = r4 + r5;
8)   r7 = min(r4, r5);
```

Die Minimumbildung ist zwar eine ALU-Instruktion, kann jedoch nie mit einer Multipliziererinstruktion parallel ausgeführt werden. Da zwischen den Multiplikationen in Instruktion 3 und 4 und der Minimumbildung in Instruktion 8 keine Datenabhängigkeiten bestehen, müssen Parallelitätsverbote erzeugt werden. Es ergeben sich die folgenden Ungleichungen:

$$\begin{aligned}
t_8 - t_3 + 17v_{83} &\geq 1 \\
t_8 - t_3 + 17v_{83} &\leq 16 \\
t_8 - t_4 + 17v_{84} &\geq 1 \\
t_8 - t_4 + 17v_{84} &\leq 16
\end{aligned}$$

□

6.2.3 Eingeschränkte Registermengenzugehörigkeit

Die Operanden von Multifunktionsinstruktionen, die sowohl ALU als auch Multiplizierer verwenden, sind auf eine Menge von vier Registern innerhalb des Registerfiles beschränkt (vgl. Kapitel 2.3.4). Damit liegt keine homogene Registermenge mehr vor, sondern es ergeben sich vier Gruppen, die aus jeweils vier Registern bestehen. Für jede dieser Vierergruppen wird ein eigener Registerknoten im Registerflußgraph benötigt. Die Menge G der zur Verfügung stehenden Register muß daher erweitert werden zu $G = \{g_1, g_2, g_3, g_4\}$, die die einzelnen Vierergruppen in aufsteigender Reihenfolge repräsentieren. Die Definition des Registerflußgraphen ist entsprechend anzupassen. Ferner müssen die Nebenbedingungen (5.27), (5.28) und (5.30) wie folgt geändert werden:

$$\Psi_g^g \leq R_g \qquad \forall\, g \in G \tag{6.14}$$
$$\sum_{g \in G} \Phi_j^g = 1 \qquad \forall\, j \in V_g \tag{6.15}$$
$$\Phi_j^g - \Psi_j^g = 0 \qquad \forall\, j \in V_F^g \quad \forall\, g \in G \tag{6.16}$$
$$t_j - t_i \geq w_i - w_j + \tau_i + (\sum_{g \in G} x_{ij}^g - 1) \cdot \alpha_{ij} \qquad \forall\, (i,j) \in E_F^g \tag{6.17}$$

Durch (6.14) wird gewährleistet, daß von jeder der vier Registermengen höchstens $R_g = 4$ Instanzen verwendet werden; die Nebenbedingung (6.15) garantiert, daß jede Variable genau einem Register zugeordnet wird. Die Flußerhaltung wird durch Nebenbedingung (6.16) aufrechterhalten, die Serialisierungsbedingungen ergeben sich nun aus (6.17).

Damit ist die Formulierung jedoch noch nicht abgeschlossen, denn es muß noch garantiert werden, daß, wenn aufgrund einer Multifunktionsinstruktion eine Variable auf eine Registermenge $g_i \in G$ beschränkt ist, die setzende Instruktion tatsächlich diese Registermenge g_i als Ziel verwendet.

Werden also die Instruktionen t_i und t_j zu einer Multifunktionsinstruktion kombiniert, so daß für eine zugehörige setzende Operation m die Zielregistermenge auf ein $g \in G$ beschränkt wird, muß eine Nebenbedingung der Form $\Phi_m^g \geq 1$ gültig werden. Da $\sum_g \Phi_j^g = 1$ gilt, ist damit gleichzeitig die Benutzung der anderen Registermengen ausgeschlossen. Wie soll jedoch die Abhängigkeit von der gleichzeitigen Ausführung formuliert werden?

Hierzu betrachten wir die folgenden drei Nebenbedingungen:

$$t_i - t_j \geq -p_{ij} T \tag{6.18}$$
$$t_i - t_j \leq q_{ij} T \tag{6.19}$$
$$p_{ij} + q_{ij} = 1 \tag{6.20}$$

Dabei sind p_{ij} und q_{ij} Binärvariable, und T hat wieder den Wert $T = 2I + 1$.

Behauptung:

1. Ist $p_{ij} = 1 \Leftrightarrow q_{ij} = 0$, dann gilt $t_i \leq t_j$.
2. Ist $q_{ij} = 1 \Leftrightarrow p_{ij} = 0$, dann gilt $t_i \geq t_j$.
3. Im Fall $t_i = t_j$ gibt es genau zwei mögliche Belegungen von p_{ij} und q_{ij}.
4. Für $t_i < t_j$ ergibt sich $p_{ij} = 1 \quad \Leftrightarrow \quad q_{ij} = 0$.
5. Für $t_i > t_j$ ergibt sich $p_{ij} = 0 \quad \Leftrightarrow \quad q_{ij} = 1$.

Beweis:

1. $\quad p_{ij} = 1 \quad \Leftrightarrow \quad q_{ij} = 0$, da $p_{ij} + q_{ij} = 1$
   $\Rightarrow \quad t_i - t_j \geq -T \quad \wedge \quad t_i - t_j \leq 0 \quad$ (aus (6.18), (6.19))
   $\Rightarrow \quad t_i \leq t_j$

2. $\quad p_{ij} = 0 \quad \Leftrightarrow \quad q_{ij} = 1$, da $p_{ij} + q_{ij} = 1$
   $\Rightarrow \quad t_i - t_j \geq 0 \quad \wedge \quad t_i - t_j \leq T \quad$ (aus (6.18), (6.19))
   $\Rightarrow \quad t_i \geq t_j$

3. Gilt $t_i = t_j$, so sind die neuen Nebenbedingungen sowohl für $p_{ij} = 1$ und $q_{ij} = 0$ als auch für $p_{ij} = 0$ und $q_{ij} = 1$ erfüllt.

   (a) Sei zunächst $p_{ij} = 1 \quad \Leftrightarrow \quad q_{ij} = 0$. Dann gilt:

   $$\begin{array}{rcrcrl} t_i - t_j & \geq & -p_{ij}T & \Leftrightarrow & 0 \geq -T & \quad \text{(wahr)} \\ t_i - t_j & \leq & q_{ij}T & \Leftrightarrow & 0 \leq \;\;\; 0 & \quad \text{(wahr)} \end{array}$$

   (b) Sei nun $p_{ij} = 0 \quad \Leftrightarrow \quad q_{ij} = 1$. Dann gilt:

   $$\begin{array}{rcrcll} t_i - t_j & \geq & -p_{ij}T & \Leftrightarrow & 0 \geq 0 & \quad \text{(wahr)} \\ t_i - t_j & \leq & q_{ij}T & \Leftrightarrow & 0 \leq T & \quad \text{(wahr)} \end{array}$$

4. Es gelte $t_i < t_j$. Dann ergibt sich:

   $$\begin{array}{rcrcllc} t_i - t_j & \geq & -p_{ij}T & \Rightarrow & p_{ij} = 1 & \Rightarrow & q_{ij} = 0 \\ t_i - t_j & \leq & 0 \cdot T & \Leftrightarrow & t_i - t_j \leq 0 & & \text{(wahr)} \end{array}$$

5. Es gelte $t_i > t_j$. Dann erhalten wir:

$$\begin{array}{rcrcccc} t_i - t_j & \geq & -p_{ij}T & \Rightarrow & p_{ij} = 0 & \vee & p_{ij} = 1 \\ t_i - t_j & \leq & q_{ij}T & \Rightarrow & q_{ij} = 1 & \wedge & p_{ij} = 0 \end{array}$$

$\square$

Unter Zuhilfenahme der so definierten p_{ij}- und q_{ij}-Werte können dann die Registerbeschränkungen mittels der beiden folgenden Nebenbedingungen formuliert werden:

$$\Phi_m^g \geq 1 - (t_i - t_j) - p_{ij}T \tag{6.21}$$

$$\Phi_m^g \geq 1 + (t_i - t_j) - q_{ij}T \tag{6.22}$$

Beweis:

Zu zeigen ist, daß die Nebenbedingungen (6.21) und (6.22) genau dann greifen, wenn $t_i = t_j$ gilt, und sowohl für $t_i < t_j$, als auch für $t_i > t_j$ keine Einschränkung darstellen.

1. Aus $t_i < t_j$ folgt sofort $p_{ij} = 1 \;\wedge\; q_{ij} = 0$. Damit ergeben sich (6.21) und (6.22) zu

$$(6.21) \Rightarrow \Phi_m^g \geq \underbrace{1 - \underbrace{(t_i - t_j)}_{\substack{\leq -1 \\ > -T+1}} - T}_{\leq 0}$$

$$(6.22) \Rightarrow \Phi_m^g \geq \underbrace{1 + \underbrace{(t_i - t_j)}_{\leq -1}}_{\leq 0}$$

Beide Nebenbedingungen sind somit stets erfüllt.

2. Aus $t_i > t_j$ folgt sofort $p_{ij} = 0 \;\wedge\; q_{ij} = 1$. Damit ergeben sich (6.21) und (6.22) zu

$$\begin{array}{lcccl}
(6.21) & \Rightarrow & \Phi_m^g & \geq & \underbrace{1 - \underbrace{(t_i - t_j)}_{\geq 1}}_{\leq 0} \\[2em]
(6.22) & \Rightarrow & \Phi_m^g & \geq & \underbrace{1 + \underbrace{(t_i - t_j)}_{\substack{\geq 1 \\ < T-1}} - T}_{\leq 0}
\end{array}$$

   Also sind wieder beide Nebenbedingungen erfüllt.

3. Für den Fall $t_i = t_j$ müssen wir zwei Fälle unterscheiden, entsprechend der jeweiligen Belegung von p_{ij} und q_{ij}.

   (a) Sind $p_{ij} = 1$ und $q_{ij} = 0$, so erhalten wir

$$\begin{array}{lcccll}
(6.21) & \Rightarrow & \Phi_m^g & \geq & 1 - T & \quad \text{(immer wahr)} \\
(6.22) & \Rightarrow & \Phi_m^g & \geq & 1 & \\
 & \Rightarrow & \Phi_m^g & \geq & 1 &
\end{array}$$

   Also muß das Ergebnis von Instruktion m in Registermenge g abgelegt werden.

   (b) Im anderen Fall $p_{ij} = 0$ und $q_{ij} = 1$ wird die erste Nebenbedingung scharf, so daß auch hier die Einschränkung der Registermenge für Instruktion j in Kraft tritt.

$$\begin{array}{lcccll}
(6.21) & \Rightarrow & \Phi_m^g & \geq & 1 & \\
(6.22) & \Rightarrow & \Phi_m^g & \geq & 1 - T & \quad \text{(immer wahr)} \\
 & \Rightarrow & \Phi_m^g & \geq & 1 &
\end{array}$$

□

Durch diese Nebenbedingungen können bei sequentiellem Eingabecode die Probleme der Instruktionsanordnung und der Registerallokation in einer einheitlichen Problembeschreibung integriert werden. Lediglich die Aufspaltung von Lebensspannen oder die Einfügung von Spillcode werden nicht berücksichtigt (vgl. hierzu Kapitel 6.7). Zwar wird innerhalb der Nebenbedingungen nur auf die vier Registermengen Bezug genommen; dennoch kann auch eine konkrete Registerzuteilung durchgeführt werden. Dies wird an Beispiel 6.6 veranschaulicht.

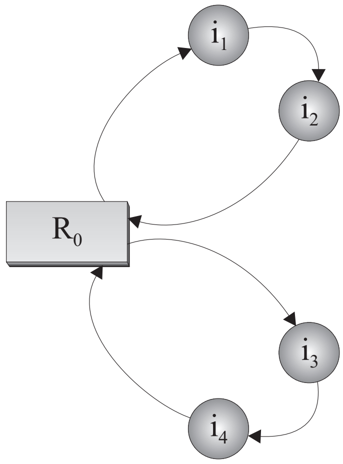


Abbildung 6.2: Resultatregisterfluß.

Beispiel 6.6 Als Ergebnis eines Optimierungsprozesses sei folgender Registerfluß bezüglich der Registermenge R_0 gegeben. Hierdurch wird festgelegt, daß die Instruktionen $i_1, \ldots, i_4$ Register der Registermenge R_0 beschreiben sollen, und zwar i_1 und i_2 ein anderes als i_3 und i_4. Die Registerverteilung wird dann durchgeführt, indem z.B. als Resultatregister bei i_1 und i_2 das Register `r0`, bei i_3 und i_4 das Register `r1` eingetragen wird. Anschließend müssen nur noch die zugehörigen angewandten Vorkommen an die definierenden Vorkommen angepaßt werden.

□

6.3 Anpassung von OASIC an den ADSP-2106x

6.3.1 Berücksichtigung der eingeschränkten Parallelität

Es seien zwei Instruktionen i und j gegeben, die nicht parallel ausgeführt werden dürfen, so daß $t_i \neq t_j$ gelten muß. Können i und j niemals im gleichen Basisblock stehen oder wird eine Parallelausführung durch Datenabhängigkeiten ausgeschlossen, so sind keine zusätzlichen Nebenbedingungen erforderlich. Ansonsten sind folgende Nebenbedingungen zum linearen Programm hinzuzufügen:

$$\sum_{k \in V_K:(i,k) \in E_R} x_{in}^k + \sum_{k \in V_K:(i,k) \in E_R} x_{jn}^k \leq 1 \qquad \forall\, n \in N(i) \cap N(j) \tag{6.23}$$

Beispiel 6.7 In diesem Beispiel werden für die bereits bekannte Beispielcodesequenz die Nebenbedingungen angegeben, die in der OASIC-Formulierung zur Modellierung der Parallelitätsverbote erforderlich sind. Zur Erinnerung:

```
1)   r1 = dm(i0, m0);
2)   r2 = pm(i8, m8);
3)   r3 = r1 * r2;
4)   MRF = MRF + r1 * r2;
5)   r4 = dm(i1, m1);
6)   r5 = pm(i9, m9);
7)   r6 = r4 + r5;
8)   r7 = min(r4, r5);
```

Dann sind die folgenden Nebenbedingungen erforderlich:

$$\begin{array}{llllllllllll}
x_{82}^S + x_{32}^{MU} & \leq & 1 & \wedge & x_{83}^S + x_{33}^{MU} & \leq & 1 & \wedge & x_{84}^S + x_{34}^{MU} & \leq & 1 & \wedge \\
x_{85}^S + x_{35}^{MU} & \leq & 1 & \wedge & x_{86}^S + x_{36}^{MU} & \leq & 1 & \wedge & x_{87}^S + x_{37}^{MU} & \leq & 1 & \wedge \\
x_{88}^S + x_{38}^{MU} & \leq & 1 & \wedge & x_{82}^S + x_{42}^{MU} & \leq & 1 & \wedge & x_{83}^S + x_{43}^{MU} & \leq & 1 & \wedge \\
x_{84}^S + x_{44}^{MU} & \leq & 1 & \wedge & x_{85}^S + x_{45}^{MU} & \leq & 1 & \wedge & x_{86}^S + x_{46}^{MU} & \leq & 1 & \wedge \\
x_{87}^S + x_{47}^{MU} & \leq & 1 & \wedge & x_{88}^S + x_{48}^{MU} & \leq & 1 & \wedge & & & &
\end{array}$$

□

6.3.2 Eingeschränkte Registermengenzugehörigkeit

Die Formulierung der in Kapitel 5.2.2 vorgestellten Nebenbedingungen zur Registerallokation kann nicht uneingeschränkt übernommen werden, da das Registerfile in Teilmengen zerfällt, die gesondert berücksichtigt werden müssen. Dazu sind neue Variablen erforderlich. Falls Instruktion i einen Schreibzugriff auf ein Register ausführt sind die zugehörigen Binärvariablen x_{in}^{k} durch Binärvariablen x_{in}^{km} gemäß der folgenden Gleichung zu ersetzten:

$$x_{in}^{k} = \sum_{m=1}^{4} x_{in}^{km}$$

Wird Instruktion i in Kontrollschritt n auf Ressourcentyp k ausgeführt und dabei ein Schreibzugriff auf ein Register der Registermenge m vollzogen, so gilt $x_{in}^{km} = 1$. Folgende zusätzliche Nebenbedingungen sind erforderlich:

- In jedem Register darf zu einem Zeitpunkt nur eine Variable gespeichert werden, in jeder der vier Teilmengen des Registerfiles also höchstens $R_m = 4$.

$$\sum_{(k,j)\in E_R} \sum_{j:n\in N(j)} x_{jn}^{km} \leq R_m \qquad \forall\, m \quad \forall\, n \tag{6.24}$$

- Für jede Registermenge darf die Anzahl der Kreuzungskanten zu jedem Kontrollschritt nicht die Anzahl der verfügbaren Register überschreiten.

$$\begin{aligned}\sum_{j_a \prec j_b} \Big(\sum_{k\in V_K} \sum_{\substack{n_1 \leq n \\ n_1\in N(j_a)}} x_{j_a n_1}^{km} + \sum_{k\in V_K} \sum_{\substack{n_2 > n \\ n_2\in N(j_b)}} x_{j_b n_2}^{km} \; - \\ \sum_{k\in V_K} \sum_{\substack{n_3 \leq n \\ n_3\in N(j_b)}} x_{j_b n_3}^{km} - \sum_{k\in V_K} \sum_{\substack{n_4 > n \\ n_4\in N(j_a)}} x_{j_a n_4}^{km} \Big) \leq 2\cdot R_m \qquad \forall n \quad \forall m\end{aligned} \tag{6.25}$$

- Wird Operation l wegen der Kombination von i und j zu einer Multifunktionsinstruktion auf Registermenge m_l beschränkt, so sind folgende Nebenbedingungen aufzunehmen:

$$t_i - t_j \geq -p_{ij}T \tag{6.26}$$

$$t_i - t_j \leq q_{ij}T \tag{6.27}$$

$$p_{ij} + q_{ij} = 1 \tag{6.28}$$

$$\sum_{n\in N(l)} x_{ln}^{km_l} \geq 1 - (t_i - t_j) - p_{ij}T \tag{6.29}$$

$$\sum_{n\in N(l)} x_{ln}^{km_l} \geq 1 + (t_i - t_j) - q_{ij}T \tag{6.30}$$

Dabei gelte $T = 2|V_D| + 1$ und $p_{ij}, q_{ij} \in \{0,1\}$. Diese Formulierung ist analog zu der in Abschnitt (6.2.3) vorgestellten; der Korrektheitsbeweis ist direkt übertragbar. Diese Nebenbedingungen greifen also genau für den Fall $t_i = t_j$ und erzwingen dann den Schreibzugriff von Instruktion l auf ein Register der Registermenge m_l.

Durch Verwendung dieser Formulierung kann die optimale Registermengenzuweisung ermittelt werden. Zudem ist durch die Struktur der Nebenbedinungen garantiert, daß genug Register vorhanden sind, um den Ausgabecode ohne zusätzliche Zwischenspeicherungen auszuführen. Es wird keine konkrete Registerzuteilung berechnet; diese muß einem nachgeschalteten Registerallokator überlassen werden, der die ermittelte Registermengenzugehörigkeit berücksichtigt.

Der Grund für den Verzicht auf die Integration der Registerzuteilung in die ILP-Formulierung liegt in der Komplexität des zu erstellenden ILP's. Alle Entscheidungsvariablen müssen ganzzahlige Werte besitzen. Wird einer solchen Variable bei Betrachtung einer Relaxation (vgl. Kapitel 5.2.3) ein nicht-ganzzahliger Wert zugeordnet, muß sie als binär spezifiziert werden. Es ist offensichtlich, daß im Branch-and-Bound-Baum im schlimmsten Fall 2^n Verzweigungsknoten berechnet werden müssen, wobei n die Anzahl der als binär spezifizierten Variablen bezeichnet. Wieviele Binärvariablen sind nun zu berücksichtigen? Bei der Modellierung einer homogenen Registermenge werden für eine Instruktion j $N(j)$ Binärvariablen definiert, nämlich eine für jeden Kontrollschritt, zu dem j ausgeführt werden kann. Wird zusätzlich die Zuordnung zu den vier Registermengen modelliert, so ergeben sich $4N(j)$ Binärvariablen für jede Instruktion j. Würde bezüglich jedes Registers eine eigene Binärvariable definiert, wären für eine Instruktion j genau $16N(j)$ Binärvariablen erforderlich. Es gilt $\mathcal{O}(2^{N(j)}) \subset \mathcal{O}(2^{16 \cdot N(j)}) = \mathcal{O}((2^{N(j)})^{16})$. Der Anstieg des Berechnungsaufwandes wäre somit hoch. Zudem würde sich die ohnehin schon große Anzahl an Registerkreuzungsbedingungen gegenüber der Beschränkung auf die Registermengenzuordnung vervierfachen. Die Testergebnisse (vgl. Kapitel 8.2) zeigen, daß die Komplexität der Formulierung bei der alleinigen Berücksichtigung der Registermengenzuordnung zu erheblichen Dateigrößen und hohen Laufzeiten führt. Würde jedes einzelne Register gesondert modelliert, so bestünde nur bei Eingabeprogrammen geringerer Größe Aussicht auf Ermittlung der optimalen Lösung.

6.4 Registerallokation unter Berücksichtigung von Schleifen

Die bisher vorgestellten Modellierungen sind nur zur Registerallokation auf Basisblockebene, d.h. bei linearem Programmfluß einsetzbar. Der Grund hierfür ist, daß bei Schleifen die bisherige Darstellung der Lebenszeiten nicht ausreichend ist. Das läßt sich mit folgender Codesequenz verdeutlichen:

```
1)       r1 = 10;
2)       lcntr = r15, do L until LCE;
3)       r2 = r1 + r2;
4)    L: r1 = r1 + 1;
5)       r3 = r1 + 1;
```

Der Wert des Registers `r15` legt die Iterationsanzahl fest, der Schleifenrumpf besteht aus den Anweisungen 3 und 4 (vgl. auch Kapitel 2.10.3). Wie sollen hier die Lebenszeiten definiert werden? Zur Veranschaulichung werden im folgenden Lebenszeiten durch Kanten von der Definition eines (symbolischen) Registers zu den angewandten Vorkommen dargestellt. Werden die in Abbildung 6.3 dargestellten lebenszeitdefinierenden Kanten berücksichtigt, so geht die Information verloren, daß das Zielregister von i_1 mit dem Operandenregister von i_5 übereinstimmen muß. Damit wäre folgende Resultatsequenz zulässig, die jedoch eine

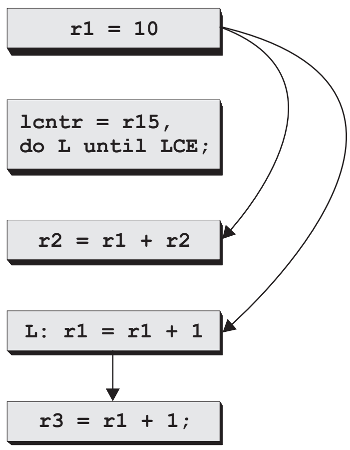


Abbildung 6.3: Fehlerhafte Lebenszeiten.

andere Semantik als das Eingabeprogramm besitzt. (Hier ist zusätzlich noch das Problem zu beachten, daß innerhalb des Schleifenrumpfes Variablen definiert werden, die in der nächsten Iteration verwendet werden; darauf wird im folgenden genauer eingegangen.)

