Mitschrift

Informatik 2 / Systemarchitektur SS10

Universität des Saarlandes, FR 6.2 - Informatik
Institut für Rechnerarchitektur und Parallelrechner
Prof. Dr. W. J. Paul

Stand: 24. September 2010, Rev. 60

Inhaltsverzeichnis

Zusammenfassung

Die vorliegende Mitschrift fasst den Inhalt der Vorlesung *„Systemarchitektur“*, gehalten von Prof. Dr. Wolfgang J. Paul an der Universität des Saarlandes, zusammen. Sie befasst sich mit dem Aufbau und der Funktionsweise von Computersystemen, bestehend aus

- Prozessor
- Compiler
- Betriebsystem

und liefert formale *Spezifikation* sowie *Konstruktionsvorschrift* für die jeweiligen Komponenten und ihre Bestandteile. Durch mathematische *Korrektheitsbeweise* wird die Konsistenz von Spezifikation und Implementierung verifiziert.

Dieses Skript wurde vorlesungsbegleitend und auf der Grundlage der Vorlesungen vom SS07, SS08 und SS10 von Christoph Baumann verfasst. Fragen und Hinweise werden dankend über die Email-Adresse *baumann@wjpserver.cs.uni-sb.de* entgegengenommen.

Kapitel 0

Einleitung

In dieser Vorlesung, wie auch in anderen Vorlesungen und dem täglichen Leben verwenden wir ganz selbstverständlich natürliche Zahlen. Seit wir zählen können benutzen wir sie, nehmen sie als gegeben hin und hinterfragen nicht, woher sie eigentlich kommen. Im folgenden soll das Konzept der natürlichen Zahl untersucht werden. Dabei wird sich herausstellen, dass diese nicht nur eine ganz abenteuerliche Konstruktion sind, sondern auch auf praktisch gesehen falschen Annahmen beruhen.

0.1 Was ist eine Zahl?

Wie schwierig der Zahlenbegriff zu deuten ist zeigt sich schon an der einfachen Zahlendarstellung "10". So könnten

- gewöhnlich denkende Menschen 10 als "Zehn" interpretieren, was der Anzahl der menschlichen Zehen entspricht. Daher auch der Name.
- wir, sofern wir zweizehige Faultiere wären, demnach auch 10 = "Vier" annehmen.
- Informatiker, die der Binärdarstellung mächtig sind, 10 als "Zwei" deuten.

Mann sollte sich also auf eine Zahlendarstellung einigen. Wir wählen Unärzahlen, die primitivste Form der Zahlendarstellung. Zahlen werden hierbei zum Beispiel durch eine Folge von *Strichen* repräsentiert, z.B.:

|||| = "Vier"

Während es sich schon schwierig gestaltet festzulegen, was alles als ein Strich durchgeht, so stößt man bei der Definition der Folge auf ein unlösbares Problem. Eine Folge im gewohnten Stil zu definieren, setzt nämlich Zählen voraus und zum Zählen benötigt man Zahlen. Da schließt sich der Teufelskreis. Hieraus lässt sich schon erkennen, dass eine traditionelle Definition der natürlichen Zahlen "von unbekannt nach bekannt" nicht möglich ist. Stattdessen kann man sich auf Eigenschaften einigen, die alle natürlichen Zahlen besitzen sollen. Die Frage, was Zahlen nun genau sind, ist philosophischer Natur.

0.2 Peano-Axiome

Die Peano-Axoime (nach Giuseppe Peano, 1858-1932) stellen eine solche Liste aus Eigenschaften der natürlichen Zahlen dar. Die Axiome sind empirisch für eine beliebige endliche Anzahl von Beispielen nachprüfbar. Dennoch verkörpern sie keine Definition der natürlichen Zahlen. Nachfolgend werden die Peano-Axiome aufgelistet.

Definition 0.1 *(Peano-Axiome)*

1. $1 \in \mathbb{N}$
   1 *ist eine natürliche Zahl.*
2. $x \in \mathbb{N} \Rightarrow N(x) \in \mathbb{N}$
   Jede natürliche Zahl hat einen Nachfolger. Dabei ist $N(x)$ *die Nachfolger-Funktion für jede natürliche Zahl* x*.*
3. $\forall x \in \mathbb{N} : 1 \neq N(x)$
   1 *ist nicht Nachfolger einer Zahl.*
4. $\forall x, y \in \mathbb{N} : x \neq y \Rightarrow N(x) \neq N(y)$
   Voneinander verschiedene natürliche Zahlen haben voneinander verschiedene Nachfolger, d.h. insbesondere, dass jede natürliche Zahl höchstens Nachfolger einer *natürlichen Zahl ist.*
5. $(A \subset \mathbb{N} \wedge 1 \in A \wedge \forall x \in A : N(x) \in A) \Rightarrow A = \mathbb{N}$
   Das Induktionsaxiom - Wenn eine Aussage für 1 *gilt und von jeder beliebigen natürlichen Zahl* x*, für die die Aussage gilt, auf deren Nachfolger geschlossen werden kann, so gilt die Aussage für alle natürlichen Zahlen.*

Darauf aufbauend kann man verschiedene Konventionen treffen. So würde die Nachfolgerfunktion unär wie folgt definiert werden:

$N(x) = x|$ (Mache 1 Strich hinter die Zahl!)

