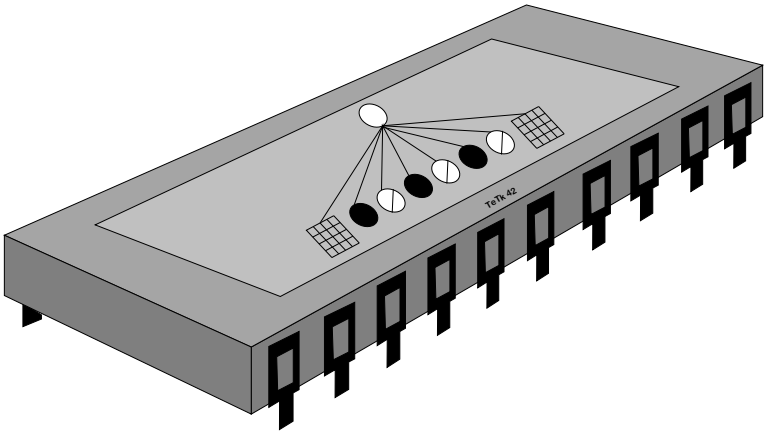
UNIVERSITÄT DORTMUND

LEHRSTUHL INFORMATIK XII

Hardwareunterstützung octreebasierter 3D–Graphik

Torsten Emmerich Thomas Kamphans

26. März 1997



Diplomarbeit

Gutachter:
Prof. Dr. Peter Marwedel
Dr. Rainer Leupers

const int answer = 42;
Bjarne Stroustrup:
The C++ programming language

Vorwort

Die vorliegende Abhandlung dokumentiert die Entwicklung einer Hardware zur Unterstützung octreebasierter 3D–Graphik; eine Arbeit, die mir — trotz allem "Implementationsärger" — sehr viel Spaß gemacht hat, vereinigte sie doch in idealer Weise meine Interessen– und Studienschwerpunkte Rechnerarchitektur und Computergraphik und gab mir die Möglichkeit, einen tiefen Einblick in jene Problematiken des Hardwareentwurfs zu erhalten, die in Lehrbüchern — wenn überhaupt — nur sehr oberflächlich behandelt werden. Um so mehr bedauere ich, daß die Zeit nicht ausreichte, den entworfenen Chip konkret herzustellen und Resultate zu erhalten, die über eine Simulation hinausgehen.

Ich danke vor allem meinen Eltern Friedrich und Brigitte Kamphans, deren ständige Unterstützung in vielen Dingen mein Studium sehr erleichtert hat. Für fachlichen Rat danke ich den Mitarbeitern des Lehrstuhls Informatik XII der Universität Dortmund, insbes. Herrn Prof. Dr. Peter Marwedel und Herrn Dr. Rainer Leupers, Herrn Dr. Peter Dietz für seine „Vorlage", Torsten Emmerich für die gute Zusammenarbeit — bereits seit dem ersten Semester —, Herrn Dipl. Inform. Stefan Wohlfeil von der Fernuniversität Hagen für die XGraphik–Bibliothek, den Mitarbeitern des Lehrstuhls VII für die Graphikpakete „MING" und „Visual", Stefan Kamphans für seine „AutoCAD–Übungen" und nicht zuletzt Dorthe Lübbert für 'Kommas', FTP und vieles andere.

Unna, im März 1997
Thomas Kamphans

Den zuvor von Thomas gemachten Ausführungen in Bezug auf diese Arbeit kann ich mich eigentlich nur noch kommentarlos anschließen.

Ganz besonders möchte ich meinen Eltern Wilhelm und Bernhardine Emmerich danken, die mir immer zur Seite standen und durch ihre Unterstützung vieles im Ablauf meines Studiums erleichtert haben. Weiterhin danke ich den Mitarbeitern des Lehrstuhls Informatik XII der Universität Dortmund, vor allem Herrn Prof. Dr. Peter Marwedel und Herrn Dr. Rainer Leupers, für ihren fachlichen Rat, sowie Herrn Dr. Peter Dietz für sein „gelbes Werk"[1]. Thomas Kamphans für die Zusammenarbeit auf unserem gemeinsamen Weg durchs Studium und meinen Freunden die mir vieles erleichtert haben.

Unna, im März 1997
Torsten Emmerich

Inhaltsverzeichnis

1 Einleitung **1**

2 Theoretische Grundlagen **5**

2.1 Octrees als Datenstruktur zur 3D-Graphik 5

2.2 Operationen im 3D-Raum 14

3 Implementierung in C **21**

3.1 Systemarchitektur 21

3.2 Datenstrukturen 26

3.3 Low–Level–Operationen 30

3.4 High–Level–Operationen 44

3.5 Laufzeitmessungen 58

4 Hardwareentwurf **59**

4.1 Spezifikation des Octchip 59

4.2 Grobentwurf 63

4.3 Grundlegende Komponenten 64

4.4 Rechenwerk 68

4.5 Steuerwerk 98

4.6 Mikroprogramme 105

5 Synthese und Simulation **113**

5.1 Hardware–Entwurfsprozeß 113

5.2 Testsynthese 120

5.3 Simulationsumgebung 126

5.4 Resultate 135

6 Zusammenfassung und Ausblick **143**

6.1 Zusammenfassung 143

6.2 Ausblick 144

A Octchip Referenzen **149**

A.1 Opcodes des Octchip 149

A.2 Register des Octchip 150

A.3 Mnemoncis des Mikrorom–Compilers 151

A.4 Mikrorom 153

B Autoren **159**

Abbildungsverzeichnis

2.1 Positionen der Subkubi 5

2.2 Rasterung einer Fläche und entsprechender Quadtree 6

2.3 Aktualisierung des Mittelpunktsvektors bei Navigationsoperationen 7

2.4 Aufbau eines Bitblocks 10

2.5 Beispiel eines Kubi auf Niveau $n = 2$ und seine Darstellung in einem Bitblock 10

2.6 Positionen eines Kubi 11

2.7 Positionen in den Bitvektoren eines Bitblocks 12

2.8 Beispieloctree 13

2.9 Speicherauszug 13

2.10 Transformationen 14

2.11 Mengenoperationen 15

2.12 Hüllenbildung 15

2.13 Verschiebung eines Kubus 16

2.14 Auswahl der adäquaten Bits bei Schiebe– bzw. Rollenoperationen entsprechend der positiven Richtung der jeweiligen Achse 19

3.1 Module der Softwarelösung 22

3.2 Benutzeroberfläche mit „angepinnten“ Menüs 25

3.3 Zeiger auf octree 28

3.4 Rasterkorrektur 35

3.5 Spiegeln am Ursprung mit Bitblöcken 40

3.6 Verschieben eines Kubus 44

3.7 Verschieben eines Bitblockes 48

3.8 Verschieben von Bitblöcken 48

3.9 Schnittebenendarstellung 51

3.10 Benachbarte Kubi 53

3.11 Drahtmodelle: Fallunterscheidung in wire_proc 53

3.12 Triangulierung einer Kubusseitenfläche 54

3.13 Visualierung unter Dos 56

3.14 Drahtmodell in MING 57

3.15 Schattiertes Modell in MING 57

4.1 Timing–Diagramm eines Speicherzugriffs 60
4.2 Anbindung des Octchip an PCI bzw. SCSI–Bus 62
4.3 Blockschaltbild des Octchip 63
4.4 Schaltbild des Input/Output-Treibers 64
4.5 Internes Timing 65
4.6 Schieberegister 67
4.7 Adreßdekodierung 69
4.8 Instruktionsregister 71
4.9 Sync. Problem 72
4.10 Flankendetektor (Schaltbild) 73
4.11 Flankendetektor (Timing) 74
4.12 Register für HOLD 74
4.13 Statusregister 75
4.14 Adreßberechnung 76
4.15 Adreßlogik 77
4.16 Register mit Stack 79
4.17 Interne Struktur des Positionsregisters 81
4.18 Mittelpunktsberechnung 83
4.19 Interne Struktur des Niveauregisters 85
4.20 Aufbau einer Niveau–Einheit 85
4.21 Register für Knoteninhalte 86
4.22 Aufbau von FullEmpty 87
4.23 Strukturaufbau der Bitblockeinheit 89
4.24 Aufbau des Speicherelements 91
4.25 weitere Komponenten eines Bitblockregisters 92
4.26 Elemente des Hilfs–Bitblockregisters 93
4.27 Verfahren des gekoppelten Schiebens 94
4.28 Elemente der Bitblockregisters der Modelle 0 und 1 94
4.29 Elemente des Bitblockregisters für Modelle 2 95
4.30 Bitblockalu 96
4.31 Realisierung von rotate 97
4.32 Blockschaltbild des Steuerwerks 98
4.33 Schaltbild des Mikroprogrammzählerblocks 102
4.34 Schaltbild der Komparatorlogik 104

5.1 Der Hardwareentwurfsprozeß 114

5.2 Die Schaltzentrale: der design analyser 116
5.3 Verzeichnisstruktur der Synopsys Arbeitsumgebung 117
5.4 Die Fenster des Simulators: Debugger und Waveform–Viewer 118
5.5 Stuck–at–fault 120
5.6 Interner Scan–Test 121
5.7 Register für internen Scan–Test 121
5.8 JTAG–Boundary–Scan–Test 123
5.9 Simulation des Gesamtentwurfs 126
5.10 Zeitbedarf und Registerzugriffe: .time–Datei 127
5.11 Beispiel einer Stimuli–Datei (Test der Navigationsroutinen) 127
5.12 Testbench 129
5.13 Mittelpunktseinheit (hier: Core) mit umgebender Boundary–Scan–Logik 136
5.14 set_node–Test 137
5.15 unite–Test: zwei Modelle und das Ergebnis 138
5.16 Time/Area–Tradeoff bei Bitblocklogik und Mittelpunktsberechnung 140

6.1 Berechnung von Left und Right 146

Tabellenverzeichnis

2.1 Grundlegende Baumoperationen 6
2.2 Aktualisierung des Mittelpunktsvektors bei Navigationen 8
2.3 Übersicht der Kenndaten des aktuellen Kubus 8
2.4 Anzahl der Knoten 9

3.1 Compileroptionen 23
3.2 Befehle im Batchmode 24
3.3 Kommandozeilenparameter 24
3.4 Fallunterscheidungen bei move_compose 47
3.5 strukturkonforme Transformationen 49
3.6 Anteil der Low–Level–Operationen an High–Level–Operationen 58

4.1 Steuerleitungen des Octchip 61
4.2 Generische Komponenten 66
4.3 Inhalt des Statusregisters 72
4.4 Codierung von output_sel(0:1) 81
4.5 Belegung des Opcode–Bus der MIDDLE-Einheit 84
4.6 Opcodes der MIDDLE–Einheit 84
4.7 Belegung von n_satus(0:8) 85
4.8 Operationen der Bitblockalu 97
4.9 Fallunterscheidung für den Folgezustand des μPC 100
4.10 strukturkonforme Transformationen 111

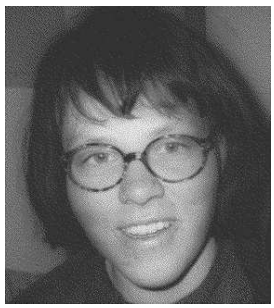
5.1 Laufzeiten ausgewählter Testoperationen 135

Kapitel 1

Einleitung

Mit der stetig steigenden Leistung moderner Mikroprozessoren gewinnt auch der Bereich der Computergraphik immer mehr an Bedeutung. Die Zeiten der ASCII–Terminals sind wohl vorüber, für jeden halbwegs zeitgemäßen Rechner sind graphische Benutzeroberflächen verfügbar; leistungsfähige und komfortable Graphiksoftware ist selbst für den Heimanwender erschwinglich; „Virtual Reality“ ist ein Schlagwort, das nicht nur im Bereich der Unterhaltungselektronik aktuell ist ([15]), selbst im „Netz der Netze“ lernen dreidimensionale Objekte mit der 'Virtual Reality Modeling Language' (VRML) das Laufen ([17]). Trotz dieser Rechenleistungen werden ehrgeizige Projekte wie das „intelligente Zimmer“[1] ([16], [19]) oder virtuelle Räume, sog. CAVEs[2] ([20]), erst mit Hilfe spezieller Graphikhardware möglich.

In diesem Kontext ist auch die vorliegende Arbeit zu sehen. Operationen auf Octrees — grob gesagt eine komprimierte Darstellung eines Voxelmodells[3] — erfordern sehr viel Rechenzeit, so daß der Versuch, solche Operationen durch geeignete Hardware zu beschleunigen naheliegt.



Doch davor stellt sich die Frage, warum es sich überhaupt lohnt, sich mit Octree–Operationen zu beschäftigen. Schließlich erweist sich jedes auf analytischen Objekten[4] basierende System[5] einem octreebasiertem System überlegen — bei gleichen Modellen wohlgemerkt! Der Vorteil octreebasierter Systeme ist derselbe wie der pixelorientierter Graphiksysteme (wie XPaint) gegenüber vektororientierten Systemen (wie XFig): Bei steigender Bildkomplexität — etwa bei eingescannten Graphiken — lassen sich bestimmte Operationen auf Pixel– bzw. Voxelmodellen schneller handhaben. Auch der Speicherbedarf dieser Modelle ist ab einer bestimmten Bildkomplexität kleiner. Betrachten wir nebenstehende Abbildungen: Das Original der oberen von beiden ist eine 286×322 Pixel große Graphik mit 8 Bit Farbtiefe, ihr Platzbedarf als GIF–Datei liegt bei $\approx 40KB$. Die untere Abbildung ist eine mit Hilfe der CAD–Software Micrografx Designer 4.0 vektorisierte Version[6]der oberen Abbildung. Ihr Platzbedarf im Designer–Dateiformat beträgt $\approx 500KB$, im Postscript–Format sogar $\approx 1MB$. Der Nachteil dieser Systeme liegt auf der Hand: Jede Information über die ein Bild komponierenden Objekte geht verloren, es ist also nicht möglich, einzelne Objekte bzw. deren Attribute (Radius eines Kreises, Schriftart eines Textes, Stützpunkte einer Bézierkurve usw.) zu modifizieren.

[1] eine Art dreidimensionale Videokonferenz mit integriertem interaktivem Informationssystem

[2] Räume, in die dreidimensionale Objekte projiziert werden, die vom Betrachter interaktiv inspiziert oder gar modifiziert werden können; Einsatzgebiete sind z.B. „Virtual Prototyping“ und Montagesimulationen

[3] das dreidimensionale Äquivalent eines Pixels

[4] geometrische Körper, Polygonflächen ...; siehe [9], [11]

[5] z.B. Micrografx Designer oder Autodesk AutoCAD im Bereich Konstruktion, Autodesk 3–D Studio im Bereich Rendering und Animation

[6] d.h. es wurde versucht, die Konturen des Bildes mit analytisch beschreibbaren Objekten zu approximieren

Die Anwendungsbereiche von Octrees liegen also überall dort, wo komplexe, dreidimensionale Objekte — etwa Bilddaten von 3–D Scannern — verarbeitet werden müssen. Ein Blick in einschlägige Fachliteratur zeigt ein weites Spektrum konkreter Anwedungsbreiche wie Maschinenbau (CNC), Schiffs– und Flugzeugbau ([5], [1], Kap. 1), Medizintechnik ([7]), Astronomie oder Robotik ([8]). In Raytracing–Systemen können Octrees zur Optimierung der Schnittpunktberechnung eingesetzt werden ([21], [22]); eine Verwendung, die im Rahmen dieser Arbeit allerdings nicht verfolgt wird.

Die Hoffnung, octreebasierte Operationen durch geeignete Hardware beschleunigen zu können, basiert auf zwei Beobachtungen:

- Bestimmte Operationen lassen sich auf klassischen Mikroprozessoren nicht optimal durchführen. Herausragendes Beispiel für diesen Fall ist die Operation rotatebit, die die Reihenfolge der Bits in einem 16 Bit Wort vertauscht:[7]

$$b = \text{rotatebit}(a) :\Leftrightarrow b_i = a_{15-i} \;\; \forall i \in \{0, \ldots, 15\}$$

- Andere Operationen könnten von einer speziellen Hardware parallel bearbeitet werden, während sie auf einem klassischen System sequentiell berechnet werden müssen. Als zeitlicher Gewinn dieser Kategorie wird sich die Berechnung von Mittelpunktsvektoren beim Durchlaufen eines Octrees erweisen.

Grundlage der Arbeit ist die 1994 an der Universität Dortmund verfaßte Dissertation „Ein Computermodell auf der Basis kubischer Volumina für die Approximation beliebig geformter körperlicher Objekte“ ([1]). Dort werden auf Octrees operierende Algorithmen sowie ein Ansatz zur Hardwareunterstützung dieser Operationen vorgestellt. Leider wurden weder die Algorithmen noch der Hardwareentwurf in der Praxis verifiziert.

Gegenstand dieser Arbeit ist es nun, jenes Versäumnis nachzuholen und sowohl die Octree–Operationen, als auch deren Hardwareunterstützung zu implementieren, mit dem Idealziel, einen Hardwareentwurf zu erstellen, der zur Fertigung eines ASIC[8] oder zur Programmierung von FPGAs[9] dienen kann (siehe auch [34]). Ein damit verbundenes Ziel war die Erstellung eines Software–Systems zur Durchführung von Operationen auf Octrees, die Arbeit an diesem System diente gleichzeitig der Einarbeitung in die Thematik „Octree“. Die in der ersten Phase implementierten Operationen ließen sich in zwei Klassen einteilen: Die erste Klasse bilden die Operationen, die direkt auf den Datenstrukturen operieren, die zweite Klasse Operationen auf höherem Niveau, die nur über die Operationen der ersten Klasse auf Datenstrukturen zugreifen.

In der zweiten Phase der Arbeit wurden Operationen der ersten Klasse in Hardware implementiert, die Operationen der zweiten Klasse sind dann als Applikationen der Hardware zu verstehen. Grundlage der Hardwareimplementierung bildete die im Anhang von [1] angeführte „Implementierung einer angepaßten Architektur“. Zur Beschreibung der Hardware diente die Hardwarebeschreibungssprache VHDL ([27], [24], [25], [26]).

Bei der Realisierung dieser Ziele traten vor allem folgende Probleme auf:

- Die in der Dissertation vorgestellten Algorithmen sind nicht erprobt und enthalten eine Reihe zum Teil schwerwiegender Fehler — oftmals nur Kleinigkeiten wie vertauschte Indices oder Druckfehler wie falsche Einrückung[10] —, Vereinfachungen oder in der Theorie nicht berücksichtige Sonderfälle.

- Die in [1] vorgestellte „Implementierung einer angepaßten Architektur“ ist recht lücken– und fehlerhaft. Vorgestellt wird ein Block von Registern und Mikroprogramme. Ein Steuerwerk fehlt ebenso wie ein Interface; hinzu kommen einige Vereinfachungen. So wird z.B. auf 24 Bit

[7] b_i bezeichne das i. Bit des Wortes b
[8] Application Specific Integrated Circuit; kundenspezifisch hergestellter Chip
[9] Field Programmable Gate Array; anwenderprogrammierbarer Chip
[10] der Autor verzichtet auf Anweisungsklammern

breite Register über 16 Bit breite Busse zugegriffen, im Schaltplan in [1] Anhang 1 sind einige Verdrahtungen falsch, andere fehlen[11].

- Die verwendete Synthese– und Simulationssoftware, das Synopsys–System, weist einige Eigenheiten und Tücken auf, die in einem folgenden Kapitel näher betrachtet werden. Hinzu kommt die Komplexität dieser Synthesesoftware, die dazu führt, daß sich gute Syntheseergebnisse nur mit einer gewissen Erfahrung im VLSI–Entwurf, in der Programmierung mit VHDL und im Umgang mit dem Synopsys–System erzielen lassen[12].

Die Arbeit ist in folgende Kapitel untergliedert:

Grundlagen: In diesem Kapitel werden die Grundlagen und Hintergründe aus dem Bereich der Computergraphik — speziell zum Thema Octrees — dargelegt, auf denen die Soft– und Hardwareimplementationen basieren.

Softwareentwurf: Dieses Kapitel stellt die Implementierung der reinen Softwarelösung vor. Es wird zunächst ein Grobentwurf der Software sowie die Entwicklungsumgebung und die Benutzung des Programms vorgestellt. Danach wird die Implementierung der Datenstrukturen für Octrees und der direkt auf den Datenstrukturen arbeitenden Operationen, die die Grundlage für die Hardwarerealisierung bilden, erläutert. Die Benutzung der in den vorhergehenden Abschnitten vorgestellten Operationen wird am Beispiel der Verschiebung, der Reflektion und der Visualisierung eines Octreemodells demonstriert. Die hier vorgestellten Operationen sind nur eine kleine Auswahl der in der Softwarelösung implementierten High–Level–Operationen.

Hardwareentwurf: Dieser Teil der Arbeit befaßt sich mit der Implementierung der Hardware. Der Entwurf wird von einer „Black–Box“ Ansicht ausgehend „top–down“ vorgestellt.

Synthese und Simulation: Der Beschreibung des Hardwareentwurfs schließt sich eine Betrachtung des Hardwareentwurfsprozesses an. Dieser unterteilt sich in Synthese — dazu gehört die Generierung von Hardwarestrukturen aus der VHDL–Beschreibung, Logikoptimierung sowie die Abbildung des Entwurfs auf die gewählte Zieltechnologie — und Simulation zur Verifikation der Korrektheit und Messungen der Performance. Die Performance der Softwarelösung wird diskutiert und mit dem Resulaten der Hardwaresimulation verglichen.

Zusammenfassung und Ausblick: Eine Zusammenfassung der gewonnenen Erkenntnisse sowie auf Ausblick auf mögliche Erweiterungen des Chips und weitere Schritte im Prozeß des Hardwareentwurfs schließen die vorliegende Arbeit.

[11] An geeigneten Stellen finden sich weitere Hinweise zu diesem Thema, die jedoch keinen Anspruch auf Vollständigkeit erheben.

[12] Unter „Audiance“ ist in den Referenzhandbüchern des Synopsys–System folgendes zu finden: „[...] You should be familiar with the language standard IEEE VHDL 1076-1987. [...]“ ([46]), „[...] The manual assumes that you have a basic knowledge of Design Compiler, *and that you are a logic designer or an electronics engineer.*“ ([47])

Kapitel 2

Theoretische Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen vorgestellt. Wir beschränken uns dabei auf einen kurzen Abriß der Themen und behandeln nur die Aspekte, die zum Verständnis der Ausführungen in späteren Kapiteln beitragen. Für eine tiefergehende Darstellung der behandelten Themen sei auf die zugrundeliegende Literatur zum Thema „Octree“ und „Computergraphik“ verwiesen.

2.1 Octrees als Datenstruktur zur 3D-Graphik

In der vorliegenden Abhandlung werden Octrees oder Kubusbäume als Datenstrukturen zur Approximation beliebig geformter Körper benutzt. Betrachten wir zunächst, was sich hinter diesem Begriff verbirgt:

Gegeben sei ein beliebig geformter Körper $\mathcal{K}$ in einem kubusförmigen Raum $\mathcal{R}$ der Kantenlänge 2^N. Wir teilen den Raum in acht gleichgroße Subkubi mit den Kantenlängen 2^n mit $n = N - 1$ ein. n nennen wir hierbei das *Niveau* des Kubus. Subkubi $\mathcal{S}$, die vollständig innerhalb des Körpers $\mathcal{K}$ liegen[1], bezeichen wir als „voll“, vollständig außerhalb liegende[2] als „leer“ und betrachten diese nicht weiter. Solche Subkubi, die weder voll noch leer sind, müssen weiter in Subkubi vom Niveau $N-2$, $N-3$ usw. unterteilt werden. Auf diese Weise läßt sich der Körper $\mathcal{K}$ solange approximieren, bis die Subkubi das Niveau 0 erreicht haben. Kubi vom Niveau 0 werden als „voll“ bezeichnet, falls ihr Mittelpunkt innerhalb des Körpers $\mathcal{K}$ liegt, anderfalls als „leer“.

Nun repräsentieren wir jeden Kubus durch einen Knoten in einem Baum[3]. Volle Kubi werden auf Blätter mit der Markierung “1”, leere Kubi auf Blätter mit der Markierung “0” abgebildet. Die acht Subkubi eines unterteilten Kubus erscheinen im Octree als die acht Söhne des korrospondierenden Knotens. Die Reihenfolge der Söhne ergibt sich aus der Numerierung der Subkubi wie in Abb. 2.1. Die Wurzel des Baumes repräsentiert den Raum $\mathcal{R}$.
Im folgenden gehen wir davon aus, daß der Ursprung des Koordinatensystems mit dem Mittelpunkt des Raumes $\mathcal{K}$ übereinstimmt und die Achsen des Koordinatensystems parallel zu den Kanten des Raumes verlaufen. n bezeichne das Niveau eines Kubus, $l = 2^n$ seine Kantenlänge.
Abb. 2.2 zeigt ein Beispiel eines Quadtrees, des zweidimensionalen Äquivalents zum Octree.

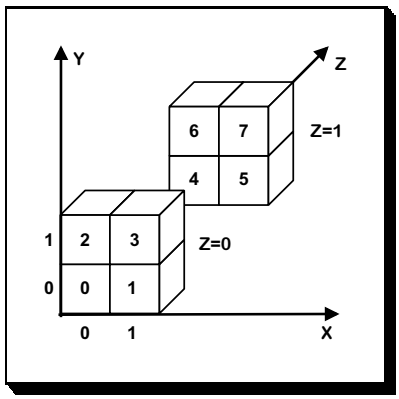


Abbildung 2.1: Positionen der Subkubi

[1] für die also $\mathcal{S} \cap \mathcal{K} = \mathcal{S}$ gilt
[2] für die also $\mathcal{S} \cap \mathcal{K} = \emptyset$ gilt
[3] wegen dieser Isomorphie werden wir die Begriffe „Knoten“ und „Kubus“ oft synonym verwenden

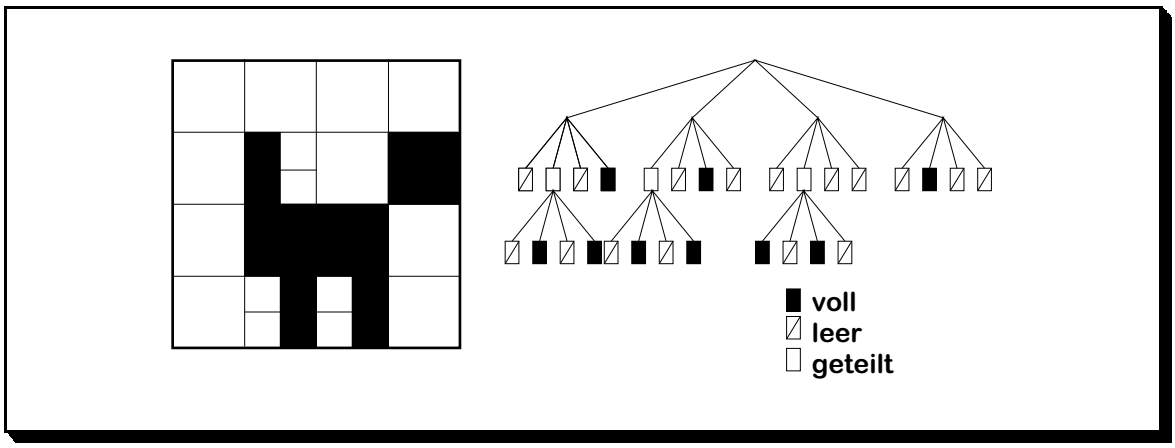


Abbildung 2.2: Rasterung einer Fläche und entsprechender Quadtree

2.1.1 Position und Index von Kubi

Jedem Subkubus ist entsprechend seiner Lage im Kubus eine eindeutige Position zugeordnet. Die Numerierung in Abb. 2.1 ergibt sich als oktale Darstellung der dreistelligen Binärzahl $P_{\text{bin}} = p_z p_y p_x$, wobei p_w die Lage des Subkubus auf der Achse w — relativ zum übergeordneten Kubus — bezeichnet.

Im folgenden bezeichne $\vec{p} = (p_z, p_y, p_x)$ mit $p_w \in \{0, 1\}$ die *vektorielle Position* eines Kubus, $p = 4p_z + 2p_y + p_x$ seine *oktale Position*.

Der Index dient zur eindeutigen Lokalisierung eines Kubus im Raum bzw. eines Knotens im Baum. Er entsteht durch Konkatenation der Positionen der auf dem Pfad von der Wurzel zum entsprechenden Knoten durchlaufenen Knoten. Analog zur Position bezeichne $\vec{P}$ den *vektoriellen Index*, P den *oktalen Index*.

2.1.2 Navigation im Octree

Bei den Algorithmen, die auf Octrees arbeiten, wird stets nur ein Knoten des Octrees betrachtet, den wir *aktuellen Knoten* nennen. Ein Wechsel des aktuellen Knotens nennen wir *Navigation* im Octree. Folgende Navigationsoperationen sind vorgesehen[4]:

root	Der Wurzelknoten wird zum aktuellen Knoten.
up	Der Vater des aktuellen Knotens wird aktueller Knoten.
down(p)	Der p. Sohn des aktuellen Knotens wird aktueller Knoten.
across(p)	p. Sohn des Vaters des aktuellen Knotens wird aktueller Knoten.
next	Der rechte Bruder[5] des aktuellen Knotens wird aktueller Knoten. Der rechte Bruder des 7. Knotens ist definiert als der 0. Knoten.

Daneben benötigen wir noch einige Operationen zum Aufbau von Octrees:

set_full	Der Inhalt des aktuellen Knotens wird auf „voll“ gesetzt.
set_empty	Der Inhalt des aktuellen Knotens wird auf „leer“ gesetzt.
attach	Der aktuelle Knoten wird geteilt; in den Kubusbaum werden 8 Söhne des aktuellen Knotens mit der Markierung „leer“ eingefügt.

Tabelle 2.1: Grundlegende Baumoperationen

[4] in [1] ist noch eine Operation upnext beschrieben, die jedoch an keiner Stelle gebraucht wird, → 3.3.5.2
[5] also der $p + 1$. Sohn des Vaters, wenn derzeit der p. Sohn aktueller Knoten ist

2.1.3 Kubusmittelpunkt

Neben dem Index ist auch der Mittelpunkt $\vec{m}$ eines Kubus in Raumkoordinaten von Interesse.

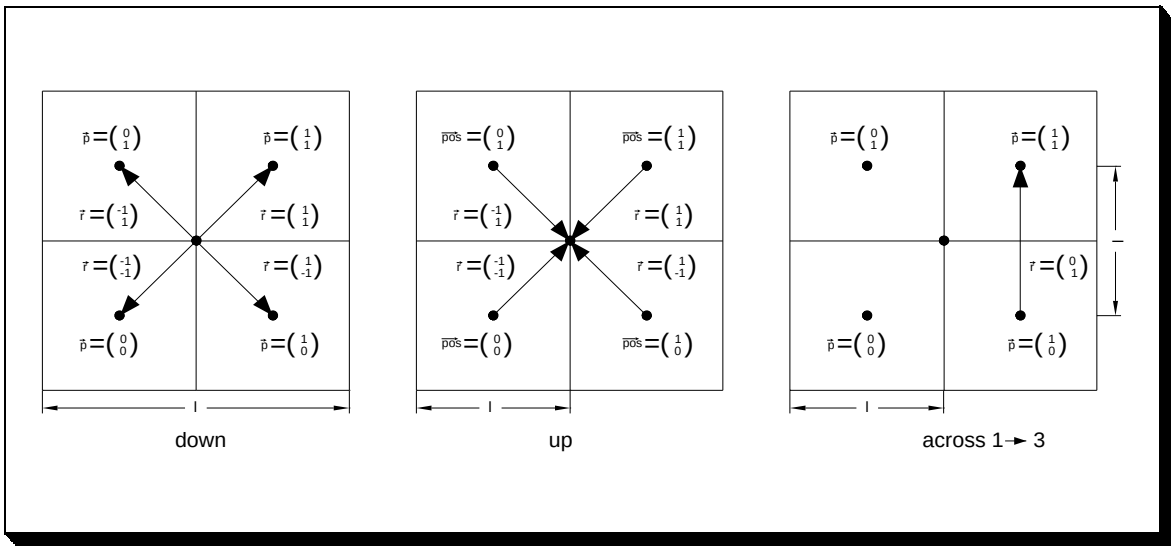


Abbildung 2.3: Aktualisierung des Mittelpunktsvektors bei Navigationsoperationen

Zur Berechnung des Mittelpunktes des aktuellen Knotens gibt es zwei Alternativen. Die erste Möglichkeit ist, einen Mittelpunktsvektor bei jeder Bewegung im Baum in der Form $\vec{m} := \vec{m} + \vec{f}$(Bewegung, Niveau) zu aktualisieren. Die andere Möglichkeit ist, eine Funktion zu implementieren, die bei Bedarf die Mittelpunktskoordinaten aus dem Index berechnet.

2.1.3.1 Aktualisierung eines Mittelpunktsvektors

Bei dieser Möglichkeit der Berechnung des Mittelpunktes des aktuellen Kubus muß ein Mittelpunktsvektor $\vec{m}$ bei jeder Navigation im Baum mittels root, up, down, across oder next aktualisiert werden[6]. Wir legen fest, daß das Niveau nach dem Mittelpunkt aktualisiert wird.

Abbildung 2.3 zeigt die Situationen in zweidimensionaler Projektion. next ist dabei nur ein Spezialfall von across.

Bei down wird zu jeder Komponente des aktuellen Mittelpunktsvektors ein Viertel der aktuellen Kubuslänge l addiert: $m_w := m_w + \frac{1}{4}\,\sigma \cdot l \quad (w = x, y, z)$. Das Vorzeichen σ hängt dabei von der Zielposition $\vec{p}$ ab. Analog dazu wird bei up mittels $m_w := m_w - \frac{1}{2}\,\sigma \cdot l$ aktualisiert, wobei σ vom Ausgangskubus abhängt.

Definieren wir also eine Funktion $\vec{u}$, die die Ausgangsposition $\overrightarrow{pos}$ bzw. die Zielposition $\vec{p}$ auf einen Richtungsvektor $\vec{r}$ abbildet, indem komponentenweise 0 auf -1 und 1 auf 1 abgebildet werden[7]:

$$\vec{u}(\vec{a}) := \big(u(a_z), u(a_y), u(a_x)\big) \qquad u(a) := \begin{cases} -1 \text{ für } a = 0 \\ 1 \text{ für } a = 1 \end{cases}$$

Auch bei across ist ein Richtungsvektor zu berechnen. Hier ergibt er sich jedoch als Differenzvektor aus der Zielposition $\vec{p}$ und der Ausgangsposition $\overrightarrow{pos}$.

[6] Diese Methode ist im Gegensatz zur Berechnung bei Bedarf nicht in [1] zu finden.
[7] Niveau 0 wird dabei bei der Implementierung als Spezialfall behandelt, → 3.3.4.1

Navigation	$\vec{r}$ berechnen	$\vec{r}$ skalieren	$\vec{m}$ aktualisieren
root			$\vec{m} := \vec{0}$
up	$\vec{r} := \vec{u}(\overrightarrow{pos})$	$\vec{r}\,' := \frac{l}{2} \cdot \vec{r} = 2^{n-1} \cdot \vec{r}$	$\vec{m} := \vec{m} - \vec{r}\,'$
down	$\vec{r} := \vec{u}(\vec{p})$	$\vec{r}\,' := \frac{l}{4} \cdot \vec{r} = 2^{n-2} \cdot \vec{r}$	$\vec{m} := \vec{m} + \vec{r}\,'$
across	$\vec{r} := \vec{p} - \overrightarrow{pos}$	$\vec{r}\,' := l \cdot \vec{r} = 2^{n} \cdot \vec{r}$	$\vec{m} := \vec{m} + \vec{r}\,'$
next	$\vec{r} := \mathrm{vec}(pos_okt + 1) - \overrightarrow{pos}$	$\vec{r}\,' := l \cdot \vec{r} = 2^{n} \cdot \vec{r}$	$\vec{m} := \vec{m} + \vec{r}\,'$

Tabelle 2.2: Aktualisierung des Mittelpunktsvektors bei Navigationen

Insgesamt sind also die in Tabelle 2.2 gezeigten Operationen durchzuführen[8]. Dabei ist vec(pos_okt+ 1)− $\overrightarrow{pos}$ so zu verstehen, daß zur aktuellen oktalen Position 1 addiert wird und von der vektoriellen Darstellung des Ergebnisses — dieses entspricht dem Zielkubus $\vec{p}$ bei across — die aktuelle Position vektoriell subtrahiert wird. Beispiel: Die aktuelle Position sei 5. Hierzu addieren wir 1 und stellen das Ergebnis in vektorieller Form[9] dar: $6_{\mathrm{okt}} = (1,1,0)_{\mathrm{vec}}$. Nun subtrahieren wir $\overrightarrow{pos}$: $(1,1,0)-(1,0,1) = (0,1,-1)$. Wie erwartet bewegen wir uns entlang der X-Achse in negativer Richtung, entlang der Y-Achse in positiver und entlang der Z-Achse gar nicht.

2.1.3.2 Berechnung des Mittelpunktes bei Bedarf

Diese Möglichkeit zur Berechnung des Mittelpunktes des aktuellen Knotens entspricht einer iterierten Anwendung der Formel für down. Dabei werden die im vektoriellen Index $\vec{P}$ gespeicherten Positionen durchlaufen:

```
FOR i := 0 TO N-n
```

$\vec{r} := \vec{u}(\vec{P}_i)$ $\qquad$ $\vec{P}_i$ bezeichnet die i.Komponente in P

$\vec{r} := \frac{l}{4} \cdot \vec{r}$

$\vec{m} := \vec{m} + \vec{r}$

n	Niveau des aktuellen Kubus
l	Seitenlänge des aktuellen Kubus, $l = 2^n$
$\vec{p}$	vektorielle Position des aktuellen Kubus
p	oktale Position des aktuellen Kubus
$\vec{P}$	vektorieller Index des aktuellen Kubus
P	oktaler Index des aktuellen Kubus
$\vec{m}$	Mittelpunkt des aktuellen Kubus
node	Inhalt des aktuellen Kubus (voll, leer, geteilt)

Tabelle 2.3: Übersicht der Kenndaten des aktuellen Kubus

[8] Die Navigation zu einem Knoten vom Niveau Null und von einem Knoten vom Niveau Null aufwärts sind dabei Spezialfälle: $\vec{r}$ wird hier nicht skaliert.

[9] praktisch also als Binärzahl

2.1.4 Bitblöcke

Dem Octree–Modell mit seiner Baumstruktur steht die Möglichkeit des Voxelmodells gegenüber, bei dem die Präsenz eines jeden Punktes des dreidimensionalen Objektraums durch ein Bit angezeigt wird. Das diese Möglichkeit für ein ganzes Modell recht speicherintensiv ist, wurde schon in der Einleitung dargelegt. Es ist aber auch nicht von der Hand zu weisen, daß auf den unteren Niveaus die Baumstruktur sehr in die Breite (siehe Tabelle 2.4) wächst und der Speicherbedarf für Zeigeroverheads auf den unteren Niveaus gegenüber der zu speichernden Information groß wird. An dieser Stelle setzen die Bitblöcke an: Es wird ein Teil der Baumstruktur und ihrer dargestellte Information durch einen Bitvektor, dessen Bits die Präsenz eines Kubi angeben, ersetzt. Zwar muß man so für jeden elementaren Kubi ein Bit einführen, d.h. Kubi auf höherem Niveau, Blätter in der Octreestruktur, müssen in ihre elementaren Kubi zerlegt werden damit die entsprechenden Bits im Bitvektor bestimmen werden können.

Formell betrachtet heißt das, daß dieser Bitvektor die Repräsentation eines Subkubus der Kantenlänge 2^b ist, der 2^{3b} elementare Kubi (Kubi mit Niveau $n = 0$) enthält. Der Bitvektor ersetzt also die Baumstruktur in den Niveaus $n < b$, b wird im weiteren Verlauf der Arbeit auch als BITBLOCK-NIVEAU bezeichnet.

Niveau	Anzahl der Knoten
10	1
9	8
8	64
7	512
6	4.096
5	32.768
4	262.144
3	2.097.152
2	16.777.216
1	134.217.728
0	1.073.741.824

Tabelle 2.4: Anzahl der Knoten auf den einzelnen Niveaus

Für die weitere Handhabung muß nun überlegt werden, ab welchem Niveau sich die Einführung der Bitvektoren anstelle der Baumstruktur lohnt. Betrachtet man die beiden Extrema, das ganze Modell durch einen Bitvektor darzustellen oder nur das unterste Niveau, so ist es ersichtlich, daß man hier keine Vorteile erreichen wird. Führt man nun die Analyse auf diesem Intervall weiter, so ergibt sich, daß das Minimum für den Speicherbedarf auf den unteren Niveaus durch den Einsatz von Bitvektoren zu erreichen ist (siehe [1], S. 123), so daß ab Niveau $n = 2$ jetzt Bitvektoren verwendet werden.

Betrachtet man unter diesem Gesichtspunkt Tabelle 2.4, so läßt sich die Einsparung auch anhand von Zahlen gut erkennen. Im worst case[10] müssen von Niveau 1 nach 0 1.073.741.824 Zeiger für die elementaren Kubi angelegt werden und auf Niveau 2 sind 134.217.728 Zeiger zu den Vaterknoten vorhanden. Das sind insgesamt 1.207.959.552 Zeiger um Information von 1.073.741.824 elementaren Kubi zu speichern, die gleiche Information beinhalten auch 16.777.216 Bitvektoren — direkt an die Knoten auf Niveau 2 angehängt — mit je 64 Bit Länge.

Zugegebenermaßen ist der worst case ein sehr unrealistischer Fall, der nur die maximale Einsparung verdeutlichen soll; Modelle die bei der Speicherausnutzung einen average case ergeben wurden in der vorhergehend erwähnten Analyse (siehe [1], S. 123) betrachtet. Die Einsparung wird aber auch so deutlich, denn ein Bitvektor von 64 Bit wird nur dann angelegt, wenn ein Knoten auf Niveau $n = 2$ auf SPLIT gesetzt wird und somit im Octree mindestens 8 neue Zeiger zu 8 Knoten erzeugt würden.

[10] Der worst case im Falle der Octreestruktur ist ein Modell, bei dem auf unterstem Niveau nur jeder zweite Kubus gesetzt ist; eine poröse Struktur ähnlich einem Schwamm.

Durch die gewählte Speicherwortgröße von 16 Bit wird der durch das Bitblockniveau $b = 2$ resultierende Bitvektor von 64 Bit auf vier Speicherstellen aufgeteilt. Mit 64 Bit spiegelt der Bitvektor einen Kubus mit der Kantenlänge 4 wieder, also kann die Zerlegung des Bitvektors so erfolgen, daß in einem Speicherwort eine Ebene von 4x4 Kubi abgelegt wird. Die so erhaltene Zerlegung des Bitvektors in vier Speicherworte entspricht auf geometrischer Ebene einer Zerlegung eines 4x4x4 Würfels in vier 4x4 Ebenen. Durch diese Äquivalenz zu dem Würfel wird der Bitvektor des weiteren als BITBLOCK bezeichnet. Diese Zerlegung wird nun so gewählt, daß sie längs der X–Achse verläuft und jedes Speicherwort eine YZ–Ebene wiederspiegelt. Abbildung 2.4 zeigt die Zerlegung eines Kubus und verdeutlicht wie die Aufteilung auf die vier Bitvektoren/Speicherstellen zu verstehen ist.

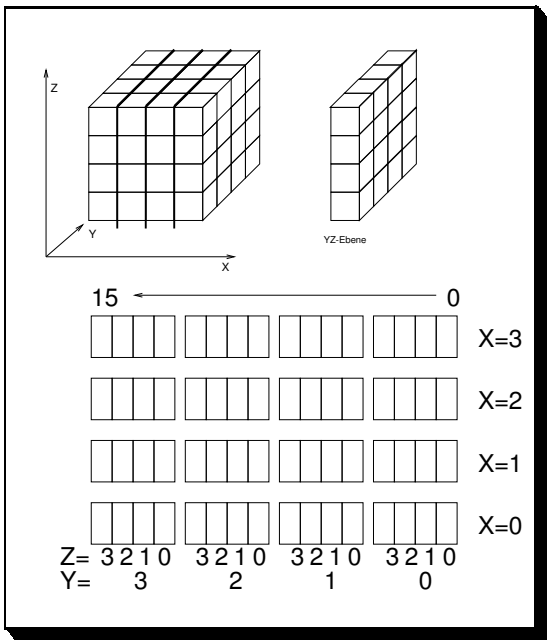


Abbildung 2.4: Aufbau eines Bitblocks

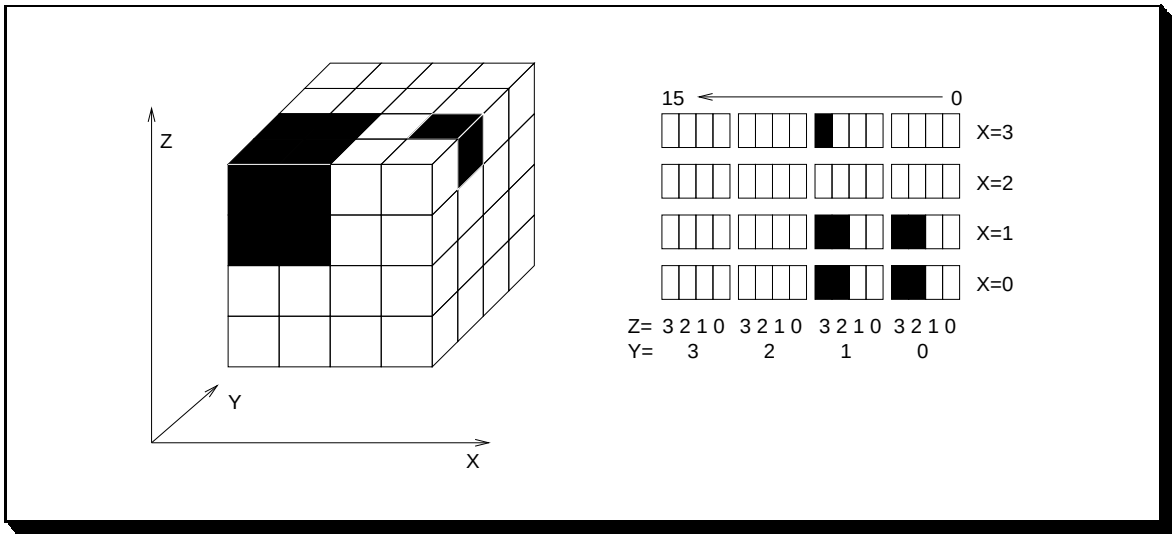


Abbildung 2.5: Beispiel eines Kubi auf Niveau $n = 2$ und seine Darstellung in einem Bitblock

2.1.4.1 Eigenschaften von Bitblöcken

Das genaue Auffinden eines Kubi bezüglich seiner Position oder über seinen Index ist in der Baumstruktur eindeutig definiert (vgl. 2.1.1), diese Eindeutigkeit muß auch für die Darstellung durch einen Bitblock gegeben sein. Dadurch, daß der Bitblock Kubi auf Niveau $n = 1$ durch seine elementaren Kubi repräsentiert, müssen für Kubi diesen Niveaus im Bitblock immer 8 Bit betrachtet werden. Auf Niveau $n = 0$, wo elementare Kubi betrachtet werden, kann direkt auf das entsprechende Bit Bezug genommen werden. Diese Positionszuordnung läßt sich durch folgende Formeln ausdrücken:

für Niveau n=1:

Bestimmung der X–Koodinate: $x = p_{1X} \cdots p_{1X} + 1$

entsprechende Werte für Y und Z durch: $y = p_{1Y} \cdots p_{1Y} + 1$

$z = p_{1Z} \cdots p_{1Z} + 1$

für Niveau n=0:

Bestimmung der X–Koordinaten: $x = 2 \cdot p_{1X} + p_{0X}$

entsprechende Werte für Y und Z durch: $y = 2 \cdot p_{1Y} + p_{0Y}$

$z = 2 \cdot p_{1Z} + p_{0Z}$

unter x, y und z sind die Indizes zu verstehen, die das Element im Bitblock identifizieren p_{1X}, p_{1Y} und p_{1Z} geben den Anteile an der octalen Position für Niveau $n = 1$ an; p_{0X}, p_{0Y} und p_{0Z} entsprechend für Niveau $n = 0$.

übertragen auf den Bitblock und dessen Darstellung bedeutet dies:

Anwahl der YZ–Ebene/des Speicherworts durch: $X = 2 \cdot p_{1X} + p_{0X}$

und der entsprechenden Bits durch: $p_{vec} = 8 \cdot p_{1Y} + 4 \cdot p_{0Y} + 2 \cdot p_{1Z} + p_{0Z}$

Abbildung 2.6 zeigt die Positionszuordnung der Subkubi eines Kubus (vgl. auch 2.1) auf Bitblockniveau $b = 2$; die eingekreisten Positionsbezeichnungen sind den Kubi auf Niveau $n = 1$ zugeordnet, die anderen Niveau $n = 0$. Die entsprechende Zuordnung dieser Positionen auf den Bitblock, welche durch die zuvor angegebenen Formeln ausgedrückt wird, wird durch Abbildung 2.7 anschaulich verdeutlicht. Diese etwas andere Art, in Bezug auf die sonstige Octreestruktur, des Zugriffs auf einen Kubus muß bei der Realisierung einer Zugriffsfunktion[11] auf die Kubusinhalte berücksichtigt werden. Auf Niveau $n = 1$ müssen 8 Bits zur Interpretation des Inhaltes beachtet werden und nötigenfalls auch gesetzt bzw. gelöscht werden. Bei dem Zugriff auf elementare Kubi (Niveau $n = 0$) muß die vorherige Position (von Niveau $n = 1$) mit beachtet werden um das entsprechende Bit im Bitblock zu bestimmen.

Abbildung 2.5 zeigt anhand eines Beispielkubis das Aussehen des entsprechenden Bitblocks.

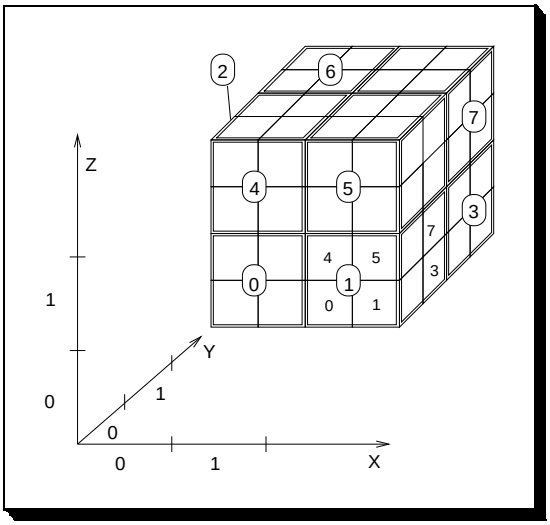


Abbildung 2.6: Positionen eines Kubi

[11] Die Zugriffsfunktion bezieht sich immer auf den aktuellen Kubus und liefert seine Zustand (SPLIT, FULL oder EMPTY) bzw. ermöglicht das setzen auf FULL oder EMPTY.

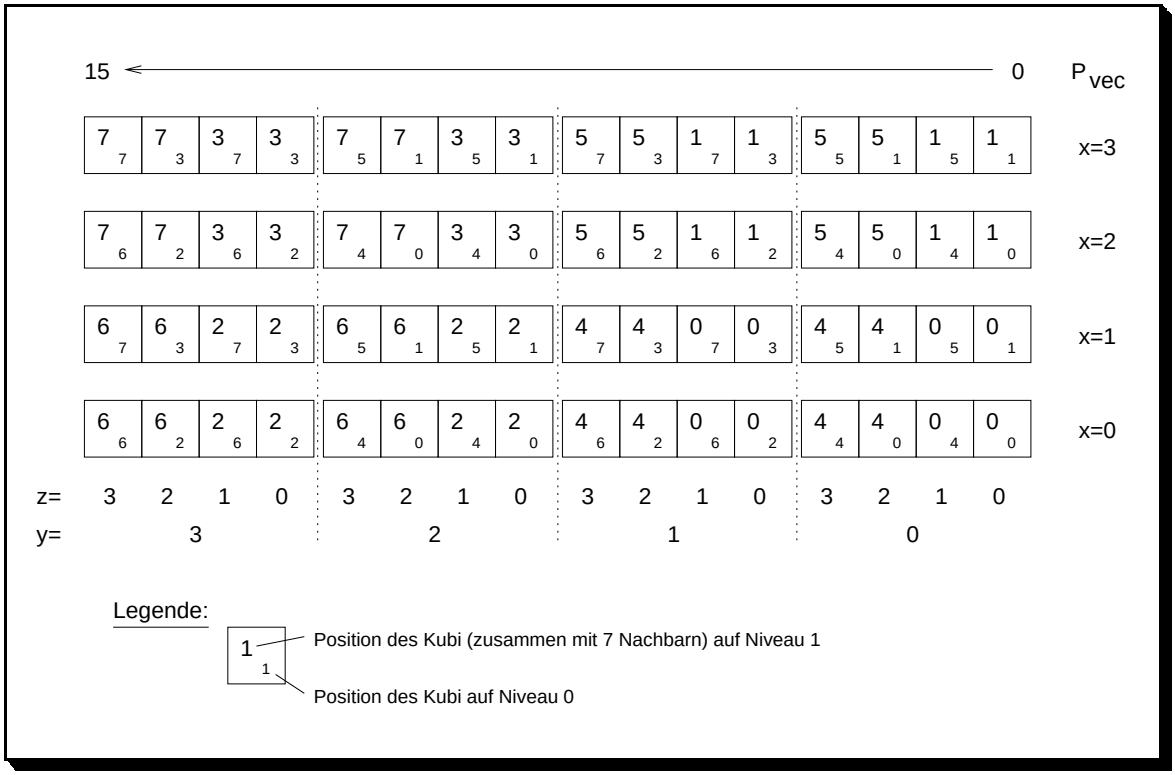


Abbildung 2.7: Positionen in den Bitvektoren eines Bitblocks

2.1.5 Octrees im Speicher

Betrachten wir nun, wie sich Octrees in einem linear adressierbaren RAM ablegen lassen. Wir folgen dabei den Vorschlägen aus [1] und legen einen 32 KB großen, in 16 Bit große Worte unterteilten Speicher zugrunde. Den Speicher teilen wir in $2^8 = 256$ Kacheln á $2^{16} = 65.536$ Worte ein. Wir speichern Kubusbäume stets in fortlaufenden Speicherzellen und an einem Kachelanfang beginnend.

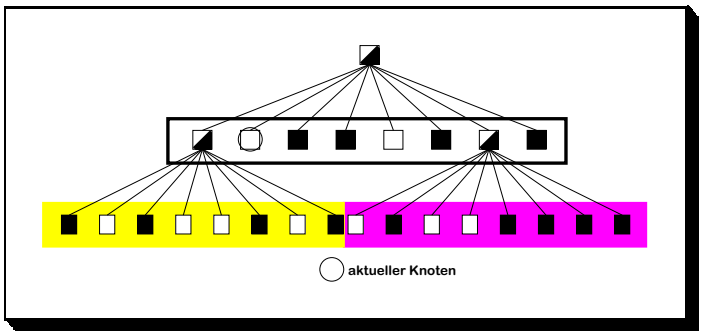


Abbildung 2.8: Beispieloctree

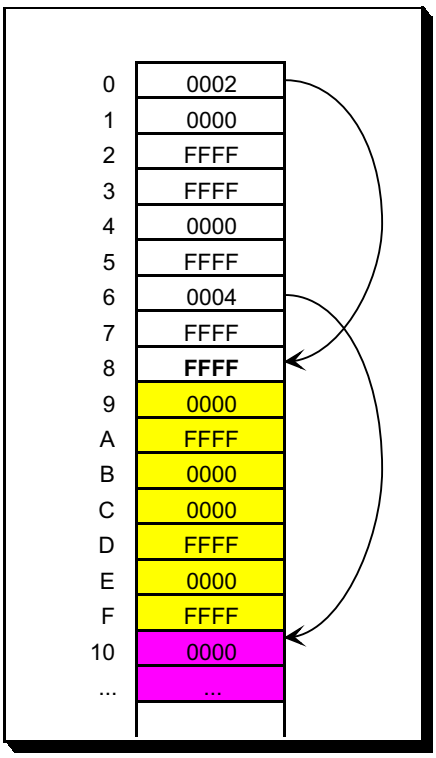


Abbildung 2.9: Speicherauszug

Wir reservieren für jeden geteilten Knoten — von den Bitblöcken, für die vier Speicherworte reserviert werden, abgesehen — acht Speicherworte, entsprechend den acht Söhnen des Knotens. Für einen vollen Subkubus beschreiben wir die zugehörige Speicherstelle mit `FFFF`, für einen leeren mit `0000`. Alle anderen Speicherinhalte werden als geteilte Kubi interpretiert, wobei der Inhalt des Speichers die *relative* Adresse des Speicherblocks enthält, in dem die 8 Söhne — bzw. der adjungierte Bitblock — des aktuellen Kubus zu finden sind. Relative Adressen können ausnahmslos als positive Adressen interpretiert werden, da Kubi eines bestimmten Niveaus erst dann erzeugt werden, wenn die übergeordneten Kubi — also diejenigen, die diesen Kubus enthalten — bereits erzeugt wurden.

Beachten wir nun, daß es ausreicht, die Anfangsadresse des aktuellen Kubusblocks zu speichern — bei dem in Abb. 2.8 markierten Knoten wäre dies die Adresse 0 — und die Adresse des aktuellen Kubus durch Addition der Anfangsadresse des Kubusblocks mit der aktuellen Position zu berechnen, so zeigt sich, daß die beiden niederwertigen Bits einer relativen Adresse stets Null sind; die Anfangsadressen der Blöcke sind stets Vielfache von Vier. Die relativen Adressen lassen sich so gemäß

$$(\text{Anfangsadresse Zielblock} - \text{Anfangsadresse akt. Block}) \ / \ 4$$

berechnen. Die Adressen für die geteilten Knoten im obigem Beispiel ergeben sich also aus $(8_{\text{hex}} - 0_{\text{hex}})/4 = 2_{\text{hex}}$ und $(10_{\text{hex}} - 0_{\text{hex}})/4 = 4_{\text{hex}}$.

2.1.5.1 Indirekte Adressen

Mit den gespeicherten 16 Bits wären Sprünge über eine Distanz von $4 * 2^{16} = 262.140 = 3\text{FFFC}_{\text{hex}}$ möglich. Bei größeren Modellen — wenn z.B. viele Kubi auf niedrigstem Niveau gesetzt werden — reicht dies jedoch unter Umständen nicht aus, besonders zwischen den Kubi auf den hohen Niveaus liegen dann sehr viele Daten. Um diesen Umständen gerecht zu werden, unterscheiden wir zwischen direkter und indirekter Adressierung: Ist das signifikanteste Bit (Bit 15) Null, so wird die Adresse als direkte Verzweigung interpretiert, andernfalls als indirekte. Direkte Adressen lassen sich so bis zu einer Distanz von $4 * 2^{15} = 131.070 = 1\text{FFFE}_{\text{hex}}$ einsetzen.

Für jeden Kubusbaum reservieren wir im Speicher eine Kachel für indirekte Adressen und adressieren mit den verbleibenden 15 Bits der indirekten Adresse ein Paar von Speicherzellen in dieser Kachel, in denen die relative Adresse des gesuchten Subkubus zu finden ist.

2.2 Operationen im 3D-Raum

Die Möglichkeit, dreidimensionale Modelle im Speicher abzulegen, reicht für die meisten Anwendungsfälle kaum aus. Im Regelfall möchte der Anwender diese Modelle in irgendeiner Weise manipulieren, sei es, daß ein Modell transformiert werden soll oder zwei Modelle zu einem dritten Modell zusammengefaßt werden sollen. Letzters läßt sich z.B. nutzen, um aus einfachen, geometrischen Objekten komplexere Objektensembles zu gestalten. Es folgt eine kurze Übersicht einiger Operationen, die sich auch auf octreebasierte Modelle anwenden lassen.

2.2.1 Geometrische Operationen

2.2.1.1 Transformationen

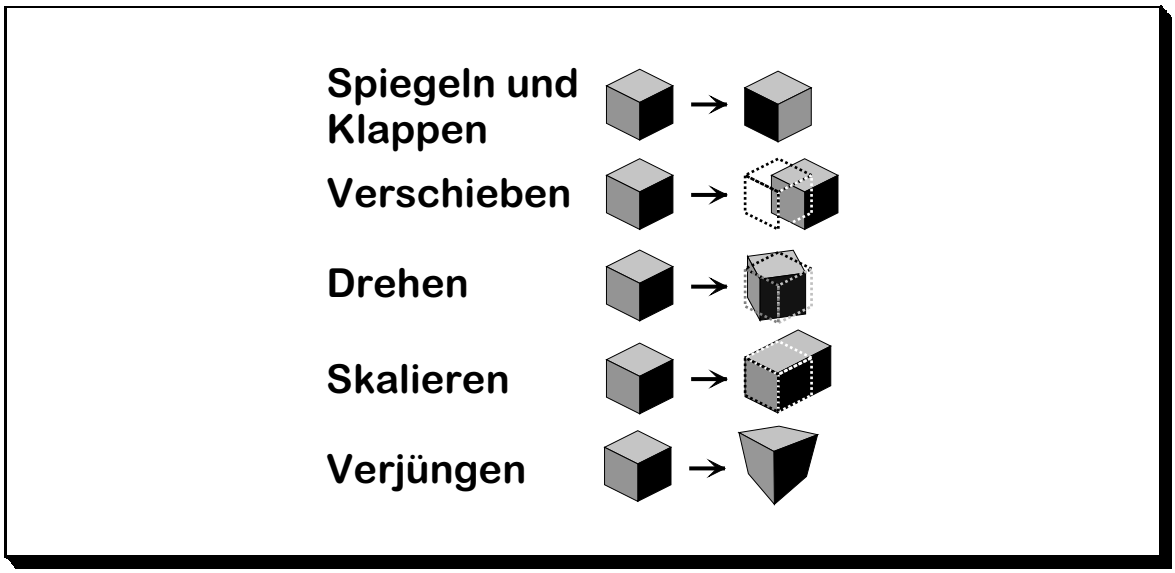


Abbildung 2.10: Transformationen

Transformationen stellen die erste Gruppe von Operationen dar. Sie sind stets Operationen vom Typ $tree' = f(tree)$, d.h. ein Octree wird auf einen anderen Octree abgebildet. Diese Operationen dürften aus einschlägiger CAD–Software bekannt sein und daher keiner weiteren Erläuterung bedürfen. Abbildung 2.10 enthält eine Auflistung der Operationen dieser Gruppe mit Andeutung ihrer Funktionsweise.

2.2.1.2 Mengenoperationen

Die zweite Gruppe der Operationen sind Mengenoperationen. Hier finden wir Operationen des Typs $tree' = tree_1 \diamond tree_2$ — im Fall Vereinigung, Differenz und Schnitt — also eine Verknüpfung zweier Octrees zu einem dritten, des Typs $tree' = f(tree)$ — im Falle der Komplementbildung — und — im Falle der Inklusion und Disjunktion — des Typs $bool = tree_1 \diamond tree_2$ mit $bool \in \{wahr, falsch\}$. Mengenoperationen sind in 2.11 aufgelistet.

2.2.1.3 Morphologische Transformationen

Von der dritten Gruppe von Operationen — den sog. morphologischen Transformationen — sei hier nur die Hüllenbildung als wichtigster Vertreter dieser Klasse von Operationen erwähnt: Bei der Hüllenbildung wird die Oberfläche eines Kubusmodells durch Kubi von kleinstem Niveau umrissen; innerhalb liegende Kubi werden durch leere Kubi ersetzt. Abbildung 2.12 deutet diese Operation in einer Schnittdarstellung an.

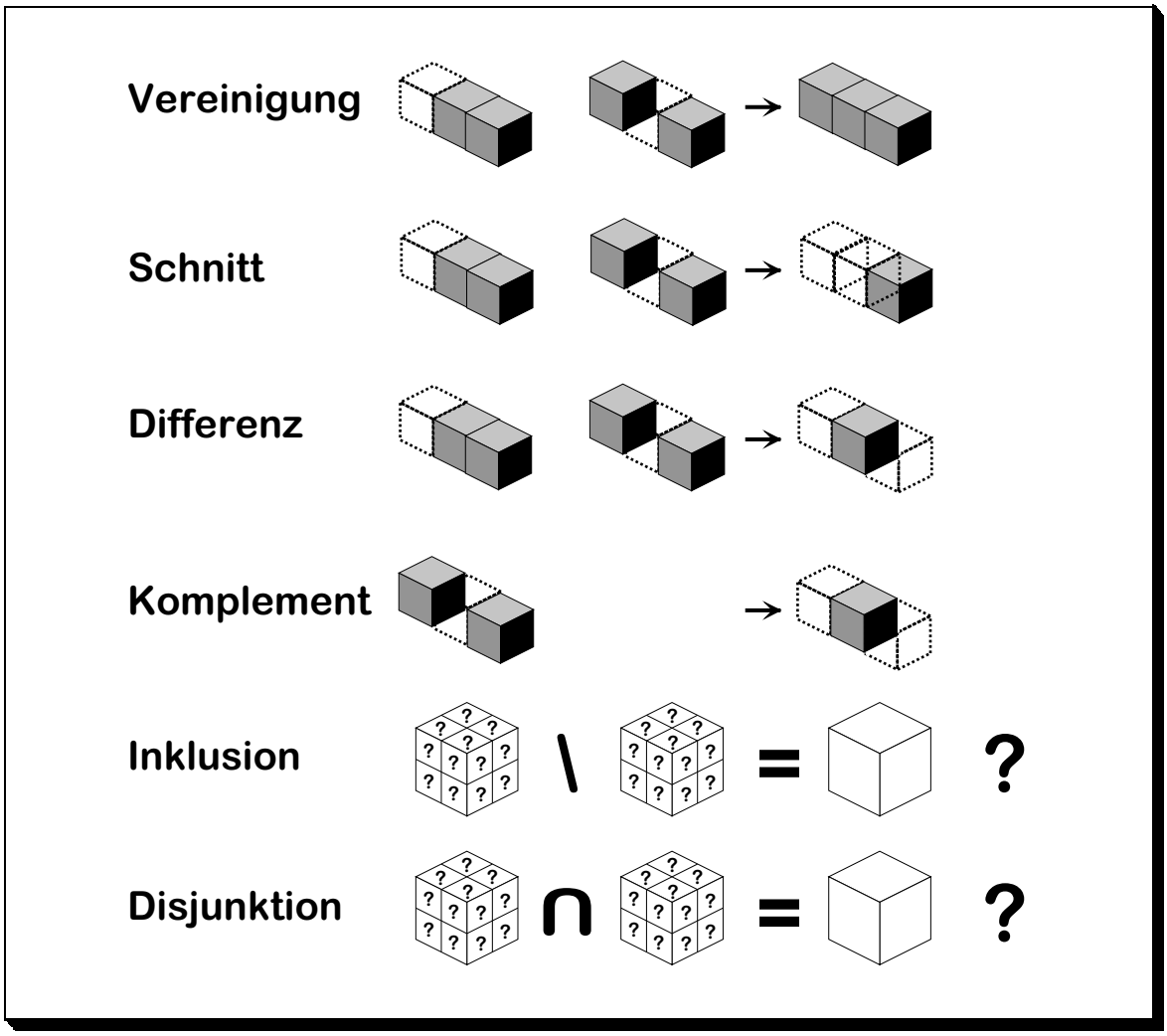


Abbildung 2.11: Mengenoperationen

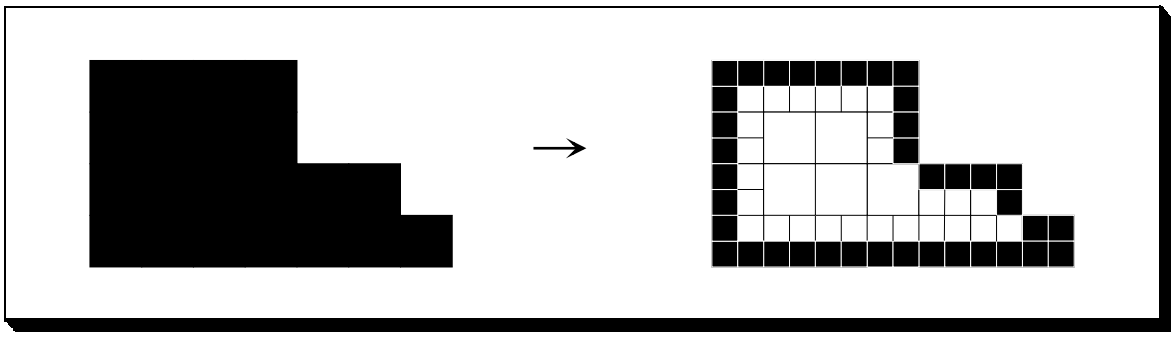


Abbildung 2.12: Hüllenbildung

2.2.1.4 Geometrische Größen

Die letzte Gruppe von Operationen auf Octrees sind vom Typ $n = f(tree)$ bzw. $\vec{v} = f(tree)$ mit[12] $n \in \mathcal{Z}, \vec{v} \in \mathcal{Z}^3$. Zu diesen Operationen gehören die Berechnung von Tiefenkoordinaten[13], Volumen, Oberfläche, Schwerpunkt und Trägheitsmoment.

2.2.2 Affine Abbildungen

In der euklidischen Geometrie lassen sich oben genannte Transformationen durch affine Abbildungen realisieren. Dabei wird jeder Punkt $\vec{p}$ des Ausgangsraumes in einen Punkt $\vec{p}\,'$ des Zielraumes abgebildet:

$$\vec{p}\,' = \mathrm{A} \cdot \vec{p} + \vec{t}$$

Durch geeignete Wahl der Transformationsmatrix A und des Translationsvektors $\vec{t}$ lassen sich folgende Transformationen durchführen:

Verschiebung:

$$\mathrm{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \vec{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

Skalierung:

$$\mathrm{A} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_z & 0 \\ 0 & 0 & s_y \end{pmatrix}, \vec{t} = \vec{0}$$

Rotation:

um X-Achse:

$$\mathrm{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{pmatrix}$$

$\vec{t} = \vec{0}$

um Y-Achse:

$$\mathrm{A} = \begin{pmatrix} \cos\varphi & 0 & \sin\varphi \\ 0 & 1 & 0 \\ -\sin\varphi & 0 & \cos\varphi \end{pmatrix}$$

$\vec{t} = \vec{0}$

um Z-Achse:

$$\mathrm{A} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$\vec{t} = \vec{0}$

Dabei bezeichnen t_w die Verschiebung, s_x die Skalierung entlang der Achse w und φ den Rotationswinkel.

Auch die Transformationen „Spiegeln“ und „Klappen“ lassen sich mittels affiner Abbildungen durchführen. Für eine vollständige Liste sei hier auf [1], S. 49 verwiesen[14].

Die Transformationen auf Octrees basieren auf den affinen Abbildungen dahingehend, daß ein Kubus des Ausgangsbaums mittels einer dieser Abbildungen auf einen Kubus im Zielbaum abgebildet wird. Es reicht jedoch nicht aus, den berechneten Zielkubus mit „voll“ zu markieren; es ist vielmehr zu berücksichtigen, daß der Zielkubus im Regelfall nicht mehr genau im Raster liegt und aus kleineren Kubi zusammengesetzt werden muß. Als Beispiel mag nebenstehende Abbildung dienen: dort wurde ein Kubus mit Kantenlänge 256 um 384 Einheiten in X-Richtung verschoben. Der abgebildete Kubus besteht nun aus 4 Kubi der Kantenlänge 128.

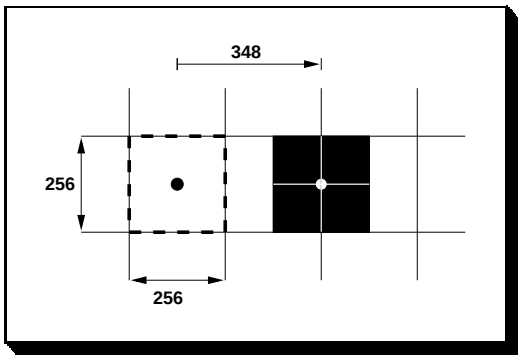


Abbildung 2.13: Verschiebung eines Kubus

[12] $\mathcal{Z}$ bezeichne die Menge der ganzen Zahlen. In der Tat ist dies eine Verallgemeinerung, da einige der erwähnten Operationen keine negativen Zahlen liefern.

[13] kleinster Abstand zw. einem vorgegebenem Punkt auf der Oberfläche des Kubus von maximalem Niveau und den Punkten auf der Oberfläche des gespeicherten Modells

[14] Allerdings sind dort die Matrizen für Klappen um $\frac{\pi}{2}$ um Z-Achse und $-\frac{\pi}{2}$ um die Y-Achse fehlerhaft; man tausche hier die Vorzeichen der nicht auf der Hauptdiagonalen liegenden Einsen!

2.2.3 Operationen auf Bitblöcken

Im Prinzip können alle zuvor beschriebenen Operationen in ihren Algorithmen unverändert bleiben, wenn man ab dem Bitblockniveau $n = 2$ die entsprechenden Details bei der Realisierung der Zugriffsfunktion berücksichtigt. Im Rahmen einer solchen Änderung würde man aber nur die Vorteile der Bitblöcke in Bezug auf den Speicherplatz nutzen; wir wollen aber auch einige Verbesserungen durch angepaßte Operationen erzielen.

Neben der Einsparung von Speicherplatz erreicht man eine effizientere Handhabung von Mengenoperationen und strukturkonformen Transformationen[15]. Um diese Vorteile nutzen zu können muß man einen kleinen Mehraufwand im Bereich der Verwaltung (Überprüfung des Niveaus und nötigenfalls Ausführen der Bitblockoperation anstelle der normalen Operationen) in Kauf nehmen. Andere Transformationen wie freies Drehen oder Skalieren bleiben ganz auf die Transformationsmatrix und die Mittelpunktsvektoren bzw. den Indexvektoren beschränkt und erforden keinerlei Modifikationen in Bezug auf die Bitblöcke, da hierfür die Anpaßung der elementaren Operationen zum Setzen bzw. Abfragen des Zustandes bezüglich der Bitblöcke ausreicht.

Die Mengenoperationen werden auf einfache logische Verknüpfungen, die sich mit den Bitblöcken aufgrund ihres an das geometrische Modell angelehnten Aufbaus gut realisieren lassen zurückgeführt. So ergibt sich, daß die Bildung der Schnittmenge auf die logische Verknüpfung UND zurückgeführt wird bzw. die Vereinigung auf das logische ODER. Ohne die Realisierung mit Bitblöcken müßten hier im worst–case 64 elementare Kubi mit weiteren 64 elementaren Kubi verglichen werden, bei Bitblöcken sind hier vier logische Verknüpfungen aufgrund der Verteilung über vier Speicherworte nötig.

Transformationen wie Spiegeln oder Klappen werden durch Schiebeoperationen mit den Bitblöcken realisiert. Zur Erläuterung sei hier recht plastisch die Vorgehensweise dargelegt: Man schneidet einen Würfel in Scheiben und fügt diese Scheiben in umgekehrter Reihenfolge wieder zusammen, als Ergebnis erhält man einen neuen Würfel, der ein Spiegelbild des Ausgangswürfels bezüglich der Schnittebene ist. Fügt man nun diese Scheiben nicht in umgekehrter Reihenfolge auf der gleichen Achse[16] zusammen, sondern längs einer anderen Achse, so ergibt sich eine Rotation um 90 Grad — auch als Klappen bezeichnet — bezüglich der dritten Raumachse. Die Transformationsvielfalt läßt sich noch dahingehend erweitern, daß jede Ebene, die von dem Ausgangskubus in den Zielkubus übertragen wird, bezüglich ihres Mittelpunktes gespiegelt bzw. um 180 Grad rotiert wird, was eine Punktsymmetrie zur Folge hat. Es ist ersichtlich, daß sich diese Vorgehensweise gut auf Bitblöcke anwenden läßt, und daß man so effizient diese Transformationen realisieren kann.

