Diplomarbeit

Karl Michael Göschka

Microcompiler Design Language

Eine Methode zur hardwaregesteuerten Generierung von Firmwarecompilern

Technische Universität Wien
Institut für Computertechnik
Gusshausstrasse 25–29
A–1040 Wien
Tel: (+43-222) 58801/3822

Siemens AG Österreich
Programm– und Systementwicklung 13
Geusaugasse 17
A–1030 Wien
Tel: (+43-222) 71600/301

DIPLOMARBEIT

Microcompiler Design Language

Eine Methode zur hardwaregesteuerten Generierung von Firmwarecompilern

ausgeführt am Institut für Computertechnik
der Technischen Universität Wien

unter der Anleitung von o.Univ.Prof. Dr. Richard Eier
und Univ.Ass. Dipl.Ing. Thomas Egger
als verantwortlich mitwirkendem Universitätsassistenten

durch

Karl Michael Göschka
Karl–Meißlstraße 7/17, 1200 Wien
Matr.Nr. 8625510

Wien, Oktober 1993 _____

MDL — *Microcompiler Design Language*

Eine Methode zur hardwaregesteuerten Generierung von Firmwarecompilern.

Kurzfassung

Offensichtlich erfordert die Entwicklung eines mikroprogrammierten Prozessors die *quasiparallele* Entwicklung von Hardware und Firmware, was insofern grundsätzlich problematisch ist, als die beiden einander sehr stark beeinflussen: Jede Hardwareänderung muß vom Compilerentwickler aufbereitet werden, damit der Firmwareentwickler auf der veränderten Hardware weiterarbeiten kann. Die lauffähigen Mikroprogramme wiederum werden vom Hardwareentwickler benötigt, um die Funktion der veränderten Hardware anhand der lauffähigen Firmware zu testen. Aus diesem Testlauf ergeben sich im allgemeinen wieder Hardwareänderungen und der Zyklus beginnt von neuem.

Das langsamste Glied in dieser Kette ist die Entwicklung des *Firmwarecompilers*: Die größten Hemmnisse für eine rasche Compilerentwicklung sind die ungeeignete Schnittstelle zwischen Hardwareentwickler und Compilerentwickler, die einen hohen *Kommunikationsaufwand* verursacht, sowie die mühevolle Suche nach hardwarebedingten *Konflikten* und die ineffiziente Formulierung der *semantischen Prüfungen*.

Im Rahmen der Diplomarbeit wurde die Sprache MDL entwickelt, um die Hardware so zu beschreiben, daß alle für die Compilerentwicklung notwendigen Informationen dargestellt werden können. Beschrieben werden die *Hardwareressourcen* des Prozessors, also die Netzstruktur aus einfachen Hardwareelementen, Registern und den dazugehörigen Verbindungen, sowie das Mikrobefehlsfeld als besondere Ressource. Außerdem gibt es die Möglichkeit der Gruppierung von Ressourcen. Darauf aufbauend wird die Verwendung der Hardwareressourcen in den *Mikrooperationen* mit Zeitangaben und unterschiedlichen Arten der Benutzung angegeben. Die Sprache ist dabei deskriptiv, effizient, einfach, flexibel und großzügig, sodaß sie im Rahmen eines Entwurfswerkzeuges Verwendung finden kann.

Zu dieser Sprache wurde ein *Analysator* implementiert, dessen Aufgabe es ist, eine MDL–Beschreibung einzulesen und daraus eine *Deskriptortabelle* aller prozessorspezifischen Bezeichner und eine *Bottom–Up–Konflikttabelle* aller konfliktbehafteten Ressourcen zu erzeugen. Darüberhinaus gibt es die Möglichkeit der *Top–Down–Konfliktanalyse*: Dabei werden zwei oder mehrere Mikrooperationen kombiniert und die entstehenden Konflikte aufgezeigt. Der Analysator beweist als *Prototyp* die Machbarkeit der geplanten Vorhaben. Dabei steckt ein erheblicher Teil des Aufwandes zur Realisierung des Analysators in der Fehlerstabilität und einem System von impliziten Annahmen für den Fall unvollständiger Eingaben, die gemeinsam den Analysator erst als *Entwurfswerkzeug* sinnvoll erscheinen lassen.

Die Vorteile dieses ersten Ansatzes sind also eine genormte Schnittstelle mit geringem Kommunikationsaufwand, die automatische Suche der Konflikte als Basis für einen Teil der semantischen Regeln sowie ein zusätzlicher *Hardwareentwicklungszyklus*: Dieser

ermöglicht die rasche Überprüfbarkeit des Entwurfes in Hinblick auf Konflikte durch den Hardwareentwickler selbst und bietet die Möglichkeit, zuerst auf einer *abstrakten Maschine* zu entwickeln.

Das Ziel der Weiterentwicklung dieser Methode ist die automatische Erzeugung der semantischen Prüfungen und der Codegenerierung für den *Mikroprogrammcompiler* sowie die Ermittlung der prozessorspezifischen *Sprachuntermenge* der Mikroprogrammsprache. Die Vorteile liegen jetzt zusätzlich in der Verlagerung der Arbeiten zur Erzeugung eines prozessorspezifischen Compilers in die Labors, in der schnelleren Verfügbarkeit der semantischen Prüfungen und letztendlich in der Einsparung von Personalkapazität für Anpassungen. Dabei muß die MDL–Beschreibung so erweitert werden, um auch die Zuordnung der Syntaxelemente der Mikroprogrammsprache sowohl zu den Mikrooperationen als auch zur Codegenerierung zu ermöglichen.

Zum Schluß werden noch die Entwicklung der Mikroprogrammierung und der *RISC–Prozessoren* diskutiert: Während sich die RISC–Architektur gegenüber der CISC–Architektur zweifellos durchgesetzt hat, bleibt dennoch die Zukunft der Mikroprogrammierung offen. Vor allem im Bereich der *Spezialprozessoren* und *Mikrocontroller* wird sie im Moment jedenfalls intensiv genutzt. In diesem Bereich bietet die Hardwarebeschreibung mittels MDL einen Ansatz, um den Firmwarecompiler weitgehend generieren zu können. Damit kann die Verflechtung zwischen Hardwareentwicklung und Firmwareentwicklung soweit aufgelöst werden, daß sie sich nicht bremsend auf den gesamten Entwicklungszyklus auswirkt.

Letztendlich verlagert sich mit der Einführung der RISC–Architektur die Problematik der Firmware in das Backend des *Hochsprachen–Compilers*: Durch die starke Hardwareabhängigkeit eines RISC–Compilers muß jetzt der Compilerentwickler Probleme bewältigen, die früher im Bereich der Firmware aufgelöst wurden und für die Software unsichtbar blieben. Auch dabei kann die der MDL zugrundeliegende Methode in angepaßter Form ebenfalls sinnvoll eingesetzt werden, um auch dem Entwickler von RISC–Prozessoren ein neues, mächtiges *Entwurfswerkzeug* in die Hand zu geben: Das Hauptziel des Entwurfes ist die möglichst rasche Ausführung üblicher Anwenderprogramme. Mit der Möglichkeit, den Hochsprachen–Compiler für eine abstrakte Maschine zu entwickeln, kann dieses Hauptziel des Entwurfsprozesses in jeder Phase der Entwicklung im Auge behalten und quantitativ überprüft werden, indem stets eine Auswahl typischer Anwenderprogramme mit dem Compiler für die in Entwicklung befindliche Maschine übersetzt wird.

Danksagung

Diese Seite soll nicht als bloße Pflichtübung verstanden werden, sondern sie ist vielmehr Ausdruck des Wunsches, jene Leistungen hervorzuheben, die neben den eigenen Anstrengungen am Gelingen der Arbeit maßgeblich beteiligt waren.

Besonderer Dank gebührt Herrn Johann Aschenbrenner, der nicht nur die Grundidee zu dieser Arbeit hatte, sondern auch während der Durchführung stets Zeit für interessante Diskussionen erübrigen konnte. Mit seiner tiefen Kenntnis der Materie und seinem umfangreichen Fachwissen hat er mittels konstruktiver Kritik stets für eine produktive Atmosphäre gesorgt, vor allem dann, wenn größere Schwierigkeiten zu überwinden waren.

Danken möchte ich auch Herrn Prof. Dr. Richard Eier und Herrn Dipl. Ing. Peter Tippold, die es ermöglicht haben, daß die Diplomarbeit in Zusammenarbeit zwischen dem Institut für Computertechnik der Technischen Universität Wien und der Abteilung PSE13 der Siemens AG Österreich durchgeführt werden konnte. Gerade für einen Techniker ist ja diese Zusammenarbeit zwischen Universität und Firma eine besonders wertvolle Erfahrung.

Ich bedanke mich auch bei Herrn Dipl. Ing. Thomas Egger, der als Hüter des universitären Niveaus die Betreuung der Diplomarbeit von Seiten des Institutes für Computertechnik innehatte. Insbesonders sein Verständnis für die Belange einer bei einer Firma durchgeführten Diplomarbeit hat für eine angenehme Atmosphäre gesorgt.

Bei Herrn Dr. Roland Schmid und Frau Hobiger als Repräsentanten des Siemens Studentenkreises möchte ich mich für die Betreuung während des zweiten Studienabschnittes bedanken, insbesonders für die kostenlose Bestellung zahlreicher Fachbücher sowie die vielen interessanten Informationsabende und nicht zuletzt für die Vermittlung des Themas der Diplomarbeit.

Mein Dank gilt auch Frau Erika Göschka und Fräulein Gabriele Grossmüller für das Opfer vieler Stunden Korrekturlesens. Meinen Eltern Albert und Erika Göschka schließlich gebührt der allergrößte Dank, da sie mein Studium überhaupt erst ermöglicht haben.

Inhaltsverzeichnis

Verzeichnisse

Kapitel 1

Spezifikation

1.1 Einleitung

Seit es Computer gibt, ist der Entwurf der Steuerung[1] des Datenpfades im Prozessor eine der schwierigsten, wenn nicht *die* schwierigste Aufgabe für die Entwickler. Nachdem Maurice Wilkes im Jahr 1949 den ersten universellen, programmierbaren Computer gebaut hatte und sich die Steuerung bereits damals als sehr komplex erwiesen hatte, suchte er nach einer besseren Methode, um die Steuerung zu entwerfen. Sein Lösungsansatz bestand darin, aus der Steuerungseinheit eine Art „Miniaturcomputer“ zu machen, wobei eine Tabelle die Steuerungssignale für den Datenpfad enthielt und eine weitere Tabelle den Kontrollfluß auf Mikroebene bestimmte. Wilkes nannte seine Erfindung *Mikroprogrammierung* und verwendete das Prefix „Mikro“ vor den üblichen Bezeichnungen, zum Beispiel Mikrobefehl, Mikrocode und Mikroprogramm[2]. Um Verwechslungen zu vermeiden, wird manchmal das Prefix „Makro“ verwendet, um die entsprechenden Bezeichnungen auf höherer Ebene darzustellen.

Mikrobefehle enthalten also die Steuerungssignale für den Datenpfad sowie die Möglichkeit bedingter Verzweigungen im Mikroprogramm. Dadurch wird die Steuerung des Datenpfades des Prozessors zu einer Programmieraufgabe: Das Mikroprogramm ist sozusagen nichts anderes als ein „Interpreter“ für den Befehlssatz des Prozessors. Es wird im Mikroprogrammspeicher[3] abgelegt und bestimmt als sogenannte *Firmware* wesentlich die Funktionalität des Prozessors. Als direkte Folge daraus kann der Befehlssatz eines Prozessors ohne Veränderungen an der Hardware durch einfachen Austausch des Inhaltes des Mikroprogrammspeichers variiert werden. Dies kann in der Praxis zweierlei Sinn haben: Erstens kann eine Familie von Prozessoren, die einander nur in der Firmware

[1] In der englischen Literatur als *control* bezeichnet.

[2] Auf englisch als *microinstruction*, *microcode* und *microprogram* bezeichnet. Bei der Verwendung der durchwegs englischen Fachausdrücke wurde folgendermaßen vorgegangen: Soweit vertretbar, wurden entsprechende deutsche Ausdrücke verwendet, also etwa „Mikrocode“ für *microcode*. Wenn jedoch die Verwendung deutscher Ausdrücke eine zu starke Verfremdung ergeben hätte (z.B. „Datensammelschiene“ für *bus*) oder unüblich gewesen wäre, wurden die englischen Fachausdrücke beibehalten und direkt in den deutschen Text integriert. Typische Beispiele sind: Computer, Compiler, Code und Controller.

[3] *control store*

unterscheiden, sehr günstig hergestellt werden. Zweitens kann es in bestimmten Fällen sinnvoll sein, dem Endbenutzer die Beeinflussung der Firmware zu gestatten. Der zweite Fall ist aber mit Vorsicht zu genießen und hauptsächlich im Bereich der Spezialprozessoren interessant. Die Veränderung der Firmware eines Prozessors, der als CPU[4] vorgesehen ist, wird hingegen eher problematisch sein.

Insgesamt entsteht durch die Verwendung der Mikroprogrammierung eine Hierarchie: Die kleinste Einheit der Software ist der Maschinenbefehl. Dieser bewirkt den Aufruf eines Mikroprogrammes, welches seinerseits aus einzelnen *Mikrobefehlen* oder Statementverbänden besteht. Ein Mikrobefehl ist eine Bitfolge, deren Länge sich aus der Breite des Mikroprogrammspeichers ergibt. Typische Werte sind etwa 32 bis über 200 Bit. Jedes Bit im Mikrobefehl entspricht einem Steuersignal für die Hardware, meist steuern mehrere Bits gemeinsam den Ablauf einer festverdrahteten Hardwarefunktion. Solcherart zusammengehörige Gruppen von Bits sind im Mikrobefehl im allgemeinen nebeneinander angeordnet und werden als Feld bezeichnet. Somit besteht jeder Mikrobefehl aus mehreren Feldern, auch *Mikrooperationen* oder Statements genannt. Solche Mikrooperationen sind etwa Registertransfers, arithmetisch-logische Verknüpfungen von Registern, Zugriffe auf den Speicher sowie die Steuerung des Mikroprogrammablaufes selbst, zum Beispiel durch bedingte Verzweigungen.

Der Mikroprogrammspeicher ist als wesentlichster Hardwarebestandteil der mikroprogrammierten Steuerung der Hauptangriffspunkt für Techniken zur Reduktion der Hardwarekosten. Diese Techniken reduzieren entweder die Breite des Mikroprogrammspeichers, die Anzahl der Mikrobefehle oder beides. Zwei Techniken zur Reduktion der Mikrobefehlsbreite werden kurz vorgestellt:

Encoding: Darunter versteht man eine Methode, bei der die Bits im Mikrobefehl nicht direkt die Steuersignale darstellen, sondern noch von festverdrahteten Hardwarefunktionen decodiert werden müssen. Ein einfaches Beispiel: Wenn stets maximal nur eine von acht Steuerleitungen pro Mikrobefehl gesetzt sein kann, genügt ein drei Bit großes Feld zur Codierung. Allerdings muß ein Decoder nachgeschaltet werden, um aus dem Mikrobefehlsfeld die Steuersignale zu erzeugen.

Vertical microcode: Wenn man darauf verzichtet, in jedem Mikrobefehl immer alle Mikrooperationen ansprechen zu können, und stattdessen nur bestimmte Gruppen von Mikrooperationen zuläßt, kann man für das Mikrobefehlsfeld unterschiedliche Formate vorsehen, die durch ein kurzes Formatfeld unterschieden werden. Im Gegensatz dazu steht die Methode des *horizontal microcode*, die zwar einen breiteren Mikroprogrammspeicher erfordert, dafür aber kürzere Ausführungszeiten erreicht.

Wie überall konkurrieren auch hier Kosten mit Geschwindigkeit, sodaß es eine der wesentlichsten Entscheidungen ist, wann die Kosten optimiert werden sollen und wann die Geschwindigkeit: Jene Maschinenbefehle, die den größten Teil der Ausführungszeit beanspruchen, werden auf Geschwindigkeit optimiert, die übrigen auf Kosten.

[4] *Central Processing Unit*

Wie wird die Firmware nun konkret erstellt? Die direkte Programmierung mittels Bitmustern ist bereits bei sehr kurzen Mikroprogrammen von nahezu unvertretbarem Aufwand. Daher wird dem Firmwareprogrammierer eine *Mikroprogrammsprache* zur Verfügung gestellt, die etwa Assemblercharakter besitzt und für jeden spezifischen Prozessor als Untermenge aus einem vorgegebenen Sprachrahmen definiert wird. Die Auswahl dieser Sprachuntermenge und die Entwicklung des zugehörigen *Mikroprogrammcompilers* sind Aufgaben des Compilerentwicklers. Er legt also die Zuordnung der Sprachuntermenge zu den Bitmustern fest und muß dabei im Compiler vor allem die notwendigen semantischen Einschränkungen implementieren. Die dazu notwendigen Informationen sind hardwareabhängig und werden dem Compilerentwickler vom Hardwareentwickler zur Verfügung gestellt. Der Hardwareentwickler benötigt aber seinerseits wiederum die Firmwareprogramme zum Testen seiner Hardware. Der dadurch entstehende *Entwicklungszyklus* von Hardware, Firmware und Compiler wird im nächsten Abschnitt genauer untersucht, wobei die Problematik verdeutlicht wird. Im darauffolgenden Abschnitt wird dann der Lösungsansatz vorgestellt und im letzten Abschnitt dieses Kapitels daraus die Anforderungen für die MDL abgeleitet.

Die Entwicklung der Mikroprogrammierung, ihre aktuelle Bedeutung und ihre Zukunft werden im letzten, zusammenfassenden Kapitel der Diplomarbeit analysiert. Nähere und weiterführende Informationen sind in [20] nachzulesen.

1.2 Darstellung der Problematik

Offensichtlich erfordert die Entwicklung eines Prozessors die quasiparallele Entwicklung von Hardware und Firmware, was insofern grundsätzlich problematisch ist, als die beiden einander sehr stark beeinflussen. Abbildung 1.1 zeigt den gesamten Entwicklungsprozeß vor Einführung des Hardwaresimulators.

Der *Hardwareentwicklungszyklus*, links im Bild, ist geprägt von der Arbeit des Hardwareentwicklers an seinem Laboraufbau: Die Schaltung wird real aufgebaut, anschließend wird die Reaktion auf verschiedene Eingangssignale mit geeigneten Meßgeräten aufgenommen. Der Hardwareentwicklungszyklus wird hier bewußt auf den für die Aufgabenstellung der Diplomarbeit wesentlichen Teil reduziert, das ist das Testen der Hardware mit einem lauffähigen Firmwareprogramm. Die anderen Teile des Hardwareentwicklungszyklus, wie etwa das Entwerfen der Schaltung in Hinblick auf Integration, Partitionierung und Fertigung, wurden bewußt weggelassen, da sie vom Hardwareentwickler alleine durchgeführt werden und daher hier nicht Gegenstand der Betrachtung sind.

Als *Firmwareentwicklungszyklus*, rechts im Bild, wird die Arbeit des Firmwareentwicklers am Compiler bezeichnet: Die eingegebenen Mikrobefehle werden übersetzt und entsprechend den ausgegebenen Meldungen solange verbessert, bis eine lauffähige Version zustande kommt.

Es gibt nun zwei Verbindungen zwischen dem Hardwareentwicklungszyklus und dem Firmwareentwicklungszyklus: Die eine besteht in der Umsetzung des vom Compiler er-

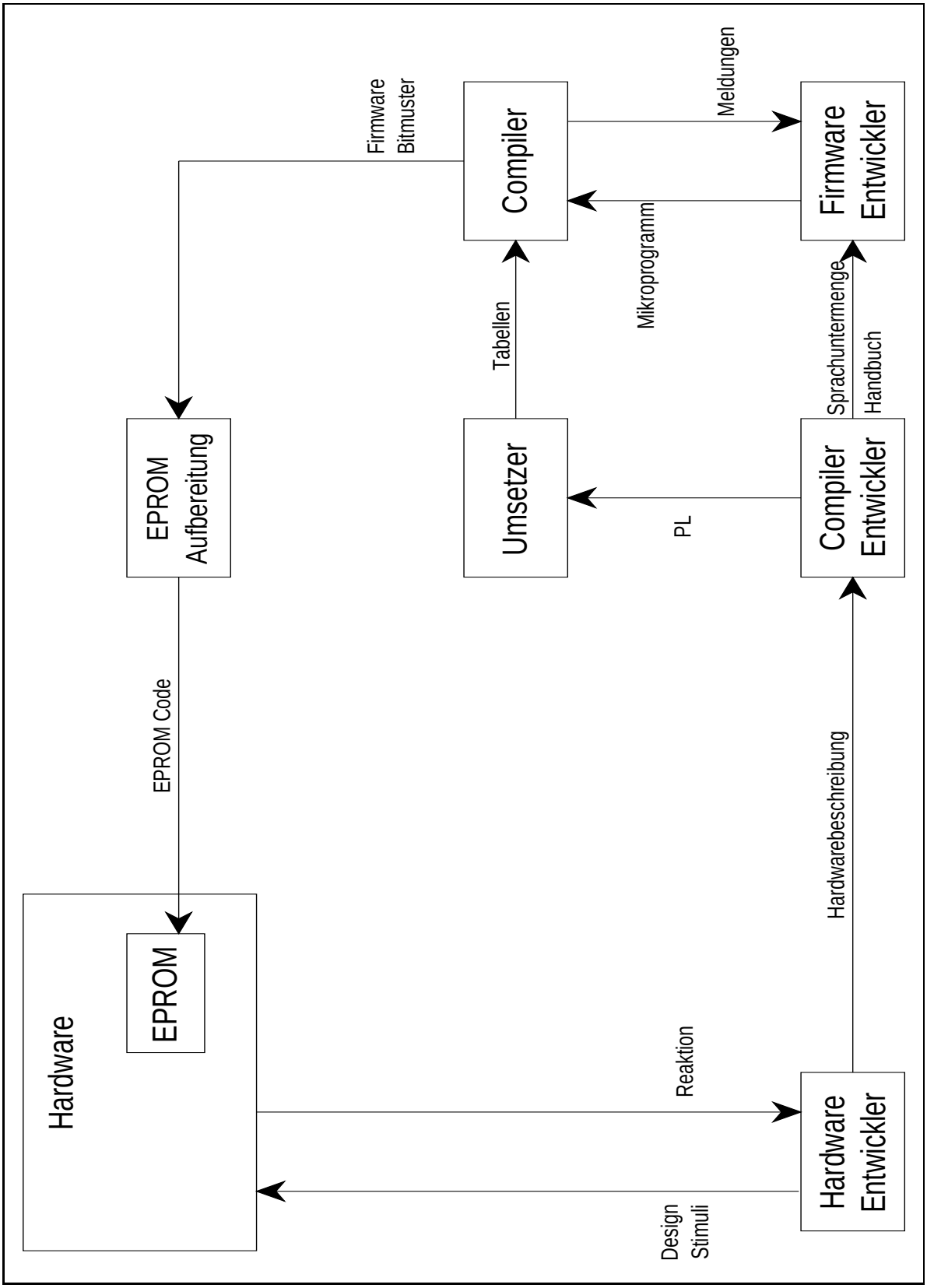


Abbildung 1.1: Der ursprüngliche Entwicklungszyklus.

zeugten Bitmusters in einen geeigneten Code, der schließlich in ein EPROM[5] einprogrammiert wird. Dieses wird im Laboraufbau vom Hardwareentwickler verwendet, um seine Probeschaltung mit lauffähigen Mikroprogrammen zu testen.

Die zweite Verbindung ist jene über den *Compilerentwickler*, im Bild in der Mitte unten: Dieser erhält vom Hardwareentwickler eine nicht formalisierte Hardwarebeschreibung, zum Beispiel anhand eines Blockschaltbildes, sowie eine Erläuterung, die gegebenenfalls durch Rückfragen ergänzt werden muß. Als Feedback erhält der Hardwareentwickler vom Compilerentwickler Information, zum Beispiel über mögliche hardwarebedingte Konflikte, wobei es im wesentlichen um die Frage geht, welche Mikrooperationen sich in einem Mikrobefehl vereinen lassen. Da dieses Feedback aber nur gering ausgeprägt ist, wurde es im Bild weggelassen. Der Compilerentwickler trifft dann die Auswahl der Sprachelemente aus dem vorgegebenen Sprachrahmen und stellt sie dem Firmwareprogrammierer zur Verfügung. Außerdem formuliert der Compilerentwickler die notwendigen semantischen Einschränkungen in Form einer Problemsprache PL[6], welche von einem Umsetzer zu Tabellen aufbereitet wird. Diese Tabellen steuern dann die semantischen Prüfungen im Firmwarecompiler.

Damit ist der Kreis geschlossen: Jede Hardwareänderung muß vom Compilerentwickler aufbereitet werden, damit der Firmwareentwickler auf der veränderten Hardware weiterarbeiten kann. Die lauffähigen Mikroprogramme wiederum werden vom Hardwareentwickler benötigt, um die Funktion der veränderten Hardware anhand der lauffähigen Firmware zu testen. Aus diesem Testlauf ergeben sich im allgemeinen wieder Hardwareänderungen und der Zyklus beginnt von neuem. Es ist daher leicht einzusehen, daß die Firmwareentwicklung immer hinter der Hardwareentwicklung einherhinkt. Da die Entwicklung der Hardware insgesamt ohne Simulator relativ langsam war, hat es sich aber nicht negativ ausgewirkt, daß eine Facette der Hardwareentwicklung, nämlich das Testen mit der Firmware, relativ langsam war.

Das hat sich aber schlagartig geändert, als die Entwicklung digitaler Hardware fast zur Gänze auf *Simulator* umgestellt wurde: Nun wird nicht mehr gelötet und gemessen, sondern das Design der Schaltung in ein Programm eingegeben und alles weitere nur noch simuliert. Lediglich solche Hardwareteile, deren Beschreibung zu kompliziert ist, werden weiterhin hardwaremäßig aufgebaut und an den Simulator angeschlossen. Mit zunehmender Rechenleistung der Simulatoren verliert dieser Sachverhalt aber an Bedeutung. Abbildung 1.2 zeigt den veränderten Entwicklungszyklus. Als Fazit wurden die anderen Hardwareentwicklungszyklen, wie etwa Integration, Layout oder Partitionierung, mittels CAD[7] und CAM[8] so stark beschleunigt, daß der Zyklus mit dem Firmwaretest in Relation dazu plötzlich viel zu langsam ist. Daran konnte auch die Einführung eines Simulatorteiles, der speziell dem Testen der simulierten Schaltung mit fertigen Firmwarebitmustern gewidmet ist, nichts ändern: Die Compilerentwicklung ist einfach zu langsam.

Die Ursache dafür ist zunächst im hohen *Kommunikationsaufwand* zwischen Hardwareentwickler und Firmwareentwickler begründet: Gerade diese zwischenmenschliche

[5] *Eraseable Programable Read Only Memory*
[6] *Problem Language*
[7] *Computer Aided Design*
[8] *Computer Aided Manufacturing*

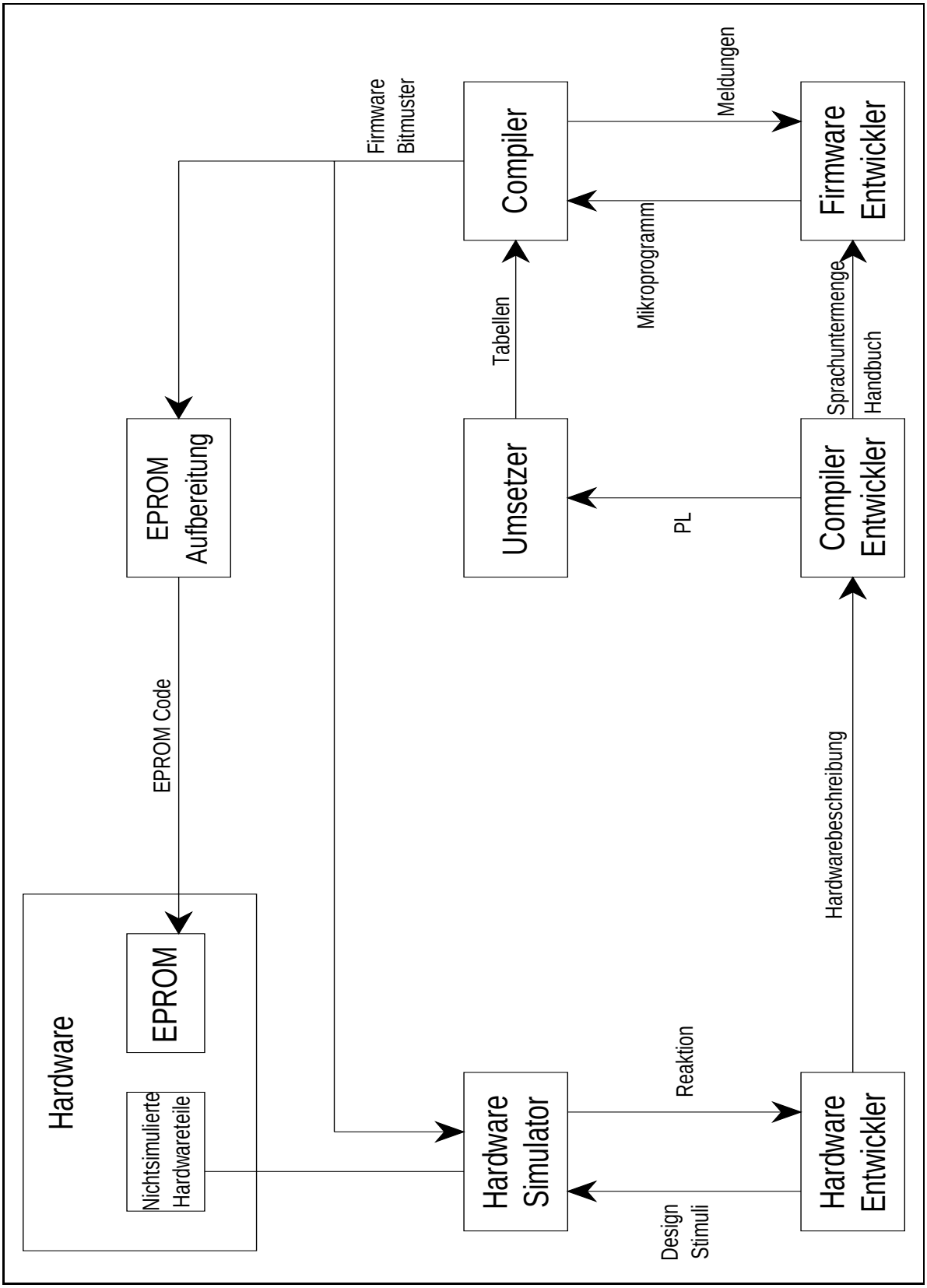


Abbildung 1.2: Der Entwicklungszyklus nach Einführung des Hardwaresimulators.

„Schnittstelle“ ist ja während des Hardwareentwicklungsprozesses laufend Änderungen unterworfen. Da sie aber keinen formalen Charakter besitzt, also in keiner Weise genormt ist, ist sie als Basis denkbar ungeeignet. Der zweite Bremsschuh ist die Entwicklung des Compilers selbst: Eigentlich sollte möglichst rasch nach dem Ende des Hardwareentwurfes ein funktionierender *Mikroprogrammcompiler* zur Verfügung stehen, um die Firmware entwickeln zu können, mit der die Hardware getestet werden soll. Dies ist aber nicht möglich, da die Anpassung des Compilers sehr mühsam und fehleranfällig ist. Zum besseren Verständnis wird nun kurz der Aufbau des Compilers in seiner bisherigen Form erläutert, die Beschreibung bezieht sich auf Abbildung 1.3.

Da der Compiler *hardwareabhängig* ist, muß er an jede Hardwareänderung angepaßt werden. Dabei wird aber nicht das Programm selbst angepaßt, sondern die Tabellen, die den Ablauf des Übersetzungsvorganges steuern: Jeder Mikrobefehl wird zunächst lexikalisch und syntaktisch überprüft. Da der Sprachrahmen und somit auch die Syntax fix vorgegeben sind, sind die Syntaxtabellen kaum hardwareabhängig, es sei denn, der Sprachrahmen muß an die Erfordernisse einer speziellen Prozessorfamilie angepaßt werden. Der semantische Vorlauf prüft die verwendeten variablen Bezeichner für Register und setzt sie gegebenenfalls um. Die hiezu verwendeten Deskriptortabellen enthalten alle für einen bestimmten Prozessor erlaubten Bezeichner und ihre Bedeutung und sind daher bereits stark hardwareabhängig.

Das größte Problem aber bereitet die *semantische Analyse* selbst: Deren Eingabe ist eine linearisierte und attributierte Form des ursprünglichen Mikrobefehls, die Ausgabe der Bitcode. Mittels der Problemsprachenbeschreibung PL werden die semantischen Regeln und Prüfungen unter Bezugnahme auf die linearisierten Mikrobefehle beschrieben. Aus der PL–Beschreibung erzeugt ein Umsetzer dann die PL–Tabellen, welche die semantischen Prüfungen im Zuge des Firmwarecompilers steuern. Dabei gibt es nun zwei Probleme:

Erstens müssen *Konflikte*, die unmittelbar die Hardwarestruktur betreffen, vom Hardwareentwickler selbst herausgefunden und dem Compilerentwickler als notwendige Einschränkungen mitgeteilt werden. Der Compilerentwickler muß sich dann darüberhinausgehende Konflikte erarbeiten, die zum Beispiel dadurch entstehen können, daß manche Sprachelemente aufgrund der Hardwarestruktur oder der Bedeutung der Bits im Mikrobefehl keine Bitcodeentsprechung besitzen. Dabei fällt auf, daß alle Konflikte von Entwicklern erkannt werden müssen. Dadurch wird die experimentelle Entwicklungsarbeit durch das ständige „händische“ Suchen nach Konflikten stark gebremst. Alle Konflikte gemeinsam werden dann mittels Problemsprache implementiert.

Diese PL–Beschreibung nun ist der zweite Grund für den hohen Aufwand der Compileranpassung an eine Hardwareänderung: Durch die hohe *Redundanz*, die der relativ simplen PL–Beschreibung innewohnt, ist das Erstellen einer PL–Beschreibung nicht nur gedanklich sehr aufwendig, sondern auch im höchsten Maße fehleranfällig. Hier liegt also einer der wichtigsten Ansatzpunkte für eine Verbesserung.

Das Hauptprogramm des Compilers besteht im wesentlichen aus einer Schleife über alle Mikrobefehle, wobei Aufgaben der Blockverwaltung und der Symboltabellenverwaltung wahrgenommen werden. Das vorläufige Endergebnis ist ein Rumpfmodul, welches durch

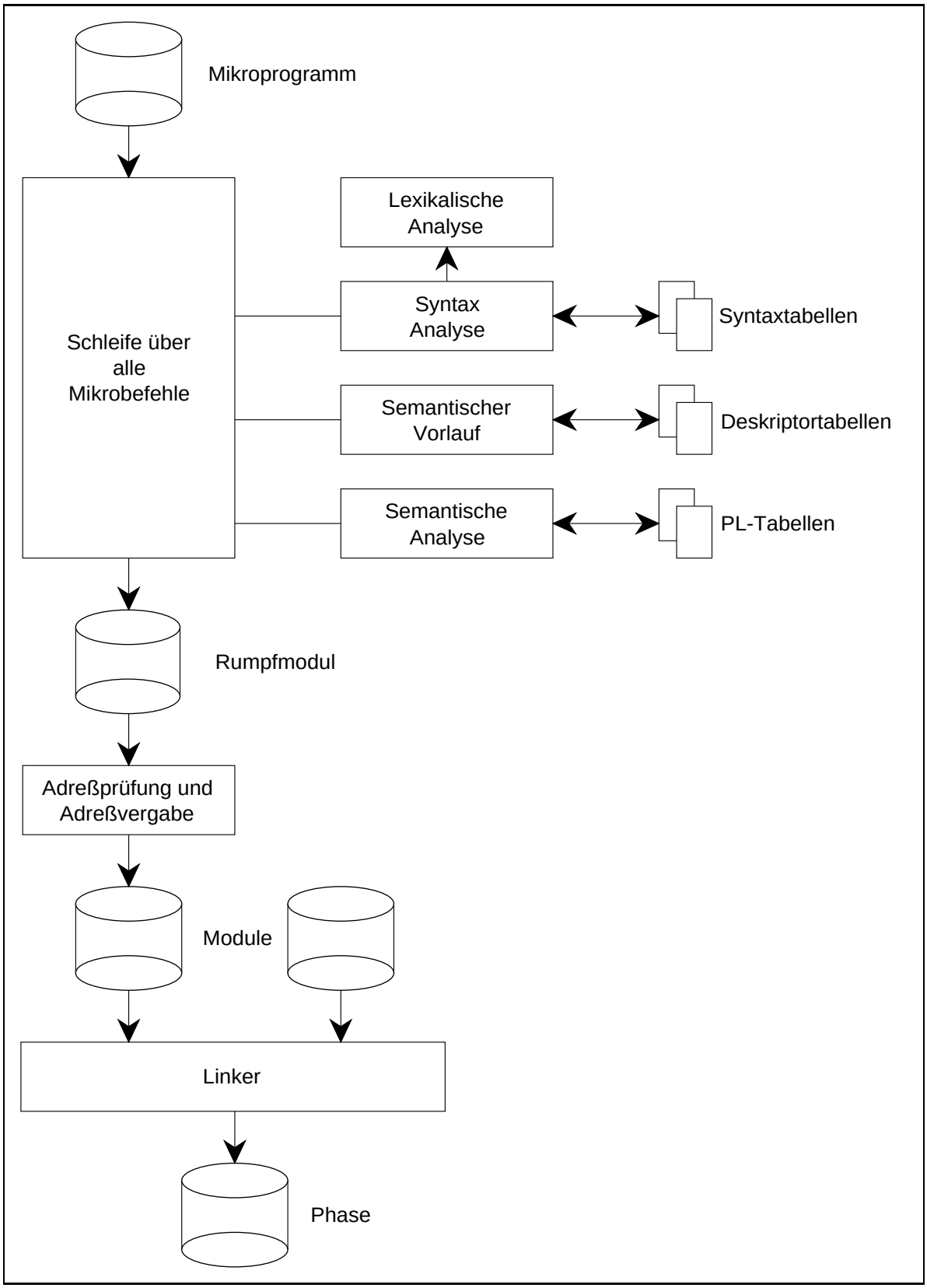


Abbildung 1.3: Der bisherige Aufbau des Firmwarecompilers.

Adreßprüfung und Adreßvergabe in ein fertiges Modul umgeformt wird. Diese Adreßbestimmungen sind ebenfalls hardwareabhängig. Mehrere Module werden schließlich vom Linker zu einer Phase gebunden, die dann schließlich dem Mikroprogrammspeicher zugeführt wird.

Zusammenfassend kann also festgestellt werden: Die größten Hemmnisse für eine rasche Compilerentwicklung sind die ungeeignete Schnittstelle zwischen Hardwareentwickler und Compilerentwickler, sowie die mühevolle Suche nach Konflikten und die Formulierung der semantischen Prüfungen mittels PL–Beschreibung. Wie nun diese Hemmnisse durch die MDL beseitigt wurden, wird im nächsten Abschnitt erläutert.

1.3 Lösungsansatz

Als erster Schritt wurde im Rahmen der Diplomarbeit die *Schnittstelle* zwischen Hardwareentwickler und Compilerentwickler formal klar definiert. Den veränderten Entwicklungszyklus zeigt Abbildung 1.4. Die Sprache MDL wurde entwickelt, um die Hardware so zu beschreiben, daß alle für die Compilerentwicklung notwendigen Informationen dargestellt werden können. Daher müssen durch die Beschreibung alle möglichen Konflikte erkannt werden können, die sowohl bei der Hardwareentwicklung, als auch bei der Übersetzung der Mikroprogramme selbst entstehen können. Diese Aufgabe erfüllt der Analysator, der automatisch entsprechende Konfliktabellen erstellt: Der Hardwareentwickler muß die Hardwarebeschreibung mittels MDL durchführen und an den Analysator weiterreichen. Dieser liefert die MDL–Beschreibung, ergänzt durch die Tabellen von erkannten möglichen Konfliktfällen, an den Compilerentwickler weiter. Dieser definiert nun, so wie auch bisher, die prozessorspezifische Sprachuntermenge und erstellt die PL–Beschreibung. Andererseits erhält der Hardwareentwickler sein Feedback jetzt nicht mehr vom Compilerentwickler, sondern vom Analysator, und zwar ebenfalls in Form von Konfliktabellen. Dabei handelt es sich um Diagnoselisten über die gegenseitige Blockierung einzelner Mikrooperationen mit Angabe der Blockierungsursache. Die dabei verwendete MDL0 ist jene Untermenge der MDL, die der Beschreibung der unmittelbaren Hardwareinformationen dient.

Daraus ergeben sich drei Konsequenzen: Erstens wird der *Kommunikationsaufwand* auf ein Minimum reduziert, indem sichergestellt wird, daß exakt die benötigte Information in klarer Darstellung übermittelt wird, nicht mehr und nicht weniger. Als zweite Konsequenz wird der Compilerentwickler von den *Konfliktabellen* des Analysators unterstützt, sodaß er nicht mehr alle Konflikte „händisch“ suchen muß, was ihm einen Teil seiner Arbeit abnimmt: Er konsultiert nunmehr die Konfliktabelle und verwendet sie als Basis für einen Teil der semantischen Regeln. Deren Implementierung in Form der PL–Beschreibung ist aber nach wie vor reformbedürftig, da ein Mensch für diese Aufgabe eigentlich zu schade ist.

Als dritte Konsequenz erhält der Hardwareentwickler einen zusätzlichen *Entwicklungszyklus*: Bisher war die Reaktion auf eine Hardwareänderung viel zu langsam, um spielerisch experimentieren zu können. Außerdem mußten bisher Schaltung und Bitcode sowie die Verwendung der Hardwareelemente in den Mikrooperationen bereits genau feststehen,

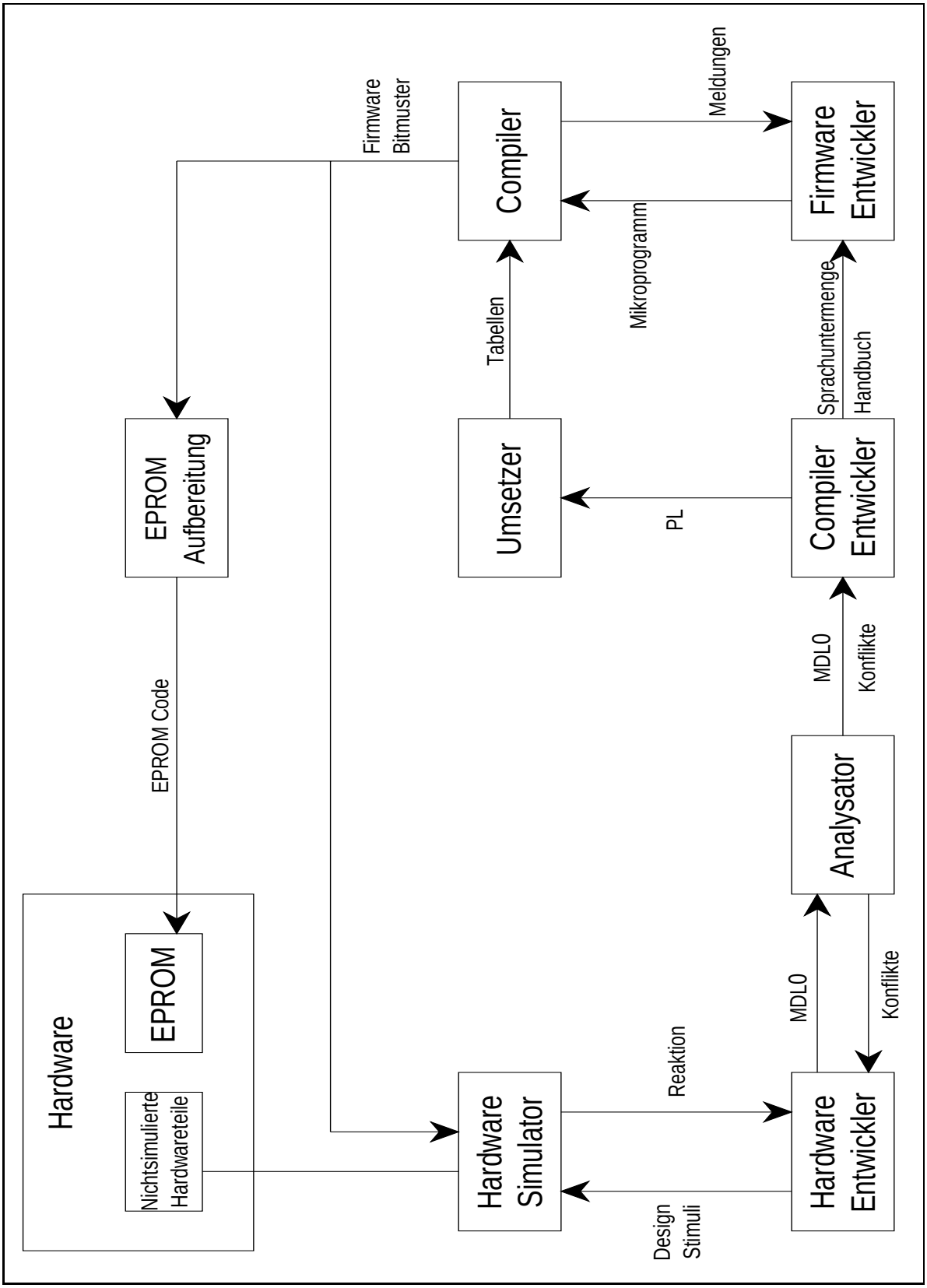


Abbildung 1.4: Der Entwicklungszyklus mit MDL.

da ja in weiterer Folge daraus der Compilerentwickler die notwendigen semantischen Einschränkungen für den Firmwarecompiler hergeleitet hat. Nun aber kann der Hardwareentwickler relativ schnell verschiedene Ansätze und Varianten durchspielen: Die Konflikte werden stets automatisch und damit rasch und unkompliziert erzeugt. Wenn die Funktion des Analysators sichergestellt ist, wird auch die Fehlerquote geringer sein, es werden kaum noch Fehler übersehen werden. Darüberhinaus wird es dem Hardwareentwickler ermöglicht zu experimentieren, sogar ohne sich in Bezug auf Bitcodemuster, Bitfelder oder Bezeichner allzusehr festzulegen: Da viele dieser Angaben erst bei der Compilergenerierung benötigt werden, beeinträchtigt ihr Fehlen nicht die Funktion des Analysators, insbesonders die Konflikterkennung. Aber auch der Analysator selbst muß großzügig und flexibel genug sein, um bei fehlenden Vorgaben sinnvolle Annahmen zu treffen.

Vor allem diese Möglichkeit, auf einer *abstrakten Maschine* mit der Entwicklung zu beginnen, bietet dem Hardwareentwickler einen zusätzlichen Freiheitsgrad bei der Wahl von Designalternativen. Ein Beispiel: Er kann zunächst mit breiten, horizontalen Bitmustern beginnen, die zwar schnell, aber auch teuer sind. Dann kann er die Auswirkung der Vertikalisierung von Mikrobefehlen unmittelbar studieren und so einen Kompromiß aus Kosten und Geschwindigkeit finden. Erst dann muß überhaupt erst die Befehlsbreite festgelegt werden und erst zum Schluß der Bitcode selbst. Auch mit der Festlegung der Register kann sich der Hardwareentwickler bis zuletzt Zeit lassen. Erst wenn er eine Variante gefunden hat, die ihm wirklich brauchbar erscheint, muß er den Compilerentwickler bemühen. Diese Möglichkeit, eine MDL–Beschreibung für eine abstrakte Maschine zu entwerfen, setzt also den Kommunikationsaufwand zusätzlich weiter herab, da viele Konflikte bereits vom Hardwareentwickler erkannt und entsprechend behandelt werden können. Daher muß nicht jedesmal der Compilerentwickler bemüht werden, der vergleichsweise immer noch relativ langsam ist, da er nach wie vor die umständliche PL–Beschreibung der semantischen Regeln erstellen muß.

Die Vorteile dieses ersten Ansatzes sind also eine genormte Schnittstelle, die rasche Überprüfbarkeit des Entwurfes im Hinblick auf Konflikte durch den Hardwareentwickler selbst, die Möglichkeit, zuerst auf einer abstrakten Maschine zu entwickeln, und die automatische Suche der Konflikte als Basis für einen Teil der semantischen Regeln. Der größte Nachteil ist die umständliche Implementierung der semantischen Regeln mittels PL–Beschreibung.

Daher ist als zweiter Schritt geplant, daß der Compilerentwickler die erhaltene MDL0–Beschreibung dahingehend zu einer MDL1–Beschreibung erweitert, daß daraus der Firmwarecompiler *generiert* werden kann. Das bedeutet, daß die MDL1–Beschreibung vor allem die semantischen Regeln enthalten muß und die Auswahl der prozessorspezifischen Sprachelemente aus dem vorgegebenen Sprachrahmen sowie die Syntax/Bitcode–Zuordnung ermöglichen muß. Diese Ergänzung der erhaltenen MDL0–Beschreibung zur MDL1–Beschreibung ersetzt dann die Erstellung der PL–Beschreibung. Dieser Schritt übersteigt zwar den Rahmen der Diplomarbeit, er wurde aber bereits konzeptionell bei der Erstellung der MDL berücksichtigt. Abbildung 1.5 zeigt den weiter beschleunigten Entwicklungszyklus.