Den verschiedenen Zahlen kann man unterschiedliche Namen geben, beispielsweise:

- $N(1) = 2$
- $N(2) = 3$
- $N(3) = 4$
- $\vdots$
- $N(8) = 9$

Aus den ersten vier Peano-Axiomen folgt, dass es unendlich viele natürliche Zahlen gibt. Dies lässt empirisch nicht überprüfen und auch praktisch gesehen kann es nicht unendlich viele Zahlen geben, da alles in der realen Welt endlich ist. Die natürlich Zahlen bauen also auf einer irrigen Annahme auf. Dennoch hat es sich gezeigt, dass die natürlichen Zahlen mit den Eigenschaften der Peano-Axiome ein nützliches und verlässliches Modell sind. Man kann daraus schließen, dass wissenschaftliche Theorie nicht immer mit der Realität übereinstimmen muss, sie muss nur praktisch funktionieren.

0.3 Sätze über natürliche Zahlen

Wir definieren zunächst Addition und Multiplikation.

Definition 0.2 *(Addition) Für alle* $x, y \in \mathbb{N}$ *gilt:*

1. $x + 1 = N(x)$
2. $x + N(y) = N(x + y)$

Definition 0.3 *(Multiplikation) Für alle* $x, y \in \mathbb{N}$ *gilt:*

1. $x \cdot 1 = x$
2. $x \cdot N(y) = x \cdot y + x$

Dann gelten für die natürlichen Zahlen folgende Sätze.

Theorem 0.1 *(Kommutativgesetz der Addition) Für alle $x, y \in \mathbb{N}$ gilt:*

$$x + y \quad = \quad y + x$$

Theorem 0.2 *(Assoziativgesetz der Addition) Für alle $x, y, z \in \mathbb{N}$ gilt:*

$$(x + y) + z \quad = \quad x + (y + z)$$

Theorem 0.3 *(Distributivgesetz) Für alle $x, y, z \in \mathbb{N}$ gilt:*

$$x \cdot (y + z) \quad = \quad x \cdot y + x \cdot z$$

Diese Sätze werden im Folgenden bewiesen. Hierbei ist das Axiom $1 \neq N(x)$ sehr wichtig. Ohne es würde eine modulo -Arithmetik entstehen, wie sie zum Beispiel in Computern vorzufinden ist ($\mod 2^{32}$ oder $\mod 2^{64}$). Dies führt unter anderem dazu, dass Überläufe (overflow) auftauchen, wenn die größte darstellbare Zahl ($2^{32} - 1$ oder $2^{64} - 1$) überschritten wird. Werden Überläufe als Fehler behandelt, so gilt das Assoziativgesetz nicht mehr.
Beispiel:

$$\underbrace{(2^{32} - 1) + 1}_{\text{Überlauf}} - 1 \quad \neq \quad (2^{32} - 1) \underbrace{+1 - 1}_{0}$$

Da aber $1 \neq N(x)$ für die natürlichen Zahlen im Allgemeinen gilt, gilt auch das Assoziativitätsgesetz und wir wollen es im Folgenden beweisen. Dazu benötigen wir zunächst mehrere Lemmas.

Lemma 0.4 *Für alle $x \in \mathbb{N}$ gilt:*

$$(x + 1) + 1 \quad = \quad x + (1 + 1)$$

Beweis:

$$\begin{array}{lcll} (x + 1) + 1 & = & N(x + 1) & \text{(Def. Addition 1.)} \\ & & x + N(1) & \text{(Def. Addition 2.)} \\ & & x + (1 + 1) & \text{(Def. Addition 1.)} \end{array} \qquad \square$$

Lemma 0.5 *Für alle $x, y \in \mathbb{N}$ gilt:*

$$(x + y) + 1 \quad = \quad x + (y + 1)$$

Beweis:

$$\begin{array}{lcll} (x + y) + 1 & = & N(x + y) & \text{(Def. Addition 1.)} \\ & = & x + N(y) & \text{(Def. Addition 2.)} \\ & = & x + (y + 1) & \text{(Def. Addition 1.)} \end{array} \qquad \square$$

Nun soll die Assoziativität der Addition bewiesen werden.

$$\forall x, y, z \in \mathbb{N}: \quad (x + y) + z \quad = \quad x + (y + z)$$

Beweis: per Induktion über z
Induktionsanfang: Für $z = 1$ gilt Lemma 0.5. ✔
Induktionsschritt: $z \to N(z)$

$$\begin{array}{lcll} (x + y) + N(z) & = & N((x + y) + z) & \text{(Def. Addition 2.)} \\ & & N(x + (y + z)) & \text{(Induktionsvoraussetzung)} \\ & & x + N(y + z) & \text{(Def. Addition 2.)} \\ & & x + (y + N(z)) & \text{(Def. Addition 2.)} \end{array} \qquad \square$$

Desweiteren definieren wir außer Addition und Subtraktion die Exponentiation von natürlichen Zahlen.