Zur Realisierung der zuvor beschriebenen strukturkonformen Transformationen müssen Operationen realisiert werden, die ein Schieben des Bitblocks bezüglich der drei Raumachsen, in Anlehnung an das räumliche Abbild, im Speicher erlauben. Diese Operationen müssen eine ähnliche Funktionalität aufweisen wie man sie von Schiebeoperationen auf Bitfolgen kennt, d.h. es muß einerseits ein Schieben möglich sein, um etwas in den Bitblock herein und auf der anderen Seite heraus zu schieben, und es muß ein Rollen durchführbar sein, um eine Ebenen aus einem Biblock heraus und diese gleichzeitig am anderen Ende wieder herein zu schieben. Nachfolgend wird kurz beschrieben was bei einer Realisierung der Rollenoperation[17] zu beachten ist.

Das Rollen in Richtung einer Achse, die entgegengesetzte Richtung erfolgt analog, gestaltet sich in der Auswahl (vgl. Abbildung 2.14) der zu rollenden Elemente unterschiedlich komplex:

X–Achse: aufgrund der Äquivalenz der vier Speicherwörter zu den YZ–Ebenen kann hier ein einfacher Tausch dieser durchgeführt werden.

Y–Achse: hier müssen immer vier zusammenhängende Bits aus einem Speicherwort heraus und am anderen Ende herein geschoben werden. Dieser Vorgang muß mit allen vier Speicherworten

[15] Transformationen, bei denen der Ausgangskubus nur an anderer Position wieder in das Kubusraster gesetzt wird (Transformationen wie Spiegeln oder Klappen); im Gegensatz dazu stehen Transformationen, bei denen der Ausgangskubs in seine Subkubi zerlegt werden muß und so wieder neu in das Raster integriert wird

[16] diese Achse bildet den Normalenvektor bezüglich der Schnittebene

[17] die Schiebeoperation läßt sich einfach hieraus ableiten, da hier nur der Übertrag des Rollens durch das hereinzuschiebende Element ersetzt werden muß

erfolgen. Damit man eine komplette XZ–Ebene erhält muß man die vier Bits der einzelnen Speicherworte zu einem kompletten 16 Bit Wort zusammenhängen.

Z–Achse: für diesen Fall werden vier einzelne Bits aus einem Speicherwort ausgewählt, dieses dann um eine Stelle verschoben, und die Bits an entsprechender Stelle wieder eingefügt. Der Vorgang muß mit allen vier Speicherworten durchgeführt werden und eine Konkatenation der einzelnen Bits jeden Speicherworts entspricht der XY–Ebene.

Das oben erwähnte Spiegeln bzw. Rotieren um 180 Grad der zu übertragenden Ebene wollen wir an dieser Stelle genauer betrachten, da es bei der Realisierung einige Mißverständnisse bezüglich der Beschreibung in [1] gab. In dortiger Beschreibung wird die Transformation als „diagonales Spiegeln“ bezeichnet, was zu einigen Irritationen führte. Bei der ersten Realisierung in C wurde versucht die Transformation durch Spiegeln an einer der Diagonalen zu realisiern. Erst eine Spiegelung an Haupt– und Nebendiagonalen lieferte das gewünschte Ergebnis. Bei näherer Betrachtung erwiesen sich diese Spiegelungen, die eine Punktsymmetrie zum Mittelpunkt der übertragenden Ebene darstellen, zu einer Umkehrung der Reihenfolge der Bits, die dem Bitvektor der Ebene entsprechen, äquivalent.

Formell läßt sich die „diagonale Spiegelung“ eines Bitvektor a in einen Bitvektor b so ausdrücken:

$$\forall i \in \{0 \ldots 15\} : b_i := a_{15-i}$$

Trotz der unterschiedlichen Komplexität der Rollen–Vorgänge und des zusätzlichen Aufwandes im Falle einer Rotation der Bitfolge lassen sich so Transformationen einfach durchführen, im Vergleich dazu muß ohne Bitblöcke der Positionsvektor zu jedem Kubi mit einer Transformationmatrix multipliziert werden, was im worst–case zu 64 Vektormultiplikationen für einen Kubus auf Niveau $n = 2$ führt.

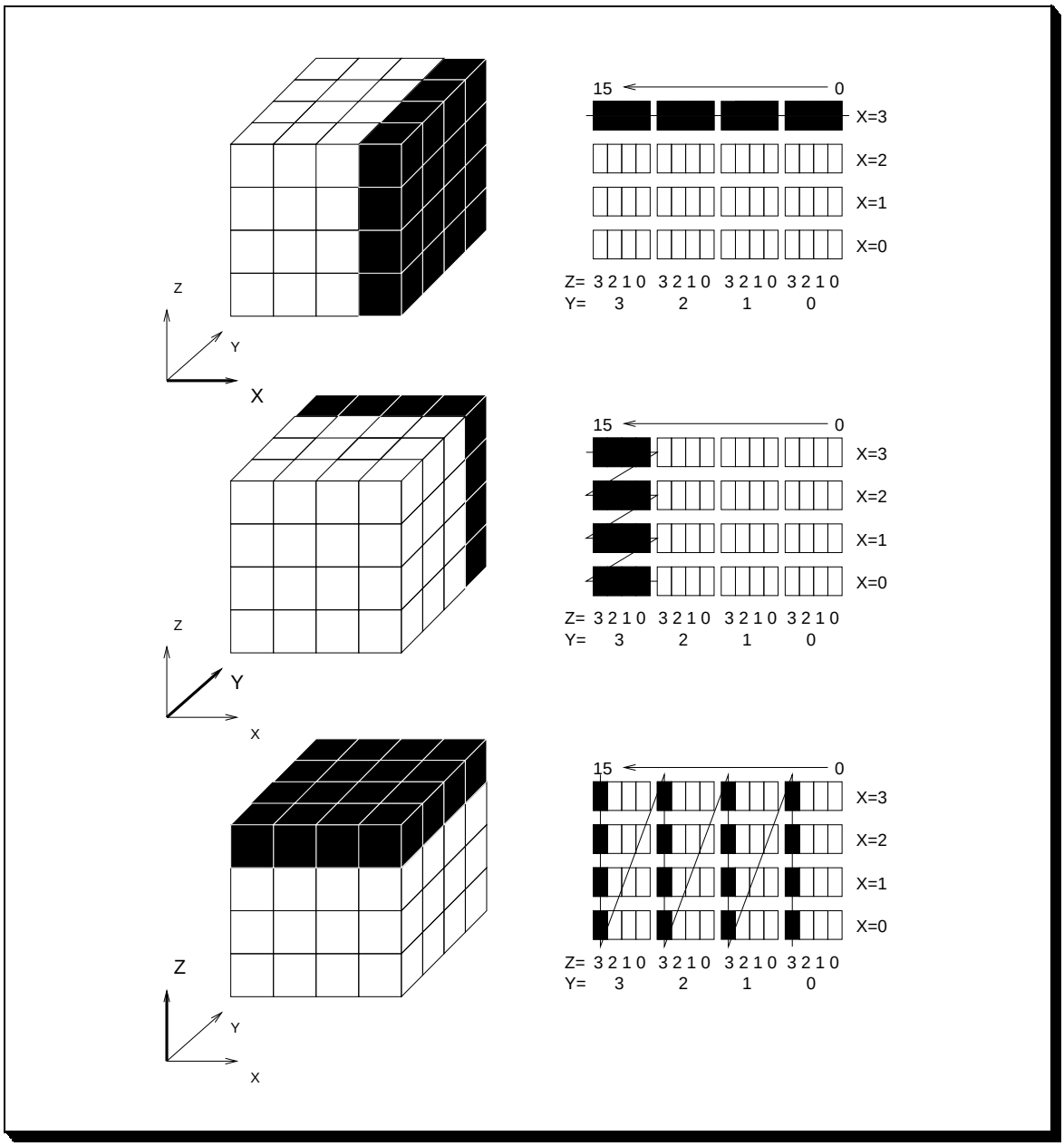


Abbildung 2.14: Auswahl der adäquaten Bits bei Schiebe– bzw. Rollenoperationen entsprechend der positiven Richtung der jeweiligen Achse

Kapitel 3

Implementierung in C

Der erste Schritt auf dem Weg zum Chip zur Unterstützung octreebasierter Operationen war eine Softwarerealisierung der in [1] vorgeschlagenen Algorithmen. Die Implementierung, zu der die Sprache C bevorzugt wurde, soll in diesem Kapitel der Abhandlung vorgestellt werden.

Eine genaue Darlegung aller implementierten Operationen würde sicherlich den Rahmen dieser Arbeit sprengen. Daher werden wir nur die Datenstrukturen und Routinen auf unterer Ebene — diese bilden die Grundlage für die Hardwareentwicklung — eingehend beschreiben und beschränken uns bei der Darstellung der Rountinen auf höherer Ebene auf eine Beschreibung der Methodik sowie Ergänzungen und Verbesserungen gegenüber [1].

Der erste Abschnitt dieses Kapitels dient der Beschreibung der Architektur des Gesamtsystems und der verwendeten Entwicklungsumgebung. In den folgenden Abschnitten werden die verwendeten Datenstrukturen sowie die Operationen auf unterer Ebene und auf höherer Ebene vorgestellt. Einige Betrachtungen zur Performance der Softwarelösung beschließen dieses Kapitel der Arbeit.

3.1 Systemarchitektur

Abbildung 3.1 zeigt die einzelnen Module der C–Implementierung. Man kann drei Blöcke unterscheiden:

Der erste Block wird durch das Modul octree.c repräsentiert und umfasst die verwendeten Datenstrukturen und die Operationen, die direkt auf den Datenstrukturen arbeiten. Hierzu gehören die in 2.1.2 vorgestellten Navigations– und Baumoperationen sowie die in 2.1.4 vorgestellten Bitblockoperationen. Diese Operationen bezeichnen wir als *Low–Level–Operationen*. Dabei sind die Operationen set_node, parse_tree usw.[1] nicht eindeutig zuzuordnen: sie werden zwar von den Operationen des zweiten Blocks benutzt — und wurden deshalb im Modul octree.c angeordnet —, greifen aber nicht direkt auf die Datenstruktur zu und gehören auch nicht zu den von der Hardwarelösung zur Verfügung gestellen Operationen.

In octree.c werden abhängig von den Compileroptionen (→ 3.1.2) folgende Dateien eingelesen[2]:

bitblock.c	Bitblockoperationen
oct_sim.c	Lesen und Schreiben von Stimuli bei Low–Level–Operationen
oct_hard.c	Benutzung der Hardwarelösung (noch nicht implementiert)

Die Datenstrukturen und Operationen dieses Blocks kann man als abstrakten Datentyp „Octree“ auffassen.

[1] in Abb. 3.1 in gestricheltem Rahmen

[2] Dabei handelt es sich nicht um Module im Sinne softwaretechnologischer Modularisierung, also nicht um getrennt kompilierbare Einheiten.

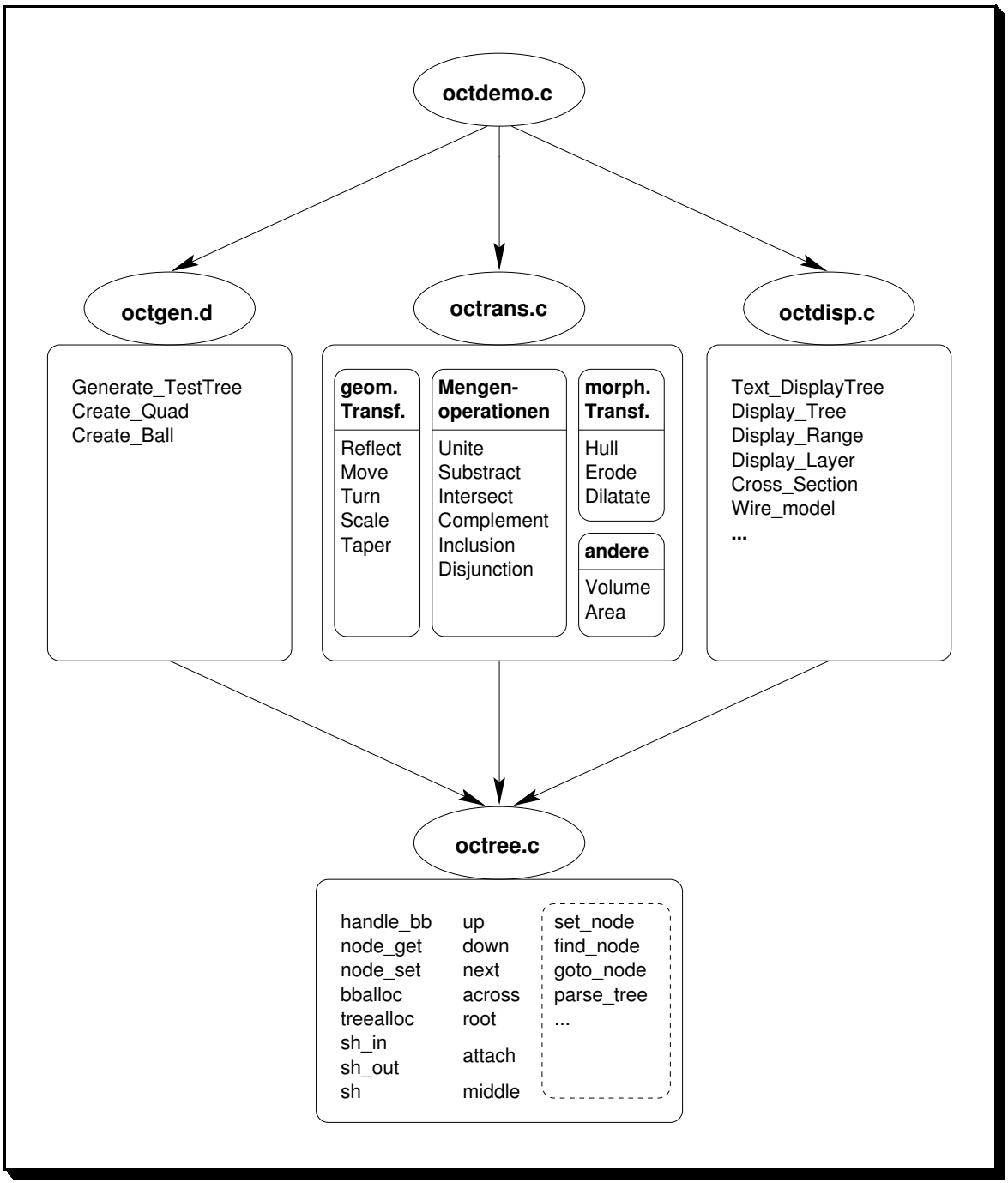


Abbildung 3.1: Module der Softwarelösung

Zum zweiten Block gehören die in den Modulen octgen.c, octrans.c und octdisp.c implementierten Transformations–, Verknüpfungs–, Generierungs– und Visualisierungsoperationen für Octrees. Diese arbeiten nur über die Low–Level–Operationen auf den Datenstrukturen. Wir nennen diese Operationen *High–Level–Operationen.*

Der dritte Block besteht aus dem Modul octdemo.c, in dem die Routinen des zweiten Blocks zu Testläufen — bestehend aus Generierung, Transformation und Visualisierung von Kubusbäumen — benutzt werden. Weiterhin besteht dieses Modul aus einer rudimentären Benutzeroberfläche zur Auswahl der Operationen.

3.1.1 Entwicklungsumgebung

Das System wurde auf einem PC (386er) unter DOS mit der Entwicklungsumgebung Borland–C 3.1 implementiert und an GNU–C unter Unix/Linux angepaßt. Die erste Uniximplementierung benutzte die Graphiklibrary „Visual“, die uns vom Lehrstuhl VII freundlicherweise zur Verfügung gestellt wurde; die aktuelle Version benutzt die am Lehrstuhl Praktische Informatik VI der Fernuniversität Hagen entwickelte XGraphik–Bibliothek ([44]), die im Gegensatz zu Visual Pulldown–Menüs zur Verfügung stellt[3].

Die im Kapitel 3.5 durchgeführten Performancemessungen wurden mit dem Borland Turbo–Profiler auf einem Rechner mit 133 MHz–Pentium vorgenommen.

3.1.2 Versionen des Programms

Durch die unterschiedlichen Realisierungen der Datenstruktur „Octree“ (→ 3.2.1), die Implementierung für verschiedene Plattformen sowie die Kopplung mit der Simulation der Hardware (→ 5.3) lassen sich unterschiedliche Versionen des Programms erzeugen. Hierbei wurde die Möglichkeit zur bedingten Compilierung des C–Präprozessors ausgenutzt: bestimmte Bereiche des Quelltextes sind durch `#ifdef` <*option*> `...` `#endif` eingerahmt und werden nur kompliert, wenn die entsprechende Compileroption gesetzt wurde[4]. Folgende Compileroptionen werden verwendet:

Option	Programmversion
zachse	Es werden dreidimensionale Algorithmen und Datenstrukturen verwendet, andernfalls zweidimensionale (→ 3.2.1).
bitblock	Kubi der Niveaus 0 und 1 werden als Bitblöcke behandelt (→ 3.2.1.2).
zeiger	Es wird die in 3.2.1.1 vorgestellte Zeiger–Datenstruktur verwendet, ansonsten die in 3.2.1.3 gezeigte Arrayversion.
unix	Das Programm wird unter UNIX compiliert, ansonsten für Borland–C.
unix_visual	Zur Graphikausgabe wird die Visual–Bibliothek benutzt, andernfalls die XGraphik–Bibliothek.
simulate	Es wird ein Programm zur Kopplung mit der VHDL–Simulation erstellt, → 5.3.
hardware	Es wird ein Programm erstellt, das die Hardwarelösung benutzt (noch nicht implementiert).

Tabelle 3.1: Compileroptionen

3.1.3 Benutzerschnittstelle

Das Programm kann entweder im interaktivem Modus oder — durch Angabe einer Kommandozeilenoption — im Batchmodus gestartet werden. Im letzteren Fall wird die in der Kommandozeile spezifizierte Datei zeilenweise bearbeitet. Jede Zeile enthält dabei den Aufruf einer High–Level–Operation.

[3] Der Vorteil dieser Bibliotheken ist, daß sie eine überschaubare Anzahl von einfachen Graphikfunktionen zur Verfügung stellen.

[4] unter UNIX in der Datei makefile, unter Borland–C im Menü Options, Compiler, Code Generation

Tabelle 3.2 enthält eine Liste aller von der Batchverarbeitung akzeptierten Befehle, Tabelle 3.3 zeigt die Kommandozeilenoptionen und Abb. 3.2 die XGraphik–Version der Benutzeroberfläche.

Generierung:	
GenTest *q n*	Generiert festen Testbaum Nr. *n*
GenBall *r*	Generiert Kugel mit Radius *r*
GenQuad *x y z*	Generiert Quader mit Kantenlängen *x y z*
Transformationen:	
Reflect *n*	Spiegeln/Klappen
Move *achse dist vz*	Verschieben
Turn *achse winkel*	Drehen
Scale *achse factor*	Skalieren
Taper *achse achse factor*	Verjüngen
Mengenoperationen:	
Unite	Vereinigung
Subtract	Differenz
Intersect	Durchschnitt
Complement	Komplement
Include	Test auf Inklusion
Disjunct	Test auf Disjunktion
Morphologische Transformationen:	
Hull	Hüllenberechnung
Geometrische Größen:	
Area	Oberfläche des Modells
Volume	Volumen des Modells
Visualisierung:	
Wire *q*	Drahtmodell
Display *q*	Schnittebenen
Dlayer *q layer*	Schnittebenen, eine Schicht
Drange *q low up*	Schnittebenen, Schichten von *low* bis *up*
Anderes:	
Modelmove *dest*	Verschieben von Modell 2 nach Modell *dest*
q gibt stets den Kubusbaum an	

Tabelle 3.2: Befehle im Batchmode

Ohne Simulationskopplung:	
`-d` *batchfile*	Batchmodus mit graphischer Ausgabe
`-b` *batchfile*	Batchmodus ohne graphische Ausgabe
Mit Simulationskopplung:	
`-w` *batchfile*	Schreiben der Stimulidatei
`-r` *batchfile*	Auswerten der Resultatdatei

Tabelle 3.3: Kommandozeilenparameter

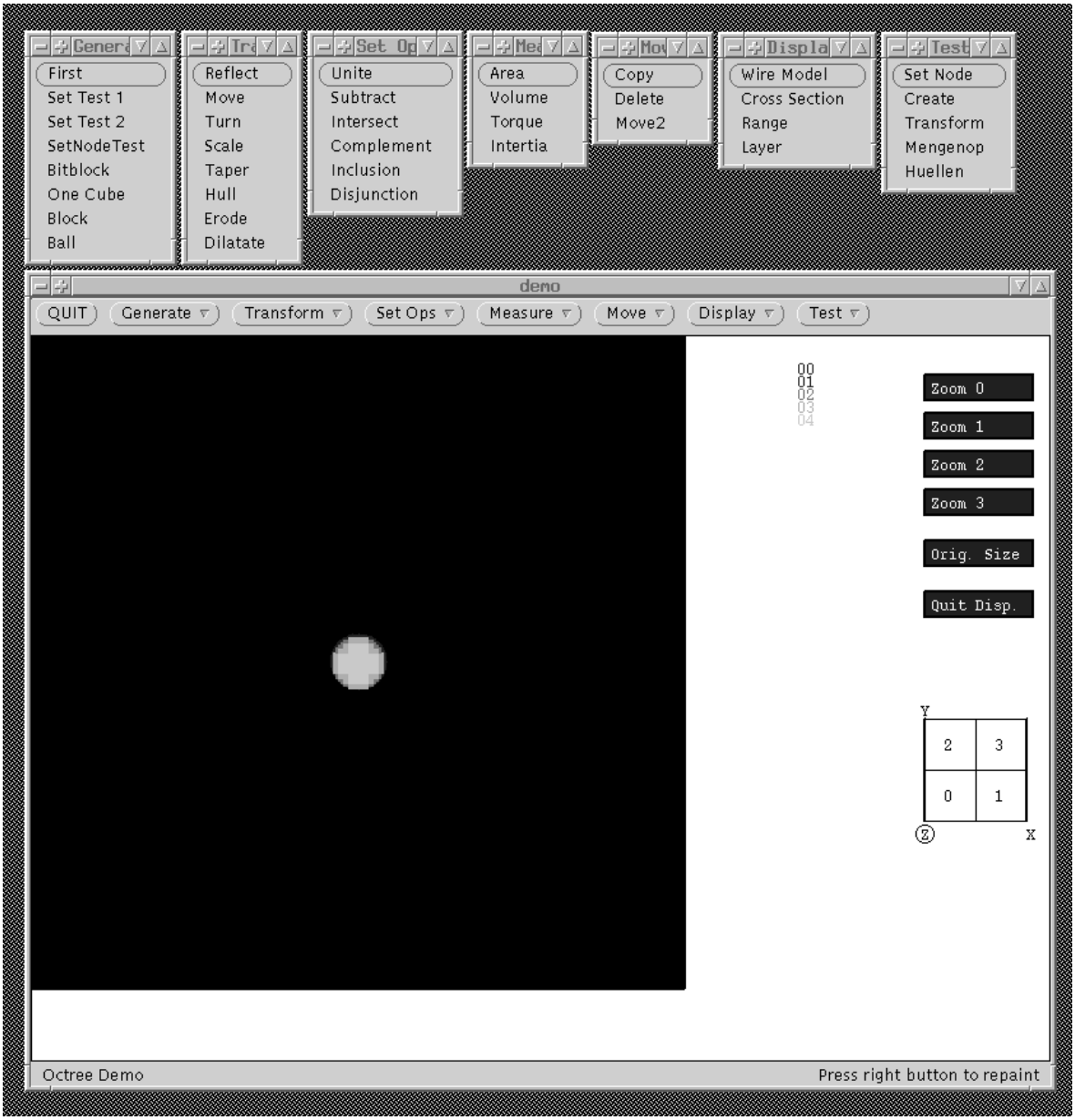
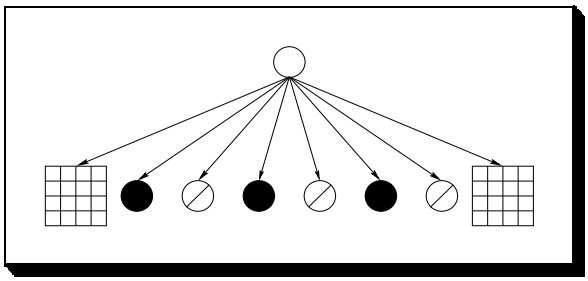


Abbildung 3.2: Benutzeroberfläche mit „angepinnten“ Menüs

3.2 Datenstrukturen

Beginnen wir die Vorstellung der C–Implementierung mit einer Erläuterung der verwendeten Datenstrukturen sowie der elementaren Zugriffsmethoden auf diese. Zusammen mit den Low–Level–Operationen aus dem folgenden Abschnitt bilden diese einen „ADT Octree“, den zu implementieren wesentliche Aufgabe des ersten Teils der vorliegenden Arbeit war.



3.2.1 Datenstruktur Octree

Die Datenstruktur „Octree“ wurde während der Erstellung des Gesamtsystems einigen Modifikationen unterworfen — von der einfachen Zeigerimplementierung bis zur „Speichersimulation“ im Array, die in der folgenden Darlegung nicht unerwähnt bleiben sollen; waren doch diese Modifikationen der Datenstruktur sowie die damit verbundene Adaption sowohl der Low– wie auch der High–Level–Operationen ein nicht unwesentlicher Teil der Entwicklung der Softwarelösung.

3.2.1.1 Implementierung als Zeigerstruktur

Bei der ersten Implementierung des Systems wurden nur Algorithmen und Datenstrukturen für zweidimensionale Objekte[5] und Operationen realisiert, da sich 2D–Objekte sehr leicht graphisch darstellen lassen und der Ablauf von Operationen auf diesen leicht nachvollzogen werden kann.
Weiterhin wurde die Datenstruktur Quad– bzw. Octree zunächst als einfache Zeigerstruktur (siehe rechts) implementiert. Der Vorteil dieser Implementierung war, daß keine eigene Speicherverwaltung implementiert werden mußte und so Fehlerquellen auf der Ebene der Datenstrukturen und deren Zugriff vermieden wurden. Bei den späteren Modifikationen konnte schnell entschieden werden, ob Fehler bei Transformationen in den Transformationsroutinen oder in den Datenstrukturzugriffen lagen, indem auf die Zeigerstruktur zurückgegriffen wurde.

```
#ifdef zachse
  #define TREETYPE 8
#else
  #define TREETYPE 4
#endif

  #define EMPTY 0
  #define FULL 1
  #define SPLIT 2

  typedef struct octreenode
                    *octree;
  typedef struct octreenode {
     int node;
     octree next[TREETYPE];
  } Octreenode;
```

Nachdem die ersten Transformationsalgorithmen brauchbare Ergebnisse lieferten, wurde eine kleine, aber speicherplatzschaffende Änderung der Datenstruktur vorgenommen: Der bisher explizit gespeicherte Knoteninhalt `node` wurde in die `next`–Zeiger verlagert, indem zwei besondere Octree–Zeiger eingeführt wurden, die als „voll“ bzw. „leer“ interpretiert wurden. „Leer“ wurde einfach durch den Nullzeiger repäsentiert, „voll“ durch den beliebigen, aber fest gewählten Zeiger `fullpointer`.

```
#define EMPTYNODE NULL
#define FULLNODE fullpointer

typedef struct octreenode
                    *octree;
typedef struct octreenode {
   octree next[TREETYPE];
} Octreenode;

octree fullpointer;
```

[5] „Quadtrees“

3.2.1.2 Einführung der Bitblöcke

Nachdem auf der bisher implementierten Datenstruktur Transformationen durchgeführt werden konnten, wurde das System um das in 2.1.4 vorgestellte Konzept der Bitblöcke erweitert. Die Einbindung der Bitblöcke erfolgte zunächst mit der rechts gezeigten „sicheren“ Methode, die den offensichtlichen Nachteil hat, für *jeden* Knoten im Baum einen Zeiger auf einen Bitblock zu reservieren.

```
typedef unsigned int bblock[4];

typedef struct octreenode
                    *octree;
typedef struct octreenode {
   octree next[TREETYPE];
   bblock *bb;
} Octreenode;
```

Dieser Nachteil wurde dadurch beseitigt, daß die Bitblockzeiger einfach in die next–Zeiger der Knoten vom Niveau 2 geschrieben werden. Dies ist insofern „gefährlich“, als daß nur noch anhand des Niveaus entschieden werden kann, ob ein von einem geteilten Knoten ausgehender Zeiger auf einen weiteren Octreeknoten oder einen Bitblock zeigt, was eine Fehlersuche sehr erschwert.

3.2.1.3 Implementierung als Array

Zuletzt wurde die Zeigerstruktur in eine Arraystruktur wie in 2.1.5 beschrieben transformiert. Die wesentliche Arbeit bei dieser Änderung lag in der Anpassung der Zugriffsroutinen, die in ihrer endgültigen Fassung im nächsten Kapitel beschrieben werden. Die Datenstruktur selbst besteht nur noch aus einem Array octmem und octmemfree, dem Index des ersten freien Elementes in diesem Array.

```
#define EMPTYNODE 0
#define FULLNODE UINT_MAX

typedef unsigned int bblock;
typedef unsigned int octree;

unsigned int octmem[MEMSIZE];
unsigned int octmemfree;
```

Vier eingeführte Makros ermöglichen den Zugriff auf die Datenstrukturen, sowohl in der Zeiger– als auch in der Arrayvariante, ohne im Programm eine entsprechende Unterscheidung durch Compileroptionen treffen zu müssen.
NODE_NEXT liefert einen auf den durch B gegebenen Sohn des durch A gegebenen Baumzeigers[6] zeigenden Baumzeiger. TREETOBB liefert den von einem Knoten vom Niveau 2 ausgehenden Bitblockzeiger und TREETOROOT die Adresse des Wurzelzeiger. Zuletzt dient das Makro BB dazu, auf die i. Zahl des q. Bitblocks zuzugreifen.

```
#ifdef zeiger
  #define NODE_NEXT(A, B)
                    (&((*A)->next[B]))
  #define TREETOBB (bblock *) *tree[q]
  #define TREETOROOT &rootp[q]
  #define BB(Q,I) ((*bb[Q])[I])
#else
  #define NODE_NEXT(A, B) (A+(*A)+B)
  #define TREETOBB NODE_NEXT(tree[q],0)
  #define TREETOROOT &octmem[q]
  #define BB(Q,I) (*(bb[Q]+(I)))
#endif
```

3.2.2 Globale Variablen

Neben den eigentlichen Octrees ist die Speicherung einiger weiterer Daten von Nutzen. Dabei handelt es sich in erster Linie um die in Tabelle 2.3 aufgelisteten Kenndaten des aktuellen Kubus.

Die High–Level–Operationen sind vom Typ $tree' = f(tree)$ (Transformationen) bzw. $tree' = tree_1 \diamond tree_2$ (Mengenop.). Daher ist es notwendig, drei Kubusbäume gleichzeitig im Speicher zu halten. Alle hier aufgeführten Variablen sind also als dreielementiges Array ausgelegt[7]. Die Zeiger auf die aktuellen Bitblöcke werden sogar als vierelementiges Array behandelt, da ein zusätzlicher „Hilfsbitblock“ benötigt wird.

Nicht festgehalten wird die vektorielle Position des aktuellen Kubus ($\vec{p}$), da sie sich bei Bedarf aus der oktalen Position p ergibt — hierzu dient der Vektor oct_to_vec — und die Kantenlänge des aktuellen Kubus, die sich mittels $l = 2^n$ berechnen läßt.

[6] ein Baumzeiger sei ein Zeiger auf einen Zeiger auf einen Octreeknoten
[7] dieser Arrayindex wird in einigen späteren Erklärungen aus Übersichtlichkeitsgründen nicht erwähnt

Zur effizienten Implementierung der Navigationsoperation up sollten der Zeiger auf den aktuellen Knoten sowie die aktuelle Position bei einer Navigation nach unten auf Stacks abgelegt werden. Zu diesem Zweck wurden die Stacks treestack und octstack eingeführt. Aus octstack läßt sich auch der oktale Index des aktuellen Knotens ableiten.

In der Statusvariablen status wird festgehalten, ob die Inhalte der Variablen v_middle und v_index gültig sind (→ 3.3.4).

```
/*****************************************************************************/
/**                          Globale Variablen                              **/
/*****************************************************************************/

#ifdef zeiger
   octree rootp[3];                 /* Zeiger auf die Wurzeln                */
#else
   unsigned int octmem[MEMSIZE];    /* gesamter Speicher fuer Octrees        */
   unsigned int octmemfree;         /* Index des 1. freien El. in octmem     */
#endif

octree *tree[3];                    /* Zeiger auf Zeiger auf aktuelle Knoten */
bblock *bb[4];                      /* Zeiger auf akt. Bitblock und Hilfsbb. */
int Node[3];                        /* Typ d. akt. Knotens (Full, Empty, Split) */
int Niveau[3];                      /* aktuelles Niveau                      */
int Pos_Oct[3]={0,0,0};             /* oktale Pos. des aktuellen Knotens     */

Vector3D v_middle[3];               /* Mittelpunkt des aktuellen Knotens     */
Vector3D v_indexv[3]={{1,1,1},{1,1,1},{1,1,1}};
                                    /* vekt. Index des aktuellen Knotens     */

int octstack[3][MAXNIVEAU];         /* fuer UP und okt. Index                */

Tree_Stack treestack[3];            /* Stack von Octreepointer, fuer UP      */

int status[3]={0,0,0};              /* Status: v_middle/v_index gueltig ?    */

Vector3D oct_to_vec[8] = {{0,0,0},{1,0,0},{0,1,0},{1,1,0},
                          {0,0,1},{1,0,1},{0,1,1},{1,1,1}};
```

Einer weiteren Erkärung mag die Struktur tree bedürfen: Zur Begründung, warum der aktuelle Knoten als octree *, also als Zeiger auf Zeiger vom Typ octreenode abgelegt wird, betrachte man Abb. 3.3 sowie nebenstehendes Programmfragment. Es seien die Definitionen in Zeile 1 und 2 sowie die Zuweisung in den Zeilen 4 und 5 erfolgt. Wird nun eine Zuweisung wie in Zeile 7 (oder Zeile 8) gezeigt durchgeführt, so erhält tree_f zwar den neuen Wert FULLNODE, p->next[1] jedoch wurde nicht verändert (Abb. 3.3 rechts). Erst die Anweisung in Zeile 10 liefert das gewünschte Resultat: p->next[1] erhält den Wert FULLNODE.

```
1  octree *tree;
2  octree tree_f;
3
4  tree = &(p->next[1]);
5  tree_f = p->next[1];
6
7  tree_f = FULLNODE;
8  oder tree_f = treealloc(2);
9
10 *tree = FULLNODE;
11 oder *tree = treealloc(2);
```

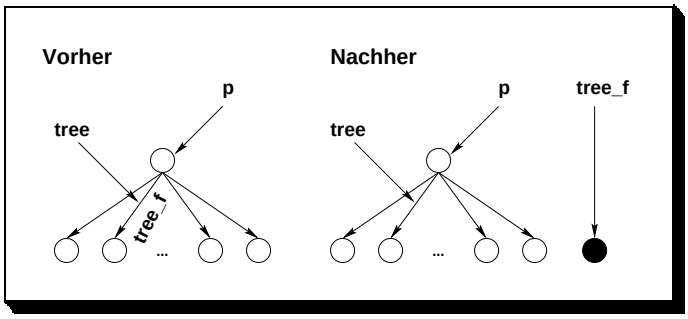


Abbildung 3.3: Zeiger auf octree

3.2.3 Initialisierung der Datenstrukturen

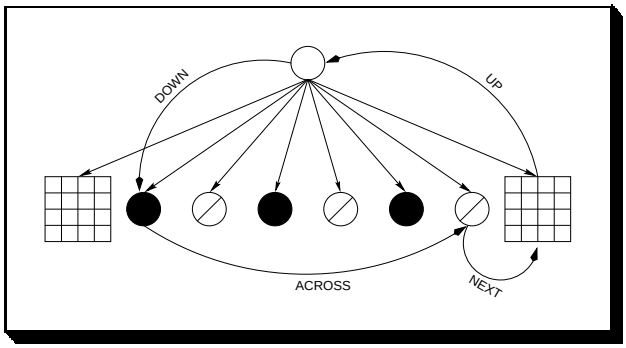
Der Vollständigkeit halber seien an dieser Stelle auch die Initialwerte der einzelnen Variablen erwähnt. Einer näheren Erläuterung mag lediglich die Initialisierung von status bedürfen; hierzu sei auf Kap. 3.3.4 verwiesen.

```
1  void init_tree(int q)
2  {
3      tree[q] = TREETOROOT;
4      *tree[q] = EMPTYNODE;
5      bb[q] = NULL;
6      Node[q] = EMPTY;
7      Niveau[q]=MAXNIVEAU;
8      v_middle[q][X]=v_middle[q][Y]=v_middle[q][Z]= 0;
9      v_indexv[q][X]=v_indexv[q][Y]=v_indexv[q][Z]= 1 << MAXNIVEAU;
10     Pos_Oct[q] = 0;
11     status[q] |= SET_DIRTIES;
12     tree_initstack(&treestack[q]);
13 }
14
15 void init_octree(void)
16 {
17   int i;
18 #ifdef zeiger
19   fullpointer = treealloc(3);
20 #else
21   bb[3] = &octmem[3];
22   octmemfree = 7;
23 #endif
24   for (i=0; i<3; i++) init_tree(i);
25 }
```

3.3 Low–Level–Operationen

Nachdem wir im vorhergehenden Abschnitt die Datenstrukturen zur Realisierung eines „ADT Octree“ eingehend betrachtet haben, folgt nun eine Behandlung der Low–Level–Operationen, also den Operationen, die unmittelbar auf den Datenstrukturen operieren.

Die hier vorgestellten Operationen bilden die Basis für die Entwicklung der High–Level–Operationen, da diese von den Implementierungsdetails der Datenstrukturen unabhängig operieren. Darüberhinaus wird ein Großteil der hier vorgestellten Operationen vom Chip zur Verfügung gestellt werden, somit sind die hier vorgestellen Algorithmen Ausgangspunkt für die spätere Hardwareentwicklung.



Bei der Erläuterung der Prozeduren wird der fast allen Prozeduren gemeinsame Parameter q zur Selektion des zu behandelnden Kubusbaumes ($q \in \{0, 1, 2\}$) nicht mehr explizit erwähnt.

3.3.1 Bitschiebeoperationen

Die High–Level–Operationen benutzen eine Reihe von Funktionen zum Test einzelner Bits oder Bitfolgen innerhalb eines Wortes.

Die Funktionen digit, leftright und right extrahieren aus einer gegebenen Interzahl a bestimmte Bits. Sei

$$a = a_n \ldots a_{i+1} a_i a_{i-1} \ldots a_0$$

die Binärdarstellung von a, so liefert digit(a, i) die Stelle a_i. Nach Aufruf von leftright(a, i, &l, &r) gilt

$$l = a_n \ldots a_{i+1} a_i 0 \ldots 0$$

und

$$r = 0 \ldots 0 a_{i-1} \ldots a_0$$

right ruft lediglich leftright auf und liefert r zurück.

```
int digit(int i, int a)
{
   return (a >>   i & 1);
}

void leftright (int i, int a,
                int *left, int *right)
{
  unsigned int t;
  t = a;  t >>= i;  t <<= i;
  *left = t;
  *right = a ^ t; /* ^: XOR */
}
```

3.3.2 Zugriffe auf die Datenstruktur

In engem Zusammenhang mit der in 3.2.1 gezeigten Entwicklung der Datenstruktur zum Speichern von Octrees stehen die elementaren Zugriffsfunktionen. Hier machen sich die Unterschiede zwischen den einzelnen Varianten deutlich bemerkbar.

3.3.2.1 Erzeugen neuer Knoten im Baum

Wie bereits in Tafel 2.1 erwähnt, dient die Prozedur attach dazu, einen neuen Knoten in den Baum einzufügen. Die wesentliche Arbeit erledigt dabei die Operation treealloc.
attach führt im Fall der Compilierung mit der Option bitblock (→ 3.1.2) eine Fallunterscheidung bezüglich des aktuellen Niveaus[8] durch (Zeilen 5–13): Ist das aktuelle Niveau kleiner oder gleich dem Niveau, auf dem die Bitblöcke beginnen, so darf kein Speicher für Zeigerblöcke alloziert werden. Sind beide Niveaus gleich, so muß Speicher für einen Bitblock alloziert werden (Zeilen 7–8) und — im Falle der Datenstruktur aus 3.2.1.1 — mit der Baumstruktur verbunden werden (Zeilen 9–11).

```
1  void attach(int q)
2  {
3     if (Node[q] != SPLIT)
4         node_set(q, SPLIT);
5  #ifdef bitblock
6     if (Niveau[q] <= BITBLOCKNIVEAU){
7      if (Niveau[q] == BITBLOCKNIVEAU)
8         bb[q]=bballoc(q);
9   #ifdef zeiger
10     *tree[q]=(octree *) bb[q];
11  #endif
12     } else
13 #endif
14        *tree[q]=treealloc(q);
15 }
```

3.3.2.2 Reservieren von Speicher

Die Funktion treealloc hat die Aufgabe, Speicherplatz für einen Zeigerknoten abhängig von der verwendeten Datenstruktur zu reservieren (Zeilen 6–7 bzw. 25). Der reservierte Speicherplatz wird mit dem Wert für leere Kubi initialisiert (Zeilen 12–13 und 23–24). Ist kein Speicher mehr verfügbar, gib treealloc eine Fehlermeldung aus und bricht die Programmausführung ab. (Zeilen 8–11 und 18–21).

Der Rückgabewert von treealloc ist ein Zeiger auf den allozierten Speicherblock.

```
1  octree treealloc(int q)
2  {
3  #ifdef zeiger   /* Zeigermodell */
4    octree p;
5    int i;
6    p = (octree) malloc(
7        (size_t) sizeof(Octreenode));
8    if (!p) {
9        printf("Speicher voll! %d",q);
10       exit(1);
11   }
12   for (i=0;i<TREETYPE;i++)
13       p->next[i]=EMPTYNODE;
14   return p;
15 #else   /* Arraymodell */
16   octree *p;
17   int i;
18   if (octmemfree == MEMSIZE) {
19       printf("Speicher voll! %d",q);
20       exit(1);
21   }
22   p = &(octmem[octmemfree]);
23   for (i=0;i<TREETYPE;i++)
24       *(p+i)=EMPTYNODE;
25   octmemfree += TREETYPE;
26   return (p-tree[q]);
27 #endif /* Array/Zeigermodell */
28 }
```

3.3.2.3 Setzen von Inhalten

Ebenfalls in Tafel 2.1 wurden die Operationen set_full und set_empty zum Setzen des aktuellen Kubus auf „voll“ bzw. „leer“ erwähnt. Beide Operationen werden in der C-Implementierung von der Funktion node_set ausgeführt.

Befindet sich der aktuelle Kubus auf einem der in Bitblöcken gespeicherten Niveaus, so wird die Prozedur handle_bitblock (→ 3.3.6.1) aufgerufen, die eine Baumidentifikation q, die aktuelle oktale Position, die oktale Position des Vaters des aktuellen Knotens, das aktuelle Niveau sowie den zu setzenden Kubustyp (cont $\in \{\mathsf{FULL}, \mathsf{EMPTY}\}$) als Parameter erwartet. Andernfalls wird der Zeiger auf den aktuellen Knoten und der Inhalt der Variablen Node (→ 3.2.2) entsprechend cont gesetzt.

[8] Die Konstante BITBLOCKNIVEAU bezeichnet dabei das Niveau, bei dem von Zeigerknoten auf Bitblöcke gewechselt wird. Werden die Niveaus 0 und 1 in Bitblöcken gespeichert, so hat BITBLOCKNIVEAU den Wert 2.

```
1  void node_set(int q, int cont)
2  {
3  #ifdef bitblock
4    if (Niveau[q] < BITBLOCKNIVEAU){
5       if (cont != SPLIT)
6          handle_bitblock(
7             q, Pos_Oct[q],
8             pos_tos(q),
9             Niveau[q], cont);
10   }
11   else
12 #endif
13    if (cont == EMPTY) {
14       *(tree[q]) = EMPTYNODE;
15       Node[q] = EMPTY;
16    }
17    else if (cont == FULL) {
18            *(tree[q]) = FULLNODE;
19            Node[q] = FULL;
20         }
21         else
22            Node[q] = SPLIT;
23 }
24
```

3.3.2.4 Auslesen von Knoteninhalten

Die Funktion node_get liefert den Typ des aktuellen Knotens (FULL, EMPTY oder SPLIT) zurück. Liegt der aktuelle Knoten auf Bitblockniveau, so wird wie in node_set die Funktion handle_bitblock aufgerufen, die bei cont = 2 keine Modifikationen am aktuellen Bitblock vornimmt, sondern den Typ des gewählten Kubus liefert.

Liegt der aktuelle Kubus auf normalem Zeigerniveau — oder werden keine Bitblöcke verwendet —, so bestimmt ein Vergleich des Zeigers auf den aktuellen Knoten mit den Zeigern auf die speziellen Knoten FULLNODE und EMPTYNODE das Funktionsresultat.

```
1  int node_get(int q)
2  {
3    octree p;
4    p = *(tree[q]);
5  #ifdef bitblock
6    if (Niveau[q] < BITBLOCKNIVEAU)
7       return handle_bitblock(
8                q, Pos_Oct[q],
9                pos_tos(q),
10               Niveau[q], 2);
11   else
12 #endif
13   if (p == EMPTYNODE)
14               return EMPTY;
15     else if (p == FULLNODE)
16               return FULL;
17          else return SPLIT;
18 }
19
20
```

3.3.3 Navigation im Octree

3.3.3.1 Navigation zum Wurzelknoten

Bei der Navigation zum Wurzelknoten durch die Prozedur root werden im wesentlichen die Variablen aus 3.2.2 initialisiert: den Baumzeigern[9] werden die Zeiger auf die Wurzeln zugewiesen, die Bitblockzeiger werden auf NULL gesetzt, Niveau, v_middle und v_indexv werden mit den für das maximale Niveau gültigen Werten versehen. Schließlich wird der Stack für Baumzeiger geleert (Zeilen 10–11), Node wird der für den Wurzelknoten geltende Typ zugewiesen und der Inhalt der Variablen v_midde und v_indexv wird für gültig erklärt (Zeile 13, vgl. 3.3.4)

```
1  void root(int q)
2  {
3     tree[q] = TREETOROOT;
4     bb[q]=NULL;
5     Niveau[q] = MAXNIVEAU;
6     v_middle[q][X] = v_middle[q][Y]
7       = v_middle[q][Z] = 0;
8     v_indexv[q][X] = v_indexv[q][Y]
9       = v_indexv[q][Z] = 1<<MAXNIVEAU;
10    while (!tree_empty(q))
11          tree_pop(q);
12    Node[q] = node_get(q);
13    status[q] &= ~SET_DIRTIES;
14 }
```

[9] also den Zeigern auf die Zeiger auf die aktuellen Knoten

3.3.3.2 Navigation zu Subkubi

Die Operation down zur Navigation zum p. Sohn des aktuellen Kubus prüft zunächst, ob eine Navigation „nach unten“ überhaupt möglich ist (Zeilen 3–8): Wenn der Zeiger auf den aktuellen Knoten mit keinem der beiden ausgezeichneten Zeiger EMPTYNODE und FULLNODE identisch ist oder wir uns — bei Verwendung von Bitblöcken — auf Bitblockniveau befinden, kann navigiert werden, ansonsten wird eine Fehlermeldung ausgegeben (Zeilen 26–30).

Im Fall der Verwendung von Bitblöcken wird bei Navigation von dem Niveau, in dem die Bitblöcke beginnen, der Zeiger auf den aktuellen Bitblock gesetzt (Z. 11–12).

In den Zeilen 16 und 17 wird der aktuelle Baumzeiger auf einen Stack gelegt und mit dem Makro NODE_NEXT (→ 3.2.1.3) aktualisiert. Werden Bitblöcke verwendet, geschieht dies nur, wenn wir uns oberhalb der Bitblöcke befinden, ansonsten in jedem Fall.

Die aktuelle, oktale Position wird auf einen Stack gelegt, falls wir nicht vom maximalen Niveau ausgehen[10].

Zum Abschluß der Prozedur werden die oktale Position, das aktuelle Niveau und der Inhalt des aktuellen Knotens mit den nun gültigen Werten belegt und der Inhalt der Variablen v_middle und v_indexv wird für ungültig erklärt (Zeile 22, → 3.3.4).

```
1  void down(int q, int p)
2  {
3    if (    (    (*tree[q] != EMPTYNODE)
4               && (*tree[q] != FULLNODE ))
5  #ifdef bitblock
6         || (Niveau[q] <= BITBLOCKNIVEAU)
7  #endif
8       )
9    {
10 #ifdef bitblock
11     if (Niveau[q] == BITBLOCKNIVEAU)
12        bb[q] = TREETOBB;
13     if (Niveau[q] > BITBLOCKNIVEAU)
14 #endif
15     {
16       tree_push(q, tree[q]);
17       tree[q] = NODE_NEXT(tree[q], p);
18     }
19     if (Niveau[q] < MAXNIVEAU)
20        pos_push(q, Pos_Oct[q]);
21     Pos_Oct[q]=p;
22     status[q] |= SET_DIRTIES;
23     Niveau[q]--;
24     Node[q] = node_get(q);
25   }
26   else {
27     printf("Zu weit down %d %d",
28             q, Niveau[q]);
29     exit(1);
30   }
31 }
32
```

3.3.3.3 Navigation zum Vaterknoten

Die Operation up macht, bildlich gesprochen, alle von down vorgenommenen Modifikationen rückgängig: das Niveau wird inkrementiert; falls wir uns nicht in Bitblöcken bewegen, wird der bei down geschriebene Baumzeiger vom Stack gelesen (Zeilen 7–9) und falls wir nicht zum maximalem Niveau zurückkehren, die aktuelle Position (Zeilen 10–11).

Werden bei der Navigation die in Bitblöcken gespeicherten Niveaus verlassen, so wird der Zeiger auf den aktuellen Bitblock auf NULL gesetzt (Zeilen 4–6). Schließlich werden noch v_middle und v_indexv wird für ungültig erklärt (Zeile 12, → 3.3.4) und Node aktualisiert.

```
1  void up(int q)
2  {
3   if (!tree_empty(treestack[q])){
4  #ifdef bitblock
5     if (Niveau[q] == BITBLOCKNIVEAU-1)
6         bb[q] = NULL;
7     else if (Niveau[q] >= BITBLOCKNIVEAU)
8  #endif
9        tree[q] = tree_pop(q);
10    if (Niveau[q] < MAXNIVEAU-1)
11       Pos_Oct[q] = pos_pop(q);
12    status[q] |= SET_DIRTIES;
13    Niveau[q]++;
14    Node[q] = node_get(q);
15  }
16 }
```

[10] Es ergibt keinen Sinn, die aktuelle Position auf maximalem Niveau zu betrachten, da dies den gesamten Raum darstellt und der Kubus auf maximalem Niveau kein Subkubus eines übergeordneten Kubus ist.

3.3.3.4 Navigation zu Bruderknoten

Die Operation next navigiert vom aktuellen Kubus mit oktaler Position p zum Kubus mit oktaler Position $p+1$ im gleichen übergeordneten Kubus. Vom Kubus der Position $p = 7$ wird zum Kubus 0 navigiert. Die Prozedur next führt die Berechnung von $p+1$ durch und ruft across auf.

```
1  void next(int q)
2  {
3     int p;
4     p = Pos_Oct[q]+1;
5     if (p == TREETYPE) p = 0;
6     across(q, p);
7  }
```

Bei across, der Navigation zum p. Sohn des Vaters des aktuellen Knotens, wird zunächst überprüft, ob der Wert des Parameters p gültig ist (Zeilen 3–6). Werden keine Bitblöcke verwendet oder befinden wir uns oberhalb der in Bitblöcken gespeicherten Niveaus, so wird der Baumzeiger mit Hilfe des Makros NODE_NEXT aktualisiert. Dabei wird jedoch nicht vom aktuellen Knoten ausgegangen (vgl. next), sondern von dessen Vater, auf den über den zuletzt auf den Stack gelegten Baumzeiger (tree_top(q)) zugegriffen wird (Zeilen 7–11).
Danach wird ggf. der Zeiger auf den aktuellen Bitblock aktualisiert. Zum Schluß bekommen — wie üblich — status, Pos_Oct und Node die derzeit gültigen Werte zugewiesen.

```
1  void across(int q, int p)
2  {
3    if (p >= TREETYPE) {
4       printf("Fehler in across!");
5       exit(1);
6    }
7  #ifdef bitblock
8    if (Niveau[q] >= BITBLOCKNIVEAU)
9  #endif
10      tree[q] = NODE_NEXT(
11                    tree_top(q), p);
12 #ifdef bitblock
13   if (Niveau[q] == BITBLOCKNIVEAU)
14      bb[q] = TREETOBB;
15 #endif
16   status[q] |= SET_DIRTIES;
17   Pos_Oct[q]=p;
18   Node[q] = node_get(q);
19 }
```

3.3.4 Mittelpunkt und vektorieller Index

In die bei den High–Level–Operationen durchzuführenden Berechnungen fließen oft die Mittelpunktskoordinaten des aktuellen Kubus ein. Wie bereits in 2.1.3 gezeigt, gibt es zur Verwaltung dieser Koordinaten zwei Ansätze: die Mittelpunktskoordinaten bei jeder Navigationsoperation zu aktualisieren (→ 2.1.3.1) oder bei Bedarf aus dem vektoriellen oder oktalen Index zu berechnen.

Erstere Methode hat sich in der Softwarerealisierung nicht bewährt: Eine Betrachtung der Algorithmen auf höherer Ebene zeigt, daß sehr viele Navigationsoperationen durchgeführt werden, bevor eine Abfrage des aktuellen Mittelpunktes stattfindet. Da die Operationen zur Aktualisierung sehr viel Rechenzeit kosten, ist eine Berechnung des Mittelpunktes bei Bedarf vorzuziehen.

Um zu verhindern, daß eine Aktualisierung stattfindet, obwohl nach der letzten Neuberechnung keine Navigation stattgefunden hat — die Werte von v_middle bzw. v_index also noch gültig sind —, wurde die Variable status (→ 3.2.2 bzw. Zeile 5) eingeführt. In Bit 0 dieser Variablen wird festgehalten, ob der Inhalt von v_middle gültig ist, in Bit 1 entsprechendes für v_index. Dabei wird „1“ als „ungültig“ interpretiert.
Die Handhabung dieses Statuswortes zeigt nebenstehendes Programmfragment: Mit den Definitionen in den Zeilen 1–3 lassen sich die beiden Statusbits durch Maskierung mittels „Oder“ wie in Zeile 7 gezeigt setzen, durch zusätzliche Invertierung der Maske zurücksetzen (Zeilen 9–10) und durch Maskierung mit „Und“ abfragen (Zeilen 12–13).

```
1  #define MIDDLEDIRTY 01
2  #define INDEXVDIRTY 02
3  #define SET_DIRTIES 03
4
5  int status[3]={0,0,0};
6
7  status[q] |= SET_DIRTIES
8
9  status[q] &= ~MIDDLEDIRTY;
10 status[q] &= ~INDEXVDIRTY;
11
12 status[q] & MIDDLEDIRTY
13 status[q] & INDEXVDIRTY
```

3.3.4.1 Rasterkorrektur

Zur Vermeidung aufwendiger Fließkommaoperationen[11] arbeiten alle Low–Level–Operationen mit ganzen Zahlen. Auch die Kubusmittelpunkte sollen als Zahlen vom Typ Integer behandelt werden.

Dabei entsteht folgendes Problem: Die Kantenlänge der Kubi vom Niveau 0 beträgt $2^0 = 1$, somit liegen die Mittelpunkte dieser Kubi zwischen den ganzzahligen Rasterlinien. Abbildung 3.4 zeigt diese Situation in zweidimensionaler Projektion; die kleinen ausgefüllten Kreise deuten die Mittelpunkte der Kubi auf Niveau 0 an. Um hier nicht auf die Verwendung realer Zahlen ausweichen zu müssen[12], legen wir fest, daß bei Kubi auf Niveau 0 statt des Mittelpunktes der — im zweidimensionalen Fall — untere rechte Eckpunkt geliefert wird (nicht ausgefüllte Kreise in der Abbildung).

Allgemein liefert middle auf Niveau 0 den Mittelpunkt des Kubus abzüglich 0.5 in jeder Komponente.

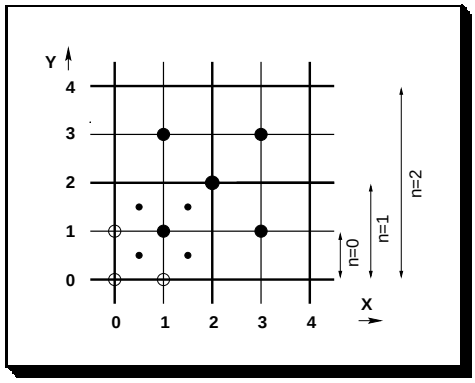


Abbildung 3.4: Rasterkorrektur

Für die High–Level–Operationen ist diese Korrektur nicht so schwerwiegend, wie es zunächst den Anschein haben mag: Die Mittelpunktskoordinaten werden häufig zu Vergleichen zwischen Kubi in unterschiedlichen Modellen, aber eines Niveaus herangezogen. In diesem Fall ist eine Berücksichtigung der Korrektur nicht notwendig[13].

3.3.4.2 Implementierung

Eine zentrale Rolle bei der Berechnung der Mittelpunktsfunktionen spielt in 2.1.3.1 definierte Funktion $\vec{u}$, die die Zielposition[14] p auf einen Richtungsvektor $\vec{r}$ abbildet. Da p bei Verwendung von Octrees nur 8 Werte annehmen kann, wurde die Funktion $\vec{u}$ als Array up_down_vec mit acht Einträgen realisiert. Jeder Eintrag ist dabei ein dreidimensionaler Vektor. Der oben genannten Sonderrolle des Niveaus 0 wird durch Einführung einer zweiten Funktion $\vec{u}_0$ — realisiert durch up_down_vec_0 — entsprochen.

```
Vector3D up_down_vec[8]   = {{-1,-1,-1},{ 1,-1,-1},{-1, 1,-1},{ 1, 1,-1},
                             {-1,-1, 1},{ 1,-1, 1},{-1, 1, 1},{ 1, 1, 1}};
Vector3D up_down_vec_0[8] = {{-1,-1,-1},{ 0,-1,-1},{-1, 0,-1},{ 0, 0,-1},
                             {-1,-1, 0},{ 0,-1, 0},{-1, 0, 0},{ 0, 0, 0}};
```

Die Implementierung der Funktion middle basiert auf dem in 2.1.3.2 angegebenen Algorithmus:

Wenn überhaupt eine Neuberechnung notwendig ist (Zeile 5), wird v_middle mit $\vec{0}$ initialisiert. Dies entspricht dem Mittelpunkt des Kubus auf höchstem Niveau. In der Variablen cl wird die Kubuskantenlänge des in der Iterationen betrachteten Niveaus gespeichert. Sie wird mit 2^{N-1} initialisiert[15] (Zeile 9) und in jeder Iteration halbiert (Zeile 16).

```
1  Vector3D *middle(int q)
2  {
3    int i, p, cl;
4    Vector3D r;
5    if (status[q] & MIDDLEDIRTY){
6      v_middle[q][X] =
7      v_middle[q][Y] =
8      v_middle[q][Z] = 0;
9      cl = 1 << MAXNIVEAU - 1;
10     ...
```

[11] vor allem in Hinblick auf eine spätere Hardwarerealisierung

[12] in 2.1.3 wurde bereits erwähnt, daß Niveau 0 als Ausnahmefall behandelt wird

[13] da $\vec{m}_0 - \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} < \vec{m}_1 - \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} \Leftrightarrow \vec{m}_0 < \vec{m}_1$

[14] auch hier wurde bei der Softwarerealisierung die Verwendung der oktalen Position vorgezogen

[15] N entspricht MAXNIVEAU

Pro Iteration wird nun die zum Niveau i korresponierende oktale Zielposition p dem Index extrahiert (Zeile 12).
Aus p wird in den Zeilen 13–20 der Vektor r berechnet und mit der Kubuskantenlänge cl skaliert (vgl. Tabelle 2.2). Hier erfolgt auch die oben beschriebene Sonderbehandlung des Niveaus 0 (r wird auf Niveau 0 wegen $2^n = 2^0 = 1$ nicht skaliert). Danach wird der berechnete Vektor r zum aktuellen Mittelpunkt v_middle addiert.
Zum Abschluß der Funktion wird noch der Inhalt von v_middle für gültig erklärt (Zeile 21).

```
9        ...
10     for (i = MAXNIVEAU-1;
11          i >= Niveau[q]; i--){
12       p = indexpos(q, i);
13       if (i > 0)
14          Vector3Scaleto(
15                  up_down_vec[p],
16                  (unsigned int) (cl >>= 1),
17                  r);
18       else Vector3Cpy(up_down_vec_0[p], r);
19       Vector3Add(v_middle[q],r,v_middle[q]);
20       }
21   status[q] &= ~MIDDLEDIRTY;
22   }
23   return &(v_middle[q]);
24 }
```

3.3.4.3 Vektorieller Index

Auch der vektorielle Index wird erst bei Bedarf berechnet und nicht bei jeder Navigation aktualisiert.

Das Grundgerüst der Funktion indexv zur Berechnung des vektoriellen Index entspricht dem der in 3.3.4.2 gezeigten Funktion middle und wird deswegen hier nicht aufgeführt. Der Unterschied zwischen beiden Funktionen liegt lediglich bei der Berechnung des Vektors r: bei der Indexberechnung wird aus der berechneten oktalen Position p die vektorielle Position bestimmt (→ 3.2.2) und um i Stellen nach links verschoben zu index_v addiert. Dies entspricht einer Konkatenation der auf dem Stack liegenden Positionen (→ Beispiel rechts).

Die Zeilen 13–18 in der Prozedur middle sind also durch folgende Zeile zu ersetzen:

```
13 Vector3Shl(i, oct_to_vec[p], r);
```

Beispiel:
Niveau
9 8 7 ...
1
1
1
  1
  0
  1
    0
    1
    0
———————
1 1 0 ...
1 0 1 ...
1 1 0 ...

3.3.5 Globale Navigation

Neben den bereits vorgestellten Navigationsoperationen, die zu adjazenten Knoten navigieren (next, down usw.) stellt der „ADT Octree“ noch eine Reihe von Operationen zur Verfügung, die in größerem Rahmen navigieren. Zur Unterscheidung zu den Erstgenannten wollen wir sie *globale Navigationsoperationen* nennen.[16]

Die Operationen dieser Gruppe sind in die bisher getroffenen Unterscheidung zwischen Low– und High–Level–Operationen nicht genau einzuordnen. Sie werden einerseits von den High–Level–Operationen benutzt, greifen aber nur über die Low–Level–Operationen auf die Datenstruktur zu (vgl. 3.1).

3.3.5.1 Suchen und Setzen von Knoten

Die erste Gruppe der globalen Navigationsoperationen bilden eine Reihe von Operationen, die einen Knoten mit gegebenem vektoriellem Index ziel aufsuchen:

```
void find_node(int q, int n, Vector3D ziel)
void search_node(int n, Vector3D ziel)
void set_node(int n, Vector3D ziel)
void goto_node(int q, Vector3D ziel)
```

[16] In [1] werden diese „elementare Navigationsoperationen“ genannt; diese Bezeichnung erscheint mir jedoch für die in 2.1.2 aufgelisteten Operationen zutreffender.

find_node und search_node suchen bei der Wurzel beginnend den Knoten, der auf Niveau n liegt und dessen Index mit dem Index ziel in den ersten n Stellen übereinstimmt; das Abbruchkriterium ist also das Erreichen des Zielniveaus, nicht die Übereinstimmung beider Indizes! search_node sucht dabei in Baum 2, find_node in Baum q[17]. Die Operation set_node arbeitet wie search_node, setzt jedoch zusätzlich den aufgesuchten Kubus auf „voll“.

Im Unterschied zu den bisher genannten Operationen beginnt goto_node nicht an der Wurzel des Baumes, sondern am aktuellen Knoten und sucht zunächst zur Wurzel hin den kleinsten Kubus auf, der sowohl den aktuellen Kubus als auch den Zielkubus enthält; bei diesem Knoten stimmen die ersten i Stellen der Indizes von aktuellem Knoten und Zielknoten überein. Von dort aus wird in Richtung der Blätter der Zielknoten aufgesucht.

```
1  void goto_node(int q, Vector3D ppe)
2  {
3    int n,i=0;
4    Vector3D a={0,0,0},b={0,0,0};
5    Vector3Cpy(ppe, a);
6    Vector3Cpy(*indexv(q), b);
7    Vector3Shr(Niveau[q], a, a);
8    Vector3Shr(Niveau[q], b, b);
9    n=i=Niveau[q];
10   for(;;){    /* Suche nach "oben" */
11      Vector3Shr(1, a, a);
12      Vector3Shr(1, b, b);
13      i++;
14      if (Vector3Cmp(a,b)) break;
15      else up(q);
16   }          /* Suche nach "unten" */
17   across(q, digit_vec(--i, ppe));
18   while ((i > n) && (Node[q]==SPLIT))
19         down(q, digit_vec(--i, ppe));
20 }
```

```
1  void find_node(int q,
2                 int n, Vector3D a)
3  {
4    int i;
5    i=MAXNIVEAU-1;
6    root(q);
7    do {
8      if (Node[q] != SPLIT) attach(q);
9      down(q, digit_vec(i--, a));}
10   while (Niveau[q] != n);
11
12 }
```

bei set_node:

```
11    node_set(2, FULL);
```

```
1  void search_node(int n, Vector3D a)
2  {
3    find_node(2, n, a);
4  }
```

3.3.5.2 Traversierung eines Octree

Grundlage jeder High-Level-Operation ist eine der Prozeduren zum Traversieren eines Octrees:

```
void parse_tree    (       int cont, void (*proc)(void))
void parse_tree_q  (int q, int cont, void (*proc)(void))
void parse_tree_b  (       int cont, void (*proc)(void), void (*procb)(void))
void parse_tree_bq (int q, int cont, void (*proc)(void), void (*procb)(void))
```

Alle Prozeduren erwarten einen Knotentyp cont (EMPTY oder FULL) sowie einen Zeiger auf eine parameterlose Prozedur proc. parse_tree_q und parse_tree_bq erwarten darüberhinaus eine Identifikation des zu traversierenden Baumes q, parse_tree_b und parse_tree_bq eine zweite Prozedur procb. Die Parameter dieser Funktionen proc und procb werden in der globalen 3×3–Matrix transform_matrix gespeichert.

Die Traversierungprozeduren wenden im Baum q[18] auf alle Knoten des Typs cont die Prozedur proc an. parse_tree_q durchläuft dabei den ganzen Baum bis zu Niveau 0, parse_tree_bq stoppt auf dem Niveau der Bitblöcke und wendet auf diese die Prozedur procb an.

Stellvertretend für alle Traversierungsprozeduren sei hier nur parse_tree_bq aufgeführt:

[17] Die Existenz von search_node ist „historisch“ bedingt: nur sie wird in [1] gefordert, tatsächlich wird aber auch ein Suchen in den anderen Bäumen benötigt.

[18] bzw. im Baum 2; die Existenz der Prozeduren parse_tree und parse_tree_b sind wie search_node „historisch“ bedingt.

```
1  void parse_tree_b_rec(int q, int cont, void (*proc)(void),
2                                         void (*procb)(void))
3  {
4    int i;
5    down(q,0);
6    for (i = 0; i < TREETYPE; i++){
7      across(q, i);
8      if (Node[q] == cont) (*proc)();
9      if (Node[q] == SPLIT)
10        if (Niveau[q] == BITBLOCKNIVEAU)
11             (*procb)();
12        else parse_tree_b_rec(q, cont, proc, procb);
13   }
14   up(q);
15 }
16
17 void parse_tree_bq(int q, int cont, void (*proc)(void), void (*procb)(void))
18 {
19    root(q);
20    if (Node[q] == SPLIT) parse_tree_b_rec(q, cont, proc, procb);
21 }
```

In [1] wird in Zeile 14 statt up die Operation upnext verwendet, was mit einem nachfolgenden next der aufrufenden Ebene dazu führen würde, daß nur jeder zweite Unterbaum bearbeitet wird (vgl. [1] Alg. 2.1-A). Tatsächlich wird an keiner Stelle ein upnext gebraucht, so daß diese Operation bei der Entwicklung der Hardwareunterstützung nicht berücksichtigt wurde.

In der vorgestellten Version wurde across statt next verwendet, da dies eventuelle Navigationsfehler in der Funktion proc (proc verändert die aktuelle Position) korrigiert.

3.3.6 Operationen auf Bitblöcken

Durch das von uns gewählte BITBLOCKNIVEAU $b = 2$ (vergl. 2.1.4) wird ein Bitblock zum Repräsentanten eines Würfels mit 64 elementaren Kubi, aufgeteilt in vier Bitvektoren entsprechend den YZ–Ebenen. Diese Darstellung wird in der C–Realisierung in der Form eines Arrays übernommen, wobei die Bitvektoren durch ein 16 Bit Wort repräsentiert werden. Der aktuelle Bitblock des jeweiligen Baumes wird ebenfalls in einem Array abgelegt, so daß eine zweidimensionale Arraystruktur der Form BitBlock[Baum][YZ–Ebene] entsteht.

Diese Struktur ist im Modul octree.c global definiert, so daß jede Low–Level–Routine durch Angabe des entsprechenden Baumes q auf einen Bitblock und durch Angabe der Ebene direkt auf den Bitvektor zugreifen kann. Gezielten Zugriff auf ein Bit ermöglichen die Routinen bit_read(Position, Bitvektor) und bit_write(Position, Wert, Bitvektor), so daß der Inhalt des entsprechenden Bits gelesen bzw. gesetzt werden kann (auf 0 für EMPTY oder 1 für FULL).

3.3.6.1 Zugriff auf Bitblöcke

Die Problematik, daß beim Zugriff auf Kubi des Niveaus $n = 1$ im Falle der Bitblöcke auf acht elementare Kubi bzw. die entsprechenden Bits zugegriffen werden muß, wurde in 2.1.4.1 dargelegt. Damit der Zugriff auf einen Kubus diesen Niveaus für High-Level Routinen transparent bleibt werden Kubi jenseits des BITBLOCKNIVEAUS durch die Funktion handle_bitblock verwaltet.

Die Funktion erwartet als Parameter den aktuellen Baum q, die aktuelle Position p0, die Position des Vaterkubis p1, sowie das aktuelle Niveau niv. Über den Parameter rw wird gesteuert, ob handle_bitblock lesend (rw= 2) — handle_bitblock liefert den aktuellen Wert zurück — oder schreibend (rw= 0: EMPTY setzen, rw= 1: FULL setzen) auf den aktuellen Kubus zugegriffen werden soll.

```
0    int handle_bitblock(int q, int p0, int p1, int niv, int rw)
1    /* rw=0 Bit loeschen, rw=1 Bit setzen, rw=2 Bit lesen */
2    {
3      Vector3D pv0, pv1;
4      int x,y,z,i=0;
5      Vector3Cpy(oct_to_vec[p0], pv0);
6      Vector3Cpy(oct_to_vec[p1], pv1);
7      if (niv){
8        Vector3Shl(1, pv0, pv0);
9        for(x=pv0[X]; x<pv0[X]+2; x++)
10          for(y=pv0[Y]; y<pv0[Y]+2; y++)
11            for(z=pv0[Z]; z<pv0[Z]+2; z++)
12              if (rw != 2) bit_write ((y<<2)+z, rw, &(BB(q,x)));
13              else i += bit_read((y<<2)+z, BB(q,x));
14       }
15     else              /* niveau 0 */
16        if (rw != 2)
17               bit_write((pv1[Y] << 3)+(pv0[Y] << 2)+(pv1[Z] << 1)+pv0[Z],
18                         rw, &(BB(q, 2*pv1[X]+pv0[X])) );
19        else i = bit_read((pv1[Y] << 3)+(pv0[Y] << 2)+(pv1[Z] << 1)+pv0[Z],
20                         BB(q, 2*pv1[X]+pv0[X]) );
21               /* 4*(2p1y+p0y)+2p1z+p0z */
22     if (i == 8)  i = FULL;
23     else if (i && niv)  i = SPLIT;
24     return i;
25   }
```

handle_bitblock läßt sich in zwei Abschnitte (Zeile 7), den unterschiedlichen Handhabungen der zwei Niveaustufen entsprechend, aufteilen:

Niveau $n = 1$: in den Zeilen 8 bis 14 werden bezüglich der aktuellen Position die 8 signifikanten Bits der jeweiligen Bitvektoren ausgewählt (vgl. 2.1.4.1). In Zeile 12 werden dann im Falle eines schreibenden Zugriffs diese Bits mittels bit_write auf den entsprechenden Wert gesetzt. Andernfalls (lesender Zugriff) werden in Zeile 13 die mit '1' gesetzten Bits gezählt um später (in Zeile 22 bis 24) ausgewertet zu werden (s.u.).

Niveau $n = 0$: in den Zeilen 16 bis 20 wird bezüglich des schreibenden bzw. lesenden Zugriffs unterschieden (Zeile 16) und entsprechend weiter verfahren. Zeilen 17 und 18 handhaben den schreibenden Zugriff und setzen das entsprechende Bit (vgl. 2.1.4.1) auf den angegebenen Wert. Im Falle des lesenden Zugriffs (Zeilen 19 und 20) wird der Wert des Bits für die weitere Auswertung (in Zeile 22 bis 24) ausgelesen.

In den Zeilen 22 bis 24 werden die gelesenen Wertigkeiten der Bits entsprechend dem möglichen Zustand eines Kubus interpretiert: Im Fall $n = 1$ ist der zu prüfende Kubus voll, falls alle Subkubi voll sind; also die in Zeile 13 gebildete Summe der Knoteninhalte seiner Subkubi gleich Acht ist. Sollte die Anzahl der vollen Subkubi von Null verschieden sein, so wird für Niveau $n = 1$ SPLIT zurückgeliefert. In allen anderen Fällen enthält i bereits das korrekte Ergebnis. So ist i im Fall $n = 0$ entweder 0 oder 1 entsprechend voll oder leer.

Der Zugriff auf die einzelnen Bits erfolgt mit der Routinen bit_write und bit_read, die es ermöglichen, gezielt auf einzelne Bits in einem 16 Bit langem Binärwort zuzugreifen.