Wodurch wird dabei die Verbesserung erreicht? Die Redundanz der PL–Beschreibung entsteht nicht durch Überflüssiges, sondern durch Ähnliches. Das bedeutet, daß etwa

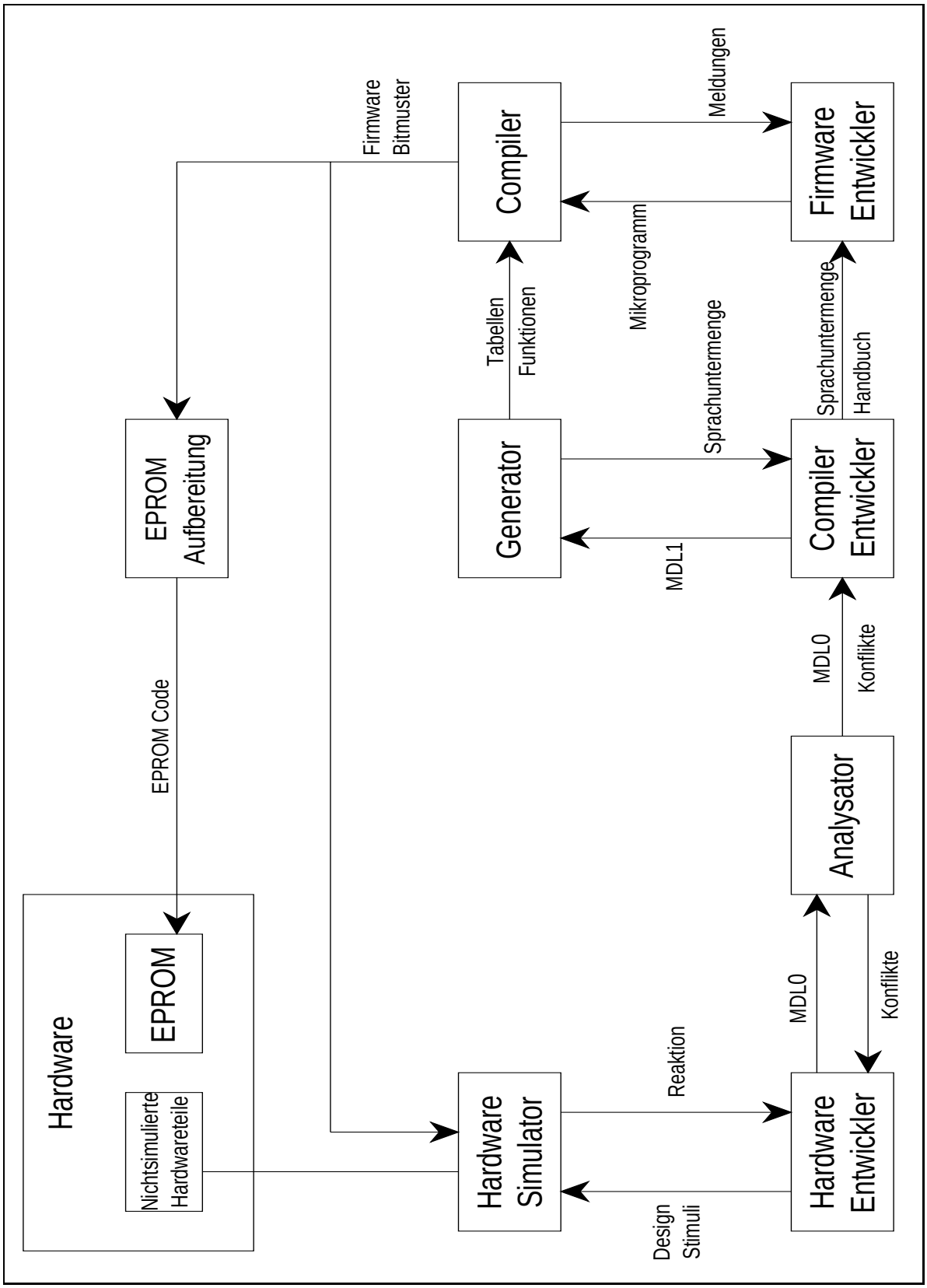


Abbildung 1.5: Der Entwicklungszyklus mit erweiterter MDL und Compilergenerator.

80% der in der Praxis vorkommenden Fälle so geartet sind, daß eine PL–Beschreibung im höchsten Maße redundant ist. Die MDL1–Beschreibung hingegen muß spezielle Mechanismen vorsehen, um diese in der Praxis häufiger auftretenden Fälle auf eine vereinfachte Weise zu beschreiben. Effizienz einer Sprache bedeutet ja nichts anderes, als die Sprachmittel für jene Elemente kurz zu halten, die am häufigsten beschrieben werden. Daher kann Effizienz aber auch nur für einen gewissen Anwendungsbereich garantiert werden. Da somit die MDL1–Beschreibung auf einer höheren Abstraktionsebene als die PL–Beschreibung stattfindet und außerdem kaum redundant ist, wird die Anpasung an eine Hardwareänderung drastisch vereinfacht.

Das Ziel des zweiten Ansatzes ist die automatische Erzeugung der semantischen Prüfungen und der Codegenerierung für den Mikroprogrammcompiler. Die Vorteile liegen jetzt zusätzlich in der Verlagerung der Arbeiten zur Erzeugung eines prozessorspezifischen Compilers in die Labors, in der schnelleren Verfügbarkeit der semantischen Prüfungen und letztendlich in der Einsparung von Personalkapazität für Anpassungen: Selbst wenn sich zusätzliche semantische Einschränkungen als notwendig erweisen sollten, ist dennoch ein neuer Compiler wieder in kürzester Zeit verfügbar. Dies ist unter anderem dann denkbar, wenn sich entweder durch Simulation oder aber durch Testen des Prototyps Einschränkungen ergeben, die auf Realisierungsbedingungen zurückgeführt werden können.

Der *Generator* ermittelt die Sprachuntermenge und generiert die hardwareabhängigen Tabellen und Unterprogrammfunktionen für den Firmwarecompiler. Der Compilerentwickler wird deutlich entlastet, er steuert den Generierungsprozeß und überwacht das Ergebnis. Sein Aufgabenbereich hat sich also drastisch verändert, weg von der „Knochenarbeit“ hin zum kreativen Design. Damit ist die Compilerentwicklung genauso rasch geworden wie die anderen Teile des gesamten Entwicklungszyklus. Da jede Kette nur so stark ist wie ihr schwächstes Glied, steht erst jetzt ein effizienter Prozessorentwicklungszyklus zur Verfügung.

Ein Maß für die Effizienz einer Sprache ist die enthaltene Redundanz. Daher stellt der Übergang von der PL–Beschreibung zur MDL1–Beschreibung eine wesentliche Verbesserung der Effizienz des Entwicklungszyklus dar. Dennoch ist noch ein gewisses Maß an Redundanz in der MDL0–Beschreibung enthalten: Der Hardwaresimulator, auf dem der Hardwareentwickler arbeitet, enthält bereits intern eine Prozessorbeschreibung in irgendeiner Hardwarebeschreibungssprache HDL[9]. Diese HDL enthält aber bereits explizit und implizit die Masse der Informationen, die in der MDL–Beschreibung enthalten sind. Daher wäre es noch sinnvoll, mittels eines *Filters* die MDL0–Beschreibung aus der simulatorinternen HDL–Beschreibung zu generieren. Mit der damit verbundenen Redundanzverminderung wäre eine weitere Effizienzsteigerung verbunden, die im Endeffekt zu einer zusätzlichen Beschleunigung des Entwicklungszyklus führt. Abbildung 1.6 zeigt den aus der Sicht der Firmwareerstellung ideal beschleunigten Entwicklungszyklus. Der Hardwareentwickler kann die MDL zunächst für eine abstrakte Maschine selbst entwerfen, wenn dann später bereits der Hardwaresimulator in Verwendung ist, wird die MDL–Beschreibung einfach extrahiert. Gegebenenfalls kann der Hardwareentwickler die extrahierte Beschreibung dann variieren, um verschiedene Möglichkeiten zu vergleichen. Er muß aber nicht mehr selbst dafür Sorge tragen, daß die MDL–Beschreibung mit der

[9] *Hardware Description Language*

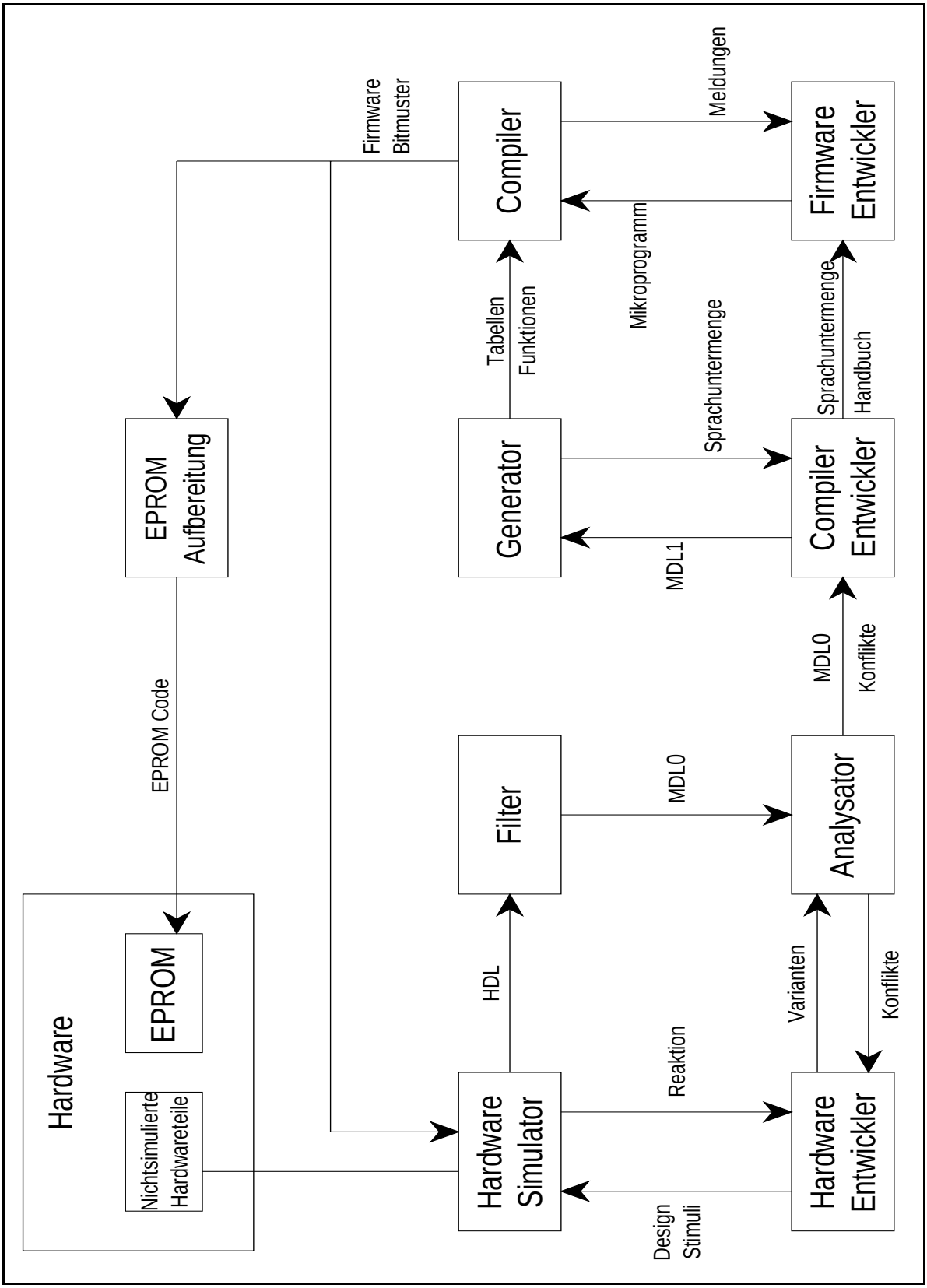


Abbildung 1.6: Der Entwicklungszyklus mit teilweise generierter MDL.

Simulator–HDL übereinstimmt. Genau diese Redundanz ist mit Einführung des Filters beseitigt.

Nun hätte man gleich eine bestehende Hardwarebeschreibungssprache wie etwa VHDL [10], ARCHI [36] oder ISPS [11] heranziehen können, um sowohl die Schnittstelle zu formalisieren als auch den Compiler zu generieren. Diese Vorgangsweise hat aber einige Nachteile. Vor allem ist es sehr wahrscheinlich, daß diese bestehende HDL hätte erweitert werden müssen, womit der Vorteil der Verwendung einer bestehenden Sprache wieder verloren gewesen wäre. Zweitens ist die HDL–Schnittstelle zum Simulator ganz allgemein eher kurzfristig und mannigfaltig veränderlich, sodaß eine enge Bindung an eine solche HDL eine unnötige Einschränkung bewirkt hätte. Drittens schließlich dient die MDL als ruhige und überprüfbare Basis für die Firmwarecompilererstellung und enthält exakt jene Information, die dafür notwendig und hinreichend ist. Das hat den Vorteil, daß im Zuge der Entwicklung der MDL deutlich geworden ist, welche Information nun tatsächlich erforderlich ist. Aufbauend auf diese MDL können nun Filter sozusagen als Adapter für bestehende Hardwarebeschreibungssprachen verwendet werden, wobei vermutlich nur die MDL0–Beschreibung generiert werden kann, die MDL1–Beschreibung muß nach wie vor vom Compilerentwickler erstellt werden, denn genau das ist ja seine neue Aufgabe. Die Anpassung an eine neue HDL bedeutet dann jedenfalls lediglich die Änderung des Adapters.

1.4 Anforderungen

In diesem Abschnitt werden nun die Anforderungen an die MDL sowie an den Analysator und den Generator zusammengefaßt. In den schließenden Abschnitten „Zielüberprüfung“ der folgenden Kapitel wird dann verglichen, wie sowohl die Sprache, als auch die Programme die Anforderungen erfüllen. Zweck der Beschreibung eines Mikroprozessors mit der Sprache MDL ist die Lösung der in den vorangegangenen Abschnitten aufgezeigten Probleme. Vor allem die folgenden Punkte sind elementar:

- Normung der Schnittstelle zwischen Hardwareentwickler und Compilerentwickler und Verminderung des Kommunikationsaufwandes.
- Automatische Auflistung der hardwarebedingten Konflikte zum Zwecke eines zusätzlichen Hardwareentwicklungszyklus mit der Möglichkeit, auch auf einer abstrakten Maschine entwickeln zu können.
- Automatische Generierung der hardwarebedingten semantischen Regeln sowie effiziente Implementierung der sonstigen semantischen Prüfungen sowie der Codeerzeugung.
- Auswahl der prozessorspezifischen Sprachuntermenge aus dem fest vorgegebenen Sprachrahmen.

Die Sprache MDL muß daher die Beschreibung folgender Sachverhalte ermöglichen:

1. Die *Hardwareressourcen* des Prozessors. Das ist die Netzstruktur bestehend aus Hardwareelementen und -verbindungen. Anders als bei anderen Hardwarebeschreibungssprachen sind aber hier weder die funktionalen Eigenschaften der Hardwareelemente noch die Breite der Verbindungen interessant. Als besondere Ressource ist auch die Beschreibung des Mikrobefehlsfeldes von großer Bedeutung.

2. Die Verwendung der Hardwareressourcen in den *Mikrooperationen* mit Zeitangaben und unterschiedlichen Arten der Benutzung.

3. Die Zuordnung der *Syntaxelemente* der Mikroprogrammsprache sowohl zu den Mikrooperationen als auch zur Codegenerierung.

Neben der Frage, *was* eine Sprache beschreiben soll, ist auch die Frage, *wie* es beschrieben werden soll, von Bedeutung. Folgende Eigenschaften soll die Sprache MDL besitzen:

deskriptiv: Im Gegensatz zu logischen, funktionalen oder prozeduralen Sprachen ist die MDL rein deskriptiv, also beschreibend: Das liegt vor allem daran, daß für die Entwicklung des Firmwarecompilers Syntax und Semantik der Mikroprogrammsprache ausreichen, erst der Firmwareentwickler benötigt zusätzliche Information über die funktionalen Zusammenhänge. Da jedoch in den meisten Fällen Hardwareentwickler und Firmwareentwickler sehr eng zusammenarbeiten, wenn nicht sogar ein und dieselbe Person sind, stellt das kein Problem dar.

effizient: Diese Eigenschaft ist gleichbedeutend mit geringer Redundanz: Die Sprache muß so beschaffen sein, daß jene Elemente und Zusammenhänge, die in der Praxis sehr oft beschrieben werden müssen, besonders einfach und übersichtlich beschrieben werden können. Damit Hand in Hand gehen leichte Änderbarkeit und geringe Fehleranfälligkeit.

einfach: Damit ist vor allem die Einfachheit der syntaktischen Struktur gemeint: Die Sprache soll eine möglichst übersichtliche und einheitliche Syntax besitzen, um leicht und rasch erlernbar zu sein. Noch günstiger wäre die Verwandtschaft zu einer bestehenden und weithin bekannten Sprache.

flexibel: Das bedeutet, daß trotz der eben angesprochenen Einfachheit die Sprache eine hohe Mächtigkeit in der Möglichkeit und Vielfalt des Beschreibbaren besitzt.

großzügig: Dieser Punkt ist die Grundlage für die Verwendung der Sprache im Rahmen eines Entwurfswerkzeuges: Eine MDL–Beschreibung soll korrekt und bearbeitbar sein, auch ohne Festlegung aller Einzelheiten bis ins kleinste Detail, wie zum Beispiel konkrete Bitcodes oder Bezeichner. Eine Menge von logischen impliziten Annahmen ergänzt Unvollständiges im Sinne des Benutzers, sodaß eine MDL–Beschreibung auch für abstrakte Maschinen möglich ist.

Die Abstraktionsebene, auf der die MDL–Beschreibung angesiedelt ist, befindet sich etwa zwischen der Architekturebene[10] und der Organisationsebene[11]. Das liegt daran, daß zwar

[10] *Instruction set level*
[11] *Register transfer level*

die Hardwarestruktur auf Organisationsebene beschrieben wird, andererseits aber die Mikrooperation auf Architekturebene das elementare Objekt zur Konflikterkennung und Syntax/Code–Umsetzung darstellt.

Nachdem nun die Anforderungen an die Sprache erläutert wurden, sollen auch die Anforderungen an den *Analysator* zusammengefaßt werden:

- Einlesen einer MDL0–Beschreibung und Prüfung der syntaktischen Korrektheit. Beim Auftreten von Syntaxfehlern soll die weitere Analyse mit entsprechenden Fehlermeldungen abgebrochen werden. Wenn hingegen die MDL–Beschreibung semantisch fehlerhaft oder unvollständig ist, soll mit Hilfe impliziter Annahmen die Analyse nach Möglichkeit fortgesetzt werden. Dies entspricht der Forderung nach einem großzügigen Entwurfswerkzeug.
- Ausgabe einer Tabelle aller verwendeten Bezeichner, die dem semantischen Vorlauf zwecks Prüfung der hardwarespezifischen Namen zur Verfügung gestellt wird.
- Ausgabe einer Tabelle aller überhaupt möglichen Konflikte sowie Erstellung von konkreten Konfliktanalysen für bestimmte Anfragen, zum Beispiel eine konkrete Kombination von Mikrooperationen.

Die Anforderungen an den *Generator* werden hier ebenfalls kurz aufgezählt, obwohl der Generator selbst nicht mehr Bestandteil der Diplomarbeit ist:

- Einlesen einer MDL1–Beschreibung und syntaktische Prüfung.
- Erzeugung von Konflikttabellen in einer Form, die zur Steuerung der semantischen Prüfungen geeignet ist. Gegebenenfalls müssen nicht nur Tabellen, sondern auch Unterprogramme zum Firmwarecompiler generiert werden.
- Auswahl der Sprachuntermenge aus dem Sprachrahmen und Angaben über Einschränkungen, die der Firmwareprogrammierer selbst berücksichtigen muß, weil sie sich der Überprüfung durch den Compiler entziehen. Dabei kann es sich zum Beispiel um Konflikte handeln, die sich erst zur Laufzeit ergeben, wie etwa bei der indirekten Adressierung.
- Generierung von Tabellen und Funktionen zur Steuerung der Umsetzung der Syntaxelemente in den Bitcode.
- Bereitstellen einer Schnittstelle, über die der Compilerentwickler steuernd in den Analysevorgang eingreifen kann, etwa um Sonderfälle abzuhandeln.

Bei der Sprachbeschreibung wird im Zuge der Syntax/Code–Zuordnung noch näher auf den Generator und seine Aufgaben eingegangen, da sich mit Einführung des Generators gleichzeitig ein Redesign des Firmwarecompilers insgesamt ergibt.

Um später näher darauf eingehen zu können, wie *Konflikte* analysiert werden, wird hier zunächst eine Einteilung der Konfliktarten getroffen. Konflikte zwischen Mikrooperationen entstehen ganz allgemein dann, wenn Hardwareressourcen zu einem bestimmten

Zeitpunkt mehrfach genutzt werden. Typische Konfliktursachen sind zum Beispiel Register und Busbenutzung oder überlappende Bitfelder im Mikrobefehl. Grundsätzlich kann man aus der Sicht der Compilergenerierung drei Arten von Konflikten unterscheiden:

Totale Konflikte entstehen, wenn zwei Mikrooperationen unabhängig von einem konkreten Mikroprogramm einen Ressourcenkonflikt verursachen. Solche Konflikte *erster Art* entstehen zum Beispiel dann, wenn Bitfelder im Mikrobefehlsfeld doppelt genutzt werden, wenn zwei Mikrooperationen die einzige ALU[12] zur selben Zeit benötigen oder ähnliches. Solche Konflikte bedeuten für den Compilerentwickler beziehungsweise später für den Generator die absolute Unvereinbarkeit dieser Mikrooperationen in einem Mikrobefehl. Wenn hingegen bereits innerhalb einer Mikrooperation ein totaler Konflikt auftritt, so handelt es sich um einen Fehler, den der Analysator entsprechend melden und behandeln muß und der bereits vom Hardwareentwickler beseitigt werden muß.

Partielle Konflikte sind abhängig von einer konkreten Mikroprogrammeingabe, man könnte sie auch als Übersetzungskonflikte bezeichnen, da sie erst zum Zeitpunkt der Übersetzung eines Mikroprogrammes im Firmwarecompiler konkret geprüft werden können. Dazu ein Beispiel: Wenn zwei Mikrooperationen je ein Register aus einer Menge von zum Beispiel 64 Registern benutzen, so handelt es sich nur um einen möglichen Konflikt, der nicht zwingend eintreten muß. Der Compiler muß dann für eine konkrete Eingabe prüfen, ob ein Konflikt vorliegt, weil beide Mikrooperationen dasselbe Register beanspruchen, oder nicht. Für den Compilerentwickler beziehungsweise den Generator bedeutet ein solcher Konflikt *zweiter Art* die Implementierung von entsprechenden semantischen Einschränkungen, die wesentlich komplexer sind, als die Überprüfung von totalen Konflikten. Im Konfliktfall gibt es dann grundsätzlich die zwei Möglichkeiten, entweder die Übersetzung mit einer Fehlermeldung abzubrechen, oder einer der beiden Mikrooperationen den Vorrang einzuräumen und die andere zu ignorieren.

Laufzeitkonflikte können vom Compiler überhaupt nicht überprüft werden, da sie zur Übersetzungszeit noch nicht feststehen. Ein typisches Beispiel bietet der Registerzugriff wie vorhin, jetzt aber mittels indirekter Adressierung: Der Inhalt der Pointerregister steht zur Übersetzungszeit nicht fest, daher kann nicht überprüft werden, ob die beiden Mikrooperationen dasselbe Register beanspruchen oder nicht. Solcherlei Konflikte sollten eigentlich von der darunterliegenden Hardware selbst erkannt und richtig behandelt werden. Sollte dies aber nicht der Fall sein, dann muß die Möglichkeit solcher Konflikte dem Firmwareprogrammierer im Handbuch mitgeteilt werden. Diese Vorgangsweise bietet sich auch als Notlösung für solche Konflikte an, die strenggenommen zwar nicht zu den Konflikten *dritter Art* gehören, aber für eine Behandlung im Compiler zu komplex sind.

Abschließend noch eine grundsätzliche Bemerkung: Insgesamt soll die Diplomarbeit eine Studie darstellen, welche die *Machbarkeit* anhand eines Prototyps zeigt, Möglichkeiten und Varianten diskutiert und die wichtigsten in der Praxis vorkommenden Fälle abdeckt.

[12] *Arithmetical Logical Unit*

Hingegen wurde bei der Realisierung keinerlei Wert auf Optimierungen aller Art, wie etwa Geschwindigkeit oder Speicherplatz, gelegt. Auch die Abdeckung *aller* auch noch so ausgefallenen Möglichkeiten würde den Rahmen der Diplomarbeit bei weitem übersteigen.

Im nächsten Kapitel wird nun die Sprache MDL in ihrer Gesamtheit vorgestellt, im darauffolgenden Kapitel wird dann die durchgeführte Implementierung beschrieben. Das letzte Kapitel schließlich analysiert zusammenfassend die Zukunft der Mikroprogrammierung sowie die Anwendbarkeit der vorgestellten Methoden auch in anderen Bereichen.

Kapitel 2

Sprachbeschreibung

2.1 Einleitung

Eine neue Sprache zu entwickeln ist nicht zuletzt deshalb problematisch, weil es schon eine Unmenge von abstrakten Sprachen für verschiedenste Zwecke gibt. Um dieses babylonische Sprachgewirr nicht weiter zu fördern, kann man die neue Sprache an eine bestehende „anlehnen“. Dies ist insbesonders dann sinnvoll, wenn die neue Sprache einem speziellen Zweck dient und die Sprachvorlage eher allgemein gehalten und günstigerweise weit verbreitet ist. Deshalb orientiert sich MDL an der Programmiersprache C[1].

Der lexikalische Aufbau von MDL entspricht dem der Programmiersprache C bis auf vier Unterschiede:

1. Es gibt keine Fließkommakonstanten.
2. Eine Binärzahlenkonstante ist gemäß Abbildung 2.1 definiert.
3. `Component` ist als spezieller Bezeichner nach Abbildung 2.2 festgelegt.
4. Es gibt zwei zusätzliche Operatoren <! und >!.

Soweit notwendig, wird bei der entsprechenden Syntaxbeschreibung noch näher auf bestimmte lexikalische Elemente eingegangen. Eine vollständige Beschreibung der lexikalischen Analyse befindet sich in Anhang L.

Die Syntax wird im Zuge der Vorstellung der einzelnen Sprachelemente stets in Form von Diagrammen präsentiert. Man erhält eine syntaktisch richtige Form, wenn man den Linien der Syntaxdiagramme folgt. Schlüsselwörter, Begrenzer, Operatoren und andere Terminalsymbole sind dabei in abgerundeten Kästchen enthalten, während Nonterminalsymbole oder Grammatikbegriffe in rechteckigen Kästchen stehen. Bei den Beispielen sind die Schlüsselwörter dann **fett** hervorgehoben. Außerdem sind die Beispiele stets durch

[1]Es wurde bereits im vorigen Kapitel darauf eingegangen, warum nicht eine gängige Hardwarebeschreibungssprache wie etwa VHDL als Vorlage für MDL gewählt wurde.

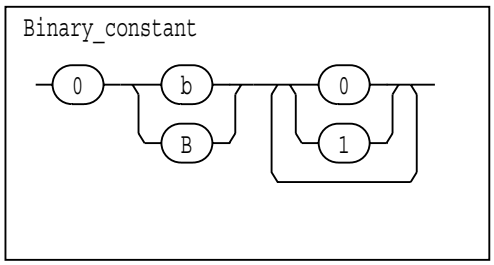


Abbildung 2.1: Die Binärzahlenkonstante entspricht im Aufbau der aus C bekannten Hexadezimalzahlkonstanten. Sie wurde eingeführt, um speziell bei der Codierung die nötigen Bitmuster direkt angeben zu können.

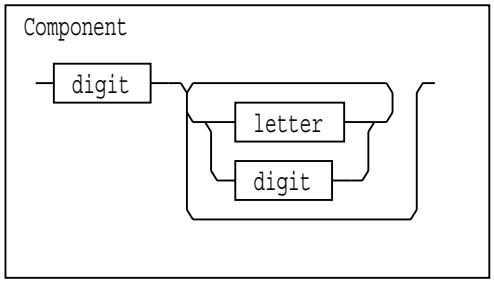


Abbildung 2.2: `Component` ergänzt `Identifier` dahingehend, daß auch Ziffern am Beginn erlaubt sind, während der „strengere" `Identifier` mit einem Buchstaben oder einem Unterstrich beginnen muß. Diese Unterscheidung wird notwendig, weil bei der Bezeichnung von Bitfeldausschnitten auch Namen erlaubt sind, die mit Ziffern beginnen oder überhaupt nur Zahlen sind.

einen Strichpunkt abgeschlossen, der in den Syntaxdiagrammen nicht aufscheint: Das liegt daran, daß der Strichpunkt in den übergeordneten Syntaxdiagrammen enthalten ist. Die vollständige Syntaxbeschreibung ist in Anhang S enthalten.

Nicht unmittelbar Bestandteil der MDL, aber dennoch sehr nützlich sind die C Präprozessoranweisungen. Diese werden eigentlich erst durch die Implementierung des Parsers eingebunden und beeinflussen nicht die Funktionalität der MDL selbst. Die Verwendung des C Präprozessors[2] wird jedoch empfohlen, um die Handhabbarkeit einer MDL–Beschreibung zu erleichtern. Hier werden die Standard C Präprozessoranweisungen verwendet, vor allem folgende:

- Der Kommentar: `/* Kommentar */`. Ausführlich mit Kommentaren versehen, kann die MDL–Beschreibung zugleich ein maßgebliches Element der Dokumentation darstellen.
- Die Makrodefinition `#define` dient vor allem der einfacheren Handhabung globaler Größen.

[2]Es kann natürlich auch jeder andere Präprozessor verwendet werden, der aber zumindest die Möglichkeit von Kommentaren, Makrodefinitionen und Dateiinkludierung bieten sollte.

- Die Dateiinkludierung: `#include`. Damit können umfangreichere Beschreibungen handlich modularisiert werden. Auch der Aufbau einer Bibliothek wird dadurch unterstützt.

In der Praxis wird die Einbindung der Präprozessoranweisungen durch einen Vorlauf des entsprechenden Präprozessors vor dem eigentlichen MDL–Parser erreicht werden, sodaß dann bei entsprechender Handhabung auch die Optionen für den Präprozessor verwendbar sind. Zusätzlich zu den hier erwähnten Anweisungen können dann alle Befehle des lokal eingebundenen Präprozessors verwendet werden. Im Sinne der Portabilität einer MDL–Beschreibung sollte man sich aber an die genormten Anweisungen halten, wie sie etwa in [17] beschrieben sind. Dort kann man auch nähere Details zu den C Präprozessoranweisungen nachlesen.

Die komplette MDL-Beschreibung eines Prozessors kann man logisch in drei Teile gliedern:

Deklarationsteil: Darin werden die Hardwareressourcen und –verbindungen beschrieben.

Operationsteil: Hier erfolgt die Beschreibung der Verwendung der Ressourcen in den verschiedenen Mikrooperationen.

Syntaxteil: Dieser Teil enthält die Syntax/Code–Zuordnung.

Diese logische Gliederung wird hier in weiterer Folge verwendet, um die Übersichtlichkeit zu erhöhen. Es besteht aber kein Zwang, daß eine MDL-Beschreibung sich an diese Gliederung halten muß. Lediglich eine Einschränkung muß bei der Eingabe der Prozessorbeschreibung beachtet werden: Die Deklaration muß stets vor der Verwendung erfolgen. Anders als in einigen Programmiersprachen gibt es also keine *forward*–Deklarationen[3].

Vor Beginn der eigentlichen MDL–Beschreibung *können* noch zwei Angaben erfolgen: Die Breite des Mikrobefehlsfeldes mit dem Schlüsselwort `MAXFIELD`, gefolgt von einer konstanten Zahlenangabe, und die Dauer eines gesamten Mikrobefehlszyklus mit dem Schlüsselwort `MAXTIME`, ebenfalls von einer konstanten Zahlenangabe gefolgt. Werden diese Angaben getätigt, dann finden sie in der Konfliktanalyse ihren Niederschlag. Werden sie weggelassen, treten implizite Annahmen an ihre Stelle, die später noch genauer erläutert werden.

Die nun folgende Sprachbeschreibung ist sehr kompakt und nicht unmittelbar als Einführungsunterricht geeignet, da ein solcher entsprechend aufbereitet den Rahmen der Diplomarbeit bei weitem sprengen würde. Zwar sind zur Erläuterung zahlreiche Bilder und Beispiele angegeben, diese sind aber ebenfalls komprimiert, um in Kürze möglichst viele Ausprägungsmöglichkeiten darzustellen. Ziel ist es vielmehr, die Sprache vollständig und in ihrer vollen Mächtigkeit vorzustellen, um somit einen Eindruck von deren Fähigkeiten zu vermitteln.

[3]Bei Programmiersprachen wird dieser Mechanismus oft benötigt, um gegenseitig verschachtelte rekursive Prozeduren oder verkettete Datentypen zu beschreiben. Da es in MDL keine derartigen Strukturen gibt, kann hier auf diese Möglichkeit verzichtet werden.

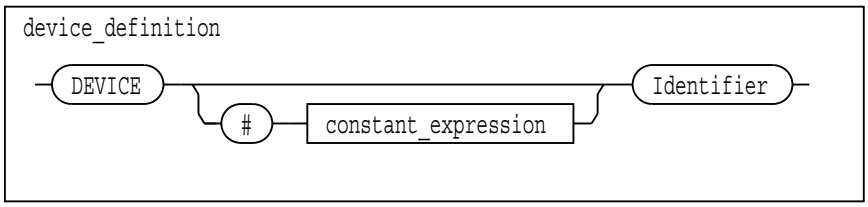


Abbildung 2.3: Die Definition eines Grundelementes ist sehr einfach, da die funktionalen Zusammenhänge für MDL nicht wichtig sind. Bei `constant_expression` handelt es sich um einen Ausdruck mit Konstanten, Operatoren und Klammern, wie er in der Programmiersprache C gebräuchlich ist.

2.2 Ressourcen

Im Deklarationsteil werden alle Hardwareressourcen beschrieben, also die Elemente der Hardwarestruktur und ihre Verbindungen. Da MDL eine deskriptive Sprache ist und somit die funktionalen Zusammenhänge unberücksichtigt bleiben, können die meisten Hardwareelemente als einfache *Grundelemente* beschrieben werden. Lediglich die *Register* werden gesondert behandelt. Eine besondere Ressource ist das *Mikrobefehlsfeld*: Bestimmte Ausschnitte davon können als eigenständige Ressourcen definiert werden. Diese elementaren Ressourcen können dann auf verschiedene Arten zu *Gruppen* zusammengefaßt werden. Außerdem müssen auch die *Verbindungen* zwischen den Hardwareelementen beschrieben werden.

Grundelemente

Alle Hardwareelemente, ausgenommen Register, können als Grundelemente definiert werden, da ihre näheren Eigenschaften für die Generierung des Compilers nicht relevant sind. Dies geschieht einfach durch Angabe des Schlüsselwortes `DEVICE`, gefolgt vom Namen des Grundelementes. Abbildung 2.3 zeigt die Syntax der Beschreibung, Beispiel 2.1 eine mögliche Anwendung. Der Ausdruck `# constant_expression` bedeutet eine entsprechende Anzahl gleichartiger Ressourcen, deren Vergabe nicht durch die Firmwareprogrammierung beeinflußt werden kann. Dies ist bei bestimmten Hardwareelementen durchaus sinnvoll, wo eine einfache Hardwaresteuerung die Verteilung mehrerer gleicher Ressourcen an die Mikrooperationen vornimmt. Fehlt diese Angabe, so wird `#1` angenommen.

Dazu noch eine Bemerkung: Überall, wo konstante Zahlenangaben erwartet werden, ist das Rechnen mit Konstanten zulässig. Die genaue Syntax ist in Anhang S beschrieben und orientiert sich an der Programmiersprache C. Zumeist wird man zwar mit einer einfachen Zahlenangabe auskommen, der Vorteil dieser Methode ist eher im Zusammenhang mit der Definition von globalen Konstanten zu sehen: Mit `#define` kann ein globaler Wert sehr übersichtlich am Anfang einer MDL–Beschreibung definiert werden. Dieser Wert kann nun an den entsprechenden Stellen unter Verwendung der Konstantenrechnung eingebracht werden. Eine Änderung dieses Wertes findet dann nur ein einzigesmal am Anfang der Beschreibung statt, der Anwender erspart sich das mühevolle und fehleranfällige Durchforsten der gesamten Beschreibung. Die Namen für solche Konstanten

```
DEVICE alu;        /* arithmetical logical unit */

DEVICE #2 c_bus;   /* result bus */

DEVICE align;      /* alignment */
```

Beispiel 2.1: Es werden hier einige Hardwareelemente beschrieben, die für MDL jedoch nur dem Namen nach unterschiedlich sind. Den c_bus gibt es zweimal bei sonst identischen Eigenschaften. Welcher c_bus im speziellen Fall tatsächlich verwendet wird, entzieht sich der Steuerung durch die Firmware und wird rein durch 'hardwired logic' entschieden. Die durch die Kommentare angedeutete Funktionalität der einzelnen Hardwareelemente dient lediglich der besseren Lesbarkeit für den Anwender.

sollten aber so gewählt werden, daß sie auf einen Blick von anderen Bezeichnern in der Beschreibung, aber auch von den großgeschriebenen Schlüsselwörtern unterscheidbar sind[4].

Besonders hingewiesen wird an dieser Stelle auf die Möglichkeit der Kommentare in der in C üblichen Syntax. Bei intensiver Nutzung dieser Kommentare kann die Prozessorbeschreibung neben der abstrakten Information auch beliebige Beschreibungen in einer für den Menschen angenehmen Form enthalten. Damit kann die MDL–Beschreibung die meisten der Informationen enthalten, die zwischen Hardwareentwickler und Compilerentwickler auszutauschen sind.

Register

Register werden mit dem Schlüsselwort `REGISTER` definiert und erhalten, ebenso wie die Grundelemente, einen Namen. Ihre Definition unterscheidet sich von der anderer Hardwareelemente aufgrund dreier Ursachen:

1. Die Möglichkeit der Bildung von Arrays[5].
2. Die Information für die Syntax/Code–Zuordnung.
3. Die Möglichkeit von Multiport–Registern.

Die Arrays können beliebig-dimensional sein, in der Praxis wird man aber zumeist mit zweidimensionalen Arrays das Auslangen finden. Fehlt die Angabe eines Arrays, so wird genau ein Register definiert. Die Information für die Syntax/Code–Zuordnung beinhaltet die Bitbreite der Register (`range`) sowie die Benennung von Teilen von Registern

[4]Der Autor bevorzugt die Methode, Konstante mit Großbuchstaben beginnen zu lassen, während sonstige Bezeichner ausschließlich klein geschrieben werden.

[5]Überall dort, wo die Gefahr der Verwechslung besteht, wird das englische Wort *Array* verwendet, um mehrdimensionale Registerfelder zu kennzeichnen, während der deutsche Begriff *Feld* zumeist mit Bitfeldern oder dem Mikrobefehlsfeld selbst assoziiert wird.

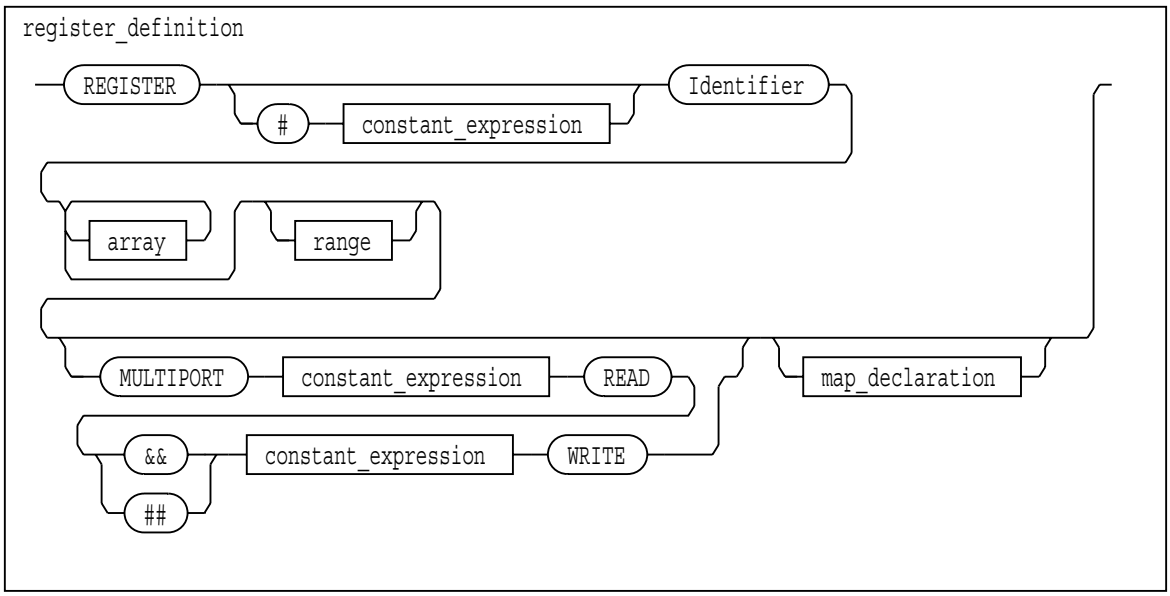


Abbildung 2.4: Die Definition eines Registers beinhaltet zusätzlich zur Beschreibung eines `DEVICE` einige weitere optionale Punkte: Ein `array`, ein `range`, eine `MULTIPORT`–Definition und eine `map-declaration`.

(`map_declaration`). Multiport–Register erlauben gleichzeitig mehrere Lese- und Schreib–Zugriffe. Abbildung 2.4 zeigt das vollständige Syntaxdiagramm, Beispiel 2.2 mögliche Definitionen. Das einfachste zuerst: Mit dem Schlüsselwort `MULTIPORT` wird die Anzahl der Lese- und Schreib–Ports des Registers festgelegt. Dies geschieht mit den Schlüsselwörtern `READ` und `WRITE` in genau dieser Reihenfolge hinter der entsprechenden Anzahl. Außerdem kann festgelegt werden, ob Lese- und Schreib–Zugriffe gleichzeitig (`&&`) oder nur alternativ (`##`) stattfinden können. Die `MULTIPORT`–Angabe ist optional. Wenn sie fehlt, wird angenommen, daß nur ein Lese- oder ein Schreib–Zugriff gleichzeitig stattfinden kann. Wenn hingegen eine `MULTIPORT`–Definition stattfindet, dann muß sie vollständig sein und der obigen Reihenfolge gehorchen.

Was bedeuten nun `array` und `range` genau? Zur Erklärung wird zunächst der Begriff des *Bereiches* vorgestellt: Ein Bereich ist eine Anzahl von aufeinanderfolgenden Elementen, Abbildung 2.5 zeigt das Syntaxdiagramm und in Tabelle 2.1 sind dazu einige Möglichkeiten angegeben. Ein Bereich kann auf mehrere Arten angegeben werden: Sind die beiden Zahlen des Bereiches durch einen Doppelpunkt getrennt, so handelt es sich um erstes und letztes Element des Bereiches. Steht hingegen ein Beistrich zwischen den beiden Zahlen, so handelt es sich um Startelement und Länge des Bereiches. Besteht der Bereich nur aus einer einzelnen Zahl, so handelt es sich um genau dieses Element. Dann gibt es noch einige Möglichkeiten von unvollständigen Angaben, die auf den ersten Blick zwar sinnlos erscheinen mögen, aber unter bestimmten Umständen sehr nützlich sind: Ein `#` vor der Zahl bedeutet eine Längenangabe bei unbekanntem Offset, ein `#` allein oder ein leerer Bereich bedeuten unbekannte Länge und unbekannten Offset. Auch diese Bereichsangaben sind in speziellen Fällen sinnvoll, wo sie durch implizite Annahmen entsprechend ergänzt werden. Diese impliziten Annahmen sind jedoch von Fall zu Fall

```
REGISTER #2 io_port;

REGISTER portreg<0:15> MAP adr;

REGISTER reg[0:3][0:15]<0:31>
         MULTIPORT 2 READ && 1 WRITE
         MAP word;
```

Beispiel 2.2: Hier sieht man die Definition einiger Register. Die erste Zeile beschreibt ein Register mit Namen `io_port`, welches zweifach vorhanden ist. Die zweite Zeile beschreibt ein Register `portreg`, dessen Breite 16 Bit beträgt. Die letzte Zeile ist die Definition eines zweidimensionalen Registerfeldes mit insgesamt 64 Registern. Jedes der Register ist 32 Bit breit und erlaubt gleichzeitig maximal zwei Lese- und einen Schreib–Zugriff. Teilbereiche von Registern, deren Bitbreite definiert ist, können gemäß der Definiton der jeweiligen `MAP` angesprochen werden. Diese definieren gewisse Bitstrukturen und werden später noch genauer erläutert.

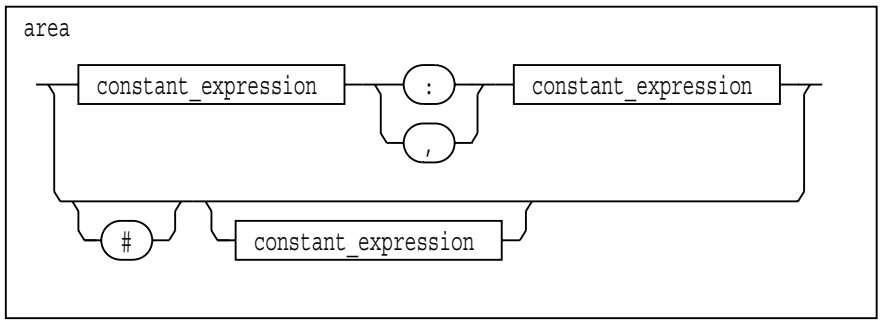


Abbildung 2.5: Die Syntax der Bereichsangabe.

Bereich	Elementenfolge	Offset/Startelement	Länge	Endelement
2:7	2 3 4 5 6 7	2	6	7
2,6	2 3 4 5 6 7	2	6	7
5	5	5	1	5
#5	? ? ? ? ?	?	5	?
#	???	?	?	?

Tabelle 2.1: Beispiele zum Bereich: Die erste Zeile beschreibt einen Bereich der Länge 6, der mit Element 2 beginnt und mit Element 7 endet. Die zweite Zeile beschreibt denselben Bereich durch Angabe von Startelement und Länge. Die dritte Zeile beschreibt das Element 5 allein, während die Zeile darunter einen Ausschnitt beschreibt, der an undefinierter Stelle beginnt, dessen Länge aber mit 5 festgelegt ist. Die letzte Zeile beschreibt einen undefinierten Bereich. Diese ganz oder teilweise undefinierten Bereiche sind nur in ganz speziellen Fällen sinnvoll.

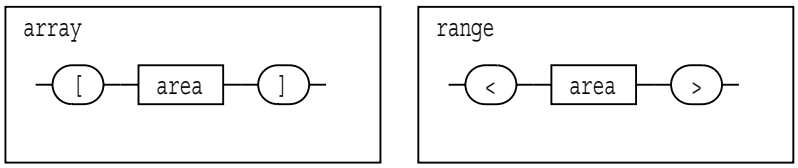


Abbildung 2.6: Die Syntax von `array` und `range`.

```
REGISTER reg[0,4][0,16]<0,32>;

#define Word 16

REGISTER reg[#4][#0x10]<#Word*2>;
```

Beispiel 2.3: Einige Beispiele für alternative Bereichsbeschreibungen. Beim zweiten Beispiel erkennt man eine mögliche Anwendung eines Bereiches mit unbestimmtem Offset. Außerdem zeigt es eine mögliche Anwendung der Konstantenrechnung.

verschieden, sodaß deren genaue Kenntnis für die Verwendung unbestimmter Bereichsangaben unumgänglich ist.

Die Abbildung 2.6 zeigt die Verwendung des Bereiches in den Definitionen für `array` und `range`. Im ersten Fall beschreibt der Bereich die Feldindizes des Registerfeldes, im zweiten Fall den Bitbereich eines einzelnen Registers. Wird kein vollständiger Bereich, sondern nur eine Längenangabe angegeben, so wird implizit Offset Null angenommen. Beispiel 2.3 zeigt einige alternative Beschreibungen für das Registerfeld aus Beispiel 2.2.

Strukturierung von Bitfeldern

Die `map_declaration` aus dem vorigen Beispiel dient der Namensgebung für Registerteile und gehört logisch zur Syntax/Code–Zuordnung (Abschnitt 2.4). Die Strukturierung von Bitfeldern ganz allgemein wird hier aber vorgezogen beschrieben, da sie nicht nur der Syntaxbeschreibung dient, sondern auch in der MDL–Beschreibung selbst intensiv Verwendung findet.

