Instruktionsanordnung unter Verwendung graphbasierter Algorithmen für den digitalen Signalprozessor ADSP-2106x

Diplomarbeit

Lehrstuhl für Übersetzerbau und Programmiersprachen
FB14 Informatik
Universität des Saarlandes

Marc Langenbach
mlangen@cs.uni-sb.de

7. Oktober 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 Oktober 1997

Marc Langenbach

Danksagung

Herrn Prof. Dr. R. Wilhelm danke ich für die Vergabe des interessanten Themas sowie für seine Hilfsbereitschaft und Unterstützung. Danken möchte ich weiterhin H. Rieder für die Bereitstellung der Prozessordokumentation, der Entwicklungsumgebung und vieler Testprogramme sowie für die informativen und anregenden Gespräche. 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 D. Kästner für die zahlreichen anregenden und ergiebigen Diskussionen und die enge Zusammenarbeit während der Entwicklung und Implementierung der gemeinsamen Programmteile.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Während der letzten zehn Jahre haben sich digitale Signalprozessoren (DSP) als die bevorzugten Prozessoren zur Implementation eingebetteter Anwendungen im Konsumgütermassenmarkt etabliert, man spricht hier von *hight volume consumer products.* Mehrere Aspekte ihres Designs tragen dazu bei: Durch ihre geringe Fläche wird der Stromverbrauch reduziert, die Prozessoren sind in der Regel an ihre Anwendung angepaßt und bieten eine hohe Leistung, wodurch sie auch in (zeit-)kritischen Bereichen einsetzbar sind.

Eingebettete Systeme werden zum einen für einfache Steuerungsfunktionen in Haushaltsgeräten, wie Waschmaschinen oder Mikrowellenherden, eingesetzt, aber auch für komplexere Aufgaben, wie Sendung und Empfang von Sprachsignalen in digitalen, portablen Funktelefonen [Cas95]. Der Einsatz auf dem Massenmarkt ergibt einen Zwang zu niedrigen Preisen. Andererseits stellen eingebettete Systeme meist hohe Leistungsanforderungen. Daher sind diese oft so konfiguriert, daß sie nur eine kleine, anwendungsspezifische Menge von Programmen ausführen können.

Die Preise von digitalen Signalprozessoren werden durch kleine Chipflächen (geringere Ausschußrate), Beschränkung auf im Chip integrierten Speicher (*on-chip memory*) und geringe Stromaufnahme (Betriebskosten) niedrig gehalten. Um dennoch den hohen Leistungsanforderungen gerecht zu werden, stellen viele DSP's Besonderheiten wie Hardwareunterstützung von Schleifen, akkumulatorbasierte Datenpfade, Instruktionen, die die parallele Ausführung unabhängiger Operationen spezifizieren oder mehrere Speicherbänke zur Datenspeicherung, auf die parallel zugegriffen werden kann, zur Verfügung [SCL95].

Ohne angepaßte Programme kann aber dieses Leistungspotential nicht ausgenutzt werden. So behalft man sich durch direkte Programmierung in Assembler. Da diese Programmiermethode fehleranfällig ist und größere Programme schwer direkt in Assembler zu implementieren sind, sind Compiler erforderlich, die eine weitverbreitete, standardisierte und portable Hochsprache, wie zum Beispiel C, in den Maschinencode eines Signalprozessors unter Ausnutzung seiner besonderen Fähigkeiten übersetzen. Allerdings sind existierende C-Compiler häufig nicht in der Lage, Code zu generieren, der die Leistungsanforderungen eingebetteter Anwendungen erfüllt [ZSWS95].

Im Laufe der Jahre wurden verschiedene graphbasierte Verfahren zur Instruktionsanordnung entwickelt, die der Codeerzeugung nachgeschaltet werden, um die parallelen Möglichkeiten der Zielmaschine zu nutzen. Neuere Verfahren bedienen sich der linearen Programmierung, wobei Beziehungen zwischen Operationen untereinander und zwischen Operationen und Registern als Flußgraphen modelliert und als lineares Programm einem Solver wie CPLEX übergeben werden.

Im Rahmen dieser Diplomarbeit werden ausgewählte klassische Verfahren zur Instruktionsanordnung untersucht und mit Ergebnissen aus [Käs97], die mit Hilfe der linearen Programmierung erzielt wurden, verglichen. [Käs97] verwendet SILP- und OASIC-Formulierungen, um das Problem der Instruktionsanordnung in ein lineares Programm zu transformieren, welches dann mit der kommerziellen Software CPLEX gelöst wird. Das Ergebnis ist eine hinsichtlich der Anzahl der Instruktionen optimale Anordnung. Da längere Programme in sehr großen Problemformulierungen resultieren, sind Laufzeiten in der Größenordnung von Tagen nicht unüblich. So werden andere Verfahren benötigt, die die gleiche Aufgabe in kürzerer Zeit lösen – allerdings mit Abstrichen bezüglich der Qualität der Lösung. Wie sich nun diese beiden Ansätze, die lineare Programmierung auf der einen und die graphbasierten Verfahren auf der anderen Seite, hinsichtlich der Ausführungszeit und der Güte des erzeugten Codes unterscheiden, ist Gegenstand dieser Arbeit.

Im folgenden Kapitel wird die zugrundeliegende Zielhardware, der ADSP-2106x, vorgestellt. Neben den verschiedenen Berechnungseinheiten werden der Instruktionszyklus und Besonderheiten der Steuerung des Kontrollflusses aufgezeigt. Kapitel 3 gibt eine kurze Einführung in die Probleme, die sich bei der Codeerzeugung stellen. Insbesondere wird der Graphfärbeansatz zur Lösung des Registerallokationsproblem vorgestellt. Im nachfolgenden Kapitel werden verschiedene Kontroll- und Datenabhängigkeitsdarstellungen eingeführt, die grundlegend für die im Kapitel 5 beschriebenen Verfahren zur Instruktionsanordnung sind. Kapitel 6 beschäftigt sich mit der Ausnutzung von aufteilbarem Speicher. Daran fügt sich ein Kapitel über `CIS`, eine Implementation verschiedener Verfahren zur Instruktionsanordnung, die in Zusammenhang mit dieser Arbeit und unter Mitarbeit von Daniel Kästner [Käs97] entstand. Neben Beschreibungen der Implementation werden dort die untersuchten Testprogramme vorgestellt und die erhaltenen Ergebnisse ausgewertet. In Kapitel 8 werden die wichtigsten Ergebnisse überblicksartig zusammengefaßt. Der Anhang enthält Einzelheiten über die Optionen von `CIS`, den Instruktionssatz des ADSP-2106xsowie einige verwendete Algorithmen. Für alle untersuchten Testprogramme wird das Quellprogramm in Assemblercode angegeben und einer optimierten Version gegenübergestellt.

Kapitel 2

Architektur

2.1 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.

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

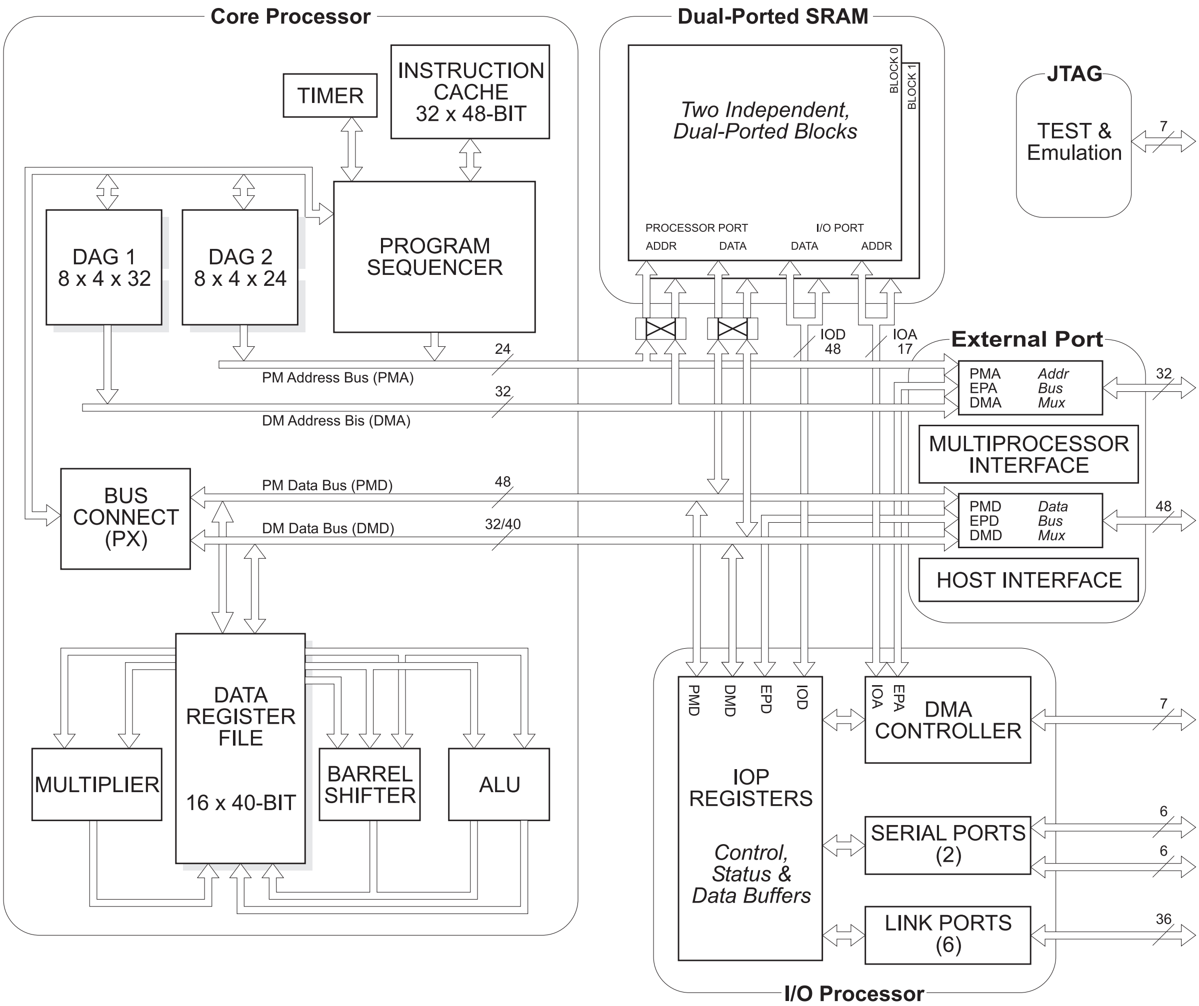


Abbildung 2.1: Blockschaltbild des ADSP-2106x.

externe Speicherbereich in vier Bänke unterteilt.

Die Hostschnittstelle erlaubt einfache Verbindungen zu 16 und 32 Bit breiten Standard-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-bittige IEEE Gleitkomma-Format, eine 32 Bit breite ganzzahlige Zahlendarstellung (2er-Komplement, sowie vorzeichenlose Darstellung) und 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 e 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. Abschnitt 2.9.3).

[1]Vektorinterrupts werden in Multiprozessorsystemen für Anweisungen zwischen verschiedenen Prozessoren verwendet.

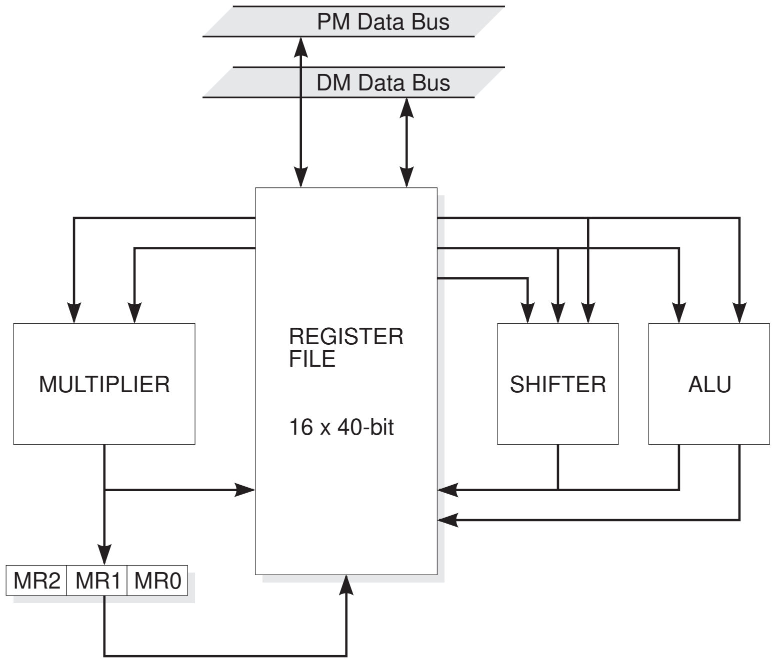


Abbildung 2.2: Die Berechnungseinheiten.

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

2.2 Die Berechnungseinheiten

Es gibt drei unabhängige Berechnungseinheiten: eine ALU, einen Multiplizierer mit Festkomma-Akkumulator und einen Shifter.

Die ALU führt arithmetische und logische Operationen 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.

2.2.1 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 C.2 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 beschreiben. Dies gilt ebenfalls für Multiplizierer und Shifter.

2.2.2 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*) bezeichnet; welches von beiden aktiv ist, wird von einem Bit eines Modusregisters bestimmt. Anders als bei anderen Register, für die alternative Registermengen existieren, kann auf beide MR-Register zur selben Zeit zugegriffen werden. In Anhang C.4 sind die Multipliziererinstruktionen mit den zugehörigen Statusbitveränderungen aufgeführt.

2.2.3 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. Bsp. in `Rz=Rz or lshift Rx by Ry`).

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

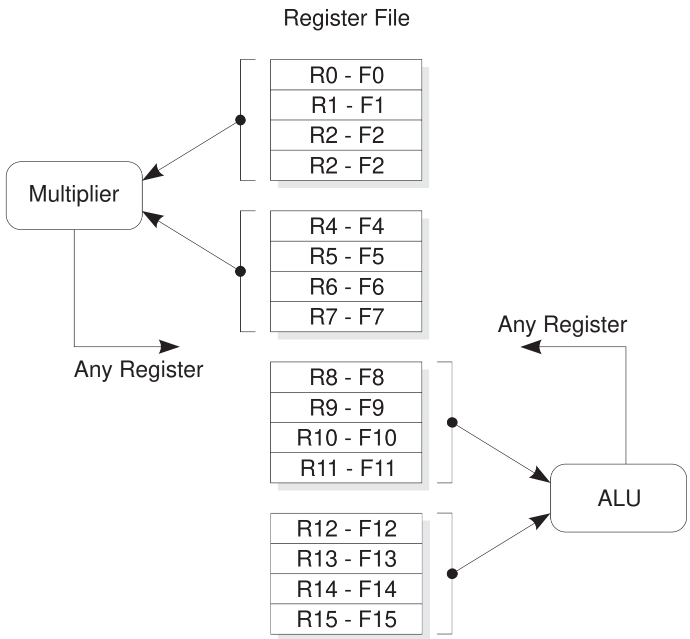


Abbildung 2.3: Eingaberegister bei Multifunktionsinstruktionen.

Der Shifter schreibt ein Ergebnis in das Registerfile zurück. Eine Liste der Instruktionen des Shifters zusammen mit den Aktualisierungen der Statusbits ist in Anhang C.5 zu finden.

2.2.4 Multifunktionsinstruktionen

Multifunktionsinstruktionen kombinieren parallele Ausführung von Multiplizierer und ALU oder parallele Funktionen innerhalb der ALU. Eine Übersicht über alle Multifunktionsinstruktionen ist in Anhang C 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.

2.3 Die Register des ADSP-2106x

Fast alle Register im Kernprozessor 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 Abschnitt 2.5), 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 Alternativmengen 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 Abschnitt 2.2.1).

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 oder Universalregister
- Programmspeicher
- 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.

2.4 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.

2.5 Kontrolleinheit und Adreßgeneratoren

Die Kontrolleinheit (*program sequencer*) liefert Instruktionsadressen an den Programmspeicher, 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. Abschnitt 2.7).

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.

Detailliertere Angaben zur Kontrolleinheit können Abbildung 2.4 entnommen werden. Das

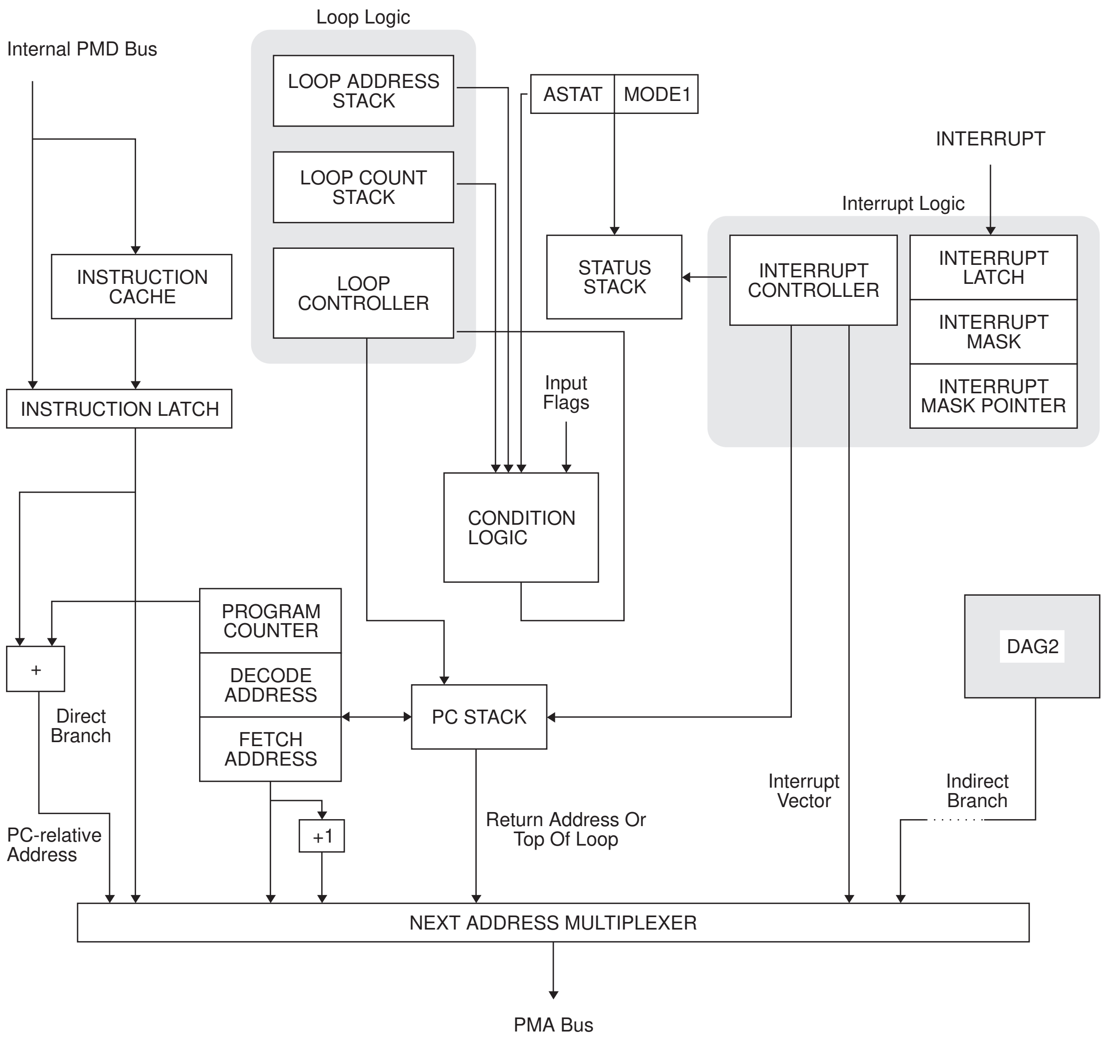


Abbildung 2.4: Die Kontrolleinheit des ADSP-2106x.

fetch address-Register, das *decode address*-Register und der PC (*program counter*) enthalten die Adressen der Instruktionen, die gerade abgerufen, dekodiert 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 Kodierung 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. Eine Auflistung verschiedener Register samt Latenzzeiten ist in Anhang B zu finden.

2.6 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 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.

2.7 Instruktionszyklus

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

1. Im *fetch*-Zyklus wird die Instruktion aus dem Programmspeicher PM oder dem Instruktionscache abgerufen.
2. Im *decode*-Zyklus wird die Instruktion dekodiert.
3. 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, dekodiert 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 von Instruktionen kollidieren
- Sprünge
- Unterprogrammaufrufe und -rücksprünge
- Interrupts und Interruptrücksprünge
- Schleifen

2.8 Befehlsvorrat

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

***compute and move or modify*-Befehle** sind Instruktionen, die eine Berechnung spezifizieren, die parallel zu einem oder zwei Datentransportbefehlen oder der Veränderung eines Indexregisters erfolgen.

***program flow control*-Instruktionen** die bestimmte Typen von Verzweigungen, Unterprogrammaufrufe, Rücksprünge und Schleifen spezifizieren. Einige dieser Instruktionen können auch eine Berechnung und/oder einen Datentransport bezeichnen.

***immediate data move*-Instruktionen** die *immediate*-Felder der Instruktion als Operanden zum Adressieren nehmen.

***miscellaneous*-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 B 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 Mikroinstruktion kombiniert werden.