Bei bit_read wird der jeweilige Wert des mit i angegebenen Bits aus der Bitfolge bf zurückgegeben. Hierfür wird die Bitfolge um i Stellen nach rechts geschoben und mit 1 logisch UND verknüpft, diese Operation liefert dann den Wert des Bits.
Ähnlich verhält es sich bei bit_write: Hier wird neben Position i und Bitfolge bf noch der zu schreibende Wert des Bits b angegeben. Für den Zugriff wird eine entsprechende Maske durch schieben eines Bits um i Stellen nach links erstellt. Diese Maske wird dann, je nachdem ob eine 1 oder 0 gesetzt werden soll logisch mit ODER verknüpft oder invertiert und logisch mit UND verknüpft.

```
0  int bit_read(int i, unsigned int bf)
1  {
2    bf >>= i;
3    return bf & 1;
4  }
```

```
0  void bit_write(int i, int b,
1                 unsigned int *bf)
2  {
3    int j = 1;
4    j <<= i;
5    if (b) *bf |= j;
6    else *bf &= ~j;
7  }
```

3.3.6.2 Transformationen mit Bitblöcken

Ein weiterer Vorteil der Bitblöcke neben der Speicherplatzeinsparung ist, daß einfache Transformationen ohne umständliche Vektormultiplikationen durch Schiebeoperationen ausgeführt werden können, und sich mengentheoretische Operationen auf logische Verknüpfungen von Bitblöcken zurückführen lassen. Der genaue Vorgang wurde schon im Theorieteil erläutert, hier wird nun die Realisierung in C dargestellt.

Als Beispiel sei hier noch einmal das Spiegeln am Ursprung vorgeführt, um zu verdeutlichen wie eine Transformation mit Bitblöcken durchgeführt wird (vergl. 2.2.3). Eine Ebene wird aus dem Ausgangsbitblock entlang der X–Achse herausgeschoben. Diese Ebene wird auf der gleichen Achse in entgegengesetzter Richtung in den Zielbitblock hereingeschoben. Dieses Schieben wird viermal durchgeführt, um jede Ebene des Ausgangsbitblocks in den Zielbitblock zu transferieren. Dieser Vorgang führt zu einer Spiegelung an der YZ-Ebene. Die Punktsymmetrie für das Spiegeln am Ursprung wird dadurch erreicht, daß mit der zu übertragenden Ebene noch einer Rotation durchgeführt wird. Der entsprechende Vorgang wird durch nebenstehende Abbildung 3.5 veranschaulicht.

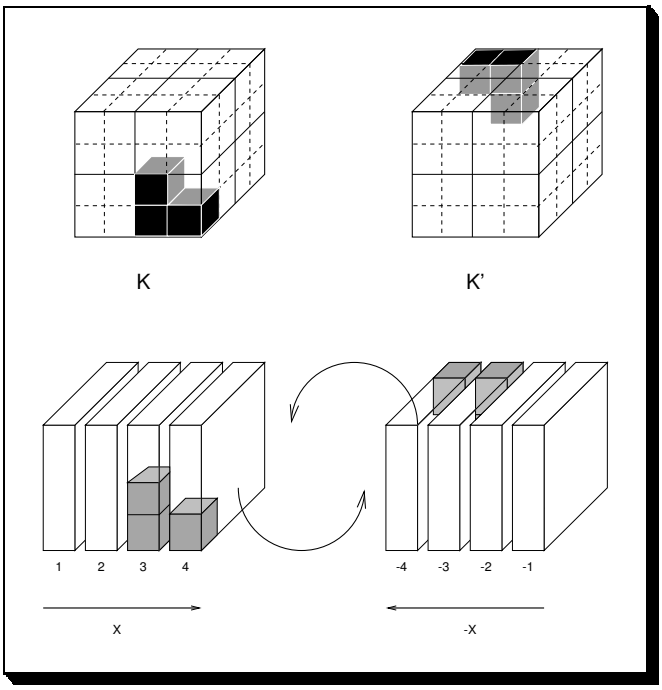


Abbildung 3.5: Spiegeln am Ursprung mit Bitblöcken

3.3.6.2.1 Schieben von Bitblöcken

Die zentrale Rolle spielen die beiden Routinen sh_in und sh_out, die ein Verschieben längs der drei Achsen in beide Richtungen entsprechend dem geometrischen Modell ermöglichen.

Die Routine sh_in mit den Parametern wtrg für die Schiebeachse, dtrg für die Schieberichtung und btrg für den Bitblock auf den die Operation ausgeführt werden soll, sowie c mit der Komponente die in den Bitblock hereingeschoben wird. Den Teil, der herausgeschoben wird, liefert sh_in als Funktionsergebnis zurück.

```
unsigned int sh_in(int wtrg, int dtrg, int btrg, unsigned int c)
{
  unsigned int j, l, ch, xh, x=0;
  switch (wtrg) {                    /* TARGET */
     case X: if (dtrg > 0) {  x = BB(btrg, 3);
                              BB(btrg, 3) = BB(btrg, 2);
                              BB(btrg, 2) = BB(btrg, 1);
                              BB(btrg, 1) = BB(btrg, 0);
                              BB(btrg, 0) = c;
             }
             else {   x = BB(btrg, 0);
                      BB(btrg, 0) = BB(btrg, 1);
                      BB(btrg, 1) = BB(btrg, 2);
                      BB(btrg, 2) = BB(btrg, 3);
                      BB(btrg, 3) = c;
             }
             break;
     case Y: ch=0;
             if (dtrg > 0)
                  for (j = 0; j < 4; j++){
                      ch = c & 0x000F;
                      c >>= 4;   x >>= 4;
                      xh = BB(btrg, j) & 0xF000;
                      x |= xh;
                      BB(btrg, j) <<= 4;
                      BB(btrg, j) |= ch;
                  }
             else for (j = 3; j > -1; j--){
                      ch = c & 0xF000;
                      c <<= 4;   x <<= 4;
                      xh = BB(btrg, j) & 0x000F;
                      x |= xh;
                      BB(btrg, j) >>= 4;
                      BB(btrg, j) |= ch;
                  }
             break;
     case Z: if (dtrg > 0)
                  for (j = 0; j < 4; j++){
                      for (l = 0; l < 4; l++)
                          bit_write(l*4+j, bit_read(l*4+3, BB(btrg, j)), &x);
                      BB(btrg, j) <<= 1;
                      for (l = 0; l < 4; l++)
                          bit_write(l*4, bit_read(l*4+j, c), &BB(btrg, j));
                  }
             else for (j = 3; j > -1; j--){
                      for (l = 0; l < 4; l++)
                          bit_write(l*4+j, bit_read(l*4, BB(btrg, j)), &x);
                      BB(btrg, j) >>= 1;
                      for (l = 3; l > -1; l--)
                          bit_write(l*4+3, bit_read(l*4+j, c), &BB(btrg, j));
                  }
             break;
     }
  return x;
}
```

sh_in gliedert sich in drei den Raumachsen entsprechende Blöcke, in denen für die jeweilige Richtung weiterverfahren wird. Der funktionale Teil besteht dann aus dem eigentlichen Verschieben: Für den Fall der X–Achse wird ein einfacher Tausch im Bitblock Array durchgeführt, da hier die Elemente genau den YZ–Ebenen entsprechen. Für die Y–Achse werden die entsprechenden Bitfolgen, immer vier hintereinander, durch Maskieren, Bit-Schieben und logischen Operationen verschoben; bei der Z–Achse wird auf die Routinen bit_write und bit_read zurückgegriffen, da hier einzelne Bits ausgewählt werden müssen (vgl. 2.1.4.1).

Im Prinzip ist sh_out identisch zu sh_in aufgebaut, nur das hier der Parameter für c fehlt. Diese Operation vollzieht ein Rollen längs der angegebenen Achse wsrc und Richtung dsrc mit dem Bitblock bsrc und liefert den Übertrag als Ergebnis zurück.

Die von den Bitblöcken unterstüzten Transformationen, die nur aus Schiebeoperationen und Rotationen (vgl. 2.2.3) bestehen, rufen die Funktion sh unter Angabe des Start– (bsrc) und Ziel–Bitblockes (btrg), sowie der Achsen (wsrc, wtrg) und Richtungen (dsrc, dtrg) und mit der Angabe ob die auszutauschende Komponente gedreht (Parameter r) werden soll auf.

sh leitet im wesentlichen die angegebenen Parameter an die Routinen sh_out und sh_in weiter, wobei über den Parameter r entschieden wird, ob zusätzlich rotatebit (eine Funktion die eine Bitfolge umdreht) aufgerufen werden soll. Die Aufrufe von sh_out und sh_in erfolgen viermal, um alle vier Ebenen eines Bitblocks zu transformieren.

```
void sh(int wsrc, int dsrc, int bsrc,
        int wtrg, int dtrg, int btrg,
        int r)
{ unsigned int i, c;
  for (i = 0; i < 4; i++){
      c = sh_out(wsrc, dsrc, bsrc);
      if (r) c = rotatebit(c);
      sh_in(wtrg, dtrg, btrg, c);
  }
}
```

3.3.6.2.2 Die Funktion rotatebit

Den Abschluß in bezug auf strukturkonforme Transformationen soll hier die Funktion rotatebit bilden. Wie schon im Theorieteil (siehe 2.2.3) erwähnt gab es hier Mißverständnisse bezüglich der korrekten Realisierung. Die letztendlich realisierte Version führt eine Umkehrung der Bitfolge durch. Die Realisierung der Bitvektoren mittels Integerzahlen erlaubt es nicht, die Bitfolge unter zur Hilfenahme einer „einfachen“ C–Funktion umzukehren, so daß hier wieder auf Maskierungs– und Bitschiebeoperationen zurückgegriffen werden muß.

```
unsigned int rotatebit(unsigned int c)
{
  unsigned long ct, ch;
  ch=ct=0l;
  ct = c & 0x0001; ct <<= 15; ch |= ct;
   .
   .  /*  14 weiter Maskierungs- und Schiebeoperationen  */
   .
  ct = c & 0x8000; ct >>= 15; ch |= ct;
  return (unsigned int) ch;
}
```

3.3.6.2.3 Mengenoperationen mit Bitblöcken

Mengentheoretische Operationen auf Bitblockniveau (vgl. 2.2.3) werden durch das Zusammenschalten von logischen Operationen gebildet, beispielsweise die Bestimmung der Schnittmenge zweier Modelle durch einfaches logisches UNDieren dieser. Zu diesem Zweck sind in dem C–Programm als Low–Level Routinen die Logischen Operationen NOT, AND und OR auf Bitblöcke realisiert. Der Aufruf der Routinen, in nachfolgenden Listings aufgeführt, erfolgt durch Angabe des Zielbitblocks und der zwei zu verknüpfenden Bitblöcke bei bb_and und bb_or, bzw. des zu negierenden Bitblocks bei bb_not.

```
void bb_and(int bb1, int bb2, int dest)
{
   BB(dest,0) = BB(bb1,0) & BB(bb2,0);
   BB(dest,1) = BB(bb1,1) & BB(bb2,1);
   BB(dest,2) = BB(bb1,2) & BB(bb2,2);
   BB(dest,3) = BB(bb1,3) & BB(bb2,3);
}

void bb_or(int bb1, int bb2, int dest)
{
   BB(dest,0) = BB(bb1,0) | BB(bb2,0);
   BB(dest,1) = BB(bb1,1) | BB(bb2,1);
   BB(dest,2) = BB(bb1,2) | BB(bb2,2);
   BB(dest,3) = BB(bb1,3) | BB(bb2,3);
}

void bb_not(int src, int dest)
{
   BB(dest,0) = ~(BB(src,0));
   BB(dest,1) = ~(BB(src,1));
   BB(dest,2) = ~(BB(src,2));
   BB(dest,3) = ~(BB(src,3));
}
```

3.4 High–Level–Operationen

Zum Abschluß der Betrachtung der C–Implementierung wollen wir einige der High–Level–Operationen vorstellen. Dabei ist das Ziel nicht eine komplette Diskussion aller implementierten Operationen dieser Ebene — dies würde über den Rahmen dieser Abhandlung hinausgehen —, sondern vielmehr eine beispielhafte Erläuterung der Benutzungsweise des in den beiden vorhergehenden Abschnitten beschriebenen „ADT Octree“. Eine weiterführende Erklärung der Transformations– und Verknüpfungsoperationen findet sich in [1], allerdings sind die dort vorgestellten Algorithmen nur grob umrissen und stellenweise fehlerhaft, so daß sie sich nicht ohne weitere Adaption für eine Implementierung eignen.

3.4.1 Transformationen am Beispiel der Verschiebung

Betrachten wir nun aus der Gruppe der Transformationsoperationen die Verschiebung als beispielhafte Anwendung des „ADT Octree“.

Bei der Transformation „Verschieben“ wird der als Modell 0 gespeicherte Octree um t Einheiten entlang der Achse w ($w \in \{x, y, z\}$) in Richtung st ($st = \{-1, 1\}$) verschoben in Modell 2 geschrieben.

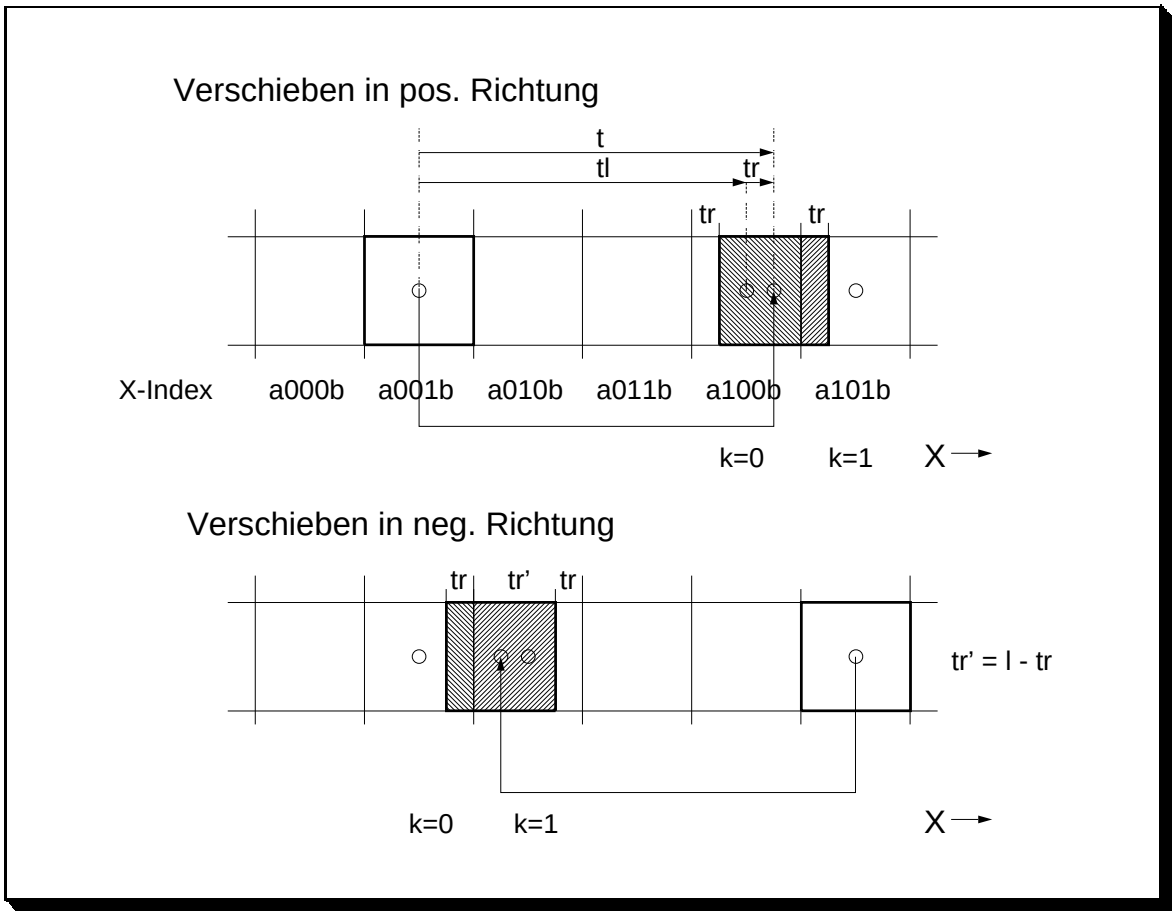


Abbildung 3.6: Verschieben eines Kubus

3.4.1.1 Aufruf der Transformation

Die Transformation wird mit move gestartet. move schreibt die Paramter w, t und st in die globale Matrix transform_matrix und startet eine Baumtraversierung, die auf jeden mit „voll“ markierten Knoten[19] die Prozedur move_model anwendet, auf die Bitblöcke die Prozedur move_model_b.

```
1  void move(int w, int t, int st)
2  {
3     transform_matrix[0][X]=w;
4     transform_matrix[0][Y]=t;
5     transform_matrix[0][Z]=st;
6     root(0);
7     parse_tree_bq(0, FULL, move_model, move_model_b);
8  }
```

3.4.1.2 Abbildung eines vollen Kubus

Die Funktionsweise der Prozedur move_model läßt sich am besten anhand eines anschaulichen Beispiels darlegen. Betrachen wir Abb. 3.6: Ein Kubus vom Niveau $n = 5$ (Kantenlänge $l = 32$) soll um $t = 104$ Einheiten entlang der Achse $w = X$ verschoben werden. Der Zielkubus liegt nun nicht mehr im Raster, er muß also aus kleineren Kubi zusammengesetzt werden.

Zunächst wird der Anteil t_l von t ermittelt, der ein Vielfaches der aktuellen Kantenlänge ist (dieser Anteil entspricht der Verschiebung um ganze Rasterkubi) und der Restanteil t_r. Dieses geschieht, indem die Binärdarstellung von t rechts von der Stelle[20] t_n aufgesplittet wird. Für t_l werden alle Stellen rechts von t_n auf 0 gesetzt, für t_r alle Stellen links von t_{n-1} (Zeile 8).

Im Beispiel:

$$t = 104_{\text{dez}} = 11\,\big|\,01000_{\text{bin}}$$

$$t_l = 11\,\big|\,00000_{\text{bin}} = 96_{\text{dez}}$$

$$t_r = 00\,\big|\,01000_{\text{bin}} = 8_{\text{dez}}$$

Mit diesen Werten können die beiden Rasterkubi ermittelt werden, in denen der Zielkubus liegt. Der Quotient[21] $\frac{t_l}{l} = t_l \gg n$ ergibt die Anzahl der Rasterkubi vom Niveau n, um die der Ausgangskubus verschoben werden soll. Dieser Wert muß um n Stellen nach links geschoben (da die Niveaus $n-1, \ldots, 0$ nicht verändert werden sollen), mit st multipliziert und zur w-Koordinate des vektoriellen Index des Ausgangskubus addiert werden. Insgesamt erhält man: $st \cdot \frac{t_l}{l} \ll n = st \cdot t_l$ (Zeilen 9 und 10). Der Zielkubus liegt in dem Kubus mit dem errechneten Index und dessen Nachbar entlang der positiven w-Achse, im folgenden der Einfachheit halber „rechter“ Nachbar genannt[22].

$$st \cdot t_l = 01100000$$

n	109876543210
$\vec{P}(0)_w$	..a001b....
+	00001100000
$\vec{P}(2)_w$	..a100b....

(Hier interessieren die den Niveaus 0 ... 4 und 8 ... 10 entsprechenden Stellen des Index nicht — sie ändern sich nicht.)

Soll der Kubus um ein Vielfaches seiner Kubuslänge verschoben werden — in diesem Fall ist der Anteil t_r gleich 0 —, so liegt der Zielkubus genau im Kubusraster und kann einfach mittels set_node auf voll gesetzt werden (Zeilen 11 und 12). Andernfalls muß er aus kleineren Kubi zusammengesetzt werden.

[19] *nicht* auf jeden vollen Kubus, da die vollen Kubi der Niveaus 0 und 1 gesondert behandelt werden!
[20] sei $t = (t_N t_{N-1} \ldots t_n t_{n-1} \ldots t_1 t_0)$ die Binärdarstellung von t
[21] $\ll$ und $\gg$ sind wie in C als Schiebeoperatoren zu verstehen
[22] analog dazu „linker“ Nachbar

Soll der Kubus in negativer Richtung verschoben werden, so werden zunächst die gleichen Ausgangsbedingungen für move_compose geschaffen wie bei der Verschiebung in pos. Richtung. (→ Abb. 3.6): Statt des bisher ermittelten Rasterkubus wird dessen „linker“ Nachbar gewählt und statt t_r rechnen wir mit $l-t_r$ weiter, da t_r in diesem Fall den *nicht* zu füllenden Anteil des rechten Rasterkubus (siehe unten) angibt (Zeilen 14–16).

Nach diesen Vorarbeiten kann das Zusammensetzen der Kubi auf niedrigerem Niveau stattfinden. Dazu dient die Prozedur move_compose, die mit dem Parameter $k = 0$ für den „linken“ Rasterkubus, $k = 1$ für den „rechten“ Rasterkubus aufgerufen wird, nachdem die berechneten Rasterkubi mit search_node aufgesucht wurden (Zeilen 18–23).

```
1  void move_model()
2  {
3    int tl, tr, w, st, t;
4    Vector3D pp2, pp3;
5    w  = transform_matrix[0][X];
6    t  = transform_matrix[0][Y];
7    st = transform_matrix[0][Z];
8    leftright(niveau(0), t, &tl, &tr);
9    Vector3Cpy(INDEXV(0), pp2);
10   pp2[w] += st * tl;
11   if ( tr != 0 )
12     set_node (niveau(0), pp2);
13   else {
14     if (st<0) {
15         pp2[w] -= 1 << niveau(0);
16         tr = cubelength(0) - tr;
17     }
18     Vector3Cpy(pp2, pp3);
19     pp3[w] += 1  << niveau(0);
20     search_node(niveau(0), pp3);
21     move_compose(1, tr, w);
22     search_node(niveau(0), pp2);
23     move_compose(0, tr, w);
24   }
25 }
```

3.4.1.3 Zusammensetzen des Zielkubus

Die Prozedur move_compose dient dazu, den Zielkubus aus kleineren Kubi zusammenzusetzen. Dabei wird über den Parameter k festgelegt, welcher Teil des Zielkubus komponiert werden soll: Im Fall $k = 0$ werden t_r Einheiten an der „rechten“ Seite[23] des aktuellen Kubus gefüllt, im Fall $k = 1$ wird genau dieser Teil nicht gefüllt, wohl aber der Rest.

In der Prozedur werden die Subkubi des aktuellen Kubus betrachtet. Die durchzuführende Aktion (Kubus auf „voll“ bzw. „leer“ setzen, rekursiv weiterarbeiten) hängt neben k von drei weiteren Faktoren ab:

- Befindet sich der betrachtete Subkubus „rechts“ oder „links“ vom Mittelpunkt des übergeordeten Kubus[24]? Dies kann anhand der w-Koordinate der vektoriellen Position des betrachteten Subkubus getestet werden (Zeile 11).
- Liegt die durch t_r definierte Schnittgerade — die Grenze zwischen den zu setzenden und den nicht zu setzenden Kubi — „rechts“ oder „links“ vom Mittelpunkt des übergeordeten Kubus? Dies wird anhand der Binärstelle[25] tr_n geprüft. (Zeile 10).
- Die Binärstellen rechts von tr_n (τ, siehe Fußnote 25) entscheiden zuletzt, ob der betrachtete Subkubus rekursiv weiterbearbeitet werden muß (Zeile 9).

Ein Übersicht der zu unterscheidenden Fälle ist Tabelle 3.4 zu entnehmen.

[23] genauer: der Seite die in pos. w–Richtung liegt
[24] also des beim Aufruf von move_compose aktuellen Kubus
[25] Sei $tr_N tr_{N-1} \ldots tr_n \underbrace{tr_{n-1} \ldots tr_1 tr_0}_{\tau}$ die Binärdarstellung von t_r.

Pos. rel. zum Kubusmittelpkt.	Pos. der Schnittgeraden	τ	k	Aktion
Links	Links	$= 0$	$= 0$	kubus := EMPTY
		$= 0$	$= 1$	kubus := FULL
		$\neq 0$	bel.	Rek. Aufruf
	Rechts	bel.	$= 0$	kubus := EMPTY
		bel.	$= 1$	kubus := FULL
Rechts	Links	bel.	$= 0$	kubus := FULL
		bel.	$= 1$	kubus := EMPTY
	Rechts	$= 0$	$= 0$	kubus := FULL
		$= 0$	$= 1$	kubus := EMPTY
		$\neq 0$	bel.	Rek. Aufruf

Tabelle 3.4: Fallunterscheidungen bei move_compose

Mit diesen Erkenntnissen läßt sich move_compose gegenüber der in [1] vorgestellten Prozedur vereinfachen:

```
1  void move_compose(int k, int tr, int w)
2  {
3    int i, tau, trn, pos;
4    static int cont[2][2]={{EMPTY,FULL},{FULL,EMPTY}};
5    if (node(2) != SPLIT) attach(2);
6    down (2,0);
7    for (i=0; i<TREETYPE; i++){
8      if (node(2) != FULL){   /* wichtig, sonst werden volle Kubi geteilt ! */
9         tau = right(niveau(2), tr);
10        trn = digit(niveau(2), tr);
11        pos = oct_to_vec[pos_oct(2)][w];
12        if ((pos == trn) && tau) move_compose(k, tr, w);
13        else node_set(2, cont[pos][k]);
14     }
15     next(2);
16   }
17   up(2);
18 }
```

3.4.1.4 Abbildung eines Bitblocks

Trifft die Prozedur parse_tree bei der Verschiebeoperation auf einen Bitblock, so wird eine spezielle Prozedur zum Verschieben von Bitblöcken aktiviert. Diese beruht im wesentlichen auf der in 3.3.6.2 beschriebenen Bitblockschiebeoperation SH_IN[26]. Die Ausgangssituation ist in Abb. 3.7 gezeigt.

Zunächst müssen analog zu move_model die zwei Rasterkubi ermittelt werden, in denen der Zielbitblock liegen wird. Die Indizes dieser Kubi seien $pp2$ und $pp3$. Von dem zu schiebenden Bitblock in Baum 0 wird eine Kopie angefertigt und in $BB(3)$ abgelegt (Zeile 17).

In Zeile 18 und 19 erfolgt das eigentliche Verschieben des Bitblocks: $BB(3)$ und $BB(2)$ werden, wie in Abb. 3.8(2) skizziert, gekoppelt verschoben: in $BB(3)$ werden entlang der w-Achse Nullen hereingeschoben. Die herausgeschobenen Werte werden in $BB(2)$ in gleicher Richtung hineingeschoben. Nach der Schiebeoperation enthält $BB(2)$ den korrekten Inhalt des Bitblocks an der Stelle $pp3$.

$BB(3)$ kann nicht einfach an die Stelle $pp2$ geschrieben werden, da dieser Bitblock — wie im Beispiel — evtl. bereits zum Teil mit Kubi vorangegangener Verschiebungen gefüllt ist (→ Abb. 3.8(1)).

[26] SH_IN(w, d, bb, c) schiebt den Bitblock bb entlang der Achse w in Richtung d. Dabei wird c hineingeschoben. Das Resultat ist der herausgeschobene Wert

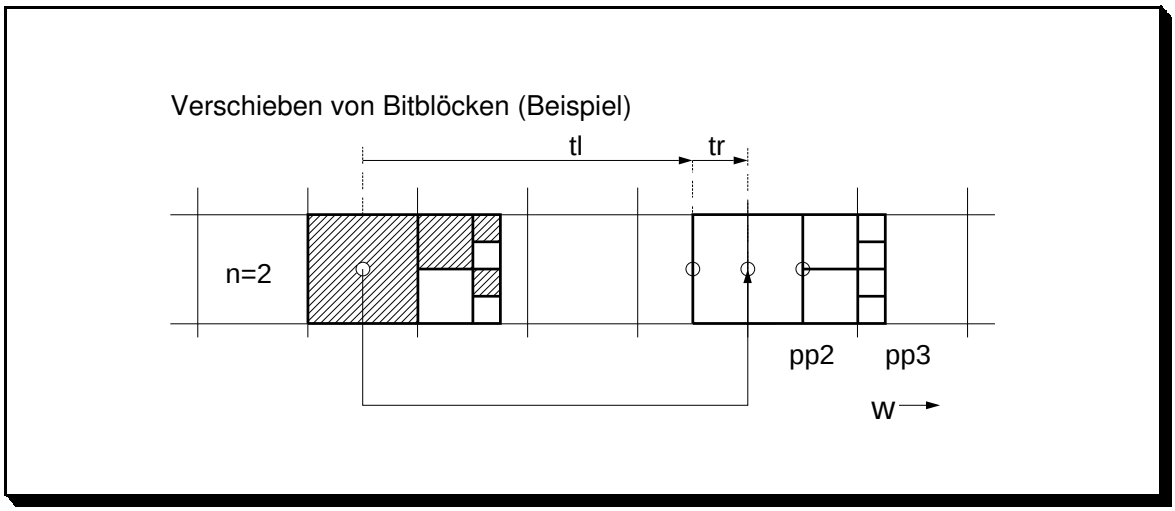


Abbildung 3.7: Verschieben eines Bitblockes

Daher muß der Bitblock mit Index $pp2$ im Baum 2 aufgesucht und mit dem Inhalt vom $BB(3)$ mittels ODER verknüpft werden (Zeilen 20–22).

Der umgekehrte Fall — $pp3$ enthält bereits auf „voll“ gesetzte Kubi — kann nicht auftreten, da der Ausgangsbaum bei der Traversierung von „links“ nach „rechts“ — also in positiver Richtung entlang der w-Achse — durchlaufen wird.

```
1  void move_model_b()
2  {
3     int tl, tr, w, st, t, i;
4     Vector3D pp2, pp3;
5     w  = transform_matrix[0][X];
6     t  = transform_matrix[0][Y];
7     st = transform_matrix[0][Z];
8     leftright(niveau(0), t, &tl, &tr);
9     Vector3Cpy(INDEXV(0), pp2);
10    pp2[w] += st * tl;
11    if (st<0) { pp2[w] -= cubelength(0);
12                tr = 4 - tr; }
13    Vector3Cpy(pp2, pp3);
14    pp3[w] += cubelength(0);
15    search_node(1, pp3);
16    attach(2);
17    copy_bb(0, 3);
18    for (i=0; i<tr; i++)
19      sh_in(w, 1, 2, sh_in(w, 1, 3, 0));
20    search_node(1, pp2);
21    if (node_get(2) != SPLIT) attach(2);
22    bb_or(3, 2, 2);
23  }
24 }
```

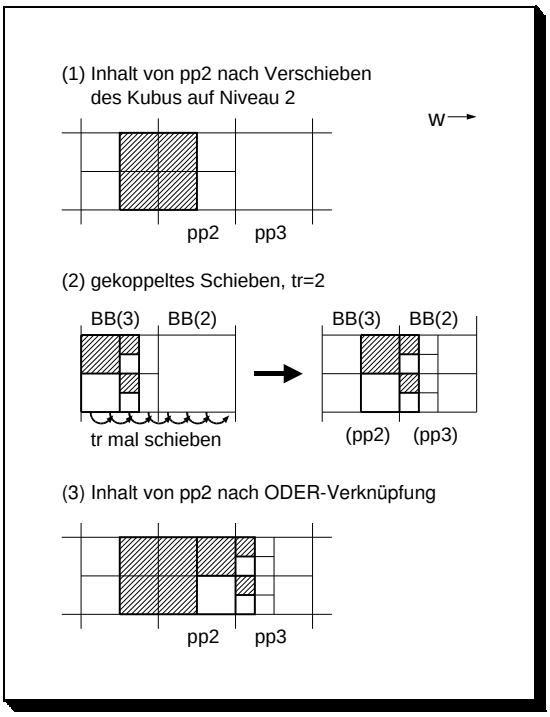


Abbildung 3.8: Verschieben von Bitblöcken

Transformation bzw. opcode	Beschreibung der Transformation	Transformationsmatrix	Form[27] des sh–Aufrufs
00	Spiegeln am Ursprung	$\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$	sh(X, 1, 0, X, -1, 2, 1)
01	Spiegeln an der xy–Ebene	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$	sh(Z, 1, 0, Z, -1, 2, 0)
02	Spiegeln an der zx–Ebene	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	sh(Y, 1, 0, Y, -1, 2, 0)
03	Spiegeln an der YZ–Ebene	$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	sh(X, 1, 0, X, -1, 2, 0)
04	Klappen $\pi/2$ um x–Achse	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$	sh(Y, 1, 0, Z, -1, 2, 0)
05	Klappen π um x–Achse	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$	sh(X, 1, 0, X, 1, 2, 1)
06	Klappen $-\pi/2$ um x–Achse	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}$	sh(Z, 1, 0, Y, -1, 2, 0)
07	Klappen $\pi/2$ um y–Achse	$\begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$	sh(X, 1, 0, Z, -1, 2, 0)
08	Klappen π um y–Achse	$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$	sh(Y, 1, 0, Y, 1, 2, 1)
09	Klappen $-\pi/2$ um y–Achse	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix}$	sh(Z, 1, 0, X, -1, 2, 0)
10	Klappen $\pi/2$ um z–Achse	$\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	sh(Y, 1, 0, X, -1, 2, 0)
11	Klappen π um z–Achse	$\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	sh(Z, 1, 0, Z, 1, 2, 1)
12	Klappen $-\pi/2$ um z–Achse	$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	sh(X, 1, 0, Y, -1, 2, 0)

Tabelle 3.5: strukturkonforme Transformationen

[27] sh erwartet die Parameter in der Reihenfolge (Schiebeachse des Ausgangsmodells, Achsrichtung des Ausgangsmodells, Ausgangsmodell, Schiebeachse des Zielmodells, Achsrichtung des Zielmodells, Zielmodell, Rotation des Übertrags?)

3.4.2 Spiegeln und Klappen

Um Spiegeln und Klappen (sogenannte strukturkonforme Transformationen, vgl. 2.2.2) mit den Modellen zu ermöglichen wurde in C die Funktion reflect realisiert. Diese Funktion ermöglicht eine genauere und effizientere Umsetzung der Transformationen als es durch die bisher gegebenen Transformationsfunktionen (turn und move) möglich ist. So kann das Klappen eines Modells um $\pi/2$ an der Y–Achse auch eine Rotation von 90 Grad um die Y–Achse mittels turn durchgeführt werden, dies ergibt aber eine ungewollte Expansion des Modells, da einige Kubi aufgrund von Rundungsfehlern auf SPLIT gesetzt werden, obwohl das Setzen eines Kubis auf höherem Niveau ausreichen würde. Führt man diese Transformation mittels einer Transformationsmatrix und dem Mittelpunktsvektor zu einem Kubus durch, so ist das Ergebnis auch wieder ein dem Kubusraster entsprechender Mittelpunkt und es ergeben sich keine unnötigen SPLIT–Knoten. Auch werden diese Transformationen aufgrund ihrer Strukturkonformität effizent von Bitblöcken unterstützt (vgl. 2.2.3).

Die Funktion reflect, die das Modell 0 entsprechend Tabelle 3.5 nach Modell 2 transferiert, gliedert sich in zwei Teile. Im ersten Abschnitt wird nach einem root–Aufruf im Ausgangsmodell entsprechend dem übergebenem Parameter opcode (siehe Tabelle 3.5) eine Transformationsmatrix generiert. Der zweite Teil initiiert dann über parse_tree einen Baumdurchlauf, der auf alle FULL gesetzten Knoten die Funktion reflect_model bzw. beim Auftreten eines Bitblocks reflect_model_b anwendet.

Nebenstehende Funktion reflect_model führt mit dem Mittelpunktsvektor des FULL–Knotens und der Transformationsmatrix eine Multiplikation durch (Zeile 3). Zu dem Ergebnis der Matrixmultiplikation, des Mittelpunktvektors des Zielkubus, wird der vektorielle Index bestimmt (Zeile 4) und mit diesem auf dem Niveau des Ausgangsknotens im Modell 2 ein Knoten gesetzt (Zeile 5).

```
0  void reflect_model()
1  {
2    Vector3D pv, pp;
3    Matrix33VectorMultiply...
  ...(transform_matrix, MIDDLE(0), pv);
4    point_to_index(pv, pp);
5    set_node(niveau(0), pp);
6  }
```

Die Funktion reflect_model_b bestimmt wie bei reflect_model über eine Matrixmultiplikation die Zielkoordinate der Transformation, allerdings wird hier kein Knoten gesetzt, sondern mittels attach ein Bitblock in den Baum eingefügt. Nun erfolgt der Aufruf der Funktion sh (vgl. 3.3.6.2), die mittels Schiebeoperationen den Ausgangsbitblock in den Zielbitblock überführt. Die Parameter mit denen sh aufgerufen wird, kann man der Tabelle 3.5 entnehmen.

3.4.3 Visualisierung

Zum Abschluß soll die Visualisierung eines Octree als eine der wichtigsten High–Level–Operation nicht unerwähnt bleiben. Aus dem Ensemble der in [1] vorgeschlagenen Methoden zur Visualisierung wurden nur die beiden vielversprechendsten implementiert — die Schnittebenendarstellung und das Drahtmodell —, allerdings wurden die dort vorgeschlagenen Algorithmen durch eigene ersetzt.

3.4.3.1 Schnittebene

Zur Schnittebenendarstellung eines Kubusmodells werden zwei Achsen u und v sowie eine Koordinate t definiert, die auf der übriggebliebenen Achse w liegt. Dargestellt wird die zur u/v–Ebene paralle Ebene, die die w–Achse an der Stelle t schneidet (→ Abb. 3.9).

Durch iterierte Schnittebenendarstellung — die Abtastebene u, v wird entlang der Achse w verschoben — erhält der Betrachter einen recht guten Einblick in das gespeicherte Kubusmodell, der durch geeignete Wahl des Abtastintervalls auf der w–Achse und der Distanz zwischen den einzelnen Abtaststellen t dahingehend optimiert werden kann, daß wenig Ebenen abgetastet werden, die vollständig außerhalb des Kubusmodells liegen oder in denen keine Änderung zur vorhergehenden Ebene stattgefunden hat.

Bei der Implementierung der Prozedur cross_section wurde ganz von der in [1] vorgeschlagenen Methode abgewichen: Anstatt für jeden Bildpunkt zu berechnen, ob sich an der entsprechenden Stelle

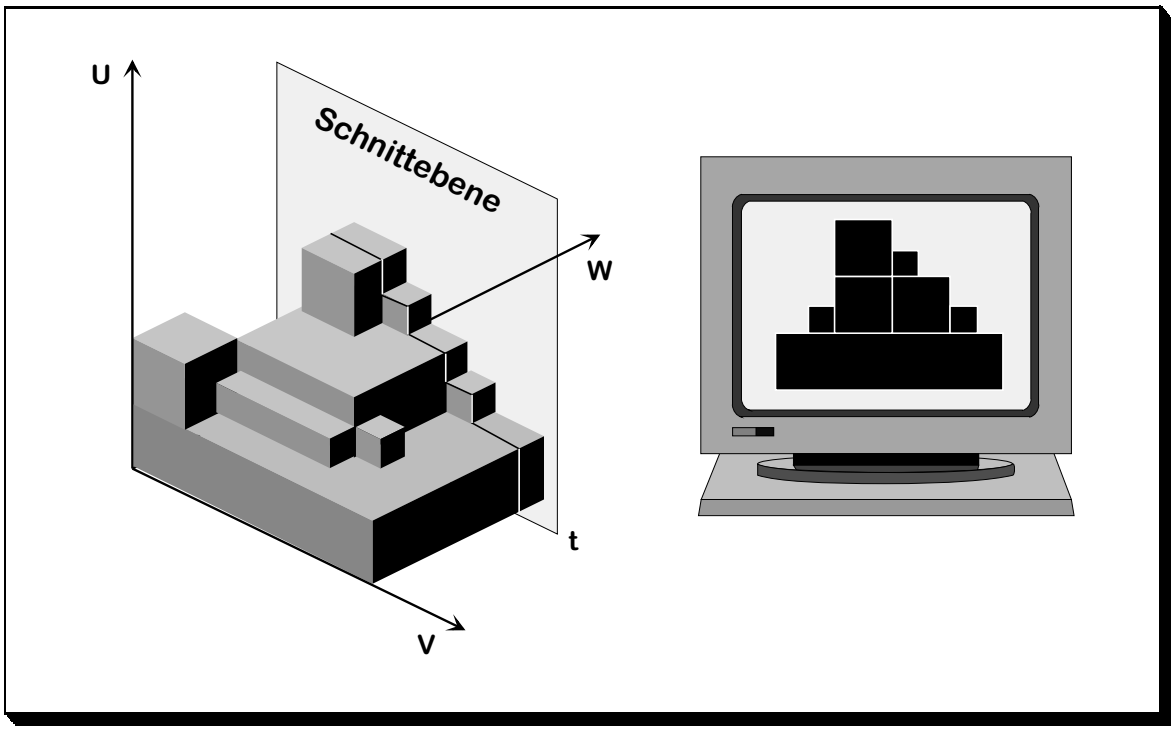


Abbildung 3.9: Schnittebenendarstellung

ein Kubus befindet, in dessen w–Koordinatenintervall t liegt, wird der Baum mittels parse_tree durchlaufen. Dabei wird bei jedem „vollen“ Knoten entschieden, ob er in der aktuellen Abtastebene liegt; ob also $t \in [m_w - \frac{l}{2}, m_w + \frac{l}{2}[$ gilt. Ist dies der Fall, wird ein Quadrat auf den Bildschirm gezeichnet, dessen Koordinaten sich aus der Kubuslänge sowie den u und v Komponenten des Mittelpunktes errechnen lassen[28]. Dadurch wurde eine erhebliche Beschleunigung der Visualisierung bei den Testläufen mit kleinen Kubusmodellen erreicht, allerdings steigt hier der Rechenaufwand mit der Anzahl der im Modell enthaltenen Kubi.

Aus Gründen der Übersichtlichkeit wollen wir die Implementierung dieser Operation nur stark verkürzt vorstellen, so wird u. a. auf die implementierte Zoomfunktion[29] und die plattformabhängigen Initialisierungen des Graphiksystems nicht eingegangen.

Die Prozedur cross_section erwartet den darzustellenden Baum q, die Vorzeichen der Achsen st, su und sv — hierdurch wird die Blickrichtung auf das Modell festgelegt — die Abtastebenen wu und wv sowie die Abtaststelle ds auf der dritten Achse.

```
1  void section_model2()
2  {
3     int q, m, cl, st, su, sv, wu, wv, wt, ds;
4     st = transform_matrix[0][X];
5     ...
6     cl = cubelength_2(q);
7     m = invf(st, MIDDLE(q)[wt]);
8     if (!niveau(q)){         /* Niveau 0 ist Spezialfall */
9           if (m==ds)                            /* Rechteck zeichnen: */
10             draw_rect(invf(su, MIDDLE(q)[wu]),  /* Mittelpunkt X      */
11                       invf(sv, MIDDLE(q)[wv]),  /* Mittelpunkt Y      */
```

[28] Das Quadrat wird in einer dem Niveau des Kubus zugeordneten Farbe gezeichnet.

[29] Mit den Tasten "0", "1", "2", "3" kann der entsprechende Quadrant der Darstellung vergrößert werden; mit "+" kehrt man zur Originalgröße zurück und mit "x" kann das Programm verlassen werden. Die XGraphik–Version ermöglicht dazu ein Zoomen durch Anklicken eines Quadranten.

```
12                      cubelength(q));          /* Kantenlaenge       */
13      }
14    else if ((m-cl <= ds) && (ds < m+cl))
15              draw_rect(invf(su, MIDDLE(q)[wu]),
16                        invf(sv, MIDDLE(q)[wv]),
17                        cubelength(q));
18 }
19
20 void cross_section(int q, int st, int su, int sv, int wu, int wv, int ds)
21 {
22    int i, xo, yo, sf, cl;
23    char c=1;
24    transform_matrix[0][X]=st;
25    ...
26    while(c){
27       parse_tree_q(q, FULL, section_model2);
28       switch( c = getch() ) {        /* Zoom */
29            ...
30       }
31       clear_graphic(0);
32    }
33 }
```

Einer näheren Erklärung bedarf hierbei nur die Funktion invf(s, n): sie negiert die Zahl n, falls s negativ ist[30]. Mit Hilfe dieser Funktion wird eine Transformation der Weltkoordinaten in das vom Benutzer spezifizierte Koordinatensystem vollzogen.

Basierend auf cross_section wurden drei weitere Prozeduren implementiert, die eine iterierte Anwendung des ebenen Schnittes durchführen. Alle drei Prozeduren wählen die X/Y–Ebene mit positiven Vorzeichen als Bildebene und durchlaufen die Z–Achse in positiver Richtung.

```
void display_tree(int q)
void display_layer(int q, ...)
void display_range(int q, int start, int end)
```

display_tree durchläuft den gesamten Raum mit logarithmischen Abständen zwischen den Abtastebenen (-512, -256, -128, -64 ...). Nur im Bereich $-8 \ldots 8$ wird an allen Stellen abgetastet.

display_layer zeigt eine beliebige Anzahl von Abtastebenen. Die drei Punkte (...) in der Funktionsdefiniton ist die Deklaration einer variablen Argumentliste (vgl. [41], Kap. 7.3 und Anhang B.7), die für einen korrekten Ablauf der Prozedur nur Zahlen vom Typ int enthalten darf und mit der Konstanten MAX_INT aus limits.h enden muß.

display_range stellt die Abtastebenen in einem linearen Bereich der Z–Achse dar.

[30] `if (s < 0) return -n; else return n;`

3.4.3.2 Drahtmodell

Bei der Darstellung als Drahtmodell wird die Oberfläche des Kubusmodells mit einem aus Dreiecken bestehenden Gitter überzogen. Ein derartiges Drahtmodell kann als Eingabe für Graphiksysteme wie Raytracer oder –shader[31] verwendet werden. Die implementierte Version erzeugt eine Eingabedatei, die für die Graphiksysteme Rayshade([13]) oder MING ([9], [43]) benutzt werden kann.

Die Grundidee der Drahtmodellerzeugung folgt dem in [1] vorgestellten Algorithmus zur Berechnung der Hülle: Bei jedem vollen Knoten betrachten wir in der Prozedur wire_proc jeden der an den 6 Seiten des gefundenen Kubus a anliegenden Kubi $b_i, i = 1 \ldots 6$ ($\rightarrow$ Abb. 3.10). Ist b_i leer, so müssen wir die Seitenfläche zwischen a und b_i in zwei Dreiecke aufteilen und diese ausgeben (wire_write), ist er voll, können wir ohne weitere Aktion fortfahren. Ist b_i dagegen geteilt, so müssen wir mit den an a anliegenden Subkubi von b_i rekursiv fortfahren (wire_split) ($\rightarrow$ Abb. 3.11).

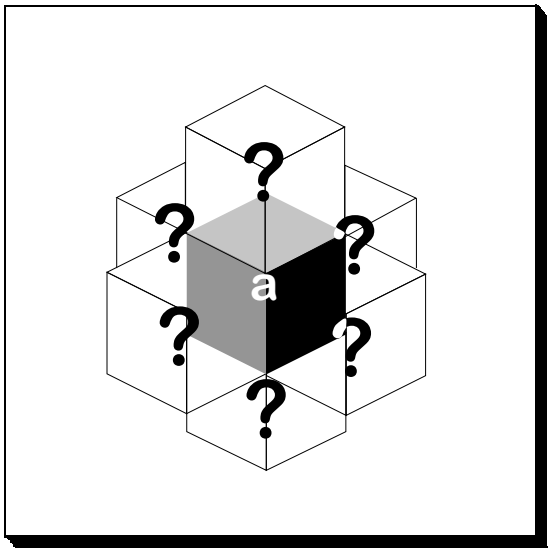


Abbildung 3.10: Benachbarte Kubi

wire_write berechnet aus Mittelpunkt m und Kubuslänge l die Koordinaten $w0, w1_0, w1_1, w2_0$ und $w2_1$ der Eckpunkte des aktuellen Kubus (vgl. Abb. 3.12 und Zeilen 6–17). Da bei den nachfolgenden Berechnungen die abstrakten Achsenbezeichungen w, $w_1 = (w + 1) \bmod 3$ und $w_2 = (w + 2) \bmod 3$ den konkreten Achsen X, Y und Z zugeordnet werden müssen, ist eine Fallunterscheidung anhand von w nötig.
Den drei Stützvektoren t_1, t_2 und t_3 werden in Abhängigkeit von der w–Achse die Koordinaten von drei Eckpunkten des auszugebenden Quadrates zugeordnet. Aus diesen werden zwei Richtungsvektoren berechnet und wiederum aus diesen ein Normalenvektor des Quadrates. Dieser wird vom MING–System zur Darstellung gouraudschattierter[32] Körper benötigt. Schließlich werden die beiden Dreiecke in eine Ausgabedatei geschrieben. Dabei hängt die Zuordnung der berechneten Koordinaten zu den ausgegebenen Koordinaten wiederum von der betrachteten Achse w ab.
Nebenbei bemerkt spielt der Parameter d in den Prozeduren wire_split und wire_write die gleiche Rolle wie der Parameter k in move_compose (vgl. 3.4.1.3): Anhand von d wird entschieden, welche der beiden Seiten der w–Achse — nennen wir sie analog zu der Terminologie im vorhergehenden Unterkapitel „rechte“ und „linke“ Seite, obwohl diese Bezeichung nur bei entsprechender Lage des Koordinatensystems korrekt ist — auszugeben ist. Im Fall $d = 0$ soll die „linke“ Seite ausgegeben werden, im Fall $d = 1$ die „rechte“.

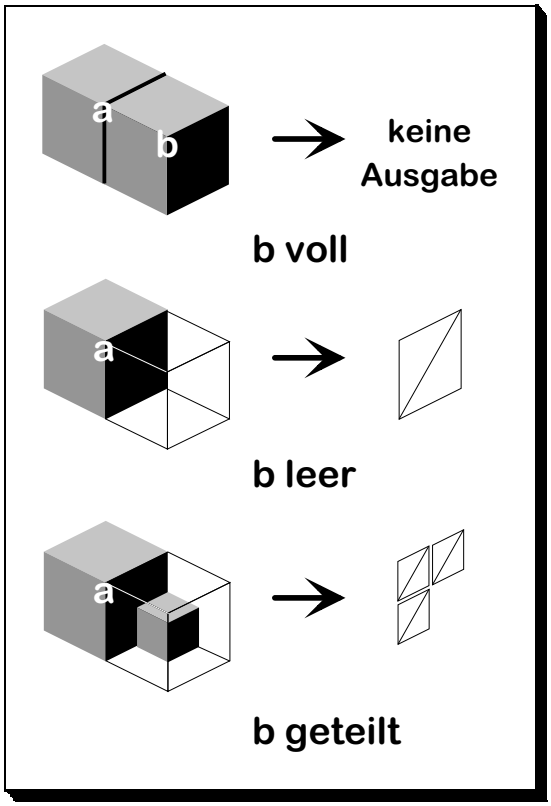


Abbildung 3.11: Drahtmodelle: Fallunterscheidung in wire_proc

[31] eine Übersicht verfügbarer Systeme findet sich in [12]
[32] Schattierungsalgorithmus von Gouraud, siehe [14], [9], [10], [11]

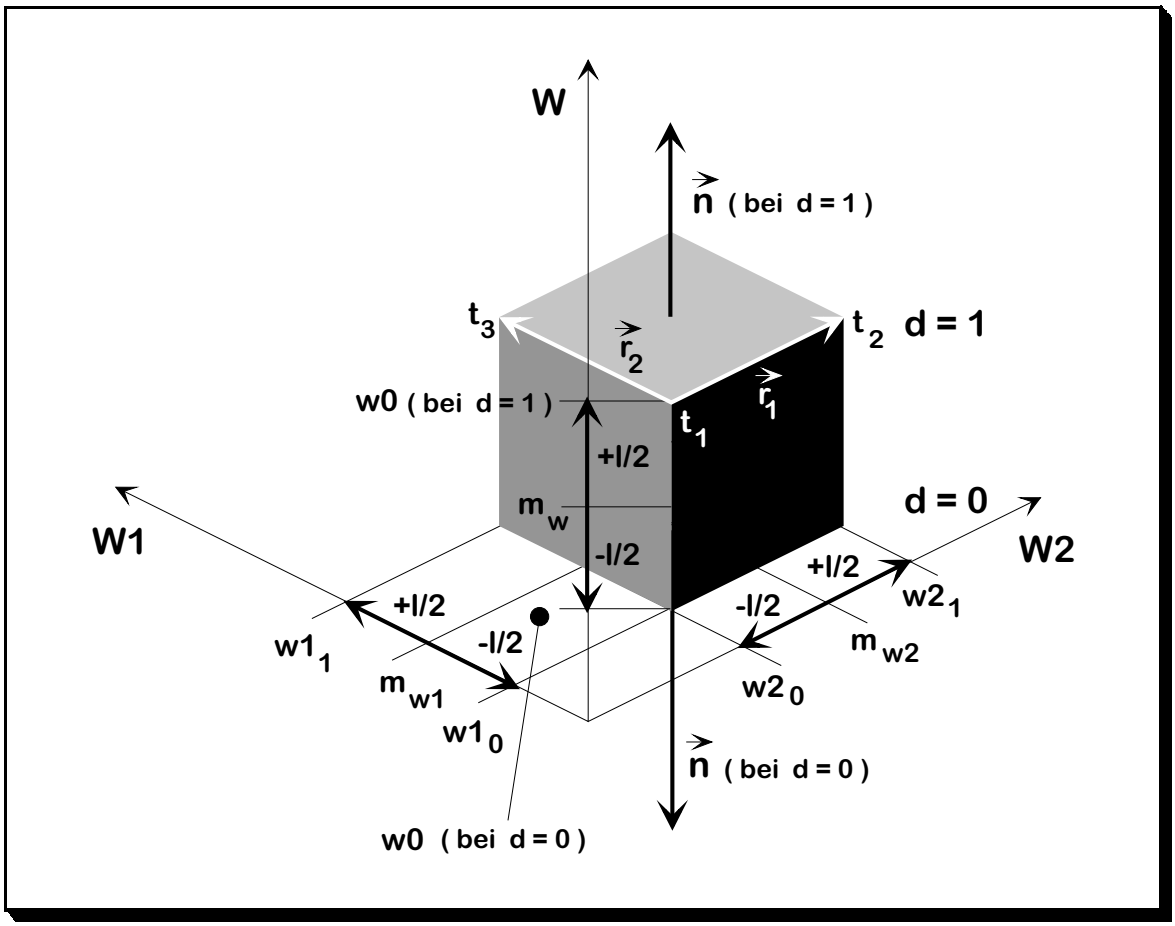


Abbildung 3.12: Triangulierung einer Kubusseitenfläche

```
1  void wire_write(int d, int w, int niv, Vector3D mi)
2  {
3     Vector3Dr t1, t2, r1, r2, n;
4     float cl2, wko, w1ko0, w1ko1, w2ko0, w2ko1, factor, m[3];
5     cl2 = (float)(1 << niv) / 2;
6     if (niv) {   /* Niveau 0 beruecksichtigen */
7       m[X]=(float)mi[X];     m[Y]=(float)mi[Y];     m[Z]=(float)mi[Z];
8     }
9     else {        /* Rasterkorrektur auf Niveau 0 */
10      m[X]=(float)mq[X]+0.5; m[Y]=(float)mq[Y]+0.5; m[Z]=(float)mq[Z]+0.5;
11    }
12    if (d) wko = (m[w] + cl2);
13      else wko = (m[w] - cl2);
14    w1ko0 = (m[(w+1)%3] - cl2);
15    w1ko1 = (m[(w+1)%3] + cl2);
16    w2ko0 = (m[(w+2)%3] - cl2);
17    w2ko1 = (m[(w+2)%3] + cl2);
18 /* w == X => X -> W    Y -> W1   Z -> W2  */
19    ...
20 /* w == Y => X -> W2   Y -> W    Z -> W1  */
21    ...
22    if (w == Z) {
23       t1[X] = w1ko0;    t1[Y] = w2ko0;    t1[Z] = wko;
24       t2[X] = w1ko0;    t2[Y] = w2ko1;    t2[Z] = wko;
```

```
25        Vector3rSubtract(t2, t1, r1);
26        t3[X] = w1ko1;    t3[Y] = w2ko0;    t3[Z] = wko;
27        Vector3rSubtract(t2, t1, r2);
28        if (d) CrossProductr(r2, r1, n);
29        else CrossProductr(r1, r2, n);
30        fprintf(fp, "triangle %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f\
31 %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f\n",
32        w1ko0, w2ko0, wko, n[X], n[Y], n[Z],
33        w1ko0, w2ko1, wko, n[X], n[Y], n[Z],
34        w1ko1, w2ko0, wko, n[X], n[Y], n[Z]);
35        fprintf(fp, "triangle %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f\
36 %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f %7.3f\n",
37        w1ko1, w2ko1, wko, n[X], n[Y], n[Z],
38        w1ko1, w2ko0, wko, n[X], n[Y], n[Z],
39        w1ko0, w2ko1, wko, n[X], n[Y], n[Z]);
40     }
41 }
42
43
44 void wire_split(int d, int w, int nq, Vector3D mq)
45 {
46    int i, q;
47    q = transform_matrix[0][0];
48    down(q, 0);
49    for(i = 0; i < TREETYPE; i++){
50       if (oct_to_vec[pos_oct(q)][w] != d)  /* "rechts" oder "links" ? */
51         switch (node(q)){
52            case EMPTY: wire_write(d, w, nq, mq); break;
53            case SPLIT: wire_split(d, w, nq, mq);
54         }
55       next(q);
56    }
57    up(q);
58 }
59
60
61 void wire_proc()
62 {
63    int niv, w, q;
64    Vector3D m, idx, idxsrc;
65    q = transform_matrix[0][0];
66    niv = niveau(q);
67    Vector3Cpy(MIDDLE(q), m);
68    Vector3Cpy(INDEXV(q), idx);     /* Kopie zur Berechnung des Zielindex */
69    Vector3Cpy(INDEXV(q), idxsrc);  /* Kopie zur Rueckkehr zum Ausgangskubus */
70    for(w = X; w <= Z; w++){        /* Alle drei Achsen durchlaufen */
71       find_node(q, niv, idxsrc)    /* Ausgangskubus aufsuchen */
72       idx[w] -= 1 << niv;          /* Kubus "links" */
73       goto_node(q, idx);           /* Nachbar aufsuchen */
74       switch (node(q)){            /* Fallunterscheidung, siehe Abbildung */
75          case EMPTY: wire_write(0, w, niv, m); break;
76          case SPLIT: wire_split(0, w, niv, m);
77       }
78       find_node(q, niv, idxsrc)    /* Zurueck zum Ausgangskubus */
79       idx[w] += 2 << niv;          /* Kubus "rechts" */
80       goto_node(q, idx);           /* Nachbar aufsuchen */
```

```
81        switch (node(q)){            /* Fallunterscheidung, siehe Abbildung */
82           case EMPTY: wire_write(1, w, niv, m); break;
83           case SPLIT: wire_split(1, w, niv, m);
84        }
85        idx[w] -= 1 << niv;          /* Index des Ausgangskubus */
86     }
87     find_node(q, nq, idxsrc);       /* Zurueck zum Ausgangskubus */
88  }
89
90
91  void wire_model(int q)
92  {
93     char name[L_tmpnam];
94     tmpnam(name);               /* temporaeren Dateinamen ermitteln */
95     fp = fopen(name, "wt");     /* Datei oeffnen */
96     transform_matrix[0][0] = q;
97     fputs("eyep 0 0 -100\n", fp);
98     ... (weitere Initialwerte fuer das Graphiksystem schreiben)
99     parse_tree_q(q, FULL, wire_proc);   /* Kubusbaum durchsuchen */
100    fclose(fp);                 /* Datei schliessen */
101 }
```

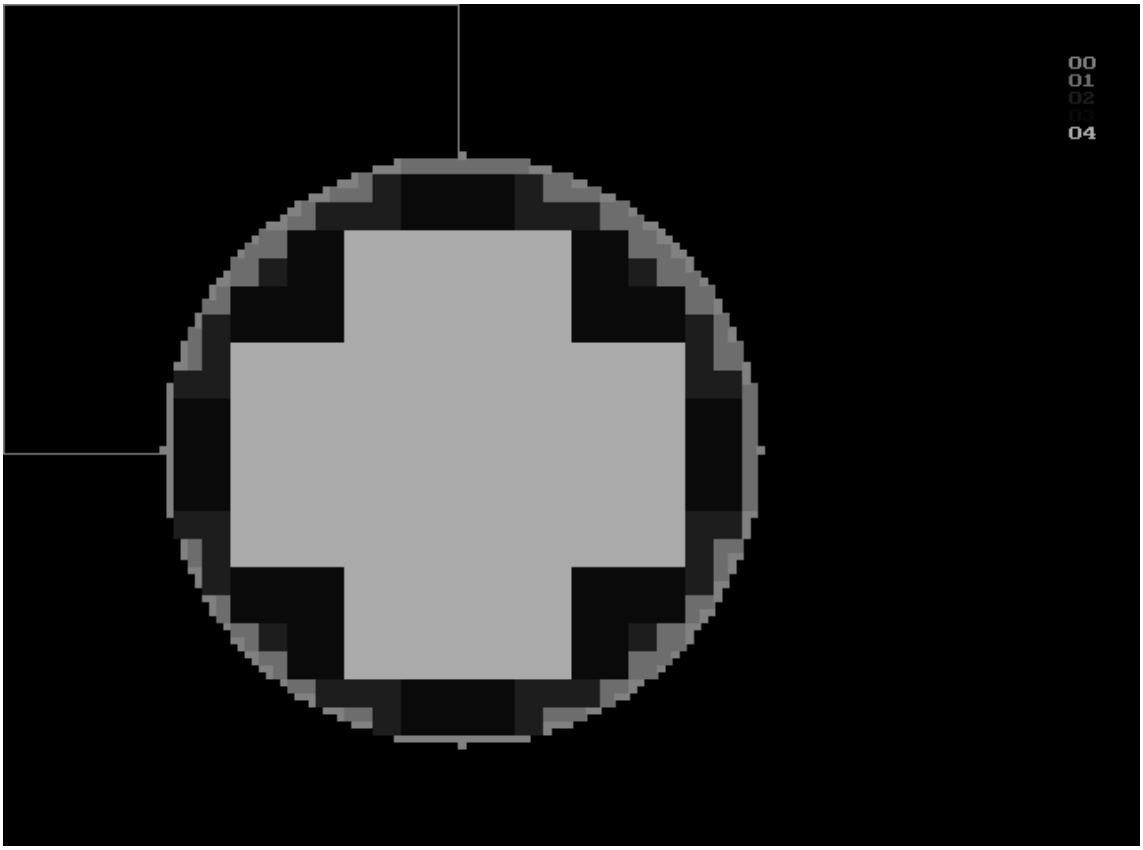


Abbildung 3.13: Visualierung unter Dos

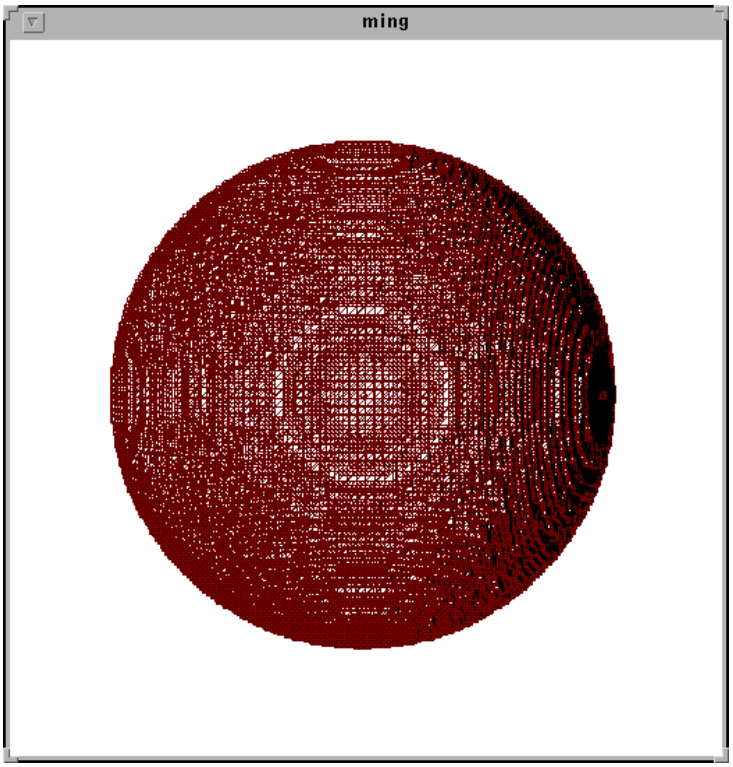


Abbildung 3.14: Drahtmodell in MING

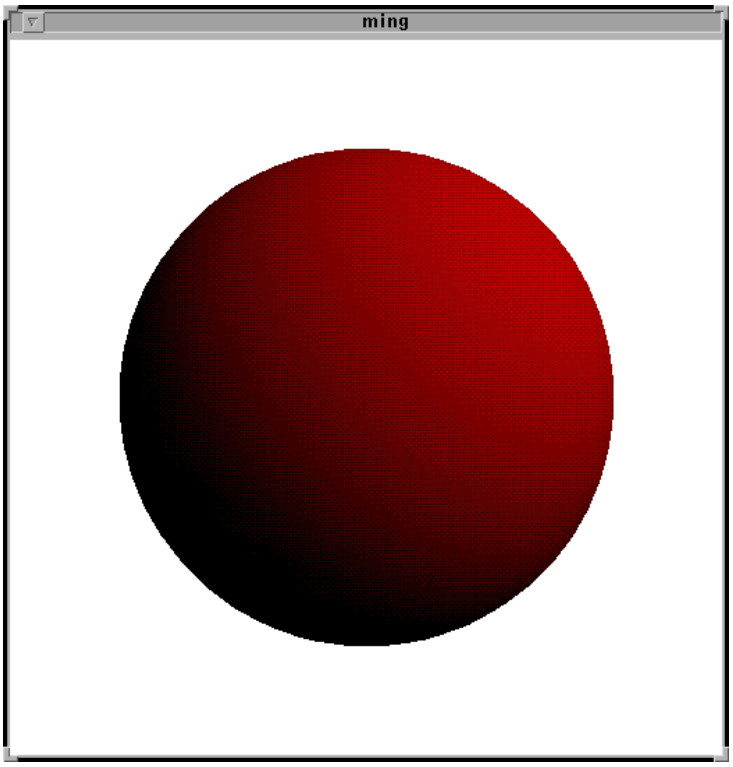


Abbildung 3.15: Schattiertes Modell in MING

3.5 Laufzeitmessungen

Bevor wir im nächsten Kapitel den Entwurf einer Hardware zur Unterstützung der hier dargelegten Operationen vorstellen, wollen wir einen Blick auf ein Laufzeitprofil der Software werfen und die Auswahl der Operation zur Hardwareimplementierung begründen. Allgemein läßt sich sagen, daß Operationen auf Octrees vergleichsweise zeitaufwendig sind, wenn man relativ einfache Modelle zugrunde legt und zum Vergleich „klassische“ Graphiksysteme heranzieht. So benötigte die Generierung einer Kugel mit einem relativ kleinem Radius ($r = 42$) auf einem 133 MHz Pentium etwa 3 sec, eine Ausgabe dieser Kugel als Drahtmodell etwa 45 sec.

Folgende Tabelle zeigt den Anteil der im Chip implementierten Routinen[33] und diesen Anteil zzgl. dem Anteil der parse–tree–Routinen an der Gesamtlaufzeit einer Operation. Zur Messung wurden die auch bei der Hardwaresimulation verwendeten Testmodelle (→ 5.4.1) herangezogen. Auf eine Visualisierung wurde verzichtet, da die Graphikausgabe einschließlich der Initialisierung des Graphiksystems bei der geringen Laufzeit der Testoperationen einen unverhältnismäßig hohen Anteil an der Gesamtlaufzeit in Anspruch nimmt. Zudem wurden die Operationen direkt vom Hauptprogramm aufgerufen, so daß die Laufzeit der High–Level–Operation als Gesamtlaufzeit angenommen werden kann.

Operation	Anteil der Chip–Operationen	Chip–Op. zzgl. parse–tree
Generierung eines Testbaumes mit einem Kubus pro Niveau (Setnode–Test)	32 %	—
Verschiebung des o.g. Baumes (Setmove–Test)	54 %	64 %
Vereinigung zweier Testmodelle (Unite–Test)	54 %	—
Erzeugen eines Testbaumes und		
– Spiegeln am Ursprung	35 %	42 %
– Verschieben	69 %	85 %
– Skalieren	62 %	96 %
– Drehen	80 %	99 %
– Verjüngen	72 %	97 %
Erzeugen einer Kugel mit $r = 42$ (Balltest)	86 %	—

Tabelle 3.6: Anteil der Low–Level–Operationen an High–Level–Operationen

Es ist zu sehen, daß die für eine Hardwareimplementierung vorgesehenen Operationen einen hohen Anteil der Rechenzeit von High–Level–Operationen beanspruchen. Die Auswahl der Operationen für die Hardwareunterstützung ist also gerechtfertigt.

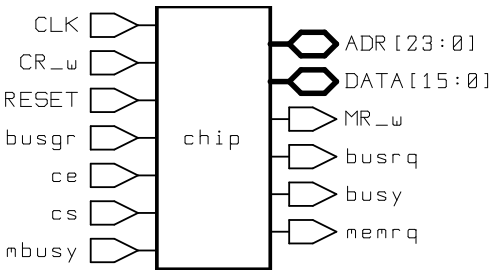
[33] Vgl. Anhang A.1. Die Auswahl dieser Operationen ist [1] entnommen.

Kapitel 4

Hardwareentwurf

In diesem Kapitel soll das eigentliche Ziel der Arbeit beschrieben werden: der Entwurf eines Chips zur Unterstützung octreebasierter Operationen, im folgenden kurz Octchip genannt. Wir werden die Entwurfseinheiten des Chips vorstellen und einige VHDL–Beschreibungen im Detail diskutieren[1]. Dabei soll der Entwurf im Großen und Ganzen Top–Down — von einer Black–Box–Ansicht bis hin zu einer Beschreibung einiger „Spezialregister“ — beschrieben werden: Im nächsten Abschnitt wird das Black–Box–Verhalten des Octchip und seine Einbindung in einen Wirtsrechner erläutert. Daran schließen sich durch den Abschnitt „Grobentwurf“ motiviert die Vorstellungen von Rechenwerk und Steuerwerk als erste Partitionen des Entwurfs an. Zur besseren Übersicht stellen wir den Abschnitten über diese komplexeren Komponenten eine Beschreibung einiger grundlegender Komponenten voran. Ein Einblick in die Mikroprogrammierung des Octchip, durch die das Zusammenspiel der einzelnen Komponenten deutlich wird, schließt dieses Kapitel.

4.1 Spezifikation des Octchip



Der zu entwickelnde Chip soll in ein existierendes Computersystem integriert werden und den Prozessor des Wirtsrechners bei der Behandlung von Octrees unterstützen. Dabei übernimmt er die Aufgaben der Low–Level–Operationen der Softwarelösung: Er soll die Daten aktueller Knoten in einem Kubusbaum verwalten, Operationen zur Deklaration eines anderen Baumknotens zum aktuellen Knoten — kurz Navigationsoperationen — ermöglichen und Manipulationen auf Bitblockebene durchführen können[2]. Die Ablage der Kubusbäume erfolgt im Hauptspeicher des Wirtsrechners.

Zur Wahrnehmung dieser Aufgaben ist im Octchip ein Satz von Registern implementiert, auf die der Prozessor des Wirtsrechners zur Vor– und Nachbereitung einer Berechnung zugreifen kann. Der

[1] In den Teilen des Entwurfs, bei denen auf Vorlagen in [1] zurückgegriffen werden konnte — im wesentlichen die Adreßberechnung und der zugehörige Teil des Mikroprogramms — werden wir explizit auf diesen Rückgriff hinweisen. Die übrigen Teile des Entwurfs sind in [1] nicht erwähnt — wie Steuerwerk oder Interfacelogik — oder wurden — wie die Bitblocklogik — vollständig neu entworfen.

[2] Eine Auflistung der vom Octchip durchführbaren Operationen befindet sich in Anhang A.1.

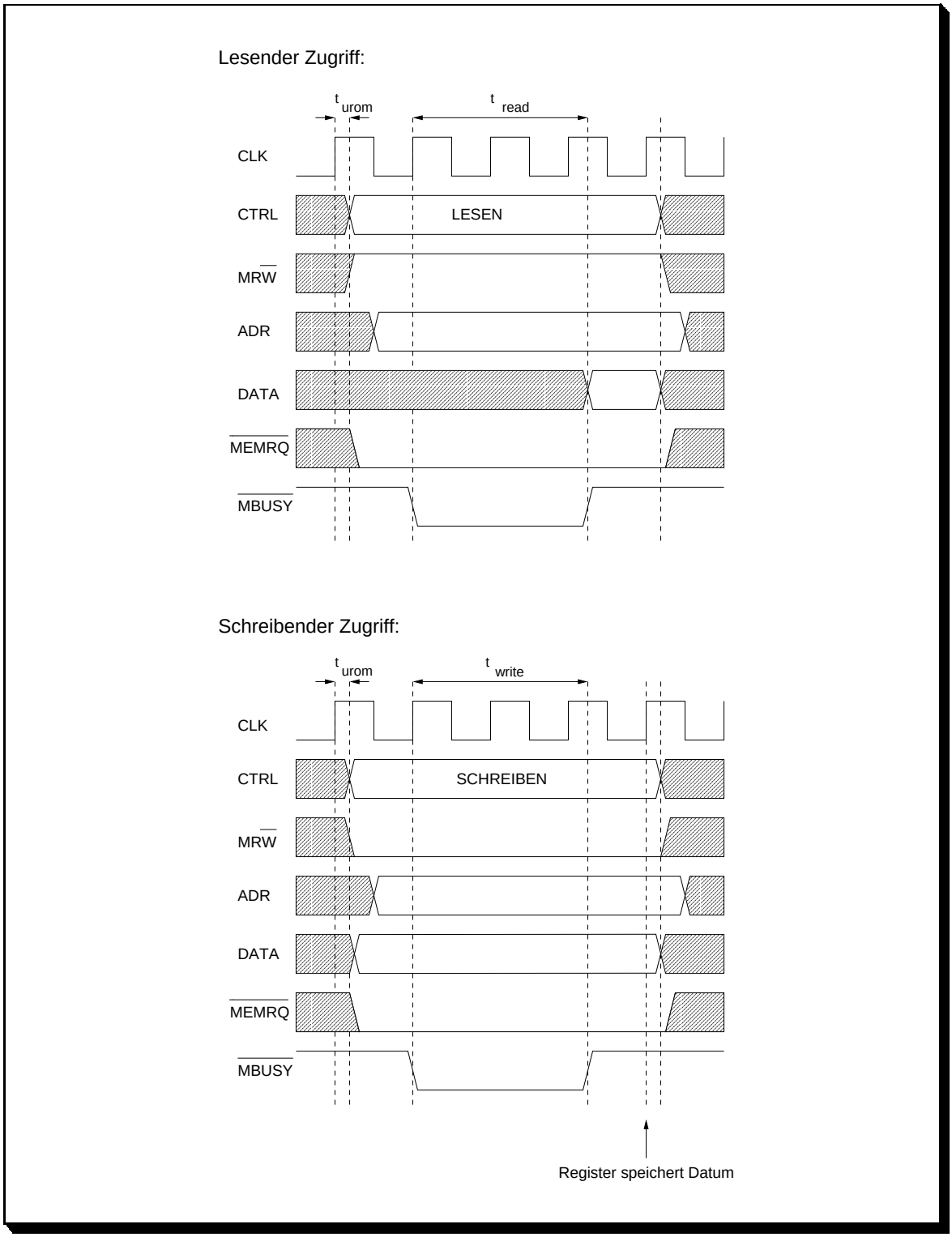


Abbildung 4.1: Timing–Diagramm eines Speicherzugriffs

Octchip selbst arbeitet dabei nicht; wir werden diesen Zustand des Octchip im folgenden „passiven Modus“ nennen. Der Octchip beginnt erst dann eine Berechnung, wenn er vom Wirtsprozessor aktiviert wird; wir bezeichnen diesen Zustand „aktiven Modus“. Da der Octchip im aktivem Modus ggf. auf den Speicher zugreift, muß der Prozessor des Wirtsrechners dabei evtl. den Systembus an den Octchip abgeben.