```
1)        r1 = 10;
2)        lcntr = r15, do L until LCE;
3)        r2 = r1 + r2;
4)     L: r4 = r1 + 1;
5)        r3 = r4 + 1;
```

Es ergibt sich also ein falsches Resultat.

Wird jedoch die schleifenübergreifende Lebenszeit mitberücksichtigt (vgl. Abbildung 6.4), könnte Instruktion i_4 als Ziel nicht Register $r1$ verwenden, da die Lebenszeit von i_1 noch nicht beendet ist. Auch auf diese Weise ist keine korrekte Behandlung möglich.

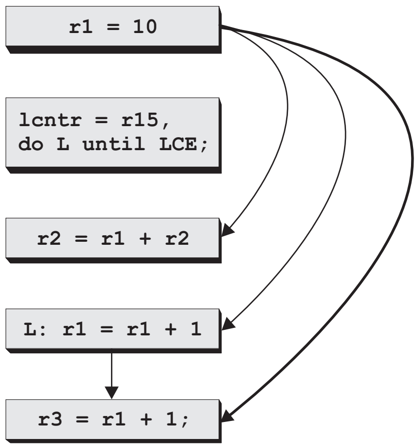


Abbildung 6.4: Fehlerhafte Lebenszeiten.

Daher muß der Erstellung der lebenszeitdefinierenden Nebenbedingungen besondere Aufmerksamkeit gewidmet werden. Tritt sowohl im Schleifenrumpf als auch hinter der Schleife ein angewandtes Vorkommen eines vor Schleifenantritt auszuführenden definierenden Vorkommens auf, so darf die schleifenumspannende Setzungs-Benutzungsabhängigkeit nicht zur Definition einer Lebenszeit herangezogen werden. Beim SILP-Ansatz ergeben sich für unser Beispiel folgende Nebenbedingungen zur Lebenszeiterfassung:

$$
\begin{aligned}
t_3 - t_1 - b_{13} &= 1 \\
t_5 - t_4 - b_{45} &= 1 \\
\tau_i - b_{13} &\geq 0 \\
\tau_4 - b_{45} &\geq 0
\end{aligned}
$$

Im Rahmen der OASIC-Modellierung wird die schleifenumspannende Setzungs-Benutzungsabhängigkeit nicht als lebenszeitdefinierend berücksichtigt und geht daher nicht in die Registerkreuzungsbedingungen (Nebenbedinungen 5.39) ein.

Zusätzlich muß gewährleistet werden, daß alle definierenden Vorkommen eines angewandten Vorkommens dasselbe Zielregister verwenden; dazu wird die Information der schleifenumspannenden Setzungs-Benutzungsabhängigkeit benötigt. Innerhalb der SILP-Formulierung kann dies dadurch erreicht werden, daß im Resultatregisterfluß die erste setzende Instruktion (i_1) ein Vorgänger der zweiten (i_2) sein muß. Hierzu müßte eine Disjunktion über alle möglichen Wege von i_1 nach i_2 formuliert werden und zwar für jede Registermenge. Da solche Disjunktionen nur mit Hilfe von Binärvariablen formuliert werden können, wäre der Komplexitätsanstieg so beträchtlich, daß von dieser Exaktheit Abstand genommen wird, um zumindest bezüglich der Registermengenzuteilung in akzeptablen Rechenzeiten zu einem Ergebnis zu gelangen. An folgendem Beispiel soll gezeigt werden, daß bereits in einem sehr einfachen Spezialfall erheblicher Aufwand erforderlich ist.

Beispiel 6.8 Flußkanten müssen im allgemeinen Fall zwischen jedem Paar von Operationen in beide Richtungen und für alle Registermengen gezogen werden; besteht zwischen zwei Operationen eine Datenabhängigkeit, so können die Kanten, die in Gegenrichtung verlaufen, weggelassen werden. Jede Kante entspricht bekanntlich einer Flußvariablen. In diesem Beispiel seien drei Instruktionen i_1, i_2 und j gegeben, wobei i_1 und i_2 definierende Vorkommen einer Variablen sein sollen, die in Instruktion j verwendet wird. i_1 werde vor Schleifeneintritt ausgeführt, und i_2 innerhalb des Schleifenrumpfes. Es sei angenommen, daß zwischen i_1 und i_2 eine Datenabhängigkeit besteht. Der zugehörige Registerflußgraph ist in Abbildung 6.5 dargestellt.

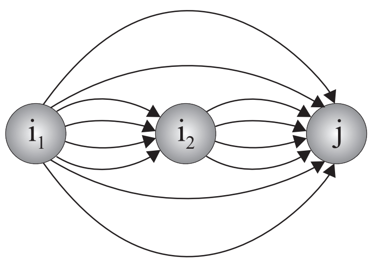


Abbildung 6.5: Der Registerflußgraph zu Beispiel 6.8.

Um nun sicherzustellen, daß i_1 und i_2 dasselbe Register beschreiben, ist folgende Disjunktion zu formulieren; die vier Registermengen werden dabei als $R_0, \ldots, R_3$ bezeichnet.

$$\begin{aligned} &(x_{i_1 i_2}^{R_0} = 1 \ \wedge \ x_{i_2 j}^{R_0} = 1) \ \vee \ (x_{i_1 i_2}^{R_1} = 1 \ \wedge \ x_{i_2 j}^{R_1} = 1) \\ \vee\ &(x_{i_1 i_2}^{R_2} = 1 \ \wedge \ x_{i_2 j}^{R_2} = 1) \ \vee \ (x_{i_1 i_2}^{R_3} = 1 \ \wedge \ x_{i_2 j}^{R_3} = 1) \end{aligned}$$

Würde dies als Nebenbedingung in einem linearen Programm formuliert, wären vier Binärvariablen erforderlich. Enthielte die Schleife eine weitere Instruktion, die einen Schreibzugriff auf ein Register durchführt, so müßte zusätzlich ein weiterer Fall berücksichtigt werden: auch diese Instruktion könnte dasselbe Register beschreiben. Somit wären weitere

Kantenfolgen zu berücksichtigen, über die das Register transportiert werden könnte. Dazu wären acht Binärvariablen erforderlich. Die Anzahl dieser disjunktiven Binärvariablen kann exponentiell zur Anzahl der zu betrachtenden Instruktionen anwachsen.

□

Aus diesem Grund werden in die ILP-Formulierung lediglich die folgenden Nebenbedingungen aufgenommen, die zumindest garantieren, daß beide setzenden Instruktionen dieselbe Registermenge verwenden; hierfür werden keine Binärvariablen benötigt.

$$\Phi_{i_1}^{g_k} - \Phi_{i_2}^{g_k} = 0 \qquad \forall\, k \in \{0, \ldots, 3\} \tag{6.31}$$

Diese Nebenbedingungen sind für alle definierenden Vorkommen eines angewandten Vorkommens zu erzeugen. Die genaue Registerverteilung muß nun ähnlich wie bei OASIC einem nachgeschalteten Registerallokator überlassen werden, der unter Berücksichtigung der aus der Lösung des ILP's bestimmten Registermengenzugehörigkeit eine exakte Lösung bestimmt.

Bei OASIC wird die Zuordnung zur richtigen Registermenge durch folgende Gleichung erzwungen:

$$\sum_{n \in N(i)} x_{in}^{kg_m} - \sum_{n \in N(j)} x_{jn}^{kg_m} = 0 \qquad \forall\, m \in \{1, \ldots, 4\} \tag{6.32}$$

Es ist noch eine weitere Problematik zu beachten: Werden innerhalb einer Schleife Setzungen ausgeführt, zu denen angewandte Vorkommen in der nächsten Schleifeniteration auftreten, so muß gewährleistet werden, daß auch diese setzenden bzw. angewandten Vorkommen auf die gleichen Register zugreifen. Der Unterschied zu den "normalen" Abhängigkeiten liegt darin, daß in diesem Fall die Lebenszeit bei einer Instruktion definiert wird, die im Programmtext, also bei normaler sequentieller Ausführung **nach** dem angewandten Vorkommen steht. Zur Formulierung der Nebenbedingungen (6.31) und (6.32) müssen also auch diese Abhängigkeiten berücksichtigt werden. Es ist jedoch noch eine Erweiterung erforderlich: Existieren zu einer Setzung nur angewandte Vorkommen in der nächsten Schleifeniteration, wird noch keine Lebenszeit für diesen Wert erzeugt. Nachfolgenden Setzungen innerhalb der Schleife könnten daher demselben Zielregister zugeordnet werden, was zu einem falschen Ausgabecode führen würde. Bei der SILP-Formulierung ist daher für eine solche Setzung eine Lebenszeit bis zum Schleifenende zu erzeugen. Bei OASIC wird ein künstlicher Schleifenendknoten erzeugt und auf diese Weise die erforderliche lebenszeitdefinierende Kante generiert, die dann in den Registerkreuzungsbedingungen zu berücksichtigen ist.

Beispiel 6.9 Bei der Codesequenz der letzten Beispiele sind auch Abhängigkeiten zwischen Schleifeniterationen zu berücksichtigen. Diese sind in Abbildung 6.6 dargestellt. Der sequentielle Programmfluß wird hierbei durch die dünnen Kanten, die Abhängigkeiten durch die fett gedruckten Kanten widergegeben.

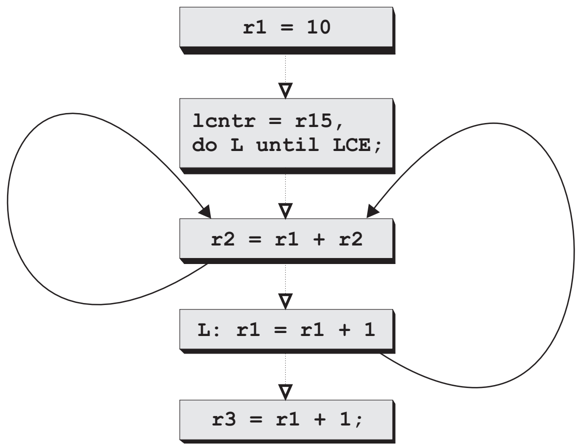


Abbildung 6.6: Beispiel für Schleifenabhängigkeiten zwischen zwei Iterationen.

□

Die zuvor vorgestellten Nebenbedingungen zur Modellierung lokaler Beschränkungen der Registermengen brauchen zur Berücksichtigung mehrerer definierender Vorkommen eines angewandten Vorkommens nicht verändert zu werden. Die Nebenbedingungen 6.14 bis 6.22 bzw. 6.26 bis 6.30 werden in Abhängigkeit einer möglichen Multifunktionsinstruktion für ein beliebiges definierendes Vorkommen erzeugt. Da aufgrund von (6.31) bzw. (6.32) alle definierenden Vorkommen dieselbe Registermenge beschreiben müssen, ist hierdurch auch das Zielregister der anderen definierenden Vorkommen bestimmt.

6.5 Registerallokation unter Berücksichtigung bedingter Anweisungen

Soll für eine Codesequenz, die bedingte Anweisungen enthält, Instruktionsanordnung mit verschränkter Registerallokation durchgeführt werden, muß der Modellierung der Lebenszeiten besondere Aufmerksamkeit gewidmet werden, ähnlich wie bei der Schleifenbehandlung (siehe Kapitel 6.4). Es sei die in Abbildung 6.7 dargestellte Codesequenz gegeben; die Kanten entsprechen den zu berücksichtigenden Setzungs-Benutzungsabhängigkeiten. Kein Pfad durch den Kontrollflußgraph kann sowohl Instruktion i_3 als auch i_5 enthalten.

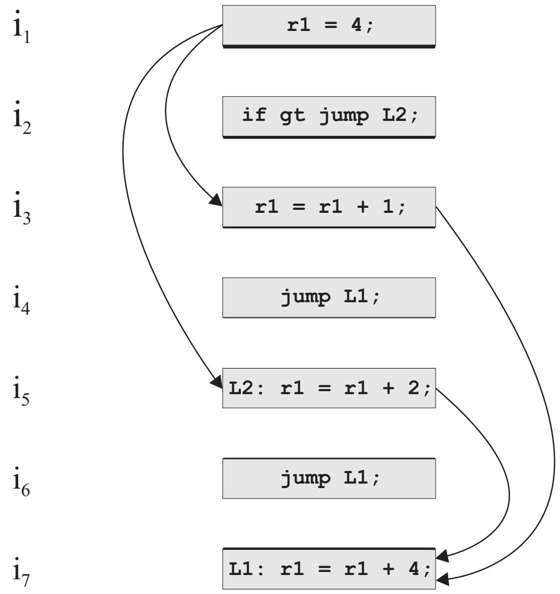


Abbildung 6.7: Problemfall.

Aufgrund der Fixierung der Basisblockstruktur wird in jedem Lösungsschedule i_3 vor i_5 angeordnet. Ohne Anpassung der ILP-Formulierung wäre die Verwendung von `r1` als Zielregister für i_3 nicht möglich, da `r1` auch als Operand in der "nachfolgenden" Instruktion i_5 verwendet wird. Somit darf die Kante (i_1, i_3) bei Betrachtung von i_5 nicht als lebenszeitdefinierend betrachtet werden. Es müssen also für alle Instruktionen, die keinen gemeinsamen Kontrollpfad besitzen, verschiedene Lebenszeiten berücksichtigt werden.

Zunächst werden die Auswirkungen auf die SILP-Formulierung betrachtet. Treten für ein definierendes Vorkommen in einer Instruktion i zwei angewandte Vorkommen j_1 und j_2 auf, die niemals auf demselben Kontrollpfad liegen können, so muß bezüglich j_1 eine Lebenszeit τ_i^1 und bezüglich j_2 eine Lebenszeit τ_i^2 definiert werden. Bezogen auf die Codesequenz aus Abbildung 6.7 ergeben sich die folgenden Nebenbedingungen zur Lebenszeitdefinition:

$$
\begin{aligned}
t_3 - t_1 - b_{13} &= 1 \\
t_5 - t_1 - b_{15} &= 1 \\
t_7 - t_3 - b_{37} &= 1 \\
t_5 - t_3 - b_{35} &= 1 \\
\tau_1^1 - b_{13} &\geq 0 \\
\tau_1^2 - b_{15} &\geq 0
\end{aligned}
$$

Desweiteren ist die Formulierung der Registerserialisierungsbedinungen anzupassen. Zwischen sich gegenseitig ausschließenden Instruktionen i und j darf keine Lebenszeit berücksichtigt werden. Bei Instruktionen i und j auf einem gemeinsamen Kontrollpfad dürfen zur Erzeugung der Registerserialisierungsbedingungen keine lebenszeitdefinierenden Kanten berücksichtigt werden, die durch Instruktionen erzeugt werden, welche niemals auf dem gleichen Pfad durch den Kontrollflußgraph wie i und j liegen können. In unserem Beispiel ergeben sich unter Vernachlässigung von i_7 folgende Registerserialisierungsbedingungen:

$$\begin{aligned} t_5 - t_1 &\geq \tau_1^2 + (\sum_{m=0}^{3} x_{15}^{R_m} - 1) \cdot 2T \\ t_3 - t_1 &\geq \tau_1^1 + (\sum_{m=0}^{3} x_{13}^{R_m} - 1) \cdot 2T \\ t_5 - t_3 &\geq (\sum_{m=0}^{3} x_{35}^{R_m} - 1) \cdot 2T \end{aligned}$$

Schließlich muß noch gewährleistet werden, daß – ähnlich wie bei der Schleifenbehandlung – alle definierenden Vorkommen eines angewandten Vorkommens dasselbe Register beschreiben. Hierzu ist das in Kapitel 6.4 geschilderte Vorgehen direkt übertragbar; es gelten dieselben Komplexitätsüberlegungen.

Auch bei der OASIC-Formulierung werden Änderungen erforderlich. Die Registerkreuzungsbedingungen (5.39) wurden bisher für alle maximalen Mengen $M'(n)$ von Kanten erzeugt, die Kontrollschritt c kreuzen und paarweise verschiedene Ausgangspunkte besitzen. Die Definition der $M'(n)$ muß nun dahingehend erweitert werden, daß in jeder Menge $M'(n)$ nur Kantenendpunkte auftreten, die zu Instruktionen gehören, welche gemeinsam auf einem Pfad durch den Kontrollflußgraph liegen können. Die Anzahl der Registerkreuzungsbedingungen steigt also weiter an.

Die Informationen, die benötigt werden, um obige Nebenbedingungen zu erstellen, können anhand des Kontrollabhängigkeitsgraphen berechnet werden. Eine solche Betrachtung von Registerallokation in Anwesenheit bedingter Anweisungen wurde jedoch aus verschiedenen Gründen nicht in die Implementierung aufgenommen. Typische DSP-Anwendungen betreffen die Durchführung von Berechnungen auf großen Datenmengen. Meist werden diese innerhalb von Schleifen durchgeführt; bedingte Anweisungen umfassen dann solche Schleifen oder große Basisblöcke. Die Laufzeitergebnisse von Kapitel 8.2 legen nahe, die ILP-Optimierung unter Berücksichtigung von Registerallokationsaspekten vorwiegend auf Schleifen- oder Basisblockebene durchzuführen, denn die Analyse vollständiger Prozeduren würde meist zu viel Zeit beanspruchen. Die Komplexität der zusätzlichen Bedingungen entspricht denen der zur Schleifenbehandlung erforderlichen Nebenbedingungen, so daß diese Laufzeitergebnisse auf Codesequenzen mit bedingten Anweisungen übertragen werden können. Dies gilt für die SILP-Modellierung; innerhalb der OASIC-Formulierung würde die Anzahl der Registerkreuzungsbedingungen weiter ansteigen. Diese ist nun bereits in der normalen Formulierung im schlimmsten Falle exponentiell, so daß die Analyse bedingter Anweisungen nur auf kleinen Eingabeprogrammen durchführbar wäre. Die praktische Relevanz dieser Untersuchungen wäre also gering.

Aus diesen Gründen werden im Rahmen der Implementierung Codesequenzen mit bedingten Anweisungen nur ohne Einbeziehung von Registerallokationsbedingungen analysiert. Konzeptionell ist eine entsprechende Erweiterung jedoch ohne weiteres realisierbar.

6.6 Unterstützung des Dual-Speichers

Sowohl bei SILP als auch bei OASIC ist durch die Art der Formulierung die Möglichkeit einer Ressourcenallokation gegeben. Es ist jedoch nicht möglich, die Zuteilung von Werten an Speicherbänke mit Hilfe solcher Allokationsbedingungen in das ILP zu integrieren. Diese Bedingungen sind ausgerichtet auf die Zuordnung einzelner Instruktionen auf unabhängige funktionale Einheiten. Kann eine Instruktion sowohl von einem Multiplizierer, als auch von einer ALU ausgeführt werden, ist es letztlich egal, wo die Ausführung der Instruktion erfolgt. Speichert jedoch eine Instruktion einen Wert in eine bestimmte Speicherbank, so müssen alle nachfolgenden lesenden Speicherzugriffe ebenfalls auf diese Speicherbank zugreifen. Wird innerhalb der ILP-Optimierung jede Instruktion der Speicherbank zugeordnet, die maximale Parallelität ermöglicht, könnte eine Speicheroperation dem DM, eine nachfolgende Ladeoperation für diesen Wert jedoch dem PM zugeordnet werden.

Der Zusammenhang zwischen Lade- und Speicheroperationen geht also verloren und kann auch durch zusätzliche Nebenbedingungen nicht modelliert werden. Der Grund hierfür liegt im **Alias-Problem**; auf jede Speicherzelle kann auf verschiedene Weisen zugegriffen werden, z.B. mittels verschiedener Indexregister oder durch direktes Adressieren. Aus der Maschinensprache läßt sich somit nicht für alle Ladeoperationen ermitteln, welcher Speicherzugriff den Wert geschrieben hat, der nun geladen werden soll – zumal sich dieser auch in einer anderen Prozedur befinden kann. Die Entscheidung, welcher Speicherbank eine Instruktion zuzuordnen ist, muß also auf Informationen der Hochsprachenebene zurückgreifen. Da als Eingaben der ILP-Optimierung bei unseren Untersuchungen Assemblercodesequenzen ohne zusätzliche semantische Informationen verwendet werden, wurde die Zuteilung von Instruktionen zu Speicherbänken nicht in die ILP-Formulierung integriert.

6.7 Automatische Modellierung von Zwischenspeicherungen

Werden im Eingabeprogramm symbolische Register verwendet, so kann es passieren, daß zu wenige reale Register vorhanden sind, um das Programm auf der Zielmaschine auszuführen. Dieser Fall kann nicht auftreten, wenn als Eingabe Assemblerprogramme verwendet werden, die mit Hilfe von `g21k` (vgl. hierzu Kapitel 8.2) erzeugt wurden; in diesem Fall wurde bereits eine Registerallokation durchgeführt. Bei Verwendung manuell erstellter Instruktionssequenzen kann dies jedoch nicht immer vorausgesetzt werden, so daß in der Regel von symbolischen Registern auszugehen ist. Konventionelle graphbasierte Registerallokatoren verwenden Methoden wie das Aufspalten von Lebensspannen oder das Einfügen von Zwischenspeicherungen, um eine Allokation von symbolischen Registern auf reale Register vorzunehmen. Dieses Vorgehen kann jedoch bei den untersuchten Formulierungen nicht innerhalb eines einheitlichen ILP's modelliert werden. Der Grund hierfür ist, daß sich die erstellten linearen Programme stets auf eine feste Menge von Instruktionen beziehen. Werden Instruktionen hinzugefügt oder entfernt, muß das ILP neu aufgebaut werden. Als Konsequenz ist es auch mit Hilfe der vorgestellten Methodik nicht möglich, eine Lösung zu berechnen, die bezüglich der Instruktionsanordnung, der Registerzuteilung und bezüglich des Aufspaltens von Lebensspannen oder des Einfügens von Zwischenspeicherungen im Rahmen der Registerallokation optimal ist. Sind die vorhandenen realen Register nicht ausreichend, um das Eingabeprogramm korrekt auszuführen, ist das erstellte ILP unlösbar und es muß durch Einsatz eines konventionellen Registerallokators bestimmt werden, welche Lebenszeit aufzuspalten ist.

Bei der SILP-Modellierung kann dieser Prozeß gesteuert werden, indem eine Lösung erzeugt wird, aus der eine Empfehlung für die aufzuspaltende Lebensspanne abgelesen werden kann. Dazu muß die Formulierung an einigen Stellen abgeändert werden. Anstelle der bisherigen Serialisierungsbedingungen wird folgende Ungleichung verwendet:

$$t_j - t_i \geq w_i - w_j + \tau_i - z_{ij} + (x_{ij}^g - 1)\alpha_{ij} \qquad \forall (i,j) \in E_F^g \tag{6.33}$$

Zusätzlich müssen noch zwei weitere Ungleichungen aufgestellt werden:

$$\begin{array}{lcll} -z_{ij} & \geq & z_{ij} - Td_{ij} & \forall \quad (i,j) \in E_F^g \\ z_{ij} & \leq & \tau_i & \forall \quad (i,j) \in E_F^g \\ d_{ij} & \in & \{0,1\} & \forall \quad (i,j) \in E_F^g \end{array} \tag{6.34}$$

Ebenso wie alle anderen Variablen seien auch die z_{ij} auf nichtnegative Werte beschränkt. Schließlich ist die Zielfunktion zu ersetzen durch

$$\min \quad M_{steps} + \beta \sum_{ij} d_{ij} \tag{6.35}$$

Sollen Zwischenspeicherungen nach Möglichkeit vermieden werden, kann $\beta = T$ gesetzt werden; besteht jedoch Grund zu der Annahme, daß genau eine zusätzliche Speicher- und eine Ladeoperation erforderlich sind, kann auch der Wert $\beta = 2$ eingesetzt werden. Gilt $z_{ij} = 0$, dann ergibt sich (6.34) zu $-0 \geq 0 - Td_{ij}$. Wegen der Minimumbildung folgt daraus $d_{ij} = 0$, die Lebensspanne wird also nicht zwischen Instruktion i und j aufgespalten. Gilt $z_{ij} > 0$, dann folgt $-z_{ij} \geq z_{ij} - Td_{ij} \Rightarrow d_{ij} = 1$. Dies ist gleichbedeutend damit, daß Instruktionen i und j das gleiche Register verwenden, obwohl die Lebensspanne von i noch nicht beendet ist. Um dennoch die Semantik zu bewahren, müssen also Zwischenspeicherungen zur Aufspaltung der Lebensspanne erzeugt werden. Die Empfehlung wird nun folgendermaßen getroffen:

Existiert kein $d_{ij} = 1$, dann ist keine Aufspaltung von Lebensspannen erforderlich.

Bestimme i^* mit $t_{i^*} = \min\{t_i \mid d_{ij} = 1\}$

Bestimme j^* mit $t_{j^*} = \min\{t_j \mid d_{i^*j} = 1\}$

Dann sollte dasselbe Register für den Schreibzugriff von i^* und j^* verwendet werden. Dazu muß das von i^* beschriebene Register unmittelbar vor Ausführung von j^* zwischengespeichert werden und vor der ersten Verwendung der von i^* erzeugten Variablen, die hinter j^* angeordnet wurde, muß der zwischengespeicherte Wert wieder in dieses Register geladen werden. Da sich durch die Einfügung dieser Instruktionen die Lösungsstruktur stark verändern kann, kann nicht garantiert werden, daß wirklich die geschickteste Aufspaltung ermittelt wurde. So ist es zum Beispiel möglich, daß die erste Verwendung des zwischengespeicherten Wertes unmittelbar nach j^* folgt, so daß vor jedem Zugriff das entsprechende Register erst aus dem Speicher geladen werden muß. Die Empfehlung kann jedoch vom Registerallokator überprüft und gegebenenfalls zugunsten einer anderen Auswahl verworfen werden. Eine solche Einbettung der ILP-Methodik in einen umgebenden Registerallokator wurde jedoch nicht in die Implementierung aufgenommen.

Kapitel 7

Approximationen und Heuristiken

Aufgrund der hohen Rechenzeiten, die zur Lösung der generierten ILPs erforderlich sind, stellt sich die Frage, ob Heuristiken eingesetzt werden können, die keine optimale Lösung mehr garantieren, dafür aber auch für größere Eingabeprogramme eine zulässige Lösung bestimmen können. Dabei beziehen sich die Abschnitte 7.1 bis 7.4 auf Approximationslösungen bezüglich der SILP-Modellierung. Der OASIC-Ansatz erweist sich für den Einsatz von Näherungsverfahren weniger zugänglich; eine Approximation bezüglich dieses Ansatzes wird in Abschnitt 7.5 vorgestellt.

7.1 Approximation durch Rundung

Die Grundidee dieses heuristischen Ansatzes ist es, nach Möglichkeit nur relaxierte Probleme zu lösen und dabei schrittweise relaxierte Binärvariablen auf den Wert $\in \{0, 1\}$ zu fixieren, den sie vermutlich auch in der optimalen Lösung haben würden. In der SILP-Basisformulierung müssen nur die Flußvariablen, die in den Serialisierungsbedingungen auftreten, als binär spezifiziert werden; diese werden nun zu der Menge M_S zusammengefaßt. Da die Variablen aus M_S mit einem hohen Faktor multipliziert werden (vgl. Nebenbedingung 5.15 auf Seite 78), liegt nach [Zha96] folgende Vermutung nahe: ein relaxierter Wert in der Nähe von 1 (0) ist ein Indiz dafür, daß diese Variable auch in der optimalen Lösung den Wert 1 (0) annimmt.

Zur Modellierung der Globalität, der Parallelisierungsverbote wie auch der Einschränkungen der Registermengenzugehörigkeit sind zusätzliche Binärvariable hinzugekommen. Da diese nur indirekt mit den binären Flußvariablen gekoppelt sind, können sehr leicht Widersprüche entstehen, wenn auch für diese Variablen der Rundungsansatz verwendet wird. So könnten z.B. aufgrund entsprechender Variablenwerte die Nebenbedingungen $x_{ij} = 1$ und $v_{ij} = 0$ erzeugt werden, die im Laufe der Berechnung zur Unlösbarkeit führen würden. Desweiteren würde insbesondere für die Variablen zur Darstellung von Disjunktionen (y-Variablen; vgl. Nebenbedingungen 6.1 auf Seite 98) eine Aufhebung der Ganzzahligkeitsbeschränkungen die Struktur des ILP's zu stark beeinträchtigen (vgl. auch [PS82]). Daher wird in der vorliegenden Implementierung die Heuristik nur für die Variablen $x \in M_S$ verwendet (Algorithmus 7.1).