Beispiel 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;
```

□

[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.

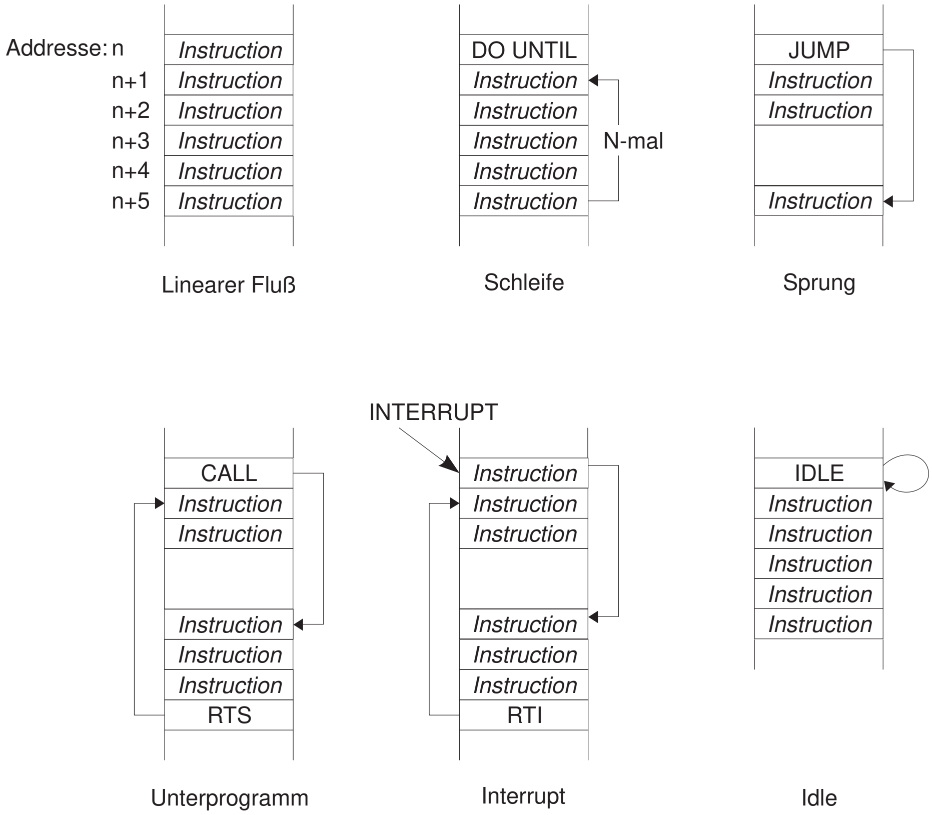


Abbildung 2.5: Mögliche Kontrollflußverläufe.

In der Assemblerdatei ist dies dadurch gekennzeichnet, daß zu einer Mikroinstruktion 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 Maschinenprogramms aufgeführt werden. Instruktionen, die verschiedene Zyklen zugeordnet sind, stehen in verschiedenen Zeilen des Assemblerprogramms. Somit repräsentiert jede Zeile des Maschinenprogramms eine gegebenenfalls aus verschiedenen Mikrooperationen zusammengesetzte Instruktion, die innerhalb eines Taktzyklus ausgeführt wird.

2.9 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.

Die `idle`-Instruktion legt den Prozessor schlafen, bis er durch einen Interrupt wieder aufgeweckt wird.

2.9.1 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 B). 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 (`curlcntr`) testet, sollte eine `not lce`-Bedingung nicht auf ein Beschreiben des Schleifenzählers erfolgen; sonst wird nämlich der alte Wert des Zählers verwendet.

2.9.2 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-Keller 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.

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.

- Sprünge und Prozeduraufrufe können direkt, indirekt oder relativ zum PC erfolgen. Direkte Verzweigungen springen zu der (24-Bit-)Adresse, die im *immediate*-Feld der Verzweigungsinstruktion 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.

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

2.9.3 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`) 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

Aufgabe der Codeerzeugung ist es, ein Quellprogramm, welches meist in einer Hochsprache geschrieben ist, in eine semantisch äquivalentes Zielprogramm zu transformieren. Die Zielsprache ist in der Regel die Assemblersprache einer gegebenen Hardware. Hierbei lassen sich drei Phasen unterscheiden: Das **Frontend** führt eine syntaktische und semantische Analyse des Quellprogramms durch und wandelt es in eine Zwischendarstellung um. Daran können sich im **Middleend** maschinenunabhängige Optimierungen anschließen, wie das Vorberechnen konstanter Ausdrücke. Im **Backend** wird die Zwischendarstellung schließlich in das Zielprogramm umgewandelt. Hier existieren drei Teilprobleme, die sich gegenseitig beeinflussen, nämlich Codeselektion, Registerallokation und Instruktionsanordnung.

Ziel der **Codeselektion** ist die Auswahl möglichst „gute" Befehle für die Ausdrücke und Anweisungen des Quellprogramms. Die meisten Rechner bieten mehrere Befehle als Übersetzung einer Quellsprachenanweisung an. Daraus gilt es, eine bezüglich Ausführungszeit, Speicherplatzbedarf und/oder Befehlsfolgenlänge möglichst gute Befehlsfolge auszuwählen. Bei RISC-Prozessoren ist diese Problem jedoch aufgrund der einfachen Befehlssatzstruktur entschärft [WM97, Bra91].

Die **Registerallokation** hat zum Ziel, die Werte von Variablen und Ausdrücken der Zwischendarstellung (siehe Kapitel 4) auf physikalische Register abzubilden, um die Anzahl der Speicherzugriffe während der Programmausführung zu miminieren. 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 benötigten Werte die Anzahl der Register. Um den Datentransfer von und zum Speicher zu minimieren, muß der **Registerallokator** Entscheidungen darüber treffen, welche Werte auf Register zu allozieren sind. 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**.

Unter **Instruktionsanordnung** schließlich versteht man die Umordnung einer Folge von Instruktionen, um effektiven Gebrauch von dem Leistungspotential der Maschine (z. Bsp. parallele Datenpfade) zu machen. Daruntern fällt auch die Kompaktifizierung parallel ausführbarer Mikroperationen in eine Maschineninstruktion bei VLIW-artigen Architekturen, wie sie mit Abstrichen auch beim ADSP-2106x vorliegt.

Da sowohl Registerallokation als auch Instruktionsanordnung $\mathcal{NP}$-schwere Probleme sind, kann in der Praxis keine optimale Lösung aller Teilaufgaben gegeben werden.

Wie schon erwähnt sind die einzelnen Phasen nicht unabhängig voneinander, sondern beeinflussen sich gegenseitig. Wurde die Codeselektion ohne Berücksichtigung von Anforderungen der Registerallokation durchgeführt, so stehen der Registerzuteilung unter Umständen nicht genügend physikalische Register zur Verfügung, so daß Ergebnisse zwischengespeichert werden müssen. Dies hat aber eine Verlängerung der Ausführungszeit zur Folge, da Speicherzugriffe – im Vergleich zu anderen Operationen – langsam sind. Umgekehrt kann der Codeselektor nach einer vorherigen Registerallokation eventuell eine billigste Befehlssequenz gar nicht auswählen, weil ihm dazu die Register fehlen [WM97].

Eine ähnlicher Beziehung besteht zwischen Registerallokation und Instruktionsanordnung: Der ADSP-2106x kann beispielsweise Addition und Multiplikation parallel ausführen, die Operanden sind allerdings an bestimmte Registermengen gebunden (vergleiche Kapitel 2). Werden die Register nicht mit Hinblick auf die Instruktionsanordnung vergeben, wird Parallelität unmöglich.

Bei RISC-Prozessoren ist aufgrund des einfacheren Befehlssatzes die Codeselektion einfacher zu lösen als bei CISC-Architekturen, so daß die Interaktion zwischen Registerallokation und Codeselektion an Bedeutung verliert. An ihre Stelle tritt die Interaktion zwischen Registerallokation und Instruktionsanordnung, da zur Instruktionsanordnung Register benötigt werden, um die Ausführung unabhängiger Operationen zu überlappen. Die Register werden verwendet, um Parallelität auf Instruktionsebene auszunutzen, im Kontext der Registerallokation, 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. Wird zum Beispiel Registerallokation zuerst ausgeführt, kann sie das Umordnungspotential des Schedulers beschneiden, indem dasselbe physikalische Register unabhängigen Zwischenergebnissen zugewiesen wird, was eine Überlappung der entsprechenden Operationen durch den Scheduler verhindert. Geht die Instruktionsanordnung der Registerallokation voran, kann die Anzahl der gleichzeitig verwendeten logischen Register so stark erhöht werden, daß viele dieser Werte im Hauptspeicher zwischengespeichert werden müssen.

Bei der Verschränkung von Registerallokation und Instruktionsanordnung lassen sich drei Methoden unterscheiden [Bra91]:

1. Die Instruktionsanordnung wird nach der Registerallokation vorgenommen, wobei zwischen den beiden Phasen keine Kommunikation stattfindet (*postpass*).
2. Die Instruktionsanordnung wird vor der Registerallokation ausgeführt. Die Menge der vom Scheduler verwendbaren Register ist lokal beschränkt (IPS).
3. Der Scheduler kommuniziert mit dem Registerallokation, indem er ihr Kostenmaße für die Verwendung von Registern mitteilt.

In der Regel liefert IPS gute Ergebnisse [Bra91]. Der Ansatz, in dem Instruktionsanordnung und Registerallokation eng miteinander verflochten sind, hat sich für Architekturen mit wenigen Registern als sinnvoll erwiesen.

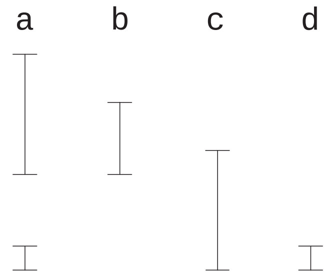
3.1 Registerallokation

In diesem Abschnitt wird genauer auf die Problemantik der Registerallokation eingegangen und ein Verfahren vorgestellt, welches zur Lösung benutzt werden kann. Die Notation ist an [WM97] angelehnt.

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 (d.h. wenn beide für eine folgende Operation benötigt werden) und nicht sicher denselben Wert enthalten. Die nachfolgenden Definitionen basieren auf der Darstellung des Eingabeprogramms in Form eines Kontrollflußgraphen (siehe Abschnitt 4.1).

Definition 1 *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*** *(Lebenszeit) eines symbolischen Registers r ist die Menge der Programmpunkte, an denen r lebendig ist.*

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



(a) Codesequenz (b) Lebenszeiten

Abbildung 3.1: Programmbeispiel und zugehörige Lebenszeiten.

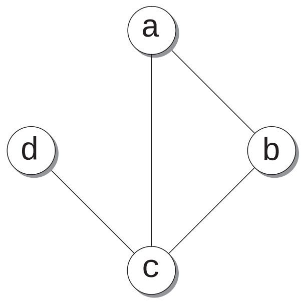


Abbildung 3.2: Der zu den Lebenszeiten aus Abbildung 3.1 resultieren Registerkollisionsgraph.

Definition 2 *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.*

Beispiel 2

Abbildung 3.1 zeigt ein Beispielprogramm und die zu den einzelnen Variablen gehörenden Lebenszeiten. Der daraus resultieren Registerkollisionsgraph ist in Abbildung 3.2 widergegeben.

□

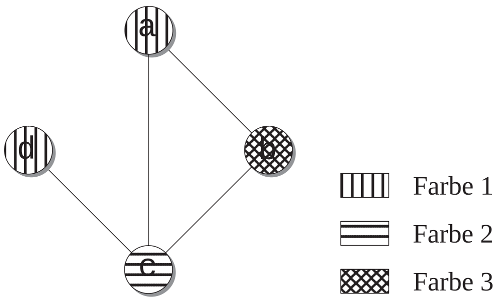


Abbildung 3.3: Eine mögliche 3-Färbung des Registerkollisiongraphen aus Abbildung 3.2

3.1.1 Graphfärbung

Beim Graphfärbeansatz wird das Registerzuteilungsproblem übersetzt in das Problem, den Registerkollisionsgraph mit k Farben zu färben, wobei k die Anzahl der realen Register ist. Die Farben entsprechen also realen Registern. Bei der Graphfärbung dürfen keine direkt verbundenen Knoten die gleiche Farbe zugeteilt bekommen. Für $k > 2$ ist das Problem, für beliebige Graphen zu entscheiden, ob sie k-färbbar sind, $\mathcal{NP}$-vollständig, weshalb eine heuristische Methode angewendet wird, 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 sicherlich 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 enhält. 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. Die hierfür erzeugte Sequenz zum Sichern und Laden des Wertes wird *spill code* genannt.

Beispiel 3

Eine mögliche 3-Färbung des Registerkollisionsgraphen aus Abbildung 3.2 ist in Abbildung 3.3 gegeben. Eine 2-Färbung hingegen existiert nicht. Dazu müßte z. Bsp. die Lebenszeit von `a` aufgespalten werden. Dies ließe sich mit einer Zwischenspeicherung von `a` erreichen (siehe Abbildung 3.4).

□

Sei x ein Knoten, dessen Nachbar die Farben $\{1, \ldots, k\}$ haben, so daß keine Farbe mehr für x übrig ist. Dann gibt es drei Ansätze zur Lösung diese Problems [Bas95]:

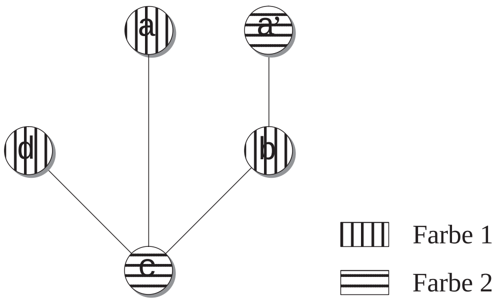


Abbildung 3.4: Durch Aufspalten der Lebenszeit von a läßt sich der Graph mit zwei Farben färben.

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

- Die Einführung von *spill code* speichert eine Variable nach jeder Definition und lädt sie vor jeder Verwendung.

- Bei der letzten Methode wird *spill code* nur an bestimmten Stellen einer Lebenszeit eingefügt und an bestimmten Stellen wieder geladen. Die Lebenszeit wird so in mehrere, disjunktive Lebenszeiten aufgespalten. Jede neue Lebenszeit stellt einen neuen Knoten im Interferenzgraph dar, wobei es gewöhnlich weniger Kollisionen gibt, als mit der ursprünglichen Lebenszeit [Bas95].

Das Einfügen von *spill code* oder das Aufspalten einer Lebensspanne führen zu einem neuen Kollisionsgraphen. Dabei kann das Einfügen von *spill code* für jedes Vorkommen als extremer Fall der Aufspaltung von Lebensspannen angesehen werden. Das Aufspalten von Lebensspannen, d.h. die Bestimmung von aufzuspaltenden Lebensspannen und die Stelle der Aufspaltung, ist jedoch sehr schwierig. Die Bestimmung von optimalen Lösungen ist für beide Probleme $\mathcal{NP}$-schwer.

Kapitel 4

Zwischendarstellungen

Zur Codeerzeugung finden verschiedene Zwischendarstellungen Verwendung. In ihnen werden Syntax und Semantik des Quellprogramms meist in Bäumen oder Graphen wiedergegeben, was der maschinellen Weiterverarbeitung zugute kommt.

Neben Kontrollflußgraphen und Basisblockgraphen, welche den Programmfluß in Graphenform darstellen, werden im diesem Kapitel auch Daten- und Kontrollabhängigkeitsgraphen erläutert sowie der Programmabhängigkeitsgraph. Die Notation ist an [WM97] und [Bas95] angelehnt.

4.1 Kontrollflußgraph

Der Kontrollflußgraph einer Prozedur stellt alle möglichen Programmabläufe in einem Graphen dar. Die Knoten repräsentieren Operationen, die Kanten einen möglichen Kontrollfluß. Der tatsächliche Ablauf kann im allgemeinen nicht dargestellt werden, da unter Umständen Informationen fehlen, die erst zur Laufzeit bekannt sind.

Definition 3 *Der **Kontrollflußgraph** einer Prozedur ist ein knotenmarkierter, kantengeordneter, gerichteter Graph $G_{kf} = (N, E, n_{start})$. Zu jeder primitiven Anweisung p der Prozedur gibt es einen Knoten n_p in der Knotenmenge N, der mit dieser Anweisung markiert ist. Kanten in E zeigen an, welche Anweisungen nacheinander ausgeführt werden könnten. Sie werden mit i (*immediate*, unmittelbar), t (*true*, wahr) oder f (*false*, falsch) markiert. n_{start} markiert den Prozedureintrittspunkt.*

Für zusammengesetzte Anweisungen ergeben sich die entsprechenden Teilgraphen wie in Abbildung 4.1 dargestellt.

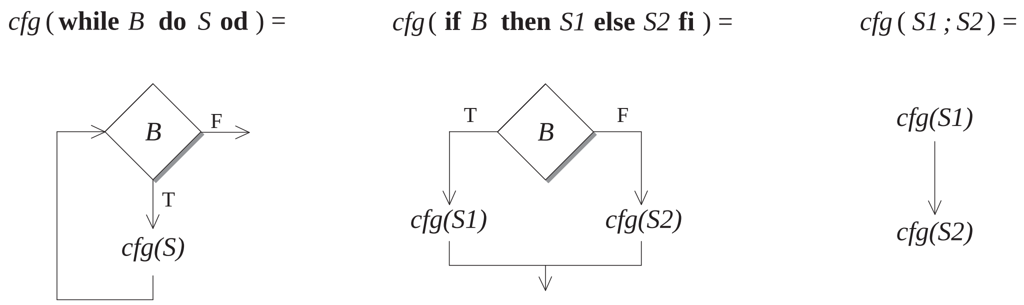


Abbildung 4.1: Kontrollflußgraph für zusammengesetzte Anweisungen.

Knoten, bei denen der Kontrollfluß zusammenläuft (die also mehr als einen Vorgänger aufweisen) werden **Verschmelzungen** genannt, Knoten, an denen er sich aufspaltet (mehrere Nachfolger) **Verzweigungen**.

Im Kontrollflußgraph sollte eine eindeutige Senke n_{ende}, das Programmende, existieren. Sind mehrere Senken vorhanden, werden diese mit dem neu hinzugefügten n_{ende}-Knoten verbunden.

Beispiel 4

Gegeben sei das folgende Codefragment:

```
   r0=dm(i0, m0);
   r1=pm(i8, m8);
   comp(r0, r1);
   if gt jump T:
   r2=r1;
   jump E;
T: r2=r0;
E:
```

Der zugehörige Kontrollflußgraph ist in Abbildung 4.2 dargestellt.

□

Aus dem Kontrollflußgraphen können – wie im folgenden gezeigt wird – weitere Darstellungsformen abgeleitet werden.

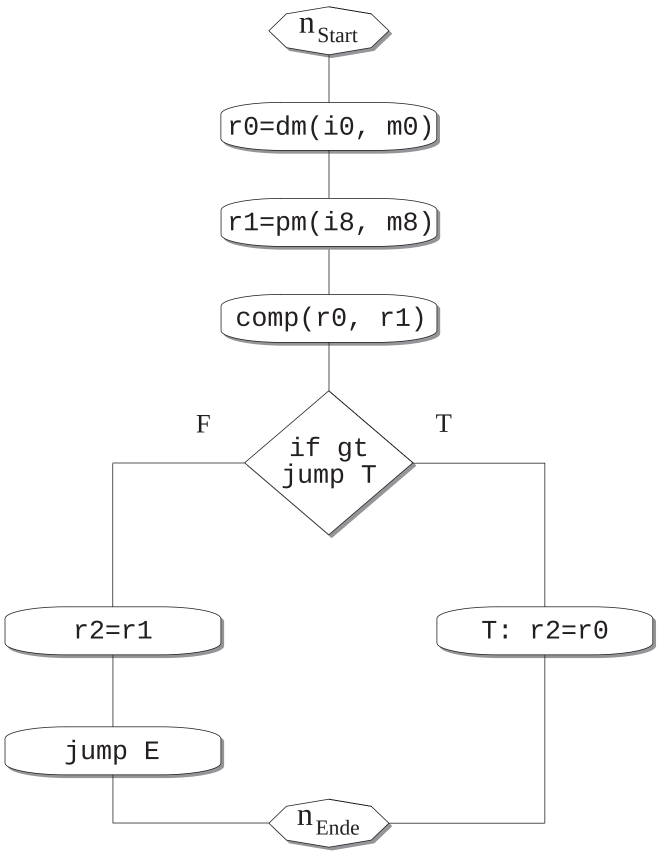


Abbildung 4.2: Kontrollflußgraph für das Programmfragment aus Beispiel 4.

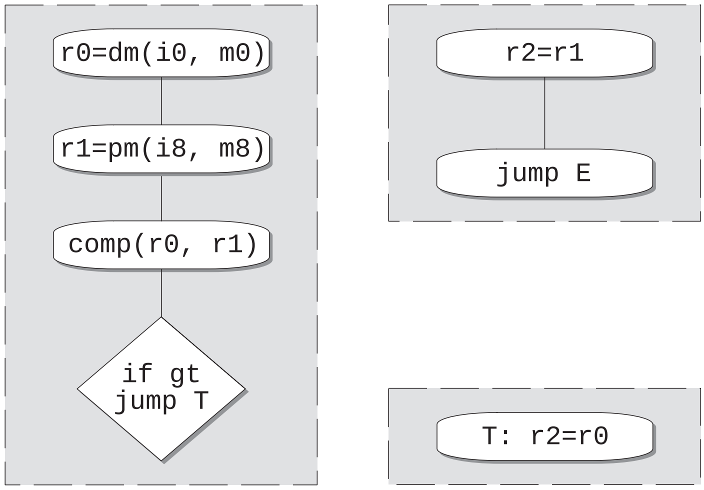


Abbildung 4.3: Die Basisblöcke des Kontrollflußgraphen aus Beispiel 4.

4.2 Basisblockgraph

Betrachtet man den Kontrollflußgraph aus Beispiel 4.2, so erkennt man Pfade, auf denen sich der Graph nicht aufspaltet oder zusammenläuft. Diese Beobachtung wird in der folgenden Definition formalisiert.

Definition 4 *Ein **Basisblock** ist ein Pfad maximaler Länge, der höchstens am Anfang eine Verschmelzung und höchstens am Ende eine Verzweigung aufweist.*

Beispiel 5

Führt man das Beispiel 4 fort, erhält man die in Abbildung 4.3 dargestellten Basisblöcke.

□

Die einzelnen Anweisungen eines Basisblocks teilen die Eigenschaft, daß wenn eine von ihnen ausgeführt wird auch alle anderen ausgeführt werden – vorausgesetzt sei hier eine sequentielle imperative Programmiersprache und das Nichtauftreten von Laufzeitfehlern und ähnlichem.

Werden im Kontrollflußgraphen die Basisblöcke durch sie repräsentierende Knoten ersetzt, erhält man den Basisblockgraph.

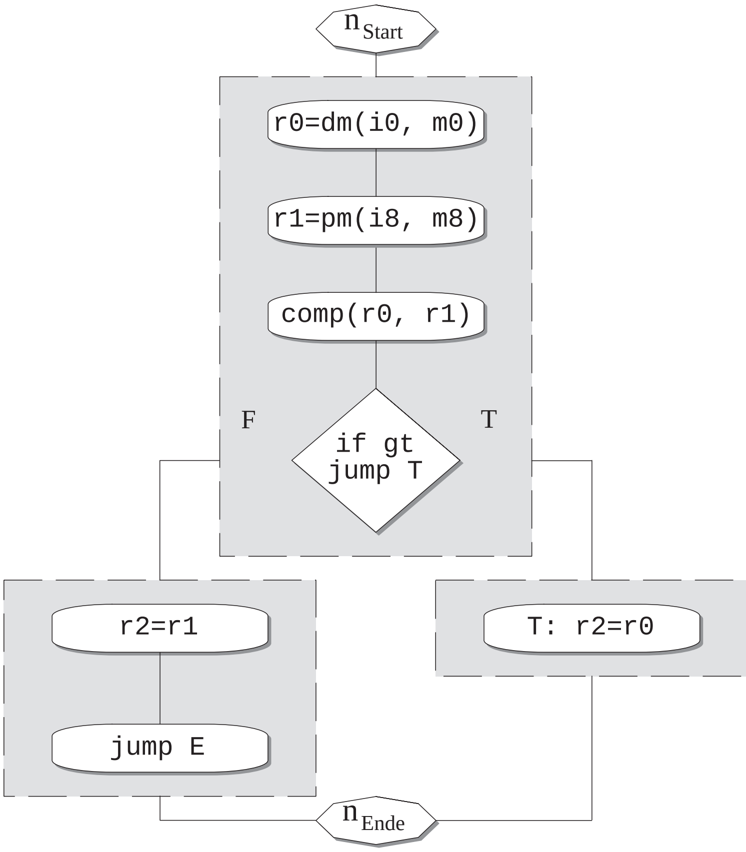


Abbildung 4.4: Der aus dem Kontrollflußgraphen entstandene Basisblockgraph.

Definition 5 *Der* ***Basisblockgraph*** G_{bb} *eines Kontrollflußgraphen* G_{kf} *entsteht aus* G_{kf}*, indem Basisblöcke zu einem Knoten zusammengefaßt werden. Kanten in* G_{kf}*, die zum ersten Knoten eines Basisblocks führen, führen in* G_{bb} *in den Knoten eines Basisblocks hinein; Kanten, die in* G_{kf} *aus dem letzten Knoten eines Basisblocks herausführen, führen in* G_{bb} *aus dem Knoten des Basisblocks heraus.*

Beispiel 6

Ersetzt man im Kontrollflußgraphen aus Beispiel 4 die Basisblöcke durch sie darstellende Knoten, erhält man den zugehörigen Basisblockgraph in Abbildung 4.4. Die Basisblöcke sind grau hervorgehoben.

□

Kontrollflußgraph und Basisblockgraph stellen mögliche Programmverläufe in Graphenform dar. Sie beinhalten in dieser Form keine Informationen über Abhängigkeiten der Instruktionen untereinander. Hierzu dienen die im Weiteren vorgestellten Datenstrukturen.

4.3 Datenabhängigkeitsgraph

Im Verlauf der Instruktionsanordnung (vergleiche Kapitel 5) müssen Informationen über Abhängigkeiten, die zwischen einzelnen Instruktionen existieren, bekannt sein. Sie werden unter anderem dazu benötigt, die sequentielle Ordnung der Anweisung im Basisblock aufzubrechen und durch eine Neuordnung – wobei keine der Abhängigkeiten verletzt wird – eine effizientere Instruktionsanordnung zu erhalten.

Abhängigkeiten treten zwischen Setzungen und Benutzungen des Zustands der Maschine. Hierbei ist eine **Setzung** (*definition*) ein schreibender Zugriff auf bestimmte Komponenten des Maschinenzustandes, wie etwa Register, Statusregister oder Speicherzellen, eine **Benutzung** (*use*) ist ein lesender Zugriff, der den Maschinenzustandes im allgemeinen nicht verändert.

Beispiel 7

Tabelle 4.1 gibt verschiedene Beispiel von Setzungen und Benutzungen in einem Maschinenprogramm wieder und stellt diese gegenüber.

□

Dementsprechend lassen sich drei verschiedene Arten von Abhängigkeiten klassifizieren:

Setzung-Setzung (*output dependence*) Wie in Abbildung 4.5 dargestellt, folgt auf das Setzen einer Maschinenressource (in diesem Beispiel Register `r0`) ein erneutes Setzen. Die Reihenfolge dieser beiden Operationen darf nicht vertauscht werden, da das Register `r0` für folgende Befehle möglicherweise einen anderen Wert enthalten würde.

Setzung-Benutzung (*true dependence*) Das Setzen einer Ressource wird von einer Benutzung gefolgt (Abbildung 4.5). Der lesende Zugriff darf nicht vor den schreibenden verschoben werden, da hier wiederum durch das Setzen der Inhalt des Registers `r0` manipuliert wird.

Setzungen	Benutzungen
Modifikation von Registerinhalten durch Lade-Befehle, durch Resultatablage nach Operationen und durch automatische In- bzw. Dekrementierung von Adreßregistern.	Benutzung von Registerinhalten in Operationen oder zur Adressierung.
Setzen von Überlauf-, Unterlauf- oder Übertragsbits und Vergleichsergebnissen im Bedingungscode.	Abfrage der Teile des Bedingungscodes in bedingten Sprüngen.
Abspeichern von Werten in Speicherzellen.	Laden des Inhaltes von Speicherzellen.
./.	Speichern von Registern.
Modifikation eines Kellerzeigers (*stackpointer*) durch Kellern oder Entkellern von Werten und durch Befehle zur Prozedurorganisation.	./.

Tabelle 4.1: Mögliche Setzungen und Benutzungen in einem Maschinenprogramm.

```
r0=dm(i0, i7);      r0=dm(i0, i7);      r2=r0*r4;
.                   .                   .
.                   .                   .
.                   .                   .
r0=42;              r2=pass r0;         r0=pm(i3, m1);
```

Abbildung 4.5: Abhängigkeiten zwischen zwei Operationen.

Benutzung-Setzung (*anti dependence*) Hier wird der Zustand einer Maschinenressource ausgelesen, bevor er beschrieben wird. Würde im Abbildung 4.5 der schreibende Zugriff auf das Register `r0` vor dem lesenden plaziert werden, resultierte dies in einer veränderten Semantik.

Verwendet man Knoten zur Darstellung der Instruktionen und Kanten für die Abhängigkeiten, erhält man den Datenabhängigkeitsgraph.

Definition 6 *Sei ein Basisblock eines Maschinenprogramms gegeben. Sein* ***Abhängigkeitsgraph*** *ist der folgende markierte, azyklische, gerichtete Graph* G_{da}*. Seine Knoten sind markiert mit den Befehlen des Basisblocks. 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, oder*
- *a eine Ressource benutzt, b sie setzt und der Weg von a nach b setzungsfrei ist, oder*
- *a und b die gleiche Ressource setzen und der Weg von a nach b setzungs- und benutzungsfrei ist.*

Beispiel 8

Ausgehend von folgendem Basisblock