Definition 0.4 *(Exponentiation) Für alle $a, b \in \mathbb{N}$ gilt:*

1. $a^1 = a$
2. $a^{N(b)} = a^b \cdot a$

Hieraus lässt sich ein weiteres Lemma beweisen.

Lemma 0.6 *Für alle $a, b, c \in \mathbb{N}$ gilt:*

$$a^b \cdot a^c = a^{b+c}$$

Erstaunlicherweise lassen sich so Regeln der Potenzrechnung herleiten, ohne dass dazu Zahlendarstellungen (insbesondere Dezimalzahlen) eingeführt werden müssen.

0.4 Die Null und ihr Wesen

Bisher haben wir bewusst eine Zahl außen vor gelassen, nämlich die *Null*. Die Zahl Null stammt aus dem arabisch-indischen Raum und setzte sich erst nach dem Mittelalter in Europa durch, dabei ist sie eine ganz außerordentliche Errungenschaft, die die Bildung von Stellenwertsystemen erst ermöglicht und so elementar zur Entwicklung der modernen Mathematik beigetragen hat. Im Vergleich zu allen anderen Zahlen nimmt die Null eine Sonderstellung ein, da man beim Zählen von Dingen nicht auf sie stößt. Fragt man sich trotzdem, was die Null denn nun zählt, so gelangt man zu der Erkenntnis:

Die Null zählt Nichts, wo etwas sein könnte.

Anhand dieser Bedeutung erschließt sich auch das Symbol der Null "0". Es stellt den Rand eines Lochs dar. Schließlich macht erst der Rand des Lochs das Loch zum Loch.

Wir erweitern nun die natürlichen Zahlen um die Zahl 0 und nennen die resultierende Menge $\mathbb{N}_0$. Die Peano-Axiome werden entprechend angepasst.

1. $0 \in \mathbb{N}_0$
2. $x \in \mathbb{N}_0 \Rightarrow N(x) \in \mathbb{N}_0$
3. $\forall x \in \mathbb{N}_0 : 0 \neq N(x)$
4. $\forall x, y \in \mathbb{N}_0 : x \neq y \Rightarrow N(x) \neq N(y)$
5. $(A \subset \mathbb{N}_0 \wedge 0 \in A \wedge \forall x \in A : N(x) \in A) \Rightarrow A = \mathbb{N}_0$

Die Verbindung zu den ursprünglichen natürlichen Zahlen erhalten wir über die Definition der 1:

$$N(0) = 1$$

Außerdem müssen die Definitionen der Rechenoperationen erweitert werden.

- Addition: $\forall x \in \mathbb{N}_0 : x + 0 = x$
- Multiplikation: $\forall x \in \mathbb{N}_0 : x \cdot 0 = 0$
- Exponentiation: $\forall x \in \mathbb{N} : x^0 = 1$

Der einfacheren Schreibweise halber soll im Folgenden die Konvention $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ gelten. Das Symbol $\mathbb{N}$ bezeichnet also ab sofort die natürlichen Zahlen inklusive Null.

0.5 Stellenwertsysteme

Stellenwertsysteme erlauben die effiziente Darstellung von Zahlen zu einer beliebigen Basis B mit $B \in \mathbb{N}, B \geq 2$. Je nach Anwendungsbereich können verschiedene Basen festgelegt werden.

- Menschen: $B = 10$ (Anzahl der Zehen)
- Zweizehige Faultiere: $B = 4$ (Anzahl der Zehen)
- Computer: $B = 2$ (Binärzahlen, Spannung HIGH$\mapsto 1$ oder LOW$\mapsto 0$)

Zu jeder Basis gibt es eine Menge von Ziffern $\{0, \ldots, B-1\}$. Zahlen werden dann durch Vektoren aus Ziffern dargestellt. So ist $a \in \{0, \ldots, B-1\}^n$ ein Vektor aus n Ziffern zur Basis B. Die verwendete Mengennotation kann wie folgt interpretiert werden (siehe Ende dieses Abschnitts für eine formale Definition).

$$\begin{aligned} A^n &= A \times A \times \cdots \times A \\ &= \{(a_{n-1}, \ldots, a_0) \mid \forall i \in \{0, \ldots, n-1\} : a_i \in A\} \end{aligned}$$

Für Vektoren aus Binärzahlen der Länge 2 gilt zum Beispiel:

$$\{0,1\}^2 = \{(0,0),(0,1),(1,0),(1,1)\}$$

Wir führen abkürzende Schreibweisen der Vektoren ein.

$$\begin{aligned} a &= a_{n-1} \cdots a_0 \\ &= a[n-1:0] \end{aligned}$$

Zur Auswahl von Teilsequenzen aus Bitvektoren definieren wir für $i, j \in \mathbb{N}, i \geq j$:

$$a[i:j] = a_i \cdots a_j$$

Zahlendarstellungen lässt sich auch immer ein Wert zuweisen, welcher nun definiert werden soll.