Es wird diejenige nicht-ganzzahlige Variable $x \in M_S$ mit dem geringsten Abstand zu einem ganzzahligen Wert gewählt und auf diesen Wert fixiert. Die Testergebnisse haben gezeigt, daß auch bei den relaxierten Flußvariablen diese Rundung problematisch sein kann. Durch eine falsche Fixierung kann das ILP unlösbar werden. In diesem Fall wird die zuletzt fixierte Variable auf ihr Komplement gesetzt. Ist das lineare Programm auch dann noch unlösbar, wurde bereits vorher eine falsche Entscheidung getroffen, deren Konsequenzen erst jetzt sichtbar werden. Es ist dann nicht sinnvoll, weiterhin mittels Backtracking zuvor fixierte Variablen auf den komplementären Wert festzusetzen, denn dadurch müßten im schlimmsten Fall $2^{|M_S|}$ Fixierungen betrachtet werden. Die normale Branch-and-Bound-

Ersetze die Integralitätsbedingung für alle Flußvariablen $x \in M_S$ durch $0 \leq x \leq 1$;

Bestimme die Lösung des daraus entstehenden gemischt ganzzahligen linearen Programmes;

Solange nicht alle $x \in M_S$ ganzzahlig sind

Setze $min = \min\{x \in M_S | 0 < x < 1\}$;

Setze $max = \max\{x \in M_S | 0 < x < 1\}$;

Wenn $|min| \leq |1 - max|$ **Dann**

Bestimme eine Variable x_j mit $x_j = min$;

Füge x_j in eine Liste L ein und setze $d_j = x_j$;

Füge $x_j = 0$ als neue Gleichung zum ILP hinzu;

Sonst

Bestimme eine Variable x_j mit $x_j = max$;

Füge x_j in eine Liste L ein und setze $d_j = 1 - x_j$;

Füge $x_j = 1$ als neue Gleichung zum ILP hinzu;

Ende Wenn

Löse das MILP;

Wenn keine Lösung existiert, ersetze die zuletzt hinzugefügte Gleichung durch eine Fixierung auf den komplementären Wert;

Löse das MILP;

Setze $d = 0.3$;

Solange $d \geq 0$

Ist das MILP unlösbar, durchlaufe die Liste L, mache die Fixierung aller Variable x_j mit Abstand $d_j \geq d$ von der nächstgelegenen ganzen Zahl rückgängig und spezifiziere die betreffende Variable als binär.

Löse das MILP;

War das MILP unlösbar, setze $d = d - 0.1$;

Sonst setze $d = -1$;

Ende Solange

Ende Solange

Algorithmus 7.1: Approximation durch Rundung.

Methode ist in diesem Fall vorzuziehen, da abhängig vom Zielfunktionswert der aktuell betrachteten Relaxation Teile des Baumes abgeschnitten werden können. Daher wird nun die Fixierung all der Variablen aufgehoben, deren Abstand zur nächstgelegenen ganzen Zahl größer als 0.3 war und diese als binär deklariert. Die übrigen Fixierungen werden beibehalten; somit wird für die Variablen, die um einen relativ großen Betrag gerundet wurden, die Branch and Bound-Methode verwendet. Dann wird das resultierende MILP erneut gelöst. Erweist sich auch dieses als unlösbar, wird das Verfahren stufenweise mit sinkendem Abstand wiederholt, bis das MILP gelöst werden kann – im schlimmsten Fall müssen alle Fixierungen rückgängig gemacht und das Ausgangs-ILP gelöst werden. Damit wird auch verständlich, warum diese Approximation nicht auf die anderen Typen von Binärvariablen angewendet wird: Da sehr leicht Widersprüche entstehen, müßten öfter Fixierungen rückgängig gemacht und die Variablen doch wieder als binär deklariert werden.

Da die Binärvariablen zur Berücksichtigung der Basisblockzugehörigkeit, der Parallelisierungsverbote und der Einschränkungen der Registermengenzugehörigkeit nicht relaxiert werden, kann auch die Lösung der betrachteten Relaxationen hohe Laufzeiten erfordern. Um diese Berechnungszeiten zu verkürzen, werden in einer ergänzenden Variante alle Instruktionen auf die Basisblöcke fixiert, denen sie bei der optimalen Lösung der ersten Relaxation zugeordnet werden. Dies wird durch Hinzufügen von Nebenbedingungen erreicht, die alle y-Variablen auf ihren aktuellen Wert festsetzen. Dadurch wird möglicherweise eine bessere Basisblockzuordnung als im Eingabeprogramm realisiert; durch die verringerte Anzahl zu berücksichtigender Binärvariablen sinkt in allen nachfolgenden Berechnungsphasen der erforderliche Berechnungsaufwand.

Die Auswertung der Testresultate (vgl. Kapitel 8.2) zeigt, daß bei den meisten Programmen eine zulässige Lösung in akzeptabler Zeit erreicht werden kann. Durch falsche Fixierungen können jedoch auch sehr hohe Laufzeiten entstehen; zudem ist die Qualität der Lösung schlechter, als bei den übrigen Approximationen.

7.2 Schrittweise Approximation

Während die zuvor vorgestellte Approximation durch Rundung nur auf den numerischen Werten der Lösungsvariablen basiert, wird in dieser Heuristik Bezug auf die Bedeutung der Variablen innerhalb des ILP's genommen. Wiederum werden zunächst alle als binär zu spezifizierenden Flußvariablen relaxiert. Das resultierende MILP wird gelöst und die optimale Lösung dieser Relaxation bestimmt. Dann wird das folgende Verfahren stufenweise für alle Kontrollschritte, beginnend beim ersten, wiederholt.

Es wird festgestellt, welche Instruktionen auf den aktuell betrachteten Kontrollschritt gelegt wurden, obwohl zwischen ihnen Serialisierungsbedingungen formuliert wurden. Es sei M_S^c die Menge der Variablen, die Flußkanten zwischen solchen kollidierenden Operationen bezüglich des aktuell betrachteten Kontrollschrittes c entsprechen. Alle $x \in M_S^c$ werden als binär deklariert und das resultierende MILP erneut gelöst. Hierdurch wird eine Sequentialisierung aller Mikrooperationen erzwungen, die nicht in Kontrollschritt c gleichzeitig ausgeführt werden können, die also nicht zu einer Instruktion kombiniert werden können. Anschließend wird für alle $x \in M_S^c$, die in der ermittelten Lösung den Wert $x = 1$ besitzen, dieser Wert mittels einer zusätzlichen Gleichung fixiert, die den Nebenbedingungen hinzugefügt wird. Die Integralitätsbedingung für alle $x \in M_S^c$ mit $x = 0$ wird aufgehoben und danach der nächste Kontrollschritt betrachtet.

Es ist nicht möglich, auch die $x \in M_S^c$ mit Wert 0 zu fixieren. Auf diese Weise könnte ein unzulässiges MILP entstehen, da unter Umständen kein korrekter Fluß mehr erreicht werden kann. Dieser Fall tritt dann ein, wenn alle Flußvariablen zwischen zwei Instruktionen auf 0 fixiert wurden und aufgrund nicht-ganzzahliger Werte anderer Variablen alle Nebenbedingungen erfüllt sind. Wird in einem späteren Schritt die Ganzzahligkeit dieser Variablen erzwungen, so ist es möglich, daß keine zulässige Lösung mehr existiert. Bei einer Fixierung der 1-wertigen Variablen hingegen tritt dieses Problem aufgrund der Struktur der Nebenbedingungen nicht auf.

In jeder Iterationsphase sind also nur diejenigen Flußvariablen binär deklariert, die zu kollidierenden Mikrooperationen (die nicht zu einer Instruktion kombiniert werden können) gehören. Nach Betrachtung aller Kontrollschritte kann der Fall auftreten, daß noch nicht alle Variablen ganzzahlige Werte besitzen. Dann wird iterativ die Menge der $x \in M_S$ bestimmt, die noch keinen ganzzahligen Wert besitzen, diese als binär deklariert, das ILP gelöst und die Werte der in diesem Schritt als binär deklarierten Variablen fixiert. Nachfolgend wird eine Darstellung dieses Algorithmus in Pseudocode angegeben (Algorithmus 7.2).

Durch dieses Vorgehen werden in der ersten Phase nur solche Instruktionen, durch deren Anordnung Serialisierungsbedingungen verletzt wurden, als binär deklariert. Dann wird eine optimale Lösung des resultierenden gemischt ganzzahligen linearen Programmes ermittelt und die Serialisierungen bezüglich der betrachteten Variablen fixiert. Da somit für jeden einzelnen Kontrollschritt bezüglich aufgetretener Kollisionen optimale Lösungen berechnet werden, ist es anzunehmen, daß die daraus erhaltenen Fixierungen auch zu einer guten Gesamtlösung führen; dies wird durch die Testresultate bestätigt.

Auch hier werden in einer ergänzenden Variante alle Instruktionen auf die Basisblöcke fixiert, denen sie bei der Lösung der letzten Relaxation zugeordnet wurden (vgl. Kapitel 7.1).

Ersetze die Integralitätsbedingung für alle $x \in M_S$ durch $0 \leq x \leq 1$;

Für $c = 1$ **bis** M_{steps}

> Stelle fest, ob zwischen den Operationen, deren Ausführung auf c angesetzt wurde, Serialisierungsbedingungen zu betrachten sind, und deklariere alle Variablen der resultierenden Menge M_S^c als binär;
>
> Löse das MILP;
>
> Fixiere die Werte der in diesem Schritt betrachteten Binärvariablen $x \in M_S^c$ mit Wert $x = 1$ und hebe die Integralitätsrestriktion der $x \in M_S^c$ mit Wert $x = 0$ wieder auf;

Ende Für

Solange nicht alle $x \in M_S$ ganzzahlig sind

> Deklariere alle $x \in M_S$ mit $0 < x < 1$ als binär ;
>
> Löse das MILP;
>
> Fixiere die Werte der in diesem Schritt betrachteten Variablen;

Ende Solange

Algorithmus 7.2: Schrittweise Approximation.

7.3 Isolierte Flußanalyse

Bei diesem Verfahren werden immer nur die Flußvariablen $x \in M_S$, die jeweils einer bestimmten Ressource r zugeordnet sind, als binär deklariert. Anderen Hardwareressourcen zugeordnete Flußvariablen werden relaxiert, d.h. es gilt:

$$\begin{array}{ll} 0 \leq x \leq 1 & \forall\ x\ \in M_S \text{ mit } res(x) \neq r \\ x \in \{0,1\} & \forall\ x\ \in M_S \text{ mit } res(x) = r \end{array}$$

Dann wird eine optimale Lösung dieses MILP's berechet und die $x \in M_S$ mit $res(x) = r$ auf ihren aktuellen Wert fixiert. Dieses Vorgehen wird nacheinander für alle Ressourcen wiederholt, so daß am Schluß eine zulässige Lösung des Gesamtproblems entsteht.

Für alle Ressourcentypen $r \in \{DM,\ PM,\ S,\ MU,\ C\}$

Ersetze die Integralitätsbedingung für alle $x \in M_S$ mit $res(x) \neq r$ durch $0 \leq x \leq 1$;

spezifiziere alle $x \in M_S$ mit $res(x) = r$ als binär;

Löse das MILP;

Fixiere die Werte der in diesem Schritt betrachteten Variablen;

Ende Für

Algorithmus 7.3: Approximation: isolierte Flußanalyse.

Auf diese Weise wird bezüglich jedes individuellen Ressourcenflusses die optimale Lösung bestimmt. Da sich die so berechnete Gesamtlösung am Schluß aus individuell optimalen Lösungen der einzelnen Ressourcenflüsse zusammensetzt, stimmt sie in den meisten Fällen mit der optimalen Lösung überein. Die Optimalität der ermittelten Lösung kann jedoch nicht garantiert werden, da bei der Analyse jedes einzelnen Ressourcenflusses die übrigen vernachlässigt werden. Somit kann sich eine optimale Lösung bezüglich der betrachteten Ressource im Zusammenwirken mit den anderen Ressourcenflüssen als suboptimal herausstellen.

7.4 Schrittweise Approximation der isolierten Flußanalyse

Bei der isolierten Flußanalyse wird im Grunde die normale Berechnungsmethode zur Lösung eines ganzzahligen linearen Programmes, jedoch eingeschränkt auf einen bestimmten Ressourcenfluß, eingesetzt. Die schrittweise Approximation wird zur Beschleunigung des optimalen Berechnungsverfahrens unter Inkaufnahme einer Verschlechterung der Lösungsqualität eingesetzt. Somit liegt es nahe, diese beiden Näherungsverfahren miteinander zu kombinieren und die im Rahmen der isolierten Flußanalyse betrachteten Teilprobleme durch Einsatz der schrittweisen Approximation zu lösen.

Für alle Ressourcentypen r

> Ersetze die Integralitätsbedingung für alle $x \in M_S$ mit $res(x) \neq r$ durch $0 \leq x \leq 1$;
>
> spezifiziere alle $x \in M_S$ mit $res(x) = r$ als binär;
>
> **Für** $c = 1$ **bis** M_{steps}
>
> > Stelle fest, ob zwischen den Operationen, deren Ausführung auf c angesetzt wurde, Serialisierungsbedingungen bezüglich Ressourcentyp r zu betrachten sind und deklariere alle Variablen der resultierenden Menge M_S^{rc} als binär;
> >
> > Löse das MILP;
> >
> > Fixiere die Werte der in diesem Schritt betrachteten Binärvariablen $x \in M_S^{rc}$ mit Wert $x = 1$ und hebe die Integralitätsrestriktion der $x \in M_S^{rc}$ mit Wert $x = 0$ wieder auf;
>
> **Ende Für**
>
> **Solange** nicht alle $x \in M_S$ mit $res(x) = r$ ganzzahlig sind
>
> > Deklariere alle $x \in M_S$ mit $res(x) = r$ und $0 < x < 1$ als binär ;
> >
> > Löse das MILP;
> >
> > Fixiere die Werte der in diesem Schritt betrachteten Variablen;
>
> **Ende Solange**

Ende Für

Algorithmus 7.4: Approximation: Kombination der Algorithmen aus 7.2 und 7.3.

Alle Ressourcenflüsse werden getrennt betrachtet, und bezüglich jeden Flusses wird die in Kapitel 7.2 vorgestellte Vorgehensweise angewendet, bis alle Flußvariablen der aktuell betrachteten Ressource ganzzahlig sind. Nach der isolierten Betrachtung aller Ressourcen werden dann iterativ alle nicht-ganzzahligen Variablen als binär deklariert, bis eine zulässige Lösung gefunden wird. Da stets nur Flußvariablen, die zur selben Ressource gehören und bezüglich eines Kontrollschrittes miteinander kollidieren, als binär spezifiziert sind, ist die Anzahl der gleichzeitig zu betrachtenden Binärvariablen stets vergleichsweise gering. Die Testresultate zeigen, daß diese Approximation bei größeren Eingabeprogrammen zu den niedrigsten Laufzeiten der untersuchten Verfahren führt. Dennoch konnte bei allen Testprogrammen eine sehr gute Lösung (meist eine optimale Lösung) ermittelt werden.

7.5 Approximation durch Rundung bei OASIC

Hierbei wird die Approximation durch Rundung auch im Rahmen der OASIC-Modellierung eingesetzt. Wiederum werden zunächst nur die disjunktiven Variablen zur Modellierung der Basisblockzugehörigkeit, sowie der Zuordnung von Operationen zu bestimmten Registermengen als binär deklariert. Bezüglich dieses gemischt ganzzahligen linearen Programmes wird eine optimale Lösung ermittelt.

Existieren in der Lösung Variablen, die nicht-ganzzahlige Werte besitzen, so wird die Variable mit maximalem Wert < 1 ermittelt und auf 1 fixiert. Das Verfahren wird dann iterativ so lange wiederholt, bis alle Variablen ganzzahlige Werte besitzen. Erweist sich das gemischt ganzzahlige lineare Programm als unlösbar, so wird die zuletzt fixierte Variable auf den Wert 0 gesetzt. Aufgrund der Struktur der OASIC-Modellierung führt eine falsche Entscheidung (eine falsche Variablenfixierung auf 1) meist schon beim nächsten Iterationsschritt zur Unlösbarkeit. Ist das resultierende MILP auch dann noch unlösbar, so wurde bereits vorher eine falsche Entscheidung getroffen, deren Auswirkungen sich erst jetzt zeigen. Dann werden - analog zum in Kapitel 7.1 beschriebenen Vorgehen - Fixierungen von Variablen, die um einen großen Wert gerundet wurden, aufgehoben und diese Variablen wieder als binär deklariert. Erweist sich auch dieses MILP als unlösbar, wird das Verfahren stufenweise mit sinkendem Abstand wiederholt, bis das MILP gelöst werden kann – im schlimmsten Fall müssen alle Fixierungen rückgängig gemacht und das Ausgangs-ILP gelöst werden.

Kapitel 8

Implementierung und Testergebnisse

8.1 Implementierung

Als Quellsprache wird der Instruktionssatz des ADSP-2106x verwendet. Ein Quellprogramm kann durch Übersetzung eines C-Programmes durch den C-Compiler `g21k`, der zum Lieferumfang des ADSP-2106x gehört und auf dem `gcc` basiert, erzeugt werden oder von Hand geschrieben werden.

Innerhalb der Implementierung können verschiedene Phasen unterschieden werden:

1. Das Quellprogramm wird durch Einsatz eines Scanners (`flex`) und eines Parsers (`bison`) syntaktisch und semantisch analysiert. Grundlage des Parsers bildet dabei eine für den Instruktionssatz des ADSP-2106x enwickelte kontextfreie Grammatik.

2. Auf der Basis der hierdurch gewonnenen Informationen werden die benötigten Programmdarstellungen generiert. Hierzu zählen neben Kontrollfluß-, und Kontrollabhängigkeitsgraphen sowie dem verallgemeinerten Datenabhängigkeitsgraphen auch weitere Graphen, die in Abhängigkeit von der verwendeten ILP-Modellierung erstellt werden. Bei Verwendung der SILP-Formulierung wird für jede Hardwareressource der benötigte Ressourcenflußgraph und bei Einschluß der Registerallokation ebenfalls der Registerflußgraph aufgebaut. Wird die OASIC-Modellierung in Verbindung mit Registerallokationsbedingungen eingesetzt, so wird der Lebenszeitgraph, der alle möglichen lebenszeitdefinierenden Kanten enthält (vgl. Kapitel 5.2.2), erstellt. Alle erzeugten Graphen können mit Hilfe des `vcg`-Tools visualisiert werden.

3. In der dritten Phase werden die ganzzahligen linearen Programme zur Lösung des Instruktionsanordnungs- und Registerallokationsproblems als Textdateien erzeugt. Durch Parameter können hierbei folgende Entscheidungen bezüglich der Lösungsstrategie getroffen werden:

   - Auswahl des Modellierungsansatzes (SILP oder OASIC).
   - Isolierte Lösung des Instruktionsanordnungsproblems oder Einbezug der Registerallokation.
   - Schließlich kann angegeben werden, ob eine optimale Lösung berechnet, eine der in Kapitel 7 vorgestellten Approximationen verwendet, oder eine untere Schranke (vgl. Kapitel 8.2) berechnet werden soll.

4. Die generierten linearen Programme werden durch Einsatz von CPLEX, einer Programmbibliothek zur Lösung von linearen Programmen, gelöst. Diese Phase wird in Kapitel 8.1.2 noch genauer beschrieben.

5. Das numerische Resultat des Optimierungsprozesses wird interpretiert und die optimierte Befehlssequenz in die Resultatdatei geschrieben.

6. Falls eine Registerallokation durchgeführt wurde, muß nun gegebenenfalls ein graphbasierter Registerallokator aufgerufen werden, der unter Beachtung der in Phase 4 bestimmten Registermengenzuteilung eine konkrete Registerverteilung ermittelt. Die Resultatdatei wird an diese Veränderungen angepaßt.

Eine Veranschaulichung dieses Arbeitsprozesses ist in Abbildung 8.1 dargestellt.

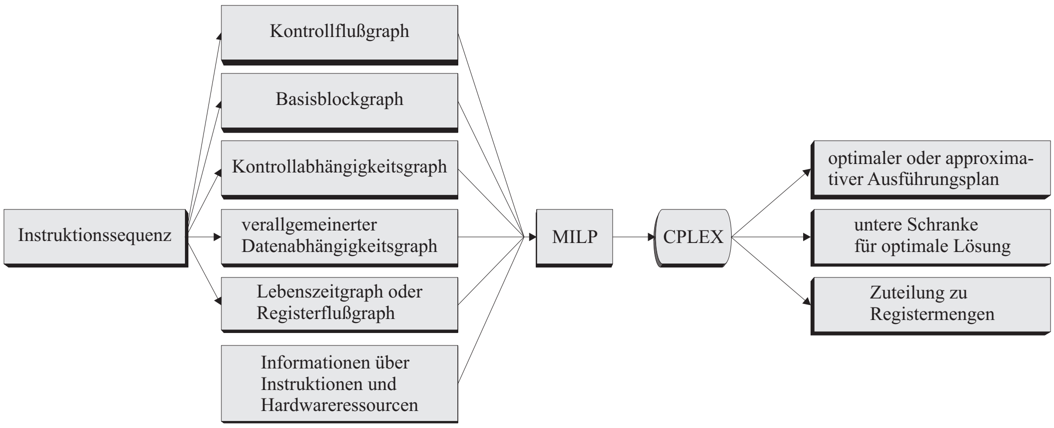


Abbildung 8.1: Ein-/Ausgabesystem des Optimierungsprozesses.

8.1.1 CIS

Die Implementierung liegt als ausführbare Datei mit Namen `CIS`(*Comparative Instruction Scheduler*) vor. Sie wurde in Zusammenarbeit mit Marc Langenbach erstellt ([Lan97]) und realisiert sowohl Verfahren der linearen Programmierung als auch klassische Algorithmen zur Lösung der Instruktionsanordnungs- und Registerallokationsproblematik. Die klassischen Verfahren werden detailliert im Rahmen von [Lan97] analysiert.

Das gewünschte Berechnungsverfahren wird mittels Kommandozeilenoptionen ausgewählt. Eine genaue Beschreibung der Optionen ist im Anhang D zu finden.

8.1.2 Schnittstelle zu CPLEX

CPLEX stellt Bibliotheksfunktionen zum Lösen, Verändern und zur Ergebnisauswertung gemischt ganzzahliger linearer Programme zur Verfügung. Die erzeugten (ganzzahligen) linearen Programme werden im sogenannten LP-Format erstellt, das im wesentlichen der algebraischen Darstellung des Optimierungsproblems entspricht. Als alternatives Format steht das MPS-Format zur Verfügung, bei dem das Problem in Matrizenschreibweise dargestellt wird. Dieses führt zwar zu einer geringeren Dateigröße, die LP-Darstellung ist jedoch leichter zu generieren und vor allem besser lesbar, was im Hinblick auf die Überprüfung der Korrektheit der erzeugten Programme von entscheidender Bedeutung ist. Eingelesene Probleme können im Berechnungsverlauf auf verschiedene Arten modifiziert werden. Es können Nebenbedinungen gelöscht oder hinzugefügt werden; Variablen können als binär, ganzzahlig oder reellwertig deklariert werden oder mit Hilfe von Schranken auf einem bestimmten Wertebereich definiert werden.

Zur Lösung von linearen Programmen wird der Simplexalgorithmus verwendet. Enthält das Problem ganzzahlige Variablen, wird eine Branch-and-Bound-Methode eingesetzt. Der Pfad, der im Branch-and-Bound-Baum durchlaufen wird, kann vom Benutzer beeinflußt werden. Sowohl Tiefensuche mit Backtracking als auch Breitensuche und verschiedene Adaptivregeln können eingesetzt werden. Im Rahmen dieser Implementierung erbrachte die *Best-Bound* Methode die besten Resultate.

Zur Bestimmung der Variable, nach der separiert wird, werden den erzeugten Binärvariablen drei verschiedenen Prioritäten zugeordnet:

- Die höchste Priorität erhalten die Binärvariblen y_i,
- Die binär zu spezifizierenden Flußvariablen x_{ij}^{kg} bei SILP, ebenso wie die Entscheidungsvariablen x_{in}^{km}, die bei OASIC die Ausführung einer Instruktion einem Kontrollschritt und eventuell einer Zielregistermenge zuordnen, erhalten die zweithöchste Priorität.
- Zuletzt soll schließlich nach den disjunktiven Binärvariablen zur Modellierung der Parallelitätsverbote, sowie den Einschränkungen der Registermengenzugehörigkeit separiert werden.

Stehen mehrere Variablen derselben Priorität zur Wahl, so wird die Entscheidung mit Hilfe des sogenannten *strong branching* getroffen. Hierbei wird versucht, bei jedem Knoten die aus verschiedenen Verzweigungen entstehenden Teilprobleme ansatzweise zu lösen und auf diese Weise die meistversprechende Verzweigungsvariable zu bestimmen. Dies hat zur Folge, daß sich die Rechenzeit pro Knoten erhöht; dafür müssen aber weniger Knoten generiert werden. Insgesamt sinkt dadurch die Rechenzeit; der Speicherbedarf ist erheblich niedriger.

8.2 Testprogramme und Ergebnisse

Bei der Auswahl der Testprogramme wurde der Schwerpunkt auf typische Anwendungen digitaler Signalprozessoren gelegt. Hierzu zählen die Berechnung von Fouriertransformationen, die Berechnung digitaler Filter, ebenso wie Algorithmen im Bereich der Bildverarbeitung. Diese Programme wurden (mit Ausnahme von `whetp3.s`) von Analog Devices entwickelt. Alle Berechnungen wurden auf einem Compute Server Ultra 2/2x200 mit 1024 MB Hauptspeicher unter dem Betriebssystem Solaris 2.5.1 ausgeführt.