```
r0=dm(i0, m0);
r4=dm(i0, m0);
r8=pm(i8, m8);
r12=pm(i8, m8);
r1=r0*r4;
r0=r1*r4;
r5=r8+r12;
dm(i0, m0)=r12;
```

läßt sich der korrespondierende Datenabhängigkeitsgraph in Abbildung 4.6 konstruieren. Die Kanten sind markiert mit „t“ für Setzung-Benutzung-, „a“ für Benutzung-Setzung- und „o“ für Setzung-Setzung-Abhängigkeiten.

Zu beachten sind hier Abhängigkeiten, die z. Bsp. zwischen den beiden Speicheroperationen `r0=dm(i0, m0)` und `r4=dm(i0, m0)` auftreten: Durch diese Speicherzugriffe wird das Indexregister `i0` inkrementiert, was einer Anweisung der Form `i0=i0+m0` entspräche. Durch diesen Sachverhalt sind die eingezeichneten Abhängigkeiten begründet.

□

Erweitert man obige Definition in dem Sinne, daß auch Abhängigkeiten zwischen verschiedenen Basisblöcken erfaßt werden, spricht man vom **erweiterten Abhängigkeitsgraph**. Dieser wird unter anderem von globalen Instruktionsanordnungsverfahren (siehe Abschnitt 5.3) benötigt, da hier Instruktionen auch über Basisblockgrenzen hinweg verschoben werden können.

Der Speicher nimmt unter den Maschinenressourcen eine gesonderte Stellung ein. Werden Register oder Statusbits durch ihren Index bzw. ihre Bitposition direkt im Befehlswort kodiert, so kann der Speicher indirekt adressiert werden, d.h. die zu verwendende Adresse wird durch Adressregister generiert, deren Inhalt im allgemeinen erst zur Laufzeit bekannt

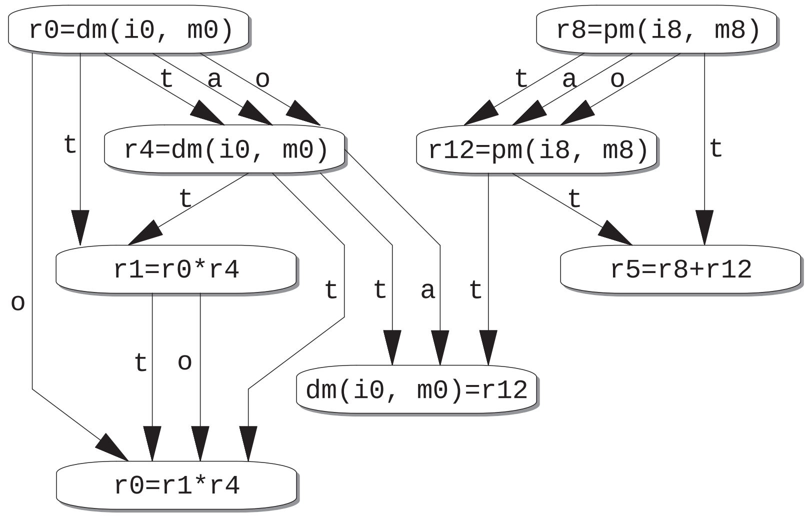


Abbildung 4.6: Der Datenabhängigkeitsgraph für das Beispiel 8.

ist. Daher werden im folgenden Speicherzugriffe nur hinsichtlich des Zugriffs auf Programm- oder Datenspeicher unterschieden.

Eine feinere Differenzierung kann man mittels **alias-Analysen** erhalten. Diese betrachtet den Speicherbereich nicht mehr als eine monolithische Maschinenressource, sondern versucht die Adressen von Speicherzugriffen genauer zu bestimmen. einzuschränken. Dies ist nicht immer exakt möglich.

4.4 Kontrollabhängigkeitsgraph

Bei globalen Analysen erweist sich der Datenabhängigkeitsgraph oft als unzureichend. Als Beispiel sei hier Abbildung 4.7 erwähnt. Werden nur Datenabhängigkeiten berücksichtigt, könnten Instruktionen aus den Basisblöcken A und B in den jeweils anderen verschoben werden (sofern keine Abhängigkeiten dies verbieten). Dadurch würde sich aber die Semantik des Programmfragments ändern, da entweder nur Instruktionen aus A oder aus B ausgeführt werden.

Daher ist neben dem Abhängigkeitsgraphen auch die Kontrollstruktur des Eingabeprogramms zu berücksichtigen. Dies wird durch den **Kontrollabhängigkeitsgraph** gewährleistet, zu dessen Definition aber noch weitere Definitionen benötigt werden.

Definition 7 *Ein Knoten x dominiert einen Knoten y ($x \Delta_d y$), genau dann, wenn jeder Pfad im Kontrollflußgraph G_{kf} vom Eintrittspunkt n_{start} zu y den Knoten x enthält. x heißt* ***Dominator*** *von y. Jeder Knoten dominiert sich selbst.*

if (*Bedingung*)
  A
else
  B
fi

Abbildung 4.7: Betrachtet man lediglich Datenabhängigkeiten, könnte ein Befehl aus Basisblock A nach B (oder umgekehrt) verschoben werden, wodurch sich aber die Semantik des Programms ändern würde.

Dies besagt, falls x den Knoten y dominiert, wird bei Ausführung von y auch immer x ausgeführt. Nach dieser Definition betrachtet man noch den Begriff des unmittelbaren Dominators.

Definition 8 *Die Menge der Dominatoren eines Knotens y bildet eine Kette. x ist **unmittelbarer Dominator** von y, genau dann wenn*

- $x \, \Delta_d \, y$ *und*
- $\nexists z : x \, \Delta_d \, z \ \wedge \ z \, \Delta_d \, y \ \wedge \ z \neq x.$

Definition 9 *Der **Dominatorbaum** eines Kontrollflußgraphen ist ein Baum, der die Knoten des Kontrollflußgraphen enthält. Seine Wurzel ist der Eintrittsknoten n_{start}. Es existiert eine Kante zwischen x und y genau dann, wenn x unmittelbarer Dominator von y ist.*

In Analogie hierzu läßt sich auch die **Post-Dominanz** definieren.

Definition 10 *Ein Knoten x **post-dominiert** einen Knoten y ($x \, \Delta_p \, y$) genau dann, wenn jeder Pfad im Kontrollflußgraphen von y zum eindeutigen Endknoten n_{ende} x enthält. Ein Knoten post-dominiert sich nie selbst.*

Definition 11 *Die Menge der Post-Dominatoren eines Knotens y bildet eine Kette. x ist **unmittelbarer Post-Dominator** von y, genau dann wenn*

- $x \, \Delta_p \, y$ *und*
- $\nexists z : y \, \Delta_p \, z \ \wedge \ z \, \Delta_p \, x \ \wedge \ z \neq y.$

Sollten im Kontrollflußgraphen mehrere Senken existieren, wird ein Knoten n_{ende} eingeführt und Kanten zwischen den Senken und n_{ende} eingefügt.

Definition 12 *Der **Post-Dominatorbaum** eines Kontrollflußgraphen ist ein Baum, der die Knoten des Kontrollflußgraphen enthält. Seine Wurzel ist der Endeknoten n_{ende}. Es existiert eine Kante zwischen x und y genau dann, wenn x unmittelbarer Post-Dominator von y ist.*

Mit Hilfe dieser Definitionen können die Kontrollabhängigkeit und der zugehörige Graph spezifiziert werden.

Definition 13 *Gegeben sei ein Kontrollflußgraph $G_{kf} = (N, E_{kf}, s)$. Ein Knoten $x \in N$ hat eine Kontrollabhängigkeit über $y \in N$ genau dann, wenn gilt:*

- $(x, a) \in E_{kf}$,
- $x \neg \Delta_p\, y$ *und*
- *Es existiert ein Pfad $p = x, a, \ldots, y$, so daß für alle $z \in p$ mit $z \neq x$ und $z \neq y$ gilt: $y \,\Delta_p\, z$.*

Definition 14 *Der **Kontrollabhängigkeitsgraph** $G_{ka} = (N, E_{ka}, s)$ des Kontrollflußgraphen $G_{kf} = (N, E_{kf}, s)$ ist ein gerichteter Graph mit Kantenmarkierung, so daß $(x, y) \in E_{ka} \Leftrightarrow y$ ist kontrollabhängig von x.*

Beispiel 9

Verwendet man wieder das Programmfragment aus Beispiel 4, erhält man den in Abbildung 4.8 dargestellten Kontrollabhängigkeitsgraph.

□

4.5 Programmabhängigkeitsgraph

Im Programmabhängigkeitsgraph [FOW87] werden sowohl Daten- als auch Kontrollabhängigkeiten in einer Datenstruktur dargestellt. In ihm existieren drei verschiedene Knotentypen: Regionsknoten, Attributsknoten und Instruktionsknoten.

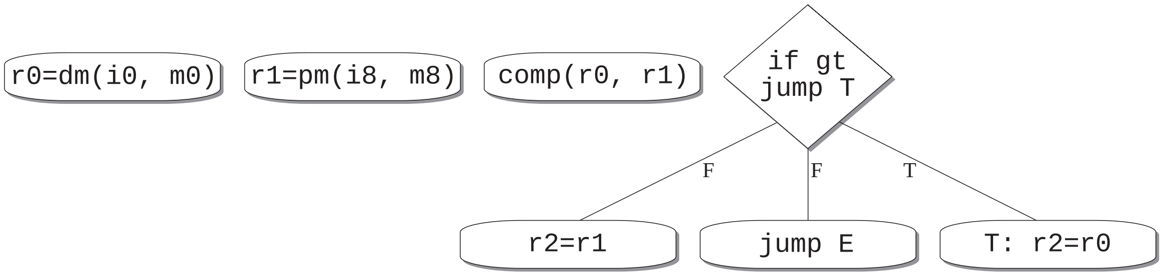


Abbildung 4.8: Der Kontrollabhängigkeitsgraph zu Beispiel 9.

Regionsknoten können alle Knotentypen als Nachfolger aufweisen. Die direkten Nachfolger liegen innerhalb des gleichen Kontrollflusses, d.h. wenn ein direkter Nachfolger ausgeführt wird, werden auch alle anderen ausgeführt.

Attributsknoten repräsentieren Verzweigungsbedingungen. Ausgehende Kanten sind mit *wahr* oder *falsch* markiert.

Anweisungsknoten stellen elementare Maschinenoperationen dar. Sie besitzen keine ausgehenden Kanten.

Der Programmabhängigkeitsgraph läßt sich durch einfache Transformationen aus dem Kontrollabhängigkeitsgraph erzeugen:

- Knoten mit gleicher Tiefe und selbem Vorgänger im Kontrollabhängigkeitsgraph, werden durch Regionsknoten zusammengefaßt.
- Verzweigungsbedingungen werden in Prädikatsknoten umgewandelt.
- Die Datenabhängigkeiten zwischen Anweisungsknoten werden eingetragen.

Beispiel 10

Gegeben sei das folgende Programmfragment:

```
    comp(r0, r1);
    if gt jump L1;
    comp(r0, r2)
    if gt jump L2;
    r2=r0+r2;
L1: r1=r0*r2;
L2: r2=dm(i0, m0);
    r0=r1+r2;
```

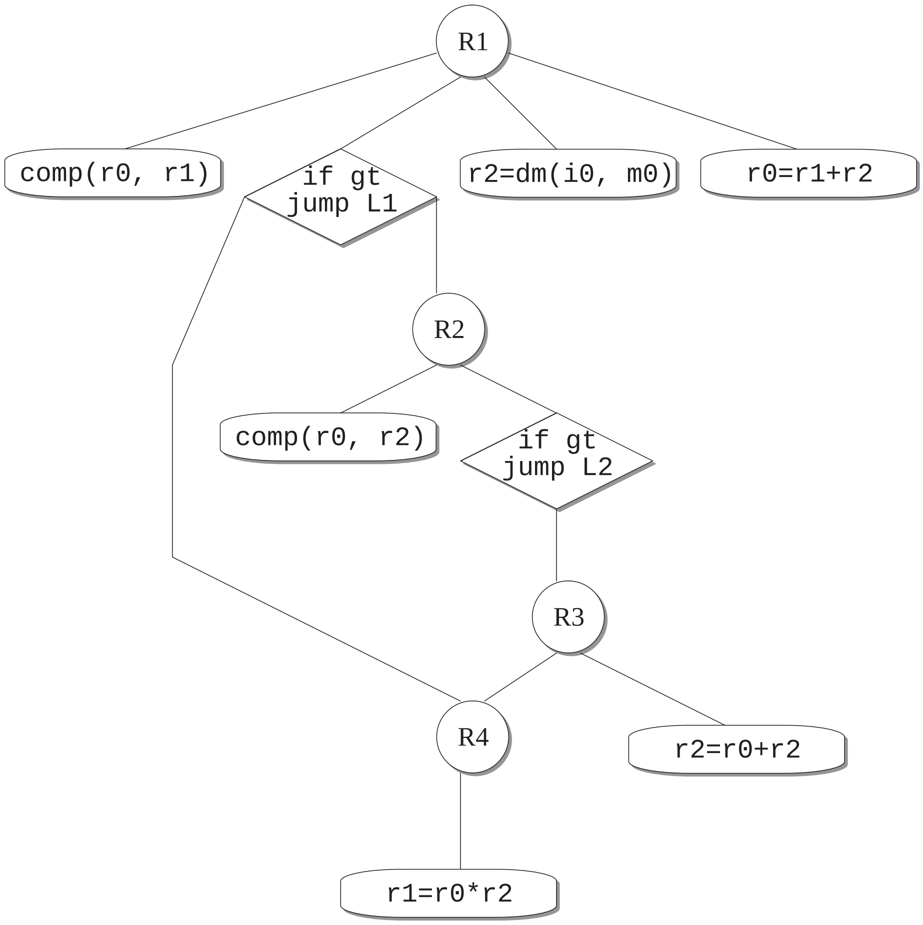


Abbildung 4.9: Der Programmabhängigkeitsgraph zu Beispiel 10.

Der zugehörige Programmabhängigkeitsgraph ist in Abbildung 4.9 dargestellt. Regionsknoten sind durch Kreise, Prädikatsknoten durch Rauten und Anweisungsknoten durch abgerundete Rechtecke dargestellt.

□

Ein weiterer Vorteil des Programmabhängigkeitsgraphen – neben der einheitlichen Datenstruktur für Kontroll- und Datenabhängigkeiten – besteht in der Darstellung von potentieller Parallelität: Direkte Nachfolger eines Regionsknoten liegen im selben Kontrollfluß und können somit (unter Wahrung der Datenabhängigkeiten) in beliebiger Reihenfolge ausgeführt werden. Diese Tatsache wird von *region scheduling* (Abschnitt 5.3.3) ausgenutzt.

Kapitel 5

Klassische Verfahren zur Instruktionsanordnung

Unter dem Begriff „Klassische Verfahren zur Instruktionsanordnung“ werden diejenigen Algorithmen zusammengefaßt, die durch Manipulation der Zwischendarstellung eines Programmes dieses in Hinblick auf die Ausführungszeit verbessern. Sie können grob in lokale und globale Verfahren aufgeteilt werden. Diese beiden Gruppen werden nun genauer gegeneinander abgegrenzt und ausgewählte Vertreter vorgestellt.

5.1 Lokale und globale Verfahren

Die klassischen Verfahren zur Instruktionsanordnung (auch Scheduling-Verfahren) lassen sich wie in Abbildung 5.1 in zwei Kategorien aufteilen: **lokale** und **globale**. Lokale Verfahren Arbeiten auf Basisblöcken – das sind solche Programmfragmente, die keine Sprünge enthalten sondern nur einen sequentiellen Fluß aufweisen (siehe Abschnitt 4.2) –, globale betrachten das gesamte Programm bzw. komplette Pfade (*traces*), die im Verlauf der Abarbeitung durchlaufen werden können.

Sowohl lokale als auch globale Verfahren nutzen üblicherweise Heuristiken[1], d.h. sie liefern oft nur eine suboptimale Lösung. Ihr Vorteil besteht in der kurzen Laufzeit, welche sie zu einer Alternative gegenüber Verfahren, die auf der linearen Programmierung aufsetzen, macht.

Als optimale Lösung wird im folgenden die Anordnung mit der minimalen Anzahl von Instruktionen bezeichnet. Die Optimierung stellt den Prozeß der Annäherung an diese

[1] Eine Ausnahme bildet hier das vollständige *branch and bound*-Verfahren.

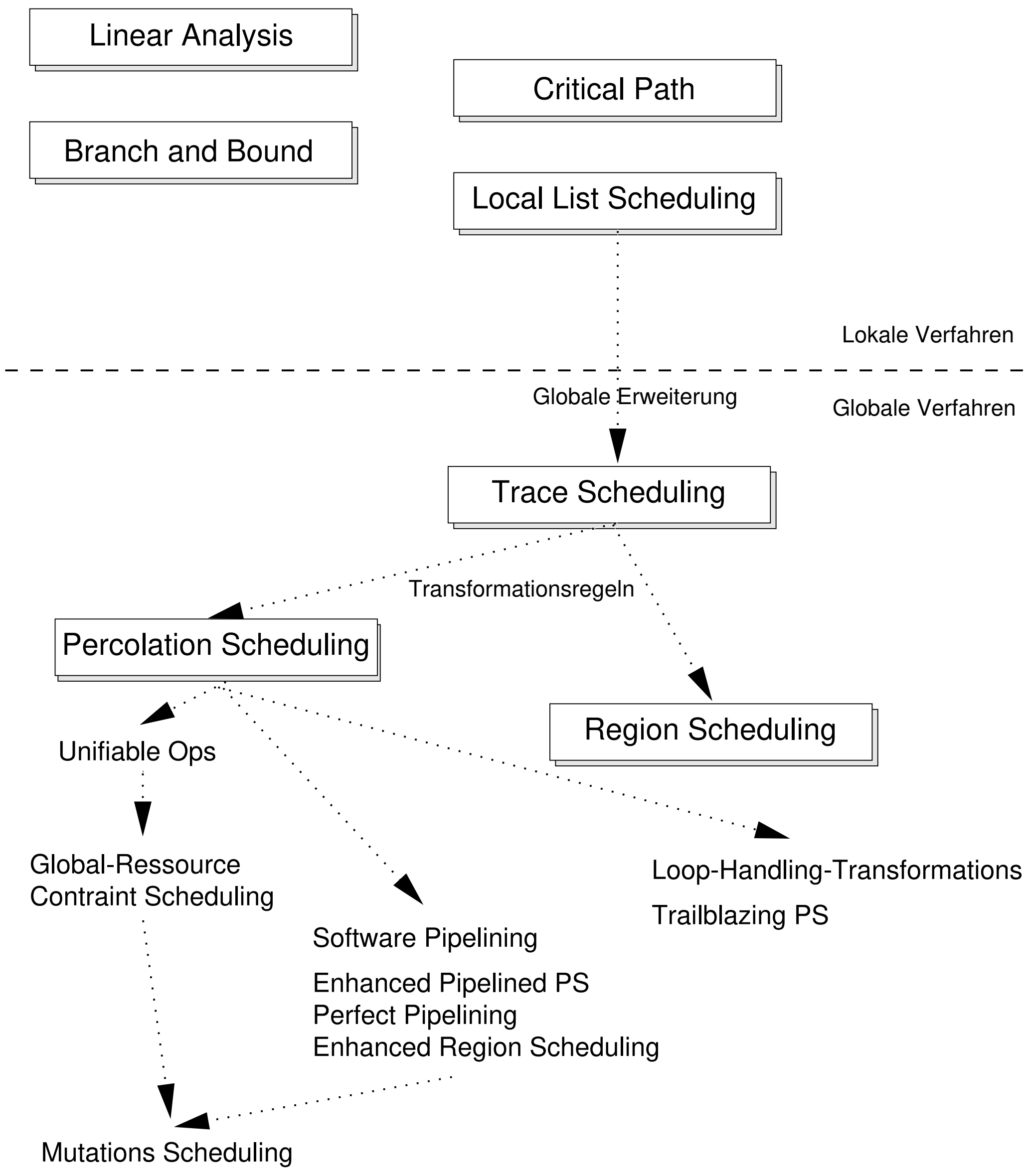


Abbildung 5.1: Klassifizierung klassischer Scheduling-Verfahren.

```
r0=dm(i0, m0);
r4=dm(i0, m0);
r8=pm(i8, m8);
r12=pm(i8, m8);
r1=r0*r4;
r0=r1*r4;
r5=r8+r12;
dm(i0, m0)=r12;
```

Abbildung 5.2: Ein Beispielprogramm.

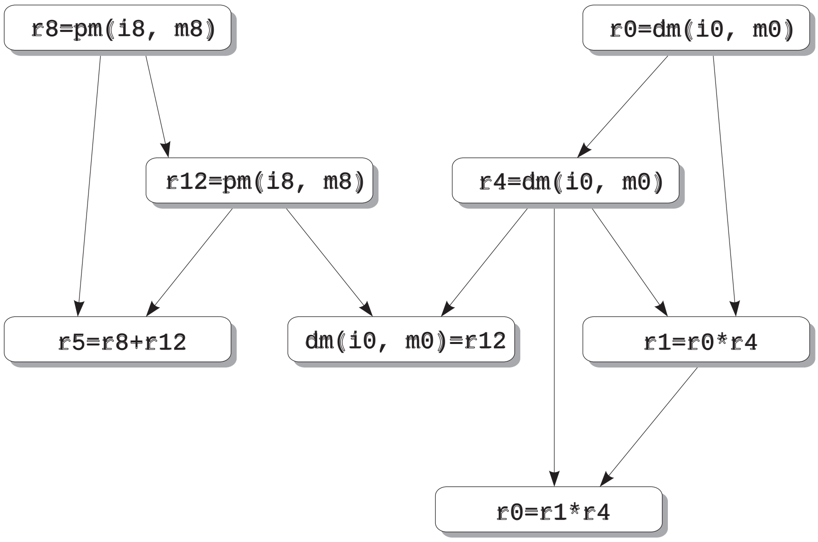


Abbildung 5.3: Datenabhängigkeitsgraph zu dem Programmfragment aus Abbildung 5.2; es sind nur Setzung-Benutzung-Abhängigkeiten dargestellt.

Instruktionsanordnung dar.

Die einzelnen Verfahren werden beispielhaft an dem Programmfragment aus Abbildung 5.2 demonstriert; der zugehörige Datenabhängigkeitsgraph ist in Abbildung 5.3 dargestellt.

5.2 Lokale Verfahren

Die lokalen Verfahren werden auf die Basisblöcke eines gegebenen Quellprogramms angewandt. Somit hängt der Optimierungsgrad u.a. von der Größe der einzelnen Basisblöcke ab: Sind diese klein, ist die Chance, Instruktionen parallel anzuordnen kleiner als bei vergleichbar größeren Blöcken.

Lokale Verfahren können vier Kategorien zugeordnet werden:

1. Lineare Analyse
2. Kritischer Pfad
3. *branch and bound*
4. *list scheduling*

Obgleich *list scheduling* ein Spezialfall des *branch and bound*-Verfahrens darstellt, wird es als eigene Klasse behandelt. Die einzelnen Kategorien werden nun genauer vorgestellt.

5.2.1 Lineare Analyse

Dieses Verfahren zur Instruktionsanordnung wurde erstmals 1976 von Dasgupta und Tartar [DT76] vorgestellt. Es arbeitet auf Basisblöcken, die als Liste dem Algorithmus zugänglich gemacht werden.

Die lineare Analyse startet mit einer leeren Liste, die Instruktionen aufnehmen kann. Im Verlauf des Verfahrens wird für jede Mikrooperation aus der Basisblockliste überprüft, ob diese zu einer Mikroinstruktion der Liste hinzugefügt werden kann, ohne Ressourcen- oder Datenkonflikte auszulösen; hierbei werden Instruktionen am Anfang der Liste bevorzugt. Kann die Mikrooperation nicht in eine bereits vorhandene Mikroinstruktion hinzugefügt werden, wird eine neue an die Liste angefügt.

Beispiel 11

Der Verlauf dieses Verfahrens angewendet auf das Beispiel 5.2 ist in Abbildung 5.4 dargestellt. Die Mikrooperationen sind in der Reihenfolge durchnumeriert, in der sie in der Basisblockliste stehen. Die Iterationen des Algorithmus sind von links nach rechts dargestellt: Operation 1 kann in den ersten Zyklus aufgenommen werden, da noch keine anderen Mikrooperationen vorhanden sind. Operation 2 kann nicht zusammen mit 1 eine Mikroinstruktion in Zyklus 1 bilden, da Datenabhängigkeiten vorliegen. Operation 3 weist keine Abhängigkeiten mit 1 oder 2 auf, ist ressourcenkompatibel mit 1 und kann somit in der ersten Zyklus aufgenommen werden. Operation 4 kann aufgrund von Datenabhängigkeiten mit 3 frühestens im Zyklus 2 verplant werden, was auch möglich ist, da 2 und 4 unterschiedliche Ressourcen verwenden. Die übrigen Operationen werden analog verplant. Der resultierende Code ist in Abbildung 5.5 dargestellt.

□

	1	2	3	4	5
Zyklus	r0=dm(i0, m0)	r4=dm(i0, m0)	r8=pm(i8, m8)	r12=pm(i8, m8)	r1=r0*r4
1	1	1	1,3	1,3	1,3
2		2	2	2,4	2,4
3					5
4					

Iterationen ⟶

	6	7	8
Zyklus	r0=r1*r4	r5=r8+r12	dm(i0, m0)=r12
1	2,3	1,3	1,3
2	2,4	2,4	2,4
3	5	5,7	5,7,8
4	6	6	6

Iterationen ⟶

Abbildung 5.4: Verlauf der linearen Analyse angewandt auf das Codebeispiel aus Abbildung 5.2.

```
r0=dm(i0, m0), r8=pm(i8, m8);
r4=dm(i0, m0), r12=pm(i8, m8);
r1=r0*r4, r5=r8+r12, dm(i0, m0)=r12;
r0=r1*r4;
```

Abbildung 5.5: Ergebnis der linearen Analyse angewandt auf das Codebeispiel aus Abbildung 5.2.

Die Güte des erzeugten Codes ist abhängig von der Reihenfolge, in der die Mikrooperation in die Basisblockliste geschrieben werden. So wird nicht immer optimaler Code (bzgl. des Basisblocks) erzeugt. Werden aber alle möglichen Anordnungen betrachtet, ergibt sich exponentielle Komplexität [LDSM80].

5.2.2 Kritischer Pfad

Kritischer Pfad-Algorithmen als Verfahren zur Intruktionsanordnung wurden bereits 1974 von Ramamoorthy und Tsuchiya eingeführt. Ihre Vorgehensweise ist einer früheren Arbeit über *processor scheduling* sehr ähnlich.

Die minimale Anzahl von Zyklen für einen Basisblock entspricht der Tiefe des zu ihm gehörenden Datenabhängigkeitsgraph. Der **kritische Pfad** bezeichnet den längsten Weg in diesem Graph [LDSM80]. Die Anordnung der im kritischen Pfad enthaltenen Operationen ist implizit durch deren Abhängigkeiten gegeben. Die restlichen Mikroperationen werden dann in diese Anordnung eingefügt, wobei zusätzliche Zyklen entstehen können.

Die Berechnung des kritischen Pfades vollzieht sich in drei Schritten. Für den zu betrachtenden Basisblock wird zunächst eine frühe Partition, danach eine späte Partition erzeugt. Bei der **frühen Partition** werden die Mikrooperationen in der frühstmöglichen Zyklus eingeplant, der aufgrund der gegebenen Datenabhängigkeiten gültig ist. Ressourcenbeschränkungen werden hierbei nicht berücksichtigt. Die Berechnung der **späten Partition** erfolgt analog, es werden lediglich die Kanten im Datenabhängigkeitsgraph umgedreht; hierbei werden Operationen in den spätestmöglichen Zyklus eingeplant. Die Anzahl der Zyklen, die bei frühen bzw. späten Partitionen benötigt wird, ist gleich [LDSM80].

Der kritische Pfad besteht nun aus solchen Mikrooperationen, die sowohl in der frühen als auch in der späte Partition in den gleichen Zyklus geplant wurden. Sie ergeben die **kritische Partition** und stellen ein Gerüst für die Anordnung der übrigen Operationen dar. Um Ressourcenkonflikte auszuschließen muß die kritische Partition noch auf Verletzungen von Ressourcebeschränkungen überprüft werden. Hierbei werden eventuell neue Zyklen eingefügt.

Die verbliebenen Mikrooperationen werden nun in die kritische Partition eingefügt, indem die Zyklen auf Verträglichkeit hinsichtlich Datenabhängigkeiten und Ressourcenbeschränkungen getestet werden, die zwischen den Zyklen der frühen und späten Partition liegen (inklusive), in denen die betrachtete Operation verplant wurde. Auch hier sind unter Umständen neue Zyklen einzufügen.

Beispiel 12

Wieder das Beispiel aus Abbildung 5.2 betrachtend, erhält man die frühen bzw. späten Partitionen aus Tabelle 5.1. Die kritischen Mikrooperationen ergeben die kritische Partition, die – nach Auflösen der Ressourcenkonflikte in Tabelle 5.2 widergegeben ist. Die verbliebenen nicht-kritischen Operation werden nun gemäß ihrer Partitionierungen eingefügt werden. Hierbei muß aber darauf geachtet werden, daß durch ihr Einfügen kein Konflikt hinsichtlich der Maschinenressourcen auslöst wird. Das Ergebnis das *critical path*-Verfahrens ist in Tabelle 5.3) zu finden.

□

Wie die lineare Analyse kann auch mittels der Kritischer Pfad-Methode nicht in allen Fällen eine optimale (in Hinblick auf die Anzahl der Instruktionen) Instruktionsanordnung generiert werden. Dies ist unter anderem darauf zurückzuführen, daß aufeinanderfolgende Unterzyklen – das sind Zyklen, die während des Verfahrens hinzugefügt werden – nicht zu

Instruktion	frühe Partition	späte Partition	kritisch?
r0=dm(i0, m0)	1	1	ja
r4=dm(i0, m0)	2	2	ja
r8=pm(i8, m8)	1	2	nein
r12=pm(i8, m8)	2	3	nein
r1=r0*r4	3	3	ja
r0=r1*r4	4	4	ja
r5=r8+r12	3	4	nein
dm(i0, i0)=r12	3	4	nein

Tabelle 5.1: Frühe und späte Partitionen für das Beispiel 5.2.

Zyklus	Mikroinstruktion
1	r0=dm(i0, m0)
2	r4=dm(i0, m0)
3	r1=r0*r4
4	r0=r1*r4

Tabelle 5.2: Kritische Partition für das Beispiel 5.2.

Zyklus	Mikroinstruktion
1	r0=dm(i0,m0), r8=pm(i8,m8)
2	r4=dm(i0,m0), r12=pm(i8,m8)
3	r1=r0*r4, r5=r8+r12, dm(i0,m0)=r12
4	r0=r1*r4

Tabelle 5.3: Instruktionsanordnung nach Hinzufügen der nicht-kritischen Operation.

einer Instruktion zusammengefaßt werden können, unter der Voraussetzung, sie könnten verschmolzen werden. Dies liegt an der strengen Wahrung der Werte der frühen und späten Partition, wodurch der Zyklenbereich fixiert wird, in den eine Operation eingefügt werden darf. Eine Abhilfe bestünde in einer Kombination aus einer vollständigen Aufzählung der Anordnungen der Operationen in der dem Algorithmus zugeführten Liste und dem wiederholten Versuchen, aufeinanderfolgende Unterzyklen zu kombinieren [LDSM80].

5.2.3 Branch and Bound

Branch and bound-Algorithmen bilden eine Klasse von Instruktionsanordnungsverfahren, die auf Baumstrukturen arbeiten. Die Knoten des Baumes bestehen aus Mikroinstruktionen, ein Pfad von der Wurzel zu einem Blatt stellt eine mögliche Anordnung der Instruktionen dar. Stehen verschiedene Nachfolger zur Auswahl, spaltet sich der Baum an dieser Stelle auf und beide Möglichkeiten werden weiterverfolgt. Ein vollständiger Baum repräsentiert alle möglichen Anordnungen, in dem ein kürzester Pfad eine optimale Lösung ist.

Die Verfahren des *branch and bound* lassen sich in **vollständige** (hier wird im Laufe des Algorithmus der komplette Baum erzeugt) und **heuristische** (es werden nur bestimmte Verzweigungen weiterverfolgt) unterteilen. Während vollständige *branch and bound*-Verfahren optimale Lösungen bei exponentieller Laufzeit finden, werden mit heuristischen Verfahren in der Regel nur suboptimale Ergebnisse erzielt, wobei aber durch günstige Wahl der zugrundeliegenden Heuristiken polynomielle Laufzeiten ereicht werden können [LDSM80].

Im folgenden wird nur der vollständige *branch and bound*-Algorithmus betrachtet.

Als Eingabe stehen dem Algortihmus neben den Instruktionen des Basisblockes auch deren Datenabhängigkeiten zur Verfügung. Davon ausgehend wird während des Verfahrens eine Menge *data_ready* unterhalten (vergleiche hierzu Abschnitt 5.2.4). In ihr werden solche Mikrooperationen abgelegt, die keine Vorgänger im Abhängigkeitsgraphen aufweisen oder deren Vorgänger bereits verplant wurden. Aus den Operationen dieser Menge wird nun jeweils eine komplette Mikroinstruktion geformt, wobei nur solche maximaler Größe in Betracht kommen. Existieren mehrere gleicher Größe resultiert daraus eine Verzweigung im *branch and bound*-Baum. Die erzeugte Mikroinstruktion wird dem Baum hinzugefügt und die Menge *data_ready* aktualisiert (entsprechend der entfernten Mikrooperationen). Dieses Verfahren wird solange fortgesetzt, bis *data_ready* keine Elemente mehr enthält. Dieses Verfahren ist in Algortihmus 5.1 widergegeben.

Es lassen sich noch zwei Optimierungen auf dieses Verfahren anwenden. Wird der Baum mittels Tiefensuche erzeugt, können Pfade, die länger sind als bereits abgeschlossene Pfade, vernachlässigt werden, da sich zu einer schlechteren Instruktionsanordnung führen. Der Aufbau des Baumes kann auch abgebrochen werden, falls ein Pfad mit einer Länge ge-