Definition 0.5 *(Wert zu Zahlendarstellungen) Der Wert einer Zahl die als Ziffern-Vektor $a \in \{0, \ldots, B-1\}^n$ der Länge n dargestellt wird, lässt sich wie folgt berechnen.*

$$\langle a \rangle_B = \sum_{i=0}^{n-1} a_i \cdot B^i$$

Beispiel:

$$\begin{aligned} \langle 101 \rangle_4 &= a_2 \cdot 4^2 + a_1 \cdot 4^1 + a_0 \cdot 4^0 \\ &= 1 \cdot 4^2 + 0 \cdot 4^1 + 1 \cdot 4^0 \\ &= 16 + 1 \\ &= 17 \\ \langle 101 \rangle_2 &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 4 + 1 \\ &= 5 \\ \langle 101 \rangle_Z &= 1 \cdot Z^2 + 0 \cdot Z^1 + 1 \cdot Z^0 \\ &= 100 + 1 \\ &= 101 \end{aligned}$$

Im jeweils vorletzten Schritt benötigen wir eine Konvention, da wir von Zahlen zu Zeichenreihen übergehen. Wir identifizieren normalerweise Zahlen mit ihren Darstellungen zur Basis $Z = N(9)$ (Zehn), also:

$$a[n-1:0] \quad \text{mit} \quad \langle a[n-1:0] \rangle_Z.$$

Die Beschäftigung mit Zahlendarstellungen macht die Unterscheidung zwischen Z und ihrer Darstellung 10 im Dezimalsystem notwendig. Für einstellige Zahlen $a_0 \in \{0, \ldots, B-1\}$ identifizieren wir grundsätzlich die Zahl mit ihrer Darstellung.

$$\langle a_0 \rangle_B \quad = \quad a_0 \cdot B^0 \quad = \quad a_0 \cdot 1 \quad = \quad a_0$$

Einschub: Summen-Notation

Die Definition der Zahlendarstellung verwendet die gebräuchliche Summennotation mit dem Summensymbol $\sum$. Eine Summe $\sum_{i=k}^{n} s_i$ addiert alle Summanden s_i der Indizes k bis n mit $k \leq n$.

$$\sum_{i=k}^{n} s_i \quad = \quad s_n + \cdots + s_k$$

Formal kann man diese Summe wie folgt rekursiv definieren:

$$\sum_{i=k}^{k} s_i \quad = \quad s_k \qquad\qquad \sum_{i=k}^{n+1} s_i \quad = \quad s_{n+1} + \sum_{i=k}^{n} s_i$$

Einschub: Bitfolgen

Wie bereits gesehen, betrachten wir außer einfachen binären Werten (Bits) auch Folgen solcher Bits, die als Zahlen interpretiert werden können. Diese Folgen lassen sich mit Hilfe der Mengenlehre wie folgt formalisieren. Seien A, B Mengen, so bezeichnet $A \times B$ die Menge aller Paare von Elementen aus A und B, formal:

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

Alle Bitfolgen der Länge zwei lassen sich dann durch die Elemente der Menge $\{0,1\} \times \{0,1\}$ darstellen.

$$\{0,1\} \times \{0,1\} = \{(0,0), (0,1), (1,0), (1,1)\}$$

Um Schreibaufwand zu sparen führen wir die Konvention ein, solche Mengen auch als $\{00, 01, 10, 11\}$ schreiben zu dürfen. Für Folgen A^n beliebiger Länge n ließe sich folgende Konstruktionsvorschrift aufstellen.

$$A^1 = A \qquad\qquad A^n = A \times A^{n-1}$$

Für $n = 3$ erhielte man dann

$$\{0,1\}^3 = \{0,1\} \times \{0,1\}^2 = \{(a, b) \mid a \in \{0,1\}, b \in \{0,1\}^2\}$$

und mit entsprechender Konvention:

$$\{(0, (0,0)), \ldots, (1, (1,1))\} = \{000, \ldots, 111\}$$

Es ist allerdings zu beachten, dass man A^n auch anders, z.B. durch $A^n = A^{n-1} \times A$, definieren könnte. Dadurch entstünden Folgen, die formal nicht identisch mit den Folgen aus dem ersten Ansatz wären. Man betrachte das Beispiel für $\{0,1\}^3$.