Die in diesem Kapitel untersuchten Eingabeprogramme basieren meist auf manuell erstellten Codesequenzen. Bei diesen wird durch geschickte Verteilung der Daten auf Daten- und Programmspeicher und durch geschickte Registerzuteilung ein hoher Parallelisierbarkeitsgrad erreicht. Alle Instruktionsfolgen werden unter Wahrung der Semantik sequentialisiert, d.h. jede Mikroinstruktion enthält genau eine Mikrooperation. Die Parallelverarbeitungskapazitäten des Prozessors werden also in den zu optimierenden Eingabeprogrammen nicht genutzt. Das Ergebnis der ILP-Optimierungen wird dann mit der manuell erstellten Optimierung und den Resultaten der im Rahmen von [Lan97] implementierten konventionellen, graphbasierten Algorithmen verglichen. Wird die Instruktionsanordnung ohne Registerallokation durchgeführt, garantiert die vorliegende Registerverteilung einen hohen Parallelisierbarkeitsgrad. Hierbei ist zu beachten, daß es sich um eine idealisierte Betrachtung handelt: Die Instruktionsanordnung wird für den Fall durchgeführt, daß zuvor eine Registerzuteilung berechnet wurde, die einen hohen (den maximalen) Parallelisierbarkeitsgrad ermöglicht. Daher stimmt die ermittelte Instruktionszahl der Optimierung mit Berücksichtigung der Registerallokation mit dem Resultat der reinen Instruktionsanordnung überein. Würde hingegen die Registerzuteilung des Eingabeprogrammes keine Parallelisierungen von ALU- und Multipliziereroperationen erlauben (vgl. Kapitel 2.3.4), würde hiervon nur das Ergebnis der reinen Instruktionsanordnung beeinträchtigt. Bei der Betrachtung der Instruktionsanordnung mit Berücksichtigung von Aspekten der Registerallokation ist die Registerzuteilung des Eingabeprogrammes irrelevant, da eine eigenständige Lösung berechnet wird.

Bei den folgenden Unterkapiteln wurde ein einheitlicher Aufbau gewählt. Zunächst wird das jeweilige Eingabeprogramm vorgestellt. Anschließend werden die Charakteristika der erstellten linearen Programme in Tabellenform aufgelistet. Von Bedeutung ist hier neben der Anzahl der Nebenbedingungen und Variablen vor allem auch die Gesamtzahl der als binär zu spezifizierenden Variablen. Diese wird weiter nach den verschiedenen Typen von Binärvariablen aufgeschlüsselt:

- Als x-Variablen werden die Binärvariablen der Basisformulierungen (vgl. Kapitel 5.1 und 5.2) bezeichnet, also die als binär zu spezifizierenden Flußvariablen bei SILP und die zentralen Entscheidungsvariablen bei OASIC.
- Die y-Variablen dienen der Modellierung der Basisblockzugehörigkeit (vgl. Kapitel 6.1).
- Die p/q-Variablen werden für Einschränkungen der Registermengenzugehörigkeit verwendet (vgl. Kapitel 6.2.3 und 6.3.2).
- v-Variablen sind in der SILP-Formulierung zur Modellierung der Parallelisierungsverbote erforderlich (vgl. Kapitel 6.2.2).

Aufgrund des iterativen Berechnungsverfahrens bei OASIC wächst die Zahl der als binär spezifizierten Variablen bei jedem Iterationsschritt an. Es müssen jedoch nicht alle Variablen als binär spezifiziert werden; daher wird die Gesamtvariablenzahl als obere Schranke für die Anzahl der Binärvariablen angegeben. Schließlich ist auch der Platzbedarf zum Anlegen der ganzzahligen linearen Programme von Interesse. Um einen Vergleich der beiden Modellierungsansätze zu erleichtern, wird in einer weiteren Tabellenspalte das Größenverhältnis korrespondierender Werte von SILP bzw. OASIC angegeben. Wegen des iterativen Berechnungsvorgehens zur Lösung der beim OASIC-Ansatz erzeugten ILPs wird bezüglich der Anzahl der Binärvariablen kein expliziter Vergleich gezogen; die entsprechenden Tabelleneinträge bleiben offen.

Anschließend werden für die verschiedenen Lösungsmethoden die insgesamt benötigte CPU-Zeit und die erreichte Instruktionszahl angegeben und dieses Ergebnis diskutiert. Dabei werden für die verwendeten Lösungsmethoden folgende Abkürzungen verwendet:

Berechnung einer optimalen Lösung	-	Default
Approximation durch Rundung	-	App1
schrittweise Approximation	-	App2
isolierte Flußanalyse	-	App3
schrittweise Approximation der isolierten Flußanalyse	-	App32

Wie in Kapitel 7 dargelegt wurde, kann die Berechnungszeit bei der Approximation durch Rundung und bei der schrittweisen Approximation weiter beschleunigt werden; hierfür muß jedoch eine Verschlechterung der Lösungsqualität in Kauf genommen werden. Nach der Lösung der ersten Relaxation werden alle y-Variablen mittels zusätzlicher Nebenbedingungen auf ihren aktuellen Wert fixiert. Auf diese Weise werden die Instruktionen den Basisblöcken zugewiesen, denen sie in der Lösung der ersten Relaxation zugeordnet worden sind. Das normale Verfahren wird dabei durch ein nachgestelltes “a”, die Fixierung der y-Variablen durch ein nachgestelltes “b” beschrieben (z.B. App1a, App1b).

Neben den optimalen bzw. approximativen Gesamtlösungen werden auch verschiedene untere Schranken für den optimalen Zielfunktionswert (die optimale Instruktionsanzahl) berechnet. Beim Verfahren B1 wird der Zielfunktionswert der Relaxation verwendet, die entsteht, wenn die Integralitätsbedingungen der Flußvariablen bei SILP aufgehoben werden. Bei OASIC wird der Zielfunktionswert nach der ersten Iteration angegeben. Beim Verfahren B2 werden zusätzlich auch die Integralitätsbedingungen der v-Variablen und p/q-Variablen aufgehoben.

8.2.1 FIR-Filter

FIR-Filter (*Finite Impulse Response Filter*) zählen zu den meistimplementierten Anwendungen der digitalen Signalverarbeitung ([GO95]). Sie berechnen im wesentlichen ein gewichtetes Mittel einer Folge von Eingangsdaten. Dies wird durch die folgende Gleichung beschrieben:

$$y[n] = \sum_{i=0}^{M} h_i x[n-i].$$

Dabei heißt M die Ordnung des Filters, die $h_i \in \mathbb{R}$ $(0 \leq i \leq M)$ werden als Filterkoeffizienten bezeichnet. $y[n]$ ist das Ausgangssignal, $x[n]$ das Eingangssignal zum Zeitpunkt n.

Im Beispielprogramm `fir.s` wird ein solcher FIR-Filter berechnet. Da die Implementierung auf Prozedurebene arbeitet, wurde die Initialisierungsroutine des Originalprogrammes vernachlässigt und nur die eigentliche Berechnungsprozedur als Eingabeprogramm verwendet. Das Eingabeprogramm besteht aus 18 Instruktionen: einer Schleifenanweisung (`do until lce`), 5 Zugriffen auf das DM, 4 Zugriffen auf das PM, 4 Multiplikationen, 3 Additionen und einem Prozedurrücksprung (`rts`). Wie die Testresultate zeigen, besteht der optimale Code aus 8 Instruktionen. Im Anhang C.1 wird das Eingabeprogramm einer optimal angeordneten Instruktionssequenz (ermittelt mit SILP/Default) gegenübergestellt.

Die wesentlichen Eigenschaften der generierten ganzzahligen linearen Programme für das Instruktionsanordnungsproblem unter Berücksichtigung der Registerallokation sind in Tabelle 8.1 aufgeführt. Auffällig ist die Größe des bei der OASIC-Formulierung erzeugten ILP's, die den Platzbedarf der SILP-Modellierung um ein Vielfaches übersteigt. Dies resultiert zum einen aus der höheren Variablen- und Nebenbedingungszahl, zum anderen daraus, daß Startzeitpunkte durch lange Summationen dargestellt werden müssen (vgl. Kapitel 5.2, Gleichung 5.2.1).

Die Berechnungen führen zu den in Tabelle 8.2 angegebenen Resultaten. Diese Tabelle enthält eine Spalte Q, in der sämtliche Ausführungszeiten bezüglich der optimalen Lösung (Default) der SILP-Codierung normiert werden. Dabei wird für ein Verfahren M folgende Gleichung verwendet:

$$Q_M = \frac{\text{Ausführungszeit}_M}{\text{Ausführungszeit}_{SILP/Default}}$$

Alle verwendeten Berechnungsmethoden außer der Approximation durch Rundung bei OASIC erreichen die optimale Lösung; der OASIC-Ansatz benötigt dabei fast die 150-fache Laufzeit der SILP-Modellierung. Auffällig ist, daß die Approximationen der SILP-Formulierung eine längere Laufzeit erfordern, als zur exakten Lösungsbestimmung nötig ist. Dies liegt daran, daß zur Berechnung der Approximationen eine größere Anzahl kleinerer ganzzahliger linearer Programme gelöst werden müssen. Der Zeitgewinn durch die

	SILP	OASIC	SILP/OASIC
Nebenbedingungen	422	1233	1 : 2,92
Variablen	675	799	1 : 1,18
Binärvariablen	464	$\leq$ 799	–
x-Variablen	454	$\leq$ 791	–
y-Variablen	4	4	–
v-Variablen	2	–	–
p/q-Variablen	4	4	–
Größe des LP [Byte]	60357	2835883	1 : 46,98

Tabelle 8.1: Charakteristika der für `fir.s` generierten ILPs mit Berücksichtigung der Registerallokation.

Modellierung	Methode	Instruktionen	CPU-Zeit	Q
OASIC	Default	8	4 min 11 sec	148,52
	App1a	11	80,35 sec	47,54
SILP	Default	8	1,69 sec	1
	App1a	8	11,09 sec	6,56
	App1b	8	8,25 sec	4,88
	App2a	8	7,8 sec	4,61
	App2b	8	9,64 sec	5,7
	App3	8	16,73 sec	9,9
	App32	8	19,58 sec	11,59

Tabelle 8.2: Laufzeitergebnisse von `fir.s` mit Berücksichtigung der Registerallokation. In der Spalte Q wird das Verhältnis der einzelnen Laufzeiten zur Berechnungsdauer der optimalen SILP-Berechnung (Default) angegeben.

Modellierung	Methode	Instruktionen	CPU-Zeit	Abweichung[%]
OASIC	B1	8	18,6	0
	B2	8	9,87	0
SILP	B1	8	1,2 sec	0
	B2	8	0,42 sec	0

Tabelle 8.3: Berechnung unterer Schranken bei `fir.s` mit Berücksichtigung der Registerallokation. Die Spalte “Abweichung” zeigt den prozentualen Abstand von der optimalen Instruktionsanzahl.

verringerte Größe der Teilprobleme ist bei diesem Programm nicht groß genug, um die erhöhte Anzahl der zu berechnenden Probleme auszugleichen.

Die ermittelten unteren Schranken für die optimale Instruktionsanzahl stimmen alle mit dem optimalen Zielfunktionswert überein (vgl. Tabelle 8.3). Die beim OASIC-Ansatz erforderliche Laufzeit zur Schrankenberechnung ist jedoch wesentlich höher, als bei Verwendung der SILP-Modellierung.

8.2.2 IIR-Filter

Dieses Beispiel wurde ausgewählt, um auf Basisblockebene eine vollständige, von externen Registerallokatoren unabhängige Registerzuteilung durchzuführen. Die im Eingabeprogramm `cascade.s` enthaltene Funktion wird bei der Berechnung eines IIR-Filters (*Infinite Impulse Response Filter*) verwendet. Für tiefergehende Informationen zu IIR-Filtern sei auf [Lüc80] verwiesen.

Kernstück des Assemblerprogrammes ist eine Schleife; aus dieser wurde die Schleifenanweisung entfernt, so daß genau eine Iteration der Schleife durchgeführt wird. Durch die Schleifenanweisung würde ein nichtlinearer Programmfluß entstehen; in diesem Fall würde nur eine optimale Registermengenverteilung, jedoch keine vollständige Registerzuteilung berechnet (vgl. Kapitel 6.4). Die resultierende Codesequenz besteht aus einem Basisblock, der 20 Mikrooperationen enthält. Diese setzen sich zusammen aus zwei unmittelbaren Datenübertragungen, 5 Zugriffen auf den Datenspeicher, 5 Datenzugriffen auf den Programmspeicher, 4 Additionen und 4 Multiplikationen.

Optimierung ohne Berücksichtigung der Registerallokation

Tabelle 8.4 beschreibt die erstellten linearen Programme; die Laufzeitergebnisse sind in Tabelle 8.5 aufgeführt. Die SILP-Codierung ist im Vergleich zur OASIC-Darstellung effizienter. Ähnlich wie bei `fir.s` wirkt sich der Einsatz von Approximationen noch negativ auf die Laufzeit aus; ein optimales Ergebnis wird jedoch überall erzielt. Das Eingabeprogramm und eine optimierte Instruktionsfolge (ermittelt mit SILP/Default) werden in Anhang C.2 aufgeführt.

Optimierung mit Berücksichtigung der Registerallokation

Die erstellten ganzzahligen linearen Programme sind in Tabelle 8.6 beschrieben; Tabelle 8.7 zeigt die Laufzeitergebnisse. Im Vergleich zum ILP für die reine Instruktionsanordnung steigt die Anzahl der Nebenbedingungen bei OASIC auf das 2,07-fache, bei SILP auf das

	SILP	OASIC	SILP/OASIC
Nebenbedingungen	161	916	1 : 5,69
Variablen	77	325	1 : 4,22
Binärvariablen	11	$\leq$ 325	–
x-Variablen	3	$\leq$ 325	–
y-Variablen	–	–	–
v-Variablen	8	–	–
p/q-Variablen	–	–	–
Größe des LP [Byte]	4250	206135	1 : 48,5

Tabelle 8.4: Charakteristika der für `cascade.s` generierten ILPs ohne Berücksichtigung der Registerallokation.

Modellierung	Methode	Instruktionen	CPU-Zeit	Q
OASIC	Default	7	5,45 sec	23,68
	App1a	7	1,04 sec	5,47
SILP	Default	7	0,19 sec	1
	App1a	7	0,22 sec	1,16
	App2a	7	0,26 sec	1,37
	App3	7	0,32 sec	1,68
	App32	7	0,27 sec	1,42

Tabelle 8.5: Laufzeitergebnisse von `cascade.s` ohne Berücksichtigung der Registerallokation.

	SILP	OASIC	SILP/OASIC
Nebenbedingungen	606	1904	1 : 3,14
Variablen	950	1200	1 : 1,26
Binärvariablen	707	$\leq$ 1208	–
x-Variablen	691	$\leq$ 1200	–
y-Variablen	–	–	–
v-Variablen	8	–	–
p/q-Variablen	8	8	–
Größe des LP [Byte]	91515	7154872	1 : 78,18

Tabelle 8.6: Charakteristika der für `cascade.s` generierten ILPs mit Berücksichtigung der Registerallokation.

Modellierung	Methode	Instruktionen	CPU-Zeit
OASIC	Default	7	11 min 7 sec
	App1a	7	65,97 sec
SILP	Default	–	> 24 h
	App1a	7	99,9 sec
	App2a	7	73,69 sec
	App3	7	3 min 46 sec
	App32	7	86,72 sec

Tabelle 8.7: Laufzeitergebnisse von `cascade.s` mit Berücksichtigung der Registerallokation.

3,76-fache an. Die Variablenzahl bei OASIC steigt auf das 3,69-fache, bei SILP sogar auf das 12,33-fache. Während auf der Basis der OASIC-Formulierung in 11 Minuten und 7 Sekunden eine optimale Lösung ermittelt werden kann, wurde die exakte Lösungsberechnung bei SILP (Default) nach 24 Stunden abgebrochen. Zu beachten ist hierbei jedoch, daß bei der SILP-Formulierung eine vollständige Registerzuteilung, bei OASIC hingegen lediglich eine Registermengenzuteilung ermittelt wird. Alle Approximationen liefern eine optimale Lösung; die schrittweise Approximation der isolierten Flußanalyse (App32) benötigte hierfür nur eine Laufzeit von 86,72 Sekunden.

Wie aus Tabelle 8.8 zu entnehmen ist, wirkt sich bei der Berechnung unterer Schranken auf der Grundlage der SILP-Modellierung die Aufhebung der Integralitätsbedingungen der p/q-Variablen auf die Qualität der Schranke aus. Die Abweichung vom optimalen Zielfunktionswert beträgt beim Verfahren B1 14 %, beim Verfahren B2 28 %. Beide Schranken können jedoch sehr schnell ermittelt werden. Die CPU-Zeit der Berechnung unterer Schranken auf der Grundlage der OASIC-Formulierung ist wiederum höher; allerdings liefert Verfahren B2 eine bessere Schranke als bei SILP.

Modellierung	Methode	Instruktionen	CPU-Zeit	Abweichung [%]
OASIC	B1	6	22,87	14
	B2	6	20,91	14
SILP	B1	6	6,76 sec	14
	B2	5	0,39 sec	28

Tabelle 8.8: Berechnung unterer Schranken bei `cascade.s` mit Berücksichtigung der Registerallokation. Die Spalte "Abweichung" zeigt den prozentualen Abstand von der optimalen Instruktionsanzahl.

8.2.3 DFT

Dieses Beispielprogramm berechnet die komplexe diskrete Fouriertransformation (DFT) der Daten einer Eingabedatei. Diese ist wie folgt definiert:

$$c_k = \sum_{j=0}^{n-1} y_j w_n^{jk} \qquad k \in \{0, 1, \ldots, n-1\}$$

Dabei stellen die c_k die Ausgabewerte für $0 \leq k \leq n-1$ dar, die y_j sind Eingabewerte zum Zeitpunkt j. Die komplexe Größe w_n stellt eine n-te Einheitswurzel dar: $w_n = e^{-i\frac{2\Pi}{n}}$.

Wiederum wird zur Analyse die Berechnungsprozedur herangezogen, während eine enthaltene Initialisierungsroutine vernachlässigt wird. Die resultierende Codesequenz besteht aus 26 Instruktionen und enthält zwei geschachtelte Schleifen. Die auftretenden Instruktionen können aufgeteilt werden in 3 unmittelbare Datenübertragungen (*immediate data move*), zwei zählerbasierte Schleifenanweisungen, 6 ALU-Instruktionen, 3 Multiplikationen, eine `rts`-Anweisung und 10 Datenübertragungsanweisungen, darunter 4 Zugriffe auf den Datenspeicher und 4 Zugriffe auf den Programmspeicher. Das Eingabeprogramm wird in Anhang C.3 einer optimierten Instruktionsfolge gegenübergestellt.

Optimierung ohne Berücksichtigung der Registerallokation

Aufgrund hoher Laufzeiten bei der Analyse unter Berücksichtigung der Registerallokation wird zunächst das reine Instruktionsanordnungsproblem betrachtet. Die Registerzuteilung des Eingabeprogrammes wird also in der Berechnung übernommen. Tabelle 8.9 beschreibt die für das Eingabeprogramm `dft.s` erzeugten ganzzahligen linearen Programme.

Die Laufzeitergebnisse werden in Tabelle 8.10 präsentiert. Die Laufzeit der OASIC-Modellierung beträgt das 265-fache der zur optimalen Lösung des ILP's der SILP-Modellierung erforderlichen Laufzeit. Die Berechnungszeit der Approximation durch Rundung (App1a) übersteigt deutlich die Berechnungsdauer aller anderen Ansätze zur Lösung der

	SILP	OASIC	SILP/OASIC
Nebenbedingungen	405	1393	1 : 3,43
Variablen	184	567	1 : 3,08
Binärvariablen	102	≤ 567	–
x-Variablen	73	≤ 551	–
y-Variablen	16	16	–
v-Variablen	13	–	–
p/q-Variablen	–	–	–
Größe des LP [Byte]	15157	244323	1 : 16,12

Tabelle 8.9: Charakteristika der für `dft.s` generierten ILPs ohne Berücksichtigung der Registerallokation.

Modellierung	Methode	Instruktionen	CPU-Zeit	Q
OASIC	Default	14	3 h 30 min	265,7
	App1a	14	1 h 39 min	125,26
SILP	Default	14	47,42 sec	1
	App1a	16	5 min 44 sec	7,25
	App1b	19	43,39 sec	0,91
	App2a	14	2 min 11 sec	2,76
	App2b	16	17,18 sec	0,36
	App3	14	31,15 sec	0,66
	App32	14	42,58 sec	0,89

Tabelle 8.10: Laufzeitergebnisse von `dft.s` ohne Berücksichtigung der Registerallokation.

SILP-Formulierung. Dies liegt daran, daß aufgrund von Unlösbarkeiten mehrmals Rundungsfixierungen rückgängig gemacht werden müssen und die betreffenden Variablen als binär deklariert werden müssen (vgl. Kapitel 7.1). Die CPU-Zeit sinkt durch die Fixierung der y-Variablen nach Berechnung der ersten Relaxation (App1b) um 87 %; dabei verschlechtert sich jedoch auch die Lösungsqualität um 18%. Sowohl die schrittweise Approximation (App2a) als auch isolierte Flußanalyse (App3) und schrittweise Approximation der isolierten Flußanalyse (App32) erreichen eine optimale Lösung. Dabei ist die Laufzeit von App3 am geringsten; sie beträgt 66% der zur exakten Lösung erforderlichen Laufzeit (beim SILP-Ansatz).

Optimierung mit Berücksichtigung der Registerallokation

Wird das Eingabeprogramm unter Einbezug der Registerallokationsbedingungen optimiert, so sind die in Tabelle 8.11 beschriebenen linearen Programme zu lösen. Im Vergleich zu den ganzzahligen linearen Programmen ohne Berücksichtigung der Registerallokation steigt die Anzahl der Nebenbedingungen bei SILP auf das 5,29-fache, bei OASIC auf das 4,55-fache. Die Anzahl der Variablen versechsfacht sich bei SILP, bei OASIC wächst sie auf das 2,83-fache. Die bemerkenswerteste Information ist die Größe des bei der OASIC-Formulierung erzeugten ILP's: Mit fast 47 MB beansprucht sie den 150-fachen Platz der SILP-Codierung.

Wie Tabelle 8.12 zu entnehmen ist, ist es möglich, durch Anwendung der Approximation durch Rundung bei der OASIC-Formulierung zu einem Ergebnis zu gelangen, während die Berechnung der SILP-Approximation durch Rundung (App1a) nach 25 Stunden abgebrochen wurde. Dies kann als Indiz dafür interpretiert werden, daß in der OASIC-Formulierung das Polytop der Relaxation die konvexe Hülle des ganzzahligen Zulässigkeitsbereiches besser approximiert (vgl. Kapitel 4.2.2). Die Laufzeitergebnisse der folgenden Testprogramme stützen diese Vermutung. Diesem Vorzug steht jedoch die Dateigröße des erzeugten ILP's gegenüber. Zudem konnte durch Anwendung der Approximation App32 (schrittweise Approximation der isolierten Flußanalyse) der SILP-Formulierung ein optimales Ergebnis in weniger als 10 Minuten Rechenzeit ermittelt werden. Durch Anwendung der Approximation durch Rundung auf das OASIC-Modell konnte in 17,5 Stunden nur ein um 14 % schlechteres Ergebnis, d.h. ein um 14 % längerer Ausführungsplan erzielt werden.

Untere Schranken für den optimalen Zielfunktionswert können beim SILP-Ansatz innerhalb von Sekunden berechnet werden (vgl. Tabelle 8.13). Beide Verfahren, B1 und B2, geben als untere Schranke für die Instruktionsanzahl den Wert 12 an; die Abweichung von der optimalen Lösung beträgt also 14 %. Werden die unteren Schranken auf der Grundlage der OASIC-Modellierung berechnet, so erhalten wir bei beiden Verfahren (B1 und B2) den Wert 13; die Abweichung von der optimalen Zielfunktionswert beträgt also nur 7 %. Die erforderliche Rechenzeit ist jedoch erheblich größer als bei Verwendung der SILP-Formulierung.

	SILP	OASIC	SILP/OASIC
Nebenbedingungen	2143	6342	1 : 2,96
Variablen	1201	1605	1 : 1,34
Binärvariablen	946	$\leq$ 1605	–
x-Variablen	909	$\leq$ 1581	–
y-Variablen	16	16	–
v-Variablen	13	–	–
p/q-Variablen	8	8	–
Größe des LP [Byte]	301662	46341471	1 : 153,62

Tabelle 8.11: Charakteristika der für `dft.s` generierten ILPs mit Berücksichtigung der Registerallokation.

Modellierung	Methode	Instruktionen	CPU-Zeit
OASIC	Default	–	> 24 h
	App1a	16	17 h 25 min
SILP	Default	–	> 24 h
	App1a	–	> 24 h
	App1b	20	1h 6 min
	App2a	14	11 h 35 min
	App2b	15	6 min 8 sec
	App3	14	1h 43 min
	App32	14	9 min 20 sec

Tabelle 8.12: Laufzeitergebnisse von `dft.s` mit Berücksichtigung der Registerallokation.

Modellierung	Methode	Instruktionen	CPU-Zeit	Abweichung [%]
OASIC	B1	13	50 min 31 sec	7
	B2	13	47 min 57 sec	7
SILP	B1	12	37,142 sec	14
	B2	12	2,9 sec	14

Tabelle 8.13: Berechnung unterer Schranken bei `dft.s` mit Berücksichtigung der Registerallokation. Die Spalte “Abweichung” zeigt den prozentualen Abstand von der optimalen Instruktionsanzahl.

8.2.4 Whetstone-Funktion

Während die übrigen Testprogramme Assemblercodesequenzen aus typischen Einsatzgebieten digitaler Signalprozessoren darstellen, wird in diesem Beispiel der Code für eine im Rahmen des bekannten Whetstone-Benchmarks verwendete Funktion optimiert. Als Eingabeprogramm wird dabei die vom `g21k` für diese Funktion erzeugte Codesequenz verwendet. Sie berechnet die folgende C-Funktion:

