

Ein Verfahren
zur Portabilität
algorithmischer
Laufzeitprogramme

EIN VERFAHREN ZUR ERSTELLUNG PORTABLER ALGORITHMISCHER LAUFZEITPROGRAMME

Dem Fachbereich Ingenieurwissenschaften der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Werner Lindstedt

Erlangen, 1981

Zusammenfassung

Die Arbeit stellt ein Konzept vor, Laufzeitprogramme für algorithmische Probleme, wie Konkatenation von Zeichenketten, Multiplikation von Matrizen und Gleitpunktarithmetik portabel zu erstellen. Sie ergänzt damit anlagenunabhängige Übersetzungssysteme, die bisher nur Compiler, generierte Codegeneratoren und graphische Treiber und Pakete für Standard-E/A enthalten.

Das Verfahren beruht darauf, daß die Laufzeitprogramme in einer assemblynahen Sprache für eine abstrakte Dreiadreßmaschine formuliert und maschinell in mehreren Stufen auf die gewünschte Zielmaschine umgesetzt werden. Um dabei möglichst effizienten Zielcode zu erhalten, ist es nötig, bereits auf der Ebene der Dreiadreßprogramme auf Eigenschaften der Zielmaschine einzugehen. Die Dreiadreßprogramme enthalten deshalb an manchen Stellen formale Zielmaschinenparameter und alternative Codestücke. Die Anpassung an die Zielmaschine nimmt ein Generierprogramm vor, das die aktuellen Werte der Zielmaschine einsetzt und die entsprechenden Codealternativen auswählt. Die so erzeugten Programme werden - je nach Zielvorgabe - auf Assemblerprogramme für eine abstrakte Ein- oder Zweiadreßmaschine abgebildet, die in Registeranzahl und -eigenschaften, Befehlssatz und Adressierungsarten mit der Zielmaschine bereits weitgehend übereinstimmt. Die Schnittstellen sind so konzipiert, daß sie leicht an verschiedene Codegeneratoren angepaßt werden können. Bei allen Umsetzstufen gibt es Möglichkeiten zur Codeoptimierung. Erfahrungen mit diesem Verfahren werden im Rahmen einer Pilotimplementation gemacht und diskutiert.

Die Arbeit entstand im Rahmen des Forschungsvorhabens P4.2/3 des Bundesministeriums für Forschung und Technologie, Projekt Prozeßlenkung durch Datenverarbeitung.

Inhaltsübersicht

| Seite | mnemotechn. Kapitel-Nr. | |
|-------|----------------------------|---|
| 4 | EINL- | Einleitung |
| 6 | ZIEL- | Problemstellung und Zielsetzung |
| 6 | ZIEL-1. | Aus historischer Entwicklung entstandene Aufgabenstellung |
| 10 | ZIEL-2. | Forderungen an algorithmische Laufzeitprogramme |
| 12 | ZIEL-3. | Vergleichende Untersuchung verschiedener Kleinrechner |
| 21 | LWEG- | Lösungsweg |
| 21 | LWEG-1. | Entwicklung einer maschinenunabhängigen Notation |
| 28 | LWEG-2. | Zielmaschinenabhängigkeit der Algorithmen |
| 31 | LWEG-3. | Konzept der Umsetzung |
| 37 | LWEG-4. | Portable Rechenergebnisse |
| 39 | LOES- | Lösung |
| 39 | LOES-1. | Abstrakte Maschinen |
| 39 | LOES-1.1 | Abstrakte Dreiadreßmaschine |
| 43 | LOES-1.2 | Aufbau eines abstrakten Laufzeitprogramms |
| 45 | LOES-1.3 | Unterprogramme |
| 47 | LOES-1.4 | Abstrakte Ein- und Zweiadreßmaschine |
| 50 | LOES-2 | Beschreibung der Umsetzung |
| 50 | LOES-2.1 | Implementationstechnik |
| 51 | LOES-2.2 | Umsetzverfahren |
| 55 | LOES-2.3 | Registerverwaltung |
| 61 | LOES-2.4 | Simulation von Adressierungsarten |
| 67 | LOES-2.5 | Simulation von Operatoren |
| 69 | BSPL- | Beispiel |
| 69 | BSPL-1. | Generierparameter |
| 72 | BSPL-2. | Programm mit Zwischenformen |
| 77 | BSPL-3. | Diskussion |

| Seite | mnemotechn. Kapitel-Nr. | |
|-------|----------------------------|---|
| 78 | ERGN- | Ergebnisse |
| 78 | ERGN-1. | Anwendungen |
| 79 | ERGN-2. | Höhere Kontrollstrukturen |
| 82 | ERGN-3. | Ausblick |
| 83 | A- | Anhang |
| 83 | A-1. | Ergänzende Beschreibung der abstrakten Maschine |
| 83 | A-1.1 | Operatoren, Ergebnisanzeigen |
| 87 | A-1.2 | Sprungbefehle, Sprungbedingungen |
| 92 | A-1.3 | Transferbefehl, Sonderbefehle |
| 94 | A-1.4 | Adressierungsarten |
| 96 | A-2. | Interne Darstellungen |
| 98 | A-3. | Dialog zum Erfassen der Generierparameter |
| 102 | A-4. | Struktogramme |
| 105 | A-5. | Listings |
| 116 | LITE- | Literaturverzeichnis |

EINL - Einleitung

Die letzte Phase eines Compilers ist die Erzeugung des Maschinencodes. Codegeneratoren delegieren einen Teil ihrer Aufgabe und rufen Unterprogramme einer Laufzeitbibliothek auf. Die benötigten Programme hängen von der höheren Programmiersprache und der Maschinenhardware ab. Arithmetische Probleme, wie sie in höheren Programmiersprachen, wie ALGOL usw. auftreten, werden auf fortschrittlichen Rechnern in Hardware gelöst. Für spezielle, sprachabhängige Funktionen, wie z.B. die Kettenarithmetik aus PEARL [DIN78], Matrixoperationen aus APL [IVE62] oder Mengenoperationen aus SETL [SCH79], sind Laufzeitprogramme weiterhin unerlässlich. Sie machen Aufwand, geben vielen Programmierern keine Befriedigung, fallen aber unangenehm auf, wenn sie nicht mit der nötigen Sorgfalt erstellt und nicht auf optimales Zeitverhalten ausgelegt sind. Da liegt die Idee nahe, lieber einmal einen gut durchdachten Entwurf anzufertigen, als immer wieder neue, wobei die anfängliche Effizienz unter Termindruck bald auf der Strecke bleibt.

Ziel der Arbeit ist es, angesichts der Vielzahl verschiedener Rechner eine Technik zu entwickeln, um den Erstellungsaufwand für algorithmische Laufzeitprogramme auf ein Minimum zu senken, ohne sofort hoffnungslos ineffizient zu werden. Die Schnittstelle zum Codegenerator und Betriebssystem wird so allgemein gehalten wie irgend möglich. Damit ordnen sich die portablen Laufzeitprogramme Implementationszielen der höheren Ebene unter. Das bedingt, daß Probleme der Speichervergabe, Ununterbrechbarkeit, Wiedereintrittsfähigkeit usw. nicht im portablen Programm gelöst werden, daß aber Schnittstellen nach oben vorhanden sein müssen.

Das Ergebnis ist ein Werkzeug für den Implementierer, der sich nicht scheut, maschinelle Hilfsmittel einzusetzen. Die Anpassung an seine Maschine erfolgt durch - im Dialog mit dem Rechner - zu erstellende Listen von Generierparametern, die von einzelnen Konstanten bis zu kurzen Befehlsfolgen reichen. Die optimale Anpassung zwingt manchmal dazu, für ein Problem mehrere Algorithmen bereitzustellen, da ein Algorithmus, der sich bei einer Maschine auf guten Code abbilden läßt, auf einer anderen

Maschine zu Platz- und/oder Laufzeitverschwendungen führt, selbst wenn die Umsetzung auf Zielmaschinencode optimal durchgeführt wird.

Die Lösung wurde darin gefunden, eine dem Problem angepaßte Assemblersprache einer abstrakten Maschine zu konstruieren, in der die Laufzeitprogramme formuliert werden. Sie ist im Niveau so niedrig, daß sie sich auf effizienten Code verschiedener Zielmaschinen abbilden läßt. Das erfolgt in mehreren Zwischenstufen, wobei jeweils eine weitere Anpassung an die Zielmaschine erfolgt. Besonderes Augenmerk wurde dabei auf eine gute Ausnutzung der Register und der vorhandenen Adressierungsarten gelegt.

ZIEL. - Problemstellung und Zielsetzung

ZIEL - 1. Aus historischer Entwicklung entstandene Aufgabenstellung

Im Juni 1978 wurde für die Echtzeit-Programmiersprache PEARL - PROZESS AND EXPERIMENT AUTOMATION REALTIME LANGUAGE - ein DIN-Normentwurf der Öffentlichkeit vorgestellt [DIN78]. Verschiedene Firmen entwickelten Compiler für diese Sprache. Für eine Vorstufe dieser Programmiersprache, PEARL Stufe 1, [GRU76] läuft bereits seit Sommer 1975 ein Compilersystem. Es wurde am 9.2.1976 der Öffentlichkeit vorgestellt. Den Compiler entwickelte die Arbeitsgemeinschaft ASME ¹⁾, bestehend aus zwei Industriefirmen und drei Hochschulinstituten. Sie hatte sich zum Ziel gesetzt, einen portablen Compiler zu erstellen, d.h. einen Compiler, der mit wenig Umstellungsaufwand auf verschiedenen Rechnern lauffähig ist. Zur Demonstration der Portabilität standen eine AEG 60/10, AEG 60/50, SIEMENS 330, SIEMENS 4004, SIEMENS 404/3 und die erwähnte SIEMENS 306 zur Verfügung.

Die Portabilität des ASME-Compilers ließ sich durch eine Zweiteilung in einen sog. Compileroberteil und einen Codeerzeuger erreichen. Der Oberteil wurde in einer Sprache, die auf jeder der geplanten Zielmaschinen verfügbar ist, formuliert. Aus Gründen der weiten Verbreitung wurde FORTRAN gewählt, obwohl diese Sprache nicht gerade für den Zweck des Compilerschreibens entwickelt worden war. Der Compileroberteil setzt Code der Zwischensprache CIMIC-1 ab [MUE76]. Für jede Zielmaschine mußte nun ein Codeerzeuger geschrieben werden, um CIMIC-1 in die Assemblersprache des Zielrechners umzusetzen. Das Werkzeug zur Erstellung des Codeerzeugers war der Makrogenerator Stage II [WAI70]. Die leichtere Testbarkeit des erzeugten Codes begründete den Umweg über den Assembler. Durch die Zweiteilung des Compilers konnte ein Portabilitätsgrad von etwa 50% des gesamten Compilersystems erzielt werden. In einer zweiten Ausbaustufe wurde FORTRAN und CIMIC-1 durch eine Sprache CIMIC-P [EIC77] ersetzt. Der Compileroberteil wurde in einer

1) ASME: Arbeitsgemeinschaft Stuttgart, München, Erlangen bestehend aus:
 Institut für Regelungstechnik und Prozeßautomatisierung
 Univ. Stuttgart - Prof. Dr. Lauber
 Institut für Verfahrenstechnik und Dampfkesselwesen,
 Univ. Stuttgart - Prof. Dr. Quack
 Electronic System Gesellschaft (ESG) München
 Gesellschaft für Prozeßprogrammierung (GPP) München
 Physikalisches Institut III, Univ. Erlangen - Prof. Dr. Fiebiger

Untermenge (mit Ausnahmen) von CIMIC-P geschrieben. Zwar entledigte man sich dadurch der Abhängigkeit von FORTRAN. Ein Gewinn an Portabilität wurde aber kaum erzielt, da wie bisher Codegeneratoren für jede Zielmaschine erstellt werden mußten. Deshalb entwickelte Pelz [PEL81] einen Generator, der aus einer geeigneten Parametrisierung einer Zielmaschine selbständig einen Codeerzeuger generiert. Dieser Codeerzeugergenerator bringt eine beträchtliche Steigerung des Portabilitätsgrades, doch bleibt immer noch eine wesentliche Lücke. Codeerzeuger, generierte oder von Hand erstellte, setzen aus Gründen der Speicherplatzeffektivität häufig Unterprogrammeinsprünge in sog. Laufzeitroutinen und die dazugehörigen Parameterversorgungsbefehle ab. Die im Laufzeitpaket zusammengefaßten Unterprogramme sollten in bindefähigem Maschinencode vorliegen und müssen an das vom Codeerzeuger erstellte Programm gebunden werden. Steht in Ausnahmefällen kein Binder zur Verfügung, so können die Laufzeitroutinen zusammen mit dem vom Codegenerator erzeugten Assemblercode neu übersetzt werden.

Die Laufzeitprogramme lassen sich in drei Gruppen einteilen:

1. Betriebssystemroutinen
2. Ein- Ausgabe und Formatierungsroutinen
 - a) für Standardperipherie
 - b) für graphische Ein-Ausgabe
 - c) für Prozeßperipherie
3. Algorithmische Laufzeitroutinen

Für die Prozeßprogrammiersprache PEARL entwickelte Roessler [ROE78] ein portables Betriebssystem, das sich entweder auf vorhandene Betriebssysteme oder auf eine "nackte" Maschine aufsetzen läßt. Die Portabilität wurde von Rössler am Beispiel des Z80, einer Siemens 306 und von Fleischmann an einer SIEMENS 310 [FLE79] demonstriert.

Für die Standardperipherie formulierte Lampe [LAM78] Laufzeitroutinen in PEARL, die einen aufbereiteten Puffer an nachgeschaltete Peripherietreiber übergeben. Die etwas mangelhafte Effizienz bezüglich Laufzeit und Speicherplatz läßt sich dabei noch vermindern, wenn einige wenige, genau definierte Routinen statt in PEARL in Assembler erstellt werden.

Zur Portabilität graphischer Ein-Ausgaberoutinen klassifizierte Prester [PRE81] am Markt befindliche graphische Peripheriegeräte, wie Plotter und Bildschirmgeräte und erstellte ein in PEARL geschriebenes Skelettprogramm und einen Stapel von Routinen zum Aufbau der verschiedenartigsten graphischen Gerätetreiber. Ein von Geräteparametern gesteuertes Generatorprogramm ergänzt das Programmgerippe durch Einfügen der geeigneten Routinen und durch Einsetzen gerätespezifischer Konstanten (in Skelettprogramm und Routinen) zu einem PEARL Prozedurmodul, dem speziellen Gerätetreiber. Diese generierten Treiber benötigen zur Realisierung der Schnittstelle zur Hardware noch eine Assembler-Schnittstelle, das die Geräte ansteuert. Der Aufwand dafür ist jedoch vernachlässigbar gegenüber dem der Neuerstellung eines graphischen Gerätetreibers.

Von Arbeiten zur Erstellung portabler Routinen zur Steuerung von Prozeßperipherie ist dem Verfasser nichts bekannt. Der Aufwand dürfte dem Erstellen von maschinenabhängigen Treibern gleichkommen.

Das Thema der vorliegenden Arbeit ist die

Portabilität algorithmischer Laufzeitprogramme.

Bei algorithmischen Laufzeitprogrammen lassen sich folgende Unterscheidungen treffen:

1. Arithmetische Routinen, für die bereits Hardwarerealisierungen existieren
 - 1.1. Mathematische Standardfunktionen
mit komplexen und reellen Zahlen als Argument
Beispiele: Sinus, Tangens
 - 1.2. Arithmetische Verknüpfungen und Typumwandlungen
Beispiele: Subtraktion zweier Gleitpunktzahlen, Multiplikation ganzer Zahlen, Umwandlung einer Gleitpunktzahl in eine ganze Zahl
2. Sprachabhängige Routinen, für die Hardwarelösungen derzeit noch nicht erhältlich sind
 - 2.1. Arithmetische Verknüpfungen von speziellen Operandentypen
Beispiele: Addition einer Zeitdauer zu einer Uhrzeit
 - 2.2. Bitketten- und Zeichenkettenroutinen
Beispiele: Zeichenselektion, Bitkettenkonkatenation
3. Operationen auf zusammengesetzten Datenstrukturen
Beispiele: Selektion eines Strukturelements, Multiplikation zweier Felder, Transfer von Strukturen, Durchschnittsbildung zweier Mengen.

Arithmetische Routinen für Standard-Datentypen werden in Zukunft von immer geringerer Bedeutung sein, da Ganzzahl- und Gleitpunktarithmetikoperatoren bereits heute auf vielen Kleinrechnern im Befehlssatz enthalten sind. Außerdem bietet der Markt Spezialhardwarebausteine dafür und zusätzlich für Standardfunktionen und Typwandlungen an, die über Ein- und Ausgabebefehle angesprochen werden z.B. [AMD79]. (Jedoch bereitet die Spezifikation des Codegenerators für PEARL auf einem Kontron-System [TRA81] Schwierigkeiten wegen der mangelhaften Synchronisationsmöglichkeiten).

Geht man davon aus, daß zur Darstellung der Operanden in arithmetischen Anweisungen eine Mindestanzahl von Bits erforderlich ist, so können auf realen Maschinen dazu ein, zwei oder mehr Maschinenworte erforderlich sein. Somit unterscheiden sich arithmetische Routinen noch in der Anzahl der Worte pro Operand. Zwar lassen sich aus dem Fall der n -fachen Maschinenwortanzahl die Spezialfälle $n = 1$ oder $n = 2$ herleiten, doch werden die Algorithmen, die auf n -fache Wortanzahl ausgelegt sind, bei dieser Spezialisierung an Effektivität verlieren. Für Mehrwortarithmetik werden in der Literatur bereits Programme, die in einer höheren Programmiersprache geschrieben sind, angeboten [HIL68, BR076]. Sie erfüllen zwar die Forderung nach Portabilität, doch Forderungen hinsichtlich Effizienz können sie nicht erfüllen.

ZIEL-2. Forderungen an algorithmische Laufzeitprogramme

Nicht an alle Laufzeitroutinen werden die gleichen Anforderungen gestellt. Bei den Ein-Ausgaberroutinen für (im Vergleich zur Prozessorgeschwindigkeit) relativ langsame Standardperipherie müssen keine großen Anforderungen an die Laufzeit gestellt werden, besonders dann nicht, wenn diese Routinen auf einem eigenen Ein-Ausgaberechner ablaufen. Graphische Ein-Ausgaberroutinen dürfen beim Betrachter eines Bildes am Monitor nicht den Eindruck eines "langsamen Bildschirms" aufkommen lassen.

Auf eine gute Laufzeit muß also geachtet werden. Besonders laufzeiteffizient müssen in einem Prozeßprogrammiersystem Betriebssystemroutinen, Prozeßperipherie-Ein-Ausgabeprogramme und arithmetische Laufzeitprogramme sein. Bei Programmen, die beispielsweise umfangreiche numerische Bearbeitung von Meßdaten erfordern, wie bei der digitalen Nachbildung analoger Regelkreise, muß die Rechengeschwindigkeit hoch genug sein, um mit der Umwelt Schritt halten zu können.

Es wäre zu begrüßen, wenn der Einsatz portabler Laufzeitprogramme portable Rechenergebnisse zur Folge hätte. Unterschiedliche Wortlänge, unterschiedliche Darstellung negativer Zahlen im Zweier- oder Einerkomplement und unterschiedliche Rundung softwaremäßig auszugleichen, steht der Erzeugung effizienten Codes entgegen. Der Wunsch nach portablen Rechenergebnissen muß in den meisten Fällen gegenüber dem Wunsch nach effizientem Code zurückstehen.

Mit der immer größeren Verbreitung von Klein- und Kleinstrechnern wächst das Verlangen nach geeigneten Programmierungshilfen. Das Ziel der Erlanger Gruppe der ASME ist daher, PEARL speziell auf solchen Rechnern zum Einsatz zu bringen. Zwar werden Speicher immer billiger, doch besteht bei Kleinstrechnern noch oft eine Grenze im Speicherausbau, die durch das Adressiervolumen gesetzt ist. Bei der weiten Verbreitung der 16 Bit Rechner können direkt maximal 64 k Worte adressiert werden. Obwohl es [FAE78] seit wenigen Jahren bei Mikrocomputern die Tendenz zur Vergrößerung des logischen und physikalischen Adressraumes gibt, sind Mechanismen

in Hard- und Software, wie beispielsweise das "Memory Management" für die PDP 11/35 & 11/40 [PDP73] auf vielen Kleinrechnern nicht vorhanden. Daraus ergibt sich neben der Forderung nach geringer Laufzeit, die nach wenig Speicherplatz.

Eine Erfahrung in der Datenverarbeitung besagt, daß sich beide Forderungen ab einer bestimmten Grenze gegenseitig ausschließen. Ebenso steht die Forderung nach einem hohen Portabilitätsgrad einer optimalen Laufzeit- und Speicherplatzeffizienz entgegen.

Es muß nach einem Kompromiß gesucht werden, wobei aber die Portabilität besondere Beachtung finden soll.

Zusätzlich gibt es noch eine Programmiereffizienz, die bei portablen Programmen nicht ins Gewicht fällt. Wenn das Programmpaket oft portiert wird, spielt die Zeit, die zur Erstellung von Laufzeitprogrammen benötigt wird, nicht mehr die Hauptrolle, weil der häufige Einsatz einmal geschriebener portabler Laufzeitprogramme bei jedem Portierungsvorgang eine, gegenüber einer Neuformulierung, beträchtliche Zeitersparnis einbringt.

Aus der Forderung nach einem geringen Speicherbedarf ergibt sich automatisch die Forderung, daß Laufzeitprogramme wiedereintrittsfähig sein müssen. Es ist nicht tragbar, daß Laufzeitprogramme bei Echtzeitsystemen an jede Task gebunden werden. (Abschreckende Beispiele könnten genannt werden.) Wenn dies zur Verbesserung des Zeitverhaltens, zumindest für einzelne Laufzeitprogramme, gemacht werden muß, so ist dies als Ausnahme anzusehen, aber die Protabilitätstechnik sollte trotzdem den speziellen Wunsch nach nicht-reentrantfähigem Code berücksichtigen können. Bei manchen Maschinen gibt es eine einfache Möglichkeit, verschieblichen Code zu erzeugen. Diese Möglichkeit ausnützen zu können, erscheint nicht nötig, da ein verschiebender Lader vorausgesetzt werden kann.

Nachdem die Forderungen an algorithmische Laufzeitprogramme zusammengestellt sind, muß geklärt werden, auf welche Maschinen die Laufzeitprogramme portiert werden sollen und können, bzw. welche Voraussetzungen bezüglich der Zielmaschine erfüllt sein müssen. Dazu ist eine vergleichende Untersuchung von einigen am Markt befindlichen Maschinen nötig, durch die in etwa das Spektrum der berücksichtigten Zielmaschinen abgesteckt ist.

ZIEL-3. Vergleichende Untersuchung einiger Kleinrechner

Der Vergleich soll die großen Unterschiede zwischen verschiedenen Rechnern einerseits und doch vorhandene Gemeinsamkeiten andererseits aufzeigen, so daß nach erstem Zurückschrecken doch die Aufgabe, portable Programme zu erstellen, mit Optimismus angegangen werden kann.

Verglichen werden einige am Markt befindliche Kleinrechner, die für PEARL-Implementationen in Frage kommen, sowie die aus historischen Gründen interessante SIEMENS 306.

Im Vergleich werden Kriterien ausgewertet, wie sie auch Pelz und Siller [PEL78], Holleczeck [HOL76], Poole und Waite [POO73] angeben. Ein wesentliches Merkmal von am Markt befindlichen Kleinrechnern ist die Registerarchitektur. Maschinen mit einem Hardwarekeller, bei denen bei Rechenoperationen ein oder sogar mehrere Operanden in den obersten Kellerzellen stehen, sind in der Theorie und bei höheren Zwischensprachen (z.B. CIMIC C) besonders zur Abarbeitung langer arithmetischer Ausdrücke zwar sehr geeignet, haben sich in der Hardwarepraxis bei Kleinrechnern jedoch nicht durchgesetzt. Beispiele für derartige Kellerrechner sind: ICL, KDF9, BURROUGHS 5000 und 5500.

Verschiedene Registermaschinen unterscheiden sich in der Anzahl und Mächtigkeit der Register. Es gibt Rechenregister, sog. Akkumulatoren, auf denen Rechenoperationen ausgeführt werden, Zwischenspeicherregister, die nur geladen, gelesen und abgespeichert werden können, Adreßregister und Indexregister, die auf einen Operanden im Speicher verweisen. Basisadreßregister dienen in den meisten Fällen nur zur Erweiterung des durch die Wortlänge festgelegten Adressierraums. (In Sonderfällen werden sie als Kellerzeiger oder Indexregister eingesetzt). Die Verwaltung von Basisregistern sollte Aufgabe des Betriebssystems sein, so daß bei algorithmischen Problemen nur Akkumulatoren, Zwischenspeicher, Index- und Adreßregister in Erscheinung treten. Moderne Maschinen kennen Mehrzweckregister, die außer der Basisregistereigenschaft alle anderen Register Eigenschaften besitzen. Die Abbildung ZIEL-3/1 gibt einen Überblick über Register Eigenschaften von sechs zum Vergleich herangezogenen Maschinen.

| Maschine \ Registerart | PDP11 | MOTOROLA 6800 | INTEL 8080 | SIEMENS 310 | SIEMENS 306 | ZILOG 80 |
|------------------------|--------------------|---------------|--------------|-----------------------------------|-------------|--------------|
| Mehrzweck | 7 ↓ ↓ ↓ ↓ ↓ ↓ ↓ | | | 14 ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ | | |
| Akkumulator | ↓ ↓ ↓ ↓ ↓ ↓ ↓ | 2 ↑ ↓ | 1 ↑ | ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ | 2 | 1 ↑ |
| Zwischenspeicher | ↓ ↓ ↓ ↓ ↓ ↓ ↓ | ↓ | 4 ↑ ↑ ↑ ↑ | ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ | | 4 ↑ ↑ ↑ ↑ |
| Index | ↓ ↓ ↓ ↓ ↓ ↓ ↓ | 1 ↓ | ↑ ↑ ↑ ↑ | ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ | | 2 ↑ ↑ |
| Adreß | ↓ ↓ ↓ ↓ ↓ ↓ ↓ | ↓ | 1 ↑ | ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ | | 1 ↑ |
| Gesamtanzahl | 7 | 3 | 6 | 14 | 2 | 8 |

Abb. ZIEL-3/1: Registerart und -anzahl von sechs Vergleichsmaschinen. Ein ausgezogener Pfeil deutet auf zusätzliche, ein gestrichelter Pfeil auf eine zusätzliche, aber eingeschränkte Verwendungsmöglichkeit hin.

Alle Maschinen haben mindestens einen Akkumulator als Voraussetzung für eine gemeinsame Behandlung. Ziel der Arbeit muß es sein, nicht nur dieses eine, allen Maschinen gemeinsame Register einzusetzen, sondern nach Möglichkeit auch die anderen Register, um die Eigenschaften einer gegebenen Zielmaschine möglichst gut ausnützen zu können.

Ein noch signifikanteres Unterscheidungsmerkmal ist, wo bei der Verknüpfung zweier Operanden oder bei Transferbefehlen die Operanden stehen und wo das Ergebnis abgelegt wird. Es werden hier nur solche Maschinen betrachtet, bei denen das Ergebnis den Platz eines der Operanden einnimmt. Maschinen dieses Typs werden auch Ein- bzw. Zweiadreßmaschinen genannt. Beide Begriffe leiten sich von der Zahl der Operanden, die im Speicher stehen können, ab. Bei Einadreßmaschinen steht ein Operand in einem von eventuell mehreren Registern, der andere meist im Speicher oder auch im Befehl. Oft stehen sogenannte Direktoperanden nicht direkt im Befehl, sondern im Befehlsfolgewart, das aber mit zum Befehl gerechnet werden kann. Unterschiedliche Länge eines Befehls auf verschiedenen realen Maschinen darf kein Hindernis für die Portabilität von Programmen sein. Im Fall mehrerer Register muß die Registernummer mit im Befehl angegeben sein. Man könnte

Einadreßmaschinen mit mehreren Registern deshalb auch als Eineinhalb-adreßmaschinen bezeichnen. Da aber die Registernummer oft im Operationsteil eines Befehls integriert ist (z.B. INTEL 8080), ist die Bezeichnung Einadreßmaschine gerechtfertigt. Die Abbildung ZIEL-3/2 gibt einen Überblick über den Standort der Operanden bei Zweioperandenbefehlen bei fünf zum Vergleich herangezogenen Maschinen. Die Zweiadreßmaschine PDP-11 ist unter den Einadreßmaschinen nochmals aufgeführt, da eine solche Maschine bei der Programmierung wie eine Einadreßmaschine behandelt werden könnte, wenn Ineffizienzen in Kauf genommen werden. Durch besondere Rahmung ist die kennzeichnende Eigenschaft aller Einadreßmaschinen hervorgehoben.

2. Operand

| | | im Befehl | in einem Register | im Speicher | Maschinentyp |
|--------------------------|-------------------|-------------------|-------------------|----------------------|--------------|
| 1. Operand = Ergebnis | in einem Register | 11 68 80 30 10 | 11 68 80 30 10 | 11 68 80 06 30 10 | Einadreß |
| | im Speicher | 11 | 11 | 11 | Zweiadreß |

Abb. ZIEL-3/2: Unterschiedliche Operandenangaben bei Zweioperandenbefehlen

11 steht für PDP-11

68 " " MOTOROLA 6800

80 " " INTEL 8080 oder ZILOG 80

06 " " SIEMENS 306

30 " " SIEMENS R30

10 " " SIEMENS 310

Speicheradressen können direkt, indirekt oder indiziert angegeben sein. Bei direkter Adressierung ist die Adresse des Operanden, bei indirekter Adressierung die eines Zeigers angegeben. Diese Angabe steht entweder im Befehl oder in einem Register. Ein Zeiger kann auf einen weiteren

| Operandenangabe | Bezeichnung des Adressierungsmodus | Wirkungsweise | Beispielsmaschinen | | | | |
|---|--|---------------|--------------------|---------------|------------|-------------|-------------|
| | | | PDP 11 | MOTOROLA 6800 | INTEL 8080 | SIEMENS 306 | SIEMENS 320 |
| Operand im Register | Registerdirekt | | + | + | + | | + |
| Hinweis auf Adresse des Operanden steht im Register R | Registerindirekt | | + | + | + | | + |
| | Registerindirekt post increment | | + | | | | + |
| | Registerindirekt predecrement | | + | | | | + |
| | Register doppelt indirekt | | | | | | |
| | Register doppelt indirekt post increment | | + | | | | |
| | Register doppelt indirekt pre decrement | | + | | | | |
| konstanter Operand im Befehl | Index | | + | + | | | + |
| | Index indirekt | | + | | | | |
| | Konstante | | + | + | + | + | + |
| Adresse M steht im Befehl B | direkt | | + | | | + | |
| | indirekt | | | | | + | |

Abb. ZIEL-3./3: Adressierungsarten verschiedener Rechner mit bildlicher Erläuterung.

Die rechteckigen Kästen in der Spalte "Wirkungsweise" stehen für ein Register R, für eine Speicherzelle M, oder für einen Teil eines Befehlswortes. Die schraffierten Kästen enthalten den Operanden. Ein Einfachpfeil \rightarrow bedeutet: Der Inhalt des linken Kastens ist die Adresse des rechten Kastens. Ein Doppelpfeil \Rightarrow steht für genau einen Datentransport. Die Kreise $(+1)$ (-1) $(+)$ sind "Rechenwerke". Bei den Adressierungsmodi mit dem Zusatz postincrement wirkt zuerst der Einfachpfeil, anschließend einmal die Doppelpfeile, bei predecrement wirken zuerst die Doppelpfeile, anschließend der Einfachpfeil.

Zeiger verweisen. Wie weit die Verzeigerung geht, kann im Befehlscode stehen, oder der Zeiger enthält ein "Substitutionsbit", das eine weitere Verzeigerung anzeigt. Beispiele für die Speichersubstitution ist die SIEMENS 305, bei der die Substitution beliebig oft erlaubt ist, und die SIEMENS 306, bei der sie auf vier Schritte begrenzt ist.