$$\{0,1\}^3 = \{0,1\}^2 \times \{0,1\} = \{((0,0),0), \ldots, ((1,1),1)\} = \{000, \ldots, 111\}$$

Nach Schreib-Konvention wären die Bitfolgen mit den obigen identisch, formal gesehen allerdings nicht. Um diesem Dilemma zu entgehen, definieren wir alternativ n-stellige Folgen von Elementen aus A nicht als geschachtelte Paare, sondern als Menge der n-stelligen Funktionen nach A.

$$A^n = \{f : \{1, \ldots, n\} \longrightarrow A\}$$

Die Bitfolge 000 würde dann zum Beispiel durch diejenige Funktion f repräsentiert werden, für die gilt: $f(1) = 0$, $f(2) = 0$, $f(3) = 0$.

Kapitel 1

Arithmetic / Logic Units

Im Folgenden sollen Schaltkreise entworfen werden, die die Berechnung von arithmetischen und logischen Operationen ermöglichen. Diese werden später zu einer Arithmetic / Logic Unit (ALU) zusammengefasst, die dann im Prozessor die "Rechenarbeit" übernehmen soll. Vorerst müssen wir allerdings die Algorithmen untersuchen, die den Schaltkreisen zugrunde liegen werden.

1.1 Additionsalgorithmus

Wir betrachten zunächst die Addition. Hier ist bereits der Schul-Algorithmus für schriftliche Addition von Dezimalzahlen bekannt. Wir werden zeigen, dass die Berechnungsvorschrift für beliebige Basen erweiterbar ist und beweisen die Korrektheit des Algorithmus. Zuvor benötigen wir allerdings noch ein wichtiges Lemma.

Lemma 1.1 *(Zerlegungslemma) Sei $m, n \in \mathbb{N}, m < n$, dann lässt sich $a[n-1:0]$ in zwei Teilvektoren $a[n-1:m]a[m-1:0]$ zerlegen. Dann gilt:*

$$\langle a[n-1:0]\rangle_B = \langle a[n-1:m]\rangle_B \cdot B^m + \langle a[m-1:0]\rangle_B$$

Beispiel: Das folgende Beispiel verdeutlicht dieses Lemma, das für alle Stellenwertsysteme gilt, anhand von Dezimalzahlen.

$$\begin{aligned} 1951 &= 19\ 51 \\ &= 19 \cdot 100 + 51 \\ &= \underbrace{19 \cdot 10^2}_{2=m} + 51 \end{aligned}$$

Beweis: Wir benutzen die Definition des Werts zu Zahlendarstellungen.

$$\begin{aligned} \langle a\rangle_B &= \textstyle\sum_{i=0}^{n-1} a_i \cdot B^i && \text{(Zahlendarstellung)} \\ &= \textstyle\sum_{i=m}^{n-1} a_i \cdot B^i + \sum_{i=0}^{m-1} a_i \cdot B^i && \text{(Assoziativität Addition)} \\ &= \textstyle\sum_{i=0}^{n-m-1} a_{m+i} \cdot B^{m+i} + \langle a[m-1:0]\rangle_B && \text{(Indexverschiebung, Zahlendarstellung)} \\ &= \textstyle\sum_{i=0}^{n-m-1} a_{m+i} \cdot B^m \cdot B^i + \langle a[m-1:0]\rangle_B && \text{(Lemma 0.6)} \\ &= \textstyle\sum_{i=0}^{n-m-1} B^m \cdot a_{m+i} \cdot B^i + \langle a[m-1:0]\rangle_B && \text{(Assoziativität Multiplikation)} \\ &= B^m \cdot \textstyle\sum_{i=0}^{n-m-1} a_{m+i} \cdot B^i + \langle a[m-1:0]\rangle_B && \text{(Distributivitätsgesetz)} \\ &= B^m \cdot \textstyle\sum_{i=0}^{n-m-1} b_i \cdot B^i + \langle a[m-1:0]\rangle_B && \text{(Umbenennung } b[n-m-1:0] = a[n-1:m]) \\ &= \langle b[n-m-1:0]\rangle_B \cdot B^m + \langle a[m-1:0]\rangle_B && \text{(Kommutativität Mult., Zahlendarstellung)} \\ &= \langle a[n-1:m]\rangle_B \cdot B^m + \langle a[m-1:0]\rangle_B && \text{(Rückbenennung)} \quad \square \end{aligned}$$

Wir kommen nun zur Schulmethode für schriftliche Addition. Tabelle 1.1 zeigt diese für ein einfaches dezimales Beispiel.

i	3	2	1	0	
a		2	0	9	
b		8	3	3	
c	1	0	1	0	$\leftarrow c_{in}$
s	1	0	4	2	

Tabelle 1.1: Schulmethode der schriftlichen Addition mit $c_{in} = 0$

Hier bezeichnet $c_{in} \in \{0, 1\}$ einen Eingangsübertrag, welcher für gewöhnliche Addition stets 0 ist. Das obige bekannte Schema lässt sich auch auf beliebige Basen ausdehnen.