```
p3(double x, double y, double *z)
{  /* t, t2 sind globale Variablen */
   x  = t * (x + y);
   y  = t * (x + y);
   *z = (x + y) /t2;
}
```

Aus der Ergebnisinstruktionsfolge wurden einige Anweisungen zur Prozedurorganisation (Sichern und Wiederherstellen von Registerinhalten) entfernt, da sie keine Auswirkungen auf das eigentliche Programm haben und sequentiell ausgeführt werden müssen. Werden zusätzlich die vorgenommenen Parallelisierungen rückgängig gemacht, erhalten wir ein aus 26 Instruktionen bestehendes Eingabeprogramm. Es wird aus Laufzeitgründen nur das reine Instruktionsanordnungsproblem untersucht. Der Vergleich mit den konventionellen, graphbasierten Algorithmen (vgl. Kapitel 8.3) ist dennoch zulässig, da auch diese mit Verwendung der vorliegenden Registerverteilung ausgeführt wurden. In Anhang C.4 wird das Eingabeprogramm einer optimal angeordneten Codesequenz (Verwendung von SILP/Default) gegenübergestellt.

Die erstellten linearen Programme werden in Tabelle 8.14 beschrieben. Wie Tabelle 8.15 zu entnehmen ist, kann eine optimale Lösung durch Verwendung der OASIC-Modellierung in nur 15 % der bei SILP erforderlichen Laufzeit ermittelt werden. Auch die Approximation durch Rundung (App1a) erfordert bei OASIC eine niedrigere Laufzeit, als alle verwendeten Approximationen der SILP-Formulierung. Der Grund für diese Abweichung ist in der Anzahl der Parallelisierungsverbote zu sehen. Das Eingabeprogramm enthält viele Instruktionen, deren Parallelisierung explizit untersagt werden muß. In der SILP-Formulierung erfordert dies zusätzliche Binärvariablen (v-Variablen; vgl. Kapitel 6.2.2), während bei OASIC zur Modellierung der Parallelitätsverbote keine Binärvariablen zu erzeugen sind (vgl. Kapitel 6.3.1). Die niedrigste Laufzeit von allen SILP-Approximationen wird bei der Approximation durch Rundung erzielt; allerdings werden in der ermittelten Lösung zwei Instruktionen mehr als bei den anderen Approximationen erzeugt. App2a, App3 und App32 führen mit einer Instruktionsanzahl von 21 Instruktionen ebenfalls zu einem suboptimalen Ergebnis; die optimale Instruktionsanzahl wird jedoch nur um 5 % überschritten.

	SILP	OASIC	SILP/OASIC
Nebenbedingungen	383	1190	1 : 3,11
Variablen	214	359	1 : 1,68
Binärvariablen	93	≤ 359	–
x-Variablen	58	≤ 359	–
y-Variablen	–	–	–
v-Variablen	35	–	–
p/q-Variablen	–	–	–
Größe des LP [Byte]	15625	162385	1 : 10,39

Tabelle 8.14: Charakteristika der für `whetp3.s` generierten ILPs ohne Berücksichtigung der Registerallokation.

Modellierung	Methode	Instruktionen	CPU-Zeit	Q
OASIC	Default	20	18 min 30 sec	0,15
	App1a	21	23,8 sec	$3,2 \cdot 10^-3$
SILP	Default	20	2h 4 min	1
	App1a	23	65,98 sec	$8,87 \cdot 10^-3$
	App2a	21	4 min 7 sec	0,03
	App3	21	13 min 35 sec	0,11
	App32	21	85,96 sec	0,01

Tabelle 8.15: Laufzeitergebnisse von `whetp3.s` ohne Berücksichtigung der Registerallokation.

Modellierung	Methode	Instruktionen	CPU-Zeit	Abweichung [%]
OASIC	B1	16	1,08	20
	B2	16	1,07	20
SILP	B1	16	2,6 sec	20
	B2	15	0,26 sec	25

Tabelle 8.16: Berechnung unterer Schranken bei `whetp3.s` ohne Berücksichtigung der Registerallokation. Die Spalte "Abweichung" zeigt den prozentualen Abstand von der optimalen Instruktionsanzahl.

Modellierung	Methode	Instruktionen	CPU-Zeit	Abweichung [%]
SILP	B1	16	24,9 sec	20
	B2	15	0,94 sec	25

Tabelle 8.17: Berechnung unterer Schranken bei `whetp3.s` mit Berücksichtigung der Registerallokation. Die Spalte "Abweichung" zeigt den prozentualen Abstand von der optimalen Instruktionsanzahl.

Die Ergebnisse der Schrankenberechnung ohne Registerallokation sind in Tabelle 8.16 dargestellt. Für die SILP-Modellierung werden die unteren Schranken auch mit Berücksichtigung der Registerallokation ermittelt. Dabei ergeben sich die in Tabelle 8.17 dargestellten Ergebnisse; die OASIC-Berechnungen mußten aus Platzgründen abgebrochen werden.

8.2.5 Histogramm

Ein Bildhistogramm ist ein Feld, das die Anzahl der Vorkommen jeder Graustufe in einem Bild angibt. Kann ein Pixel eines typischen monochromatischen Bildes 256 verschiedene Werte annehmen, sind für das Histogramm 256 Feldelemente erforderlich. In jedem Element wird die Anzahl der Vorkommen der entsprechenden Graustufe gespeichert. Der im Eingabeprogramm `histo.s` verwendete Algorithmus geht davon aus, daß das Graustufenbild in einem Puffer des Datenspeichers liegt; das Histogramm wird ebenfalls im Datenspeicher aufgebaut. Das Programm besteht aus 43 Mikrooperationen, darunter zwei Schleifenanweisungen. Aus Laufzeitgründen wurde wiederum auf eine vollständige Untersuchung der Instruktionsanordnung mit Berücksichtigung der Registerallokation verzichtet. Wieder werden Eingabeprogramm und eine optimal angeordnete Instruktionsfolge (ermittelt durch isolierte Flußanalyse) gegenübergestellt (vgl. Anhang C.5).

Eine beweisbar optimale Lösung konnte weder bei der SILP- noch bei der OASIC-Modellierung innerhalb von 24 Stunden ermittelt werden; auch die Approximation durch Rundung (App1a) auf der Basis der SILP-Formulierung kam in dieser Zeit zu keinem Ergebnis. Die Approximationen App2a, App2b, App3 und App32 hingegen konnten Ergebnisse

	SILP	OASIC	SILP/OASIC
Nebenbedingungen	716	4536	1 : 6,33
Variablen	434	1344	1 : 3,1
Binärvariablen	168	$\leq$ 1344	–
x-Variablen	130	1328	–
y-Variablen	16	16	–
v-Variablen	22	–	–
p/q-Variablen	–	–	–
Größe des LP [Byte]	30373	1794288	1 : 59,07

Tabelle 8.18: Charakteristika der für `histo.s` generierten ILPs ohne Berücksichtigung der Registerallokation.

Modellierung	Methode	Instruktionen	CPU-Zeit
OASIC	Default	–	> 24 h
	App1a	31	21 h 20 min
SILP	Default	–	> 24 h
	App1a	–	> 24 h
	App1b	37	3 h 15 min
	App2a	31	3 h 44 min
	App2b	31	15 min 2 sec
	App3	31	1 h 59 min
	App32	31	1 h 2 min

Tabelle 8.19: Laufzeitergebnisse von `histo.s` ohne Berücksichtigung der Registerallokation.

Modellierung	Methode	Instruktionen	CPU-Zeit	Abweichung [%]
SILP	B1	27	156,46 sec	12,9
	B2	24	16,19 sec	22,6

Tabelle 8.20: Berechnung unterer Schranken bei `histo.s` mit Berücksichtigung der Registerallokation. Die Spalte "Abweichung" zeigt den prozentualen Abstand von der optimalen Instruktionsanzahl.

ermitteln, die, wie die Betrachtung der Ergebnisinstruktionsfolgen zeigt, optimale Lösungen darstellen. Das schnellste Verfahren ist App2b mit einer Laufzeit von 15 Minuten. Die Basisblockfixierung nach der Lösung der ersten Relaxation (vgl. Kapitel 7.2) führte hier also zu einer erheblichen Beschleunigung (die Laufzeit von App2b beträgt 6 % der Laufzeit von App2a), ohne die Qualität des Ergebnisses zu beeinträchtigen. Das schnellste globale Verfahren ist App32; auch hier wird eine optimale Lösung ermittelt.

Die Berechnung unterer Schranken wird für die SILP-Formulierung mit Berücksichtigung der Registerallokation durchgeführt. Dabei ergeben sich die in Tabelle 8.17 dargestellten Ergebnisse. Wieder können bei vergleichsweise geringer Laufzeit gute untere Schranken der optimalen Instruktionsanzahl berechnet werden.

8.2.6 Konvolutionsberechnung

Das Beispielprogramm `conv.s` entstammt aus dem Bereich der Bildverarbeitung; es wird eine Funktion berechnet, die zur Bildrekonstruktion eingesetzt wird. Bildrekonstruktion aus Projektionen erlaubt es, von geeigneten Messungen ausgehend die interne Struktur eines untersuchten Objektes bildlich darzustellen ([Now90]). Die wichtigste Anwendung der Bildrekonstruktion ist die Computertomographie; die meistverbreitete Bildrekonstruktionsmethode ist die Konvolutionsmethode. Sie rekonstruiert ein Bild in zwei Schritten: Konvolution der Projektionsdaten mit einer gegebenen Funktion und Rückprojektion der transformierten Projektionsdaten. Die formale Definition einer Konvolution lautet wie folgt ([Her80]): Es seien zwei Funktionen Φ und Ψ in $\mathbb{R}$ gegeben. Ihre Konvolution ist eine Funktion $\Phi * \Psi$ mit

$$[\Phi * \Psi](v) = \int_{-\infty}^{\infty} \Phi(u)\Psi(v-u)du.$$

In sequentialisierter Form besteht das Assemblerprogramm aus 124 Mikroinstruktionen. Um die Berechnungszeit einzuschränken, wurde aus diesem Programm ein Basisblock extrahiert und die resultierende Codesequenz als Eingabe der ILP-Optimierung verwendet. Der Basisblock enthält 49 Instruktionen, darunter fast ausschließlich Additionen, Multiplikationen, sowie Datenzugriffe aus DM und PM. Aus Gründen der Laufzeit wird dieses Eingabeprogramm - mit Ausnahme der Berechnung unterer Schranken - nur für das rei-

	SILP	OASIC	SILP/OASIC
Nebenbedingungen	963	5559	1 : 5,77
Variablen	452	1905	1 : 4.21
Binärvariablen	163	$\leq$ 1905	–
x-Variablen	160	$\leq$ 1905	–
y-Variablen	–	–	–
v-Variablen	3	–	–
p/q-Variablen	–	–	–
Größe des LP [Byte]	41049	3347178	1 : 81,54

Tabelle 8.21: Charakteristika der für `conv.s` generierten ILPs ohne Berücksichtigung der Registerallokation.

Modellierung	Methode	Instruktionen	CPU-Zeit	Q
OASIC	Default	17	1 h 45 min	0,87
	App1a	18	2 min 26 sec	0,02
SILP	Default	17	2 h 1 min	1
	App1a	21	77 sec	0,01
	App2a	17	5 min 47 sec	0,05
	App3	17	11 min 42 sec	0,0968
	App32	17	53,66 sec	$7,4 \cdot 10^{-3}$

Tabelle 8.22: Laufzeitergebnisse von `conv.s` ohne Berücksichtigung der Registerallokation.

ne Instruktionsanordnungsproblem untersucht. Das Eingabeprogramm ist in Anhang C.6 aufgeführt. Dort ist ebenfalls eine optimale Anordnung (ermittelt mit SILP/Default) angegeben.

Zur Berechnung der optimalen Lösung auf der Basis der OASIC-Formulierung werden nur 87 % der bei SILP erforderlichen Laufzeit benötigt; der OASIC-Ansatz erweist sich hier also als effizienter. Bei der Approximation durch Rundung für den SILP-Ansatz kommt es nur selten zur Unlösbarkeit; meistens war die Rundung korrekt in dem Sinne, daß keine zulässige Lösung verhindert wird. Es brauchen im Laufe des Berechnungsverfahrens nur wenige Binärvariable berücksichtigt zu werden, so daß die Laufzeit sehr niedrig ist. Hier zeigt sich jedoch auch eine Schwäche dieser Approximation: ihr Ergebnis ist um 23 % schlechter als das der anderen Berechnungsverfahren. Bei App3 enthalten auch die zu lösenden Teil-ILPs viele Binärvariablen; die Laufzeit ist daher höher als bei den anderen Approximationen. Die schnellste Laufzeit erreicht die Kombination der schrittweisen Approximation und der isolierten Flußanalyse: App32 benötigt weniger als 0,1 % der zur exakten Lösung der SILP-Formulierung erforderlichen Laufzeit.

Modellierung	Methode	Instruktionen	CPU-Zeit	Abweichung [%]
OASIC	B1	15	36,47	11
	B2	15	36,04	11
SILP	B1	12	0,553 sec	29
	B2	12	0,379 sec	29

Tabelle 8.23: Berechnung unterer Schranken bei `conv.s` ohne Berücksichtigung der Registerallokation. Die Spalte "Abweichung" zeigt den prozentualen Abstand von der optimalen Instruktionsanzahl.

Modellierung	Methode	Instruktionen	CPU-Zeit	Abweichung [%]
SILP	B1	12	30 min 22 sec	29
	B2	12	11,1 sec	29

Tabelle 8.24: Berechnung unterer Schranken bei `conv.s` mit Berücksichtigung der Registerallokation. Die Spalte "Abweichung" zeigt den prozentualen Abstand von der optimalen Instruktionsanzahl.

Die Berechnung unterer Schranken kann bei Verwendung der SILP-Modellierung auch unter Berücksichtigung der Registerallokation schnell durchgeführt werden. Die OASIC-Berechnung mußte jedoch abgebrochen werden, da das generierte ILP zu groß wurde (> 100 MB). Daher werden in Tabelle 8.23 die Ergebnisse der Schrankenberechnung ohne Einbezug der Registerallokation dargestellt. Die Tabelle 8.24 zeigt die Ergebnisse der Schrankenberechnung auf der Grundlage der SILP-Modellierung mit Berücksichtigung der Registerallokation.

8.3 Gesamtauswertung

Vergleicht man die Resultate der untersuchten Modellierungsansätze SILP und OASIC, so zeigt sich, daß die Verwendung von SILP zur Lösung des Instruktionsanordnungs- und Registerallokationsproblems besser geeignet ist. Die Testresultate zeigen zunächst, daß die SILP-Modellierung zur Berechnung der optimalen Lösung bei kleinen Eingabeprogrammen dem OASIC-Ansatz überlegen ist. Bei steigender Problemgröße kehrt sich dieses Verhältnis jedoch um; zur Ermittlung der optimalen Lösung größerer Eingabeprogramme scheint also der Einsatz von OASIC vorzuziehen zu sein. Wird jedoch Instruktionsanordnung unter Berücksichtigung der Registerallokation ausgeführt, so kann der Speicherbedarf zum Anlegen der ILPs der OASIC-Formulierung stark anwachsen. Es kann sich also als günstiger erweisen, die höheren Laufzeiten der SILP-Formulierung in Kauf zu nehmen.

Die Überlegenheit der SILP-Modellierung tritt erst bei der Betrachtung der Approxi-

Programm	list scheduling ff[a]	lr[b]	md[c]	hl[d]	critical path	optimal
fir	10	10	10	10	10	8
dft	16	16	16	16	16	14
conv3x3	17	17	18	18	18	17
histo	31	31	31	31	31	31
cascade	9	9	15	9	11	7
whetp3	22	22	25	21	23	20

[a] *first fit*
[b] *longest remain*
[c] *max depend*
[d] *highest level first*

Tabelle 8.25: Ergebnisse von konventionellen graphbasierten Algorithmen.

mationen deutlich hervor. Der OASIC-Ansatz ist approximativen Berechnungen weniger zugänglich; auf der Grundlage der SILP-Modellierung hingegen können durch verschiedene Approximationen in relativ kurzer Zeit sehr gute Lösungen (in den getesteten Programmen mit einer Ausnahme stets optimale Lösungen) ermittelt werden – auch für Problemgrößen, die keine exakte Lösungsbestimmung mehr zulassen. Auch bei der Berechnung unterer Schranken ist die SILP-Modellierung dem OASIC-Ansatz deutlich überlegen. Für alle getesteten Programme konnte unter Berücksichtigung der Registerallokation eine gute untere Schranke meist bei einer Laufzeit im Sekundenbereich ermittelt werden. Die entsprechenden Berechnungen bei OASIC mußten hingegen in drei Fällen (`whetp3.s, histo.s, conv.s`) wegen zu hohen Platzbedarfes abgebrochen werden, so daß untere Schranken nur für das reine Instruktionsanordnungsproblem ohne Registerallokationsbedingungen ermittelt werden konnten. Zudem beträgt die erforderliche Laufzeit ein Vielfaches der bei SILP gemessenen Werte.

Vergleicht man die SILP-Approximationen untereinander, so zeigt sich, daß die schrittweise Approximation der isolierten Flußanalyse (App32) am effizientesten ist. Mit diesem Verfahren konnte in den Testprogrammen stets eine optimale Lösung bei niedriger Laufzeit ermittelt werden. Die Überlegenheit dieser Approximation zeigt sich erst bei größeren Problemgrößen; bei kleineren Programmen kann die reduzierte Binärvariablenzahl der zu lösenden Teilprobleme den organisatorischen Aufwand noch nicht ausgleichen (vgl. Kapitel 7.4).

Abschließend werden die untersuchten ILP-Ansätze mit konventionellen, graphbasierten Algorithmen verglichen. Die im Rahmen von [Lan97] analysierten graphbasierten Algorithmen benötigen stets eine Laufzeit von weniger als eine Sekunde. Die Qualität der Lösungen kann an Tabelle 8.25 (entnommen aus [Lan97]) abgelesen werden.

Bei dem besten Verfahren (List-Scheduling unter Verwendung der Heuristik "highest level"(hl)) beträgt die Abweichung von der optimalen Lösung durchschnittlich 13 %; die größte Abweichung wurde mit 28 % bei `cascade.s` gemessen. Die konventionellen Algorithmen überschreiten somit die optimale Instruktionsanzahl um durchschnittlich mindestens 13 %. Durch die Berechnung unterer Schranken mit Hilfe der ganzzahligen linearen Programmierung wird ein Zielfunktionswert ermittelt, der durchschnittlich 15 % unter der optimalen Instruktionsanzahl liegt[1]. Die Güte der Berechnung unterer Schranken durch Einsatz der ganzzahligen linearen Programmierung entspricht also der Lösungsqualität der konventionellen, graphbasierten Algorithmen.

Als eigenständiges Verfahren zur Instruktionsanordnung und Registerallokation ist die ILP-Optimierung aufgrund der hohen Komplexität nicht denkbar. Die Untersuchungen zeigen jedoch, daß gegenüber den konventionellen, graphbasierten Algorithmen durchaus Verbesserungen in der Codequalität bei akzeptablen Laufzeiten erzielt werden können. Hierzu ist eine Integration der ILP-Optimierung in einen graphbasierten Algorithmus erforderlich. Dieser muß überprüfen, ob eine bestimmte Instruktionsteilfolge vielversprechend für eine ILP-Optimierung ist (Schleifen- oder Basisblockstruktur). Gegebenenfalls muß dann die ILP-Optimierung der ausgewählten Codesequenz angestoßen werden.

[1] Dieser Wert wurde als Mittelwert der beim Verfahren B1 auf der Grundlage der SILP-Formulierung auftretenden Abweichungen für die Analysen unter Berücksichtigung der Registerallokation ermittelt.

Kapitel 9

Zusammenfassung und Ausblick

Es konnte gezeigt werden, daß durch Erweiterungen sowohl des SILP- als auch des OASIC-Modellierungsansatzes die für die Instruktionsanordnung relevanten Berechnungsvorgänge eines realen Prozessors vollständig und korrekt in Form von ganzzahligen linearen Programmen modelliert werden können. Instruktionsanordnung ist auf beliebigen Eingabeprogrammen möglich; bei der Integration der Registerallokation hingegen ergeben sich Einschränkungen. Da die linearen Programme für eine feste Codesequenz erstellt werden, ist es bei den untersuchten Ansätzen nicht möglich, das Hinzufügen oder Entfernen von Instruktionen in einer geschlossenen ILP-Darstellung zu berücksichtigen. Aufspaltungen von Lebensspannen sowie Einfügungen von Zwischenspeicherungen werden daher nicht berücksichtigt. Zwei andere Teilaufgaben der Registerallokation, nämlich die Registermengenzuweisung und die konkrete Registerzuteilung können hingegen berechnet werden. Aus Komplexitätsgründen erweist es sich als sinnvoll, bei der SILP-Modellierung für Eingabeprogramme mit nichtlinearem Programmfluß sowie grundsätzlich bei der OASIC-Modellierung auf die Ermittlung der vollständigen Lösung der Registerzuteilung zu verzichten. Es wird dann nur die optimale Registermengenzuteilung berechnet. Die konkrete Registerzuteilung wird einem nachgestellten Registerallokator überlassen, der die aus der Lösung des ILP's erhaltenen Registermengenverteilung berücksichtigen muß.

Die Analyse der Testresultate hat gezeigt, daß in den meisten Fällen die SILP-Modellierung der in OASIC verwendeten Darstellung überlegen ist. Von der SILP-Formulierung ausgehend können verschiedene Approximationen berechnet werden, die bei geringerer Laufzeit auch zu guten Resultaten führen. Die Optimalität des Ergebnisses ist allerdings bei solchen Approximationen nicht mehr garantiert; dennoch können bessere Ergebnisse erzielt werden, als bei den in [Lan97] untersuchten konventionellen, graphbasierten Algorithmen. Die Verwirklichung einer völlig eigenständigen Instruktionsanordnung und Registerallokation auf der Basis ganzzahliger linearer Programmierung ist aus Komplexitätsgründen nicht erfolgversprechend. Als vielversprechend ist hingegen die Einbettung in einen konventionellen Algorithmus anzusehen, der für passende Codesequenzen (Schleifen, Basisblöcke) als Unterfunktion eine ILP-Optimierung anstößt.

Ein weiteres wichtiges Verwendungsgebiet des ILP-Ansatzes besteht in der Ermittlung unterer Schranken für die optimale Lösung. Bei konventionellen, graphbasierten Algorithmen ist es nicht möglich, eine Abschätzung für die Güte der Lösung anzugeben. Durch das Lösen von gemischt ganzzahligen linearen Programmen können untere Schranken für die optimale Lösung ermittelt werden. Die Güte dieser unteren Schranken entspricht bei den untersuchten Programmen der Güte der Lösungen, die durch Einsatz konventioneller, graphbasierter Algorithmen berechnet werden können. Es ist somit möglich, ein Intervall anzugeben, in dem sich die optimale Lösung mit Sicherheit befindet und eine Abschätzung für die Qualität einer approximativen Lösung zu erhalten, auch wenn das optimale Ergebnis aus Komplexitätsgründen nicht ermittelt werden kann.

Anhang A

Instruktionssatz des ADSP-2106x

In den folgenden Kapiteln wird der Instruktionssatz des ADSP-2106x im Detail vorgestellt. Zunächst werden die Instruktionsformate beschrieben. Viele dieser Formate enthalten ein Feld zur Spezifizierung einer Berechnungsoperation. Da es hierfür eine große Anzahl von Möglichkeiten gibt, werden sie in Anhang A.2 gesondert aufgelistet. Auch die Multifunktionsinstruktionen werden in einem eigenen Abschnitt (Anhang A.3) beschrieben.

A.1 Instruktionsformate

Die Instruktionsformate sind von 1 bis 23 durchnumeriert; einige besitzen mehr als eine syntaktische Form. Viele Instruktionen können in Abhängigkeit einer Bedingung ausgeführt werden, was durch ein vorgestelltes *IF condition* dargestellt wird. Eine Übersicht über die möglichen Bedingungscodes ist in Anhang A.4 gegeben. Kursiv gedruckte Instruktionsteile sind optional; sie können auch weggelassen werden. Die Instruktionsformate werden getrennt nach den in Kapitel 2.9 eingeführten Kategorien aufgelistet. Die von einer Instruktion durchgeführten Aktionen werden kurz beschrieben; eine Übersicht über die verwendeten Notation ist in Anhang A.1.5 gegeben.

A.1.1 Compute- und Move or Modify-Instruktionen

1. $$\textit{compute},\ \left|\begin{array}{l}\text{DM(Ia, Mb)=dreg1}\\ \text{dreg1=DM(Ia, Mb)}\end{array}\right|,\ \left|\begin{array}{l}\text{PM(Ic, Md)=dreg2}\\ \text{dreg2=PM(Ic, Md)}\end{array}\right|;$$

Es wird eine parallele Datenübertragung zwischen Datenspeicher und Registerfile sowie zwischen Programmspeicher und Registerfile ausgeführt. Datenspeicher (DM) und Programmspeicher (PM) werden über die angegebenen I-Register adressiert. Die Werte der I-Register werden nachträglich um die Werte der angegebenen M-Register in- bzw. dekrementiert.

2. *IF condition* compute;

   Es wird eine Berechnungsoperation ausgeführt. Wird eine Bedingung angegeben (*condition*), wirkt sie sich auf die gesamte Instruktion aus.

3a. *IF condition compute,* $\left|\begin{array}{l}\text{DM(Ia, Mb)}\\ \text{PM(Ic, Md)}\end{array}\right|\text{=ureg;}$

3b. *IF condition compute,* $\left|\begin{array}{l}\text{DM(Mb, Ia)}\\ \text{PM(Md, Ic)}\end{array}\right|\text{=ureg;}$

3c. *IF condition compute,* $\text{ureg=}\left|\begin{array}{l}\text{DM(Ia, Mb)}\\ \text{PM(Ic, Md)}\end{array}\right|\text{;}$

3d. *IF condition compute,* $\text{ureg=}\left|\begin{array}{l}\text{DM(Mb, Ia)}\\ \text{PM(Md, Ic)}\end{array}\right|\text{;}$

Es wird eine Datenübertragung zwischen Daten- oder Programmspeicher und einem Universalregister durchgeführt. Das I-Register gibt die Speicheradresse an, auf die zugegriffen werden soll. Der Wert des I-Registers wird entweder im voraus (Stellung M,I) oder im nachhinein (Stellung I,M) um den Wert des M-Registers in- bzw. dekrementiert. Wird eine Berechnungsoperation spezifiziert (*compute*), wird sie parallel zum Datenzugriff ausgeführt. Wird eine Bedingung angegeben (*condition*), so wirkt sie sich auf die gesamte Instruktion aus.

4a. *IF condition compute,* $\left|\begin{array}{l}\text{DM(Ia, <data6>)}\\ \text{PM(Ic, <data6>)}\end{array}\right|\text{=dreg;}$

4b. *IF condition compute,* $\left|\begin{array}{l}\text{DM(<data6>, Ia)}\\ \text{PM(<data6>), Ic}\end{array}\right|\text{=dreg;}$

4c. *IF condition compute,* $\text{dreg=}\left|\begin{array}{l}\text{DM(Ia, <data6>)}\\ \text{PM(Ic, <data6>)}\end{array}\right|\text{;}$

4d. *IF condition compute,* $\text{dreg=}\left|\begin{array}{l}\text{DM(<data6>, Ia)}\\ \text{PM(<data6>), Ic}\end{array}\right|\text{;}$

Dieses Instruktionsformat spezifiziert eine Datenübertragung zwischen Daten- oder Programmspeicher und einem Universalregister. Das I-Register gibt die Speicheradresse an, auf die zugegriffen werden soll. Der Wert des I-Registers wird entweder im voraus (Stellung Datum,I) oder im nachhinein (Stellung I,Datum) um den unmittelbaren Datenwert <data6> in- bzw. dekrementiert. Wird eine Berechnungsoperation spezifiziert (*compute*), wird sie parallel zum Datenzugriff ausgeführt. Wird eine Bedingung angegeben (*condition*), so wirkt sie sich auf die gesamte Instruktion aus.

5. *IF condition compute*, ureg1=ureg2;

Durchgeführt wird ein Datentransfer zwischen zwei Unversalregistern. Wird eine Berechnungsoperation spezifiziert, wird sie parallel zur Datenübertragung ausgeführt. Wird eine Bedingung angegeben, so wirkt sie sich auf die gesamte Instruktion aus.

6a. *IF condition* shiftimm, $\left|\begin{array}{l}\text{DM(Ia, Mb)}\\ \text{PM(Ic, Md)}\end{array}\right|$=dreg;

6b. *IF condition* shiftimm, dreg=$\left|\begin{array}{l}\text{DM(Ia, Mb)}\\ \text{PM(Ic, Md)}\end{array}\right|$;

Eine unmittelbare Shifteroperation verwendet als Y-Operand einen unmittelbaren Datenwert. Wird ein Datentransfer zwischen Daten- oder Programmspeicher und Registerfile spezifiziert, wird er parallel zur Shifteroperation ausgeführt. Das I-Register wird im nachhinein um den Wert des M-Registers in- bzw. dekrementiert. Wird eine Bedingung angegeben, wirkt sie sich auf die gesamte Instruktion aus.

7. *IF condition compute*, MODIFY$\left|\begin{array}{l}\text{(Ia, Mb)}\\ \text{(Ic, Md)}\end{array}\right|$;

Der Wert des angegebenen I-Register wird um den Wert des M-Registers in- bzw. dekrementiert. Wird eine Berechnungsoperation angegeben, wird sie parallel ausgeführt. Wird eine Bedingung angegeben, so wirkt sie sich auf die gesamte Instruktion aus.

A.1.2 Kontrollflußinstruktionen

8\. $$\textit{IF condition}\ \text{JUMP} \left|\begin{array}{l} \text{<addr24>} \\ \text{(PC, <reladdr24>)} \end{array}\right| \left|\begin{array}{l} \textit{(DB)} \\ \textit{(LA)} \\ \textit{(CI)} \\ \textit{(DB, LA)} \\ \textit{(DB, CI)} \end{array}\right|;$$

$$\textit{IF condition}\ \text{CALL} \left|\begin{array}{l} \text{<addr24>} \\ \text{(PC, <reladdr24>)} \end{array}\right| \textit{(DB)};$$

Es wird ein Sprung oder Prozeduraufruf zu der spezifierten absoluten oder PC-relativen Adresse ausgeführt. Ist der Modifikator *(DB)* angegeben, wird die Instruktion verzögert (*delayed*), sonst unverzögert (*non-delayed*) ausgeführt. Ist der Schleifenabbruchmodifikator *(LA)* angegeben, werden die obersten Einträge von Schleifen- und PC-Keller bei der Ausführung des Sprunges entfernt. Der *Clear Interrupt*- Modifikator (*CI*) erlaubt die erneute Ausführung eines Interrupts, während die Serviceroutine dieses Interrupts ausgeführt wird.