Die `MAP`–Definition leistet diese Strukturierung von Bitfeldern, indem sie bestimmten Teilbereichen eigene Namen zuweist. In Abbildung 2.7 sind die entsprechenden Syntaxdiagramme gezeigt, Beispiel 2.4 zeigt einige typische Anwendungen. Jede `MAP`–Struktur erhält bei ihrer Definition einen Namen zugewiesen, mit dem sie später angesprochen werden kann. Jede Zeile (`map_def`) einer `MAP`–Definition definiert einen Namen (`symbol`) für einen bestimmten Bitbereich (`range`). Wenn der Bereich nur eine Längenangabe enthält, wird angenommen, daß der Bereich an den vorangegangenen anschließt. Enthält bereits die erste Zeile der `MAP`–Definition nur eine Längenangabe, wird angenommen, daß der Bereich bei Null beginnt. Die Bereiche in einer `MAP` dürfen durchaus lückenhaft oder auch

```
MAP portmap {
    <0,2> high;
    <2,2> low;
};

MAP adr1 {                  /* mapping for port field B */
    <#2>   switch;          /* group */
    <3:5>  word;            /* 8 words total*/
    <#3>   digit;           /* 8 digits per word */
    <10>   direction;       /* left = 0, right = 1 */
    <12,4> port             /* port selector */
                MAP portmap;
    <3:8>  dir_digit;       /* 64 digits total */

};

MAP adr2 {                  /* mapping for port field A */
    <#2>   switch;          /* group */
    <2:8>  immediate;       /* 0..127 */
    <12,4> port             /* port selector */
                MAP portmap;
    <3:8>  dir_digit;       /* 64 digits total */

};

MAP adr OF adr1, adr2;   /* mapping for total port field */
```

Beispiel 2.4: Typische Anwendungen der MAP–Definition: Die MAP portmap definiert eine einfache Struktur mit dem Feld high (Bits 0 und 1) und dem Feld low (Bits 2 und 3). Die MAP adr1 definiert folgende Felder: Das Feld switch besteht aus den Bits 0 und 1, das Feld word aus den Bits 3 bis 5. Das Feld digit schließt direkt daran an und umfaßt die Bits 6 bis 8. Das Feld direction besteht aus dem Bit 10 allein, das Feld port aus den Bits 12 bis 15. Das Feld direct_digit schließlich enthält die Bits 3 bis 8. Wie man sieht, gibt es sowohl Überlappungen als auch Lücken. Interessant ist das Feld port, welches seinerseits mit einer Bitstruktur versehen ist: Über die MAP portmap, die bereits zuvor definiert wurde, können die Bits 12 und 13 als Subfeld high angesprochen werden, die Bits 14 und 15 als Subfeld low. Beispiele für die Verwendung dieser MAP–Definition folgen später. Die MAP adr2 ist ähnlich definiert. Die MAP adr schließlich stellt die Vereinigung der beiden MAP–Definitionen adr1 und adr2 dar. Zwar sind die Felder switch, port und dir_digit in beiden zu vereinigenden Bitfeldstrukturen definiert, da sie aber in beiden Strukturen völlig identisch definiert sind, gibt es bei der Vereinigung keine Probleme. Die MAP adr stellt somit die Obermenge der beiden vereinigten Bitfeldstrukturen dar, sie enthält die Felder switch, word, digit, direction, port, dir_digit und immediate.

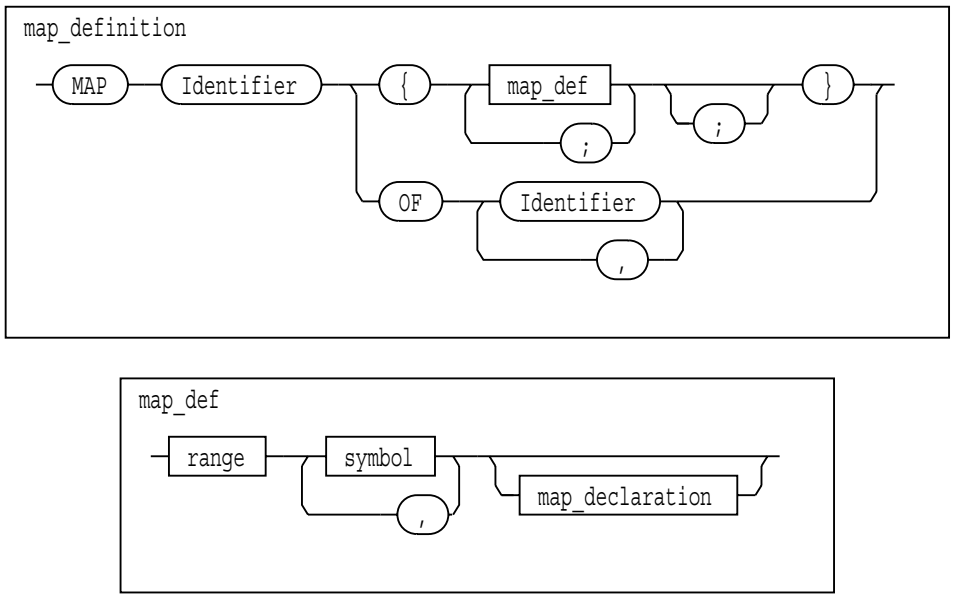


Abbildung 2.7: Die Syntax der MAP–Definition erinnert an die Definition einer `struct` in der Programmiersprache C. Die Semantik unterscheidet sich aber wesentlich.

überlappend sein. Es dürfen aber keine zwei verschiedenen Bereiche den gleichen Namen besitzen, wobei zwei Bereiche auch dann als verschieden gelten, wenn sie unterschiedliche MAP–Deklarationen besitzen.

Eine alternative Form der MAP–Definition ist die Zusammensetzung einer MAP aus bereits definierten MAP–Definitionen mit dem Schlüsselwort OF. Dabei ist zu beachten, daß in den zusammenzusetzenden MAP–Definitionen keine gleichen Bereichsnamen verschiedene Bereiche beschreiben. Es ist jedoch durchaus zulässig, daß in den zusammenzusetzenden MAP–Definitionen gleiche Bereichsnamen vorkommen, wenn sie auch den gleichen Bereich beschreiben und gegebenenfalls auch dieselbe optionale MAP–Struktur besitzen.

Jeder in einer MAP–Definition enthaltene Bereich kann seinerseits wieder mittels einer `map_declaration` strukturiert werden. Die einfachste Form einer `map_declaration` ist die Angabe des Namens einer bereits definierten MAP hinter dem Schlüsselwort MAP. Darüberhinaus gibt es jedoch die Möglichkeit, zugleich mit der Angabe die MAP auch zu definieren. Abbildung 2.8 zeigt die volle Mächtigkeit der `map_declaration`. Eine derart definierte MAP besitzt globale Gültigkeit und kann an beliebiger Stelle verwendet werden. Wenn hingegen die MAP im Zuge der `map_declaration` anonym, also ohne Angabe eines `Identifiers` deklariert wird, besitzt sie nur lokale Gültigkeit und kann sonst nirgends verwendet werden. Dies ist nur sinnvoll, wenn eine bestimmte Strukturierung nur ein einzigesmal benötigt wird.

Stehen mehrere Namen (`symbol`) in einer Zeile hinter dem Bereich (`range`), dann gilt für den ersten das bisher Gesagte, während die folgenden Namen jene Bereiche beschreiben, die an den ersten anschließen und gleiche Länge besitzen. Eine optionale `map_declaration` gilt für alle aufgelisteten Namen der Zeile. Diese Methode stellt vor allem eine effiziente Abkürzung für den häufigen Fall dar, daß alle Teilbereiche gleich

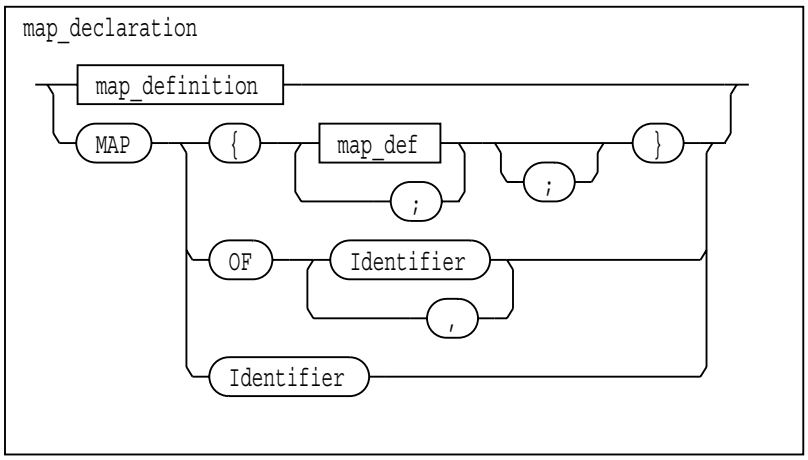


Abbildung 2.8: Die Syntax für eine `map_declaration`, das ist die Strukturierung einer Ressource wie z.B. eines Registers mittels einer `MAP`. Dabei kann entweder eine bereits definierte `MAP` angesprochen werden oder aber eine neue `MAP` definiert werden. Wird eine `MAP` neu definiert, erhält sie globale Gültigkeit, wenn ein Name angegeben wird, ansonsten ist sie nur lokal gültig.

lang sind. Beispiel 2.5 zeigt einige „fortgeschrittenere“ `MAP`–Definitionen.

Zurück zum Beispiel des Registers aus dem vorigen Abschnitt: Dieses wurde mit der `map_declaration MAP word` mit einer Bitfeldstruktur versehen. Wie werden nun die entsprechenden Registerteile angesprochen? Dies geschieht einfach mittels Dereferenzierung durch Angabe des Namens des Teilbereiches, der durch einen Punkt vom Namen der Ressource getrennt ist. Abbildung 2.9 zeigt das zugehörige Syntaxdiagramm des `resource_name` und beschreibt, wie eine bereits definierte Ressource in weiterer Folge ganz oder teilweise angesprochen werden kann.

Noch eine Bemerkung zur Namensgebung: Der Name einer Ressource ist syntaktisch ein `Identifier`, wie er in den meisten Programmiersprachen üblich ist: Er darf Ziffern und Buchstaben enthalten, muß aber mit einem Buchstaben[6] beginnen. Anders der Name eines Teilbereiches einer Bitstruktur: Da dieser Name, syntaktisch ein `symbol`(siehe Abbildung 2.10), immer nur gemeinsam mit einem `Identifier` auftreten kann, ist hier größere Freiheit gegeben: Er darf auch mit einer Ziffer beginnen (`Component`) oder gänzlich nur aus einer Zahl bestehen. Diese Art der Benennung ist im Bereich der Firmwareprogrammierung auch durchaus üblich.

Liegt also über einem Register eine mittels `MAP` definierte Struktur, so kann man durch Dereferenzierung mit dem Namen des Teilbereiches den entsprechenden Registerausschnitt ansprechen. Wie funktioniert das nun, wenn bei verschachtelten `MAP`–Definitonen ein Registerteil nochmals strukturiert ist? Dazu eine Bemerkung zum Aufbau der `MAP` selbst: Eine `MAP` beginnt stets bei Bit 0 und endet mit dem höchsten Bit, das im Zuge der Teilbereiche angesprochen wurde, sodaß also alle definierten Teilbereiche in der `MAP` Platz haben. Ergo kann man eine `MAP`–Struktur ganz allgemein nur über ein Bitfeld legen, das mindestens genauso groß ist wie die `MAP` selbst. Wird nun ein ganzes Register

[6] Auch der Unterstrich _ wird hier zu den Buchstaben gezählt.

```
MAP wbytes {
    <#8> 0,1,2,3 MAP bdigits{
                      <#4> h,l MAP {
                                       <#1> ax,ay,bx,by;
                                   };
                  };
};

MAP wdigits {
    <0:3> 0h,0l,1h,1l,2h,2l,3h,3l ;
};

MAP word OF wbytes, wdigits;
```

Beispiel 2.5: Fortgeschrittenere Anwendungen der MAP–Definition: Die `MAP wbytes` ist insgesamt 32 Bits lang und besteht aus 4 Teilbereichen der Länge 8. Der erste Teilbereich 0:7 heißt `0`, der zweite Teilbereich 8:15 heißt `1`, usw.... Jeder dieser Teilbereiche ist seinerseits mit der `MAP bdigits` strukturiert, die an dieser Stelle zugleich definiert wird: Die Bits 0 bis 3 von jedem der 8 Bit langen Teilbereiche der Hauptstruktur können mit dem Namen `h` weiter dereferenziert werden, die Bits 4 bis 7 jeweils mit `l`. Diese 4 Bit langen Bereiche, es gibt insgesamt 8 davon in der `MAP wbytes`, sind nun wiederum weiter strukturiert: Sie bestehen aus vier Bereichen zu je 1 Bit mit den Namen `ax`, `ay`, `bx` und `by` in dieser Reihenfolge. Diese Struktur besitzt aber keinen Namen, sie kann also sonst nirgends verwendet werden. Die `MAP wdigits` ist ebenfalls 32 Bit breit und besteht aus den 8 Bereichen `0h`, `0l`, `1h`, `1l`, `2h`, `2l`, `3h` und `3l`, die jeweils 4 Bit lang sind. Die `MAP word` schließlich stellt die Zusammenfassung der beiden Strukturen dar.

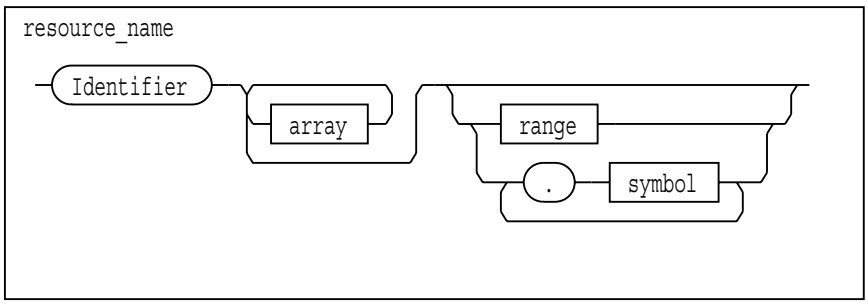


Abbildung 2.9: Dieses Syntaxdiagramm zeigt, wie eine bereits definierte Ressource als ganzes oder auch nur teilweise angesprochen werden kann: Der Name allein spricht die Ressource als ganzes an, so wie sie definiert wurde. Sofern es sich um ein mehrdimensionales Registerfeld handelt, können einzelne Felder oder Feldbereiche mit `array` angesprochen werden. Schließlich kann ein Bitausschnitt mittels `range` angegeben werden. Alternativ dazu kann der Bitteilbereich auch mit Hilfe der definierten MAP–Strukturen angesprochen werden, indem der gewünschte Teilbereich mit seinem in der MAP–Definition festgelegten Namen angesprochen wird. Dieser Name ist vom Namen der Ressource durch einen Punkt getrennt. Sofern verschachtelte MAP–Strukturen definiert wurden, sind auch mehrfache Dereferenzierungen möglich.

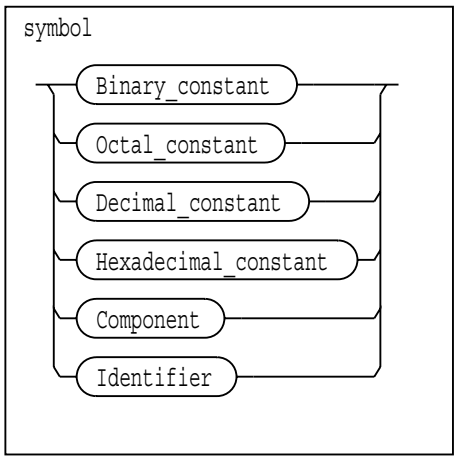


Abbildung 2.10: Die Syntax der Bezeichnung eines Bereiches in einer MAP–Definition.

strukturiert, das seinerseits mit Bit 0 beginnt, so sprechen die Bereichsdefinitionen der MAP–Struktur direkt die Bits des Bitfeldes an. Wenn hingegen ein Register*teil* strukturiert wird, der ja bei einem anderen Bit beginnt, so beschreiben die Teilbereiche der MAP stets die Bits *relativ* zum Anfang des Register*teiles*. Beginnt etwa der Registerteil mit Bit 8, so korrespondiert das Bit 0 der MAP mit dem Bit 8 im Registerteil, welches ja gerade das Bit 0 *relativ* zum Anfang des strukturierten Registerteiles ist. Im Normalfall ist das aber genau das, was man möchte: Egal welche Nummern die Bits in einem Registerteil tragen, die MAP–Struktur bezieht sich immer auf den Anfang des Registerteiles. Beispiel 2.6 zeigt einige Dereferenzierungen, die an die vorangegangenen Beispiele anknüpfen.

Die intensive Verwendung von MAP–Strukturen und Dereferenzierungen an Stelle der Angabe von expliziten Bitfeldbereichen hat zwei Vorteile: Erstens wird die MDL–Beschreibung dadurch verständlicher und leichter lesbar, da Namen nun einmal aussagekräftiger sind als Zahlenangaben. Zum zweiten wird auch die Änderung einer MDL–Beschreibung einfacher und damit weniger fehleranfällig: Die Korrektur an einer einzigen Stelle genügt, um die Änderung in der gesamten MDL–Beschreibung wirksam werden zu lassen!

Felder

Das Mikrobefehlsfeld ist eine besondere Ressource, die es genau einmal gibt: Jede Mikrooperation benötigt zur Codierung bestimmte Teile des gesamten Mikrobefehlsfeldes. Um diese Teile namentlich ansprechen zu können, gibt es die Möglichkeit der Feld–Definition: Mit dem Schlüsselwort FIELD wird einer Menge von Bits des Mikrobefehlsfeldes ein eindeutiger Name zugeordnet. Abbildung 2.11 zeigt die zu beachtende Syntax, in Beispiel 2.7 sind einige Beispiele angeführt. Auch bei der FIELD–Definition können, ebenso wie bei der REGISTER–Definition, unbestimmte Bereiche verwendet werden. Dabei gelten aber andere implizite Annahmen als beim Register, die in der Besonderheit des Mikrobefehlsfeldes begründet sind: Wird bei einem Feld nur die Länge angegeben, so wird ein Feld

```
io_port

portreg                 portreg<0:15>
portreg.digit           portreg<6:8>
portreg.port            portreg<12:15>
portreg.port.high       portreg<12:13>

reg
reg[0]
reg[0][2:5]             reg[0][2:5]<0:31>
reg[0][2].2             reg[0][2]<16:23>
reg[0][2].2.h           reg[0][2]<16:19>
reg[0][2].2h            reg[0][2]<16:19>
reg[0][2].2.h.ay        reg[0][2]<17>
```

Beispiel 2.6: Die Dereferenzierungen in diesem Beispiel verwenden die in den vorangegangenen Beispielen getätigten Definitionen: In der ersten Zeile wird einfach das Register `io_port` angesprochen, genauer gesagt, eines der beiden Register. Welches tatsächlich verwendet wird, entscheidet die Hardware selbst und entzieht sich dem Einfluß der Firmware. Der zweite Block beschreibt einige Verwendungen des Registers `portreg`, welches in Beispiel 2.2 definiert und dessen Bitstruktur in Beispiel 2.4 vorgestellt wurde. Links ist die Dereferenzierung mittels Bereichsnamen vorgestellt, rechts wird jeweils eine völlig äquivalente Ansprache der Registerteile unter Verwendung der direkten Bereichsangabe gezeigt. Besonders beachtenswert ist die letzte Zeile mit `portreg.port.high`. Man erkennt, wie die `MAP portmap` den 4 Bit breiten Bereich `portreg.port` weiter strukturiert: `high` beschreibt den Bereich 0:1 einer 4 Bit breiten Struktur, das ergibt in diesem Fall die Bits 12:13 des Registers insgesamt, da der strukturierte Teilbereich mit Bit 12 beginnt. Der letzte Block zeigt einige Ansprachen des Registerfeldes `reg`, dessen `MAP`–Struktur in Beispiel 2.5 beschrieben wurde. Bei der `array`–Angabe ist zu beachten, daß nicht nur einzelne Indizes, sondern ganze Feldteile beschrieben werden können: Die erste Zeile beschreibt alle 64 Register des Feldes, die zweite Zeile die 16 Register der Reihe 0 und die dritte Zeile die Register 2 bis 5 der Reihe 0, jeweils über die volle Bitbreite. Die weiteren Zeilen vergleichen wieder Dereferenzierung und Bereichsangaben. Noch eine Bemerkung zur letzten Zeile: Obwohl `reg[0][2].2.h` und `reg[0][2].2h` exakt denselben Registerausschnitt beschreiben, kann nur `reg[0][2].2.h` weiter dereferenziert werden, da für den Bereichsnamen `2h` keine weitere `MAP`–Struktur definiert wurde!

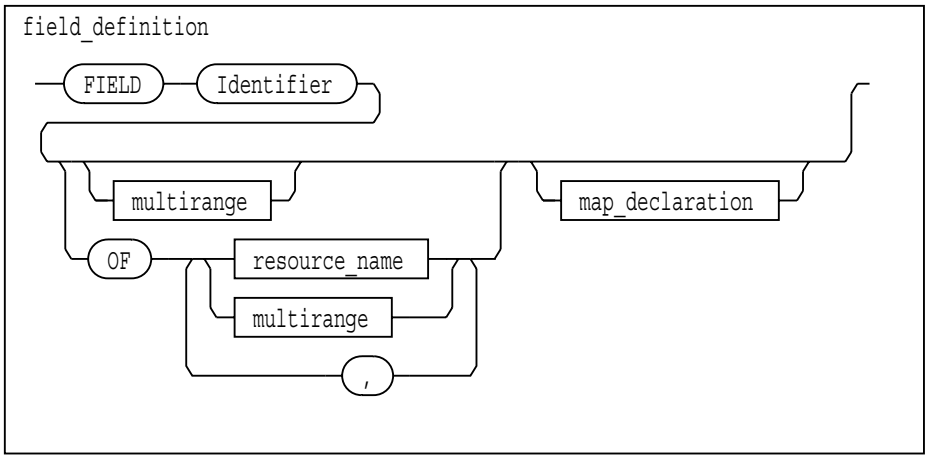


Abbildung 2.11: Das Syntaxdiagramm zur Definition von Feldern im Mikrobefehl: Die Bereichsangabe von der Registerdefinition wurde hier zum `multirange` erweitert. Außerdem gibt es die Möglichkeit, eine Felddefinition aus bestehenden Felddefinitionen und Bereichen neu zusammenzusetzen, dafür ist das Schlüsselwort `OF` verantwortlich. Auch eine Strukturierung mittels `map_declaration` ist möglich.

```
MAP fieldmap {<#3> high, low};

FIELD xyz<3,6>;
FIELD a<2:7> MAP fieldmap;
FIELD u<#6> MAP fieldmap;
FIELD v;  /* no mapping possible */

FIELD d<6:8; 4; 0:1> MAP fieldmap;
FIELD e<5:8; 1:2> MAP fieldmap;
```

Beispiel 2.7: Einige Varianten zur Felddefinition. Da nur Bitfelder mit einer Breite von mindestens 2 Bit sinnvoll mit einer `MAP`–Definition strukturiert werden können, ist eine solche bei `FIELD`–Definitionen mit völlig undefiniertem Bereich nicht möglich. Solche `FIELD`–Definitionen werden ja mit Länge 1 angenommen, wie das `FIELD v` in diesem Beispiel. Tabelle 2.2 beschreibt genau, welche Bits im Mikrobefehlsfeld von den obigen Definitionen angesprochen werden.

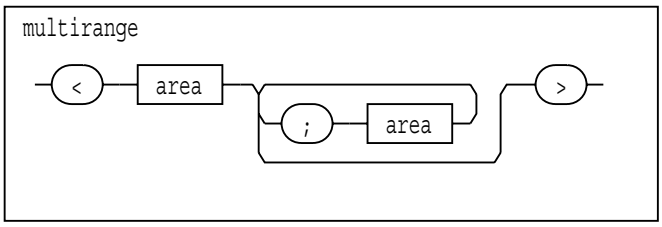


Abbildung 2.12: Die Syntax des Mehrfachbereiches.

Feld	Bits im Mikrobefehlsfeld						Gesamtlänge
xyz	3	4	5	6	7	8	6
a	2	3	4	5	6	7	6
b	u0	u1	u2	u3	u4	u5	6
c	v0						1
d	6	7	8	4	0	1	6
e	5	6	7	8	1	2	6

Tabelle 2.2: Erläuterung zu den Definitionen aus Beispiel 2.7: 'Bits im Mikrobefehlsfeld' bedeutet die Nummer des jeweiligen Bits im ursprünglichen, gesamten Mikrobefehlsfeld. Da die Felder `u` und `v` nicht im Mikrobefehlsfeld gelegen sind, kann bei ihnen nur die relative Nummerierung der Bits im Feldausschnitt angegeben werden.

dieser Länge aber außerhalb des Mikrobefehlsfeldes definiert. Der Effekt ist, daß dieses Feld keine Konflikte mit anderen Feldern hervorrufen kann, sehr wohl aber die notwendige Mindestlänge des Mikrobefehlsfeldes beeinflußt. Felder, bei denen außerdem keine Längenangabe erfolgt, erhalten die Länge 1 zugewiesen. Diese „Großzügigkeit“ der Sprache ist notwendig, da die MDL–Beschreibung bereits während des Entwurfes verwendet wird. Zu diesem Zeitpunkt sind aber viele Informationen noch nicht verfügbar, man weiß etwa noch nicht, wie lang das Feld sein wird oder wo im Mikrobefehlsfeld es disloziert sein wird. Eine korrekte MDL–Beschreibung soll aber auch mit möglichst wenig Information *korrekt* sein.

Auffallend ist bei der `FIELD`–Definition vor allem der `multirange`, ein Mehrfachbereich, der es ermöglicht, auch nicht zusammenhängenden Bitbereichen einen gemeinsamen Namen zu geben. Dieser besteht einfach aus einer Aufzählung von „normalen“ Bereichen (`range`), die durchaus ungeordnet und lückenhaft sein dürfen, sich aber nicht überlappen sollten. Abbildung 2.12 zeigt die Syntax des Mehrfachbereiches, Tabelle 2.2 zeigt die Bitbereiche, die von den Bereichsdefinitionen im Beispiel 2.7 definiert werden.

Jedes `FIELD` kann mit einer `MAP`–Struktur versehen sein, es gibt nur eine Einschränkung: Die verwendete `MAP` darf nicht größer sein, als das `FIELD` selbst. Als Besonderheit dürfen auch `FIELD`–Definitionen mit unbekanntem Offset mit einer `MAP`–Definition strukturiert werden. Feldteile können dann genauso beschrieben werden, wie Registerteile: Entweder durch direkte Angabe des Bitbereiches oder unter Verwendung einer `MAP`–Struktur. Dabei beziehen sich die Angabe des Bitbereiches oder die Dereferenzierung mit Teilbereichsnamen stets auf den Anfang des jeweiligen strukturierten Feldes und nicht

Feld	Bits im Mikrobefehlsfeld Bits im `FIELD` e					
e	5	6	7	8	1	2
	0	1	2	3	4	5
e.high	5	6	7			
	0	1	2			
e.low				8	1	2
				3	4	5
e<2:4>			7	8	1	
			2	3	4	

Tabelle 2.3: Beispiel für die Verwendung von Feldteilen: Die obere Zeile gibt jeweils die Bitnummer im Mikrobefehlsfeld an, die untere Zeile die relative Bitnummer im Feld e. Die `MAP fieldmap` bezieht sich hier auf den Anfang des Feldes e: Somit spricht der Teilbereich `high` die Bits 0:2 des Feldes e an, das sind aber gerade die Bits 5:7 des Mikrobefehlsfeldes. Der Teilbereich `low` spricht ergo die Bits 8, 1 und 2 des ursprünglichen Mikrobefehlsfeldes an. Analog werden mit `e<2:4>` die Bits 7, 8 und 1 des Mikrobefehlsfeldes angesprochen. Man erkennt dabei den großen Vorteil dieser Methode: Durch geeignete Definition können logisch zusammengehörige Felder, die im Mikrobefehlsfeld irgendwie verteilt sind, geschlossen angesprochen werden. Der Anwender erspart sich die mühselige 'Bitbitzlerei'.

auf die ursprüngliche Bezeichnung der Bits im Mikrobefehlsfeld. Tabelle 2.3 illustriert das an Hand eines Beispiels.

Alternativ kann eine Felddefinition mit dem Schlüsselwort `OF` auch als Zusammensetzung bereits bestehender Felder oder Feldteile (`resource_name`) und zusätzlicher expliziter Mehrfachbereiche definiert werden. Insgesamt entsteht dadurch wieder ein Mehrfachbereich, der keine Überlappungen enthalten darf. Auch `FIELD`–Definitionen mit undefiniertem Bereich dürfen ganz oder mit Teilen in anderen `FIELD`–Definitionen verwendet werden. Beispiel 2.8 zeigt die volle Mächtigkeit der Felddefinitionen, Tabelle 2.4 zeigt die Bitbereiche des ursprünglichen Mikrobefehlsfeldes, die dadurch definiert werden.

Die Definition von Feldern mit einer Längenangabe alleine wurde als Lösung vorgestellt für den Fall, daß man zwar die Länge des Feldes kennt, nicht aber seine genaue Lage. Was aber, wenn man von zwei Feldern nur die Länge und nicht die genaue Lage kennt, sehr wohl aber die relative Lage der beiden Felder zueinander bekannt ist? Für diesen Fall definiert man sich einfach mittels `#define` einen gemeinsamen Nullpunkt, der weit außerhalb des Mikrobefehlsfeldes liegt, etwa `#define Refpoint 1000` und definiert die beiden Felder exakt mit Offset und Länge unter Verwendung des definierten Referenzpunktes. Dann werden zwar Konflikte zwischen diesen beiden Feldern erkannt, es können aber keine Konflikte zu anderen Feldern entstehen. Diese Methode wird später bei den

```
FIELD f OF a, v;

FIELD g OF a.high, <8:9;0> MAP fieldmap;

FIELD k OF <1:2>, a.low, u.high;

FIELD l OF e<4:5>, e.high, u.high;
```

Beispiel 2.8: 'Fortgeschrittenere' Anwendung der `FIELD`–Definition: Die dadurch beschriebenen Ausschnitte des Mikrobefehlsfeldes sind in Tabelle 2.4 zusammengefaßt. Auch die Felder `k` und `l` könnten mit einer `MAP`–Struktur versehen werden, obwohl sie Teile undefinierter Bereiche enthalten.

Feld	Bits im Mikrobefehlsfeld								Gesamtlänge
f	2	3	4	5	6	7	v0		7
g	2	3	4	8	9	0			6
k	1	2	5	6	7	u0	u1	u2	8
l	1	2	5	6	7	u0	u1	u2	8

Tabelle 2.4: Bits des ursprünglichen Mikrobefehlsfeldes, wie sie von den `FIELD`–Definitionen in Beispiel 2.8 definiert werden: Bemerkenswert ist hier vor allem, daß `FIELD k` und `FIELD l` exakt denselben Ausschnitt des Mikrobefehlsfeldes beschreiben, was aufgrund der Definition nicht auf den ersten Blick abzulesen ist. Das aber ist ja gerade der Vorteil dieser Methode: Obwohl die zwei Felder genau denselben Ausschnitt aus dem Mikrobefehlsfeld beschreiben, ist die unterschiedliche *logische* Zuordnung der beiden Felder durch die Beschreibung mit Namen sofort zu erkennen. Bei einer reinen Beschreibung mit Bitbereichen wäre nur mühsam aus dem Zusammenhang zu erkennen, um welches *logische* Feld es sich dabei handelt. Soll nun eines der beiden Felder geändert werden, ist die Gefahr der Verwechslung sehr groß. Bei der Beschreibung mit Namen hingegen gibt es diese Gefahr nicht!

```
GROUP unit1 { reg[3][8:9], reg[0:1][5] };

SET pointer { reg[0][2], reg[0][3] };

GROUP unit2 { pointer, align, alu, c_bus, unit1 };
```

Beispiel 2.9: Beispiele für Gruppierungen. Diese Definitionen verwenden die DEVICE– und REGISTER–Definitionen vorangegangener Beispiele. Die erste Definition faßt die Register reg[3][8], reg[3][9], reg[0][5] und reg[1][5] unter dem Namen unit1 zusammen. Das SET pointer besteht nur aus zwei Registern. Die dritte Zeile verwendet die Möglichkeit, Gruppierungen hierarchisch zu strukturieren: Die GROUP unit2 besteht insgesamt aus den 4 Grundelementen align, alu und c_bus(existiert doppelt!), sowie den sechs Registern reg[3][8], reg[3][9], reg[0][2], reg[0][3], reg[0][5] und reg[1][5], insgesamt also aus 10 Ressourcen.

Zeitbereichen nochmals verwendet.

Auf den ersten Blick mögen die Möglichkeiten der FIELD–Definition etwas verwirrend erscheinen. In der Praxis helfen sie aber dem Anwender, Verwirrung zu vermeiden: Felder können nach ihrer *logischen* Zusammengehörigkeit hierarchisch und strukturiert definiert werden. Einmal festgelegt, muß sich der Anwender dann nicht mehr um die eigentliche Verteilung der Bits im Mikrobefehl kümmern. Mehr noch: Die früher aufgrund ihrer Redundanz höchst fehleranfällige Änderung einer solchen Verteilung wird jetzt mühelos und sicher an einer einzigen Stelle der Beschreibung vorgenommen. Dies ist vor allem dann eine große Erleichterung, wenn Felder aufgrund spezieller Hardwareerfordernisse im Mikrobefehlsfeld nicht als geschlossene Bitfolge auftreten. Aber auch wenn das nicht der Fall sein sollte, ist es einfacher und somit weniger fehleranfällig, sich einen Feldnamen zu merken und logisch zuzuordnen, als den reinen Bitbereich zu verwenden. Da in weiterer Folge Konflikte zwischen überlappenden Feldbereichen automatisch erkannt werden und auch bei der Codierung die definierten Feldnamen verwendet werden, braucht sich der Anwender nach der einmaligen Definition um die tatsächlichen Bits im Mikrobefehlsfeld überhaupt nicht mehr zu kümmern!

Gruppierungen

Wenn eine bestimmte Gruppe von Ressourcen des öfteren gemeinsam Verwendung findet, ist es naheliegend, diese Ressourcen auch unter einem gemeinsamen Namen zusammenzufassen. Diese Zusammenfassung wird von der GROUP–Definition bewerkstelligt. Bei der SET–Definition handelt es sich um eine analoge Zusammenfassung, die aber ausschließlich Register oder Registerteile enthalten darf. Abbildung 2.13 zeigt die Syntaxdiagramme für diese Definitionen, Beispiel 2.9 enthält einige Anwendungsmöglichkeiten. Wie man sieht, dürfen Gruppierungen außer Registern und Grundelementen auch wiederum Gruppierungen enthalten. Dies gilt für die GROUP- und SET–Definition gleichermaßen mit einer

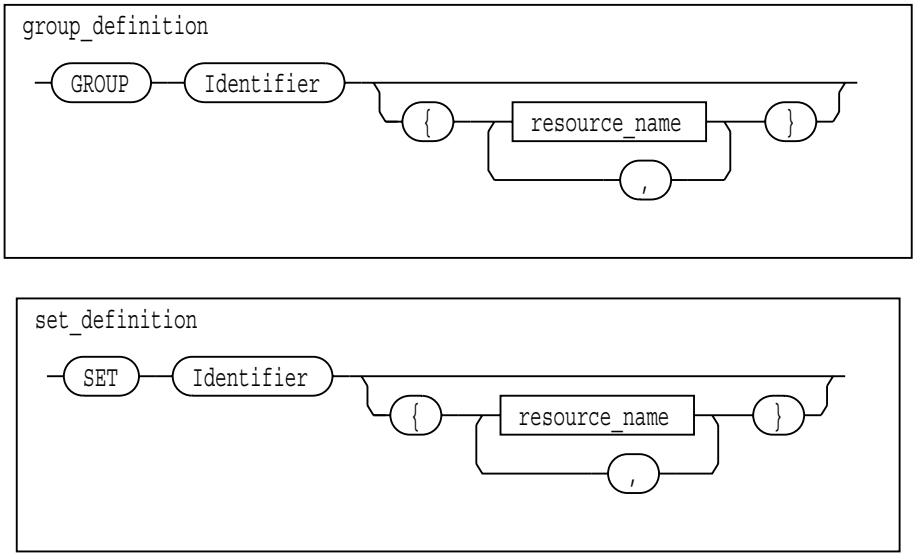


Abbildung 2.13: Syntax für Gruppierungen: Die zusammenzufassenden Ressourcen oder Ressourcenteile werden mittels `resource_name` angesprochen und mit einem gemeinsamen `Identifier` als Gruppierung gekennzeichnet.

Einschränkung: Löst man alle inneren Gruppierungen in ihre Elemente auf, dürfen im `SET` letztendlich nur Register enthalten sein. Das bedeutet aber nicht, daß in einem `SET` keine `GROUP` enthalten sein darf: Wenn diejenige `GROUP` nur Register enthält, wie die `GROUP unit1` im Beispiel, gibt es keine Probleme.

Verbindungen

Da die Verbindungen zwischen den Hardwareelementen Basis für den von den Mikrooperationen verursachten *Datenfluß* sind, kommt ihnen ein durchaus hoher Stellenwert zu. Dennoch wurde für die Eingabe der Verbindungen nur ein relativ einfacher Mechanismus vorgesehen. Der Grund dafür ist, daß in der Praxis für die Entwicklung der Hardware graphische Oberflächen zur Verfügung stehen, auf denen die Verbindungen sehr komfortabel definiert werden können. Es kann daher davon ausgegangen werden, daß die Verbindungsdeklaration für die MDL–Beschreibung einer solchen graphischen Oberfläche entnommen, also generiert werden kann. In weiterer Folge könnte sogar der gesamte Deklarationsteil der MDL–Beschreibung aus einer graphischen Oberfläche heraus generiert werden, wenn zu den Hardwareelementen die für die MDL–Beschreibung benötigten Attribute direkt im Rahmen der graphischen Oberfläche eingegeben werden könnten.

Im Rahmen der MDL werden Verbindungen mit dem Schlüsselwort `PATH` definiert. Abbildung 2.14 zeigt die Möglichkeiten der Syntax, Beispiel 2.10 liefert einige Muster dazu. Verbindungen können gerichtet definiert werden, dann gibt der angedeutete Pfeil die Richtung der Verbindung an. Ungerichtete Verbindungen werden wie ein Paar antiparallel gerichteter Verbindungen behandelt. Wie immer gilt auch hier, daß auf Ressourcen erst nach ihrer Definition Bezug genommen werden kann: Daher müssen alle Ressourcen definiert sein, bevor sie in einer Verbindungsdefinition angesprochen werden können.

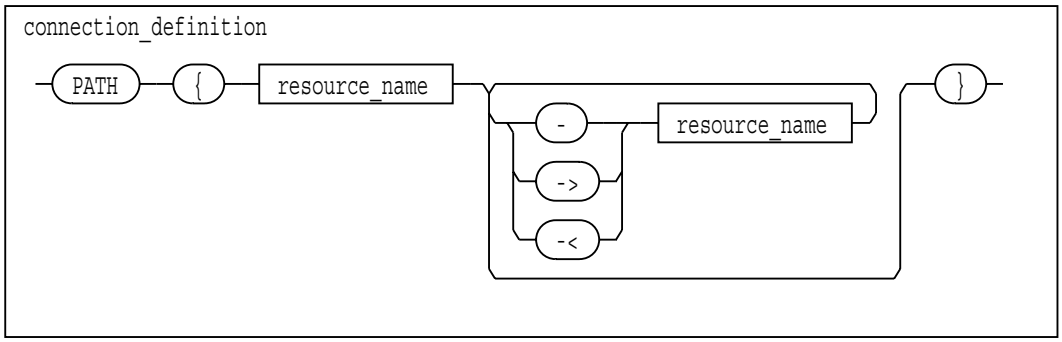


Abbildung 2.14: Die Syntax der Verbindungsdefinition: Die zu verbindenden Ressourcen oder Ressourcenteile werden syntaktisch wieder als `resource_name` angesprochen. Ungerichtete Verbindungen werden in der Form `a-b` definiert, gerichtete Verbindungen mit einem Pfeil: `a->b` bedeutet eine gerichtete Verbindung von `a` nach `b`, `a-<b` eine gerichtete Verbindung in der umgekehrten Richtung.

```
DEVICE a_bus;
DEVICE b_bus;

PATH { reg -> a_bus -> alu -> b_bus -> reg };
PATH { pointer - portreg };
PATH { unit1 -> c_bus -< portreg };
```

Beispiel 2.10: Einige Verbindungsdefinitionen: Die erste `PATH`–Definition definiert 130 gerichtete Verbindungen, nämlich 64 Verbindungen von allen Registern des Registerfeldes `reg` zu `a_bus`, eine Verbindung von `a_bus` nach `alu`, eine Verbindung von `alu` nach `b_bus` und wiederum 64 Verbindungen von `b_bus` zurück ins Registerfeld `reg`. Die zweite Zeile definiert zwei ungerichtete Verbindungen, die wiederum vier gerichteten Verbindungen entsprechen: Von `reg[0][2]` nach `portreg` und zurück sowie von `reg[0][3]` nach `portreg` und zurück. Die letzte Zeile definiert völlig analog insgesamt 10 einzelne gerichtete Verbindungen: `unit1` enthält 4 Elemente, das ergibt insgesamt 8 Verbindungen zu den zwei Elementen, die durch `c_bus` beschrieben werden (siehe Beispiel 2.1). Dazu kommen noch die zwei Verbindungen vom `portreg` zum `c_bus`.

Der Begriff „Verbindung“, wie er hier verwendet wird, ist aber nicht etwa geeignet, um einen Systembus zu beschreiben: Busverbindungen oder andere Datenwege sind vielmehr als `DEVICE` zu definieren, da sie ja gerade besonders interessant für die Konflikterkennung sind. Eine Verbindung ist vielmehr im Sinne einer Lötverbindung zu verstehen, etwa eine Verbindung zwischen einem Registerport und einem Bus oder zwischen Bus und Multiplexer. Sinn der Verbindungsdefinition ist es, später überprüfen zu können, ob ein beabsichtigter Datenfluß auch tatsächlich möglich ist.

Werden im Zuge der Verbindungsdefinition Gruppierungen verwendet, so werden die entsprechenden Verbindungen für alle Elemente der Gruppierung eingerichtet. Dabei wird aber nicht überprüft, ob für ein Element eine Verbindung zu sich selbst eingerichtet wird, was bei dem hier verwendeten Verbindungsbegriff ja sinnlos ist. Geschickt eingesetzt können Gruppierungen jedoch die „händische“ Eingabe der Verbindungen einer Prozessorstruktur maßgeblich erleichtern. Auch Arrays sind in diesem Sinne als Gruppierungen aufzufassen.

2.3 Mikrooperation

Jede Mikrooperation entspricht einer festverdrahteten Hardwarefunktion, gesteuert von bestimmten Bits im Mikrobefehlsfeld. Das Mikrobefehlsfeld insgesamt steuert mit seinem Bitmuster im allgemeinen mehrere Mikrooperationen. Diese bilden gemeinsam einen Mikrobefehl. Um die Konflikte bei der Zusammensetzung von Mikrooperationen zu einem Mikrobefehl erkennen zu können, muß jede Mikrooperation entsprechend beschrieben werden. Dies geschieht mit dem Schlüsselwort `MICROP`, die Syntax ist in Abbildung 2.15 dargestellt. Folgende Punkte können bei der Beschreibung einer Mikrooperation angegeben werden:

Mikrobefehlsfeld: Welche Teile des Mikrobefehlsfeldes eine bestimmte Mikrooperation für sich beansprucht.

Ressourcennutzung: In welchen Zeiträumen welche Hardwareressourcen von der Mikrooperation belegt sind.

Datenfluß: Über welche Pfade der Datenfluß läuft.

Bedingungen: Welche zusätzlichen Bedingungen eingehalten werden müssen.

Darüberhinaus kann die Angabe von *Ressourcenvariablen* notwendig sein, wenn die Ressourcennutzung syntaxabhängig ist. Ein komplettes Beispiel einer `MICROP`–Definition wird erst in Abbildung 2.16 auf Seite 53 vorgestellt, zuvor werden die einzelnen Bestandteile einer solchen Definition genauer beschrieben.

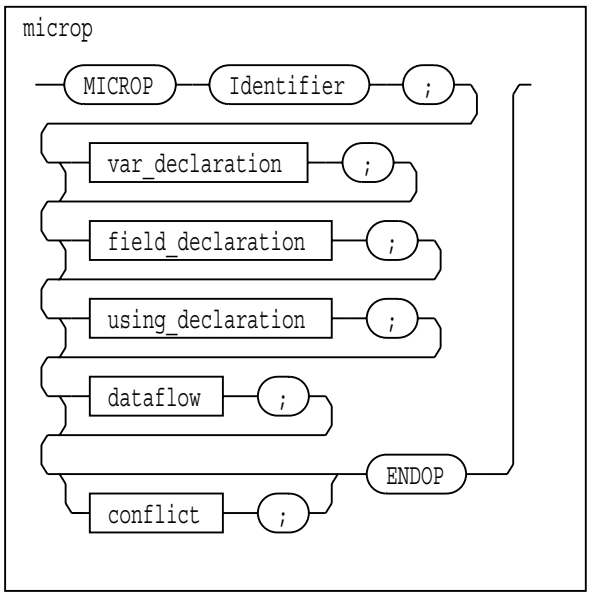


Abbildung 2.15: Die Syntax der Mikrooperation: Die einfachste Beschreibung einer Mikrooperation besteht nur aus der Angabe ihres Namens, alle anderen Punkte sind optional. Sie können voneinander unabhängig definiert oder weggelassen werden. Dies ist notwendig, um die erwähnte Großzügigkeit der Sprache zu gewährleisten, die für ein Entwurfswerkzeug unerläßlich ist.

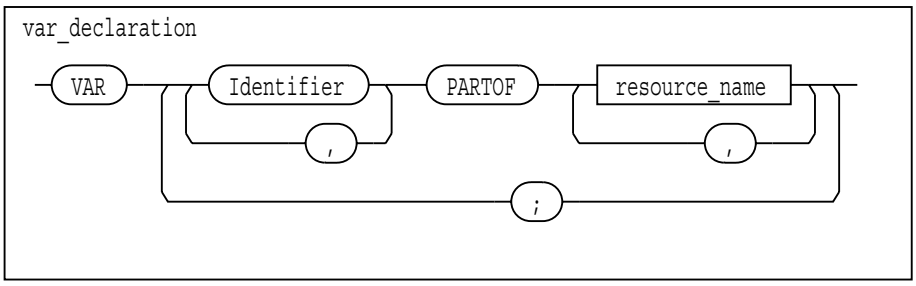


Abbildung 2.16: Die Syntax der Variablendeklaration: Eine Ressourcenmenge wird durch Angabe der einzelnen Ressourcen definiert, syntaktisch als `resource_name`–Aufzählung. Zu jeder derartigen Menge können mehrere `Identifier` für Variablen definiert werden.

Ressourcenvariable

Die Variablendeklaration dient mit dem Schlüsselwort `VAR` der Beschreibung syntaxgesteuerter Ressourcennutzungen. Das bedeutet, daß erst der Compiler entscheiden kann, welche Ressource aus einer bestimmten Menge möglicher Ressourcen nun tatsächlich benutzt wird. Vom Standpunkt der Konflikterkennung bedeutet das einen Konflikt 2.Art. Zu diesem Zweck definiert die Variablendeklaration eine Ressourcenvariable: Das ist ein Platzhalter für eine Ressource und steht für genau ein Element aus der Menge jener Ressourcen, die an der jeweiligen Stelle in Frage kommen. In Abbildung 2.16 ist das Syntaxdiagramm dargestellt, Beispiel 2.11 zeigt die Möglichkeiten.

```
VAR     p1,p2 PARTOF pointer;
        reg1, reg2 PARTOF reg;
        op1 PARTOF unit1, portreg;
        op2 PARTOF op1, portreg;
        op3 PARTOF a_bus, b_bus;
```

Beispiel 2.11: Einige Variablendeklarationen: Die erste Zeile definiert zwei Variablen `p1` und `p2`, die beide je genau ein Element aus dem `SET pointer` beschreiben. Die beiden Variablen `reg1` und `reg2` stehen jeweils für ein Register aus den 64 Registern des Registerfeldes `reg`. Die Variable `op1` repräsentiert ein Register aus der Auswahl: `reg[3][8]`, `reg[3][9]`. `reg[1][5]` und `reg[1][5]` von `unit1`(siehe Beispiel 2.9) sowie `portreg`. Zwar darf eine Variable sowohl für ein Grundelement als auch für ein Register stehen, die Mischung von Registern und Grundelementen ist allerdings nicht zulässig! Auch die nächste Variable `op2` beschreibt genau eine Ressource: Entweder das Register `portreg`, oder exakt jene Ressource, die von der Variablen `op1` beschrieben wird. Die letzte Variable schließlich ist ein Beispiel für eine Variable von Grundelementen: Sie kann entweder `a_bus` oder `b_bus` repräsentieren.

Die Syntax ist so ausgelegt, daß mehrere Variablen zu einer bestimmten Ressourcenmenge angegeben werden dürfen. Das Schlüsselwort `PARTOF` trennt die Angabe der Variablen–Bezeichner von der Deklaration der Ressourcenmenge. Die Ressourcenmenge wiederum besteht aus der Angabe von Registern, Grundelementen, Gruppierungen und sogar Ressourcenvariablen selbst, sofern sie bereits zuvor definiert wurden. Die Angabe einer Ressourcenmenge ist nur sinnvoll, wenn tatsächlich mindestens zwei Ressourcen beschrieben werden. Das bedeutet aber nicht, daß unbedingt zwei `resource_name`–Angaben erforderlich sind: Bereits die Angabe eines Registerfeldes oder einer Gruppierung allein reicht aus, um eine Ressourcenmenge zu definieren. Außerdem muß eine Ressourcenvariable „reinrassig“ sein, sie darf also nach Auflösung aller Gruppierungen und anderen Variablen entweder nur Register oder nur Grundelemente enthalten. Diese Menge der möglichen Ressourcen wird in weiterer Folge als *Trägermenge* der zugehörigen Ressourcenvariable bezeichnet.