Bei indizierter Adressierung steht im Befehl eine konstante Adreßabstandszahl (Offset), die zum Inhalt eines Indexregisters addiert, eine Adresse ergibt. Bei indizierter indirekter Adressierung stellt das Ergebnis dieser Addition einen Zeiger dar.

Die Abbildung ZIEL-3/2 muß noch durch eine ausführliche Liste der Adressierungsangabe, die Abbildung ZIEL-3/3 enthält, ergänzt werden. Sie ist so in der Tabelle enthalten, wie sie der Assemblerprogrammierer sieht, auch wenn die Operanden in Wirklichkeit relativ zum Befehlszähler adressiert werden. (Dem Assemblerprogrammierer erscheint der PDP-11 Mode "relativ" als direkt, da er nur eine symbolische Adresse anzugeben braucht. Den Abstand zum Befehlszähler berechnet der Assembler).

Zum leichteren Verständnis der Spalte Wirkungsweise in der Abbildung ZIEL-3/3 seien drei Beispiele herausgegriffen:

1. Bei der Adressierungsart "Register indirekt pre-decrement" steht im Register R eine Adresse. Diese wird um eine Adreßeinheit vermindert und ins Register R zurückgeschrieben. Unter der verminderten Adresse ist der Operand zu finden.
2. Bei der Adressierungsart "Register doppelt indirekt post-increment" steht im Register die Adresse einer Speicherzelle, die nicht den Operanden, sondern erst die Adresse des Operanden enthält. Nach der Referenzierung des Operanden wird das Register um eine Adreßeinheit weitergeschaltet.
3. Bei der Adressierungsart "direkt", der häufigsten Adressierungsart, steht die Adresse eines Operanden im Befehl.

Die Tabellen in Abbildung ZIEL-3/2 und /3 beispielsweise weisen auf die Möglichkeit der PDP-11 hin, daß bei einer Verknüpfung der eine Operand über ein Indexregister erreicht werden, der andere vielleicht indirekt angegeben sein kann.

So gibt es den Additionsbefehl

| Operator | Operand1 | Zielloperand = Operand2 |
|----------|----------|-------------------------|
| ADD | 10(R1) | SUMME |

wobei 10 der Offset zur im Register 1 angegebenen Adresse und SUMME eine symbolische Adresse bedeutet.

Beim Exklusiv-Oder-Befehl kann der Operand1, beim arithmetischen Schiebepfeil der Zielloperand aber ausschließlich im Register direkt-Modus angegeben werden. Dabei ist gerade die PDP-11 noch "recht orthogonal" aufgebaut, d.h. trägt man die Operation über den möglichen Adressierungsarten in einem Diagramm auf, so enthält bei dieser Maschine der dadurch aufgespannte Raum nur kleine Lücken. Ein solches Diagramm, in dem die Lücken deutlich auffallen, zeigt die Abbildung ZIEL-3/4.

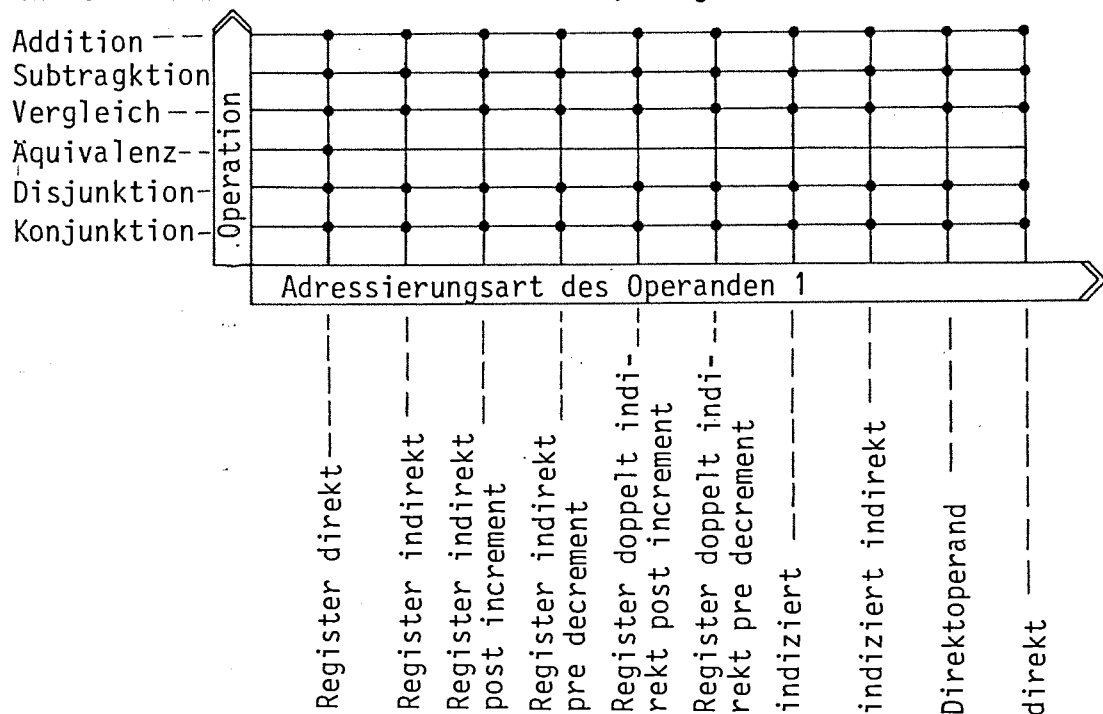


Abb. ZIEL-3/4: Beispiel für Nicht-Orthogonalität der PDP-11 bezüglich der Adressierungsart des Quelloperanden bei dyadischen Operationen.

Erweitert man das Diagramm durch eine dritte Achse, welche die Register-eigenschaften enthält, so sind der INTEL 8080 und der ZILOG 80 Musterbeispiele für Nicht-Orthogonalität. (Siehe dazu Tabelle ZIEL-3/1, Registerart und -anzahl von sechs Vergleichsmaschinen).

Ein weiterer Unterschied zwischen mehreren Maschinen ist durch die Länge der Operanden gegeben. Zwar geht die Tendenz bei Kleinrechnern derzeit zu 16-bit Wortlänge, jedoch sind 8-bit Rechner noch stark am Markt vertreten. Einige Rechner haben die Möglichkeit, bei fast jedem Befehl wahlweise ein halbes oder ganzes Wort zu adressieren, was allerdings meist auf Kosten des Adressierraumes geht und diesen halbiert. Beim ZILOG 80 unterscheiden sich die Register durch die Wortlänge der Operanden, die sie bearbeiten können.

Große Unterschiede, die sich allerdings weniger auf den Algorithmus eines Laufzeitprogramms, sondern mehr auf die Schnittstelle zum aufrufenden Programm auswirken, liegen in der Art, wie Rücksprungadressen bei Unterprogrammsprüngen gerettet werden, speziell, ob dafür ein Stack vorhanden ist und wie er ansprechbar ist. In Abbildung ZIEL-3/5 sind die Rettmechanismen der Rücksprungadresse für einige Beispielsmaschinen gelistet.

| Hinterlegung der Rücksprungadresse | PDP-11 | MOTOROLA 6800 | SIEMENS 306 | SIEMENS 310 |
|------------------------------------|---|---------------|-------------|-------------|
| im Stack | + | + | | + |
| in einem Register | ⁺ push alter Registerinhalt | | | + |
| im Unterprogramm | | | + | |

Abb. ZIEL-3/5: Rettung der Rücksprungadresse bei einigen Beispielsmaschinen

Die Rettung der Rücksprungadresse im Unterprogramm selbst erschwert die Programmierung wiedereintrittsfähiger Programme. Sie ist bei modernen Rechnern durch die Einführung eines Kellers ersetzt. Wie auch Wirth in [WIR76] feststellt, bringt die Hinterlegung der Rücksprungadresse in einem Register zusätzlich zu der Kellerlösung (PDP-11) keine Vorteile.

Als weiteres Vergleichsmerkmal kann der Operatorenvorrat herangezogen werden. Die Abbildung ZIEL-3/6 enthält eine Liste verschiedenartiger Befehle mit Beispielen.

| Befehlsart | Beispiele |
|-----------------------|---|
| Einoperandenbefehle | Löschen |
| Arithmetische Befehle | Addieren |
| Logische Befehle | boole'sches Verodern |
| Schiebebefehle | arithmetisch rechtsverschieben mit Rundung, logisch rechtsverschieben |
| Sprungbefehle | springen, falls Ergebnis = \emptyset , um 5 Adreßeinheiten weiterspringen |
| Sonderbefehle | 1 addieren, negieren, Einerkomplement bilden |
| Ein-Ausgabebefehle | ausgeben des Akkuinhalts |
| Systembefehle | rückspringen aus Interruptroutine |

Abb. ZIEL-3/6: Liste verschiedenartiger Befehle mit Beispielen

Ein-Ausgabebefehle und Systembefehle können im Zusammenhang mit algorithmischen Programmen unberücksichtigt bleiben.

Selbst wenn zwei Rechner in Befehlssatz, Adressierungsmodus und Wortlänge übereinstimmen, wie z.B. die PDP-11 und die SIEMENS 310, so kann doch noch ein gleiches Programm auf verschiedenen Rechnern ganz verschiedene Wirkungen zeigen, wie das folgende Programmstück in PDP-11 Notation zeigt. Es soll die Konstante KONST zum Inhalt des Registers R3 addiert werden und das Ergebnis, falls es positiv ist, in Zelle ERG abgespeichert werden.

```

ADD R3, KONST      /*
BMI MINUS          /* Bedingter Sprung
MOV R3, ERG        /* Transfer

```

MINUS:

Steht im Register R3 vor der angegebenen Befehlsfolge eine "große" positive Zahl (z.B. die Oktalzahl 012345) und wird eine "große" positive Zahl (z.B. 065433) addiert und überschreitet das Ergebnis den darstellbaren Zahlenbereich, so erhält man auf der PDP-11 und auf der SIEMENS 310 die gleiche negative Zahl (in unserem Beispiel 100000).

Die Sprungbedingung ist auf der PDP-11 erfüllt, auf der SIEMENS 310 wird aber nur das Überlaufereignis erkannt, was programmabhängig einen Einsprung ins Betriebssystem bewirkt, oder vom Anwenderprogramm abgefragt werden kann. Eine Abfrage, ob das Ergebnis negativ ist, ist bei der SIEMENS 310 erst nach einem zusätzlichen Testbefehl, bei der PDP-11 gleich nach dem Subtraktionsbefehl möglich.

Dieses kurze Beispiel lehrt, daß beim Vergleich von Maschinen die Erzeugung von Ergebnisanzeigen mit berücksichtigt werden muß, wenn man es mit so maschinennahen Problemen wie Laufzeitprogrammen zu tun hat. Wegen anders gearteter Ziele ist dieses Unterscheidungskriterium in den oben genannten Literaturstellen nicht aufgenommen.

LWEG - Lösungsweg

LWEG-1. Entwicklung einer maschinennahen Notationssprache

Trotz der Mannigfaltigkeit verschiedener Rechner wird nun nach einem Verfahren gesucht, algorithmische Laufzeitprogramme portabel zu machen. Zur Lösung dieser Aufgabe gehört es, Algorithmen zu beschreiben. Die Beschreibungsmethode könnte verschiedenartig aussehen:

1. Umgangssprachliche Beschreibung
 2. Graphische Darstellung in Form von Flußdiagrammen oder Struktogrammen
- Beide Beschreibungsarten sind aber noch zu weit von der Programmierung entfernt. Es sind aber Vorleistungen, die zur Programmierung nötig sind. Etwas maschinennäher wäre, die Laufzeitprogramme in einer höheren Programmiersprache, z.B. PEARL, zu formulieren und bei jedem Programmstück eine semantische Beschreibung mitzuliefern. Die einzelnen Programmstücke lassen sich dann auf Grund der Beschreibung in einer Assemblersprache nachahmen, was dadurch vereinfacht wird, daß bereits ein Vorbild in der höheren Programmiersprache vorliegt. Dieses Portabilitätsprinzip erfordert aber trotzdem sehr viel Handarbeit. Andere Portabilitätsverfahren gehen wie der ASME-Stufe 1 Compiler von einer Zwischensprache aus, der ein abstrakter Rechner, ein Begriff, der von Pool [P0071] geprägt wurde, zugrunde liegt. Parnas [PAR78, KIM79] schlägt zum Entwurf von Software-spezifikation als Anhaltspunkte unter anderem die Bildung von abstrakten Maschinen vor.

Poole und Waite machen in [P0071] Vorschläge zum Entwurf abstrakter Maschinen. Diese Vorschläge beinhalten Gleitpunktzahlen mit den dazugehörigen Operationen, Zeichenketten mit der Konkatenation und fordern somit Operatoren, deren Realisierung Aufgabe der Laufzeitprogramme ist. Sie sind dem hier gegebenen Problem nicht angemessen und verletzen Poole's und Waite's oberste Forderung nach einer dem Problem angepaßten Zwischensprache. Diese Forderung erfüllt eine Sprache auf Assembler-niveau. Brown [BR072] hat mit einer solchen Sprache Erfahrungen gesammelt: Er vergleicht "LOWL", eine Assemblersprache für eine Maschine mit einem Akku und einem Indexregister mit "L", einer Sprache auf hoher Ebene.

L ist eine Mischung von vertrauten Anweisungen wie IF, SET, CALL und Anweisungen, die speziell zur Implementation von ML/I benötigt werden, wie STACK, MOVE und BACKSPACE. Die folgenden Codestücke sind aus [BR072] entnommen.

```
IF NTYPSW = 6 & IDPT + IDLEN GR SIZE THEN
    SET DIFF = IND(SPT + 1) NM-IDLEN + 1
    STACK NTYPSW (SW) ON FSTACK
    MOVE FROM IDPT TO STAKPT LENG 13
        BACKWARDS
END
```

```
LAB CALL SCAN(IDPT)PT
    SETSW NTYPSW = TYPSW & MASKSW
    BACKSPACE ARGPT
```

Die ersten beiden Zeilen ergeben in LOWL folgendes Codestück

| | | |
|------|------------|--|
| LAV | NTYPSW | (load A with variable) |
| CAL | 6 | (compare A with a literal) |
| GONE | GL 20, 100 | (go if not equal to label 20 which is 100 LOWL statements ahead) |
| LAV | IDPT | |
| AAV | IDLEN | (add to A a variable) |
| CAV | SIZE | (compare A with variable) |
| GOLE | GL 20, 96 | (go if less or equal to label GL20 which is 96 LOWL statements ahead) |
| LBC | SPT | (load B with variable) |
| LAM | 1 | (load A with address, given by B modified by a numerical operand) |
| SAV | IDLEN | (subtract from A a variable) |
| AAL | 1 | (add to A a literal) |
| STV | DIFF,X | (store A in variable; X means, A need not be preserved) |

Brown codiert den Makroprozessor ML/I [BR067] abgesehen von einigen handgeschriebenen maschinenabhängigen Teilen in LOWL und L und setzt den LOWL- sowie den L- Code mit demselben Makroprozessor auf eine ICL 4130, IBM 360 und PDP-11 um. Der Vergleich des aus LOWL und L erzeugten Maschinencodes ergibt, daß aus der niederen Zwischensprachenebene

keine Effizienzverbesserungen auf den Zielmaschinen resultieren. Brown selbst sieht einen Grund für die Ineffizienz des aus LOWL erzeugten Codes in dem auf der LOWL-Maschine fehlenden "Dekrementiere um 1" Befehl. Ein gewichtigerer Grund dürfte in der speziellen Einakkuarchitektur der LOWL-Maschine zu suchen sein. Diese Untersuchungen entmutigen daher noch nicht beim Versuch, für algorithmische Laufzeitprogramme eine assemblernahe Notationssprache zu konstruieren, obwohl man sich wieder der mühsamen und fehleranfälligen Assemblerprogrammierung nähert. Bedenkt man aber beispielsweise die Nähe einer effizienten Zweiwortaddition zum Übertragungsbit (auch Link- oder Carrybit genannt) einer realen Maschine, so erscheint nur eine Sprache, die solche Einzelbits berücksichtigt, dem Problem angemessen. Der Nachteil einer so starken Maschinennähe fällt aber bei portablen Programmen, die mehrfach verwertet werden sollen, nicht mehr so stark ins Gewicht und wird in Kauf genommen, wenn es zur Effizienzsteigerung dient.

Die erste Idee bei der Suche nach einer assemblernahen Notationssprache besteht darin, einen "gemeinsamen Durchschnitt" oder größte Maschine gemeinsamer Eigenschaften aller in Frage kommenden Zielmaschinen zu bilden. Wenn man aber an die in Kapitel ZIEL-3 dargestellte Vielfalt verschiedenartiger Rechner denkt, wird sich ein nicht leerer Durchschnitt nur innerhalb von noch zu bildenden Klassen finden lassen. Abbildung LWEG-1/1 verdeutlicht diese Tatsache bildhaft. Jede Klasse erfordert eine eigene Sprache. Das hat zur Folge, daß die Laufzeitprogramme mehrmals neu zu formulieren sind. Die Sprache liegt bei den Durchschnittsmaschinen genau auf dem Niveau der Zielmaschinen. Deshalb wird der erzeugte Code nicht besonders effizient sein können, da kein "Abstieg" möglich ist, oder anders ausgedrückt, weil bei den Durchschnittsmaschinen viele Eigenschaften spezieller Zielrechner verloren gehen. Die mehrfache Formulierung und die Niveaugleichheit sprechen also gegen Klassen- und Durchschnittsbildung. Das Gegenteil wäre, eine "Vereinigungsmaschine" zu konstruieren, welche die wichtigsten Eigenschaften aller Maschinen in sich vereinigt. Eine solche Maschine liegt im Niveau über den meisten in der Vereinigungsmenge enthaltenen Rechnern und somit ist ein Abstieg zu den meisten realen Zielmaschinen möglich, was eine bessere Effizienz des Zielcodes erwarten läßt. Bei der Abbildung von Eigenschaften der Vereinigungsmaschine, die auf einer realen

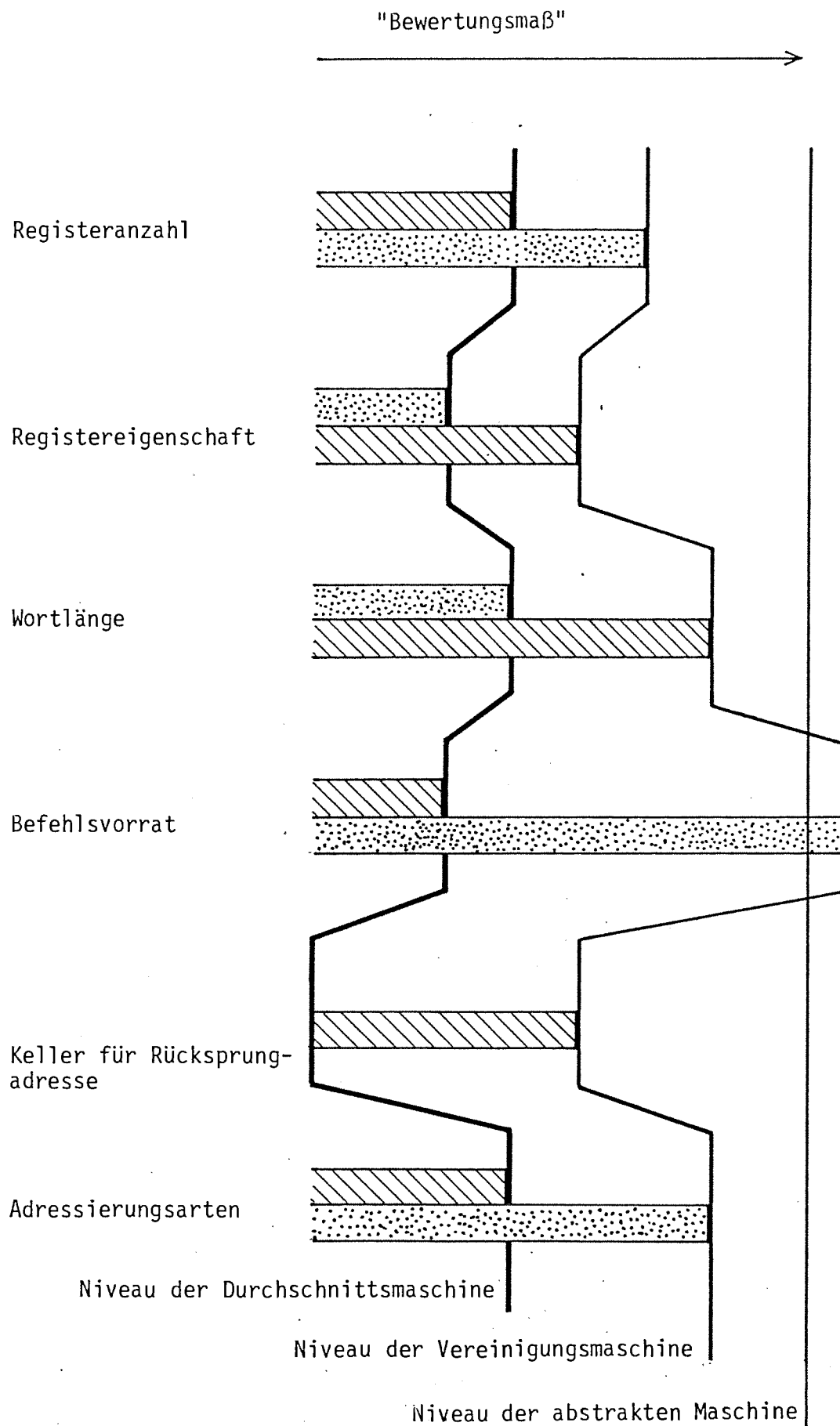


Abb. LWEG-1/1: Bildhafte Darstellung des Durchschnitts und der Vereinigung zweier Maschinen

Zielmaschine nicht wiederzufinden sind, muß eine Ersatzprogrammierung vorgenommen werden. Dabei könnte wahlweise oder vermischt eine Makro- oder Unterprogrammtechnik angewandt werden.

Bessere Ergebnisse als bei einer Vereinigungsmaschine kann man noch erzielen, wenn man die Maschinenstruktur verschiedener Zielrechner berücksichtigt. Dazu ist wieder eine Klasseneinteilung nötig. Dann führt jeweils eine Vereinigung zu einer die Klasse repräsentierenden abstrakten Maschine. Man muß sich auf möglichst wenige Klassen beschränken, da jeweils eine zeitaufwendige Neubearbeitung der Laufzeitprogramme erforderlich ist.

Ein Kompromiß wird darin gesehen, nur zwei Klassen einzuführen:

Klasse der Einadreßmaschinen

Klasse der Zweiadreßmaschinen

Die doppelte Formulierung der Laufzeitprogramme läßt sich noch vermeiden, wenn sie für eine etwas höhere abstrakte Maschine formuliert werden und die Umsetzung der Programme in die Assemblersprachen der beiden niedereren Maschinen automatisiert wird. Über den abstrakten Ein- und Zweiadreßmaschinen liegt eine abstrakte Dreiadreßmaschine. Sie wird im Anhang 1 genauer beschrieben, hier wird nur Grundsätzliches gesagt.

Die abstrakte Dreiadreßmaschine hat keine Register. Alle Verknüpfungen werden direkt auf dem Speicher ausgeführt. Jede Speicherzelle läßt sich zur Adressierung in der Art eines Zeigerregisters mit und ohne Predecrement und Postincrement und in der Art eines Indexregisters verwenden. Bezüglich der vorhandenen Adressierungsarten stellt die hier definierte abstrakte Maschine in etwa die Vereinigung der meisten am Markt befindlichen Kleinrechner dar. Gleiches gilt für den Operatorensatz. Die Dreiadreßmaschine liegt im Niveau über den in Frage kommenden Zielmaschinen, so daß ein Abstieg möglich ist. Als Operanden sind nur Konstante, Parameter und symbolische Namen möglich, so daß man sich über eine optimale Registerverwendung oder Stackverwaltung auf dieser Ebene noch keine Gedanken zu machen braucht.

Dreiadreßbefehle haben die Form

Quelle1 verknüpft mit Quelle → Ziel

Neben diesen Befehlen kennt die abstrakte Dreiadressmaschine natürlich noch andere Befehle, wie Transferbefehle etc.. Dreiadressbefehle sind aus Compilerbüchern z.B. [GRI71, SCH75] als interne Form des Quellcodes bekannt.

Eine Dreiadreß maschinensprache ist auch Johnson und Mueller's Konstruktion Definition Language [JOH71], in der sie bei der Simulation von Mikrocomputern auf Großrechnern die Maschinenbefehle der Mikrocomputer auf Register-Ebene beschreiben. Sie ziehen diese Register-Transfer-Sprache auf Befehlsebene der auf Gatter-Ebene liegenden "Computer design Language" CDL von YAOH und Chu [YAO72] vor. Eine auf den ersten Blick ganz andere Zwischensprache verwendet Hofmann [HOF79] in einem Prototyp seines portablen Betriebssystems. Es ist in einem Dialekt von PASCAL beschrieben und ist somit auf Maschinen, auf denen ein PASCAL Compiler läuft, sofort ablauffähig, oder wird mit Hilfe des STAGE II [WAI70] auf einen Rechner gebracht. Die hauptsächliche Einschränkung der PASCAL-Untermenge gegenüber der Gesamtsprache liegt in den Zuweisungsanweisungen. Auf der rechten Seite dürfen maximal zwei Operanden stehen, eine Eigenschaft einer Dreiadreßmaschine.

Die Frage, warum bei der Portabilität arithmetischer Laufzeitprogramme nicht ebenfalls ein PASCAL Subset verwendet wird, läßt sich erst nach Kenntnis der später genauer beschriebenen Optimierungshinweise, die PASCAL nicht kennt, beantworten.

Ein wichtiges Kriterium bei der Auswahl einer Zwischensprache, das Poole und Waite nicht erwähnen, sollte die Möglichkeit einer möglichst dokumentationsfreundlichen Gestalt sein. Dazu müssen alle Zwischensprachenanweisungen menschenlesbare Form haben. Die Sprache sollte formatfrei sein und an jeder beliebigen Programmstelle müssen sich Kommentare einschieben lassen.

Die Rettung der Rücksprungadresse ist bei einem realen Rechner hardwaremäßig fest vorgegeben. Hier könnten Vorschriften seitens der abstrakten Maschine nur die Effizienz verschlechtern. Deshalb wird in einem abstrakten Unterprogramm nur eine Kennzeichnung des Unterprogrammein- und -aussprungs programmiert. Bei der Umsetzung muß eine Anpassung an die Gegebenheiten der Hardware erfolgen. Sie erfolgt durch den Benutzer der abstrakten Programme, indem er, Generierparametern ähnlich, Assemblercodestücke angibt.

Das soll am Beispiel einer Maschine, welche die Rücksprungadresse im Unterprogramm hinterlegt, näher erläutert werden. Eine solche Architektur verhindert natürlich das Ablegen von Laufzeitprogrammen in einem billigen Nur-Lese-Speicher. Will man trotzdem mit solchen Speichern arbeiten, so ist statt eines einfachen Unterprogrammsprungs eine kleine Befehlsfolge zu durchlaufen, um die Rücksprungadresse etwa in einen simulierten Keller zu retten. Beim Rücksprung sind entsprechende Maßnahmen nötig. Stört die Hinterlegung der Rücksprungadresse nicht aus dem genannten Speichergrund, sondern bei der Erzielung der Wiedereintrittsfähigkeit, so kann eine Kellersimulation auch im Unterprogramm selbst erfolgen. Beim Einsprung muß natürlich eine Unterbrechbarkeitssperre gesetzt sein, damit die Rücksprungadresse nicht durch erneuten Aufruf sofort überschrieben wird. Verzichtet man auf Wiedereintrittsfähigkeit, so kann der Unterprogrammsprung der realen Maschine abgesetzt werden. Ähnliche Unabhängigkeit von hardwaremäßigen Gegebenheiten sowie Implementationswünschen sollte auch bei der Behandlung von Parametern gewährleistet sein.

Wie die Abbildung der Parameterversorgung und des Zugriffs auf formale Parameter aussieht, wird vom Benutzer ebenfalls durch Assemblercodestücke beschrieben. Dabei sind zur optimalen Anpassung an verschiedene Codegeneratoren verschiedenartige Routinen denkbar.

Ein solches Codestück könnte beispielsweise einen Parameterversorgungsblock laden und ein Indexregister auf den Blockanfang stellen. Der Zugriff auf einen Parameter, egal ob Ein- oder Rückgabeparameter, hat dann immer indiziert zu erfolgen. Viele Compiler legen den Parameterversorgungsblock direkt vor die lokalen Daten. Dann können lokale Daten und Parameter auf gleiche Weise referenziert werden. Solche Methoden sind bei der Übersetzung von höheren Programmiersprachen sicherlich angebracht und schematisch bei allen Prozeduraufrufen einsetzbar. Bei Laufzeitprogrammen ist aber auch eine Parameterübergabe in Registern üblich. Eine unterschiedliche Übersetzung von Anwender- und Laufzeitprozeduren erschwert die Codegenerierung. Beim Einsatz eines generierten Codegenerators [PEL81] muß man sich wohl auch auf eine Art der Parameterübergabe festlegen. Bei Laufzeitprogrammen sind die Unterprogramme aber alle bekannt, so daß eine individuelle Behandlung zur Erzielung

eines möglichst effizienten Codes leichter möglich ist. So können für verschiedene Unterprogramme, z.B. für direkt vom Codegenerator abgesetzte Aufrufe und innerhalb der abstrakten Routinen aufgerufene Unterprogramme, verschiedenartige Parameterbehandlungen vorgesehen sein. Darüber hinaus können die abstrakten Laufzeitroutinen auf mehrere Gruppen aufgeteilt werden, so daß sich für jede Gruppe eine eigene Parameterbehandlung berücksichtigen läßt. Ein Beispiel für unterschiedliche Parameterübergabe findet man in der Spezifikation des von Hand erstellten CIMIC-1 Codegenerators für die SIEMENS 310. [TRA80]. Hier sieht der Codegenerator Parameterversorgungen über Register vor, wobei die Registeranzahl und die Registernummern von einem Laufzeitprogramm zum anderen wechseln.