4.1.1 Interface

Die Schnittstelle des Chips zur „Außenwelt“ besteht aus einem 16 Bit breitem Daten– und einem 24 Bit breitem Adreßbus. Beide sind bidirektional, da der Adressbus sowohl zur Adressierung eines Registers im Octchip als auch beim Zugriff des Octchip auf den Speicher genutzt wird. Hinzu kommen bestimmte Steuerleitungen — sie sind nebenstehender Tabelle zu entnehmen — und eine Taktleitung CLK[3]. Weitere Erläuterungen der Steuerleitungen geben die folgenden Abschnitte.

Leitung	Funktion
$\overline{\mathsf{CE}}$	Octchip aktivieren
$\overline{\mathsf{CS}}$	Zugriff auf Chipregister initiieren
$\mathsf{CR}\overline{\mathsf{W}}$	Zugriffsmodus auf Chipregister
$\overline{\mathsf{MBUSY}}$	Speicherzugriff dauert an
$\overline{\mathsf{BUSGR}}$	Chip hat Bus zugeteilt bekommen
$\overline{\mathsf{BUSY}}$	Chip signalisiert aktiven Modus
$\overline{\mathsf{BUSRQ}}$	Chip fordert Bus an
$\overline{\mathsf{MEMRQ}}$	Chip greift auf Speicher zu
$\mathsf{MR}\overline{\mathsf{W}}$	Zugriffsmodus auf Speicher
RESET	Chip zurücksetzen

Tabelle 4.1: Steuerleitungen des Octchip

4.1.1.1 Passiver Modus

Im passiven Modus kann der Wirtsrechner auf die Register des Octchip zugreifen, indem er die Registeradresse auf den Adreßbus legt und — bei schreibendem Zugriff — das Datum auf den Datenbus. Über die Steuerleitung $\mathsf{CR}\overline{\mathsf{W}}$ wird die Übertragungsrichtung festgelegt und der Registerzugriff mittels $\overline{\mathsf{CS}}$ initiiert. $\overline{\mathsf{CS}}$ muß dabei solange aktiv sein, bis der Octchip das Datum übernommen hat — dies geschieht bei der nächsten steigenden Taktflanke —, bzw. bis der Prozessor des Wirtsrechners das vom Octchip gelieferte Datum übernommen hat.

Eine Auflistung der Registeradressen befindet sich in Anhang A.2. Zu beachten ist, daß auf einige Register nur lesend zugegriffen werden kann.

4.1.1.2 Aktiver Modus

Der Wirtsrechner schaltet den Octchip vom passiven in den aktiven Modus, indem mindestens ein Takt lang das Signal $\overline{\mathsf{CE}}$ aktiviert wird. Vor der Aktivierung sollte der Wirtsrechner das Instruktionsregister des Octchips mit dem Opcode der durchzuführenden Operation beschreiben (vgl. Kap. 4.4.2 und Anhang A.1). Der Octchip setzt dann das Signal $\overline{\mathsf{BUSY}}$, fordert in Abhängigkeit vom Opcode im Instruktionsregister über das Signal $\overline{\mathsf{BUSRQ}}$ den Systembus an und wartet, bis ihm über $\overline{\mathsf{BUSGR}}$ mitgeteilt wird, daß er den Bus benutzen darf.

Der Octchip benutzt zum Speicherzugriff ein einfaches Handshake–Verfahren: Nach einer steigenden Taktflanke liegt um t_{urom} ($\rightarrow$ 4.2.2) verzögert ein neuer Mikrobefehl ($\rightarrow$ 4.5) an: Der Chip legt die gewünschte Adresse auf den Adreßbus, bei schreibendem Zugriff ein Datum auf den Datenbus, legt über $\mathsf{MR}\overline{\mathsf{W}}$ die Zugriffsrichtung fest und initiiert einen Speicherzugriff durch eine fallende Flanke an $\overline{\mathsf{MRQ}}$. Der Speicher bestätigt diese Anforderung durch eine fallende Flanke an $\overline{\mathsf{MBUSY}}$, wobei nicht spezifiziert ist, wieviel Zeit zwischen diesen Flanken vergeht. Im Falle des lesenden Zugriffs indiziert der Speicher durch eine steigende Flanke an $\overline{\mathsf{MBUSY}}$, daß das auf dem Datenbus liegende Wort gültig ist und das adressierte Register speichert dieses Datum bei der nächsten steigenden Taktflanke. Bei schreibendem Zugriff indiziert eine steigende Flanke an $\overline{\mathsf{MBUSY}}$ den abgeschlossenen Schreibvorgang.

[3] Zur Schreibweise: Signale mit negativer Logik (Low–Aktiv) werden — in dieser Arbeit — in den VHDL–Texten i.d.R. `klein` geschrieben, Signale mit positiver Logik `GROSS`. In der dem Synthesesystem entnommenen Black–Box–Ansicht des Chips ist das Signal $\mathsf{CR}\overline{\mathsf{W}}$ daher `CR_W` notiert.

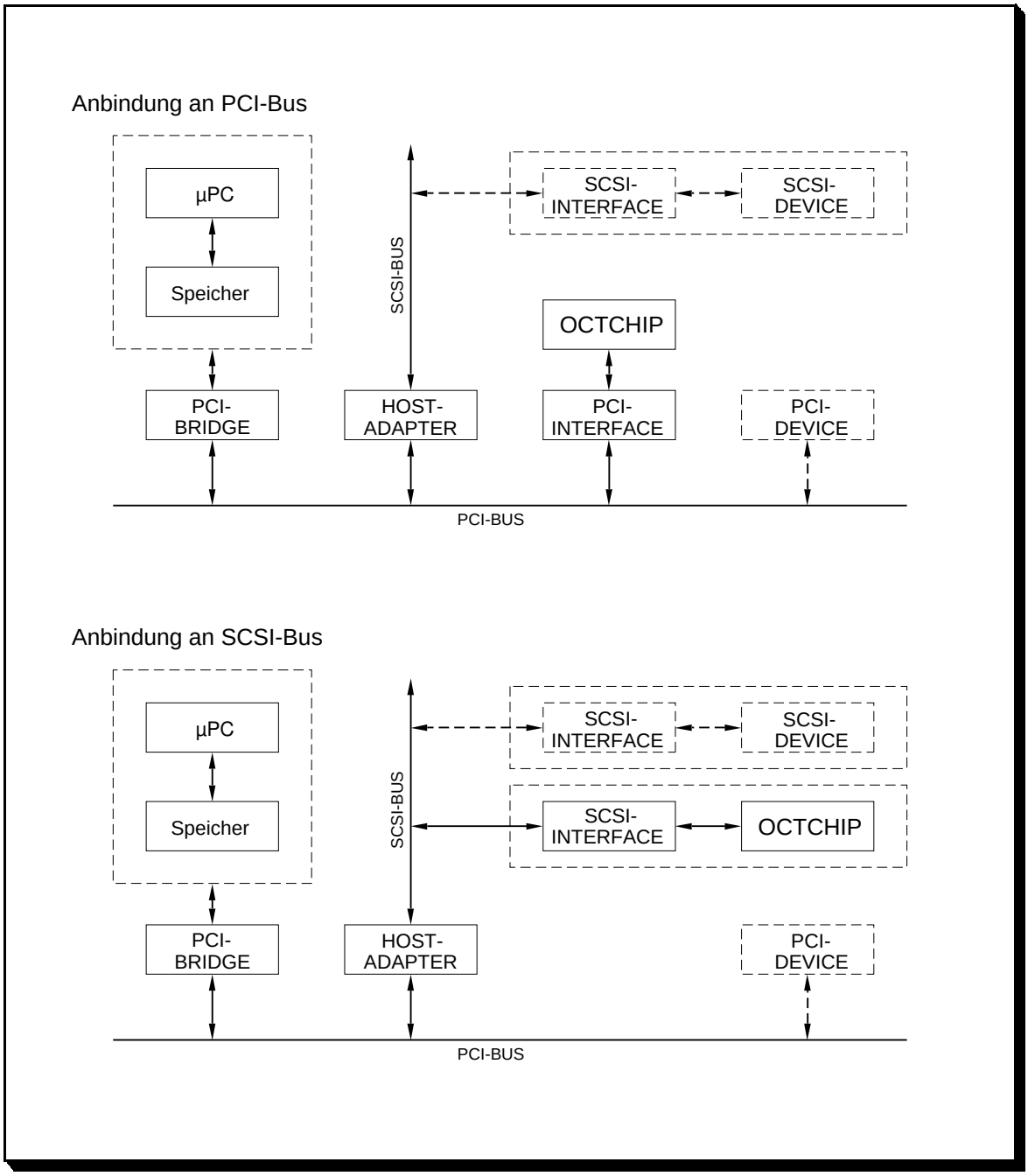


Abbildung 4.2: Anbindung des Octchip an PCI bzw. SCSI–Bus

In beiden Fällen beendet der Octchip den Speicherzugriff durch eine steigende Flanke an $\overline{\text{MRQ}}$. Abb. 4.1 zeigt die Timing–Diagramme eines lesenden und eines schreibenden Zugriff des Octchip auf den Speicher.

4.1.2 Einbindung in einen Wirtsrechner

Das in 4.1.1.2 beschriebende einfache Kommunikationsprotokoll wird in den meisten Anwendungsfällen zur Integration des Octchip in einen Wirtsrechner nicht ausreichen. Vielmehr wird der Bedarf nach Anschluß des Octchip an einen standardisierten Bus wie SCSI oder PCI ([39], [40]) bestehen. In einem solchen Fall ist also ein Interface zwischen Bus und Octchip zu entwerfen. Abb. 4.2 zeigt die Anbindung des Octchip an einen PCI– oder SCSI–Bus. Der Vorteil dieser Vorgehensweise ist eine höhere Flexibilität des Octchip durch Unabhängigkeit von konkreten Bussystemen.

4.2 Grobentwurf

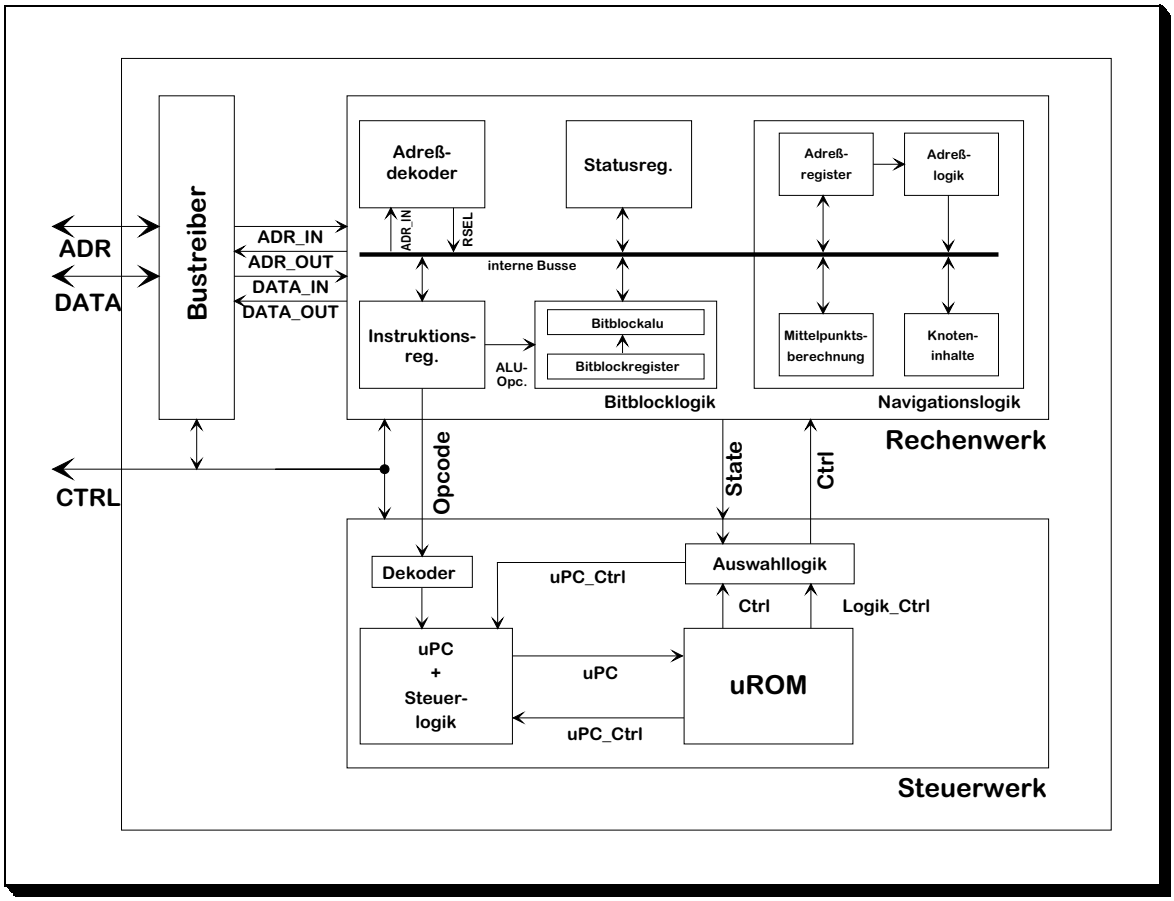


Abbildung 4.3: Blockschaltbild des Octchip

Wie das Blockschaltbild in Abb. 4.3 zeigt, besteht der Chip auf oberster Ebene aus zwei großen Blöcken: Ein Rechenwerk enthält die Speicher– und Funktionseinheiten des Chips und ein Steuerwerk übernimmt die Koordination der Datenflüsse und Operationen des Rechenwerks. Rechenwerk und Steuerwerk kommunizieren über die Busse STATE und CTRL miteinander. Ersterer dient dazu, dem Steuerwerk den aktuellen Status des Rechenwerks kenntlich zu machen, letzterer zur Übertragung der Kommandos vom Steuerwerk an das Rechenwerk. Ferner wird der aktuelle Opcode aus dem Instruktionsregister des Rechenwerks in das Steuerwerk übertragen.

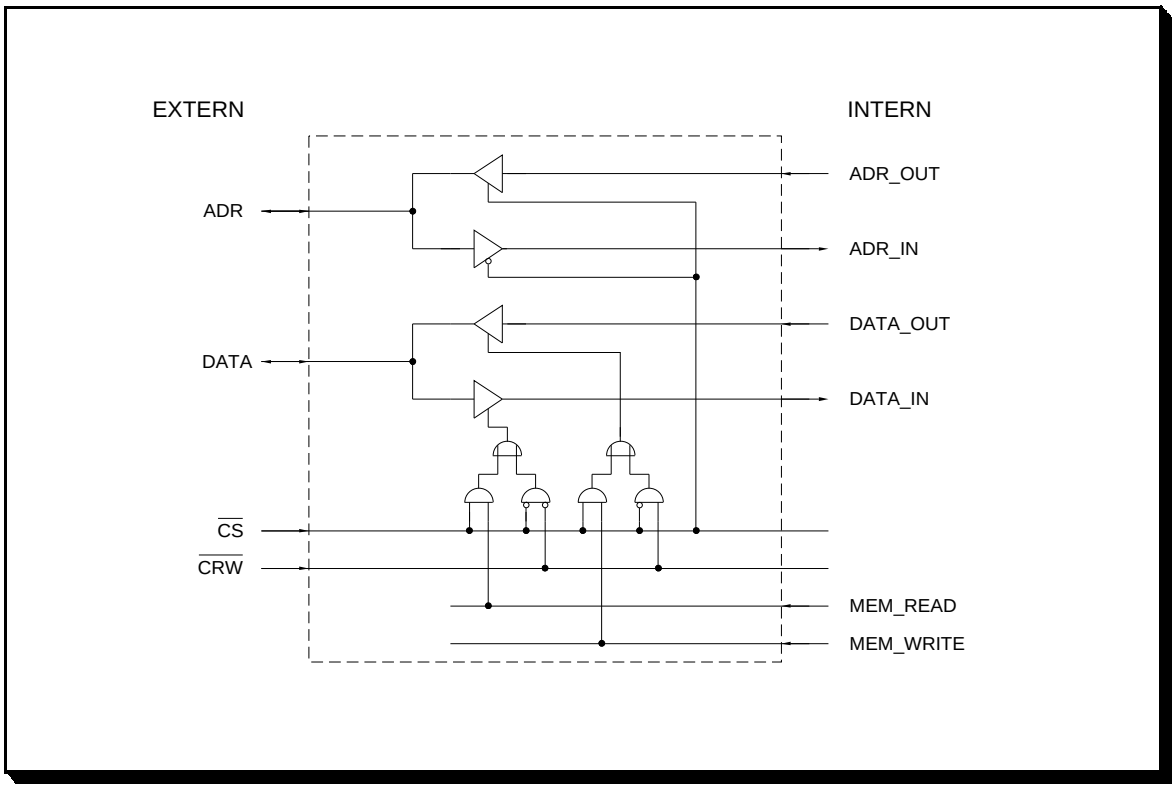


Abbildung 4.4: Schaltbild des Input/Output-Treibers

4.2.1 Bustreiber

Die dritte Einheit auf oberster Ebene des Chips ist ein Treiber, der die Verbindung der chipinternen Busse DATA_IN, DATA_OUT, ADR_IN und ADR_OUT mit den externen Bussen DATA und ADR herstellt. Ein Schaltbild dieses Treibers findet sich in Abb. 4.4.

4.2.2 Internes Timing

Beim Entwurf von Rechenwerken entsteht folgendes als „critical race“ bekannte Problem: Das Resultat einer Verknüpfung $\diamond$ zweier Register A und B ist in Register B abzulegen (Abb. 4.5(1), vgl. [36], Bild 1.6-2). Die für A und B verwendeten Register speichern bei steigender Taktflanke das am Eingangsbus liegende Datum. Da der Speichervorgang eine gewisse Zeit andauert, muß das Datum für eine bestimmte Zeit t_{hold} nach der Taktflanke gültig sein. Während dieser Zeit ist der Ausgang des Registers und — ohne die gestrichelt gezeichneten Hilfsregister — der Ausgang der ALU unbestimmt. Es besteht nun die Gefahr, daß das Register dieses geänderte Resultat der ALU übernimmt. Zur Vermeidung solcher Effekte schaltet man in die Eingangsbusse der ALUs Hilfsregister, die bei *fallender* Taktflanke die Daten an ihren Eingängen übernehmen (Abb. 4.5(2)). Die Adressierung der Register und ALUs wird in der Regel von einem Steuerwort (Mikrobefehl, $\rightarrow$ 4.5) vorgenommen. Dieses Steuerwort sollte während der Zeit t_{hold} ebenfalls konstant sein. Aus diesem Grund wird die Ausgabe des Steuerwortes CTRL durch das Steuerwerk um eine Zeit $t_{\text{urom}} > t_{\text{hold}}$ verzögert.

4.3 Grundlegende Komponenten

Beginnen wir die Präsentation des VHDL–Entwurfs mit einer Reihe von Komponenten, die in verschiedenen Teilen des Chips instantiiert werden. Da es sich um vergleichsweise triviale Komponenten handelt, sollen sie nicht in aller Ausführlichkeit diskutiert werden, allerdings baut der Entwurf von

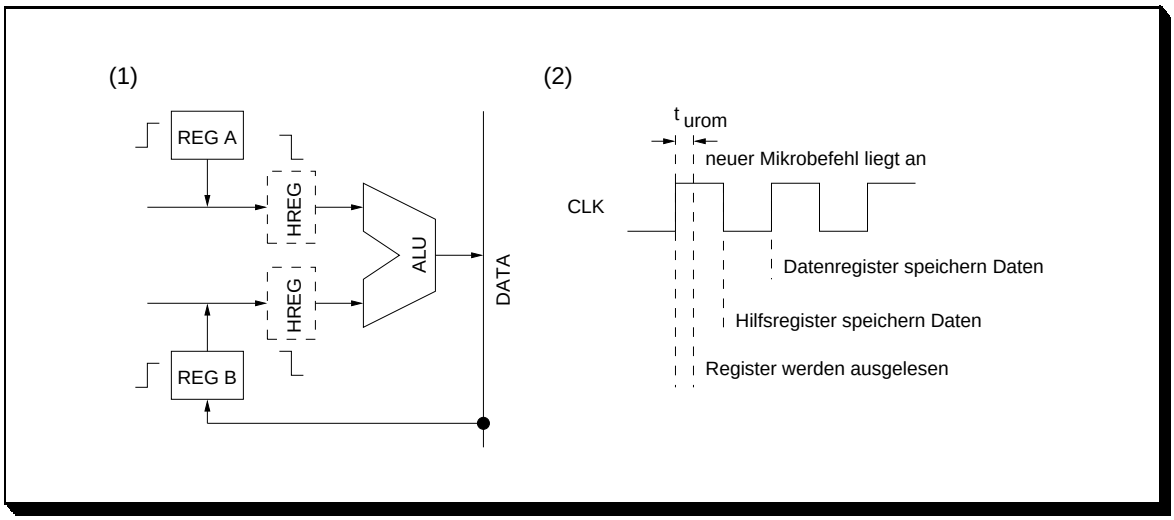


Abbildung 4.5: Internes Timing

komplexeren Einheiten auf ihnen auf, so daß auf eine kurze Vorstellung nicht ganz verzichtet werden soll.

4.3.1 Paket Octchip

Das VHDL–Package OCTCHIP enthält keine Komponente des Chips, vielmehr ist es Sammelstelle für im Design verwendete Typen, Subtypen und Konstanten. So wurden nicht nur Adreß– und Datenbusse als Subtypen von std_logic_vector definiert, sondern auch die meisten Steuer– und Statusbusse wie CTRL oder STATE. Dies ermöglichte eine Erweiterung dieser Busse, ohne in jedem Subdesign die Busbreite anpassen zu müssen. Weiterhin wurden für die einzelnen Leitungen der Steuer– und Statusbusse Konstanten definiert, wodurch der Zugriff auf diese Busse erheblich vereinfacht wurde[4]. So kann dank der Definition von

```
-- Belegung des State-Bus:
   constant NR_LE_1   : integer :=  0;
   constant NR_LQ_1   : integer :=  1;
   constant NR_EQ_1   : integer :=  2;
   ...
```

z.B. auf die Leitung, die anzeigt, ob das aktuelle Niveau gleich 1 ist (vgl. 4.4.4.3) durch `STATE(NR_EQ_1)` zugegriffen werden.

4.3.2 Generische Komponenten

Eine Reihe von Komponenten wurden mit einem oder mehreren generischen Parametern beschrieben — in der Regel die Breite des Eingabe– und Ausgabebusses —, was eine Anpassung der Komponenten an verschiedene Einsatzbedingungen erlaubt, da die generischen Parameter erst bei der Instantiierung der Komponente an einen Wert gebunden werden. So wurde z.B. ein n–Bit Tristate–Treiber beschrieben, der bei ena='1' seinen Eingang auf seinen Ausgang durchschaltet, anderfalls wird der Ausgang hochohmig:

[4]Leider wird der ALIAS–Mechanismus von VHDL nicht vom Synopsys–System unterstützt.

Entity	Generics	Ports
NEGATOR	BREITE: Positive := 16	EINGANG: in Std_Logic_Vector(BREITE-1 downto 0) COMPL : in Std_Logic AUSGANG: out Std_Logic_Vector(BREITE-1 downto 0)
ENA_REG	BREITE: Positive := 16	DIN : in std_logic_vector(BREITE-1 downto 0) ENA : in std_logic CLK : in std_logic DOUT: out std_logic_vector(BREITE-1 downto 0)
RESET_REG	BREITE: Positive := 16	DIN : in std_logic_vector(BREITE-1 downto 0) ENA1 : in std_logic ENA2 : in std_logic CLK : in std_logic OUT_ENA: in std_logic RESET : in std_logic DOUT : out std_logic_vector(BREITE-1 downto 0)
H_REG	BREITE: Positive := 16	DIN : in std_logic_vector(BREITE-1 downto 0 CLK : in std_logic DOUT: out std_logic_vector(BREITE-1 downto 0)
KONSTANTE _TO_B	BREITE: Positive := 16	ENA : in std_logic KON : in std_logic OUTPUT: out std_logic_vector(BREITE-1 downto 0)
DELAY	BREITE: Positive := 16	A: in std_logic_vector(n-1 downto 0) Z: out std_logic_vector(n-1 downto 0)
SCHIEBEREG_N	BREITE: Positive := 16 TIEFE: Positive := 10	EIN: in std_logic_vector(breite-1 downto 0) PUSH: in std_logic POP: in std_logic AUS: out std_logic_vector(breite-1 downto 0) CLK: in std_logic
SHIFTER42	BREITE: Positive := 16 FACTOR: Positive := 2 SHIN : Positive := 0	INPUT : in std_logic_vector(BREITE-1 downto 0) OUT_ENA: in std_logic OUTPUT : out std_logic_vector(BREITE-1 downto 0)

Tabelle 4.2: Generische Komponenten

```
entity NDRIVER is
   generic (N: Positive := 16);
   port (EINGANG: in  Std_Logic_Vector(N-1 downto 0);
         ENA    : in  Std_Logic;
         AUSGANG: out Std_Logic_Vector(N-1 downto 0));
end NDRIVER;

architecture STRUC of NDRIVER is
...
begin
  G: for I in EINGANG'range generate
       D: DRIVER port map (EINGANG(I), ENA, AUSGANG(I));
  end generate;
end STRUC;
```

Der so beschriebene Treiber kann nun in allen Bussen unabhängig von deren Breite eingesetzt werden. Bei der Instantiierung wird die Busbreite über den generischen Parameter angegeben:

```
  OUTPUTDRIVER: NDRIVER generic map(16) port map(INPUT, INPUT_TO_OUTPUT, OUTPUT);
```

Tabelle 4.2 zeigt weitere Komponenten dieser Art: NEGATOR ist ein steuerbarer Inverter, d.h. bei COMPL='1' wird der Ausgang invertiert durchgeschaltet, ansonsten nicht invertiert. ENA_REG ist

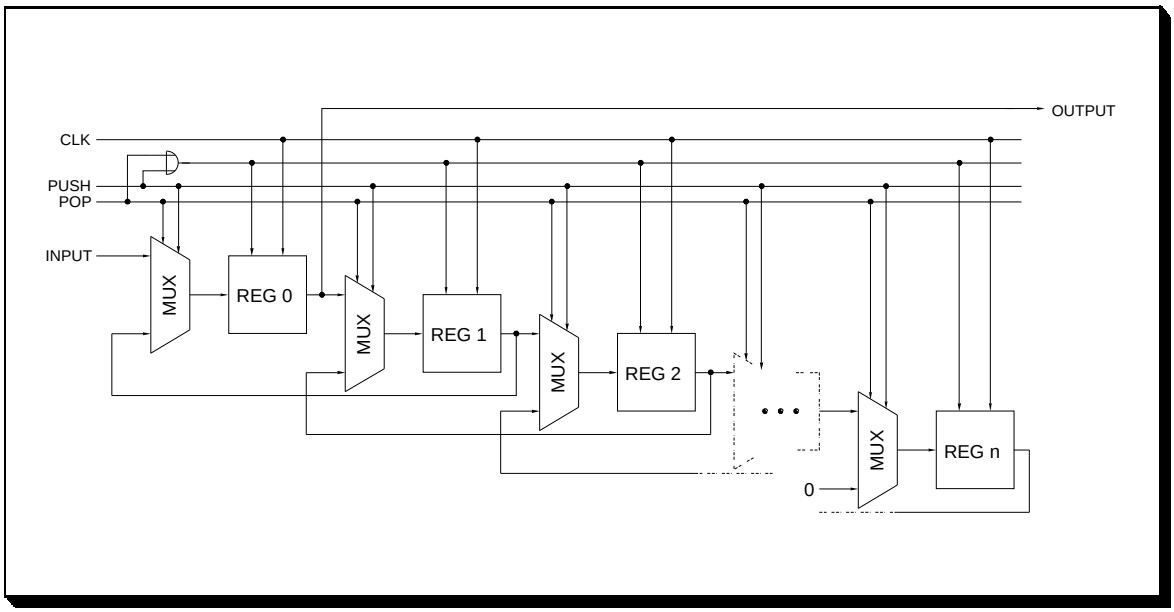


Abbildung 4.6: Schieberegister

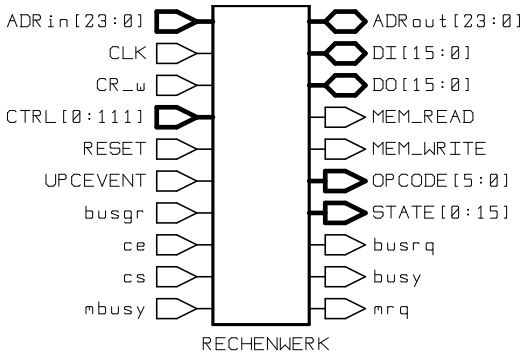
ein Register, das das an DIN anliegende Datum bei ENA='1' und steigender Flanke an CLK speichert und ständig an DOUT ausgibt. Eine Erweiterung dieses Registers ist RESET_REG, das einen weiteren Enable–Eingang hat — beide müssen aktiv sein, damit das Register speichert — und einen Eingang RESET, mit dem das Register asynchron zurückgesetzt werden kann. H_REG dient zur Konstruktion von Master–Slave–Registerstrukturen wie Hilfsregister in Alu–Eingängen (→ 4.2.2): Es übernimmt bei jeder *fallenden* Taktflanke den Wert an seinem Eingang. Mit Hilfe der Komponente KONSTANTE_TO_B kann ein Bus auf einen durch KON spezifizierten, konstanten Wert ('0 ... 0' oder '1 ... 1') gelegt werden. Ist ENA='0', so ist der Ausgang hochohmig. DELAY realisiert lediglich eine Zeitverzögerung durch zwei Inverter zwischen den Ein– und Ausgängen[5].

Die Komponente SCHIEBEREG_N ist ein Schieberegister mit generischer Anzahl der Schiebestufen (Tiefe) und Busbreite zur Implementierung von Stacks. Das Schieben wird durch eine der Leitungen PUSH oder POP, durch die auch die Schieberichtung spezifiziert wird, initiiert. Beim Hineinschieben (PUSH = '1') wird das an EIN liegende Datum als oberstes Stackelement übernommen. AUS liefert ständig das oberste Stackelement. Abb. 4.6 zeigt die Struktur des Schieberegisters.

Gleich drei generische Parameter hat die Komponente SHIFTER42, die einen Multiplizierer bzw. Dividierer um die Faktoren 4 oder 2 als Shifter um ein oder zwei Bit nach rechts oder links realisiert. Über FACTOR wird dabei Faktor und Schieberichtung spezifiziert, über SHIN wird ausgewählt, ob eine Eins oder eine Null hineingeschoben wird.

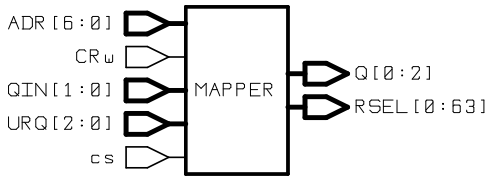
[5]Wichtig beim Entwurf dieser Komponente war, diese von jeder Logikoptimierung durch das Synopsys–System auszuschließen. Im Kapitel 4.4.3.1 wird dieses Problem näher behandelt.

4.4 Rechenwerk



In diesem Abschnitt soll die wichtigste Entwurfseinheit auf oberster Ebene des Octchip — das Rechenwerk — beschrieben werden. Wie bereits Abb. 4.3 zu entnehmen ist, besteht das Rechenwerk aus zwei großen Funktionsblöcken, die Bitblock– und die Navigationslogik, die von einem Adreßdekoder, einem Status– und einem Instruktionsregister begleitet werden.

4.4.1 Adreßdekoder



Die Aufgabe der Adreßdekodierung ist es, die Eingangs– und Ausgangstreiber (Tristate–Treiber) der Register des Rechenwerkes unter Berücksichtigung des vom Steuerwerk gelieferten Mikrobefehls (vgl. Kap. 4.5) oder der anliegenden Adresse — je nach Betriebsmodus des Chips — anzusteuern.

Wie in Abb. 4.7 zu sehen, kann man die Adreßdekodierung in zwei Stufen betrachten: Abb. 4.7(1) zeigt die unten erläuterte Berechnung von RSEL und q, Abb. 4.7(2) zeigt exemplarisch, wie mit RSEL, q und dem Steuerbus CTRL ein Register ausgewählt wird. RSEL im passivem Modus und CTRL im aktivem Modus wählen dabei u.U. nur einen Block von drei Registern — eines für jedes gespeicherte Kubusmodell — aus. Innerhalb dieses Blocks wird durch q ein Register selektiert.

Die Spezifikation des Demultiplexers für q erlaubt es, die drei Register eines Blocks gleichzeitig zu adressieren:

q	Selektiert:
00	Baum 0
01	Baum 1
10	Baum 2
11	alle

4.4.1.1 Passiver Modus

Im passiven Modus — indiziert durch $\overline{\mathsf{CS}}$ = '0' — wird der anzusteuernde Tristate–Treiber durch die unteren 7 Bits der anliegenden Adresse und der Übertragungsrichtung $\mathsf{CR}\overline{\mathsf{W}}$ selektiert. Diese 7 Bits werden folgendermaßen interpretiert[6]:

6	5	4	3	2	1	0
User/Test	q		Adresse			

[6] q dient wie bei der Softwareimplementierung zur Auswahl eines Baumes

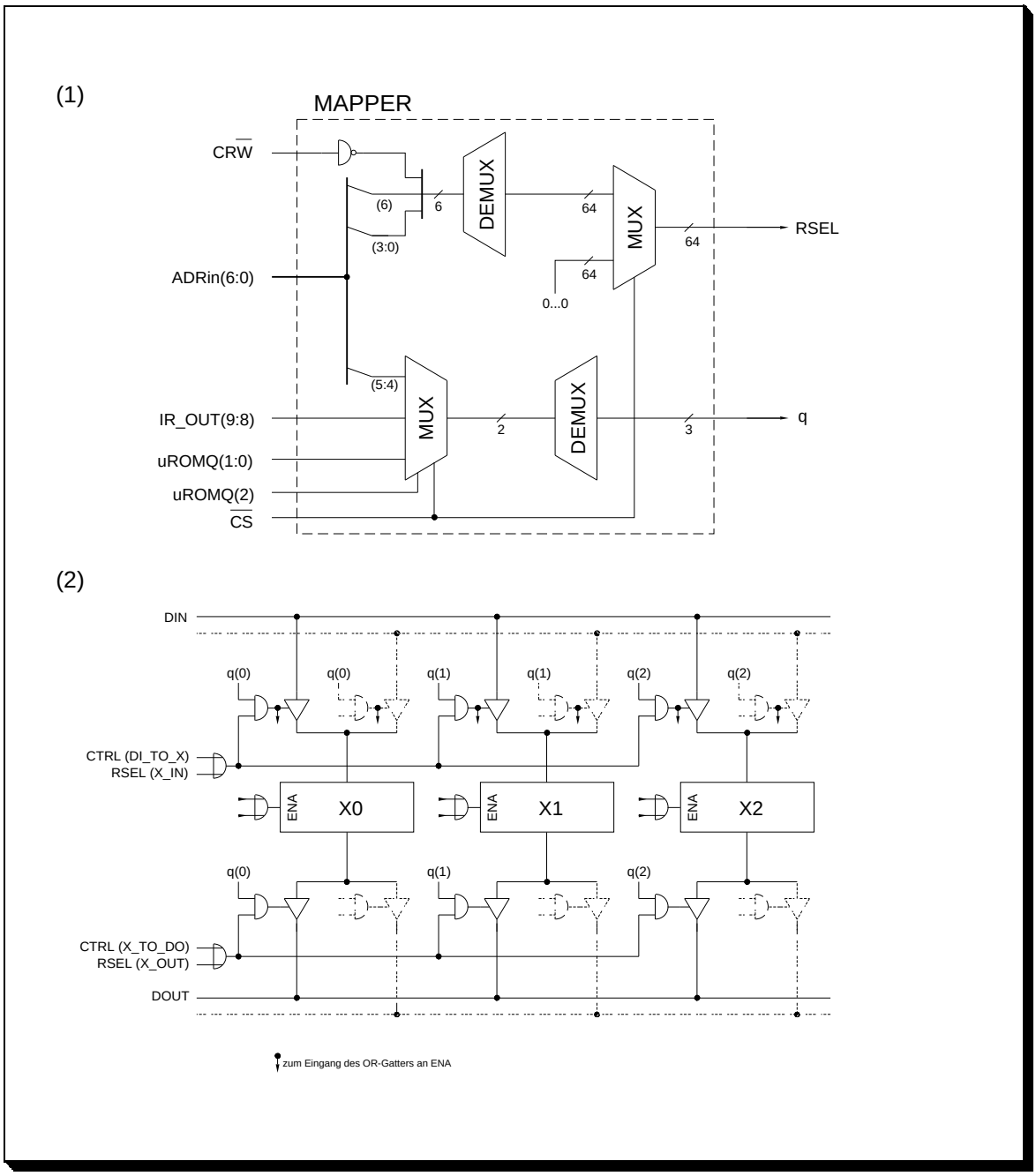


Abbildung 4.7: Adreßdekodierung

Die Entscheidung, ein separates Bit zur Unterscheidung zwischen Benutzeradressen und Testmodusadressen einzuführen — alternativ hätten ADRin(6:5) als q und ADRin(4:0) als Adresse interpretiert werden können — entstand aus der Überlegung, die Register, auf die ein Anwender nicht zugreifen sollte und auf die nur zum Testen des Chips zugegriffen werden darf, in einer Anwendungsumgebung gänzlich vom externen Zugriff auszuschließen, indem etwa diese Leitung im passivem Modus vom Adreßbus des Wirtsrechners getrennt und explizit auf Null gelegt wird. Eine solche Adressleitung sollte dann das führende Bit der externen Adresse enthalten, so daß keine Lücken im Adreßraum des Chips auftreten. Damit teilt sich der Adreßraum des Chips wie folgt auf:

Adressbereich (Hex)	Zugriff	
00–0F	Benutzer	Baum 0
10–2F	Benutzer	Baum 1
20–2F	Benutzer	Baum 2
40–4F	Test	Baum 0
50–5F	Test	Baum 1
60–6F	Test	Baum 2

Chipintern wurde dagegen der Übertragungsmodus CR$\overline{\text{W}}$ zum führenden Bit der Treiberselektierung gewählt. Der Vorteil dieser Wahl ist, daß die unteren 5 Bits der internen Adresse ein Register auswählen, während Bit 5 zwischen Eingangs– und Ausgangstreibern eines Registers entscheidet:

5	4	3	2	1	0
CR$\overline{\text{W}}$	User/Test	Adresse			

Die interne Adresse wird zur endgültigen Auswahl eines Treibers an den Eingang eines 6–zu–64–Demultiplexers gelegt, dessen Ausgang RSEL bei $\overline{\text{CS}}$ = '0' genau einen Treiber[7] ansteuert, bei $\overline{\text{CS}}$ = '1' keinen.

Nebenbei bemerkt kann bei dieser Art der Adressierung auf die Register, die nur einfach auftreten und deren Selektierung demzufolge nicht von q abhängt, mittels mehrerer Adressen zugegriffen werden (siehe auch Anhang A.2).

4.4.1.2 Aktiver Modus

Im aktivem Modus wird durch RSEL kein Tristate–Treiber selektiert (RSEL(i) = '0' $\forall i$), die Ansteuerung dieser obliegt vielmehr dem Steuerwerk, welches über den Bus CTRL die Tristate–Treiber und Funktionseinheiten des Rechenwerks kontrolliert. Der Baumidentifikator q wird in dieser Betriebsart entweder vom Instruktionsregister oder vom Steuerwerk geliefert, die Selektion zwischen beiden Alternativen ist Bestandteil des Mikrobefehls.

Im Gegensatz zum passivem Modus kann im aktivem Modus auch über andere Busse auf die Register zugegriffen werden wie in 4.7 gestrichelt angedeutet ist. So können einige Register über den Adreßbus ADRout geladen werden, andere über die Eingangsbusse der Addierer der Navigationslogik ausgelesen werden (vgl. Abb. 4.15).

Aus Gründen der Übersichtlichkeit wollen wir auf eine Darstellung der vollständigen Registerbank des Octchip verzichten. Eine Übersicht der Adressierungsmöglichkeiten im passivem Modus kann Anhang A.2, im aktivem Modus A.3 entnommen werden. Hier sei nur auf zwei Spezialfälle hingewiesen: Zur Speicherung der relativen Adressen gemäß 2.1.5 müssen das Register AXR durch 4 dividiert und das Register IP2 durch 2 dividiert und mit führender Eins versehen[8] auf den Datenbus DOUT gelegt werden können, was Instantiierungen der Komponente SHIFTER42 ermöglichen:

```
IP2oDO2: SHIFTER42 generic map(16,12,1) port map(IP2_OUT, IP2_TO_DO, DO);
AXroDO2: SHIFTER42 generic map(16,14,0) port map(AXr_OUT, AX4_TO_DO, DO);
```

[7] genauer: einen Treiberblock, aus dem dann durch q ein Treiber selektiert wird
[8] vgl. [1] Alg. 5.2–A. In [1] Abb. A.1(1) fehlen diese Komponenten allerdings.

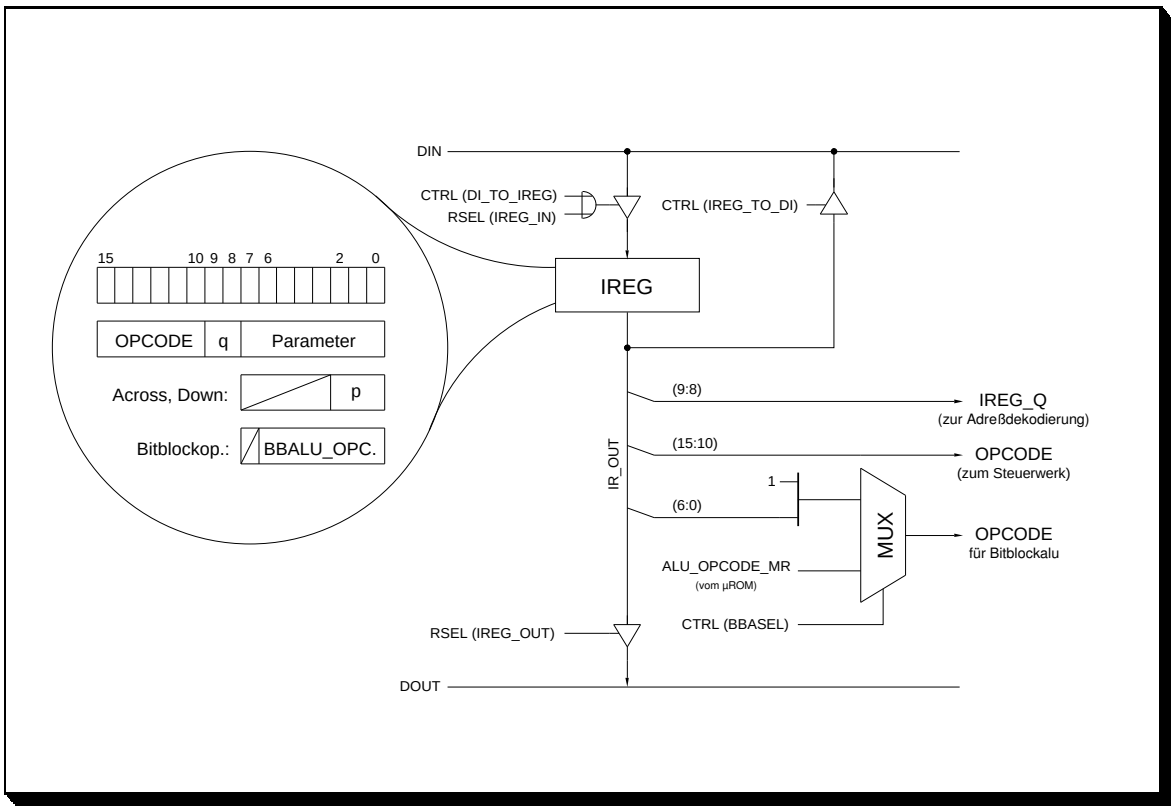


Abbildung 4.8: Instruktionsregister

4.4.2 Instruktionsregister

In das Instruktionsregister schreibt der Wirtsrechner ein den auszuführenden Befehl spezifizierendes Wort. Dieses Wort enthält im allgemeinen eine der gewünschten Operation zugeordnete Bitfolge, den sogenannten Opcode. Bei einigen Chiparchitekturen enthält das Instruktionsregister neben dem Opcode weitere Parameter des Befehls. So auch im Fall des Octchips: Im Instruktionsregister des Octchip werden neben dem Opcode (6 Bits) noch — abhängig von der Operation — eine Identifikation des zu bearbeitenden Baumes q, der Zielkubus p bzw. eine Kodierung der von der Bitblockalu auszuführenden Operation gespeichert.

Abb. 4.8 zeigt die Einbindung des Instruktionsregisters. Die Kopplung von IR_OUT mit DIN ist notwendig, da das Positionsregister und die Mittelpunktsberechnung den in IREG gespeicherten Zielkubus p von DIN lesen.

Die vom Octchip durchführbaren Operationen können Anhang A.1 entnommen werden.

4.4.3 Statusregister

Die Aufgabe des Statusregisters ist die Speicherung bestimmter Informationen über den aktuellen Zustand des Rechenwerkes. Dabei wollen wir nur die Informationen dem Statusregister zuordnen, die explizit gespeichert werden; jene Informationen, die Ausgaben kombinatorischer Funktionen sind — wie z.B. die Information, ob der Inhalt des Registers AXL Null ist —, sind zwar Teil des Statuswortes STATE, nicht jedoch des Statusregisters. Eine Übersicht der im Statusregister gespeicherten Informationen ist Tabelle 4.3 zu entnehmen.

Das Setzen und Rücksetzen der einzelnen Register im Statusregister hängt von bestimmten Ereignissen ab bzw. wird vom Steuerwerk ausgeführt. Da die Bedingungen zum Setzen und Rücksetzen der Register teilweise recht komplex sind, sind Standard–Flip–Flops zur Realisierung dieser Register nicht ausreichend. Dies wird im nächsten Unterkapitel diskutiert.

Interne Informationen:	
STATE(BCmod)	Bitblockregister oder Knoteninhalt im Baum 2 wurde seit dem letzten Laden verändert
STATE(CHIP_ENA)	Wirtsrechner hat Wechsel vom passivem zum aktivem Modus initiiert
STATE(HOLD)	Chip befindet sich im Wartezustand
Externe Informationen:	
BUSY	Chip befindet sich im aktivem Modus
BREQ	Chip fordert Bus des Wirtsrechners
MREQ	Chip initiiert Speicherzugriff

Tabelle 4.3: Inhalt des Statusregisters

4.4.3.1 Spezielle Register

Die Synchronisation des Steuerwerkes mit dem Speicher warf einige Probleme auf. Da ein Speicherzugriff in der Regel einen Taktzyklus überschreitet, wurde eine Statusleitung HOLD eingeführt, mit der Teile des Rechenwerkes und das Steuerwerk in einen Wartezustand gesetzt werden können.

Der erste Ansatz, $\overline{\text{MRQ}}$ als NOR–Verknüpfung der Steuerleitungen MEM_READ und MEM_WRITE zu betrachen sowie HOLD bei fallender Flanke[9] an $\overline{\text{MRQ}}$ zu setzen und steigender Flanke an $\overline{\text{MBUSY}}$ — sprich bei beendetem Speicherzugriff — zurückzusetzen zeigte seine Mängel bei aufeinanderfolgenden Speicherzugriffen: Hier wurde HOLD zwar durch die steigende Flanke an $\overline{\text{MBUSY}}$ zurückgesetzt, jedoch nicht erneut gesetzt, da an $\overline{\text{MRQ}}$ keine Änderung auftrat.

Die weitere Bedingung, HOLD bei steigender Taktflanke und aktivem Memory–Request zu setzen, zeigte auch nicht den gewünschten Erfolg: HOLD wurde nach Abschluß eines Speicherzugriffs bei der steigenden Flanke an CLK wieder aktiv, da an CTRL noch kein neuer Mikrobefehl anliegt, wodurch der Chip in einen ewigen Haltezustand versetzt wurde; da kein Speicherzugiff initiiert worden war, kam es zu keiner weiteren steigenden Flanke an $\overline{\text{MBUSY}}$ und die Bedingung zum Beenden des Haltezustands konnte nicht mehr erfüllt werden (siehe nebenstehende Abbildung).

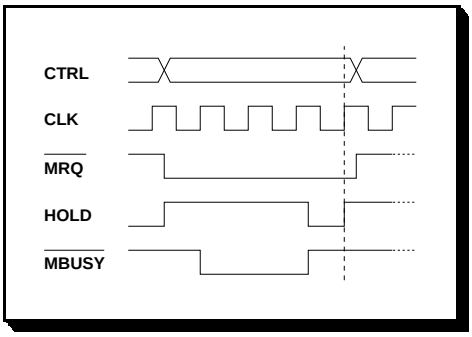


Abbildung 4.9: Sync. Problem

Durch die Einführung des Signals uPCevent, auf dem bei jedem Wechsel des Mikrobefehls eine Zustandsänderung erfolgt (vgl. 4.5.2), läßt sich dieses Problem lösen und ein spezielles Register für HOLD beschreiben:

```
m <= MEM_READ nor MEM_WRITE;

REG: process(m, mbusy, uPCevent)
     variable q: std_logic := '0';
     begin
       if (uPCevent'event and m='0')
          then q := '1';
       end if;
       if (mbusy'event and mbusy = '1')
          then q := '0';
       end if;
       HOLD <= q;
     end process;
```

[9] HOLD <= not $\overline{\text{MRQ}}$ wäre an dieser Stelle fatal gewesen, da MEM_READ und MEM_WRITE Signale des Steuerwerkes sind, diese sich bei aktiver HOLD–Leitung jedoch nicht ändern

Die RT–Simulation lieferte zwar das gewünschte Ergebnis, jedoch wurde diese Beschreibung vom Synopsys Design–Compiler als nicht synthetisierbar zurückgewiesen. Dieser akzeptiert VHDL–Konstrukte mit dem Attribut `'event` nur in der Form `if (`*signal*`'event and` *signal* `=` *wert*`)`.

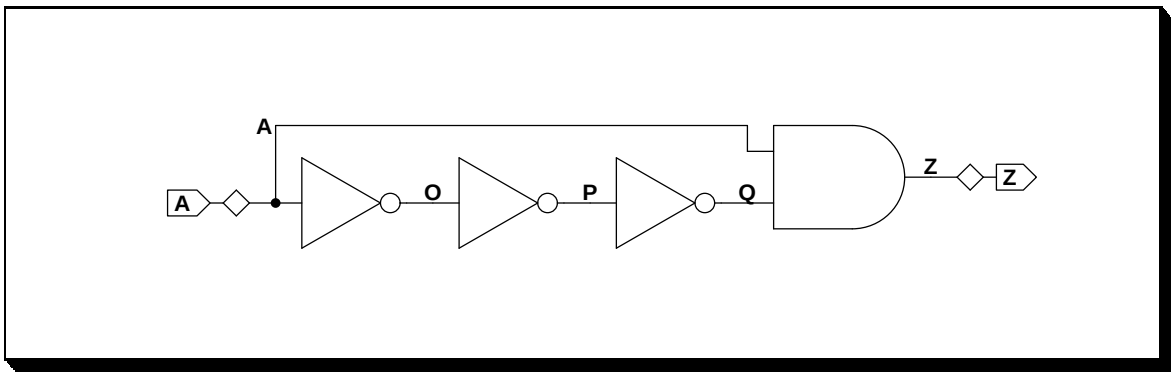


Abbildung 4.10: Flankendetektor (Schaltbild)

Eine Lösung dieses Problems[10] war die Beschreibung eines Flankendetektors, mit dessen Hilfe ein Flip–Flop in einer der obigen Beschreibung äquivalenten Weise angesteuert werden kann (vgl. Abb. 4.10):

```
1  entity SLEWDETECT is
2     port (A: in  std_logic;
3           Z: out std_logic
4          );
5  -- pragma dc_script_begin
6  -- set_dont_touch slewdetect
7  -- pragma dc_script_end
8  end SLEWDETECT;
9
10 architecture STRUC of SLEWDETECT is
11    component IVA
12       port( A : in std_logic;  Z : out std_logic);
13    end component;
14
15    component AN2
16       port(A, B: in std_logic; Z: out std_logic);
17    end component;
18
19    signal O, P, Q: std_logic;
20
21 begin
22   U1: IVA port map( A => A, Z => O);
23   U2: IVA port map( A => O, Z => P);
24   U3: IVA port map( A => P, Z => Q);
25   U4: AN2 port map( A => A, B => Q, Z => Z);
26 end STRUC;
```

Dieser Flankendetektor liefert bei einer steigenden Flanke im Eingangssignal einen kurzen Impuls am Ausgang (vgl. Abb. 4.11). Die drei hintereinandergeschalteten Inverter werden dazu genutzt, das Eingangssignal zu verzögern, so daß die beiden Eingänge des Und–Gatters das „aktuelle“ Eingangssignal[11] A^{t_n} und das invertierte „alte“ Eingangssignal $\neg A^{t_{n-3}}$ führen — dabei gilt

[10] nach mehreren vergeblichen Versuchen, das Verhalten des Registers Design–Compiler konform zu beschreiben
[11] bezeichne X^{t_n} den Wert des Signals X zum Zeitpunkt t_n

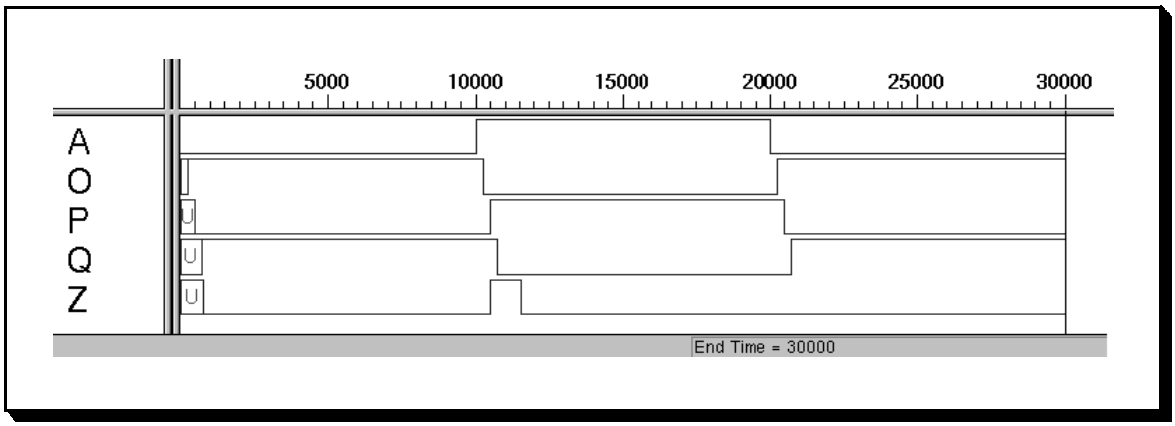


Abbildung 4.11: Flankendetektor (Timing)

$t_n - t_{n-3} = 3 \cdot t_{\text{Inverterdelay}}$. Liegt zwischen diesen eine steigende Flanke, gilt also $\mathsf{A}^{t_n} = '1'$ und $\mathsf{A}^{t_{n-3}} = '0'$, liegt an beiden Eingängen des Und–Gatters eine Eins.

Wichtig ist, daß dieser Flankendetektor von jeder Logikoptimierung ausgenommen wird; er würde auf eine Komponente mit offenem Eingang und auf Null liegendem Ausgang abgebildet. Dies wird mit den Compiler–Direktiven in den Zeilen 5 bis 7 sichergestellt: Eine mit dem Attribut don't–touch versehene Komponente wird vom Design–Analyzer nicht betrachtet. Dies impliziert allerdings auch die Abbildung der Komponente auf die selektierte Zieltechnologie, so daß die Komponente bereits mit den Bausteinen der Zieltechnologie beschrieben werden muß[12].

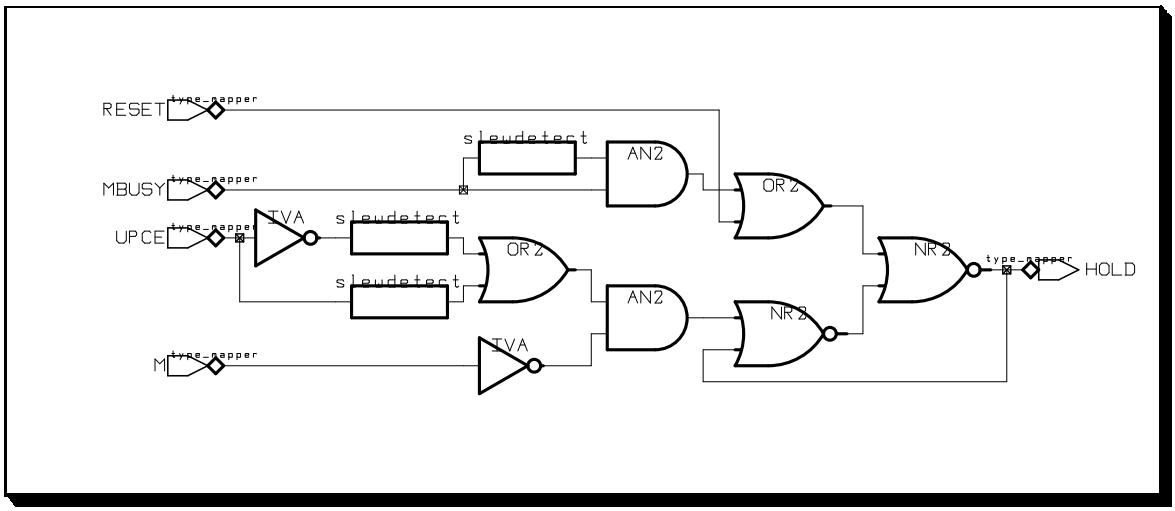


Abbildung 4.12: Register für HOLD

Der Flankendetektor wurde zur Konstruktion eines speziellen Registers für HOLD eingesetzt. Dabei zeigte sich, daß die Impulslängen der gelieferten Impulse nicht ausreichen, um ein Flip–Flop aus der LSI10k–Bibliothek zu schalten. Zur Vermeidung dieses Problems wurde ein mit zwei NOR–Gattern beschriebenes Flip–Flop verwendet (Abb. 4.12).

Ähnliche Überlegungen führten dazu, das Register HOLDREG auch für STATE(CHIP_ENABLE) zu

[12] Das Attribut map–only, das nach der Synopsys–Dokumentation den Design–Analyzer veranlaßt, zwar ein Technology–Mapping durchzuführen, jedoch keine Logikoptimierung, funktionierte hier aus unersichtlichen Gründen nicht. Nebenbei kann man die in der Zieltechnologie verwendeten Gatterbezeichnungen wie IVA für den Inverter der LSI–10k Bibliothek durch Speicherung eines in die Zieltechnologie abgebildeten Designs als VHDL–Beschreibung erhalten.

verwenden[13] sowie zwei weitere spezielle Register für $\overline{\mathsf{MRQ}}$ und BUSY einzuführen. Abb. 4.13 zeigt das gesamte Statusregister.

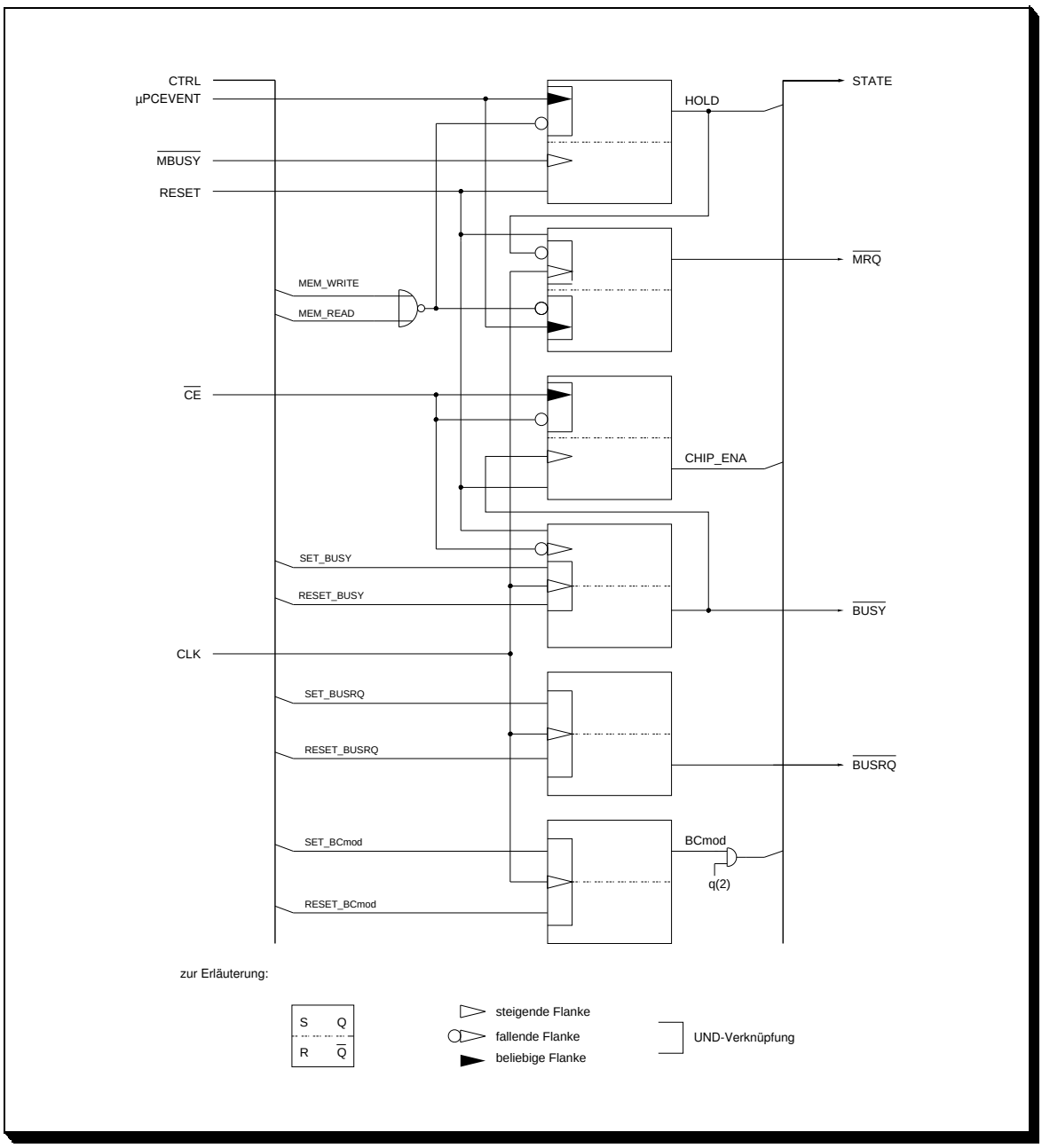


Abbildung 4.13: Statusregister

[13] Alternativ hätte auch ein Register mit der Setzbedingung „fallende Flanke an Signal“ statt der Setzbedingung „Flanke an Signal 1 und Signal = '0'“ entworfen werden können.

4.4.4 Navigationslogik

In der Navigationslogik sind die zur Durchführung der ersten Gruppe von Operationen des Octchip — den Navigationsoperationen — benötigten Register und Funktionsblöcke implementiert. Die Aufgaben der Navigationslogik sind die Berechnung von Adressen der Kubusbäume im Speicher sowie die Aktualisierung von Mittelpunkt und vektoriellem Index.

4.4.4.1 Adreßberechnung

Abb. 4.15 zeigt den Satz der zur Adreßberechnung implementierten Register[14] und deren Verbindung zu der aus zwei Addierern bestehenden Adreßlogik. Die vor die Eingänge der Addierer geschalteten Hilfsregister übernehmen bei *fallender* Taktflanke den am Eingang liegenden Wert und dienen bei Operationen des Typs $R1 := R1 + R2$ zur Vermeidung von „critical races“ (→ 4.2.2). Um die Abbildung nicht zu überladen, sind die Ein– und Ausgangstreiber der Register (vgl. Abb. 4.7) sowie Verschaltung mit den Bussen DIN und DOUT nicht eingezeichnet.

Das Register AB(q) enthält die Kachelnummer der Anfangsadresse von Modell q, im folgenden Basisadresse genannt. Wie bereits in 2.1.5 erwähnt, werden Kubusmodelle stets an fortlaufendenden Speicheradressen und an einem Kachelanfang beginnend im Speicher abgelegt werden.
Das Register AR(q) enthält die zum Modellanfang relative Adresse des aktuellen Zeiger– oder Bitblocks, aufgeteilt in ARL(q) und ARR(q). PR(q) enthält die aktuelle Position, so daß sich die Adresse des aktuellen Kubus aus AB(q) + AR(q) + PR(q) ergibt.
Die Nummer der Kachel mit den indirekten Adressen wird — ebenfalls relativ zum Modellanfang — in IB(q) festgehalten. Für Modell 2 werden darüberhinaus die Adressen der ersten freien Speicherzelle im Speicherbereich für Zeiger– und Bitblöcke und im Speicherbereich für indirekte Adressen in AP2 — aufgeteilt in AP2L und AP2R — und IP2 festgehalten. Schließlich rundet ein Register für Zwischenergebnisse (AX) das Ensemble der Adreßregister ab.
Nebenstehende Abbildung (vgl. [1] Abb. 5.2-1) verdeutlicht das Zusammenspiel der einzelnen Adreßregister.

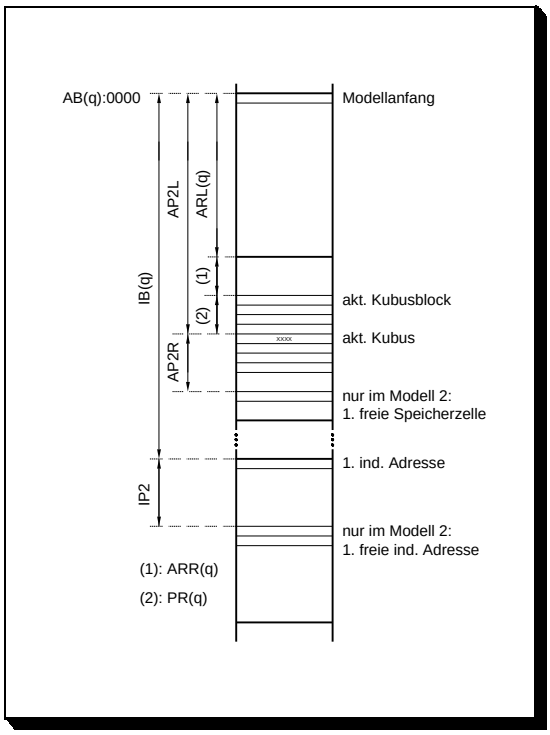


Abbildung 4.14: Adreßberechnung

Zur effizienten Implementierung der Operation UP müssen die Adresse des aktuellen Kubusblock AR und die aktuelle Position PR bei einer Abwärtsnavigation auf einem Stack abgelegt werden. Dieser Stack muß parallel ausgelesen werden können, um den Index des aktuellen Kubus zu erhalten. Weiterhin muß die Option, das Register AR subtrahieren zu können, implementiert werden, was gemäß $A - B = A + \overline{B} + 1$ durch Invertierung des Inhaltes von AR und Addition von 1 (gesetztem „Carry In“ des ersten Addieres) erreicht werden kann. Das Register ARR weist eine weitere Besonderheit auf: Da die Zeiger– bzw. Bitblöcke acht bzw. vier Speicherzellen belegen und ein Kubusbaum an einem Kachelanfang beginnend gespeichert wird, ist die Anfangsadresse AR des aktuellen Blocks stets ein Vielfaches von Vier, die beiden unteren Bits der Adresse also stets Null. Eine Speicherung dieser Bits in ARR ist also nicht notwendig.

[14] Dieser Registersatz orientiert sich an [1] Taf. 5.2-II, allerdings mußte die Verschaltung gegenüber [1]A.1(1) erheblich korrigiert und erweitert werden.

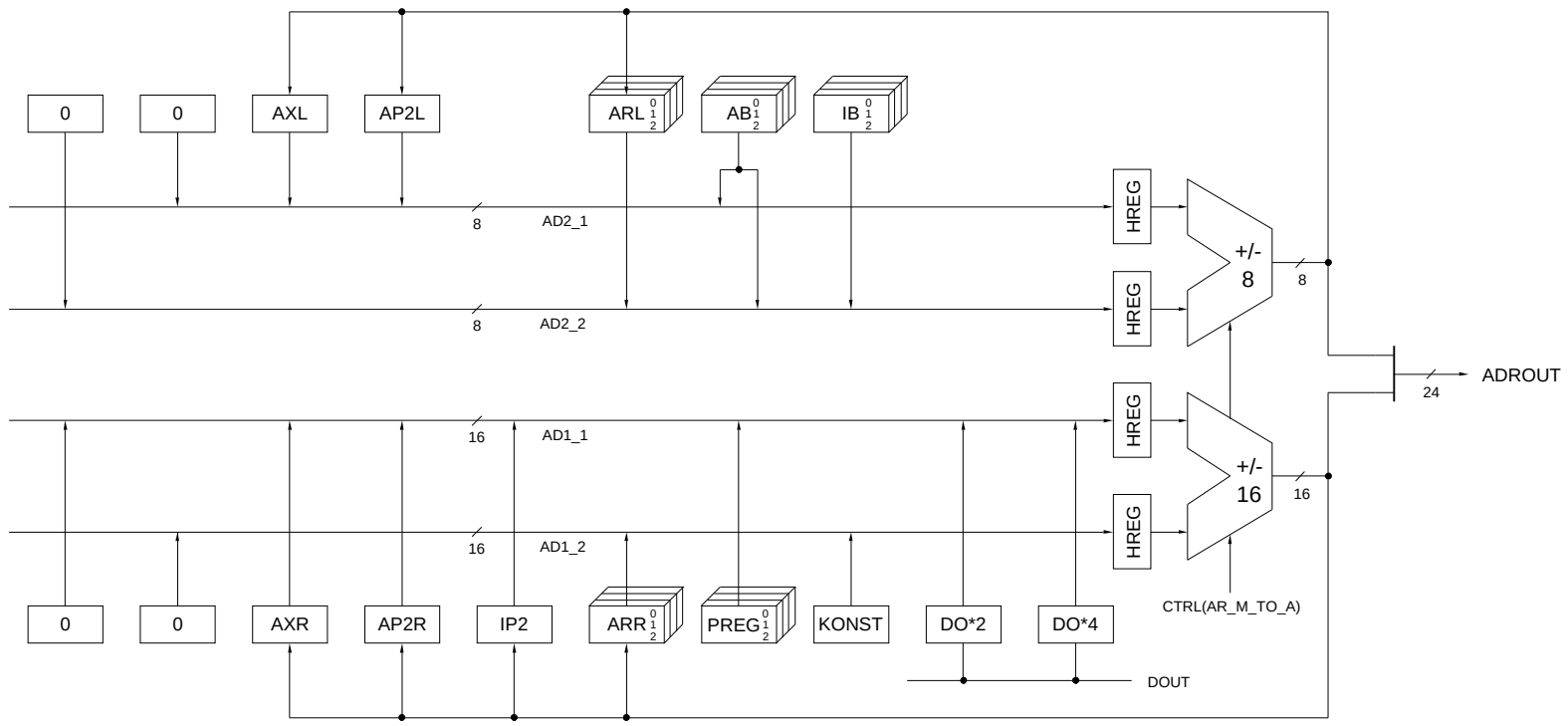


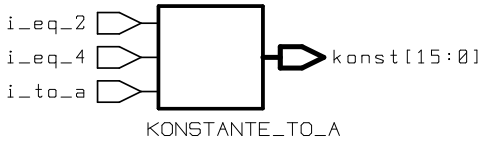
Abbildung 4.15: Adreßlogik

Zur Realisierung der Register ARL und ARR wurde daher die Komponente STACK_REG mit integriertem Schieberegister entworfen (→ 4.4.4.1.2), zur Realisierung von PR eine spezielle Komponenten PREG (→ 4.4.4.1.3).

Wie in Abb. 4.15 zu sehen lassen sich die Eingänge der Addierer der Adreßlogik über die Adreßregister hinaus mit einer Konstanten $k \in \{1, 2, 4, 8\}$ und dem mit 2 oder 4 multipliziertem Datenbus belegen. Dies ermöglichen entsprechende Instantiierungen der Komponenten KONSTANTE_TO_A (→ 4.4.4.1.1) und SHIFTER42.

Die in Abb. 4.15 mit 0 bezeichneten Komponenten sind Instantiierungen von KONSTANTE_TO_B, durch die ein nicht anderweitig beschalteter Addierereingang mit dem definierten Wert Null belegt werden kann. Dies ist notwendig, da sonst in der Simulation der Eingangsbus hochohmig wäre (`'Z...Z'`), wodurch der Ausgang des Addierers unbestimmt wird. In einer realen Schaltung ist der „Wert" von nicht beschalteten Eingängen technologieabhängig; so täuschen z.B. unbeschaltete TTL–Eingänge oft High–Pegel vor.

4.4.4.1.1 Generierung einer Konstanten



Die Komponente KONSTANTE_TO_A legt bei aktivem Signal I_TO_A eine von der Belegung der Signale I_EQ_2 und I_EQ_4 abhängige Konstante an ihren Ausgang (vgl. nebenstehende Tabelle). Bei inaktivem I_TO_A ist der Ausgang hochohmig.