9\. $$\textit{IF condition}\ \text{JUMP} \left|\begin{array}{l} \text{(Md, Ic)} \\ \text{(PC, <reladdr6>)} \end{array}\right| \left|\begin{array}{l} \textit{(DB)} \\ \textit{(LA)} \\ \textit{(CI)} \\ \textit{(DB, LA)} \\ \textit{(DB, CI)} \end{array}\right|, \left|\begin{array}{l} \textit{compute} \\ \textit{ELSE compute} \end{array}\right|;$$

$$\textit{IF condition}\ \text{CALL} \left|\begin{array}{l} \text{(Md, Ic)} \\ \text{(PC, <reladdr6>)} \end{array}\right| \textit{(DB)}, \left|\begin{array}{l} \textit{compute} \\ \textit{ELSE compute} \end{array}\right|;$$

Ausgeführt wird ein Sprung oder Prozeduraufruf zur spezifierten PC-relativen Adresse bzw. zu der durch das zuvor modifizierte Indexregister angegebenen Adresse. Ist eine Berechnungsoperation ohne *ELSE* angegeben, wird sie parallel zum Sprung ausgeführt. Ansonsten wird sie nur ausgeführt, wenn die angegebene Bedingung nicht erfüllt ist.

10\. $$\textit{IF condition}\ \text{JUMP} \left|\begin{array}{l} \text{(Md, Ic)} \\ \text{(PC, <reladdr6>)} \end{array}\right|, \text{ELSE} \left|\begin{array}{l} \textit{compute}\text{, DM(Ia, Mb)=dreg)} \\ \textit{compute}\text{, dreg=DM(Ia, Mb))} \end{array}\right|;$$

Ist die Bedingung erfüllt, so wird ein bedingter Sprung zu der angegebenen PC-relativen Adresse oder zu der Adresse, die durch das im voraus modifizierte I-Register spezifiziert ist, ausgeführt. Ansonsten wird eine optionale Berechnungsoperation parallel zu einer Datenübertragung zwischen dem Datenspeicher und dem Registerfile durchgeführt.

11. $$\textit{IF condition}\ \mathrm{RTS}\ \left|\begin{array}{l} \textit{(DB)}\\ \textit{(LR)}\\ \textit{(DB, LR)}\end{array}\right|,\ \left|\begin{array}{l} \textit{compute}\\ \textit{ELSE compute}\end{array}\right|;$$

$$\textit{IF condition}\ \mathrm{RTI}\ \textit{(DB)},\ \left|\begin{array}{l} \textit{compute}\\ \textit{ELSE compute}\end{array}\right|;$$

Es wird ein Prozedur- bzw. Interruptrücksprung durchgeführt.

12. $$\mathrm{LCNTR}{=}\left|\begin{array}{l} \text{<data16>}\\ \text{ureg}\end{array}\right|,\ \mathrm{DO}\ \left|\begin{array}{l} \text{<addr24>}\\ \text{(PC, <reladdr24>)}\end{array}\right|\ \text{UNTIL LCE;}$$

Es wird eine zählerbasierte Schleife ausgeführt. Der Schleifenzähler LCNTR wird mit dem Datenwert <data16> oder dem Inhalt eines Universalregisters geladen. Die Schleifeneinsprungadresse wird auf den PC-Keller gelegt; die Adresse der letzten Schleifeninstruktion und die LCE-Bedingung werden auf dem Schleifenadreßkeller abgelegt. Der Wert des LCNTR-Wert wird auf den Schleifenzählerkeller gelegt und dem Schleifenzähler CURLCNTR zugewiesen. Die Schleife wird so lange ausgeführt, bis der CURLCNTR den Wert 0 erreicht.

13. $$\mathrm{DO}\ \left|\begin{array}{l} \text{<addr24>}\\ \text{(PC, <reladdr24>)}\end{array}\right|\ \text{UNTIL termination;}$$

Es wird eine Schleife in Abhängigkeit einer expliziten Bedingung ausgeführt. Die Schleifeneinsprungadresse wird auf den PC-Stack gelegt; die Adresse der letzten Schleifeninstruktion und die Abbruchbedingung werden auf den Schleifenkeller gelegt. Die Schleife wird so lange ausgeführt, bis die Abbruchbedingung erfüllt ist.

A.1.3 Unmittelbare Datentransportinstruktionen

14a. $\left|\begin{array}{l}\text{DM(<addr32>)}\\ \text{PM(<addr24>)}\end{array}\right|\text{=ureg;}$

14b. $\text{ureg=}\left|\begin{array}{l}\text{DM(<addr32>)}\\ \text{PM(<addr24>)}\end{array}\right|\text{;}$

Ausgeführt wird eine Datenübertragung zwischen Daten- oder Programmspeicher und einem Universalregister durch Verwendung von direkter Adressierung.

15a. $\left|\begin{array}{l}\text{DM(<addr32>, Ia)}\\ \text{PM(<addr24>, Ic)}\end{array}\right|\text{=ureg;}$

15b. $\text{ureg=}\left|\begin{array}{l}\text{DM(<addr32>, Ia)}\\ \text{PM(<addr24>, Ic)}\end{array}\right|\text{;}$

Es wird eine Datenübertragung zwischen Daten- oder Programmspeicher und einem Universalregister durch Verwendung von indirekter Adressierung durch ein Indexregister durchgeführt. Dieses wird im voraus um den angegebenen unmittelbaren Datenwert <data32> in- oder dekrementiert.

16. $\left|\begin{array}{l}\text{DM(Ia, Mb)}\\ \text{PM(Ic, Md)}\end{array}\right|\text{=<data32>;}$

Ein unmittelbarer Datenwert <data32> wird im Daten- oder Programmspeicher abgespeichert. Es wird indirekte Adressierung verwendet; das Indexregister wird nachträglich um den Wert des M-Registers verändert (postmodifiziert).

17. ureg=<data32>;

Ein unmittelbarer Datenwert <data32> wird einem Unversalregister zugewiesen.

A.1.4 Verschiedene Instruktionen

18. $\text{BIT} \left|\begin{array}{l}\text{SET}\\ \text{CLR}\\ \text{TGL}\\ \text{TST}\\ \text{XOR}\end{array}\right| \text{ sreg <data32>;}$

    Hier werden Bitmanipulationen auf Systemregistern spezifiziert. Diese Instruktion kann einzelne Bits setzen, löschen, kippen oder testen, oder das Systemregister mit einem angegebenen Datenwert vergleichen (XOR-Verknüpfung).

19a. $\text{MODIFY} \left|\begin{array}{l}\text{(Ia, <data32>)}\\ \text{(Ic, <data24>)}\end{array}\right|;$

19b. $\text{BITREV} \left|\begin{array}{l}\text{(Ia, <data32>)}\\ \text{(Ic, <data24>)}\end{array}\right|;$

    Das spezifizierte I-Register wird um den angegebenen unmittelbaren Datenwert in- bzw. dekrementiert.

20. $\left|\begin{array}{l}PUSH\\ POP\end{array}\right| LOOP, \left|\begin{array}{l}PUSH\\ POP\end{array}\right| STS, \left|\begin{array}{l}PUSH\\ POP\end{array}\right| PCSTK, FLUSH\ CACHE;$

    Es wird ein *Push* bzw. *Pop* auf Schleifenadreß-, Schleifenzähler-, Status- und PC-Keller ausgeführt. Alle Mikrooperationen können auch isoliert ausgeführt werden.

21. NOP;

    Eine Leeroperation; nur die Abrufadresse (*Fetch*-Adresse) wird erhöht.

22. IDLE;

    Der Prozessor wird schlafengelegt, bis er durch einen Interrupt wieder aufgeweckt wird.

23. $\text{CJUMP} \left|\begin{array}{l}\text{function}\\ \text{(PC, <reladdr24>)}\end{array}\right| \text{(DB);}$

RFRAME;

Die CJUMP-Instruktion wird vom C-Compiler für Funktionsaufrufe verwendet. CJUMP kombiniert einen direkten oder PC-relativen Sprung mit Registertransferoperationen zum Abspeichern von Rahmen- und Kellerzeiger. Für detaillierte Informationen hierzu sei auf [Ana95c] verwiesen. Die RFRAME-Instruktionen stellt die Werte von Rahmen- und Kellerzeiger wieder her.

A.1.5 Notation

Notation	Bedeutung
GROSS	Explizite Syntax – Assembler-Schlüsselwort
;	Semikolon (Instruktionsterminator)
kursiv	Optionaler Teil der Instruktion
$\left\lvert\begin{array}{l}\text{Option1}\\ \text{Option2}\end{array}\right\rvert$	Liste von Optionen, von denen eine auszuwählen ist
compute	ALU-, Multiplizierer-, Shifter- oder Multifunktionsoperation
shiftimm	Shifter Immediate-Operation
condition	Statusbedingungen (siehe Kapitel A.4)
termination	Schleifenterminierungs-Bedingungen (siehe Kapitel A.4)
ureg	Universalregister
sreg	Systemregister
dreg	Datenregister (Registerfile): R15-R0 oder F15-F0
Ia	I7-I0 (DAG1 Indexregister)
Mb	M7-M0 (DAG1 Modifyregister)
Ic	I15-I8 (DAG2 Indexregister)
Md	M15-M8 (DAG2 Modifyregister)
<data$n$>	n-bit Konstante
<addr$n$>	n-bit Adresse
<reladdr$n$>	n-bit relative Adresse
(DB)	Verzögerung einer Sprunganweisung
(LA)	Schleifenunterbrechung (entfernt oberstes Element von Schleifen- und PC-Keller)
(CI)	Lösche Interrupt

A.2 Instruktionen der Berechnungseinheiten

Das 23-Bit breite Teilfeld einer einfachen Berechnungsoperation besitzt folgende Gestalt:

<table>
<tr><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td colspan="2">CU</td><td colspan="8">OPCODE</td><td colspan="4">RN</td><td colspan="4">RX</td><td colspan="4">RY</td></tr>
</table>

Die durch das OPCODE-Feld spezifizierte Instruktion wird in der durch CU bestimmten Berechnungseinheit ausgeführt. Die beiden Operanden sind durch RX bzw RY festgelegt, das Zielregister wird im RN-Feld angegeben. Die Codierung der Berechnungseinheiten kann der folgenden Tabelle entnommen werden.

CU=00	Arithmetisch-logische Einheit
CU=01	Multiplizierer
CU=10	Shifter

Im folgenden werden die arithmetischen Instruktionen des ADSP-2106x aufgelistet; gegliedert sind sie nach den Berechnungseinheiten, auf denen sie ausgeführt werden. Dabei werden für jede Instruktion mögliche Änderungen von Kontroll- und Statusbits angegeben. Die Semantik der in den Testprogrammen (vgl. Kapitel 8.2) auftretenden arithmetischen Operationen ist intuitiv verständlich; für eine detaillierte Beschreibung aller Berechnungsoperationen sei daher auf [Ana95c] verwiesen.

A.2.1 ALU

Instruktion	*ASTAT Status Flags*								*STKY Status Flags*			
	AZ	AV	AN	AC	AS	AI	AF	CACC	AUS	AVS	AOS	AIS
Fixkomma:												
c Rn=Rx+Ry	*	*	*	*	0	0	0	–	–	–	**	–
c Rn=Rx-Ry	*	*	*	*	0	0	0	–	–	–	**	–
c Rn=Rx+Ry+CI	*	*	*	*	0	0	0	–	–	–	**	–
c Rn=Rx+Ry+CI-1	*	*	*	*	0	0	0	–	–	–	**	–
Rn=(Rx+Ry)/2	*	0	*	*	0	0	0	–	–	–	–	–
COMP(Rx, Ry)	*	0	*	0	0	0	0	*	–	–	–	–
Rn=Rx+CI	*	*	*	*	0	0	0	–	–	–	**	–
Rn=Rx+CI-1	*	*	*	*	0	0	0	–	–	–	**	–
Rn=Rx+1	*	*	*	*	0	0	0	–	–	–	**	–
Rn=Rx-1	*	*	*	*	0	0	0	–	–	–	**	–
c Rn=-Rx	*	*	*	*	0	0	0	–	–	–	**	–
c Rn=ABS Rx	*	*	0	0	*	0	0	–	–	–	**	–
Rn=PASS Rx	*	0	*	0	0	0	0	–	–	–	–	–
c Rn=Rx OR Ry	*	0	*	0	0	0	0	–	–	–	–	–
c Rn=Rx XOR Ry	*	0	*	0	0	0	0	–	–	–	–	–
c Rn=NOT Rx	*	0	*	0	0	0	0	–	–	–	–	–
Rn=MIN(Rx, Ry)	*	0	*	0	0	0	0	–	–	–	–	–
Rn=MAX(Rx, Ry)	*	0	*	0	0	0	0	–	–	–	–	–
Rn=CLIP Rx BY Ry	*	0	*	0	0	0	0	–	–	–	–	–
Fließkomma:												
Fn=Fx+Fy	*	*	*	0	0	*	1	–	**	**	–	**
Fn=Fx-Fy	*	*	*	0	0	*	1	–	**	**	–	**
Fn=ABS(Fx+Fy)	*	*	0	0	0	*	1	–	**	**	–	**
Fn=ABS(Fx-Fy)	*	*	0	0	0	*	1	–	**	**	–	**
Fn=(Fx+Fy)/2	*	0	*	0	0	*	1	–	**	–	–	**
COMP(Fx, Fy)	*	0	*	0	0	*	1	*	–	–	–	**
Fn=-Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn=ABS Fx	*	*	0	0	*	*	1	–	–	**	–	**
Fn=PASS Fx	*	0	*	0	0	*	1	–	–	–	–	**
Fn=RND Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn=SCALB Fx BY Fy	*	*	*	0	0	*	1	–	**	**	–	**
Rn=MANT Fx	*	*	0	0	*	*	1	–	–	**	–	**
Rn=LOBG Fx	*	*	*	0	0	*	1	–	–	**	–	**
Rn=FIX Fx BY Ry	*	*	*	0	0	*	1	–	**	**	–	**
Rn=FIX Fx	*	*	*	0	0	*	1	–	**	**	–	**
Fn=FLOAT Rx BY Ry	*	*	*	0	0	0	1	–	**	**	–	–
Fn=FLOAT Rx	*	0	*	0	0	0	1	–	–	–	–	–
Fn=RECIPS Fx	*	*	*	0	0	*	1	–	**	**	–	**
Fn=RSQRTS Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn=Fx COPYSIGN Fy	*	0	*	0	0	*	1	–	–	–	–	**
Fn=MIN(Fx, Fy)	*	0	*	0	0	*	1	–	–	–	–	**
Fn=MAX(Fx, Fy)	*	0	*	0	0	*	1	–	–	–	–	**
FN=CLIP Fx BY FY	*	0	*	0	0	*	1	–	–	–	–	**

Rn, Rx, Ry = Beliebiges Fixkomma-Register
Fn, Fx, Fy = Beliebiges Fließkomma-Register
c = ADSP-21xx-kompatible Instruktionen

* wird entsprechend dem Resultat der Operation gesetzt oder gelöscht
** wird entsprechend dem Resultat der Operation gesetzt aber nie gelöscht
– keine Auswirkung

A.2.2 Multiplizierer

Instruktion	*ASTAT Flags* MU	MN	MV	MI	*ASTAT Flags* MUS	MOS	MVS	MIS
Festkomma:								
$\left\lvert\begin{array}{l}\mathrm{Rn}\\ \mathrm{MRF}\\ \mathrm{MRB}\end{array}\right\rvert = \mathrm{Rx{*}Ry}\ \ (\left\lvert\begin{array}{l}\mathrm{S}\\ \mathrm{U}\end{array}\right\rvert \left\lvert\begin{array}{l}\mathrm{S}\\ \mathrm{U}\end{array}\right\rvert \left\lvert\begin{array}{l}\mathrm{F}\\ \mathrm{I}\\ \mathrm{FR}\end{array}\right\rvert)$	*	*	*	0	–	**	–	–
$\left\lvert\begin{array}{l}\mathrm{Rn} = \mathrm{MRF}\\ \mathrm{Rn} = \mathrm{MRB}\\ \mathrm{MRF} = \mathrm{MRF}\\ \mathrm{MRB} = \mathrm{MRB}\end{array}\right\rvert + \mathrm{Rx{*}Ry}\ \ (\left\lvert\begin{array}{l}\mathrm{S}\\ \mathrm{U}\end{array}\right\rvert \left\lvert\begin{array}{l}\mathrm{S}\\ \mathrm{U}\end{array}\right\rvert \left\lvert\begin{array}{l}\mathrm{F}\\ \mathrm{I}\\ \mathrm{FR}\end{array}\right\rvert)$	*	*	*	0	–	**	–	–
$\left\lvert\begin{array}{l}\mathrm{Rn} = \mathrm{MRF}\\ \mathrm{Rn} = \mathrm{MRB}\\ \mathrm{MRF} = \mathrm{MRF}\\ \mathrm{MRB} = \mathrm{MRB}\end{array}\right\rvert - \mathrm{Rx{*}Ry}\ \ (\left\lvert\begin{array}{l}\mathrm{S}\\ \mathrm{U}\end{array}\right\rvert \left\lvert\begin{array}{l}\mathrm{S}\\ \mathrm{U}\end{array}\right\rvert \left\lvert\begin{array}{l}\mathrm{F}\\ \mathrm{I}\\ \mathrm{FR}\end{array}\right\rvert)$	*	*	*	0	–	**	–	–
$\left\lvert\begin{array}{l}\mathrm{Rn} = \mathrm{SAT\ MRF}\\ \mathrm{Rn} = \mathrm{SAT\ MRB}\\ \mathrm{MRF} = \mathrm{SAT\ MRF}\\ \mathrm{MRB} = \mathrm{SAT\ MRB}\end{array}\right\rvert \ \left\lvert\begin{array}{l}\mathrm{(SI)}\\ \mathrm{(UI)}\\ \mathrm{(SF)}\\ \mathrm{(UF)}\end{array}\right\rvert$	*	*	*	0	–	**	–	–
$\left\lvert\begin{array}{l}\mathrm{Rn} = \mathrm{RND\ MRF}\\ \mathrm{Rn} = \mathrm{RND\ MRB}\\ \mathrm{MRF} = \mathrm{RND\ MRF}\\ \mathrm{MRB} = \mathrm{RND\ MRB}\end{array}\right\rvert \ \left\lvert\begin{array}{l}\mathrm{(SF)}\\ \mathrm{(UF)}\end{array}\right\rvert$	*	*	*	0	–	**	–	–
$\left\lvert\begin{array}{l}\mathrm{MRF}\\ \mathrm{MRB}\end{array}\right\rvert = 0$	0	0	0	0	–	–	–	–
$\left\lvert\begin{array}{l}\mathrm{MRxF}\\ \mathrm{MRxB}\end{array}\right\rvert = \mathrm{Rn}$	0	0	0	0	–	–	–	–
$\mathrm{Rn} = \left\lvert\begin{array}{l}\mathrm{MRxF}\\ \mathrm{MRxB}\end{array}\right\rvert$	0	0	0	0	–	–	–	–
Fließkomma:								
Fn=Fx*Fy	*	*	*	*	**	–	**	**

Rn, Rx, Ry = R0-R15; Fixkomma-Register
Fn, Fx, Fy = F0-F15; Fließkomma-Register
MRxF=MR0F, MR1F, MR2F; multiplier result accumulator, Vordergrund
MRxB=MR0B, MR1B, MR2B; multiplier result accumulator, Hintergrund

- * wird entsprechend dem Resultat der Operation gesetzt oder gelöscht
- ** wird entsprechend dem Resultat der Operation gesetzt aber nie gelöscht
- – keine Auswirkung

Optionale Modifizierer für Fixpunkt:

X-Operand, Y-Operand, Datenformat, Rundung

$(\lvert\bullet\rvert \quad \lvert\bullet\rvert \quad \lvert\bullet\rvert)$

S	Vorzeichenbehafteter Operand
U	Vorzeichenfreier Operand
I	Ganzzahliger Operand
F	Reeller Operand
FR	Reeller Operand, gerundetes Ergebnis
(SF)	Standardformat für 1-Operand-Operation
(SSF)	Standardformat für 2-Operanden-Operation

A.2.3 Shifter

Bei einigen Shifterinstruktionen spezifiziert das *RN*-Feld einen dritten Operanden, der nach der Berechnung aktualisiert wird. In diesem Fall wird das angegebene Register als Operanden- und Zielregister verwendet.

	Instruktion	*Flags* SZ	SV	SS
c	Rn=LSHIFT Rx BY Ry	*	*	0
c	Rn=LSHIFT Rx BY <data8>	*	*	0
c	Rn=Rn OR LSHIFT Rx BY Ry	*	*	0
c	Rn=Rn OR LSHIFT Rx BY <data8>	*	*	0
c	Rn=ASHIFT Rx BY Ry	*	*	0
c	Rn=ASHIFT Rx BY <data8>	*	*	0
c	Rn=Rn OR ASHIFT Rx BY Ry	*	*	0
c	Rn=Rn OR ASHIFT Rx BY <data8>	*	*	0
	Rn=ROT Rx BY Ry	*	0	0
	Rn=ROT Rx BY <data8>	*	0	0
	Rn=BCLR Rx BY Ry	*	*	0
	Rn=BCLR Rx BY <data8>	*	*	0
	Rn=BSET Rx BY Ry	*	*	0
	Rn=BSET Rx BY <data8>	*	*	0
	Rn=BTGL Rx BY Ry	*	*	0
	Rn=BTGL Rx BY <data8>	*	*	0
	BTST Rx BY Ry	*	*	0
	BTST Rx BY <data8>	*	*	0
	Rn=FDEP Rx BY Ry	*	*	0
	Rn=FDEP Rx BY <bit6>:<len6>	*	*	0
	Rn=Rn OR FDEP Rx BY Ry	*	*	0
	Rn=Rn OR FDEP Rx BY <bit6>:<len6>	*	*	0
	Rn=FDEP Rx BY Ry (SE)	*	*	0
	Rn=FDEP Rx B> <bit6>:<len6> (SE)	*	*	0
	Rn=Rn OR FDEP Rx BY Ry (SE)	*	*	0
	Rn=Rn OR FDEP Rx B> <bit6>:<len6> (SE)	*	*	0
	Rn=FEXT Rx BY Ry	*	*	0
	Rn=FEXT Rx BY <bit6>:<len6>	*	*	0
	Rn=FEXT Rx BY Ry (SE)	*	*	0
	Rn=FEXT Rx BY <bit6>:<len6> (SE)	*	*	0
c	Rn=EXP Rx (EX)	*	0	*
c	Rn=EXP Rx	*	0	*
	Rn=LEFTZ Rx	*	*	0
	Rn=LEFTO Rx	*	*	0
	Rn=FPACK Fx	0	*	0
	Fn=FUNPACK Rx	0	0	0

Rn, Rx, Ry = Beliebiges Fixkomma-Register
Fn, Fx, Fy = Beliebiges Fließkomma-Register; Fließkomma-Wort
c = ADSP-2100-kompatible Instruktionen

* wird entsprechend dem Resultat der Operation gesetzt oder gelöscht

22	21–20	19–16	15–12	11–8	7–4	3–0
0	00	x111	RS	RA	RX	RY

Abbildung A.1: Das 23 Bit breite Teilfeld einer kombinierten Addition/Subtraktion.

22	21–16	15–12	11–8	7–6	5–4	3–2	1–0
1	OPCODE	RM	RA	RXM	RYM	RXA	RYA

Abbildung A.2: Das 23 Bit breite Teilfeld bei der Parallelausführung ganzzahliger Multiplizierer- und ALU-Operationen.

A.3 Multifunktionsinstruktionen

Multifunktionsinstruktionen erscheinen in drei Ausprägungen, die sich hinsichtlich der Form des 23 Bit breiten Teilfeldes der Instruktion unterscheiden:

1. kombinierte Addition/Subtraktion
2. kombinierte Multiplikation und ALU-Operation
3. Multiplikation mit gleichzeitiger kombinierter Addition/Subtraktion

Die kombinierte Addition/Subtraktion berechnet Summe und Differenz zweier Operanden und schreibt die Ergebnisse in zwei verschiedene Register. Die Syntax lautet wie folgt:

Ra=Rx+Ry, Rs=Rx-Ry

Fa=Fx+Fy, Fs=Fx-Fy

Die Aufteilung des Berechnungsfeldes im Instruktionswort ist in Abbildung A.1 dargestellt.

Die parallele Ausführung von ALU und Multiplizierer ist nicht für beliebige Operationen möglich. Zulässige Kombinationen bestehen aus einer Multiplikation bzw. akkumulierenden Multiplikation und einer der folgenden ALU-Operationen: Addition, Subtraktion und Durchschnittsbildung im ganzzahligen Fall; Addition, Subtraktion, Multiplikation, Zahlenformatkonvertierung, Absolutbetrag-, Minimum- und Maximumbildung bei Fließkommaoperationen. Das Berechnungsfeld des zugehörigen Instruktionsformates ist für Festkommazahlen in Abbildung A.2 und für Fließkommazahlen in Abbildung A.3 dargestellt. Tabelle A.1 zeigt alle zulässigen Operationskombinationen.

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	OPCODE						FM				FA				FXM		FYM		FXA		FYA	

Abbildung A.3: Das 23 Bit breite Teilfeld bei Parallelausführung von Multiplizierer und ALU für Gleitkommazahlen.

Rm=R3-0*R7-4 (SSFR), Ra=R11-8+R15-12
Rm=R3-0*R7-4 (SSFR), Ra=R11-8–R15-12
Rm=R3-0*R7-4 (SSFR), Ra=(R11-8+R15-12)/2

MRF=MRF+R3-0*R7-4 (SSF), Ra=R11-8+R15-12
MRF=MRF+R3-0*R7-4 (SSF), Ra=R11-8–R15-12
MRF=MRF+R3-0*R7-4 (SSF), Ra=(R11-8+R15-12)/2

Rm=MRF+R3-0*R7-4 (SSFR), Ra=R11-8+R15-12
Rm=MRF+R3-0*R7-4 (SSFR), Ra=R11-8–R15-12
Rm=MRF+R3-0*R7-4 (SSFR), Ra=(R11-8+R15-12)/2

MRB=MRB+R3-0*R7-4 (SSF), Ra=R11-8+R15-12
MRB=MRB+R3-0*R7-4 (SSF), Ra=R11-8–R15-12
MRB=MRB+R3-0*R7-4 (SSF), Ra=(R11-8+R15-12)/2

Rm=MRB+R3-0*R7-4 (SSFR), Ra=R11-8+R15-12
Rm=MRB+R3-0*R7-4 (SSFR), Ra=R11-8–R15-12
Rm=MRB+R3-0*R7-4 (SSFR), Ra=(R11-8+R15-12)/2

Fm=F3-0*F7-4, Fa=F11-8+F15-12
Fm=F3-0*F7-4, Fa=F11-8–F15-12
Fm=F3-0*F7-4, Fa=FLOAT F11-8 BY R15-12
Fm=F3-0*F7-4, Fa=FIX F11-8 BY R15-12
Fm=F3-0*F7-4, Fa=(F11-8+F15-12)/2
Fm=F3-0*F7-4, Fa=ABS F11-8
Fm=F3-0*F7-4, Fa=MAX(F11-8, F15-12)
Fm=F3-0*F7-4, Fa=MIN(F11-8, F15-12)

Tabelle A.1: Mögliche Kombinationen von Multiplizierer- und ALU-Operationen zu Multifunktionsinstruktionen.

22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2	1 0
1	10	RS	RM	RA	RXM	RYM	RXA	RYA

Abbildung A.4: Das 23 Bit breite Teilfeld bei Parallelausführung von Multiplizierer und kombinierter Addition/Subtraktion für Festkommazahlen.

22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2	1 0
1	11	FS	FM	FA	FXM	FYM	FXA	FYA

Abbildung A.5: Das 23 Bit breite Teilfeld bei Parallelausführung von Multiplizierer und kombinierter Addition/Subtraktion für Gleitkommazahlen.

Auch die gleichzeitige Ausführung einer Multiplikation bzw. einer akkumulierenden Multiplikation mit einer kombinierten Addition/Subtraktion ist zulässig. Abbildung A.4 zeigt den relevanten Instruktionsformatausschnitt für ganzzahlige Berechnungen, Abbildung A.5 für Fließkommaberechnungen.

A.4 Bedingungen und Terminierungscodes

In einer bedingten Anweisung ist die Ausführung der gesamten Instruktion abhängig von der angegebenen Bedingung.

Bedingung	Bedeutung
EQ	ALU gleich Null
NE	ALU ungleich Null
LT	ALU kleiner Null
GE	ALU größer oder gleich Null
LE	ALU kleiner oder gleich NULL
GT	ALU größer Null
AC	ALU carry gesetzt
NOT AC	ALU carry nicht gesetzt
AV	ALU overflow gesetzt
NOT AV	ALU overflow nicht gesetzt
MV	Multiplizierer overflow gesetzt
NOT MV	Multiplizierer overflow nicht gesetzt
MS	Multiplizierer sign gesetzt
NOT MS	Multiplizierer sign nicht gesetzt
SV	Shifter overflow gesetzt
NOT SV	Shifter overflow nicht gesetzt

SZ	Shifter zero gesetzt
NOT SZ	Shifter zero nicht gesetzt
FLAG0_IN	Flag 0 Eingabe gesetzt
NOT FLAG0_IN	Flag 0 Eingabe nicht gesetzt
FLAG1_IN	Flag 1 Eingabe gesetzt
NOT FLAG1_IN	Flag 1 Eingabe nicht gesetzt
FLAG0_IN	Flag 2 Eingabe gesetzt
NOT FLAG2_IN	Flag 2 Eingabe nicht gesetzt
FLAG0_IN	Flag 3 Eingabe gesetzt
NOT FLAG3_IN	Flag 3 Eingabe nicht gesetzt
TF	Bittest-Flag gesetzt
NOT TF	Bittest-Flag nicht gesetzt
BM	Busmaster-Flag gesetzt
NBM	Busmaster-Flag nicht gesetzt
LCE	Schleifenzählen abgelaufen (DO UNTIL)
FOREVER	Immer wahr (DO UNTIL)
NOT LCE	Schleifenzählen nicht abgelaufen (IF)
TRUE	Immer wahr (IF)

A.5 Universalregister

Register	**Funktion**
Datenregister	
R15-R0	Ganzzahlregister
F15-F0	Fließkommaregister
Kontrolleinheit	
PC	Programmzähler (nur lesen)
PCSTK	Oberstes Element des PC-Kellers
PCSTKP	Zeiger auf oberstes Element des PC-Kellers
FADDR	Fetch-Adresse (nur lesen)
DADDR	Decode-Adresse (nur lesen)
LADDR	Schleifenterminierungsadresse
CURLCNTR	Aktueller Schleifenzähler
LCNTR	Schleifenzähler für nächste verschachtelte zählergesteuerte Schleife
Datenadreßgeneratoren	
I7-I0	DAG1 Indexregister

M7-M0	DAG1 Modifyregister
L7-L0	DAG1 Längenregister
B7-B0	DAG1 Basisregister
I15-I8	DAG2 Indexregister
M15-M8	DAG2 Modifyregister
L15-L8	DAG2 Längenregister
B15-B8	DAG2 Basisregister

Busaustausch

PX1	PMD-DMD Busaustausch 1 (16-bit)
PX2	PMD-DMD Busaustausch 2 (32-bit)
PX	48-bit Kombination aus PX1 und PX2

Zeitgeber

TPERIOD	Periode
TCOUNT	Zähler

Systemregister

MODE1	Moduskontrolle und Status
MODE2	Moduskontrolle und Status
IRPTL	Interruptlatch
IMASK	Interruptmaske
IMASKP	Interruptmaskenzeiger (für Schachtelung)
ASTAT	Arithmetisches Statusflag, Bittest-Flag, etc.
STKY	Sticky-Arithmetisches Statusflag, Bittest-Flag, etc.
USTAT1	Benutzerstatus-Register 1
USTAT2	Benutzerstatus-Register 2

Anhang B

Algorithmen

B.1 Berechnung des Dominatorbaumes

Es sei ein Kontrollflußgraph G mit Knotenmenge N, Kantenmenge E und Startknoten n_{start} gegeben. Wir berechnen iterativ die Menge $D(n)$ der Dominatoren aller Knoten n ([ASU88]). Am Ende der Berechnung ist d genau dann in $D(n)$ enthalten, wenn gilt $d \, \Delta_d \, n$.

$D(n_{start}) = \{n_{start}\}$;

Für alle $n \in N - \{n_{start}\}$

  $D(n) = N$;

Ende Für

Solange Veränderungen in einem $D(n)$ auftreten

  Für alle $n \in N - \{n_{start}\}$

    Sei P die Menge aller Vorgänger von n.
    $D(n) = \{n\} \cup \bigcap_{p \in P} D(p)$; (*)

  Ende Für

Ende Solange

Algorithmus B.1: Berechnung des Dominatorbaumes.

Man kann zeigen, daß die in (*) erzeugte Menge $D(n)$ immer eine Teilmenge der zuletzt berechneten Menge $D(n)$ ist. Da $D(n)$ nicht unendlich oft verkleinert werden kann, muß die Schleife terminieren.

B.2 Berechnung des Kontrollabhängigkeitsgraphen

Sei ein Kontrollflußgraph mit zugehörigem Postdominatorbaum gegeben und sei S die Menge aller Kanten (A, B) des Kontrollflußgraphen, so daß B kein Vorfahr von A im Postdominatorbaum ist (d.h. B postdominiert A nicht). Jede dieser Kanten ist mit T oder F markiert. Der Algorithmus untersucht jede Kante (A, B) in S.

Sei L der früheste gemeinsame Vorgänger von A und B im Postdominatorbaum. Nach Konstruktion gilt $L \neq B$. Es gilt weiter: $L = A$ oder L ist Vorgänger von A im Postdominatorbaum ([FOW87]). Es gibt zwei Fälle für L:

1. $L =$ Vorgänger von A: Alle Knoten im Postdominatorbaum auf dem Pfad von L nach B, darunter B, aber nicht L, sollten kontrollabhängig von A sein.
2. $L = A$: Alle Knoten im Postdominatorbaum auf dem Pfad von A nach B, darunter A und B, sollten kontrollabhängig von A sein. (Dieser Fall erfaßt Schleifenabhängigkeiten.)

Ist nun eine Kante (A, B) gegeben, laufen wir von B ausgehend im Postdominatorbaum rückwärts, bis wir den Vorgänger von A erreichen (sofern er existiert) und markieren alle Knoten, die vor dem Vorgänger von A besucht wurden, als kontrollabhängig von A. Im ersten Fall liegt A nicht auf dem Pfad zu L und wird daher nicht markiert. Nachdem alle Kanten in S untersucht wurden, sind alle Kontrollabhängigkeiten bestimmt.

Die Laufzeit des Algorithmus beträgt $\mathcal{O}(n^2)$, wobei n die Anzahl der Knoten des Kontrollflußgraphen angibt ([FOW87]).

B.3 Vorgehensweise des Simplex-Algorithmus

Im folgenden wird eine intuitive Beschreibung der Vorgehensweise des Simplex-Algorithmus vorgestellt. Für eine detailliertere Behandlung sei auf [Chv83], [Sch93] oder [Sei96] verwiesen.

Die Idee des Simplex-Algorithmus läßt sich anschaulich wie folgt beschreiben: Die Nebenbedingungen eines linearen Programmes LP definieren ein Polytop in $\mathbb{R}^n$. Das **Simplex-Theorem** besagt, daß mindestens eine der Ecken dieses Polytops eine optimale Lösung von LP darstellt. Um eine solche Ecke zu finden, wählt man eine beliebige Ecke des Polytops als Startpunkt und wechselt zu einer benachbarten Ecke, wenn diese einen besseren Wert bezüglich der Zielfunktion ergibt. Dieses Vorgehen wiederholt man so lange, bis die Ecke erreicht ist, die der optimalen Lösung entspricht. Ein möglicher Berechnungsverlauf im zweidimensionalen Fall ist in Abbildung B.1 gegeben.

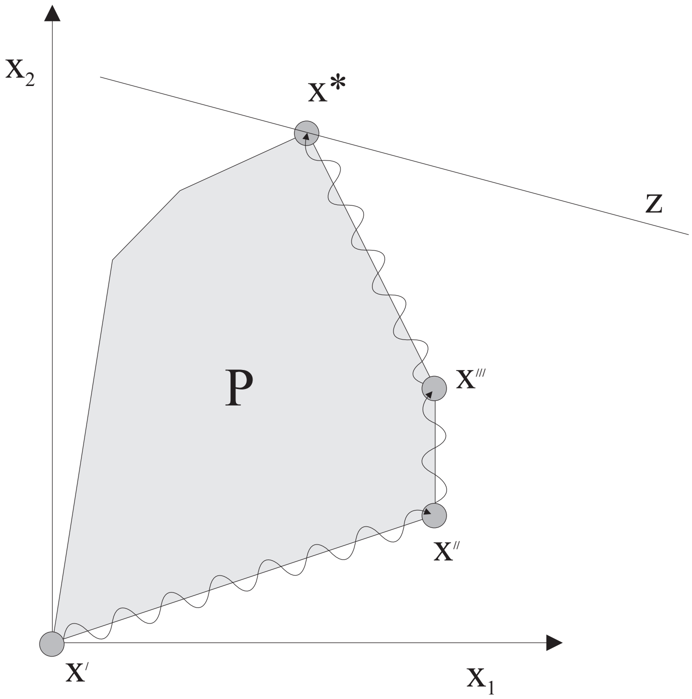


Abbildung B.1: Veranschaulichung der Idee des Simplex-Algorithmus im zweidimensionalen Fall. Dabei stellt P den Zulässigkeitsbereich des linearen Programmes dar; z ist die Zielfunktion. Als Startecke wird x' berechnet; danach über x'' und x''' die optimale Lösung x^* erreicht.

B.4 Der Bellman-Ford Algorithmus

Es sei ein gewichteter Graph $G = (V, E)$ mit einem Startknoten $s \in V$ gegeben. Zu berechnen ist der längste Weg von s zu allen anderen Knoten $v \in V$. Das Gewicht einer Kante $(u, v) \in E$ wird mit $w(u, v)$ bezeichnet. Am Ende des Algorithmus gibt der Wert $d(v)$ den längsten Weg von s nach v an und $\pi(v)$ ist der unmittelbare Vorgänger von v auf dem längsten Weg von s nach v. Im Verlaufe des Algorithmus werden die Distanzen $d(v)$ sukzessiv verbessert; das verwendete Verfahren wird als Relaxation[1] bezeichnet. Hierbei wird für eine Kante (u, v) überprüft, ob der bis zum aktuellen Zeitpunkt gefundenen längste Weg von s nach v verbessert werden kann, indem man zuerst von s nach u geht und dann die direkte Kante zwischen u und v benutzt. Sind Zyklen im Graphen enthalten, so werden diese erkannt und das Verfahren bricht ab. Die Laufzeit des Algorithmus beträgt $\mathcal{O}(|V| \cdot |E|)$.