LWEG-2. Zielmaschinenabhängigkeit

Bevor mit der Programmierung begonnen wird, sind Algorithmen zu überlegen. Um effiziente Programme zu erstellen, müssen diese an gewisse Eigenschaften der Zielschienen angepaßt werden. So hängen der Grad sowie die Koeffizienten von Approximationspolynomen von der Wortlänge ab. [RAL66, HAS55]

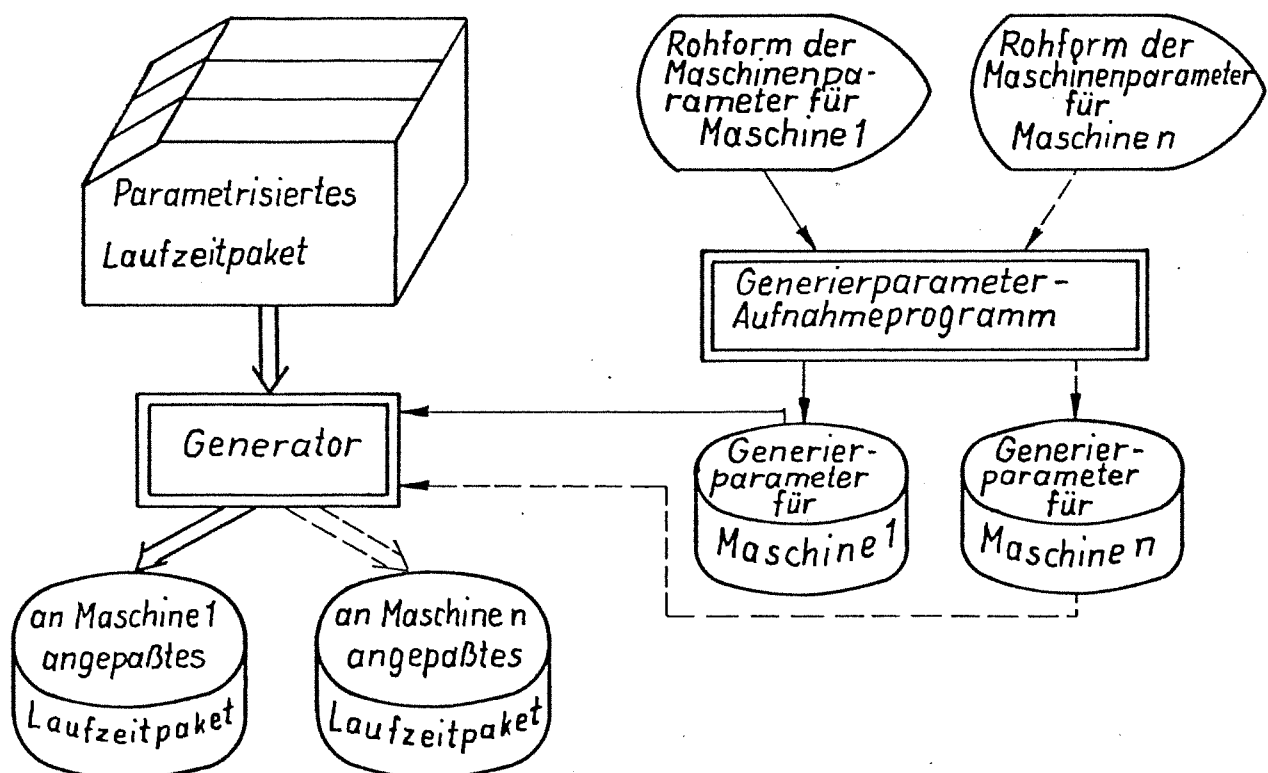


Abb. LWEG-2/1: Graphische Veranschaulichung des Generierprogramms

Abhängig von den im Dialog mit einem Generierparameter-Aufnahmeprogramm erzeugten Dateien, erstellt der Generator aus einem mit maschinenabhängigen Verzweigungen und Konstanten parametrisierten Stapel abstrakter Laufzeitprogramme, ein an die gewünschte Zielmaschine angepaßtes Laufzeitpaket.

In vielen Fällen kann der für eine Maschine konzipierte Algorithmus zur Realisierung eines Laufzeitprogramms im Prinzip für alle Maschinen verwendet werden, jedoch sind an sehr vielen Stellen kleine maschinenabhängige Verzweigungen nötig. Nur selten werden ganze Programme auszutauschen sein. Ein solch krasses Beispiel stammt aus der Zeichenkonkatenation. Hier unterscheidet sich der Algorithmus wesentlich, wenn pro Wort ein oder mehr als ein Zeichen gespeichert wird. Zwar ist der Fall für ein Zeichen pro Wort im allgemeinen Fall enthalten, doch führt dieser Spezialfall zu Codeineffizienz, und zwar schon auf der Ebene der abstrakten Dreiadreßmaschine.

Hat man eine bestimmte Zielmaschine im Auge, so ist klar, daß nur einige der Verzweigungen durchlaufen werden können, da die anderen auf Grund der Zielmaschinenparameter von vornherein ausscheiden. Es ist daher nicht sinnvoll, alle Abfragen zur Laufzeit durchzuführen.

Vielmehr sollten sie einmal für jede Zielmaschine von einem Generatorprogramm ausgewertet werden. Dieses Programm erstellt durch bedingte Generierung aus einem sog. parametrisierten ein sog. angepaßtes abstraktes Laufzeitprogramm. Der Generiervorgang ist in Abb. LWE-2 /1 durch eine Graphik veranschaulicht. Die zielmaschinenabhängigen Verzweigungen sind Generierbedingungen, die sich natürlich von den bedingten Sprüngen der abstrakten Maschine syntaktisch unterscheiden müssen.

Die folgende Syntax läßt wegen der Anforderung aus der Praxis auch geschachtelte Verzweigungen zu. Die Syntax ist hier, wie auch in anderen Kapiteln in erweiterter Backus-Naur-Notation niedergelegt. Hochgestellte Punkte bedeuten eine beliebige Anzahl von Wiederholungen. 'name' und 'virtueller-Dreiadreß befehl' werden nicht weiter abgeleitet.

```

<zielmaschinenabhängige Verzweigung> ::=
    IF<boole'scher Ausdruck> THEN
    { virtueller-Dreiadreß befehl | <zielmaschinenabhängige Verzweigung> }
    ELSE { <virtueller-Dreiadreß befehl | <zielmaschinenabhängige> }
        Verzweigung

FI

<boole'scher-Ausdruck> ::=
    <vergleichsausdruck> | <boole'scher-Ausdruck> { AND | OR }
        <vergleichsausdruck>

<vergleichsausdruck> ::= [ NOT ]
    { <Generierparameter> { EQ | NE | LT | LE | GE | GT }
        { <generierparameter> | <ganze-zahl> | TRU | FALSE } |
        boole'scher-Ausdruck

<Generierparameter> ::= name

```

Die Reihenfolge der Abarbeitung ist von links nach rechts definiert.

Oft lassen sich Anpassungen des abstrakten Programms an die Verhältnisse der realen Maschine durch einfaches Einsetzen von maschinenabhängigen Konstanten erreichen. (Beispiel dafür: Anzahl der Zeichen, die ein Wort bei gepackter Darstellung faßt.) Diese Generierparameter sind von Fluchtsymbolen (z.B. ↑, ') eingeschlossen. Um die Anzahl der Generierparameter einzuschränken, kann zwischen diesen Fluchtsymbolen auch Arithmetik getrieben werden, wie das folgende Beispiel mit den nicht notwendigerweise unterschiedlichen Zeichen ' und ↑ zeigt.

↑ 'WORTL IN BITS'+1 - ('ZEICHEN PRO WORD'*'BIT PRO ZEICHEN') ↑

Nur bei häufig auftretenden derartigen arithmetischen Ausdrücken mag es zweckmäßig sein, dafür eigene Generierparameter zu definieren.

Die Generierparameter werden dem Generator in Form von Listen zur Verfügung gestellt. Zu deren Aufbau dient zur Vermeidung von Fehlern ein Aufnahmeprogramm, welches einfache Parameter zu komplexeren zusammenfaßt.

Beispiel: Bei der Gleitpunktarithmetik werden verschiedene Masken, u.a. zum Ausblenden des Exponenten, benötigt. Das Generierparameter-Erfassungsprogramm errechnet sich daraus Verschiebezahlen, die angeben, wie weit eine ausgeblendete Mantisse bis zur Linksbündigkeit verschoben

werden muß. Werden als Folge der Erfindung neuer Programmiersprachen neue Laufzeitprogramme benötigt, ist denkbar, daß auch neue Generierparameter erfunden werden müssen. Dann wird das Aufnahmeprogramm unter Verwendung vorhandener Prozeduren ergänzt.

LWEG-3: Konzept der Umsetzung

In LWEG-1 ist auf die automatische Umsetzung der abstrakten Laufzeitprogramme von der Notationssprache, der Assemblersprache einer abstrakten Dreiadreßmaschine, auf die Assemblersprache von abstrakten Ein- bzw. Zweiadreßmaschinen hingewiesen worden. Hier wird das Konzept der Umsetzung genauer erläutert. Sie erfolgt in mehreren Stufen. Ihr geht das in LWEG-2 erwähnte Programm zur Erfassung der Generierparameter voran. Es prüft nur den Quelltext auf die Symmetrie der Fluchtsymbole hin und wertet die boole'schen und arithmetischen Ausdrücke aus, läßt aber ansonsten den Quelltext ungeprüft. Über die weiteren Stufen gibt Abbildung LWEG-3/1 eine Übersicht. Die eigentliche Umsetzung beginnt mit der zweiten Stufe, der Syntaxanalyse. Neben der Prüfung auf korrekten Aufbau des vom Generator hinterlassenen Programms wandelt sie zum Zweck der besseren Weiterverarbeitung mit dem Rechner die dokumentationsfreundliche in eine maschinenfreundliche Darstellung um. Dabei setzt sie Operatoren, Adressierungsarten und Ergebnisanzeigen in einen Zifferncode um. Dezimale Konstante verschlüsselt sie binär. Variablen- und Markennamen bleiben in der Originalfassung erhalten.

An die Syntaxanalyse schließt sich eine Optimierungsstufe an, mit der Aufgabe, den Generator zu ergänzen. Dieser ersetzt "ohne zu denken" arithmetische Ausdrücke durch Konstante. Dadurch können zwangsläufig Redundanz enthaltende Befehle entstehen, wie sie in der Abbildung LWEG-3 /2 in der linken Spalte aufgeführt sind. Die Optimierungsstufe beseitigt diese Redundanz und erzeugt die Transferbefehle der rechten Spalte in Abbildung LWEG-3 /2. Bei manchen Befehlen bleibt ein Transferbefehl übrig, bei dem Quelle und Senke identisch sind; solche Befehle eliminiert diese Redundanzbeseitigungsstufe. Optimierungen dieser Art sind aus der Theorie der Übersetzung von Ausdrücken her bekannt.

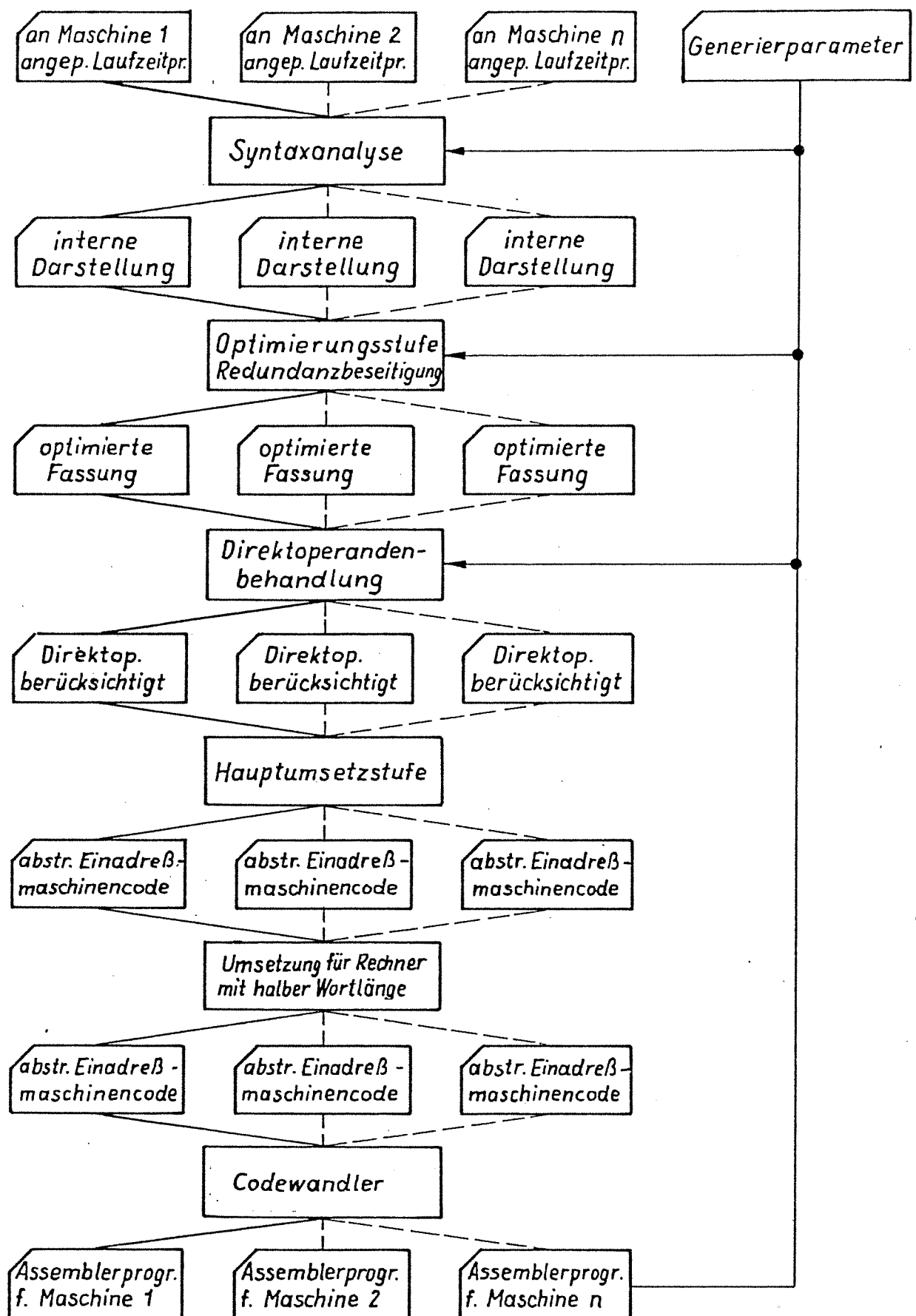


Abb. LWEG-3/1: Weitere Umsetzung der vom Generator an Maschine 1,2..n angepaßten Laufzeitprogramme.

| Mit Redundanz behafteter Befehl | Befehl nach Verbesserung durch die Vorverarbeitungsstufe | Kommentar |
|--|---|---|
| $Q1 + \emptyset \Rightarrow S$ $\emptyset + Q2 \Rightarrow S$ $Q1 - \emptyset \Rightarrow S$ $Q1 * 1 \Rightarrow S$ $1 * Q2 \Rightarrow S$ $Q1 / 1 \Rightarrow S$ $Q1 \text{ OR } \emptyset \Rightarrow S$ $\emptyset \text{ OR } Q2 \Rightarrow S$ $Q1 \text{ AND } 1..1 \Rightarrow S$ $1..1 \text{ AND } Q2 \Rightarrow S$ $Q1 \text{ SHIFT } \emptyset \Rightarrow S$ $\emptyset \text{ SHIFT } Q1 \Rightarrow S$ $Q1 * \emptyset \Rightarrow S$ $\emptyset * Q2 \Rightarrow S$ $Q1 \text{ AND } \emptyset \Rightarrow S$ $\emptyset \text{ AND } Q2 \Rightarrow S$ $Q1 \text{ OR } 1..1 \Rightarrow S$ $1..1 \text{ OR } Q2 \Rightarrow S$ | $Q1 \Rightarrow S$ $Q2 \Rightarrow S$ $Q1 \Rightarrow S$ $Q1 \Rightarrow S$ $Q2 \Rightarrow S$ $Q1 \Rightarrow S$ $Q1 \Rightarrow S$ $Q2 \Rightarrow S$ $Q1 \Rightarrow S$ $Q2 \Rightarrow S$ $Q2 \Rightarrow S$ $\emptyset \Rightarrow S$ $\emptyset \Rightarrow S$ $\emptyset \Rightarrow S$ $\emptyset \Rightarrow S$ $\emptyset \Rightarrow S$ $1..1 \Rightarrow S$ $1..1 \Rightarrow S$ | 1..1 ein Wort voller Einsen 1..1 ein Wort voller Einsen SHIFT steht stellvertr.f. alle Schiebepfeile 1..1 ein Wort voller Einsen 1..1 ein Wort voller Einsen |

Abb. LWEG-3/2: Befehle, die der Generator möglicherweise hinterläßt und Verbesserung durch die Optimierungsstufe
 $Q1$ = Quelloperand1, $Q2$ = Quelloperand2, S = Senke
Die Befehle im oberen Teil der Tabelle entfallen, falls $Q1=S$ bzw. $Q2 = S$

| vom Generator hinterlassener Befehl | Befehl nach Verbesserung durch die Vorverarbeitung | Kommentar |
|--|--|--|
| $Q1 \text{ XOR } 1..1 \Rightarrow S$ $1..1 \text{ XOR } Q2 \Rightarrow S$ $\emptyset - Q2 \Rightarrow S$ $Q1 / 2 \Rightarrow S$ $Q1 / 2^i \Rightarrow S$ | $\text{NOT } Q1 \Rightarrow S$ $\text{NOT } Q2 \Rightarrow S$ $\text{NEG } Q2 \Rightarrow S$ $Q1 \text{ SAR } 1 \Rightarrow S$ $Q1 \text{ SAR } i \Rightarrow S$ | Vergleiche 2.4 portable Rechenergebnisse $i = 2..(WL-1)$ |

Abb. LWEG-3/3: "Strength Reduction" der Optimierungsstufe

Die Befehle in Abbildung LWEg-3 /3 werden sinnvollerweise nur dann ausgeführt, wenn die reale Zielmaschine über die entsprechenden Ersatzbefehle verfügt. Aufgaben dieser Art sind aus Compilerbüchern unter dem Namen "Strength Reduction" [AH078] bekannt. Falls die umzuwandelnden Befehle Ergebnisanzeigen setzen sollen, was in einem abstrakten Programm explizit vermerkt sein muß, so darf die Ersetzung nur dann durchgeführt werden, wenn man sich die Mühe macht, die zur Auswertung im abstrakten Programm vorgesehenen Ergebnisanzeigen mit den Anzeigen der Ersatzbefehle auf Übereinstimmung zu überprüfen (z.B. Abb. LWEg-3 /2, - /3). Der Start der nächsten Stufe, der Direktoperandenbehandlung, unterbleibt, wenn die reale Zielmaschine Direktoperanden in voller Wortlänge verarbeiten kann. Manche Maschinen verkraften keine Direktoperanden oder nur solche bis zu einem bestimmten Wert. Dann müssen sie gegebenenfalls aufgesammelt, mit einem symbolischen Namen versehen und am Ende eines zusammen übersetzten Laufzeitsystems abgelegt werden. Gleiche Konstanten sollten dabei nur einmal abgelegt werden.

Die nächste Stufe, im folgenden mit Hauptumsetzstufe betitelt, ist die wichtigste. Hier erfolgt die angekündigte Aufteilung in zwei Klassen von Maschinen:

1. abstrakte Einadreßmaschinen
2. abstrakte Zweiadreßmaschinen.

Die Umsetzungsprogramme müssen für jede Klasse gesondert geschrieben werden. Auf welche Klasse umgesetzt werden soll, ist durch Generierparameter festzulegen. Ziel dieser Stufe ist eine abstrakte Maschine, die bereits wesentliche Architektureigenschaften der realen Zielmaschine aufweist. Das wichtigste Kriterium sind die Anzahl und Eigenschaften der Register. Dabei sind nicht alle Register, die die Hardware anbietet, gemeint, sondern nur diejenigen, die Betriebssystem und Codegenerator den Laufzeitprogrammen zur beliebigen Verwendung übrig lassen. Es ist nicht erforderlich, daß für alle Laufzeitprogramme dieselben und die gleiche Anzahl von Registern zur Verfügung stehen. Dem Anwender des portablen Systems ist freigestellt, einige Register, die der Codegenerator nicht freigibt, umzuladen. Das erfordert aber wegen der nötigen Ununterbrechbarkeit tiefere Einblicke in das Betriebssystem.

Eine weitere Umsetzstufe könnte die auf der realen Zielmaschine nicht vorhandenen Adressierungsarten nachbilden und das verlangte Ansprechen der Parameter besorgen. Leider ergeben sich, wie sich an einem Test eines Unterprogramms zur Zeichenkettenbearbeitung demonstrieren läßt, bei Schleifen Ineffizienzen. Diese können zwar von Hand wieder beseitigt werden, doch erscheint der Aufwand dafür nicht tragbar. Deshalb muß die Hauptumsetzstufe um die Simulation nicht vorhandener Adressierungsarten erweitert werden. Dabei kann gleichzeitig die Adressierung über Indexregister mit erledigt werden. Eine ausführliche Beschreibung dieser Simulation findet der Leser in Abschnitt LOES-2. 4 "Simulation nicht vorhandener Adressierungsarten".

Die folgende Umsetzstufe kann wahlweise angeschlossen werden. Sie setzt den bisher erzeugten Code auf eine abstrakte Maschine mit nur der halben Wortlänge um. Dies ist dann einfach, wenn zu jedem Register ein Hilfsregister zur Verfügung gestellt werden kann. In den folgenden Beispielen sind D, E, F, G Variablennamen, R1L, R2L, R1H, R2H Register, C ist das Übertragungsbit. Die Punkte sind als "Inhalt von", der Pfeil als "geht nach" zu lesen. Die Zusätze H und L beziehen sich auf den hohen, bzw. niederen Teil einer Variablen bzw. eines Registers.

| Befehlsnr. | ursprüngliche abstrakte Einadreß befehlsfolge | Code auf Maschine mit der halben Wortlänge |
|------------|---|---|
| n | .D→R1 | .DL → R1L .DH → R1H |
| n+1 | .R1 AND .E→R1 | .R1L AND EL → R1L .R1H AND EH → R1H |
| n+2 | .F→R2 | .FL → R2L .FH → R2H |
| n+3 | .R1+.R2→ R1 | .R1L + .R2L → R1L, C .R1H + C + .R2H → R1H |
| n+4 | .R1 → G | .R1L → GL .R1H → GH |

Schwierig wird die Umsetzung, wenn die Registeranzahl nicht ausreicht. Daß man in manchen Fällen trotzdem guten Code erzeugen kann, zeigt das nächste Beispiel. Der Trick liegt darin, daß Code für das hohe Teilwort vorläufig gemerkt und erst bei Bedarf abgesetzt wird. Zwischen Befehl n und n+1 sowie zwischen n+4 und n+5 sind Kontextprüfungen nötig, die feststellen, ob das Ergebnis des niederen Auswirkungen auf den hohen Teil hat.

| Befehls-Nr. | ursprüngliche Einadreß befehlsfolge | Code auf Maschine mit der halben Wortlänge | später abzu- legender Code | Kommentar |
|-------------|-------------------------------------|--|----------------------------------|--|
| n | .D → R1 | .DL → R1 | .DH → R1 | |
| n+1 | .R1 AND .E → R1 | .R1 AND .EL → R1 | .R1 AND .EH → R1 | |
| n+2 | .F → R2 | .FL → R2 | .FH → R2 | |
| n+3 | .R1 + R2 → R1 | .R1 + R2 → R1, C | .R1 + C + R2 → R1 | |
| n+4 | .R1 → G | .R1 → GL, SAVE C .DH → R1, SAVE C .R1 AND .EH → R1, SAVE C .FH → R2, SAVE C .R1 + C + R2 → R1 .R1 → GH | | C muß erhalten bleiben erst jetzt wird der hohe Teil be- nötigt. Vorausge- setzt wird, daß die mit SAVEC C ge- kennzeichneten Befehle C nicht verändern |
| n+5 | .R1 + D → R1 | .GL → R1 .R1 + DL → R1, C | .GH → R1 .R1 + C + DH → R1 | R1 regenerieren |
| n+6 | .R1 → E | .R1 → EL, SAVE C .GH → R1, SAVE C .R1 + C + EH → R1 .R1 → EH | | gemerkter Code gemerkter Code |

Die nächste Stufe, der Codewandler, wandelt den maschineninternen Code in Assemblercode um. Es wird vorausgesetzt, daß auf der Zielmaschine ein Assemblerprogramm vorhanden ist. Wenn nicht, ist die Umwandlung in menschenlesbaren Code zwar nicht nötig, aber zumindest in der Testphase sehr angebracht.

LWEG-4 Portable Rechenergebnisse

Über die Länge eines Datenworts der abstrakten Dreiadreßmaschine werden keine Aussagen gemacht. Ebenso gibt es auch für die abstrakten Zwei- und Dreiadreßmaschinen keine Aussage über die Wortlänge. Bei der Umsetzung von der abstrakten auf reale Maschinen wird davon ausgegangen, daß ein Datenwort des abstrakten auf ein Datenwort des realen Rechners abgebildet wird. Dementsprechend können Aussagen über die Portabilität der Rechenergebnisse auf den realen Zielmaschinen höchstens dann gemacht werden, wenn von einer bestimmten Wortlänge auf der abstrakten und realen Maschine und einer bestimmten Darstellung der Operanden ausgegangen wird.

Zur Darstellung einer Gleitpunktzahl werden ein oder zwei Datenworte der abstrakten Maschine benötigt. Die Aufteilung auf Exponent und Mantisse und die Darstellung von Exponent und Mantisse ist weitgehend frei wählbar und damit an eventuell auf dem Zielrechner vorhandene Software anpaßbar. Das liegt daran, daß in den abstrakten Dreiadreßprogrammen Anpassungen weitgehend berücksichtigt sind, die vom Generator durchgeführt werden (Siehe dazu das Beispiel in Kapitel BSPL).

Ganze Zahlen werden in einem Datenwort der abstrakten Maschine abgespeichert. Für die Darstellung von negativen ganzen Zahlen wird eine Zweierkomplementdarstellung angenommen, was insbesondere die eindeutige Darstellung der Null betrifft.

Wenn abstrakte Programme wahlweise auf Ein- oder Zweierkomplementmaschinen umgesetzt und gleiche Rechenergebnisse geliefert werden sollten, so wären einige recht aufwendige Anpassungen ins abstrakte Programm aufzunehmen. Bei realen Zweierkomplementmaschinen wäre nach arithmetischen Befehlen die negative Größtzahl auszuschließen. Ein geeignetes Programmstück dazu ist:

```
IF ZWEIERKOMPLEMENT AND PORTABLE-RECHENERGEBNISSE THEN
    NEG ... → , V      /* Nur Überlaufanzeige des Negationsbefehls
                        interessant*/
    JV → UEB           /* Ausschalten der negativen Größtzahl */
FI
```

Weitere Schwierigkeiten bereitet die negative Null. Nach boole'schen Befehlen könnte auf einer Einerkomplementmaschine die Sprungbedingung "Ergebnis = 0" auch erfüllt sein, wenn die negative Null (lauter Einsen) das Ergebnis bildet. Umgekehrt, d.h. wenn die Bedingung nur bei der positiven Null erfüllt ist, müßte der Befehl

JEQ → NULL

durch eine explizite Abfrage auf die negative Null ergänzt werden.

Man erkennt, daß dem Ersteller abstrakter Programme viel Arbeit aufgebürdet würde. Manche Schreibaarbeit ließe sich reduzieren, wenn der Generator solch kritische Programmstellen erkennen würde und dann selbständig vom Ersteller des abstrakten Programms bereitgestellte Makrokörper einsetzte.

Probleme bereiten auch die Ganzzahl-Division und die arithmetischen Schiebebefehle nach rechts. Bei der Division soll grundsätzlich der Betrag abgerundet werden. Beim arithmetischen Rechtsschieben bleiben "rechts herausfallende Bits" unberücksichtigt. Das hat zur Folge, daß der Betrag positiver und negativer arithmetisch rechts verschobener Zahlen unterschiedlich gerundet wird. Um dem Ersteller abstrakter Programme die Wahl zu lassen, ob er an einer bestimmten Programmstelle auf Portabilität der Rechenergebnisse verzichten kann, oder ob die Rundung wie oben definiert sein muß, kann er zwischen zwei verschiedenen Divisions- und arithmetischen Rechtsschiebebefehlen mit exakter oder beliebiger Rundung unterscheiden. Wählt er die exakte Rundung, so ist für anders rundende Maschinen bei der Umsetzung statt eines einfachen Befehls eine kleine Ersatzbefehlsfolge abzusetzen.

LOES Lösung

LOES-1 Abstrakte Maschinen

LOES-1.1 Assemblersprache der abstrakten Dreiadreßmaschine

Wie in LWEG gesagt, werden die algorithmischen Laufzeitprogramme in der Assemblersprache einer abstrakten Dreiadreßmaschine formuliert. Sie werden je nach Generierparameter für eine abstrakte Zwei- oder Einadreßmaschine umgesetzt. Die Umsetzprogramme gehen von einer dokumentationsfreundlichen Darstellung der abstrakten Programme auf Lochkarten bzw. lochkartenorientierten Dateien aus. Ein Befehl wird durch eine Abschlußzeichenfolge (z.B. \$ Zeichen) und beliebigen Kommentar abgeschlossen; er darf auf mehrere Lochkarten verteilt sein. Kommentarzeilen beginnen den Kommentar mit einer Einleitungszeichenfolge an beliebiger Stelle. Zur Auflockerung des Druckbildes dürfen nach jedem Befehl Lochkarten eingeschoben werden. Die Darstellung der Befehle ist zur Erzielung eines übersichtlichen Druckbildes formatfrei.

Die abstrakte Dreiadreßmaschine kennt ausführbare Anweisungen mit Optimierungszusätzen und Pseudobefehle. Ausführbare Anweisungen können mit einer Marke versehen werden. Sie steht durch Doppelpunkt getrennt vor dem Befehl.

Die ausführbaren Anweisungen umfassen

| | |
|-----------------------|--------------------------------------|
| Transferbefehle: | Quelle1 → Senke |
| Dreioperandenbefehle: | Quelle1 Operator Quelle2 → Senke |
| Sprungbefehle: | Sprungoperator → Sprungziel |
| Unterprogrammaufrufe: | CALL ... |
| Sonderbefehle: | monadischer Operator Quelle1 → Senke |
| | Quelle1 Sonderoperator → Senke |

Die Schreibweise mit dem Symbol "→", das bei Sprungbefehlen als "nach" und sonst als "geht nach" zu lesen ist, erscheint anschaulicher, als der von dem IEEE Gremium Task P 694/D11 erarbeitete Standardisierungsvorschlag [FIS79]. Dort sind, wie auch sonst weit verbreitet, Quelle1, Quelle2 und Senke als Tripel notiert, wobei die Reihenfolge maßgebend ist. Ebenfalls anschaulich wäre die Notation mit dem "ergibt sich aus"-Zeichen (z.B. "==" bei ALGOL oder "=" bei FORTRAN).

Als Quellen sind Konstante, Marken und Referenzen auf Konstante, Variable und Parameter zugelassen. Eine Referenz besteht aus einem Namen und der Angabe einer der folgenden Adressierungsarten:

direkt

indirekt

doppelt indirekt

indiziert

indiziert indirekt

indirekt pre-dekrement

doppelt indirekt pre-dekrement

indirekt post-inkrement

doppelt indirekt post-inkrement

Eine genaue Beschreibung findet der Leser im Anhang A1.1.

Variablenamen haben den Aufbau, den man von höheren Programmiersprachen her kennt. Parameter sind durch das Schlüsselwort "PAR", dem eine Nummer folgt, gekennzeichnet. Konstante und Marken sind als Senke nicht zugelassen. Schleifenzähler, als besondere Variablen, sind durch das Schlüsselwort "SCHZ", gefolgt von einer Nummer, gekennzeichnet. Ihre maximale Anzahl ist implementationsabhängig. Sie wurden zur Optimierungsmöglichkeit eingeführt, da manche Rechner spezielle Hardwarevorkehrungen zur Realisierung von Schleifen besitzen: Der bedingte Sprungbefehl "Dekrementiere das i-te Schleifenzählerregister und springe, falls Inhalt gleich Null" (DiJNZ), ist weit verbreitet. Allerdings ist oft nur ein Schleifenzähler vorhanden. Der DIETZ 621 kennt einen D0-Befehl, der (leider nur) die folgende Anweisung mehrfach ausführt. Die DiJNZ-Befehle sind auch im Befehlsvorrat der abstrakten Ein- bis Dreiaдресmaschine enthalten. Zu jedem Schleifenzähler gibt es den entsprechenden Dekrementierebefehl. Diese Befehle dürfen nur zur Schleifenbildung verwendet werden. Dazu muß das Sprungziel natürlich vorher vereinbart worden sein, d.h. es sind keine Vorwärtssprünge erlaubt. Ein DiJNZ Befehl ist nicht sinnvoll, ohne daß vorher dem i-ten Schleifenzähler ein definierter Wert zugewiesen wurde.

Im Anhang A1 findet der Leser die Beschreibung der Operatoren, der Transfer- und Sonderbefehle, der Sprungbefehle und -bedingungen. Unterprogrammen wird Abschnitt LOES-1.3 gewidmet. Hier seien nur noch die

zur Optimierung eingeführten Ergänzungsmöglichkeiten "L" und "U" zu den Transfer-, Dreioperanden- und Sonderbefehlen beschrieben. Der Zusatz L bedeutet gleichzeitiges Löschen des Operanden.