Die Angabe der Ressourcenvariablen ist aus zweierlei Sicht sinnvoll: Der Konflikterkennung wird damit mitgeteilt, daß nur irgendeine Ressource aus einer Menge benötigt wird, nicht aber die Menge insgesamt. Dies führt eben genau zu den Konflikten zweiter Art. Noch wichtiger ist diese Angabe für die Syntax/Code–Zuordnung, ergo für die Generierung des Compilers: Jede Ressourcevariable wird im fertigen Compiler mit der Ressource assoziiert, die von dem konkreten Firmwareprogramm beschrieben wird. Damit wird der Platzhalter konkretisiert und es kann die Konfliktprüfung erfolgen. Durch diese Zuweisung einer konkreten Ressource an die Ressourcevariable werden also die Konflikte 2.Art im Compiler abgehandelt.

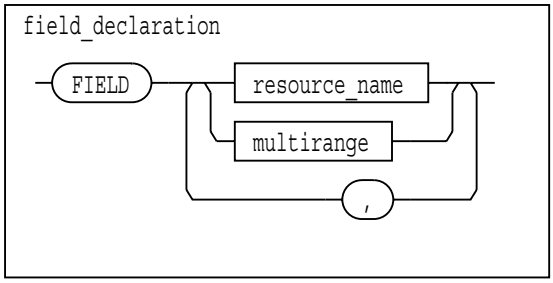


Abbildung 2.17: Die Syntax der Felddeklaration entspricht der Syntax, die bei der Zusammensetzung von Feldern zu einem neuen Feld im Rahmen der FIELD–Definition vorgestellt wurde (siehe Abbildung 2.11).

```
FIELD l, v, g.low, xyz<0:1>, <10:12;#4;14>;
```

Beispiel 2.12: Hier sind einige mögliche Felddeklarationen im Mikrobefehl angeführt: Die Feldnamen beziehen sich auf die Definitionen aus den Beispielen 2.7 und 2.8. Insgesamt beansprucht die Mikrooperation mit der obigen Felddeklaration die Bits 0 bis 12 und das Bit 14 des Mikrobefehlsfeldes fix: Das Feld `l` beschreibt die Bits 1 und 2 sowie 5 bis 7. Der Bereich `g.low` deklariert die Bits 8, 9 und 0, der Bereich `xyz<0:1>` schließlich die Bits 3 und 4. Ergänzt werden diese Bits durch den explizit angegebenen Mehrfachbereich. Außerdem werden die Bereiche `v0`, `u0` bis `u2` (als Bestandteil des Feldes `l`) sowie ein weiterer 4 Bit langer anonymer Bereich beansprucht. Diese Mikrooperation benötigt also eine minimale Befehlsbreite von 22 Bits für sich alleine.

Mikrobefehlsfeld

Obwohl syntaktisch relativ einfach (siehe Abbildung 2.17), ist die Felddeklaration ein zentraler Punkt bei der Beschreibung einer Mikrooperation: Durch Nennung von bereits definierten Feldern oder Feldteilen und durch explizite Angabe von Mehrfachbereichen wird eine Menge von Bits aus dem Mikrobefehlsfeld definiert. Dabei sollte es innerhalb der Felddeklaration einer Mikrooperation zu keinerlei Überschneidungen kommen. Genau diese definierten Bits sind es nämlich, die die Mikrooperation im Mikrobefehlsfeld für sich beansprucht. Und genau diese Bits sind es auch, die im Rahmen der Syntax/Code–Zuordnung mit einem konkreten Bitcode belegt werden dürfen. Beispiel 2.12 zeigt die Möglichkeiten der Felddeklaration an einem Beispiel.

Dabei sind selbstverständlich auch teilweise oder ganz undefinierte Bereiche zugelassen. Für die Konflikterkennung wirkt sich das wie folgt aus:

- Bereiche, die mit Offset und Länge definiert sind, wirken sich nicht nur auf die minimale Feldbreite des gesamten Mikrobefehles aus, sondern können auch auf Überlappungen geprüft werden.
- Felder, deren Länge zwar bekannt ist, deren Offset aber noch nicht feststeht, können

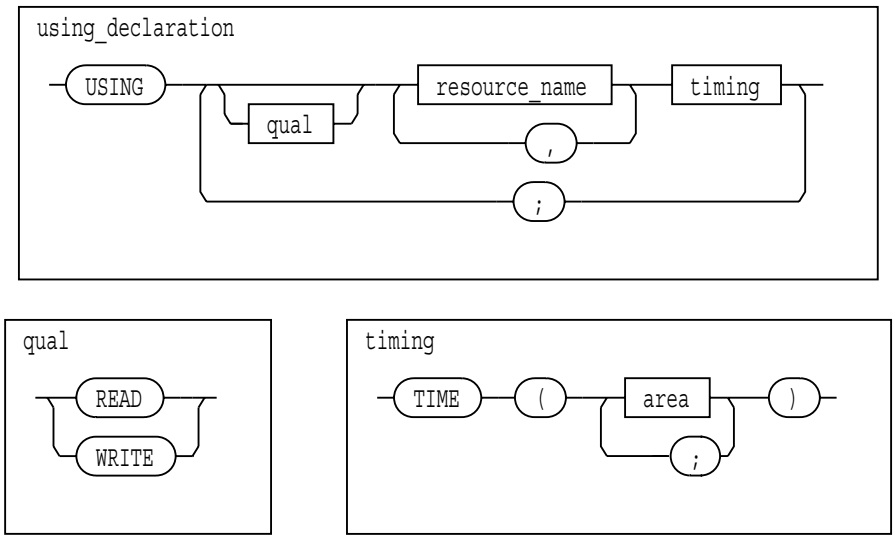


Abbildung 2.18: Die Nutzung der Ressourcen durch eine Mikrooperation wird mit obiger Syntax beschrieben: Die benutzten Ressourcen werden mittels `resource_name` angegeben. `qual` ist nur bei Registern sinnvoll und unterscheidet zwischen Lese- und Schreib–Zugriffen. Das `timing` schließlich enthält einen oder mehrere Zeitbereiche in einer an den `multirange` angelehnten syntaktischen Form.

nur bei Überlappung mit sich selbst einen Konflikt hervorrufen. Sehr wohl aber geht ihre Länge in die minimale Feldbreite des Mikrobefehlsfeldes ein.

- Felder, die nur dem Namen nach ohne Länge oder Offset definiert wurden, können ebenfalls nur bei Überlappung mit sich selbst einen Konflikt hervorrufen. Zur minimalen Feldbreite des Mikrobefehlsfeldes können sie mit Länge 1 beitragen, da ein Bereich wohl keine kleinere Breite besitzen kann.

Diese Zulassung undefinierter Feldbereiche bezieht sich nicht nur auf die Möglichkeit der Konflikterkennung. Um aus der MDL–Beschreibung einen Firmwarecompiler zu generieren, ist es noch nicht unbedingt notwendig, alle Feldbereiche exakt festzulegen. Erst wenn der durch den Compiler erzeugte Bitcode einer Simulation oder einem EPROM zugeführt werden soll, ist es nötig, die Feldbereiche exakt festzulegen.

Ressourcennutzung

Die Nutzung der Hardwareressourcen durch die Mikrooperation ist die Ursache für mannigfaltige Konflikte, sie steht daher im Mittelpunkt der Beschreibung der Mikrooperation. Hinter dem Schlüsselwort `USING` wird aufgezählt, welche *Ressourcen* in welchen *Zeiträumen* auf welche *Art und Weise* beschrieben werden. Die hiezu festgelegte Syntax zeigt Abbildung 2.18. In Beispiel 2.13 sind dazu einige Möglichkeiten aufgezeigt.

Jede Zeile einer `USING`–Deklaration beschreibt die Nutzung einer oder mehrerer Ressourcen. Diese Ressourcen werden in der syntaktischen Form eines `resource_name` aufgezählt und können Register, Grundelemente, Gruppierungen und Ressourcenvariablen

```
/* time granularity: 1 = 1ns */

#define T 25 /* clock cycle 25ns, clock frequency 40MHz */

#define T1 0,T
#define T2 T,T
#define T3 2*T,T
#define T4 3*T,T

#define Strobe 1000


USING
        READ p1,p2 TIME (0,12);
        READ reg1, reg2 TIME (T1);
        a_bus, b_bus TIME (T,10);
        alu TIME (T2;T3);
        op1, op2, op3 TIME (Strobe-5,Strobe+3*T);
        unit2 TIME (Strobe+T,4;Strobe+2*T,4);
        c_bus TIME (3*T,10);
        WRITE reg2 TIME (T4);
```

Beispiel 2.13: Hier werden die Möglichkeiten der `USING`–Deklaration vorgestellt. Vorneweg werden einige Definitionen zur Erleichterung der Zeitangaben getätigt: `T` steht für 25 dimensionslose Zeiteinheiten, die Kommentare geben dem erst einen Sinn: Durch sie wird `T` als Dauer des Taktzyklus mit 25ns festgelegt. Analog werden `T1` bis `T4` als vier Taktzyklen festgelegt, jeder mit der Länge `T`. Die `USING`–Deklaration beschreibt folgende Ressourcennutzungen: Die erste Zeile beschreibt einen Lesezugriff auf die beiden Register `p1` und `p2` im Zeitintervall 0 bis 12. Da es sich dabei um Ressourcenvariablen handelt, steht erst zum Zeitpunkt der Übersetzung eines Firmwareprogrammes im Compiler fest, um welche Register es sich tatsächlich handelt. Vorerst ist nur festgelegt, daß beide Variablen aus der Gruppierung `pointer` stammen müssen. Bei der zweiten Zeile handelt es sich analog um einen Lesezugriff auf zwei der 64 Register des Registerfeldes `reg` im Zeitintervall von 0 bis 25. Die dritte Zeile bedeutet die Beanspruchung der beiden Grundelemente `a_bus` und `b_bus` im Zeitintervall von 5 bis 15, die vierte Zeile eine analoge Beanspruchung der `alu` in den Zeitintervallen 25 bis 50 und 50 bis 75. Die nächsten beiden Zeilen bieten ein Beispiel für die Definition von Zeitpunkten weit außerhalb der Taktzyklen zum Zwecke des *relativen* Vergleichs von Ressourcennutzungen: Die obere Zeile deklariert die Nutzung der drei Ressourcenvariablen `op1`, `op2` und `op3` im Zeitintervall 995 bis 1075, die untere Zeile die Nutzung aller Ressourcen aus der Gruppierung `unit2` in den Zeitintervallen 1025 bis 1029 und 1050 bis 1054. Interessant ist noch die letzte Zeile, wo über die Ressourcenvariable `reg2` ein Schreibzugriff auf eines der 64 Register des Registerfeldes `reg` beschrieben wird.

umfassen. Jede Ressource wird dabei stets zur Gänze beansprucht. Daraus ergibt sich auch ein Rückschluß auf die sinnvolle Definition von Ressourcen: Ressourcen, die stets nur gemeinsam oder gar nicht benötigt werden, können als *eine* Ressource definiert werden, wenn dem sonst keinerlei Gründe entgegensprechen. Sollte sich aber herausstellen, daß eine Mikrooperation eine Ressource nur zum Teil beansprucht und es konfliktfrei möglich ist, daß eine andere Mikrooperation zur gleichen Zeit einen anderen Teil derselben Ressource beansprucht, so wurde diese Ressource ungünstig definiert: Sie sollte vielmehr in solche Ressourcen aufgespalten werden, die nur entweder ganz oder gar nicht benutzt werden können.

Drei Besonderheiten sind dabei allerdings zu beachten. Die erste betrifft die Ressourcenvariablen: Gemäß Definition der Ressourcenvariablen bedeutet die Nutzung einer Variablen die Nutzung genau einer Ressource aus der Trägermenge, die bei der Deklaration der Variablen angegeben wurde. Diese Ressource jedoch, die erst im Rahmen des Compilers konkretisiert werden kann, wird dann wiederum zur Gänze genutzt. Als zweite Besonderheit soll an die Möglichkeit erinnert werden, daß Ressourcen mehrfach bestehen, die Vergabe derselben aber unbeinflußbar ist: In Analogie zur Ressourcenvariablen bedeutet die Benutzung einer solchen Ressource stets die Benutzung genau eines Repräsentanten. Dieser wird aber wiederum zur Gänze beansprucht. Ein Konflikt entsteht erst dann, wenn die Ressource öfter benötigt wird, als sie vorhanden ist. Die dritte Besonderheit betrifft die Nutzung der Register: Da es Multiport–Register gibt, die mehrere Lese- und Schreibzugriffe zur selben Zeit zulassen, besteht ein Register aus der Sicht der Ressourcennutzung aus der entsprechenden Anzahl von Lese- und Schreib–Ports. Ein Konflikt entsteht erst, wenn die Anzahl der Lese- oder Schreibzugriffe die Anzahl der entsprechenden Ports übersteigt. Daher kann die Nutzung von Registern auch durch `READ` oder `WRITE` qualifiziert werden. Unterbleibt eine solche Qualifikation, so wird das Register zur Gänze beansprucht, egal wieviele Ports auch immer es besitzt.

Jede Zeile einer `USING`–Deklaration wird von einer Zeitangabe abgeschlossen, die den syntaktischen Regeln des Bitbereiches gehorcht (`area`): Jede Zeitangabe kann dabei aus mehreren disjunkten Zeitintervallen bestehen. Zeitangaben, also Intervallgrenzen und Intervallängen, sind dimensionslose ganze Zahlen. Diese Einschränkung stellt in der Praxis kein echtes Problem dar, da stets eine Zeitkörnigkeit gewählt werden kann, welche die Beschreibung aller Zeitangaben in Form von ganzen Zahlen gestattet. Die gewählte Zeitkörnigkeit sollte aber in Form eines Kommentars in der MDL–Beschreibung festgehalten werden. Sollte die Einschränkung der ganzen Zahlen aus irgendeinem Grund eine echte Beschränkung darstellen, bereitet auch die Erweiterung zu Fließkommazahlen kein grundsätzliches Problem.

Um die Zeitangaben einfacher und übersichtlicher handhaben zu können, empfiehlt sich der intensive Einsatz von Definitionen mittels `#define`. Damit wird die MDL–Beschreibung verständlicher und somit leichter wartbar. Darüber hinaus sollten die Bereiche einer Zeitangabe stets vollständig definiert werden, undefinierte Bereiche sind nicht zugelassen. Ist man sich noch nicht sicher, in welchem Zeitintervall eine Ressource benötigt wird, kann man sich Zeitpunkte und Intervalle definieren, die sicher weit außerhalb des betrachteten Zeitbereiches liegen. Damit sind Konflikte mit anderen Ressourcen ausgeschlossen. Diese Vorgangsweise ist auch dann sinnvoll, wenn man die Zeitverhält-

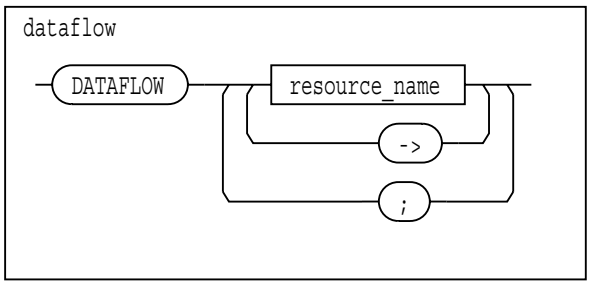


Abbildung 2.19: Die Syntax der Datenfluß–Deklaration entspricht einer Vereinfachung der Syntax der PATH–Definition.

```
DATAFLOW
        reg2 -> a_bus -> alu -> c_bus -> reg2;
        reg1 -> b_bus -> alu;
        op1 -> unit2 -> op2;
```

Beispiel 2.14: Eine mögliche Anwendung der Datenfluß–Deklaration: Die Variable reg2, eines der 64 Register des Registerfeldes reg, ist Ausgangspunkt der ersten Verbindung. Von ihr wird eine Verbindung zum a_bus, von dort weiter zur alu, zu den beiden Grundelementen c_bus und schließlich zurück zum Register reg2 benötigt. Interessant noch die dritte Zeile: Während op1 und op2 jeweils als Platzhalter für genau eine Ressource stehen, steht unit2 für alle Ressourcen, die in dieser Gruppierung enthalten sind. Es werden somit Verbindungen von op1 zu allen Ressourcen in unit2 und von diesen zu op2 benötigt. Dies sind insgesamt 20 Verbindungen, da die unit2 10 Ressourcen enthält (siehe Beispiel 2.9).

nisse einiger Ressourcennutzungen relativ zueinander aber unabhängig von den übrigen untersuchen will: Man definiert einen Zeitpunkt weit außerhalb des betrachteten Zeitbereiches der Taktzyklen und setzt die zu untersuchenden Ressourcennutzungen bezüglich dieses Zeitpunktes zueinander in Relation.

Datenfluß

Der Datenfluß ist das logische Gegenstück zur Verbindungsdefinition: Mit dem Schlüsselwort DATAFLOW dient er der Angabe der Verbindungen, die für den Ablauf der Mikrooperation erforderlich sind. In Abbildung 2.19 ist die Syntax angegeben, Beispiel 2.14 zeigt eine mögliche Anwendung. Der Datenfluß beschreibt die gerichteten Verbindungen, die notwendig sind, um die jeweilige Mikrooperation durchzuführen. Somit wird also der „Fluß“ der Daten über die Hardwareressourcen beschrieben. Es ist offensichtlich, daß jede im Datenfluß genannte Ressource auch im Rahmen der USING–Deklaration genannt werden muß. Die Umkehrung gilt nicht unbedingt, da Ressourcen benutzt werden können, die nicht im Datenfluß aufgezählt werden. Dies liegt an der relativ hardwarefernen Abstraktionsebene der MDL–Beschreibung. Wenn nun eine Verbindung benötigt wird, die

Operator	Verwendung	Bedingung
==	`var1 == var2`	`var1` muß identisch gleich `var2` sein
!=	`var1 != var2`	`var1` darf nicht gleich `var2` sein
<=	`var1 <= list`	`var1` muß in der Ressourcenmenge `list` enthalten sein
>=	`list >= var1`	äquivalent zu `var1 <= list`
<!	`var1 <! list`	`var1` darf nicht in der Ressourcenmenge `list` enthalten sein
>!	`list >! var1`	äquivalent zu `var1 <! list`

Tabelle 2.5: Die Operatoren, die bei der expliziten Bedingungsangabe Verwendung finden. Dabei bedeuten `var1` und `var2` irgendwelche Ressourcenvariablen und `list` eine Menge von Ressourcen.

nicht exisitiert, gibt es zwei Möglichkeiten: Entweder wird ein Fehler gemeldet oder die Verbindung wird mit einer Warnung eingerichtet. Die zweite Methode wird in weiterer Folge als automatische Datenpfadsynthese bezeichnet.

Ansonsten gelten bei der `DATAFLOW`–Beschreibung dieselben Regeln wie bei der `USING`–Deklaration: Verbindung zu einer Ressourcenvariablen bedeutet Verbindung zu genau einer Ressource aus der Menge der möglichen Ressourcen. Ebenso steht die Verbindung zu einer Gruppierung symbolisch als Verbindung zu allen einzelnen Ressourcen dieser Gruppierung. Dies gilt analog für Ressourcen, die mehrfach identisch existieren (`#`–Definition).

Bedingungen

Zusätzlich zur bisherigen Beschreibung können explizit angegebene Bedingungen für zusätzliche Konflikte sorgen: Die Angabe solcher Bedingungen geschieht nach der Syntax in Abbildung 2.20. Es handelt sich dabei um Bedingungen für Ressourcenvariablen. Jede Ressourcenvariable unterliegt aufgrund ihrer Definition bereits der Einschränkung, daß sie Element der Trägermenge sein muß. Unter bestimmten Umständen können aber zusätzliche Einschränkungen notwendig sein, welche die Variablen untereinander und in Bezug zu gewissen Ressourcenmengen in Relation setzen. Einzelne Bedingungen werden gemäß Tabelle 2.5 angegeben. Mehrere Bedingungen können mittels `&&` (UND), `||` (ODER) beziehungsweise `##` (EXKLUSIVES ODER) miteinander logisch verknüpft oder mittels `!` (NEGATION) invertiert werden. Die Klammerung zwischen `(` und `)` schließlich ermöglicht die Konstruktion komplexerer Bedingungen. Mehrere solcher Bedingungsangaben können durch je einen Strichpunkt getrennt aneinandergereiht werden. Diese Aneinanderreihung entspricht einer UND–Verknüpfung und dient der übersichtlicheren Darstellung durch Vermeidung allzuvieler Klammerungen.

Die Angabe von Bedingungen für Ressourcenvariablen bedeutet dabei stets eine Bedingung für genau diejenige Ressource, die dann im Compiler bei der Bearbeitung eines Firmwareprogrammes der Variable zugeordnet wird, und *nicht* etwa eine Bedingung für alle Elemente der Trägermenge. Daher kann zum Beispiel durchaus Ungleichheit für zwei Variablen gefordert werden, die exakt gleich definiert wurden, die also dieselbe Träger-

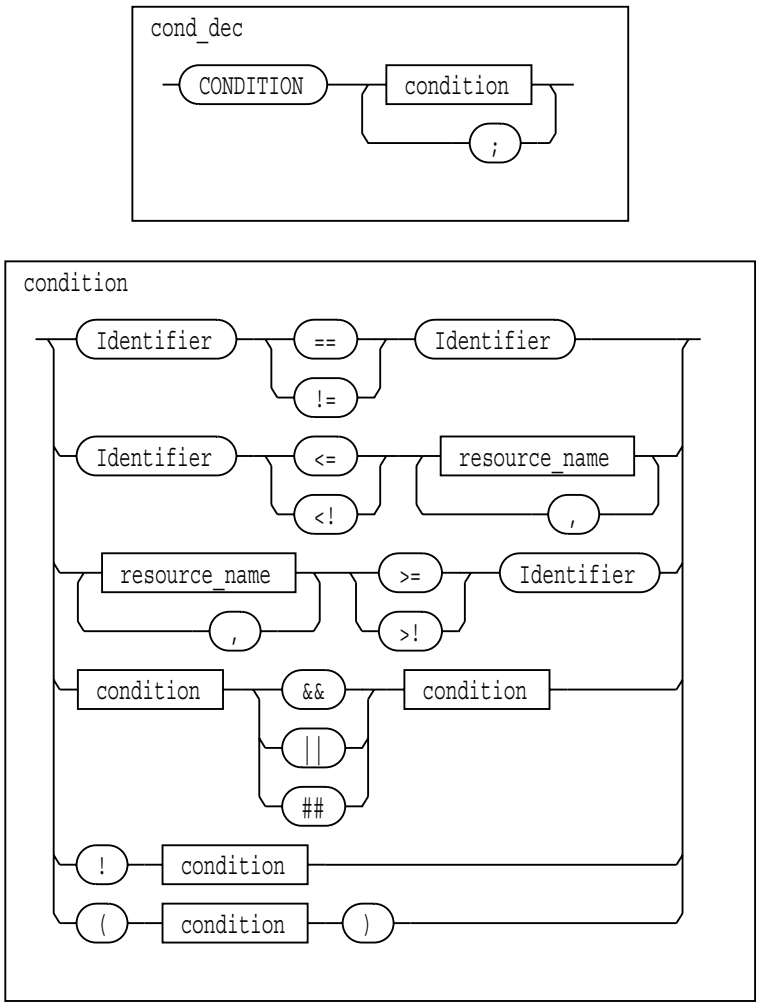


Abbildung 2.20: Die Syntax für die Angabe zusätzlicher Bedingungen. Achtung: Im Rahmen der `CONDITION`–Deklaration haben die Operatoren <= und >= eine andere Bedeutung als im Bereich der `constant_expression`, wo sie sich an der Programmiersprache C orientieren! Die genaue Bedeutung der Operatoren wird in Tabelle 2.5 vorgestellt.

```
CONDITION
        (op1 <= unit1 && op2 != op1) ||
        (op1 <! unit1 && op2 == op1);
        ! reg1 <= pointer;
        ! reg2 <= pointer;
```

Beispiel 2.15: Einige Möglichkeiten zur `CONDITION`–Deklaration: Die erste Bedingung ist erfüllt, wenn mindestens eine der beiden Bedingungen in den Klammern erfüllt ist. Die erste Klammer stellt die Bedingung, daß die Variable `op1` eines der Register der Gruppierung `unit1` sein muß und gleichzeitig nicht mit der Variablen `op2` übereinstimmen darf. Die zweite Klammer fordert, daß die Variablen `op1` und `op2` gleich sind, aber `op1` keines der Register aus `unit1` sein darf. Mit dem exklusiv–oder–Operator hätte man das noch kürzer als `(op1 <= unit1) ## (op1 == op2)` formulieren können. Die letzten beiden Bedingungen bedeuten, daß `reg1` und `reg2` keine der Ressourcen aus der Gruppierung `pointer` sein dürfen. Diese Bedingungen sind grundsätzlich vermeidbar: Man könnte ja gleich bei der Deklaration den Bereich der für `reg1` und `reg2` zugelassenen Ressourcen entsprechend einschränken. Allerdings ist die gewählte Form der Darstellung effizienter: Erstens ist die Deklaration sehr einfach und zweitens muß bei einer Änderung der Gruppierung `pointer` nicht gleich die Variablendeklaration in mehreren Mikrooperationen ebenfalls geändert werden. Auch hier gilt der Grundsatz, stets so zu deklarieren, daß es den *logischen* Gegebenheiten möglichst gut entspricht. Damit wird die MDL–Beschreibung leichter wartbar und weniger fehleranfällig, die Mühe der Umsetzung hingegen bleibt dem Computer. Insgesamt müssen schließlich alle drei durch Strichpunkte voneinander getrennten Bedingungen erfüllt sein.

menge besitzen. Im konkreten Fall müssen die beiden Ressourcen, die den Variablen zugewiesen wurden, dann verschieden sein, auch wenn sie derselben Trägermenge entstammen. Beispiel 2.15 zeigt einige Möglichkeiten.

Konflikte, die von `CONDITION`–Deklarationen verursacht werden, können also erst im Compiler konkret *geprüft* werden. Dennoch sind diese Bedingungs–Deklarationen nicht Teil der Syntax/Code–Zuordnung, weil sie unabhängig von der Syntax *beschrieben* werden können. Im Unterschied dazu gibt es auch Konflikte, die sinnvoll erst im Zuge der Syntax/Code–Zuordnung beschrieben werden können, man denke etwa an die Bedingung, daß ein konstanter Wert in der Syntax, der direkt einem Feld im Mikrobefehl zugewiesen wird, nur geradzahlig sein darf.

Eine komplette Mikrooperation

Als Zusammenfassung aller Beispiele aus diesem Abschnitt gibt Beispiel 2.16 die Definition einer kompletten Mikrooperation an. Damit wird nun die Gesamtheit aller Möglichkeiten dargestellt, die durch die Beschreibung der Mikrooperationen geboten werden. Den Rahmen einer solchen Beschreibung bieten das Schlüsselwort `MICROP` mit dem Namen der Mikrooperation am Beginn der Beschreibung und das Schlüsselwort `ENDOP` als Abschluß.

Es mag nicht immer von vornherein offensichtlich sein, welche Hardwarefunktionen

```
/* time granularity: 1 = 1ns */

#define T 25 /* clock cycle 25ns, clock frequency 40MHz */

#define T1 0,T
#define T2 T,T
#define T3 2*T,T
#define T4 3*T,T

#define Strobe 1000

MICROP  addition;
VAR     p1,p2 PARTOF pointer;
        reg1, reg2 PARTOF reg;
        op1 PARTOF unit1, portreg;
        op2 PARTOF op1, portreg;
        op3 PARTOF a_bus, b_bus;
FIELD l, v, g.low, xyz<0:1>, <10:12;#4;14>;
USING
        READ p1,p2 TIME (0,12);
        READ reg1, reg2 TIME (T1);
        a_bus, b_bus TIME (T,10);
        alu TIME (T2;T3);
        op1, op2, op3 TIME (Strobe-5,Strobe+3*T);
        unit2 TIME (Strobe+T,4;Strobe+2*T,4);
        c_bus TIME (3*T,10);
        WRITE reg2 TIME (T4);
DATAFLOW
        reg2 -> a_bus -> alu -> c_bus -> reg2;
        reg1 -> b_bus -> alu;
        op1 -> unit2 -> op2;
CONDITION
        (op1 <= unit1 && op2 != op1) ||
        (op1 <! unit1 && op2 == op1);
        ! reg1 <= pointer;
        ! reg2 <= pointer;
ENDOP;
```

Beispiel 2.16: Eine vollständige Mikrooperation.

man in Mikrooperationen zusammenfassen soll. Eine möglichst einfache Beschreibung wird man aber stets dann erhalten, wenn man solche Funktionen in einer Mikrooperation zusammenfaßt, die zwar funktional unterschiedlich sein mögen, aber in der Nutzung des Mikrobefehlsfeldes gleichartig und in ihrer Ressourcennutzung zumindest ähnlich sind.

2.4 Syntax und Code

Die Syntax/Code–Zuordnung dient primär der Generierung des Firmwarecompilers und gehört daher nicht zur Aufgabenstellung der Diplomarbeit. Dennoch wurden jene Teile davon entworfen und implementiert, die auch im Rahmen der MDL–Beschreibung selbst Verwendung finden. Die übrigen Teile werden hier der Vollständigkeit halber genannt und spezifiziert, soweit das mit den bisherigen Erfahrungen möglich ist. Zunächst werden jedoch einige Eigenschaften des Firmwarecompilers zusammengefaßt, die für das Verständnis der Syntax/Code–Zuordnung relevant sind. Dabei handelt es sich um eine Beschreibung des *neuen* Firmwarecompilers, wie er vom Generator erzeugt wird. Der Aufbau des Firmwarecompilers, wie er bisher verwendet wurde, ist ja bereits im ersten Kapitel beschrieben worden.

Firmwarecompiler

Der Firmwarecompiler dient der Umsetzung eines in Assembler–Form gegebenen Firmwareprogrammes in den Bitcode, der schließlich im Firmwarespeicher abgelegt wird und der Steuerung der Hardwareabläufe dient. In Analogie zu Hochsprachencompilern kann man den Firmwarecompiler in ein Frontend und ein Backend unterteilen:

Das Frontend führt die lexikalische und syntaktische Analyse durch. Dabei wird die Eingabe in eine linearisierte Folge von Terminalsymbolen der Grammatik umgeformt, sogenannte Token. Diese Token werden zudem attributiert, also mit aufbereiteter Information aus der Eingabe gewichtet. Die Ausgabe des Frontend ist somit eine linearisierte attributierte Tokenfolge. Sie wird als *Zwischensprache* oder *Intermediate Representation* (IR) bezeichnet, da sie sozusagen zwischen Frontend und Backend angesiedelt ist.

Das Backend analysiert die erhaltenen Tokenfolgen und führt auf Basis der Attribute semantische Aktionen durch. Diese umfassen vor allem die Prüfung von *Konflikten* sowie die Erzeugung des *Bitcode*.

Aus der MDL–Beschreibung müssen sowohl das Frontend als auch das Backend generiert werden können.

Beim Frontend ist das noch relativ einfach. Wenn sowohl die Syntax der Eingabesprache als auch die Zwischensprache wohldefiniert sind, gibt es für das Frontend nur noch einen Freiheitsgrad: Die Festlegung der Bezeichner. Dies bedeutet, daß das Frontend ein hardwareunabhängiger Programmteil ist, der aber auf einer hardwareabhängigen Tabelle

operiert, welche die Erkennung und Umsetzung der Bezeichner steuert. Diese Tabelle muß ergo aus der MDL–Beschreibung generiert werden können, es handelt sich dabei um die *Deskriptortabelle*.

Zunächst muß geklärt werden, welche Bezeichner überhaupt in einem Firmwareprogramm vorkommen:

Variablen: Das sind Namen für Register oder Registerteile, frei definierbare Variablen wie etwa in einer Hochsprache gibt es nicht[7].

Funktionen: Dies sind Bezeichner für komplexere Hardwareabläufe, die mit der Syntax der Firmwareprogrammiersprache nicht anders beschrieben werden können. Man nennt sie Builtinfunktionen, in gewissen Situationen auch Pseudovariablen, und übergibt ihnen gegebenenfalls auch eine bestimmte Anzahl von Parametern.

Symbole: Darunter versteht man Bezeichner, die erst vom Firmwareprogramm selbst definiert werden. Es handelt sich dabei etwa um Marken, symbolische Konstanten oder die Deklaration zusätzlicher Namen für Register und Registerteile.

Das Frontend muß also zunächst einmal eine Symboltabelle unterhalten, in der alle vom Firmwareprogramm definierten Symbole enthalten sind. Wird ein solches Symbol eingelesen, wird es entsprechend umgesetzt: Marken durch Adreßwerte, symbolische Konstanten durch die zugehörigen Zahlenangaben und deklarierte Zusatzbezeichner durch den ursprünglichen Namen des Registers oder Registerteiles.

Nach dieser ersten Umsetzung bleiben nur noch Variablen und Funktionen. Die Namen der Funktionen werden unverändert als Attribute dem Backend übergeben, somit müssen die Namen der Variablen mit Hilfe der Deskriptortabelle den entsprechenden Registern und Registerteilen zugeordnet und diese als Attribute der Tokenfolge mitgegeben werden. Wie die dafür nötige Information in der MDL–Beschreibung enthalten ist, wird im nächsten Abschnitt erläutert.

Die Generierung des Backend ist etwas komplizierter und wird im übernächsten Abschnitt behandelt.

Deskriptoren

Um die Deskriptortabelle generieren zu können, müssen folgende Informationen in der MDL–Beschreibung enthalten sein:

- Die Bezeichner für die Register selbst. Diese werden im Zuge der `REGISTER`–Definition angegeben.

[7] Der Begriff der Variablen bedeutet in diesem Abschnitt stets die Nennung eines Registers oder eines Registerteiles und darf nicht mit dem Begriff *Ressourcenvariable* verwechselt werden, der im vorigen Abschnitt geprägt wurde. Im Zweifelsfall wird daher der Begriff *Registervariable* verwendet, um Verwechslungen auszuschließen.

- Die Bezeichnung für Registerteile. Diese werden mit Hilfe der Bitfeldstrukturierung durch `MAP`–Definitionen bereitgestellt.
- Zusätzliche Beschreibungsmöglichkeiten für Register und Registerteile. Diese werden mit dem Schlüsselwort `ALIAS` definiert.

Da die MDL–Beschreibung viel übersichtlicher wird, wenn man diese Deskriptoren für Register und Registerteile nicht erst im Firmwareprogramm, sondern bereits in der MDL–Beschreibung selbst zuläßt, wurden diese Beschreibungsmöglichkeiten bereits in der vorliegenden Version der Sprache vorgesehen. Die Registerdefinition mittels `REGISTER` sowie die Strukturierung von Bitfeldern mittels `MAP` wurden bereits ausführlich beschrieben. Daher wird hier in weiterer Folge die zusätzliche Benennung von Registern und Registerteilen mittels Aliasnamen vorgestellt.

Natürlich kann man auch mittels `#define` bestimmten Registern oder Registerteilen neue Namen zuweisen. Diese werden aber im Zuge des Präprozessorlaufes beseitigt und sind somit für den Analysator unsichtbar. Sie eignen sich daher nur für solche Namen, die nicht in die Deskriptortabelle eingehen sollen. Die `ALIAS`–Definition hingegen stellt die neuen Bezeichner im Rahmen der Deskriptortabelle auch dem Firmwarecompiler zur Verfügung. Darüberhinaus besitzt die `ALIAS`–Definition zusätzliche mächtige Eigenschaften, die mittels `#define` nicht realisiert werden können. In Abbildung 2.21 wird das vollständige Syntaxdiagramm der `ALIAS`–Definition vorgestellt. Es wird zunächst die Form vorgestellt, die im Syntaxdiagramm von der unteren Zeile beschrieben wird, erst am Ende dieses Abschnittes wird die andere Form erläutert, die de facto nur eine effiziente Abkürzung in bestimmten Fällen darstellt.

Beispiel 2.17 zeigt einige Beispiele, um die prinzipiellen Möglichkeiten vorzustellen: Jede Zeile einer `ALIAS`–Definition beginnt mit der Angabe des neuen Bezeichners, gefolgt vom Schlüsselwort `ALIAS`. Daran anschließend wird in Form eines `resource_name` angegeben, welches Register, welcher Registerteil oder welche Registermenge diesen neuen Bezeichner erhalten soll. Dabei kann auch ein bereits definierter Aliasname verwendet werden. Schließlich kann noch eine `map_declaration` erfolgen, welche als zusätzliche Bitfeldstruktur dem Aliasnamen zugeordnet wird. Bereits in dem gegebenen einfachen Beispiel kann man jene beiden Fähigkeiten der `ALIAS`–Definition erkennen, welche die Mächtigkeit dieses Schlüsselwortes ausmachen:

Bereichsübersetzung bedeutet, daß Bitbereiche oder Array[8]–Bereiche bei der `ALIAS`–Definition entsprechend umgesetzt werden.

`MAP`–**Vererbung** ist jener Mechanismus, der einem Aliasnamen implizit und logisch strukturiert die entsprechende Bitfeldstruktur zuordnet. Eine allfällige explizite `map_declaration` gilt stets *zusätzlich*.

In weiterer Folge werden nun die `ALIAS`–Fähigkeiten in ihrer vollen Mächtigkeit vorgestellt. Dabei wird allerdings eine gewisse Vertrautheit und Erfahrung mit verschachtelten

[8]Um keine Verwechslung zwischen dem Begriff des Bitfeldes und dem des mehrdimensionalen Registerfeldes zu provozieren, wird hier für letzteres stets der englische Begriff *Array* verwendet.

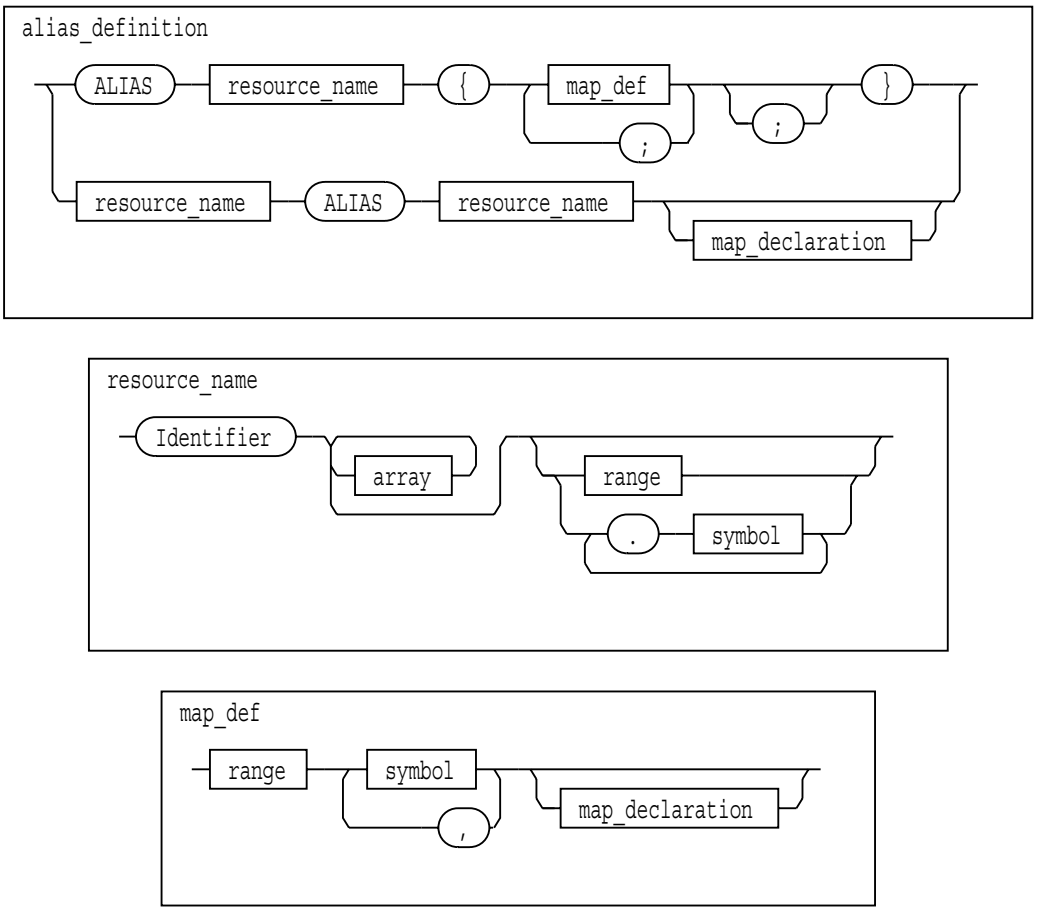


Abbildung 2.21: Die syntaktische Form der `ALIAS`–Definition: Die Syntaxdiagramme für `resource_name` und `map_def` werden hier nochmals gezeigt, um die verschiedenen Varianten der `ALIAS`–Definition besser mitverfolgen zu können.

`MAP`–Strukturen und den zugehörigen Dereferenzierungen vorausgesetzt. Noch eine Bemerkung zu den verwendeten Begriffen: Die Bezeichnung *Alias* bezieht sich stets auf den neuen Namen, der vor dem Schlüsselwort `ALIAS` steht, während mit *Original* stets der Bezeichner rechts vom Schlüsselwort gemeint ist, der das neu zu benennende Register beschreibt.

Für die folgenden Erläuterungen wurde die `MAP word` des Registerfeldes `reg` etwas erweitert, um komplexere Zusammenhänge darstellen zu können. Beispiel 2.18 zeigt die vorgenommenen Definitionen.

Bitbereiche

Bei der Umsetzung der Bitbereiche im Zuge der `ALIAS`–Definition gelten folgende Regeln:

- Wenn beim Original kein expliziter Bitbereich angegeben ist, so wird der *gesamte* Bitbereich des Originals implizit angenommen.

```
pt0  ALIAS reg[0][2];
br0  ALIAS pt0<5>;
dp1  ALIAS pt0.1;
hp1l ALIAS dp1.l;
lit  ALIAS reg[0][1] MAP {<#16> h,l};
ar6  ALIAS reg[2];
```

Beispiel 2.17: Hier einige Anwendungen der `ALIAS`–Definition: `pt0` wird als neuer zusätzlicher Name für das Register `reg[0][2]` eingeführt. `br0` wird als Aliasname für das Bit 5 von `reg[0][2]` definiert. Wie man sieht, können Aliasnamen auch als Basis zur Bildung weiterer Aliasnamen verwendet werden. Die dritte Zeile legt `dp1` als neuen Namen für `reg[0][2]<8:15>` fest. Dabei kann man erkennen, daß `pt0` dereferenziert angesprochen wird: `pt0` hat offensichtlich die Bitfeldstruktur `MAP word` von `reg` „geerbt". Die nächste Zeile deklariert `hp1l` als Aliasnamen für `reg[0][2]<12:15>`. Auch `dp1` erbt ebenfalls eine Bitfeldstruktur, nämlich `MAP bdigits` von `pt0.1`. Daher kann es auch mit `.l` dereferenziert werden. Die vorletzte Zeile zeigt die Möglichkeit einer zusätzlichen `MAP`–Deklaration: `lit` ist ein neuer Name für das Register `reg[0][1]` und erbt auch dessen `MAP word`, zusätzlich kann `lit` aber auch mit der explizit angegebenen und zugleich anonym definierten Bitfeldstruktur `{<#16> h,l }` dereferenziert werden. Interessant ist auch die letzte Zeile: `ar6` ist ein Aliasname für `reg[2]`. Dies ist aber ein eindimensionales Registerfeld, sodaß `ar6[0:15]` genau jenes eindimensionales Registerfeld beschreibt, welches durch `reg[2][0:15]` gegeben ist.

```
MAP sub1  { <#2> a,b };
MAP sub4a { <#2> x4a,y4a };
MAP sub4b { <#2> x4b,y4b };

MAP wbytes {
      <#8> 0,1,2,3  MAP bdigits { <#4> h,l MAP sub1; };
      };

MAP wdigits {
      <#4> 0h,0l,1h,1l MAP sub4a;
      <#4> 2h,2l,3h,3l MAP sub4b;
      };

MAP word OF wbytes, wdigits;
```

Beispiel 2.18: Die erweiterte Definition der `MAP word`, wie sie für die folgenden Erläuterungen Verwendung findet.

```
br0 ALIAS pt0<5>;
br1<3> ALIAS pt0<5>;

br2 ALIAS pt0<8:15>;
br3<18:25> ALIAS pt0<8:15>;

br4 ALIAS pt0;
br5<100:131> ALIAS pt0;

br6 ALIAS pt0.1;
br7<18:25> ALIAS pt0.1;

br8a ALIAS br7<0:3>; /* error */
br8b ALIAS br7<18:21>;
br9 ALIAS br7.h;
```

Beispiel 2.19: Die Möglichkeiten bei der Umsetzung des Bitbereiches: Die Namen `br` mit gerader Nummer erhalten dabei den impliziten Aliasbereich, während bei den ungeraden `br`–Namen (außer bei `br9`) stets ein expliziter Aliasbereich angegeben ist. In den ersten beiden Zeilen bezeichnet das Original genau das Bit 5 des Registers `pt0`. Die nächsten beiden Zeilen definieren explizit den Originalbereich 8:15, während die darauffolgenden beiden Zeilen implizit den vollen Originalbereich 0:31 beschreiben. Die Zeilen 6 und 7 zeigen, daß man natürlich auch mittels Dereferenzierung einen Registerteilbereich beschreiben kann. `br8a` ist fehlerhaft definiert, weil der Alias `br7` nur mit Bereichen aus 18:25 angesprochen werden kann, wie die Definition von `br8b` richtig zeigt. Die Definition von `br9` verwendet stattdessen die viel sicherere Methode der Dereferenzierung: Mittels `.h` werden die ersten 4 Bits von `br7` angesprochen, egal welche explizite oder implizite Nummerierung dafür deklariert wurde. Tabelle 2.6 zeigt die Umsetzung der Bereiche für dieses Beispiel.

- Wenn beim Alias kein expliziter Bereich angegeben ist, so beginnt der Bitbereichindex für den Aliasnamen stets bei 0. Ist hingegen ein expliziter Bereich angegeben, so wird dieser zur Ansprache von Teilen des Alias verwendet. Dann muß dieser explizite Bereich des Alias aber gleich lang sein wie der Bereich des Originals.

Beispiel 2.19 zeigt alle Möglichkeiten, die sich daraus ergeben, in Tabelle 2.6 sind die zugehörigen Bereiche angegeben.

Das Beispiel enthält eine Besonderheit (`br7` bis `br9`): Wenn ein Alias einen expliziten Bitbereich zugewiesen bekommt, so ist dies zugleich der erlaubte Bereich für die explizite Ansprache von Teilbereichen des Alias. Besitzt der Alias aber eine `MAP`–Struktur, so ist diese stets mit dem linken Rand des Alias ausgerichtet. Das bedeutet, daß die einzelnen Dereferenzierungsnamen der Bitstruktur stets dieselben Teile des Alias ansprechen, egal welche *Nummern* die Bits im Alias zugewiesen bekommen haben. Das ist aber im Normalfall genau das, was man eigentlich möchte. Im übrigen sollte die Zuweisung expliziter Alias–Bereiche nur dann vorgenommen werden, wenn es den logischen Zusammenhang verdeutlicht. Ansonsten sollte man die impliziten, stets bei Null beginnenden Bereiche

Alias	Aliasbereich	Originalbereich	Bitstruktur
br0	0:0	5:5	
br1	3:3	5:5	
br2	0:7	8:15	
br3	18:25	8:15	
br4	0:31	0:31	word
br5	100:131	0:31	word
br6	0:7	8:15	bdigits
br7	18:25	8:15	bdigits
br8a	fehlerhaft		
br8b	0:3	8:11	
br9	0:3	8:11	sub1

Tabelle 2.6: Erläuterungen zur Umsetzung des Bitbereiches. Die erste Spalte enthält den Alias–Namen, die zweite Spalte den erlaubten Bereich des Alias in direkter Ansprache. Die dritte Spalte zeigt, welchen Bereich des Registers `pt0` (selbst ein Alias auf das Register `reg[0][2]`) der jeweilige Alias neu benennt. Die letzte Spalte schließlich zeigt, welche Bitstrukturen für die jeweiligen Alias bestehen, wie also die Alias–Namen selbst weiter dereferenziert werden können. Eine genau Erklärung dazu wird im Abschnitt über die Vererbung von `MAP`–Strukturen gegeben.

bevorzugen.

Mehrdimensionale Registerfelder

Bei der Umsetzung mehrdimensionaler Arrays gelten prinzipiell dieselben Regeln wie bei der Umsetzung der Bitbereiche. Aufgrund der Möglichkeit mehrerer Dimensionen, die es ja beim Bitbereich nicht gibt, sehen die Regeln im Detail etwas anders aus:

- Explizite Bereichsangaben beim Original werden stets der Reihe nach den jeweiligen Dimensionen zugeordnet, beginnend bei den höherwertigen[9].
- Fehlende niederwertige Dimensionen werden implizit mit den vollen Bereichen des Originals in den jeweiligen Dimensionen ergänzt.
- Werden beim Original bei bestimmten Dimensionen keine Bereiche, sondern konkrete Indizes angegeben, so verringert das die Dimensionszahl des Alias. Die entsprechende Dimension wird beim Alias weggelassen.
- Das Array des Alias besteht aus sovielen Dimensionen, wie beim Original Bereiche (nicht konkrete Indizes!) angegeben wurden. Die impliziten Bereiche stimmen in der Länge mit den korrespondierenden Bereichen des Originals überein und beginnen stets bei Null.