```
STRUCT PAIR = { SET data_ready, LIST instr };

LIST paths OF PAIR;

Erzeuge Eintrag von Typ PAIR, initialisiere dessen Menge data_ready und füge
    diesen paths an

while (∃ unvollständiger Pfad) do

    forall p in paths do

        entferne p aus paths
        erzeuge Menge I möglicher Instruktionen, die aus data_ready gebildet
            werden können.
        forall i in I do
            erzeuge Kopie p' von p
            füge i and p'.instr an
            Neuberechnung der Menge p'.data_ready
            füge p' zu paths hinzu
        od

    od

od
```

Algorithmus 5.1: Algorithmus zum *branch and bound*-Verfahren.

funden wird, der dem längsten Weg im Datenabhängigkeitsgraphen entspricht, da dies die Anordnung mit minimalen Zyklen darstellt.

5.2.4 List Scheduling

List scheduling ist ein lokales Verfahren, welches – wie schon angesprochen – auf Basisblöcken arbeitet. Dieses Verfahren ist aus einem heuristischen *branch and bound*-Verfahren hervorgegangen und dient u.a. auch für globale Verfahren wie *trace scheduling* als Grundlage [Fis81].

Neben einer Liste der Basisblöcke wird der Datenabhängigkeitsgraph benötigt. Während der Ausführung unterhält *list scheduling* eine Menge *data_ready*, in der sich Mikrooperationen befinden, die keinen Vorgänger im Datenabhängigkeitsgraphen besitzen oder deren Vorgänger bereits verplant wurde.

Die Funktionsweise ist nun einfach zu erklären: Beginnend mit dem ersten Zyklus des zu betrachtenden Basisblocks wird solange eine Mikrooperation aus der *data_ready*-Menge ausgewählt, bis diese Menge leer ist oder die ausgewählte Operation mit den bereits verarbeiteten Operationen einen Ressourcenkonflikt auslöst. In diesem Fall wird der aktuelle Zyklus erhöht, die Menge *data_ready* aktualisiert und mit dem Verfahren solange fortgefahren, bis sämtliche Mikrooperationen verplant wurden (vergleiche Abbildung 5.2).

Bei der Auswahl der Mikrooperationen wird eine Heuristik verwendet, die Prioritäten zuteilt. Anweisungen mit höherer Priorität werden bevorzugt. So kann z. Bsp. die Verweildauer in *data_ready* verwendet werden oder die Anzahl der von der Mikropoperation ausgehenden Abhängigkeiten. Dies hat den Vorteil, daß viele neue „Kandidaten“ in *data_ready* aufgenommen werden können. Eine andere Möglichkeit, die Priorität zuzuweisen, wurde in [Fis81] vorgeschlagen und trägt den Namen *highest level first*: Jede Operation erhält die Länge des von ihr ausgehenden längsten Weges im Datenabhängigkeitsgraph als Gewicht. Diese einfache Heuristik hat sich als gut erwiesen und die damit erzielten Ergebnisse weichen nur wenig von der optimalen Lösung ab [Fis81].

Die Komplexität von *list scheduling* ist $\mathcal{O}(n^2)$ [LDSM80].

Beispiel 13

Tabelle 5.4 listet die Mikrooperationen aus Beispiel 5.2 mit den Prioritäten nach der *highest level first*-Heuristik auf. Der Ablauf des Verfahrens ist in Abbildung 5.6 visualisiert.

□

```
SET data_ready;
INT zyklus=0;

Füge Instruktionen ohne Vorgänger im
Datenabhängigkeitsgraph zu data_ready hinzu.

while (data_ready nicht leer) do
    zyklus = zyklus + 1;

    Wähle solange Operationen aus data_ready aus und
    füge sie in den aktuellen Zyklus ein, bis data_ready leer
    oder durch das Hinzufügen ein Ressourcenkonflikt ausgelöst
    wird.