Beispiel:

Die Befehlsfolge

$..A, L + .B \Rightarrow C$

$.D + .E, L \Rightarrow F$

$.G, L + ..H, L \Rightarrow I$

$..J, L \Rightarrow K$

führt zum selben Ergebnis wie die Befehlsfolge

$..A + .B \Rightarrow C$

$\emptyset \Rightarrow .A$

$.D + .E \Rightarrow F$

$\emptyset \Rightarrow E$

$.G + ..H \Rightarrow I$

$\emptyset \Rightarrow G$

$\emptyset \Rightarrow .H$

$..J \Rightarrow K$

$\emptyset \Rightarrow .J$

Dieses gleichzeitige Löschen dient neben der eventuellen Optimierungsmöglichkeit hauptsächlich zur Verkürzung der abstrakten Programme. Auf der SIEMENS 306 wurde der Transfer- und ODER-Befehl mit gleichzeitigem Löschen wohl hauptsächlich wegen der damit verbundenen Ununterbrechbarkeit geschaffen. Auf der abstrakten Maschine sagt das L nichts über Unterbrechbarkeit aus.

Der Zusatz U bedeutet, daß die zur Referenzierung benutzte Speicherzelle nach Ausführung des Befehls undefiniert ist.

U kann nach jeder Quelle stehen.

Beispiele: $..A, U + .B, U \Rightarrow C$ bedeutet, daß ab jetzt der Inhalt der Speicherzellen A und B undefiniert ist, der ursprünglich durch $..A$ referenzierte Operand bleibt erhalten. Auf ihn kann aber nicht mehr über $..A$ zugegriffen werden, ohne A vorher neu zu laden.

Die Angabe der Optimierungshinweise L und U bei einer Quelle schließen sich gegenseitig aus. Bei einem Dreioperandenbefehl kann jedoch die eine Quelle den Zusatz U und die andere den Zusatz L tragen. Beide Zusätze werden von der Hauptumsetzstufe ausgewertet. Der Zusatz L dient zur Optimierung, wenn die reale Zielmaschine über einen entsprechenden Befehl verfügt. Der Zusatz U wirkt, wenn sich zufällig die durch U gekennzeichnete Variable in einem Register befindet, das dann für anderweitige Verwendung freigegeben werden kann. Der Zusatz U wäre nicht unbedingt nötig, denn eine Datenflußanalyse könnte in den meisten Fällen ¹⁾ feststellen, wann eine Variable nicht mehr benötigt wird, eine Information, die der abstrakte Programmierer aber schon früher hat. Die Auswertung dieser Information erspart also die komplizierte Datenflußanalyse.

Der Zusatz U hat noch eine weitere Bedeutung. An sich nicht sehr sinnvolle Befehle der Art

$$.X, U \rightarrow X$$

deuten darauf hin, daß der Wert der Variablen X ab sofort nicht mehr benötigt wird. Ist X als wichtige Variable ²⁾ gekennzeichnet, verliert sie gleichzeitig diese Eigenschaft.

- 1) Es gibt trickreiche Programme, bei denen sich diese Information vor einer Datenflußanalyse versteckt. Eine Analyse des folgenden Programms muß A (ohne Beachtung des Optimierungszusatzes U) weiterhin als wichtig ansehen, denn A könnte ja beim zweiten Schleifendurchlauf benötigt werden:

```

    Ø → I  $
M1: ...
    :
    .I → ,Z,GT $
    JGT⇒M2  $
    .A,U +.B⇒C  $
M2: .I + 1⇒I  $
    JEQ⇒M1  $

```

- 2) wichtige Variablen sind besonders deklariert. Siehe LOES-1.2

LOES-1.2 Aufbau eines abstrakten Laufzeitprogramms

Bevor mit der Programmierung begonnen werden kann, muß noch etwas zum Aufbau abstrakter Laufzeitprogramme gesagt werden. Laufzeitprogramme sind Unterprogramme. Sie beginnen mit dem Kopfbefehl

```
SUBIN : Ø : <programmname>[SAVE <Ergebnisanzeigen>][liste_formaler_Parameter ]
                                           <befehlsende>
```

Das Schlüsselwort SAVE besagt, daß Parameterübergabe oder-übernahme die aufgeführten Ergebnisanzeigen des aufrufenden Programms nicht zerstören dürfen, da sie als Parameter mit i.A. schneller Übergabe angesehen werden können. Die Liste der formalen Parameter enthält Eingabe, Ausgabe oder gemischte Parameter. Werden Adressen übergeben, sind diese besonders gekennzeichnet.

```
liste_formaler_Parameter ::=
    { IN [i]|OUT [i]|INOUT [i] | INADR [i]|OUTADR [i] | INOUTADR [i] } ...
                                           <befehlsende>
```

Die Programme könnten etwas dokumentationsfreundlicher sein, wenn Parameter durch frei wählbare Namen statt durch Nummern identifiziert wären. Der Kopfbefehl wird bis zum Codewandler unverändert übernommen. Erst diese Stufe bildet ihn beispielsweise auf ein Parameterübernahmeprogramm ab.

Das Befehlsende besteht aus einem Endezeichen und Kommentar bis zum Zeilenende.

An den Kopfbefehl schließt sich eine Liste der in einem abstrakten Programm verwendeten symbolischen Namen und Schleifenzähler an.

Die Deklarationen erleichtern eine eventuelle Speicherplatzverwaltung, erhöhen die Programmiersicherheit und können bei einer Datenflußanalyse nützlich sein. ¹⁾

An die Deklarationsliste schließen sich ausführbare abstrakte Dreiadreß-befehle oder zu Abschnitten zusammengefaßte Dreiadreß-befehle an. Diese Abschnitte haben nichts mit der Blockstruktur blockorientierter Sprachen

¹⁾ In der Pilotimplementation sind die Deklarationen durchnummeriert. Die Hauptumsetzstufe fügt nämlich zur Erleichterung der Adressierung über Indexregister jeder Variablen diese Nummer als Adreßabstandszahl bei. (Siehe Beispielprogramm in Abschnitt BSPL-3). Wenn Adressen in eine Speicherzelle passen, stimmt diese Zahl mit der Deklarationsnummer überein.

wie ALGOL60 oder PEARL zu tun. Sie dienen vielmehr als Umsetzhinweise für die Hauptumsetzstufe.

Abschnitte beginnen mit BEGIN: <befehlsende>
und enden mit ENDE: <befehlsende>

Bei Beginn und Verlassen eines Abschnitts werden alle sich augenblicklich nur in Registern befindlichen Variablen in die mittels der Deklarationsanweisung eingerichteten Speicherzellen transferiert. Zur Erleichterung einer günstigen Registerverwaltung kann am Abschnittsanfang auf wichtige, d.h. häufig gebrauchte Variablen hingewiesen werden. Diese werden bis zum Abschnittsende bevorzugt in Registern gehalten. Durch die Reihenfolge wird eine Priorität festgelegt. Der Hinweis auf eine wichtige Variable erfolgt durch Nennung des Variablennamens hinter dem Schlüsselwort "IMP:".

Bei der Implementation ist zu beachten, daß noch ausreichend viele Variablen der Verwaltung der Hauptumsetzstufe unterstellt bleiben. Die Entscheidung, welche Variablen am häufigsten benutzt werden, kann am besten der Ersteller des abstrakten Programms treffen. Eine Datenflußanalyse kann exakte Ergebnisse nur liefern, wenn sie abschätzen könnte, wie oft Schleifen durchlaufen werden. Die Benutzung von Variablen wird meist nicht gleichmäßig über das gesamte Programm verteilt sein. Deshalb darf ein abstraktes Programm in mehrere Abschnitte eingeteilt sein.

Ein Laufzeitprogramm wird mit einem Rücksprungbefehl verlassen. Da der Rücksprung in den meisten Fällen mit Organisation verbunden ist, ist nur ein Rücksprungbefehl erlaubt. Das vorzeitige Verlassen erfordert einen Sprungbefehl auf eine Marke vor dem Rücksprungbefehl mit dem reservierten Namen RETURN. Die Syntax für den Rücksprungbefehl lautet:

```
<Rücksprungbefehl> ::= RET: Ø :<programmname>
                        SAVE <Ergebnisanzeigen> <Liste_formaler_Parameter>
                        <befehlsende>
```

Die Liste der formalen Parameter ist redundant, da sie bereits im Kopf angegeben ist. Sie erleichtert aber die weitere Umsetzung. Die Übereinstimmung beider Listen zu überprüfen, ist Aufgabe der Syntaxanalyse. Je nach Implementation des Codegenerators verbirgt sich dahinter eine mehr oder weniger aufwendige Befehlsfolge. Bei der PEARL-Implementation auf der SIEMENS 310 [TRA80] müssen Hardwareanzeigen gesetzt

sein. Bei der Implementation unter Einsatz des portablen Codegenerators von Pelz [PEL81] müssen Wahrheitswerte in speziell vorgesehene Zellen eingeschrieben sein.

Ein zusammen zu übersetzender Stapel von abstrakten Programmen wird mit einer implementationsbedingten Programmende-Zeichenfolge abgeschlossen. Zur Trennung zweier abstrakter Programme genügt der SUBIN-Befehl des nachfolgenden Programms.

LOES-1.3 Unterprogramme

Abstrakte Laufzeitprogramme können ihrerseits Unterprogramme aufrufen. Dabei ist eine Hierarchie erklärt. Routinen, die der Codegenerator aufruft, bekommen die Nummer Null zugeordnet, worauf die Null im Kopf- und Rücksprungbefehl hinweist (SUBIN:Ø:...) . Andere Routinen erhalten je nach Schachtelungstiefe eine höhere Nummer. Unterprogramme werden mit einem CALL-Befehl aufgerufen:

<CALL-Befehl>:=

CALL:<hierarchiestufennummer>: <unterprogrammname>[SAVE <ergebnisanzeigen>]
 (<eingabeparameterliste>) → (<rückgabeparameterliste>)<befehlsende>

Die Unterprogramme selbst haben den gleichen Aufbau wie ein in LOES-1.2 beschriebenes Laufzeitprogramm, außer der anderen Hierarchiestufennummer im Kopfbefehl. Diese Nummer erlaubt, Unterprogramme unterschiedlicher Hierarchie bezüglich Parameter- und Variablenadressierung unterschiedlich zu übersetzen. Dadurch wäre beispielsweise möglich, daß in Laufzeitunterprogrammen der Stufe Null Variable über Indexregister adressiert werden und in eventuell aufgerufenen Hilfsprogrammen Variable einige taskspezifische Hilfszellen belegen können, die relativ zum Taskanfang adressiert werden. Zum Zweck unterschiedlicher Übersetzung einzelner ausgewählter Programme kann die Hierarchienummer, nun besser Übersetzungsmodusnummer genannt, leicht mißbraucht werden, indem sie durch eine bisher noch nicht vergebene Nummer ersetzt wird. Wenn sich beispielsweise bei kurzen und oft aufgerufenen Programmen aufwendige Mechanismen zur Erzielung der Wiedereintrittsfähigkeit nicht lohnen, oder wenn einige Unterprogramme, vielleicht Zeichenketten verarbeitende, besonders effizient werden sollen, können diese anders als die übrigen übersetzt werden. Vielleicht sollen mehr Register zur Verfügung gestellt werden, oder die Variablen werden

direkt adressiert, statt wie sonst über Indexregister. Die Übersetzungsmodusnummern kann der Anwender des portablen Systems durch einen Editor einbringen. Ebenso ließe sich ein Spezialeditierprogramm beispielsweise ins Generierparameter-Aufnahmeprogramm einbauen.

Zur Anpassung an verschiedene Implementationswünsche müssen nicht alle Parameter von Unterprogrammen einer Hierarchiestufe auf gleiche Weise adressiert sein. So wäre beispielsweise möglich, wenn auch nicht unbedingt sinnvoll, die Adresse einer ersten Zeichenkette in einem Register zu übergeben, die Länge vor dem ersten Wort der Zeichenkette, die Adresse der zweiten Zeichenkette in einem zweiten Register und deren Länge in einem weiteren Register. Bei Bitkettenunterprogrammen beispielsweise mag die Adressierung ganz andersartig aussehen und bei Gleitpunktarithmetik wiederum anders.

Der Unterprogrammname im Aufruf kann nur direkt angegeben werden, da zielmaschinenabhängige Unterprogramme bereits durch den Generator eingesetzt werden. Eine laufzeitabhängige Auswahl eines Unterprogramms über indirekte Adressierung wird aus Gründen der Übersichtlichkeit eines abstrakten Programms abgelehnt.

"SAVE Ergebnisanzeigen" weist darauf hin, daß an das Unterprogramm Ergebnisanzeigen übergeben werden. Die Eingabe- bzw. Rückgabeparameterlisten enthalten Quellen und Senken wie normale Dreiadreß befehle.

Über den Parameterübergabe- oder Übernahmemechanismus ist auf der abstrakten Maschine bewußt nichts ausgesagt. Das läßt die Möglichkeit einer optimalen Anpassung an die jeweilige Zielmaschine zu. Das bedeutet natürlich auch, daß die algorithmischen Laufzeitprogramme kaum Unterstützung bei der Parameterbehandlung von Anwenderprozeduren bieten. Nur ein abstraktes Programm zum Transfer von Datenbereichen könnte Verwendung finden, wenn die Hardware des Zielrechners keine Blocktransferbefehle kennt.

LOES-1.4 Abstrakte Ein- und Zweiadreß maschinen

Im Abschnitt LOES-1.1 ist eine abstrakte Dreiadreß maschine vorgestellt worden, in deren Assemblersprache Laufzeitprogramme erstellt werden. Wie in LWEG-3 (Konzept der Umsetzung) beschrieben, werden die Dreiadreß programme maschinell auf abstrakte Ein- oder Zweiadreß - maschinen umgesetzt. Wegen der Vielzahl von Generierparametern gibt es auch eine Vielzahl von abstrakten Ein- und Zweiadreß maschinen. Innerhalb jeder Klasse gibt es natürlich starke Gemeinsamkeiten.

Einadreß maschinen kennen wie die abstrakte Dreiadreß maschine Transferbefehle

Dreioperandenbefehle

Sonderbefehle und

Sprungbefehle.

Bei den Transferbefehlen der Einadreß maschine muß Quelle oder Senke ein Register sein:

Quelle \rightarrow Register oder:
Registerinhalt \rightarrow Senke

Die Verknüpfungsbefehle haben folgende Gestalt:

Registerinhalt \otimes Quelle \rightarrow Register

und die inversen Verknüpfungsbefehle:

Quelle \otimes Registerinhalt \rightarrow Register

Dabei bedeutet \otimes den boole'schen, arithmetischen oder Schiebeoperator. Die Existenz des inversen Befehls ist für jeden Operator gesondert als Generierparameter anzugeben.

Manche reale Maschinen kennen Beispiele, bei denen sofort das Ergebnis einer Verknüpfung in den Speicher zurücktransfertierte wird.

Registerinhalt \otimes Quelle \rightarrow Register & Senke. ¹⁾

Bei der Siemens 306 existiert diese Möglichkeit nur beim Additionsbefehl, bei der TRIUMPH-ADLER 1000 bei allen Verknüpfungsbefehlen.

Damit lassen sich Dreiadreß befehle, bei denen Quelle gleich Senke ist, besonders gut umsetzen. Jedoch ist eine automatische Verwendung solcher Befehle nicht zu empfehlen, wie das folgende Beispiel lehrt:

¹⁾ & lies als: sowohl, als auch

M1: .A + .B → A \$

M2: .A + .C → A \$

M3: .A,U + .D → E \$

Wegen der erforderlichen Kontextprüfung und dem Bestreben nach einfachen Protabilitätsverfahren wurde auf die Aufnahme von Befehlen mit gleichzeitigem Rücktransfer verzichtet.

Bei den Sonderbefehlen muß Quelle und Senke das gleiche Register sein.

Für Quelle und Senke gibt es die gleichen Möglichkeiten wie bei der Dreiadreßmaschine (siehe Abbildung A1.4/1 im Anhang). Falls Generierparameter dies zulassen, gibt es aber darüber hinaus als Quelle und Senke die in der Abbildung LOES-1.4/1 gelisteten Adressierungsmodi über Register.


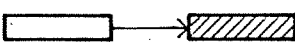
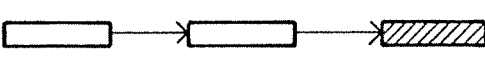
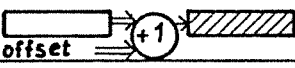
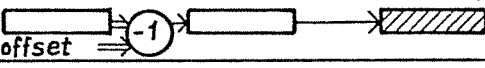
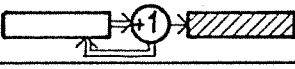
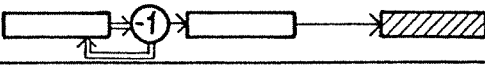
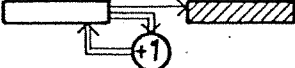
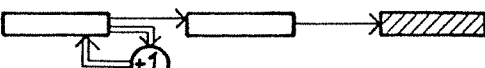
| Registerart | Darstellung als Quelle | Wirkungsweise |
|--------------------------------------|------------------------|---|
| Rechenregister | . Ri | Ri  |
| Adreßregister | .. Rj | ..Rj  |
| Adreßregister | ... Rj | ...Rj  |
| Indexregister | .. Rx offset | Rx  offset |
| Indexregister | ... Rx offset | Rx  offset |
| Register m. predecrementmöglichkeit | .↓. Rk | Rk  |
| Register m. predecrementmöglichkeit | ..↓. Rk | Rk  |
| Register m. postincrementmöglichkeit | .. Rk 1 | Rk  |
| Register m. postincrementmöglichkeit | ... Rk 1 | Rk  |

Abb. LOES-1.4/1: Adressierungsmöglichkeiten über Register
Legende zur Spalte Wirkungsweise siehe Abb. ZIEL-3/3.

Durch Generierparameter muß bei realen Zielmaschinen, bei denen nicht alle Register in den gleichen Adressierungsarten ansprechbar sind, noch zwischen verschiedenartigen Registern unterschieden werden. In Abbildung LOES-1.4/1 kommt dies durch die Indices i, j, k, x zum Ausdruck. Die Operatoren können direkt von der Dreiadreßmaschine übernommen werden.

Die abstrakten Zweiadreßmaschinen unterscheiden sich von den abstrakten Dreiadreßmaschinen hauptsächlich dadurch, daß in den Verknüpfungsbefehlen die Senke mit einer der Quellen identisch ist. Der Normalfall eines Verknüpfungsbefehls hat die Gestalt:

Quelle \otimes Senke \rightarrow Senke,

wobei Quelle und/oder Senke auch ein Register sein darf.

LOES-2 Beschreibung der Umsetzung

LOES-2.1. Implementationstechnik

Im ersten Teil dieses Kapitels sind die abstrakten Maschinen vorgestellt worden. Der zweite Teil beschreibt nach einigen Bemerkungen über die Implementationstechnik die Hauptstufe der Umsetzung abstrakter Dreiadreßprogramme auf abstrakte Einadreßprogramme.

Zur Implementation der in Kapitel LWEG-3 (Konzept der Umsetzung) erwähnten Umsetzprogramme ist eine SIEMENS 306 mit 32K Worten Kernspeicher und einer Magnetplatteneinheit zur Verfügung gestanden. Auf diesem Rechner existieren Compiler für FORTRAN, ALGOL, PEARL Stufe 1 und eine einfache Systemimplementierungssprache ERLAN [ROE78] (Erlanger _Language).

Der ERLAN-Compiler liegt in ERLAN selbst und in einem CIMIC-C Dialekt vor und läßt sich daher mittels Bootstrap portieren. Die ERLAN-Laufzeitbibliothek wurde später noch durch ein in ERLAN geschriebenes Standard E/A- und ein sequentielles Filehandlingpaket ergänzt [ROE78], welches auf Assemblerunterprogramme zurückgreift, und ist somit ebenfalls portabel. Da die vorgesehene Implementation aber nur als Prototyp angesehen werden darf, ist die Neuerstellung in einer modernen, weit verbreiteten Systemsprache (z.B. PASCAL) einem Portieren von ERLAN vorzuziehen. Die Umsetzprogramme sind noch durch ein kurzes Dialogprogramm, das verschiedene Dateinamen erfragt, und ein Ablaufsteuerungsprogramm ergänzt.

Die Ablaufsteuerung ist nötig, da die einzelnen Programme (insgesamt ca. 70K Worte gebunden, ungebunden ca. 40K Worte) nicht zusammen in den Speicher passen und außerdem ein ERLAN-Programm auf dem vorhandenen Implementationsrechner SIEMENS 306 nicht größer als 16K Worte (à 24 Bit) werden darf. Die einzelnen Stufen von Hand zu laden und starten erscheint zu mühevoll. Die Schnittstelle zwischen dem Steuer- und dem Dialogprogramm bildet ein Datenpuffer im Steuerprogramm. Die Schnittstelle zwischen den Umsetzstufen einschließlich Generierparameteraufnahme und Protokollprogramm bilden Dateien, deren Namen sich die einzelnen Programme aus einem als COMMON-Bereich fungierenden Datenpuffer im Steuerprogramm holen.

Folgende Dateien werden benötigt:

1. Generierparameter in Rohform. Falls diese über Lochkarte oder Blattschreiber eingegeben werden, werden sie auf dieser Datei gespeichert.
2. Generierparameter, die das Generierparameteraufnahmeprogramm aus den Eingabedaten erzeugt und die zur Auswertung durch den Generator bestimmt sind.
3. Generierparameter, welche die auf den Generator folgenden Umsetzstufen auswerten. Die Generierparameter nach 1 und 2 überschneiden sich.
4. Abstraktes maschinenabhängiges Dreiadreßprogramm (vor Generator-durchlauf).
- 5.-6. Abstraktes maschinenangepaßtes Dreiadreßprogramm nach Durchlauf des Generators, der Optimierungsstufe und der Direktoperandenbehandlung.
- 7.-8. Abstraktes Einadreßprogramm in interner Darstellung nach Hauptumsetzstufe und der Stufe zur Wortlängen Anpassung.
9. Programm in Assemblersprache der Zielmaschine.

LOES-2.2 Umsetzverfahren

Das Konzept der Umsetzung von den parameterisierten über die maschinenangepaßten bis zu den bindefähigen Laufzeitprogrammen ist bereits im zweiten Kapitel vorgestellt worden. Die Abbildung LOES-2.2/1 zeigt die einzelnen Umsetzstufen noch einmal. Darin sind die in der Pilotimplementation realisierten Teile durch besondere Rahmung hervorgehoben. Die Abbildung enthält auch die Symbole für Schnelldruckerausdrucke, um die in BSPL-2 gezeigten Protokolle besser einordnen zu können. Die meisten Stufen sind bei der Vorstellung des Konzepts bereits hinreichend beschrieben worden. In den folgenden Abschnitten soll über die Hauptumsetzstufe berichtet werden. Die Stufen davor modifizieren den Code einer abstrakten Dreiadreßmaschine. Die Hauptumsetzstufe erzeugt, von Generierparametern beeinflusst, Code in einer Assemblersprache, die bis auf die SUBIN-, RET- und CALL-Befehle eine Unter-

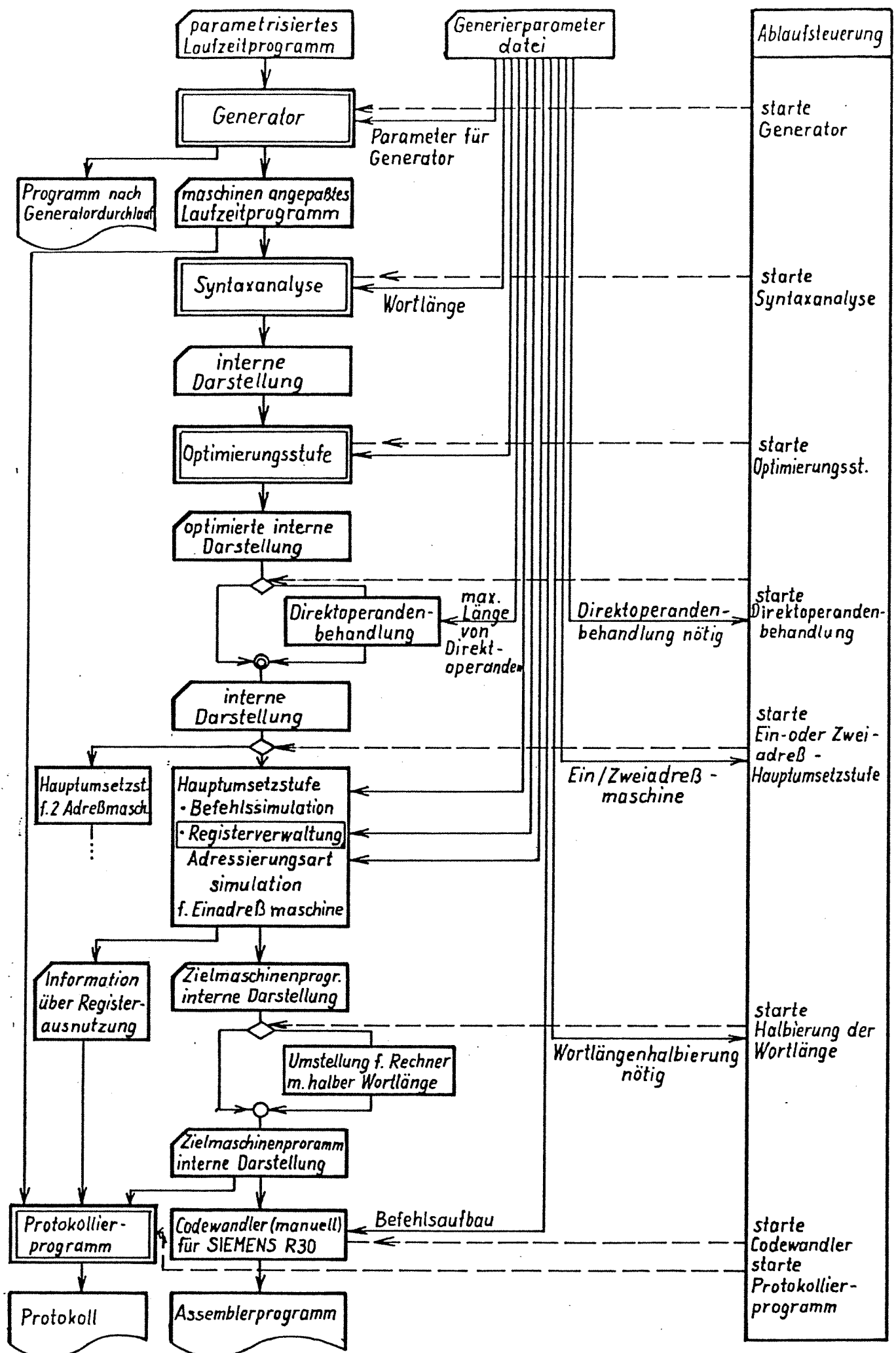


Abb. LOES-2.2/1: Schematische Übersicht über das stufenweise Umsetzverfahren

menge der Assemblersprache der realen gewünschten Zielmaschine ist. (Vorausgesetzt, diese Zielmaschine besitzt eine ausreichende Wortlänge. Andernfalls gilt das Gesagte nach der Stufe zur Halbierung der Wortlänge.)

Der erzeugte Code hat nur noch nicht die syntaktische Form, die der Assembler der Zielmaschine verlangt. Diese Umformung führt der anschließende Codewandler durch. Ziel der Hauptumsetzstufe ist, daß sich die Untermenge möglichst weitgehend mit der Zielmaschine deckt, d.h. daß der reale Rechner gut ausgenutzt wird. Dieser Abschnitt gibt einen Überblick über den Aufbau des implementierten Programms. Man erhält ihn am besten durch Flußdiagramme bzw. Struktogramme. Hier werden nur Struktogramme betrachtet, da sie einen strukturierten Programmaufbau unterstützen und ERLAN entsprechende Sprachmittel besitzt. Die Bilder LOES-2.2/2 und /3 zeigen die Umsetzung von abstrakten Dreioperandenbefehlen auf abstrakte Einadreßmaschinen. Das Hauptprogramm besteht aus nur wenigen ERLAN - Programmzeilen. Das zugehörige Struktogramm zeigt Abbildung LOES-2.2/2.

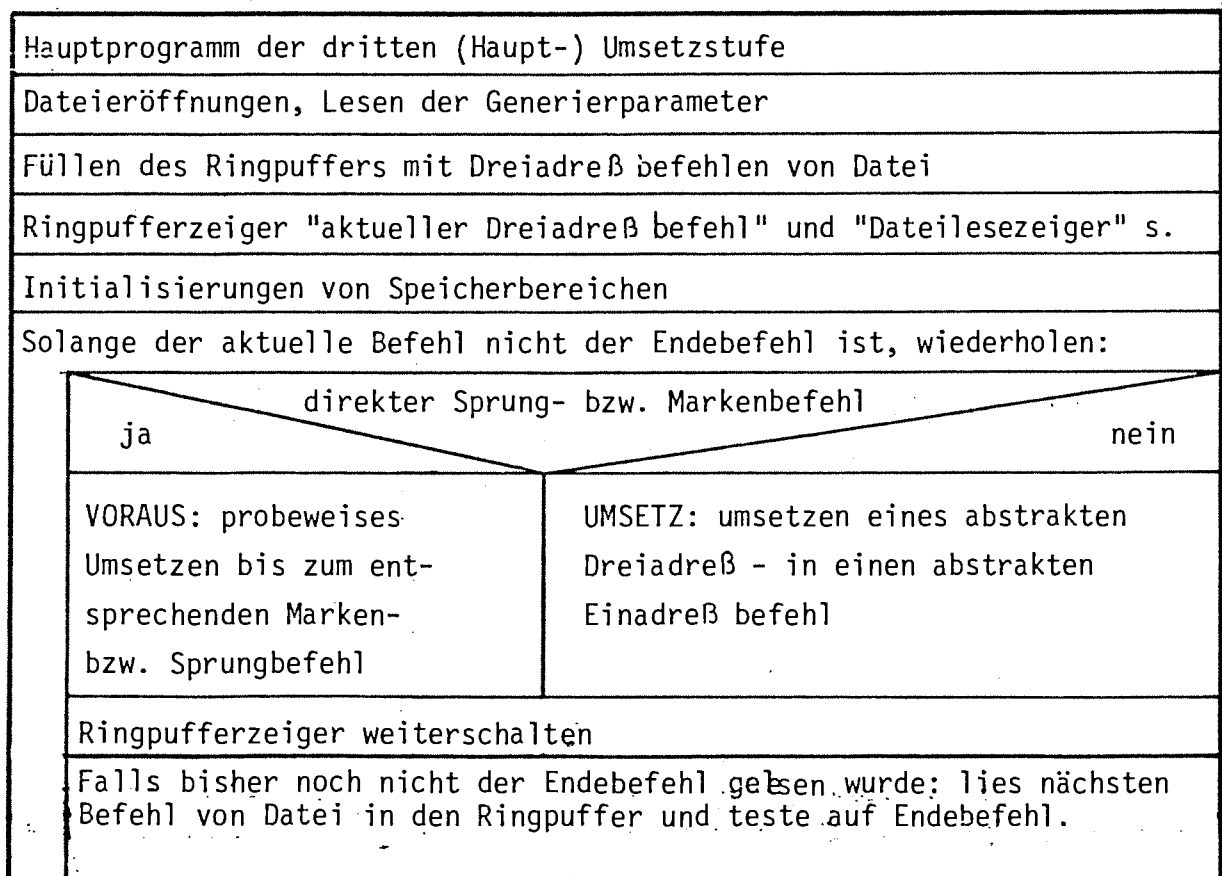


Abb. LOES-2.2/2: Struktogramm der Hauptumsetzstufe

Von der Prozedur UMSETZ soll nur das Kernstück betrachtet werden, das sich mit der Umsetzung eines Dreioperandenbefehls beschäftigt. Zuerst wird getestet, ob sich Quelle1 oder Quelle2 in einem Register befindet. Abhängig vom Ergebnis wird eines von vier Unterprogrammen ausgewählt. Dies zeigt das Struktogramm in Abbildung LOES-2.2/3.