[9]Mit höherwertig sind jene Dimensionen gemeint, die bei der Definition des Registerfeldes weiter links stehen.

Alias	Alias		Original	
ar1	—	—	2	9
ar2	—	0:7	2	8:15
ar3	—	4:11	2	8:15
ar4	—	0:1	2:3	9
ar5	—	1:2	2:3	9
ar6	—	0:15	2	0:15
ar7	—	16:31	2	0:15
ar8	0:1	0:7	2:3	8:15
ar9	1:2	0:7	2:3	8:15
ar10	1:2	4:11	2:3	8:15
ar11	0:1	0:15	2:3	0:15
ar12	1:2	0:15	2:3	0:15
ar13	1:2	16:31	2:3	0:15
ar14	0:3	0:15	0:3	0:15
ar15	4:7	0:15	0:3	0:15
ar16	4:7	16:31	0:3	0:15

Tabelle 2.7: Die Bereichsumsetzungen, die durch die Definitionen in Beispiel 2.20 bewirkt werden: Die erste Spalte enthält den Namen des jeweiligen Alias. Die zweite und dritte Spalte geben die höher- und niederwertige Array–Dimension des Alias an: Felder und Feldausschnitte des Alias dürfen nur Untermengen dieser Bereiche sein. Die letzten beiden Spalten geben die dazu korrespondierenden Bereiche des Registerfeldes an.

- Die Angabe expliziter Bereiche ersetzt die impliziten Bereiche, wieder bei den höherwertigen Bereichen beginnend. Dann muß aber der Anwender dafür Sorge tragen, daß die korrespondierenden Bereichslängen übereinstimmen.

Dies scheint auf den ersten Bilck ziemlich kompliziert, das liegt aber zu einem großen Teil an der allgemeinen Formulierung beliebigdimensionaler Registerfelder. In der Praxis werden zumeist nur ein- oder zweidimensionale Registerfelder vorherrschend sein. Daher wird der zweidimensionale Fall hier an Hand von Beispielen näher erläutert, der eindimensionale Fall kann daraus als Untermenge abgeleitet werden. Er unterscheidet sich schließlich auch kaum von der Umsetzung der Bitbereiche, die ja grundsätzlich immer eindimensional sind[10].

Beispiel 2.20 zeigt alle möglichen `ALIAS`–Definitionen im zweidimensionalen Fall. Tabelle 2.7 gibt die dadurch definierten Bereichsumsetzungen an.

[10]Der Unterschied ergibt sich daraus, daß bei einem Bitfeld, welches nur aus einem Bit besteht, dennoch ein `range` angegeben werden kann, zum Beispiel `br0<0>`, während bei einem Registerfeld, das nur aus einem einzigen Register besteht, dieses nicht mehr mittels `array` angesprochen werden kann: `pt0[0]` etwa ist nicht zulässig, der Alias kann nur mittels `pt0` allein verwendet werden.

```
ar1 ALIAS reg[2][9];

ar2 ALIAS reg[2][8:15];
ar3[4:11] ALIAS reg[2][8:15];

ar4 ALIAS reg[2:3][9];
ar5[1:2] ALIAS reg[2:3][9];

ar6 ALIAS reg[2];
ar7[16:31] ALIAS reg[2];

ar8 ALIAS reg[2:3][8:15];
ar9[1:2] ALIAS reg[2:3][8:15];
ar10[1:2][4:11] ALIAS reg[2:3][8:15];

ar11 ALIAS reg[2:3];
ar12[1:2] ALIAS reg[2:3];
ar13[1:2][16:31] ALIAS reg[2:3];

ar14 ALIAS reg;
ar15[4:7] ALIAS reg;
ar16[4:7][16:31] ALIAS reg;
```

Beispiel 2.20: Die möglichen ALIAS–Definitionen im Fall zweidimensionaler Registerfelder: Die erste Zeile spricht als Original genau ein Register an, es handelt sich um den nulldimensionalen Fall, das heißt der Alias besitzt keine Dimension, er kann nur als `ar1` angesprochen werden. Die nächsten sechs Zeilen beschreiben eindimensionale Fälle: Beim Original enthält nur eine Dimension einen Bereich, die andere ist fixiert. Damit ist der Alias eindimensional, seine Dimension korrespondiert mit jener Dimension des Originals, für die ein Bereich angegeben wurde. Die ersten vier eindimensionalen Fälle beschreiben explizite Array–Angaben beim Original, die letzten beiden halb implizite. Beim Alias wechseln implizite und explizite Arrays regelmäßig. Die übrigen drei Dreiergruppen beschreiben zweidimensionale Fälle: Durch Angabe von zwei Bereichen beim Original ist auch der Alias zweidimensional: Bei der ersten Dreiergruppe ist das Array des Originals voll explizit gegeben, bei der zweiten Dreiergruppe halb implizit und bei der letzten Dreiergruppe ganz implizit. Umgekehrt ist innerhalb jeder Dreiergruppe in der ersten Zeile der Alias mit voll implizitem Array definiert, in der jeweils zweiten Zeile halb explizit und in der jeweils letzten Zeile jeder Dreiergruppe ganz explizit. Die sich daraus ergebenden Array–Umsetzungen zeigt Tabelle 2.7.

Vererbung von MAP–Strukturen

Für die Vererbung der MAP–Strukturen gibt es nur eine einzige Regel: Der Alias erbt genau jene Bitfeldstruktur, welche die dereferenzierte Komponente des Originals besitzt. Wenn daher ein Original einen Registerteil mittels range definiert, wird überhaupt keine MAP vererbt, egal welcher Registerteil definiert wurde. Selbst wenn der range die volle Registergröße angibt, wird die MAP des Registers nicht vererbt. Dies ist eine Möglichkeit, die für ein Register definierte MAP–Struktur für ein Alias dieses Registers *unwirksam* zu machen.

Noch eine zweite Besonderheit soll vermerkt werden. Da die Vererbung der *logischen* Strukturierung der MAP–Definition folgt, diese aber ein und denselben Registerteil möglicherweise auf verschiedene Arten dereferenzieren läßt, kann es vorkommen, daß zwei verschiedene Dereferenzierungen zwar denselben Registerteil beschreiben, aber dennoch unterschiedliche MAP–Strukturen besitzen. Dies ist aber kein Nachteil. Im Gegenteil! Dadurch wird die *logisch hierarchische* Namensgebung von Registerteilen unterstützt: Je nach *Kontext* kann eine unterschiedliche Bitstruktur wirksam werden, obwohl unter Umständen der selbe Registerteil beschrieben wird. Beispiel 2.21 zeigt etliche Variationen zur Vererbung von Bitstrukturen, Voraussetzung ist die Definition der MAP word aus Beispiel 2.18 auf Seite 58. In Tabelle 2.8 wird schließlich zusammengefaßt, welche MAP–Strukturen bei den einzelnen Definitionen vererbt wurden.

Wird zusätzlich zur hier beschriebenen impliziten Vererbung der Bitfeldstruktur eine explizite map_declaration angegeben, so werden die beiden Bitfeldstrukturen gemischt. Dies geschieht nach denselben Regeln, wie sie bei der Zusammensetzung von MAP–Strukturen mit dem Schlüsselwort OF erläutert wurden.

Die alternative ALIAS–Definition

Die alternative ALIAS–Definition stellt nur eine Abkürzung dar, die dann besonders sinnvoll eingesetzt werden kann, wenn äquidistante Registerteile mit Alias–Namen zu versehen sind. Die Syntax erinnert an die MAP–Definition und tatsächlich gibt es eine gewisse Analogie: Bei der MAP–Definition bezeichnen die Namen der Bereiche die Komponentennamen der Registerteile, während sie bei der ALIAS–Definition neue Alias–Namen für die Registerteile definieren. In Beispiel 2.22 ist ein Beispiel angeführt, welches die beiden Arten der ALIAS–Definition für zwei konkrete Fälle vergleicht. Ganz allgemein stellt der resource_name hinter dem Schlüsselwort ALIAS das Original dar, während die einzelnen Zeilen zwischen den geschwungenen Klammern jeweils den Bitbereich des *Originals*, den Aliasnamen (hier nur mit impliziten Bereichen, da nur der Name angegeben werden darf) und eine optionale explizite map_declaration enthalten.

Bei dieser ALIAS–Definition findet naturgemäß keine MAP–Vererbung statt, da ja auch keine Komponenten angegeben werden. Benötigte Bitfeldstrukturen sind dann eben explizit anzuführen. Auch das wird im Beispiel gezeigt. Diese Methode ist jedenfalls nur dann zu empfehlen, wenn das Register in gleiche Teile geteilt wird, andernfalls ist sie zwar genauso möglich, bewirkt aber keine wesentliche Vereinfachung mehr. Selbstverständlich

```
pt00 ALIAS pt0<0:31>; /* no mapping */

ap1     ALIAS reg[0][2]<8:15>;
bp1     ALIAS reg[0][2].1;
cp1     ALIAS pt0<8:15>;
dp1     ALIAS pt0.1;
ep1     ALIAS pt00<8:15>;
fp1     ALIAS pt00.0; /* error */

ap1l    ALIAS reg[0][2]<12:15>;
bp1l    ALIAS reg[0][2].1.l;
cp1l    ALIAS reg[0][2].1l;
dp1l    ALIAS pt0<12:15>;
ep1l    ALIAS pt0.1.l;
fp1l    ALIAS pt0.1l;
gp1l    ALIAS bp1<4:7>;
hp1l    ALIAS bp1.l;
ip1l    ALIAS ep1<4:7>;

jp1l    ALIAS pt0.5; /* error */
kp1l    ALIAS pt0.1.x; /* error */
```

Beispiel 2.21: Einige Variationen zur Vererbung von `MAP`–Strukturen: Der Alias `pt00` besitzt keine `MAP`, diese wird aufgrund der Angabe des `range` nicht vererbt. Analoges gilt für alle anderen Alias–Namen, wo das Original nicht dereferenziert wird, sondern mittels eines `range` die Registerteile anspricht. Daher ist auch die Definition von `fp1` fehlerhaft, `pt00` kann ja nicht dereferenziert werden. Die letzten beiden Zeilen sind fehlerhaft, weil die `MAP word` keine Komponente `.5` enthält und die `MAP bdigits` der Komponente `.1` der `MAP word` keine Komponente `.x` besitzt. Die Vererbung der Bitstruktur für die dereferenzierten `ALIAS`–Definitionen wird in Tabelle 2.8 erläutert.

Alias	Bitbereich	Bitstruktur
pt00	0:31	—
ap1	8:15	—
bp1	8:15	bdigits
cp1	8:15	—
dp1	8:15	bdigits
ep1	8:15	—
fp1	fehlerhaft	
ap1l	12:15	—
bp1l	12:15	sub1
cp1l	12:15	sub4a
dp1l	12:15	—
ep1l	12:15	sub1
fp1l	12:15	sub4a
gp1l	12:15	—
hp1l	12:15	sub1
ip1l	12:15	—
jp1l	fehlerhaft	
kp1l	fehlerhaft	

Tabelle 2.8: Die Vererbung der `MAP`–Strukturen für die Definitionen aus Beispiel 2.21: Der Alias `bp1` erhält die `MAP bdigits`, weil die Komponente `.1` der `MAP word` genau diese `MAP` besitzt. Dasselbe gilt für `dp1`. Interessant ist ein Vergleich der Definitionen von `bp1l` und `cp1l`: Der Aliasname `bp1l` erhält die `MAP sub1` von der Komponente `.l` der Komponente `.1` der `MAP word`. `cp1l` hingegen wird durch die `MAP sub4a` strukturiert, die sie von der Komponente `.1l` der `MAP word` erhält. Obwohl also `bp1l` und `cp1l` denselben Registerteil desselben Registers beschreiben, besitzen sie aufgrund der logisch hierarchischen Vererbung der `MAP`–Definitionen eine unterschiedliche Bitstruktur. Auch `ap1l` beschreibt denselben Registerteil, besitzt aber gar keine Bitstruktur, da bei der Definition keine Komponentenangabe, sondern die explizite Bereichsangabe verwendet wurde. Für die übrigen `ALIAS`–Definitionen gilt das völlig analog.

```
pt1 ALIAS reg[0][3];

/* standard */

p4 ALIAS pt1.0;
p5 ALIAS pt1.1;
p6 ALIAS pt1.2;
p7 ALIAS pt1.3;

p4h ALIAS pt1.0h;
p4l ALIAS pt1.0l;
p5h ALIAS pt1.1h;
p5l ALIAS pt1.1l;
p6h ALIAS pt1.2h;
p6l ALIAS pt1.2l;
p7h ALIAS pt1.3h;
p7l ALIAS pt1.3l;

/* alternative */

ALIAS pt1 {
   <#8>  p4,p5,p6,p7 MAP bdigits;
   <0,4> p4h,p4l,p5h,p5l MAP sub4a;
   <#4>  p6h,p6l,p7h,p7l MAP sub4b;
};
```

```
/* standard */

m0 ALIAS reg[1:3].0;
m1 ALIAS reg[1:3].1;
m2 ALIAS reg[1:3].2;
m3 ALIAS reg[1:3].3;

/* alternative */

ALIAS reg[1:3]
   {<#8> m0, m1, m2, m3
         MAP bdigits};
```

Beispiel 2.22: Hier werden die beiden Arten der ALIAS–Definition für zwei konkrete Beispiele verglichen: Links ein eher einfacheres Beispiel, das nur ein Register betrifft, rechts eine etwas fortgeschrittenere Nutzung der alternativen ALIAS–Definition in Kombination mit der Array–Bereichsübersetzung. In beiden Fällen wird oben die bisher erläuterte Methode und unten die abgekürzte Methode verwendet: Lediglich die im ersten Fall implizit ererbten MAP–Strukturen müssen im zweiten Fall explizit angegeben werden. Durch Vergleich der Standarddefinition mit der alternativen Definition kann man leicht die Wirkungsweise der alternativen ALIAS–Definition erkennen.

kann diese Methode auch mit den anderen kombiniert werden, etwa mit den Möglichkeiten der Beschreibung mehrdimensionaler Felder, wie Beipiel 2.22 ebenfalls zeigt.

Nachdem nun alle Möglichkeiten der `ALIAS`–Definition vorgestellt wurden, soll noch darauf hingewiesen werden, daß alle diese Möglichkeiten voneinander weitgehend unabhängig auch kombiniert auftreten können. Zwar kann man mit der `ALIAS`–Definition Konstrukte erzeugen, die schon in den Bereich der Denksportaufgaben gehören, dies ist aber nicht der Sinn der Sache. Vielmehr bietet die `ALIAS`–Definition gemeinsam mit der `MAP`–Definition die Möglichkeit, sich von der Angabe absoluter Bereiche zu lösen und logisch–hierarchisch strukturierte Namen zu verwenden. Einmal definiert können die Bereiche dann immer mit den logischen Namen angesprochen werden. Dies hat nicht nur den Vorteil der Übersichtlichkeit und leichteren Lesbarkeit, auch Änderungen werden wesentlich unproblematischer: An einer einzigen Stelle geändert, wirkt sich die neue Definition automatisch an allen Stellen ihres Auftretens aus. In der MDL–Beschreibung eines Prozessors sollten ergo absolute Bereichsangaben eine Seltenheit sein, wenn man von den `MAP-` und `ALIAS`–Definitionen selbst einmal absieht.

Semantische Aktionen

Dieser Abschnitt enthält keine Lösung, sondern eine Spezifikation. Er zählt die Aufgaben des Backends auf und leitet daraus die Anforderungen an die MDL–Beschreibung ab. Dieser Abschnitt stellt also gewissermaßen die Aufgabenstellung für weiterführende Arbeiten dar.

Folgenden Aufgaben müssen vom Backend jedenfalls durchgeführt werden:

- Prüfung, ob ein Mikrobefehl überhaupt in der prozessorspezifischen Sprachuntermenge des Sprachrahmens enthalten ist.
- Prüfung syntaxorientierter Konflikte, ob etwa zum Beispiel eine Konstante eine gerade Zahl ist.
- Prüfung der Ressourcenkonflikte.
- Ermittlung des Bitcodes des Mikrobefehls.

Um das Backend generieren zu können, muß daher die MDL–Beschreibung folgende Anforderungen erfüllen:

- Angabe der erlaubten beziehungsweise codierbaren Tokenfolgen. Diese legen dann gemeinsam die Sprachuntermenge fest.
- Allgemeine Formulierung von Algorithmen in einer Programmiersprache, wobei auf die Attribute der Token mittels einer Metasymbolik zugegriffen werden kann. Damit sind die syntaxorientierten Konflikte behandelbar.

- Für jede Mikrooperation müssen die entsprechenden Tokenattribute an die jeweiligen Ressourcenvariablen zugewiesen werden. Anschließend muß eine Konfliktprüfung mit allen Mikrooperationen des Mikrobefehls durchgeführt werden. Damit werden die Ressourcenkonflikte behandelt und insbesonders die Konflikte zweiter Art aufgelöst. Dabei müssen außerdem die explizit angegebenen `CONDITION`–Definitionen berücksichtigt werden.

- Aus den Attributen kann der Bitcode der einzelnen Mikrooperationen algorithmisch ermittelt werden. Er ist dann den entsprechenden Bitbereichen im Mikrobefehlsfeld zuzuordnen, woraus sich insgesamt der Bitcode des Mikrobefehls ergibt.

Um die Vorteile der MDL–Beschreibung voll nutzen zu können, ist es weiters sinnvoll, innerhalb der Programmcodesegmente die `FIELD`-, `REGISTER`-, `MAP`- und `ALIAS`–Definitionen mittels einer Metasymbolik zugänglich zu machen. Darüberhinaus sollte die Verwendung des `multirange` sowie auch der `Binary_constant` möglich sein. Vermutlich wird auch die Einführung eines neuen Schlüsselwortes notwendig sein, um die Tokenfolgen und die Programmcodesegmente von der übrigen MDL–Beschreibung entsprechend abheben zu können.

2.5 Überprüfung der Entwurfsziele

In diesem Abschnitt wird nun zusammengefaßt, wie die Sprache MDL die im Abschnitt 1.4 ab Seite 15 an sie gestellten Anforderungen erfüllt. Zunächst einmal wird aufgezeigt, *was* mit der Sprache MDL beschrieben werden kann:

1. Die Hardwareressourcen werden im Deklarationsteil beschrieben: Die Elemente werden mit den Schlüsselwörtern `DEVICE` und `REGISTER` definiert, die Verbindungen mit dem Schlüsselwort `PATH`. Die besondere Ressource des Mikrobefehlsfeldes wird mit dem Schlüsselwort `FIELD` gehandhabt. Außerdem bieten die Schlüsselwörter `SET` und `GROUP` noch die Möglichkeit der Gruppierung von Ressourcen.

2. Die Verwendung der Hardwareressourcen in den Mikrooperationen wird im Operationsteil beschrieben: Das Schlüsselwort `MICROP` definiert die Mikrooperationen, mit dem Schlüsselwort `USING` wird die Verwendung der Ressourcen und mit dem Schlüsselwort `DATAFLOW` die Verwendung der benötigten Verbindungen beschrieben. Die Verwendung der Mikrobefehlsfelder wird mit dem Schlüsselwort `FIELD` beschrieben. Das Schlüsselwort `TIME` dient der Angabe der Zeitangaben, mit den Schlüsselwörtern `READ` und `WRITE` wird die Art der Benutzung von Registern angegeben. Die Handhabung der subtilen Konflikte zweiter Art erfolgt mit den Schlüsselwörtern `VAR` und `CONDITION`.

3. Die Zuordnung der Syntaxelemente der Mikroprogrammsprache sowohl zu den Mikrooperationen als auch zur Codegenerierung wurde im Zuge der Diplomarbeit nur ansatzweise ausgeführt. Der Syntaxteil der MDL–Beschreibung enthält die benötigten Informationen. Außerdem wird auch mit den Schlüsselwörtern `REGISTER`, `MAP`

und `ALIAS` Information angegeben, die für die Syntax/Code–Zuordnung relevant ist.

Insoweit erfüllt also die Sprache MDL die an sie gestellten Anforderungen. Daher werden nun die Eigenschaften der Sprache überprüft:

deskriptiv: Diese Eigenschaft ist offensichtlich gegeben, da die funktionalen Eigenschaften der Hardwareelemente und der Mikrooperationen nicht beschrieben werden. Lediglich die Syntax/Code–Zuordnung wird eine funktionale Beschreibung erfordern, da es sich dabei aber ebenfalls nicht um die Beschreibung von Hardwarefunktionen handelt, ist die MDL bezüglich der Hardware rein deskriptiv.

effizient: Die geringe Redundanz wird vor allem durch die mächtigen Möglichkeiten der Schlüsselwörter `FIELD`, `MAP` und `ALIAS`, aber auch durch `GROUP` und `SET` erreicht. Die Syntax/Code–Zuordnung ist ebenfalls mit geringer Redundanz konzipiert. Lediglich eine gewisse Redundanz besteht zwischen dem Simulator, der ja bereits eine Hardwarebeschreibung enthält, und der Hardwarebeschreibung durch die MDL. Wie im Abschnitt 1.3 ab Seite 9 aufgezeigt, kann diese Redundanz durch Einführung eines Filters überwunden werden.

einfach: Diese Eigenschaft ist leider nicht restlos erfüllt, gewisse Sprachelemente der MDL sind nicht ganz trivial. Vielmehr wurde versucht, die Eigenschaft der Einfachheit durch die Eigenschaft der *Vertrautheit* zu ersetzen, indem die Programmiersprache C als syntaktisches und lexikalisches Vorbild gewählt wurde. Durch die weite Verbreitung dieser Programmiersprache wird auf diesem Weg der Zugang zur MDL zumindest erleichtert.

flexibel: Die Forderung nach hoher Flexibilität und Mächtigkeit ist der Hauptgrund für die mangelnde Einfachheit der Sprache: Da Flexibilität und Mächtigkeit grundsätzlich im Gegensatz zur Einfachheit stehen, wurde hier der Mächtigkeit der Vorrang eingeräumt. Flexibilität und Mächtigkeit sind insofern gegeben, als die Sprache keine Beschränkung auf eine spezielle Prozessorfamilie enthält. Dennoch ist es denkbar, daß sie für bestimmte Prozessoren noch erweitert werden müßte.

großzügig: Diese Eigenschaft ist die Voraussetzung für die Verwendung der Sprache im Rahmen eines Entwurfswerkzeuges: Sie wird durch mannigfache implizite Annahmen erfüllt, die bei fehlenden oder unvollständigen Angaben getroffen werden. Ja die Sprache selbst erlaubt sogar die korrekte Darstellung von sehr mangelhaften Informationen, sodaß die Möglichkeit der Beschreibung einer abstrakten Maschine ohne Festlegung von Bezeichnern, Bitcodes oder ähnlichem möglich ist. Hier soll vor allem darauf aufmerksam gemacht werden, daß bei den Schlüsselwörtern `DEVICE`, `REGISTER`, `FIELD`, `GROUP`, `SET` und `MICROP` bis auf den Namen alle Angaben optional sind!

Im letzten Abschnitt des nun folgenden Kapitels zur Implementierung des Analysators werden die Entwurfsziele des Analysators überprüft. Im Kapitel 4 wird dann noch ein etwas allgemeinerer Ansatz für Sprache und Analysator vorgestellt, um den Mangel an struktureller Einfachheit bei gleichzeitiger hoher Flexibilität zu beseitigen.

Kapitel 3

Implementierung des Analysators

3.1 Einleitung

Mit der Implementierung schlägt die Stunde der Wahrheit: Erst die reale Implementierung eines *Prototyps* beweist die tatsächliche Durchführbarkeit der geplanten Vorhaben. Und genau das war auch der Zweck der Implementierung des Analysators, nämlich die grundsätzliche Machbarkeit zu beweisen sowie die Möglichkeiten und Fähigkeiten aufzuzeigen. Daher haben sich auch im Zuge der Programmierung des Analysators zahlreiche Rückwirkungen auf das Sprachdesign ergeben, bevor die Sprache ihr jetziges Aussehen angenommen hat.

Auf Optimierungen aller Art, wie zum Beispiel hinsichtlich Geschwindigkeit oder Speicherplatz, auf eine komfortable Bedienungsoberfläche oder auf ein effizientes Datenbankdesign hingegen wurde überhaupt kein Wert gelegt. Lediglich die Funktionen zur Konfliktanalyse wurden soweit optimiert, daß ihr Laufzeitverhalten erträglich ist. Verglichen mit einem Auto, entspricht der Prototyp etwa einem Geländewagen mit offenem Fahrgastraum: Man kommt zwar überall hin damit, aber der Fahrer sieht nachher auch entsprechend aus.

Die Implementierung des Analysators erfolgte in Standard C, ausgelegt für ein UNIX–Betriebssystem. Konkret wurde der Analysator sowohl auf einer Apollo WS30 als auch auf einer NeXTstation getestet. Die Portierung zwischen unterschiedlichen UNIX–Rechnern ist unkompliziert, lediglich an zwei Stellen müssen gegebenenfalls Anpassungen erfolgen: Der Pfad für den verwendeten C–Präprozessor muß in `default.h` angepaßt werden und das Zusammenspiel der verbesserten Fehlerbehandlung mit dem Parsergenerator `yacc` muß eventuell überprüft werden, je nachdem welche Version von `yaccpar` vorliegt. Details dazu werden in weiterer Folge erläutert.

In den folgenden Abschnitten wird beschrieben, was der Analysator leistet, welche Eingaben möglich sind und welche Ausgaben erwartet werden können. Danach wird kurz der innere Aufbau des Analysators dargestellt, ohne jedoch auf Details einzugehen. Zuletzt wird noch überprüft, inwieweit die im Abschnitt 1.4 ab Seite 15 gestellten Anforderungen auch erfüllt werden.

```
localhost> mdl -u

##### MDL Analyzer
usage: mdl [options] [source [destination]]
sourcefile must have extension .mdl.
source and destination must not have any extension!
        options:
                -c          generate conflict table
                -nc         suppress conflict table
                -d          generate descriptor table
                -nd         suppress descriptor table
                -f[name]    use name as control file name
                -nf         suppress use of control file
                -h          interactive input mode
                -i          enable interactive top-down analysis
                -ni         disable interactive top-down analysis
                -p          proof connections
                -np         ignore connections
                -t<value>   set maximum cycle time to <value>
                -u          get this usage message
                -w          print warning diagnostics
                -nw         suppress warning diagnostics
                            ... and all C-preprocessor options.
localhost>
```

Beispiel 3.1: Diese Meldung gibt in geraffter Form die Möglichkeiten des Programmaufrufes des Analysators wieder: Zuerst können Optionen angegeben werden [options], dann erfolgt die optionale Angabe des Dateinamens der MDL–Beschreibung als Quelldatei [source] sowie der Dateiname der Tabellen als Zieldatei [destination]. localhost> ist das Promptzeichen der Shell.

3.2 Eingabemöglichkeiten und Steuerung

Optionen und Dateien

Der Aufruf des Programmes erfolgt mit dem Namen `mdl`, gefolgt von mehreren optionalen Parametern. Der Aufruf `mdl -u` sowie jeder falsche Aufruf des Programmes bewirken eine Meldung der Form von Beispiel 3.1, in der die Aufrufmöglichkeiten des Programmes kurz zusammengefaßt sind. Diese werden nun in weiterer Folge im Detail vorgestellt.

Die grundsätzliche Funktionsweise des Analysators ist es, eine MDL–Beschreibung einzulesen und daraus eine *Deskriptortabelle* und eine *Konflikttabelle* zu erzeugen. Darüberhinaus gibt es die Möglichkeit der *Top–Down–Konfliktanalyse*: Dabei werden zwei oder mehrere konkrete Mikrooperationen kombiniert und die entstehenden Konflikte aufgezeigt. Die Angabe, welche Mikrooperationen kombiniert werden sollen, kann auf zwei Arten erfolgen: Erstens mit einer Steuerdatei und zweitens durch interaktive Eingabe nach der Hauptanalyse. Die verschiedenen Möglichkeiten, die der Analysator dabei bietet, werden mit den Parametern beim Programmaufruf gesteuert.

Zunächst zur Bedeutung der Optionen[1], die stets mit einem `-` beginnen und unmittelbar auf den Programmnamen folgen müssen:

`-c` Es soll eine Konflikttabelle generiert werden. Diese Option ist Standard und kann daher weggelassen werden. Der Name der Konflikttabelle ist der Zieldateiname mit der Erweiterung `.con`.

`-nc` Es soll keine Konflikttabelle generiert werden.

`-d` Es soll eine Deskriptortabelle erzeugt werden. Diese Option kann als Standard weggelassen werden. Der Name der Deskriptortabelle entspricht dem Zieldateinamen, erweitert um die Endung `.des`.

`-nd` Es soll keine Deskriptortabelle generiert werden.

`-f[name]` Mit dieser Option wird angegeben, daß eine *Steuerdatei* für die Top–Down–Konfliktanalyse verwendet werden soll. Wird nur `-f` angegeben, so ist der Name der verwendeten Steuerdatei gleich dem Namen der MDL–Beschreibung, aber mit der Erweiterung `.ctl`. Wird hingegen unmittelbar hinter `-f` ein Name angegeben, so wird dieser als Dateiname für die einzulesende Steuerdatei angenommen. Gleichzeitig wird auch eine zusätzliche Tabelle als Ausgabe erzeugt, die Top–Down–Konflikttabelle. Diese erhält als Namen den Zieldateinamen, um den Anhang `.tdn` erweitert.

`-nf` Es soll kein Steuerfile verwendet werden. Diese Einstellung ist Standard und kann daher weggelassen werden.

`-h` Die Eingabe der Parameter soll *interaktiv* erfolgen. Alle nach dieser Option angegebenen Parameter, auch die Dateinamen, werden ignoriert und sind daher sinnlos. Alle davor angegebenen MDL–Parameter werden überschrieben und sind daher ebenfalls sinnlos. Ein sinnvoller Aufruf mit dieser Option wird also die Gestalt `mdl -h` besitzen. Lediglich Optionen für den Präprozessor müssen zwischen `mdl` und `-h` angegeben werden. Die interaktive Eingabe wird im Anschluß besprochen.

`-i` Mit dieser Option wird die interaktive Top–Down–Analyse aktiviert: Nach der Hauptanalyse können Kombinationen von Mikrooperationen auf Konflikte geprüft werden. Diese Option ist also der Option `-f` verwandt, nur daß hier die Eingabe der zu prüfenden Kombinationen über das Terminal an Stelle der Steuerdatei erfolgt. Dabei wird, sofern am Beginn der interaktiven Eingabe selbst nichts anderes gewünscht wird, eine weitere Top–Down–Konflikttabelle erstellt, die den Zieldateinamen, erweitert um `.itd`, besitzt.

`-ni` Es wird keine interaktive Top–Down–Analyse gewünscht. Diese Einstellung ist Standard und muß daher nicht angegeben werden.

`-p` Die beim Schlüsselwort `DATAFLOW` angegebenen benötigten Verbindungen werden mit den Verbindungen, die mittels `PATH` definiert wurden, auf Konsistenz geprüft. Dabei findet die automatische *Datenpfadsynthese* statt, wobei nicht existente aber benötigte Verbindungen eingerichtet werden. Diese Einstellung kann als Standard weggelassen werden.

[1] In der englischen Literatur *flags* genannt.

`-np` Verbindungsprüfung und Datenpfadsynthese finden nicht statt. Da bei der Einrichtung einer benötigten, nicht definierten Verbindung jedesmal eine Warnung ausgegeben wird, kann durch die automatische Datenpfadsynthese in einem frühen Entwurfsstadium der MDL, wo so detaillierte Angaben noch nicht vorliegen, eine Flut von Warnmeldungen die Arbeit mit dem Analysator unübersichtlich werden lassen. Um das zu verhindern, können alle mit den Verbindungen im Zusammenhang stehenden Prüfungen mit dieser Option deaktiviert werden.

`-t<value>` Auf die Option `-t` muß zwingend unmittelbar die Angabe einer ganzen Zahl folgen. Diese Zahl gibt dann die Dauer eines Maschinenzyklus an und wird anstelle des Wertes von 10000 verwendet, der als Standard bei Weglassen von `-t` verwendet wird. Diese Maschinenzyklusdauer wird für verschiedene interne Prüfungen benötigt. Wenn aber in der MDL–Beschreibung das Schlüsselwort `MAXTIME` angegeben ist, so ist diese Angabe stärker als Standard oder Option.

`-u` Diese Option bewirkt die Meldung gemäß Beispiel 3.1.

`-w` Es sollen Warnmeldungen ausgegeben werden. Diese Einstellung ist Standard und daher nicht unbedingt erforderlich.

`-nw` Die Warnmeldungen sollen unterdrückt werden. Da oft am Beginn der Entwicklung einer MDL–Beschreibung sehr viele implizite Annahmen getroffen werden, um Fehlendes und Unvollständiges zu ergänzen, kann es zu einer Flut von Warnmeldungen kommen, weil jede implizite Annahme von einer Warnung begleitet wird. Mit der Option `-nw` kann die Ausgabe dieser Warnmeldungen verhindert werden. Dennoch erhält man am Ende der Analyse im Rahmen der Diagnoseausgabe die Anzahl der unterdrückten Warnmeldungen. Wenn sich eine MDL–Beschreibung der Vervollkommnung nähert, sollten jedenfalls auch die Warnmeldungen überprüft und beseitigt werden.

Die Optionen wirken in der Reihenfolge ihres Auftretens, sodaß bei widersprüchlichen Angaben immer die *letzte* gültig ist.

Darüberhinaus können alle Optionen des verwendeten *Präprozessors* benutzt werden. Diese werden hier nicht angegeben, da sie lokal unterschiedlich sein können, stattdessen hier einige Worte über die Einbindung des lokalen Präprozessors: Grundsätzlich kann jeder Präprozessor verwendet werden, es muß lediglich der vollständige Pfad in `default.h` angegeben werden. Allerdings gibt es eine Einschränkung: Der Präprozessor darf keine zusätzlichen Ausgaben erzeugen, die dann dem Analysator eingegeben würden. Lediglich eine Ausnahme gibt es: Die Datei- und Zeileninformationen, die der C–Präprozessor zum Beispiel in der Gestalt `# 6 "sample.mdl" 2` einfügt, werden vom Analysator erkannt und auch berücksichtigt. Daher wird die Einbindung eines Standard–C–Präprozessors bevorzugt. Bei der Übergabe der Optionen geht der Analysator folgendermaßen vor: Alle ihm unbekannten Optionen werden dem Präprozessor übergeben. Sollten fehlerhafte Optionen verwendet werden, so wird das erst vom Präprozessor erkannt, der dann seinerseits eine entsprechende Fehlermeldung liefert. Bei der konkreten Implementierung des Analysators wurde stets ein Standard–C–Präprozessor verwendet.

Datei	Dateiname
MDL–Beschreibung	`<source>.mdl` —
Steuerdatei	`<source>.ctl` `beliebig`
Konflikttabelle	`<dest>.con`
Deskriptortabelle	`<dest>.des`
gesteuerte Top–Down–Konflikttabelle	`<dest>.tdn`
interaktive Top–Down–Konflikttabelle	`<dest>.itd`

Tabelle 3.1: Die Namenskonventionen des Analysators: <source> ist der Quelldateiname, alternativ kann auch von der Standardeingabe gelesen werden, wobei die MDL–Beschreibung dann gar keinen Namen besitzt. Der Name der Steuerdatei ergibt sich entweder implizit aus dem Quelldateinamen oder er wird explizit beliebig angegeben. Ist kein Quelldateiname gegeben, so ist auch kein impliziter Name für die Steuerdatei verfügbar. <dest> ist der Zieldateiname. Wird er nicht angegeben, so ist er gleich dem Quelldateinamen. Wenn auch der Quelldateiname fehlt, so wird `mdl_table` als Zieldateiname <dest> angenommen.

Die Standardeinstellungen, also die vorgegebenen Namen der Dateien sowie der Namenserweiterungen und auch die vorgegebenen Einstellungen bei Fehlen der entsprechenden Optionen können in der Datei `default.h` verändert werden. Danach muß aber mittels `make mdl` der Analysator neu erzeugt werden.

Nun zu den beiden Parametern `[source]` und `[destination]`: Die Quelldatei ist jene Datei, welche die MDL–Beschreibung enthält. Dabei wird der Dateiname ohne Erweiterung angegeben, es wird implizit die Erweiterung `.mdl` angenommen. Auch diese Annahme kann in `default.h` verändert werden. Der schon mehrfach angegebene Zieldateiname ist der letzte Parameter. Mit den verschiedenen Erweiterungen dient dieser Name der Bezeichnung der Tabellen als Ausgabe des Analysators. Wird der an sich optionale Zieldateiname weggelassen, so wird als Standard der Name der Eingabe, also der Quelldateiname, angenommen.

Wenn auch noch der Quelldateiname weggelassen wird, so liest der Analysator aus der Standardeingabe. Da wohl niemand eine MDL–Beschreibung direkt eintippen möchte, dient diese Möglichkeit eher der Verwendung des Analysators als Filter. In diesem Fall gibt es also keinen Quelldateinamen und keinen expliziten Zieldateinamen. Da dennoch ein Name für die Ausgaben benötigt wird, wird als Standard der Name `mdl_table` als Zieldateiname angenommen, der dann mit den verschiedenen Erweiterungen der Namensgebung der Tabellen dient. Wie alle Standards kann auch dieser Name in `default.h` den persönlichen Wünschen angepaßt werden. Die Option `-f` ohne Angabe eines Namens für das Steuerfile wird bei der Verwendung als Filter einfach unterdrückt, da kein sinnvoller impliziter Name angenommen werden kann. Wird jedoch unmittelbar hinter `-f` ein Name angegeben, so bleibt die Option wirksam. Tabelle 3.1 zeigt die Namenskonventionen nochmals zusammengefaßt.

Auch die Option `-h` kann für den Fall der Verwendung des Analysators als Filter

```
localhost> mdl -I/mike/MDL/mdl -Dswitch=alpha -f -np -nw sample

##### MDL Analyzer

##### preprocessing with /lib/cpp -I/mike/MDL/mdl -Dswitch=alpha sample.mdl.

##### starting analysis.

##### analysis finished. [no errors][7 warnings]

##### analysis of sample.mdl.
##### control file sample.ctl.
##### descriptor table sample.des generated.
##### conflict table sample.con generated.
##### top-down conflict table sample.tdn generated.
##### connections ignored.
localhost>
```

Beispiel 3.2: Aufruf und Diagnoseausgaben des Analysators für einen typischen, fehlerfreien Fall: Die ersten beiden Flags sind für den Präprozessor bestimmt. Dann wird dem Analysator mitgeteilt, daß eine Steuerdatei mit dem Standardnamen (hier `sample.ctl`) für die Top–Down–Konfliktanalyse verwendet werden soll. Verbindungen sollen nicht geprüft werden, auch die Ausgabe von Warnungen soll unterdrückt werden. Als Eingabe dient die MDL–Beschreibung in `sample.mdl`, als Zieldateiname wird ebenfalls der Standardwert verwendet. Daher haben die Ausgabedateien die Namen `sample.des`, `sample.con` und `sample.tdn`. Als Präprozessor wird `/lib/cpp` herangezogen, die Analyse ist fehlerfrei, lediglich 7 Warnungen wurden im Hintergrund gezählt.

sinnvoll genutzt werden: Dann wird die interaktive Eingabe ebenfalls aus der Filtereingabe gelesen. Die Filtereingabe müßte in diesem speziellen Fall die korrekten Daten für die interaktive Eingabe enthalten. Wenn zudem noch die interaktive Top–Down–Konfliktanalyse durchgeführt werden soll, müssen auch dafür die korrekten Daten in der Filtereingabe enthalten sein. Legt man alle diese Daten in einer Datei ab, zum Beispiel in `control`, so genügt der Aufruf von `cat control | mdl -h`, um die gewünschten Analysen durchzuführen. In weiterer Folge werden nun die Details der interaktiven Eingabe sowie der interaktiven Top–Down–Analyse beschrieben, bevor zuletzt noch ein Beispiel für die Filteranwendung gegeben wird.

Um den vollständigen Aufruf des Analysators mit allen Optionen nicht jedesmal komplett angeben zu müssen, sei hier an die zweite Möglichkeit der Verwendung von sogenannten *Shellscripts* erinnert. Diese Methode ermöglicht es, den Analysator mit den entsprechenden Parametern mit einem Befehl aufzurufen, wobei die interaktive Top–Down–Konfliktanalyse im Gegensatz zu vorhin interaktiv bleibt. Diese Methode ist also der Verwendung des Analysators als Filter mit der Option `-h` eindeutig vorzuziehen. Weitere Möglichkeiten in diesem Zusammenhang sind sehr genau in [19] beschrieben.

Abschließend noch ein Beispiel für einen Aufruf des Analysators, wobei die typischen Diagnoseausgaben für einen fehlerfreien Fall gezeigt werden (Beispiel 3.2).

Interaktive Eingabe

Bei der interaktiven Eingabe bestehen dieselben Möglichkeiten wie bei der Eingabe der Optionen, nur werden hier alle Möglichkeiten interaktiv abgefragt. Diese Methode ist selbsterklärend und kann am besten im Beispiel 3.3 mitverfolgt werden. Wird bei den ja/nein–Abfragen eine falsche Eingabe getätigt, so wird der Standard gewählt, wie er auch für die Optionen gilt und in `default.h` festgelegt ist. Immer wenn eine Standardeinstellung gewählt wird, dann wird die getroffene implizite Angabe auch angezeigt.

Top–Down–Analyse

Die Top–Down–Analyse untersucht die Konflikte, die durch die Kombination einiger konkreter Mikrooperationen entstehen. Die Eingabe für diese Analyse ist von sehr einfacher Gestalt: Es werden die Namen der zu kombinierenden Mikrooperationen angegeben, jeweils gefolgt vom Schlüsselwort `#end`. Es können mehrere solcher Listen angegeben werden, bevor mit dem Schlüsselwort `#quit` das Ende der Eingabe angezeigt wird.

Diese Eingabe kann nun entweder interaktiv erfolgen, oder aus einer Steuerdatei. Bei der interaktiven Eingabe folgt zuerst noch eine Abfrage, ob die Ausgabe auf den Bildschirm oder in eine Datei erfolgen soll. Wird die Datei gewählt, kann man entweder einen Namen angeben, oder den Standardnamen akzeptieren, der aus dem Zieldateinamen und der Endung `.itd` zusammengesetzt ist. Beispiel 3.4 zeigt eine MDL–Analyse mit interaktiver Top–Down–Konfliktsuche. Die beiden Formen der Top–Down–Konfliktanalyse beeinflussen einander nicht, sodaß bei einem Aufruf des Analysators sowohl die gesteuerte als auch die interaktive Top–Down–Konfliktanalyse gemeinsam stattfinden können. In Beispiel 3.5 ist zum Vergleich eine Steuerdatei abgebildet. Die Steuerdatei ist speziell konzipiert, um umfangreichere, gegebenenfalls automatisch generierte Testfolgen zu prüfen, während die interaktive Eingabe dazu dient, rasch einige konkrete Kombinationen zu testen.

Bei diesen Top–Down–Konflikttabellen werden, anders als bei der Hauptkonflikttabelle, nur jene Ressourcen aufgezeigt, deren Benutzung einen Konflikt ergibt. Wünscht man alle von den kombinierten Mikrooperationen benutzten Ressourcen aufgelistet, so muß man die Liste der Namen der Mikrooperationen mit dem Schlüsselwort `#endfull` an Stelle von `#end` abschließen. Das funktioniert bei der gesteuerten und bei der interaktiven Top–Down–Konfliktanalyse gleichermaßen.

Zum Schluß noch das versprochene Beispiel der Verwendung des Analysators als Filter mit der Option `-h` und interaktiver Top–Down–Konfliktanalyse. Beispiel 3.6 zeigt die Datei `control`. Mittels Aufruf von `cat control | mdl -I/mike/MDL/mdl -Dswitch=alpha -h` wird dann genau dieselbe Analyse wie in Beispiel 3.3 durchgeführt, nur wird zusätzlich noch eine interaktive Top–Down–Konfliktanalyse analog zu Beispiel 3.4 durchgeführt. Die Datei `control` enthält genau jene Daten in der richtigen Reihenfolge, die vorher händisch eingegeben wurden. Beispiel 3.7 zeigt die entstehende Ausgabe, die hier nicht so leicht lesbar ist, da die Eingabe über den Filter nicht wiedergegeben wird.

```
localhost> mdl -I/mike/MDL/mdl -Dswitch=alpha -h

##### MDL Analyzer
##### interactive control:

enter source file name (extension .mdl assumed):sample

use default destination file name? (y/n):y
default destination file name assumed: sample
generate conflict table? (y/n):x
 default yes assumed
generate descriptor table? (y/n):x
 default yes assumed
use conflict control file? (y/n):y

use default control file name? (y/n):y
default control file name assumed: sample.ctl
enter interactive conflict mode after analysis? (y/n):n

use default maximum cycle time? (y/n):y
default maximum cycle time assumed: 10000
check connections? (y/n):n

print warning diagnostics? (y/n):n

repeat this interactive control input? (y/n):n


##### preprocessing with /lib/cpp -I/mike/MDL/mdl -Dswitch=alpha sample.mdl.

##### starting analysis.

##### analysis finished. [no errors][7 warnings]

##### analysis of sample.mdl.
##### control file sample.ctl.
##### descriptor table sample.des generated.
##### conflict table sample.con generated.
##### top-down conflict table sample.tdn generated.
##### connections ignored.
localhost>
```

Beispiel 3.3: Dieses Beispiel zeigt eine mögliche interaktive Eingabe, die exakt der Eingabe aus Beispiel 3.2 entspricht. Die Optionen für den Präprozessor müssen jedenfalls zwischen `mdl` und `-h` angegeben werden. Die Angabe von Optionen oder Parametern für den Analysator ist sinnlos, wenn `-h` gewählt wird. Nun werden interaktiv alle weiteren Angaben abgefragt. Bei den Fragen, ob Deskriptortabelle und Konflikttabelle erzeugt werden sollen, wurde beidemale die falsche Antwort `x` gegeben. In diesem Fall wird der Standard gewählt (hier 'ja') und auch gemeldet. In jedem Fall muß zumindest irgendeine Eingabe getätigt werden, ein bloßer Zeilenumbruch genügt nicht. Im Zweifelsfalle kann man sich dann eben mit einem `x` helfen, wenn man die Standardeinstellung erreichen will. Die übrigen Eingaben sind selbsterklärend. Wenn man zum Schluß erkennt, daß bei der Eingabe ein Fehler passiert ist, kann mit einem `y` die Eingabe von neuem beginnen, wobei alle bisherigen Einstellungen wieder vergessen werden. Natürlich wäre die Eingabe über ein interaktives Fenster komfortabler, aber, wie bereits erwähnt, war das nicht die Aufgabenstellung an diesen Prototyp des Analysators.

```
localhost> mdl -np -i sample

##### MDL Analyzer

##### preprocessing with /lib/cpp sample.mdl.

##### starting analysis.

##### entering interactive top down analysis:
output to screen? (y/n):n

use default file name? (y/n):y
default file name assumed: sample.itd
enter names of microps now. type #end for a new analysis
and #quit for finishing interactive top down analysis.
add acc #end
add sub #end
acc sub #end
abb
[warning] abb is no microp, ignored!
add acc sub #end
#quit

##### analysis finished. [no errors][7 warnings]

##### analysis of sample.mdl.
##### no control file.
##### descriptor table sample.des generated.
##### conflict table sample.con generated.
##### interactive top-down conflict table sample.itd generated.
##### connections ignored.
localhost>
```

Beispiel 3.4: Ein Beispiel für eine interaktive Top–Down–Konfliktanalyse. Wie man sieht, könnte die Ausgabe auch an den Bildschirm erfolgen. Davon ist aber nur dann sinnvoll Gebrauch zu machen, wenn relativ wenige Konflikte erwartet werden. Ansonsten ist die Ausgabe in eine Datei zu bevorzugen. Die Eingabe der zu kombinierenden Mikrooperationen erfolgt nach denselben Regeln wie bei der Steuerdatei. Der Name `abb` ist nicht als Mikrooperation definiert worden, daher wird er mit einer Warnung ignoriert. Diese Warnung wird ausgegeben, da in diesem Beispiel die Warnmeldungen nicht unterdrückt wurden. Die anderen 6 Warnungen wurden zwar auch ausgegeben, sind aber hier nicht relevant und wurden daher im Beispiel weggelassen. Nach der Eingabe von `#quit` wird die Analyse abgeschlossen.

```
add acc #end
add sub #end
acc sub #end
abb add acc sub #end
#quit
```

Beispiel 3.5: Die Steuerdatei `sample.ctl`: Das Schlüsselwort `#end` zeigt das Ende einer Liste von zu kombinierenden Mikrooperationen an. Die Namen davor sind Namen von Mikrooperationen, die in der zugrundeliegenden MDL–Beschreibung `sample.mdl` definiert wurden. Das Schlüsselwort `#quit` beendet die Analyse. Wird das Ende der Datei erreicht, ohne das Schlüsselwort `#quit` gefunden zu haben, so wird die Analyse ebenfalls beendet. In der vorletzten Zeile wird der Name `abb` verwendet, obwohl in `sample.mdl` gar keine Mikrooperation mit diesem Namen definiert wurde. Dies führt dann bei der Bearbeitung der Steuerdatei zu einer Warnmeldung, der entsprechende Name wird ignoriert. Zwar wurde in Beispiel 3.2 die Ausgabe der Warnungen unterdrückt, dieselbe Warnung kann aber in Besipiel 3.4 bei der interaktiven Analyse beobachtet werden.

```
sample y x x y y y y n n n
n y
add acc #end
add sub #end
acc sub #end
abb add acc sub #end
#quit
```

Beispiel 3.6: Die Datei `control` dient als Eingabe für den Aufruf des Analysators als Filter: Sie enthält in der ersten Zeile genau jene Zeichen, die in Beispiel 3.3 interaktiv eingegeben wurden. Lediglich bei der Frage nach der interaktiven Top–Down–Konfliktanalyse wird jetzt `y` angegeben. Die folgenden Zeilen enthalten dann die Eingabe für die interaktive Konfliktanalyse selbst, wobei hier dieselben Kombinationen wie im Beispiel 3.4 untersucht werden. Die Eingaben `y` und `n` in der zweiten Zeile beantworten die beiden Fragen nach der Ausgabedatei am Beginn der interaktiven Konfliktanalyse.

```
localhost> cat control | mdl -I/mike/MDL/mdl -Dswitch=alpha -h

#####  MDL Analyzer
#####  interactive control:

enter source file name (extension .mdl assumed):
use default destination file name? (y/n):default destination file name assumed: sample
generate conflict table? (y/n): default yes assumed
generate descriptor table? (y/n): default yes assumed
use conflict control file? (y/n):
use default control file name? (y/n):default control file name assumed: sample.ctl
enter interactive conflict mode after analysis? (y/n):
use default maximum cycle time? (y/n):default maximum cycle time assumed: 10000
check connections? (y/n):
print warning diagnostics? (y/n):
repeat this interactive control input? (y/n):

#####  preprocessing with /lib/cpp -I/mike/MDL/mdl -Dswitch=alpha sample.mdl.

#####  starting analysis.

#####  entering interactive top down analysis:
output to screen? (y/n):
use default file name? (y/n):default file name assumed: sample.itd
enter names of microps now. type #end for a new analysis
and #quit for finishing interactive top down analysis.

#####  analysis finished.  [no errors][8 warnings]

#####  analysis of sample.mdl.
#####  control file sample.ctl.
#####  descriptor table sample.des generated.
#####  conflict table sample.con generated.
#####  top-down conflict table sample.tdn generated.
#####  interactive top-down conflict table sample.itd generated.
#####  connections ignored.
localhost>
```

Beispiel 3.7: Die Diagnoseausgabe des Analysators bei Verwendung als Filter mit der Datei `control` gemäß Beispiel 3.6.