    Füge diejenigen Instruktionen zu data_ready hinzu, deren
    Vorgänger im aktuellen Zyklus verplant wurden.
od
```

Algorithmus 5.2: Pseudocode für *list scheduling.*

Mirkooperation	Priorität
r0=dm(i0, m0)	3
r4=dm(i0, m0)	2
r8=pm(i8, m8)	3
r12=pm(i8, m8)	2
r1=r0*r4	2
r0=r1*r4	1
r5=r8+r12	1
dm(i0, i0)=r12	1

Tabelle 5.4: Die Mikroinstruktionen aus Beispiel 5.2 mit zugehörigen Prioritäten gemäß *highest level first*-Heuristik.

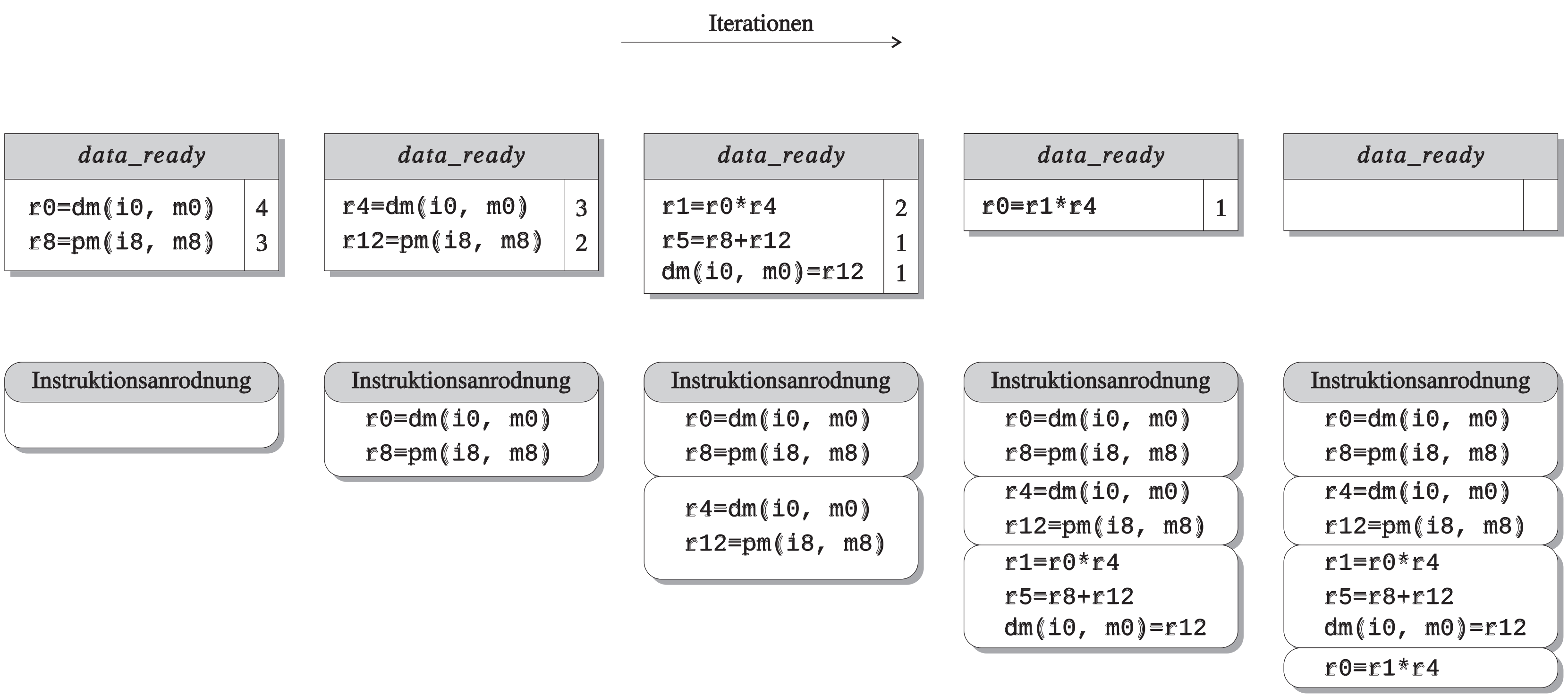


Abbildung 5.6: Veranschaulichung des *list scheduling*-Verfahrens am Beipiel 5.2.

Pfad	enthaltene Basisblöcke
1	B1, B3, B5, B6, B7
2	B4
3	B2

Tabelle 5.5: Pfade im Basisblockgraph aus Abbildung 5.7

5.3 Globale Verfahren

Der große Nachteil der lokalen Verfahren zur Instruktionsanordnung liegt in der ausschließlichen Betrachtung von Basisblöcken. Da diese sehr klein sein können, wird unter Umständen nur eine geringe Parallelität erzielt. Im folgenden werden nun drei globale Scheduling-Verfahren vorgestellt, die auf verschiedene Weise die Basisblockgrenze aufheben.

5.3.1 Trace Scheduling

Bei *trace scheduling* werden an Stelle der Basisblöcke komplette Pfade (*traces*) im Kontrollflußgraphen betrachtet. Ein **Pfad** ist hierbei eine schleifenfreie Anordnung von Instruktionen, die aufgrund gegebener Daten nacheinander ausgeführt werden könnten (Pfade mit Schleifen werden später betrachtet). Diese Pfade lassen sich am besten im Basisblockgraphen verdeutlichen, bei dem die Knoten Basisblöcke und die Kanten einen möglichen Kontrollfluß repräsentieren (siehe Abbildung 5.7).

Die einzelnen Basisblöcke erhalten eine Priorität, die die Ausführhäufigkeit des entsprechenden Blocks widerspiegelt. Diese Prioritäten lassen sich durch Probeläufe des Programms bestimmen oder durch Sprungvorhersagetechniken [BL93]. Aufgrund dieser Prioritäten wird nun der Basisblockgraph in eine geordnete Menge sich nicht überlappender, schleifenfreier Pfade partitioniert, wobei wieder Heuristiken zum Einsatz kommen. Diese konstruierte Menge überdeckt den Basisblockgraphen vollständig. Die Pfade zum Basisblockgraphen aus Abbildung 5.7 sind in Tabelle 5.5 aufgelistet.

Die einzelnen Pfade in der Menge werden nun wie Basisböcke behandelt und *list scheduling* zugeführt. Dabei muß besonders auf Sprünge – sowohl bedingte als auch unbedingte – geachtet werden, was gegebenenfalls zur Einfügung von **Kompensationscode** führt. Ein Beispiel dafür sei Abbildung 5.8: Hier wurde im Laufe des Scheduling-Verfahrens die bedingte Verzweigung vor die Zuweisung geschoben. In einem solchen Fall muß Kompensationscode eingefügt werden, der in diesem Beispiel aus der übersprungenen Zuweisung besteht.

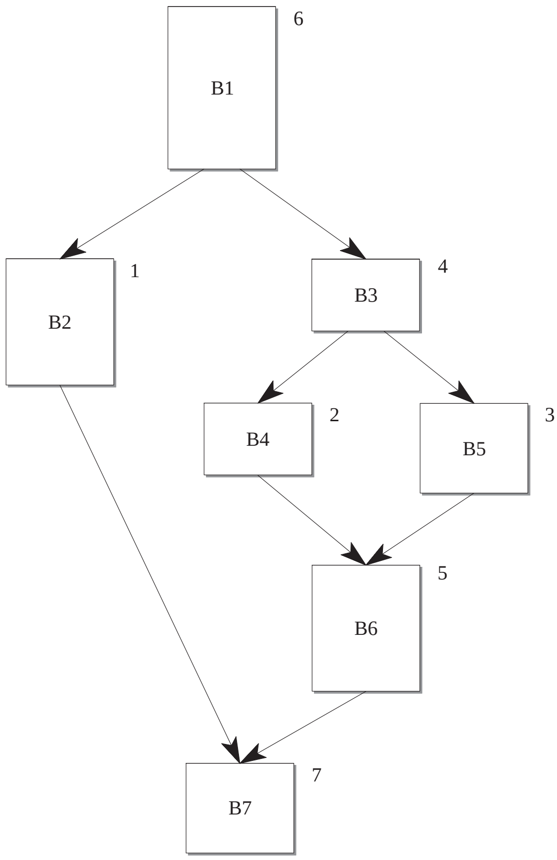


Abbildung 5.7: Der für *trace scheduling* mit Prioritäten markierte Basisblockgraph.

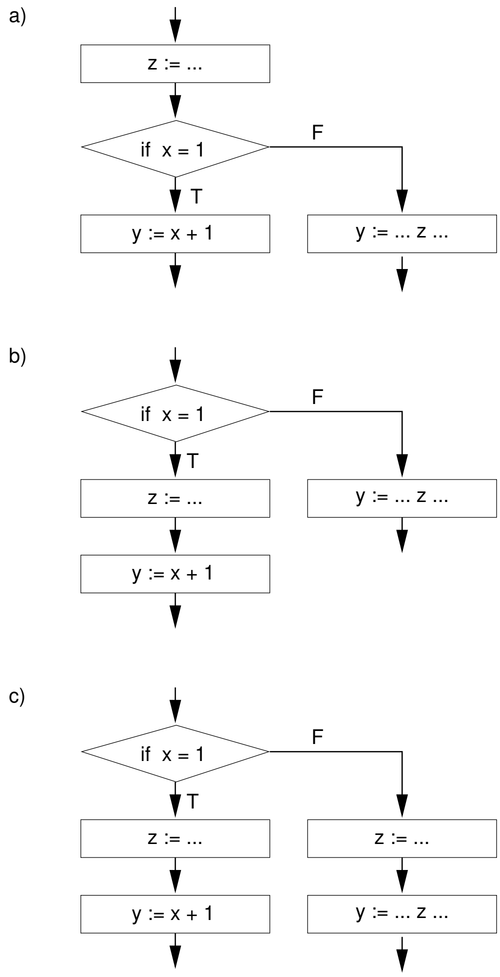


Abbildung 5.8: Einfügung von Kompensationscode: a) ursprünglicher Code, b) Code nach Scheduling ohne Kompensationscode, c) mit Kompensationscode.

Schleife	Blöcke	innere Schleife(n)	Blöcke im mod. BBG
L1	B7, B8	—	B7, B8
L2	B9, B10	—	B9, B10
L3	B6–B12	L1, L2	B6, B11, B12, lr1, lr2
L4	B2, B3	—	B2, B3
L5	B2–B12	L3, L4	B4, B5, lr3, lr4
L6	B1–B12	L5	B1, lr5

Tabelle 5.6: Die verschiedenen Schleifen in Abbildung 5.9

Trace scheduling läßt sich so erweitern, daß Schleifen korrekt behandelt werden können. Betrachten wir hierzu Abbildung 5.9: Es existieren – wenn wir das komplette Programm selbst als Schleife auffassen – 6 Schleifen. Als Voraussetzung wird angenommen, daß sich die Schleifen nicht überlappen. Mit dieser Prämisse lassen sich die Schleifen topologisch sortieren unter der *Ist-innere-Schleife-von*-Relation.

Die einzelnen Schleifen und die zugehörigen Blöcke sind in Tabelle 5.6 aufgelistet. Mit diesen Bezeichnungen ergibt eine topologische Sortierung die Reihenfolge L4-L1-L2-L3-L5-L6. Damit läßt sich nun ein erster, einfacher Algorithmus angeben, der Schleifen korrekt behandelt:

- Sortiere die Schleifen topologisch unter der *Ist-innere-Schleife-von*-Relation. Man erhält eine Folge $L_1, \ldots, L_p$.
- Benutze *trace scheduling* zur Kompaktifizierung der Schleife L_i, wobei die inneren Schleifen L_j mit $j < i$ bereits angeordnet und als *gepackt* markiert wurden.

Durch das Markieren als *gepackt* wird eine Schleife durch einen Schleifenrepräsentanten ersetzt, der dann als eine Anweisung gehandhabt werden kann.

5.3.2 Percolation Scheduling

Percolation scheduling arbeitet auf Programmgraphen, wie z. Bsp. dem Kontrollflußgraphen. Die Motivation hinter diesem Verfahren war, das Problem der Codeexplosion, das bei *trace scheduling* durch Einfügen von Kompensationscode auftrat, zu lösen. Es wird aber unter Umständen immer noch zusätzlicher Code erzeugt.

Percolation scheduling wurde von Nicolau [Nic85] entwickelt. Basisblöcke werden gar nicht mehr betrachtet, vielmehr gibt es vier Transformationen – später wurden diese auf drei reduziert –, die den Graphen lokal verändern: $\tau_{\textit{move-op}}$, $\tau_{\textit{move-cj}}$, $\tau_{\textit{delete}}$ und $\tau_{\textit{unify}}$. Diese

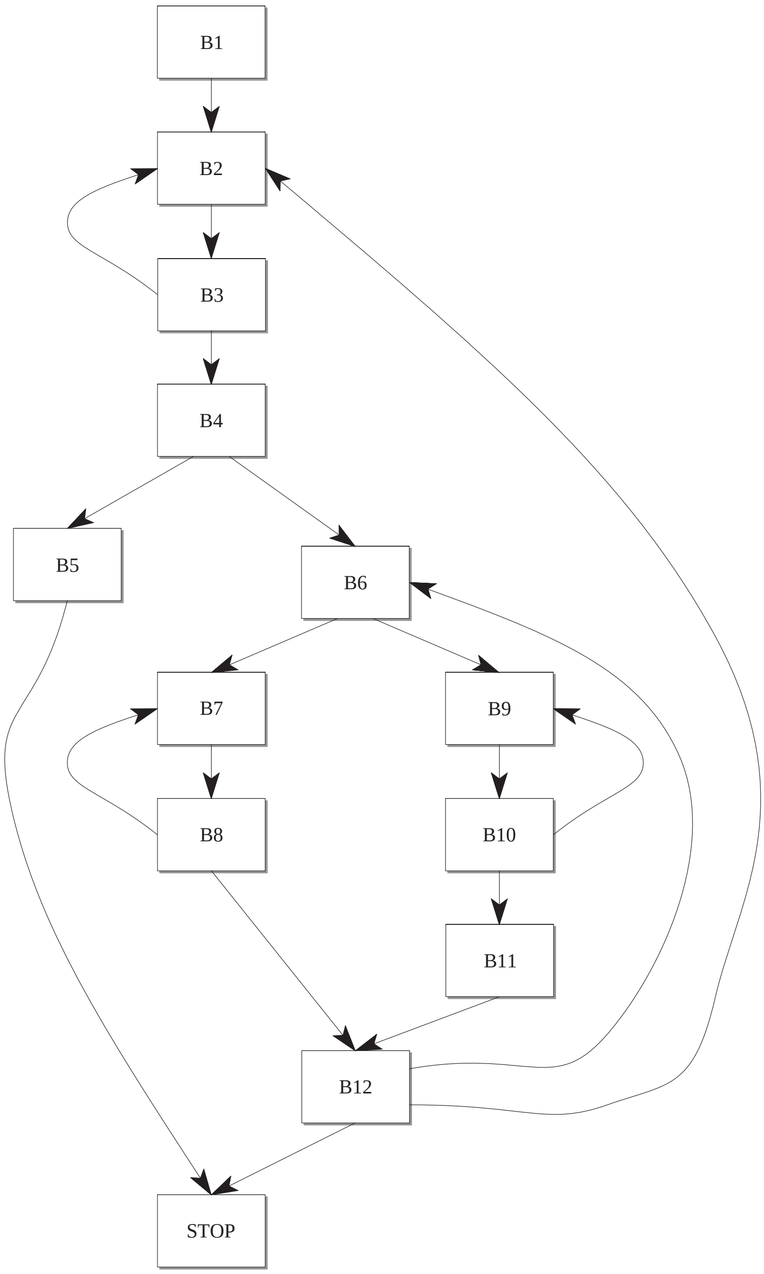


Abbildung 5.9: Programm mit Schleifen.

Transformationen werden – durch eine Heuristik gesteuert – auf den Programmgraphen angewandt. Restriktionen der Ressourcen werden nicht berücksichtigt, für Operationen wird eine Ausführungszeit von einem Zyklus angenommen. Die Parallelisierung wird durch Aufsteigen von Operationen erreicht, wobei die Semantik des ursprünglichen Programms bewahrt wird. Operationen werden durch sukzessives Anwenden der Transformationen in Richtung Programmstart bewegt. Jede Transformation wandelt das Programm in ein semantisch identisches aber paralleleres um.

Es werden verschiedene Schichten unterschieden. Den Kern von *percolation scheduling* stellen die vier Transformation dar, die die Programmdarstellung lokal verändern. Darüber liegen Schichten, die die Anwendung der Transformationen steuern oder Transformationen vornehmen, die globaler Natur sind.

Transformationen

Die im folgenden beschriebenen vier Transformationen bilden den Kern von *percolation scheduling* in der untersten Schicht. Das Wiederholte Anwenden dieser Transformationen läßt Operationen zum Programmstart hin aufsteigen (*percolate*), die Programmsemantik bleibt dadurch gewahrt.

τ_{delete} Diese Transformation dient dazu, Knoten, die keine Operation mehr enthalten, aus dem Graph zu entfernen. Solche Knoten können bereits von Beginn des Verfahrens an enthalten oder durch Applizierung einer anderen Transformation entstanden sein. Durch ihr Entfernen ändert sich die Semantik nicht (siehe Abbildung 5.10).

$\tau_{move\text{-}op}$ Hiermit können einzelne Operationen in Richtung Programmstart verschoben werden, sofern dies Datenabhängigkeiten zulassen. Hierbei muß besonders auf solche Kanten geachtet werden, die in den Knoten laufen, aus dem die Operation entfernt wird. Sie werden auf eine Kopie des ursprünglichen Knotens umgelengt, wobei seine ausgehenden Kanten erhalten bleiben. Abbildung 5.11 zeigt eine solche Verschiebung einer Operation.

$\tau_{move\text{-}cj}$ Analog zur $\tau_{move\text{-}op}$-Transformation werden hierdurch bedingte Sprünge in Vorgängerknoten verschoben. Um die Semantik zu bewahren werden Kopien des Knotens, aus dem der bedingte Sprung entfernt wurde, erzeugt und der ursprüngliche Knoten ausgespalten (siehe hierzu Abbildung 5.12).

τ_{unify} Durch diese Transformation wird eine einzige Kopie einer Operation, die in allen Kindknoten vorhanden ist, in den Elternknoten eingefügt. Die Operation wird in der anderen Knoten gelöscht. Die Transformation kann nur unter Wahrung der Datenabhängigkeiten durchgeführt werden. Gegebenenfalls müssen wieder Knoten mit einer Kopie der verschobenen Operation eingefügt werden. Dies ist der Fall, wenn

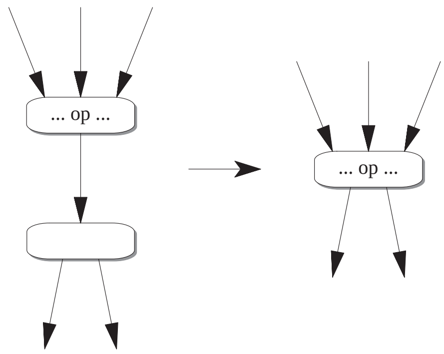


Abbildung 5.10: τ_{delete}-Transformation.

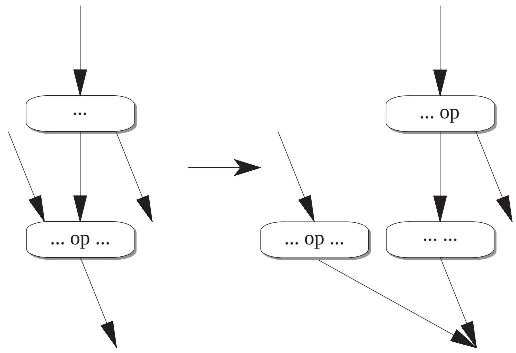


Abbildung 5.11: $\tau_{move\text{-}op}$-Transformation.

die Kindknoten eingehende Kanten besitzen, die nicht vom Elternknoten ausgehen (Abbildung 5.13).

Die Wahl der als nächstes zu verwendenden Transformation wirkt sich aus zwei Gründen auf die Güte des erzeugten Codes aus.

1. Werden zwei verschiedene Transformationen auf einen Ausgangsgraphen G mit Resultat G' bzw. G'' angewendet, kann nicht garantiert werden, daß sich G' und G'' in den gleichen Ergebnisgraphen umwandeln lassen.
2. Die Transformationen sind nicht vollständig; d.h. ein Programmgraph kann nicht immer in einen semantisch äquivalenten mit größtmöglicher Parallelität transformiert werden.

Das Auffinden einer Folge von Transformationen, die zu einer optimalen Lösung führt, hat sich als $\mathcal{NP}$-vollständig herausgestellt. Daher wurden verschiedene Heuristiken entwickelt, die die Wahl der folgenden Transformation treffen, u.a. *compact-block*, *compact-path* oder *compact-global*.

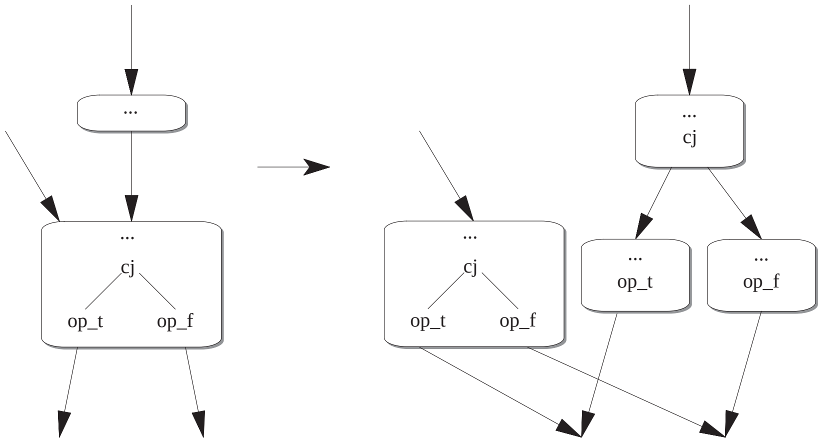


Abbildung 5.12: $\tau_{move\text{-}cj}$-Transformation.

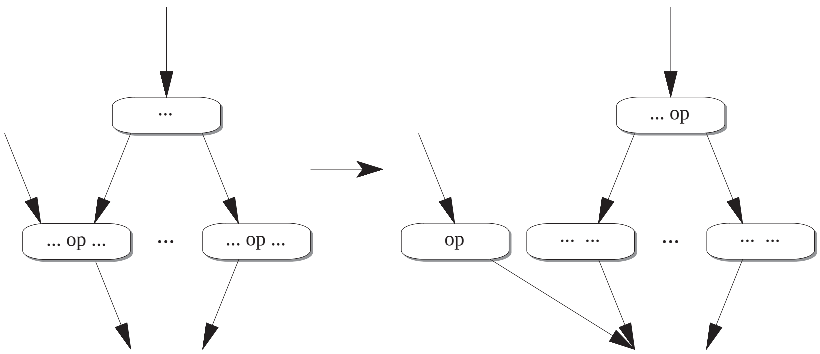


Abbildung 5.13: τ_{unify}-Transformation.

Höhere Schichten

Die oben genannten Transformationen verrichten den Großteil der Codebewegung. Die über ihnen liegenden Schichten von *percolation scheduling* leiten ihre Anwendung und ordnen den Programmgraph so um, daß weitere Veränderungen vorgenommen werden können.

Zwischen folgenden Schichten wird unterschieden:

Basisverfahren Hier werden Datenabhängigkeiten ausfindig gemacht und aufgezeichnet. Generelle Codeoptimierungen wie etwa Entfernung nicht erreichbaren Codes werden vorgenommen.

Leitregeln In dieser Schicht wird eine Menge von Regeln zusammengefaßt, die die Anwendung der verschiedenen Kerntransformationen steuern. Da diese Schicht unabhängig von den primitiven Transformationen ist, kann eine Vielzahl von Heuristiken verwendet werden.

Globale Transformationen nehmen solche Veränderungen am Programmgraph vor, die durch die Kerntransformationen nicht getätigt werden können. Dazu zählen u.a. auch Techniken zum Schleifenaufrollen.

Meta-Schicht dient zum Behandeln von gröberem Parallelismus, wie er etwa auch Schleifen- oder Modulebene zu finden ist.

Wie bereits erwähnt, werden während des Scheduling-Prozesses keine Ressourcen berücksichtigt, so daß das Ergebnis wieder in Hinblick auf vorgegebene Restriktionen partitioniert werden muß. Dies, und die Tatsachte, daß bei der Anwedung der verschiedenen Transformationen der Programmgraph mehrfach traversiert werden muß, zeigen sich in einer langen Ausführungszeit des Verfahrens.

5.3.3 Region Scheduling

Region scheduling verwendet – ähnlich wie *percolation scheduling* – einen erweiterten Programmabhängigkeitsgraphen und führt auf diesem verschiedene Transformationen aus. Das Scheduling wird von Schätzungen der Parallelität der verschiedenen Regionen gesteuert. Die Transformationen werden solange vorgenommen, bis die geschätzte Parallelität mit denen von der Zielmaschine vorgegebenen Möglichkeiten übereinstimmt oder keine Transformation mehr möglich ist.

Vergleicht man *region scheduling* mit *trace scheduling*, sieht man, daß beide Verfahren versuchen, Programme durch Umordnen von Instruktionen zu parallelisieren. *Trace scheduling* orientiert sich bei der Umordnung an der Ausführhäufig – gemessen oder geschätzt

– verschiedener Basisblöcke, das *region scheduling*-Verfahren orientiert sich nur an der Programmstruktur. Nach Gupta [GS90] kann *region scheduling* mindestens soviel Parallelität erzeugen wie *trace scheduling*.

Folgende Erweiterungen werden am Programmabhängigkeitsgraph vorgenommen:

- Knoten, auf die ein Regionsknoten zeigt, sind geordnet. Die Ordnung korrespondiert mit der Ordnung, in der Anweisungen im Quellprogramm vorkommen. So werden bei Verschiebungen (siehe unten) von Knoten aus Region R_i nach R_j zuerst diejenigen Knoten betrachtet, die R_j am nächsten sind.

- Knoten werden als *strukturiert* markiert, wenn nach Entfernung aller Schleifenkanten für alle Kinder folgende Bedingungen erfüllt sind: Der Kindknoten ist *strukturiert* und auf ihn zeigt genau eine Kante. Schleifenkanten sind Kanten, die den Kontrollfluß zwischen zwei aufeinanderfolgenden Schleifeniterationen repräsentiert. Die Strukturiertheit dient dazu, komplette Subgraphen zu verschieben. Hierfür muß gewährleistet sein, daß sich zwei Subgraphen nicht überschneiden. Dies wird durch das *strukturiert*-Attribut sichergestellt.

- Schleifen, die nur aus Anweisungsknoten bestehen werden gekennzeichnet. Sie sind mögliche Kandidaten zum Abrollen (*loop unrolling*), d.h. eine oder mehrere Iterationen der Schleife werden aus der Schleife entfernt und als sequentieller Code vor die Schleife gesetzt.

- Regionsknoten werden mit einem Attribut versehen, das eine Schätzung der in ihnen enthaltenen Parallelität darstellt. Die Schätzung wird für die Entscheidung, welche Region als nächste transformiert werden soll, benötigt. Diese Schätzung einer Region R_i ist definiert als $\delta_i = \frac{O_i}{D_i}$, wobei O_i die Anzahl der Operationen in Region R_i und D_i die maximale Tiefe des Abhängigkeitsgraphen dieser Region sind. Die Schätzung der Anzahl der physikalisch parallel ausführbaren Operationen ist durch δ_h gegeben. Ziel der Instruktionsanordnung ist es, die einzelnen δ_i der Parallelität der Hardware δ_h möglichst gut anzunähern.

Dieser erweiterte Programmabhängigkeitsgraph wird nun solange Transformationen unterworfen, bis sich die Schätzungen der Parallelität der einzelnen Regionen einer vorgegebenen Zahl angenähert haben oder keine Transformationen mehr vorgenommen werden kann. Die einzelnen Transformationen, die im folgenden beschrieben werden, versuchen auf unterschiedliche Weise, die Parallelität[2] in den einzelnen Regionen zu erhöhen, z. Bsp. durch Abrollen von Schleifen oder Verschmelzen von Regionen.

[2] Genauer gesagt die Schätzung der Parallelität, die durch δ_i definiert ist.

Transformationen

Durch die Transformationen werden entweder einzelne Anweisungsknoten oder Teilgraphen manipuliert, die als Wurzel einen Regionsknoten aufweisen. Sie lassen sich aufteilen in

Schleifen-Transformationen Hierzu zählen die Transformationen zum Entrollen einer Schleife und zum Herausziehen von schleifeninvarianten Anweisungen vor die Schleife.

Kopier- und Vereinigungs-Transformationen dienen zum Duplizieren von Codesequenzen zum Vergrößern von Regionen bzw. zum Zusammenführen von Anweisungen, um die Anzahl der Regionen zu Verkleinern.

Verschiebe-Transformationen Durch diese Transformation kann Code von Regionen hoher Parallelität zu denen mit vergleichsweise niedriger bewegt werden.

Die beiden Schleifen-Transformationen dienen zum Entrollen von Schleifen und zum Verschieben von schleifeninvarianten Anweisungen. Die Motivation hinter diesen beiden Modifikationen ist die Erhöhung der Parallelität der betroffenen Regionen, indem die Anzahl der in ihnen enthaltenen Operationen erhöht wird.

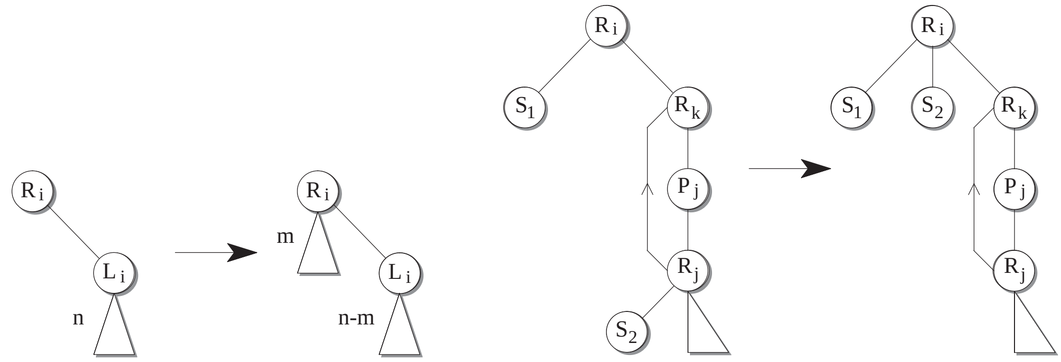
τ_{unroll} Entrollt m Iterationen der Schleife in Region L_i und fügt sie der Region R_i zu, dem unmittelbaren Vorfahr von L_i (siehe Abbildung 5.14 (a)). Dies resultiert in einer erhöhten Parallelität in Region R_i. Mit dieser Transformation läßt sich auch die Parallelität innerhalt von Schleifen steigern.

τ_{invar} Verschiebt eine schleifen-invariante Anweisung S_2 aus Region R_j nach R_i (Abbildung 5.14 (b)). Durch diese Transformation ist möglicherweise eine Revision des erweiterten Programmabhängigkeitsgraphen notwendig (siehe unten).

Transformationen zum Kopieren und Verschmelzen von Regionen stellen weitere Möglichkeiten dar, die Parallelität zu erhöhen. Der Kopiervorgang erhöht die Anzahl der Operationen in einer Region, wo hingegen durch die Verschmelzung die Anzahl der Regionen verringert wird. Beides resultiert in einer Erhöhung der Parallelität.

τ_{copy} Erzeugt eine Kopie aller Knoten im strukturierten Subgraphen, dessen Wurzel der Regionsknoten R_i ist, in allen Elternknoten. Hierdurch werden gemeinsame Knoten eliminiert, wodurch die Subgraphen, die mit einem der Elternknoten beginnen, *strukturiert* werden. Es werden zwei Fälle unterschieden:

1. Falls ein Vorfahr von R_i ein Regionsknoten R_j ist, wird R_j zum Vorfahr aller Kopien von Knoten aus R_i (Abbildung 5.15 (a)).



(a) τ_{unroll} (b) τ_{invar}

Abbildung 5.14: Schleifen-Transformationen

2. Falls ein Vorfahr von R_i ein Prädikatsknoten P_j ist, wird eine Kopie vom Subgraphen mit Wurzel R_i erstellt und P_j zum Vorfahr dieser Kopien (Abbildung 5.15 (b)).

τ^1_{merge} Vereint $P_k - R_j$ mit R_i, wobei P_k der einzige Vorfahr der beiden Regionsknoten ist. Das Ergebnis ist ein Knoten S_k (siehe Abbildung 5.16). Nach dieser Transformation werden P_k, R_i und R_j mittels S_k als eine Anweisung behandelt. Das Prädikat wird zusammen mit den Regionsknoten ausgeführt. Abhängig vom Resultat der Bedingung wird ein Ergebnis der Auswertung der Regionsknoten verworfen, das andere zur weiteren Berechnung übernommen.

Diese Transformation kann nur vorgenommen werden, falls die zugrundeliegende Hardware diese Vorgehensweise unterstützt.

τ^2_{merge} Vereinigt Regionen R_i und R_j zu einer Region (Abbildung 5.17). Hierdurch werden redundante Regionsknoten eliminiert, die aufgrund bereits ausgeführter Transformationen entstanden.

Zum Bewegen von Codefragmenten innerhalb des Programmabhängigkeitsgraphen dienen die Verschiebe-Transformationen. Sie sind auf strukturierte Knoten immer anwendbar, sofern Datenabhängigkeiten gewahrt bleiben.

τ_{move} Verschiebt Code aus Region R_i in eine angrenzende Region R_j. Es existieren zwei Transformationen (τ^1_{move} und τ^2_{move}), die Code in tiefergelegenere Regionen verschieben (vergleiche hierzu Abbildung 5.18). τ^{b1}_{move} und τ^{b2}_{move} bewegen diesen in Richtung der unmittelbaren Vorgänger (Abbildung 5.19). τ^{f1}_{move} und τ^{b1}_{move} werden angewandt, wenn die Regionen R_j und R_i unmittelbar miteinander verbunden sind. Die Transformationen τ^{f2}_{move} und τ^{b2}_{move} kommen dann zum Einsatz, wenn R_j und R_i durch einen

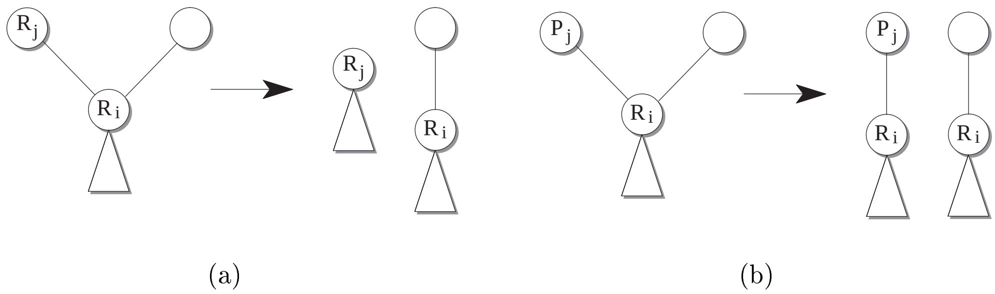


Abbildung 5.15: Transformation τ_{copy}

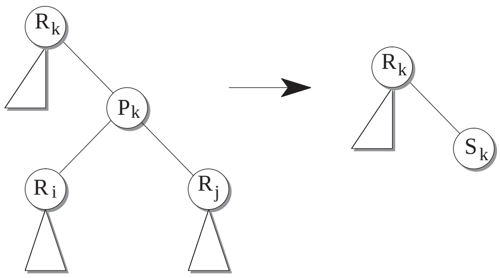


Abbildung 5.16: Transformation τ^{1}_{merge}

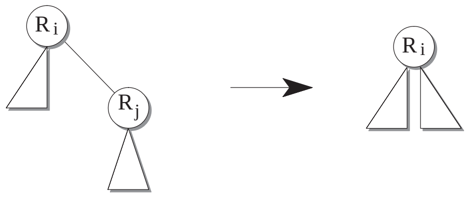


Abbildung 5.17: Transformation τ^{2}_{merge}

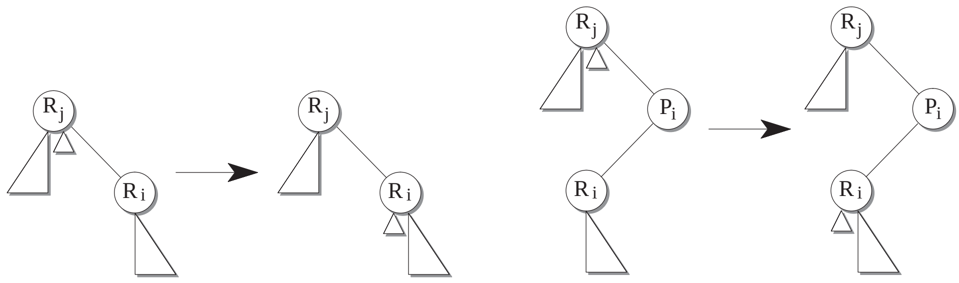


Abbildung 5.18: Transformation τ_{move} in Richtung der Nachfolger

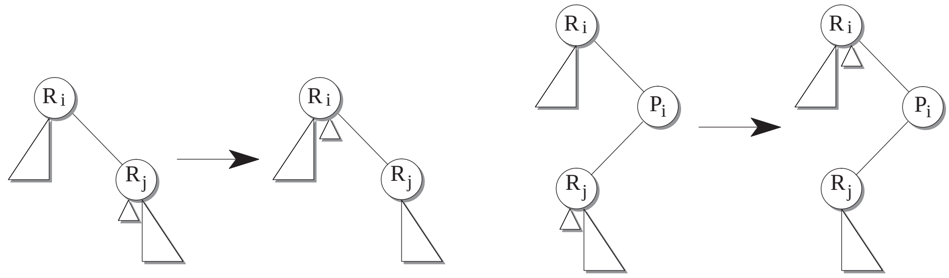


Abbildung 5.19: Transformation τ_{move} in Richtung der Vorgänger

Prädikatsknoten P_i verbunden sind. Diese Transformationen werden verwendet, wenn dadurch eine Steigerung der Parallelität in beiden Regionen erzielt werden kann, oder wenn dadurch „überschüssige“ Parallelität umgeleitet werden kann.

Verschiedene Transformationen erfordern eine **Revision** des erweiterten Programmabhängigkeitsgraph, um die Semantik des dargestellten Programmms zu wahren. Dies wird notwendig, wenn verschobener Code nicht ausgeführt werden würde, obgleich die Bedingungen zu seiner Ausführung erfüllt sind. Wurde Code von Region R_{fr} nach R_{to} mittels der Transformationen τ_{invar} oder τ_{move}^{b1}, müssen zusätzlich Kopien davon für alle Vorgänger von R_{fr} erzeugt werden, wie in Abbilding 5.20 dargestellt. Bei Anwendung der Transformation τ_{move}^{f2} müssen Kopien an die Kinder des R_{fr} nachfolgenden Prädikatsknoten P angefügt werden (Abbildung 5.21). τ_{copy} in Verbindung mit oben genannten Revisionen können zu Redundanzen führen, welche aber durch Anwedung von τ_{merge}^{2} eliminiert werden. Die Revision des erweiterten Programmabhängigkeitsgraphen entspricht dem Kompensationscode, der bei *traces scheduling* Verwendung findet.

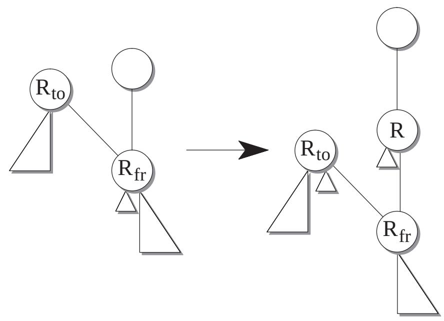


Abbildung 5.20: Erste Revision des erweiterten Programmabhängigkeitsgraphen.

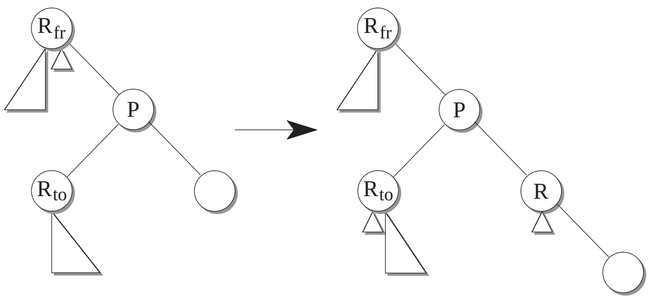


Abbildung 5.21: Zweite Revision des erweiterten Programmabhängigkeitsgraphen.

Scheduling-Algorithmus

Die oben beschriebenen Transformationen können wiederholt angewandt werden, um die Parallelität im erweiterten Programmabhängigkeitsgraph zu erhöhen. Durch die Modifikation der Zwischendarstellung können Transformationen, die zuvor noch nicht möglich waren, verwendet werden, wodurch die Iteration des Schedulers zwingend vorgegeben ist.

Das Anwenden der Transformationen wird solange fortgesetzt, bis keine mehr anwendbar ist oder das Potential der Hardware ausgebeutet ist. In einem ersten Schritt wird zunächst auf lokaler Ebene die Parallelität erhöht, d.h. auf Ebene einzelner Regionen (siehe Abbildung 5.3). Nach der Berechnung der geschätzten Parallelität δ_i werden immer wieder solche Regionen transformiert, die noch „nicht genug“ ($< \delta_h$) Parallelität aufweisen, bis die Ressourcen der Maschine ausgeschöpft werden oder keine Applikation der Transformationen mehr möglich ist. Schleifen werden nur bei Bedarf entrollt, d.h. wenn sich dadurch die Parallelität einer Region erhöhen läßt. Dies unterscheidet *region scheduling* von den üblichen Verfahren, bei denen eine fixe Anzahl von Schleifeniterationen entrollt wird.

In einem zweiten Schritt werden Pfade im Programmabhängigkeitsgraph betrachtet und gegebenenfalls einer Serie von Transformationen unterworfen. Wiederum wird das Verfahren fortgesetzt, bis der Parallelität der Maschine ausgeschöpft wird oder keine der Transformationen mehr angewandt werden kann. Der Algorithmus für diese globale Phase ist in

LocalTransformationPhase()

{

Berechne δ_i *für alle Regionen* R_i

do *Erzeuge Liste* $L = \{R_i | \delta_h - \delta_i \geq \varepsilon\}$

forall $R_i \in L$ **do**

apply $\tau_{unroll}(R_i - L_i^n \longrightarrow R_i - L_i^m - L_j^{n-m})$

else apply $\tau_{invar}(R_j \longrightarrow R_i)$

else apply $\tau_{move}(R_j \longrightarrow R_i)$, *falls* $\tau_{move}(R_i \longrightarrow R_j)$ *nicht zuvor angewandt wurde*

else apply $\tau_{copy}(R_i)$

else apply $\tau^{1}_{merge}(R_i - P_k - R_j \longrightarrow S_k)$, *falls* $\delta_i + \delta_j \leq \delta_h$

Revision des EPAG falls notwendig; dann τ^{2}_{merge} *auf applizierbare Knoten anwenden*

od

until $S = \emptyset$ *oder keine Transformation mehr anwendbar ist*

}

Algorithmus 5.3: *Region scheduler – Lokale Phase.*

GlobalTransformationPhase()

{

Berechne δ_i *für alle Regionen* R_i

do *Erzeuge Liste* $L = \{R_i | \delta_h - \delta_i \geq \varepsilon\}$

forall $R_i \in L$ **do**

if $\exists$ *ein Pfad* $R_i - R_{i+1} - \cdots - R_k$ *mit* $\delta_k > \delta_h$ *oder dies durch Anwendung der Transformationen* $\tau_{unroll}, \tau_{invar}, \tau_{copy}$ *oder* τ^1_{merge} *erreicht*

then apply *die Folge von Transformationen*

$\tau_{move}(R_k \longrightarrow R_{k-1}), \tau_{move}(R_{k-1} \longrightarrow R_{k-2}), \ldots, \tau_{move}(R_{i+1} \longrightarrow R_i)$

Revision des EPAG falls notwendig; dann τ^2_{merge} *auf applizierbare Knoten anwenden*

od

until $S = \emptyset$ *oder keine Transformation mehr anwendbar ist*

}

Algorithmus 5.4: *Region scheduler – Globale Phase.*

Abbildung 5.4 aufgelistet.

Beispiel 14

Der Verlauf des Algorithmus sei hier anhand der Programmausschnitts aus Abbildung 5.22 verdeutlicht. Der benötigte Programmabhängigkeitsgraph ist in Abbildung 5.23 wiedergegeben, der auch die Ausgangsbasis für *region scheduling* darstellt.

Der in Abbildung 5.24 (a) grau hervorgehobene Subgraph stellt eine einfache Schleife dar, die durch einen Schleifenknoten L ersetzt wird, um sie ggf. Schleifentransformationen unterziehen zu können. Unter der Annahme, daß Region R_6 zu wenig Parallelität enthält, wird in einem ersten Schritt (5.24 (b)) die Transformation τ_{copy} auf R_6 angewendet – man erhält damit einen strukturierten Subgraphen. Danach wird durch $\tau^1_{merge}(R_6, P_3, R_7)$ der Anweisungsknoten S_6 erzeugt (in Abbildung 5.24 (c)).

Wieder unter der Annahme, R_4 habe unzureichend Parallelität, werden im folgenden die Transformationen τ_{unroll} und zweimal τ_{move} angewandt. Mit Hilfe dieser Sequenz wird überschüssige Parallelität, die durch das Entrollen der Schleife entsteht, in niedere Regionen verschoben.

Vergleicht man den ursprünglichen Programmabhängigkeitsgraphen mit den, der durch die Transformationen erzeugt wurde, fällt auf, daß zwei Regionsknoten eingespart werden konnten (5 im Vergleich zu 7). Zusätzlich die die neuen Regionen größer als vorher und ent-

L : **for** $i = 1$ **to** N **do** X_i;

if P_1 **then** S_1;

  if P_2 **then** S_3; **goto** $< label >$;
  else
    if $P_3 < label >$: S_4;
    else S_5;
    endif
  endif

else S_2;

endif

Abbildung 5.22: Ein Beispielprogramm zur Veranschaulichung des Scheduling-Verfahrens.

halten somit in der Regel mehr Parallelität. Der daraus erzeugt Code ist in Abbildung 5.25 wiedergegeben.

□

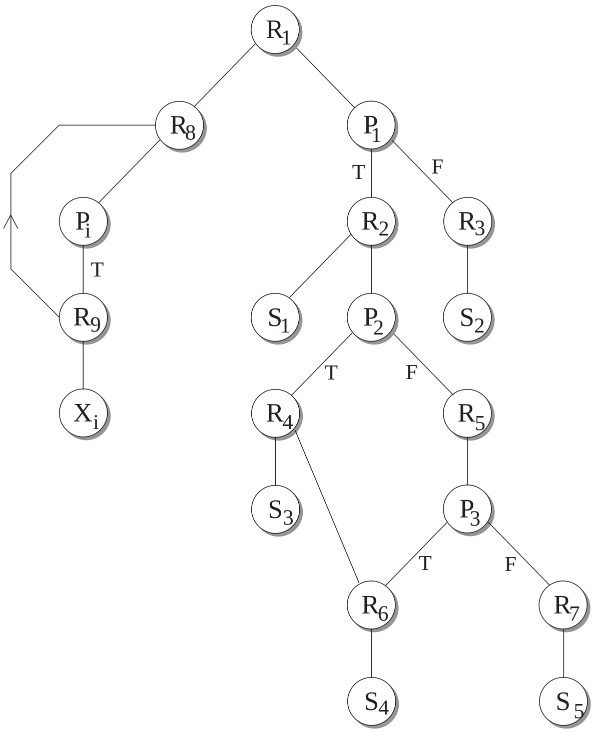


Abbildung 5.23: Der aus Abbildung 5.22 resultierende Programmabhängigkeitsgraph.

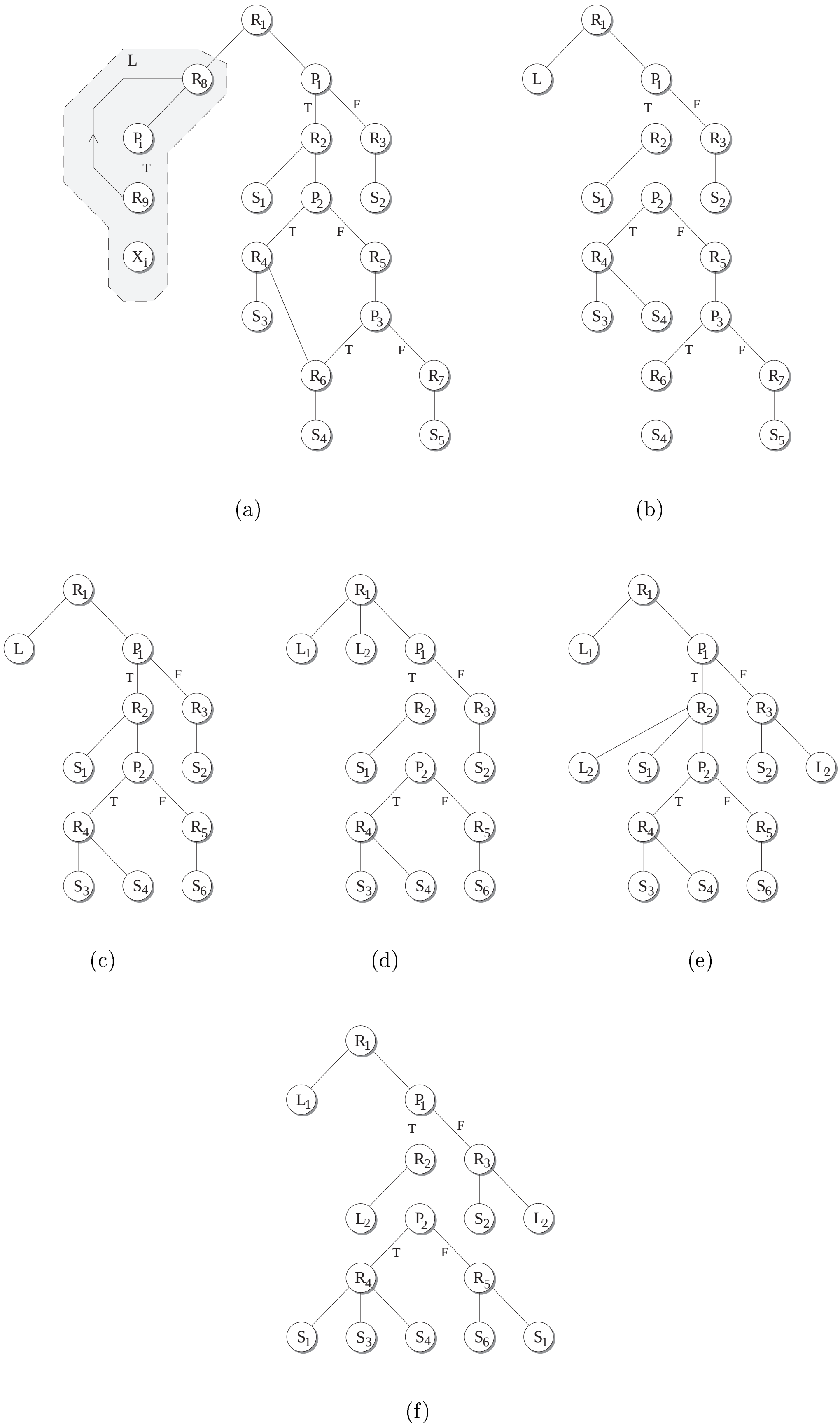


Abbildung 5.24: Verlauf des *region scheduling* anhand des Programms aus Abbildung 5.22.

$L_1 : X_1;\ X_2;\ \ldots X_{N/2};$

if P_1 **then** L_2: $X_{N/2+1};\ \ldots X_N;$

  if P_2 **then** $S_1;\ S_3;\ S_4;$
  else $S_1;\ S_6;$
  endif

else L_2: $X_{N/2+1};\ \ldots X_N;$

 $S_2;$

endif

Abbildung 5.25: Das Beispielprogramm aus Abbildung 5.22 nach der Instruktionsanordnung durch *region scheduling*.

Kapitel 6

Dual-Banked Memory

Der Speicher des ADSP-2106x läßt sich in zwei Bereiche aufteilen, man spricht hier von *dual-banked memory*: einen für den Programmcode (PM) und einen für die Daten (DM), wobei der erste auch als Datenbereich genutzt werden kann. Auf diese beiden Speicherbereiche kann gleichzeitig zugegriffen werden, wodurch sich – bei geschickter Wahl des Speicherplatzes einer Variablen – die Ausführungszeit eines Programmes reduzieren läßt.

Im folgenden wird das Problem der Zuweisung von Variablen an den zweigeteilten Speicher genauer spezifiziert, Verfahren zur Lösung vorgestellt und deren Vor- und Nachteile aufgezeigt.

6.1 Problem

Digitale Signalprozessoren (DSP) werden häufig in zeitkritischen oder Realzeit-Anwendungen eingesetzt. Da Speicherzugriffe in der Regel sehr zeitintensiv sind, liegt es nahe, an dieser Stelle Optimierungen vorzunehmen.

Eine solche Optimierung besteht darin, gleichzeitig (also im gleichen Zyklus) auf zwei oder mehrere Daten zugreifen zu können. Dies läßt sich durch sogenannte *dual-ported* Speicher realisieren, die in der Lage sind, Daten an verschiedenen Adressen auszulesen. Diese Form des Speichers ist allerdings sehr teuer, so daß man an seiner Stelle oft *dual-banked* Speicher verwendet.

Hierbei wird der gesamte Speicherbereich in zwei Teile aufgespalten, die unabhängig voneinander adressiert und somit gleichzeitig ausgelesen oder beschrieben werden können. Hiermit läßt sich im besten Falle eine Verdoppelung der Speicherbandbreite erzielen.