```
INIT(G, s)
{
    Für alle u ∈ V
        d[u] = −∞;
        π[u] = NIL;
    Ende Für
    d[s] = 0;
}

RELAX(u, v)
{
    Wenn d(v) < d(u) + w(u, v) Dann
        d[v] = d[u] + w(u, v);
        π[v] = u;
    Ende Wenn
}
```

[1]Diese Relaxation darf nicht mit dem im Gebiet der ganzzahligen linearen Programmierung verwendeten Relaxationsbegriff verwechselt werden.

```
BELLMAN-FORD(G, s)
{
    INIT(G, s);
    Wiederhole |V| − 1 mal
        Für alle (u, v) ∈ E
            RELAX(u, v);
        Ende Für
    Ende Wiederhole
    Für alle (u, v) ∈ E
        Wenn d(v) < d(u) + w(u, v) Dann
            Es liegt ein Zyklus vor;
            Verlasse Funktion;
        Ende Wenn
    Ende Für
}
```

B.5 Berechnung der transitiven Hülle

Die transitive Hülle des Datenabhängigkeitsgraphen wird mittels eines leicht modifizierten Floyd-Warshall-Algorithmus berechnet [CLR90]. Benötigt wird eine Adjazenzmatrix M, deren Einträge m_{ij} angeben, ob eine Abhängigkeitskante zwischen den Mikrooperationen i und j besteht. Ist m_{ij} gleich 1, existiert eine direkte Abhängigkeit, andernfalls nicht. Der Algorithmus liefert die Adjazenzmatrix D^n der transitiven Hülle des Datenabhängigkeitsgraphen zurück. Gilt $d_{ij}^n = 1$, so ist Mikrooperation j transitiv abhängig von Mikrooperation i. Die Laufzeit des Algorithmus beträgt $\mathcal{O}(n^3)$, wobei n die Anzahl der Knoten des Abhängigkeitsgraphen angibt.

$D^0 = M$;

Für $k = 1$ **Bis** n

    Für $i = 1$ **Bis** n

        Für $j = 1$ **Bis** n

            $D_{ij}^k = D_{ij}^{k-1} \vee (D_{ik}^{k-1} \wedge D_{kj}^{k-1})$;

        Ende Für

    Ende Für

Ende Für

return D^n;

Algorithmus B.2: Modifizierter Floyd-Warshall-Algorithmus zur Berechnung der transitiven Hülle bei gegebener nxn-Adjazenzmatrix M.

Anhang C

Testprogramme

Die folgenden Seiten enthalten eine zweispaltige Auflistung der untersuchten Beispielprogramme. Die linke Spalte enthält die Instruktionsfolge, die als Eingabe des Optimierungsprozesses verwendet wird; die rechte Spalte zeigt jeweils eine optimale Anordnung. Zu einer Instruktion zusammengefaßte Mikrooperationen werden durch Kommata getrennt; Instruktionen werden mittels Strichpunkten abgegrenzt (vgl. Kapitel 2.9). Eine detaillierte Beschreibung der einzelnen Programme ist in Kapitel 8.2 gegeben. Die wichtigsten Charakteristika der Eingabeprogramme werden überblicksartig in Tabelle C.1 zusammengefaßt (vgl. auch [Lan97]).

Alle Assemblercodesequenzen werden eingeleitet und abgeschlossen durch Assemblerdirektiven, die Beginn bzw. Ende eines Codesegmentes spezifizieren. Für nähere Angaben hierzu sei auf [Ana95a] verwiesen. Die Direktiven stellen keine Instruktionen dar und werden daher bei der Ermittlung der Instruktionsanzahl nicht berücksichtigt. Labels stehen in eigenen Zeilen der Maschinenprogramme; sie beziehen sich stets auf die in der nächsten Zeile stehende Instruktion.

Programm	Instruktionen	Basisblöcke	Schleifen	Abhängigkeiten
fir	18	3	1	59
cascade	20	1	0	63
dft	26	4	2	63
whetp3	26	1	0	103
histo	43	4	2	186
conv	49	1	0	193

Tabelle C.1: Übersicht über die verwendeten Beispielprogramme.

C.1 fir.s