3.3 Ausgabe der Tabellen und Meldungen

Die hier zur Illustration als Beispiel verwendete MDL–Beschreibung knüpft an die Beispiele aus dem Kapitel „Sprachbeschreibung“ an. Sie stellt im wesentlichen eine Zusammenfassung der dort präsentierten kurzen Beispiele dar und hat ebenfalls die Aufgabe, in Kürze möglichst viele Varianten vorzustellen. Daher ist sie nicht als Beispiel einer typischen Prozessorbeschreibung zu verstehen, sondern als Kurzfassung der Masse der syntaktischen und semantischen Möglichkeiten. Beispiel 3.8 zeigt diese MDL–Beschreibung zusammengefaßt. Es werden nun die erzeugten Tabellen zu diesem Beispiel vorgestellt, am Ende des Abschnittes wird noch auf die ausgegebenen Fehlermeldungen und Warnungen näher eingegangen.

Deskriptortabelle

Die Deskriptortabelle dient dem *Frontend* des Firmwarecompilers zur Prüfung und Umsetzung hardwarespezifischer Bezeichner. Da die Generierung des Compilers aber nicht mehr Aufgabenstellung der Diplomarbeit war, wurde die Deskriptortabelle vorerst als menschenlesbare Tabelle aller in der MDL–Beschreibung getroffenen Definitionen angelegt. In dieser Form diente sie der Überprüfung der Funktionsweise des Analysators während der Entwicklung. Es bereitet dann keine grundsätzlichen Schwierigkeiten mehr, die in der Deskriptortabelle in ihrer jetzigen Form enthaltene Information so aufzubereiten, daß daraus eine Tabelle für das Frontend des Firmwarecompilers erzeugt werden kann. Diese endgültige Form der Deskriptortabelle wird aber erst dann sinnvoll zu programmieren sein, wenn die Schnittstelle zum Frontend des neuen Firmwarecompilers wohldefiniert ist.

Beispiel 3.9 zeigt einen Auszug der Deskriptortabelle `model.des`, die man durch Analyse der MDL–Beschreibung `model.mdl` aus Beispiel 3.8 erhält. Dabei können sehr gut auch nochmals die im Kapitel „Sprachbeschreibung“ dargestellten Umsetzungen im Zusammenhang mit dem Einlesen einer MDL–Beschreibung nachvollzogen werden. Doch nun zum Aufbau der Deskriptortabelle im Detail.

Nach einem kurzen Tabellenkopf mit der Information, wann und auf Basis welcher MDL–Beschreibung die Tabelle generiert wurde, folgt die Angabe der Register. Am Zeilenanfang ist in den eckigen Klammern angegeben, in welcher Datei in welcher Zeile die zugehörige Definition zu finden ist. Wenn man an die Möglichkeit der Modularisierung mit der Präprozessoranweisung `#include` denkt, erweist sich diese Angabe vor allem bei umfangreicheren Eingaben als sehr hilfreich beim Auffinden von Ungereimtheiten. Danach werden die Register in einer Form wiedergegeben, die abgesehen vom klein geschriebenen Schlüsselwort exakt der Eingabesyntax entspricht: Zuerst der Hardwaremultiplikator, dann der Name, anschließend Felddimensionen und Bitfeldgröße und zum Schluß die Bitfeldstruktur.

Nach den Registern werden die `ALIAS`–Definitionen wiedergegeben: In der ersten Zeile wird der Name gemeinsam mit den erlaubten Bereichen sowie einer möglichen Bitfeldstruktur angegeben, in der zweiten Zeile wird die Zuordnung zum zugehörigen Register

```
REGISTER reg[0:3][0:15]<0:31>
        MULTIPORT 2 READ && 1 WRITE
        MAP word;

MAP fieldmap {<#3> high, low};

FIELD xyz<3,6>;
FIELD a<2:7> MAP fieldmap;                    /* warning */
FIELD u<#6> MAP fieldmap;
FIELD v;  /* no mapping possible */ /* warning */

FIELD d<6:8; 4; 0:1> MAP fieldmap;
FIELD e<5:8; 1:2> MAP fieldmap;

FIELD f OF a, v;

FIELD g OF a.high, <8:9;0> MAP fieldmap;

FIELD k OF <1:2>, a.low, u.high;

FIELD l OF e<4:5>, e.high, u.high;

DEVICE a_bus;
DEVICE b_bus;

PATH { reg -> a_bus -> alu -> b_bus -> reg };
PATH { pointer - portreg };
PATH { unit1 -> c_bus -< portreg };

GROUP unit1 { reg[3][8:9], reg[0:1][5] };

SET pointer { reg[0][2], reg[0][3] };

GROUP unit2 { pointer, align, alu, c_bus, unit1 };

/* time granularity: 1 = 1ns */

#define T 25 /* clock cycle 25ns, clock frequency 40MHz */

#define T1 0,T
#define T2 T,T
#define T3 2*T,T
#define T4 3*T,T

#define Strobe 1000

MICROP  addition;
VAR
    p1,p2 PARTOF pointer;
    reg1, reg2 PARTOF reg;
    op1 PARTOF unit1, portreg;
    op2 PARTOF op1, portreg;
    op3 PARTOF a_bus, b_bus;
```

```
DEVICE alu;     /* arithmetical logical unit */

DEVICE #2 c_bus; /* result bus */

DEVICE align;   /* alignment */

MAP portmap {
    <0,2> high;
    <2,2> low;
};

MAP adr1 {              /* mapping for port field B */
    <#2>   switch;      /* group */
    <3:5>  word;        /* 8 words total*/
    <#3>   digit;       /* 8 digits per word */
    <10>   direction;   /* left = 0, right = 1 */
    <12,4> port         /* port selector */
      MAP portmap;
    <3:8>  dir_digit;   /* 64 digits total */
};

MAP adr2 {              /* mapping for port field A */
    <#2>   switch;      /* group */
    <2:8>  immediate;   /* 0..127 */
    <12,4> port         /* port selector */
      MAP portmap;
    <3:8>  dir_digit;   /* 64 digits total */
};

MAP adr MAP adr1, adr2;   /* mapping for total port field */

MAP sub1  { <#2> a,b };
MAP sub4a { <#2> x4a,y4a };
MAP sub4b { <#2> x4b,y4b };

MAP wbytes {
    <#8> 0,1,2,3  MAP bdigits { <#4> h,l MAP sub1; };
    };

MAP wdigits {
    <#4> 0h,0l,1h,1l MAP sub4a;
    <#4> 2h,2l,3h,3l MAP sub4b;
    };

MAP word OF wbytes, wdigits;

REGISTER #2 io_port;

REGISTER portreg<0:15> MAP adr;
```

Beispiel 3.8: Die MDL–Beschreibung `model.mdl`.

```
FIELD l, v, g.low, xyz<0:1>, <10:12;#4;14>;
USING
    READ p1,p2 TIME (0,12);
    READ reg1, reg2 TIME (T1);
    a_bus, b_bus TIME (T,10);
    alu TIME (T2;T3);
    op1, op2, op3 TIME (Strobe-5,Strobe+3*T);
    unit2 TIME (Strobe+T,4;Strobe+2*T,4);
    c_bus TIME (3*T,10);
    WRITE reg2 TIME (T4);
DATAFLOW
    reg2 -> a_bus -> alu -> c_bus -> reg2;
    reg1 -> b_bus -> alu;
    op1 -> unit2 -> op2;
CONDITION
    (op1 <= unit1 && op2 != op1) ||
    (op1 <! unit1 && op2 == op1);
    ! reg1 <= pointer;
    ! reg2 <= pointer;
ENDOP;

pt0  ALIAS reg[0][2];
lit  ALIAS reg[0][1] MAP {<#16> h,1};

br0 ALIAS pt0<5>;
br1<3> ALIAS pt0<5>;

br2 ALIAS pt0<8:15>;
br3<18:25> ALIAS pt0<8:15>;

br4 ALIAS pt0;
br5<100:131> ALIAS pt0;

br6 ALIAS pt0.1;
br7<18:25> ALIAS pt0.1;

br8a ALIAS br7<0:3>; /* error */
br8b ALIAS br7<18:21>;
br9 ALIAS br7.h;

ar1 ALIAS reg[2][9];

ar2 ALIAS reg[2][8:15];
ar3[4:11] ALIAS reg[2][8:15];

ar4 ALIAS reg[2:3][9];
ar5[1:2] ALIAS reg[2:3][9];

ar6 ALIAS reg[2];
ar7[16:31] ALIAS reg[2];

ar8 ALIAS reg[2:3][8:15];
ar9[1:2] ALIAS reg[2:3][8:15];

ar10[1:2][4:11] ALIAS reg[2:3][8:15];

ar11 ALIAS reg[2:3];
ar12[1:2] ALIAS reg[2:3];
ar13[1:2][16:31] ALIAS reg[2:3];

ar14 ALIAS reg;
ar15[4:7] ALIAS reg;
ar16[4:7][16:31] ALIAS reg;

pt00 ALIAS pt0<0:31>; /* no mapping */

ap1   ALIAS reg[0][2]<8:15>;
bp1   ALIAS reg[0][2].1;
cp1   ALIAS pt0<8:15>;
dp1   ALIAS pt0.1;
ep1   ALIAS pt00<8:15>;
fp1   ALIAS pt00.0; /* error */

ap11  ALIAS reg[0][2]<12:15>;
bp11  ALIAS reg[0][2].1.1;
cp11  ALIAS reg[0][2].11;
dp11  ALIAS pt0<12:15>;
ep11  ALIAS pt0.1.1;
fp11  ALIAS pt0.11;
gp11  ALIAS bp1<4:7>;
hp11  ALIAS bp1.1;
ip11  ALIAS ep1<4:7>;

jp11  ALIAS pt0.5; /* error */
kp11  ALIAS pt0.1.x; /* error */

/* alternative */

pt1 ALIAS reg[0][3];

ALIAS pt1 {
    <#8>  p4,p5,p6,p7 MAP bdigits;
    <0,4> p4h,p41,p5h,p51 MAP sub4a;
    <#4>  p6h,p61,p7h,p71 MAP sub4b;
};

/* alternative */

ALIAS reg[1:3]
    {<#8> m0, m1, m2, m3
     MAP bdigits};
```

Beispiel 3.8 (Fortsetzung): Die MDL–Beschreibung `model.mdl`.

```
symbol table model.des generated by mdl from model.mdl
on Sun Aug 15 20:23:43 GMT+0100 1993

*****************************************************************************
R E G I S T E R S
*****************************************************************************

    [file model.mdl, line 50] register #2 io_port;
    [file model.mdl, line 52] register #1 portreg<  0: 15> map adr;
    [file model.mdl, line 54] register #1 reg(  0:  3)(  0: 15)<  0: 31> map word;

*****************************************************************************
A L I A S E S
*****************************************************************************

    [file model.mdl, line 128] pt0<  0: 31> map word
      is alias reg(  0:  0)(  2:  2)<  0: 31>;
    [file model.mdl, line 129] lit<  0: 31> map $50006
      is alias reg(  0:  0)(  1:  1)<  0: 31>;
    [file model.mdl, line 131] br0<  0:  0>
      is alias reg(  0:  0)(  2:  2)<  5:  5>;
    [file model.mdl, line 132] br1<  3:  3>
      is alias reg(  0:  0)(  2:  2)<  5:  5>;
    [file model.mdl, line 134] br2<  0:  7>
      is alias reg(  0:  0)(  2:  2)<  8: 15>;
    [file model.mdl, line 135] br3< 18: 25>
      is alias reg(  0:  0)(  2:  2)<  8: 15>;
    [file model.mdl, line 137] br4<  0: 31> map word
      is alias reg(  0:  0)(  2:  2)<  0: 31>;
    [file model.mdl, line 138] br5<100:131> map word
      is alias reg(  0:  0)(  2:  2)<  0: 31>;
    [file model.mdl, line 140] br6<  0:  7> map bdigits
      is alias reg(  0:  0)(  2:  2)<  8: 15>;
    [file model.mdl, line 141] br7< 18: 25> map bdigits
      is alias reg(  0:  0)(  2:  2)<  8: 15>;
    [file model.mdl, line 143] br8a<  0:  7>
      is alias reg(  0:  0)(  2:  2)<  8: 15>;
    [file model.mdl, line 144] br8b<  0:  3>
      is alias reg(  0:  0)(  2:  2)<  8: 11>;
    [file model.mdl, line 145] br9<  0:  3> map sub1
      is alias reg(  0:  0)(  2:  2)<  8: 11>;
    [file model.mdl, line 147] ar1<  0: 31> map word
      is alias reg(  2:  2)(  9:  9)<  0: 31>;
    [file model.mdl, line 149] ar2(  0:  7)<  0: 31> map word
      is alias reg(  2:  2)(  8: 15)<  0: 31>;
    [file model.mdl, line 150] ar3(  4: 11)<  0: 31> map word
      is alias reg(  2:  2)(  8: 15)<  0: 31>;
    [file model.mdl, line 152] ar4(  0:  1)<  0: 31> map word
      is alias reg(  2:  3)(  9:  9)<  0: 31>;
    [file model.mdl, line 153] ar5(  1:  2)<  0: 31> map word
      is alias reg(  2:  3)(  9:  9)<  0: 31>;
    [file model.mdl, line 155] ar6(  0: 15)<  0: 31> map word
      is alias reg(  2:  2)(  0: 15)<  0: 31>;
    [file model.mdl, line 156] ar7( 16: 31)<  0: 31> map word
      is alias reg(  2:  2)(  0: 15)<  0: 31>;
    [file model.mdl, line 158] ar8(  0:  1)(  0:  7)<  0: 31> map word
      is alias reg(  2:  3)(  8: 15)<  0: 31>;
    [file model.mdl, line 159] ar9(  1:  2)(  0:  7)<  0: 31> map word
      is alias reg(  2:  3)(  8: 15)<  0: 31>;
    [file model.mdl, line 160] ar10(  1:  2)(  4: 11)<  0: 31> map word
      is alias reg(  2:  3)(  8: 15)<  0: 31>;
    [file model.mdl, line 162] ar11(  0:  1)(  0: 15)<  0: 31> map word
      is alias reg(  2:  3)(  0: 15)<  0: 31>;
    [file model.mdl, line 163] ar12(  1:  2)(  0: 15)<  0: 31> map word
      is alias reg(  2:  3)(  0: 15)<  0: 31>;
    [file model.mdl, line 164] ar13(  1:  2)( 16: 31)<  0: 31> map word
      is alias reg(  2:  3)(  0: 15)<  0: 31>;
    [file model.mdl, line 166] ar14(  0:  3)(  0: 15)<  0: 31> map word
      is alias reg(  0:  3)(  0: 15)<  0: 31>;
    [file model.mdl, line 167] ar15(  4:  7)(  0: 15)<  0: 31> map word
      is alias reg(  0:  3)(  0: 15)<  0: 31>;
    [file model.mdl, line 168] ar16(  4:  7)( 16: 31)<  0: 31> map word
      is alias reg(  0:  3)(  0: 15)<  0: 31>;
    [file model.mdl, line 170] pt00<  0: 31>
      is alias reg(  0:  0)(  2:  2)<  0: 31>;
    [file model.mdl, line 172] ap1<  0:  7>
      is alias reg(  0:  0)(  2:  2)<  8: 15>;
    [file model.mdl, line 173] bp1<  0:  7> map bdigits
      is alias reg(  0:  0)(  2:  2)<  8: 15>;
    [file model.mdl, line 174] cp1<  0:  7>
      is alias reg(  0:  0)(  2:  2)<  8: 15>;
    [file model.mdl, line 175] dp1<  0:  7> map bdigits
      is alias reg(  0:  0)(  2:  2)<  8: 15>;
    [file model.mdl, line 176] ep1<  0:  7>
      is alias reg(  0:  0)(  2:  2)<  8: 15>;
    [file model.mdl, line 177] fp1<  0: 31>
      is alias reg(  0:  0)(  2:  2)<  0: 31>;
    [file model.mdl, line 179] ap11<  0:  3>
      is alias reg(  0:  0)(  2:  2)< 12: 15>;
    [file model.mdl, line 180] bp11<  0:  3> map sub1
      is alias reg(  0:  0)(  2:  2)< 12: 15>;
    [file model.mdl, line 181] cp11<  0:  3> map sub4a
      is alias reg(  0:  0)(  2:  2)< 12: 15>;
    [file model.mdl, line 182] dp11<  0:  3>
      is alias reg(  0:  0)(  2:  2)< 12: 15>;
    [file model.mdl, line 183] ep11<  0:  3> map sub1
      is alias reg(  0:  0)(  2:  2)< 12: 15>;
    [file model.mdl, line 184] fp11<  0:  3> map sub4a
      is alias reg(  0:  0)(  2:  2)< 12: 15>;
    [file model.mdl, line 185] gp11<  0:  3>
      is alias reg(  0:  0)(  2:  2)< 12: 15>;
    [file model.mdl, line 186] hp11<  0:  3> map sub1
      is alias reg(  0:  0)(  2:  2)< 12: 15>;
    [file model.mdl, line 187] ip11<  0:  3>
      is alias reg(  0:  0)(  2:  2)< 12: 15>;
    [file model.mdl, line 189] jp11<  0: 31>
      is alias reg(  0:  0)(  2:  2)<  0: 31>;
    [file model.mdl, line 190] kp11<  0:  7>
      is alias reg(  0:  0)(  2:  2)<  8: 15>;
    [file model.mdl, line 194] pt1<  0: 31> map word
      is alias reg(  0:  0)(  3:  3)<  0: 31>;
    [file model.mdl, line 196] p4<  0:  7> map bdigits
      is alias reg(  0:  0)(  3:  3)<  0:  7>;
    [file model.mdl, line 196] p5<  0:  7> map bdigits
      is alias reg(  0:  0)(  3:  3)<  8: 15>;
    [file model.mdl, line 196] p6<  0:  7> map bdigits
      is alias reg(  0:  0)(  3:  3)< 16: 23>;
```

Beispiel 3.9: Die Deskriptortabelle `model.des`.

```
[file model.mdl, line 196] p7<  0:  7> map bdigits
   is alias reg(  0:  0)(  3:  3)< 24: 31>;
[file model.mdl, line 196] p4h<  0:  3> map sub4a
   is alias reg(  0:  0)(  3:  3)<  0:  3>;
[file model.mdl, line 196] p41<  0:  3> map sub4a
   is alias reg(  0:  0)(  3:  3)<  4:  7>;
[file model.mdl, line 196] p5h<  0:  3> map sub4a
   is alias reg(  0:  0)(  3:  3)<  8: 11>;
[file model.mdl, line 196] p51<  0:  3> map sub4a
   is alias reg(  0:  0)(  3:  3)< 12: 15>;
[file model.mdl, line 196] p6h<  0:  3> map sub4b
   is alias reg(  0:  0)(  3:  3)< 16: 19>;
[file model.mdl, line 196] p61<  0:  3> map sub4b
   is alias reg(  0:  0)(  3:  3)< 20: 23>;
[file model.mdl, line 196] p7h<  0:  3> map sub4b
   is alias reg(  0:  0)(  3:  3)< 24: 27>;
[file model.mdl, line 196] p71<  0:  3> map sub4b
   is alias reg(  0:  0)(  3:  3)< 28: 31>;
[file model.mdl, line 204] m0(  0:  2)(  0: 15)<  0:  7> map bdigits
   is alias reg(  1:  3)(  0: 15)<  0:  7>;
[file model.mdl, line 204] m1(  0:  2)(  0: 15)<  0:  7> map bdigits
   is alias reg(  1:  3)(  0: 15)<  8: 15>;
[file model.mdl, line 204] m2(  0:  2)(  0: 15)<  0:  7> map bdigits
   is alias reg(  1:  3)(  0: 15)< 16: 23>;
[file model.mdl, line 204] m3(  0:  2)(  0: 15)<  0:  7> map bdigits
   is alias reg(  1:  3)(  0: 15)< 24: 31>;

*****************************************************************************
S E T S
*****************************************************************************

[file model.mdl, line 85] set pointer
{
   register reg(  0:  0)(  2:  2)
   register reg(  0:  0)(  3:  3)
};

*****************************************************************************
D E V I C E S
*****************************************************************************

[file model.mdl, line 2] device #1 alu;
[file model.mdl, line 4] device #2 c_bus;
[file model.mdl, line 6] device #1 align;
[file model.mdl, line 76] device #1 a_bus;
[file model.mdl, line 77] device #1 b_bus;

*****************************************************************************
G R O U P S
*****************************************************************************

[file model.mdl, line 83] group unit1
{
   register reg(  3:  3)(  8:  8)
   register reg(  3:  3)(  9:  9)
   register reg(  0:  0)(  5:  5)
   register reg(  1:  1)(  5:  5)
};
[file model.mdl, line 87] group unit2
{
   register reg(  0:  0)(  2:  2)
   register reg(  0:  0)(  3:  3)
   device align
   device alu
   device c_bus
   register reg(  3:  3)(  8:  8)
   register reg(  3:  3)(  9:  9)
   register reg(  0:  0)(  5:  5)
   register reg(  1:  1)(  5:  5)
};

*****************************************************************************
F I E L D S
*****************************************************************************

[file model.mdl, line 60] field xyz<  3:  8> with length 6;
[file model.mdl, line 61] field a<  2:  7> with length 6 map fieldmap;
[file model.mdl, line 62] field u< -6: -1> with length 6 map fieldmap;
[file model.mdl, line 63] field v< -7: -7> with length 1;
[file model.mdl, line 65] field d<  6:  8>< 4:  4>< 0:  1> with length 6 map
fieldmap;
[file model.mdl, line 66] field e<  5:  8>< 1:  2> with length 6 map fieldmap;
[file model.mdl, line 68] field f<  2:  7>< -7: -7> with length 7;
[file model.mdl, line 70] field g<  2:  4>< 8:  9>< 0:  0> with length 6 map
fieldmap;
[file model.mdl, line 72] field k<  1:  2>< 5:  7>< -6: -4> with length 8;
[file model.mdl, line 74] field l<  1:  2>< 5:  7>< -6: -4> with length 8;

*****************************************************************************
M A P P I N G S
*****************************************************************************

[file model.mdl, line 8] map portmap<  0:  3>
{
   <  0:  1> high
   <  2:  3> low
};
[file model.mdl, line 13] map adr1<  0: 15>
{
   <  0:  1> switch
   <  3:  5> word
   <  6:  8> digit
   < 10: 10> direction
   < 12: 15> port map portmap
   <  3:  8> dir_digit
};
[file model.mdl, line 24] map adr2<  0: 15>
{
   <  0:  1> switch
   <  2:  8> immediate
   < 12: 15> port map portmap
   <  3:  8> dir_digit
};
[file model.mdl, line 33] map adr<  0: 15>
{
   <  0:  1> switch
   <  3:  5> word
   <  6:  8> digit
```

Beispiel 3.9 (Fortsetzung): Die Deskriptortabelle `model.des`.

```
  < 10: 10> direction
  < 12: 15> port map portmap
    < 3:  8> dir_digit
    < 2:  8> immediate
  };
  [file model.mdl, line 35] map subl<  0:  3>
  {
    < 0:  1> a
    < 2:  3> b
  };
  [file model.mdl, line 36] map sub4a<  0:  3>
  {
    < 0:  1> x4a
    < 2:  3> y4a
  };
  [file model.mdl, line 37] map sub4b<  0:  3>
  {
    < 0:  1> x4b
    < 2:  3> y4b
  };
  [file model.mdl, line 39] map wbytes<  0: 31>
  {
    < 0:  7> 0 map bdigits
    < 8: 15> 1 map bdigits
    < 16: 23> 2 map bdigits
    < 24: 31> 3 map bdigits
  };
  [file model.mdl, line 40] map bdigits<  0:  7>
  {
    < 0:  3> h map subl
    < 4:  7> l map subl
  };
  [file model.mdl, line 43] map wdigits<  0: 31>
  {
    < 0:  3> 0h map sub4a
    < 4:  7> 0l map sub4a
    < 8: 11> 1h map sub4a
    < 12: 15> 1l map sub4a
    < 16: 19> 2h map sub4b
    < 20: 23> 2l map sub4b
    < 24: 27> 3h map sub4b
    < 28: 31> 3l map sub4b
  };
  [file model.mdl, line 48] map word<  0: 31>
  {
    < 0:  7> 0 map bdigits
    < 8: 15> 1 map bdigits
    < 16: 23> 2 map bdigits
    < 24: 31> 3 map bdigits
    < 0:  3> 0h map sub4a
    < 4:  7> 0l map sub4a
    < 8: 11> 1h map sub4a
    < 12: 15> 1l map sub4a
    < 16: 19> 2h map sub4b
    < 20: 23> 2l map sub4b
    < 24: 27> 3h map sub4b
    < 28: 31> 3l map sub4b
  };
  [file model.mdl, line 58] map fieldmap<  0:  5>
  {
    < 0:  2> high
    < 3:  5> low
  };
  [file model.mdl, line 129] map $$0005<  0: 31>
  {
    < 0: 15> h
    < 16: 31> l
  };
  [file model.mdl, line 129] map $$0006<  0: 31>
  {
    < 0: 15> h
    < 16: 31> l
    < 0:  7> 0 map bdigits
    < 8: 15> 1 map bdigits
    < 16: 23> 2 map bdigits
    < 24: 31> 3 map bdigits
    < 0:  3> 0h map sub4a
    < 4:  7> 0l map sub4a
    < 8: 11> 1h map sub4a
    < 12: 15> 1l map sub4a
    < 16: 19> 2h map sub4b
    < 20: 23> 2l map sub4b
    < 24: 27> 3h map sub4b
    < 28: 31> 3l map sub4b
  };
```

Beispiel 3.9 (Fortsetzung): Die Deskriptortabelle `model.des`.

angegeben. Durch Vergleich der `ALIAS`–Definitionen mit den Ausgaben in der Deskriptortabelle kann man die teilweise nichttrivialen Umsetzungen sehr gut mitverfolgen, die im Kapitel „Sprachbeschreibung“ erläutert wurden. Wenn Bezeichner auftreten, die mit `$$` beginnen, so handelt es sich um sogenannte *anonyme Bezeichner*, die der Analysator selbst vergeben hat. Ein typisches Beispiel dafür sind `MAP`–Strukturen, die im Zuge der `MAP`–Vererbung implizit aufgebaut wurden. Die `map $$0006` ist eine solche implizit im Zuge der `ALIAS`–Definition von `lit` entstandene Bitfeldstruktur.

Registernamen und Aliasnamen stellen gemeinsam jene Informationen dar, die für den Firmwarecompiler relevant sind. Die folgenden Angaben dienen nur der Information des Benutzers und zur Überprüfung der Funktion des Analysators. Die Angabe der `SET-`, `DEVICE-` und `GROUP`–Definitionen erfolgt analog zum bisher Gesagten.

Interessant ist wieder die Darstellung der Ausschnitte des Mikrobefehlsfeldes: Nach dem Namen erfolgt die Angabe des Mehrfachbereiches als Auflistung mehrerer einfacher Bereiche. Den Schluß macht die Angabe der Gesamtlänge des Feldes sowie der möglichen Bitfeldstruktur. Betrachtet man die einfachen Bereiche genauer, so fallen vor allem einige Bereiche mit *negativen* Bereichsgrenzen auf. Der Grund dafür ist relativ einfach: Der Analysator legt Felder mit unbekanntem Offset einfach links vom Nullpunkt ab. Dabei erhält jedes *anonyme Feld* seinen eigenen einzigartigen Platz links von Null, sodaß es zwischen verschiedenen anonymen Feldern zu keiner Überlappung kommen kann. Der Vergleich dieser Ausgaben mit den zugehörigen Definitionen erleichtert auch hier das Verständnis für die manchmal nicht sofort offensichtlichen Zusammenhänge.

Den Abschluß bildet die Angabe aller Bitfeldstrukturdefinitionen. Dabei hält sich die Ausgabe weitgehend an die Syntax der Eingabe, dennoch gibt es einen kleinen Unterschied: Die Gesamtgröße der `MAP` wird am Ende der ersten Zeile angegeben.

Abschließend noch eine Bemerkung: Die Deskriptortabelle sollte auch dann in ihrer jetzigen Form bestehen bleiben, wenn bereits die Tabellen für das Compilerfrontend generiert werden. Der Grund dafür ist, daß die Deskriptortabelle in der momentanen Gestalt nichts anderes ist als ein Abbild der internen Symboltabelle des Analysators, und sich daher besonders gut für die Überprüfung der Funktion des Analysators eignet.

Konflikttabellen

Die Konflikttabelle dient dem *Backend* des Firmwarecompilers zur Durchführung der semantischen Prüfungen. Sie wurde ebenfalls menschenlesbar ausgeführt, da sie ja wesentlich für das Feedback des Hardwareentwicklers ist. Für den Einbau dieser semantischen Prüfungen in den generierten neuen Firmwarecompiler genügt es vermutlich nicht, nur Tabellen zu generieren: Für eine effiziente Implementierung der semantischen Regeln werden aus der MDL–Beschreibung auch einige Unterprogramme des Compilers selbst generiert werden müssen.

Zurück zur Konflikttabelle in ihrer jetzigen Form. Dabei handelt es sich um eine Bottom–Up–Konflikttabelle: Das bedeutet, daß die Konflikte an den Ressourcen selbst (also von *unten* her) untersucht werden und nicht die Kombination spezieller Mikrooperationen (von *oben* her). Dabei werden für jede einzelne Hardwareressource alle Konflikte

aufgezeigt, die zwischen irgendwelchen Mikrooperationen entstehen können. Beispiel 3.10 zeigt die Konflikttabelle `model.con`. Diese Tabelle ist für gewöhnlich sehr umfangreich, sodaß hier nur ein kurzer Auszug vorgestellt und beschrieben wird.

Nach dem Tabellenkopf folgt die Beschreibung der Auslastung des Mikrobefehlsfeldes: Zuerst wird angegeben, welche Bitbereiche von welchen Mikrooperationen benötigt werden, anschließend werden die dadurch möglichen Konflikte aufgezeigt. Aus diesen Konflikten ergibt sich bereits, welche Mikrooperationen einander zwingend ausschließen. Im Fall des hier gewählten Beipieles gibt es keine Konflikte, da ja eine Mikrooperation allein keine Konflikte bei der Nutzung des Mikrobefehlsfeldes verursachen darf. Im allgemeinen Fall der Definition vieler Mikrooperationen aber wird vor allem die Nutzung des Mikrobefehlsfeldes schon eine erste Klasseneinteilung der Mikrooperationen nach sich ziehen.

Danach folgt die Beschreibung der Mikrooperationen, hier nur eine einzige. Am Anfang wird angegeben, welche Bitbereiche des Mikrobefehlsfeldes die Mikrooperation benötigt und welche minimale Mikrobefehlsbreite sich für diese Mikrooperation daraus ergibt. Dann erfolgt eine Aufzählung der definierten Ressourcenvariablen, auf die hier nicht näher eingegangen werden soll, da erst mit der Syntax/Code–Zuordnung die endgültige Handhabung der Ressourcenvariablen feststeht. Danach folgt die Angabe, welche Ressourcen von wann bis wann auf welche Art und Weise benutzt werden. Dabei fällt auf, daß Registerfelder in einzelne Register aufgespalten wurden, da ja jedes Register für sich eine eigene unteilbare Ressource darstellt. Für dieses Beispiel wurde nur die Nutzung von drei Registern beschrieben, nämlich `reg[1][6]`, `reg[1][5]` sowie `reg[0][2]`. Die Information über die übrigen Register wurde weggelassen, da die Tabelle sonst deutlich länger gewesen wäre. Den Abschluß der Beschreibung der Mikrooperation bildet der Datenfluß, der hier aber wegen der Verwendung der Option `-np` ignoriert wird.

Die Darstellung der Mikrooperationen bedeutet eigentlich einen Bruch in der Bottom–Up–Darstellung. Sie hat sich aber bewährt, um die Beziehung zwischen Mikrooperationen und Ressourcen besser mitverfolgen zu können. Nach der Beschreibung der Mikrooperationen erfolgt noch die Beschreibung der eigentlichen Hardwareressourcen, nämlich der einfachen Hardwareelemente und der Register.

Für jede Hardwareressource wird nach der kurzen Wiedergabe ihrer Definition zuerst angegeben, welche Mikrooperationen in welchen Zeiträumen die Ressource auf welche Art und Weise benutzen. Bei den Hardwareelementen gibt es dabei nur die Möglichkeiten `use` und `use partly`: `use` bedeutet die sichere totale Inanspruchnahme der Ressource, während `use partly` die mögliche Inanspruchnahme der Ressource im Zuge eines Konfliktes zweiter Art bedeutet. Im Anschluß daran werden die dadurch entstehenden Konflikte nach Zeitintervallen geordnet ausgegeben und zuletzt noch die definierten Verbindungen wiederholt, sofern nicht, so wie in diesem Beispiel, die Option `-np` angegeben wurde.

Bei der Benutzung der Register durch die Mikrooperationen gibt es neben der Benutzungsart `use` auch noch die Möglichkeiten `read` und `write`. Die dabei geltenden Regeln wurden bei der Beschreibung der *Multiportregister* im vorigen Kapitel angegeben. Vor allem bei Registern sind Konflikte zweiter Art sehr häufig, in diesem Fall ist als Be-

```
conflict table model.con generated by mdl from model.mdl
on Mon Aug 16 17:45:51 GMT+0100 1993

*****************************************************************************
B I T F I E L D   O F   M I C R O I N S T R U C T I O N
*****************************************************************************

[ ] field $$field
    used by:
    [file model.mdl, line 100] micro operation addition from -11 to -8 (use)
    [file model.mdl, line 100] micro operation addition from -7 to -7 (use)
    [file model.mdl, line 100] micro operation addition from -6 to -4 (use)
    [file model.mdl, line 100] micro operation addition from 0 to 0 (use)
    [file model.mdl, line 100] micro operation addition from 1 to 2 (use)
    [file model.mdl, line 100] micro operation addition from 3 to 4 (use)
    [file model.mdl, line 100] micro operation addition from 5 to 7 (use)
    [file model.mdl, line 100] micro operation addition from 8 to 9 (use)
    [file model.mdl, line 100] micro operation addition from 10 to 12 (use)
    [file model.mdl, line 100] micro operation addition from 14 to 14 (use)
    no conflicts reported

*****************************************************************************
M I C R O O P E R A T I O N S
*****************************************************************************

[file model.mdl, line 100] micro operation addition
    minimum field length 22
        with absolute field locations:
        <-11: -8>< -7: -7>< -6: -4><  0:  0><  1:  2><  3:  4><  5:  7><  8:  9>< 10: 12><
14: 14>
    with parameters:
        p1 defined in group $$0000
        p2 defined in group $$0000
        reg1 defined in group $$0001
        reg2 defined in group $$0001
        op1 defined in group $$0002
        op2 defined in group $$0003
        op3 defined in group $$0004
    uses:
    [file model.mdl, line 54] register reg(  0:  0)(  2:  2) from 0 to 12 (read partly)
    [file model.mdl, line 54] register reg(  0:  0)(  2:  2) from 0 to 12 (read partly)
    [file model.mdl, line 54] register reg(  1:  1)(  6:  6) from 0 to 25 (read partly)
    [file model.mdl, line 54] register reg(  1:  1)(  5:  5) from 0 to 25 (read partly)
    [file model.mdl, line 54] register reg(  0:  0)(  2:  2) from 0 to 25 (read partly)
    [file model.mdl, line 54] register reg(  1:  1)(  6:  6) from 0 to 25 (read partly)
    [file model.mdl, line 54] register reg(  1:  1)(  5:  5) from 0 to 25 (read partly)
    [file model.mdl, line 54] register reg(  0:  0)(  2:  2) from 0 to 25 (read partly)
    [file model.mdl, line 76] device a_bus from 25 to 35 (use)
    [file model.mdl, line 77] device b_bus from 25 to 35 (use)
    [file model.mdl, line 2] device alu from 25 to 50 (use)
    [file model.mdl, line 2] device alu from 50 to 75 (use)
    [file model.mdl, line 4] device c_bus from 75 to 85 (use)
    [file model.mdl, line 54] register reg(  1:  1)(  6:  6) from 75 to 100 (write partly)
    [file model.mdl, line 54] register reg(  1:  1)(  5:  5) from 75 to 100 (write partly)
    [file model.mdl, line 54] register reg(  0:  0)(  2:  2) from 75 to 100 (write partly)
    [file model.mdl, line 54] register reg(  1:  1)(  5:  5) from 995 to 2070 (use partly)
    [file model.mdl, line 54] register reg(  1:  1)(  5:  5) from 995 to 2070 (use partly)
    [file model.mdl, line 77] device b_bus from 995 to 2070 (use partly)
    [file model.mdl, line 76] device a_bus from 995 to 2070 (use partly)
    [file model.mdl, line 54] register reg(  1:  1)(  5:  5) from 1025 to 1029 (use)
    [file model.mdl, line 4] device c_bus from 1025 to 1029 (use)
    [file model.mdl, line 2] device alu from 1025 to 1029 (use)
    [file model.mdl, line 6] device align from 1025 to 1029 (use)
    [file model.mdl, line 54] register reg(  0:  0)(  2:  2) from 1025 to 1029 (use)
    [file model.mdl, line 54] register reg(  1:  1)(  5:  5) from 1050 to 1054 (use)
    [file model.mdl, line 4] device c_bus from 1050 to 1054 (use)
```

Beispiel 3.10: Die Konflikttabelle `model.con`.

```
    [file model.mdl, line 2] device alu from 1050 to 1054 (use)
    [file model.mdl, line 6] device align from 1050 to 1054 (use)
    [file model.mdl, line 54] register reg(  0:  0)(  2:  2) from 1050 to 1054 (use)
    without any dataflow

*******************************************************************************
D E V I C E S
*******************************************************************************

[file model.mdl, line 4] device #2 c_bus
    used by:
    [file model.mdl, line 100] micro operation addition from 75 to 85 (use)
    [file model.mdl, line 100] micro operation addition from 1025 to 1029 (use)
    [file model.mdl, line 100] micro operation addition from 1050 to 1054 (use)
    no conflicts reported
    without any connections

-------------------------------------------------------------------------------
[file model.mdl, line 6] device #1 align
    used by:
    [file model.mdl, line 100] micro operation addition from 1025 to 1029 (use)
    [file model.mdl, line 100] micro operation addition from 1050 to 1054 (use)
    no conflicts reported
    without any connections

-------------------------------------------------------------------------------
[file model.mdl, line 2] device #1 alu
    used by:
    [file model.mdl, line 100] micro operation addition from 25 to 50 (use)
    [file model.mdl, line 100] micro operation addition from 50 to 75 (use)
    [file model.mdl, line 100] micro operation addition from 1025 to 1029 (use)
    [file model.mdl, line 100] micro operation addition from 1050 to 1054 (use)
    no conflicts reported
    without any connections

-------------------------------------------------------------------------------
[file model.mdl, line 77] device #1 b_bus
    used by:
    [file model.mdl, line 100] micro operation addition from 25 to 35 (use)
    [file model.mdl, line 100] micro operation addition from 995 to 2070 (use partly)
    no conflicts reported
    without any connections

-------------------------------------------------------------------------------
[file model.mdl, line 76] device #1 a_bus
    used by:
    [file model.mdl, line 100] micro operation addition from 25 to 35 (use)
    [file model.mdl, line 100] micro operation addition from 995 to 2070 (use partly)
    no conflicts reported
    without any connections

*******************************************************************************
R E G I S T E R S
*******************************************************************************

[file model.mdl, line 54] register #1 reg(  1:  1)(  6:  6)
    used by:
    [file model.mdl, line 100] micro operation addition from 0 to 25 (read partly)
    [file model.mdl, line 100] micro operation addition from 0 to 25 (read partly)
    [file model.mdl, line 100] micro operation addition from 75 to 100 (write partly)
    conflicts:
        interval from 0 to 25:
            [file model.mdl, line 100] micro operation addition (read partly)
            [file model.mdl, line 100] micro operation addition (read partly)
    without any connections

-------------------------------------------------------------------------------
```

Beispiel 3.10 (Fortsetzung): Die Konflikttabelle `model.con`.

```
[file model.mdl, line 54] register #1 reg(  1:  1)(  5:  5)
   used by:
    [file model.mdl, line 100] micro operation addition from 0 to 25 (read partly)
    [file model.mdl, line 100] micro operation addition from 0 to 25 (read partly)
    [file model.mdl, line 100] micro operation addition from 75 to 100 (write partly)
    [file model.mdl, line 100] micro operation addition from 995 to 2070 (use partly)
    [file model.mdl, line 100] micro operation addition from 995 to 2070 (use partly)
    [file model.mdl, line 100] micro operation addition from 1025 to 1029 (use)
    [file model.mdl, line 100] micro operation addition from 1050 to 1054 (use)
    conflicts:
        interval from 0 to 25:
            [file model.mdl, line 100] micro operation addition (read partly)
            [file model.mdl, line 100] micro operation addition (read partly)
        interval from 995 to 1025:
            [file model.mdl, line 100] micro operation addition (use partly)
            [file model.mdl, line 100] micro operation addition (use partly)
        interval from 1025 to 1029:
            [file model.mdl, line 100] micro operation addition (use partly)
            [file model.mdl, line 100] micro operation addition (use partly)
            [file model.mdl, line 100] micro operation addition (use)
        interval from 1029 to 1050:
            [file model.mdl, line 100] micro operation addition (use partly)
            [file model.mdl, line 100] micro operation addition (use partly)
        interval from 1050 to 1054:
            [file model.mdl, line 100] micro operation addition (use partly)
            [file model.mdl, line 100] micro operation addition (use partly)
            [file model.mdl, line 100] micro operation addition (use)
        interval from 1054 to 2070:
            [file model.mdl, line 100] micro operation addition (use partly)
            [file model.mdl, line 100] micro operation addition (use partly)
    without any connections

--------------------------------------------------------------------------------
[file model.mdl, line 54] register #1 reg(  0:  0)(  2:  2)
   used by:
    [file model.mdl, line 100] micro operation addition from 0 to 25 (read partly)
    [file model.mdl, line 100] micro operation addition from 0 to 25 (read partly)
    [file model.mdl, line 100] micro operation addition from 0 to 12 (read partly)
    [file model.mdl, line 100] micro operation addition from 0 to 12 (read partly)
    [file model.mdl, line 100] micro operation addition from 75 to 100 (write partly)
    [file model.mdl, line 100] micro operation addition from 1025 to 1029 (use)
    [file model.mdl, line 100] micro operation addition from 1050 to 1054 (use)
    conflicts:
        interval from 0 to 12:
            [file model.mdl, line 100] micro operation addition (read partly)
            [file model.mdl, line 100] micro operation addition (read partly)
            [file model.mdl, line 100] micro operation addition (read partly)
            [file model.mdl, line 100] micro operation addition (read partly)
        interval from 12 to 25:
            [file model.mdl, line 100] micro operation addition (read partly)
            [file model.mdl, line 100] micro operation addition (read partly)
    without any connections
```

Beispiel 3.10 (Fortsetzung): Die Konflikttabelle `model.con`.

nutzungsart `read partly` oder `write partly` angegeben. Das Register `reg[1][5]` zeigt zum Beispiel einige interessante Konfliktsituationen.

Ob es nun tatsächlich zu einem Konflikt kommt, ist nur bei einem Konflikt erster Art im Analysator selbst überprüfbar und hängt vom Hardwaremultiplikator sowie von den Multiporteigenschaften des betreffenden Registers ab. Für die Konflikte zweiter Art können hingegen vom Analysator selbst *keine* Aussagen getroffen werden, vielmehr müssen vom Analysator Tabellen und Unterprogrammfunktionen für den Compiler generiert werden, sodaß die Konflikte zweiter Art im Compiler bei Vorliegen eines konkreten Firmwareprogrammes entsprechend geprüft werden können. Zur Illustration kann man den Zeitbereich 1025 bis 1029 bei den Konflikten des Registers `reg[1][5]` betrachten: In diesem Zeitbereich wird das Register einmal total und zweimal partiell benötigt. Wenn nun ein konkretes Firmwareprogramm jenen Ressourcenvariablen, die zur partiellen Nutzung geführt haben, ebenfalls dieses Register zuweist, kommt es zu einem Konflikt. Wenn hingegen die konkrete Eingabe den Ressourcenvariablen andere Register zuordnet, kommt es zu keinem Konflikt. Das ist eben das Wesen der Konflikte zweiter Art, daß sie erst zur Übersetzungszeit im Compiler konkret geprüft werden können.

Abschließend noch eine Bemerkung zu den Top–Down–Konflikttabellen. Diese besitzen denselben Aufbau wie die Bottom–Up–Konflikttabelle, nur daß eben nicht die Konflikte aller Mikrooperationen geprüft werden, sondern nur die Kombination einiger weniger konkreter Mikrooperationen. Daher auch die Bezeichnung Top–Down, da jetzt das Hauptaugenmerk *oben* bei den Mikrooperationen und nicht *unten* bei den Ressourcen liegt.

Meldungen

Bei einem Aufruf des Analysators werden nicht nur die eben besprochenen Tabellen erzeugt, im allgemeinen erhält man auch eine Anzahl von Meldungen an die Standardfehlerausgabe, die meistens gleich der Standardausgabe ist. Dabei gibt es vier Arten von Meldungen:

`error` Eine *Fehlermeldung* erhält man dann, wenn der Analysator eine Ungereimtheit entdeckt, die er durch bestimmte implizite Annahmen ausgleichen muß. Im Unterschied zur Warnung liegt ein Fehler dann vor, wenn die erkannte Eingabe einen Widerspruch enthält.

`warning` Die *Warnmeldung* ist mit der Fehlermeldung eng verwandt: Der Analysator muß ebenfalls eine implizite Annahme treffen. Nur ist der erkannte Sachverhalt hier so gelegen, daß angenommen werden kann, daß nur eine unvollständige oder fehlende Angabe zu dem Problem geführt hat.

`fatal` Diese Meldung beschreibt einen *fatalen* Fehler, der zum sofortigen Programmabbruch führt. Der Analysator hat keine Möglichkeit gefunden, sich durch implizite Annahmen „weiterzuretten“.

`system error` Der *Systemfehler* sollte eigentlich im normalen Betrieb überhaupt nicht auftreten. Systemfehlermeldungen wurden so im Programm angebracht,

daß sie bei erwartungsgemäßem Ablauf des Programmes niemals aufgerufen werden dürften. Kommt es dennoch einmal zu einer Systemfehlermeldung, so ist höchstwahrscheinlich ein Programmfehler die Ursache.

Beispiel 3.11 zeigt den Aufruf von `model.mdl` mit einigen typischen Fehlermeldungen und Warnungen.