```
summe=0;
for (c=0; c<256; c++)
  summe+=A[c]*B[c];
```

```
mrf=0;
lcntr=256, do $L1-1 until lce;
  r0=dm(i0, m0), r4=pm(i8, m8);
  mrf=mrf+r0*r4;
$L1:
```

Abbildung 6.1: Beispiel eines parallelen Speicherzugriffs: Die einzelnen Komponenten der Vektoren A und B können gleichzeitig aus dem Speicher geladen werden.

Sind diese Voraussetzungen gegeben, liegt es nun am Programmierer bzw. Compiler, diese Möglichkeiten auch zu nutzen. Da das Programmieren in Assembler mühsam und fehleranfällig ist, wird diese Aufgabe mehr und mehr dem Compiler zuteil. Dieser muß dafür sorgen, daß Variablen sinnvoll in einem der beiden Speicherbereiche untergebracht werden, so daß Speicherzugriffe parallelisiert werden können.

Beispiel 15

Als Beispiel für einen parallelen Speicherzugriff sei auf Abbildung 6.1 verwiesen. Bei der Berechnung des Skalarprodukts der Vektoren A und B müssen sukzessive die einzelnen Komponenten geladen werden. Befinden sich die zu A und B gehörigen Daten in verschiedenen Speicherbereichen, kann dies gleichzeitig erfolgen.

□

Leider wird diese Besonderheit von vielen Compilern (insbesondere vom GNU C-Compiler für den ADSP-2106x) nicht berücksichtigt und unterstützt.

6.2 Datenduplizierung

Eine intuitive und einfache Möglichkeit, *dual-banked* Speicher auszunutzen, liegt darin, sämtliche Daten sowohl im Programmspeicher (PM) als auch im Datenspeicher (DM) abzulegen. So kann auf alle Paare von Daten parallel zugegriffen werden.

Diese Vorgehensweise besitzt aber mehrere Nachteile. So werden, um die Konsistenz der beiden Speicherbereiche zu gewährleisten, zusätzliche Speicheroperationen benötigt. Da beim ADSP-2106x auch solche Schreibzugriffe auf den Speicher parallelisiert werden können, ist dieser Umstand noch kein Grund, Datenduplizierung zu vermeiden.

Da aber DSP's dazu ausgelegt, klein und energiesparend zu sein, ist ihre Speicherkapazität beschränkt. Gängige Prozessoren besitzen etwa nur 16 bis 256 kB Hauptspeicher. Durch die Datenduplizierung wird also der verfügbare Speicher halbiert, was in den meisten Fällen eine nicht annehmbare Tatsache darstellt.

6.3 Datenpartitionierung

Im Idealfall sollten die Daten so auf die beiden Speicherbereiche verteilt sein, daß mögliche parallele Speicherzugriffe auch parallel ausgeführt werden können – ohne die Notwendigkeit, Daten zu duplizieren. Bei der Datenpartitionierung [SCL95] können mehrere Phasen unterschieden werden: In einen ersten Schritt müssen Partitionierungsbeziehungen zwischen den einzelnen Daten gesammelt werden, dies beinhaltet das Auffinden von potentiellen parallelen Speicherzugriffen. Diese Partitionierungsbeziehungen werden dann mit einem geeigneten Kostenmaß bewertet, welches Rückschlüsse auf die Ausführungszeit zuläßt. Schließlich werden die Daten unter Berücksichtigung des Kostenmaßes partitioniert, wodurch eine bessere Laufzeit erzielt wird.

6.3.1 Interferenzgraph

Zur Darstellung der Partitionierungsbeziehungen eignet sich ein **Interferenzgraph**. Hier werden die einzelnen Variablen (Daten) durch Knoten repräsentiert und die Beziehungen zwischen ihnen durch Kanten.

Der in Abbildung 6.1 vorgestellte Algorithmus aus [SCL95] dient zur Erzeugung des Interferenzgraphen. Er ist eng verflochten mit *list scheduling* (vergleiche Kapitel 5) und arbeitet auf Basisblöcken.

Zunächst wird für jeden Basisblock der Datenabhängigkeitsgraph erzeugt, der zur Bestimmung der Menge *data_ready* notwendig ist. In ihr werden solche Instruktionen aufgenommen, die keine Abhängigkeiten besitzen oder nur solche Abhängigkeiten mit bereits verplanten Instruktionen. Gemäß *list scheduling* werden nun Instruktionen aus *data_ready* zusammengefaßt, die parallel ausgeführt werden können.

An dieser Stelle weicht der Algorithmus aus Abbildung 6.1 vom reinen *list scheduling* ab. Werden Instruktionen entdeckt, die zwar datenkompatibel aber nicht kompatibel bezüglich

```
for (jeden Basisblock) {
      erzeuge den Datenabhängigkeitsgraph;
      weise jeder Operation eine Priorität zu;
      berechne die Menge data_ready;
     while (data_ready nicht leer) {
          sortiere die Operationen in data_read gemäß ihrer Priorität;
         for (jede op in data_ready) {
             if (op ist datenkompatibel) {
                if (op ist kompatibel bezüglich der funktionalen Einheit) {
                   füge op den Instruktionen zu und markiere sie als angeordnet;
                  if (op ist load/store)
                   ld/st_i = op;
                 }
                else if (op ist load/store) {
                   ld/st_j = op;
                  if (ld/st_i und ld/st_j greifen auf verschiedene Daten/Felder zu)
                   füge dem Interferenzgraphen ein Kante hinzu;
                  else
                   markiere Datum/Feld als zu duplizieren;
                 }
             }
         }
      berechne data_ready neu;
     }
 }
```

Algorithmus 6.1: Pseudocode zur Erzeugung des Interferenzgraphen.

ihrer funktionalen Einheit sind, wird überprüft, ob es sich um Speicherzugriffe handelt. Ist dies der Fall, wird weiterhin getestet, ob diese Zugriffe auf verschiedene Variablen erfolgen, da dann diese Zugriffe parallel erfolgen könnten; es wird eine Kante in den Interferenzgraphen eingefügt.

Wird auf die gleiche Variable zugegriffen, so läßt sich durch eine Kombination von Datenduplizierung und Datenpartitionierung ein Laufzeitgewinn erzielen (vergleiche [SCL95]).

6.3.2 Kostenmaß

Anhand des Kostenmaßes wird der im vorherigen Abschnitt erzeugte Interferenzgraph partitioniert. Somit hat seine Wahl auf die Güte des erzeugten Codes einen großen Einfluß. Als gute Wahl hat sich die Schleifenschachtelungstiefe erwiesen [SCL95]; hier wird jede Kante mit der Anzahl der sie umgebenden Schleifen gewichtet. Dies macht Sinn, da mit wachsender Schachtelungstiefe Instruktionen häufiger ausgeführt werden und sich durch vermeidbare Zyklen (aufgrund nicht parallelisierter Speicherzugriffe) die Ausführungszeit stark erhöht.

6.3.3 Partitionierung

Der erzeugte und mit Kantengewichten versehene Graph muß nun noch in zwei Knotenmengen partitioniert werden. Das Problem der Partitionierung bei minimalen Kosten (wobei sich die Kosten als Summe der Kantengewichte berechnen lassen, deren Knoten sich innerhalb derselben Menge befinden) ist $\mathcal{NP}$-vollständig [GJ79]. Um dennoch schnell eine Näherungslösung zu finden, verwenden [SCL95] eine Heuristik, genauer gesagt einen Greedy-Algorithmus.

Diese Wahl ist einerseits begründet in der Einfachheit des Algorithmus. Andererseits wurden mit ihm Ergebnisse erzielt, die der optimalen Lösung – in den meisten Fällen – sehr nahe kamen.

Die Vorgehensweise des Greedy-Algorithmus ist einfach zu beschreiben: Zunächst befindet sich der komplette Interferenzgraph in einer der beiden Mengen. Die Kosten der Partitionierung berechnen sich aus der Summe der Kantengewichte der Kanten, die Knoten innerhalb derselben Partition verbinden. Nun werden nach und nach Knoten aus der Menge, die zu Beginn den kompletten Graph enthielt, entfernt und der anderen zugeführt. Es wird jeweils derjenige Knoten überführt, der die größtmögliche Kostenersparnis einbringt. Können die Kosten nicht mehr gesenkt werden, terminiert das Verfahren.

```
D[c]=A[d]+B[e];
B[c]=B[d]+D[e];
C[c]=B[d]+C[e];
C[c]=A[d]+C[e];

for (c=0; c<5; c++)
  C[c]=A[c]+D[c];

A[c]=C[d]+D[e];
```

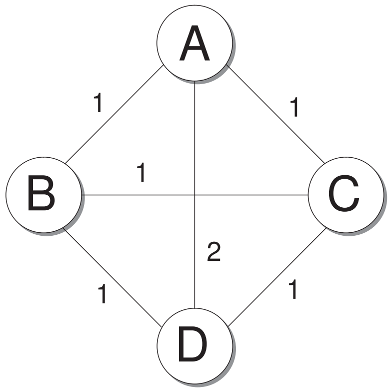


Abbildung 6.2: Beispielprogramm mit zugehörigem Interferenzgraph

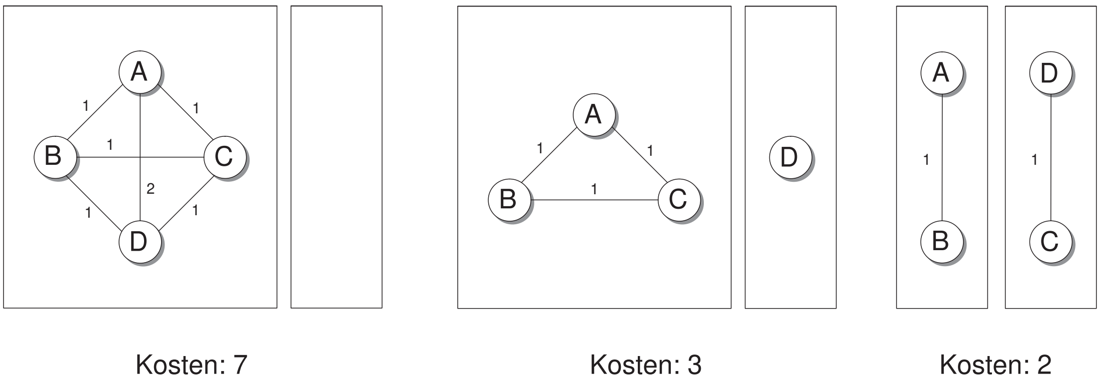


Abbildung 6.3: Schritte des Greedy-Algorithmus bei der Partitionierung

Beispiel 16

Zur Verdeutlichung sei hier auf das Beispiel in Abbildung 6.2 hingewiesen, das ein Codefragment und dessen Interferenzgraph darstellt. In Abbildung 6.3 wird der Ablauf des Greedy-Algorithmus visualisiert: Gestartet wird mit einer Partition, in der der komplette Graph in der einen Hälfte enthalten ist. Die Kosten belaufen sich auf 7. Durch Wahl des Knotens D kann die größtmögliche Kostenreduktion von 4 erzielt werden. Als nächster Knoten kann dann Knoten C gewählt werden, was eine Verringerung der Kosten um 1 bewirkt. B würde das gleiche bewirken, nur die Wahl von A würde die Kosten nicht verändern.

□

6.4 Ergebnisse

[SCL95] erzielten in ihren Benchmarks eine Leistungssteigerung hinsichtlich der Ausführungszeit von 13-49% im Kernelbereich und 3-15% bei Applikationen. Letztere stellen komplette Anwendungen dar, während Kernels kleine Programmfragmente sind, die eine oder mehrere Schleifen enthalten.

Der größere Leistungsgewinn im Kernelbereich läßt sich dadurch erklären, daß Zykleneinsparungen aufgrund paralleler Speicherzugriffe durch Schleifen vervielfacht werden. Anwendungen weisen in der Regel einen linearen Kontrollfluß auf, so daß parallele Operationen sich nicht so stark auf die Gesamtlaufzeit auswirken.

Tabelle 6.1 zeigt eine Übersicht über die in [SCL95] betrachteten Kernel, Tabelle 6.2 listet die untersuchten Applikationen auf. Die Einsparungen, die bei der Laufzeit erzielt wurden, sind in Abbildung 6.4 für die Kernel und in Abbildung 6.5 für die Applikationen dargestellt. „Ideal“ bezeichnet jeweils ein Referenzprogramm, bei dem mögliche Speicherzugriffe auch parallel getätigt werden können. „CB“ kennzeichnet den in diesem Kapitel vorgestellten Algorithmus, wobei die Kanten im Interferenzgraph mit der Schleifenschachtelungstiefe gewichtet werden; „Pr“ benutzt hingegen Informationen, die aus einem Profile-Durchgang gewonnen wurden, um die Kanten zu gewichten. „Dup“ indiziert Datenduplizierung.

Zusammenfassend kann festgehalten werden, daß mittels Datenpartitionierung Programmvariablen verschiedenen Speicherbänken zugewiesen werden können. Dies kann die Parallelität eines Programms steigern und somit sich positiv auf dessen Laufzeit auswirken.

Kernel		Beschreibung
k1	fft_1024	Schnelle Fourier-Transformation
k2	fft_256	
k3	fir_256_64	FIR-Filter
k4	fir_32_1	*finite impulse response*
k5	iir_64_4	IIR-Filter
k6	iir_1_1	*infinite impulse response*
k7	latnrm_32_64	Normalisierter Gitterfilter
k8	latnr_8_1	*normalized lattice*
k9	lmsfir_32_64	LSM-adaptiver FIR-Filter
k10	lmsfir_8_1	*least mean squared*
k11	mult_10_10	Matrizenmultiplikation
k12	mult_4_4	

Tabelle 6.1: DSP-Kernel

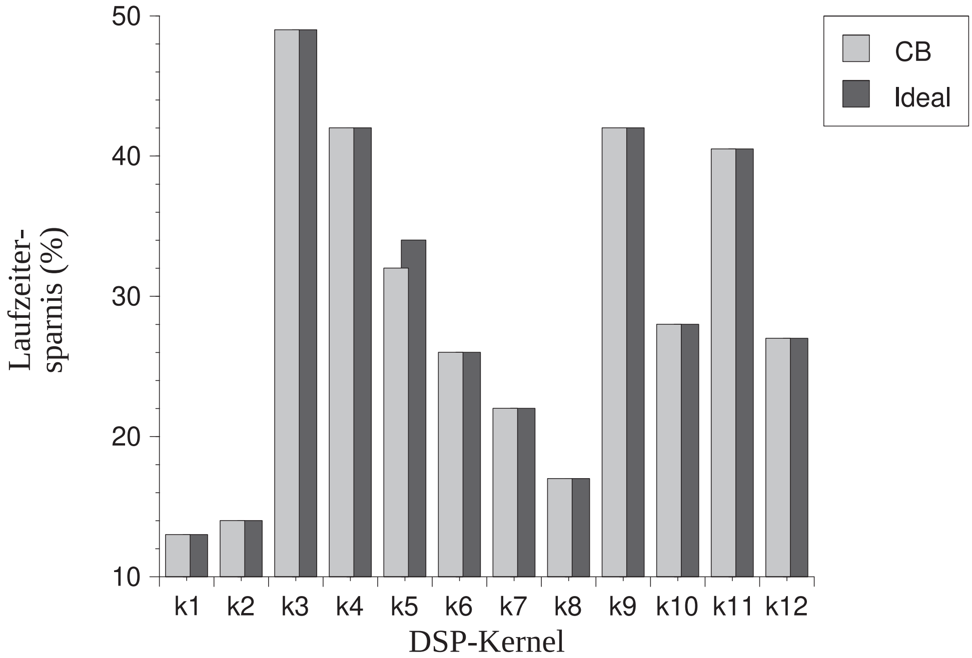


Abbildung 6.4: Laufzeitersparnis für DSP-Kernel.

Applikationen		Beschreibung
a1	adpcm	Sprachverschlüssler *adaptive, differential pulse-code modulation speech encoder*
a2	lpc	Sprachverschlüssler *linear predictive coding speech encoder*
a3	spectral	Spektrale Analyse
a4	edge_detect	Kantenerkennung
a5	compress	Bildkompression
a6	histogramm	Bildverbesserung mittels Histogramm
a7	V32encode	V.32 Modemverschlüssler
a8 a9 a10	G721MLencode G721MLdecode G721WFencode	Verschiedene Implementierungen des CCITT G.721 ADPCM Sprachverschlüsslers
a11	trellis	Trellis-Dekoder

Tabelle 6.2: DSP-Applikationen

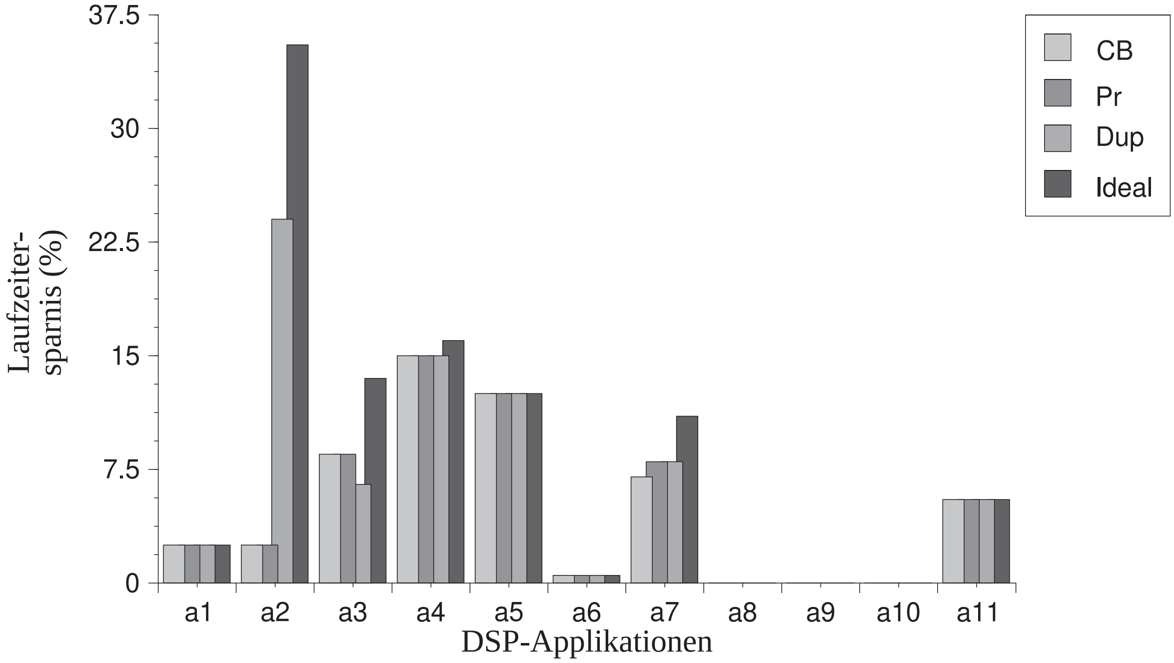


Abbildung 6.5: Laufzeitersparnis für DSP-Applikationen.

Kapitel 7

CIS

`CIS` (*comparative instruction scheduler*) ist die Implementierung verschiedener Verfahren zur Instruktionsanordnung, wobei sowohl in dieser Arbeit vorgestellte klassische Algorithmen als auch solche, die sich der linearen Programmierung bedienen (vorgestellt in [Käs97]), verwendet wurden. Mittels Kommandozeilenoptionen (eine genaue Beschreibung der einzelnen Optionen ist in Anhang A zu finden) kann des gewünschte Verfahren ausgewählt werden.

`CIS` arbeitet in vier Phasen:

1. Einlesen des Quellprogramms, das in der Assemblersprache des ADSP-2106x vorliegt; hierbei findet auch eine Syntaxüberprüfung statt.

2. Erzeugung der Zwischendarstellungen wie Kontrollflußgraph und Datenabhängigkeitsgraph sowie der Zwischendarstellungen für die Verfahren der linearen Programmierung [Käs97].

3. Anordnen der Instruktionen mittels des ausgewählten Verfahrens.

4. Erzeugen einer Ergebnisdatei in Assembler.

Auf die Implementierung der einzelnen Phasen wird nun näher eingegangen. Daran fügt sich eine Auflistung der während der verschiedenen Schritte erzeugten VCG-Dateien. Abschließend werden Testprogramme vorgestellt und die mittels `CIS` erzielten Ergebnisse diskutiert.

7.1 Einlesen des Quellprogramms

Als Quellprogramm erwartet `CIS` eine in der Assemblersprache des ADSP-2106x geschriebene Textdatei. Diese kann entweder durch Übersetzen eines C-Programms mit dem GNU C-Compiler `gcc` für den ADSP-2106x bereitgestellt werden, oder es wird eine handgeschriebene Assemblerdatei verwendet.

Der Scanner wurde unter Verwendung des lexikalischen Analysegenerators `flex` erzeugt, die Grammatik der Assemblersprache mittels `bison` in einen Parser übersetzt.

Entsprechend der Struktur der Assemblerdateien, liefert der Parser eine Liste von Segmenten. Die Segmente bestehen ihrerseits aus einer Liste von Direktiven, Sprungmarken und Mikroinstruktionen. Letztere beinhalten zu diesem Zeitpunkt jeweils nur eine Mikrooperation, Multifunktionsinstruktionen werden unter Wahrung der Antidependenzen (Benutzung-Setzung-Abhängigkeiten) serialisiert.

7.2 Zwischendarstellungen

Die im ersten Schritt erzeugte Segmentliste ist sehr unhandlich und bietet in ihrem Aufbau keine Informationen über die Programmstruktur. Darum wird in einem ersten Schritt ein Kontrollflußgraph (siehe Abschnitt 4.1) für jedes Segment und jede darin enthaltene Prozedur erzeugt. Im folgenden werden Datenabhängigkeiten analysiert und ein Datenabhängigkeitsgraph wie in 4.3 aufgebaut.

Weitere Zwischendarstellungen werden nach Bedarf erzeugt. So benötigen *list scheduling*, *critical path* und sämtliche Schedulingverfahren, die auf der linearen Programmierung aufbauen, den Basisblockgraphen aus Abschnitt 4.2. Der Kontrollabhängigkeitsgraph wird wiederum von den SILP- und OASIC-Formulierungen verwendet.

Bei Verfahren, die eine erneute Registerallokation durchführen, finden Lebenszeiten und der zugehörige Interdependenzgraph Verwendung.

Eine Übersicht der Algorithmen, die in `CIS` implementiert wurden, findet sich in Anhang D.

7.3 Verfahren

7.3.1 List Scheduling

List scheduling ist wie in Abbildung 7.1 schematisch dargestellt implementiert. Von zentraler Bedeutung ist die Funktion `choose_microop`, die aus der Menge *data_ready* die Mikrooperation mit höchster Priorität und nicht gesetztem Flag zurückliefert. Dieses Flag wird für die ausgewählte Operation gesetzt, so daß sie in einem folgenden Aufruf nicht wieder verwendet wird. Sind alle Mikrooperation markiert, liefert `choose_microop` `NULL` und die Flags werden wieder gelöscht.