I_EQ_4	I_EQ_2	Konstante
0	0	1
0	1	2
1	0	4
1	1	8

4.4.4.1.2 Register mit Stack

Die Komponente STACK_REG deckt den Bedarf eines Registers mit integriertem Stack und invertierbarem Ausgang. Die o.g. Sonderrolle von ARR wird durch Verwendung zweier generischer Parameter HIGH und LOW zur Angabe der Busbreite berücksichtigt. Ungeachtet des Parameters LOW ist der Ausgabebus vom Typ std_logic_vector(HIGH downto 0), um eine korrekte Ausgabe des komplementierten Registerinhaltes zu gewährleisten: Bei gewünschter Komplementierung ist für die nicht gespeicherten Bits '1' auszugeben. Bei Instantiierung eines Stack–Registers mit LOW > 0 werden daher die Leitungen LOW $- 1, \ldots, 0$ des Ausgangsbusses mit COMPL verbunden:

```
entity STACK_REG is
   generic(HIGH: Natural := 15;
           LOW : Natural := 0);
   port (CLK      : in std_logic;
         DI       : in std_logic_vector(HIGH downto LOW);
         AR       : in std_logic_vector(HIGH downto LOW);
         DOUT     : out std_logic_vector(HIGH downto 0);
         q        : in std_logic_vector(0 to 2);
         COMPL    : in std_logic;
         PUSH, POP: in std_logic;
         A_TO     : in std_logic;
         DI_TO    : in std_logic
        );
end STACK_REG;

architecture TOMS of STACK_REG is
```

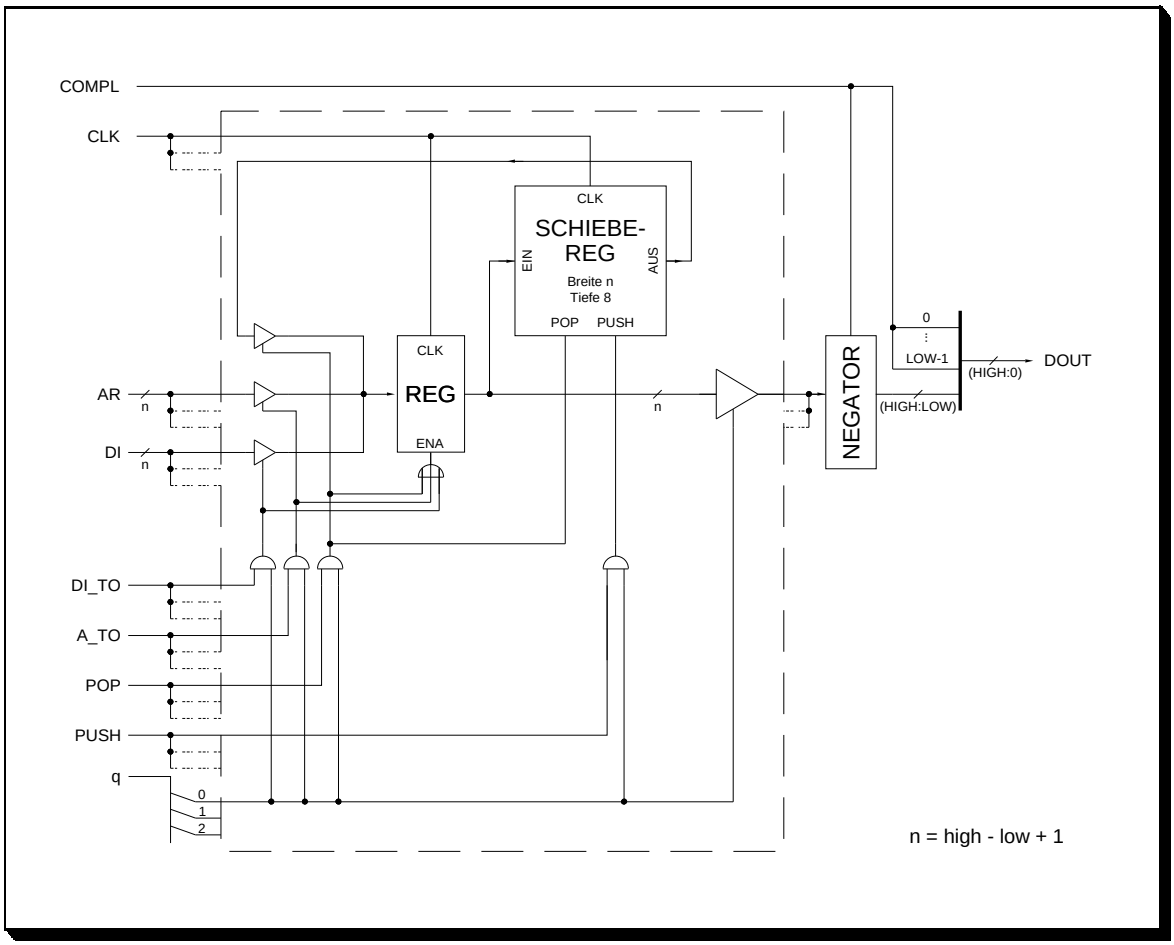


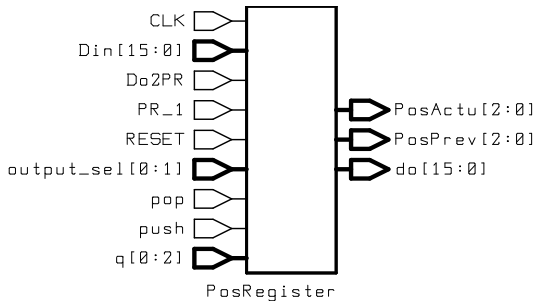
Abbildung 4.16: Register mit Stack

```
...
begin
...
  DO1: if LOW = 0 generate
          DO <= DOUT_0;        -- DOUT_0 ist Ausgang des Negators
       end generate;
  DO2: if LOW > 0 generate
          DO(HIGH downto LOW) <= DOUT_0;
          DOUTBUF_i: for I in LOW-1 downto 0 generate
                        DO(I) <= COMPL;
                     end generate;
       end generate;
  DOUT <= DO;
end TOMS;
```

Abb. 4.16 zeigt die Struktur eines Stack–Registers. Der gestrichelt eingerahmte Block ist dabei für jeden Kubusbaum einmal implementiert.

4.4.4.1.3 Positionsregister

Die Position eines jeden Kubus ist nicht nur für das Traversieren durch den Octree wichtig und wird für Transformationsoperationen benötigt, sondern sie fließt auch bei der Berechnung der Speichadressen ein. Im Chip wurde ein Register realisiert, welches die Position der Kubi entsprechend ihrem Niveau für jedes der drei Modelle auf einem Stack verwaltet. Die Stacks wurden dabei so realisiert, daß sie neben der Möglichkeit des parallelen Auslesens (zur Bestimmung des Indexvektors; vgl. 2.1.1) auch das zweite (falls vorhanden) Stackelement (zur Bestimmung der Position in den Bitblöcken; vgl. 2.1.4.1) zur Verfügung stellen.



Die Ein– und Ausgänge, sowie die Steuerleitungen des Positionsregister beziehen sich immer auf das aktuelle Modell, welches über q(0:2) entsprechend angewählt werden kann. Die Steuerleitung des Positionsregisters beeinflussen analog zu den Navigationsoperationen des Octrees den Inhalt des Registers. So wird beim Setzen der Leitung RESET der Stack des aktiven Modells entleert und die aktuelle Position auf 0 gesetzt, ganz analog zu einem root Aufruf. Diese Leitung ermöglicht es auch der RESET–Routine des μROMs, das Positionsregister zurückzusetzen, allerdings wird hierbei q=111 gesetzt um die Stacks aller Modelle zurückzusetzen. Zur Realisierung der Operationen UP und DOWN sind die Leitungen POP und PUSH vorhanden, um die aktuelle Position vom Stack zu holen bzw. sie auf dem Stack abzulegen. Im Falle eines DOWN Aufrufs muß neben der Leitung PUSH auch Do2PR aktiv sein, um das an Din anliegende Datum als neue aktuelle Position zu übernehmen. Durch alleiniges Setzen der Signalleitung Do2PR wird das an Din anliegende Datum zur aktuellen Position, entsprechend einem ACROSS–Aufruf. Bei einem NEXT wird die aktuelle Position immer um 1 erhöht, was einem Wechsel zum Nachbarknoten entspricht; das Inkrementieren der aktuellen Position im Positionsregister wird durch aktivieren der Leitung PR_1 erreicht.

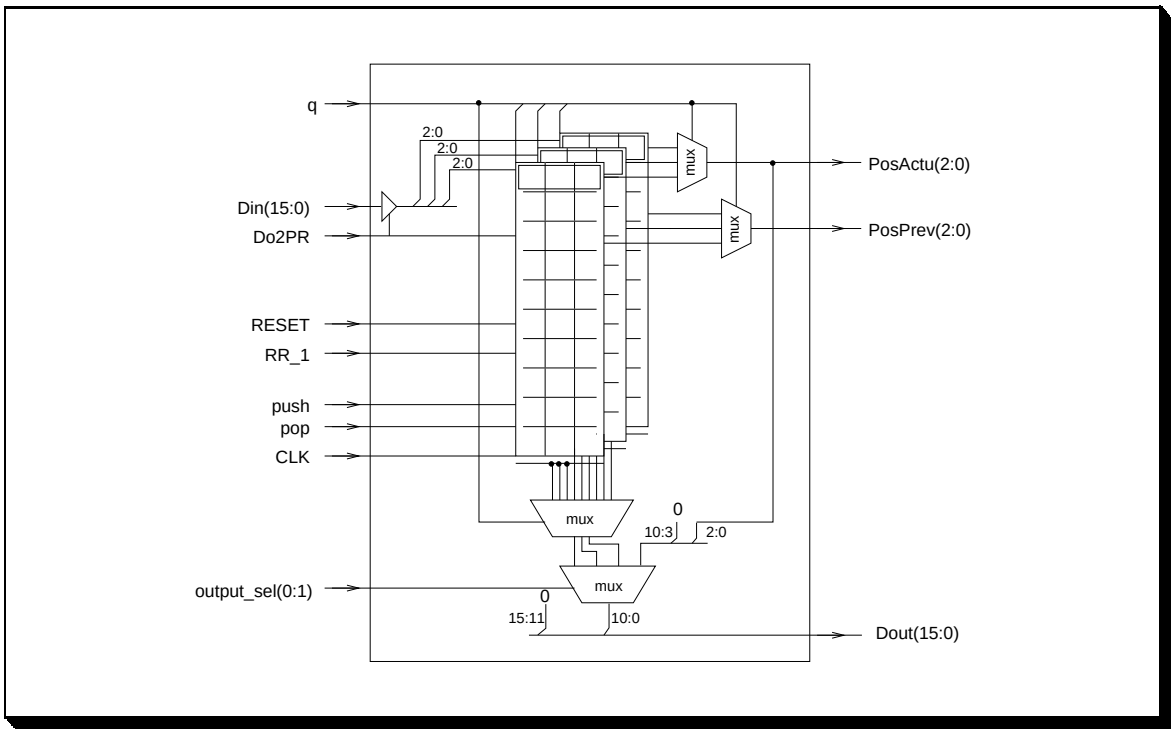


Abbildung 4.17: Interne Struktur des Positionsregisters

Die Ausgängen PosActu(2:0) und PosPrev(2:0) des Positionsregisters stellen, entsprechend ihrer Nomenklatur, die aktuelle sowie, falls vorhanden, die Position des vorherigen Niveaus[15] — sollte diese nicht vorhanden sein, da das aktuelle Niveau $n = 10$ ist, so ist dieser Ausgang hochohmig — zur Verfügung.

Die am Ausgang do anliegende Information kann über die Leitungen output_sel(0:1) entsprechend ausgewählt werden. Nebenstehende Tabelle 4.4 zeigt die Codierung für output_sel(0:1) und den entsprechenden Wert des dann an do anliegenden Datums.

output_sel(0:1)	Datum an do(15:0)
00	0000 0.....index X.....
01	0000 0.....index Y.....
10	0000 0.....index Z.....
11	0000 0000 0000 0pos

Tabelle 4.4: Codierung von output_sel(0:1)

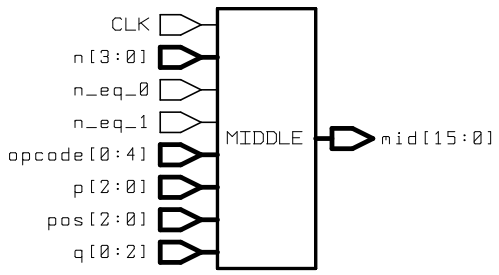
Der interne Aufbau des Positionsregisters (siehe Abbildung 4.17) gliedert sich in drei identische Einheiten, die die Verwaltung der Position für jedes Modell übernehmen. Diese drei Einheiten werden über Multiplexer, durch q(0:2) gesteuert, mit den Ein– und Ausgängen verbunden und über die entsprechende Leitung aus q(0:2) angewählt.

Die Einheiten bestehen aus einer Inkrementor und einem Stapelregister, das bei steigender Taktflanke Daten übernimmt. Das Stapelregister weist bezüglich seines Aufbaus zwei Besonderheiten auf: Falls mehr als ein Datum im Stapel gespeichert ist, so kann neben dem Zugriff auf das oberste Element auch das vorherige ausgelesen werden. Weiterhin ist es möglich den Stapel parallel auszulesen, d.h. man kann die auf dem Stapel abgelegte Folge[16] an Bits zu einer Stelle des drei–stelligen Bitwortes auslesen, ist der Stapel nicht ganz gefüllt so wird hier mit 0 aufgefüllt. Aufgrund dieser Besonderheiten wurden das Positionsregister in VHDL als Verhaltensbeschreibung realisiert, da ein Entwurf mittels einer Strukturbeschreibung sehr komplex geworden wäre.

[15] die Notwendigkeit, daß das Positionsregister dieser Information für die Bitblöcke zur Verfügung stellen muß, wurde in der Realisierung des Positionsregisters in [1] völlig vernachlässigt (vgl. auch 4.4.5.1)

[16] entspricht somit dem unter 2.1.1 beschriebenem Indexvektor

4.4.4.2 Mittelpunktsberechnung



Im Gegensatz zur Softwarelösung erlaubt die Hardwareimplementierung eine parallele Aktualisierung des Mittelpunktsvektors bei jeder Navigation, wodurch die Abfrage einer Komponente des Mittelpunktes lediglich die Zeit eines Registerzugriffs kostet. Daher wurde diese Variante der in der Softwarelösung (→ 3.3.4) verwendeten und in [1] vorgeschlagenen Methode, den Mittelpunkt erst bei Bedarf zu berechnen, vorgezogen.

Die in Abb. 4.18 skizzierte Funktionseinheit zur Berechnung der Mittelpunktskoordinaten des aktuellen Kubus ist eine direkte Umsetzung der in 2.1.3.1 hergeleiteten Formeln[17]: Zunächst wird aus den Eingaben $\vec{p}$ und $\overrightarrow{pos}$ der Vektor $\vec{r}$ berechnet. MAP ist dabei eine Implementierung der in 2.1.3.1 eingeführten Funktion[18] u; SUB dient zur Subtraktion zweier 1 Bit Zahlen, INC zur Inkrementierung einer dreistelligen Binärzahl. Die MAP und INC Blöcke liefern je eine einstellige Binärzahl plus Vorzeichenbit, der obere Multiplexer wählt hieraus abhängig vom Opcode[19] eine aus.

In den darauf folgenden Einheiten wird $\vec{r}$ entsprechend den Angaben in Tabelle 2.2 skaliert. SH_MAP bildet dabei die Binärstellen von $\vec{r}$ auf Bit 0, die Vorzeichenbits von $\vec{r}$ auf die Bits 1 . . . 15 der Eingabe des Shifters ab:

```
SHMAP: block  /* Vektor r = (cx, cy, cz) auf Shiftereingabe abbilden */
       begin
         F: for i in 1 to 15 generate
               Gx: shinx(i) <= cx(1);
               Gy: shiny(i) <= cy(1);
               Gz: shinz(i) <= cz(1);
            end generate;
         shinx(0) <= cx(0);
         shiny(0) <= cy(0);
         shinz(0) <= cz(0);
       end block;
```

Die Anzahl der Schiebeoperationen m wird von zwei Multiplexern und zwei Dekrementoren nach folgender Tabelle berechnet:

Opcode	Niveau	Anzahl Schiebeoperationen
DOWN	$n > 1$ $n = 1$	$m := n - 2$ $m := 0$
UP	$n > 0$ $n = 0$	$m := n - 1$ $m := 0$
ACROSS, NEXT		$m := n$

[17] Zu beachten ist, daß der gestrichelt eingezeichnete Block für jede Komponente des Mittelpunktsvektors einmal vorhanden ist.
[18] auch hier wird Niveau 0 als Spezialfall behandelt
[19] von oben nach unten: down, up, across, next

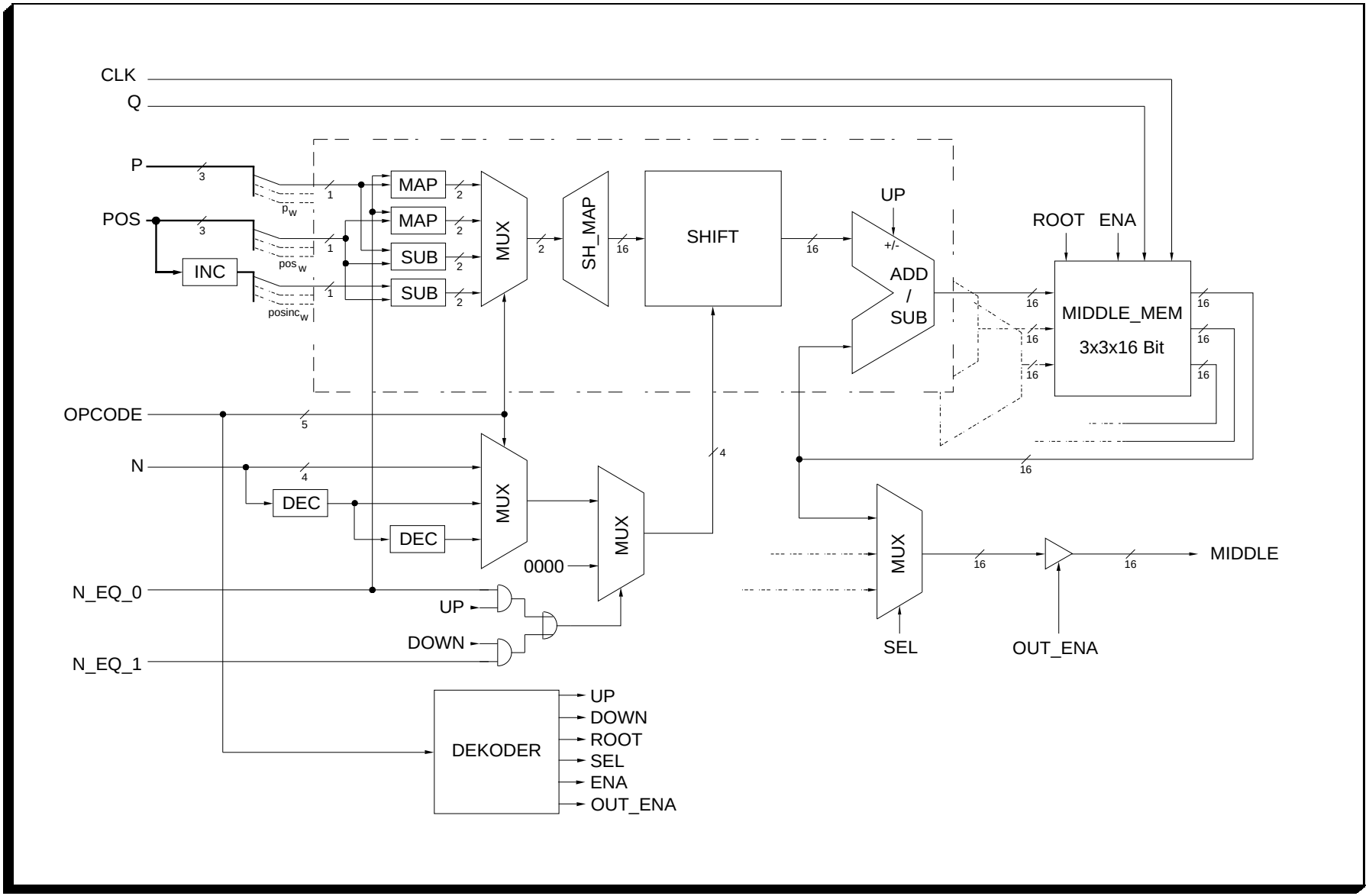


Abbildung 4.18: Mittelpunktsberechnung

Die Ausgabe des Shifters wird zum aktuellen Wert addiert — bzw. im Fall up hiervon abgezogen — und in MIDDLE_MEM gespeichert. Letztere Einheit dient zur Speicherung von 3 (Anzahl der Kubusbäume) · 3 (drei Komponenten von $\vec{m}$) Worten in Registern mit Rücksetzeingang (RESET_REG, → 4.3.2).

Die Ausgabe MIDDLE besteht aus dem mit Q und SEL gewählten Speicherwort.

Die folgenden Tabellen geben die Interpretation der Bits im Opcode–Bus der MIDDLE–Einheit sowie die von der MIDDLE–Einheit ausführbaren Operationen und die assoziierten Opcodes an.

Bit	
0	Enable
1	Reset
2	0: Navigation, 1: Ausgabe
3,4	Navigationsopcode bzw. X, Y, Z Selektor

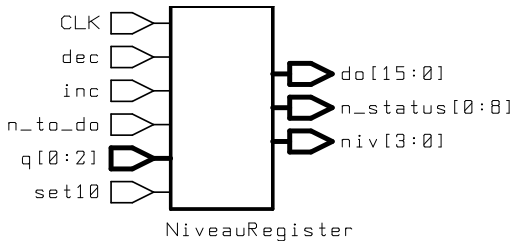
Tabelle 4.5: Belegung des Opcode–Bus der MIDDLE-Einheit

Opcode Bit 01234	Operation
0----	Nichts berechnen
110--	Reset
10000	next
10001	across
10010	down
10011	up
10100	X ausgeben
10101	Y ausgeben
10110	Z ausgeben

Tabelle 4.6: Opcodes der MIDDLE–Einheit

4.4.4.3 Niveauregister

Um die im C–Programm genutzte globale Variable Niveau auf den Chip zu übertragen wurde ein Register für das Niveau eingeführt, das über eine Reihe von Statusleitungen Steueraufgaben im Chip übernimmt.



Das Niveauregister übernimmt die Verwaltung des aktuellen Niveaus für alle drei Modelle, welche über die Leitungen q(0:2) ausgewählt werden können. Die Leitungen inc und dec sorgen für das In– bzw. Dekrementieren des Niveaus des aktuellen Modells entsprechend eines UP bzw. DOWN Aufrufes. Die Leitung set10 bewirkt ein Zurücksetzen des Niveaus für das aktuelle Modell auf 10, analog zu einem ROOT–Aufruf. Über die Ausgänge niv und do wird das Niveau des aktuellen Modells zur Verfügung gestellt, hierbei wird der Ausgang do, der mit dem Datenbus verbunden ist, über ein Tri–State mittels n_to_do geschaltet; der Ausgang niv stellt immer das aktuelle Niveau zur Verfügung.

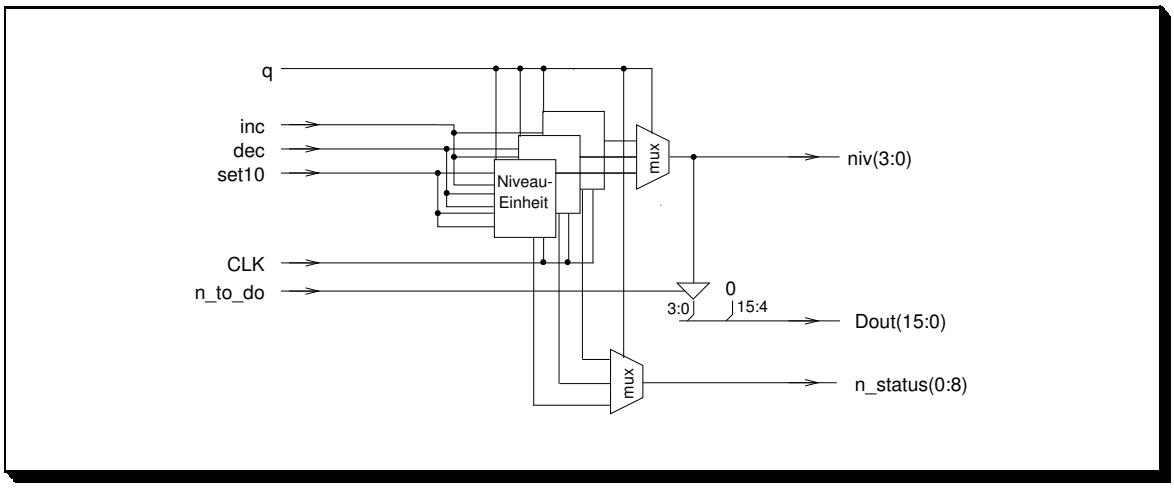


Abbildung 4.19: Interne Struktur des Niveauregisters

Der eigentliche Kern des Niveauregisters besteht aus drei identischen Blöcken, den Niveau–Einheiten. Diese drei Niveau–Einheiten werden über Multiplexer, die über q(0:2) gesteuert werden, mit den Ein– und Ausgängen des Niveauregisters verbunden; Abbildung 4.19 zeigt die interne Struktur des Registers mit den Multiplexern.

Eine Niveau–Einheit (vgl. Abbildung 4.20) besteht aus einem Speicherelement, einem In–/Dekrementor, einer Einheit die konstant 10 liefert sowie einer Komperatorlogik. Das Speicherelement übernimmt bei steigender Taktflanke an CLK und aktiver enable–Leitung Daten. Die enable Leitung wird aktiv, wenn die entsprechende q Leitung dieser Einheit **und** eine Operation (inc, dec **oder** set10) ausgewählt wurde. Der In–/Dekrementorbaustein modifiziert den aktuellen Inhalt des Speicherelements, ebenso wird die Konstante bei Aktivierung an das Speicherelement geliefert. Die Komperatorlogik setzt die jeweiligen Statusleitungen (in n_status(0:8)) dem aktuellem Inhalt des Speicherlements (dem Niveau) entsprechend.

Die Leitungen n_status(0:8) geben also ständig Aufschluß über das aktuelle Niveau und ermöglichen so einige gesteuerte Mikroprogrammschritte, bzw. regeln Abläufe in Funktionseinheiten (Bitblockregister, middle–Einheit und FullEmpty); über die Belegung der einzelnen Leitungen gibt Tabelle 4.7 näheren Aufschluß.

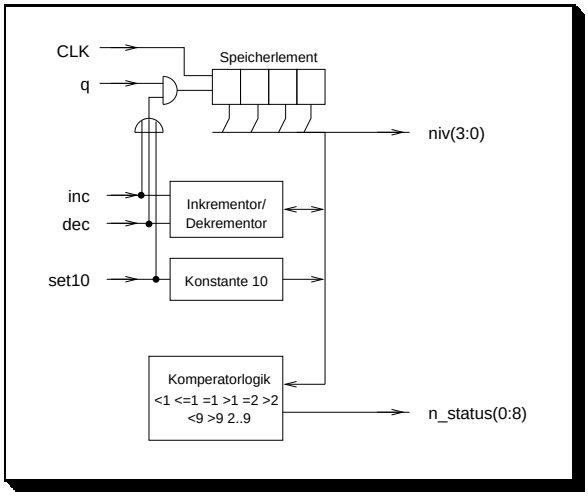


Abbildung 4.20: Aufbau einer Niveau–Einheit

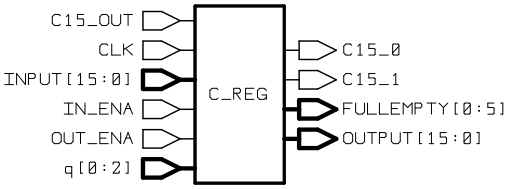
Leitungsnummer in n_status(0:8)	Zustand des Niveaus n
0	$n < 1$
1	$n <= 1$
2	$n = 1$
3	$n > 1$
4	$n = 2$
5	$n > 2$
6	$n < 9$
7	$n > 9$
8	$n = 2 \cdots 9$

Tabelle 4.7: Belegung von n_satus(0:8)

Eine Sonderrolle fällt noch dem RESET zu, für diesen Fall wird nämlich durch die RESET–Routine des μROM $q = 111$ gesetzt, was zur Folge hat, daß alle Niveau–Einheiten aktiv sind. Gleichzeitig ist die Signalleitung set10 aktiviert, was dazu führt, daß das Niveauregister für alle Modell auf $n = 10$ gesetzt wird.

Abschließend sei noch erwähnt, daß das Niveauregister als Verhaltensbeschreibung in VHDL realisiert wurde und die Niveau-Einheit einzeln beschrieben und dreimal instantiiert wurde.

4.4.4.4 Knoteninhalte



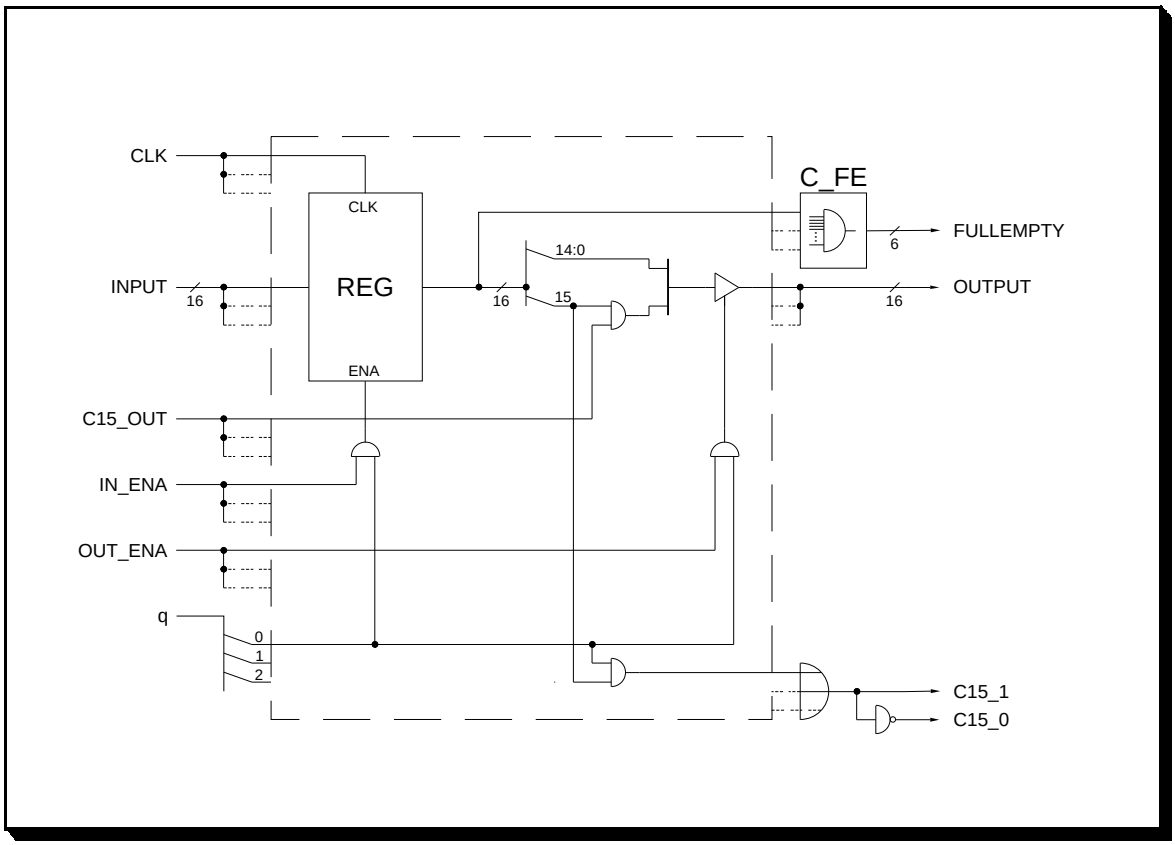


Abbildung 4.21: Register für Knoteninhalte

Zur Speicherung der aktuellen Knoten der drei Kubusbäume dient die in Abb. 4.21 skizzierte Komponente C_REG, die im wesentlichen aus drei 16 Bit Registern mit adäquater Treiberlogik besteht.

Wie bereits in 2.1.5 erläutert, wird ein Knoteninhalt von 0000_{hex} als „leer“, FFFF_{hex} als „voll“ interpretiert, in allen anderen Fällen liegen geteilte Kubi vor und der gespeicherte Wert gibt die *relative* Adresse des Speicherblocks an, in dem die Söhne des aktuellen Knotens zu finden sind. Dabei wird anhand des führenden Bits zwischen direkter und indirekter Folgeadresse unterschieden. Aus diesem Grund wird dieses Bit im Inhaltsregister gesondert behandelt: Zum einen ist die Möglichkeit vorgesehen, das Register ohne dieses Bit auszulesen — es wird nur bei bei C15_OUT = '1' ausgegeben —, zum anderen steht das zum aktuellen Baum gehörende Bit[20]: $C(q)_{15}$ im Rechnenwerk explizit als Teil des Statusbus STATE zur Verfügung.

Ein weiterer Bestandteil des Knotenregisters ist die kombinatorische Schaltung C_FE, die aus den Registerinhalten $C(q)$, $q \in \{0, 1, 2\}$ den Status $FULL(q)$ und $EMPTY(q)$ berechnen. Die Kompo-

[20] A_i bezeichne das Bit i des Wortes A

nente fullempty (→ 4.4.4.4.1) berechnet aus diesem Statuswort in Abhängigkeit des Baumselektors q und des Niveaus die Statusleitungen FULL und EMPTY.

$$FULL(q) = \bigwedge_{i=0}^{15} C(q)_i \qquad EMPTY(q) = \bigwedge_{i=0}^{15} \neg C(q)_i = \neg \bigvee_{i=0}^{15} C(q)_i$$

4.4.4.4.1 Full–Empty Einheit

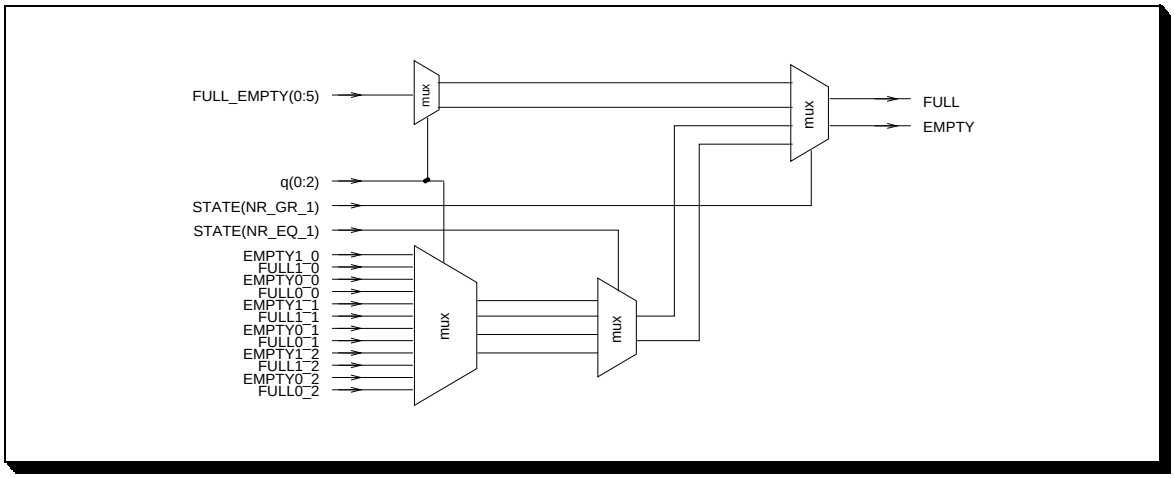


Abbildung 4.22: Aufbau von FullEmpty

Abbildung 4.22 zeigt den strukturellen Aufbau der Einheit FullEmpty. Die Einheit FullEmpty wählt aus den Statusleitungen für FULL und EMPTY des C–Registers und der Bitblockregister in Abhängigkeit des Niveaus und von q die gültigen aus und legt den entsprechenden Wert an die Ausgänge FULL und EMPTY. Aus den sechs Leitungen des C–Registers werden durch einen Multiplexer die zwei ausgewählt, die für das aktuelle q Gültigkeit haben. Ebenso wird mit den 12 Leitungen der Bitblockregister verfahren, nur hier sind es vier Leitungen die weiterführen (FULL bzw. EMPTY für Niveau $n = 0$ bzw. $n = 1$). Ein weitere Multiplexer, der über die Statusleitung STATE(NR_EQ_1) des Niveauregisters gesteuert wird, leitet die dem Niveau entsprechenden Daten weiter. Diese führen zusammen mit den Leitungen des Multiplexers, der die entsprechenden Leitungen des C–Registers ausgewählt hat, in einen weiteren Multiplexer. Dieser entscheidet nun wiederum über eine Statusleitung des Niveauregisters (STATE(NR_GR_1))gesteuert, ob die Informationen der Bitblockregister oder des C–Registers gültig sind.

4.4.5 Bitblocklogik

Damit die Bitblöcke im Chip realisiert werden können sind, vier Bitblockregister sowie eine Bitblockalu im Chip integriert. Für jedes Modell gibt es ein Register, zusätzlich ist noch ein Hilfs–Bitblockregister vorhanden, um das gekoppelte Schieben bei der Move–Routine zu realisieren, oder um als Zwischenspeicher für Operationen zu dienen. Die Register für die Modelle 0 und 1 ermöglichen keine Modifikationen ihres Inhaltes. Das Register des Modells 2 hingegen läßt diese Modifikationen zu, die dann auch in den Speicher zurück geschrieben werden. Diese Realisierung entspricht den Konventionen, das die Modelle 0 und 1 als Operanden dienen und dabei selbst nicht modifiziert werden sollen, wohingegen das Modell 2 als Ergebnis– bzw. Zielmodell dient.

Der strukturelle Aufbau der Bitblöcke untereinander und die Verbindung mit der Bitblockalu gestaltet sich recht einfach. Alle Register sind mit Din verbunden und über Tristate–Treiber an Dout gelegt, um so eine Verbindung zum internen Datenbus herzustellen; gleichzeitig führen die Ausgänge der

Bitblockregister zur Bitblockalu, deren Ausgang wiederum zurück an Din führt und so Operationen der Bitblöcke untereinander ermöglicht.

Die Abbildung 4.23 zeigt die Zusammenschaltung der einzelnen Bitblockregister und der Bitblockalu sowie die Steuerleitungen vom Rechenwerk und die Anbindung an den Datenbus.

Die Eingliederung der Bitblocklogik in die Struktur der einzelnen Funktionen des Mikroprogramms sei hier nur kurz vorgestellt. Ausführlich wird das Mikroprogramm und seine Funktionalität in 4.6.3 beschrieben. Dort werden auch die Routinen für die Transformationen mit Bitblöcken genauer dargestellt. Im folgenden soll nur ein kurzer Abriß der Vorgehensweise gegeben werden, wenn beim Traversieren des Octrees das Bitblockniveau erreicht bzw. verlassen wird, und wie Transformationen mit der Bitblocklogik durchzuführen sind.

Sobald beim Durchlaufen eines Baumes das Bitblockniveau $n = 2$ erreicht ist, wird nicht mehr das C–Register mit der nächsten Speicherzelle geladen, sondern das Bitblockregister des aktuellen Modells wird in den darauffolgenden vier Taktzyklen mit dem entsprechenden Inhalt geladen. Zum Laden eines Bitblockregisters wird die SHIFT–Leitung des entsprechenden Registers aktiviert und das Register „schiebt“ den an Din liegende Wert in seinen Speicher. Da ein Bitblock in vier Speicherworten abgelegt ist, sorgt die Zugriffslogik für das Ansprechen der Speicherzellen und das Bitblockregister ist nach weitern drei Schiebeoperationen mit einem gültigem Bitblock geladen. Per Definition wird ein Register immer „vorwärts“ in X–Achsen Richtung (Angabe der Richtung über back_FORW und Achse über axe(0:2)[21]) geladen.

Nachfolgende DOWN–, NEXT– oder ACROSS–Aufrufe haben keinen weiteren Speicherzugriff mehr zur Folge, da der Bitblock die gesamten Knoteninformationen der unteren beiden Niveaustufen beinhaltet, was in den Mikroprogrammschritten über Niveauabfragen entsprechend gesteuert wird (vgl. 4.6.2). Niveau, Mittelpunkt und Position werden auch weiterhin aktualisiert; ein Speicherzugriff für dieses Modell wird allerdings erst wieder bei einem UP–Aufruf im Niveau $n = 1$ bzw. bei einem ROOT–Aufruf angestoßen, da hier wieder auf den Inhalt des C–Registers zurückgeriffen werden muß. Bevor allerdings das C–Register geladen wird, muß für das Bitblockregister des Modells $q = 2$ im Falle von Modifikationen der neue Inhalt in den Speicher zurückgeschrieben[22] werden. Wenn dieses Modell modifiziert wurde, ist das Statusbit STATE(BC_mod) gesetzt. Der Vorgang des Schreibens in den Speicher verläuft ähnlich dem des Lesens: Über die Leitung B2_TO_DO wird der Ausgang des Registers auf Dout gelegt und in vier Schritten wird über ROLL der Inhalt des Registers ebenenweise wieder vorwärts in X–Achsen Richtung ausgegeben, die Zugriffslogik sorgt für das Ansprechen der richtigen Speicheradressen. Danach wird das C–Register mit dem entsprechenden Inhalt geladen und STATE(BC_mod) wird wieder zurückgesetzt.

Das Flag STATE(BC_mod) wird bei allen Bitblock–Operationen, die das Modell $q = 2$ als Ziel haben[23] gesetzt, dies sind neben Transformationsoperationen und mengentheoretischen Verknüpfungen auch jene, die im Bitblock des Modells $q = 2$ einen Knoten auf FULL oder EMPTY setzen.

Auf Bitblockniveau aufgerufen bewirken die elementaren Operationen set_empty und set_full ein Aktivieren der Leitung S_EMPTY bzw. S_FULL des Bitblockregisters 2, welches daraufhin die entsprechenden Veränderungen im Bitblock durchführt.

Bei Transformationsoperationen werden die Steuerleitungen für Richtung und Achse des Quell– und Zielbitblocks[24] mit den entsprechenden Werten gesetzt (vgl. 2.2.3 und 3.4.2). Nun wird die Bitblockalu so geschaltet, daß die Daten am Ausgang des Quellbitblocks wieder an Din anliegen. Am Quellbitblockregister wird nun die ROLL– und am Zielbitblockregister die SHIFT–Leitung aktiviert. In den folgenden vier Taktzyklen schiebt das Quellbitblockregister immer ein Datum heraus. Dieses wird durch die Bitblockalu zurück an Din geführt[25], von wo es in das Zielbitblockregister hereingeschoben wird. Bei mengentheoretische Verknüpfungen liefert ein zweites Quellbitblockregister, auch mittels ROLL–Leitung angesteuert, zusätzlich Daten. Die Bitblockalu führt nun eine logische Verknüpfung durch und das Ergebnis wird an Din bereitgestellt, von wo es in den Zielbitblock gelangt.

[21] Den Autoren ist bekannt, daß in der Englischen Sprache „Achse“ mit ‚axis‘ zu übersetzen ist.
[22] Der Fall eines ROOT–Aufrufs wurde in [1] nicht beachtet!
[23] Wir erinnern uns, Modifikationen sind nur im Modell q=2 möglich.
[24] In der Vorlage [1] sind hier Modell $q = 0$ als Quelle und $q = 2$ als Ziel vorgesehen.
[25] Je nach Transformation kann es sein, daß die Daten vorher in der ALU rotiert wurden.

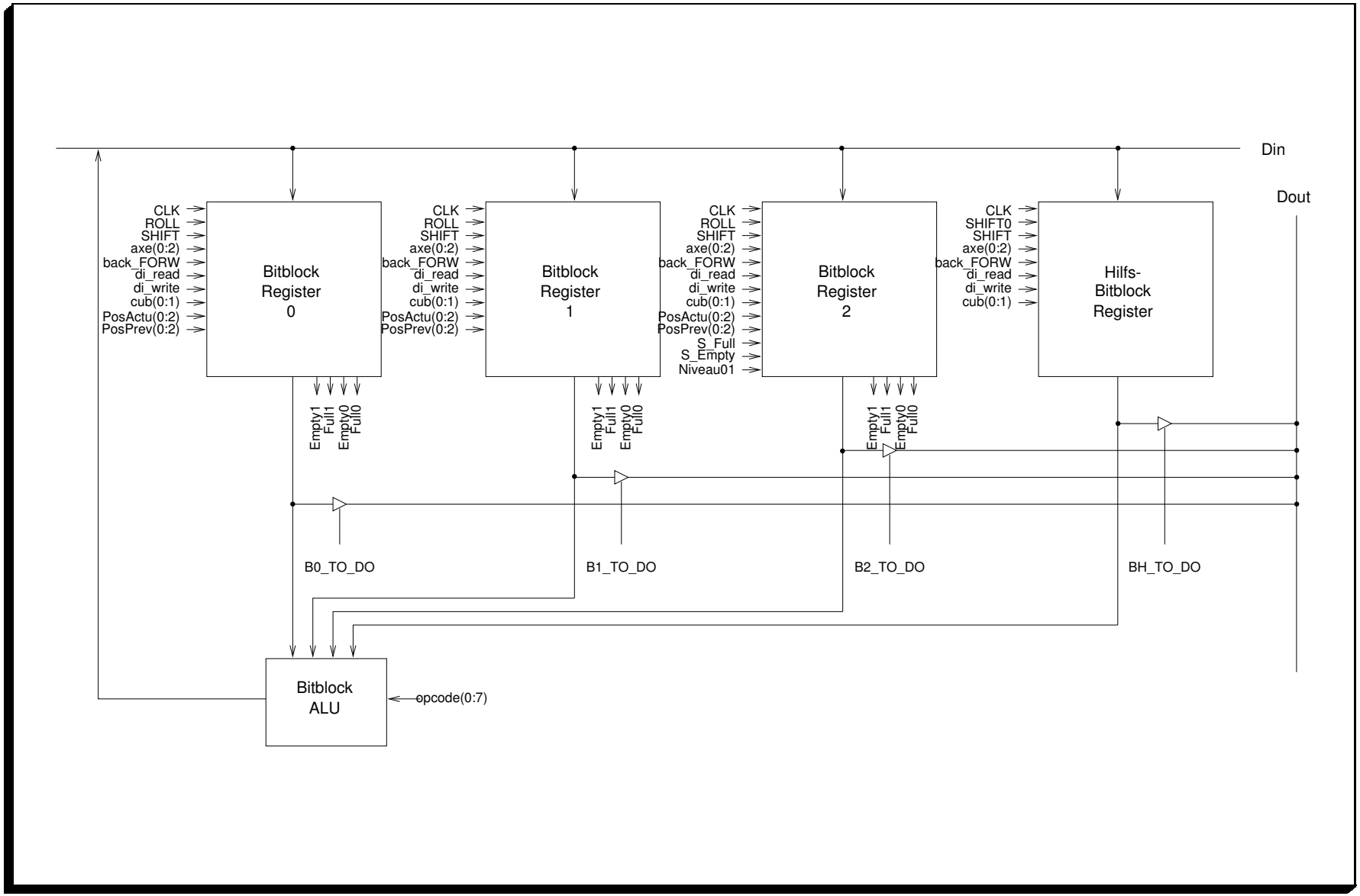


Abbildung 4.23: Strukturaufbau der Bitblockeinheit

Bei mengentheoretischen Verknüpfungen ist noch zu beachten, daß alle Bitblockregister die Rollen– bzw. Schiebeoperationen bezüglich der gleichen Achse und Richtung ausführen.

Damit diese Operationen korrekt durchgeführt werden, müssen die Bitblockregister bestimmte Anforderungen bezüglich der Schiebeoperationen erfüllen. Die Daten bezüglich Schiebeachse und Richtung müssen nach kurzer Verzögerungszeit am Ausgang zur Verfügung stehen, so daß diese Daten, auch nach Verzögerung durch die Bitblockalu, lange genug stabil an den Eingängen der anderen Bitblockregister anliegen, damit sie von dort übernommen werden können. Desweitern müssen die Register die Daten trotz der internen Schiebevorgänge stabil am Ausgang halten, so daß keine Einschwingvorgängen, sogenannten critical races, auftreten. Die Realisierung der Register wird im folgenden beschrieben.

4.4.5.1 Bitblockregister

Die Umsetzung der Bitblöcke und der damit verbundenen Funktionen in VHDL wurde angelehnt an das C–Programm und die dabei gewonnenen Erfahrungen durchgeführt, d.h. jeder Bitblock besteht aus vier YZ–Ebenen entsprechend den 16–Bit Worten (Integer) in C (vergl. 2.1.4, 3.3.6). Die Komponenten wurden als reine Verhaltensbeschreibung realisiert, da zur Umsetzung der von Bitblöcken unterstützten Funktionen (Rollen, Schieben längs der Achsen in beide Richtungen) ein Ansprechen jeden elementaren Kubus (in Hardware einem Speicherelement entsprechend) notwendig ist und dies für eine Strukturbeschreibung zu unübersichtlich bzw. zu fehlerbehaftet geworden wäre.

Die Realisierungen der im C–Programm durch globale Variablen dargestellten Bitblöcke erfolgt in VHDL durch vier Bitblockregister. Zwei Register für die Modelle 0 und 1, ein Register für das Modell 2 und ein Hilfsregister. Im Gegensatz zu der Softwarelösung, bei der über entsprechende Funktionsaufrufe die Schiebeoperationen eines Bitblockes durchgeführt wurden, ist diese Funktionalität beim Hardware–Entwurf im Register umgesetzt worden. Diese Funktionalität setzt sich in der Realisierung der in der Definition der einzelnen Modelle vorgegebenen[26] Operationen fort, was zur Folge hat, daß die einzelnen Bitblockregister unterschiedlich komplex realisiert worden sind. Im Großen und Ganzen ist in allen drei Arten der Bitblockregister das zentrale Speicherelement, das eigentliche Register, gleich konstruiert. Zu diesem kommt noch eine Logikeinheit, die das zu rollende bzw. schiebende Element zur Verfügung stellt (die Einheit bb_shifter), sowie eine Einheit die den Inhalts des aktuellen Kubus (die Einheit bb_fullempty) berrechnet.

Das Speicherelement (Abbildung 4.24) der Bitblockregister ist so aufgebaut, daß es bei einer steigenden CLK–Flanke Daten an die eigentlichen Speicherelemente (QUBE(0)[0:15] bis QUBE(4)[0:15]) übergibt. Der aktuelle Inhalt dieser Speicherelemente steht über die Leitungen h0 bis h3 im gesamten Bitblockregister zur Verfügung. Die anderen Komponenten eines Bitblockregisters greifen so auf den aktuellen Inhalt zurück. Das Speicherelement übernimmt das an Cin anliegende Datum entsprechend den Steuerleitungen in die Speicherzellen. Für den Fall, daß die Leitungen ROLL **oder** SHIFT aktiv sind muß das an Cin anliegende Datum entsprechend den Werten für die Achse (axe(2:0)) und Richtung (back_FORW) in den Speicherzellen abgelegt und die anderen Speicherzellen müssen entsprechend weitergeschoben werden (vergl. 2.2.3). Im Speicherelement des Modells 2 ist ein Baustein realisiert, der das Setzen bzw. Löschen eines Kubus ermöglicht; sollten die hierfür vorgesehenen Steuerleitungen (S_FULL und S_EMPTY) aktiv sein, so entfällt die Übernahme des an Cin anliegenden Datums und der nachfolgende Schiebevorgang der Speicherelemente. Über die Leitung Niveau01 wird entschieden, ob der zu setzende Kubus auf Niveau $n = 1$ (Niveau01=1) oder auf Niveau $n = 0$ (Niveau01=0) ist und entsprechend den Angaben für die aktuelle Position (PosActu(2:0)) und vorherige Position (PosPrev(2:0)) werden die jeweiligen Speicherzellen auf 1 bzw. 0 gesetzt (vergl. 2.1.4.1).

Die Möglichkeit des direkten Zugriffs auf die einzelnen Ebenen des Registers ist in allen vier Modellen realisiert. Die Auswahl der jeweiligen YZ–Ebene erfolgt über cub(1:0) und erlaubt es das an Cin anliegende Datum mittels di_write direkt zu schreiben. Sollte diese Möglichkeit des Registerzugriffs aktiv sein, so werden alle anderen Funktionen des Speicherelements (Rollen und im Modell 2 auch das

[26] Für Modelle 0 und 1 Schieben und Rollen, sowie Informationen über den Inhalt; bei Modell 2 kommt zusätzlich noch das Setzen und Löschen der Kubi hinzu; das Hilfsregister ermöglicht außer dem Schieben keine weiteren Funktionen.

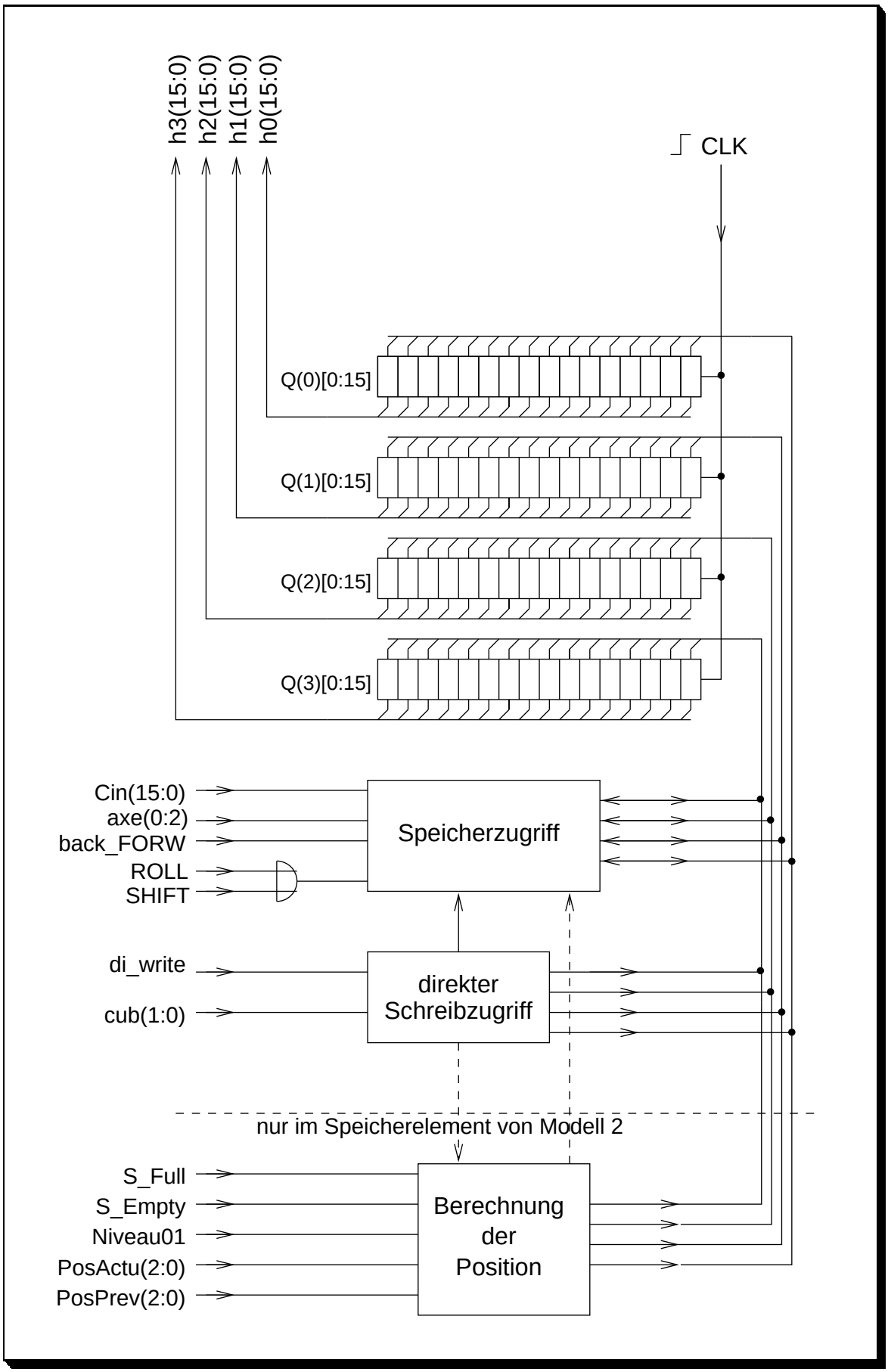


Abbildung 4.24: Aufbau des Speicherelements

Setzen bzw. Löschen) nicht beachtet. Der direkte lesende Zugriff wird von dem Baustein bb_shifter realisiert, der im nachfolgendem beschrieben wird.

Der Baustein bb_shifter (siehe Abbildung 4.25) wählt aus din(15:0) und dem aktuellen Inhalt des Speicherelements (über h0 bis h3), die Daten aus, die an das Speicherelement weitergereicht werden, und die, die an Dout zur Verfügung gestellt werden. Entsprechend den Werten von axe(0:2) und back_FORW werden aus h0 bis h3 die zu dieser Schiebe–Operation passenden Daten ausgewählt. Das so entstandene Datum wird an Dout bereitgestellt und im Falle einer „Rollen"-Operation über Cin auch wieder an das Speicherelement weitergereicht, sollte statt des Rollens das Schieben des Registers vorgesehen sein, so wird stattdessen Din direkt an Cin geschaltet; im Hilfsbitblockregister, bei dem das Rollen entfällt wird hier 0 an Cin gelegt. Die Ausgangsleitungen Cin sind mit Hilfsregistern gepuffert, so daß bei der Weiterführung der Daten an das Speicherelement Einschwingvorgänge abgefangen und so critical races vermieden werden.

Im Falle einer Setzen–/Löschen–Operation eines Kubus im Modell 2 werden keine der zuvor beschriebenen Operationen ausgeführt, da auch das Speicherelement in diesem Falle keine andere Operation ausführt. Der Ausgang Dout wird in diesem Fall hochohmig. Eine Ausnahme bildet in diesem Baustein, wie auch schon bei dem Speicherelement, die Funktion des direkten Zugriffs, in diesem Fall des direkten Lesens. Hierbei werden die Rollen–/Schiebe–Operationen bzw. das Setzen/Löschen eines Kubis nicht ausgeführt, sondern am Ausgang wird die über cub(1:0) angewählte Speicherebene bereit gestellt.

Dem direkten Zugriff fällt, wie man bei der Beschreibung des Speicherelements und von bb_shifter schon erkennen konnte, eine Sonderrolle zu. Diese Möglichkeit des Zugriffs dient zur Fehlersuche und –korrektur und soll hier kurz spezifiert werden. Im Falle des C–Programms war es möglich sich an den kritischen Stellen den Inhalt der einzelenen Variablen auf dem Bildschirm ausgeben zu lassen oder mittels eines Debuggers zu überwachen. Bei einem VHDL–Entwurf hat man auch die Möglichkeit sich entsprechende Werte von Signalleitungen und Speicherzellen anzeigen zu lassen. Diese Möglichkeit des direkten Zugriffs gestattet es aber auch bei einem fertigen Chip die Inhalte der Register auszulesen oder für einen gewünschten Startzustand entsprechend zu schreiben. So kann man auch später bei der Verwendung des Chips die Richtigkeit von Anwendungen überprüfen.

Über die Leitungen di_write und di_read aktiviert man den direkten Zugriff, entsprechend der Signalleitung schreibend oder lesend, auf die über cub(1:0) angewählte Speicherebene (nebenstehende Tabelle verdeutlicht die Zuordnung).

cub(1:0)	**entspr. Speicherebene**
00	CUBE(0)
01	CUBE(1)
10	CUBE(2)
11	CUBE(3)

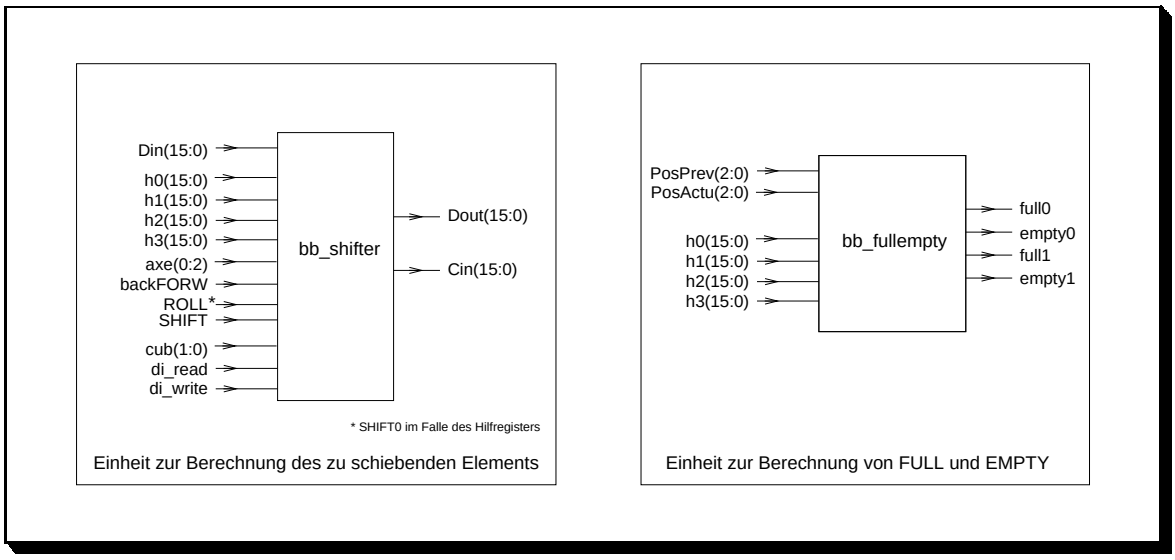


Abbildung 4.25: weitere Komponenten eines Bitblockregisters

Als weiters Grundelement der Bitblockregister gibt es die Einheit bb_fullempty (siehe Abbildung 4.25), welche aus den Signalleitungen h0 bis h3 und den Positionsangaben von PosActu(2:0) und PosPrev(2:0) den Zustand des aktuellen Kubis angibt. Da dieser Einheit das aktuelle Niveau nicht zur Verfügung steht, wird aus den vorhandenen Informationen der Zustand des Kubis für Niveau 0 und 1 berechnet (vergl. 2.1.4.1). Das Ergebnis liegt an vier Leitungen an und muß von einem nachgeschalteten Baustein (siehe 4.4.4.4.1) dem das Niveau zur Verfügung steht entsprechend bewertet werden. full1 und empty1 geben für das Niveau 1 den entsprechenden Zustand an, sind beide Leitungen auf 0 gesetzt, so ist der aktuelle Kubi SPLIT; für das Niveau 0 sind dies die Leitungen full0 und empty0.

4.4.5.1.1 Hilfs–Bitblockregister

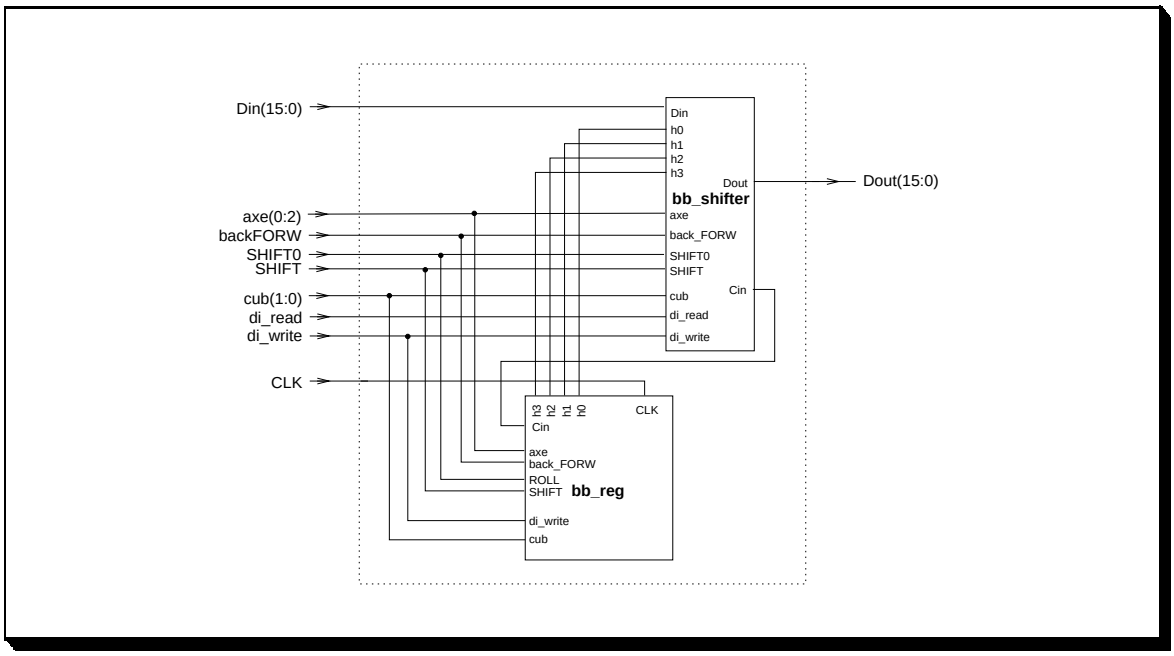


Abbildung 4.26: Elemente des Hilfs–Bitblockregisters

Das Hilfsbitblockregister, welches nur zur Unterstützung des gekoppelten Schiebens für die Move–Operation (vgl. 3.4.1.4)realisiert wurde — es kann aber auch zum Zwischenspeichern von Ergebnissen genutzt werden — ist am einfachsten aufgebaut, beinhaltet aber auch einige Abweichungen vom Entwurf der anderen Register. Es besteht eigentlich nur aus dem Speicherelement und dem Logikbaustein bb_shifter, der das Herausschieben des Registers in jede Richtung bezüglich der Koordinatenachsen realisiert. Auf eine Logik zur Berechnung von FULL und EMPTY für die aktuelle Position wurde verzichtet. Da das Hilfsbitblockregister für das gekoppelte Schieben realisiert wurde, sei hier dieser Vorgang kurz aufgezeichnet (Abbildung 4.27): Aus dem Hilfsregister wird der Inhalt „schrittweise“ herausgeschoben, die Bitblockalu führt diese Daten zurück an Din, von dort gelangen sie in das Bitblockregister 2. Nun muß das Hilfsbitblockregister mit 0 aufgefüllt werden, da dieser Wert aber nicht von Din kommen kann, ist im Hilfsbitblockregister ein Hereinschieben von 0 direkt über eine Steuerleitung anwählbar; hier weicht der Baustein bb_shifter etwas von denen der übrigen Bitblockregister ab, es wurde auf die Realisierung des Rollens verzichtet, statt des herausgeschobenen Datums wird konstant 0 hereingeschoben[27]. Die Besonderheiten der Elemente eines Bitblockregisters bezüglich des Hilfsregisters ist ja schon bei der Beschreibung dieser erwähnt worden; Abbildung 4.26 zeigt wie die Elemente im Falle des Hilfsbitblockregisters zusammengeschaltet wurden.

[27] der verbleibende Inhalt (es wird nicht immer viermal geschoben) des Hilfsregisters wird abschließend mit dem Inhalt eines anderen Kubis ODER verknüpft und 0 ist das Neutrale–Element bezüglich ODER

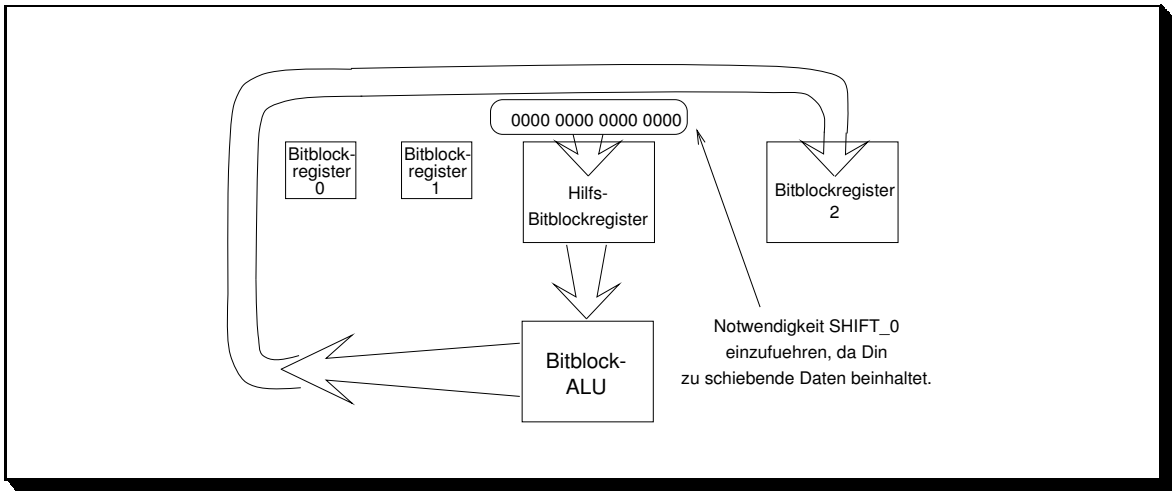


Abbildung 4.27: Verfahren des gekoppelten Schiebens

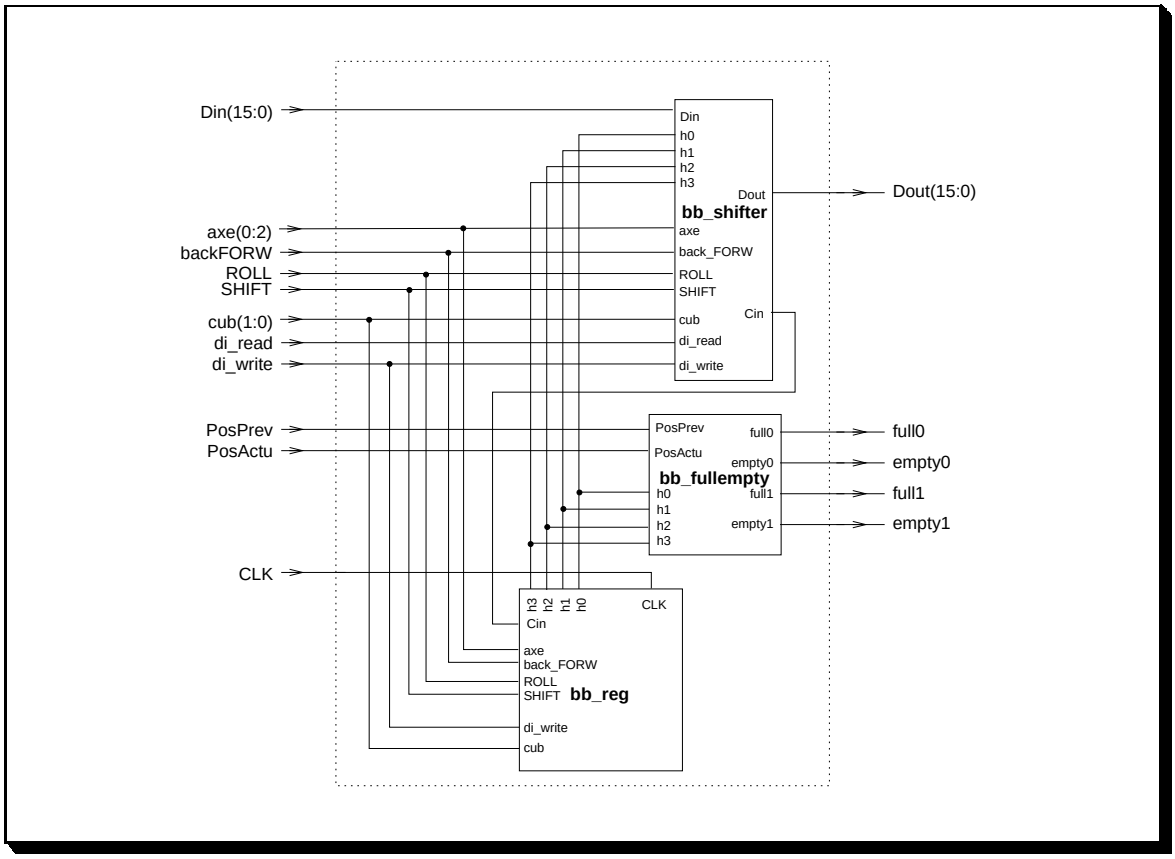


Abbildung 4.28: Elemente der Bitblockregisters der Modelle 0 und 1

4.4.5.1.2 Bitblockregister der Modelle 0 und 1

Die Bitblockregister für die Modelle q=0 und q=1 sind identisch aufgebaut (siehe Abbildung 4.28). Da Modifikationen für diese Modelle nicht vorgesehen sind, fehlt hier auch die entsprechende Funktionalität, die ein Setzen bzw. Löschen eines Kubis ermöglicht. Auch die Transforamtionsoperationen mit diesen Modellen als Quelle sind im μROM so implemeniert, daß immer ein komplettes Rollen des Registers durchgeführt wird, d.h. der Bitblock wird einmal ganz durchrotierte und ist am Ende wieder im Ausgangszustand. Abbildung 4.28 zeigt die Zusammenschaltung der zuvor beschriebenen Einheiten wie sie zur Realisierung der Bitblockregister der Modelle 0 und 1 benötigt werden.

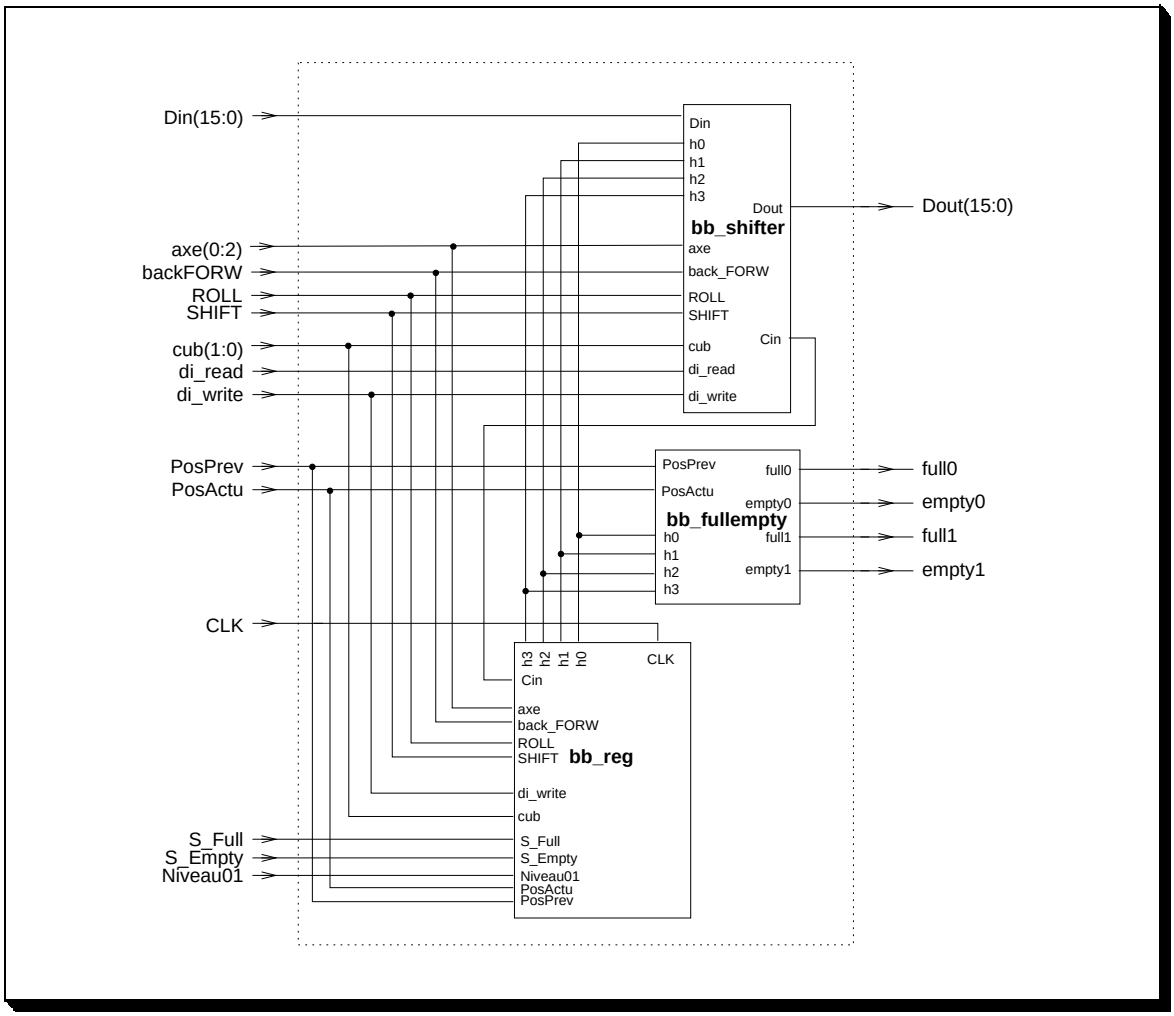


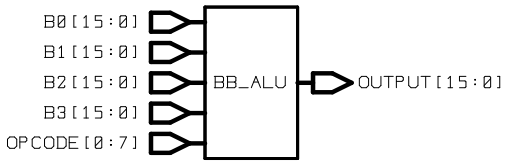
Abbildung 4.29: Elemente des Bitblockregisters für Modelle 2

4.4.5.1.3 Bitblockregister des Modells 2

Das Bitblockregister für $q = 2$ zeichnet sich durch die größte Funktionalität aus, es ist das einzige Register, das Modifikationen seines Inhaltes durch explizite Chip Befehle ermöglicht[28].Abbildung 4.29 verdeutlicht die Integration der einzelnen Komponenten des Registers.