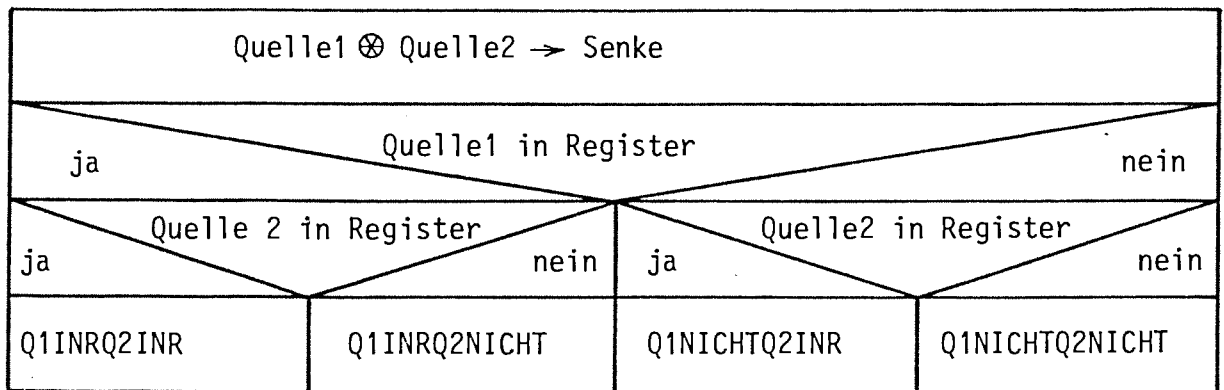


Abb. LOES-2.2/3: Umsetzung von Befehlen der Art Quelle1 \otimes Quelle2 \rightarrow Senke

Wenn sich bei Befehlen der Art

$$.A + .B \rightarrow C$$

die Senke zufällig in einem Register befindet und außerdem Senke=Quelle1 und Senke=Quelle2, wird dieses Register als frei gekennzeichnet und steht der Registerverwaltung zur Verfügung. Dieser Fall ist nur in linearen Programmstücken ein Programmierfehler.

Die vier Unterprogramme Q1INRQ2INR bis Q1NICHTQ2NICHT sind in Anhang A4 beschrieben.

Sie werten eine Reihe von Generierparametern aus.

- a) Anzahl der zur Verfügung stehenden Register
- b) Möglichkeit des Transfers zwischen Registern
- c) Möglichkeit von Verknüpfungen zwischen Registern
- d) Existenz des zu jedem Befehl inversen Befehls.

Weitere Details der Programmierung sollen den Leser nicht belasten. Die beiden folgenden Abschnitte sollen stattdessen nur die Prinzipien der Registerverwaltung und der Simulation nicht vorhandener Adressierungsarten zeigen.

LOES-2.3 Registerverwaltung

Bei der Umsetzung von der abstrakten Drei- auf die abstrakte Einadreßmaschine werden Register-Abspeicheroperationen so lange wie möglich aufgeschoben in der Hoffnung, daß sie möglicherweise ganz entfallen können. Das ist am Beispiel der Umsetzung der Dreiadreß befehlsfolge

D01 : .Y + .Z \Rightarrow X

D02 : .X + .Y \Rightarrow X

zu sehen.

Dij und Eij (ij = 0,1,2...) soll hier nicht als Marke sondern als Befehlsnummer eines Drei- oder Einadreß befehls verstanden werden.

Sei Ri ein freies Rechnerregister. Wird der Befehl D01 zusammenhanglos übersetzt, ergeben sich die folgenden drei Einadreß befehle

E01 : .Y \Rightarrow Ri

E02 : .Ri + .Z \Rightarrow Ri

E03 : .Ri \Rightarrow X

Der letzte Speicherbefehl kann zusammen mit einem eventuell folgenden Ladebefehl entfallen. Der Ladebefehl muß nicht direkt anschließen, wenn nur das Register Ri zwischenzeitlich nicht umgeladen wird. Dies vorausgesetzt, erhält man die folgende Einadreß befehlsfolge. Die in eckiger Klammer stehenden Befehle sind überflüssig.

E01: .Y \Rightarrow Ri

E02: .Ri + .Z \Rightarrow Ri

E03: [.Ri \Rightarrow X]

⋮

E04: [.X \Rightarrow Ri]

E05: .Ri + .Y \Rightarrow Ri

E06: .Ri \Rightarrow X

Die gleiche Ersparnis in der Einadreß befehlsfolge erhält man, wenn der Befehlszusatz U (für undefiniert) ausgenützt wird, wie im Beispiel der folgenden Dreiadreß befehlsfolge:

D11: .Y + .Z \Rightarrow X

⋮

D12: .X,U + .Y \Rightarrow Z

Statt E06 wird hier der Befehl E16 .Ri \Rightarrow Z abgesetzt.

Dieselbe Einadreß.befehlsfolge erhält man, wenn der Dreiadreß.befehl D12 durch

D22: $.Y + .X, U \Rightarrow Z$

ersetzt wird, weil bei der Umsetzung die Kommutativität der Addition erkannt wird. Bei nichtkommutativen Operatoren könnte es bei realen Einadreßmaschinen z.B. statt des normalen Subtraktionsbefehls

$.R_i - .Z \Rightarrow R_i$ den inversen Befehl

$.Z - .R_i \Rightarrow R_i$ geben, was wiederum zur Einsparung des Lade- und zuweilen auch des Abspeicherbefehls führt.

Beispiel:

Steht statt D02

D32: $.Y - .X \Rightarrow X$

so kann der Befehl wie oben auf zwei Befehle abgebildet werden.

E34: $[.X \Rightarrow R_i]$

E35: $.Y - .R_i \Rightarrow R_i$

E36: $.R_i \Rightarrow X$

Die Existenz eines inversen Befehls gibt, wie bereits in LOES-1.4 gesagt, ein Generierparameter an.

Im folgenden Beispiel einer Einakkumaschine, das man bereits bei McKeeman [McK65] findet, kann zwar der Speicherbefehl nicht vermieden werden, aber wenigstens der Ladebefehl.

Die Dreiadreß.befehlsfolge

D31: $.Y \Rightarrow X$

D32: $.X + .Z \Rightarrow Z$

wird in die Folge $.Y \Rightarrow AKK_u$

E31: $.AKK_u \Rightarrow X$

E32: $[.X \Rightarrow AKK_u]$

E33: $.AKK_u + Z \Rightarrow AKK_u$

E34: $.AKK_u \Rightarrow Z$

umgesetzt.

Um Einsparungsmöglichkeiten bei der Umsetzung zu erkennen, wird über den augenblicklichen Registerzustand Buch geführt. Für jedes Register wird ein Speicherbereich angelegt, der den Namen der Variablen enthält, die Adressierungsart einschließlich Offset und zusätzlich einen Merker,

der angibt, ob der Wert nur im Register geführt wird, oder ob er bereits abgespeichert wurde, aber außerdem noch im Register vorhanden ist. Das Verfahren geht so lange gut, bis keine freien Register mehr zur Verfügung stehen. Dann muß ein Register freigeräumt werden. Um nicht eine Variable abzuspeichern, die innerhalb des gerade zu übersetzenden Programmabschnitts besonders häufig benötigt wird, dienen die in Kapitel LOES-1.1 beschriebenen Optimierungsanweisungen der Abschnittseinteilung und der Angabe wichtiger Variablen. Sie dürfen nicht zur Blockierung von Rechenregistern führen. Daher werden bei Abschnittsbeginn nur die ersten Prioritätsvariablen berücksichtigt, so daß noch die Hälfte der Rechenregister der Verwaltung der Umsetzstufe unterworfen werden kann.

Welches Register freigeräumt wird, entscheidet die Prozedur SUCHRR. In einem ersten Programmtest ist in SUCHRR folgender einfacher Algorithmus programmiert worden. Jedem Register, das eine noch nicht abgespeicherte Variable enthält, wird eine "Altersnummer" zugeordnet, die angibt, wie lange eine Variable bereits ein Register belegt. Zur Abspeicherung wird immer die Variable gewählt, die bereits am längsten einen Registerplatz belegt. Leider sagt das nichts darüber aus, wann sie das nächste Mal wieder benötigt wird, so daß diese Registerverwaltung nicht optimal ist.

Bessere Ergebnisse werden erst durch eine Vorausschau über eine implementationsmäßig bedingte Anzahl von Folgebefehlen erreicht ¹⁾. Es wird dann das Register ausgewählt, welches am spätesten wieder benötigt wird. Schwieriger wird die Registerverwaltung bei Sprüngen und Marken. Man macht sich die Arbeit einfach, wenn man sich auf lokale Optimierungen beschränkt und generell vor Sprüngen und Marken alle Registerinhalte abspeichert. Damit sind sie an jedem Programmpunkt, unabhängig von dem Weg, auf dem er erreicht wurde, klar bestimmt, z.B. leer. Dieses Verfahren ist bei Programmen, die nur relativ wenig Sprünge enthalten, sicherlich vertretbar. Bei arithmetischen Laufzeitprogrammen kommen aber laufend Sprünge, oft nur über wenige Befehle hinweg vor. Eine Probeimplementation hat gezeigt, daß dann die Länge von so übersetzten Programmen mit der Anzahl der zur Verfügung stehenden Rechenregistern steigen kann - wahrlich keine optimale Registerverwaltung. Abhilfe ist nötig. Beim Auftreten eines Sprunges wird nun vorläufig probeweise bis zum Sprungziel weiter übersetzt. Diese Vorausschau zeigt, welche Register bis zum Sprungziel

¹⁾ Aus Speicherplatzgründen beschränkt sich die vorliegende Implementation auf fünf Befehle. Diese müssen im Speicher gehalten werden, da das ERLAN-Filehandling nur sequentiellen Zugriff erlaubt.

benötigt werden und welche unberührt bleiben. Die benötigten werden vor Übersetzung des Sprunges und außerdem unmittelbar vor dem Sprungziel "befreit". Wenn während der Probeübersetzung ein nicht erwartetes Ziel oder ein weiterer Sprung auftritt, wird die Vorausschau abgebrochen. Ebenfalls abgebrochen wird, wenn zwischen Sprung und -ziel mehr als eine implementationsmäßig bedingte Anzahl von Dreiaßbefehlen liegen. In diesen Fällen tritt doch wieder die "Holzhammermethode" der Beschränkung auf lokale Optimierung in Kraft, jedoch mit einer Verbesserung: Register, die als wichtig gekennzeichnete Variablen enthalten, sollen nicht befreit werden. Bei Rückwärtssprüngen (Schleifen) wird analog verfahren, nur sieht hier der Übersetzer zuerst das Sprungziel (Marke) und dann den zugehörigen Sprungbefehl.

Am Ende eines Abschnitts (Definition eines Abschnitts siehe LOE-1.2) werden alle Register befreit, auch die mit wichtigen Variablen. Hier ist noch eine kleine Optimierung möglich, indem man die Abspeicherung der Variablen, die im Folgeabschnitt als wichtig gekennzeichnet sind, unterläßt. Eine weitere Befehlersparnis ergibt sich, wenn bei Sprüngen an das Unterprogrammende nur die Register befreit werden, die formale Rückkehrparameter enthalten, so wie das beim Rücksprungbefehl auch gehandhabt wird. Solche Befehle werden an dem reservierten Namen RETURN für das Sprungziel erkannt. Der letzte Programmabschnitt sollte ausnahmsweise nicht mit dem Abschnittsende-Befehl abgeschlossen werden, um unnötiges Registerbefreien zu vermeiden.

Die Griff zur "Holzhammermethode" läßt sich bei indirekten Sprüngen nicht vermeiden, da hier die Sprungziele nicht bekannt sind und die Methode mit der Vorausschau versagt. Das Problem, beim Sprungziel alle Register zu befreien, erledigt sich von selbst, wenn der Ersteller eines abstrakten Programms jedem Sprungziel, das durch einen indirekten Sprung erreicht werden kann, eine eigne Marke zuordnet. Dadurch erkennt die Hauptumsetzstufe bei der linearen Suche während der Vorausschau eine Unsymmetrie, d.h. eine Marke ohne zugehörigen Sprungbefehl bzw. umgekehrt und räumt automatisch alle Register.

Ein Beispiel soll dies Verdeutlichen:

| | | |
|---|---|-------------|
| /* alle Register freimachen | | |
| MINDIREKT1: \$ |] | Unsymmetrie |
| SCHLEIFE1: \$ | | |
| /* alle Register freimachen | | |
| MINDIREKT2: \$ |] | Unsymmetrie |
| | | |
| /*alle Register freimachen | | |
| D1JNZ→SCHLEIFE1 \$ |] | Unsymmetrie |
| /* nur die in der Schleife benötigten | | |
| /* Register freimachen | | |
| SCHLEIFE2: \$ |] | Unsymmetrie |
| | | |
| /*benötigte Register freimachen |] | Symmetrie |
| D1JNZ→SCHLEIFE2 | | |
| | | |
| /*alle Register befreien, | | |
| J → .VERTEILER \$ /*indirekter Sprungbefehl | | |

Wird MINDIREKT1 oder MINDIREKT2 über indirekte Sprungbefehle erreicht, werden wegen der Unsymmetrie immer wieder alle Register leer geräumt. Sollte jedoch die Marke SCHLEIFE2 indirekt erreichbar sein, läuft die Registerverwaltung in die Irre. Indirekte Sprünge dürfen nicht aus einem Block herausführen, da dann wichtige Variablen enthaltende Register nicht befreit werden. Ein Ersteller abstrakter Programme muß, wie man erkennt, hohe Anforderungen erfüllen, hat er es doch mit Assemblerprogrammierung zu tun.

Bei geschachtelten Sprüngen wirkt ebenfalls der Holzhammer. Nur die innerste Schachtel wird mit der Vorausschaumethode übersetzt. Hiergegen läßt sich jedoch leicht Abhilfe schaffen, wenn das Vorausschauprogramm rekursiv aufgerufen werden kann und die Vorausschau auch über eine hinreichend große Anzahl von Befehlen möglich ist.

Falls in der Registerbelegungstabelle nur der gleiche Variablenname wie im Befehl notiert ist, aber die Adressierungsarten nicht übereinstimmen, wird das Register freigegeben. In der Pilotimplementation

bedingt das, wie das folgende Beispiel zeigt, noch nicht optimalen Code.
Wenn vor der Umsetzung des Befehls

$$.A + ..B \rightarrow A$$

die Variable A und der Zeiger B in den Registern R1 und R2 geführt werden,
könnte nämlich der Befehl

$$.R1 + ..R2 \rightarrow R1$$

erzeugt werden. Welche Programmergänzungen nötig sind, beschreibt der
folgende Abschnitt über Simulation nicht vorhandener Adressierungsarten.

LOES-2.4 Simulation von Adressierungsarten

Im Anhang A1.4 sind die Adressierungsarten der abstrakten Dreiadreßmaschine gelistet. Auf realen Maschinen, deren Hardware nur einen Teil davon kennt, sind Ersatzprogrammierungen durch manchmal recht lange Befehlsfolgen nötig. Ursprünglich ist geplant gewesen, zur Simulation der Adressierungsarten eine eigene Umsetzstufe einzusetzen, ein naheliegender und gangbarer Weg. Am praktischen Beispiel hat sich aber gezeigt, daß es bei Schleifen manchmal aus Optimierungsgründen sinnvoll ist, Teile der Adreßsimulation vorzuziehen. Diese Aufgabe erinnert an die Aufgabe der Vorausschau in der Hauptumsetzstufe. Wegen der ähnlichen Aufgabe sollte die Simulation der Adressierungsarten mit in diese Stufe einbezogen werden.

Eng mit dem Thema Adressierungsart hängt die Erzeugung der Wiedereintrittsfähigkeit über Indexregister zusammen. Auch dafür ist ursprünglich eine eigene Umsetzstufe geplant gewesen. Da aber in Laufzeitprogrammen bereits die Adressierung über Indexregister vorkommt, ist eine doppelte Indizierung nötig, die auf keinem der gängigen Kleinrechner möglich ist und daher eine Ersatzprogrammierung erfordert. Die Simulation der Adressierungsarten läßt sich mit der Erzeugung der indizierten Adressierung koppeln, wenn man für alle Adressierungsarten, also auch für die direkte, Ersatzbefehlsfolgen fordert.

Die Art der Adreßsimulation kann vom Befehlstyp abhängig sein.

Bei Transferbefehlen können andere Adressierungsarten zulässig sein als bei arithmetischen und logischen Verknüpfungen. Deshalb sind mehrere Tabellen für die Adressierungsarten-Ersatzprogrammierung bereitzustellen.

| | | |
|------------|-------------|--|
| Tabelle.1: | Für Befehle | Quelle \rightarrow Register |
| 2: | | Register \rightarrow Ziel |
| 3. | | Register \otimes Quelle \rightarrow Register |
| 4. | | { NOT } Register \rightarrow Register |
| 5. | | Register $\left\{ \begin{array}{c} +c \\ -c \\ +1 \\ -1 \end{array} \right\} \rightarrow$ Register |

Reale Maschinen lassen manchmal nicht alle Befehle in jeder Adressierungsart zu, so daß obige Tabellen nicht für alle Operatoren geeignet sind (vergl. ZIEL-3). Man kann aber davon ausgehen, daß ein Satz primitiver

Adressierungsarten ausreicht, um für alle Ausnahmen Ersatzmöglichkeiten anzugeben. Deshalb ist eine weitere Tabelle nötig, um alle Befehle zu erfassen. Natürlich benötigt man noch eine Tabelle, die aussagt, welcher Befehl zu welcher Adressierungsarten-Ersatztablette gehört. Als Adressierungsart wurde bisher nur eine Adressierung über eine Speicherzelle implementiert, z.B.

$$.R + ..X \rightarrow R$$

Wie bereits in LOE-2.3 (Registerverwaltung) angedeutet, ist eine Ergänzung zur Adressierung über Register nötig (z.B. $.R + ..AR \rightarrow R$ /* AR=Adreßregister)

Beispiel: es möge der Befehl

$$.R \text{ AND } ..M \rightarrow R$$

umzusetzen sein.

Die Ersatztablette 3 biete beispielsweise:

$$..IX\%m \rightarrow AR$$

$$.R \otimes ..AR \rightarrow R$$

wobei IX auf die lokalen Daten zeigt; m ist der zu M gehörende Adreßabstand, AR ein als Adreßregister verwendbares freies oder zu befreiendes Register. R das Register des zu ersetzenden Befehls. \otimes stehe für AND oder OR. Der AND-Befehl existiere auf der realen Maschine in der Adressierungsart $..AR$. Möge das bei OR nicht zutreffen.

Die Tabelle für die Ersatzprogrammierung in primitiven Adressierungsarten könnte beispielsweise folgende Befehle vorschlagen, bei denen Arithmetik nur zwischen Registern betrieben wird.

$$..IX\%m \rightarrow AR$$

$$..AR \rightarrow RF \quad /* RF \text{ steht für ein freies Register}$$

$$.R \otimes RF \rightarrow RF$$

Aus den Beispielen geht hervor, daß die in LOES-1.4 definierte Einadreßmaschine noch einer Erweiterung bedarf. Ein Register ist nicht nur durch eine Nummer gekennzeichnet, sondern auch durch die Art der Verwendung. Die Register können Spezialregister sein (z.B. Indexregister oder Register, auf denen Postinkrementfunktion möglich ist). Die unterschiedlichen Registerarten erfordern eine etwas andere Registerverwaltung, u.a. für jede Registerart einen eigenen Belegtzähler; die Prozedur SUCHR, welche ein freies Register beschafft, erhält den Parameter Registerart.

| Nr. | Quelle | quelle \rightarrow R | $.R \otimes \text{quelle} \rightarrow R$ | $.R \Rightarrow$ Senke | Optimier.- hinweise |
|-----|--------------|--|---|---|------------------------|
| 1 | C | $C \rightarrow R$ | $R \otimes C \rightarrow R$ | | |
| 2 | ADR | $ADA \rightarrow R$ | $R \otimes ADR \rightarrow R$ | | |
| 3 | .M | $..IX\%m \rightarrow R$ | $.R \otimes ..IX\%m \rightarrow R$ | $.R \rightarrow ..IX\%m$ | |
| 4 | ..M | $..IX\%m \rightarrow AR$ $..AR \rightarrow R$ | \sim $.R \otimes ..AR \rightarrow R$ | \sim $..AR \rightarrow R$ | Teste! |
| 5 | ...M | $..IX\%m \rightarrow AR$ $..AR \rightarrow AR$ $..AR \rightarrow R$ | \sim \sim $.R \otimes ..XR\%of \rightarrow R$ | \sim \sim $.R \rightarrow ..XR\%of$ | Teste! |
| 6 | ..M%of | $..IX\%m \rightarrow XR$ $..XR\%of \rightarrow R$ | \sim $.R \otimes ..XR\%of \rightarrow R$ | \sim $.R \rightarrow ..XR\%of$ | Teste! |
| 7 | ...M%of | $..IX\%R \rightarrow XR$ $..IX\%of \rightarrow AR$ $..AR \rightarrow R$ | \sim \sim $.R \otimes ..AR \rightarrow R$ | \sim \sim $.R \rightarrow ..AR$ | Teste! |
| 8 | . Ψ .M | $..IXm \rightarrow AR$ $.AR - 1 \rightarrow AR$ $.AR \rightarrow ..IX\%m$ $..AR \rightarrow R$ | \sim \sim \sim $.R \otimes ..AR \rightarrow R$ | \sim \sim \sim $.R \rightarrow AR$ | Teste! Merke! |
| 9 | .. Ψ .M | $..IX\%m \rightarrow AR$ $.AR - 1 \rightarrow AR$ $.AR \rightarrow IX\%m$ $..AR \rightarrow AR$ $..AR \rightarrow R$ | \sim \sim \sim \sim $.R \otimes ..AR \rightarrow R$ | \sim \sim \sim \sim $.R \rightarrow ..AR$ | Teste! |
| 10 | ..M' | $..IX\%m \rightarrow PI$ $..PI' \rightarrow R$ $.PI \rightarrow ..IX\%m$ | \sim $.R \otimes ..PI' \rightarrow R$ \sim | \sim $.R \rightarrow ..PI'$ \sim | Teste! Merke! |
| 11 | ...M' | $..IX\%m \rightarrow PI$ $..PI' \rightarrow PI$ $..PI \rightarrow R$ $.PI \rightarrow IX\%m$ | \sim \sim $.R \otimes ..PI \rightarrow R$ \sim | \sim \sim $.R \rightarrow ..PI$ \sim | Teste! Merke! |

Abb. LOES-2.4/1: Tabellen zur Simulation der Adressierungsarten für die SIEMENS 310 bei Lade-, Verknüpfungs- und Speicherbefehlen
 AR = Adreß register, XR = Indexregister, PI = Postincrementregister, IX= Indexregister. m ist der zu M gehörende Adreß -
 Abstand

In vielen Fällen unterscheiden sich die Befehle der Tabellen 1 bis 5, die zu jeder Befehlsart Ersatzadressierungsarten bereitstellen, nur dadurch, daß jeweils ein Befehl sinnvoll abgewandelt wird (z.B. statt Transfer-Verknüpfungsbefehl). Solche Kennungen können Tabellen einsparen. Weitere Kennungen weisen darauf hin, daß sinnvollerweise ein Befehl vor eine Schleife zu ziehen wäre, oder daß eine Abspeicherung vorläufig aufgeschoben werden sollte.

Zum Ansprechen von Parametern sind weitere Tabellen nötig. Die Abbildung LOES-2.4/1 zeigt, wie auf der SIEMENS 310 bei Lade-, Verknüpfungs- und Speicherbefehlen nicht vorhandene Adressierungsarten und gleichzeitig die Adressierung über Indexregister simuliert werden könnte. Zur Darstellung ist die menschenlesbare Form gewählt. Die Tabelle gilt auch für ähnliche Maschinen, die zwar die gleichen Adressierungsarten wie die SIEMENS 310 besitzen, aber bei der sie an bestimmte Registertypen gebunden sind. Bei der SIEMENS 310 sind die Adressierungsarten nicht an Befehle gebunden, daher genügt eigentlich eine Tabelle, die z.B. nur die Spalte "Quelle \rightarrow R", aber mit einem Hinweis, wann ein Befehl sinn- gemäß abgewandelt werden muß, um aus einem Lade- einen Speicher- oder Verknüpfungsbefehl zu erhalten. In Abbildung LOES-2.4/1 sind die abgewandelten Befehle explizit aufgeführt. Auf direkt zu übernehmende Befehle weist das Symbol \hookrightarrow hin. Selbst in einer Spalte tauchen öfter gleiche Befehle auf. Daraus ließe sich eine Verdichtung der Tabelle ableiten, indem die Tabelle nur Zeiger auf einige wenige explizit anzugebende Befehle enthält. Die Spalte Optimierungshinweise gibt an, daß manche Befehle unter Umständen entfallen können. Die mit "Teste!" gekennzeichneten Ladebefehle entfallen, wenn das Register zufällig bereits richtig geladen ist. "Teste! -Befehle" werden, wenn die Register nicht knapp werden, möglichst vor Schleifen gezogen.

Die Anwendung dieser Adressierungsarten-Ersatztabellen sollen an Hand der Umsetzung einiger Befehlsfolgen gezeigt werden. Seien R1, R2 und R3 freie Register

```
.D  $\rightarrow$  A $
..A' + ...B  $\rightarrow$  .C%5 $
ENDE: $/*Ende eines Blocks.
```

Die bisher implementierte Umsetzung liefert die Einadreß befehlsfolge

```
.D→R1 $
[.R1→A $]  Merken
.R1→A $    Da A zur Adressierung im nächsten Befehl
            benötigt wird, muß R1 nun doch freigemacht
            werden.
..A'→R1 $
.R1 + ...B→R1 $
[.R1 → .C%5 $] Merken
.R1 → .C%5 $  Am Blockende werden alle Register
               befreit.
```

Die Umsetzung nach den Adressierungstabellen Abb. LOES-2.4/1 liefert folgende Befehlskette, die obige Befehle als Kommentar enthält

| | | |
|--|---|--|
| ..IX%d→R1 \$ [.R1 → .IX % a \$] | Merken | .D→R1 \$ [R1→A] |
| [..I%a →R1 \$] ..R1'→R2 [.R1 → IX%a \$] | Der Befehl .R1→A wird nicht benötigt, dafür aber ein neues Rechenregister R2 | .R1→A \$..A'→R1 \$ |
| ..IX%b→R3 \$..R3→R3 \$..R2+..R3→R2 \$ [.R2 → C%5] | | .R1 + ...B →R1 \$ [.R1→.C%5 \$] |
| .R1 →.IX%a \$..IX%C→R3 \$.R2 → .R3%5 \$ | R1 und R2 am Blockende befreien | . .R1→.C%5 |

Wenn die Dreiadreß befehlsfolge Schleifen enthält, ergeben sich neue Überlegungen. Sei die folgende besonders einfache Schleife gegeben

```
SCHL: ..A' + .B→.C
      D1JNZ → SCHL $
```

Die Übersetzung ohne Schleifenoptimierung führt, wenn genügend freie Register vorhanden sind, zu folgendem Codestück:

| | |
|---|------------------|
| SCHL: \$ | SCHL: \$ |
| .IX%a → R2 \$ ✕ ..R2' → R1 \$ [.R2 → IX%a \$] | ..A' → R1 \$ |
| .R1 + ..IX%b R1 \$ | .R1 + .B → R1 \$ |
| ..IX %C → R3 \$ ✕ .R1 → .R3 \$.R3 → .IX%C | .R1 → .C \$ |
| .R2 → .IX%a \$.R3 → .IX% \$ D1JNZ → SCHL \$ | D1JNZ → SCHL |

Die mit ✕ gekennzeichneten Befehle lassen sich vor die Schleife ziehen und man erhält:

| |
|---|
| ..IX%a → R2 \$..IX%c → R3 \$ |
| SCHL: \$..R2' → R1 \$.R1 + ..IX%b R1 \$.R1 → .R3' \$ D1JNZ → SCHL \$ [.R2 → .IX%a \$] [.R3 → .IX%c \$] |

LOES-2.5 Simulation von Operatoren

Wie für Adressierungsarten muß auch für Operatoren, die auf einer realen Maschine nicht existieren, eine Ersatzprogrammierung angegeben werden. Eine eigene Umsetzstufe kann den Einbau in das maschinenangepaßte Dreiadreß programm durchführen.

Beispiel:

Sei der EXOR-Befehl auf der realen Maschine nicht vorhanden und der Befehl

.A EXOR .B \rightarrow C

zu simulieren.

Es wird eine Hilfszelle H eingeführt, die automatisch mit in die Deklarationsliste aufzunehmen ist. Dann ist folgender Ersatzcode möglich

.A OR .B \rightarrow C

.A AND .B \rightarrow H

.C - .H \rightarrow C

Das Beispiel lehrt nebenbei, daß es in Laufzeitprogrammen durchaus sinnvoll sein kann, logische mit arithmetischen Operatoren zu mischen und bewußt das Typkonzept höherer Programmiersprachen zu verletzen. Die direkte Umsetzung der Formeln

$$a \vee b = \overline{a} \cdot b \vee a \cdot \overline{b}$$

$$\text{oder } a \vee b = \overline{\overline{a} \cdot b} \cdot \overline{a} \vee b$$

verletzt das Typkonzept nicht, führt aber zu einer längeren Befehlsfolge.

Wenn aber A = C, d.h. der Befehl

.A EXOR .B \rightarrow A

ersetzt werden soll, ist eine Abwandlung der obigen Ersetzbefehlsfolge nötig, die auch für den ersten Fall geeignet ist, die aber bei der Umsetzung auf Einadreß befehle nur optimal ist, wenn die Einadreß maschine inverse Befehle hat.

.A OR .B \rightarrow H

.A AND .B \rightarrow C

.H - .C \rightarrow C

Dem Anwender des portablen Systems wird ziemlich viel zugemutet. Seine Aufgabe wird einfacher, wenn er Ersatz in Einadreß code eingeben darf.

Für den Befehl

.R EXOR .Q → R

ist die Codefolge

.R → RF

.R OR .Q → R

.RF AND .Q → RF

.R ← .RF → R als Ersatz geeignet.

RF ist ein freies oder freizuräumendes Register, das auch in die Registerverwaltung der Hauptumsetzstufe einzubeziehen ist. Das spricht dafür, auch die Befehlssimulation in die Hauptumsetzstufe zu verlegen.

Die Hauptumsetzstufe kann auch erkennen, wenn Q bereits in einem Register R2 zwischengespeichert ist und die optimale Codefolge

.R → RF

.R OR .R2 → R

.R AND .R2 → R

.R ← RF → R absetzen.

BSPL Beispiel

BSPL-1 Generierparameter

Der in die Pilotimplementation aufgenommene Teil des Umsetzverfahrens soll an einem Beispielprogramm näher betrachtet werden. Es wird in der Gleitpunktarithmetik laufend verwendet. Um die Generierparameter zu verstehen, muß noch etwas über Gleitpunktzahlen gesagt werden. Die Bearbeitung vieler technisch-wissenschaftlicher Probleme mit Hilfe von Rechnern erfordert, einen weiten Zahlenbereich darstellen zu können. Dazu dient die Gleitpunktarithmetik, die deshalb in keiner Programmiersprache für technisch-wissenschaftliche Anwendungen fehlen darf. Auf Kleinrechnern ist sie jedoch oftmals weder hard- noch softwaremäßig implementiert. Simulationsroutinen sind nötig. Sie können ebenfalls verwendet werden, wenn die vorhandene Gleitpunktarithmetik wegen der Zerstörung der Portabilität von Rechenergebnissen nicht verwendet werden soll.