```
list_scheduling(BASICBLOCK bb)
{
  LIST mops;
  int cycle=0;
  MICROOP m;

  update_data_ready();
  while (entries(data_ready) > 0) {
    cycle++;

    while ((m=choose_microop()) != NULL)
      if (are_parallel(m, mops))
        append(mops, m);

    forall(mops, m)
      m->cycle=cycle;

    remove_from_data_ready(mops);
    update_data_ready();
    empty(mops);
  }
}
```

Algorithmus 7.1: Schema der Implementierung des *list scheduling*-Verfahrens.

Die Auswahl der Mikrooperation erfolgt mit absteigender Priorität. Diese kann durch die folgenden Heuristiken bestimmt werden:

first fit Die erste Mikrooperation in *data_ready*, die noch nicht verplant oder abgewiesen wurde, wird ausgewählt.

longest remain Nach jeder Aktualisierung wird ein Zähler in den Mikrooperation der Menge *data_ready* um eins erhöht. Diejenige Operation mit höchstem Zähler, also die mit der längsten Verweildauer in *data_ready*, wird ausgewählt.

max depend Die Priorität einer Operation wird durch die Anzahl der ausgehenden Abhängigkeitskanten, die zu noch nicht verplanten Operationen führen, bestimmt. Die Mikrooperation mit der höchsten Priorität wird verwendet.

highest level Hierbei erhält eine Mikrooperation als Priorität die Länge des längsten von ihr ausgehenden Pfads im Datenabhängigkeitsgraph.

Wird beim Aufruf von `CIS` die Options `-r` angegeben, so wird dem *list scheduling*-Verfahren eine Registerallokation vorgeschaltet, welche durch Ergebnisse eines *list scheduling*-Probelaufs gesteuert wird. In diesem Probelauf werden solche Instruktionen ausfindig gemacht, die aufgrund ihrer Registerindizes einen Ressourcenkonflikt auslösen und somit eine Parallelisierung zweier Operationen verhindern. Diese werden markiert und ihnen wird eine Registergruppe zugewiesen, die diesen Konflikt auflösen würde.

Während der Registerallokation wird nun versucht, diese Gruppenzugehörigkeiten zu wahren. Sollten hier allerdings Probleme bezüglich der Registerverfügbarkeit auftreten, werden auch andere Gruppen verwendet. Das folgende Lauf der Instruktionsanordnung mittels *list scheduling* beendet die Instruktionsanordnung.

7.3.2 Critical Path

Die Implementation richtet sich nach dem im Abschnitt 5.2.2 vorgestellten Verfahren (Abbildung 7.2). In einem ersten Schritt werden frühe und späte Partition berechnet, wobei für die späte Partition der Datenabhängigkeitsgraph, der als Matrix vorliegt, lediglich zu transponieren und dem gleichen Verfahren wie zur Berechnung der frühen Partition zuzuführen ist. Die Werte der frühen und späten Partition werden in den Mikrooperation selbst gespeichert. Danach werden kritische Operationen in den zugehörigen Zyklus eingeordnet, nichtkritische in eine temporäre Liste aufgenommen. Nun wird versucht, die Mikrooperationen der Liste in die Zyklen zwischen früher und später Partition einzuordnen. Können sie dort nicht parallel eingeführt werden, wird in letztmöglichen Zyklus ein Unterzyklus eingefügt. Dies bedeutet, daß in diesem Zyklus eine neue Mikroinstruktion angefügt wird. Diese Aufgabe übernimmt die Funktion `insert_microop`.

7.3.3 Globale Verfahren

Mit Hinblick auf die ausgewählten Testprogramme wurden in `CIS` nur Verfahren zur Instruktionsanordnung implementiert, die auf Basisblöcken arbeiten. Die in der Testsuite

```
critical_path(BASICBLOCK bb)
{
  LIST frames[ddg_depth];
  LIST non_critical;
  MICROOP m;
  int c;

  create_partitions(bb);

  forall(bb, m)
    if (m->ep == m->lp)
      insert_mop(frames[m->ep], m);
    else
      append(non_critical, m);

  forall(non_critical, m)
    for (c=m->ep; c<=m->lp; c++)
      if (insert_mop(frames[c], m))
        break;
}
```

Algorithmus 7.2: Übersicht der Implementierung des *critical path*-Verfahrens.

enthaltenen Programme weisen größtenteils wenige bis keine Verzweigungen auf, so daß die Basisblöcke im Vergleich zum kompletten Programmtext groß sind. Ein anderer Grund, globale Verfahren nicht zu unterstützen, liegt in dem vergleichenden Charakter dieser Diplomarbeit: Programme, die eine globale Analyse rechtfertigen würden, können von den Verfahren, die auf der linearen Programmierung aufsetzen, nicht verarbeitet werden (aufgrund langer Berechnungszeiten und großen Problemformulierungen).

7.3.4 Registerallokation

Das Verfahren, welches zur Registerallokation verwendet wird, ist ein leicht modifiziertes Graphfärbeverfahren, wie es in Abschnitt 3.1.1 vorgestellt wurde. Hierbei werden zuerst die Nachbarn des zu betrachtenden Knotens untersucht und anhand deren Färbung und der Registergruppe, aus der der aktuelle Knoten ein Register erhalten soll, wird eine Farbe und somit ein Register ausgewählt.

7.3.5 Dual-banked Memory

Die Assemblersprache des ADSP-2106x kennt nur einfache Datenstrukturen. Felder werden beim Übersetzungsprozeß vom Compiler implizit einem Speicherbereich zugewiesen und mittels der Adreßgeneratoren zur Laufzeit adressiert. So fällt es schwer, Felder und Zugriffe auf diese ausfindig zu machen. Abhilfe würde eine alias-Analyse schaffen, durch die Speicherzugriffe auf aufeinanderfolgende Speicherzellen gefunden werden könnten.

Aus diesem Grund wurde auf die Implementation des in Kapitel 6 vorgestellten Verfahrens verzichtet. Der geeignete Ausgangspunkt für eine solche Analyse wäre vielmehr das Backend eines Codeerzeugers, in dem strukturelle Informationen über die einzelnen Variablen vorliegen.

7.4 VCG-Dateien

Während der Erzeugung der Zwischendarstellungen werden verschiedene VCG-Dateien generiert (in Abhängigkeit des ausgewählten Scheduling-Verfahrens). Dazu gehören die folgenden Dateien:

PARSE.vcg spiegelt die Struktur des Quellprogramms wider, wie es vom Parser eingelesen wurde.

LIST.vcg Sequentialisierte Darstellung des Quellprogramms. Das Programm wird als Liste von Segmenten aufgefaßt, die wiederum Listen von Direktive, Labels und Mikrooperationen sind.

CFG.vcg Der Kontrollflußgraph des Quellprogramms. Dieser beinhaltet schon die Datenabhängigkeiten.

BBG.vcg ist eine graphische Version des Basisblockgraphen.

PDOM.vcg ist eine Visualisierung des Postdominator-Baumes, der zur Erzeugung des Kontrollabhängigkeitsgraphen benötigt wird.

CDG.vcg beinhaltet den Kontrollabhängigkeitsgraph.

FLOW.vcg Stellt die Flüsse bezüglich der Maschinenressourcen dar, wie sie bei SILP- und OASIC-Formulierungen entstehen.

REG.vcg Hier werden ebenfalls Flüsse aber diesmal bezüglich der Register dargestellt, wie sie bei Verfahren der linearen Programmierung mit Registerallokation auftreten.

Lifetime.vcg Stellt die Lebenszeiten bei OASIC-Formulierungen dar.

7.5 Testprogramme

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.

Der Großteil der getesteten Codesequenzen basiert auf handgeschriebenem Assemblercode, der durch geschickte Aufteilung der Daten auf die beiden Speicherbänke und Wahl der Registergruppen eine hohe Parallelität aufweist. Diese Programme wurden im Vorfeld – unter Wahrung der Semantik – sequentialisiert, d.h. Multioperations-Instruktionen in eine Folge von einzelnen Anweisungen umgesetzt.

Eine Übersicht der untersuchten Programme ist in Tabelle 7.1 gegeben. Die einzelnen Testprogramme werden nun genauer vorgestellt. Ihr Quelltext – sowohl in unoptimierter als auch in optimierter Form – ist in Anhang E zu finden.

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
conv3x3	49	1	0	193

Tabelle 7.1: Die Beispielprogramme.

7.5.1 fir.s

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 lineare Differentialgleichung beschrieben:

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

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

Das Beispielprogramm `fir.s` (Anhang E.1) stellt eine Implementierung eines solchen FIR-Filters dar. Da die Implementierung auf Prozedurebene arbeitet, wurde die Initialisierungsroutine des Originalprogrammes vernachlässigt (sie ist in einer externen Bibliothek untergebracht) und nur die eigentliche Berechnungsprozedur als Eingabeprogramm verwendet. Das sequentialisierte Eingabeprogramm umfaßt 18 Instruktionen. Nach einem Initialisierungsteil wird eine Schleife bestehend aus 4 Instruktionen durchlaufen. Daran schließt sich ein Epilog an.

7.5.2 cascade.s

Die im Eingabeprogramm cascade.s (Anhang E.2) 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 der eigentlichen cascade-Funktion ist eine Schleife; aus dieser wurde die Schleifenanweisung entfernt, so daß genau eine Iteration der Schleife durchgeführt wird. Dies resultiert in einem Basisblock bestehend aus 20 Instruktionen.

7.5.3 dft.s

Dieses Beispielprogramm berechnet die komplexe diskrete Fourier-Transformation (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: $w_n = e^{-i\frac{2\Pi}{n}}$.

Dieses Programm besteht in der Hauptsache aus zwei geschachtelten Schleifen und weist – abgesehen vom Prozedurrücksprung – keine Verzweigungen auf. Die Anzahl der Instruktionen nach der Sequenzialisierung beträgt 26. Der vollständige Programmtext ist in Anhang E.3 zu finden.

7.5.4 whetp3.s

Während die übrigen Testprogramme manuell erstellte (und somit „voroptimierte“) Codesequenzen aus typischen Einsatzgebieten digitaler Signalprozessoren darstellen, ist dieses Beispiel dem bekannten Whetstone-Benchmarks entnommen. Als Eingabeprogramm wird dabei die vom `g21k`[1] für diese Funktion erzeugte Codesequenz verwendet. Sie berechnet die folgende C-Funktion:

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

Das vom `g21k` gelieferte Assemblerprogramm (Anhang E.4) besteht aus 20 Instruktionen; werden die vorgenommenen Parallelisierungen wieder rückgängig gemacht, erhalten wir ein aus 26 Instruktionen bestehendes Eingabeprogramm, welches aus einem Basisblock besteht.

[1]Dies ist der von Analog Devices adaptierte `gcc`.

7.5.5 histo.s

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 verwendete Algorithmus geht davon aus, daß das Graustufenbild in einem Puffer des Datenspeichers liegt; das Histogramm wird ebenfalls im Datenspeicher aufgebaut.

Das Programm, welches in Anhang E.5 zu finden ist, besteht aus 43 Instruktionen, darunter zwei Schleifenanweisungen: eine mit 12 Schleifeninstruktionen und die folgende mit 4. Dazwischen erfolgen einige verzweigungsfreie Berechnungen.

7.5.6 conv3x3.s

Das Beispielprogramm `conv3x3.s` (Anhang E.6) 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 $I\!R$ gegeben. Ihre Konvolution ist eine Funktion $\Phi * \Psi$ mit

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

Das Beispielprogramm berechnet eine 3x3-Konvolution. In sequentialisierter Form besteht es aus 124 Mikroinstruktionen. Um die Berechnungszeit für die Verfahren, die sich der linearen Programmierung bedienen, einzuschränken, wurde aus diesem Programm ein Basisblock extrahiert und für die resultierende Codesequenz als Eingabe der Instruktionsanordnung verwendet. Der Basisblock enthält 49 Instruktionen, die sehr stark voneinander abhängig sind.

7.6 Ergebnisse

Die Ergebnisse der unterschiedlichen graphbasierten Verfahren zur Instruktionsanordnung hinsichtlich der Laufzeit und der erzielten Anzahl von Instruktionen sind in den Tabellen 7.2 und 7.3 aufgelistet. Die Anzahl der Instruktionen in der optimalen[2] Anordnung wurde aus [Käs97] entnommen. Die Ergebnisse wurden auf einem Computeserver Ultra 2/2x200 mit 1024 MByte Hauptspeicher unter Solaris 2.5.1 erzielt.

Auf die Darstellung der Ergebnisse von *list scheduling* mit integrierter Registerallokation wurde verzichtet, da sich dadurch bei den gewählten Testprogrammen keine Instruktionseinsparung erzielen ließ. Dies liegt u.a. daran – betrachtet man ausschließlich die manuell erstellten Programme –, daß der Registerallokator die bereits vorgenommene Registerzuteilung als Ausgangsbasis übernimmt und nur an solchen Stellen Änderungen vornimmt, die zu einer höheren Parallelität führen. Die Registerverteilung der handgeschriebenen Programme ist aber schon so durchgeführt, daß keine weiteren parallelen Berechnungen mehr möglich sind. Bei whetp3.s lassen sich durch eine neue Registerallokation keine Instruktionen einsparen, da viele der vorkommenden Operationen ein Register lesen und das Ergebnis im gleichen Register wieder ablegen. Dies verhindert in den meisten Fällen die Bildung einer größeren Instruktion.

Bei Ausnutzung der beiden Speicherbänke läßt sich lediglich bei whetp3.s ein Datenzugriff parallel ausführen und dadurch eine Instruktion einsparen. Die handgeschriebenen Programme schöpfen den simultanen Speicherzugriff bereits aus.

Alle Berechnungen benötigen weniger als eine Sekunde. Auffällig sind hierbei die Zeiten für *list scheduling* mit der *highest level first*-Heuristik. Hier sind die Laufzeiten erheblich größer als die der anderen Heuristiken oder von *critical path*. Eine Ursache ist in der Berechnung der längsten Wege im Datenabhängigkeitsgraph zu finden. Das verwendete Verfahren (siehe Anhand D) ist kubisch in der Anzahl der Instruktionen. *Critical path* benötigt zur Berechnung der Partitionen zwar auch die Tiefe der einzelnen Operationen im Datenabhängigkeitsgraph, bei *list scheduling* werden aber beim Aktualisieren der *data_ready*-Menge Datenabhängigkeiten dahingehend überprüft, ob diese einer Aufnahme der Instruktion in diese Menge im Wege stehen. An dieser Stelle beeinflußt die Anzahl der Datenabhängigkeiten die Laufzeit noch zusätzlich. Die Zeiten der übrigen Verfahren sind nahezu identisch.

First fit und *longest remain* stimmen in ihren Ergebnissen überein. Dies hat zwei Gründe. Einerseits ist die Menge *data_ready* als Liste implementiert; neue Operationen werden am Ende angefügt. So befinden sich die Instruktionen, die sich am längsten in *data_ready* befinden am Anfang der Liste. Aus dieser werden vom Anfang der Liste an jeweils Operationen

[2] Hierbei wird der Begriff der Optimalität gleichgesetzt mit derjenigen Anordnung, die die minimale Anzahl von Instruktionen beinhaltet.

Programm	*list scheduling* ff[a]	lr[b]	md[c]	hl[d]	*critical path*
fir	0,014s	0,014s	0,014s	0,038s	0,014s
cascade	0,012s	0,013s	0,013s	0,019s	0,013s
dft	0,021s	0,021s	0,021s	0,096s	0,021s
whetp3	0,030s	0,030s	0,031s	0,116s	0,031s
histo	0,062s	0,063s	0,063s	0,40s	0,067s
conv3x3	0,069s	0,069s	0,070s	0,116s	0,075s

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

Tabelle 7.2: Ausführungszeiten der verschiedenen Verfahren zur Instruktionsanordnung.

genommen und auf Verträglichkeit mit denen im aktuellen Zyklus bereits verplanten Operationen getestet. Der zweite Grund ist in den Testprogrammen zu finden, welche (bis auf eine Ausnahme) handgeschrieben sind und sequentialisiert wurden. Dadurch entsprechen sich die erste passende Operation und diejenige mit der längsten Verweildauer.

Der letztere Grund erklärt auch die Tatsache, daß die beiden ersten Heuristiken bei allen manuell erstellten Programmen besser oder gleich gut abschneiden im Vergleich zu *max depend* und *highest level first*. Bei whetp3.s, welches mittels `g21k` erzeugt wurde, kann *highest level first* ein besseres Ergebnis aufweisen. Durch die Bevorzugung von Operationen, die auf längeren Wegen im Datenabhängigkeitsgraph liegen, werden Parallelisierungen möglich, die von den anderen Verfahren nicht vorgenommen werden können.

An den Testprogrammen cascade.s und whetp3.s ist der Einbruch von *max depend* auffällig (die Abweichungen vom optimalen Ergebnis betragen 114,3% bzw. 25%). In diesen beiden Programmen existieren viele Abhängigkeiten zwischen zwei aufeinanderfolgenden Operationen. Somit werden Parallelisierungen mit späterfolgenden Operationen erschwert, was zu dieser schlechteren Instruktionsanordnung führt.

Die optimale Instruktionsanordnung wird nur in von 6 von 30 Fällen erreicht. Tabelle 7.4 stellt die Abweichungen des Ergebnisses der verschiedenen Verfahren vom Optimum dar. Hierin zeigt sich auch, daß die etwas aufwendigere *highest level first*-Heuristik bessere Instruktionsfolgen generiert als die übrigen Heuristiken. *First fit* und *longest remain* weisen immer noch bessere Ergebnisse als der *critical path*-Algorithmus auf.

Tabelle 7.4 sollte nicht Anlaß dazu geben, die klassischen Verfahren als ineffizient zu deklarieren. Bei der Größe der betrachteten Programme und der daraus resultierenden (verhältnismäßigen) geringen Anzahl von Instruktionen in der optimalen Anordnung fallen geringe

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

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

Tabelle 7.3: Anzahl der Instruktionen nach den verschiedenen Scheduling-Algorithmen.

Abweichungen eher ins Gewicht. So resultiert eine Abweichung von 2 Instruktionen bei cascade.s mit *first fit* in 25%, obgleich eine Einsparung von 11 Instruktionen gegenüber dem ursprünglichen Code erreicht werden konnte. Diese Übersicht dient vielmehr dazu, die einzelnen Verfahren der graphbasierten Scheduling-Algorithmen untereinander vergleichen zu können.

Eine Gegenüberstellung der minimalen und maximalen Laufzeiten sowohl klassischer Verfahren als auch der SILP- und OASIC-Formulierungen (siehe hierzu [Käs97]) findet sich in Tabelle 7.5. Letztere Verfahren liefern (bis auf eine Ausnahme) die optimale Lösung. Auffällig hieran sind die recht unterschiedlichen Laufzeiten zur Berechnung der linearen Programme. Während die Ausführungszeiten der klassischen Verfahren mit zunehmender Programmgröße steigen, kann zwischen Größe des Programms und Dauer der Berechnung bei den SILP- und OASIC-Formulierungen – sofern auch die Heuristiken betrachtet werden – keine eindeutige Aussage getroffen werden.

Man erkennt, daß sich die Größe solcher Programme, die mittels SILP oder OASIC angeordnet werden sollen, nicht beliebig steigern läßt. Schon bei relativ kleinen Programmen (histo.s, 43 Operationen) dauert eine Berechnung mittels OASIC mehr als 24 Stunden. Die graphbasierten Verfahren benötigen weniger als eine Sekunde.

Die Verfahren, die sich der linearen Programmierung bedienen, haben den Vorteil, Aussagen über die Codequalität liefern zu können. Terminiert das Verfahren mit einer zulässigen Lösung, ist der erzeugte Code optimal hinsichtlich der Anzahl der Instruktionen. Auch lassen sich untere Schranken für die minimal erforderliche Anzahl von Instruktionen mittels linearer Programme berechnen, mit deren Hilfe dann die Qualität von Näherungslösungen angegeben werden kann [Käs97]. Auf der Seite der graphbasierten Verfahren kann lediglich durch die Tiefe des Datenabhängigkeitsgraphen eine untere Schranke für die mini-

Programm	*list scheduling* ff[a]	lr[b]	md[c]	hl[d]	*critical path*	
fir	25%	25%	25%	25%	25%	25%
cascade	28,6%	28,6%	114,3%	28,6%	57,1%	53,6%
dft	14,3%	14,3%	14,3%	14,3%	14,3%	14,3%
whetp3	10%	10%	25%	5%	15%	13,0%
histo	0%	0%	0%	0%	0%	0%
conv3x3	0%	0%	5,9%	5,9%	5,9%	3,5%
	13,0%	13,0%	30,75%	12,3%	19,6%	

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

Tabelle 7.4: Abweichungen der Ergebnisse der verschiedenen Scheduling-Algorithmen vom Optimum ausgedrückt in Prozent des optimalen Ergebnisses.

male Anzahl von Instruktionen gegeben werden, Ressourcenkonflikte werden hierbei nicht berücksichtigt. Untere Schranken auf Grundlage des Datenabhängigkeitsgraphen werden in Tabelle 7.6 der optimalen und der besten, durch die klassischen Verfahren erzielten Anordnung gegenübergestellt. Die Abweichung der Anzahl der Instruktionen in der optimalen Instruktionsanordnung von der Tiefe des Datenabhängigkeitsgraphen beträgt bis zu 54%, wodurch die Güte dieser Schranke in Frage gestellt wird.

Der große Vorteil klassischer Verfahren liegt in ihrem Laufzeitverhalten. Auf Grund dieser Tatsache lassen sich auch größere Programme in eine effizientere Form transformieren. Dort kann dann auch durch den Einsatz globaler Verfahren weiterer Parallelismus aufgedeckt werden. Formulierungen wie SILP und OASIC, die sich der linearen Programmierung bedienen, sind in der Lage, für relativ kleine Programme die optimale Instruktionsanordnung zu finden. Bei größer werdender Instruktionsanzahl erreichen die ILP-Formulierungen schnell gewaltige Größen ($>$ 20 MByte), der Aufwand zur Lösung dieser linearer Programme ist dementsprechend groß.

Programm	klassisch		SILP		OASIC	
	min	max	min	max	min	max
fir	0,014s	0,038	1,69s	19,58s	251s	251s
cascade	0,012s	0,019s	0,19s	0,32s	1,04s	5,45s
dft	0,021s	0,096s	47,42s	131s	210s	210s
whetp3	0,030s	0,116s	3h 23min	3h 23min	24min 53s	24min 53s
histo	0,062s	0,400s	16min 23s	5h 9min	> 24h	> 24h
conv3x3	0,069s	0,116s	347s	2h 1min	145s[a]	1h 45min

[a]nicht optimal

Tabelle 7.5: Vergleich der Laufzeiten der verschiedenen Scheduling-Algorithmen. Die Laufzeiten der SILP- und OASIC-Ansätze beziehen sich auf optimale Anordnungen.

Programm	untere Schranke	klassisch	optimal
fir	8	10	8
cascade	5	9	7
dft	10	16	14
whetp3	13	21	20
histo	23	31	31
conv3x3	12	17	17

Tabelle 7.6: Gegenüberstellung unterer Schranken, die durch die Tiefe des Datenabhängigkeitsgraphen gegeben sind, und der Anzahl der Instruktionen in einer durch graphbasierte Verfahren erzielten Anordnung.

Kapitel 8

Zusammenfassung und Ausblick

Klassische Verfahren zur Instruktionsanordnung können die Parallelität eines Programms steigern. Die nach der Anzahl der Instruktionen gewichtete mittlere Abweichung der in dieser Arbeit untersuchten Programme beträgt 11,89% vom Optimum. Eine größere Ausbeute an parallelen Operationen ließe sich mittels globaler Scheduling-Verfahren erreichen. Die hierfür typischen großen Programme können mittels der ILP-Ansätze aufgrund der hohen Komplexität nicht mehr verarbeitet werden. In Verbindung mit den sehr guten Laufzeiten der graphbasierten Algorithmen liefert dies einen Grund für die Tatsache, daß diese Verfahren weltweit in optimierenden Compilern Verwendung finden, somit also von praktischer Bedeutung sind.

Eine Interaktion zwischen Instruktionsanordnung und Registerallokation läßt sich bei klassischen Verfahren relativ leicht realisieren und bedeutet einen geringen Mehraufwand hinsichtlich der Laufzeit. Auch die Ausnutzung zweier Speicherbänke kann in graphbasierte Verfahren einfach integriert werden (vergleiche hierzu Algorithmus 6.1), was bei ILP-Ansätzen aufgrund der in [Käs97] genannten Gründe nur schwer oder gar nicht möglich ist. Hierdurch läßt sich die Güte der Instruktionsanordnung bei maschinell erstelltem Assemblercode verbessern.

Ein Vorteil der ILP-Verfahren ist, daß diese bei Terminierung mit einer zulässigen Lösung die Optimalität der erzeugten Anordnung garantieren. Allerdings wird die Optimalität der Lösung durch lange Laufzeiten bezahlt. Die Größe der zugeführten Programme beeinflußt die Laufzeit so stark, daß nur kleinere Programmfragmente angeordnet werden können. Wird zusätzlich noch eine Registerallokation vorgenommen, wird die Anzahl der maximal verarbeitbaren Instruktionen nochmals herabgesetzt. Die Güte der Instruktionsanordnung, die mittels graphbasierter Verfahren erstellt wurde, kann nicht näher bestimmt werden. Eine untere Schranke auf Basis der Tiefe des Datenabhängigkeitsgraphen liefert nur einen groben Anhaltspunkt.

Somit bleiben ILP-Verfahren auf kleine Programmstücke, wie etwa Funktionen oder innere Schleifen, die im DSP-Bereich stark vertreten sind, beschränkt. Ihre Attraktivität besteht in der optimalen Anordnung zusammen mit einer vertretbaren Laufzeit. Ihre hohe Komplexität macht einen Einsatz auf globaler Ebene unmöglich. Hier sind vielmehr die heuristischen Verfahren vorzuziehen, sowohl lokale als auch globale. Diese erzeugen zwar in der Regel keine optimale Instruktionsanordnung, durch Verschränkung des Schedulers mit der Registerallokation sowie Berücksichtigung zweier Speicherbänke lassen sich dennoch gute Resultate erzielen.

Zur Zeit stellen klassische Verfahren zur Instruktionsanordnung die einzige Möglichkeit dar, größere Programme bei annehmbarer Laufzeit anzuordnen. Eine Verbesserung der Qualität des erzeugten Codes ließe sich durch Verknüpfung der beiden, bisher getrennten Lösungsansätze erzielen: graphbasierte Verfahren auf der einen, ILP-Ansätze auf der anderen Seite. Denkbar wäre z. Bsp. eine Integration der linearen Programmierung innerhalb eines graphbasierten Verfahrens. Dort könnte diese dann auf geeignete Codestücke angewandt und für diese Teilprobleme eine optimale Anordnung erzeugt werden.

Anhang A

CIS Optionen

`-b` Ermittlung einer unteren Schranke für die optimale Lösung (Verfahren B1; vgl. [Käs97]).

`-c` Nur gültig in Verbindung mit den Optionen `-o` und `-s`. Eine Verschiebung der Instruktionen über die Basisblockgrenzen wird unterdrückt (→ vollständiges *branch and bound*-Verfahren).

`-d` Hierdurch werden während der Laufzeit von `CIS` Debuginformationen ausgegeben.

`-h` Dient zur Auswahl der Heuristik, die zur Auswahl der Operationen aus der Menge *data_ready* verwendet werden soll. Mögliche Optionen sind `firstfit`, `longestremain`, `maxdepend` und `highestlevel` (Voreinstellung).

`-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 an potentieller Parallelität orientiert.

`-m` Gibt die Heuristik an, die zur Berechnung einer Approximation verwendet werden soll. Optionen sind `approx1a`, `approx1a`, `approx1b`, `approx2a` `approx2b`, `approx3`, `approx32` und `default` (hierbei wird keine Approximation verwendet). Nur gültig in Verbindung mit den Optionen `-o` oder `-s`.

`-r` Erzwingt – falls möglich – eine neue Registerallokation bei der ausgewählten Optimierungsmethode.

`-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 (siehe auch [Käs97]).

`-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 (siehe auch [Käs97]).

`-v` Mit dieser Optionen werden VCG-Dateien für Verschiedene Phasen und Zwischendarstellungen erzeugt. Eine Übersicht über erzeugte Graphen ist in Abschnitt 7.4 zu finden.

`-x` Ermittlung einer unteren Schranke für die optimale Lösung (Verfahren B2; vgl. [Käs97]).

`-P` Als Verfahren zur Instruktionsanordnung wird *critical path* verwendet.

`-R` Findet Verwendung, um annotierte Assemblerdateien zu verarbeiten. Die ursprüngliche Grammatik mußte um eine Direktive und ein Steuersequenz erweitert werden[1]. Mittels `.cycle <nummer>` werden alle nachfolgenden Instruktionen dem Zyklus `<nummer>` zugeordnet. Die Setzung eines Ganzzahl- oder Gleitkommaregisters kann durch die Sequenz `#<nummer>#` in der Art erweitert werden, daß die Zahl `<nummer>` bei der Registerallokation die Registergruppe der Setzung bestimmt. Mögliche Werte reichen von eins bis vier.

[1]Bei den Verfahren, die sich der linearen Programmierung bedienen, wird – im Zusammenhang mit der Registerallokation – eine temporäre, annotierte Assemblerdatei erzeugt, die `CIS` zur Registerallokation zugeführt wird.

Anhang B

Instruktionssatz des ADSP-2106x

B.1 Überblick

Dieser und der folgende Anhang beschreiben den Instruktionssatz des ADSP-2106x im Detail. Zuerst werden die Instruktionstypen samt Assemblersyntax erklärt. Viele Instruktionstypen enthalten ein Feld zur Spezifizierung einer *compute*-Operation[1]. Da es hierfür eine große Anzahl von Möglichkeiten gibt, werden sie in Anhang C getrennt aufgelistet. (Datentransfers zwischen den MR-Register und dem Registerfile werden als Operationen des Multiplizierers betrachtet.)

Im folgenden wird neben der Syntax auch die Funktion einer jeden Instruktion erklärt. Die Instruktionen werden in vier Kategorien eingeteilt:

1. *compute and move or modify*-Instruktionen, die eine *compute*-Operation parallel zu einem Datentransfer ausführen.

2. *program flow control*-Instruktionen, die verschiedene Arten von Verzweigungen, Prozeduraufrufen und -rücksprüngen und Schleifenoperationen darstellen. Einige können *compute*- und/oder Datentransferoperationen dazu parallel ausführen.

3. *immediate data move*-Instruktionen, die Konstanten, die im Instruktionswort kodiert sind, zur Adressierung verwenden.

4. *miscellaneous*-Instruktionen. Darunter fallen alle übrigen Instruktionen wie Tests, Bitmanipulationen oder NOP (*no operation*).

[1]Dazu gehören Operationen die die ALU, den Multiplizierer oder Shifter verwenden.

Die Instruktionen sind von 1 bis 23 numeriert, einige besitzen mehr als eine syntaktische Form. Viele Instruktionen sind bedingt, was sich an einem vorangestellten „IF“ mit folgender Bedingung erkennen läßt. Hierbei hängt die Ausführung der kompletten Instruktion von der Auswertung der Bedingung ab.

B.2 Instruktionen

Die folgenden Seiten listen die Instruktionen des ADSP-2106x auf. Die *compute*-Operationen, die innerhalb der Instruktionen Verwendung finden, sind in Anhang C beschrieben.

Die folgende Tabelle listet die Notation auf, die in den folgenden Abschnitten verwendet wird.

Notation	*Bedeutung*
GROSS	Explizite Syntax – Assembler-Schlüsselwort
;	Semikolon (Instruktionsterminator)
kursiv	Optionaler Teil der Instruktion
|Option1| |Option2|	Liste von Optionen
compute	ALU-, Multiplizierer- oder Shifter-Multifunktionsoperation
shiftimm	Shifter Immediate-Operation
condition	Statusbedingungen (siehe unten)
termination	Schleifenterminierungs-Bedingungen (siehe unten)
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)	Verzögerte Verzweigung
(LA)	Schleifenunterbrechung (entfernt oberstes Element von Schleifen- und PC-Keller)
(CI)	Lösche Interrupt

B.2.1 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. $$\textit{IF condition compute},\ \left|\begin{array}{l}\text{DM(Ia, Mb)}\\ \text{PM(Ic, Md)}\end{array}\right|\text{=ureg;}$$

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

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

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

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

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

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

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

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.

B.2.2 Program Flow Control-Instruktionen

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

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

    Es wird ein Prozedur- bzw. Interruptrücksprung durchgeführt.

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

B.2.3 Immediate Data Move-Instruktionen

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.

B.2.4 Miscellaneous-Instruktionen

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

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

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

    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 der Programmzähler wird erhöht.

22. IDLE;

    Der Prozessor geht in einen Wartezustand und setzt die Verarbeitung des Maschinenprogramms fort, sobald ein Interrupt eintrifft.

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

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 [Ana91] verwiesen. Die RFRAME-Instruktionen stellt die Werte von Rahmen- und Kellerzeiger wieder her.

B.2.5 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 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)

B.2.6 Universalregister

Register	**Funktion**
Datenregister	
R15-R0	Ganzzahlregister
F15-F0	Gleitkommaregister
Programm-Sequenzer	
PC	Programmzähler (nur lesen)
PCSTK	Oberstes Element des PC-Stapels
PCSTKP	Zeiger auf oberstes Element des PC-Stapels
FADDR	Fetch-Adresse (nur lesen)
DADDR	Decode-Adresse (nur lesen)
LADDR	Schleifenterminierungs-Adresse
CURLCNTR	Aktueller Schleifenzähler
LCNTR	Schleifenzähler für nächste verschachtelte zählergesteuerte Schleife
Daten-Adreßgeneratoren	
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

In Tabelle B.1 sind nochmals wichtige Systemregister und Register der Kontrolleinheit mitsamt Registerbreite und Latenzzeiten aufgeführt.

Registers	*Contents*	*Bits*	*Read Latency*	*Effect Latency*
Kontrolleinheit				
FADDR*	fetch address	24	–	–
DADDR*	decode address	24	–	–
PC*	execute address	24	–	–
PCSTK	top of PC stack	24	0	0
PCSTKP	PC stack pointer	5	1	1
LADDR	top of loop address stack	32	0	0
CURLCNTR	top of loop count stack (current loop count)	32	0	0
LCNTR	loop count for next DO UNTIL loop	32	0	0
Systemregister				
MODE1	mode control bits	32	0	1
MODE2	mode control bits	32	0	1
IRPTL	interrupt latch	32	0	1
IMASK	interrupt mask	32	0	1
IMASKP	interrupt mask pointer (for nestings)	32	1	1
ASTAT	arithmetic status flag	32	0	1
STKY	sticky status flags	32	0	1
USTAT1	user-defined status flags	32	0	0
USTAT2	user-defined status flags	32	0	0

* read-only

Tabelle B.1: Wichtige Systemregister und Register der Kontrolleinheit.

Anhang C

Die Berechnungseinheiten des ADSP-2106x und ihre Instruktionen

C.1 Überblick

Compute-Operation werden in der ALU, dem Multiplizierer oder dem Shifter ausgeführt. Das Instruktionwort beinhaltet ein 23 Bit breites Unterfeld, in dem die auszuführende Operation kodiert wird.

Compute-Operation lassen sich einer der folgenden Gruppen zuordnen:

- Operationen, die eine einzelne Berechnungeinheit beanspruchen.
- Multifunktionsoperationen, die Operationen in ALU und Multiplizierer oder zwei Operationen in der ALU parallel ausführen.
- Operationen, die das MR-Register verwenden, um Gebrauch vom Ganzzahlakkumulator des Multuplizierers zu machen.

C.2 ALU-Operationen

Das 23 Bit breite Unterfeld einer einfachen Berechnungsoperation ist in Abbildung C.1 dargestellt.

Eine Operation, die durch OPCODE spezifiziert wurde, wird in der durch CU bestimmten Berechnungseinheit ausgeführt. X- und y-Operanden sind durch RX und RY indiziert, das

22	21–20	19–12	11–8	7–4	3–0
0	CU	OPCODE	RN	RX	RY

Abbildung C.1: Das 23 Bit breite Unterfeld einer *compute*-Operation.

CU=00	Arithmetisch-logische Einheit
CU=01	Multiplizierer
CU=10	Shifter

Tabelle C.1: Kodierung der Berechnungseinheit im CU-Feld.

Zielregister durch RN. Die Berechnungseinheit ist gemäß Tabelle C.1 kodiert.

Bei einigen Shifter-Operationen wird RN sowohl als Zielregister als auch als Quellregister für einen dritten Operanden verwendet. Die mittels OPCODE auswählbaren Befehle sind in den Tabellen C.2 und C.3 aufgeführt.

Instruktion	*ASTAT Status Flags*								*STKY Status Flags*			
Fixkomma:	AZ	AV	AN	AC	AS	AI	AF	CACC	AUS	AVS	AOS	AIS
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	–	–	–	–	–

Tabelle C.2: Übersicht der verwendbaren ganzzahligen ALU-Operationen inklusive Seiteneffekte auf Statusregister.

C.3 Multiplizierer-Operationen

Ein Überblick über die verfügbaren Operationen ist in Tabelle C.4 gegeben.

Instruktion	*ASTAT Status Flags*								*STKY Status Flags*			
Gleitkomma:	AZ	AV	AN	AC	AS	AI	AF	CACC	AUS	AVS	AOS	AIS
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 Gleitkomma-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

Tabelle C.3: Übersicht der verwendbaren Gleitkomma-ALU-Operationen inklusive Seiteneffekte auf Statusregister.

Instruktion	*ASTAT Flags* MU	MN	MV	MI	*ASTAT Flags* MUS	MOS	MVS	MIS
Festkomma:								
Rn = Rx*Ry (S S F) MRF U U I MRB FR	*	*	*	0	–	**	–	–
Rn = MRF + Rx*Ry (S S F) Rn = MRB U U I MRF = MRF FR MRB = MRB	*	*	*	0	–	**	–	–
Rn = MRF - Rx*Ry (S S F) Rn = MRB U U I MRF = MRF FR MRB = MRB	*	*	*	0	–	**	–	–
Rn = SAT MRF (SI) Rn = SAT MRB (UI) MRF = SAT MRF (SF) MRB = SAT MRB (UF)	*	*	*	0	–	**	–	–
Rn = RND MRF (SF) Rn = RND MRB (UF) MRF = RND MRF MRB = RND MRB	*	*	*	0	–	**	–	–
MRF = 0 MRB	0	0	0	0	–	–	–	–
MRxF = Rn MRxB	0	0	0	0	–	–	–	–
Rn = MRxF MRxB	0	0	0	0	–	–	–	–
Gleitkomma:								
Fn=Fx*Fy	*	*	*	*	**	–	**	**

Rn, Rx, Ry = R0-R15; Fixkomma-Register
Fn, Fx, Fy = F0-F15; Gleitkomma-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

(| • | | • | | • |)

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

Tabelle C.4: Übersicht über die Multiplizierer-Operationen inklusive Seiteneffekte auf Statusregister.

Die Operationen des Multiplizierers können mit einem optionalen Modifikator, der der Operation in Klammern nachgestellt wird, versehen werden. So können x- und y-Operanden als vorzeichenbehaftet (S) oder nicht (U) ausgewiesen werden. Beide können entweder ganzzahlig (I) oder gebrochen rational (F) sein. Durch ein Anfügen von „R“ wird das Ergebnis gerundet. Gültige Kombinationen dieser Modifikatoren sind: (SSI), (SUI), (USI), (UUI), (SSF), (SUF), (USF), (UUF), (SSFR), (SUFR), (USFR) und (UUFR).

C.4 Shifter-Operationen

Die verfügbaren Shifter-Operationen sind in Tabelle C.5 aufgelistet. Die Operationen arbeiten auf 32 Bit der Datenregister. Shifter-Operationen, die zwei Operanden benötigen, können ihren y-Operanden aus dem Registerfile oder der im Instruktionswort kodierten Konstanten beziehen.

C.5 Multifunktions-Operationen

Multifunktions-Operationen gibt es in drei Ausprägungen, die sich hinsichtlich der Form des 23 Bit Unterfeldes unterscheiden:

- Duale Addition/Subtraktion
- Multiplizierer und ALU parallel
- Multiplizierer und duale Addition/Subtraktion parallel

Duale Addition/Subtraktion berechnet die Summe und Differenz zweier Operanden und gibt die Ergebnisse in zwei verschiedenen Registern zurück. Die Syntax ist wie folgt:

Ra=Rx+Ry, Rs=Rx-Ry

Fa=Fx+Fy, Fs=Fx-Fy

Das 23 Bit Berechnungfeld ist in Abbildung C.2 widergegeben.

Ganzahlige Operationen, bei denen Multiplizierer und ALU parallel laufen, bestehen einerseits aus Multiplikation bzw. akkumulierender Multiplikation und eine der folgenden ALU-Operationen: Addition, Subtraktion, Duchschnitt, Zahlenformatkonvertierung oder

	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 Gleitkomma-Register; Gleitkomma-Wort
c = ADSP-2100-kompatible Instruktionen

* wird entsprechend dem Resultat der Operation gesetzt oder gelöscht

Tabelle C.5: Übersicht über die Shifter-Operationen inklusive Seiteneffekt auf Statusregister.

<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">00</td><td colspan="4">x111</td><td colspan="4">RS</td><td colspan="4">RA</td><td colspan="4">RX</td><td colspan="4">RY</td></tr>
</table>

Abbildung C.2: Das 23 Bit breite Unterfeld einer dualen Addition/Subtraktion.

<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>1</td><td colspan="6">OPCODE</td><td colspan="4">RM</td><td colspan="4">RA</td><td colspan="2">RXM</td><td colspan="2">RYM</td><td colspan="2">RXA</td><td colspan="2">RYA</td></tr>
</table>

Abbildung C.3: Das 23 Bit breite Unterfeld bei Parallelschaltung von Multiplizierer und ALU (ganzzahlig).

<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>1</td><td colspan="6">OPCODE</td><td colspan="4">FM</td><td colspan="4">FA</td><td colspan="2">FXM</td><td colspan="2">FYM</td><td colspan="2">FXA</td><td colspan="2">FYA</td></tr>
</table>

Abbildung C.4: Das 23 Bit breite Unterfeld bei Parallelschaltung von Multiplizierer und ALU (Gleitkomma).

Absolutbetrag (ABS), Minimum (MIN) oder Maximum (MAX) für Gleitkommazahlen. Das Format des *compute*-Feldes der Instruktion ist in Abbildung C.3 gezeigt, für Gleitkommazahlen in Abbildung C.4. Eine Auflistung möglicher paralleler Instruktionen ist in Tabelle C.6 enthalten.

Duale Addition/Subtraktion kann mit Multiplikation bzw. akkumulierenden Multiplikation parallel ausgeführt werden. Die Struktur des Unterfeldes ist in Abbildung C.5 für ganzzahlige bzw. Abbildung C.6 für Gleitkomma-Berechnungen aufgezeigt.

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 C.6: Mögliche Kombinationen bei Parallelschaltung von Multiplizierer und ALU.

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 C.5: Das 23 Bit breite Unterfeld bei Parallelschaltung von Multiplizierer und dualer Addition/Subtraktion (ganzzahlig).

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 C.6: Das 23 Bit breite Unterfeld bei Parallelschaltung von Multiplizierer und dualer Addition/Subtraktion (Gleitkomma).

Anhang D

Algorithmen

D.1 Berechnung des Dominatorbaumes

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 $d \; \Delta_d \; n$.

> $D(n_{start}) = \{n_{start}\};$
>
> **forall** $n \in N - \{n_{start}\}$ **do**
>
> $D(n) = N;$
>
> **od**
>
> **while** *Veränderungen in einem* $D(n)$ *auftreten* **do**
>
>   **forall** $n \in N - \{n_{start}\}$ **do**
>
>   *Sei* P *die Menge aller Vorgänger von* n.
> $D(n) = \{n\} \;\cup\; \bigcap_{p \in P} D(p);$ (*)
>
>   **od**
>
> **od**

Algorithmus D.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.

D.2 Berechnung des Kontrollabhängigkeitsgraphen

Sei ein Kontrollflußgraph mit zugehörigem Post-Dominatorbaum gegeben und sei S die Menge aller Kanten (A, B) des Kontrollflußgraphen, so daß B kein Vorfahr von A im Post-Dominatorbaum ist (d.h. B post-dominiert A nicht). Jede dieser Kanten ist mit T oder F markiert. Der Algorithmus untersucht jede Kante (A, B) in S.

Sei L der frühste gemeinsame Vorgänger von A und B im Post-Dominatorbaum. Nach Konstruktion gilt $L \neq B$. Es gilt weiter: $L = A$ oder L ist Vorgänger von A im Post-Dominatorbaum [FOW87]. Es gibt zwei Fälle für L:

1. $L =$ Vorgänger von A: Alle Knoten im Post-Dominatorbaum auf dem Pfad von L nach B, darunter B, aber nicht L, sollten kontrollabhängig von A sein.
2. $L = A$: Alle Knoten im Post-Dominatorbaum 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 Post-Dominatorbaum rückwärts, bis wir A's Vorgänger erreichen (sofern er existiert) und markieren alle Knoten, die vor A's Vorgänger 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 worden sind, sind alle Kontrollabhängigkeiten bestimmt.

Die Laufzeit des Algorithmus beträgt $\mathcal{O}(n^2)$, wobei n die Anzahl der Knoten des Kontrollflußgraphen angibt [FOW87].

D.3 Berechnung längster Wege

Längste Wege im Datenabhängigkeitsgraph werden u.a. von *critical path* und der *highest level first*-Heuristik benötigt. Der Graph wird dem Algorithmus als Adjazenzmatrix zugeführt. Eine Komponente m_{ij} ist hierbei gesetzt (d.h. gleich 1), falls es eine direkte Verbindung von Knoten i nach Knoten j gibt; anderenfalls ist m_{ij} gleich 0. Die Länge des längsten Wege von i nach j ist nach Beendigung des Verfahrens in w_{ij}^n abgelegt. Es wird vorausgesetzt, daß der Graph keine positiven Zyklen enthält.

Der Algorithmus ist in Abbildung D.2 gegeben. Seine Laufzeit ist $\mathcal{O}(n^3)$. Nähere Informationen zum Floyd-Warshall-Algorithmus zur Berechnung längster Wege sind [CLR90] zu entnehmen.

$D^0 = M$

for k=1 **to** n **do**

  for i=1 **to** n **do**

    for j=1 **to** n **do**

      $D^k_{ij} = \min\left(D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\right)$

    od

  od

od

return D^n

Algorithmus D.2: Floyd-Warshall-Algorithmus zur Berechnung längster Wege in einem Graphen bei gegebener nxn-Adjazenzmatrix M.

Anhang E

Testprogramme

Auf den folgenden Seiten folgt eine dreispaltige Auflistung der verwendeten Testprogramme. Die linke Spalte enthält das Programm in seiner sequentiellen Form, die mittlere das Ergebnis der Instruktionsanordnung mittels *list scheduling* unter Verwendung der *highest level first*-Heuristik. In der rechten Spalte befindet sich die optimale Anordnung, wie sie in [Käs97] zu finden ist. Eine Beschreibung der einzelnen Programme ist in Abschnitt 7.5 gegeben.

E.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;
   f12=f0*f4, f8=f8+f12;
   f0=dm(i0,m0);
macs:
   f4=pm(i8,m8);
   f12=f0*f4, f8=f8+f12;
   rts, f0=f8+f12;
.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;
```