[28] das direkte Schreiben von Bitblockebenen für die anderen Register bildet hier eine Ausnahme bezüglich der üblichen Konvention, das Schreiben nur in Modell 2 möglich ist

4.4.5.2 Bitblock–ALU



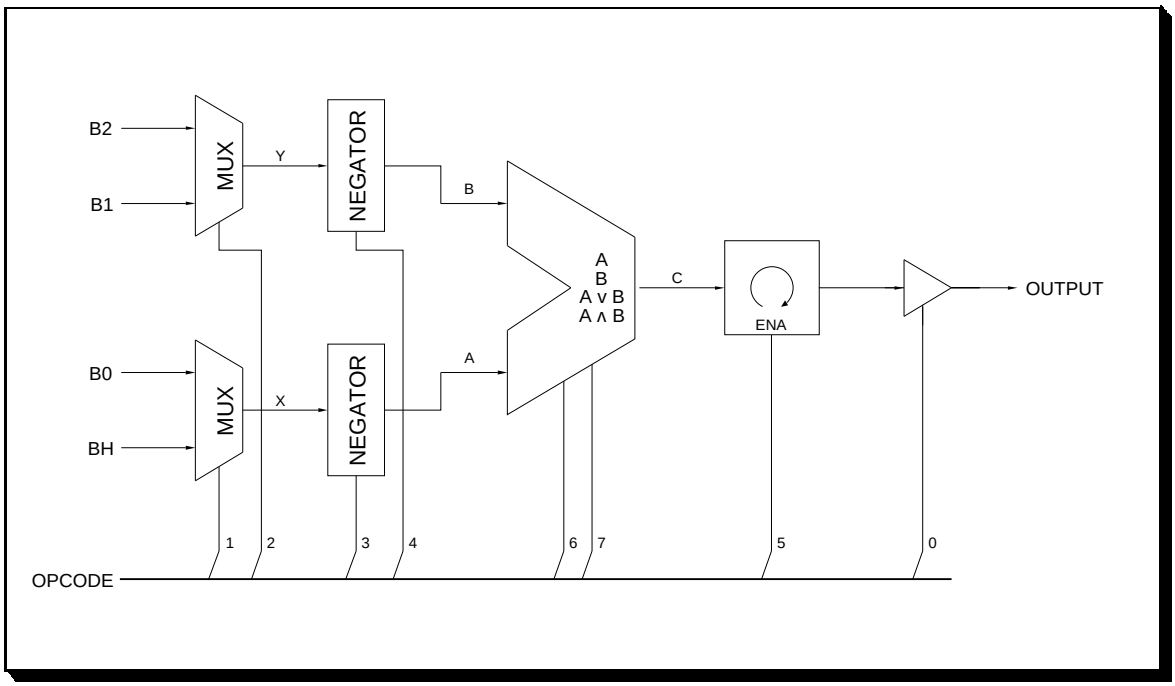


Abbildung 4.30: Bitblockalu

Die Bitblockalu wurde entworfen, um Bitblöcke logisch verknüpfen zu können. Dabei wurde das Ziel verfolgt, eine hohe Flexibilität bezüglich der durchführbaren Verknüpfungen zu erreichen: Die Menge der zur Verfügung gestellten Operationen sollte eine echte Obermenge der von den High–Level–Operationen geforderten bilden, um zukünftige Weiterentwicklungen der Software nicht einzuschränken. So entstand das Konzept einer „benutzerprogrammierbaren“ Bitblockalu: Bei Bitblockverknüpfungen wird der Opcode der Alu vom Benutzer als Parameter übergeben; ein entsprechendes Mikroprogramm steuert das Rotieren der Bitblöcke. Bei Bedarf kann die Bitblockalu jedoch auch vom Mikroprogramm gesteuert werden (→ 4.4.2). Zu den logischen Verknüpfungen wurde auch die Option des Rotierens[29] eines Bitblockdatums (4.4.5.2.1) in die Alu integriert.

Die Struktur der Alu läßt sich in 5 Stufen einteilen (vgl. Abb. 4.30): in der ersten Stufe werden zwei der vier Eingänge zur Verknüpfung selektiert, die in der zweiten Stufe invertiert werden können. Die dritte Stufe besteht aus der eigentlichen Verknüpfung, wobei die Operationen „UND“ und „ODER“ sowie einfaches Durchreichen der beiden Eingänge zur Verfügung stehen. In der Vierten Stufe kann das Resultat der logischen Verknüpfung rotiert werden und wird in der letzten Stufe über ein Tristate–Puffer ausgegeben. Tabelle 4.8 enthält eine Aufschlüsselung des Opcodes der Bitblockalu.

[29] Dies ist nicht mit dem Rotieren von Bitblöcken zu verwechseln!

Opcode Bit								
0	1	2	3	4	5	6	7	Operation
1	-	-	-	-	-	-	-	Output Enable
-	0	-	-	-	-	-	-	$X = BB_0$
-	1	-	-	-	-	-	-	$X = BB_{\text{Hilf}}$
-	-	0	-	-	-	-	-	$Y = BB_1$
-	-	1	-	-	-	-	-	$Y = BB_2$
-	-	-	0	-	-	-	-	$A = X$
-	-	-	1	-	-	-	-	$A = \neg X$
-	-	-	-	0	-	-	-	$B = Y$
-	-	-	-	1	-	-	-	$B = \neg Y$
-	-	-	-	-	0	-	-	C ausgeben
-	-	-	-	-	1	-	-	C rotiert ausgeben
-	-	-	-	-	-	0	0	$C = A$
-	-	-	-	-	-	0	1	$C = B$
-	-	-	-	-	-	1	0	$C = A \wedge B$
-	-	-	-	-	-	1	1	$C = A \vee B$
1	0	0	0	1	1	1	0	Beispiel: rotate$(B_0 \wedge \neg B_1)$

Tabelle 4.8: Operationen der Bitblockalu

4.4.5.2.1 Rotieren von Bitblockdaten

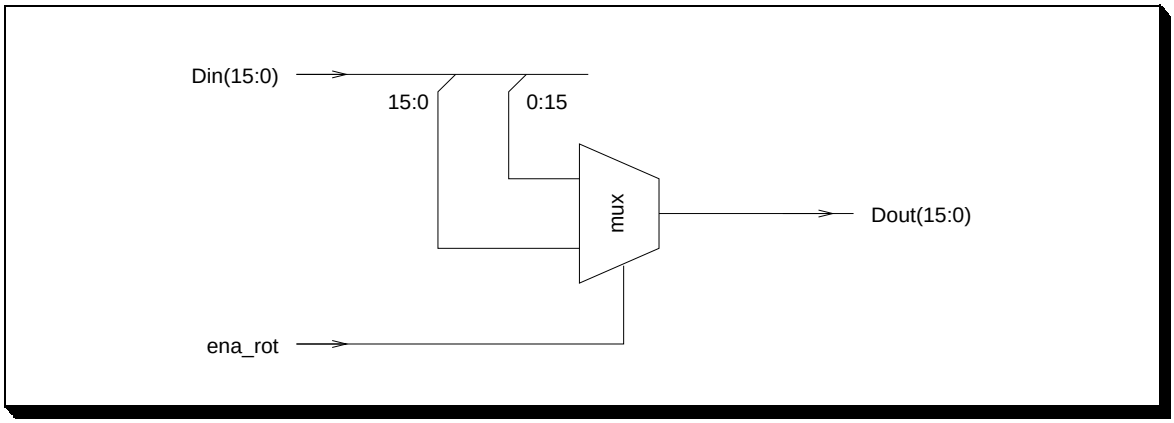
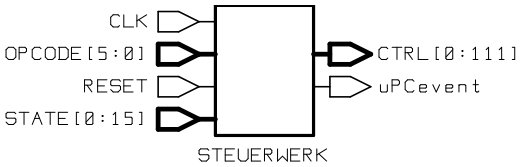


Abbildung 4.31: Realisierung von rotate

Das im C Programm (vgl. 3.3.6.2.2) realisierte Rotieren einer Bitfolge gestaltete sich recht aufwendig und rechneintensiv, da es in C keine Funktion gibt, die die Umkehrung einer Bitfolge ermöglicht. Abbildung 4.31 stellt die Realisierung einer Schaltung mit gleicher Funktionalität dar. Der Entwurf gestaltet sich sehr einfach, von einem Bus werden die Daten einmal vorwärts (15:0) und einmal rückwärts (0:15) abgegriffen und durch einen Multiplexer adäquat weitergeleitet.

4.5 Steuerwerk



Die Aufgabe des Steuerwerkes besteht darin, durch adäquates Ein– und Ausschalten von Tristate–Treibern den Datenfluß im Rechenwerk zu steuern und den Funktionseinheiten des Rechenwerkes ihre Aufgaben mitzuteilen. Die Gesamtheit aller dem Rechenwerk vom Steuerwerk in einem Zeitpunkt gegebenen Instruktionen bezeichen wir als Mikrobefehl und eine Folge von Mikrobefehlen als Mikroprogramm. Ein Mikroprogramm enthält dabei die Instruktionen zur Ausführung eines Befehls im Sinne von Kap. 4.4.2. Darüberhinaus ist das Steuerwerk in der Lage zu entscheiden, ob — in Abhängigkeit vom Status des Rechenwerkes — ein Mikrobefehl ausgeführt wird und welcher Mikrobefehl als nächster ausgeführt wird.

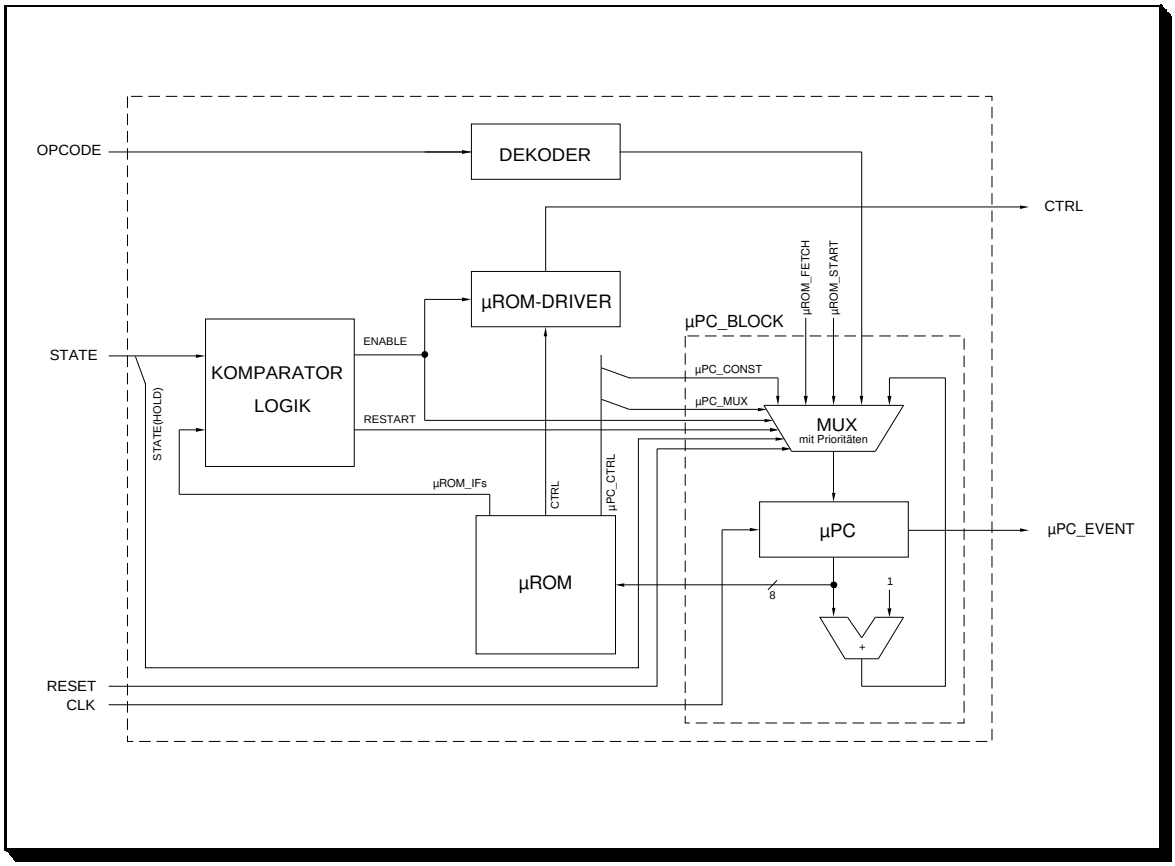
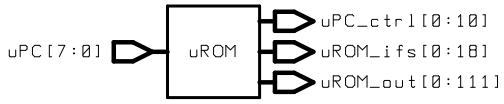


Abbildung 4.32: Blockschaltbild des Steuerwerks

Die zentrale Einheit im Steuerwerk ist der Speicher für die Mikroprogramme, das Mikrorom (μROM). Daneben findet sich ein Block zur Berechnung und Speicherung der aktuellen Adresse des μROMs (μPC_Block) sowie eine Comparator–Logik zur Berechung der Signale ENABLE und RESTART, durch die sich bedingte Anweisungen in einem Mikroprogramm realisieren lassen, ein Treiberblock zur Verknüpfung des CTRL–Bus mit dem ENABLE–Signal und ein Dekodierer, der den Opcode aus dem Instruktionsregister auf die Anfangsadresse des zugehörigen Mikroprogramms abbildet[30]. (vgl. Abb. 4.32).

[30] Die Alternative zu einem solchen Dekoder wäre gewesen, die unteren Adressen des Mikroroms als Startadressen der Mikroprogramme zu definieren und in den Mikroprogrammen nach dem ersten Programmschritt zu höheren Mikroromadressen zu verzweigen. Dies untersagt jedoch bedingte Mikroprogrammschritte (→ 4.5.3) in der ersten

4.5.1 Mikrorom



Das Mikrorom[31] liefert abhängig von der anliegenden Rom–Adresse — gespeichert im Mikroprogrammzähler μPC — das zur Ausführung eines Mikrobefehls notwendige Steuerwort auf dem Ausgang CTRL sowie die zur Steuerung des Steuerwerkes benötigen Kontrollworte μROM–IFS und μPC–CTRL. Ersteres teilt der Comparatorlogik mit, von welchen vom Rechenwerk gelieferten Statusinformationen die Ausführung des anliegenden Mikrobefehls abhängt (→ 4.5.3), letzteres beeinflußt die Wahl des nächsten Mikrobefehls durch den μPC_Block und besteht aus den Kommandoleitungen μPC_MUX und einer Konstanten μPC_CONST.

Den drei Kommandoleitungen ist folgende Bedeutung assoziiert:

μPC_INC: Mikroprogrammzähler soll inkrementiert werden

μPC_IREG: Mikroprogrammzähler soll einen Wert abhängig vom Opcode im Instruktionsregister bekommen (Start eines neuen Mikroprogramms)

μPC_N: Mikroprogrammzähler soll auf eine vom Mikrorom gelieferte Konstante μPC_CONST gesetzt werden

Der eigentliche Inhalt des Mikroroms — die Mikroprogramme — werden ob ihrer Relevanz in einem gesonderten Unterkapitel (→ 4.6) betrachtet.

4.5.1.1 Mikrorom–Compiler

Da eine manuelle Beschreibung der Mikroprogramme als 0/1–Matrix nicht nur mühsam, sondern auch fehleranfällig gewesen wäre — ist doch bereits der Steuerbus 112 Bits breit —, wurde ein rudimentärer Compiler zur Generierung desselben in C programmiert.

Jeder einfachen Steuerleitung ist ein Mnemonic zugeordnet, die zur Steuerung von ALUs und anderen Funktionseinheiten benötigten Opcodes werden durch einfache Befehle beschrieben. Eine Auflistung der Befehle des Mikrorom–Compilers befindet sich in Anhang A.3.

Der „Compiler“ liest nun eine Eingabedatei, in der pro Zeile ein Mikrobefehl durch Mnemonics beschrieben wird. Mit einem Minuszeichen (-) beginnende Zeilen werden als Kommentarzeilen betrachtet und lediglich mit einem weiteren Minuszeichen versehen als VHDL–Kommentar in die Zieldatei geschrieben. Vor der Behandlung einer Zeile werden alle Steuerleitungen auf inaktiv gesetzt; konkret einfache Steuerleitungen auf 0, Opcodes auf „nichts berechnen“ bzw. „irgendwas berechnen, aber nichts ausgeben“. Der Opcode der Bitblock–Alu (→ 4.4.5.2) z.B. wird mit `0-------` initialisiert, der Tristate–Treiber der ALU wird also deaktiviert; die restlichen Bits des Opcodes sind „don't care“–Fälle[32]. Für jeden in der abzuarbeitenden Zeile enthaltenen Mnemonic setzt der Mikrorom–Compiler die korrespondierende Steuerleitung auf aktiv. Jede derart bearbeitete Zeile wird als VHDL–Quelltext in die Zieldatei geschrieben. Vor und nach der Bearbeitung der Eingabedatei generierter VHDL–Text ergänzen die Ausgabe zu einer kompletten VHDL–Beschreibung.

Auf diese Weise kann das Mikroprogramm in einfach lesbarer Form beschrieben werden, was sowohl dessen Erstellung als auch dessen Fehlerkorrektur und Dokumentation erheblich vereinfacht.

Zeile, da die Folgezeile bereits dem Mikroprogramm des nächsten Opcodes zugeordnet wäre. Eine Abbildung des Opcodes auf die Bits 6 bis 1 der Mikroromadresse (somit wäre jede zweite Zeile der unteren Mikroromadressen eine Startadresse) verursachte zuviele ungenutzte Mikroromzeilen.

[31] Die Bezeichnung „Mikrorom“ ist insofern inkorrekt, als von der Synthese kein echtes ROM generiert wird, die Bezeichnung „festverdrahtetes Mikroprogramm“ also angebrachter wäre.

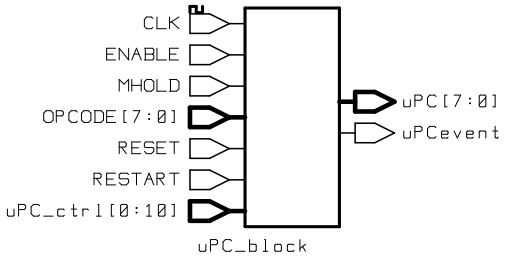
[32] Es bleibt also dem VHDL–Compiler überlassen, einen bezüglich der Minimierung der Hardware optimierten Wert einzusetzen, siehe auch [23], [32].

4.5.1.2 Struktur des Mikroroms

Auch das Mikrorom wurde in VHDL beschrieben, obwohl eine Beschreibung im Synopsys–PLA–Format wegen der zugrundeliegenden Struktur zunächst naheliegender erscheinen mag. Dagegen spricht jedoch — neben dem Wunsch, eine sprachlich homogene Beschreibung des Gesamtentwurfs zu erstellen —, daß das PLA–Format keine Ein– und Ausgabe von Bussen zuläßt, was bei der Anzahl der Ausgangsleitung den Mikroroms etwas umständlich gewesen wäre. Somit wurde Mikrorom als VHDL Case–Anweisung beschrieben:

```
1  architecture GEN of uROM is
2
3  begin
4   process(uPC)
5   begin
6    case uPC is
7      when " ... " => uROM_out <= " ... ";
8                      uROM_ifs <= " ... ";
9                      uPC_ctrl <= " ... ";
10     ...
11     when others  => ...
12   end case;
13  end process;
14 end GEN;
```

4.5.2 Mikroprogrammzählerblock



Der Funktionsblock zur Verwaltung des Mikroprogrammzählers besteht aus einem Auswahlbaustein und einem Speicher für den aktuellen Stand des Mikroprogrammzählers, der zur Vermeidung von Hazards als Master–Slave–Flipflop ausgelegt ist. Neben dem Mikroprogrammzähler indiziert dieser Block auf der Statusleitung μPCevent einen Wechsel des Mikroprogrammzählers und somit des Mikrobefehls als Pegeländerung auf dieser Leitung.

Bedingung	Priorität	Folgezustand	Bemerkung
MHOLD = 1	1	keiner	Chip wartet auf Speicher
RESET = 1	2	Einsprungsadresse der Reset–Routine	
RESTART = 1	3	Einsprungsadresse „Fetch–“Zyklus	
ENABLE = 0	4	μPC+1	
μPC_INC = 1	5	μPC+1	
μPC_IREG= 1	5	Einsprungsadresse abh. vom Instr.Reg.	
μPC_N = 1	5	vom Mikrorom gelieferte Adresse	

Tabelle 4.9: Fallunterscheidung für den Folgezustand des μPC

Der Auswahlbaustein — in Abb. 4.32 als „MUX mit Prioritäten“ abbreviiert —, wählt anhand eines Steuerbefehles vom Mikrorom und den Statusleitungen von Rechenwerk und Comparatorlogik aus

einer Menge möglicher Folgewerte für den Mikroprogrammzähler einen Wert aus. Bei dieser Auswahl wird eine Reihe von Prioritäten berücksichtigt, z.B. wird, falls die Entscheidung der Comparatorlogik negativ ausgefallen ist, wird immer der Programmzähler inkrementiert, unabhängig von der vom Mikrorom gewünschten Verzweigung, vorausgesetzt natürlich, der Chip befindet sich nicht im durch MHOLD angezeigten Wartezustand (vgl. 4.4.3.1) und soll nicht zurückgesetzt werden. Eine Auflistung der den Folgezustand des Mikroprogrammzählers beeinflussenden Faktoren findet sich in Tabelle 4.9.

```
1     P_uPC: process(CLK, MHOLD)
2     begin
3       if (MHOLD='0') then
4         if (CLK 'event and CLK='0') then
5           if (RESET = '1')
6             then uPCSlave <= uROM_START;        -- Anfangsadresse RESET-Routine
7                  UPE <= '0';
8             else
9                 if (RESTART = '1')
10                  then uPCSlave <= uROM_FETCH;  -- Anfangsadresse "FETCH"-Zyklus
11                       UPE <= not UPE;
12                  else if (uPC_INC = '1') OR (ENABLE = '0')
13                          then uPCSlave <= uPCinc;
14                               UPE <= not UPE;
15                          else if (uPC_IREG = '1')
16                                  then uPCSlave <= OPCODE;
17                                       UPE <= not UPE;
18                                  else if (uPC_N='1')
19                                          then uPCSlave <= N;
20                                               UPE <= not UPE;
21                                       end if;
22                               end if;
23                       end if;
24              end if;  -- if (RESTART = '1')
25          end if;   -- (RESET = '1')
26        end if;   -- if (CLK 'event and clk='0')
27      end if;   --  if (MHOLD='0')
28    end process;
29
30    P_uPCM: process(CLK, uPCSlave)
31    begin
32      if (CLK 'event and clk='1') then
33         uPC0 <= uPCSlave;
34         uPE0 <= uPE;
35      end if;   -- if (CLK 'event and clk='1')
36    end process;
37
38    DEL0: DELAY generic map (8) port map (uPC0, uPC1);   -- Verzoegerung
39    DEL1: DELAY0 port map (uPE0, uPCevent);
40    uPC <= uPC1;   -- sonst Fehlermeldung
```

Hierzu ist anzumerken, daß die ursprüngliche Formulierung `if (CLK'event and CLK='0' and MHOLD='0')` (Zeile 4) vom Synopsys Design–Analyzer als nicht synthetisierbar zurückgewiesen wurde. Ein ähnliches Problem trat bereits beim Entwurf der speziellen Register des Statusregisters auf (vgl. 4.4.3), ist aber an dieser Stelle einfacher zu lösen.

Zur Funktion der Verzögerungsglieder DELAY und DELAY0 siehe Kap. 4.2.2.

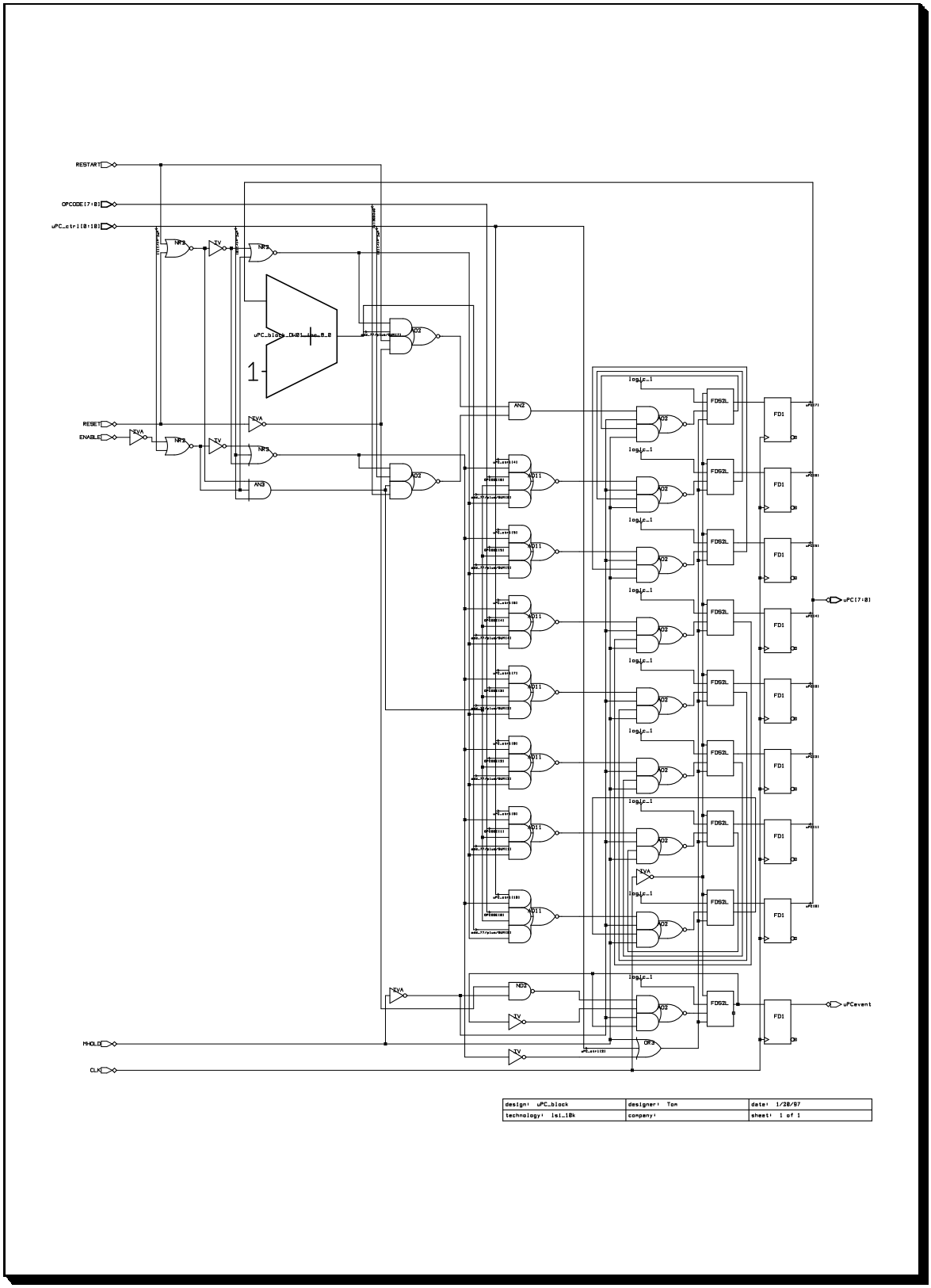
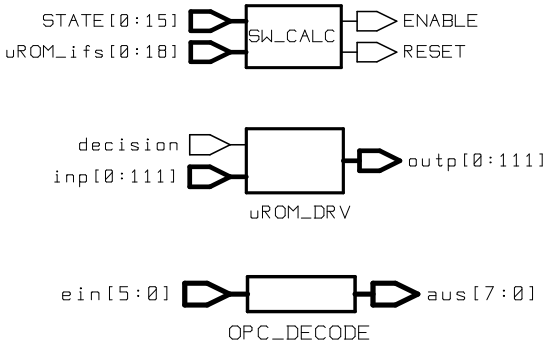


Abbildung 4.33: Schaltbild des Mikroprogrammzählerblocks

4.5.3 Comparator–Logik, Treiber und Dekoder



Die verbleibenden Blöcke des Steuerwerkes sind vergleichsweise trivial und sollen daher hier nur in aller Kürze erwähnt werden.

Die Comparator–Logik ist eine rein kombinatorische Schaltung, die in Abhängigkeit vom Status des Rechenwerkes (STATE) berechnet, ob ein Mikrobefehl ausgeführt wird (ENABLE) und ob mit der Bearbeitung des aktuellen Mikroprogrammes vorzeitig abgebrochen werden kann (RESTART). Das vom Mikrorom gelieferte Steuerwort μROM_IFs entscheidet dabei, welche der Statusleitungen auf die getroffenen Entscheidungen Einfluß haben soll und welche nicht.

ENABLE berechnet sich dabei als Konjunktion der Implikationen der Signale aus μROM_IFs und den zugehörigen Statussignalen, RESTART als Disjunktion von Konjunktionen dieser Signale: [33]

$$\mathsf{ENABLE} = \bigwedge_{i=1}^{15} s_i \quad \text{mit} \quad s_i = \mu\mathsf{ROM_IFs}\,(i) \Rightarrow \mathsf{STATE}\,(\sigma\,(i))$$

$$= \overline{\mu\mathsf{ROM_IFs}\,(i)} \;\vee\; \mathsf{STATE}\,(\sigma\,(i))$$

$$\mathsf{RESTART} = \bigvee_{i=16}^{18} s_i \quad \text{mit} \quad s_i = \mu\mathsf{ROM_IFs}\,(i) \;\wedge\; \mathsf{STATE}\,(\sigma\,(i))$$

Wie bereits erwähnt beeinflussen beide Leitungen den Folgezustand des Mikroprogrammzählers: Ist ENABLE Null, so wird er ungeachtet anderer Anweisungen des Mikroroms inkrementiert[34], ist RESTART Eins, so wird die Ausführung des aktuellen Mikroprogrammes abgebrochen und der μPC verzweigt in den Fetch–Zyklus. ENABLE beeinflußt zudem die Steuerung des Rechenwerkes: Ist ENABLE Null, sind also die über μROM_IFs ausgewählten Bedingungen nicht erfüllt, so führt das Rechenwerk in diesem Takt keine Operation aus, alle Kontrolleitungen müssen auf Null gesetzt werden[35]. Dies leistet der Block μROM–DRV, der sich wie folgt beschreiben läßt:

$$\forall i : \mathsf{CTRL_OUT}(i) = \mathsf{ENABLE} \wedge \mathsf{CTRL_IN}(i)$$

Der Dekoderblock dient lediglich dazu, die Anfangsadresse des zur auszuführenden Instruktion gehörenden Mikroprogramms zu ermitteln. Technisch handelt es sich also um ein ROM, das ähnlich wie das Mikrorom in einer Case–Anweisung beschrieben wurde.

---

[33] $\Rightarrow$ bezeichne die Implikation und $\sigma(i)$ die zum Steuersignal i korrespondierende Statusleitung; z.B. `S1 := (not uROM_ifs(IF_NR_EQ_1)) or STATE(NR_EQ_1)`

[34] Dies ist bei der Erstellung von Mikroprogrammen zu beachten! Insbesondere ist zu berücksichtigen, daß im Falle einer unerfüllten Bedingung in der letzten Zeile des Mikroroms (FF_{hex}) in die erste Zeile (0_{hex}) verzweigt wird, in der sich bei der aktuellen Implementierung die RESET–Routine befindet.

[35] Mit anderen Worten: Der Octchip steht einen Takt lang still, was Ansatzpunkt für Verbesserungen sein kann. Denkbar wäre z.B., den nächsten Mikrobefehl auszuführen. Das Problem dabei ist, daß dieser u.U. auch ein bedingter Programmschritt ist. Eine einfachere Variante wäre, statt der Festlegung, den μPC im Fall EN = 0 stets zu inkrementieren, eine zweite Sprungadresse μPC_CONST_ELSE oder sogar ein zweites Mikroprogrammzähler–Steuerwort μPC_CTRL_ELSE einzuführen, das in diesem Fall zum Tragen kommt. Somit ließen sich Anweisungen des Typs `IF` *bedingung* `THEN` *befehl* `upc:=` *sprung1* `ELSE upc:=` *sprung2* programmieren.

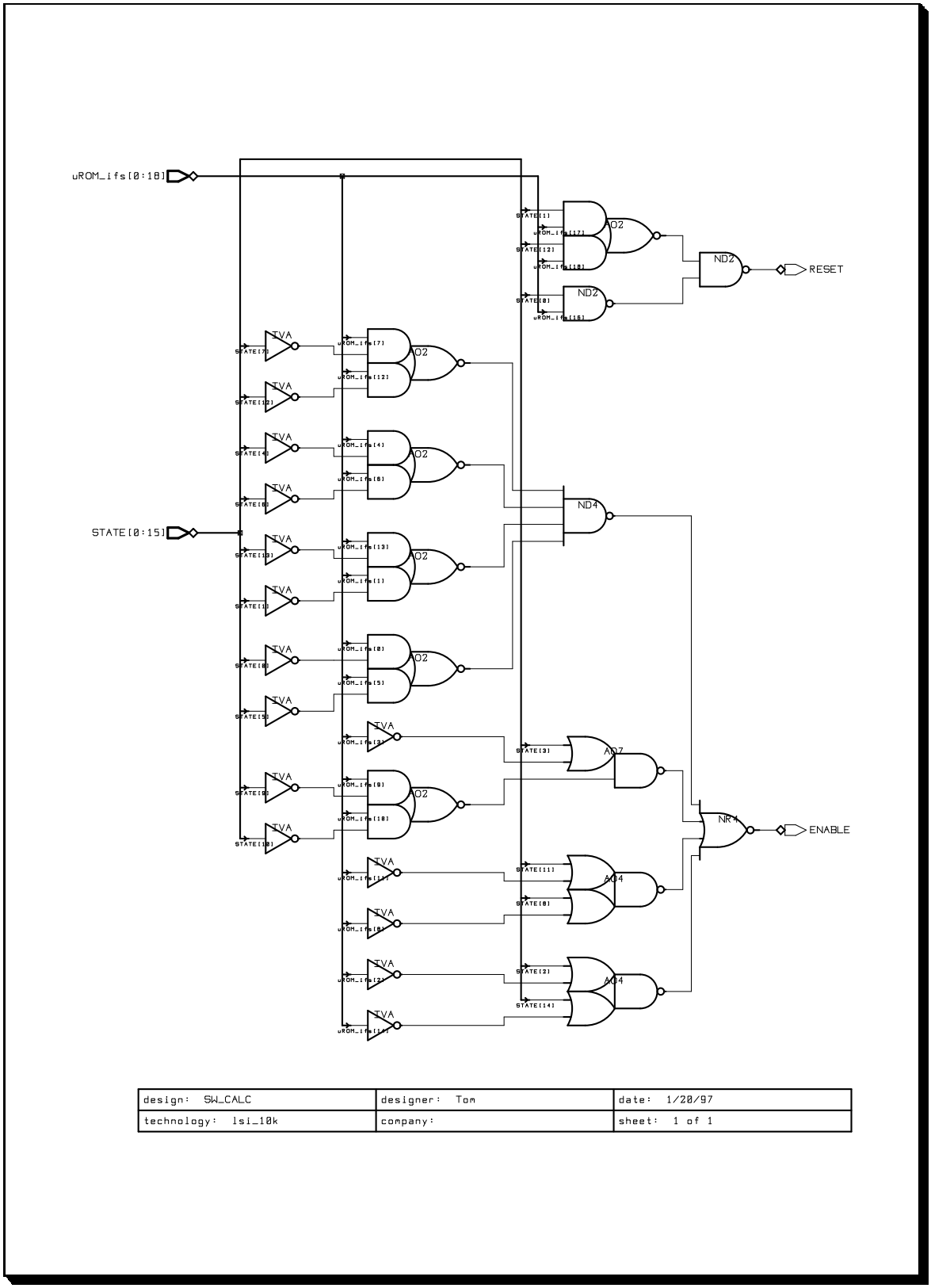


Abbildung 4.34: Schaltbild der Komparatorlogik

4.6 Mikroprogramme

Den Abschluß der Vorstellung des Octchip bilden die Mikroprogramme, durch die letztlich alle Abläufe im Chip gesteuert werden.

Jede Zeile des Mikroroms enthält Belegungen der Busse CTRL, μPC–CTRL und μROM-IFS mit den Steuerworten für das Rechenwerk, den Mikroprogrammzähler sowie Bedingungen zur Ausführung des Mikrobefehls (`if_...`) und zum Beenden des Mikroprogramms (`endif_...`). Die Auswertung einer Mikroromzeile kann durch folgenden Algorithmus beschrieben werden:

```
if ( Bedingung1 and Bedingung2 and ... and BedingungN )
then {
     führe Mikrobefehl CTRL aus
     if ( Endbedingung1 or Endbedingung2 or ... or EndbedingungN )
     then uPC:=128;
     else uPC:=Dekodiere( µPC-CTRL )
}
else uPC++;
```

Bei der Mikroprogrammierung ist zu beachten, daß bei Verwendung des Adreßbus ADROUT alle vier Eingangsbusse der Addierer der Navigationslogik (Abb. 4.15) mit einem definiertem Wert zu belegen sind. Durch die Mikrorombefehle 0_to_ad*mn* können diese Busse mit Null belegt werden.

Zur Beschreibung von Auszügen der Mikroprogramme verwenden wir eine an C–Notation angelehnte Beschreibung auf Register–Transfer–Ebene. An angemessenen Stellen wird der bei einem Registertransfer benutzte Bus explizit benannt:

```
BUS := REGISTERa + REGISTERb;     // statt REGISTERc := REGISTERa + REGISTERb;
REGISTERc := BUS;
```

Eine komplette Beschreibung des Mikroroms als Eingabe für den μROMcompiler ist in Anhang A.4 zu finden.

4.6.1 Fetch–Zyklus

Beginnen wir die Betrachtung der Mikroprogramme mit der Routine des „Fetch–Zyklus“. Übliche Prozessoren laden in diesem Zyklus Instruktionscode und Operanden aus dem Speicher. Eine ähnliche Vorgehensweise war ursprünglich auch für den Octchip vorgesehen — bei Aktivierung des Chips sollte dieser das auf dem Datenbus liegende Wort in das Instruktionsregister schreiben —, wurde jedoch zugunsten einer einheitlichen Spezifikation für Registerzugriffe verworfen; der Wirtsrechner greift auf das Instruktionsregister wie auf alle anderen Register zu. Damit degeneriert der „Fetch–Zyklus“ des Octchip zu einer einfachen Wartescheife, gefolgt von einer Verzweigung zur Einsprungsadresse des abzuarbeitenden Mikroprogramms:

```
BUSY := 0; BUSRQ := 0;
DO WHILE (not octchip_enabled());    /* Warteschleife */
BUSY := 1;
GOTO Mikroprogrammbeginn(OPCODE);
```

4.6.2 Navigation

Die erste Klasse der vom Octchip unterstützten Operationen sind die Navigationsoperationen[36]. Dabei handelt es sich um Pendants der Prozeduren aus Kap. 3.3.

[36] Die Basis zur Implementierung dieser Operationen bildete [1] Anhang A.2. Allerdings mussten die dort vorgestellten Mikroprogramme stark korrigiert und ergänzt werden, wie ein Vergleich des oben angegebenen Mikroprogramms mit dem in Anhang A.4 aufgelisteten zeigen dürfte.

4.6.2.1 Reset

Wird der Opcode 0 in das Instruktionsregister geschrieben oder liegt die Eingangsleitung RESET des Chips mindestens einen Takt lang auf Null, so setzt sich der Octchip in den Urzustand zurück: Die Register der Navigationslogik werden mit Null beschrieben und die Statusregister zurückgesetzt. Das Instruktionsregister wird mit $\mathrm{FFFF}_{\mathrm{hex}}$ initialisiert, so daß insbes. IREG_Q='11' gilt und sich nachfolgendes root auf alle drei Kubusbäume bezieht.

4.6.2.2 Navigation zum Wurzelknoten

In dem Mikroprogramm zu root wird zunächst anhand der Statusleitung BC_mod überprüft, ob der Inhalt des aktuellen Knotens oder des aktuellen Bitblocks seit dem Laden verändert wurden[37]. Ist dies der Fall, fordert der Octchip den Systembus an und schreibt die veränderten Register zurück:

```
if (BC_mod)
  do while (not busgr());   // Warte auf Buszuteilung
  if (niveau > 1) {         // Knoteninhalt schreiben
      ADROUT :=   AB[2]     // Basisadresse des Kubusmodells
                + AR[2]     // Anfangsadr. des akt. Kubusblocks
                + PR[2];    // aktuelle Position
      MEM[ADROUT] := C[2];  // Inhalt des akt. Knotens
  }
  else {                    // Bitblock schreiben
      ADROUT := AB[2] + AR[2];  AX := ADROUT;  MEM[ADROUT] := BB[2];  rotiereBB(2);
      ADROUT := AX + 1 ;        AX := ADROUT;  MEM[ADROUT] := BB[2];  rotiereBB(2);
      ADROUT := AX + 1 ;        AX := ADROUT;  MEM[ADROUT] := BB[2];  rotiereBB(2);
      ADROUT := AX + 1 ;        AX := ADROUT;  MEM[ADROUT] := BB[2];  rotiereBB(2);
  }
```

Dabei liefert `AB + AR + PR` die oft benötigte Adresse der dem aktuellen Kubus zugeordnete Speicheradresse (vgl. 4.4.4.1).

Abschließend werden die Register für Niveau, aktuelle Kubusblockadresse, aktuelle Position, vektorieller Index und Mittelpunkt zurückgesetzt.

4.6.2.3 Navigation zu Subkubi

Bei down werden die Adresse des aktuellen Kubusblocks und die aktuelle Position auf den entsprechenden Stack gelegt[38]; Niveau, Position und Mittelpunkt werden aktualisiert. Anschließend wird — falls `niveau[q]` > 1 [39] — der Inhalt des Zielkubus geladen, wobei zwischen direkter und indirekter Adressierung sowie zwischen Zeiger– und Bitblöcken unterschieden wird:

```
if (niveau[q] < 0){                // hier niveau _nach_ Aktualisierung!
   if (C[q][15] = '0'){            // Bit 15 = '0' => direkte Adressierung
       ADROUT := 4*C[q]            // C enthaelt relative Adresse,
                                   //  da die unteren beiden Bits einer Kubusblock-
```

[37] Da nur im Modell q = 2 Änderungen vorgenommen werden können, kann STATE(BC_mod) = '1' nur bei q = 2 gelten, vgl. Abb. 4.3. In [1] ist ein Speichern an dieser Stelle nicht vorgesehen, was zu Verlust geänderter Daten bei Navigation zum Wurzelknoten führt.

[38] Da das Ablegen der Position nur bei $n < 9$, der Kubusadresse nur bei $n \in \{2 \ldots 9\}$ sinnvoll ist, werden die Steuerleitungen CTRL(AS_PUSH) und CTRL(PS_PUSH) im Rechenwerk mit entsprechenden Statusleitungen durch UND–verknüpft. Gegenüber der in [1] vorgeschlagenen mikroprogrammierten Niveauüberprüfung erlaubt diese Lösung eine höhere Parallelisierung der durchzuführenden Operationen und damit weniger auszuführende Mikrobefehle. Ähnliches gilt für AS_POP und PS_POP.

[39] Hier wird das Niveau *vor* der Aktualisierung des Niveauregisters überprüft. Dies erklärt die auf den ersten Blick widersprüchlich erscheinenden Zeilen 21 (endif_nr<=1) und 25 (if_nr>1) des Mikroroms.

```
                                    //  Anfangsadresse stets 0 sind, werden sie nicht
                                    //  explizit gespeichert
                + AR[q];            // Anfangsadr. des akt. Kubusblocks
        AR[q] := ADROUT;
    }
    else {                          // Bit 15 = '0' => indirekte Adressierung
        ADROUT := AB[q]             // Modell Basisadresse
                + IB[q]             // Basisadresse der indirekten Adressen
                + 2*C;              // C enthaelt indirekte Adresse,
                                    //  bei ind. Adr. ist das unterste Bit 0
        ARL[q] := MEM[ADROUT];      // Lade indirekte Adresse "lower byte"
        AX := ADROUT;
        ADROUT := AX + 1;
        ARR[q] := MEM[ADROUT];      // Lade indirekte Adresse "higher byte"
    }
if (niveau[q] > 1){                 // Zeigerblock
        ADROUT := AB[q] + AR[q] + PR[q];
        C[q] := MEM[ADROUT];
    }
    else {                          // Bitblock
        ADROUT := AB[q] + AR[q];
                            AX := ADROUT;  BB[q] := MEM[ADROUT];   shiftBB(q);
        ADROUT := AX + 1;   AX := ADROUT;  BB[q] := MEM[ADROUT];   shiftBB(q);
        ADROUT := AX + 1;   AX := ADROUT;  BB[q] := MEM[ADROUT];   shiftBB(q);
        ADROUT := AX + 1;   AX := ADROUT;  BB[q] := MEM[ADROUT];   shiftBB(q);
    }
```

4.6.2.4 Navigation zum Vaterknoten

Bei der Navigation zum Vaterknoten wird analog zu root zunächst evtl. ein modifizierter Knoteninhalt oder Bitblock gespeichert. Anschließend werden Position und Kubusblockadresse vom Stack geladen sowie Niveau und Mittelpunkt aktualisiert. Ist $n > 1$, wird das Register C mit dem Inhalt des aktuellen Knotens neu geladen.

4.6.2.5 Navigation zu Bruderknoten

Wie die Softwarelösung stellt auch der Octchip zwei Operationen zur Navigation zu Bruderknoten zur Verfügung, die überwiegend identisch implementiert sind: Beide speichern zunächst ggf. einen modifizierten Knoteninhalt ab — der Fall eines veränderten Bitblocks braucht hier nicht betrachtet zu werden, da bei einer Navigation innerhalb eines Bitblocks der evtl. modifizierte Bitblock weiterhin im Bitblockregister gehalten wird. Erst bei Verlassen des Bitblockniveaus durch up ist dieser in den Speicher zu schreiben. Danach werden Mittelpunkt und Position aktualisiert. Hier liegen die Unterschiede zwischen next und across: Bei next wird die Position inkrementiert, bei across wird der im Instruktionsregister übergebene Parameter p (→ 4.4.2) gespeichert. Auch die Aktualisierung des Mittelpunktes ist unterschiedlich (→ 4.4.4.2). Abschließend wird für $n > 1$ der aktuelle Knoteninhalt in das Register C geladen.

4.6.2.6 Erzeugen neuer Knoten im Baum

Die Erzeugung eines neuen Knotens beginnt mit der Berechnung der relativen Adresse als Differenz zwischen der Adresse des aktuellen Kubusblocks und der ersten freien Speicherzelle. Ist die Differenz kleiner als 20000_{hex}, so kann eine direkte Adresse verwendet werden, anderfalls muß indirekte Adressierung angewendet werden[40]. Die relativen Adressen werden im Fall einer direkten Adresse durch

[40] Da in [1] $AXL = 0$ als Kriterium für direkte Adressen verwendet wird, werden bereits ab einer Differenz von 10000_{hex} indirekte Adressen verwendet; die maximale direkte Adresse ist also FFFF_{hex}. Da die Anfangsadresse eines

vier, im Fall einer indirekten Adresse durch zwei geteilt in den Speicher geschrieben (→ 4.4.4.1). Bei indirekten Adressen wird darüberhinaus das Bit 15 auf '1' gesetzt. Diese Schiebeoperationen werden von Instantiierungen der Komponente SHIFTER42 vorgenommen, → 4.4.1.2.

Nach der Aktualisierung des Inhaltes des aktuellen Knotens werden 8 Speicherplätze für einen Zeigerblock bzw. 4 Speicherstellen für einen Bitblock alloziert und — um bösen Überraschungen vorzubeugen — mit 0000_{hex} initialisiert.

```
if (niveau[2] < 10){                    // Der Kubus auf Niveau 10 wird nicht
                                        // explizit gespeichert
    AX := AP2 - AR[2];                  // Berechne relative Adresse:
    if (AXL <= 1) {                     // direkte Adresse
        ADROUT := AB[2] + AR[2] + PR[2];
        C := '00'.(AX / 4);             // vgl. 'down'
        MEM[ADROUT] := '00'.(AX / 4);   // ( . sei Konkatenation)
    }
    else {                              // indirekte Adresse
        ADROUT := AB[2] + AR[2] + PR[2];
        C := '1'.(IP2 / 2);             // IP2: erster freier Speicherplatz
                                        //       fuer indirekte Adresse
                                        // Bit 15 wird dabei zur Charakterisierung
                                        // einer indirekten Adresse auf 1 gesetzt
        MEM[ADROUT] := '1'.(IP2 / 2);
        ADROUT := AB[2] + IB[2] + IP2;  // Schreibe indirekte Adresse:
        MEM[ADROUT] := AP2L;            //   "lower Byte"
        AX := ADROUT;
        ADROUT := AX + 1;
        MEM[ADROUT] := AP2R;            //   "higher Byte"
        IP2 := IP2 + 2;
    }
}
ADROUT := AB[q] + AP2;                  // initialisiere allozierten Speicher:
                     AX := ADROUT;   MEM[ADROUT] := 0;
ADROUT := AX + 1;    AX := ADROUT;   MEM[ADROUT] := 0;
ADROUT := AX + 1;    AX := ADROUT;   MEM[ADROUT] := 0;
ADROUT := AX + 1;    AX := ADROUT;   MEM[ADROUT] := 0;
AP2 := AP2 + 4;                         // 4 Speicherzellen alloziert
if (niveau[2] > 2) {                    // kein Bitblock, also 4 weitere
                                        //   Speicherzellen allozieren
   ADROUT := AX + 1;    AX := ADROUT;   MEM[ADROUT] := 0;
   ADROUT := AX + 1;    AX := ADROUT;   MEM[ADROUT] := 0;
   ADROUT := AX + 1;    AX := ADROUT;   MEM[ADROUT] := 0;
   ADROUT := AX + 1;    AX := ADROUT;   MEM[ADROUT] := 0;
   AP2 := AP2 + 4;                      // 4 Speicherzellen alloziert
}
```

Kubusblocks stets durch vier teilbar ist, werden die unteren zwei Bits nicht gespeichert und im Falle der maximalen Adresse wird $\text{FFFF}_{\text{hex}}/4 = 3\text{FFF}_{\text{hex}}$ im Speicher abgelegt. Zur Differenzierung zwischen direkten und indirekten Adressen dient das höchstwertige Bit, also werden Knoteninhalte im Bereich 0000_{hex} bis 7FFF_{hex} als direkte Adressen interpretiert. Ein Bit bliebe somit bei dieser Methode ungenutzt. Berechnet man die maximale direkte Adresse gemäß $7\text{FFF}_{\text{hex}} * 4 = 1\text{FFFC}_{\text{hex}}$ wird deutlich, daß erst ab einer Differenz von 20000_{hex} auf indirekte Adressierung ausgewichen werden muß. Die Semantik der Statusleitung STATE(AXL_EQ_0) wurde daher — ohne eine Umbenennung — zu $AXL \leq 1$ „umdefiniert“: $\text{STATE(AXL_EQ_0)} := \bigwedge_{i=1}^{15} \neg\, \text{AXL}_i = \neg \bigvee_{i=1}^{15} \text{AXL}_i$.

4.6.2.7 Setzen von Knoteninhalten

Sehr trivial ist die Implementierung der Routinen zum Setzen eines Kubus auf „voll“ bzw. „leer“: Beide entscheiden anhand des Niveaus, ob in einem Zeigerblock oder in einem Bitblock zu setzen ist. Im ersten Fall wird das Register C modifiziert, im zweiten Fall wird ein Setzen durch das Bitblockregister initiiert.

4.6.2.8 Umbennen von Modellen

Die Spezifikation des Octchip sieht vor, daß nur im Modell q=2 Modifikationen vorgenommen werden dürfen. Um die Modelle q=0 und q=1 mit Inhalten zu füllen, muß ein Modellwechsel vorgenommen werden, d.h. ein als Modell 2 generierter Kubusbaum wird nach Modell 0 bzw. 1 verschoben. In einer realen Umgebung kann dies der Wirtsrechner vornehmen: Er kann neue Speicherbereiche reservieren und dem Octchip neue Basisadressen der Modelle mitteilen, er kann Kubusbäume im Speicher verschieben — dies wird durch ausschließliche Verwendung relativer Adressen möglich — und nicht mehr benötigte Kubusbäume löschen. Im einfachsten Fall — das Zielmodell ist leer — kann ein Verschieben durch Umkopieren der baumspezifischen Register erreicht werden. Der in 5.3 beschriebenen Simulationsumgebung stehen diese Möglichkeiten nicht zur Verfügung: Das C–Programm kann nicht auf den Speicher der Testbench zugreifen, und eine bidirektionale Kommunikation in dem Sinne, daß der Wirtsrechner Chipregister ausliest und deren Inhalt in andere Chipregister schreibt, ist nicht möglich. Aus diesem Grund wurde die Operation MoveModel2 implementiert, die den beschriebenen einfachsten Fall eines Modellwechsels vornimmt. Neben dem Umkopieren der Basisadressen von Modell 2 nach Modell q ($q \in \{0, 1\}$) wird die Basisadresse von Modell 2 auf dem Nachfolger der zuletzt benutzten Kachel gesetzt, die Kacheladresse der indirekten Adressen von Modell 2 wird inkrementiert.

Da die Verschiebeoperation nicht häufig benötigt wird, sollte sie ohne Erweiterung des Rechenwerkes implementiert werden, wodurch einige „Umwege“ nötig wurden:

```
// AB[q] := AB[2], Umweg ueber Hilfsregister AX, da
// AB[2] und AB[q] nicht gleichzeitig selektiert werden koennen
  ADROUT := AB[2];   AX := ADROUT;
  DO := AXL; DI := DO; AB[q] := DI;
// IB[q] := IB[2], Umweg wie oben
  ADROUT := IB[2];   AX := ADROUT;
  DO := AXL; DI := DO; IB[q] := DI;
// AB2 := AB2 + AP2L + 1, Umweg ueber AXR, da Inkrementieren
// in den oberen 8 Bit des Adressbus nicht moeglich ist
  ADROUT := AB2 + AP2L;  AX := ADROUT;
  DO := AXL;  DI := DO;  AP2R := DI;
  ADROUT := AXR + 1;  AX := ADROUT;
  DO := AXR;  DI := DO;  AB[2] := DI;
// IB2 := IB2 + 1, Umweg ueber AXR
  ADROUT := IB2;  AX := ADROUT;
  DO := AXL;  DI := DO;  AP2R := DI;
  ADROUT := AXR + 1;  AX := ADROUT;
  DO := AXR;  DI := DO;  IB[2] := DI;
// Zeiger auf freie Speicherstellen zuruecksetzen
  DI := 0;  AP2L := DI;  AP2R := 0;  IP2 := 0;
```

4.6.2.9 Testroutine: Schreiben der Registerinhalte

Zum Abschluß der Beschreibung der Navigationsroutinen sei noch die Operation DUMP erwähnt, die die Inhalte der relevanten Register der Navigationslogik in den Speicher schreibt und vor allem zu Testzwecken implementiert wurde: Die ersten Tests der Navigationslogik bestand aus Vergleichen

zwischen Registerdumps nach einer Folge von Navigationsoperationen und einem errechneten Sollergebnis. Um die Registerinhalte im Speicher leichter lokalisieren zu können, werden sie von zwei mit $FFFF_{hex}$ beschriebenen Speicherworten eingerahmt.

```
ADROUT := AB + AP2;
                    AX := ADROUT;  MEM[ADROUT] := "FFFF";
ADROUT := AX + 1;  AX := ADROUT;  MEM[ADROUT] := "FFFF";
ADROUT := AX + 1;  AX := ADROUT;  MEM[ADROUT] := middle_x;
...
ADROUT := AX + 1;  AX := ADROUT;  MEM[ADROUT] := "FFFF";
ADROUT := AX + 1;  AX := ADROUT;  MEM[ADROUT] := "FFFF";
AP2 := AP2 + 8;
AP2 := AP2 + 8;
AP2 := AP2 + 8;
```

4.6.3 Bitblockoperationen

In diesem Teil sollen die Mikroprogrammschritte der Bitblocktransformationen und –operationen, die im Abschnitt 4.4.5 schon kurz aufgezeichnet wurden, detailierter beschrieben werden.

Nachfolgender Ausschnitt aus dem Mikroprogramm verdeutlicht die Vorgehensweise der Realisierung von Bitblockoperationen. Die Abfrage IF_NR<= 1 in jedem Mikroprogrammschritt verhindert, daß der Befehl „zufällig“[41] ausgeführt wird. Die nachfolgenden Befehle (AXIS_0X, BB0_roll, BB0_bF, AXIS_1X, BB1_roll, BB1_bF) sorgen dafür, daß die möglichen Quellbitblöcke[42] die benötigten Daten der Bitblockalu zur Verfügung stellen. Durch den Befehl IREG_TO_BBA werden die letzten 7 Bit des Instruktionsregisters direkt auf die Leitungen für den Opcode der Bitblockalu geschaltet, so ist es möglich, die Bitblockoperationen möglichst variabel zu gestalten. Durch die Befehle (AXIS_2X, BB2_shift, BB2_bF) werden die an Din anliegenden Ergebnisse der Bitblockalu (durch Ausführen von IREG_TO_BBA oder auch direktes Setzen des Opcodes für die Alu, wird vom Steuerwerk automatisch die Leitung out_ena der Bitblockalu gesetzt) des im Bitblockregister 2 aufgenommen. Da die Operation Bitblockregister 2 als Ziel hat versteht sich das Setzen des BCMOD–Flags durch set_bcmod von selbst.

```
--BB0 BB1 -> BB2 --11000011
line:=195 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF ...
                        ... AXIS_2X BB2_shift BB2_bF IREG_TO_BBA set_bcmod  upc++
line:=196 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF ...
                        ... AXIS_2X BB2_shift BB2_bF IREG_TO_BBA            upc++
line:=197 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF ...
                        ... AXIS_2X BB2_shift BB2_bF IREG_TO_BBA            upc++
line:=198 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF ...
                        ... AXIS_2X BB2_shift BB2_bF IREG_TO_BBA            upc:=128
```

Bei den Transformationsoperationen wird ähnlich vorgegangen, nur daß hier als Quelle immer der Bitblock 0 gewählt wird und die Richtungen in denen die Bitblöcke durchgeschoben werden nicht identisch sein müssen. Die Bitblockalu führt hier als einzige Operation neben dem Durchschalten der Daten von Bitblock 0 nach Dout unter Umständen eine Rotation des Datums durch.

Zur Verdeutlichung wie eine Transformation durchgeführt wird sei hier ein kurzer Ausschnitt des Mikroprogramms erklärt, der Rest des Mikroprogramm ist im Anhang zu finden. Bei dieser Transformation handelt es sich um die Transformation 0 (Spiegeln am Ursprung).

[41] Zur weiteren Fehlertoleranz ist am Ende des Mikroprogramms noch ein Rücksprung in den FETCH–Zyklus realisiert, um zu verhindern, daß der Chip im Falle eines Fehlers in einen undefinierten Zyklus gerät; so setzt der Chip seine Arbeit mit dem nächsten Befehl fort, auch kein Garant für ein fehlerfreies beenden des Programms, aber ein definiert Zustand.

[42] Bitblockregister der Modelle 0 und 1

```
--T00 --11001011
line:=203 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ...
                                                ... ALU_0_A__R set_bcmod  upc++
line:=204 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ...
                                                ... ALU_0_A__R            upc++
line:=205 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ...
                                                ... ALU_0_A__R            upc++
line:=206 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ...
                                                ... ALU_0_A__R            upc:=128
```

In Tabelle 4.10 sind nochmal die restlichen strukturkonformen Transformationen aufgeführt, ihre Realisierung im Mikrorom läßt sich im Anhang aus dem gesamten Mikroprogramm ablesen.

Transformation	**Beschreibung der Transformation**	**im μROM–Programm**
T00	Spiegeln am Ursprung	203
T01	Spiegeln an der xy–Ebene	207
T02	Spiegeln an der zx–Ebene	211
T03	Spiegeln an der YZ–Ebene	215
T04	Klappen $\pi/2$ um x–Achse	219
T05	Klappen π um x–Achse	223
T06	Klappen $-\pi/2$ um x–Achse	227
T07	Klappen $\pi/2$ um y–Achse	231
T08	Klappen π um y–Achse	235
T09	Klappen $-\pi/2$ um y–Achse	239
T10	Klappen $\pi/2$ um z–Achse	243
T11	Klappen π um z–Achse	247
T12	Klappen $-\pi/2$ um z–Achse	251

Tabelle 4.10: strukturkonforme Transformationen

Kapitel 5

Synthese und Simulation

Nachdem im letzten Kapitel die Implementierung des Octchip beschrieben wurde, widmet sich dieses Kapitel dem Prozeß des Hardwareentwurfs. Beschrieben wird der Ablauf einer Hardwareentwicklung, die Synthese und Simulation des Octchip sowie deren Resultate. Die hier verfolgten Ziele sind:

- Korrektheitsverifikation des Hardwareentwurfs.
- Messung des Zeitverhaltens der Hardware zur Ermittlung der Zeitersparnis gegenüber einer reinen Softwarelösung.
- Quantitative Analyse des Trade–Offs zwischen Zeit[1]– und Platz[2]optimierter Lösung.

5.1 Hardware–Entwurfsprozeß

Im Nachfolgenden soll allgemein der Ablauf eines Hardware–Entwurfsprozesses geschildert werden. Später wird dann der Hardwareentwurf mit dem im Rahmen der Arbeit verwendeten Synthesewerkzeug Synopsys näher verdeutlicht und erläutert.

5.1.1 Entwurf im allgemeinen

Bevor man das Resultat, ein Stück Hardware, fertig und funktionsfähig produzieren kann, muß der Weg des Hardwareentwurfs (in Kurzform: von den Spezifikationen über die Synthese zum Ergebnis) durchlaufen werden. Abbildung 5.1 zeigt den Ablauf des Hardwareentwurfsprozeß.

Am Anfang stehen Spezifikationen und Anforderungen des gewünschten Entwurfs. Diese müssen in eine für das Syntheseprogramm verständlichen Form dargebracht werden. Hierfür wurden Hardwarebeschreibungssprachen wie **VHDL**, **Verilog** oder **MIMOLA** entwickelt, die es ermöglichen eine Struktur– oder eine Verhaltensbeschreibung einer Schaltung zu geben. Die Hardwarebeschreibungssprachen erlauben einen modularen Aufbau, so daß man auf schon beschriebene Elemente[3] zurückgreifen kann, analog zu fertigen Bausteinen bei richtigen Schaltungen.

Als nächster Schritt erfolgt die Simulation des beschriebenen Designs. Für diesen Zweck kann das Verhalten der syntaktisch korrekten Beschreibung unter zur Hilfenahme eines Simulators überprüft werden. Die Simulation des Designs erfolgt in einer Testumgebung, der `testbench`, in der man „reale“ Bedingungen und Eingabewerte vorgeben kann und so das Verhalten des Entwurfs überprüft. In dieser Phase des Entwurfsprozesses dient die Simulation nur einer groben Überprüfung des korrekten Verhaltens. Zwar können in VHDL auch Verzögerungszeiten von Schaltungselementen modelliert

[1] sprich Taktrate
[2] sprich Chipfläche
[3] hier kann auch auf Elemente aus Bausteinbibliotheken zurückgegriffen werden

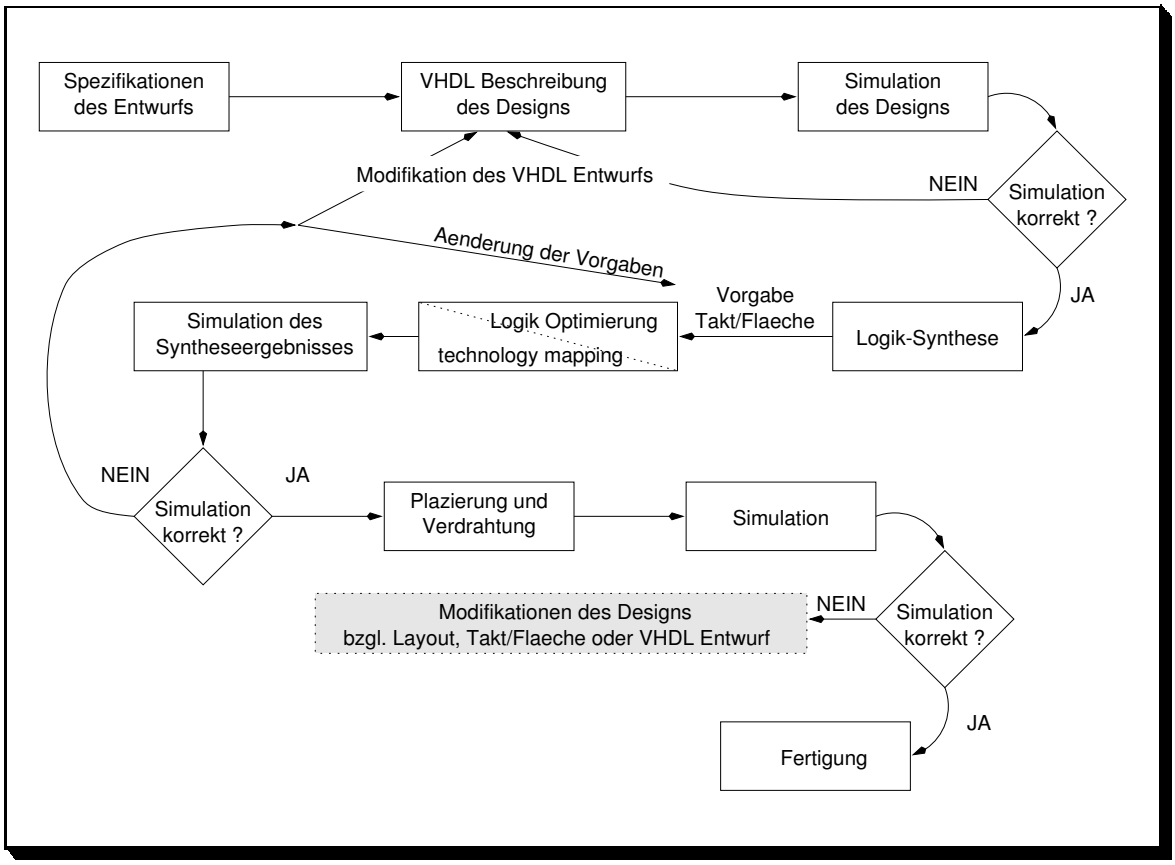


Abbildung 5.1: Der Hardwareentwurfsprozeß

werden, die in die Simulation mit einfließen, aber ansonsten werden hier keine technologiebedingten Restriktionen berücksichtigt. Die modellierten Verzögerungszeiten werden bei der nachfolgenden Synthese auch nicht weiter berücksichtigt, sie dienen nur dazu die Simulation in dieser Phase des Entwurfs möglichst „realistisch“ zu gestalten.

Sollte die Simulation des Designs nicht das erwartete Verhalten wiederspiegeln, so muß der VHDL–Entwurf überarbeitet werden. Verlief die Simulation zufriedenstellend, so kann mit der Synthese fortgefahren werden.

Der eigentliche Synthesevorgang, bei dem das Design auf eine spezielle Zieltechnologie übertragen wird, gestaltet sich sehr komplex. Als Zieltechnologie können hier verschiedene Halbleitertechnologien oder FPGAs[4] dienen. Bevor das Syntheseprogramm das Design auf die Zieltechnologie umsetzt, das sogenannte „technology mapping“, wird eine Logiksynthese durchgeführt. In dieser Logiksysnthese wird die VHDL Beschreibung auf logische Gatter umgesetzt. Das darauffolgende „technology mapping“ setzt diese logischen Gatter in Elemente der Zielthechnologie um. Durch diese Zweiteilung des Syntheseprozesses wird neben der Umsetzung auch eine Optimierung der beschriebenen Logik durchgeführt. Weiterhin ist es möglich bei dem Synthesevorgang Kriterien vorzugeben, so daß gewisse Vorgaben in Bezug auf Fläche oder Zeitverhalten eingehalten werden. Die so erhaltene Gatternetzliste muß nun erneut simuliert werden. Sollten hier Diskrepanzen auftreten, so kann man dem Syntheseprogramm Vorgaben in Bezug auf die Logikoptimierung machen oder schwächere Restriktionen, falls möglich in Bezug auf Fläche oder Zeit, vorgeben. Läßt sich mit keinem der Syntheseergebnisse eine erfolgreiche Simulation durchführen, so kann es sein, daß der gesamte VHDL–Ansatz modifiziert oder gar verworfen werden muß.

Liegt nun eine erfolgreich simulierte Gatternetzliste vor, so kann diese weiterbehandelt werden. Je nach Technologiegrundlage und Fertigungskonzept wird die Gatternetzliste in ein Layout– und Verdrahtungsprogramm übergeleitet oder auf ein FPGA übertragen.

Im Falle des Layout– und Verdrahtungsprogramms wird das Design entsprechend den Anforderungen eines Chiplayouts weiterbearbeitet und muß dann nocheinmal simuliert werden bevor mit der Fertigung des Chips begonnen werden kann. Sollte das Design auf ein FPGA transferiert werden, so wird auch hier ein Layout-Werkzeug angewendet um eine möglichst optimale Ausnutzung der FPGA–Zellen zu erreichen.

5.1.2 Synopsys

Das Synthesewerkzeug Synopsys ist kein einzelnes Programm, sondern ein Programmpaket mit vielen Modulen. Die zentrale Rolle übernimmt hierbei der design analyser(siehe Abbildung 5.2), die Schaltzentrale von der aus sich viele Funktionen des Syntheseprozesses, die auch als shell–Kommandos ausgeführt werden können, menügesteuert aufrufen lassen.

Bevor man mit dem Hardware–Entwurfsprozeß durch Synopsys beginnen kann, müssen einige Voraussetzungen geschaffen werden (siehe auch [48]):

Um einen möglichst reibungslosen und korrekten Ablauf zwischen den einzelnen Synopsys–Modulen zu erhalten sollte man eine Verzeichnisstruktur wie sie Abbildung 5.3 darstellt generieren. Hierbei sollten VHDL Beschreibungen im Verzeichnis rtl abgelegt werden. Die Ergebnisse der Logiksynthese legt Synopsys im Verzeichnis WORK und die des „technology mappings“ im Verzeichnis gate ab. Die Bibliotheken, die das Zeitverhalten der Zieltechnologien beschreiben, erwartet Synopsys im Verzeichnis libraries. Unter Umständen müssen aus einer VHDL Beschreibung der Zieltechnologie mittels liban erst noch die Bibliotheken mit den einzelnen Verhalten der verschiedenen Zeitmodelle[5] generiert werden, in diesem Fall sollte für die genaue Vorgehenweise das Synopsys Handbuch ([48]) zu Rate gezogen werden. Die Programme des Synopsys–Pakets sollten immer im Verzeichnis Entwurf

[4] FPGAs (Field Programmable Gate Arrays) sind frei programmierbare Logikzellen, die eine große Flexibilität erlauben. Eingesetzt werden sie vorallem beim *rapid prototyping* und bei der Fertigung kleiner Stückzahlen.

[5] Synopsys berücksichtigt bei der Synthese und Simulation verschiedene Zeitmodelle für Gatterverzögerungen und Signallaufzeiten. Eine genauere Beschreibung der Theorien hinter diesen Modellen würde den Rahmen dieser Arbeit überschreiten, hier sei auf weiterführende Literatur verwiesen [28][23].

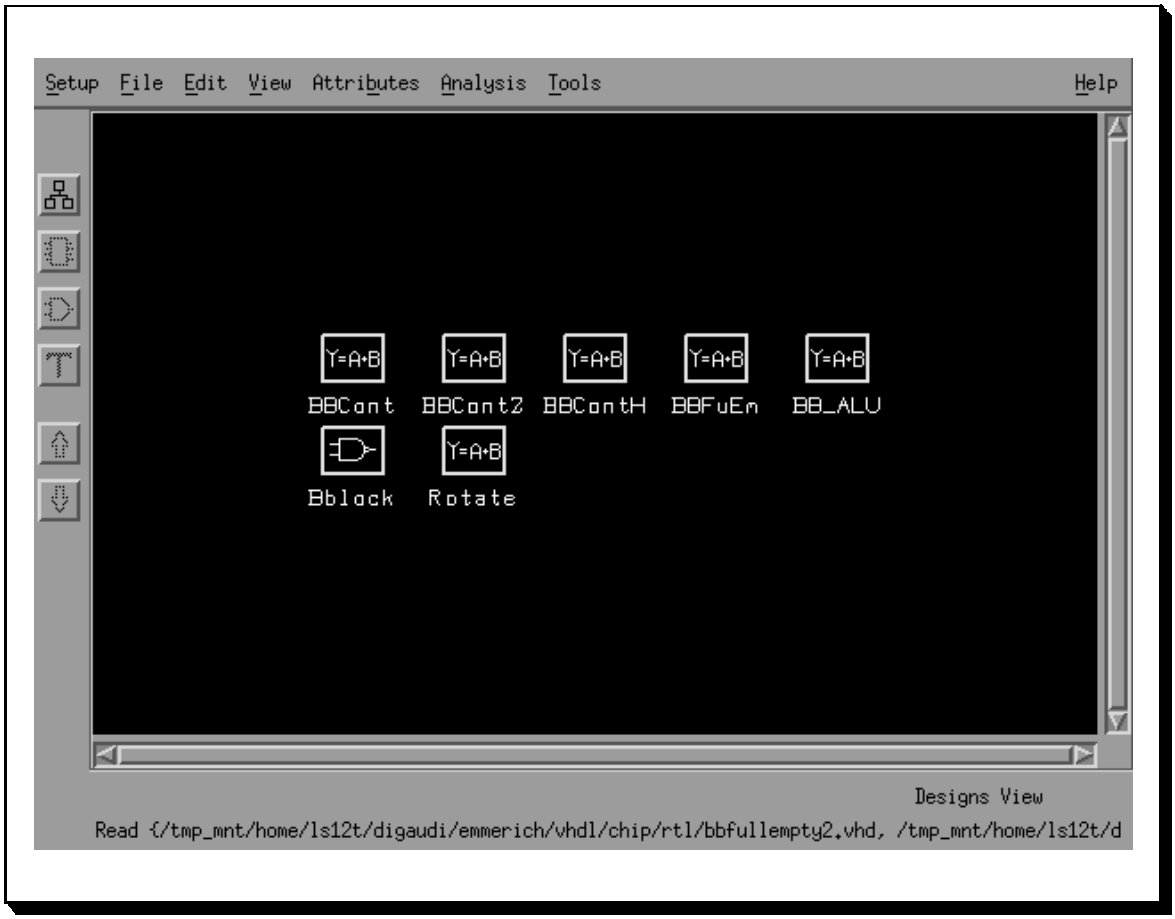


Abbildung 5.2: Die Schaltzentrale: der design analyser

der beschriebenen Verzeichnisstruktur aufgerufen werden. Hält man diese Struktur ein, dann reduziert sich die Anpassung der einzelnen setup–Dateien die das Zusammenspiel der Programme und Module steuern auf die Angabe der Zieltechnologie.