Unter Gleitpunktzahlen werden Zahlen der Form

$$\text{Mantisse} * \text{Basis}^{\text{Exponent}}$$

verstanden. Die abstrakten Programme gehen von der Basis 2 und von einer ganz bestimmten Darstellungsform einer abstrakten Gleitpunktzahl aus.

Eine abstrakte Gleitpunktzahl ist normiert:

positive Zahlen liegen im Intervall $1/2 \leq |\text{Mantisse}| < 1$

negative Zahlen liegen im Intervall $1/2 < |\text{Mantisse}| \leq 1$.

Da negative Zahlen im Zweierkomplement dargestellt werden, beginnen positive Mantissen immer mit 0.1, negative immer mit 1.0. Die Stelle vor dem Punkt gibt das Vorzeichen an. Die erste Stelle nach dem Punkt wird zur Vermeidung von Redundanz nicht mit abgespeichert. Dieses Bit wird deshalb "verstecktes Bit" genannt. Statt des Exponenten wird die sog. Charakteristik abgespeichert. Sie ergibt sich aus

$$\text{Exponent} + 2^{(\text{EXL}-1)}$$

wobei EXL die für den Exponenten zur Verfügung stehende Anzahl von Bits ist. Wenn bei negativen Gleitpunktzahlen (negative Mantisse) der Exponent im Einerkomplement dargestellt ist, hat das den Vorteil, daß Gleitpunktzahlen wie im Zweierkomplement abgespeicherte ganze Zahlen geordnet sind. Außerdem funktioniert die Negation wie bei Ganzzahlen. Aus dem folgenden Beispiel in Abbildung BSPL-1/1, das wegen der geringen Wortlänge von nur drei Bits

natürlich nicht praktisch eingesetzt wird, gehen weitere Einzelheiten hervor.

| normiert binär Vorzeichen u. Betrag | Vor- zei- chen | Expo- nent +1 | Mantisse im Zweier Komple- ment | abstrakte Gleitpunkt- zahl | Ordnung wie Ganz- zahl | Dezimal Wert | Kommentar |
|--|----------------------|---------------------|--|----------------------------------|---------------------------------|-----------------|--|
| -1.00 2^0 | 1 | 1 | 00 | 100 | -4 | -1 | kein pos. Äquivalent |
| -0.11 2^0 | 1 | 1 | 01 | 101 | -3 | 0.75 | |
| -1.00 2^{-1} | 1 | 0 | 00 | 110 | -2 | 0.5 | |
| -0.11 2^{-1} | 1 | 0 | 01 | 111 | -1 | 0.375 | |
| 0.00 2^{-1} | 0 | 0 | 00 | 000 | 0 | 0 | eindeutige Null |
| +0.10 2^{-1} | 0 | 0 | 10 | 000 | 0 | 0.25 | verboten, da für Null reserviert |
| +0.11 2^{-1} | 0 | 0 | 11 | 001 | 1 | 0.375 | |
| +0.10 2^0 | 0 | 1 | 10 | 010 | 2 | 0.5 | |
| +0.11 2^0 | 0 | 1 | 11 | 011 | 3 | 0.75 | |

Abbildung BSPL-1/1

Beispiel für die Gleitpunktzahldarstellung auf einer abstrakten 3-Bit-Maschine mit Exponentenlänge 1 Bit.

Bereits an diesem einfachsten Beispiel erkennt man das Abbildungsgesetz, die Ordnung und Negierungsmöglichkeit wie bei ganzen Zahlen. Die Kommentarspalte weist auf Sonderheiten hin.

Ein solcher Aufbau von Gleitpunktzahlen wird auch bei manchen realen Rechnern verwendet [GÜN72].

Zur arithmetischen Weiterverarbeitung zerlegt das abstrakte Beispielpogramm MAANP (Maschinenanpassung) die Gleitpunktzahl in die Bestandteile Mantisse und Exponent. Bei dieser Zerlegung läßt sich gleichzeitig eine Anpassung an weitgehend beliebige Zahlendarstellungen durchführen. Die möglichen Anpassungen werden durch Generierparameter beschrieben.

Das Generierparameteraufnahmeprogramm fragt zuerst, ob die Gleitpunktzahl zufällig die auf der abstrakten Maschine gewünschte Form hat. Dies ist sicher dann der Fall, wenn Gleitpunktzahlen auf der betrachteten Maschine bisher weder hard- noch softwaremäßig implementiert sind.

Bei der Antwort TRUE fehlen nur noch wenige Zusatzfragen zum vollständigen Bestimmen der Zahlendarstellung. Für die Länge einer Gleitpunktzahl in Maschinenworten ist derzeit als Antwort nur "1" oder "2" vorgesehen. Für Gleitpunktzahlen höherer Genauigkeit ist das Aufnahmeprogramm noch zu ergänzen. Bei Kleinrechnern mit 16 Bit Wortlänge wird die Antwort immer "2" sein. Die nächsten Fragen, ob die Mantisse in einem Wort untergebracht werden kann, nach der Nummer des Wortes, in dem das Vorzeichen steht, stellt das Aufnahmeprogramm nur, wenn die Gleitpunktzahl in zwei Worten untergebracht ist.

Weitere Fragen nach der Marke für Vorzeichen, Mantisse hoher und niederer Teil usw. erübrigen sich, wenn die Gleitpunktzahl die abstrakte Form hat.

In den Abbildungen A3/1 und /2 im Anhang ist für zwei Fälle der Dialog des Generierparameter-Aufnahmeprogramms bezüglich Gleitpunktzahldarstellung in einem Schnelldruckerprotokoll festgehalten. Aus den eingegebenen Parametern erzeugt sich das Generierparameter-Aufnahmeprogramm weitere Parameter, die das Programm MAANP oder Arithmetikprogramme benötigen. Dazu gehören hauptsächlich nicht explizit erfragte Masken und Verschiebezahlen, die angeben, wie die Teile der Mantisse an die leitenden Bits 0.1 (bzw. 1.0 bei negativen Zahlen) herangerückt werden. Die Parameter POSRUNDH und-L und NEGRUNDH und-L benötigen die Arithmetikprogramme zum Runden. In den Abbildungen A3/3 und /4 sind Schnelldruckerprotokolle der eingegebenen und der vom Aufnahmeprogramm erzeugten Generierparameter gezeigt.

BSPL-2 Programm mit Zwischenformen

Im vorausgegangenen Abschnitt wurde der Sinn des Programms MAANP erläutert. Jetzt soll das Programm in seinen verschiedenen Phasen vorgestellt werden. Vergleiche dazu Kapitel LWEG-3 - Konzept der Umsetzung. Abbildung BSPL-2/1 zeigt einen Ausschnitt aus dem Programm in seiner ursprünglichen Form mit den von Maschinenparametern abhängigen Verzweigungen und Konstanten. Das gesamte Programm ist im Anhang A5.1 aufgenommen. Die Abbildung BSPL-2/2 zeigt denselben Ausschnitt nach Durchlaufen des Generators unter Verwendung der im Anhang A3 vorgestellten Generierparameter. Die nächste Abbildung BSPL-2/3 zeigt in der Kommentarspalte eines Assemblerausdrucks das Ergebnis der Umsetzung durch die Optimierungs- und Hauptumsetzstufe nach Rückübersetzung des internen in mnemotechnischen Code. Die Hauptumsetzstufe ging bei dem gezeigten Beispiel von folgenden Parametern aus:

1. Vier Register
2. Register-Register Transfer möglich
3. Verknüpfungen zwischen Registern möglich.

Das Ergebnis der Optimierungsstufe ist nicht abgebildet, da es nur in interner Darstellung vorliegt. Die linke Spalte aus dem Assemblerprotokoll gibt die von Hand durchgeführte Umsetzung in die Assemblersprache der SIEMENS R30 wieder. Die Handumsetzung geht davon aus, daß Parameter in den Registern G1, G2 und G3 übergeben werden, daß die Register R2, R3 und R6 dem Laufzeitprogramm zur beliebigen Verwendung zur Verfügung stehen, daß die Rücksprungadresse im Register R7 übergeben wird und daß Register R5 auf einen freien Datenbereich für Variable zeigt. Diese Annahmen sind realitätsbezogen, wie ein Vergleich mit der Spezifikation des Codegenerators für PEARL auf der SIEMENS 310 [TRA80] zeigt.

```

.PARAMLA AND #MLMAA -> MLHZ $      MANTISSE NIEDERER TEIL
.PARAMHA AND #MHMAA -> PAR2 $      MANTISSE HOHER TEIL
/. DER LETZTE BEFEHL ENTFÄLLT BEI DER UMSCHZUNG
/. FALLS ER ZU .PAR2 AND 11...1 -> PAR2 ENTARTET
.MLHZ,U SL #MLVL4 -> PAR1 $      NIEDERER TEIL LINKSBUENDIG
.PAR2 SR #MHVR4 -> PAR2 $      HOHER TEIL RECHTSBUENDIG
/. JETZT STEHT Z.B. EINE 10-BIT MANT. SO IN ZWEI
/. 8 BIT WORTEN 00001111,11111100

/. SIE WIRD NUN DOPPELT LINKS BZW. RECHTS VERSCHOBEN, DASS NOCH
/. PLATZ FUER ZWEI VERSTECKTE BITS BLEIBT

```

```

IF MVL GT 0 THEN
  IF MVL GT 1 THEN
    #MVL4 -> SCHZ1$
  FI

  SL: .PAR1 SL 1 -> PAR1,C,C $
    .PAR2 SLC 1 -> PAR2 $

  IF MVL GT 1 THEN
    D1JNZ -> SL$
  FI
FI

IF MVL LT 0 THEN
  .PAR2 SR 1 -> PAR2,C,C $
  .PAR1 SRC 1 -> PAR1 $
FI

IF HBGESP EQ FALSE THEN
  /. VERSTECKTE BITS EINBLENDEN
  IF MBETR EQ TRUE
    #HBITS4 + .PAR2 -> PAR2 $
  FI

  IF MBETR EQ FALSE THEN
    .VZHZ -> ,ZN,EQ $
    JEQ -> POSHB $
    #NEGHBITS4 + .PAR2 -> PAR2 $
    J -> HBENDE $
    POSHB: #HBITS4 + .PAR2 -> PAR2 $
    HBENDE: $
  FI
FI

IF MBETR EQ TRUE THEN
  /. BEI NEGATIVER GLEITPUNKTZAHL WIRD DER
  /. BETRAG DER MANTISSE NEGIIERT
  .VZHZ,U -> ,F,EQ $
  /. DER TESTBEFEHL (TRANSFERBEFEHL OHNE ZIEL) ERZEUGT V=0
  JEQ -> RETURN,SAVE F,EQ $
  NEG .PAR1 -> PAR1,C,C $

```

Abb. BSPL-2/1: Ausschnitt aus dem parametrisierten Programm MAANP. Man erkennt die IF-Abfragen und die in Hochpfeilen eingeschlossenen Konstanten.

```

.PAR2 AND 255 -> MLHZ $      MANTISSE NIEDERER TEIL
.PAR1 AND 1 -> PAR2 $      MANTISSE HOHER TEIL
/. DER LETZTE BEFEHL ENTFALLT BEI DER UMSETZUNG
/. FALLS ER ZU .PAR2 AND 11...1 -> PAR2 ENTARTET
.MLHZ,U SL 0 -> PAR1 $      NIEDERER TEIL LIKSBUENDIG
.PAR2 SR 0 -> PAR2 $      HOHER TEIL RECHTSBUENDIG
/. JETZT STEHT Z.B. EINE 10-BIT MANT. SO IN ZWEI
/. 8 BIT WORTEN 00001111,11111100

```

```

/. SIE WIRD NUN DOPPELT LINKS BZW. RECHTS VERSCHOBEN, DASS NOCH
/. PLATZ FUER ZWEI VERSTECKTE BITS BLEIBT
S -> SCHZ1$

```

```

SL: .PAR1 SL 1 -> PAR1,C,C $
.PAR2 SLC 1 -> PAR2 $

```

```

D1JNZ -> SL$

```

```

/. VERSTECKTE BITS EINBLENDEN

```

```

.VZHZ -> ,ZN,EO $
JEQ -> POSHB $
128 + .PAR2 -> PAR2 $
J -> HBENDE $
POSHB: 64 + .PAR2 -> PAR2 $
HBENDE:$

```

```

RETURN:$

```

```

RET:1:MAANP,SAVE F,EQ *(OUT,OUT,OUT) $

```

```

$ENDE$

```

Abb. BSPL-2/2: Programmausschnitt aus Abb. BSPL-2/1 nach Generatordurchlauf

/MAANP;

'NAME' MAANP
'SATZ' MAANP
'IA' MAANP/

*** SCHZ1 NUR IN R6
VZHZ/ 'IDENT' 2
*** MLHZ WIRD NICHT BEN DA IN REG

R2 := G1
R2 := R2 .U 126
R3 := G1
R3 := R3 .U 128
=: SP POS
R2 := R2 .X 126
POS/ R4 :T G2

(VZHZ+R5):=R3
#: SP NENULL
R2 :T G1
=: SP RETURN
NENULL/ R2 := G2

R3 := G1
R3 := R3 .U 1
R6 := 5
SL/ R2 := R2 .V + 1

R3 := R3 .V + 1
R6:DS SL
R4 :T (VZHZ+R5)
G1 := R2
G2 := R3
=: SP POSHB
R2 := 128
R2 := R2 + G2
G2 := R2
=: SP HBENDE
POSHB/ R2 := 64

R2 := R2 + G2
G2 := R2
HBENDE/
RETURN/
=: SP R7

*** Kommentar

*** 1 SUBIN:1:MAANP (IN1,IN2,
*** OUT3) \$
*** 2 DCL:1: SCHZ1 \$
*** 3 DCL:2: VZHZ \$
*** 4 DCL:3: MLHZ \$
*** 5 DCL: ENDE \$
*** 6 BEGIN: \$
*** 7 IMP: SCHZ1 \$
*** 7 .PAR1 -> R2 \$
*** 8 .R2 AND 126 -> R2 \$
*** 9 .PAR1 -> R3 \$
*** 10 .R3 AND 128 -> R3, ZH, EQ \$
*** 11 JEQ -> POS \$
*** 12 .R2 XOR 126 -> R2 \$
*** 13 POS: \$
*** 14 .PAR2 -> R4 \$
*** 15 .R4 - 0 -> (R4), ZH, NE \$
*** 16 .R2 -> PAR3, SAVE ZH, NE \$
*** 17 .R3 -> VZHZ, SAVE ZH, EQ \$
*** 18 JNE -> NENULL \$
*** 19 .PAR1 -> R2 \$
*** 20 .R2 XOR 0 -> (R2), F, EQ \$
*** 21 JNE -> RETURN, SAVE ZH, NE \$
*** 22 NENULL: \$
*** 23 .PAR2 -> R2 \$
*** 24 .PAR1 -> R3 \$
*** 25 .R3 AND 1 -> R3 \$
*** 26 5 -> R1 \$
*** 27 SL: \$
*** 28 .R2 SL 1 -> R2, F, C \$
*** 29 .R3 SLC 1 -> R3 \$
*** 30 D1(R1)JNZ -> SL \$
*** 31 .VZHZ -> (R4), ZH, EQ \$
*** 32 .R2 -> PAR1, SAVE ZH, EQ \$
*** 33 .R3 -> PAR2, SAVE ZH, EQ \$
*** 34 JEQ -> POSHB \$
*** 35 128 -> R2 \$
*** 36 .R2 + .PAR2 -> R2 \$
*** 37 .R2 -> PAR2 \$
*** 38 J -> HBENDE \$
*** 39 POSHB: \$
*** 40 64 -> R2 \$
*** 41 .R2 + .PAR2 -> R2 \$
*** 42 .R2 -> PAR2 \$
*** 43 HBENDE: \$
*** 44 RETURN: \$
*** 45 RET:1:MAANP, SAVE F, EQ, V (\$
*** OUT1, OUT2, OUT3) \$

Um die Umsetzung beurteilen und das Verfahren der Registerbelegung verfolgen zu können, wurde ein Protokollierprogramm geschrieben, welches die maschinenangepaßten Dreiadreß befehle (abstraktes Programm nach Generatordurchlauf) und die augenblickliche Registerbelegung zwischen die erzeugten Einadreß befehle einblendet.

Das vollständige Programm ist in dem Anhang A5 aufgenommen. Einige Besonderheiten der Notation müssen noch erläutert werden:

- 1) Steht bei Ergebnisanzeigen erzeugenden Befehlen das Senkenregister in Klammern, bedeutet das, daß nur die Ergebnisanzeigen, nicht aber der Registerinhalt interessiert. Die Registerverwaltung stellt ein Register zur Verfügung, obwohl es bei manchen realen Maschinen nicht immer benötigt wird.
- 2) Die in Klammern stehende Nummer hinter Variablen ist die Nummer der Deklaration.
- 3) In den Registerbelegungszeilen bedeutet AN Altersnummer, ANI Altersnummer eines Registers, das eine wichtige Variable enthält, oder falls die Variable eingeklammert steht, zur Aufnahme einer wichtigen Variable vorgesehen ist.

BSPL-3 Diskussion

An diesem Beispielprogramm sei der Leser auf einige Glanzpunkte, aber auch noch auf Verbesserungsmöglichkeiten in der Hauptumsetzstufe hingewiesen, die nur teilweise in vorangegangenen Kapiteln besprochen wurden. Die folgenden Punkte beziehen sich auf die Abbildung A5.3a bis e im Anhang:

Zwischen Sprungbefehl 37 und Marke 39 wird erkannt, daß das Register R2 mit dem gleichen Objekt PAR3 wie vor dem Sprung belegt wird und sonst keine Register innerhalb der Schleife benötigt werden. Damit können Register-Räumoperationen entfallen. Zwischen Sprungbefehl 44 und 47 stört der Sprungbefehl 46, der das Freimachen der Register R2 und R3 vor Übersetzung des Sprungbefehls veranlaßt. Da der störende Sprung aber auf das Programmende zielt, dürften die Registerinhalte erhalten bleiben, weil am Schluß genau die Register, die Parameter enthalten, ohnehin freigeräumt werden. In Befehl 53 wird der Inhalt der Hilfszelle MLHZ durch "U" als unwichtig gekennzeichnet. Man stellt fest, daß diese Hilfszelle bis zu dieser Programmstelle nie belegt und auch bis zum Programmende nie mehr benötigt wird. Daher hätte auf die Deklaration dieser Variablen ganz verzichtet werden können. Bei der Übertragung auf die SIEMENS R30 erspart das aber nur eine "IDENT" Anweisung an den Assembler und führt zu keiner Programmverkürzung oder Laufzeiterparnis. Weiter fällt am Befehl 53 auf, daß das Linksverschieben um Null Bits keinen Einadreß - befehl erzeugt. Die Umsetzung dieses Befehls drückt sich allein in den Registerbelegungszeilen aus. Der Befehl 54 hat ein leeres Abbild. In der Schleife 63 bis 66 werden nur Variable benutzt, die in Registern stehen. Die Befehle 72 bis 76 bilden eine zweiseitige IF-Anweisung nach. Bei optimaler Übersetzung könnten einige Register-Abspeicherbefehle entfallen, da das Register R3 in keinem Zweig gebraucht wird und das Register R2 in beiden Zweigen mit PAR2 belegt bleibt. Beim Rücksprung befinden sich keine Parameter mehr in Registern; der Rücksprungbefehl wird kopiert, ohne daß zuvor Register abgespeichert werden. Betrachtet man rückschauend die Registerbelegung, stellt man fest, daß das vierte Register nie belegt wird. Man könnte eine erneute Umsetzung durchführen und dieses Register fest für eine Variable (VZHZ) benutzen. Das brächte in realen Laufzeitsystemen eine Ersparnis, da dann vor dem Aufruf des Laufzeitprogramms kein Zeiger auf lokale Daten gesetzt werden müßte.

ERGN Ergebnisse

ERGN-1. Anwendungen

Wie in ZIEL-1 berichtet, ist die Arbeit in die Entwicklung von PEARL-Übersetzungssystemen eingebettet. Portable Laufzeitprogramme wurden erstmals bei dem PEARL-Kompilierer für die SIEMENS 310 angewandt. Er wurde im Sommer 1979 der Öffentlichkeit vorgestellt. Da die SIEMENS 310 Simulationsroutinen für Gleitpunktrechnung im Betriebssystem integriert hat, wurden nur einige Routinen zur Zeichenkettenverarbeitung benötigt:

1. Laden von Zeichenketten in einen Zeichenkettenakku
2. Abspeichern des Zeichenkettenakkus mit Längen Anpassung durch Abschneiden bzw. Auffüllen mit Zwischenraumzeichen
3. Vergleich von Zeichenketten mit Rückmeldung in einem Hardware-Ergebnisanzeige bit
4. Konkatenation von Zeichenketten.

Die Programme wurden portabel formuliert und vom Prototyp des Umsetzverfahren bearbeitet. Die weitere Umsetzung erfolgte von Hand zur SIEMENS 306 und 310. Dabei wurden die in LOES-2 beschriebenen Optimierungen manuell durchgeführt, so daß der endgültige Code recht gut wurde.

Bei der Konkatenation wurde ein interessanter Vergleich mit dem vorhandenen Laufzeitprogramm des PEARL-Systems auf der SIEMENS 306 gemacht. Erstaunlicherweise war der generierte Code etwa um die Hälfte kürzer als der bestehende. Die Verbesserung ist auf den Algorithmus zurückzuführen. Das alte Programm entpackte die Zeichenketten, führte dann auf einfache Weise die Verkettung durch und packte anschließend wieder vier Zeichen in ein Wort. Das hatte seinerzeit den gewichtigen Vorteil einer optimalen Programmerstellungs- und -testzeit. Beim portablen Algorithmus mußte eine längere Programmerstellungszeit toleriert werden nach dem Motto: lieber einmaliger hoher, als mehrmals niedriger Aufwand.

ERGN-2 Höhere Kontrollstrukturen

Das erfreuliche Vergleichsergebnis der portablen mit der "alten" Zeichenkettenkonkatenation soll die noch vorhandenen Schwächen der Umsetzung, welche die Diskussion des Beispielprogramms gezeigt hat, nicht überdecken. Dort wurde auf die nicht erkannte zweiseitige IF-Struktur hingewiesen, die einige überflüssige Abspeicheroperationen zur Folge hatte. Ein naheliegender Verbesserungsvorschlag wäre, das Vorausschauprogramm, das bisher nur einfache Sprünge erkennt, zu erweitern, oder vor der Übersetzung ein Analyseprogramm mit der Suche solcher Kontrollstrukturen zu beauftragen. Man sollte jedoch bedenken, daß dabei eine Erkenntnis zu Tage gefördert wird, die der Ersteller des abstrakten Programms bereits in einem Struktogramm oder evtl. Flußdiagramm vor sich liegen hat. Um nun diese Strukturkenntnisse zwanglos an die Umsetzstufen weiterzugeben, sollte die Assemblersprache der abstrakten Dreiadreßmaschine um höhere Kontrollstrukturen erweitert werden, die aus höheren Programmiersprachen bekannt sind. Bisher erkennt die Hauptumsetzstufe die UNTIL- und FOR-Schleifen und die einseitige IF-Anweisung. Es fehlt die WHILE-Schleife, die zweiseitige IF-Struktur und das CASE-Element. In der Abbildung ERGN-2/1 sind die Kontrollstrukturen zusammen mit Assemblercode, den die Hauptumsetzstufe erzeugen müßte, aufgelistet. Zur Anpassung an das Assemblerniveau der abstrakten Maschine muß die Berechnung der boole'schen Ausdrücke (eq, lt in Abbildung ERGN-2/1) vorgezogen werden, um Ergebnisanzeigen zu setzen, so wie bisher auch vor den bedingten Sprüngen. Bei der Umsetzung der Kontrollstrukturen auf abstrakte Ein- bzw. Zweiadreßmaschinen müssen Namen vergeben werden. Die Erfindung der Namen kann auch dem Ersteller des abstrakten Programms überlassen werden, wenn man fordert, daß alle Kontrollstruktur-Anweisungen eine Marke tragen müssen. Dieser Name kann dann wie in der Abbildung ERGN-2/1 je nach Bedarf noch ergänzt werden. In der Praxis werden diese Ergänzungen maximal ein Zeichen lang sein. Es soll darauf hingewiesen werden, daß nicht in den Fehler verfallen wird, diese Kontrollstrukturen als Makros und den Code der Spalte "abstrakte Maschine" der Abbildung ERGN-2/1 als Makrokörper aufzufassen und den zur Auswertung der Generierparameter vorgesehenen Generator mit der Makroexpansion zu beauftragen. Dadurch wird die zur Registerverwaltung nötige Information über die Kontrollstruktur

wieder versteckt, so wie das bei Compilern noch üblich ist, die zur nachträglichen Optimierung wieder eine Kontrollflußanalyse nötig haben.





| Höhere Programmiersprache | Sprung- richtung | abstrakte Maschine | Kommentar |
|------------------------------------|---|--|---|
| M1: WHILE eq DO b OD |  | M1WHILE: \$... →,ZN,NE \$ JNE→M1 OD \$ b J → M1 WHILE M1OD: \$ | ...Befehlsfolge zur Berechnung von \overline{eq} beachte die boole'sche Negation |
| M2: UNTIL eq DO b OD |  | M2UNTIL: \$ b ... →,ZN,EQ \$ JEQ→M2UNTIL \$ | ...Befehlsfolge zur Berechnung von eq |
| M3: FOR I=1 STEP 1 TO e DO b OD |  | ... → SCHZ1 \$ M3FOR: \$ b D1JNZ→M3FOR | ...Befehlsfolge zur Berechnung von e |
| M4: IF eq THEN b FI |  | ... →,ZN,NE\$ JNE → M4FI \$ b M4FI: \$ | ...Befehlsfolge zur Berechnung von \overline{eq} beachte die boole'sche Negation |

Abb. ERGN-2/1: Konstrukte höherer Programmiersprachen und ihre Übersetzung auf abstrakte Einadreß maschinen

eq und lt steht für boole'sche Ausdrücke, b, b_1 , b_2 für beliebige Befehlsfolgen,

e ist ein arithmetischer Ausdruck für eine ganze Zahl


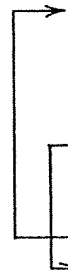
| | | | |
|---|---|--|--|
| <pre> M5: IF eq THEN b1 ELSE b2 FI oder IF eq THEN b2 ELSE b1 FI </pre> |  | <pre> ... → ,ZN, EQ JNE → M5ELSE \$ b1 J → M5FI \$ M5ELSE: \$ b2 M5FI: \$ </pre> | <p>...Befehlsfolge zur Berechnung von eq Die Negation erlaubt, daß der "ELSE Zweig" abgesetzt werden kann</p> |
| <pre> M6: CASE e OF n OUT : b₀ ALT1: b₁ [END1] : ALTn : b_n ESAC </pre> | | <pre> ... → I, ZN, LE \$ JLE → M6OUT \$ I - n → ,ZN, LT \$ JLT → M6 TAB → POINTER \$.POINTER + I → POINTER \$ J → .POINTER CONST: TAB: M6OUT \$ CONST: : M6ALT1 \$ CONST: : 6ALTn \$ OUT: b₀ J⁰ → M6ESAC M6ALT1: b₁ [J → M5ESAC] M6ALTn : b_n M6ESAC: \$ </pre> | <p>...Befehlsfolge zur Berechnung von e falls Adressen zwei Speichereinheiten belegen, muß I verdoppelt werden. Überlauf testen! Event. Laufzeitfehler</p> <p>fehlt END1, wird in die nächst. Alternative gesprungen</p> |
| <pre> M7: UNTIL eq DO b₁ EXIT lt b₂ OD </pre> |  | <pre> M7UNTIL: \$ b₁ ... → F, LT JLT → M7EXIT \$ b₂ ... → F, EQ \$ JEQ M7UNTIL \$ M7EXIT: \$ </pre> | <p>...Befehlsfolge zur Berechnung von eq ...Befehlsfolge zur Berechnung von lt. Bedingte vorzeitige Beendigung d. Schleife ...Befehlsfolge zur Berechnung von eq</p> |

Abb. ERGN-2/1: Fortsetzung

ERGN-3 Ausblick

Zu Beginn der Arbeit wurden von mehreren Seiten Zweifel am Gelingen des Vorhabens angemeldet: "Algorithmische Laufzeitprogramme können gar nicht portabel erstellt werden!" oder "Das gibt doch hoffnungslos ineffizienten Code!" Dennoch konnte gezeigt werden, daß auch algorithmische Laufzeitprogramme portabel erstellt werden können, wenn man gängige Kleinrechner wie SIEMENS 310, ZILOG 80, PDP-11 usw. als Zielrechner vorsieht. Mit Hilfe geeigneter Umsetzverfahren läßt sich auch effizienter Code erzeugen. Wegen des erzielten Erfolges wäre zu wünschen, daß die Arbeit keine reine Studie bleibt, obwohl denkbar wäre, daß in Zukunft Portabilitätsaufgaben an die Konstrukteure der Hardware übergeben werden. Bescheidene Ansätze dazu sind ja vorhanden. Es gibt einen IEEE Standardisierungsvorschlag für Gleitpunktoperationen [C0080], einen Vorschlag für Assemblersprachen für Mikrorechner [FIS79]. Die moderneren Kleinrechner tendieren alle zu 16-Bit-Wortlänge; allerdings glaubte man noch vor wenigen Jahren bei der Normierung von Prozeßperipherie [CAM72], daß 24-Bit-Rechner das Rennen machen. Trends können also täuschen. Der Wunsch nach standardisierter Hardware bleibt sicher noch jahrelang ein Traum.

A-1 Ergänzende Beschreibung der abstrakten Maschine

A-1.1 Operatoren, Ergebnisanzeigen

Die dyadischen Operatoren der abstrakten Maschine sind in Abbildung A-1.1/1 gelistet. Zusätzlich ist angegeben, welche Ergebnisanzeigen beeinflusst werden. Ein Stern bedeutet Erzeugung gemäß dem Ergebnis der vorliegenden Operation, ein Strich Erhaltung des Zustandes vor der Operation. Null steht für Löschen.

C bedeutet Übertragsbit (im Englischen Carry). Es zeigt bei Verschiebeoperationen und bei arithmetischen Befehlen einen Übertrag über die höchstsignifikante Bitstelle hinaus an.

Bei Addition zeigt es das aus der Grundschararithmetik bekannte "Merken", bei Subtraktion das "Borgen" an. Bei Subtraktion von Betragszahlen läßt sich aus dem Carrybit die größer-kleiner Relation bestimmen. V bedeutet Überlauf (Overflow). Es zeigt an, wenn der darstellbare Zahlenbereich vorzeichenbehafteter Zahlen überschritten wird ¹⁾. Z zeigt an, daß ein Ergebnis Null (Zero) und N, daß ein Ergebnis negativ ist (Negative). Zur Bestimmung der kleiner-größer Relation von vorzeichenbehafteten Zahlen dient die Exklusiv-Oder Verknüpfung der Überlauf- und Negativ-Bits.