Abschließend noch eine Bemerkung zur Unterscheidung von Fehlermeldungen und Warnungen. Während Fehlermeldungen darauf hindeuten, daß wirklich etwas nicht stimmt, können Warnmeldungen, vor allem am Beginn des Entwurfes einer MDL–Beschreibung, ruhig in Kauf genommen werden. Unvollständige oder fehlende Angaben werden durch ein komplexes System von impliziten Annahmen ergänzt, sodaß eine analysierbare MDL–Beschreibung entsteht. Immerhin steckt ein *erheblicher* Teil des Aufwandes zur Realisierung des Analysators in der Fehlerstabilität und dem System der impliziten Annahmen, die gemeinsam den Analysator erst als Entwurfswerkzeug sinnvoll erscheinen lassen. Dennoch sollte man auch von Zeit zu Zeit die Warnmeldungen überprüfen: Da der Analysator ja nicht *weiß*, was sein Benutzer *denkt*, könnte es sonst passieren, daß eine implizite Annahme die Analyse in einer Art und Weise verändert, die der Benutzer nicht erkennt. Das wiederum könnte zur Folge haben, daß der Benutzer einen Fehler woanders vermutet und dort vergeblich sucht. Im allgemeinen sind die impliziten Annahmen aber so ausgelegt, daß sie die Eingabe sinnvoll und im Sinne des Benutzers ergänzen.

3.4 Aufbau des Analysators

Der gesamte Quellcode des Analysators umfaßt etwa 200kB reinen C–Code, dazu kommen noch etwa 40kB C–Kommentare als Quellcodedokumentation. Dieser Umfang gestattet es nicht, die Struktur des Analysators bis in alle Einzelheiten zu erläutern, daher wird in diesem Abschnitt nur eine grobe Übersicht über die Dateien des Analysators und ihre Aufgaben gegeben. Die Detailinformation ist ja im Quellcode in Form von umfangreichen Kommentaren enthalten. Besonders die Definitionsdateien `symtab.h` und `contab.h` enthalten auch Überblicksinformation über Aufbau und Funktionsweise ganzer Module und die darin verwendeten Datenstrukturen.

Tabelle 3.2 enthält eine Aufstellung aller Quelldateien des Analysators, grob gegliedert nach den vier hauptsächlichen Aufgabenbereichen Syntaxprüfung, Symboltabelle, Konfliktanalyse und Benutzerschnittstelle. Dazu kommt noch die Datei `makefile`, die gemeinsam mit dem UNIX–Werkzeug `make` der Konstruktion des Analysators nach Veränderung einiger Quelldateien dient. Außerdem werden noch die UNIX–Werkzeuge `lex` und `yacc` zur Generierung der Syntaxprüfung benötigt. Nun werden die Aufgaben der einzelnen Dateien kurz zusammengefaßt:

`mdl.l` Diese Datei dient als Eingabe für den Übersetzer `lex`, der die darin enthaltene Beschreibung der lexikalischen Analyse in ein tabellengetriebenes Programm `lex.yy.c` umsetzt, das in der Lage ist, Folgen von Eingabezeichen zu erkennen, die den Mustern genügen. Nähere Details sind in [18] nachzulesen.

```
localhost> mdl -np model

##### MDL Analyzer

##### preprocessing with /lib/cpp model.mdl.

##### starting analysis.
[warning] file model.mdl, line 62 at ";":
          bitrange <-1,6> in fieldlist: anonymous offset assumed!
[warning] file model.mdl, line 63 at ";":
          bitrange in field v: anonymous offset, length=1 assumed!
[warning] file model.mdl, line 105 at ";":
          register portreg in group $$0003: redefinition ignored!
[warning] file model.mdl, line 107 at ">":
          bitrange <-1,4> in fieldlist: anonymous offset assumed!
[error 1] file model.mdl, line 143 at ">":
          incorrect subrange <0,4> of alias br7, default <18,8> assumed !
[error 2] file model.mdl, line 177 at ";":
          alias pt00 has no component!
[error 3] file model.mdl, line 189 at ";":
          component 5 in map word not defined, <0,32> assumed!
[error 4] file model.mdl, line 190 at ";":
          component x in map bdigits not defined, <0,8> assumed!

##### analysis finished. [4 errors][4 warnings]

##### analysis of model.mdl.
##### no control file.
##### descriptor table model.des generated.
##### conflict table model.con generated.
##### connections ignored.
localhost>
```

Beispiel 3.11: Hier nun der Aufruf des Analysators mit der MDL–Beschreibung `model.mdl` und die aufgetretenen Warnungen und Fehlermeldungen: Jede Meldung beginnt mit dem Dateinamen, der Zeilennummer und dem Zeichen, wo der Fehler aufgetreten ist, sofern eine solche Zuordnung eindeutig möglich ist. Danach folgt die Meldung selbst. Die ersten beiden Warnungen sind auf implizite Annahmen bei der Felddefinition zurückzuführen. Die dritte Warnung resultiert aus der Definition der Ressourcenvariablen op2, die das Register `portreg` einmal direkt und einmal indirekt über die Ressourcenvariable `op1` insgesamt zweimal enthält. Die letzte Warnung ist auf eine implizite Annahme bei der `FIELD`–Angabe innerhalb der `MICROP`–Definition zurückzuführen. Die vier Fehlermeldungen entsprechen allesamt genau den Fehlern, die im Kapitel Sprachbeschreibung bei der `ALIAS`–Definition erklärt wurden.

Aufgabenbereich	Dateinamen
Syntaxprüfung	`mdl.l, mdl.y, ytab.1, ytab.2, yyerror.c, yywhere.c, creatok, better.l`
Symboltabelle	`symtab.h, symtab.c`
Konflikttabelle	`contab.h, contab.c`
Benutzerschnittstelle	`main.c, message.c, default.h`

Tabelle 3.2: Übersicht über die Quelldateien des Analysators.

`mdl.y` Darin ist die Grammatik der Sprache in einer Form dargestellt, die der Parsergenerator `yacc` als Eingabe akzeptiert. Neben den Syntaxregeln sind darin auch die Aufrufe der Funktionen enthalten, welche die semantischen Aktionen des Analysators bewirken. [18] enthält eine ausgezeichnete Beschreibung der Wirkungsweise von `yacc`. Die folgenden sechs Dateien gehören logisch zur Syntaxanalyse und sind ebenfalls in [18] genauestens beschrieben.

`ytab.1` Ein Teil der verbesserten Fehlerbehandlung.

`ytab.2` Ein Teil der verbesserten Ablaufverfolgung.

`yyerror.c` Funktionen zu den Fehlermeldungen des Parsers.

`yywhere.c` Funktionen zur Angabe, wo ein Fehler passiert ist.

`creatok` Shellscript zur verbesserten Ablaufverfolgung.

`better.l` Lexscript zur verbesserten Fehlerbehandlung.

`symyab.h` Die Definitionsdatei zu `symtab.c` enthält neben den Datenstrukturdefinitionen und mannigfachen `#define`–Anweisungen vor allem auch die Deklarationen aller Funktionen in `symtab.c` sowie eine Überblicksbeschreibung der Symboltabelle.

`symtab.c` Diese Datei enthält die Funktionen zum Aufbau der internen Symboltabelle des Analysators sowie zur Ausgabe der Deskriptortabelle.

`contab.h` Die Definitionsdatei zu `contab.c` enthält neben den Deklarationen analog zu `symtab.h` auch eine Beschreibung der internen Konflikttabelle.

`contab.c` Darin sind die Funktionen zum Aufbau der internen Konflikttabelle, insbesonders also die Konfliktanalyse selbst, sowie die Funktionen zur Top–Down–Analyse und zur Ausgabe sämtlicher Konflikttabellen enthalten.

`main.c` Diese Datei enthält das Hauptprogramm mit der Behandlung der Optionen sowie die Funktionen zur interaktiven Eingabe. Von hier werden dann sowohl der Präprozessor als auch die Syntaxanalyse aufgerufen und schließlich die Diagnosemeldungen ausgegeben.

`message.c` Darin sind die Funktionen zur Ausgabe der Fehlermeldungen, der Warnmeldungen, der fatalen Fehlermeldungen und der Systemfehlermeldungen enthalten.

`default.h` Diese Datei enthält mittels `#define`–Anweisungen die Möglichkeit, bestimmte Standardnamen den persönlichen Wünschen anzupassen. Sie ist ausführlich dokumentiert, nach einer Änderung muß aber mittels `make mdl` der Analysator neu übersetzt werden.

3.5 Überprüfung der Entwurfsziele

In diesem Abschnitt wird nun zusammenfassend überprüft, wie der Analysator die im Abschnitt 1.4 ab Seite 15 an ihn gestellten Anforderungen erfüllt. Die Anforderungen waren:

- Einlesen einer MDL0–Beschreibung und Prüfung der syntaktischen Korrektheit. Beim Auftreten von Syntaxfehlern soll die weitere Analyse mit entsprechenden Fehlermeldungen abgebrochen werden. Wenn hingegen die MDL–Beschreibung semantisch fehlerhaft oder unvollständig ist, soll mit Hilfe impliziter Annahmen die Analyse nach Möglichkeit fortgesetzt werden. Dies entspricht der Forderung nach einem großzügigen Entwurfswerkzeug.

- Ausgabe einer Tabelle aller verwendeten Bezeichner, die dem semantischen Vorlauf zwecks Prüfung der hardwarespezifischen Namen zur Verfügung gestellt wird.

- Ausgabe einer Tabelle aller überhaupt möglichen Konflikte sowie Erstellung von konkreten Konfliktanalysen für bestimmte Anfragen, zum Beispiel eine konkrete Kombination von Mikrooperationen.

Der erste Punkt wird einerseits durch die Syntaxanalyse erfüllt und andererseits durch die bereits beschriebene Fehlerstabilität und Großzügigkeit bei unvollständigen Eingaben: Durch mannigfache implizite Annahmen wird nach Möglichkeit ein korrekter Zustand hergestellt. Die zweite Anforderung wird von der Deskriptortabelle erfüllt, das endgültige Aussehen der Deskriptortabelle ist dann lediglich ein Schnittstellenproblem zwischen Analysator und Compilerfrontend. Die letzte Anforderung schließlich wird durch die Konflikttabellen erfüllt. Die Bottom–Up–Konflikttabelle stellt dabei alle überhaupt möglichen Konflikte dar, während die dateigesteuert oder interaktiv erstellten Top–Down–Konflikttabellen konkrete Kombinationen von Mikrooperationen untersuchen.

Alles in allem stellt somit der Analysator einen Prototyp dar, der zwar nicht gerade ein Muster an Benutzerfreundlichkeit ist, aber dafür beweist, daß die Sprache MDL die benötigten Informationen enthält und die geforderten Konflikttabellen *erstellbar* sind. Im nun folgenden letzten Kapitel werden noch einige Varianten und Möglichkeiten diskutiert, bevor abschließend die weitere Entwicklung der MDL sowie alternative Einsatzmöglichkeiten vorgestellt werden.

Kapitel 4

Möglichkeiten und Zukunft

Ausgehend von den ursprünglichen Anforderungen wird in diesem Kapitel zunächst zusammengefaßt, welche Teile im Zuge der Diplomarbeit realisiert wurden und welche Aufgabenstellungen noch weiter erfüllt werden müssen, um die volle Leistungsfähigkeit der konzipierten Methode zu erreichen. Zusätzlich werden einige sinnvolle Verbesserungen und Erweiterungen dargelegt. Dann werden die Einsatzmöglichkeiten der MDL vorgestellt, und zwar sowohl im Bereich der Mikroprogrammierung als auch im Bereich der RISC[1]–Prozessoren. Zuletzt erfolgt noch ein allgemeiner Ausblick vor dem Schlußwort.

4.1 Zusammenfassung und Weiterentwicklung

Hier noch einmal die elementarsten Anforderungen an die Methode der MDL, wie sie sich im Zuge der Spezifikation ergeben haben:

- Normung der Schnittstelle zwischen Hardwareentwickler und Compilerentwickler und Verminderung des Kommunikationsaufwandes.
- Automatische Auflistung der hardwarebedingten Konflikte zum Zwecke eines zusätzlichen Hardwareentwicklungszyklus mit der Möglichkeit, auch auf einer abstrakten Maschine entwickeln zu können.
- Automatische Generierung der hardwarebedingten semantischen Regeln sowie effiziente Implementierung der sonstigen semantischen Prüfungen sowie der Codeerzeugung.
- Auswahl der prozessorspezifischen Sprachuntermenge aus dem fest vorgegebenen Sprachrahmen.

In den Abschnitten 2.5 und 3.5 wurde dargestellt, wie Sprache und Analysator die an sie gestellten Anforderungen erfüllen, sodaß hier nur noch eine Zusammenfassung erfolgt: Die

[1] *Reduced Instruction Set Computer*

ersten beiden Punkte werden von der bisherigen Ausprägung der Sprache, auch MDL0 genannt, und dem zugehörigen Analysator erfüllt. Die fehlenden beiden Punkte müssen noch durch Erweiterung der MDL0 zur MDL1 und durch Entwicklung des Generators ausgearbeitet werden, der folgende Abschnitt formuliert dazu die Aufgabenstellungen für eine weitere Entwicklung. Danach werden noch einige sonstige Verbesserungen und Erweiterungen vorgeschlagen, wie sie aus der derzeitigen Sicht sinnvoll erscheinen.

Die Generierung des Firmwarecompilers

Folgende Anforderung unterscheidet die MDL0 von der MDL1: Die Zuordnung der Syntaxelemente der Mikroprogrammsprache sowohl zu den Mikrooperationen als auch zur Codegenerierung. Im Abschnitt 2.4 wurde der neue Aufbau des Firmwarecompilers genau dargestellt. Daraus ergaben sich folgende Verfeinerungen der obigen Anforderung an die MDL1:

- Angabe der erlaubten beziehungsweise codierbaren Tokenfolgen. Diese legen dann gemeinsam die Sprachuntermenge fest.
- Allgemeine Formulierung von Algorithmen in einer Programmiersprache, wobei auf die Attribute der Token mittels einer Metasymbolik zugegriffen werden kann. Damit sind die syntaxorientierten Konflikte behandelbar.
- Für jede Mikrooperation müssen die entsprechenden Tokenattribute an die jeweiligen Ressourcenvariablen zugewiesen werden. Anschließend muß eine Konfliktprüfung mit allen Mikrooperationen des Mikrobefehls durchgeführt werden. Damit werden die Ressourcenkonflikte behandelt und insbesonders die Konflikte zweiter Art aufgelöst. Dabei müssen außerdem die explizit angegebenen `CONDITION`–Definitionen berücksichtigt werden.
- Aus den Attributen kann der Bitcode der einzelnen Mikrooperationen algorithmisch ermittelt werden. Er ist dann den entsprechenden Bitbereichen im Mikrobefehlsfeld zuzuordnen, woraus sich insgesamt der Bitcode des Mikrobefehls ergibt.
- Zugriff auf die `FIELD`-, `REGISTER`-, `MAP`- und `ALIAS`–Definitionen mittels einer Metasymbolik innerhalb der Programmcodesegmente. Darüberhinaus sollte die Verwendung des `multirange` sowie auch der `Binary_constant` möglich sein.

Diese erweiterte MDL1–Beschreibung wird dann von einem Generator eingelesen, der folgende Anforderungen erfüllen muß:

- Einlesen einer MDL1–Beschreibung und syntaktische Prüfung.
- Erzeugung von Konflikttabellen in einer Form, die zur Steuerung der semantischen Prüfungen geeignet ist. Gegebenenfalls müssen nicht nur Tabellen, sondern auch Unterprogramme zum Firmwarecompiler generiert werden.

- Auswahl der Sprachuntermenge aus dem Sprachrahmen und Angaben über Einschränkungen, die der Firmwareprogrammierer selbst berücksichtigen muß, weil sie sich der Überprüfung durch den Compiler entziehen. Dabei kann es sich zum Beispiel um Konflikte handeln, die sich erst zur Laufzeit ergeben, wie etwa bei der indirekten Adressierung.
- Generierung von Tabellen und Funktionen zur Steuerung der Umsetzung der Syntaxelemente in den Bitcode.
- Bereitstellen einer Schnittstelle, über die der Compilerentwickler steuernd in den Analysevorgang eingreifen kann, etwa um Sonderfälle abzuhandeln.

Verbesserungen und Erweiterungen

Die erste Erweiterung wurde bereits im Zuge der Spezifikation erwähnt: Die Einführung eines *Filters* oder Adapters, der aus der Hardwarebeschreibungssprache, die als Basis des Hardwaresimulators dient, die MDL0–Beschreibung weitgehend generiert. Abbildung 1.6 auf Seite 14 zeigt den zugehörigen Entwicklungszyklus.

Die zweite Verbesserung betrifft die *Benutzeroberfläche* des Analysators: Hier wäre eine graphische Oberfläche wünschenswert, welche nach Einlesen der MDL–Beschreibung aus einer Datei die Variation derselben gestattet. Dabei sollte auch die interaktive Konfliktanalyse sowie die Durchsicht der generierten Tabellen möglich sein. Es wäre dann auch die graphische Aufbereitung der Konflikttabellen in Form von Diagrammen wünschenswert, um den Entwickler rascher die kritischen Konflikte und ihre Ursachen erkennen zu lassen.

Die dritte Erweiterung schließlich stellt einen etwas allgemeineren *objektorientierten* Ansatz vor. Dieser kommt vor allem der Forderung nach hoher Flexibilität bei gleichzeitiger struktureller Einfachheit nach: Jedes Sprachelement wird intern durch ein Objekt repräsentiert, so zum Beispiel alle Hardwareressourcen oder die Mikrooperation. Jede Verwendung eines Sprachelementes in der MDL–Beschreibung bewirkt dann die Aktivierung eines entsprechenden Objektes mit den konkreten Eigenschaften durch den Parser. Der Analysator besteht in diesem Fall aus einem Parser zum Aufruf der Objekte und einer entsprechenden *Objektbibliothek* sowie aus einigen Basisobjekten zur Durchführung der Analyse oder zur Ausgabe der Tabellen. Dabei könnte auch das Hierarchiesystem, also die Möglichkeit der Vererbung, sinnvoll eingesetzt werden. Ein Beispiel wäre etwa die Definition einer ALU mit etwas spezielleren Eigenschaften als Unterobjekt zum allgemeinen Objekt „ALU“.

Jedes Objekt der Objektbibliothek besitzt bestimmte Eigenschaften, die in Form interner Variablen und lokaler Funktionen implementiert werden. Mit der Außenwelt tritt das Objekt nur über eine Schnittstelle in Verbindung, die das Senden und Empfangen von Signalen gestattet. Die lokalen Funktionen dienen dann der Abarbeitung der empfangenen Signale abhängig von den inneren Zuständen, dem Senden von Signalen sowie der Verwaltung der inneren Zustände.

Im Zuge der Konfliktanalyse könnte die Verwendung einer Hardwareressource durch eine Mikrooperation dann mittels eines Signals von dem Objekt der Mikrooperation an das Objekt der Ressource realisiert werden. In diesem Signal wären dann auch Zeitangaben und Information über die Art der Verwendung enthalten. Zuletzt wäre es allein Sache des Objektes der Hardwareressource, einen Konflikt zu erkennen und entsprechend zu melden, zum Beispiel als Signal an das Objekt der Konfliktttabelle. Dies entspricht übrigens auch am ehesten der physikalischen Realität, wo a priori nur die Hardwareressource selbst erkennt, ob sie mehrfach benutzt wird. Die Benutzer selbst erhalten lediglich eine falsche Information, unter Umständen sogar ohne dies zu erkennen.

Da sich objektorientierte Ansätze durch eine ausgesprochen geringe Redundanz auszeichnen, sind sie besonders gut zur Implementierung von *Entwurfswerkzeugen* geeignet. Vor allem die Eigenschaft der *Kapselung* sorgt hier für eine unkomplizierte Wartbarkeit und leichte Erweiterbarkeit des Analysators: Sollte ein neues Sprachelement hinzugefügt werden, sind nur zwei Anpassungen notwendig: In der Grammatikbeschreibung muß die Syntax des neuen Sprachelementes ergänzt werden und in der Objektbibliothek muß das zugehörige Objekt definiert werden. Danach wird das neue Objekt ohne Änderungen im Rest des Programmes überall berücksichtigt!

4.2 Einsatzmöglichkeiten bei RISC–Prozessoren

Hier werden nun nach einer kurzen Betrachtung der Entwicklung der Mikroprogrammierung und der RISC–Prozessoren die Einsatzmöglichkeiten der MDL in diesen Bereichen beschrieben. Dabei wird auch dargelegt, warum Mikroprogrammierung und RISC einander nicht unbedingt ausschließen müssen.

Mikroprogrammierung und RISC–Prozessoren

Zunächst wird das Konzept der RISC–Prozessoren mit ihren wichtigsten Eigenschaften soweit zusammengefaßt, daß daraus die Voraussetzungen für eine Anwendung der MDL in diesem Bereich abgeleitet werden können.

RISC ist die Abkürzung für *Reduced Instruction Set Computer* und bezeichnet einen Computerarchitekturstil, der sich vor allem durch seine Einfachheit und Effizienz auszeichnet. Folgendes Konzept liegt dieser Architektur zugrunde: Der Befehlssatz enthält nur die notwendigen und ausreichenden Operationen. Der Grund dafür ist, daß für fast alle Berechnungen einige wenige, einfache Operationen ausreichen. Diese müssen dann aber auch sehr schnell sein. Die Hardware ist relativ einfach. Das ist aber kein Selbstzweck, sondern bringt neben einem deutlichen Geschwindigkeitszuwachs[2] auch den Vorteil einfacherer Implementierungen.

Die RISC–Architektur kann auch als verzögerte Reaktion auf den Übergang vom Assembler zu *Hochsprachen* aufgefaßt werden: Während einerseits beim Assembler–Programmieren ausführliche, komplexe und im allgemeinen mikrocodierte Befehle von

[2]Faktor zwei bis fünf gegenüber vergleichbarer traditioneller Hardware.

Vorteil sind, benötigt die Masse der aus einer Hochsprache übersetzten Programme nur etwa 20% bis 30% der zur Verfügung stehenden Befehle. Besonders typisch für RISC–Architekturen ist auch die enge Beziehung der Maschinen–Architektur zum Compiler–Design, im wesentlichen zum Compiler-Backend. Dabei zeigt sich wieder der Vorteil der einfacheren Hardware. Ziel ist eine maximale effektive Geschwindigkeit. Seltene Funktionen werden in die Software verlagert, es gibt keine komplizierten, mikrocodierten Befehle, wie z.B. Texteditierfunktionen oder aufwendige Unterprogramme. Lediglich die Fließkommaarithmetik läßt sich in der Hardware effizienter implementieren als in der Software.

Aufgrund ihrer Einfachheit kann sich die RISC–Architektur besser an neue Technologien anpassen als das bei CISC[3]–Architekturen möglich ist. Daher verbessert sich die Geschwindigkeit bei CISC–Architekturen deutlich langsamer als die Technologie. Als logische Folgerung aus diesem Konzept ergeben sich einige Eigenschaften, die für RISC–Architekturen typisch sind. Diese werden mit ihren englischen Originalbezeichnungen angeführt, da diese Ausdrücke gebräuchlicher sind als allfällige deutsche Übersetzungen:

Single cycle execution: Die meisten Instruktionen können in einem Taktzyklus ausgeführt werden. Ausnahmen sind z.B. Lese- und Schreib–Zugriffe auf den Hauptspeicher.

Hardwired control: Es wird wenig bis kein Mikrocode verwendet.

Register to register design: Es gibt nur zwei Speicherzugriffe: `LOAD` und `STORE`. Alle anderen Instruktionen arbeiten mit den Registern.

Fixed format instructions: Die Befehle sind alle gleich breit, typisch 32 bis 64 Bit. Ebenso gibt es relativ wenige Adressierungsarten.

Pipelining: Mehrere Befehle befinden sich zeitlich gestaffelt zur selben Zeit in Bearbeitung.

High performance memory: Es gibt typisch mindestens 32 allgemeine Register und große Cache–Speicher.

Vertical migration: Nur jene Funktionen, die nachweislich die Geschwindigkeit verbessern, werden in der Hardware implementiert, die übrigen werden in die Software verlagert, wo sie dann als Sequenzen einfacher Instruktionen enthalten sind.

Parallelism: Es wird mehr interne Gleichzeitigkeit für die Software sichtbar, zum Beispiel bei der verzögerten Verzweigung.

Compiletime Functionality: Es wird Funktionalität von der Laufzeit weg zur Übersetzungszeit hin verlagert: Optimierende Compiler bewirken zum Beispiel die Neuanordnung gepufferter Befehlssequenzen oder eine geeignete Anordnung von Register/Register–Operationen. Somit liegt mehr Verantwortung beim Compiler–Backend und es wird mehr Wissen um die Hardware in diesem Bereich notwendig.

[3] *Complex Instruction Set Computer*

Der Compiler wird also in einem Maße *hardwareabhängig*, wie das früher nur bei Mikroprogrammcompilern der Fall war. Diese Tatsache ist Ansatzpunkt für weitere Überlegungen um den Einsatz der MDL in diesem Bereich. Die Details werden im nächsten Abschnitt diskutiert. Zuvor soll aber noch die Frage erörtert werden, warum sich die RISC–Architektur weitgehend durchgesetzt hat.

Ursprünglich wurden nur ganz einfache Funktionen in der Hardware implementiert, kompliziertere Funktionen wurden mittels Mikrocode als Firmware in ROM[4]–Speichern abgelegt. Da das beim Auftreten von Fehlern nicht ganz unproblematisch war, wurde Anfang der 70er Jahre die Firmware in RAM[5]–Halbleiterspeichern abgelegt, die viel rascher waren als die damals für den Hauptspeicher üblichen Ferritkerne. Um den Inhalt des relativ langsamen Hauptspeichers klein zu halten, war die CISC Architektur damals die logische Konsequenz, um die Programme kurz und effizient zu halten. Folgende Vorteile sprachen damals für die Mikroprogrammierung:

- Einfache Anpassung bei Änderungen auch in der Endphase der Hardwareentwicklung.
- Die Emulation anderer Befehlssätze im Mikrocode vereinfacht die Software–Kompatibilität.
- Um zu einer Architektur komplexere Instruktionen hinzuzufügen, benötigt man lediglich einen größeren Mikroprogrammspeicher.
- Diese Flexibilität ermöglicht es, daß die Hardwareentwicklung beginnt, noch bevor Befehlssatz und Mikrocode fertig sind.
- Nicht zuletzt steht auch für die Mikroprogrammierung mittlerweile eine größere Anzahl an Werkzeugen zur Verfügung.

Folgende Ursachen führten allerdings später zur RISC–Architektur und weg von der Mikroprogrammierung:

- Mikrocode ist nicht schneller als eine Sequenz von Hardware-Instruktionen. Die Verlagerung von Software in die Firmware bringt keinen Gewinn, sondern nur den Nachteil der schwierigeren Veränderung.
- Die billigen und schnellen Halbleiterspeicher machen ein Sparen von Speicherplatz sinnlos. Große Cache–Speicher machen nicht mikrocodierte Programme sogar schneller als solche mit Mikrocode.
- Optimierende Compiler benutzen nur wenige einfache Instruktionen.

[4] *Read Only Memory*
[5] *Random Access Memory*

	traditionell	RISC
C	4 – 10	1.3 – 1.7
I	100%	120% – 140%
P	1	2 – 5

Tabelle 4.1: Vergleich der Geschwindigkeiten von traditionellen Architekturen und RISC–Architekturen: Dabei bedeutet C die durchschnittliche Anzahl der Taktzyklen pro Befehl, I die Anzahl der Befehle[a], P die Geschwindigkeit nach einer bestimmten Benchmark[b] und s die Taktfrequenz. Die Geschwindigkeit wurde dabei nach der Formel

$$P = \frac{1}{IC\frac{1}{s}}$$

berechnet. Man erkennt aus dem Vergleich, daß bei der traditionellen Architektur die Befehle zwar mächtiger und effizienter sind, sodaß die Programme kürzer sind, dennoch sind die Programme auf der RISC Architektur deutlich schneller.

[a] Hier mit Index 100% bei der traditionellen Architektur.
[b] Hier ebenfalls mit Index 1 bei der traditionellen Architektur.

Diese und weitere Ursachen führten ab Mitte der 70er Jahre zur Entwicklung der ersten RISC-Maschinen IBM801, Berkeley RISC-I und RISC-II sowie Stanford MIPS[6]. Dabei wurde erstmals das Management der Pipeline der Software ergo dem Compiler anvertraut. Ab 1986 begann dann die Computerindustrie RISC–Prozessoren zu veröffentlichen: MIPS R2000 (1986), AMD29000 (1987), Motorola 88000 und SPARC (1988) sowie IBM RS6000 (1990), um nur einige zu nennen. Seit 1990 bietet nahezu jede namhafte Computerfirma RISC–Produkte an, woraus man schließen kann, daß sich die RISC–Prozessoren in der industriellen Praxis bewährt haben. Tabelle 4.1 zeigt einen Vergleich der Geschwindigkeit von traditionellen Architekturen und RISC–Architekturen. Weitere Information über RISC–Architekturen allgemein kann in [20] und [21] nachgelesen werden, die auch als Basis für diesen Abschnitt dienten.

Bedeutet nun der Sieg der RISC–Architektur über die CISC–Architektur das endgültige Aus für die Mikroprogrammierung? Nicht unbedingt! Es wurde bereits festgestellt, daß der eigentliche Nachteil der Mikroprogrammierung bei der Geschwindigkeit liegt. Das liegt daran, daß die Mikroprogrammierung von der *Speichertechnologie* abhängig ist: Die Dauer des Taktzyklus ist durch die Zeit beschränkt, die benötigt wird, um einen Mikrobefehl aus dem Mikroprogrammspeicher zu lesen. In den späten 60er und frühen 70er Jahren waren schnelle Halbleiterspeicher für den Mikroprogrammspeicher verfügbar, während der Hauptspeicher noch mit Ferritkernen aufgebaut war. Da sich die beiden Technologien bezüglich der Zyklusdauer um den Faktor zehn unterscheiden, war die Mikroprogrammierung damals eine rentable Vorgangsweise. In den späten 70er Jahren mit dem Aufkommen der Cache–Speicher war aber dieser Vorteil dahin, weil Hauptspeicher und Mikroprogrammspeicher mit derselben Technologie aufgebaut waren. Daher erfolgte der oben beschriebene Übergang zur RISC–Architektur.

[6] MIPS bedeutet in diesem Zusammenhang eigentlich *Microprocessor without Interlocked Pipeline Stages* an Stelle der sonst üblichen Bedeutung von *Millions of Instructions Per Second.*

Allerdings ist die Mikroprogrammierung nach wie vor eng mit der Speichertechnologie verbunden: Es wäre durchaus denkbar, daß zum Beispiel irgendwann ROM–Speicher wesentlich schneller als RAM–Speicher werden, oder daß die Cache–Speicher aus irgendeinem Grund ineffizient würden. In diesem Fall würde die Mikroprogrammierung wieder rentabel eingesetzt werden können, auch RISC–Prozessoren könnten wieder in stärkerem Maße mikroprogrammiert werden.

Darüberhinaus gibt es eine große Anzahl von *Spezialprozessoren*, die mit den Methoden der Mikroprogrammierung eine bequeme Anpassung an die Bedürfnisse des Benutzers ermöglichen. Als aktuelles Beispiel sei hier der Mikrocontroller SAB 88C166 genannt, der ein Umprogrammieren der Firmware im System und ohne Hardware–Veränderungen ermöglicht (siehe [24]). Auch hier bietet sich der Methode der Generierung des Firmwarecompilers mittels MDL ein weites Betätigungsfeld.

Letztendlich verlagert sich mit der Einführung der RISC–Architektur die Problematik der Firmware in das Backend des Hochsprachen–Compilers: Durch die starke Hardwareabhängigkeit eines RISC–Compilers muß jetzt der Compilerentwickler Probleme bewältigen, die früher im Bereich der Firmware aufgelöst wurden und für die Software unsichtbar blieben. Wie dabei die Methode der MDL in angepaßter Form ebenfalls sinnvoll eingesetzt werden kann, zeigt der folgende Abschnitt.

Anwendung der Methode bei RISC–Prozessoren

Es wird zunächst einmal angenommen, daß die Hardware eines RISC–Prozessors so beschrieben werden kann, daß daraus die hardwareabhängigen Teile des Compilerbackends für den Hochsprachencompiler generiert werden können. Ob diese Annahme gerechtfertigt ist, wird im Anschluß erläutert, zunächst soll hingegen die Frage nach dem *Nutzen* einer solchen Vorgangsweise gestellt werden. Zwei Vorteile ergeben sich unmittelbar aus einer solchen Methode:

1. Fertige, hochentwickelte Compiler wären mit geringstem Aufwand auf jeder beliebigen Maschine verfügbar. Vor allem jede Verbesserung am Compiler und insbesonders jede hardwareunabhängige Optimierungsmethode könnte sofort auf allen Maschinen ausgenutzt werden. Da heutzutage nicht nur Anwenderprogramme, sondern auch Betriebssysteme nahezu ausschließlich in Hochsprachen programmiert werden, reduziert sich das Problem der *Portabilität* auf die Suche nach einer geeigneten Hardwarebeschreibungsmethode.

2. Mit der Möglichkeit, den Compiler auch für eine abstrakte Maschine zu entwickeln, könnte der Compiler bereits in der Entwurfsphase eines neuen Prozessors zur Verfügung stehen. Damit könnte bereits der entstehende Prozessor mit fertigen Programmen getestet werden, der Compiler wird somit zu einem *Entwurfswerkzeug* für den Hardwareentwickler. Das Zitat "A computer architect designes machines to run programs" aus [20] zeigt die Bedeutung dieses Entwurfswerkzeuges als Maßstab für die Prozessorentwicklung: Es kann das Hauptziel des Entwurfsprozesses in jeder Phase der Entwicklung im Auge behalten und überprüft werden, indem stets eine

Auswahl typischer Anwenderprogramme mit dem Compiler für die in Entwicklung befindliche Maschine übersetzt und getestet wird.

Vor allem auf den zweiten Vorteil wird noch etwas näher eingegangen.

Voraussetzung für beide Vorteile ist die gestiegene Bedeutung des *Compilers*: Da fast nur noch in Hochsprachen programmiert wird, ist es vor allem die Ausgabe des Compilers, die von den Prozessoren abgearbeitet wird. Daher stehen Hardwareentwicklung und Compilertechnologie in einem engen Zusammenhang, ja sind sogar gegenseitig voneinander abhängig und die Unterstützung des Compilers wird zu einem wesentlichen Designziel der Hardware. Eine Methode analog zur MDL könnte hier die Brücke zwischen Hardware und Compiler schlagen. Diese Methode wird in weiterer Folge als CIAD – *Compiler Integrated Architecture Design* bezeichnet, die Beschreibungsform, die dieser Methode zugrundeliegt, als HCDL – *High level Compiler Design Language*. Zunächst wird die Methode des CIAD näher beschrieben, danach werden Anforderungen an die HCDL im Vergleich zur MDL analysiert sowie die Machbarkeit beurteilt.

Um den Entwurf der Hardware während der gesamten Entwicklung zu unterstützen, muß das CIAD hierarchisch gegliedert sein und die HCDL die Beschreibung der Hardware auf mehreren *Abstraktionsebenen* ermöglichen:

Architektur: Auf dieser Abstraktionsebene muß die HCDL die Architektur des Befehlssatzes des Prozessors beschreiben. Daraus muß der Compiler für eine *abstrakte Maschine* generiert werden. Dieser ermöglicht dann das Testen des Befehlssatzes, zum Beispiel die Ausnutzung der einzelnen Befehle durch den entstehenden Code, insbesonders die dynamische Häufigkeit des Auftretens einzelner Befehle. Bisher wurden kurze, künstlich konstruierte Programme händisch übersetzt, um diese Messungen durchzuführen. Der Nachteil war, daß kein noch so subtil konstruiertes Beispiel eine Auswahl realer Anwenderprogramme ersetzen kann, ja manchmal waren gerade diese konstruierten Beipiele für eine Irreführung der Entwickler verantwortlich.

Organisation: Hier muß die funktionelle Organisation der Hardware auf *Register–Transfer–Ebene* beschrieben werden. Es wird die Hardware besonders im Hinblick auf Ressourcenauslastung und Ressourcenkonflikte untersucht. Im Vordergrund steht hier die Messung der erreichbaren Geschwindigkeit, allerdings noch auf Basis des Taktzyklus als abstrakte Größe. Dabei muß berücksichtigt werden, daß ein Design mit niedriger Taktzyklenzahl im Zuge der Implementierung eine Verlängerung der Taktzyklendauer zur Folge haben kann, die den erreichten Vorteil unter Umständen mehr als zunichte macht.

Implementierung: Hier erfolgt eine Beschreibung auf *Gatterebene*, das logische Design, das Packaging und die Entflechtung stehen jetzt im Vordergrund. Jetzt kann auch die *reale* Geschwindigkeit der getesteten Programme als eigentliches Designziel überprüft werden. Ist die Entwicklung des Prozessors abgeschlossen, steht auch sofort ein ausgereifter Compiler zur Verfügung.

Dabei sind die einzelnen Hierarchieebenen nicht isoliert zu sehen. Vielmehr sollte eine geschlossene Beschreibung die Hardware auf allen drei Hierarchieebenen modellieren. Dies

ist vor allem dann sinnvoll, wenn sich während der Entwicklung verschiedene Teile des Prozessors in unterschiedlichen Entwurfsstadien befinden. Es soll jedenfalls stets möglich sein, den Einfluß des Compilers zu testen und quantitative Untersuchungen auf Basis der Anwenderprogramme durchzuführen.

Zuletzt noch ein Vergleich zwischen MDL und HCDL, der die Unterschiede und Gemeinsamkeiten untersucht und schließlich die Machbarkeit der CIAD darstellt. Speziell für die RISC–Prozessoren gilt: Noch nie gab es eine Klasse von Mikroprozessoren, die untereinander so *ähnlich* gewesen sind. Dies läßt zugleich erwarten, daß *eine* Beschreibungsform gefunden werden kann, mit der alle RISC–Prozessoren beschrieben werden können. Eine weitere Ähnlichkeit zur Mikroprogrammierung ist die oft *fixe* Aufteilung des Befehlsfeldes mit typisch etwa einem bis fünf verschiedenen Befehlsformaten. Dabei werden oft einzelne Bitfelder direkt als Steuerleitungen in die Hardware übernommen. Ein typisches Beispiel ist die Registernummer, die bei RISC–Prozessoren häufig einen fixen Platz im Befehlsfeld besitzt und von dort über einen Decoder direkt der Registerselektion dient.

Eine weitere Analogie findet man vor allem bei RISC–Prozessoren mit den Eigenschaften *superpipelined*[9] oder *superscalar*[10]: Durch parallel ausgeführte Befehle kann es zu Ressourcenkonflikten kommen. Dies verursacht zwar keine Fehler, aber immerhin bewirkt es, daß Befehle auf ihre Ausführung warten müssen, was in einer Verminderung der Gesamtgeschwindigkeit resultiert. Hier ist es eine der wichtigsten Aufgaben des Compilers, die Befehle so anzuordnen, daß solche Verzögerungen vermieden werden. An die Stelle der Konflikte zwischen Mikrooperationen treten also hier die Konflikte zwischen Maschinenbefehlen.

Diese Gemeinsamkeiten lassen von der Machbarkeit der MDL auf eine Machbarkeit der CIAD schließen. Dennoch wird diese Methode deutlich komplexer sein als die MDL, was auf folgende Unterschiede zwischen Mikroprogrammierung und Hochsprache zurückzuführen ist:

- Wenn ein Konflikt erkannt wird, bedeutet das bei der Mikroprogrammierung, daß die beteiligten Mikrooperationen nicht in einem Mikrobefehl vereinbar sind. Bei der Makroprogrammierung hingegen muß nach geeigneten Algorithmen eine *Umordnung* der Befehle gefunden werden, sodaß alle Konflikte beseitigt sind. Anders als bei der Mikroprogrammierung ist es dem Makroprogrammierer nicht zumutbar, sich in der Anordnung seiner Hochsprachenbefehle nach Hardwareerfordernissen zu richten.

- Während bei der Mikroprogrammierung die Konflikterkennung dominiert und die Syntax/Code–Zuordnung eher einfach ist, stellt bei der Makroprogrammierung die Syntax/Code–Zuordnung den wesentlichsten Punkt dar. Vielfältige Optimierungsstrategien wie etwa *instruction scheduling*, *register allocation*, *loop unrolling* oder

[9]Hohe zeitliche Parallelität durch besonders tiefe Pipeline ermöglicht die Parallelität vieler Befehle auf einer Hardware.

[10]Hohe räumliche Parallelität durch Reduplikation wichtiger Ressourcen ermöglicht einen Durchsatz von mehr als einem Befehl pro Taktzyklus.

software pipelining müssen unter einen Nenner gebracht werden. Diese Aufgabe wäre so schon schwer genug, dennoch sind auch die Optimierungsstrategien teilweise hardwareabhängig: Beim *register renaming* zum Beispiel ist die Anzahl der verfügbaren Register die elementare Größe.

- Die Zwischensprache zwischen Frontend und Backend besitzt bei der Makroprogrammierung *Baumstruktur* anstelle der linearisierten Form bei der Mikroprogrammierung. Daher muß die Syntax/Code–Zuordnung mit den Methoden des *tree pattern matching* die Relation der Zwischensprache zum Maschinencode herstellen. Details sind in [23] und [22] beschrieben.

- Die durch die HCDL zu beschreibenden Organisationselemente sind teilweise *komplexer* als die durch die MDL beschriebenen: Man denke etwa an die Beschreibung einer vierstufigen Pipeline oder einer superskalaren *instruction dispatch and branch unit* wie beim MC88110 (siehe [25]).

Alles in allem ist die Methode der CIAD deutlich komplexer als die Verwendung der MDL im Bereich der Mikroprogrammierung. Andererseits würde der Aufwand für die Entwicklung dieser Methode durch gewaltige Vorteile in den Bereichen der Hardwareentwicklung und der Compilerentwicklung gleichermaßen belohnt werden.

4.3 Ausblick und Schluß

Während sich die RISC–Architektur gegenüber der CISC–Architektur zweifellos durchgesetzt hat, bleibt dennoch die Zukunft der Mikroprogrammierung offen. Vor allem im Bereich der *Spezialprozessoren* und *Mikrocontroller* wird die Mikroprogrammierung im Moment jedenfalls intensiv genutzt. In diesem Bereich bietet die Hardwarebeschreibung mittels MDL einen Ansatz, um den Firmwarecompiler weitgehend generieren zu können. Damit kann die Verflechtung zwischen Hardwareentwicklung und Firmwareentwicklung soweit aufgelöst werden, daß sie sich nicht bremsend auf den gesamten Entwicklungszyklus auswirkt.

Die Entwicklung der MDL0 sowie des zugehörigen Analysators im Rahmen der Diplomarbeit stellt den ersten Schritt auf dem Weg zur Verwirklichung der beschriebenen Methode dar. Aber erst mit der Erweiterung zur MDL1 und der Implementierung des Compilergenerators steht ein *wirklich* effizienter Entwicklungszyklus zur Verfügung. Mit den dabei gewonnenen Erfahrungen müßte es in weiterer Folge möglich sein, auch für RISC–Prozessoren ein ähnliches Werkzeug zu konzipieren. Dieses könnte dann den Entwurf der Hardware selbst *maßgeblich* unterstützen und somit letztendlich mithelfen, noch bessere Prozessoren zu entwerfen. □

Anhang L

Lexikalische Regeln

Allgemeines

MDL besteht, wie jede Sprache, aus einer Menge von Sätzen (siehe auch Anhang S), die ihrerseits Folgen von *Eingabesymbolen* sind. Normalerweise besitzt eine Programmiersprache drei Arten von Eingabesymbolen: *Operatoren und Begrenzer* werden als kurze Folge von Sonderzeichen dargestellt, *reservierte Schlüsselwörter* werden als vordefinierte Folge von Buchstaben dargestellt, deren Bedeutung nicht verändert werden kann, und schließlich vom Benutzer eingeführte Eingabesymbole, als da sind: *Konstanten* und *Identifier oder Bezeichner*. Zusätzlich gibt es noch Zwischenraumzeichen[1], die zwar die Eingabesymbole gegeneinander abgrenzen, sonst aber unwesentlich sind.

In diesem Anhang werden die lexikalischen Regeln und ihre Implementierung vorgestellt, also jener Teil des Analysators, der aus der unstrukturierten Folge der Eingabezeichen jene Eingabesymbole konstruiert, die schließlich das *Alphabet* der Grammatik darstellen (siehe auch Anhang S). In der Datei `mdl.l` ist die komplette lexikalische Definition in Form einer Spezifikationssprache enthalten, die der Übersetzer `lex` als Eingabe akzeptiert und in ein tabellengetriebenes C–Programm umsetzt, das in der Lage ist, Folgen von Eingabezeichen zu erkennen, die den Mustern genügen. Nähere Details sind in [18] nachzulesen.

Hier soll nun diese Musterbeschreibung der Einfachheit halber nicht so vorgenommen werden, wie in `mdl.l`, sondern mit den Mitteln der Syntaxdiagramme aus Anhang S. Zwar ist die Musterbeschreibung effizienter, die Darstellung mittels Diagrammen jedoch übersichtlicher. Gegenüber der in Anhang S gegebenen Erläuterung zu den Diagrammen wird noch eine Symbolik hinzugenommen: `'A'...'Z'` bezeichnet eines der Zeichen von `'A'` bis `'Z'`, die Ränder eingeschlossen, in der Reihenfolge ihres ASCII–Codes.

Zunächst werden nun die reservierten Schlüsselwörter zusammengefaßt vorgestellt, danach folgt die Darstellung des lexikalischen Aufbaues der Konstanten und Bezeichner. Ziel dieser Darstellungen ist es zu definieren, welchen *lexikalischen Vorschriften* eine MDL–Beschreibung genügen muß. Gleichzeitig soll aber auch gezeigt werden, wie diese

[1] z. B. Leerzeichen, Tabulatorzeichen und Zeilentrenner.

ALIAS	CONFLICT	DATAFLOW
DEVICE	ENDOP	FIELD
GROUP	MAP	MAXFIELD
MAXTIME	MICROP	OF
PARTOF	PATH	READ
REGISTER	SET	TIME
USING	VAR	WRITE

Tabelle L.1: Die MDL–Schlüsselwörter.

Alternative	Bedeutung
CON	CONFLICT
FLOW	DATAFLOW
EOP	ENDOP
STATEMENT	MICROP
PART	PARTOF
RD	READ
REG	REGISTER
WR	WRITE

Tabelle L.2: Alternative Schlüsselwörter.

lexikalischen Elemente zum Alphabet der Grammatik umgesetzt werden, aus welchem dann die Eingabesymbole in den Syntaxdiagrammen stammen (siehe Anhang S).

Reservierte Wörter

Alle Schlüsselwörter von MDL sind großgeschrieben, für einige gibt es auch Alternativschreibweisen. Tabelle L.1 zeigt die MDL–Schlüsselwörter, Tabelle L.2 die möglichen Alternativen.

Konstante und Bezeichner

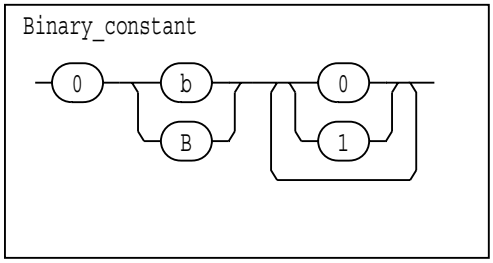
Die folgenden Regeln L1 bis L8 beschreiben den detaillierten lexikalischen Aufbau der beiden Bezeichner `Identifier` und `Component` sowie der vier Arten von konstanten Zahlen, nämlich der `Binary_constant`, der `Octal_constant`, der `Decimal_constant` und der `Hexadecimal_constant` und schließlich die Form der beiden Textkonstanten `Char_constant` und `String_constant`. Die übrigen der fünfzehn Regeln verfeinern die bei diesen Regeln verwendeten Begriffe bis zu elementaren Zeichen. Muster, die diesen Regeln entsprechen, werden zu den angegebenen acht Eingabewörtern im Alphabet der Grammatik umgesetzt. Neben dem optischen Unterschied, daß diese Wörter hier nicht zur Gänze groß geschrieben werden, sondern nur einen großen Anfangsbuchstaben besitzen, gibt es noch einen wichtigen Unterschied: Diese Wörter enthalten als Attribute, sozusagen im Hintergrund, die Information, wie die Konstante bzw. der Bezeichner ausgesehen hat. Diese Hintergrundinformation wird dann weiterverarbeitet, Details hiezu

sind in [18] nachzulesen. Ansonsten besteht aber rein grammatikalisch im Alphabet kein Unterschied zu den MDL–Schlüsselwörtern. Eine kurze Erläuterung noch zur Regel L13: Diese bedeutet, daß ein `printing_char` jedes Zeichen aus dem ASCII–Zeichensatz sein kann, mit Ausnahme der Zeichen `'`, `"` und `CR` (Zeilenumbruch).

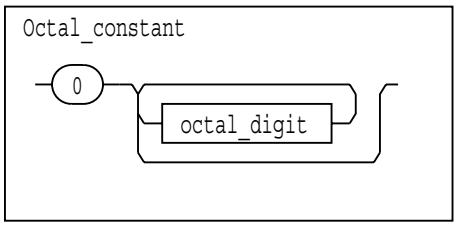
Als Zwischenraumzeichen zwischen den Eingabesymbolen dienen das Leerzeichen, das Tabulatorzeichen und der Zeilenumbruch. Diese werden natürlich nicht in das Alphabet übernommen. Alle Operatoren oder Begrenzer hingegen werden direkt in das Alphabet der Grammatik übernommen.