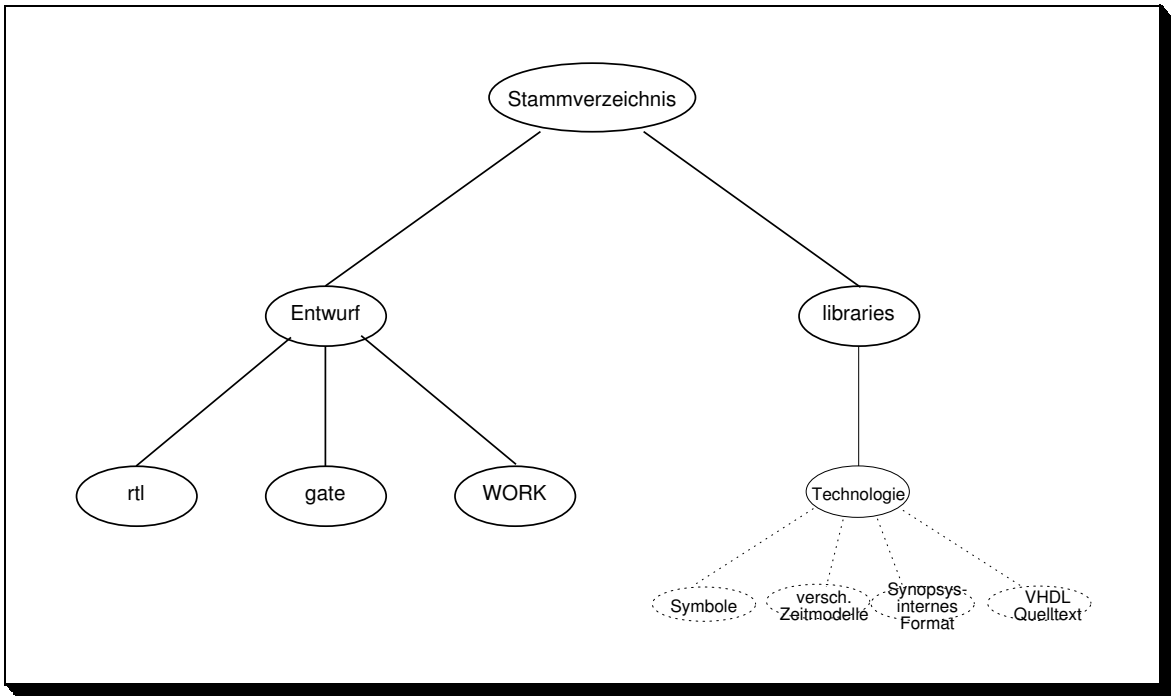


Abbildung 5.3: Verzeichnisstruktur der Synopsys Arbeitsumgebung

Bevor die verschiedenen Möglichkeiten des Entwurfs mittels Synopsys vorgestellt werden, sei an dieser Stelle noch ein VHDL Tutorial der Firma doulus (VHDL PaceMaker) erwähnt. Die trial-out Version dieses Programmes ermöglicht einen guten Einstieg in VHDL. Da das Synopsys Tutorial doch einiges an Basiswissen bezüglich VHDL voraussetzt, sollte man auch andere Quellen zur Einarbeitung in VHDL zu Rate ziehen.
Das Synopsys Tutorial liefert anhand *eines* kleinen Beispiels den Ablauf des kompletten Entwurfsprozesses. Um hier einen besseren Einblick zu erhalten ist dem Einsteiger weitere Literatur ([25], [26] und [28]) zu empfehlen.

5.1.2.1 Beschreibung eines Designs

Neben der Möglichkeit ein Designentwurf als Verhaltens– oder Strukturbeschreibung in VHDL vorzugeben, stellt Synopsys mit dem Schematic–Editor ein Werkzeug zur Verfügung, das es erlaubt aus einer Menge von Standardzellen mittels drag–and–drop–Operationen komplexere Designs zu erstellen. Diese „grafischen“ Entwürfe werden dann in VHDL umgesetzt. Ohne Grundkenntnisse von VHDL kann man so Designs entwerfen, aber man wird recht schnell feststellen, daß komplexere Entwürfe nur recht mühselig und aufwendig umzusetzen sind, so daß man besser direkt eine Beschreibung in VHDL erstellt.

5.1.2.2 Analyse/Simulation einer Beschreibung

Damit mittels Synopsys eine Simulation durchgeführt werden kann, muß der vorliegende VHDL–Code bezüglich seiner Syntax von Synopsys analysiert werden. Dies kann durch den design analyser oder durch Aufruf des Programms vhdlan erfolgen. Nach erfolgreicher Analyse kann eine Simulation des Entwurfs durchgeführt werden. Zu diesem Zweck sollte zuvor noch eine test bench erstellt werden. Zwar erlaubt der Simulator auch das manuelle Setzen von Signalen, aber die test bench sollte als

Simulationsumgebung das spätere „reale“ Umfeld des Designs wiederspiegeln. Außerdem kann die test bench für weitere Simulationen im Verlauf des Entwurfprozeß weiterverwendet werden.

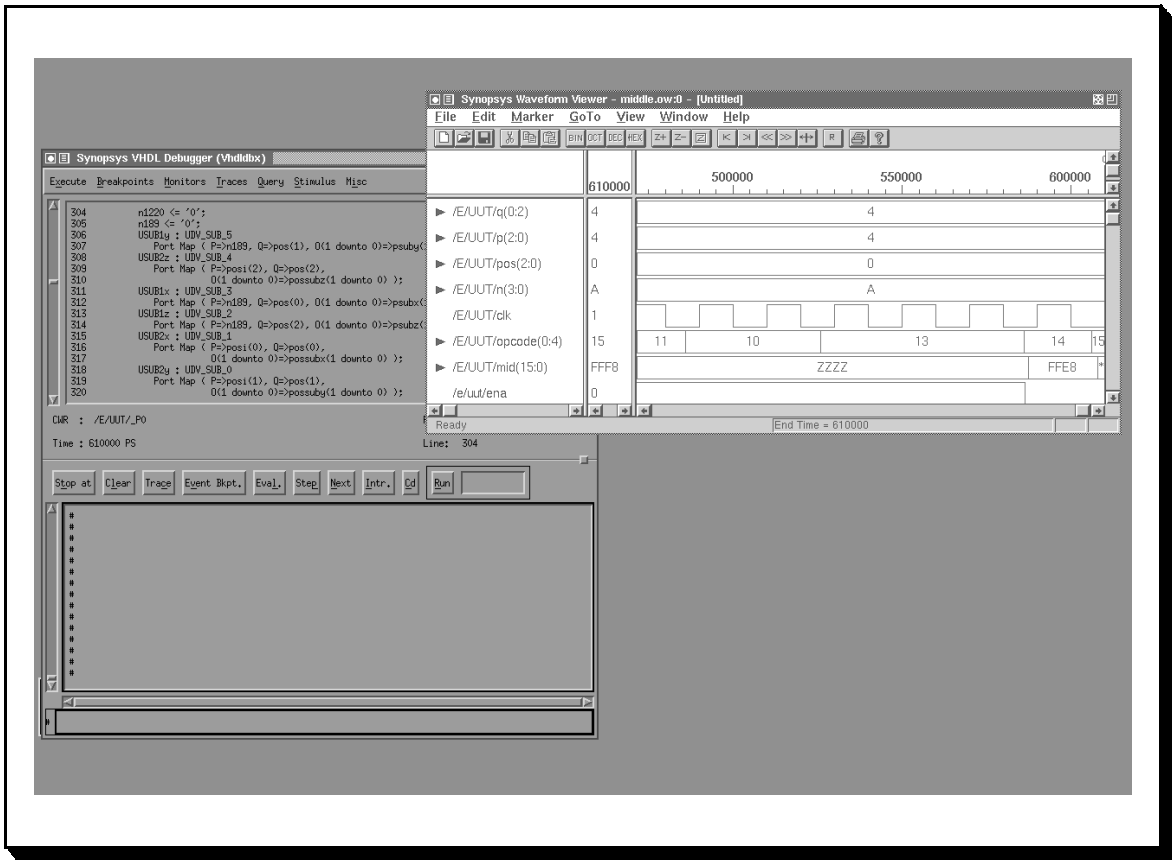


Abbildung 5.4: Die Fenster des Simulators: Debugger und Waveform–Viewer

Der Simulator wird über den Aufruf vhdldbx gestartet und ermöglicht es neben den Ein– und Ausgängen des Entwurfs auch interne Signalleitungen sowie Variablen zu überwachen. Nach dem Start und der Auswahl des zu simulierenden Designs erscheinen zwei Fenster (siehe Abbildung 5.4): der waveform–viewer dient zur Ausgabe von Signalverläufen; der Debugger ermöglicht im Rahmen der Steuerung der VHDL–Simulation eine schrittweise Abarbeitung der VHDL–Beschreibung.

5.1.2.3 Optimierung/Synthese

Nach erfolgreichem Verlauf der Simulation kann mit der Synthese fortgesetzt werden. Der Entwurf wird in den design analyser eingelesen; bei diesem Vorgang wird vom design analyser automatisch eine Logik–Synthese durchgeführt, d.h. die VHDL–Beschreibung wird auf einfache Bausteine und Logikelemente abgebildet. Diese Umsetzung erfolgt technologieunabhängig auf die interne Bausteinbibliothek von Synopsys, die GTECH–Library. Daneben bietet das Synthesesystem die Designware–Bibliothek mit vorgefertigten Bausteinen zur Synthese. Auf Elemente dieser Bibliothek kann auch schon bei der Beschreibung des Entwurfs zurückgegriffen werden, so daß Elemente wie z.B. Addierer nicht extra beschrieben werden müssen. Da dieser Vorgang zeitintensiv ist, sollte man das Design nach diesem Vorgang sichern. Als Dateiformat sollte das Synopsys eigene Format (.db–Dateien) verwendt werden auch wenn andere Formate unterstützt werden, da das Einlesen eines anderen Formates wieder eine Logiksynthese nach sich zieht.

Bevor mit dem „technology mapping“ fortgefahren wird, können noch Vorgaben in bezug auf die zu benötigende Fläche oder die zu erreichende Taktzeit gemacht werden. Während der Logikoptimierung und des Technologietransfers wird dann versucht diese Restriktionen einzuhalten bzw. nicht allzu

sehr von ihnen abzuweichen[6].

Damit der Optimierungsprozeß durchgeführt werden kann, müssen an dem Designentwurf noch einige Veränderungen vorgenommen werden. Ist eine Komponente in einem Design mehrfach instantiiert, so zeigt der design analyser das Design dieser Komponente nur einmal an, für den Optimierungsprozeß muß aber entsprechend jeder Instantiierung ein Design vorhanden sein; dies kann man mittels uniquify erreichen. Zuvor beschriebene Notwendigkeit verdeutlicht, daß während des Optimierungsprozesses die Struktur eines Designs unverändert bleibt, d.h. wenn ein Design aus mehreren Komponenten besteht, so wird jede Komponente für sich optimiert. Um einen möglichst hohen Optimierungsgrad zu erreichen kann man diese Strukturen auflösen, der design analyzer stellt hierfür die Funktion ungroup zur Verfügung. Dabei ist darauf zu achten, daß die Eingaben zur Optimierung weder zu groß noch zu klein bzgl. der Gatteranzahl sein sollten. Es sei an dieser Stelle erwähnt, daß die Funktion flatten hiermit nicht in Zusammenhang zu bringen ist; mittels flatten wird eine Auflösung und Gruppierung des Designs nach UND– und ODER–Gattern entsprechend einer PLA–Struktur erreicht.

5.1.2.4 Simulation des Synthese–Ergebniss

Nachdem der Synthese– und Optimimerungsprozeß erfolgreich durchgeführt wurde, liegt die Gatternetzliste in Form des Synopsysinternen Formats als .db–Datei vor. Eine abschließende Simulation der Designs steht nun an.

Bevor das Synthese–Ergebnis simuliert werden kann, muß es mit dem Programm SGE[7] bearbeitet werden. Im design analyser wird über das Skript *DA to SGE transfer*[8] der Transfer des Designs angestoßen. SGE beinhaltet ein Werkzeug, das zu einem Design eine Strukturbeschreibung in VHDL generiert. Dieser VHDL Beschreibung liegt die Technologiebibliothek mit ihrem Zeitverhalten zugrunde, so daß zu jedem Gatter eine entsprechende Konfiguration angegeben wird.

Der „Umweg“ über SGE ist erforderlich, da der design analyser zwar Optimierungen bezüglich einer Technologie durchführen kann, aber beim Speichern des Designs in VHDL nur die Beschreibung, ohne die jeweilige Konfiguration erzeugt.

SGE ermöglicht nach der Codierung in VHDL den Aufruf des Simulators mit der zuvor schon generierten testbench. Es werden wieder die beiden Fenster des Waveform–Viewers und des Debuggers geöffnet. Letzterem fällt nun nur noch die Rolle der Steuerung zu, da der jetzt vorhandene VHDL–Code mit dem ursprünglichen Entwurf nicht mehr viel gemein hat und so eine Kontrolle des Ablaufs so gut wie unmöglich wird.

Nach fehlerfreier Simulation kann mit der „Weiterverarbeitung“ der Gattenetzliste fortgesetzt werden; ansonsten müssen noch Modifikationen durchgeführt werden um ein korrektes Design zu erhalten. An dieser Stelle können mittels des design analysers noch Reports erstellt werden, die Auskunft über das Zeitverhalten und den zu erwartenden Flächenbedarf geben. Diese Informationen sind aber nur als Richtwert anzusehen, da ihnen kein reales Layout mit seinen speziellen Anforderungen zugrunde liegt.

[6] Es sollte ersichtlich sein, daß ein Design eine gewisse Fläche benötigt und daß Gatter minimale Verzögerungszeit haben, die nicht unterschritten werden können.

[7] Synopsys Graphical Environment

[8] zu finden im Menu setup unter Scripts

5.2 Testsynthese

Während der Herstellung integrierter Schaltungen können Fehler auftreten, darum sind Methoden zum Test gefertigter Chips für den Herstellungsprozeß von großer Bedeutung. Da eine vollständige Verifikation eines Chips undurchführbar ist, ist das Ziel der Testerstellung, Hardware nach der Herstellung auf möglichst optimale Weise — also mit einem Minimum an Testeingaben und einem Maximum an untersuchten Fehlerquellen — auf Fertigungsfehler untersuchen zu können. Geeignete Testschaltungen im Chip können dabei diese Prüfverfahren unterstützen.

Das Synopsys–Synthesesystem enthält ein Werkzeug, den Test–Compiler, zur Erweiterung eines Entwurfs um Strukturen zur Unterstützung bestimmter Hardware–Testmethoden und zur automatischen Generierung von Testmustern, auf die die Testhardware angewendet werden kann. Die vom Test–Compiler zur Verfügung gestellten Testmethoden sind der JTAG–Boundary–Scan–Test, der durch Modifikation der I/O–Ports erlaubt, Chips auf einer Leiterplatte zu Test–Ketten zu verschalten, sowie verschiedene Varianten interner Scan–Tests, bei denen die chipinternen Register sequentiell verdrahtet werden.

Es folgt ein kurzer Einblick in die Grundideen beider Testverfahren, gefolgt von der prinzipiellen Vorgehensweise bei der Arbeit mit dem Synopsys–System. Weiterführende Abhandlungen zu diesem Thema sind in [29], [39], und [45] zu finden. Ansätze von Algorithmen zur Berechnung optimaler Mengen von Testmustern werden in [32] behandelt.

5.2.1 Interne Scan–Tests

Herstellungsfehler in einem Chip können durch sog. Stuck–at–faults modelliert werden: Ein Stuck–at–fault liegt vor, wenn eine Leitung durch einen Hardwaredefekt einen konstanten Wert ('0' oder '1') führt. Ein Testmuster ist dann eine Eingabe, die im Fehlerfall eine andere Ausgabe erzeugt als in einer fehlerfreien Schaltung.

Betrachten wir nebenstehende Abbildung: Auf der Leitung L1 sei — etwa durch eine Leitungunterbrechung — ein Stuck–at–0 Fehler. Der Ausgang A des UND–Gatters liefert somit unabhängig von der Belegung der Eingänge E1 und E2 den Wert Null.
Ein Testmuster, mit dem sich dieser Fehlerfall überprüfen läßt, ist E1 = E2 = '1': Im Fehlerfall ist die Ausgabe '0', im korrekten Fall '1'.

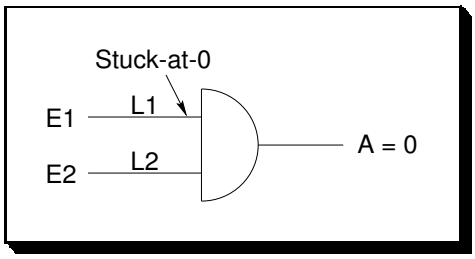


Abbildung 5.5: Stuck–at–fault

Zur Anwendung von Testmustern bei sequentiellen Schaltungen wird die logische Struktur der Schaltung so geändert, daß *alle* kombinatorischen Blöcke im Testmodus mit einen einzigen Eingabevektor versehen werden können und einen einzigen Ausgabevektor liefern. Durch sequentielle Verkettung der Flip–Flops kann ein Eingabevektor E_R in die Flip–Flops geschoben werden — dieser bildet zusammen mit der Belegung E des Schaltungseingangs den Testvektor — und nach einem Taktzyklus[9] kann das Ergebnis A_R aus den Flip–Flops herausgeschoben werden, das mit dem Ergebnis A den Ausgabevektor bildet. Dabei werden die Flip–Flops durch äquivalente Flip–Flops mit Scan–Test Unterstützung ersetzt. Abb. 5.6 verdeutlicht dies: Eine Schaltung enthalte zwei Register R_1 und R_2, die die Resultate der Schaltnetze L_1 und L_2 speichern. Ein weiteres Schaltnetz berechne die Ausgabe A. Die Flip–Flops sind, wie in 5.6(1) dargestellt, sequentiell verkettet. Die logische Struktur[10] der Schaltung im Testmodus verdeutlicht 5.6(2): Ein Eingabevektor E_R kann sequentiell in die Register R_1 und R_2 geschoben werden, und nach einem Taktschritt kann ein Ausgabevektor A_R sequentiell herausgeschoben werden. Neben diesen Test der kombinatorischen Blöcke können auch die Register

[9] Dabei ist zu beachten, daß ein Testtaktzylus nicht notwendigerweise mit einem Taktzylus im normalem Betrieb übereinstimmt. Der Testtakt muß so gewählt sein, daß auch der kombinatorische Block mit der höchsten Zeitverzögerung ein korrektes Ergebnis liefert.

[10] Die gewählte Darstellungform dient lediglich dem Verständnis der Methode des Scan–Test; an der Verschaltung der Komponenten wird nichts verändert.

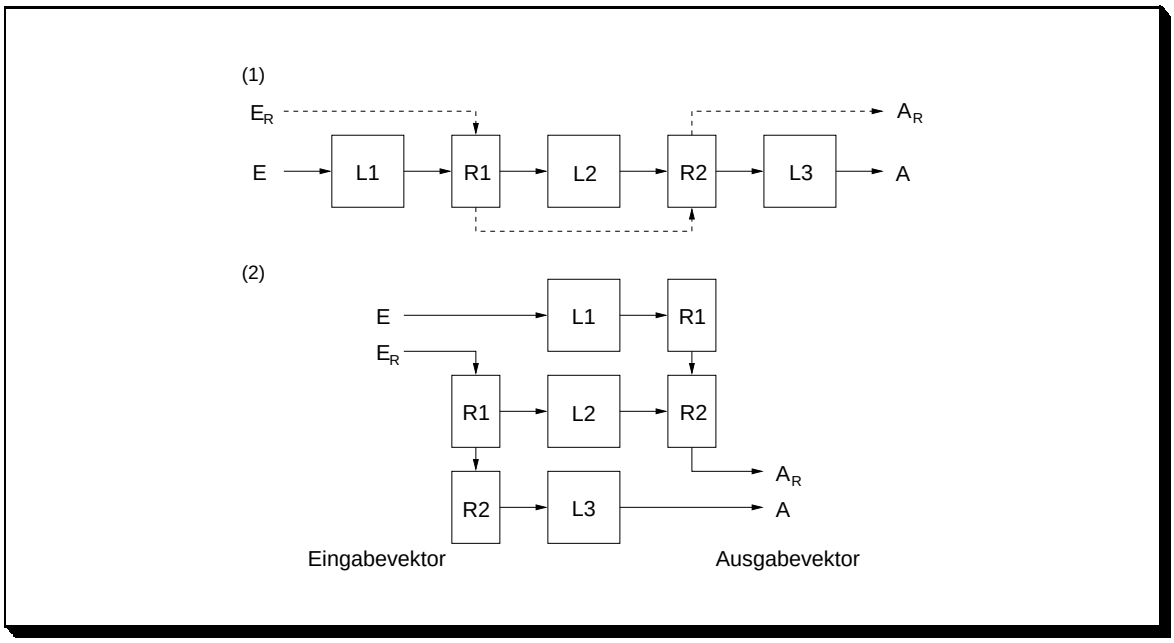


Abbildung 5.6: Interner Scan–Test

selbst getestet werden, indem ein Bitmuster die gesamte Scan–Kette durchläuft und das ausgegebene Bitmuster mit dem eingegebenen verglichen wird.

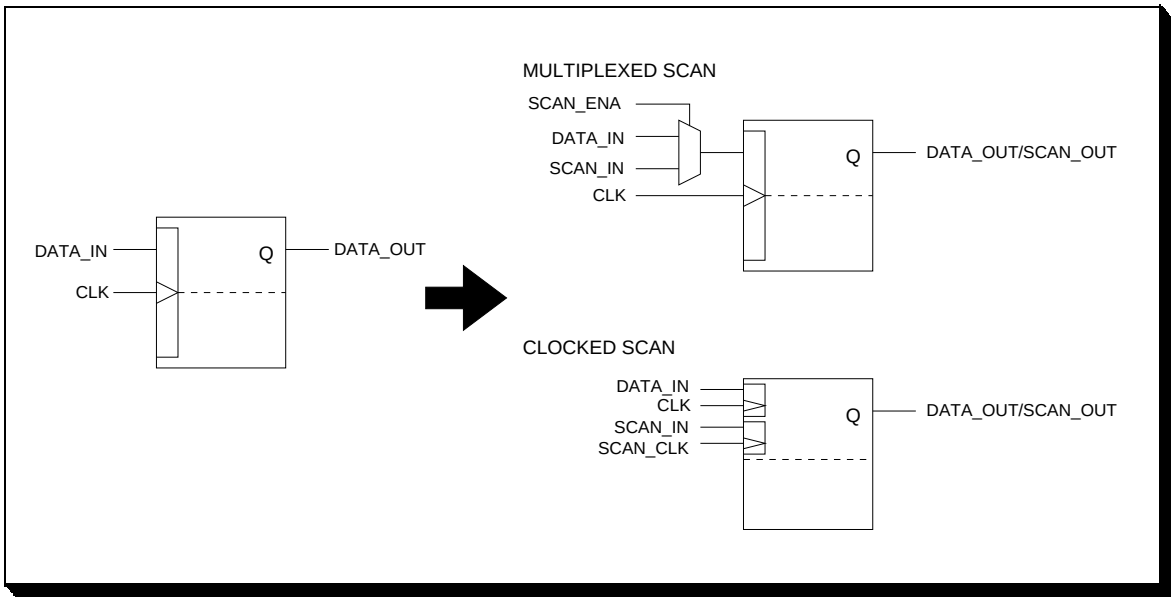


Abbildung 5.7: Register für internen Scan–Test

Es gibt verschiedene Ansätze zur Erweiterung eines Flip–Flops zu einem Flip–Flop mit Scan–Test Unterstützung. Abb. 5.7 zeigt zwei dieser Ansätze[11] anhand der Konvertierung eines D_T–Flip–Flops: Bei der Multiplexed–Flip–Flop Methode wird vor den Dateneingang des Flip–Flops ein Multiplexer geschaltet, der zwischen dem Eingang im Normalbetrieb (DATA_IN) und dem Eingang im Testbetrieb (SCAN_IN) umschaltet. Beim Clocked–Scan Verfahren wird zum Scan–Test eine zweite Taktleitung SCAN_CLK benutzt. Das Verhalten des hier benutzten Flip–Flop läßt sich folgendermaßen beschreiben:

[11] Die komplexeren Methoden Level–Sensitive Scan und Auxilliary Clock LSSD Scan sind in [45], Kap. 3 zu finden.

```
if (CLK'event and CLK='1')
   then q := DATA_IN;
   else if (SCAN_CLK'event and SCAN_CLK='1')
           then q := SCAN_IN;
        end if;
end if;
```

Der Vorteil des Clocked–Scan Verfahrens gegenüber dem Multiplexed–Flip–Flop Verfahren ist ein geringerer Zeitverlust durch zusätzliche Logik. Im Gegenzug erlaubt der Clocked–Scan keine Master–Slave–Flip–Flops in Scan–Ketten.

5.2.2 JTAG–Boundary–Scan–Test

Eine sehr viel umfangreichere[12] Testmethode ist der durch die IEEE–Norm 1149.1 standardisierte Boundary–Scan–Test. Bei dieser Methode erfolgt im Gegensatz zum internen Scan–Test kein Eingriff in die erstellte Schaltung[13] — in der Boundary–Scan Terminologie „Chip–Kern“ genannt —, sondern sie wird durch Erweiterung der Schaltung um bestimmte Funktionsblöcke realisiert.

Der Boundary–Scan ermöglicht es, Chips zu testen, die sich bereits auf einer Leiterplatte befinden (In–Circuit–Tests); er erlaubt sowohl den Test einzelner mit Boundary–Scan–Logik versehener Komponenten eines Boards — etwa durch Laden der Eingabeports und Auslesen der Ausgabeports nach einer Berechnung — als auch ein Test der Interaktion zwischen den Chips oder ein Test der Leiterbahnen durch Laden der Ausgabeports. Dazu wird die Aktivierung chipinterner Selbsttests und das Auslesen deren Resultate unterstützt.

Abb. 5.8 zeigt die prinzipielle Vorgehensweise: Der Chip muß um 4 (optional 5) Anschlüsse erweitert werden. Zwei davon dienen zur Ein– und Ausgabe von Testdaten (TDI, Test Data In und TDO, Test Data Out), einer zur Auswahl des Testmodus (TMS, Test Mode Select) und einer als separater Takteingang (TCK, Test Clock). Der optionale Eingang dient dem Rücksetzen der Testlogik ($\overline{\text{TRST}}$, Test Reset). Mit Boundary–Scan–Logik versehene Chips einer Leiterplatte können durch Verbindung des TDO–Ausgangs eines Chips mit den TDI–Eingang des nächsten zu sequentiellen Testketten verbunden werden. Alle Chips einer Testkette werden dabei mit gemeinsamen TCK, TMS und — soweit vorhanden — $\overline{\text{TRST}}$ versehen.

In die Ein– und Ausgangsleitungen des Chip–Kerns werden Boundary–Registerzellen geschaltet, die sequentiell verbunden werden und so ein Schieberegister bilden: das Boundary–Register[14]. Im Falle bidirektionaler Ports werden als Ein– oder Ausgang konfigurierbare Zellen eingefügt. Diese Zellen können nun die Belegung der Anschlußleitungen zu einem bestimmten Zeitpunkt speichern. Über TDO kann diese Belegung dann seriell ausgelesen werden. Umgekehrt können die Boundary–Registerzellen über TDI mit einem Wert beschrieben werden, um eine bestimmte Belegung der Ports simulieren zu können. Die Zellen sind dabei so aufgebaut, daß serielles Ein– und Auslesen die Belegung der Anschlußleitungen nicht beeinflußt; eine Kopplung zwischen den seriellen Einheiten einer Zelle und den Ports erfolgt erst durch explizites Aktivieren.

In den Datenpfad zwischen TDI und TDO können parallel zum Boundary–Register weitere Register geschaltet werden. Der Standard fordert die Implementierung eines Instruktions– und eines Bypass–Registers. Begleitet werden diese Register von einem durch TMS und TCK gesteuerten endlichen Automaten mit 16 Zuständen — den TAP–Controller —, in dem das vom Standard definierte Kommunikationsprotokoll implementiert ist, sowie einem Dekoder für das Instruktionsregister. Weitere Register wie ein Identifikationsregister oder ein Register zum Speichern des Resultats eines internern Selbsttests sind optional.

Der TAP–Controller unterscheidet im wesentlichen die vier Modi „Testlogik inaktiv/zurücksetzen“, „Test durchführen“, „Instruktionsregister seriell laden/auslesen“ und „Datenregister[15] seriell laden/auslesen“. Die meisten beim Boundary–Scan verwendeten Register — u.a. das Boundary– und

[12] und — um die Begeisterung des Autors über diese Verfahren auszudrücken — sehr viel interessantere
[13] Zumindest nicht für den Boundary–Scan selbst; für interne Selbsttest–Funktionen gilt dies natürlich nicht.
[14] Dieses an der „Grenze“ des Chips liegende Register verlieh dem Boundary–Scan–Test seinen Namen.
[15] Zu den Datenregistern zählen das Boundary– und Bypass–Register sowie die optionalen Register.

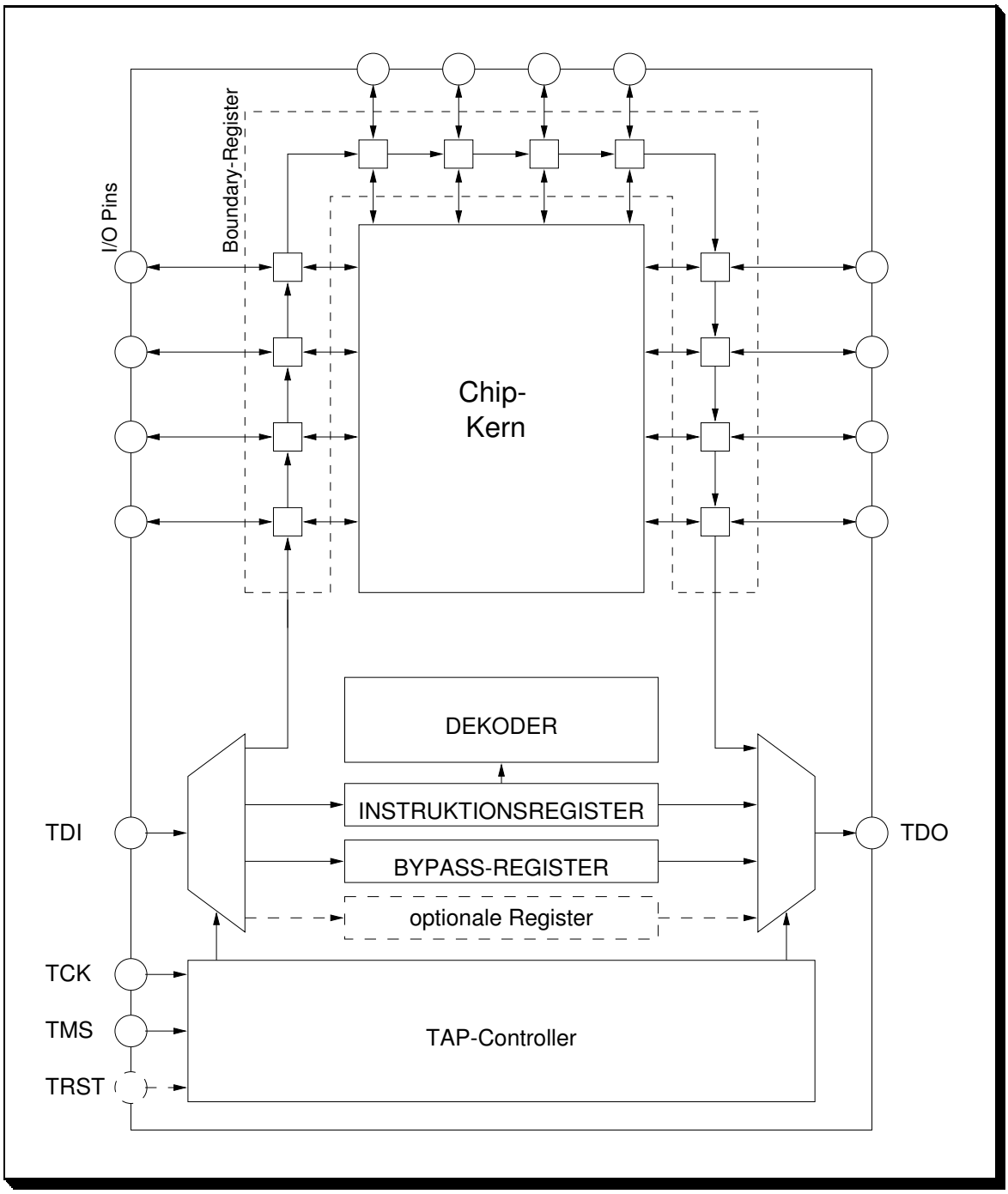


Abbildung 5.8: JTAG–Boundary–Scan–Test

das Instruktionsregister — bestehen aus einem seriellen und einem parallelen Teil. Ein– und Auslesen der Register über TDI und TDO bezieht sich dabei stets auf den seriellen Teil. Die letzten beiden Modi unterteilen sich daher in die Zustände „Kopieren des Inhaltes des parallelen Teils in den seriellen“ (Capture), „Schieben der Daten“ (Shift) und „Kopieren des Inhaltes des seriellen Teils in den parallelen Teil“ (Update).

Zur Durchführung einer Testoperation wird das Instruktionsregister mit dem Opcode der gewünschten Operation geladen. Der Standard verlangt die Implementierung folgender Instruktionen:

SAMPLE/PRELOAD: Zunächst wird die Belegung der Anschlußleitungen in den Boundary–Registerzellen gespeichert, danach können diese Werte seriell ausgelesen werden, während die Zellen gleichzeitig mit neuen Werten geladen werden. Die geladenen Werte werden jedoch vorerst nur zwischengespeichert; sie werden nicht an die Anschlußleitungen weitergereicht.

EXTEST: Die an den Eingabeleitungen des Chips liegenden Werte werden in den Registerzellen gespeichert. Ausgabeleitungen werden mit den in den entsprechenden Zellen gespeicherten Werten belegt. So kann ein bestimmtes Ausgabeverhalten des Chips zum Test nachfolgender Einheiten simuliert werden.

BYPASS: TDI und TDO werden durch das Bypass–Register „kurzgeschlossen“, d.h. jeder über TDI hineingeschobene Wert wird im nächsten Takt über TDO herausgeschoben. Mit dieser Instruktion können Testmuster an in einer Testkette weiter hinten liegende Chips oder Testresultate eines weiter vorn liegenden Chips weitergereicht werden.

Darüber hinaus können weitere Instruktionen implementiert werden[16]:

IDCODE: Das 32 Bit breite Identifikationsregister wird in den Datenpfad zwischen TDI und TDO geschaltet und mit einem festverdrahtetem Wert zur Identifikation des Chips geladen. Dieses Identifikationswort enthält genormte Angaben über Hersteller, Baustein und Version. Selbstverständlich setzt die Implementierung dieses Befehls die Implementierung des Identifikationsregisters voraus.

USERCODE: Das Identifikationsregister wird mit einem benutzerdefinierten Wert geladen.

INTEST: Das Verhalten der Scan–Logik beim INTEST kann man als zum EXTEST komplementär beschreiben: Hier werden die Werte an den Ausgangsleitungen in den Zellen gespeichert und die Eingangsleitungen mit den Werten in den Zellen belegt. Auf diese Weise kann eine bestimmte Eingabe an den Chip gelegt und die Ausgabe des Chips abgegriffen werden.

RUNBIST: Bei dieser Instruktion wird ein Selbsttest des Chips durchgeführt. Resultate dieses Tests werden in einem Selbsttestregister abgelegt und können nach Beendigung des Tests seriell ausgelesen werden. Während des Tests werden die Ausgangsleitungen mit den in den Zellen gespeicherten Werten geladen, so daß der Testlauf keine Auswirkungen auf nachfolgende Einheiten hat.

HIGHZ: Dies versetzt die Ausgabeleitungen in einen hochohmigen Zustand, so daß diese von einer externen Testeinrichtung (In–Circuit–Tester) mit einem Wert belegt werden können ohne daß die Gefahr besteht, die Ausgangstreiber des Chips zu zerstören.

CLAMP: Dieser Testmodus verhält sich ähnlich wie EXTEST, die Ausgabeleitungen werden mit den in den Zellen gespeicherten Werten belegt, allerdings wird nicht das Boundary–Register zwischen TDI und TDO geschaltet, sondern das Bypass–Register. Um die Vorteile dieser Instruktion zu erkennen, stelle man sich eine Testsituation vor, in der ein Chip A ständig einen konstanten Wert α ausgeben soll. Ein Chip B in der gleichen Testkette wie A soll während des Test verschiedene Werte $\beta_1, \ldots, \beta_n$ ausgeben. Wird nur EXTEST verwendet, so muß der Wert α bei jeder Änderung der Ausgabe von B in A hineingeschoben werden, da β_i auf dem Weg zu B das Boundary–Register von A passieren muß.

[16] Im Pentium–Prozessor von Intel sind z.B. die Operationen IDCODE und RUNBIST implementiert ([39]).

5.2.3 Testsynthese mit Synopsys

Die Erzeugung von Testhardware mit Synopsys ist recht einfach. Alle wichtigen Parameter für den Test–Compiler lassen sich über die Dialogfelder des Design–Analyzers einstellen. Die übliche Vorgehensweise dabei ist, nach einer ersten Optimierung der Hardware die zum internen Scan–Test notwendigen Modifizierungen vorzunehmen. Dazu wird im Menü Tools→Test Synthesis die Schaltfäche Insert Internal Scan Circuitry ausgewählt. Eingestellt werden kann hier u.a. die Art des Scan–Tests (Multiplexed, Clocked–Scan usw.) und die Anzahl der Scan–Ketten. Danach kann über Insert Boundary Scan (JTAG) Circuitry eine minimale Boundary–Scan Hardware hinzugefügt werden[17]. Die Dialogfelder bieten hier die Optionen, ein ID–Register hinzuzufügen und das Signal $\overline{\text{TRST}}$ vorzusehen. Weiterhin kann die Breite des Instruktionsregisters und die Kodierung der Boundary–Scan–Instruktionen angegeben werden. Zum Schluß können durch Anwahl von Test Manager, Create Test Patterns Testvektoren erzeugt werden. Diese werden zunächst im Synopsys internen Format .vdb geschrieben und können über Format Vectors in andere Formate (z.B. VHDL) umgewandelt werden.

Die hier — in aller Kürze — beschriebene Testsynthese wurde auf die Bitblocklogik und die Mittelpunktsberechnung angewendet, dabei interessierte in erster Linie der Mehraufwand an Hardware für die Testunterstützung. Für den gesamten Chip wurde Testhardware generiert und Testvektoren berechnet. Die Resultate dieser Synthesen sind in Kapitel 5.4 zusammengefasst.

[17] Die Synthese von komplexeren Boundary–Scan Strukturen ist möglich, wird jedoch nicht von den Dialogfeldern des Design–Analyzers angeboten.

5.3 Simulationsumgebung

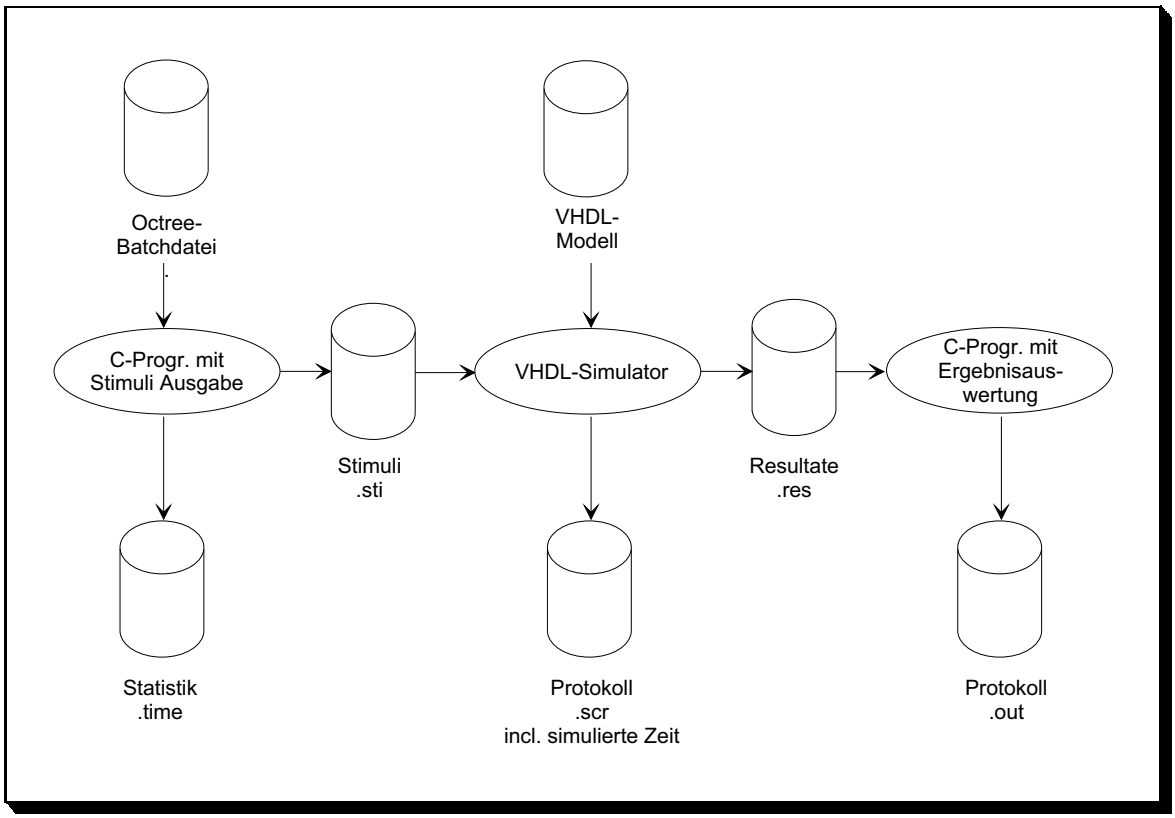


Abbildung 5.9: Simulation des Gesamtentwurfs

Um die Hardware mit möglichst anforderungsgetreuen Eingaben — Stimuli genannt — simulieren zu können, wurde eine enge Kopplung zwischen der Hardwaresimulation und der implementierten Softwarelösung erstellt.

Die Grundidee dieser Kopplung ist, zwei Durchläufe des C–Programms mit derselben Eingabe im Batchmodus zu starten. Im ersten Durchlauf wird jeder Aufruf einer vom Chip zur Verfügung gestellten Operation und jeder Zugriff auf Register des Chips in einer Stimulidatei (.sti, vgl. Abb. 5.9) protokolliert, die von der VHDL–Simulation ausgewertet werden kann. Die VHDL–Simulation schreibt ihrerseits alle vom Chip zurückgelieferten Ergebnisse eine weiteren Datei (.res), die dann im zweiten Durchlauf des C–Programms ausgewertet wird.

Neben der Stimuli– und Resultatdatei wird pro Simulationslauf eine Datei mit einer Statistik über Anzahl und Zeitbedarf von Low–Level–Operationen sowie über die Anzahl von Registerzugriffen erstellt (.time, vgl. Abb. 5.10) sowie die Protokolldateien des Simulationsablaufes (.scr) — dieser ist u.a. die simulierte Zeit zu entnehmen — und des zweiten Programmdurchlaufes (.out). Letztere Datei enthält Meldungen über Inkonsistenzen zwischen den den von der Simulation gelieferten und den von der Software errechneten Resultaten.

5.3.1 Testbench

Zur Simulation eines in VHDL beschriebenen Bausteins (Entity) muß eine Testumgebung (Testbench) erstellt werden, die im einfachsten Fall aus einer Instantiierung des zu testenden Bausteins besteht. Darüberhinaus kann die Testbench weitere zur Simulation sinnvolle Funktionsblöcke enthalten.

Die für den Octchip entworfene und in Abb. 5.12 skizzierte Testbench besteht neben der Instantiierung des Octchip aus einem Block zur Auswertung der Stimulidatei und einem Speichermodell.

```
Octree-Simulation      test4                    Tue Feb 11 20:58:41 1997

Memory used: 95 words
Total running:                                        4 sec  606019 usec
Simulation overhead:                                  0 sec  208192 usec

Operation Reset      called     1 times used          0 sec     388 usec
Operation Root       called    16 times used          0 sec     371 usec
Operation Down       called    55 times used          0 sec    1051 usec
Operation Next       called    20 times used          0 sec     357 usec
Operation Across     called   260 times used          0 sec    4109 usec
Operation Up         called    34 times used          0 sec     733 usec
Operation Attach     called    11 times used          0 sec     198 usec
Operation Set_Full   called    17 times used          0 sec     186 usec
Operation Set_Empty  called     8 times used          0 sec      85 usec
Operation Move2      called     1 times used          0 sec       0 usec
Operation Dump       called     3 times used          0 sec       0 usec
Register MIDDLE_X        read    31 times
Register MIDDLE_Y        read    31 times
Register MIDDLE_Z        read    39 times
Register INDEX_X         read     5 times
Register INDEX_Y         read     5 times
Register INDEX_Z         read     5 times
Register NODE            read   550 times
Register POS             read    20 times
Register NIVEAU          read   215 times
```

Abbildung 5.10: Zeitbedarf und Registerzugriffe: .time–Datei

```
000000000000-100000000100000000000000000      AB0 := 0
000000000000-100000001000000000000000000      IB0 := 0
...
000000000000-100000000000011101000000000      Instruktionsregister := attach
000000000000-011ZZZZZZZZZZZZZZZZZZZZZZZZ      Octchip aktivieren
000000000000-100000000000000101000000100      down 4
000000000000-011ZZZZZZZZZZZZZZZZZZZZZZZZ
000000000000-100000000000001111000000000      set_full
000000000000-011ZZZZZZZZZZZZZZZZZZZZZZZZ
000000000000-100000000000000111000000000      next
000000000000-011ZZZZZZZZZZZZZZZZZZZZZZZZ
000000000000-100000000000001101000000000      attach
000000000000-011ZZZZZZZZZZZZZZZZZZZZZZZZ
...
000000000000-100000000000010101000000000      dump
000000000000-011ZZZZZZZZZZZZZZZZZZZZZZZZ
```

Abbildung 5.11: Beispiel einer Stimuli–Datei (Test der Navigationsroutinen)

Hinzu kommen vier kleinere Blöcke: zwei Generatoren für die Signale CLK und RESET sowie zwei Verzögerungsglieder zur Modellierung der Antworten $\overline{\text{BUSGR}}$ und $\overline{\text{MBUSY}}$ auf die Anforderungen $\overline{\text{BUSRQ}}$ und $\overline{\text{MEMRQ}}$:

```
1  architecture toms of test_chip is
2  ...
3  begin
4  -- Power-On Reset:
5    RESET <= '1', '0' after RSTDELAY;
6
7  -- Taktgenerator:
8    TAKT: process
9          begin
10           if (ENABLE = '1')
11              then CLK <= '0', '1' after CLKDELAY;
12              else wait;
13           end if;
14           wait for CLKDELAY+CLKDELAY;
15         end process;
16
17 -- Antwort auf busrq
18   BRQ: process
19        begin
20          busgr <= '1';
21          wait until busrq = '0';
22          busgr <= '0' after BRQDELAY;
23          wait until busy = '1';
24        end process;
25
26 -- Antwort auf mbusy
27   MBSY: process
28         begin
29           mbusy <= '1';
30           wait until (clk'event and clk='1' and memrq = '0');
31           mbusy <= '0';
32           wait for MEMDELAY;
33         end process;
```

Das in Zeile 10 verwendete Signal ENABLE dient zum „Abschalten“ des Taktgenerators und somit zum Beenden einer mit run gestarteten Simulation: Dieser Befehl veranlaßt den Simulator erst dann abzubrechen, wenn auf allen Signalen keine Änderungen mehr auftreten, was bei einem Taktgenerator ohne ENABLE nie erreicht wird. Das Signal ENABLE wird mit '1' initialisiert und beim Erreichen des Endes des Stimulidatei auf '0' gesetzt[18].

Die Stimuli–Auswertung wertet die Eingabedatei `octchip.sti` aus, die pro Zeile je einen Stimulus für die Signale $\overline{\text{CE}}$, $\overline{\text{CS}}$ und CR_$\overline{\text{W}}$, eine Belegung des Adress– und des Datenbus sowie bei Registerzugriffen eine Prüfziffer („Magic Number“, vgl. 5.3.2), die lediglich zusammen mit dem Ergebnis des Registerzugriffs in die Resultatdatei geschrieben wird:

38 26	25	24	23	22 16	15 0
Magic–Number	$\overline{\text{CE}}$	$\overline{\text{CS}}$	CR_$\overline{\text{W}}$	Adresse	Datum

Abb. 5.11 zeigt ein Beispiel einer (handgeschriebenen) Stimulidatei. So geartete Dateien wurden als Eingaben für ersten Tests der Navigationslogik verwendet: Nach einer Folge von Operationen

[18] Eine einfachere, aber „brutale“ Methode wäre gewesen, die Stimulidatei mit einer Leerzeile zu beenden. Der Simulator hätte dann mit der Fehlermeldung `STD.TEXTIO.READ(CHARACTER) failed, trying to extract a value from '' ''`. abgebrochen.

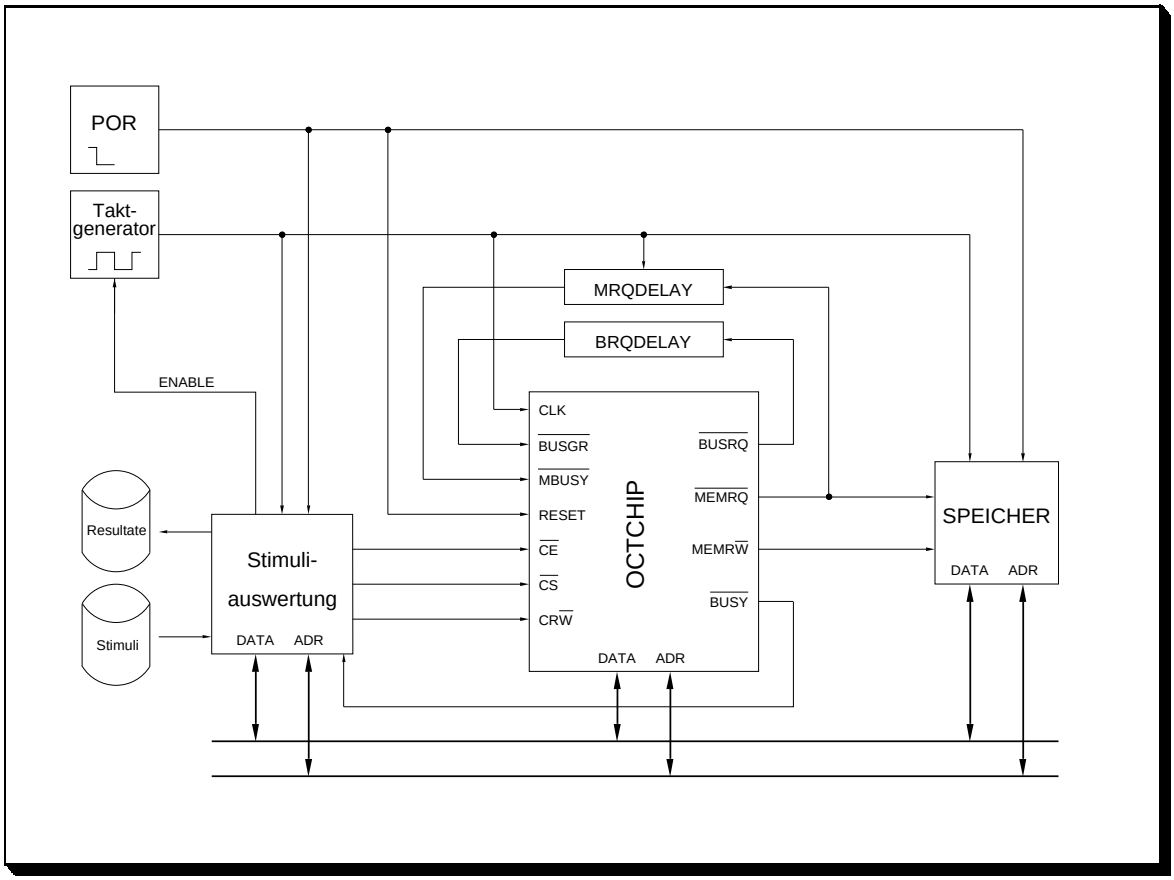


Abbildung 5.12: Testbench

wurde der Speicher der Testbench (s.u.) ausgegeben. Die vom Chip veränderten Speicherzellen und die von dump ausgegebenen Werte der Register der Navigationslogik konnten dann mit errechneten Sollwerten verglichen werden.

Der Stimuli Auswertungsprozeß wartet zunächst, bis der Chip inaktiv wird (Zeile 39). Ist der aktuell bearbeitete Stimulus ein Registerzugriff, was durch $\overline{CS}$ = '0' und CR_$\overline{W}$ = '1' indiziert wird, werden der Inhalt des Datenbus und die „Magic Number“ in die Resultatdatei geschrieben (Zeilen 40 – 44). Anschließend wird — falls nicht das Ende der Stimulidatei erreicht ist — der nächste Stimulus gelesen und den Signalen des Chips zugeordnet (Zeilen 45 – 61). Ist das Ende der Eingabedatei erreicht, wird durch Rücksetzen von ENABLE der Taktgenerator deaktiviert und der Simulationslauf endet.

Zur Ein– und Ausgabe werden Funktionen der Pakete STD.TEXTIO und IEEE.STD_LOGIC_TEXTIO benutzt: In STD.TEXTIO sind die Datentypen TEXT (ASCII–Datei) und LINE (Zeiger auf Zeichenkette) definiert. Nach den Deklarationen

```
file Fin : TEXT is in  "octchip.sti";
file Fout: TEXT is out "octchip.res";
```

kann mit der Funktion READLINE eine Zeichenkette aus Fin gelesen und mit WRITELINE in Fout geschrieben werden. Die Konvertierung zwischen einer Zeichenkette und einem std_logic_vector nehmen die Funktionen READ und WRITE aus IEEE.STD_LOGIC_TEXTIO vor (vgl. Zeilen 43 und 48).

```
34   SIM: process
35        variable L: line;
36        variable STIM: std_logic_vector(38 downto 0);
37        variable RES: std_logic_vector(28 downto 0);
38        begin
39            wait on CLK until (clk='0' and busy='1' and RESET ='0');
40            if (cs='0' and CR_w='1')
41               then  RES(28 downto 16) := STIM(38 downto 26);
42                     RES(15 downto 0 ) := DBUS(15 downto 0);
43                     write(L, RES); writeline(Fout, L);
44            end if;
45            if endfile(Fin)
46               then ENABLE <= '0';
47                    wait;
48               else readline(Fin, L); read(L, STIM);
49                    ce   <= STIM(25);
50                    cs   <= STIM(24);
51                    CR_w <= STIM(23);
52                    if (STIM(24)='0') then
53                      ABUS(23 downto 7) <= NULADR(23 downto 7);
54                      ABUS( 6 downto 0) <= STIM(22 downto 16);
55                      DBUS <= STIM(15 downto 0);
56                    else
57                      ABUS <= "ZZZZZZZZZZZZZZZZZZZZZZZZ";
58                      DBUS <= "ZZZZZZZZZZZZZZZZ";
59                    end if;
60            end if;
61        end process;
```

Da die Antwortszeit des Speichers bereits durch oben beschriebenes Verzögerungsglied modelliert wird, kann der Speicher durch ein einfaches Array

```
type MEMTYPE is array(0 to MEMSIZE) of DATAbus;
```

beschrieben werden, auf das bei $\overline{\text{MEMRQ}}$ = '0' ∨ MR$\overline{\text{W}}$ = '1' lesend, bei $\overline{\text{MEMRQ}}$ = '0' ∨ MR$\overline{\text{W}}$ = '0' schreibend zugegriffen wird. Die Umrechnung der auf dem Adreßbus liegenden Adresse vom Typ std_logic_vector in einen Arrayindex vom Typ integer übernimmt die triviale Funktion ABUS_TO_INT.

```
62   RAM: process(CLK)
63           variable MEM: MEMTYPE;
64           variable ADRESSE: integer range 0 to 16777216;
65
66         function ABUS_TO_INT (stdl: in ADRbus) return integer is
67           variable i, akk: integer;
68           begin
69             akk := 0;
70             for i in stdl'range loop
71                 if stdl(i) = '0'
72                    then akk := akk * 2;
73                    else akk := (akk * 2) + 1;
74                 end if;
75             end loop;
76             return akk;
77           end ABUS_TO_INT;
78
79         begin    -- RAM process
80           if (clk'event and clk = '1' and memrq = '0' and RESET = '0')
81              then ADRESSE := ABUS_TO_INT(ABUS);
82                   if (MR_w = '0')
83                      then MEM(ADRESSE) := DBUS;
84                      else DATA <= MEM(ADRESSE);
85                   end if;
86           end if;
87         end process;
```

Eine Entkopplung zwischen dem Speicher und dem Datenbus beim Lesezugriff nimmt der in einem separaten Prozeß beschriebene Tristate–Treiber RAMDRV vor:

```
88   RAMDRV: process(DATA, memrq, MR_w)
89            begin
90              if (memrq = '0' and MR_w = '1')
91                 then DBUS <= DATA;
92                 else DBUS <= "ZZZZZZZZZZZZZZZZ";
93              end if;
94            end process;
```

Den Abschluß der Testbench bildet die Instantiierung des Octchip und die Konfiguration der Testbench:

```
95   octchip: chip port map(CLK, DBUS, ABUS, ce, busgr, RESET,
96                          cs, CR_w, mbusy, busy, busrq, memrq, MR_w);
97 end toms;
98
99 configuration CFG_TB_CHIP of test_chip is
100   for toms
101       for octchip: chip
102           use entity work.chip(STRUC);
103       end for;
104   end for;
105 end CFG_TB_CHIP;
```

5.3.2 Anpassung der Softwarelösung

Die Softwarelösung wurde zur Kopplung mit der Simulation um die Ausgabe von Stimuli und der Messung der während der Bearbeitung von Low–Level–Operationen verstrichenen Zeit erweitert. Zusätzlich werden die Anzahl der Aufrufe von Low–Level–Operationen und die Zugriffe auf die Chipregister gezählt.

Bei der Erweiterung der Softwarelösung sollte einerseits die Möglichkeit bestehen, die Simulationskopplung über eine Compileroption zu deaktivieren, um die „Standalone"–Variante frei von Abfragen der Art `if (simulate) ...` zu halten, andererseits sollte die Adaption der vorhandenen Routinen möglichst einfach gehalten werden. Beide Anforderungen wurden durch Einführung einer Reihe in beiden Modi unterschiedlich definierter Makros erfüllt. Ist der Simulationsmodus bei der Compilierung aktiviert, werden die Makros vom Präprozessor durch Aufrufe von Prozeduren zur Simulationskopplung ersetzt, andernfalls durch eine Leerzeile bzw. eine einfache `return` Anweisung.

```
1  /*
2    WRITESTIM(O, Q, P) : Opcode O, Baumnr. Q, Parameter P
3                         und "Chip enable" in Stimulidatei schreiben
4    ENDSTIM(O)         : Abschluss fuer WRITESTIM mit Opcode O
5    RETURNREG(N, Q, S) : ein Datum aus der Resultatdatei der Simulation lesen
6                         N: Registercode, Q: Baum, S: Sollergebnis
7  */
8
9  #ifdef simulate
10     #define WRITESTIM(O, Q, P)    WriteStimulus((O), (Q), (P));
11     #define ENDSTIM(O)            EndStimulus((O));
12     #define RETURNREG(N, Q, S)    ReadRegister((N), (Q));\
13                                  return GetRegister((S), (N));
14 #else
15     #define WRITESTIM(O, Q, P)
16     #define ENDSTIM(O)
17     #define RETURNREG(N, Q, S)    return (S);
18 #endif
```

Damit beschränken sich die Modifikationen in den Low–Level–Routinen darauf, den die Octree–Operation simulierenden Programmcode mit WRITESTIM und ENDSTIM einzuklammern:

```
void down(int q, int p)
{
  WRITESTIM(OPC_DOWN, q, p)
  if (((*tree[q] != EMPTYNODE) && (*tree[q] != FULLNODE))
  ...
  ENDSTIM(OPC_DOWN)
}
```

RETURNREG ersetzt bei den Funktionen, die eine einem Register entsprechende globale Variable[19] liefern, die `return`–Anweisung:

```
int niveau(int q)
{
   RETURNREG(REG_NIVEAU, q,  Niveau[q])
}
```

[19] Genauer: im Modul `octree` globale Variable; diese sind von den Modulen der High–Level–Operationen nur über Zugriffsfunktionen verfügbar.

Wir wollen die einzelnen Simulationsprozeduren nicht in ihrer gesamten Länge auflisten, sondern beschränken uns auf einen kurzen Abriß ihrer Funktionsweise:

sim_writemode und sim_readmode initialisieren die Simulationskopplung zum Schreiben der Stimuli bzw. zum Lesen der Resultate. Die Zähler für Registerzugriffe und Operationsaufrufe sowie die Zeitakkumulatoren werden initialisiert, die benutzten Dateien geöffnet und der Zeitpunkt zum Start des Programms wird festgehalten. sim_close schließt den Programmdurchlauf durch Ausgabe der Zähler und Laufzeiten. Insbesondere der Zeitpunkt des Programmendes wird festgehalten. Durch Subtraktion der Startzeit ergibt sich die zur Durchführung des Programms benötigte Zeit.

WriteStimulus schreibt den Stimulus zum Laden des Instruktionsregisters mit dem spezifiziertem Opcode und den Parametern q und p (vgl. 4.4.2). Der dem Opcode korrespondierende Zähler für die Operationsaufrufe wird inkrementiert und die Startzeit der Operation festgehalten. EndStimulus hält die Endzeit einer Operation fest und addiert die Differenz zwischen Start– und Endzeit zum Zeitakkumulator dieser Operation.

Den Stimulus zum Auslesen eines Registers schreibt die Prozedur ReadRegister. Neben den Stimuli für den Octchip wird eine „Magic–Number“ geschrieben — eine aus der Binärkodierung eines fortlaufenden Zählers für Registerzugriffe und der Registeradresse bestehenden Prüfziffer —, die von der Testbench in die Resultatdatei übernommen wird. GetRegister kann beim zweiten Programmdurchlauf anhand dieser Zahl prüfen, ob das im zweiten Durchlauf gelesene Datum mit dem im ersten Durchlauf angefragtem Register korrespondiert[20]. Weiterhin vergleicht GetRegister im zweiten Programmdurchlauf das gelesene Datum mit dem von der Software errechneten Wert. Stimmen diese nicht überein, wird der errechnete Wert zurückgeliefert und eine Warnung ausgegeben.

Zur Zeitmessung wird die Systemroutine

```
int gettimeofday(struct timeval *tp, struct timezone *tzp)
```

aus `<sys/time.h>` verwendet (vgl. Unix–Manualpage „gettimeofday(2)“), die die aktuelle Zeit in *tp und die hier nicht weiter benötigte Zeitzone in *tzp zurückliefert. Zum Inkrementieren der Zeitakkumulatoren — dabei handelt es sich um einen Arrayeintrag vom Typ timeval pro Opcode — um eine Zeitdifferenz wurde die Prozedur

```
void calctime(struct timeval *start, struct timeval *end, struct timeval *erg)
```

erstellt, die Überläufe bei der Zeitaddition und –Subtraktion berücksichtigt.

```
struct timeval  sim_start, sim_end;
struct timezone sim_timezone;
...
gettimeofday (&sim_start, &sim_timezone);  /* in WriteStimulus: Startzeit bestimmen */
...
gettimeofday(&sim_end, &sim_timezone);     /* in EndStimulus: Endzeit bestimmen     */
calctime(&sim_one, &sim_end, &sim_time[opcode]);
```

5.3.3 Automatisierung des Ablaufes

Der gesamte Simulationslauf wird durch ein Shellskript automatisiert, das die beiden Durchläufe der Softwarelösung und den Simulator startet sowie die vom C–Programm verwendeten Dateien *name*.sti und *name*.res mit den von der Testbench verwendeten Dateien octchip.sti und octchip.res verbindet. Das Skript wird mit dem Namen des Octree–Batchfiles als Parameter gestartet[21]:

```
#!/bin/sh
#
# Shellscript zum Aufruf der Octchipsimulation                    201296 TK
#
```

[20] Ferner ist die „Magic–Number“ beim manuellem Auffinden eines Registerzugriffs in der Resultatdatei oder einer Waveform–Ausgabe des VHDL–Debuggers sehr nützlich.

[21] $1 liefert in Shellskripten den ersten Kommandozeilenparameter des Aufrufs.

```
octsim -w $1
rm octchip.sti
ln -s $1.sti octchip.sti
vhdlsim -t ns -i octsim.inc -s $1.scr cfg_tb_chip
mv octchip.res $1.res
octsim -r $1 | tee $1.out
```

Eine weitere Automatisierung wird durch Starten des Simulators im Batchbetrieb erreicht. Der Parameter -s *includefile* bewirkt, daß die in der Datei *includefile* enthaltenen Simulatorbefehle abgearbeitet werden. Darüberhinaus werden alle Ausgaben des Simulators in einer Datei protokolliert (-s *scriptfile*).

Die Batchdatei für den VHDL–Simulator enthält schließlich die Befehle zum Starten der Simulation, zum Schreiben des Speicherinhaltes in die Datei mem.dump, zum Ausgeben der simulierten Zeit und zum Beenden des Simulators:

```
-- Simulator-Batchfile fuer Octchip Testbench                    141196 TK
-- zur Verwendung mit Shellscript                                201296 TK
run
cd /test_chip/ram
dump_memory mem mem.dump
fprint "\n\nSimulated Time: %d %t\n" $NOW $TIMEBASE
quit
```

5.3.4 Einschränkungen der Simulationsumgebung

Die Kopplung zwischen der Softlösung und der Simulation erlaubt zwar eine automatische Generierung von Stimuli für das VHDL–Modell des Octchips, jedoch ist diese Einschränkungen unterworfen. So kann das C–Programm nicht auf den Speicher der VHDL–Simulation zugreifen, wodurch Löschen und Verschieben von Kubusbäumen unmöglich wird. Mehrfache Operationen auf Octrees, z.B. die Verschmelzung zweier Kubusmodelle zu einem dritten und anschließende Tranformation des entstandenen Modells sind damit ausgeschlossen.

Für Korrektheitsverifikationen und Laufzeitmessungen am Chip sind diese Einschränkungen jedoch nicht gravierend.

5.4 Resultate

5.4.1 Simulation

Zum Test der Funktionsweise des Chips und zur Messung der Performance wurden bestimmte Modelle und Operationen auf diesen ausgewählt. Einige diese Tests wurden bereits bei der Entwicklung des C–Programms verwendet, um die implementierten Funktionen auf Korrektheit zu überprüfen.

Die zum Test herangezogenen Kubusbäume sind zwar klein, jedoch insofern repräsentativ, als daß das Verhältnis der High–Level–Operationen zu den Low–Level–Operationen sowie das Verhältnis der Softwarelösungen der Low–Level–Operationen zu deren Hardwarependants proportional zur Modellgröße ist. Die Simulation größerer Modelle ist wegen des hohen Verbrauchs an Ressourcen kaum möglich: Bereits die Erzeugung einer Kugel mit Radius $r = 42$ — also einer verhältnismäßig kleinen Kugel — und deren Darstellung als Schnittebene zu $z = 0$ erzeugt eine etwa 50 MB große Stimulidatei. Die Simulation dieses Testlaufs dauert mehrere Stunden.

Es folgen kurze Erkärungen einigen Testläufen sowie eine tabellarische Auflistung ausgewählter Testergebnisse. Da bei den Messungen der Laufzeit der Softwarelösung das Problem auftritt, daß die Routinen zur Kopplung mit der Hardwaresimulation durch das Schreiben der Stimulidatei sehr viel Zeit in Anspruch nehmen, ist in Tabelle 5.1, die die Laufzeiten der Software im Vergleich zu der von der Hardwaresimulation gelieferten Zeit zeigt, neben der Gesamtlaufzeit des C–Programms und dem Anteil der Low–Level–Operationen der durch die Simulationskopplung entstehende Zeitoverhead aufgelistet.

Die große Differenz zwischen Gesamtlaufzeit und Anteil der Low–Level–Routinen läßt sich so erklären, daß mit Rücksicht auf die Laufzeit der Hardwaresimulation relativ kleine Testmodelle ausgewählt wurden und so der Anteil der Routinen für Graphiksystem und Benutzerschnittstelle den Anteil der eigenlichen Octree–Operationen überwiegt. Der hohe Anteil der Low–Level–Routinen an High–Level–Operationen wurde bereits in Kap. 3.5 gezeigt. Die Messungen der Softwarelösung wurden auf einem 133 MHz–Pentium unter Linux vorgenommen, die Hardwaresimulation legt eine Taktperiode von 30 ns zugrunde.

Test	Laufzeit Softwarelösung			simulierte Zeit
	Gesamt	Overhead	Low–Level–Op.	
Kubi setzen („setnodetest“)	$\approx 0,52$sec	$\approx 0,11$sec	2990μsec	84μsec
Setnodetest mit Verschiebung	$\approx 0,87$sec	$\approx 0,30$sec	8082μsec	258μsec
Vereinigung zweier Testmodelle	$\approx 0,78$sec	$\approx 0,11$sec	3368μsec	270μsec
Spiegeln am Ursprung	$\approx 0,98$sec	$\approx 0,10$sec	3002μsec	103μsec
Generierung einer Kugel und Visualisierung der Schnittebene $z = 0$	≈ 80sec	≈ 60sec	$\approx 2,3$sec	$0,061$sec

Tabelle 5.1: Laufzeiten ausgewählter Testoperationen

set_node–Test

Der set_node–Test soll die Korrektheit der Funktionen attach und set_node sowie der Navigationsoperationen sowohl auf Zeiger– als auch auf Bitblockniveaus prüfen und setzt auf jedem Niveau im Modell $q = 2$ einen Kubus auf „voll“.

Damit wird ein Funktionstest der Zugriffslogik, des Knoteninhaltsregisters C, des Bitblockregisters für Modell 2 sowie der μROM-Programme für die Navigationsoperationen erreicht. Da die Ausgaberoutine cross_section Werte wie Mittelpunkt, Index und Knoteninhalt aus den Chipregistern liest, dient dieser Testlauf auch zur Überprüfung der middle–Einheit und des Positionsregisters. Darüber hinaus müssen das Kommunikationsprotokoll des Chips (→ 4.1.1) und die Simulationskopplung (→ 5.3) ihre Korrektheit beweisen.

Zur Verdeutlichung sei hier der Ablauf des Tests skizziert; in Klammern ist die durch den jeweiligen Schritt getestete Operation des Chips aufgeführt:

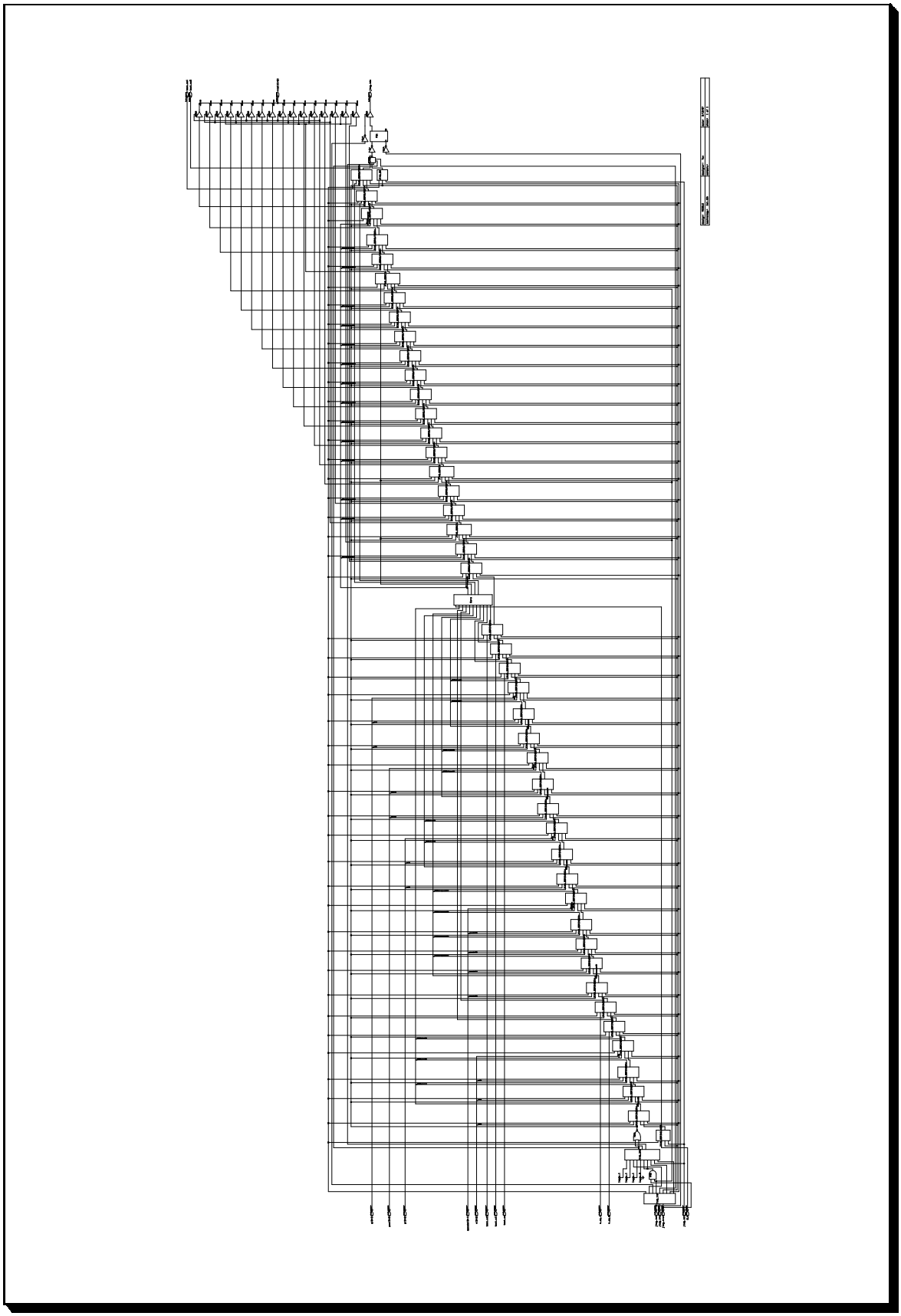


Abbildung 5.13: Mittelpunktseinheit (hier: Core) mit umgebender Boundary–Scan–Logik

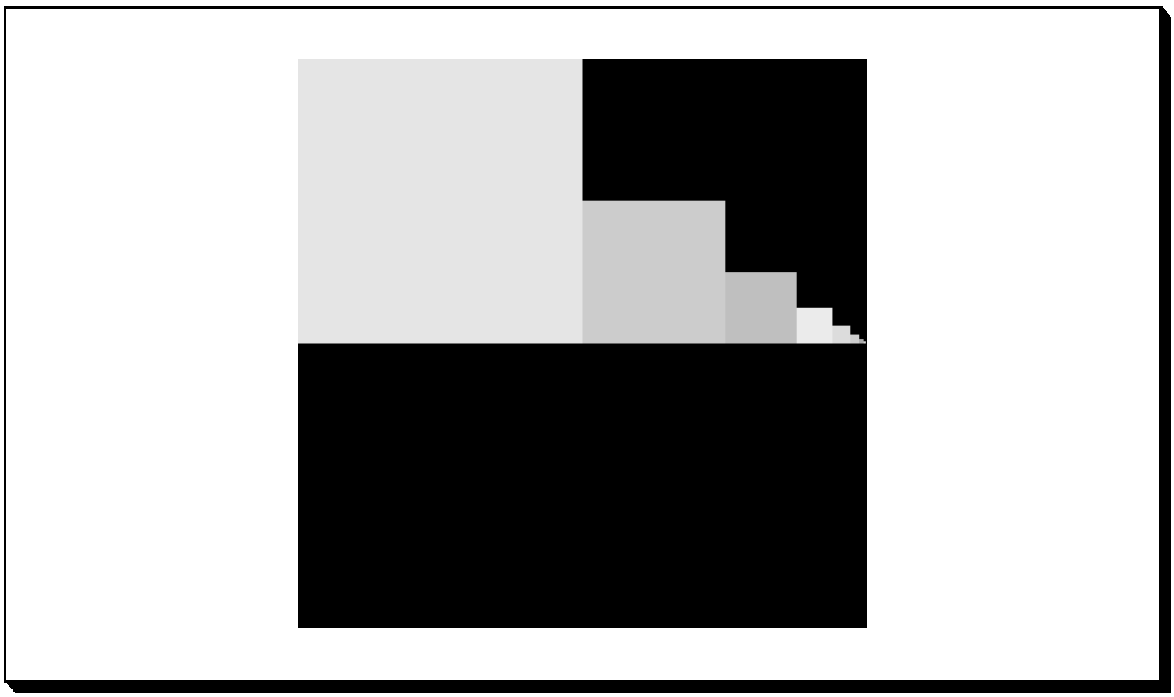


Abbildung 5.14: set_node–Test

- für Niveaus $n = 10$ bis $n = 1$:
  - Setze aktuellen Knoten auf FULL (set–full).
  - Navigiere zu Bruderknoten (Navigation next).
  - Erzeuge Subkubi des aktuellen Knotens (attach).
  - Navigiere in den erzeugten Sub–Baum (Navigation down)
- für Niveau $n = 0$:
  - Setze aktuellen Knoten auf FULL.
- Stelle die Schnittebene $z = 0$ graphisch dar (Auswertung der Resultate).