Definition 1.1 *(Additionsalgorithmus) Die formale Definition der Schulmethode lautet wie folgt:*

- $c_0 = c_{in}$
- $\langle c_{i+1}\ s_i \rangle = a_i + b_i + c_i$
- $s_n = c_n$

Die Definition kann auch als Fallunterscheidung

$$\langle c_{i+1}\ s_i \rangle \quad = \quad \begin{cases} a_i + b_i & : \quad c_i = 0 \\ N(a_i + b_i) & : \quad c_i = 1 \end{cases}$$

geschrieben werden.

Einschub: Das Kleine 1+1

Hinter der Addition $a_i + b_i$ verbirgt sich das Kleine 1+1, das als eine Tabelle aus Definitionen und Sätzen aufgefasst werden kann. Die Zeilen haben die Form

$$a_i + b_i \quad = \quad \langle c_{i+1}\ s_i \rangle_Z.$$

So ist zum Beispiel $2 + 2 = \langle 0\ 4 \rangle_Z$ ein Satz, den wir bereits über das Assoziativitätsgesetz bewiesen haben. Andererseits stellt $3+1 = \langle 0\ 4 \rangle_Z$ die Definition der 4 dar. Der Satz $1+3 = \langle 0\ 4 \rangle_Z$ wiederum folgt aus dem Kommutativgesetz. Für zweistellige Ergebnisse sei exemplarisch folgendes Beispiel gegeben

$$9 + 1 \quad = \quad N(9) \quad = \quad Z \quad = \quad 1 \cdot Z + 0 \quad = \quad \langle 1\ 0 \rangle_Z.$$

Einschub: Minus

Bisher wurde stehts nur addiert, also von Zahlen aus weitergezählt. Möchte man dagegen etwas abziehen, also zum Beispiel $x \dot{-} 1$ berechnen, so muss eine Vorgängerfunktion $V(x)$ definiert werden, so dass $x \dot{-} 1 = V(x)$. Die Subtraktion wird dabei auf die natürlichen Zahlen beschränkt, was durch das spezielle Symbol "$\dot{-}$" angezeigt wird. Die Vorgängerfunktion wird dann folgendermaßen rekursiv definiert.

- $V(0) = 0$
- $V(N(x)) = x$

Korrektheit des Additionsalgorithmus

Nun soll die Korrektheit der Schulmethode zur schriftlichen Addition gezeigt werden. Für die Summenbits und den obersten Übertrag soll gelten:

$$\langle c_n\ s[n-1:0] \rangle_B \quad = \quad \langle a[n-1:0] \rangle_B + \langle b[n-1:0] \rangle_B + c_0$$

Beweis: per Induktion über n
Induktionsanfang: $n = 1$

$$\begin{aligned}\langle a_0\rangle_B + \langle b_0\rangle_B + c_0 &= a_0 \cdot B^0 + b_0 \cdot B^0 + c_0 \quad \text{(Def. } \langle\ \rangle\text{)} \\ &= a_0 + b_0 + c_0 \quad \text{(Potenzrechnung)} \\ &= \langle c_1\ s_0\rangle_B \quad \text{(Additionsalgorithmus)} \quad \checkmark\end{aligned}$$

Induktionsschritt: $n-1 \to n$

$$\begin{aligned}&\langle a[n-1:0]\rangle_B + \langle b[n-1:0]\rangle_B + c_0 \\ &= a_{n-1} \cdot B^{n-1} + \langle a[n-2:0]\rangle_B + b_{n-1} \cdot B^{n-1} + \langle b[n-2:0]\rangle_B + c_0 \quad \text{(Zerlegungslemma)} \\ &= a_{n-1} \cdot B^{n-1} + b_{n-1} \cdot B^{n-1} + \underbrace{\langle a[n-2:0]\rangle_B + \langle b[n-2:0]\rangle_B + c_0}_{\langle c_{n-1}\ s[n-2:0]\rangle_B} \quad \text{(Kommutativgesetz)} \\ &= a_{n-1} \cdot B^{n-1} + b_{n-1} \cdot B^{n-1} + \quad\quad \langle c_{n-1}\ s[n-2:0]\rangle_B \quad \text{(Induktionsvoraussetzung)} \\ &= a_{n-1} \cdot B^{n-1} + b_{n-1} \cdot B^{n-1} + c_{n-1} \cdot B^{n-1} + \langle s[n-2:0]\rangle_B \quad \text{(Zerlegungslemma)} \\ &= (a_{n-1} + b_{n-1} + c_{n-1}) \cdot B^{n-1} + \langle s[n-2:0]\rangle_B \quad \text{(Distributivgesetz)} \\ &= \langle c_n\ s_{n-1}\rangle_B \cdot B^{n-1} + \langle s[n-2:0]\rangle_B \quad \text{(Additionsalgorithmus)} \\ &= \langle c_n\ s[n-1:0]\rangle_B \quad \text{(Zerlegungslemma)} \quad \square\end{aligned}$$