E.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:
   f12=0.000000;
   f8=0.000000;
   f2=dm(i0,m1), f4=pm(i8,m1), f8=f8+f12;
   f3=dm(i0,m1), f4=pm(i8,m8), f12=f2*f4;
   f12=f3*f4, dm(i1,m1)=f3, f8=f8+f12;
   f4=pm(i8,m8), f8=f8+f12;
   f2=dm(i0,m1), f4=pm(i8,m8), f12=f2*f4;
   f8=f8+f12, dm(i1,m1)=f8;
   f4=pm(i8,m8), f12=f3*f4;
.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;
```

E.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:
   i10=0;
   l10=0;
   f15=0.000000;
   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;
   f12=f0*f5, f8=f8-f13;
inner:
   f4=pm(i8,m8);
   f13=f0*f4, f9=f9+f12;
   f8=f8-f13, modify(i10,m9);
   dm(i2,m1)=f9;
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;
```

E.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:
   modify(i7,-3);
   f13=f8;
   f1=dm(100);
   f5=dm(200);
   f4=f4+f13, i4=r12;
   f9=1073741824.000000;
   f2=recips f5;
   f8=f1*f4;
   f4=f8+f13, f12=f2*f5;
   f13=f1*f4;
   f2=f9-f12;
   f4=f8+f13;
   f12=f2*f12;
   f4=f2*f4;
   f2=f9-f12, f4=f2*f4;
   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;
```

```
.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;
```

E.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;
   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;
```

```
.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;
   r1=pm(i8,m15), r0=dm(i0,m0);
   r1=r1+1, i9=r2;
   r3=pm(i9,m15);
   pm(i8,m15)=r1, r3=r3+1;
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, r2=l1;
   r3=pm(i9,m15), r2=r2-1;
   r3=r3+1, pm(i8,m15)=r1;
   pm(i9,m15)=r3;
   i8=b8;
   i9=b9;
   r0=pm(i8,m8);
   r1=pm(i9,m8);
   lcntr=r2, DO combine until lce;
   r0=pm(i8,m8);
   r2=r0+r1;
   dm(i1,m0)=r2;
combine:
   r1=pm(i9,m8);
   r2=r0+r1;
   rts, dm(i1,m0)=r2;
.endseg;
```

```
.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;
```

E.6 conv3x3.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;
   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;
```

```
.segment /pm pm_code;
    f12=f2*f6, f7=dm(i0,m1), f0=pm(i8,m8), f8=f8+f12;
    f4=dm(i0,m0), f8=f8+f12, f1=pm(i8,m8), f13=f1*f6;
    f13=f2*f7, f5=dm(i0,m0), f9=f9+f13;
    f12=f0*f4, f9=f9+f13, f6=dm(i0,m0), f2=pm(i8,m8);
    f8=f8+f12, f7=dm(i0,m1), f0=pm(i8,m8), f13=f0*f5;
    f9=f9+f13, f4=dm(i0,m0), f12=f1*f5;
    f8=f8+f12, f5=dm(i0,m0), f1=pm(i8,m8), f13=f1*f6;
    f12=f2*f6, f9=f9+f13, f6=dm(i0,m0);
    f8=f8+f12, f2=pm(i8,m8), f7=dm(i0,m2), f13=f2*f7;
    f9=f9+f13, f12=f0*f4;
    f8=f8+f12, f13=f0*f5, modify(i0,m3);
    f9=f9+f13, f12=f1*f5;
    f13=f1*f6, f8=f8+f12;
    f9=f9+f13, f12=f2*f6;
    f13=f2*f7, f8=f8+f12;
    f9=f9+f13, dm(i1,m0)=f8;
    dm(i1,m0)=f9;
.endseg;
```

```
.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;
```

Anhang F

Glossar

Backend Letzter Teil eines Übersetzers, der üblicherweise in Frontend (syntaktische und semantische Analyse), Middleend (Optimierungen) und Backend (Codeerzeugung) aufgeteilt werden kann.

CISC (*complex instruction set computer*) bieten komplexe Befehle zur Unterstützung des Zugriffs auf Datenstrukturen und der Prozedurorganisation. Weiterhin bieten sie viele und komplexe Adressierungsarten, verschiedene Varianten von Operationen für Operanden unterschiedlicher Länge und in der Regel wenig Register.

DMA (*direct memory access*) Durch diese Architektur können Maschinenkomponenten auf den Speicher zugreifen, ohne dafür die Dienste des Kernprozessors in Anspruch nehmen zu müssen.

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].

ILP (*integer linear programming*) Lineare Programme, deren Lösungsvariablen ganzzahlig sind. Siehe hierzu [Käs97].

LRU (*least recently used*) Stellt eine Strategie dar, bei der solche Einträge im Cache ersetzt werden, die am wenigsten genutzt wurden.

Mikroinstruktion In einer Mikroinstruktion werden mehrere Mikrooperationen zusammengefaßt. Sie stellt ein Befehlswort dar, welches beim ADSP-2106x eine Länge von 48 Bit hat.

MILP (*mixed integer linear programming*) Lineare Programme, deren Lösungsvariablen sowohl ganzzahling als auch reell sein können. Siehe hierzu [Käs97].

MIPS (*million instructions per second*) Für ein gegebenes Programm kann ein in MIPS gemessener Leistungswert nach folgender Gleichung ermittelt werden:

$$\text{MIPS} = \frac{\text{Befehlsanzahl}}{\text{Ausführungszeit in Sekunden} \cdot 10^6}$$

Für eine genauere Beschreibung insbesondere auch der mit diesem Maß verbundenen Probleme sei auf [HP94] verwiesen.

Mikrooperation Eine elementare Operation auf Prozessorniveau, wie z. Bsp. Addition zweier Register, Laden aus Speicherbereichen. Mehrere Mikrooperationen können als Mikroinstruktion zusammengefaßt parallel ausgeführt werden, sofern dies die zugrundeliegende Hardware erlaubt.

Registergruppe Die Register des ADSP-2106x sind logisch in Vierergruppen zusammengefaßt. Bei parallelen Operationen benötigen die verschiedenen funktionalen Einheiten bestimmte Registergruppen (vergleiche Kapitel 2).

RISC (*reduced instruction set computer*) Operationen können meist nur auf Registern ausgeführt werden, die Operanden müssen zuvor aus dem Speicher geladen werden (*load/store*-Architektur). Durch wenige Adressierungsarten wird die Kontrollogik verkleinert, die meist fest verdrahtet ist. Die Registermenge ist meist groß (> 32 Register).

Scheduler ist eine Bezeichnung für den Übersetzerteil, der für die Instruktionsanordnung zuständig ist.

VCG (*visualization tool for compiler graphs*) Ein Werkzeug zum Anzeigen von Graphen, die mittels der VCG-Spezifikation beschrieben sind.

VLIW (*very large instruction word*) Durch die Größe des Intruktionsworts (48 Bit und mehr) lassen sich mehrere funktionale Einheiten, wie etwa Addierer, Shifter u.ä. zugleich ansteuern.

Literaturverzeichnis

[AN88] Aiken, Alexander und Alexandru Nicolau: *A Development Environment for Horizontal Microcode.* IEEE Transactions on Software Engineering, 14(5):584–593, 1988.

[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, Alfred V., Ravi Sethi und Jeffrey D. Ullman: *Compilerbau, Teil 2.* Addison-Wesley, 1988.

[Bas95] Bashford, Steven: *Code Generation Techniques for Irregular Architektures.* Technischer Bericht 596, Lehrstuhl Informatik XII, University of Dortmund, 1995.

[BL93] Ball, Thomas und James R. Larus: *Branch Prediction For Free.* In: *Proceedings of the Conference on Programming Language Design and Implementation*, Seiten 300–313. ACM Press, Juni 1993.

[Bra91] Bradlee, David Gordon: *Retargetable Instruction Scheduling for Pipelined Processors.* Technischer Bericht 91-08-07, Department of Computer Science and Engeneering, FR-35, University of Washington, 1991.

[Cas95] Castelli, Gianluigi: *Guest Editor's Introduction: The Seemingly Unlimited Market for Microcontroller-Based Embedded Systems.* IEEE Micro, 15(5):6–8, Oktober 1995.

[CLR90] Cormen, Thomas H., Charles E. Leiserson und Ronald L. Rivest: *Introduction To Algorithms.* MIT Press, 1990.

[DT76] Dasgupta, S. und J. Tartar: *The Identification of Maximal Parallelism in straight-line Microprograms.* IEEE Transactions on Computers, 25:986–992, 1976.

[Fis81] Fisher, A.: *Trace Scheduling: A Technique for Global Microcode Compaction.* IEEE Transactions on Computers, 30(7):478–490, 1981.

[FOW87] Ferrante, Jeanne, Karl J. Ottenstein und Joe D. Warren: *The Program Dependence Graph and its Use in Optimization.* ACM Transactions on Programming Languages and Systems, 9(3):310–349, Juli 1987.

[GJ79] Garey, Michael R. und David S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[GO95] Goodby, L. und A. Orailoglu: *Towards 100% Testable FIR Digital Filters.* International Test Conference. Washington D.C., USA, 1995. IEEE Computer, Seiten 394–402, 1995.

[GS90] Gupta, Rajiv und Mary Lou Soffa: *Region Scheduling: An Approach for Detecting and Redistributing Parallelism.* IEEE Transactions on Software Engineering, 16(4):421–431, 1990.

[Her80] Herman, G.T.: *Image Reconstruction From Projections. The Fundamentals of Computerized Tomography.* Academic Press, New York, 1980.

[HP94] Hennessy, John L. und David A. Patterson: *Rechnerarchitektur.* Vieweg, 1994.

[Käs97] Kästner, Daniel: *Instruktionsanordnung und Registerallokation auf der Basis ganzzahliger linearer Programmierung für den digitalen Signalprozessor ADSP-2106x.* Diplomarbeit, Universität des Saarlandes, 1997.

[LDSM80] Landskov, David, Scott Davidson, Bruce Shriver und Patrick W. Mallet: *Local Microcode Compaction Techniques.* ACM Computing Surveys, 12(3):261–294, 1980.

[Lüc80] Lücker, Raimund: *Grundlagen digitaler Filter. Einführung in die Theorie linearer zeitdiskreter Systeme und Netzwerke*, Band 7. Springer Verlag, 1980.

[Nic85] Nicolau, Alexandru: *Uniform Parallelism Exploitation in Ordinary Programs.* In: *International Conference on Parallel Processing*, Seiten 614–618. IEEE Computer Society Press, August 1985.

[Now90] Nowinski, W. L.: *Parallel Implementation of the Convolution Method in Image Reconstruction.* Springer Verlag, 1990. CONPAR90 - VAPP IV. pp. 355-364, Zurich, Switzerland, 1990.

[PLNG87] POTASMAN, RONI, JOSEPH LIS, ALEXANDRU NICOLAU und DANIEL GAJSKI: *Percolation Based Synthesis.* 27th ACM/IEEE Design Automation Conference, 1987.

[SCL95] SAGHIR, MAZEN A. R., PAUL CHOW und CORINNA G. LEE: *Exploiting Dual Data-Memory Banks in Digital Signal Processors.* http://www.eecg.toronto.edu/˜ saghir/papers/asplos7.ps, 1995.

[WM97] WILHELM, REINHARD und DIETER MAURER: *Übersetzerbau. Theorie, Konstruktion, Generierung; zweite, überarbeitete und erweiterte Auflage.* Springer Verlag, 1997.

[ZSWS95] ZIVOJNOVIC, VOJIN, HARALD SCHRAUT, M. WILLEMS und R. SCHOENEN: *DSPs, GPPs, and Multimedia Applications - An Evaluation Using DSPstone.* Proceedings of the International Conference on Signal Processing Applications and Technology, Seiten 1779–1783, Oktober 1995.

Index