```
.segment /PM pm_code;
fir:
 dm(i0,m0)=f0;
 f0=dm(i0,m0);
 f4=pm(i8,m8);
 f8=f0*f4;
 f0=dm(i0,m0);
 f4=pm(i8,m8);
 f12=f0*f4;
 f0=dm(i0,m0);
 f4=pm(i8,m8);
 lcntr=r1, do macs until lce;
  f8=f8+f12;
  f12=f0*f4;
  f0=dm(i0,m0);
macs:
  f4=pm(i8,m8);
 f8=f8+f12;
 f12=f0*f4;
 f0=f8+f12;
 rts;
.endseg;
```

```
.segment /pm pm_code;
fir:
 dm(i0,m0)=f0, f4=pm(i8,m8);
 f0=dm(i0,m0);
 f0=dm(i0,m0), f4=pm(i8,m8), f8=f0*f4;
 f0=dm(i0,m0), f4=pm(i8,m8), f12=f0*f4;
 lcntr=r1, DO macs until lce;
macs:
  f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8), f12=f0*f4;
 f12=f0*f4, f8=f8+f12;
 rts, f0=f8+f12;
.endseg;
```

C.2 cascade.s

```
.segment /PM pm_code;
cascaded_biquad:
 f12=0;
 f8=0;
 f2=dm(i0,m1);
 f4=pm(i8,m1);
 f8=f8+f12;
 f12=f2*f4;
 f3=dm(i0,m1);
 f4=pm(i8,m8);
 f8=f8+f12;
 f12=f3*f4;
 dm(i1,m1)=f3;
 f4=pm(i8,m8);
 f8=f8+f12;
 f12=f2*f4;
 f2=dm(i0,m1);
 f4=pm(i8,m8);
 dm(i1,m1)=f8;
 f8=f8+f12;
 f12=f3*f4;
 f4=pm(i8,m8);
.endseg;
```

```
.segment /pm pm_code;
cascaded_biquad:
 f8=0.000000;
 f2=dm(i0,m1), f4=pm(i8,m1);
 f12=0.000000;
 f3=dm(i0,m1), f4=pm(i8,m8), f8=f8+f12, f12=f2*f4;
 dm(i1,m1)=f3, f4=pm(i8,m8), f8=f8+f12, f12=f3*f4;
 f2=dm(i0,m1), f4=pm(i8,m8), f8=f8+f12, f12=f2*f4;
 f4=pm(i8,m8), dm(i1,m1)=f8, f8=f8+f12, f12=f3*f4;
.endseg;
```

C.3 dft.s

```
.segment/PM pm_code;
dft:
 i10=0;
 l10=0;
 f15=0.0;
 lcntr=64, do outer until lce;
  f8=pass f15;
  m8=i10;
  f9=pass f15;
  f0=dm(i0,m1);
  f5=pm(i9,m8);
  f12=f0*f5;
  f4=pm(i8,m8);
  lcntr=63, do inner until lce;
   f13=f0*f4;
   f9=f9+f12;
   f0=dm(i0,m1);
   f5=pm(i9,m8);
   f12=f0*f5;
   f8=f8-f13;
inner:
   f4=pm(i8,m8);
  f13=f0*f4;
  f9=f9+f12;
  f8=f8-f13;
  dm(i2,m1)=f9;
  modify(i10,m9);
outer:
  dm(i1,m1)=f8;
 rts;
.endseg;
```

```
.segment /pm pm_code;
dft:
 f15=0.000000;
 l10=0;
 i10=0;
 lcntr=64, DO outer until lce;
  f8=pass f15, m8=i10;
  f9=pass f15, f0=dm(i0,m1), f5=pm(i9,m8);
  f12=f0*f5, f4=pm(i8,m8);
  lcntr=63, DO inner until lce;
   f9=f9+f12, f0=dm(i0,m1), f5=pm(i9,m8), f13=f0*f4;
inner:
   f12=f0*f5, f8=f8-f13, f4=pm(i8,m8);
  f13=f0*f4, f9=f9+f12, modify(i10,m9);
  dm(i2,m1)=f9, f8=f8-f13;
outer:
  dm(i1,m1)=f8;
 rts;
.endseg;
```

C.4 whetp3.s

```
.segment /pm seg_pmco;
_p3:
 modify(i7,-3);
 f13=f8;
 f1=dm(100);
 f4=f4+f13;
 f5=dm(200);
 f8=f1*f4;
 f2=recips f5;
 i4=r12;
 f4=f8+f13;
 f13=f1*f4;
 f9=0x40000000;
 f12=f2*f5;
 f4=f8+f13;
 f4=f2*f4;
 f2=f9-f12;
 f12=f2*f12;
 f4=f2*f4;
 f2=f9-f12;
 f12=f2*f12;
 f4=f2*f4;
 f2=f9-f12;
 f4=f2*f4;
 f2=f2-f12;
 f2=f2*f4;
 f2=f2+f4;
 dm(i4,m5)=f2;
.endseg;
```

```
.segment /pm seg_pmco;
_p3:
 f5=dm(200);
 f13=f8, f2=recips f5;
 f1=dm(100);
 f4=f4+f13, i4=r12;
 f9=1073741824.000000;
 f8=f1*f4;
 f4=f8+f13;
 f13=f1*f4;
 f12=f2*f5, f4=f8+f13;
 f2=f9-f12, f4=f2*f4;
 f12=f2*f12;
 f2=f9-f12, f4=f2*f4;
 modify(i7,-3);
 f12=f2*f12;
 f2=f9-f12, f4=f2*f4;
 f4=f2*f4;
 f2=f2-f12;
 f2=f2*f4;
 f2=f2+f4;
 dm(i4,m5)=f2;
.endseg;
```

C.5 histo.s

```
.segment /pm pm_code;
histo:
 r3=l1;
 r3=lshift r3 by -1;
 r14=i8;
 r3=r3-1;
 r15=i9;
 r0=dm(i0,m0);
 lcntr=r3, do hloop until lce;
  r0=r0+r14;
  r2=dm(i0,m0);
  r2=r2+r15;
  i8=r0;
  i9=r2;
  r1=pm(i8,m15);
  r1=r1+1;
  r3=pm(i9,m15);
  r3=r3+1;
  pm(i8,m15)=r1;
  r0=dm(i0,m0);
hloop:
  pm(i9,m15)=r3;
 r0=r0+r14;
 r2=dm(i0,m0);
 r2=r2+r15;
 i8=r0;
 i9=r2;
 r1=pm(i8,m15);
 r1=r1+1;
 r3=pm(i9,m15);
 r3=r3+1;
 pm(i8,m15)=r1;
 pm(i9,m15)=r3;
 i8=b8;
 i9=b9;
 r2=l1;
 r2=r2-1;
 r0=pm(i8,m8);
 r1=pm(i9,m8);
 lcntr=r2, do combine until lce;
```

```
.segment /pm pm_code;
histo:
 r3=l1;
 r15=i9;
 r3=lshift r3 by -1, r0=dm(i0,m0);
 r14=i8, r3=r3-1;
 lcntr=r3, DO hloop until lce;
  r0=r0+r14, r2=dm(i0,m0);
  r2=r2+r15, i8=r0;
  r1=pm(i8,m15);
  i9=r2;
  r1=r1+1, r3=pm(i9,m15), r0=dm(i0,m0);
  pm(i8,m15)=r1, r3=r3+1;
hloop:
  pm(i9,m15)=r3;
 r0=r0+r14, r2=dm(i0,m0);
 i8=r0;
 r2=r2+r15, r1=pm(i8,m15);
 i9=r2, r1=r1+1;
 r3=pm(i9,m15);
 r2=l1;
 pm(i8,m15)=r1, r3=r3+1;
 pm(i9,m15)=r3, r2=r2-1;
 i8=b8;
 r0=pm(i8,m8);
 i9=b9;
 r1=pm(i9,m8);
 lcntr=r2, DO combine until lce;
  r0=pm(i8,m8);
  r2=r0+r1;
combine:
  dm(i1,m0)=r2, r1=pm(i9,m8);
 r2=r0+r1;
 dm(i1,m0)=r2;
 rts;
.endseg;
```

```
  r0=pm(i8,m8);
  r2=r0+r1;
  dm(i1,m0)=r2;
combine:
  r1=pm(i9,m8);
 r2=r0+r1;
 dm(i1,m0)=r2;
 rts;
.endseg;
```

C.6 conv.s

```
.segment /pm pm_code;
f8=f8+f12;
f12=f2*f6;
f13=f1*f6;
f8=f8+f12;
f7=dm(i0,m1);
f9=f9+f13;
f13=f2*f7;
f4=dm(i0,m0);
f0=pm(i8,m8);
f12=f0*f4;
f9=f9+f13;
f5=dm(i0,m0);
f1=pm(i8,m8);
f13=f0*f5;
f8=f8+f12;
f6=dm(i0,m0);
f12=f1*f5;
f9=f9+f13;
f2=pm(i8,m8);
f8=f8+f12;
f12=f2*f6;
f13=f1*f6;
f8=f8+f12;
f7=dm(i0,m1);
f9=f9+f13;
f13=f2*f7;
f4=dm(i0,m0);
f0=pm(i8,m8);
f12=f0*f4;
f9=f9+f13;
f5=dm(i0,m0);
f1=pm(i8,m8);
f13=f0*f5;
f8=f8+f12;
f6=dm(i0,m0);
f12=f1*f5;
f9=f9+f13;
f2=pm(i8,m8);
f8=f8+f12;
f12=f2*f6;
```

```
.segment /pm pm_code;
f8=f8+f12, f13=f1*f6, f7=dm(i0,m1), f0=pm(i8,m8);
f9=f9+f13, f4=dm(i0,m0), f12=f2*f6, f1=pm(i8,m8);
f12=f0*f4, f5=dm(i0,m0), f8=f8+f12;
f12=f1*f5, f8=f8+f12;
f13=f2*f7, f8=f8+f12;
f6=dm(i0,m0), f13=f0*f5, f2=pm(i8,m8), f9=f9+f13;
f9=f9+f13, f12=f2*f6, f7=dm(i0,m1);
f8=f8+f12, f13=f1*f6, f4=dm(i0,m0), f0=pm(i8,m8);
f9=f9+f13, f12=f0*f4;
f13=f2*f7, f5=dm(i0,m0), f1=pm(i8,m8), f8=f8+f12;
f9=f9+f13, f12=f1*f5;
f6=dm(i0,m0), f13=f0*f5, f8=f8+f12;
f2=pm(i8,m8), f13=f1*f6, f9=f9+f13;
f7=dm(i0,m2), f12=f2*f6, f9=f9+f13;
f8=f8+f12, f13=f2*f7, modify(i0,m3);
f9=f9+f13, dm(i1,m0)=f8;
dm(i1,m0)=f9;
.endseg;
```

```
  f7=dm(i0,m2);
  f13=f1*f6;
  f8=f8+f12;
  f9=f9+f13;
  f13=f2*f7;
  modify(i0,m3);
  f9=f9+f13;
  dm(i1,m0)=f8;
  dm(i1,m0)=f9;
.endseg;
```

Anhang D

CIS-Optionen

`-b` Ermittlung einer unteren Schranke für die optimale Lösung (Verfahren B1; vgl. Kapitel 8.2)

`-c` Nur gültig in Verbindung mit den Optionen `-o` und `-s`. Eine Verschiebung der Instruktionen über die Basisblockgrenzen wird unterdrückt.

`-d` Hierdurch werden während der Laufzeit von `CIS` Debuginformationen ausgegeben.

`-l` Instruiert `CIS`, List-Scheduling als Anordnungsverfahren zu verwenden. Die Auswahl der Mikrooperationen aus der Menge *data_ready* kann durch die `-h` Option beeinflußt werden. `-r` erzwingt in diesem Kontext eine neue Registerallokation, die sich im Wesentlichen am potentiellem Parallelismus orientiert (vgl. [Lan97]).

`-r` Erzwingt eine Registerallokation bei der ausgewählten Optimierungsmethode.

`-s` Lösungsberechnung mit Hilfe ganzzahliger linearer Programmierung. Als ILP-Formulierung wird SILP verwendet. Mit der `-m` Option wird festgelegt, ob die exakte Lösung ermittelt oder eine Approximation berechnet werden soll.

`-o` Lösungsberechnung mit Hilfe ganzzahliger linearer Programmierung. Als ILP-Formulierung wird OASIC verwendet. Mit der `-m` Option wird festgelegt, ob die exakte Lösung ermittelt oder eine Approximation berechnet werden soll.

`-v` Mit dieser Optionen werden VCG-Dateien für verschiedene Phasen und Zwischendarstellungen erzeugt.

`-x` Ermittlung einer unteren Schranke für die optimale Lösung (Verfahren B2; vgl. Kapitel 8.2)

Anhang E

Glossar

CISC (*Complex Intruction Set Computer*) Eine Prozessorarchitektur, die komplexe Befehle zur Unterstützung des Zugriffs in Datenstrukturen und der Prozedurorganisation anbietet. Sie ist charakterisiert durch viele und komplexe Adressierungsarten, verschiedene Varianten von Operationen für Operanden unterschiedlicher Länge, unterschiedliche Ausführungszeiten für Befehle und in der Regel wenige Prozessorregister.

Harvard-Architektur Ursprünglich war dies eine Bezeichnung für eine Prozessorarchitektur mit getrenntem Speicher für Befehle und Daten. Abweichend von der ursprünglichen Bedeutung wird dieser Begriff heute auch für Maschinen mit einem Hauptspeicher, aber getrennten Befehls- und Datencaches benutzt ([HP94]).

Instruktion In einer Instruktion werden mehrere Mikrooperationen zusammengefaßt. Sie stellt ein Befehlswort dar, welches beim ADSP-2106x eine Länge von 48 Bit hat.

Mikrooperation Eine elementare Operation auf Prozessorniveau, wie z.B. Addition zweier Register oder Laden aus Speicherbereichen. Mehrere Mikrooperationen können – als Instruktion zusammengefaßt – parallel ausgeführt werden, sofern die zugrundeliegende Hardware dies erlaubt.

MIPS (*Million Instructions Per Second*) Für ein gegebenes Programm kann ein in MIPS gemessener Leistungswert nach folgender Gleichung ermittelt werden:

$$\mathrm{MIPS} = \frac{\text{Befehlsanzahl}}{\text{Ausführungszeit} \cdot 10^6}$$

Für eine genauere Beschreibung insbesondere auch der mit diesem Maß verbundenen Probleme sei auf [HP94] verwiesen.

RISC (*Reduced Instruction Set Computer*) Eine Prozessorarchitektur mit folgenden charakteristischen Merkmalen: rechnende Operationen arbeiten nur auf Registeroperanden und legen ihr Ergebnis ebenfalls in Registern ab. Zugriffe auf den Speicher erfolgen nur in Lade-/Speicheroperationen (*load/store*-Architektur). Durch wenige Adressierungsarten wird die Kontrollogik vereinfacht, die meist fest verdrahtet ist. RISC-Prozessoren besitzen oft sehr große Registermengen und verfügen über parallel arbeitende funktionale Einheiten.

Scheduler ist eine Bezeichnung für das Verfahren, welches für die Instruktionsanordnung (*Scheduling*) zuständig ist.

Taktfrequenz Reziproker Wert der Taktzykluszeit, gewöhnlich in MHz gemessen.

VLIW (*Very Large Instruction Word*) Architektur, die durch sehr lange Befehlswörter charakterisiert ist. Es sind mehrfache unabhängige Funktionseinheiten vorhanden; anstatt zu versuchen, diesen Einheiten mehrere unabhängige Instruktionen zu übergeben, packt eine VLIW-Architektur mehrere Mikrooperationen in eine sehr lange Instruktion.

LRU (*Least Recently Used*) Strategie, nach der der Block aus dem Cache verdrängt wird, auf den am längsten nicht mehr zugegriffen wurde.

Index

Literaturverzeichnis

[Ana91] ANALOG DEVICES: *ADSP-21020 User's Manual*, 1991.

[Ana95a] ANALOG DEVICES: *ADSP-21000 Family Assembler Tools and Simulator Manual*, 1995.

[Ana95b] ANALOG DEVICES: *ADSP-21000 Family C Tools Manual*, 1995.

[Ana95c] ANALOG DEVICES: *ADSP-2106x SHARC User's Manual*, 1995.

[Ana96] ANALOG DEVICES: *ADSP-2106x SHARC DSP Microcomputer Family*, 1996.

[ASU88] AHO, A. V., R. SETHI und J. D. ULLMAN: *Compilerbau, Teil 2.* Addison Wesley, 1988.

[Bas95] BASHFORD, S.: *Code Generation Techniques for Irregular Architectures.* Technischer Bericht596, Universität Dortmund, November 1995.

[Ben94] BENITEZ, M. E.: *Register Allocation and Phase Interactions in Retargetable Optimizing Compilers.* Doktorarbeit, University of Virginia, Mai 1994.

[Bra91] BRADLEE, D. G.: *Retargetable Instruction Scheduling for Pipelined Processors.* Doktorarbeit, Technischer Bericht 91-08-07, University of Washington, 1991.

[Brü95] BRÜGGEMANN, W.: *Ausgewählte Probleme der Produktionsplanung.* Physica Verlag, Heidelberg, 1995.

[Cas95] CASTELLI, G.: *The Seemingly Unlimited Market for Microcontroller-Based Embedded Systems.* IEEE Micro, 15(5):6 – 8, Oktober 1995.

[CH90] CHOW, F.C. und J.L. HENNESSY: *The Priority-Based Coloring Approach to Register Allocation.* ACM Transactions on Programming Languages and Systems, 12(4):501 – 536, 1990.

[Chv83] CHVATAL, V.: *Linear Programming.* Freeman and Company, New York, 1983.

[CJP83] CROWDER, H., E. JOHNSON und M. PADBERG: *Solving Large Scale Zero-One Linear Programming Problems.* Operations Research, 31(5):803 – 834, 1983.

[CLR90] Cormen, T. H., C. E. Leiserson und R.L. Rivest: *Introduction to Algorithms.* MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1990.

[CPL95] CPLEX Optimization: *Using the CPLEX Callable Library*, 1995.

[CWM94] Chaudhuri, S., R.A. Walker und J.E. Mitchell: *Analyzing and Exploiting the Structure of the Constraints in the ILP-Approach to the Scheduling Problem.* IEEE Transactions on Very Large Scale Integration (VLSI) System, 2(4):456 – 471, Dezember 1994.

[Din92] Dinkelbach, W.: *Operations Research.* Springer, 1992.

[DK96] Dinkelbach, W. und A. Kleine: *Elemente einer betriebswirtschaftlichen Entscheidungslehre.* Springer, 1996.

[Fis81] Fisher, J.A.: *Trace Scheduling: A Technique for Global Microcode Compaction.* IEEE Transactions on Computers, C-30(7):478 – 490, Juli 1981.

[FOW87] Ferrante, J., K. J. Ottenstein und J. D. Warren: *The Program Dependence Graph and its Use in Optimization.* ACM Transactions on Programming Languages and Systems, 9(3):319 – 349, Juli 1987.

[Gas89] Gasperoni, F.: *Compilation Techniques for VLIW-Architectures.* Technischer Bericht Courant Institute of Mathematical Science, New York University, März 1989.

[GE92] Gebotys, C. H. und M.I. Elmasry: *Optimal VLSI Architectural Synthesis.* Kluwer Academic, 1992.

[GE93] Gebotys, C. H. und M.I. Elmasry: *Global Optimization Approach for Architectural Synthesis.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, CAD-12(9):1266 – 1278, September 1993.

[GJ79] Garey, M. und D. S. Johnson: *Computers and Intractability. A Guide to the Theory of NP-Completeness.* Freemann and Company, 1979.

[GO95] Goodby, L. und A. Orailoglu: *Towards 100 % Testable FIR Digital Filters.* In: *International Test Conference*, Seiten 394 – 402. IEEE Computer Society Press, 1995.

[GP92] Girkar, M. und C. Polychronopoulos: *Optimization of Data/Control Conditions in Task Graphs.* In: Banerjee, Uptal, David Gelernter, Alex Nicolau und David Padua (Herausgeber): *Proceedings of Languages and Compilers for Parallel Computing*, LNCS 589, Seiten 152 – 168, Berlin, 1992. Springer.

[Her80] Herman, G.T.: *Image Reconstruction from Projections. The Fundamentals of Computerized Tomography.* Academic Press, New York, 1980.

[HP94] Hennessy, J.L und D.A. Patterson: *Rechnerarchitektur.* Vieweg, Braunschweig; Wiesbaden, 1994.

[Lan97] Langenbach, M.: *Instruktionsanordnung unter Verwendung graphbasierter Algorithmen für den digitalen Signalprozessor ADSP-2106x.* Diplomarbeit, Universität des Saarlandes, 1997. In Vorbereitung.

[Law76] Lawler, E.L.: *Combinatorial Optimization Networks and Matroids.* Holt – Rinehart – Winston, New York, 1976.

[LDSM80] Landskov, D., S. Davidson, B.D. Shriver und P.W. Mallet: *Local Microcode Compaction Techniques.* ACM Computing Surveys, 12(3):261 – 294, 1980.

[Lee88] Lee, E. A.: *Programmable DSP Architectures: Part 1.* IEEE ASSP Magazine, Seiten 4 – 19, Oktober 1988.

[LLKS85] Lawler, E.L., J.K. Lenstra, A.H.G. Rinnooy Kan und D.B. Shmoys: *The Traveling Salesman Problem.* John Wiley and Sons, New York, 1985.

[Lüc80] Lücker, R.: *Grundlagen digitaler Filter. Einführung in die Theorie linearer zeitdiskreter Systeme und Netzwerke.* Nummer7 in *Nachrichtentechnik.* Springer, Berlin; Heidelberg; New York, 1980.

[Meh88] Mehlhorn, K.: *Datenstrukturen und effiziente Algorithmen 1.* Teubner, Stuttgart, 1988.

[NKT89] Nemhauser, G.L., A.H.G. Rinnooy Kan und M.J. Todd (Herausgeber): Band 1 der Reihe *Handbooks in Operations Research and Management Science.* North-Holland, Amsterdam; New York; Oxford, 1989.

[Now90] Nowinsiki, W.L.: *Parallel Implementation of the Convolution Method in Image Reconstruction.* In: *CONPAR 90 – VAPP IV*, Seiten 355 – 364, Berlin; Heidelberg; New York, 1990. Springer.

[NW88] Nemhauser, G.L. und L.A. Wolsey: *Integer and Combinatorial Optimization.* John Wiley and Sons, New York, 1988.

[PF92] Proebsting, T. A. und C. N. Fisher: *Probabilistic Register Allocation.* In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, Seiten 300 – 310, Januar 1992.

[PS82] Papadimitriou, C.H. und K. Steiglitz: *Combinatorial Optimization, Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, 1982.

[Rei95] REISSENWEBER, B.: *Prozeßdatenverarbeitung.* Oldenbourg, München, 1995.

[Sch93] SCHWARZ, H.R.: *Numerische Mathematik.* Teubner, Stuttgart, 1993.

[SCL96] SAGHIR, M.A.R., P. CHOW und C.G. LEE: *Exploiting Dual Data-Memory Banks in Digital Signal Processors.* http://www.eecg.toronto.edu/saghir/papers/asplos7.ps, 1996.

[Sei96] SEIDEL, R.: *Skript zur Vorlesung Optimierung.* http://www.cs.uni-sb.de/users/mdenny/files/optSs96/skript.ps.gz, 1996.

[WM97] WILHELM, R. und D. MAURER: *Übersetzerbau. Theorie, Konstruktion, Generierung; zweite, überarbeitete und erweiterte Auflage.* Springer, Berlin; Heidelberg; New York, 1997.

[Zha96] ZHANG, L.: *SILP. Scheduling and Allocating with Integer Linear Programming.* Doktorarbeit, Technische Fakultät der Universität des Saarlandes, 1996.

[ZSMS95] ZIVOJNOVIC, V., H. SCHRAUT, WILLEMS M. und R. SCHOENEN: *DSPs, GPPs, and Multimedia Applications – An Evaluation Using DSPstone.* In: *Proceedings of the International Conference on Signal Processing Applications and Technology*, Seiten 1779 – 1783. DSP Associates, Oktober 1995.