Der Beweis zeigt, dass die Schulmethode auch für beliebige Basen funktioniert. Im Folgenden wollen wir Binärzahlen betrachten es soll also $B = 2$ gelten und somit die Konvention:

$$\langle a\rangle \quad = \quad \langle a\rangle_2$$

Für den binären Additionsalgorithmus und $a, b, c \in \{0,1\}$ gibt es dann acht Möglichkeiten.

$a\ b\ c$	$a+b+c$	$\langle c'\ s\rangle$
0 0 0	0	0 0
0 0 1	1	0 1
0 1 0	1	0 1
0 1 1	2	1 0
1 0 0	1	0 1
1 0 1	2	1 0
1 1 0	2	1 0
1 1 1	3	1 1

Dieser Zusammenhang lässt sich auch als Funktion

$$f \quad : \quad \{0,1\}^3 \quad \longrightarrow \quad \{0,1\}^2$$

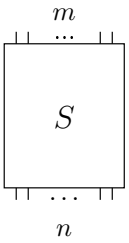
darstellen. Über die Definition solcher Schaltfunktionen und die Konstruktion entsprechender Schaltkreise wird im nächsten Abschnitt berichet.

1.2 Schaltfunktionen und Schaltkreise

Schaltfunktionen der Form

$$f \quad : \quad \{0,1\}^m \quad \longrightarrow \quad \{0,1\}^n$$

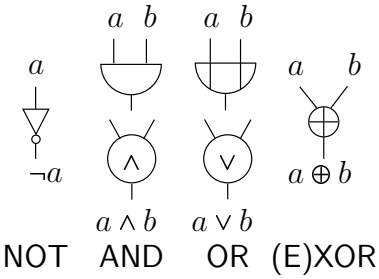
beschreiben Algorithmen mit m Ein- und n Ausgangswerten. Um sie in Hardware zu realisieren, benötigen wir einen *Schaltkreis* S, der f *berechnet*.



Im folgenden sollen beide Konzepte genauer untersucht werden.

1.2.1 Gatter und boolesche Ausdrücke

Die grundlegenden Bausteine von Schaltkreisen sind Gatter.



Diese Schaltungen berechnen die elementaren booleschen Funktionen wie folgt.

Zeile	x	y	$x \wedge y$	$x \vee y$	$x \oplus y$	$\neg x$
0	0	0	0	0	0	1
1	0	1	0	1	1	1
2	1	0	0	1	1	0
3	1	1	1	1	0	0

Mit Hilfe der Operationen, lassen sich boolesche Ausdrücke zusammensetzen. Dabei gibt es folgende

- Konstanten:

  $$0, 1$$

- Prioritäten:
  - Unäre Operatoren ($\neg$) haben Vorrang vor binären Operatoren ($\wedge, \vee, \oplus$)
  - Es gilt *„Punkt vor Strich“*, wobei $\wedge$ und $\oplus$ als Punkt- und $\vee$ als Strichrechnung angesehen werden.
  - Binäre Operationen werden stets von links nach rechts geklammert.

- Rechenregeln (Sätze):
  - Kommutativgesetze

    $$\begin{aligned} x \wedge y &\equiv y \wedge x \\ x \vee y &\equiv y \vee x \\ x \oplus y &\equiv y \oplus x \end{aligned}$$

  - Assoziativgesetze

    $$\begin{aligned} (x \wedge (y \wedge z)) &\equiv ((x \wedge y) \wedge z) \\ (x \vee (y \vee z)) &\equiv ((x \vee y) \vee z) \\ (x \oplus (y \oplus z)) &\equiv ((x \oplus y) \oplus z) \end{aligned}$$

– Distributivgesetze

$$\begin{aligned}(x \wedge (y \vee z)) &\equiv ((x \wedge y) \vee (x \wedge z))\\ (x \vee (y \wedge z)) &\equiv ((x \vee y) \wedge (x \vee z))\end{aligned}$$

– Komplementärgesetze

$$\begin{aligned}x \wedge \neg x &\equiv 0\\ x \vee \neg x &\equiv 1\end{aligned}$$

Beweis: durch Wahrheitstabellen

- Abkürzungen:

$$\begin{aligned}x \wedge y &\mathrel{\widehat{=}} xy\\ \neg x &\mathrel{\widehat{=}} \overline{x}\\ &\mathrel{\widehat{=}} /x\\ &\mathrel{\widehat{=}} {\sim}x\end{aligned}$$