Nun die lexikalischen Regeln im Detail:

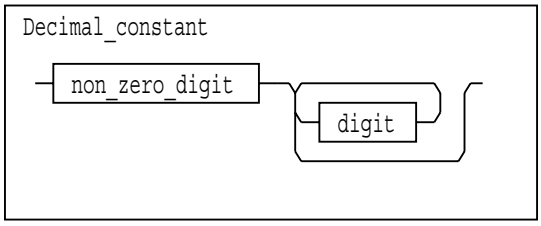
L1



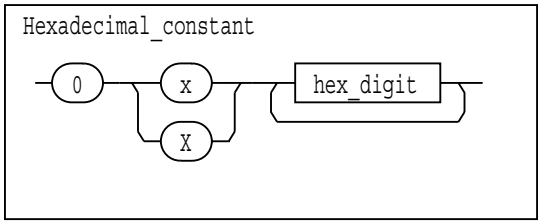
L2



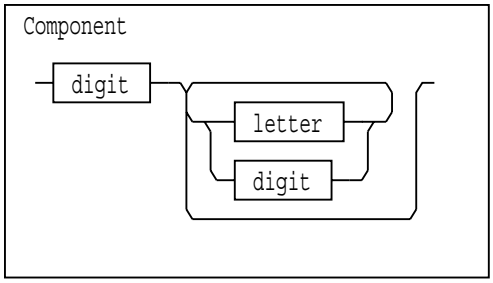
L3



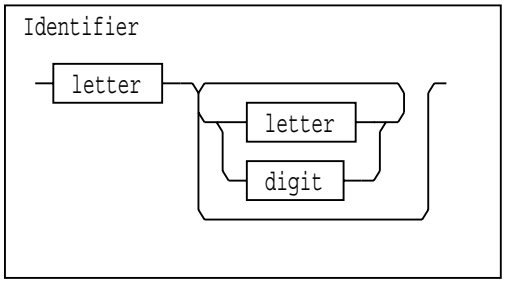
L4


Hexadecimal_constant
0
x
X
hex_digit

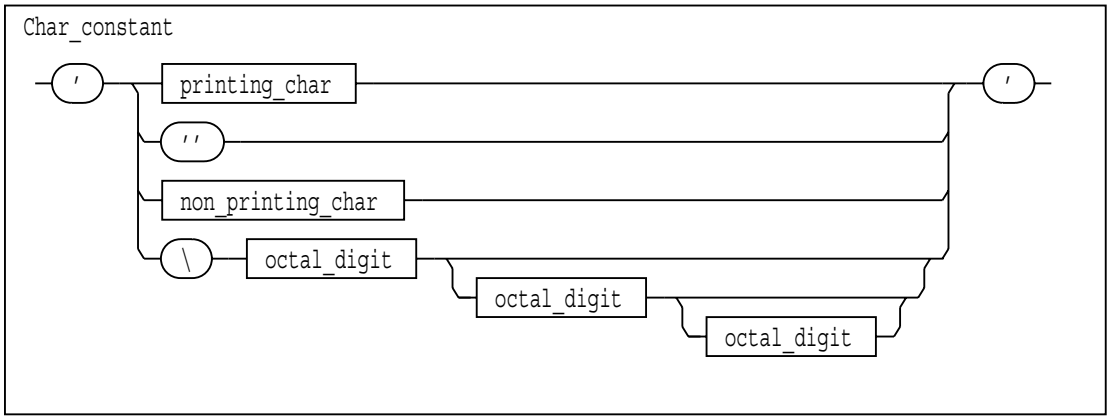
L5


Component
digit
letter
digit

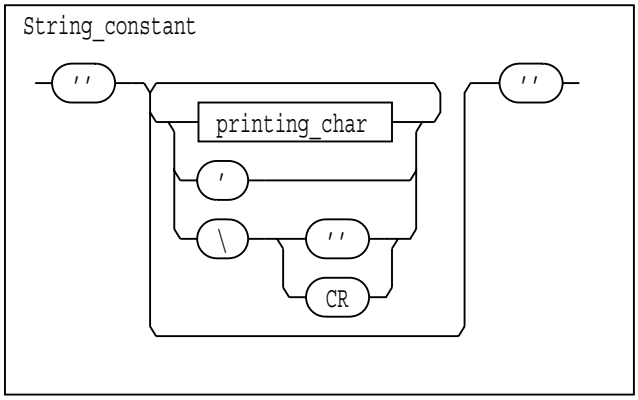
L6


Identifier
letter
letter
digit

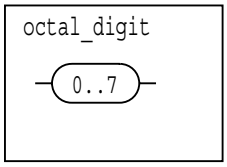
L7


Char_constant
'
printing_char
''
non_printing_char
\
octal_digit
octal_digit
octal_digit
'

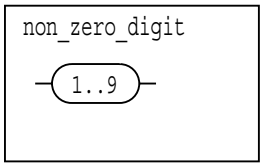
L8



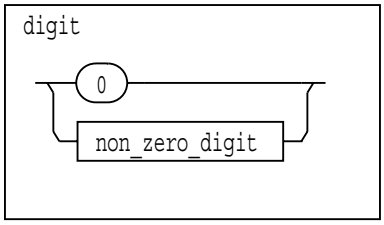
L9



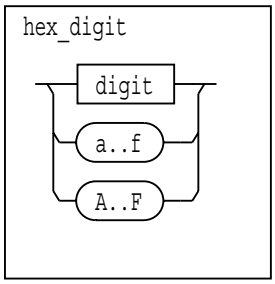
L10



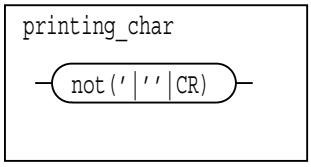
L11



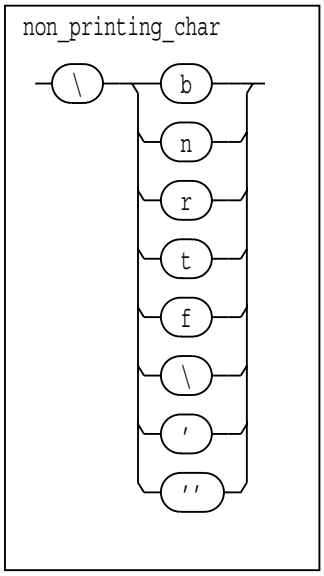
L12



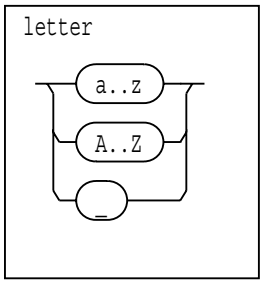
L13



L14



L15



Anhang S

Syntaxregeln

Allgemeines

In diesem Anhang wird die Syntax von MDL und ihre Implementierung beschrieben. In der Datei `mdl.y` ist die komplette Syntaxdefinition zwar auch enthalten, aber nicht so gut lesbar. Das liegt daran, daß die Datei `mdl.y` ja die Eingabe für den Parsergenerator `yacc` [18] darstellt. Daher sind neben den Syntaxbegriffen auch mannigfache C–Anweisungen enthalten, die später Teil des Analysators sind. Außerdem wird in `mdl.y` die Syntax mittels BNF[1] dargestellt, während in diesem Anhang die wesentlich übersichtlichere Darstellung mittels Diagrammen verwendet wird.

Die BNF–Definition einer Sprache, in weiterer Folge auch *Grammatik* genannt, besteht aus einer Folge von *Regeln*. Eine Regel besteht aus einer linken Seite und einer rechten Seite, die durch einen Doppelpunkt getrennt sind. Die linke Seite besteht aus einem einzigen, eindeutigen *Grammatikbegriff* (non-terminal-symbol). Die rechte Seite besteht aus einer Folge von einer oder mehreren *Formulierungen*, die voneinander durch einen `|` getrennt sind. Eine Formulierung in einer Regel kann leer sein; sonst besteht eine Formulierung aus einer Folge von Grammatikbegriffen oder *Eingabesymbolen* (terminal symbols). Grammatikbegriffe und Eingabesymbole sollen hier unter dem gemeinsamen Begriff *Symbole* zusammengefaßt werden.

Eine Grammatik beschreibt eine Sprache dadurch, daß sie erklärt, welche Sätze gebildet werden können. Der Grammatikbegriff auf der linken Seite der ersten Regel ist dabei das *Startsymbol*. Für jeden Grammatikbegriff muß eine Regel existieren und jede Formulierung aus dieser Regel kann an Stelle des Grammatikbegriffes eingefügt werden. Diese Ersetzung geschieht solange, bis eine Folge von Eingabesymbolen, ein *Satz*, aus dem Startsymbol der Grammatik *produziert* wurde. Die Eingabesymbole wiederum bilden das Alphabet der Grammatik. In unserem Fall werden diese Eingabesymbole aus der lexikalischen Analyse gewonnen, siehe Anhang L. Weitere Details sind in [18] nachzulesen.

Die hier verwendeten Syntaxdiagramme sind nichts anderes als eine alternative Darstellung der Regeln: Jede Regel wird von einem Rechteck umrahmt, der Name links

[1] Backus Naur Form

oben im Rechteck stellt die linke Seite der Regel dar. Die Formulierung der Regel wird durch die Zeichnung ersetzt. Gültige Produktionen erhält man, indem man den Linien der Zeichnung von links beginnend folgt. Dabei darf man bei Verzweigungen stets nur solchen Wegen folgen, die eine Wende um weniger als 90 Grad erfordern. Die gültige Produktion ist vollständig, wenn man die Zeichnung rechts verlassen hat. Zusätzlich zu der hier getroffenen Konvention, Eingabezeichen mit Großbuchstaben und Grammatikbegriffe mit Kleinbuchstaben beginnen zu lassen, sind Terminalsymbole von Ovalen umgeben, während Nonterminalsymbole in Rechtecken eingeschlossen sind.

Zunächst werden nun alle Elemente des Alphabets geschlossen dargestellt, bevor im darauffolgenden Abschnitt die Regeln der Grammatik vollständig aufgelistet werden.

Alphabet

Als Alphabet wird die Menge aller möglichen Eingabesymbole bezeichnet. Das Alphabet ist das Ergebnis der lexikalischen Analyse und in Anhang L bereits angedeutet worden. Es besteht aus Schlüsselwörtern, Bezeichnern und Konstanten und allgemeinen Eingabezeichen wie Operatoren und Begrenzern. Die Schlüsselwörter sind bereits in Tabelle L.1 auf Seite L-2 vorgestellt worden, die Bezeichner und Konstanten in den lexikalischen Regeln L1 bis L8. Die Operatoren zeigt Tabelle S.1 mit Vorrang und Assoziativität, Tabelle S.2 zeigt schließlich die übrigen Begrenzer.

!	rechts	höchste Priorität
* / %	links	
+ -	links	
>> <<	links	
< > >= <= >! <!	links	
== !=	links	
&	links	
^	links	
|	links	
&&	links	
||	links	
##	links	
? :	rechts	niedrigste Priorität

Tabelle S.1: Vorrang der Operatoren: Die erste Spalte zeigt die Operatoren, die zweite Spalte gibt die Assoziativitätsrichtung und die dritte Spalte den Vorrang an. Der erste Eintrag besitzt höchste Priorität, der letzte die niedrigste.

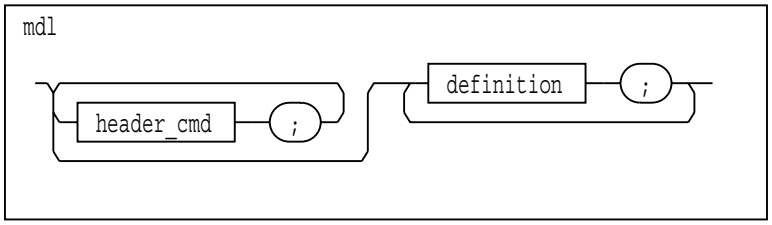
{	}	(	)	[	]
.	,	;	~	=	#

Tabelle S.2: Begrenzer.

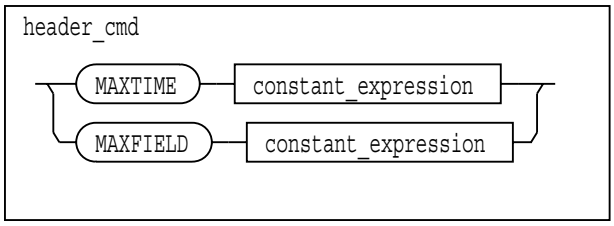
Syntaxregeln

Hier sind nun alle Regeln der Grammatik aufgezählt. Das Startsymbol ist `mdl` in Regel S1.

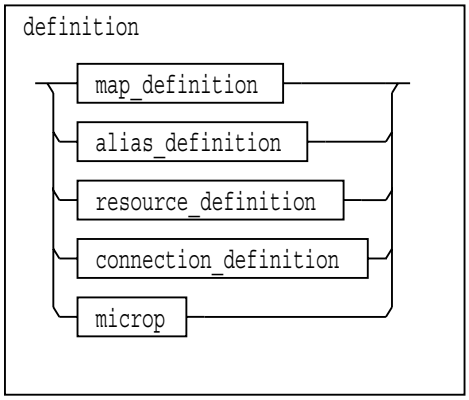
S1



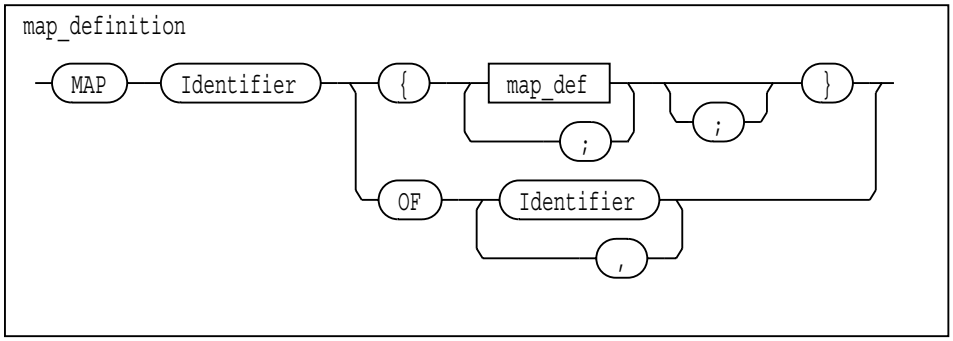
S2



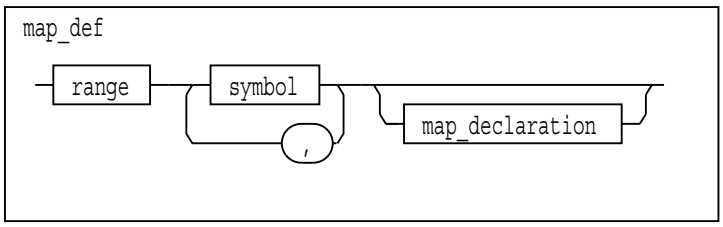
S3



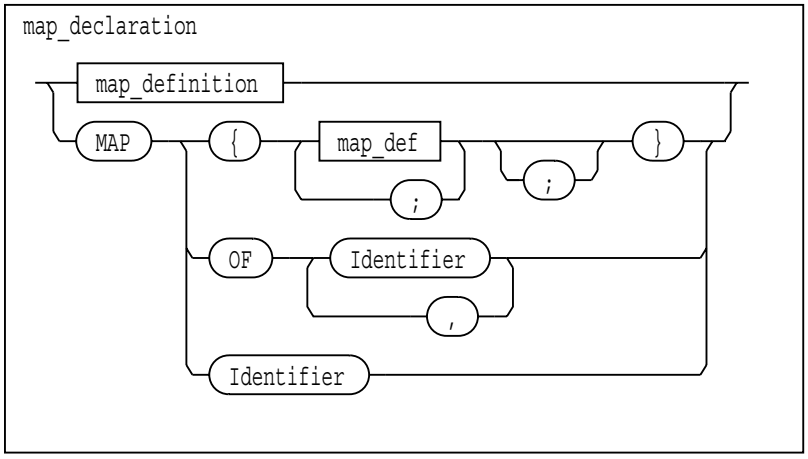
S4



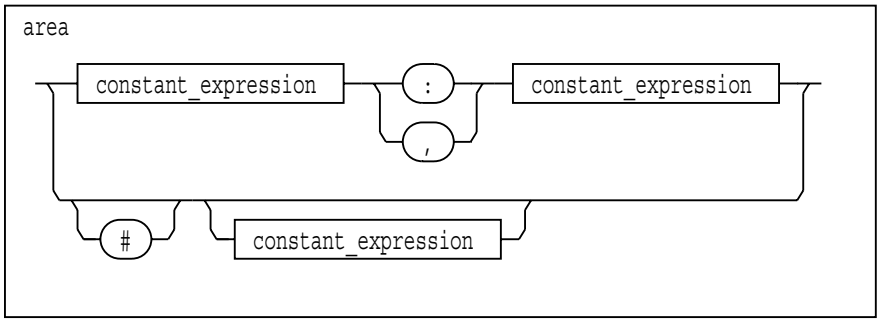
S5


map_def
range
symbol
,
map_declaration

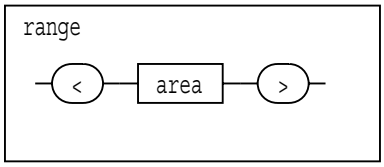
S6


map_declaration
map_definition
MAP
{
map_def
;
;
}
OF
Identifier
,
Identifier

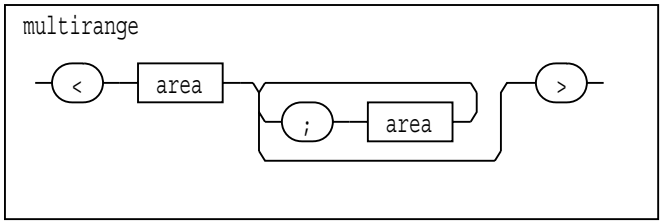
S7


area
constant_expression
:
constant_expression
,
#
constant_expression

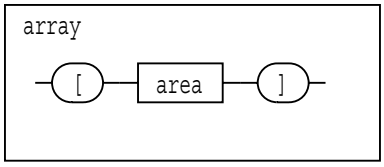
S8


range
<
area
>

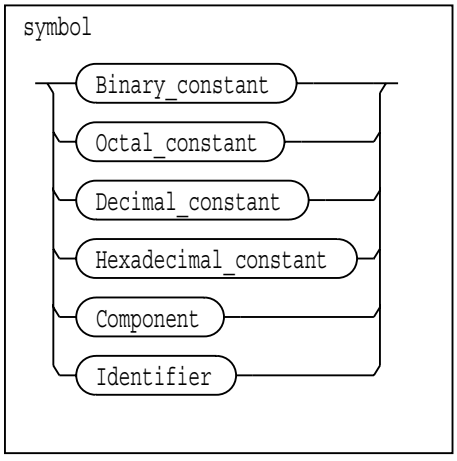
S9



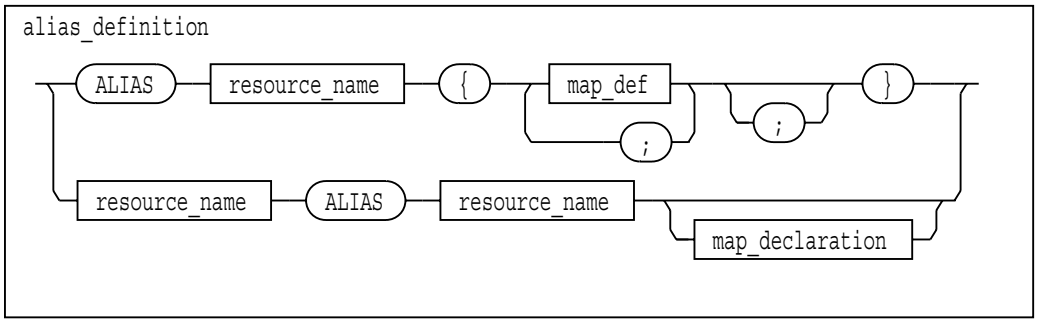
S10



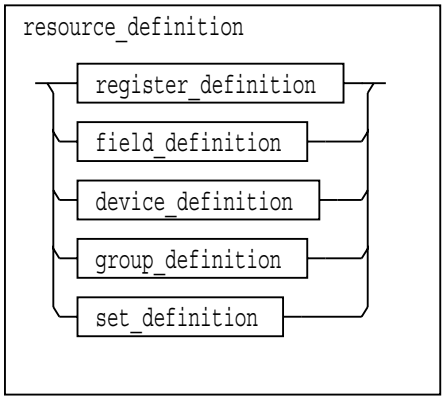
S11



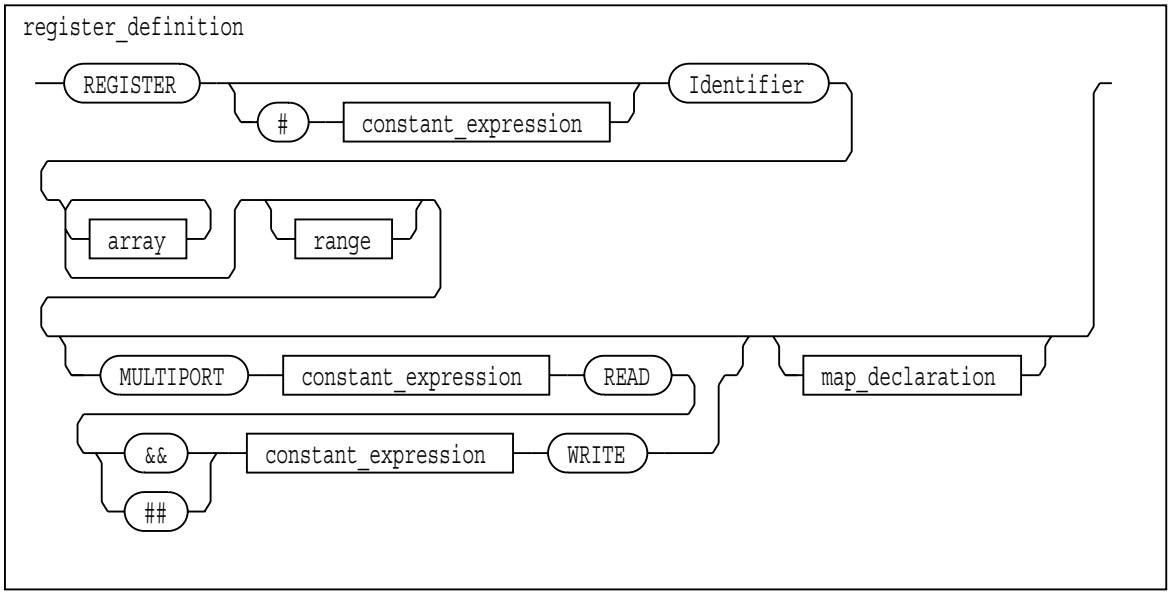
S12



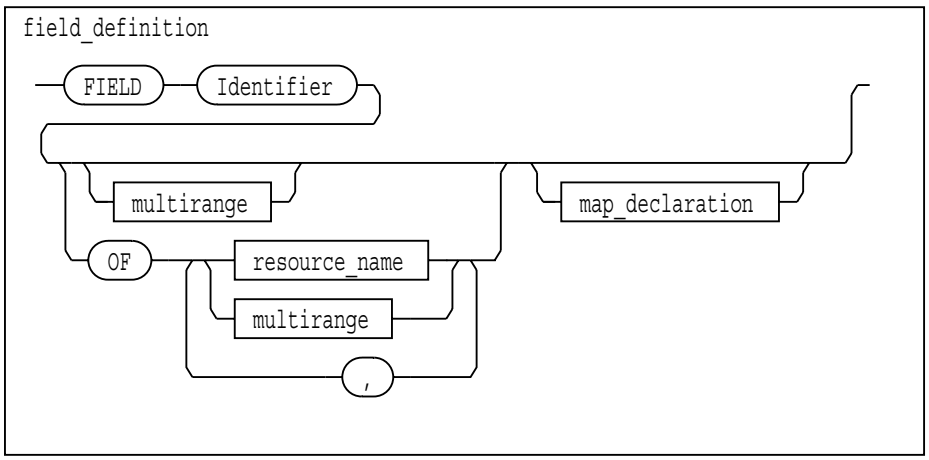
S13


resource_definition
register_definition
field_definition
device_definition
group_definition
set_definition

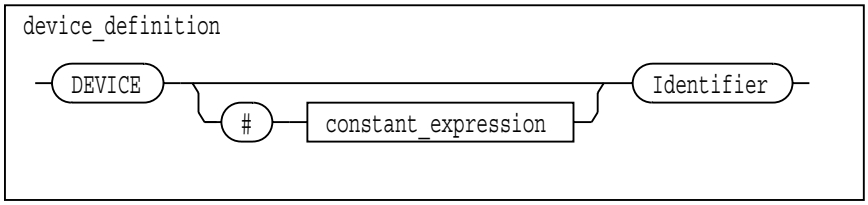
S14


register_definition
REGISTER
#
constant_expression
Identifier
array
range
MULTIPORT
constant_expression
READ
map_declaration
&&
constant_expression
WRITE
##

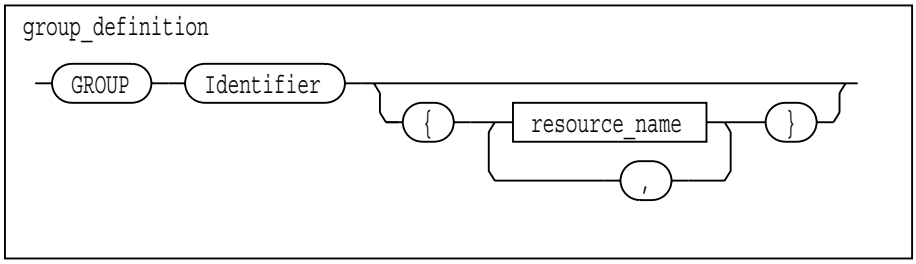
S15


field_definition
FIELD
Identifier
multirange
map_declaration
OF
resource_name
multirange
,

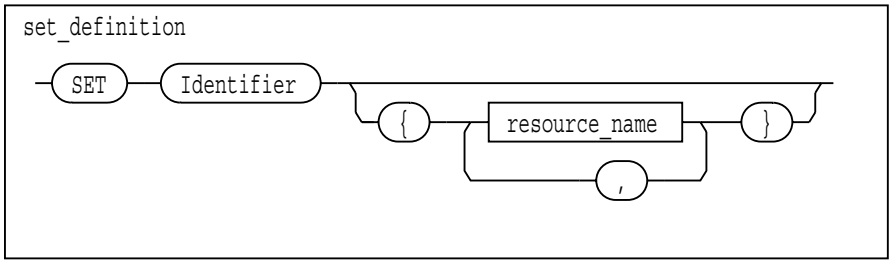
S16



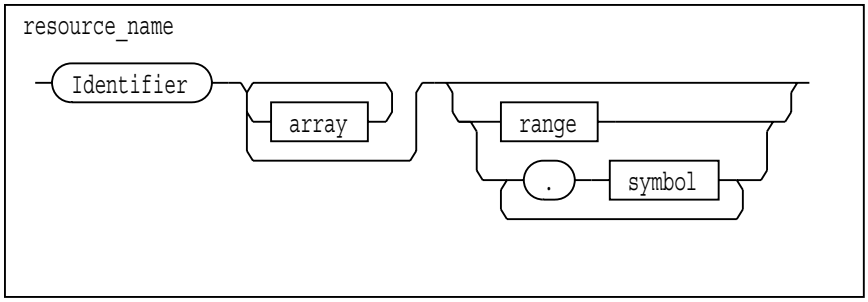
S17



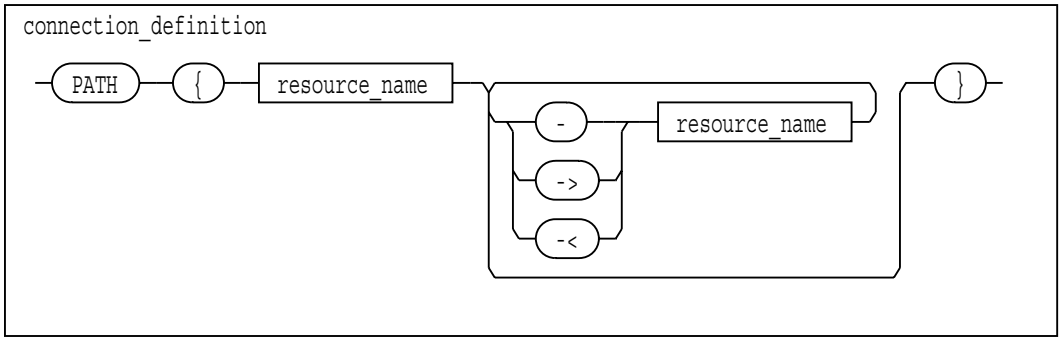
S18



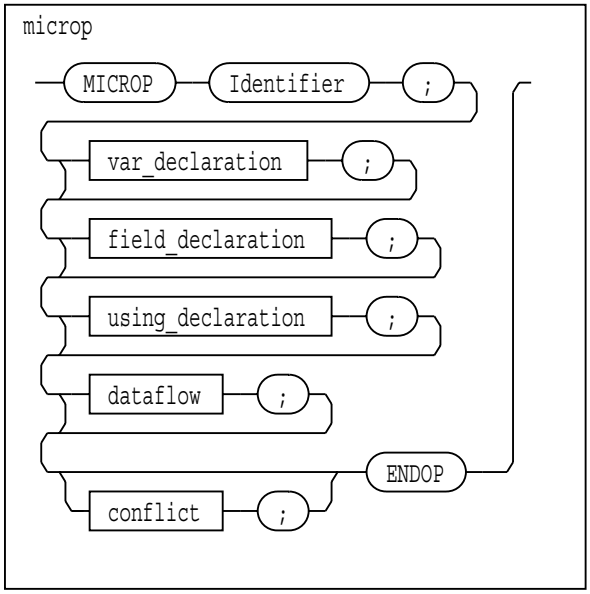
S19



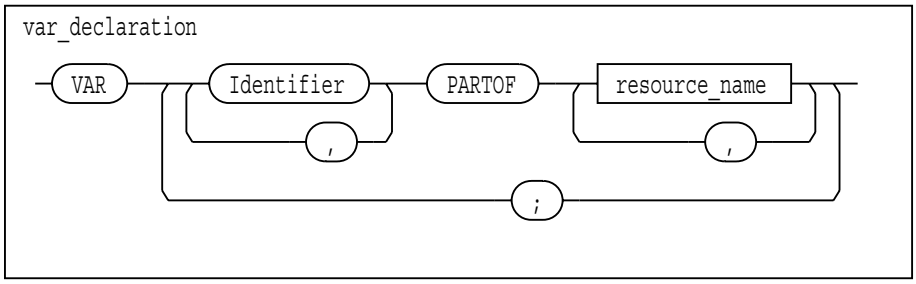
S20


connection_definition
PATH
{
resource_name
-
->
-<
resource_name
}

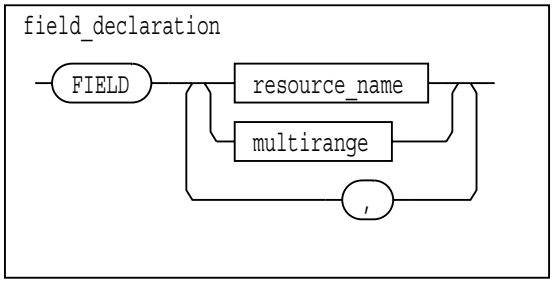
S21


microp
MICROP
Identifier
;
var_declaration
;
field_declaration
;
using_declaration
;
dataflow
;
ENDOP
conflict
;

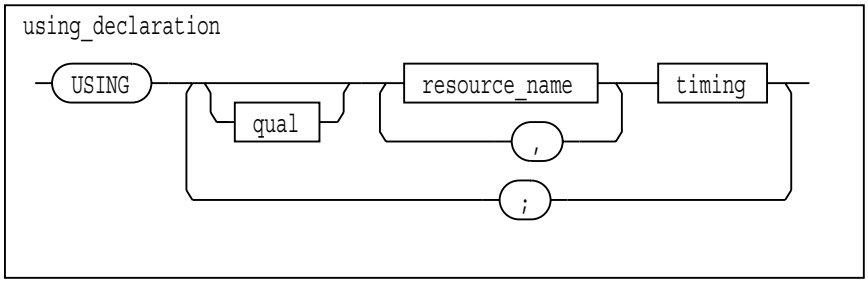
S22


var_declaration
VAR
Identifier
,
PARTOF
resource_name
,
;

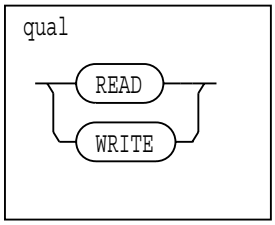
S23



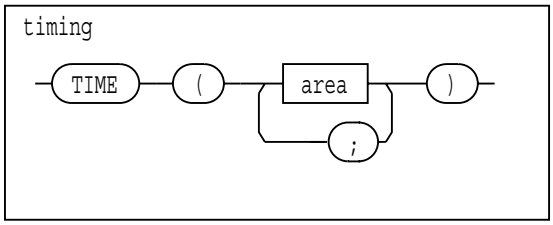
S24



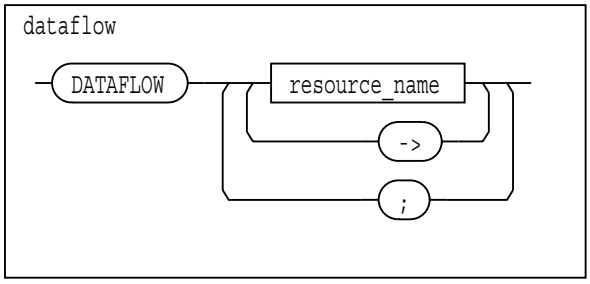
S25



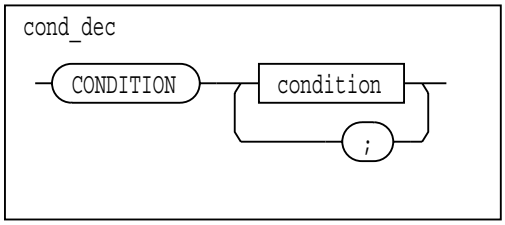
S26



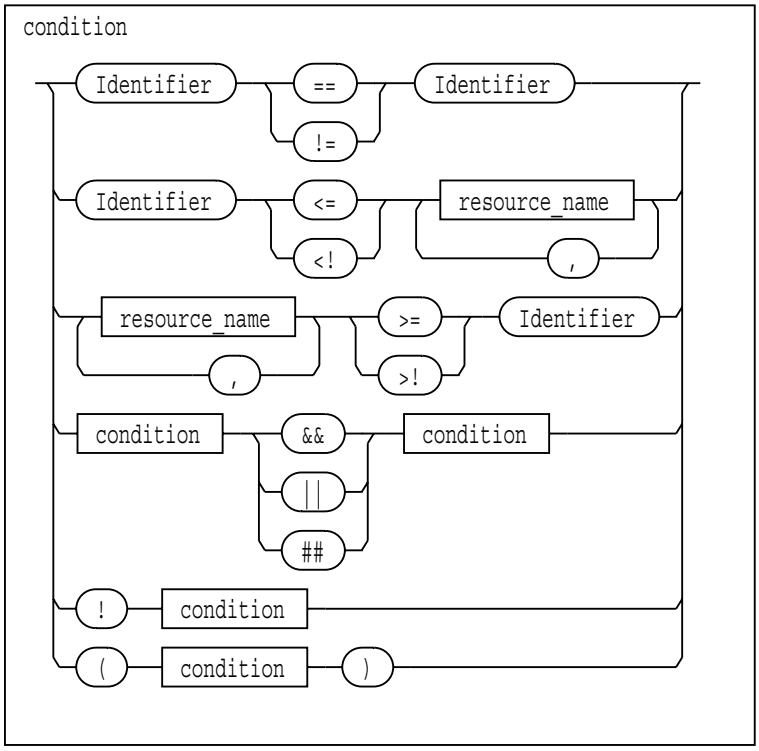
S27


dataflow
DATAFLOW
resource_name
->
;

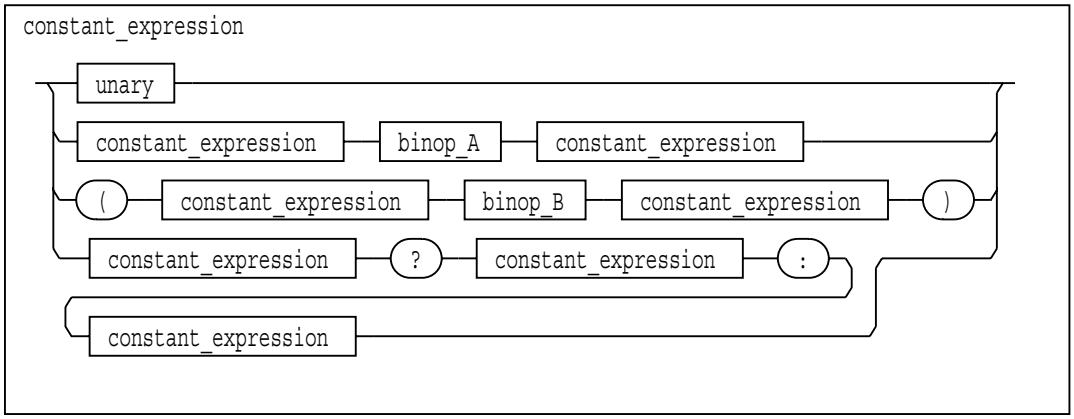
S28


cond_dec
CONDITION
condition
;

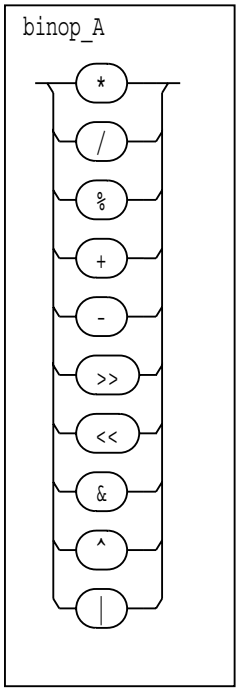
S29


condition
Identifier
==
!=
Identifier
Identifier
<=
<!
resource_name
,
resource_name
,
>=
>!
Identifier
condition
&&
||
##
condition
!
condition
(
condition
)

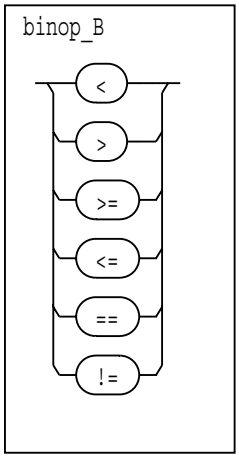
S30


constant_expression
unary
constant_expression
binop_A
constant_expression
(
constant_expression
binop_B
constant_expression
)
constant_expression
?
constant_expression
:
constant_expression

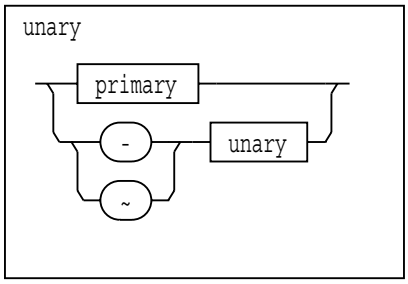
S31


binop_A
*
/
%
+
-
>>
<<
&
^
|

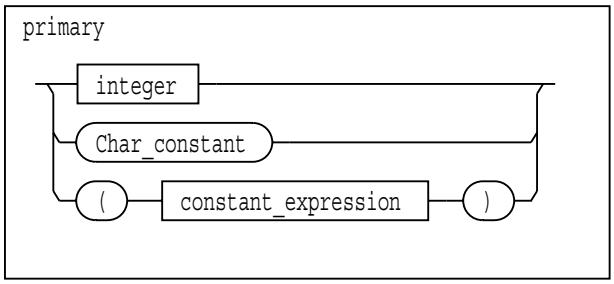
S32



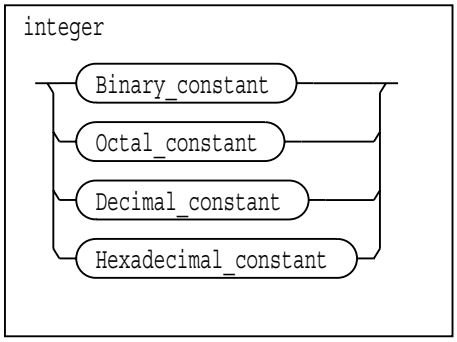
S33



S34



S35



Abbildungsverzeichnis

Tabellenverzeichnis

Beispielverzeichnis

Literaturverzeichnis

[1] Strutynski/Haff: *Funktionsbeschreibung und Mikrobefehlsliste P1.* Siemens DFY 3.3.2.3.27 **1979**

[2] Baumgartner: *Mikroprogrammsprache des Y–Prozessors P1.* Siemens DFY 3.3.2.3.25 **1979**

[3] Handbuch: *Mikado: MPL–Y Sprachbeschreibung.* Siemens **1991**

[4] Lutz: *Sprachbeschreibung: Problemsprache für Mikrobefehlsnotation.* Siemens **1981**

[5] W. Wölfel: *Programmgenerator PG – Benutzerhandbuch und Beispiele.* Siemens AG **1988**

[6] M. Sonntag: *Scannergenerator SG.* Siemens AG **1990**

[7] M. J. Eager: *M29 – An Advanced Retargetable Microcode Assembler.* MICRO–16 **1983**

[8] R. W. Beauchamp/N. R. Firth: *UDSYS a microcode development system.* MICRO–15 **1982**

[9] K. Schmidt u. a.: *Mikroprogrammierbare Schnittstellen.* Teubner Stuttgart **1984**

[10] Manual: *IEEE Standard VHDL Language Reference Manual.* IEEE **1988**

[11] M. R. Barbacci a. o.: *The ISPS Computer Description Language.* Technical Report, Department of Computer Science, Carnegie–Mellon University **1977**

[12] J. L. Gieser/R. J. Sheraga: *Experiments in Automatic Microcode Generation.* IEEE Transactions on Computers, C–32(6), 557–569 **1983**

[13] J. L. Gieser/R. J. Sheraga: *Microarchitecture Description Techniques.* MICRO–15, 23–32 **1982**

[14] W. Damm: *Design and Specification of Microprogrammed Computer Architecture.* MICRO–18, 3–9 **1985**

[15] S. Dasgupta/M. Olafsson: *Towards a Family of Languages for the Design and Implementation of Machine Architecture.* 9th Symposium on Computer Architecture, SIGARCH Notices, 158–170 **1982**

[16] S. Dasgupta: *On the Verification of Computer Architecture Using an Architecture Description Language.* 10th Symposium on Computer Architecture, SIGARCH Notices, 32–38 **1983**

[17] B. W. Kerninghan/D. M. Ritchie: *Programmieren in C.* Carl Hanser Verlag München Wien **1983**

[18] A. T. Schreiner/G. Friedman: *Compiler bauen mit UNIX – Eine Einführung.* Carl Hanser Verlag München Wien **1985**

[19] M. Banahan/A. Rutter: *UNIX – lernen, verstehen, anwenden.* Carl Hanser Verlag München Wien **1984**

[20] D. A. Patterson/J. L. Hennessy: *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers, Inc. **1990**

[21] B. J. Catanzaro *The SPARC Technical Papers.* Springer Verlag **1991**

[22] C. W. Fraser/R. R. Henry/T. A. Proebsting: *BURG — Fast Optimal Instruction Selection and Tree Parsing.* `ftp: kaese.cs.wisc.edu in pub/burg.shar.Z` **1991**

[23] H. Emmelmann/F. W. Schröer/R. Landwehr: *BEG — a Generator for Efficient Back Ends.* ACM SIGPLAN **1989**

[24] T. Staudinger: *SAB 88C166: Flash-EPROM-Controller setzt neue Maßstäbe.* Siemens Components März/93, 95–97 **1993**

[25] K. Diefendorff, M. Allen *Organization of the Motorola 88110 Superscalar RISC Microprocessor.* IEEE Micro **April 1992**

[26] *MC88100 RISC Microprocessor User's Manual.* Motorola, Inc. **1990**

[27] *R4000 User's Manual.* MIPS Computer Systems, Inc. **1991**

[28] *Alpha Architecture Handbook.* Digital Equipment Corporation **1992**

Microprogramming and Firmware Engineering

[29] T. G. Rauscher/P. M. Adams: *Microprogramming: A Tutorial and Survey of Recent Developments.* IEEE Transactions on Computers, Volume C–29, Number 1, 2–20 **1980**

[30] N. Tredennick: *The "Cultures" of Microprogramming.* MICRO–15, 79–83 **1982**

[31] N. Tredennick: *The Impact of VLSI on Microprogramming.* MICRO–19, 2–5 **1986**

[32] D. Landskov a. o.: *Local Microcode Compaction Techniques.* Computing Surveys, Volume 12, Number 3, 261–294 **1980**

[33] D. T. Wang: *Defensive Microprogramming.* MICRO–15, 84–88 **1982**

[34] W. J. Tracz: *Advances in Microcode Support Software.* MICRO–18, 57–59 **1985**

[35] A. van Dam a. o.: *Simulation of a Horizontal Bit–Sliced Processor Using the ISPS Architecture Simulation Facility.* IEEE Transactions on Computers C–30 **1981**

[36] J. F. Nixon a. o.: *A microarchitecture description language for retargeting firmware tools.* MICRO–19 **1986**

[37] M. Sint: *A survey of High Level Microprogramming Languages.* MICRO–13, 141–152 **1980**

[38] S. Dasgupta: *Some Aspects of High Level Microprogramming.* Computing Surveys, Volume 12, Number 3, 296–323 **1980**

[39] S. Davidson: *High Level Microprogramming – Current Usage, Future Prospects.* MICRO–16, 193–200 **1983**

[40] S. R. Vegdahl: *The Design of an Interactive Compiler for Optimizing Microprograms.* MICRO–18, 129–135 **1985**

Microprocessing and Microprogramming 1990

[41] A. T. Balou/A. N. Refenes: *Designing a Parallel Object–Oriented Compiler Target Language (TOOL).* Department of Computer Science, University College London **1990**

[42] M. Auguin a. o.: *From Program to Hardware: A parallel Architecture Compiler.* Laboratoire de Signaux et Systèmes, Université de Nice et de Sophia–Antipolis **1990**

[43] E. Accomazzo a. o.: *Integrated Intermediate Code Optimization with Retargetable Code Generation.* Dipartimento di Matematica, Universita di Genova **1990**

[44] P. J. de Graaff: *On the Formal Specification and Verification of Digital Circuits.* Eindhoven University of Technology **1990**

[45] S. Antoniazzi/M. Mastretti: *An Interactive Environment for Hardware/Software System Design at the Specification Level.* Central Research Laboratories, Italtel SIT **1990**

[46] E. E. Pissaloux: *A Rational Methodology for Design of New Computer Structures.* Université Paris XI **1990**

[47] U. Wienkop: *Behavioral Circuit Description on System Level.* Siemens AG **1990**

[48] R. Gajda: *Enhancing a Control Graph Based HDL for Performance Evaluation of Simulated Architectures.* Institute of Computer Science, Polish Academy of Sciences **1990**

Microprocessing and Microprogramming 1991

[49] E. S. T. Fernandes a. o.: *Micro-instruction Placement by Stimulated Annealing.* Programa de Engenharia de Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro **1991**

[50] W. Wrona/A. Pawlak: *VLSI Integrated Circuit Design Representation in an Object–Oriented CAD Environment.* Poland/Germany **1991**

[51] C. Charlton a. o.: *Object–Oriented Modelling in Digital Circuit CAD Systems.* Department of Computer Science, University of Liverpool **1991**

[52] Yun–Chao Hu a. o.: *Object Oriented System Analysis for VLSI.* Eindhoven University of Technology **1991**

[53] M. Schäfer/G. Klein–Hessling: *A Design Concept for Verified Concurrent Controllers.* Siemens AG **1991**

[54] P. Tsanakas a. o.: *A PROLOG Based Design Environment for the High Level Synthesis of Application Specific Architectures.* National Technical University of Athens **1991**

[55] S. Antoniazzi/M. Mastretti: *An Architectural Design Support Environment for High Performance Digital Systems.* Central Research Laboratories, ITALTEL **1991**

[56] L. P. M. Benders/M. P. J. Stevens: *Task Level Behavioral Hardware Description.* Eindhoven University of Technology **1991**

[57] R. D. Dowsing/R. Elliot: *A Higher Level of Behavioral Specification: An Example in Interval Temporal Logic.* School of Information Systems, UEA, Norwich **1991**

[58] J. Sziray/Z. Nagy: *OPART: A Hardware Description Language for Test Generation.* Computer Research and Innovation Center, Budapest **1991**

Wescon Conference Record 1990

[59] P. D. Lindemann: *Top–Down Design Synthesis using VHDL.* ASIC Business Group, Racal–Redac, Westford **1990**

[60] D. Jakopac: *VHDL Modelling: From Concept to Gates.* Vista Technologies, Inc. **1990**

[61] G. House: *Mixed–Signal, Multi–Level Simulation with VHDL.* Mentor Graphics Corporation, Oregon **1990**

[62] D. R. Coelho: *Advanced Simulation and Debugging in the VHDL Environment.* Vantage Analysis Systems, Inc. **1990**

[63] G. M. Nurie: *Linking VHDL and Test.* ExperTest, Inc. **1990**