- 1) Altgediente Systemprogrammierer, die sich nur am Rande mit Arithmetik befassen und Anwenderprogrammierer, die ihre Probleme nur in höheren Programmiersprachen formulieren, sind oftmals überrascht, daß zwischen Übertrag und Überlauf ein Unterschied besteht. (Es scheint auch Hardwarekonstrukteure gegeben zu haben, die kein Carry kannten. Die Softwareleute mußten dafür einen Ausgleich durch umständliche Programmierung schaffen). Der Unterschied wird sofort an einem kleinen Beispiel mit nur drei Bit Wortlänge klar. Bei der Addition von -2 und +3 entsteht ein Carry, das Ergebnis +1 ist jedoch durchaus richtig. Mathematisch korrekt lautet die Rechnung nämlich:
 $2^3 - 2 + 3 = 2^3 + 1$

$$\begin{array}{rcl} \text{Binäre Darstellung} & & \\ & 1\ 1\ 0 & = -2 = 2^3 - 2 \\ + & 0\ 1\ 1 & = +3 = +3 \\ \hline & & \\ \text{Carrybit } 1\ 0\ 0\ 1 & = & 1 = 2^3 + 1 \end{array}$$

Die Addition von 2+3 ergibt 5, ein Carry tritt nicht auf, aber das Ergebnis liegt nicht mehr im erlaubten Zahlenbereich.

$$\begin{array}{rcl} & 0\ 1\ 0 & = +2 \\ + & 0\ 1\ 1 & = +3 \\ \hline & & \end{array}$$

$$\text{Carrybit } 0\ 1\ 0\ 1 = -3 \text{ Überlauf!}$$

Unbekannt ist oft, daß mit einem überlaufbehafteten Ergebnis mitunter sogar weitergerechnet werden kann, wenn das Ergebnis wieder in den darstellbaren Zahlenbereich gelangt.

$$\begin{array}{rcl} 2 + 3 + (-2) & & \\ & 1\ 0\ 1 & \text{falsches Zwischenergebnis} \\ & 1\ 1\ 0 & = -2 \\ 1 & 0\ 1\ 1 & = +3 \text{ richtiges Endergebnis} \end{array}$$

| Mnemo code | Bedeutung | Ergebnisanzeigen | | | |
|---------------|---|------------------|---|---|---|
| | | C | V | Z | N |
| + | plus | * | * | * | * |
| +C+ | plus Übertragsbit plus | * | * | * | * |
| - | minus | * | * | * | * |
| -C- | minus Übertragsbit minus | * | * | * | * |
| * | multipliziert mit | * | * | * | * |
| / | dividiert durch mit Rundung zur Null hin | * | * | * | * |
| DIV | dividiert durch mit beliebiger Rundung | * | * | * | * |
| MOD | modulo | * | - | * | * |
| AND | logisch UND verknüpft mit | - | Ø | * | * |
| OR | logisch ODER verknüpft mit | - | Ø | * | * |
| EXOR | logisch exklusiv ODER verknüpft mit | - | Ø | * | * |
| BCL | Bits löschen | - | Ø | * | * |
| SL | logisch links geschoben | * | - | * | * |
| SR | logisch rechts geschoben | * | - | * | * |
| RL | links rotiert um | * | - | * | * |
| RR | rechts rotiert um | * | - | * | * |
| SAL | arithmetisch links geschoben ¹⁾ | * | * | * | * |
| SAR | arithmetisch rechts geschoben ²⁾ ohne Rundungsmaßnahmen | * | * | * | * |
| SARB | arithmetisch rechts geschoben mit beliebiger Rundung | * | * | * | * |
| SLC | logisch links verschoben | * | - | * | * |
| SRC | analog SLC nur rechts und "höchstwertige" | * | - | * | * |
| RLC | links rotiert um re.Op. unter Verlängerung des Wortes durch Übertragungsbit | * | - | * | * |
| RRC | analog RLC, nur rechts | * | - | * | * |

A-1.1/1 Dyadische Operatoren mit Ergebnisanzeigen

| | | |
|---|----------------------------|--|
| C | steht für Übertrag (Carry) | Ein Stern bedeutet Erzeugung gemäß der |
| V | " " Überlauf (Overflow) | vorliegenden Operation |
| Z | " " Null (Zero) | Ein Strich bedeutet Erhaltung des |
| N | " " Negativ (Negative) | Zustands vor der Operation |
| | | Null bedeutet Löschen |

- 1) SAL und SL unterscheiden sich durch die Ergebnisanzeigen.
- 2) Beachte: Es gibt Rechner (z.B. SIEMENS 310, 320, 330, 306), die beim arithmetischen Rechtsschieben negativer Zahlen eine Eins addieren, wenn während der Verschiebung eine Eins "herausgefallen" ist. Damit bleibt der Betrag des Ergebnisses beim Rechtsverschieben vom Vorzeichen unabhängig. Beachte den Einfluß auf die Portabilität von Rechenergebnissen.

Die Schiebeoperatoren werden in Abbildung A1-1/2 durch Bilder verdeutlicht:

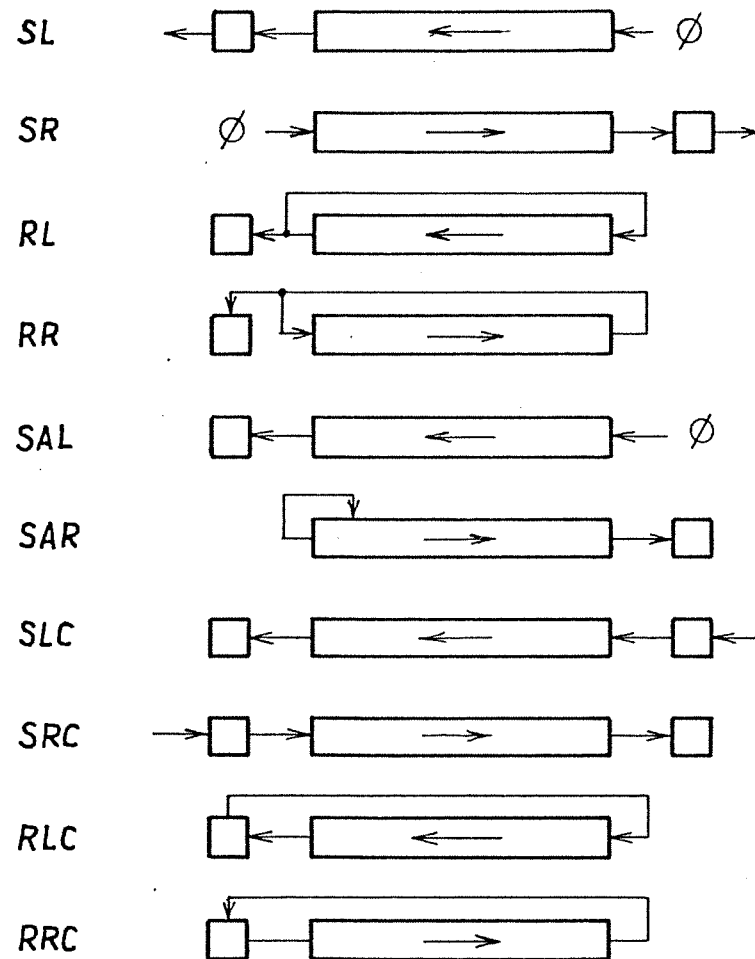


Abb. A-1.1/2: Schiebefehle mit Veranschaulichung durch Bilder

A-1.2 Sprungbefehle, Sprungbedingungen

Die abstrakte Dreiaßmaschine kennt unbedingte Sprünge:

J → Sprungziel

bedingte Sprünge:

J Bedingung → Sprungziel

und die in LOES-1.3 beschriebenen Unterprogrammsprünge und DiJNZ-Befehle aus Kapitel LOES-1.1.

Das Sprungziel verweist direkt oder in einer anderen Adressierungsart auf eine Marke. Marken stehen mit Doppelpunkt getrennt vor einem Befehl oder bilden selbständige Befehle (z.B. MARKE: \$). Zielen mehrere Sprünge auf die gleiche Programmstelle, sollte für jedes Sprungziel eine eigene Marke stehen, da die Hauptumsetzstufe dies fordert.

Als Bedingung wird eine bestimmte Auswahl boole'scher Verknüpfungen der vier Ergebnisanzeigen

C für (Carry) Übertrag

V für (Overflow) Überlauf des darstellbaren Zahlenbereichs

Z für (Zero) Null

N für Negativ

herangezogen.

Wie sich diese Anzeigen bilden, läßt sich aus den Befehlstabellen (Abbildung A-1.1/1 und /2 und A1.3) entnehmen.

Hier wird beschrieben, welche boole'schen Verknüpfungen dieser Anzeigen-Bits zugelassen sind. Die möglichen Verknüpfungen sind noch in Gruppen unterteilt.

Diese Gruppeneinteilung beruht auf einem Zugeständnis an die Architektur der SIEMENS 310, 320, 330. Bei den meisten Rechnern erfolgt eine Verknüpfung von Ergebnisanzeigebits erst nach Maßgabe einer in einem Sprungbefehl angegebenen Bedingung (z.B. erfolgt beim BGT Befehl (Branch if Greater Than) der PDP-11 ein Sprung genau dann, wenn $Z \vee (N \wedge V) = \emptyset$). Bei der genannten SIEMENS-Serie erfolgt parallel mit der Erzeugung der genannten Ergebnisanzeigen eine Umcodierung auf eine von vier Anzeigennummern. Nur die codierten Anzeigen können durch bedingte Sprungbefehle abgefragt werden. Sprungbedingungen sind

Veroderungen der vier Anzeigennummern. Die Umcodierung hängt vom Befehl ab. Es wird zwischen Operationen auf Betragszahlen, vorzeichenbehafteten Zahlen (Festpunktrechnung), Vergleichsoperationen, logischen Befehlen und für algorithmische Laufzeitprogramme nicht relevanten Ein/Ausgabebefehlen unterschieden. Abbildung A1.2/1 zeigt die Codierung der Ergebnisanzeigenbits auf der SIEMENS 310.

In der Abbildung A-1.2/2 sind die erlaubten Verknüpfungen und die in der Sprungbedingung anzugebenden mnemotechnischen Abkürzungen gelistet.

| An- zei- gen Nr. | Sprung- maske (F1- feld) | PZR- Bit 0 1 | E r g e b n i s b e i | | | | |
|---------------------------|------------------------------------|------------------------|---|---------------------------------------|---|---|--|
| | | | Fest- punkt- rech- nung (F) | Be- trags- rech- nung (B) | Ver- gleichs- befehlen (V) | boole- schen Be- fehlen (L) | Ein/Aus- gabe- Befehlen (EA) |
| 0 | 1000 | 0 0 | =0 | =0 | 1. Ope- rand gleich 2. Ope- rand | =0 | Quittung ohne Anzeigen, Datenan- forderung |
| 1 | 0100 | 0 1 | < 0 | - | 1. Ope- rand kleiner 2. Ope- rand | - | - |
| 2 | 0010 | 1 0 | > 0 | > 0 | 1. Ope- rand größer 2. Ope- rand | ≠ 0 | Quittung mit Anzeigen, Organi- sationsan- forderung |
| 3 | 0001 | 1 1 | Über- lauf | Über- lauf | - | - | - |

Abb. A-1.2/1: Codierung der Condition Code Bits auf der SIEMENS 310.

Aus: SIEMENS SYSTEME 300-16 BIT, Zentraleinheit 310S

Bestell-Nr. E STE 4-121/000

| | | | |
|---------------------------------------|-----|------------------|-------------------|
| $V \vee N = 1$ | LT | < | |
| $V \vee N = \emptyset$ | GE | \geq | |
| $Z = 1$ | EQ | = | Gruppe VF |
| $Z = \emptyset$ | NE | = | Vergleich bei |
| $Z \vee (N \vee V) = 1$ | LE | \leq | Festpunktrechnung |
| $Z \vee (N \vee V) = \emptyset$ | GT | \geq | |
| $C \vee Z = \emptyset$ | HIS | \geq | |
| $C \vee Z = 1$ | LO | < | |
| $Z = 1$ | EQ | = | Gruppe VB |
| $Z = \emptyset$ | NE | \neq | Vergleich bei |
| $C = 1$ | LOS | \leq | Betragsrechnung |
| $C = \emptyset$ | HI | > | |
| $V = 1$ | V | | |
| $V = \emptyset, N = 1$ | LT | < \emptyset | |
| $V = \emptyset, N = \emptyset$ | GE | $\geq \emptyset$ | |
| $V = \emptyset, Z = 1$ | EQ | = \emptyset | Gruppe F |
| $V = \emptyset, Z = \emptyset$ | NE | $\neq \emptyset$ | Festpunktrechnung |
| $V = \emptyset, N \vee Z = 1$ | LE | $\leq \emptyset$ | |
| $V = \emptyset, N \vee Z = \emptyset$ | GT | > \emptyset | |
| $C = 1$ | C | | |
| $C = 0$ | CQ | | |
| $V = 1$ | V | | |
| $V = \emptyset, Z = 1$ | EQ | = \emptyset | Gruppe B |
| $V = \emptyset, Z = \emptyset$ | NE | = \emptyset | Betragsrechnung |
| $N = 1$ | LT | < \emptyset | |
| $N = \emptyset$ | GE | $\geq \emptyset$ | |
| $Z = 1$ | EQ | = \emptyset | Gruppe ZN |
| $Z = \emptyset$ | NE | $\neq \emptyset$ | |
| $N \vee Z = 1$ | LE | $\leq \emptyset$ | |
| $N \vee Z = \emptyset$ | GT | > \emptyset | |

Abb.: A-1.2/2 Gruppenbildung bei Auswertung der Ergebnisanzeigen

Da die Abbildung einer Maschine dieses Typs auf eine Maschine des bezüglich der Ergebnisanzeigen anderen Typs einfach, der umgekehrte Fall aber nur durch umständliche Programmierung möglich ist, mußte sich die abstrakte Dreiadreßmaschine an die SIEMENS-Serie anlehnen

Die abstrakte Dreiadreßmaschine kennt fünf Gruppen von erlaubten Verknüpfungen der Ergebnisanzeigen.

Der Gruppe F entspricht den Ergebnisanzeigen der Festpunktrechnung, die Gruppe B entspricht den Ergebnisanzeigen der Betragsrechnung, der Gruppe VF und VB entspricht den Ergebnisanzeigen der Vergleichsbefehle der Siemens Lösung. Die Unterscheidung zwischen Vergleich für Festpunkt und Vergleich von Betragzahlen ist bei der SIEMENS im Operationscode enthalten.

Die Gruppe ZN entspricht einer Obermenge über die Ergebnisanzeige der boole'schen Befehle der SIEMENS-Maschinen.

Ob die Ergebnisanzeigen ausgewertet werden sollen und gegebenenfalls nach welcher Gruppe, muß bei jedem Befehl angegeben sein. Die Gruppenangabe kann durch eine Liste der Bedingungen ergänzt werden, auf die später abgefragt wird. Fehlt dieser Zusatz, so darf auf jede innerhalb der Gruppe erlaubte Bedingung abgefragt werden.

Es ist auch eine leere Senke möglich, dann ist aber die Angabe von Ergebnisanzeigen obligatorisch. Die Auswertung der Ergebnisanzeigen muß nicht unmittelbar an den erzeugenden Befehl anschließen. In diesem Fall müssen jedoch hinter jedem Befehl bis zum bedingten Sprungbefehl die Ergebnisanzeigen nach dem Schlüsselwort SAVE wiederholt werden.

Beispiel:

```
.I  -.J⇒K, F, GE, EQ$
.K  + 1⇒L, SAVE F, GE, EQ, C $
JGE ⇒ GREQ,      SAVE F, EQ, C $
JC  ⇒ CARR $
:
GREQ : JEQ ⇒ EQU $
EQU  : ...
:
CARR : ...
```

Das Beispiel zeigt, daß die Ergebnisanzeigen, die das Schlüsselwort SAVE über den nächsten Befehl hinweg rettet, beim Sprungbefehl weder bei erfüllter Sprungbedingung noch bei linearer Programmfortsetzung verloren gehen dürfen.

A-1.3 Transferbefehl, Sonderbefehle

Neben den Transfer-, Dreioperanden- und Sprungbefehlen gibt es auf der abstrakten Dreiadreßmaschine noch einige Sonderbefehle, die auch auf den meisten realen Rechnern vorhanden sind. Sie sind zusammen mit den Ergebnisanzeigen-Bits in der Abbildung A-1.3 gelistet.

| Sonderbefehl | Ergebnisanzeigen | | | | Kommentar |
|--------------------------------|------------------|---|---|---|------------------------------|
| | C | V | Z | N | |
| Quelle+1 \Rightarrow Senke | * | * | * | * | |
| Quelle-1 \Rightarrow Senke | * | * | * | * | |
| Quelle+C \Rightarrow Senke | * | * | * | * | |
| Quelle-C \Rightarrow Senke | * | * | * | * | |
| NEG Quelle \Rightarrow Senke | * | * | * | * | Negation im Zweierkomplement |
| NOT Quelle \Rightarrow Senke | - | Ø | * | * | Boole'sche Funktion |
| 1 \Rightarrow C | 1 | - | - | - | |
| 0 \Rightarrow C | 0 | - | - | - | |
| 1 \Rightarrow V | - | 1 | - | - | |
| 0 \Rightarrow V | - | 0 | - | - | |

Abb. A-1/3: Sonderbefehle

Die Sonderbehandlung der Addition von +1 bzw. -1 erfolgt in der Syntaxanalysestufe, wo ein Spezialoperator eingeführt wird. Sie kommt nicht in der Syntax zum Ausdruck.

Der Transferbefehl setzt dieselben Anzeigen wie die boole'sche Negation (NOT).


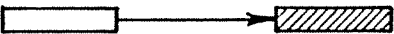


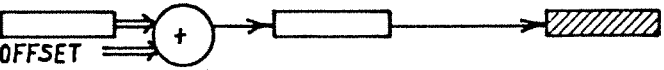

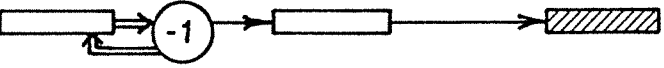
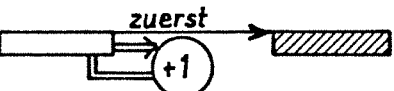
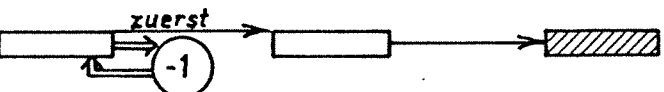
| Nummer der Adressierungsart | Menschen-lesbare Darstellung eine Quelle | Menschen-lesbare Darstellung eine Senke | Wirkungsweise |
|-----------------------------|--|---|--|
| 1a | 5 | - | Konstante |
| 1b | M | - | Adresse der Speicherzelle M |
| 2 | .M | M | M  |
| 3 | ..M | .M | M  |
| 4 | ...M | ..M | M  |
| 5 | ..M%offset | .M%offset | M  |
| 6 | ...M%offset | ..M%offset | M  |
| 7 | .v.M | v.M | M  |
| 8 | ..v.M | .v.M | M  |
| 9 | ..M' | .M' | M  |
| 10 | ...M' | ..M' | M  |

Abb. A-1.4: Adressierungsarten der abstrakten Dreiadreßmaschine

Die rechteckigen Kästen stellen Speicherzellen dar. Ein Einfachpfeil bedeutet: Der Inhalt der linken Zelle ist die Adresse der rechten Zelle. Die schraffierten Zellen enthalten den Operanden, bzw. bieten Platz für das Ergebnis. Ein Doppelpfeil steht für genau einen Datentransport. Die Kreise sind Rechenwerke, die als Ergebnis eine Adresse liefern. Die Punkte sind als "Inhalt von" zu lesen. Ein Pfeil nach unten bedeutet predecrement, das Hochkomma steht für postincrement.

A-1.4 Adressierungsarten

Die Abbildung A-1.4 enthält alle möglichen Referenzen und eine Veranschaulichung durch eine Zeichnung. Der Punkt zur Kennzeichnung des Inhalts einer Speicherzelle ist aus BLISS [WUL70] entnommen.

Das Ziel enthält immer einen Punkt weniger als eine Quelle. Konstante und Adreßkonstante (Adressierungsart 1 und 2) können verständlicherweise keine Senke bilden.

Beispiele zur Interpretation der Sonderzeichen der Abbildung 2-2/1:

- 1) `.M` Inhalt der Speicherzelle M
- 2) `↓.M` " " " " wird dekrementiert und bildet dann eine Adresse
- 3) `↓.M` Inhalt der in 2) definierten Adresse
- 4) `..M` Inhalt der Speicherzelle M ist eine Adresse. Inhalt der so adressierten Zelle.
- 5) `..M'` Inhalt der Speicherzelle M (innerer Punkt; `.M`) bildet eine zu merkende Adresse.
Der Inhalt der Zelle M ist um 1 zu erhöhen (Postincrementzeichen, `.M'`)
Der Inhalt (äußerer Punkt, `..M`) der durch die gemerkte Adresse bestimmten Speicherzelle.
- 6) `..M%` offset Inhalt der Speicherzelle M (innerer Punkt) addiert mit der Adreßabstandskonstante offset bildet eine Adresse.
Inhalt der so adressierten Zelle (äußerer Punkt).

Die logische Reihenfolge der Abarbeitung von innen nach außen erfordert eigentlich ein Klammerpaar z.B. `..(M% offset)`, das aber wegen Vermeidung von Schreibarbeit entfällt.

A-1.5 Direktoperanden, Konstante

Die abstrakte Maschine kennt als Direktoperanden Dezimalzahlen, Adreßkonstante und zwischen Fluchtsymbolen eingeschlossene arithmetische Ausdrücke (siehe LWEG-2). Bitmuster und Oktalzahlen können vom Generierparameter-Aufnahmeprogramm in Dezimalzahlen umgewandelt werden. Bitmuster (Oktalzahlen) bestehen aus einer Folge von Binärziffern (Oktalziffern), die durch bestimmte Sonderzeichenfolge (z.B. zwei Punkte) unterbrochen

oder beendet werden. Die Punkte bedeuten eine Wiederholung der letzten Ziffer links des Punktes zur Auffüllung auf die Wortlänge der realen Maschine. Durch diese Notation läßt sich auch ausdrücken, ob ein Bitmuster (Oktalzahl) links- oder rechtsbündig abgelegt sein soll. Beispielsweise gelangt das Bitmuster '101'B1 mit der Notation '1010..'B1 linksbündig und mit '0..101'B1 rechtsbündig in den Speicher. Bitmuster (Oktalzahlen), die sich nicht so darstellen lassen, (z.B. Null und Eins alternierend) prüft das Generierparameter-Aufnahmeprogramm auf Übereinstimmung mit der Wortlänge.

Indirekt zu adressierende Konstante und Adreß konstante sind wie folgt definiert

$$\left. \begin{array}{l} \text{CONST} \\ \text{ADRC} \end{array} \right\} \left[\begin{array}{l} \left[\langle \text{Konstantenname} \rangle :: \langle \text{Dezimalzahl} \rangle \mid \langle \text{Arithmetischer Ausdruck} \rangle \right] \\ \left[\langle \text{Adreß konstantenname} \rangle \mid \left[\langle \text{Markenname} \rangle \mid \langle \text{Konstantenname} \rangle \right] \right] \end{array} \right]$$

Selbstverständlich müssen die nach dem zweiten Doppelpunkt stehenden Marken-, Konstanten-, oder Adreß konstantennamen im Programm definiert sein. Dies zu überprüfen, ist Aufgabe des Assemblers.

A-2 Interne Darstellungen

Die Syntaxanalysestufe liest die abstrakten Dreiadreß programme satzweise von einer Lochkartenorientierten Eingabedatei. Dabei wird jeder Befehl einer Syntaxüberprüfung unterworfen und zu einer internen Struktur umcodiert. Der Aufbau der Struktur ist in der Abbildung A-2/1 am Beispiel eines Dreioperandenbefehls gezeigt. Die anderen Befehle sind ähnlich codiert.

Die Einadreß befehle werden in ähnlicher Weise intern in eine Struktur umcodiert. Statt der Strukturelemente für drei Adressen genügt eine Adresse. Der Optimierungshinweis U wird nicht mehr benötigt, da er von der Hauptumsetzstufe bereits ausgewertet wurde. Eine Fehlernummer ist nicht vorgesehen, jedoch werden zusätzliche Strukturelemente für die Registernummern und den Registeradressierungsmodus benötigt.

| Selektor | Offset zum Struktur- anfang | Inhalt des Struktur- elements | Typ bzw. Zahlenbereich | Quelle 1/2 Senke | |
|----------|-----------------------------------|---------------------------------------|---------------------------|---------------------|----|
| LAENGE | 0 | Länge der Struktur in Datenworten | 63 | Quelle 1 | |
| S1MODUS | 1 | Adressiermodus | 1 - 11 | " | |
| S1NAME | 2 | symbolischer Name | Zeichenkette | " | |
| S1OFFSET | 9 | Offsetkonstante oder Direktoperand | $0 - 2^{WL} - 1$ | " | 1) |
| S1L | 10 | Befehlszusatz löschen | True/False | " | |
| S1U | 11 | Befehlszusatz undefiniert | True/False | " | |
| S2MODUS | 12 | s.o. | s.o. | Quelle 2 | |
| S2NAME | 13 | s.o. | s.o. | " | |
| S2OFFSET | 20 | s.o. | s.o. | " | |
| S2L | 21 | s.o. | s.o. | " | |
| S2U | 22 | s.o. | s.o. | " | |
| DMODUS | 23 | s.o. | s.o. | Senke | |
| DNAME | 24 | | Zeichenkette | " | |
| DOFSET | 31 | s.o. | s.o. | " | |
| DOPTOR | 32 | Operator | 0 | - | |
| DCOND | 33 | Hinweise auf Er- gebnisanzeigen | 0-64 | - | |
| STNR | 34 | Befehlsnummer | 0-MAX | - 2) | |
| KNR | 35 | Kartenummer | 0-MAX | - 2) | |
| DMARKE | 36 | Marke | Zeichenkette | - | |
| FEHLER | 43 | Fehleranzeige | True/False | - 3) | |

Abb. A-2/1: Interne Darstellung eines Dreiadreß-Dreioperandenbefehls

1) WL = Wortlänge der Zielmaschine. Implementationsbedingt darf WL die Wortlänge der SIEMENS 306, 24 Bit nicht übersteigen

2) Global vereinbarte Konstante

3) Der Fehlertext überschreibt die Struktur

A-3 Dialog zum Erfassen der Generierparameter

In den folgenden vier Abbildungen sind Dialoge und Ergebnisse des Generierparameter-Aufnahmeprogramms wiedergegeben.

GLEITPUNKTZAHLENDARSTELLUNG

```

SOLL STANDARD-WORTLAENGE 16 VERWENDET WERDEN GIB FINES VON TF
F
GIB WL IN BITS
8
HAT DIE GPZ DIE DIE ABSTRAKTE FORM
TRUE -1
WIE LANG (IN MASCHINENWORTEN) IST EINE GPZ
2 2
PASST DIE MANTISSE IN EIN MASCHINENWORT
FALSE 0
IN WELCHEM WORT STEHT DAS VORZEICHEN
1 1
GIB BITMUSTER FUER HOCHES WORT DER MANTISSE
'00000001'B1 1

```

Abb. A-3/1: Ausschnitt aus dem Protokoll des Generierparameter-Aufnahmeprogramms, wenn die erste, Gleitpunktzahlen betreffende, Frage bejaht wird. Die Antwort ist unterstrichen. Sie wird in eine Dezimalzahl umgewandelt, die ebenfalls protokolliert ist.

HAT DIE GPZ DIE DIE ABSTRAKTE FORM
FALSE 0

WIE LANG (IN MASCHINENWORTEN) IST EINE GPZ
1 1

GIB BITMUSTER FUER MANTISSENMASKE
'11110000'B1 240

GIB BITMUSTER FUER EXPONENTENMASKE
'00001110'B1 14

GIB BITMUSTER FUER VORZEICHENMASKE
'00000001'B1 1

IST DER EXPONENT IN CHARAKTERISTIKFORM ABGESP
FALSE 0

IST DAS VERSTECKTE BIT GESPEICHERT
TRUE -1

GIB BITMUSTER FUER EXAKTE NULL
'00000001'B1 1

GIB BITMUSTER FUER MAXIMALEN EXPONENT
'00001100'B1 12

GIB BITMUSTER FUER MINIMALER EXPONENT
'00000010'B1 2

SIND BEI NEG. GPZ DIE EXPONENTEN EINERKOMPL.
FALSE 0

WIRD BEI NEG. GPZ VZ. UND BETRAG GESPEICHERT
TRUE -1

Abb. A-3/2: Wie Abb. A-3/1, nur Verneinung der ersten Frage

KONTROLLE DER GENERIERPARAMETER

| | | |
|--------------|-----------------------------|---|
| APWLA | :=8; 8 | Wortlänge |
| APGPABSTRA | :=TRUE; -1 | abstrakte Form? |
| APGPLA | :=2; 2 | Länge in Maschinenworten |
| APM1WA | :=FALSE; 0 | Mantisse in einem Maschinenwort |
| APVZA | :=1; 1 | Nr. des Wortes, in dem das Vorzeichen steht |
| APEXA | :=1; 1 | Nr. des Wortes, in dem der Exponent steht |
| APMA | := nur falls GPL=1 relevant | 0 |
| APNHA | :=1; 1 | Nr. Maschinenwort der Mantisse hoher Teil |
| APMLA | :=2; 2 | Nr. Maschinenwort der Mantisse niederer Teil |
| APVZMA | := '10000000'B1; 128 | Vorzeichenbit-Maske |
| APNMA | := nur falls GPL=1 relevant | 0 |
| APMLMA | := '11111111'B1; 255 | Maske für Mantisse niederer Teil |
| APMHMA | := '00000001'B1; 1 | Maske für Mantisse hoher Teil |
| APEXMA | := '01111110'B1; 126 | Maske für Exponent |
| APNULLA | := '00000000'B1; 0 | nur bei GPL=1 relevant |
| APNULLHA | := '00000000'B1; 0 | exakte Null hoher Teil |
| APNULLLA | := '00000000'B1; 0 | exakte Null niederer Teil |
| APCAREXA | :=TRUE; -1 | Exponent in Charakteristikform? |
| APCARVALA | := '01000000'B1; 64 | Wert der Charakteristik |
| APHBGESPA | :=FALSE; 0 | verstecktes Bit gespeichert? |
| APHBITSA | := '01000000'B1; 64 | verstecktes Bit |
| APNEGBITSA | := '10000000'B1; 128 | verstecktes Bit negiert |
| APHBA | :=1; 1 | entspricht HBGESP=FALSE |
| APMLVLA | :=0; 0 | Verschiebez. damit Mant.nied.Teil linksbündig |
| APMHVLA | :=0; 0 | Verschiebez. damit Mant.hoher Teil rechtsb. |
| APMVLA | :=5; 5 | Verschiebez. um gesamte Mant. linksb. |
| APMAXEXPA | := '01111110'B1; 126 | maximaler Exponent |
| APMINEXPA | := '00000000'B1; 0 | minimaler Exponent |
| APMAXEXDA | := '00010100'B1; 20 | maximale Exponentendifferenz |
| APEXPL1A | := '00000010'B1; 2 | Wert um Exponent zu inkrementieren |
| APPOSROUNDHA | := '00000000'B1; 0 | Wert um pos. Mant.hoher Teil zu runden |
| APPOSROUNDLA | := '00010000'B1; 16 | Wert um pos.Mant. niederer Teil zu runden |
| APNEGRUNDHA | := '00000000'B1; 0 | Wert um neg.Mant. hoher Teil zu runden |
| APNEGRUNDLA | := '00001111'B1; 15 | Wert um neg.Mant. niederer Teil zu runden |
| APNMEKPLA | :=TRUE; -1 | bei neg.Gleitpunktzahl Exponent komplementiert? |
| APMBETRA | :=FALSE; 0 | bei Mantisse Betrag und Vorzeichen |
| APEXLA | :=6; 6 | Länge des Exponenten |
| APMHLA | :=1; 1 | Länge Mantisse hoher Teil |
| APMLLA | :=8; 8 | Länge Mantisse niederer Teil |

Abb. A-3/3: Generierparameter, die das Generierparameter-Aufnahmeprogramm aus den wenigen eingegebenen Generierparametern erzeugt.
Vergleiche Abb. A-3/1.