Abbildung 5.14 zeigt einen mittels `display_layer(2, 0)` (→ 3.4.3.1) erzeugten Schnitt durch das vom set_node–Test erzeugten Objekt.

setmove–Test

Der setmove–Test führt ein Verschiebung auf dem durch set_node–Test generierten Objekt durch. Damit die Transformation durchgeführt werden kann, muß zunächst ein Modellwechsel stattfinden, d.h. das in Modell $q = 2$ generierte Objekt muß mittels move_model2 zu Modell $q = 0$ umbenannt werden (→ 4.6.2.8). Nun kann die Transformation move auf das Modell $q = 0$ angewendet werden, das Ergebnis wird in Modell $q = 2$ abgelegt.

Bei diesem Test wird durch das Verschieben von Modell $q = 2$ nach $q = 0$ sichergestellt, daß der Chip auch einen korrekten Zugriff auf mehrere Modelle im Speicher durchführen kann. Des weiteren wird das Verhalten des Chips bei Transformationen und Wechsel zwischen den Modellen überprüft, z.B. wird getestet, ob die Register beim Umschalten zwischen den Modellen die richtigen Daten liefern und die Daten zum „richtigem“ Zeitpunkt zur Verfügung gestellt werden (internes Timing → 4.2.2). Auch wird festgestellt, ob die Bitblocklogik korrekt arbeitet, da die move Operationen ein gekoppeltes Schieben mit Bitblöcken durchführt (vgl. 4.4.5).

unite–**Test**

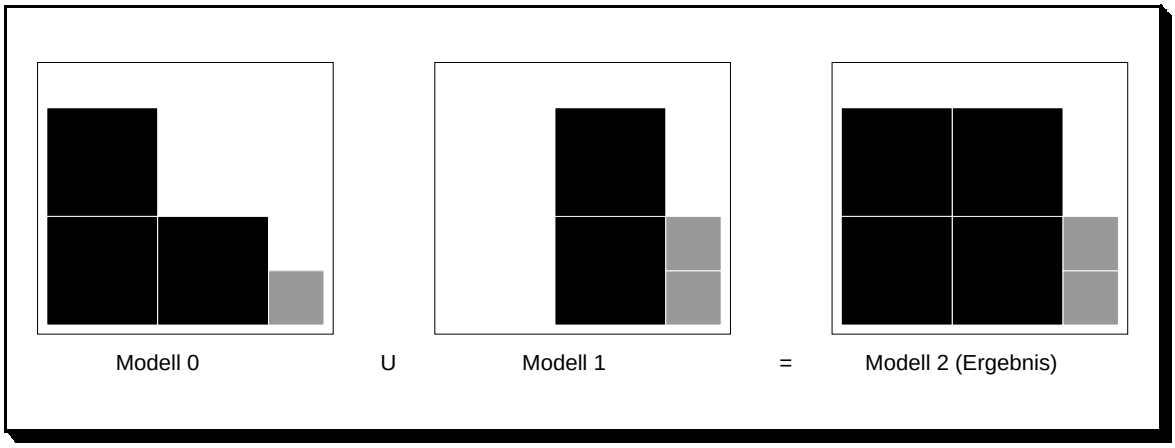


Abbildung 5.15: unite–Test: zwei Modelle und das Ergebnis

Beim unite–Test werden alle drei vom Chip verwalteten Kubusbäume einbezogen, um die Speicherverwaltung sowie alle Register vollends zu überprüfen. Die Bitblocklogik wird getestet, da hier durch die Bitblockalu eine Operation mit drei Registern durchgeführt wird und alle Register synchron ihre Schiebeoperationen durchführen müssen.

Der Test besteht aus der Berechnung der Vereinigung zweier Objekte. Abbildung 5.15 zeigt einen Schnitt duch die zwei Ausgangsobjekte und das Ergebnis.

5.4.2 Synthese

5.4.2.1 Synthese ausgewählter Komponenten

Zur Untersuchung des Trade–Offs zwischen maximalem Platzbedarf und maximaler Taktrate wurden exemplarisch die Einheit zur Mittelpunktsberechnung und die Bitblocklogik ausgewählt. Die Wahl läßt sich damit begründen, daß es sich bei der middle–Einheit um eine sehr rechenintensive Komponente handelt; bei der also Zeit eine kritische Rolle spielt. Ähnlich verhält es sich mit der Bitblocklogik: Hier werden weniger rechenintensive Operationen als aufwändige zeitabhängige Speichertransfers durchgeführt, so das auch hier bei korresponierenden Taktraten Rückschlüsse auf die korrekte Arbeitsweise des jeweiligen Synthese–Ergebnisses des Gesamtentwurfs gezogen werden können.

Die korrekte Arbeitsweise beider Einheiten wurde durch Teststimuli, die Eingabesignalen des Chips entsprechen, überprüft: Die middle–Einheit wurde mit den Opcodes einiger Navigationsoperationen getestet. Dabei mussten in der Testumgebung die von anderen Registern gelieferten Werte adäquat verwaltet werden — z.B. muß nach einem next–Stimulus die aktuelle Position inkrementiert werden —, um nachvollziehbare Simulationsergebnisse zu erhalten. So konnten die von der middle–Einheit berechneten Werte mit einem theoretischen Sollwert verglichen werden.

Darüber hinaus waren einige Fehler im Design bereits in den Ausgaben des Debugger und im Waveformviewer zu erkennen. Es kam hier zu Warnungen, falls die Timing–Voraussetzungen der Flip–Flops (Hold–Time) verletzt wurden, woduch auf eine fehlerhafte (zu langsame) Arbeitsweise geschlossen werden konnte.

Ähnlich wurde bei der Bitblocklogik vorgegangen, allerdings wurden hier zuerst die vier einzelnen Bitblockregister und die Bitblockalu entsprechend ihres Einsatzes im Chip verschaltet (vgl. 4.4.5). Zum Testen wurden einige typische Operationen mit Bitblöcken durchgeführt: das Setzen von Kubi, Kopieren von Bitblöcken sowie das Verknüpfen zweier Bitblöcke über die Bitblockalu. Auch hier ließ

sich die fehlerhafte Arbeitsweise einiger Synthese–Ergebnisses schon beim Durchlauf im Debugger dahingehend erkennen, daß bei den internen Schiebeoperationen der Register zu große Verzögerungen auftraten und vom Debugger fehlerhafte Inhalte der Register angemerkt wurden. Insbesondere fällt hier das Bitblockregister des Modells $q = 2$ auf, welches bei einigen Synthese–Ergebnissen als einziges fehlerhaft war. Dieses Ergebnis läßt sich auf die größerer Komplexität und somit auch auf die längere Verzögerunge des Speicherelements, bedingt durch die Möglichkeit in diesem Register einzelne Kubi setzen zu können, zurückführen.

Constraints		Bitblocklogik		middle–Einheit	
vorgeg. Wert		ermittelter Wert	Anz. Zellen	ermittelter Wert	Anz. Zellen
CLK = 5 ns		AREA = 12640	4704	AREA = 4969	1597
CLK = 10 ns		AREA = 12035	4198	AREA = 5341	1831
CLK = 20 ns		AREA = 11485	4120	AREA = 5407	1873
CLK = 30 ns		AREA = 11485	4120	AREA = 5110	1700
CLK = 30 ns	'ungroup'	AREA = 11464	4137	AREA = 4530	1331
CLK = 30 ns	'flatten logic'	AREA = 11485	4120	AREA = 5119	1701
CLK = 30 ns	mit Scan–Test	AREA = 13247	5385	AREA = 5330	1992
CLK = 30 ns	mit Scan/JTAG	AREA = 15942	5920	AREA = 6450	2227
CLK = 40 ns		AREA = 11485	4120	AREA = 5207	1732
CLK = 50 ns		AREA = 11485	4120	AREA = 4650	1483
MAX. AREA = 4400		—	—	AREA = 4233 CLK > 23 ns	1067
MAX. AREA = 4000		—	—	AREA = 4230 CLK > 23 ns	1067
MAX. AREA = 10000		AREA = 10692 CLK > 14.45 ns	3670	—	—
MAX. AREA = 9000		AREA = 10692 CLK > 14.45 ns	3670	—	—

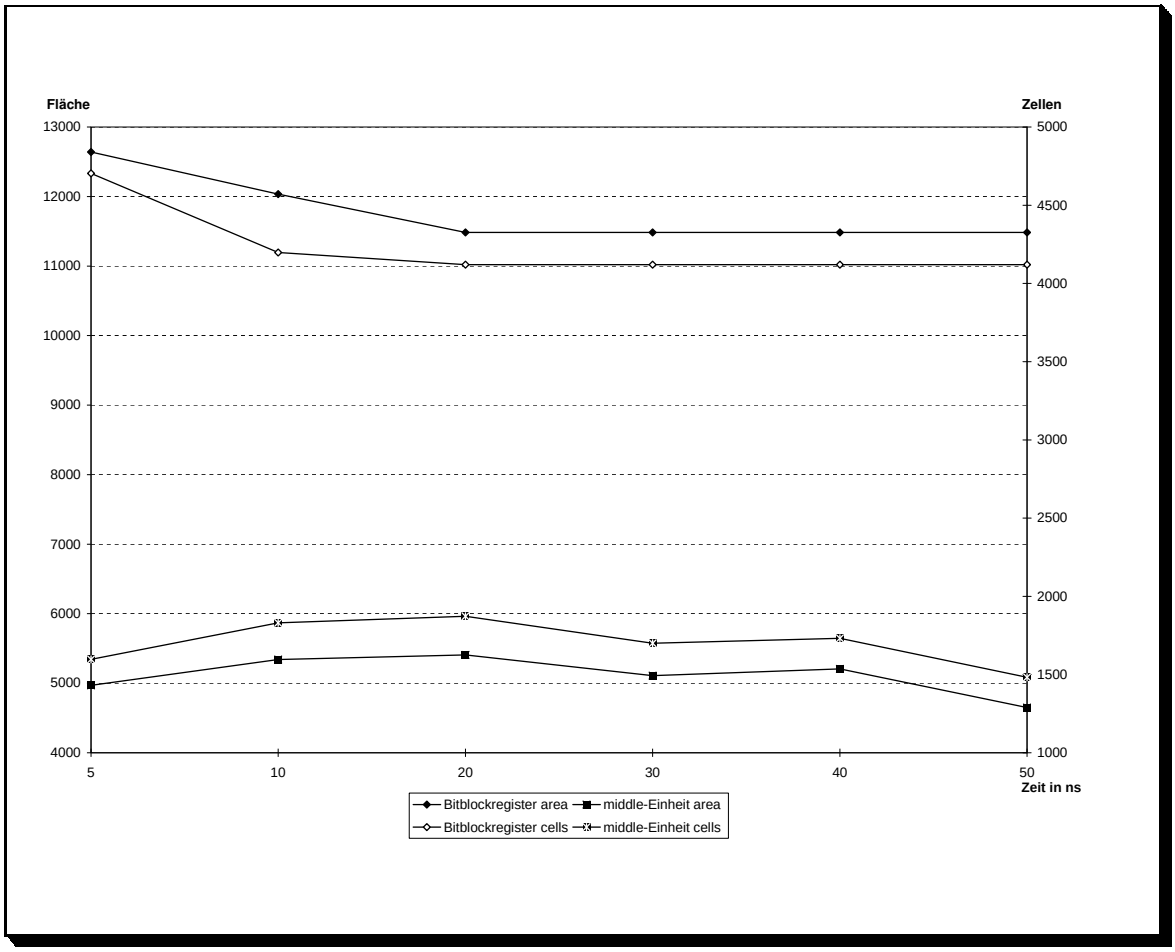
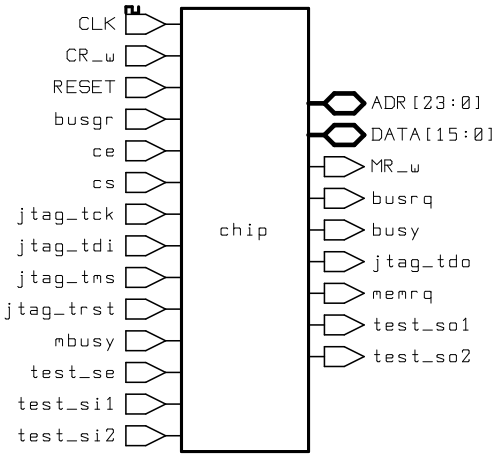


Abbildung 5.16: Time/Area–Tradeoff bei Bitblocklogik und Mittelpunktsberechnung

5.4.2.2 Synthese des Gesamtentwurfs



Den Abschluß der Vorstellung der Ergebnisse aus Synthese und Simulation sollen die Syntheseergebnisse des gesamten Octchip bilden. Der Entwurf wurde zunächst mit einer Taktvorgabe von 30 ns berechnet. Die Vorgabe eines Taktes statt einer maximalen Chipfläche ist dadurch begründet, daß kein Anhaltspunkt über die Größe des Chips vorlag und demzufolge keine Synthesen mit beschänkter Fläche erfolgen konnten. Die Festlegung der Taktrate auf 30 ns ist durch einen späteren Anschluß des Chips an mit 33 MHz getakteten PCI–Busse motiviert. Der 30 ns–Entwurf wurde um interne Scan–Test–Methoden und eine minimale JTAG–Boundary–Scan–Logik erweitert. Ausgehend von der errechneten Fläche wurde eine zweite Synthese mit der Vorgabe einer maximalen Chipfläche durchgeführt.

Design	vorgeg.	errechnet	Anz. Zellen
Chip	CLK = 30 ns	AREA = 52142	15042
Chip mit Scan–Logik	CLK = 30 ns	AREA = 60956	21569
Chip mit Scan– und JTAG–Logik	CLK = 30 ns	AREA = 63247	22019
Chip	AREA = 52000	CLK > 116.66 ns (s.u.) AREA = 50110	15179

Die Ressourcen des mit CLK = 30 ns berechneten Entwurfs verteilen sich wie folgt:

	Fläche	Zellen
Mikrorom	692	504
Steuerwerk (inkl. Mikrorom)	1328	777
I/O–Treiber	247	84
Rechenwerk	Rest	

Die bei der Arearestriktion errechnete Taktrate ist nicht realistisch, da der Design–Analyzer keine bidirektionalen Pfade erkennt. Der zur Ermittlung dieser Taktrate zugrundegelegte maximale Pfad führt z.B. über Leitungen von DATA_OUT in den I/O–Treiber (vgl. 4.2.1) und dort über DATA nach DATA_IN, was nicht auftreten kann, da die entsprechenden Treiber exklusiv DATA_IN oder DATA_OUT mit DATA verbinden. In [28] wird zu diesem Problem vorgeschlagen, die Ein– und Ausgangsleitungen der Tristate–Treiber mit Input– bzw. Outputdelays zu versehen und so den Pfad explizit zu unterbrechen; allerdings konnte auf diese Weise keine Verbesserung erzielt werden[22].

[22] Eine tiefergehende Beschäftigung mit dieser Problematik wurde von der ablaufenden Abgabefrist für diese Arbeit unterbrochen.

Kapitel 6

Zusammenfassung und Ausblick

Zum Abschluß dieser Abhandlung wollen wir einen Blick zurück werfen und einen Blick nach vorn wagen: Wir wollen die Resultate unserer Arbeit noch einmal zusammenfassen und Möglichkeiten vorstellen, mit der Entwicklung des Octchip fortzufahren.

6.1 Zusammenfassung

Das Resultat unserer Arbeit läßt sich nicht auf den Simulationswert eines Chips beschränken; es bezieht viele Erfahrungen im Bereich des Hardwareentwurfs mit ein. Im folgenden wollen wir daher — neben dem „Schlußwort“ zum Octchip — einige unserer Beobachtungen notieren.

- Zum Thema Octchip darf man wohl behaupten, daß die Ergebnisse der Simulation vielversprechend sind und man auf einen Prototypen des Chips gespannt sein darf. Ein Vergleich der Laufzeiten einer reiner Softwarelösung mit den von der Hardwaresimulation errechneten Zeiten zeigt, daß eine angepaßte Architektur durchaus in der Lage ist, die geforderten Aufgaben beim Umgang mit Octrees schneller zu bewältigen als eine reine Softwarelösung.

- Unsere Erfahrung zeigt, daß die größten Probleme beim Hardwareentwurf in kleinen Details stecken; jenen Details, die in einem theoretischen Entwurf gern übersehen werden. Dies ist wohl der Grund dafür, daß sich der Hauptaugenmerk unserer Arbeit immer mehr von einer Synthese und Simulation des in [1] vorgestellten Entwurfs zu einer Korrektur und Erweiterung jener Vorlage verlagert hat.

- Rückblickend stellt sich uns die Frage, ob die Vorgehensweise[1] bei der Realisierung des Octchip sinnvoll war. Es ist nicht klar, ob die Umsetzung eines Entwurfs auf Gatterebene nicht mehr Probleme aufgeworfen als gelöst hat und wir nicht mit Werkzeugen der High–Level–Synthese schneller zu einem Ziel gekommen wären — vielleicht sogar zu einem besseren.

- Die Arbeit mit Synopsys zeigt[2], daß der Umgang mit diesem System einiges an „Fingerspitzengefühl“ erfordert, daß nur durch viel „Experimentieren“ mit den zur Verfügung gestellten Werkzeugen erworben werden kann. Dies betrifft sowohl die benutzten Formulierungen in VHDL[3]

[1] Durch die Aufgabenstellung war der Entwurf in [1] Anhang A als Ausgangspunkt der Realisierung vorgegeben.

[2] und die Lektüre von [28] — einem wirklich empfehlenswertem Buch, das uns leider erst gegen Ende der Arbeit zur Verfügung stand — bestätigt dies

[3] In unserer Praxis wurden z.B. unerwünschte Latches in Designs eingefügt. Der Grund dafür war — wie auch [28] zu entnehmen ist —, daß if–Ausdrücke ohne else–Zweig verwendet wurden:

```
if (ENABLE = '1')
 then Q <= INPUT;
end if;
```

als auch die Vorbereitung eines Designs zur Optimierung im Hinblick auf Strukturierung und Größe von Komponenten[4].

- Bei der Arbeit mit VHDL sollte man sich stets vor Augen halten, daß es sich um eine Hardwarebeschreibungssprache und nicht um eine Programmiersprache im herkömmlichen Sinne handelt. Gerade durch die an klassische Programmiersprachen angelehnte Syntax besteht oft die Gefahr, VHDL–Beschreibungen als sequentiell ablaufendes Programm zu sehen und nicht als Beschreibung einer parallel arbeitenden Hardware. Daneben sollte man sich im Bezug VHDL als Eingabe für Synopsys darüber im Klaren sein, daß nicht alle VHDL–Sprachkonstrukte von Synopsys unterstützt werden[5] — ein Phänomen, daß in klassischen Programmiersprachen eher selten auftritt[6] und dementsprechend gewöhnungsbedürftig ist.

6.2 Ausblick

Mit dem Abgabetermin dieser Arbeit ist die Entwicklung des Octchip zwar an einen Punkt gelangt, der eine Dokumentation des Erarbeiteten rechtfertigt, endgültig abgeschlossen ist sie jedoch nicht. Sowohl in den Prozeß der Hardwaresynthese als auch in die Erweiterung der Funktionalität des Chips kann noch Arbeit investiert werden.

6.2.1 Weitere Schritte beim Hardwareentwurf

Als Weiterführung der Arbeit kann im Prozeß des Hardwareentwurfs mit der gewonnenen Gatternetzliste fortgeschritten werden. Eine Möglichkeit wäre es mit einem ‚Place– and Route–Tool‘, einem Plazierungs– und Verdrahtungswerkzeug, den Herstellungsprozeß weiterzuführen. Ein solches Programm[7] bildet eine Gatternetzliste in ein Chiplayout ab und dient dazu die für die Fertigung eines ASICs[8] nötigen Masken zu entwerfen. Solche Programme berücksichtigen auch durch die Fertigungstechnik gegebene Vorgaben (design–rules). So muß z.B. bei der Erstellung der Masken die für die Fertigung benötigt werden eine Mindestgröße und ein Mindestabstand der Leiterbahnen gegeben sein damit der Fertigungsprozeß[9] fehlerfrei erfolgen kann.

Die Einheiten des Designs müssen so plaziert werden, daß es zu keinen bzw. wenigen Überschneidungen der Leiterbahnen kommt, andererseits dürfen die Leiterbahnen nicht zu lang werden, um sowohl die benötigte Chipfläche gering zu halten wie auch zu große Differenzen in den Signallaufzeiten zu verhindern. Die Signallaufzeiten sind ein allgemeines Problem im Layout, nicht nur für Datenleitungen, sondern auch für die Taktleitung, die die zentrale Rolle der Synchronisation übernimmt. Als Beispiel sei hier der Fall aufgezeigt, daß zwei Register in Bezug auf ihre Datenpfade nahe beieinander liegen, aber unterschiedlich „weit“ von der Taktleitung entfernt sind. Bei einem Datentransfer zwischen diesen beiden Registern kann es nun aufgrund der längeren Laufzeit des Taktsignals zu dem einen Register dazu kommen, daß die Daten den Takt „überholen“. Aus diesem Grund wird bei einem Layout meißt das Taktsignal zentral in das Layout geführt und von dort baumartig gleichmäßig im Design verteilt.

Aus den oben aufgeführten Gründen ist es ersichtlich, daß nach Fertigstellung des Layouts vor der eigentlichen Fertigung eine weitere Simulation des Designs erfolgt um festzustellen, ob die vorgegebenen Spezifikationen immer noch Gültigkeit haben.

Ein anderer Weg der Fortsetzung der Arbeit in Richtung Hardwareentwurfsprozeß ist die Möglichkeit der Programmierung eines FPGAs. Der Vorteil liegt auf der Hand, es muß nicht der lange und

[4] Konkret ist abzuwägen, ob die Hierarchie eines Entwurfs erhalten bleibt — in diesem Fall wird der uniquify–Befehl verwendet — oder ob kleinere Entwurfseinheiten mit dem ungroup–Befehl zusammengefasst werden. In [28] wird eine Anzahl von 250–5000 Gattern als Größe für eine Einheit zum Optimieren empfohlen.

[5] z.B. Formulierungen der Art `A <= B when C = '0' else D`

[6] Man stelle sich z.B. einen C–Compiler vor, der keine `WHILE`–Schleifen kennt!

[7] z.B. CADENCE

[8] ASIC: (Application Specific Integrated Circuit) eine individuell entwickelte integrierte Schaltung

[9] Gemeint ist hier der Vorgang des Belichtens und Ätzens der einzelnen Materialschichten.

auch kostspielige Weg der Fertigung eines ASICs durchlaufen werden und es können schnell „reale“ Messungen an einer Hardware durchgeführt werden. Von der Theorie her sind alle Voraussetzungen von Seiten des Lehrstuhls gegeben. Mit Synopsys könnte durch Auswahl einer anderen Zieltechnologie eine Gatternetzliste bzw. der Entwurf einer FPGA Programmierung erstellt werden. Diese Gatternetzliste wird dann von einem Layoutprogramm[10] für FPGAs hinsichtlich der Ausnutung der FPGA–Zellen optimiert.

Allerdings sind auch hier einige Hindernisse zu erwarten: Die Anzahl der benötigten FPGA Zellen wird sehr hoch sein, so daß das Design über mehrere FPGAs, die dann entsprechend verschaltet werden, verteilt werden muß. Auch ist vom späteren Einsatz her durch die FPGAs gegenüber einem ASIC eine Grenze[11] gesetzt, da FPGAs ihre Grenzfrequenz bei etwa 10 MHz erreichen.

Der nächste Schritt des Hardwareentwurfs bleibt also offen und unterliegt der Motivation des nächsten Ziels: Möchte man möglichst schnell einen Prototypen des Chips, so ist die Programmierung von FPGAs das näherliegende Ziel. Aber auch das Plazieren und Verdrahten des Designs und der weitere Weg zu einem „richtigem“ Chip ist ein interessantes Ziel.

6.2.2 Erweiterung der Funktionalität

In diesem Abschnitt wollen wir Möglichkeiten auflisten, den Octchip funktional zu erweitern. Die hier aufgelisteten Punkte sind Varianten, die bei der ersten Realisierung des Chips nicht einbezogen wurden, weil sie für eine erste Implementierung nicht essentiell wichtig waren.

- Die Spezifikation des Chips verlangt, daß die drei Kubusbäume in fortlaufender Reihenfolge im Speicher abgelegt werden, zudem kann nur auf Modell $q = 2$ schreibend zugegriffen werden. Bei der Hintereinanderaufführung von Operationen entsteht nun das Problem, daß Modelle im Speicher gelöscht oder verschoben[12]. Eine Implementierung solcher Operationen im Octchip ist sicherlich nicht sinnvoll, da sie sich durch den Prozessor des Wirtsrechners ausführen lassen und sogar evtl. direkt vom Befehlssatz dieses Prozessors unterstützt werden. Der Octchip könnte jedoch dazu beitragen, die Anzahl solcher Operationen zu verringern, indem er den für Kubusbäume zur Verfügung stehenden Speicher als Ringpuffer betrachtet und neben der ersten freien Adresse auch die erste belegte Adresse verwaltet. Wird beim Anlegen eines Modells das Ende des Octree–Speichers erreicht, so wird mit Allozierung von Speicher am Anfang des Octree–Speichers fortgefahren — jedenfalls solange die Endadresse die Anfangsadresse nicht „überholt“, also der Speicher voll ist. Löschen von Modell 0 würde dann bedeuten, die Adresse des ersten belegten Speicherplatzes auf die Anfangsadresse von Modell 1 (falls vorhanden, andernfalls Modell 2) zu setzen. So ließen sich folgender Typ von Anwendungen effizient durchführen:
  - Generiere gewünschten Kubusbaum in Modell 2
  - Verschiebe Modell 2 nach Modell 0
  - Führe Transformation 1 durch (der transformierte Kubusbaum ist in Modell 2 zu finden)
  - Lösche Modell 0
  - Verschiebe Modell 2 nach Modell 0
  - Führe Transformation 2 durch
  - (wiederhole die letzten drei Schritte für weitere Transformationen)
- Die Übersicht der Laufzeiten in Kapitel 3.5 zeigt, daß ein nicht unbeachtlicher Teil der Rechenzeit von den Routinen zur Baumtraversierung verbraucht wird, was angesichts ihrer rekursiven Formulierung auch nicht verwundert. Eine iterative Version von parse_tree zur Behandlung voller Kubi wäre:

[10] z.B. X–ACT

[11] hiermit ist nicht gemeint, daß FPGAs keine Alternative zu ASICs sind; ganz im Gegenteil: FPGAs erlauben die schnellere und flexiblere Umsetzung von Designs, sowie das schnellere und günstigere Beheben von Fehlern im Design und sind besonders bei kleinere Stückzahlen eines Entwurfs kostengünstiger.

[12] Das Verschieben von Modellen im Speicher ist nicht der Transformation move zu verwechseln!

```
void parse_tree (int q, void (*proc)(void), void (*procb)(void))
{
   search_start(q);
   search_next_full(q);
   while( search_found(q) ){
       if (Niveau[q] == BITBLOCKNIVEAU) (*procb)();
       else (*proc)();
       search_next_full(q);
   }
}
```

search_start initialisiert die zur Baumtraversierung benötigten Statusbits und navigiert zum Wurzelknoten. Danach kann mit search_next_full der nächste volle Knoten im Baum gesucht werden (analog dazu search_next_empty). Liefert search_found wahr, so wurde ein Knoten gefunden, andernfalls ist die Suche beendet. Die hier verwendeten Operationen könnten leicht in Hardware implementiert werden; hierzu wären eine Erweiterung des Mikroroms und des Opcode–Dekoders sowie die Einführung von Statusbits search_finished und Mikrorom–Abfragen if_node_full/ if_node_empty nötig.

- Die Bitblockregister werden derzeit bei strukturkonformen Transformationen (Spiegeln und Klappen) vom Mikrorom angesteuert. Durch Einführung eines zweiten Instruktionsregisters, in das ein Opcode zur Ansteuerung der Bitblockregister geschrieben wird, könnte der Benutzer alle von den Bitblockregistern durchführbaren Schiebeoperationen veranlassen. Dies würde zum einen die Flexibilität der Bitblocklogik erhöhen, zum anderen das Mikrorom verkleinern, da nur noch ein Opcode zur Durchführung von Schiebeoperationen vorgesehen werden muß. Dieses Bitblock–Instruktionsregister müßte die folgenden Informationen enthalten:
  - Quellbitblock 1: Identifikation (2 Bit), Achse (2 Bit), Richtung (1 Bit)
  - Quellbitblock 2: Identifikation (2 Bit), Achse (2 Bit), Richtung (1 Bit)
  - Zielbitblock: Identifikation (2 Bit), Achse (2 Bit), Richtung (1 Bit)
- In Kapitel 3.3.1 wurden Operationen zum Test einzelner Bits oder Bitfolgen innerhalb eines Wortes vorgestellt:

```
void leftright (int i, int a,
            int *left, int *right)
```

Ein Durchsuchen des Quelltextes der Softwareimplementierung zeigt, daß bei diesen Operationen stets für den Parameter i das Niveau eingesetzt wird. Im Chip könnte man eine Einheit vorsehen, die genau diese Operationen dürchführt, indem ein reserviertes Register mit dem Wert für a beschrieben wird und danach Register mit den errechneten Werten für left und right ausgelesen werden können (vgl. Abb. 6.1).

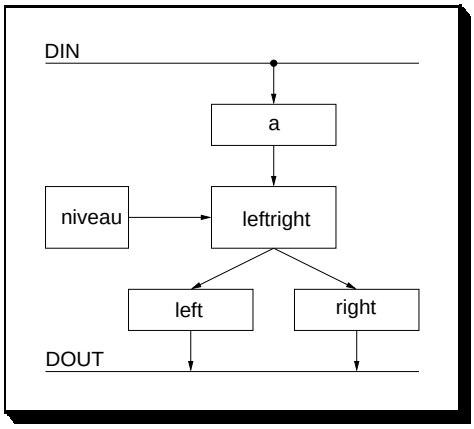


Abbildung 6.1: Berechnung von Left und Right

- Eine sehr interessante Aufgabe dürfte es sein, die einfachen Boundary–Scan Strukturen des Octchip zu erweitern. Dazu gehört die Implementierung weiterer Boundary–Scan Instruktionen wie INTEST oder die Realisierung eines internen Selbsttests und dessen Einbindung in die Boundary–Scan–Logik (RUNBIST). Auch der Entwurf von geeigneten Testeingaben für einen Boundary–Scan Test fällt in diesen Bereich.
- Es wäre zu untersuchen, welchen Nutzen die Aufhebung der Beschränkung von Modifikationen auf das Modell $q = 2$ hätte oder welche Vorteile die Erweiterung des Chips auf die Speicherung von vier Modellen haben würde.

- Ferner könnte man überlegen, ob durch Einführung von Pipeling–Strukturen eine weitere Parallelisierung erreicht werden kann. Denkbar wäre eine weitgehende Entkopplung der Navigation– und Bitblockeinheiten, so daß Bitblockoperationen und Navigationsoperationen parallel ablaufen können. Fest steht allerdings, daß solche Erweiterungen sich nur mit sehr viel zusätzlicher Hardware realisieren lassen und der Nutzen dieses Aufwandes eher fragwürdig ist.

Anhang A

Octchip Referenzen

A.1 Opcodes des Octchip

Folgende Operationen sind vom Octchip durchführbar:

Navigation:			
00	000000	reset	Zurücksetzen des Chips
01	000001	root (q)	Navigation zum Wurzelknoten
02	000010	down (q, p)	Navigation zum p. Sohn
03	000011	next (q)	Navigation zum Bruderknoten
04	000100	across (q, p)	Navigation zum Bruder p
05	000101	up (q)	Navigation zum Vaterknoten
06	000110	attach (q)	Erzeugen neuer Knoten
07	000111	set_full (q)	Aktuellen Knoten auf „voll“ setzen
08	001000	set_empty (q)	Aktuellen Knoten auf „leer“ setzen
09	001001	move_model2 (q)	Modellwechsel (Modell 2 umbenennen)
0A	001001	dump	Test: Inhalte der Register ausgeben
Bitblockoperationen:			
0D	001101	bbh $\rightarrow$ bb2 x	Gekoppeltes Schieben
0E	001110	bbh $\rightarrow$ bb2 y	vom Hilfsbitblock nach
0F	001111	bbh $\rightarrow$ bb2 z	Bitblock 2
10	010000	bbh $\diamond$ bb2 $\rightarrow$ bb2	Bitblockverknüpfung
11	010001	bb0 $\diamond$ bb1 $\rightarrow$ bb2	Bitblockverknüpfung
12	010010	bb0 $\diamond$ bb1 $\rightarrow$ bbh	Bitblockverknüpfung
13	010011	t00	Bitblock: Spiegeln am Ursprung
14	010100	t01	Bitblock: Spiegeln an x/y–Ebene
15	010101	t02	Bitblock: Spiegeln an x/z–Ebene
16	010110	t03	Bitblock: Spiegeln an y/z–Ebene
17	010111	t04	Bitblock: Klappen um $\pi/2$ um x–Achse
18	011000	t05	Bitblock: Klappen um π um x–Achse
19	011001	t06	Bitblock: Klappen um $-\pi/2$ um x–Achse
1A	011010	t07	Bitblock: Klappen um $\pi/2$ um y–Achse
1B	011011	t08	Bitblock: Klappen um π um y–Achse
1C	011100	t09	Bitblock: Klappen um $-\pi/2$ um y–Achse
1D	011101	t10	Bitblock: Klappen um $\pi/2$ um z–Achse
1E	011110	t11	Bitblock: Klappen um π um z–Achse
1F	011111	t12	Bitblock: Klappen um $-\pi/2$ um z–Achse

A.2 Register des Octchip

Die folgende Tabelle enthält eine Übersicht der im Octchip implementierten Register. Aufgelistet sind die externe Adresse des Registers, seine in der Dokumentation und in der Implementierung verwendete Bezeichnung, seine Breite in Bits, sein Inhalt sowie mögliche Zugriffsarten. R steht dabei für lesenden Zugriff, W für schreibenden Zugriff; runde Klammern deuten an, daß eine Zugriffsart nur zu Testzwecken erfolgen sollte.

Register des Octchip				
[-10pt] Adresse	Bez.	Breite	Inhalt	Zugriff
00	IREG	16	Instruktionsregister	W (R)
10	dto.		(15:10) Opcode, (9:8) q, (7:0) Parameter	
20	dto.			
01	AB0	8	Modell Anfangsadresse Baum 0	R, W
11	AB1	8	Modell Anfangsadresse Baum 1	R, W
21	AB2	8	Modell Anfangsadresse Baum 2	R, W
02	IB0	8	Basisadr der indirekten Adressen Baum 0	R, W
12	IB1	8	Basisadr der indirekten Adressen Baum 1	R, W
22	IB2	8	Basisadr der indirekten Adressen Baum 2	R, W
03	MIDx0	16	Mittelpunkt X-Achse Baum 0	R
04	MIDy0	16	Mittelpunkt Y-Achse Baum 0	R
05	MIDz0	16	Mittelpunkt Z-Achse Baum 0	R
13	MIDx1	16	Mittelpunkt X-Achse Baum 1	R
14	MIDy1	16	Mittelpunkt Y-Achse Baum 1	R
15	MIDz1	16	Mittelpunkt Z-Achse Baum 1	R
23	MIDx2	16	Mittelpunkt X-Achse Baum 2	R
24	MIDy2	16	Mittelpunkt Y-Achse Baum 2	R
25	MIDz2	16	Mittelpunkt Z-Achse Baum 2	R
06	INDx0	16	Oktaler Index X-Achse Baum 0	R
07	INDy0	16	Oktaler Index Y-Achse Baum 0	R
08	INDz0	16	Oktaler Index Z-Achse Baum 0	R
16	INDx1	16	Oktaler Index X-Achse Baum 1	R
17	INDy1	16	Oktaler Index Y-Achse Baum 1	R
18	INDz1	16	Oktaler Index Z-Achse Baum 1	R
26	INDx2	16	Oktaler Index X-Achse Baum 2	R
27	INDy2	16	Oktaler Index Y-Achse Baum 2	R
28	INDz2	16	Oktaler Index Z-Achse Baum 2	R
09	C0	16	Knoteninhalt (Full, Empty oder Split) Baum 0	R
19	C1	16	Knoteninhalt (Full, Empty oder Split) Baum 1	R
29	C2	16	Knoteninhalt (Full, Empty oder Split) Baum 2	R
0A	PR0	3	Aktuelle Position Baum 0	R
1A	PR1	3	Aktuelle Position Baum 1	R
2A	PR2	3	Aktuelle Position Baum 2	R
–	PS0	9*3	Stapel für PR Baum 0	-
–	PS1	9*3	Stapel für PR Baum 1	-
–	PS2	9*3	Stapel für PR Baum 2	-
0B	NR0	4	Aktuelles Niveau Baum 0	R
1B	NR1	4	Aktuelles Niveau Baum 1	R
2B	NR2	4	Aktuelles Niveau Baum 2	R

Register des Octchip				
[-10pt] Adresse	Bez.	Breite	Inhalt	Zugriff
0C	AP2l	8	erste freie Speicherzelle (q=2) HI	R, W
1C	dto.			
2C	dto.			
0D	AP2r	16	erste freie Speicherzelle (q=2) LO	R, W
1D	dto.			
2D	dto.			
0E	IP2	8	erste freie ind. Adresse (q=2)	R, W
1E	dto.			
2E	dto.			
40	ARL0	8	Relative Anfangsadr Zeigerblock HI Baum 0	(R, W)
50	ARL1	8	Relative Anfangsadr Zeigerblock HI Baum 1	(R, W)
60	ARL2	8	Relative Anfangsadr Zeigerblock HI Baum 2	(R, W)
41	ARR0	14	Relative Anfangsadr Zeigerblock LO Baum 0	(R, W)
51	ARR1	14	Relative Anfangsadr Zeigerblock LO Baum 1	(R, W)
61	ARR2	14	Relative Anfangsadr Zeigerblock LO Baum 2	(R, W)
–	AS0	7*22	Stapel für AR Baum 0	-
–	AS1	7*22	Stapel für AR Baum 1	-
–	AS2	7*22	Stapel für AR Baum 2	-
42	C0	16	Inhalt aktueller Knoten Baum 0	(R, W)
52	C1	16	Inhalt aktueller Knoten Baum 1	(R, W)
62	C2	16	Inhalt aktueller Knoten Baum 2	(R, W)
44-47	BB0	4*16	Bitblock 0	(R, W)
54-57	BB1	4*16	Bitblock 1	(R, W)
64-67	BB2	4*16	Bitblock 2	(R, W)
68-6B	BBH	4*16	Hilfsbitblock	(R, W)
6C	AXL	8	Temporäre Adresse LO	(R, W)
6D	AXR	16	Temporäre Adresse HI	(R, W)

A.3 Mnemoncis des Mikrorom–Compilers

Die zusammengesetzten Befehle des μROM–Compilers sind in EBNF–ähnlicher Schreibweise notiert: *schräg gedruckte* Bestandteile eines Befehles sind Nichtterminale und bei Verwendung des Befehles zu konkretisieren. Mögliche Werte dafür ergeben sich entweder implizit (z.B. *zeilennr.*) oder durch eine mit | getrennte Auflistung der Alternativen (z.B. *opcode* = `opc1` | `opc2` | `opc3`). Zur Semantik der zusammengesetzten Befehle sei auf die entsprechenden Erläuterungen der Hardware verwiesen.

Entscheidungen im Steuerwerk:
IF_NR<1
IF_NR<=1
IF_NR=1
IF_NR>1
IF_NR=2
IF_NR>2
IF_NR<9
IF_NR>9
IF_NR=2.9
IF_C15=0
IF_C15=1
IF_BCMOD
IF_AXL=0
IF_HOLD
IF_BUSWAIT
IF_MBUSY
ENDIF_NR<1
ENDIF_NR<=1
ENDIF_AXL=0

DIN → Register:
DI_TO_B
DI_TO_B*n*
DI_TO_PR
DI_TO_C
DI_TO_ARL
DI_TO_ARR
DI_TO_AXL
DI_TO_IR
DI_TO_AB
DI_TO_IB
DI_TO_AXR
DI_TO_AP2L
DI_TO_AP2R
DI_TO_IP2

Register → DOUT:
IP2_TO_DO
AP2L_TO_DO
AP2R_TO_DO
PR_TO_DO
B_TO_DO
C_TO_DO
C15_TO_DO
AXR_TO_DO
AXL_TO_DO
AX4_TO_DO
B_TO_DO
N_TO_DO
FE_TO_DO

Register → Addierwerk:
AB_TO_A
AP2L_TO_A
AP2R_TO_A
IB2_TO_A
IP2_TO_A
AR_P_TO_A
AR_M_TO_A
PR_TO_A
AXR_TO_A
AXL_TO_A

Konstante → Addierwerk:
I_TO_A
I_SET_2
I_SET_4
0_TO_AD11
0_TO_AD12
0_TO_AD21
0_TO_AD22

ADRin → Register:
A_TO_AP2
A_TO_IP2
A_TO_AX
A_TO_AR

DIN beschalten:
IREG_TO_DI
K_TO_DI
KONSTANTE

Speicherzugiff initiieren:
MEM_WRITE
MEM_READ

Bitblockeinheiten:
BB0_shift
BB0_roll
BB0_bF
BB1_shift
BB1_roll
BB1_bF
BB2_shift
BB2_roll
BB2_bF
BB2_SF
BB2_SE
BBH_shift
BBH_shift0
BBH_bF
IREG_TO_BBA

Bus → Bus:
DO4_TO_A
DO2_TO_A
DI_TO_DO

μPC steuern:
UPC:=IREG
UPC++
UPC:=*zeilennr.*

Statusregister:

SET_BCmod	RESET_BCmod
SET_BUSY	RESET_BUSY
SET_BUSRQ	RESET_BUSRQ

Spezialregister: NR:=1

NR_INC	NR_DEC
PR_INC	PR:=P
PS_PUSH	PS_POP
AS_PUSH	AS_POP

Index/Pos.–ausgeben:
INDEX_*auszugeben*
auszugeben ::= P | X | Y | Z

Bitblock–Achsen:
AXIS_*bbnr. achse*
bbnr. ::= 0 | 1 | 2 | H
achse ::= X | Y | Z

Bitblock–Opcode: (vgl. 4.4.5.2)
ALU_*src op dest rotate*
src ::= 0_ | H_ | 0K | HK | __
op ::= ^ | v | A | B
dest ::= 1_ | 2_ | 1K | 2K | __
rotate ::= R | _

x**K** bedeutet: Datum von Bitblock *x* komplementiert verknüpfen; die Opertoren sind Und–Verknüpfung, Oder–Verknüpfung, Eingang A durchschalten und Eingang B durchschalten. Wird nur ein Eingang durchgeschaltet, so wird der Selektor des anderen Eingangs auf „dont't care“ geschaltet.

Middle–Opcode: (vgl. 4.4.4.2)
MIDDLE_*opcode*
opcode ::= R(eset) | N(ext) | A(cross) |
D(own) | U(p) |
X | Y | Z (auslesen)

Zeilennummer festlegen:
line:=*zeilennr.*

A.4 Mikrorom

```
- Mikrorom fuer Octchip
-
-
- Navigation                                                                                                                    18o796 TK
-
- RESET
line:=0 set_busy reset_busrq reset_bcmod k_to_di konstante di_to_ir                      upc++
line:=1 k_to_di di_to_ab di_to_ib di_to_ap2l di_to_ap2r di_to_ip2                upc++
- ROOT
line:=2 if_buswait if_bcmod  set_busrq upc:=2
line:=3 if_nr>1    if_bcmod  c_to_do c15_to_do  mem_write  ab_to_a ar_p_to_a pr_to_a                                    upc:=8 reset_busrq
line:=4            if_bcmod  b_to_do bb2_roll   mem_write  ab_to_a ar_p_to_a         0_to_ad11   a_to_ax  axis_2x      upc++
line:=5            if_bcmod  b_to_do bb2_roll   mem_write  axr_to_a axl_to_a i_to_a   0_to_ad22   a_to_ax  axis_2x      upc++
line:=6            if_bcmod  b_to_do bb2_roll   mem_write  axr_to_a axl_to_a i_to_a   0_to_ad22   a_to_ax  axis_2x      upc++
line:=7            if_bcmod  b_to_do bb2_roll   mem_write  axr_to_a axl_to_a i_to_a   0_to_ad22   a_to_ax  axis_2x      upc++
line:=8 nr:=10 k_to_di di_to_c di_to_pr 0_to_ad11 0_to_ad21 0_to_ad22 0_to_ad12 a_to_ar indexreset middle_r reset_bcmod upc:=128
-
- DOWN
line:=20  if_buswait          set_busrq upc:=20
line:=21                      ps_push as_push nr_dec ireg_to_di pr:=p  middle_d   reset_busrq                                    upc++    endif_nr<=1
line:=22  if_nr<9   if_c15=0  c_to_do c15_to_do do4_to_a ar_p_to_a 0_to_ad21 a_to_ar                                           upc:=25
line:=23  if_nr<9   if_c15=1  c_to_do do2_to_a ab_to_a ib_to_a 0_to_ad12   mem_read    di_to_arl a_to_ax   di_to_arl              upc++
line:=24  if_nr<9   if_c15=1  axl_to_a axr_to_a i_to_a         0_to_ad22   mem_read    di_to_arr                                  upc++
line:=25  if_nr>1             ab_to_a  ar_p_to_a pr_to_a                    mem_read    di_to_c                                    upc:=128
line:=26                      ab_to_a  ar_p_to_a              0_to_ad11   mem_read    di_to_b   a_to_ax   axis_0x axis_1x axis_2x  upc++
line:=27                      axr_to_a axl_to_a i_to_a        0_to_ad22   mem_read    di_to_b   a_to_ax   axis_0x axis_1x axis_2x  upc++
line:=28                      axr_to_a axl_to_a i_to_a        0_to_ad22   mem_read    di_to_b   a_to_ax   axis_0x axis_1x axis_2x  upc++
line:=29                      axr_to_a axl_to_a i_to_a        0_to_ad22   mem_read    di_to_b   a_to_ax   axis_0x axis_1x axis_2x  upc:=128
-
- NEXT
line:=30 if_buswait           set_busrq upc:=30
line:=31 if_nr>1 if_bcmod     c_to_do c15_to_do mem_write ab_to_a ar_p_to_a pr_to_a    reset_bcmod     reset_busrq  upc++
line:=32                      pr_inc ireg_to_di middle_n                                                                 upc++     endif_nr<=1
line:=33                      ab_to_a ar_p_to_a pr_to_a                    mem_read    di_to_c                       upc:=128
-
- ACROSS
line:=35 if_buswait           set_busrq upc:=35
line:=36 if_nr>1 if_bcmod     c_to_do c15_to_do mem_write ab_to_a ar_p_to_a pr_to_a    reset_bcmod     reset_busrq  upc++
line:=37                      ireg_to_di middle_a pr:=p                                                                  upc++     endif_nr<=1
line:=38                      ab_to_a ar_p_to_a pr_to_a                    mem_read    di_to_c                       upc:=128
-
- UP
line:=40 if_buswait           set_busrq upc:=40
line:=41 if_nr>1 if_bcmod     c_to_do c15_to_do  mem_write  ab_to_a  ar_p_to_a pr_to_a    reset_busrq                reset_bcmod  upc:=46
line:=42 if_nr=1 if_bcmod     b_to_do bb2_roll   mem_write  ab_to_a  ar_p_to_a         0_to_ad11  a_to_ax   axis_2x                upc++
line:=43 if_nr=1 if_bcmod     b_to_do bb2_roll   mem_write  axr_to_a axl_to_a i_to_a   0_to_ad22  a_to_ax   axis_2x                upc++
line:=44 if_nr=1 if_bcmod     b_to_do bb2_roll   mem_write  axr_to_a axl_to_a i_to_a   0_to_ad22  a_to_ax   axis_2x                upc++
line:=45 if_nr=1 if_bcmod     b_to_do bb2_roll   mem_write  axr_to_a axl_to_a i_to_a   0_to_ad22  a_to_ax   axis_2x   reset_bcmod  upc++
line:=46                      ps_pop as_pop nr_inc middle_u                                                                                upc++     endif_nr<=1
line:=47                      ab_to_a ar_p_to_a pr_to_a                    mem_read   di_to_c                                      upc:=128
-
- ATTACH
line:=50  if_buswait          set_busrq                                                                                    q:=2  upc:=50
line:=51  if_nr=2.9           ap2l_to_a ap2r_to_a ar_m_to_a a_to_ax              reset_busrq                               q:=2  upc:=53   endif_nr<=1
line:=52                                                                                                                   q:=2  upc:=58   endif_nr<=1
line:=53  if_axl=0            ax4_to_do                 mem_write ab_to_a ar_p_to_a pr_to_a      do_to_di di_to_c         q:=2  upc:=58
line:=54                      ip2_to_do                 mem_write ab_to_a ar_p_to_a pr_to_a      do_to_di di_to_c         q:=2  upc++
line:=55                      ap2l_to_do                mem_write ab_to_a ib_to_a ip2_to_a 0_to_ad12 a_to_ax              q:=2  upc++
line:=56                      ap2r_to_do                mem_write axr_to_a axl_to_a i_to_a 0_to_ad22                      q:=2  upc++
line:=57                      ip2_to_a i_to_a i_set_2 a_to_ip2                                                             q:=2  upc++
line:=58                      ap2l_to_a ap2r_to_a ab_to_a2 0_to_ad12  a_to_ax                                              q:=2  upc++
line:=59                      k_to_di di_to_do          mem_write   axl_to_a axr_to_a 0_to_ad12 0_to_ad22 a_to_ax         q:=2  upc++
line:=60                      k_to_di di_to_do          mem_write   axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax         q:=2  upc++
line:=61                      k_to_di di_to_do          mem_write   axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax         q:=2  upc++
line:=62                      k_to_di di_to_do          mem_write   axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax         q:=2  upc++
line:=63                      ap2l_to_a ap2r_to_a i_to_a i_set_4  0_to_ad22   a_to_ap2                                     q:=2  upc++
line:=64  if_nr=2                                                                                                            q:=2  upc:=128
line:=65                      k_to_di di_to_do          mem_write   axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax         q:=2  upc++
line:=66                      k_to_di di_to_do          mem_write   axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax         q:=2  upc++
line:=67                      k_to_di di_to_do          mem_write   axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax         q:=2  upc++
line:=68                      k_to_di di_to_do          mem_write   axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax         q:=2  upc++
```

```
line:=69                      ap2l_to_a ap2r_to_a i_to_a i_set_4  0_to_ad22   a_to_ap2                                q:=2  upc:=128
-
- SET_FULL
line:=70 if_nr=2.9            k_to_di konstante di_to_c  set_bcmod    q:=2   upc:=128
line:=71 if_nr<=1             bb2_sf                     set_bcmod    q:=2   upc:=128
line:=72 upc:=128
-
- SET_EMPTY
line:=75 if_nr=2.9            k_to_di           di_to_c  set_bcmod    q:=2   upc:=128
line:=76 if_nr<=1             bb2_se                     set_bcmod    q:=2   upc:=128
line:=77 upc:=128
-
- MOVE Modell2
-   Basisadressen umkopieren, AB(q) := AB(2), IB(q) := IB(2)
line:=80   ab_to_a   0_to_ad22  0_to_ad11 0_to_ad12              a_to_ax       q:=2   upc++
line:=81   axl_to_do                                  do_to_di   di_to_ab              upc++
line:=82   ib_to_a   0_to_ad21  0_to_ad11 0_to_ad12              a_to_ax       q:=2   upc++
line:=83   axl_to_do                                  do_to_di   di_to_ib              upc++
-   AB2 := AB2 + AP2L + 1, Umweg ueber AXR, da Inkrementieren in den oberen 8 Bit nicht moeglich
line:=84   ap2l_to_a  ab_to_a2  0_to_ad11 0_to_ad12              a_to_ax       q:=2   upc++
line:=85   axl_to_do                                  do_to_di   di_to_axr     q:=2   upc++
line:=86   axr_to_a  i_to_a     0_to_ad21  0_to_ad22             a_to_ax       q:=2   upc++
line:=87   axr_to_do                                  do_to_di   di_to_ab      q:=2   upc++
-   IB2 := IB2 + 1, Umweg ueber AXR
line:=88   ib_to_a   0_to_ad21  0_to_ad11 0_to_ad12              a_to_ax       q:=2   upc++
line:=89   axl_to_do                                  do_to_di   di_to_axr     q:=2   upc++
line:=90   axr_to_a  i_to_a     0_to_ad21  0_to_ad22             a_to_ax       q:=2   upc++
line:=91   axr_to_do                                  do_to_di   di_to_ib      q:=2   upc++
-   AP2L := 0, AP2R := 0, IP2 := 0
line:=92   k_to_di  di_to_ap2l di_to_ap2r  di_to_ip2                            q:=2   upc:=128

- Testroutine: DUMP
line:=100  if_buswait  set_busrq upc:=100
line:=101  ap2l_to_a ap2r_to_a ab_to_a2 0_to_ad12  a_to_ax                                          upc++
line:=102  k_to_di konstante di_to_do   mem_write axl_to_a axr_to_a 0_to_ad12 0_to_ad22 a_to_ax   upc++ reset_busrq
line:=103  k_to_di konstante di_to_do   mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=104  middle_x                     mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=105  middle_y                     mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=106  middle_z                     mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=107  index_x  pr_to_do            mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=108  index_y  pr_to_do            mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=109  index_z  pr_to_do            mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=110  pr_to_do                     mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=111  n_to_do                      mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=112  ap2l_to_do                   mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=113  ap2r_to_do                   mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=114  ip2_to_do                    mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=115  c_to_do c15_to_do            mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=116  axl_to_do                    mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=117  axr_to_do                    mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=118  k_to_di konstante di_to_do   mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=119  k_to_di konstante di_to_do   mem_write axl_to_a axr_to_a i_to_a    0_to_ad22 a_to_ax   upc++
line:=120  ap2l_to_a ap2r_to_a i_to_a i_set_4 i_set_2 0_to_ad22 a_to_ap2                           upc++
line:=121  ap2l_to_a ap2r_to_a i_to_a i_set_4 i_set_2 0_to_ad22 a_to_ap2                           upc++
line:=122  ap2l_to_a ap2r_to_a i_to_a i_set_4 i_set_2 0_to_ad22 a_to_ap2                           upc:=128
-
-
- FETCH
line:=128         reset_busy reset_busrq upc++
line:=129 if_hold                        upc:=129
line:=130         set_busy               upc:=ireg
-
-
---- Bitblockteil  0197 TE
-
-- BB-Transformationen, ab line 188 bis 202
-- f"ur die Transformationen (immer in Verbindung mit BB0 BB1 -> BB2)
-- mu"s der Rest des Instruktionsregisters direkt an die Bitblock--ALU
-- gelegt werden, m"ogliches "Aussehen:
-- ..0000010 bildet die Schnittmenge (subtract) zwischen BB0 und BB1, Ergebnis in BB2
-- ..0000011 bildet die Vereinigung (unite) zwischen .".
```

```
-- ..0001010 bildet BB0 ohne BB1 (isect)
-- ..0010010 bildet BB1 ohne BB0 (isect)
--BB1 BBH -> BB2
--line:=197 IF_NR<=1 AXIS_1X BB1_roll BB1_bF AXIS_HX BBH_roll BBH_bF AXIS_2X BB2_shift BB2_bF IREG_TO_BBA set_bcmod  upc++
--line:=198 IF_NR<=1 AXIS_1X BB1_roll BB1_bF AXIS_HX BBH_roll BBH_bF AXIS_2X BB2_shift BB2_bF IREG_TO_BBA            upc++
--line:=199 IF_NR<=1 AXIS_1X BB1_roll BB1_bF AXIS_HX BBH_roll BBH_bF AXIS_2X BB2_shift BB2_bF IREG_TO_BBA            upc++
--line:=200 IF_NR<=1 AXIS_1X BB1_roll BB1_bF AXIS_HX BBH_roll BBH_bF AXIS_2X BB2_shift BB2_bF IREG_TO_BBA            upc:=128
--BB0 BB2 -> BBH
--line:=201 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_roll BB2_bF AXIS_HX BBH_shift BBH_bF IREG_TO_BBA set_bcmod  upc++
--line:=202 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_roll BB2_bF AXIS_HX BBH_shift BBH_bF IREG_TO_BBA            upc++
--line:=203 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_roll BB2_bF AXIS_HX BBH_shift BBH_bF IREG_TO_BBA            upc++
--line:=204 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_roll BB2_bF AXIS_HX BBH_shift BBH_bF IREG_TO_BBA            upc:=128


--BBH -> BB2 x-Richtung --10111100
line:=188 IF_NR<=1 AXIS_HX BBH_shift0 BBH_bF AXIS_2X BB2_shift BB2_bF ALU_H_A___ set_bcmod  upc:=128
--BBH -> BB2 y-Richtung --10111101
line:=189 IF_NR<=1 AXIS_HY BBH_shift0 BBH_bF AXIS_2Y BB2_shift BB2_bF ALU_H_A___ set_bcmod  upc:=128
--BBH -> BB2 z-Richtung --10111110
line:=190 IF_NR<=1 AXIS_HZ BBH_shift0 BBH_bF AXIS_2Z BB2_shift BB2_bF ALU_H_A___ set_bcmod  upc:=128
--BBH or BB2 -> BB2 --10111111
line:=191 IF_NR<=1 AXIS_HX BBH_shift0 BBH_bF AXIS_2X BB2_shift BB2_bF ALU_H_v2__ set_bcmod  upc++
line:=192 IF_NR<=1 AXIS_HX BBH_shift0 BBH_bF AXIS_2X BB2_shift BB2_bF ALU_H_v2__            upc++
line:=193 IF_NR<=1 AXIS_HX BBH_shift0 BBH_bF AXIS_2X BB2_shift BB2_bF ALU_H_v2__            upc++
line:=194 IF_NR<=1 AXIS_HX BBH_shift0 BBH_bF AXIS_2X BB2_shift BB2_bF ALU_H_v2__            upc:=128
--BB0 BB1 -> BB2 --11000011
line:=195 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF AXIS_2X BB2_shift BB2_bF IREG_TO_BBA set_bcmod  upc++
line:=196 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF AXIS_2X BB2_shift BB2_bF IREG_TO_BBA            upc++
line:=197 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF AXIS_2X BB2_shift BB2_bF IREG_TO_BBA            upc++
line:=198 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF AXIS_2X BB2_shift BB2_bF IREG_TO_BBA            upc:=128
--BB0 BB1 -> BBH --11000111
line:=199          AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF AXIS_HX BBH_shift BBH_bF IREG_TO_BBA            upc++
line:=200          AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF AXIS_HX BBH_shift BBH_bF IREG_TO_BBA            upc++
line:=201          AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF AXIS_HX BBH_shift BBH_bF IREG_TO_BBA            upc++
line:=202          AXIS_0X BB0_roll BB0_bF AXIS_1X BB1_roll BB1_bF AXIS_HX BBH_shift BBH_bF IREG_TO_BBA            upc:=128
--Tranformationen ab line 203 bis 254
--T00 --11001011
line:=203 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A__R set_bcmod  upc++
line:=204 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A__R            upc++
line:=205 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A__R            upc++
line:=206 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A__R            upc:=128
--T01 --11001111
line:=207 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___ set_bcmod  upc++
line:=208 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___            upc++
line:=209 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___            upc++
line:=210 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___            upc:=128
--T02 --11010011
line:=211 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___ set_bcmod  upc++
line:=212 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___            upc++
line:=213 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___            upc++
line:=214 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___            upc:=128
--T03 --11010111
line:=215 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___ set_bcmod  upc++
line:=216 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___            upc++
line:=217 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___            upc++
line:=218 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___            upc:=128
--T04 --11011011
line:=219 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___ set_bcmod  upc++
line:=220 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___            upc++
line:=221 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___            upc++
line:=222 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___            upc:=128
--T05 --11011111
line:=223 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Y BB2_shift BB2_bF ALU_0_A__R set_bcmod  upc++
line:=224 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Y BB2_shift BB2_bF ALU_0_A__R            upc++
line:=225 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Y BB2_shift BB2_bF ALU_0_A__R            upc++
line:=226 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Y BB2_shift BB2_bF ALU_0_A__R            upc:=128
--T06 --11100011
line:=227 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___ set_bcmod  upc++
line:=228 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___            upc++
line:=229 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___            upc++
line:=230 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___            upc:=128
```

```
--T07 --11100111
line:=231 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___ set_bcmod  upc++
line:=232 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___            upc++
line:=233 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___            upc++
line:=234 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Z BB2_shift ALU_0_A___            upc:=128
--T08 --11101011
line:=235 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Y BB2_shift BB2_bF ALU_0_A__R set_bcmod  upc++
line:=236 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Y BB2_shift BB2_bF ALU_0_A__R            upc++
line:=237 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Y BB2_shift BB2_bF ALU_0_A__R            upc++
line:=238 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2Y BB2_shift BB2_bF ALU_0_A__R            upc:=128
--T09 --11101111
line:=239 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___ set_bcmod  upc++
line:=240 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___            upc++
line:=241 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___            upc++
line:=242 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___            upc:=128
--T10 --11110011
line:=243 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___ set_bcmod  upc++
line:=244 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___            upc++
line:=245 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___            upc++
line:=246 IF_NR<=1 AXIS_0Y BB0_roll BB0_bF AXIS_2X BB2_shift ALU_0_A___            upc:=128
--T11 --11110111
line:=247 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Z BB2_shift BB2_bF ALU_0_A__R set_bcmod  upc++
line:=248 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Z BB2_shift BB2_bF ALU_0_A__R            upc++
line:=249 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Z BB2_shift BB2_bF ALU_0_A__R            upc++
line:=250 IF_NR<=1 AXIS_0Z BB0_roll BB0_bF AXIS_2Z BB2_shift BB2_bF ALU_0_A__R            upc:=128
--T12 --11111011
line:=251 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___ set_bcmod  upc++
line:=252 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___            upc++
line:=253 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___            upc++
line:=254 IF_NR<=1 AXIS_0X BB0_roll BB0_bF AXIS_2Y BB2_shift ALU_0_A___            upc:=128
-- und dann war da noch
line:=255  upc:=128
-ende
```

Anhang B

Autoren

Torsten Emmerich

2.1.4 (Grundlagen: Bitblöcke)
2.2.3 (Grundlagen: Operationen auf Bitblöcken)

3.3.6 (Low–Level–Operationen: Operationen auf Bitblöcken)
3.4.2 (High–Level–Operationen: Spiegeln und Klappen)

4.4.4.1.3 (Rechenwerk: Positionsregister)
4.4.4.3 (Rechenwerk: Niveauregister)
4.4.4.4.1 (Rechenwerk: Full–Empty)
4.4.5 (Bitblocklogik) ausgenommen 4.4.5.2 (Bitblockalu)
4.4.5.2.1 (Rotieren von Bitblockdaten)
4.6.3 (Mikroprogramme, Bitblockoperationen)

5.1 (Hardwareentwurfsprozeß)

Thomas Kamphans

2 (Grundlagen), ausgenommen 2.1.4 (Bitblöcke) und 2.2.3 (Operationen auf Bitblöcken)

3.1 (Systemarchitektur)
3.2 (Datenstrukturen)
3.3 (Low–Level–Operationen), ausgenommen 3.3.6 (Operationen auf Bitblöcken)
3.4 (High–Level–Operationen), ausgenommen 3.4.2 (Spiegeln und Klappen)

4.1 (Spezifikation des Octchip)
4.2 (Grobentwurf)
4.3 (Grundlegende Komponenten)
4.4 (Rechenwerk) ausgenommen 4.4.4.1.3 (Positionsregister), 4.4.4.3 (Niveauregister), 4.4.4.4.1 (Full–Empty) und 4.4.5 (Bitblocklogik)
4.4.5.2 (Bitblockalu), ausgenommen 4.4.5.2.1 (Rotieren von Bitblockdaten)
4.5 (Steuerwerk)
4.6 (Mikroprogramme) ausgenommen 4.6.3 (Bitblockoperationen)

5.2 (Testsynthese)
5.3 (Simulationskopplung)

Gemeinsam erstellte Abschnitte

1 (Einleitung)
5.4 (Resultate)
6 (Zusammenfassung und Ausblick)

Literaturverzeichnis

[1] Peter Dietz: *Ein Computermodell auf Basis kubischer Volumina für die Approximation beliebig geformter körperlicher Objekte*, Verlag Shaker 1994

Literatur zum Thema Octrees:

[2] Hanan Samet: *The design and analysis of spatial data structures.*, Addison–Wesley, 1990

[3] Hanan Samet: *Applications of spatial data structures. computer graphics, image processing, and GIS.*, Addison–Wesley, 1990

[4] Andrew Marriott: *Lectures on 3–D Computer Graphics*, http://www.cs.curtin.edu.au/units/cg351-551/notes/.index.html

[5] K. Yamaguchi u.a.: *Computer–Integrated Manufacturing of Surfaces using Octree Encoding*, IEEE Computer–Graphics and Applications, Januar 1984

[6] Louis J. Doctor, John G. Torborg: *Display Techniques for Octree–Encoded Objects*, IEEE Computer–Graphics and Applications, Juli 1981

[7] Jean-Louis Amans, Pierre Darrier: *Processing and Display of Medical Three Dimensional Arrays of Numerical Data Using Octree Encoding*, Medical Image Processing, SPIE Vol. 593, Dezember 1985

[8] Shawn Bohn, Erin Thornton *Environment Reconstruction for Robot Navigation*, Proceedings of the SPIE, Vol. 2217, 1994

Literatur zum Thema Computergraphik:

[9] Heinrich Müller: *Skriptum zur Vorlesung Graphische Systeme*, Universität Dortmund, 1992

[10] James Foley et al.: *Computer Graphics — Principles and Practice*, Addison–Wesley, 1996

[11] Wolf Dieter Fellner: *Computergrafik*, BI Wissenschaftsverlag, 1992

[12] Ekkehard Beier: *Objektorientierte 3D-Grafik*, Thompson Publishing, 1994

[13] Craig E. Kolb: *Rayshade User's Guide*, 1992 http://www-graphics.stanford.edu/~cek/rayshade/doc/guide/guide.html

[14] H. Gouraud: *Computer Display on Curved Surfaces*, IEEE Computer Graphics & Applications (6), 1971

[15] Tom Sperlich, Christian Bauer: *Künstliche Welten — Virtual Reality: schneller und echter*, c't Magazin, April 1996

[16] Anne Foerst: *Zimmervermittlung — Intelligente Mensch–Computer–Interaktion*, c't Magazin, Juni 1996

[17] Andreas Steinbruch: *Holodeck im Web — VRML 2.0: lebendiges 3D*, c't Magazin, Juli 1996

[18] *Intelligentes Zimmer*, 1996
http://www.ai.mit.edu/projects/hci/hci.html

[19] Mark C. Torrance: *Advances in Human–Computer Interaction: The Intelligent Room*, MIT, 1995

[20] Tom Sperlich, Gernot Schärmeli: *Höhlenbewohner — 3D–Umgebung der virtuellen Art*, c't Magazin, Januar 1997

[21] Andreas Stabinger: *Raytrace*, Data Becker, 1992

[22] Gregory J. Ward: *The RADIANCE Lighting Simulation and Rendering System*, 1994
http://radsite.lbl.gov/radiance/papers/sg94.1/octree.html

Literatur zu Hardwareentwurf und VHDL:

[23] Peter Marwedel: *Synthese und Simulation von VLSI-Systemen*, Hanser, 1993

[24] Peter Marwedel, Georg Baumann *Software-Werkzeuge für den Entwurf Digitaler Systeme*, 1991

[25] A. Maeder: *VHDL Kurzbeschreibung*, Universität Hamburg

[26] R. Lipsett, C. Schaefer, C. Ussery: *VHDL: Hardware Description and Design*, Kluver Academic Press, 1989

[27] *IEEE Standard 1076–1993: VHDL Language*, 1994

[28] P. Kurup, T. Abbasi: *Logic Synthesis Using Synopsys*, Kluver Academic Press, 1997

[29] Kenneth P. Parker: *The Boundary–Scan Handbook*, Kluver Academic Press, 1992

[30] Projektgruppe Enigma: *Zwischenbericht*, Universität Dortmund, 1994

[31] Projektgruppe Enigma: *Endbericht*, Universität Dortmund, 1994

[32] Ingo Wegener: *Skriptum zur Vorlesung Theorie des Logikentwurfes*, Universität Dortmund, 1993

[33] Ingo Wegener: *Effiziente Algorithmen für grundlegende Funktionen*, Teubner–Verlag, 1989

[34] Frank Hellwig: *Vom Sand zum Chip — ASIC–Entwicklung heute*, c't Magazin, Mai 1996

Literatur zum Thema Rechnerarchitektur und Hardware:

[35] Peter Marwedel: *Skriptum zur Vorlesung Rechnerarchitektur*, Universität Dortmund, 1993

[36] Helmut Bähring: *Mikrorechnersysteme*, Springer, 1991

[37] Wilhelm Lütze: *Skriptum zur Vorlesung Datentechnik*, Universität Dortmund, 1992

[38] Otto Limann, Horst Pelka: *Elektronik ohne Ballast*, Franzis–Verlag, 1991

[39] Hans-Peter Messmer: *PC–Hardwarebuch*, Addison–Wesley, 1995

[40] Adaptec Inc.: *Quick Reference to SCSI*, Adaptec EZ–SCSI 4.0 CD–ROM, 1995

Sonstiges:

[41] Brian Kernighan, Dennis Ritchie: *Programmieren in C*, Hanser, 1990

[42] Borland Inc. *Referenzhandbuch Borland C++ 3.0*, 1992

[43] *Einführung in das Graphiksystem MING*, aus: Übungen zur Vorlesung Graphische Systeme 1993

[44] Stefan Wohlfeil, Alexander Schölling: *Programmierung mit der XGraphic–Bibliothek*, Fernuniversität Hagen, 1996

[45] Synopsys Inc.: *Test Compiler and Test Compiler Plus Reference Manual*, 1992

[46] Synopsys Inc.: *VSS Reference Manual*, 1992

[47] Synopsys Inc.: *Design Compiler Family Reference Manual*, 1992

[48] Synnopsys Inc.: *Synthesis to Simulation Interfacing Application Note*, 1993

Index

across, *siehe* Navigation
Adreßberechnung, 76
Adreßdekoder, 68
Affine Abbildungen, **16**
ASIC, 144
attach, *siehe* Baumoperation

Baumoperation
  attach, **6**, 31, 107
  set_empty, **6**, 31, 109
  set_full, **6**, 31, 109
Bitblock, **9**
  Bitblockalu, 96
  C–Implementierung, 27
  Transformation, 40
  Zugriff, 38
Bitblockalu, 96
Boundary–Scan–Test, 122

Datenstruktur
  Arraymodell, 27
  Bitblock, 27
  Octree, 26–27
Designware, 118
down, *siehe* Navigation

FPGA, 144

Geometrische Größen, **16**
GTECH, 118

Hardware–Entwurfsprozes, 113
Hardwarebeschreibungssprache, 113
Hardwareentwurf
  Grobentwurf, 63

Index
  oktal, **6**, 28
  vektoriell, **6**, 28, 36
Instruktionsregister, 71

Knoten
  aktueller, **6**
  leer, **5**
  setzen, 36
  suchen, 36
  voll, **5**
Knoteninhalte, 86
Kubusmittelpunkt, **7**
  Hardwarerealisierung, 82
  sw, 34

Laufzeitmessung, 135
Logikoptimierung, 115

Magic–Number, 128, 133
Mengenoperationen, **14**
middle, *siehe* Kubusmittelpunkt
Mikroprogramm, *siehe* Steuerwerk
Mikroprogrammzähler, *siehe* Steuerwerk
Mikrorom, *siehe* Steuerwerk
Morphologische Transformationen, **14**

Navigation, **6**, 27
  across, **6**, 34, 107
  down, **6**, 33, 106
  global, 36
  next, **6**, 34, 107
  root, **6**, 32, 106
  up, **6**, 33, 107
Navigationslogik, 76
next, *siehe* Navigation
Niveau, **5**
Niveauregister, 84

Octchip
  Interface, 61
  Opcodes, 149
  Register, 150
  Spezifikation, 59
Octree, **5**, 26
Octree, Traversierung, 37
Octrees
  Abbildung auf den Speicher, 13, 27

Plazierung/Verdrahtung, 144
Position
  oktal, **6**, 27
  Register, 80
  vektoriell, **6**, 27
Positionsregister, 80

Rasterkorrektur, 35
Rechenwerk, 68
Reservieren von Speicher, 31
Reset, 106
root, *siehe* Navigation

Scan–Test, 120
Schnittebene, 50
set_empty, *siehe* Baumoperation
set_full, *siehe* Baumoperation
Simulationsumgebung, 126
Statusregister, 71
Steuerwerk, 98
    Mikroprogramm, 105
    Mikroprogrammzähler, 100
    Mikrorom, 99
Synopsys, 73, 115, 125, 143
Synthese, 138

Testbench, 126
Testmuster, 120
Testsynthese, 120
Transformationen, **14**
treealloc, 31

up, *siehe* Navigation

VHDL, 113, 144
Visualisierung, 50