KONTROLLE DER GENERIERPARAMETER

```

APWL#      :=8; 8
APGPABSTRA :=FALSE; 0
APGPL#     :=1; 1
APM1WA#    :=TRUE; -1
APVZA#     :=1; 1
APEXA#     :=1; 1
APMA#      :=1; 1
APIHA#     :=1; 1
APML#      :=1; 1
APVZMA#    :='00000001'B1; 1
APMMA#     :='11110000'B1; 240
APMLMA#    :='11110000'B1; 240
APMHMA#    :='00000000'B1; 0
APEXMA#    :='00001110'B1; 14
APNULL#    :='00000001'B1; 1
APNULLHA#  :='00000000'B1; 0
APNULLLL#  :='00000001'B1; 1
APCAREXA#  :=FALSE; 0
APCARVALA# :=
APHBGESPA# :=TRUE; -1
APHBITS#   :='01000000'B1; 64
APNEGHBITS#:= '10000000'B1; 128
APHB#      :=0; 0
APMLVL#    :=0; 0
APMHVR#    :=0; 0
APMVL#     :=7; 7
APMAXEXPA# :='00001100'B1; 12
APMINEXPA# :='00000010'B1; 2
APMAXEXDA# :='00001000'B1; 8
APEXPL1#   :='00000010'B1; 2
APPOSRUNDH#:= '00000100'B1; 4
APPOSRUNDL#:= '00000000'B1; 0
APNEGRUNDH#:= '00000011'B1; 3
APNEGRUNDL#:= '11111111'B1; 255
APNMEKPL#  :=FALSE; 0
APMBETRA#  :=TRUE; -1
APEXL#     :=3; 3
APMHL#     :=0; 0
APMLL#     :=4; 4

```

Abb. A-3/4: Generierparameter, die nach dem Dialog in Abb. A-3/2 entstehen. Die Kommentare sind aus Abb. A-3/3 zu entnehmen.

A-4 Struktogramme

Da die vier Unterprogramme Q1INRQ2INR bis Q1NICHTQ2NICHT des Struktogramms in Abbildung LOES-2.2/3 alle ähnlichen Aufbau besitzen, sei hier nur das Programm Q1INRQ2INR weiter ausgeführt, das sich wie die anderen drei Programme je nach Art des Befehls in vier Fälle gliedert:

- 1) $Q1 \otimes Q2 \leftrightarrow S$
- 2) $Q1, U \otimes Q2 \rightarrow S$ oder $S \otimes Q2 \rightarrow S$
- 3) $Q1 \otimes Q2, U \rightarrow S$ oder $Q1 \otimes S \rightarrow S$
- 4) $Q1, U \otimes Q2, U \rightarrow S$ oder $S \otimes S \rightarrow S$ oder $Q1, U \otimes S \rightarrow S$ oder $S \otimes Q2, U \rightarrow S$

Auch diese vier Struktogramme haben ähnlichen Aufbau, wie das Struktogramm in Abbildung A-4/1 zeigt. Da die Abbildung zwei Seiten füllt, sind beide Teilstruktogramme mit Kommentarspalten versehen, die den ähnlichen Aufbau unterstreichen.

| siehe Seite 72 siehe Seite 73 | | Q1/NR Q2/NR (1, 2) | | | |
|---------------------------------------|----------------------------------|--------------------|----------------------------------|--------------------|------------|
| Kommentar | $Q1 \oplus Q2 \rightarrow S$ | | $Q1, U \oplus Q2 \rightarrow S$ | | |
| | | | $S \oplus Q2 \rightarrow S$ | | |
| | | | | | |
| $R \oplus R \rightarrow R$ existiert | $R * R \rightarrow R \text{ ex}$ | | $R * R \rightarrow R \text{ ex}$ | | |
| inverser Operator | | | | | |
| freies Register 1 Registertransfer | $R \rightarrow R$ | | | | |
| Behandlung von Quelle 1 | ja | ex freies Register | ja | ex freies Register | ja |
| | RF = SUCHR | RF = SUCHR | RF = SUCHR | RF = SUCHR | RF = SUCHR |
| | RF = SUCHR | RF = SUCHR | RF = SUCHR | RF = SUCHR | RF = SUCHR |
| Behandlung von Quelle 2 | ja | ex freies Register | ja | ex freies Register | ja |
| | RF = SUCHR | RF = SUCHR | RF = SUCHR | RF = SUCHR | RF = SUCHR |
| | RF = SUCHR | RF = SUCHR | RF = SUCHR | RF = SUCHR | RF = SUCHR |
| \oplus - Befehl | $RQ1 * RQ2 \rightarrow RQ1$ | | $RQ1 * RQ2 \rightarrow RQ1$ | | |
| Merke | | | | | |
| | | | | | |
| | | | | | |
| Registerbelegung | RF mit S belegen | | RF mit S belegen | | |

Abb. A-4/1 Teil 1: Verfeinerung des Struktogramms in Abb. LOES-2.2/3
SUCHR ist ein Unterprogramm zum Suchen eines freien Registers

| | | | | | | | | | | |
|----------------------------------|----------------------|------|--|--|--|--|--|--|--|--|
| siehe Seite 72 siehe Seite 73 | Q1 INR Q2 / NR (3,4) | | | | | | | | | |
| Kommentar | Q1 * Q2, U → S | | | | | | | | | |
| | Q1 * S → S | | | | | | | | | |
| | Q1, U * S → S | | | | | | | | | |
| | S * Q2, U → S | | | | | | | | | |
| R ⊗ R → R existiert | R * R → R ex | | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| inverser Operator | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| freies Register | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| Behandlung von Quelle 1 | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| Behandlung von Quelle 2 | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| ⊗ - Befehl | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| Merke Register-belegung | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |
| | ja | nein | | | | | | | | |

A-5 Listings

Die folgenden Abbildungen zeigen die verschiedenen Stufen der Umsetzung eines abstrakten Programms zum Ausblenden von Exponent und Mantisse unter gleichzeitiger Anpassung an die Zahlendarstellung verschiedener Rechner.

A-5.1a-c zeigt das parametrisierte Programm

A-5.2a-b zeigt das maschinenangepaßte Programm

A-5.3a-e zeigt das Programm nach Hauptumsetzstufe und Protokollierablauf.

```

/.*****-----*****-----*****-----*****-----*****
/.      GLEITPUNKT-MANTISSENANPASSUNG
/.*****-----*****-----*****-----*****

```

```

/. EINGABEPARAMETER:
/. NIEDERES ( UND HOHES WORT) EINER GLEITPUNKTZAHL <GPZ>
/. IN WEITGENÜND BELIEBIGER FORM
/. RUECKGABEPARAMETER:
/. 1.) NIEDERER TEIL DER MANTISSE <ML> UND
/. 2.) HOHER TEIL DER MANTISSE <MH>
/. ERGAENZT DURCH ZWEI 'VERSTECKTE BITS' 0.1 BEI POSITIVEN
/. UND 1.0 BEI NEGATIVEN GPZ. DIE MANTISSE STEHT LINKSBUENDIG
/. SODASS EVENTUELL ML=0.
/. 3.) EXPONENT AUSGEBLENDET, EVENTUELL EINERKOMPLEMENTIERT

```

```

/. ERGEBNISANZEIGEN DER GRUPPE F

```

```

      SUBIN:1:MAANP*
IF GPL EQ 1 THEN
      (IN1,OUT2,OUT3)$
/.      GPZ      EINGABE
/.      ML ,MH ,EXP      RUECKGABE
FI
IF GPL EQ 2 THEN
      (IN1,IN2,OUT3)$
/.      GPZ,GPZ      EINGABE
/.      ML ,MH ,EXP      RUECKGABE
FI

```

```

DCL:1: SCHZ1 $
DCL:2: VZHZ $
DCL:3: MLHZ $
DCL:ENDE $

```

```

BEGIN: $
IMP: SCHZ1 $

```

```

      .PARAEXA AND AEXMAA -> PAR3 $      EXPONENT AUSBLENDEN

```

```

IF NMEKPL EQ FALSE THEN
      .PARAVZA AND AVZMAA -> VZHZ $      VORZEICHEN AUSBLENDEN
FI

```

```

IF NMEKPL EQ TRUE THEN
      .PARAVZA AND AVZMAA -> VZHZ, ZN.EQ $      VORZEICHEN AUSBLENDEN
      JEQ -> POS $
      .PAR3 XOR AEXMAA -> PAR3 $
      POS:$
FI

```

```

      /. TEST AUF EXAKTE NULL
IF M1W EQ TRUE THEN
      .PARAMLA XOR ANULLLA -> ,F,EQ $      XOR-BEFEHL ERZEUGT V=0
      JEQ -> RETURN,SAVE F,EQ $
FI

```

```

IF M1W EQ FALSE THEN

```

Abb. A-5/1a: Abstraktes Dreiadreß programm zum Ausblenden von Exponent und Mantisse unter gleichzeitiger Anpassung an die Zahlendarstellung verschiedener Zielrechner

```

.PARAMLA - #VULLLA -> ,ZN,NE $
JNE -> NENULL $
.PARAMHA XOR #VULLHA -> ,F,EO $      XOR-BEFEHL ERZEUGT V=0
JEQ -> RETURN,SAVE F,EO $
NENULL:$

```

FI

```

.PARAMLA AND #MLMA -> MLHZ $      MANTISSE NIEDERER TEIL
.PARAMHA AND #MHMA -> PAR2 $      MANTISSE HOHER TEIL
/. DER LETZTE BEFEHL ENTFAEHLT BEI DER UMSETZUNG
/. FALLS ER ZU .PAR2 AND 11...1 -> PAR2 ENTARTET
.MLHZ,U SL #VLVLA -> PAR1 $      NIEDERER TEIL LINKSBUENDIG
.PAR2 SR #MHVRA -> PAR2 $      HOHER TEIL RECHTSBUENDIG
/. JETZT STEHT Z.B. EINE 10-BIT MANT. SO IN ZWEI
/. 8 BIT WORTEN 00001111,11111100

/. SIE WIRD NUN DOPPELT LINKS BZW. RECHTS VERSCHOBEN,DASS NOCH
/. PLATZ FUER ZWEI VERSTECKTE BITS BLEIBT

```

```

IF MVL GT 0 THEN
  IF MVL GT 1 THEN
    #MVL -> SCHZ1$
  FI

```

```

SL:.PAR1 SL 1 -> PAR1,C,C $
.PAR2 SLC 1 -> PAR2 $

```

```

IF MVL GT 1 THEN
  DIJNZ -> SL$

```

FI

FI

```

IF MVL LT 0 THEN
  .PAR2 SR 1 -> PAR2,C,C $
  .PAR1 SRC 1 -> PAR1 $

```

FI

```

IF HBGESP EQ FALSE THEN
  /. VERSTECKTE BITS EINBLENDEN
  IF MBETR EQ TRUE
    #HBITS + .PAR2 -> PAR2 $
  FI

```

```

IF MBETR EQ FALSE THEN
  .VZHZ -> ,ZN,EQ $
  JEQ -> POSHB $
  #NEGBITS + .PAR2 -> PAR2 $
  J -> HBENDE $
  POSHB: #HBITS + .PAR2 -> PAR2 $
  HBENDE:$

```

FI

FI

```

IF MBETR EQ TRUE THEN
  /. BEI NEGATIVER GLEITPUNKTZAHL WIRD DER
  /. BETRAG DER MANTISSE NEGIIERT
  .VZHZ,U -> ,F,EQ $
  /. DER TESTBEFEHL (TRANSFERBEFEHL OHNE ZIEL) ERZEUGT V=0
  JEQ -> RETURN,SAVE F,EQ $
  NEG .PAR1 -> PAR1,C,C $

```

```

/. SONDERBEHANDLUNG FÜR MANTISSE 1/2
.PAR2 +C → PAR2 $
NEG .PAR2 → PAR2 $
.PAR2 - 1NEGHBITS1+1HBITS1 → .F,NE $ MANTISSE WAR 1/2
/. DER SUBTRAKTIONSBEFEHL ERZEUGT ALTERNATIV AUCH V
JNE → RETURN,SAVE F,NE$
.PAR3 - 1MINEXP1 → .ZN,NE $
/. 1/2 MIT MINIMALEM EXPONENTEN IST NICHT ERLAUBT
JNE → EXMIN1 $
NEG 1NEGHBITS1 → .F,V $ V=1 ZEIGT UNTERLAUF AN
JV → RETURN $

```

```
EXMIN1:
```

```
.PAR3 - 1EXPL11 → PAR3,F,V $
```

```
FI
```

```
RETURN:$
```

```
RET:1:MAANP,SAVE F,EQ *(OUT,OUT,OUT) $
```

```
1ENDE1$
```

Abb. A-5/1c: Fortsetzung

```

/.....
/.      GLEITPUNKT-MANTISSENANPASSUNG
/.....

/. EINGABEPARAMETER:
/. NIEDERES ( UND HOHERS WORT) EINER GLEITPUNKTZAHL (GPZ)
/. IN WEITGENEEND BELIEBIGER FORM
/. RUECKGABEPARAMETER:
/. 1.) NIEDERER TEIL DER MANTISSE (ML) UND
/. 2.) HOHER TEIL DER MANTISSE (MH)
/. ERGAENZT DURCH ZWEI "VERSTECKTE BITS" 0.1 BEI POSITIVEN
/. UND 1.0 BEI NEGATIVEN GPZ. DIE MANTISSE STEHT LINKSBUENDIG
/. SODASS EVENTUELL ML=0.
/. 3.) EXPONENT AUSGELENDET, EVENTUELL EINERKOMPLEMENTIERT

/. ERGEBNISANZEIGEN DER GRUPPE F

      SUBIN:1:MAANP*
      (IN1,IN2,OUT3)S
/.      GPZ,GPZ      EINGABE
/.      ML, MH ,EXP  RUECKGABE

DCL:1: SCHZ1 $
DCL:2: VZHZ $
DCL:3: MLHZ $
DCL:ENDE $

BEGIN: $
IMP: SCHZ1 $

      .PAR1 AND 126 -> PAR3 $      EXPONENT AUSBLENDEN

      .PAR1 AND 128 -> VZHZ, ZN,EQ $  VORZEICHEN AUSBLENDEN
      JEQ -> POS $
      .PAR3 XOR 126 -> PAR3 $
POS:$

      /. TEST AUF EXAKTE NULL

      .PAR2 - 0 -> ,ZN,NE $
      JNE -> NENULL $
      .PAR1 XOR 0 -> ,F,EQ $      XOR-BEFEHL ERZEUGT V=0
      JEQ -> RETURN,SAVE F,EQ $
NENULL:$

      .PAR2 AND 255 -> MLHZ $      MANTISSE NIEDERER TEIL
      .PAR1 AND 1 -> PAR2 $      MANTISSE HOHER TEIL
      /. DER LETZTE BEFEHL ENTFAEHLT BEI DER UMSETZUNG
      /. FALLS ER ZU .PAR2 AND 11...1 -> PAR2 ENTARTET
      .MLHZ,U SL 0 -> PAR1 $      NIEDERER TEIL LINKSBUENDIG
      .PAR2 SR 0 -> PAR2 $      HOHER TEIL RECHTSBUENDIG
      /. JETZT STEHT Z.B. EINE 10-BIT MANT. SO IN ZWEI
      /. 8 BIT WORTEN 00001111,11111100

      /. SIE WIRD NUN DOPPELT LINKS BZW. RECHTS VERSCHOBEN,DASS NOCH
      /. PLATZ FUER ZWEI VERSTECKTE BITS BLEIBT

```

S → SCHZ14

SL: .PAR1 SL 1 → PAR1, C, C \$
 .PAR2 SLC 1 → PAR2 \$

D1JNZ → SL\$

/ . VERSTECKTE BITS EINBLENDEN

.V7HZ → ,ZN,EO \$
 JEQ → POSHB \$
 12K + .PAR2 → PAR2 \$
 J → HBENDE \$
 POSHB: 64 + .PAR2 → PAR2 \$
 HBENDE:\$

RETURN:\$

RET:1:MAANP,SAVE F,EQ *(OUT,OUT,OUT) \$

4ENDE4\$

Abb. A-5/2b: Fortsetzung

FAP-DAB-DAZ-NUMBER

```

-----
1 /.*****
2 /.      Gleitpunkt-Mantissenanpassung
3 /.*****
4
5 /. EINGABEPARAMETER:
6 /. NIEDERES ( UND HOHES WORT ) FINER GLEITPUNKTZAHL {GPZ}
7 /. IN WEITGENÜND BELIEBIGER FORM
8 /. RUECKGABEPARAMETER:
9 /. 1.) NIEDERER TEIL DER MANTISSE {ML} UND
10 /. 2.) HOHER TEIL DER MANTISSE {MH}
11 /. ERGAENZT DURCH ZWEI 'VERSTECKTE BITS' 0.1 BEI POSITIVEN
12 /. UND 1.0 BEI NEGATIVEN GPZ. DIE MANTISSE STEHT LINKSBUENDIG
13 /. SODASS EVENTUELL ML=0.
14 /. 3.) EXPONENT AUSGEBLENDET, EVENTUELL EINERKOMPLEMENTIERT
15
16 /. ERGEBNISANZEIGEN DER GRUPPE F
17
18      SUBIN:1:MAANP*
19      (IN1,IN2,OUT3)$

      1      1      18      :SUBIN:1:MAANP*(IN1,IN2,OUT3,)$
-----
20 /.      GPZ,GPZ      EINGABE
21 /.      ML, MH ,EXP  RUECKGABE
22
23      DCL:1: SCHZ1 $

      2      2      23      :DCL:1:SCHZ1$
-----
24      DCL:2: VZHZ $

      3      3      24      :DCL:2:VZHZ$
-----
25      DCL:3: MLHZ $

      4      4      25      :DCL:3:MLHZ$
-----
26      DCL:ENDE $

      5      5      26      :DCL:ENDE$
-----
27
28
29      BEGIN: $

      6      6      29      :BEGIN:$
-----
30      IMP:SCHZ1 $

      7      7      30      :IMP:SCHZ1$

```

Abb. A-5/3a: Programm von Abbildung A-5/1 nach Hauptumsetzstufe und Protokollierlauf. Die Zeilen mit der Kommentarzeichenfolge "/." sind aus dem Profiadressprogramm

31
32
33

.PAR1 AND 126 → PAR3 \$ EXPONENT AUSBLENDEN

7 8 33 : .PAR1 → R2\$
5 8 33 : R2 AND 126 → R2\$

***** ANI1(.SCHZ1) , AN2 .PAR3 , -- ,--

34
35
36

.PAR1 AND 128 → VZHZ, ZN.EQ \$ VORZEICHEN AUSBLENDEN

9 9 36 : .PAR1 → R3\$
10 9 36 : R3 AND 128 → R3, ZN.EQ\$

***** ANI1(.SCHZ1) , AN2 .PAR3 , AN3 .VZHZ ,
***** --

37 JEQ → POS \$

11 10 37 : JEQ → POS\$

38 .PAR3 XOR 126 → PAR3 \$

12 11 38 : R2 XOR 126 → R2\$

***** ANI1(.SCHZ1) , AN3 .PAR3 , AN2 .VZHZ ,
***** --

39 POS: \$

13 12 39 POS: \$

40

41 /. TEST AUF EXAKTE NULL

42

43 .PAR2 - 0 → ,ZN,NE \$

14 13 43 : .PAR2 → R4\$
15 13 43 : R4 - 0 → (R4), ZN,NE\$

***** ANI1(.SCHZ1) , AN3 .PAR3 , AN2 .VZHZ ,
***** --

44 JNE → NENULL \$

16 14 44 : R2 → PAR3 ,SAVE ZN,NE\$
17 14 44 : R3 → VZHZ(2) ,SAVE ZN,NE\$
18 14 44 : JNE → NENULL\$

Abb. A-5/3b: direkt übernommen (siehe Abb. A-5/2). Die schwach eingerückten Zeilen sind die Dreiadreß befehle, die stark eingerückten Zeilen enthalten die durchnumerierten Einadreß befehle

45 .PAR1 XOR 0 -> ,F, EQ \$ XOR-BEFEHL ERZEUGT V=0

14 15 45 : .PAR1 -> R2\$
20 15 45 : R2 XOR 0 -> (R2), F, EQ \$

***** ANI1(.SCHZ1) , -- , -- , --

46 JEQ -> RETURN, SAVE F, EQ \$

21 16 46 : JEQ -> RETURN , SAVE F, EQ \$

47 NENULL: \$

22 17 47 NENULL: \$

48

49 .PAR2 AND 255 -> MLHZ \$ MANTISSE NIEDERER TEIL

23 18 49 : .PAR2 -> R2\$

***** ANI1(.SCHZ1) , AN2 .MLHZ , -- , --

50 .PAR1 AND 1 -> PAR2 \$ MANTISSE HOHER TEIL

24 19 50 : .PAR1 -> R3\$
25 19 50 : R3 AND 1 -> R3\$

***** ANI1(.SCHZ1) , AN2 .MLHZ , AN3 .PAR2 ,
***** --

51 / . DER LETZTE BEFEHL ENTFALLT BEI DER UMSETZUNG

52 / . FALLS ER ZU .PAR2 AND 11...1 -> PAR2 ENTARTET

53 .MLHZ, U SL 0 -> PAR1 \$ NIEDERER TEIL LINKSBUENDIG

***** ANI1(.SCHZ1) , AN3 .PAR1 , AN2 .PAR2 ,
***** --

54 .PAR2 SR 0 -> PAR2 \$ HOHER TEIL RECHTSBUENDIG

55 / . JETZT STEHT Z.B. EINE 10-BIT MANT. SO IN ZWEI

56 / . 8 BIT WORTEN 00001111, 11111100

57

58 / . SIE WIRD NUN DOPPELT LINKS BZW. RECHTS VERSCHOBEN, DASS NOC

59 / . PLATZ FUER ZWEI VERSTECKTE BITS BLEIBT

60

61 5 -> SCHZ1\$

26 22 61 : 5 -> R1\$

***** ANI1.SCHZ1 , AN3 .PAR1 , AN2 .PAR2 ,
***** --

Abb. A-5/3c Nach der Einadreß befehls- (EAB-) Nummer folgen die
Dreiadreß befehls- (OAB-)nummer und die Dreiadreß zeilen-

62

63 SL: .PAR1 SL 1 → PAR1, C, C \$

27 23 63 SL: \$

28 23 63 : R2 SL 1 → R2, C, C \$

***** ANI1.SCHZ1 , AN3 .PAR1 , AN2 .PAR2 ,

***** --

64

.PAR2 SLC 1 → PAR2 \$

29 24 64 : R3 SLC 1 → R3 \$

***** ANI1.SCHZ1 , AN2 .PAR1 , AN3 .PAR2 ,

***** --

65

66 D1JNZ → SL \$

30 25 66 : D1(R1)JNZ → SL \$

67

68

69 / . VERSTECKTE BITS EINBLENDEN

70

71 .VZHZ → ,ZN,EQ \$

31 26 71 : .VZHZ(2) → (R4), ZN,EQ \$

***** ANI1.SCHZ1 , AN2 .PAR1 , AN3 .PAR2 ,

***** --

72

J EQ → POSHB \$

32 27 72 : R2 → PAR1 ,SAVE ZN,EQ \$

33 27 72 : R3 → PAR2 ,SAVE ZN,EQ \$

34 27 72 : J EQ → POSHB \$

73

128 + .PAR2 → PAR2 \$

35 28 73 : 128 → R2 \$

36 28 73 : R2 + .PAR2 → R2 \$

***** ANI1.SCHZ1 , AN2 .PAR2 , -- , --

74

J → HBENDE \$

37 29 74 : R2 → PAR2 \$

38 29 74 : J → HBENDE \$

75

POSHB: 64 + .PAR2 → PAR2 \$

Abb. A-5/3d: Die mit zehn Sternen beginnenden Zeilen enthalten die augenblickliche Registerbelegung.

```

32 30 75 PUSHB: $
40 30 75      :64 -> P2$
41 30 75      : R2 + .PAR2 -> R2$

```

```

***** ANI1.SCH21 , AN2 .PAR2 , -- ,--

```

```

-----
76 HBENDE:$

```

```

42 31 76      :R2 -> PAR2$
43 31 76 HBENDE: $

```

```

-----
77
78
79 RETURN:$

```

```

44 32 79 RETURN: $

```

```

-----
80
81 RET:1:MAANP,SAVE F,EQ *(OUT,OUT,OUT) $

```

```

45 33 81      :RET:1:MAANP*(OUT1,OUT2,OUT3,)$

```

```

-----
82
83
84 ENDE$

```

```

46 33 81      :ENDE$

```

Abb. A-5/3e AN steht für Altersnummer, ANI heißt Altersnummer eines Registers für eine wichtige Variable. (Altersnummern sind in Abschnitt 3.3 erklärt.) Auf die Altersnummer folgt die gespeicherte Variable mit Adressierungsart, notiert wie bei Quellen. Die Variable steht in Klammern, wenn das Register einstweilen für wichtige Variable reserviert, aber noch nicht belegt ist. Freie Register sind durch "--" gekennzeichnet.

LITE- Literaturverzeichnis

- AH078 Aho, A.V., Ullman, J.D.:
Principles of Compiler Design.
Addison Wesley Publishing Company, Reading Mass., 1978
- AMD79 Advanced Micro Devices:
AM 9511A Arithmetic Processor, Advanced Micro Devices,
Advanced MOS/LSI, Datenblatt 1979
- BEA72 Beatty, J.C.:
A global Register Assignment Algorithm.
In R. Rustin (ed) Design and Optimization of Compilers,
Prentice Hall, Englewood Cliffs, N.J. 1972
- BR067 Brown, P.J.:
The ML/I Macro Processor
Communications of the ACM, 10, 10 (1967)
- BR072 Brown, P.J.:
Levels of Languages for Portable Software
Communications of the ACM 15, 12 (1972)
- BR076 Brown, W.S., Hall, A.D.:
FORTRAN Portability via Models and Tools.
In Lecture Notes in Computer Science, Portability of
Numerical Software, Workshop, Oak Brook, Illinois, 1976
- CAM72 CAMAC, A Modular Instrumentation System for Data Handling,
Commission of the European Communities, EUR 4100e,
revised version 1972
Joint Nuclear Research Centre Ispra Establishment - Italy
ESONE Committee
- CO080 Coonen, J.T.:
An Implementation Guide to a Proposed Standard for Floating-
Point-Arithmetic
Computer, 1 (1980)
- DIN78 DIN 66253
Basic PEARL Draft Standard
Beuth Verlag GmbH, Berlin/Cologne, 1978
- EIC77 Eichenauer, B.F., Henn, R.:
Spezifikation der Zwischensprache CIMIC-P (PEARL Set).
Gesellschaft für Prozessprogrammierung mbH, 2 (1977)
- FAE78 Faerber, G.:
Fachtagung Prozeßlenkung mit Mikroprozessoren in Karlsruhe
Projekt PDV der Gesellschaft für Kernforschung mbH, Karlsruhe
Bericht KfK-PDV-165, 12 (1978)
- FIS79 Fischer, W.P.:
Microprocessor Assembly Language Draft Standard
Computer, 12 (1979)
- FLE79 Fleischmann, A.:
Anpassung eines rechnerunabhängigen PEARL-Betriebssystems
für die Siemens 310
Studienarbeit im Fach Informatik, Erlangen, 1979
- GRI71 Fries, D.:
Compiler Construction for Digital Computers
John Wiley & Sons, Inc., New York, London, Sydney, Toronto, 1971

- GRU76 Gruber, Inderst, Piche:
 Programmierung für das ASME 1-PEARL-Subset.
 Projekt PDV der Gesellschaft für Kernforschung mbH Karlsruhe,
 Bericht KfK-PDV-100, 11 (1976)
- GUN72 Güntsch, F.R., Schneider, H.-J.:
 Einführung in die Programmierung digitaler Rechenautomaten
 Walter de Gruyter, Berlin-New York, 1972
- HAS55 Hastings, C.JR.:
 Approximations for Digital Computers
 Princeton New Jersey, Princeton University Press 1955
- HIL68 Hill, I.D.:
 Procedures for the Basic Arithmetic Operations in Multiple
 Length Working. Algorithm 34 in Computer Journal, Vol. 11 (1969)
- HOF78 Hofmann, F.:
 Persönliche Mitteilung, Erlangen 1978
- HOL76 Holleczeck, P.:
 Stufe 2 - ASME-PEARL-Compilersystem-Grundideen.
 Projekt PDV der Gesellschaft für Kernforschung mbH, Karlsruhe,
 Entwicklungsnotiz PDV-E89, 10 (1976)
- IVE62 Iverson, K.E.:
 A programming language
 Wiley, New York, 1962
- JOH71 Johnsohn, G.R., Mueller, R.A.:
 Automated Generation of Cross-System Software for Mikrocomputers,
 COMPUTER, Microprocessors and Education, 1 (1977)
- McK65 McKeemann, W.M.
 Peephole Optimization
 Communication of the ACM, 8 (1965)
- KIM79 Kimm, Koch, Simonsmeier, Tontsch:
 Einführung in Software Engineering
 Walter de Gruyter, Berlin, New York 1979
- LAM78 Lampe, K.:
 Erstellung der Standard-E/A Laufzeitroutinen des PEARL-ASME-
 Stufe-1-Compilersystems in PEARL.
 Diplomarbeit im Fach Informatik, Erlangen 1978
- MUE76 Mühlhahn:
 Spezifikation CIMIC-1
 Projekt PDV der Gesellschaft für Kernforschung mbH, Karlsruhe,
 Bericht KfK-PDV-75, 5 (1976)
- PDP73 Digital Equipment Corporation: Processor Handbook PDP-11, 1973
- PEL78 Pelz, K., Siller, F.:
 Katalogisierung von Mikroprozessoren
 Interne Notiz am Physikalischen Institut der Universität
 Erlangen-Nürnberg
- PEL81 Pelz, K.:
 Ein portabler Codegenerator. Dissertation am Lehrstuhl für
 Programmiersprachen, Erlangen, in Vorbereitung

- P0071 Poole, P.C.:
Hierarchical Abstract Machines.
Proceedings Software Engineering Conference, Culham, England,
HMSO, London, 1971
- P0073 Poole, P.C., Waite, W.M.:
Portability and Adaptability.
In Lecture Notes in Economics and Mathematical Systems
Advanced Course on Software Engineering, 1973
- PAR78 Parnas, D.L.
Designing Software for Ease of Extension and Contraction,
ICSE 1978
- PRE81 Prester, F.J.:
Ein portabler Treiber für graphische Geräte. Dissertation am
Lehrstuhl für Programmiersprachen, Erlangen, in Vorbereitung
- RAL66 Rolston, H., Wilf, H.S.:
Mathematical Methods for Digital Computers.
John Wiley & Sons, Inc., New York, London, Sydney
- ROE78 Rössler, R.:
ERLAN Handbuch
Interner Bericht am Physikalischen Institut Erlangen, 1978
- ROE78 Rössler, R.:
PEARL Betriebssystem für den Z80-Bus Subset
Projekt PDV der Gesellschaft für Kernforschung mbH, Karlsruhe,
Entwicklungsnotiz PDV-E119, 6 (1978)
- SCH75 Schneider, H.J.:
Compiler Aufbau und Arbeitsweise
Walter de Gruyter, Berlin, New York 1975
- SCH79 E. Schonberg et al.:
Automatic data structure selection in SETL
Proc. 6th Annual ACM Symposium Principles of Programming languages
San Antonio, Jan. 1979), p. 197-210
New York: Assoc. for Computer Mach.; 1979
- TRA80 Trautner, M.:
Codegenerator- und Systembeschreibung der SIEMENS 310-
PEARL-Implementation
Projekt PDV der Gesellschaft für Kernforschung mbH, Karlsruhe,
Entwicklungsnotiz PDV-E142, 6 (1980)
- TRA81 Trautner, M.:
Spezifikation des Codegenerators für PEARL auf einem KONTRON
System.
Interner Bericht am Physikalischen Institut, Erlangen, 1981
- WAI70 Waite, W.M.:
The Mobile Programming System Stage 2
In Communications of the ACM 13, p. 415, 7 (1970)
- WIR76 Wirth, N.:
Design and Implementation of MODULA. Bericht des Instituts
für Informatik, ETH Zürich, 19, 6 (1976)

- WUL70 Wulf, W.A., et al.:
 BLISS Reference Manual
 Bericht des Computer Science Department
 Carnegie-Mellon University Pittsburgh, Pennsylvania, 1970
- YA072 Yaoh, Chu:
 Introduction in the Computer Design Language
 Digit of Papers, COMPCON72,
 Sixth Annual IEEE Computer Society International Conference,
 San Francisco, 9 (1972)