Die Menge der booleschen Ausdrücke kann wie folgt formalisiert werden.
Definition:

- Menge der Variablen:

$$V = \{X_0, X_1, X_2, \ldots\} = \{X_i \mid i \in \mathbb{N}\}$$

- Boolesche Ausdrücke der Stufe 0:

$$B_0 = V \cup \{0, 1\}$$

- Boolesche Ausdrücke der Stufe $i+1$:

$$B_{i+1} = B_i \cup \{(\neg e) \mid e \in B_i\} \cup \{(e_1 \circ e_2) \mid e_1, e_2 \in B_i, \circ \in \{\wedge, \vee, \oplus\}\}$$

- Vollständig geklammerte boolesche Ausdrücke:

$$B = \bigcup_{i=0}^{\infty} B_i$$

Wie angegeben beschreibt diese Definition vollständig geklammerte boolesche Ausdrücke. Unvollständig geklammerte boolesche Ausdrücke verwenden Abkürzungen für geklammerte Ausdrücke. Hier sind Prioritätenregelungen notwendig für eine eindeutige Auswertung der Ausdrücke.

Einschub: Das "=" und sein Wesen

Das Gleichheitszeichen besitzt nicht nur eine Bedeutung sondern zwei. Zum Einen kann es in Sätzen und Definitionen verwendet werden, wie z.B.:

$$(a+b)^2 = a^2 + 2ab + b^2$$

Diese Gleichung gilt immer und für alle möglichen Werte von a und b. Man bezeichnet eine solche Aussage als *Identität*. Zur Verdeutlichung oder Betonung dessen, kann auch das Identitätssymbol "≡" verwendet werden. Betrachtet man dagegen die Gleichungen.

$$\begin{aligned}a + 2 = 3a &\Leftrightarrow 2 = 2a\\ &\Leftrightarrow a = 1,\end{aligned}$$

so wird hier das Gleichheitszeichen mit einer anderen Bedeutung verwendet. Die Gleichheit gilt nur für bestimmte a, die bestimmt werden können. Man spricht daher von *Bestimmungsgleichungen*.
Im Allgemeinen unterscheidet man nicht explizit zwischen Identitäten und Bestimmungsgleichungen um Schreibarbeit zu sparen. Es wird dasselbe Gleichheitszeichen "=" verwendet, solange aus dem Kontext klar ist, wann mit Identitäten und wann mit Bestimmungsgleichungen gerechnet wird.

Anhand von zwei booleschen Ausdrücken e_1, e_2 lässt sich der Unterschied von Identität und Bestimmungsgleichung folgendermaßen aufzeigen.

- Identität:

  $$e_1 \quad \equiv \quad e_2$$

  Für alle $X_i \in V$ gilt, ...

- Bestimmungsgleichung:

  $$e_1 \quad = \quad e_2$$

  Bestimme $X_i \in V$ so, ...

... dass beim *Einsetzen* das *Gleiche* herauskommt.
Dabei bedeutet das Gleiche, dass beide Ausdrücke entweder zu 0 oder zu 1 ausgewertet werden. Das Einsetzen in und Auswerten von booleschen Ausdrücken wird nun definiert.

$$\varphi : V \quad \longrightarrow \quad \{0, 1\}$$

deklariert eine Einsetzungsfunktion φ, die booleschen Variablen eine bestimmte Belegung zuordnet, zum Beispiel:

V	X_0	X_1	X_2	X_3	$\ldots$
$\varphi[X_i]$	1	0	0	1	$\ldots$

Zu n booleschen Variablen existieren 2^n mögliche Einsetzungen. Wir dehnen nun die Definition von φ auf boolesche Ausdrücke aus, um die Auswertung jener zu ermöglichen.

Definition 1.2 *(Wert von Ausdrücken)*

- $\varphi[0] = 0$
- $\varphi[1] = 1$
- $\varphi[(\neg e)] = \neg\varphi[e]$
- *Für* $\circ \in \{\wedge, \vee, \oplus\} : \varphi[(e_1 \circ e_2)] = \varphi[e_1] \circ \varphi[e_2]$

Diese Definition ist keine Definition im konventionellen Sinne, da nicht etwas Neues mit Hilfe alter Begriffe und bekannter Kombinationsmöglichkeiten definiert wird. Ähnlich wie bei den Peano-Axiomen, wo für das Induktionsaxiom schon die Fähigkeit zu Zählen vorausgesetzt wurde, benutzen wir hier bereits in der Definition axiomatisch die Fähigkeit Ausdrücke auszuwerten. Nach allen Einsetzungen durch φ erhält man einen booleschen Ausdruck über Konstanten, den man wiederum auswerten muss. Eine konventionelle aber längere rekursive Definition zur Auswertung von $e \circ e'$ ohne den Vorgriff auf das Verständnis von Ausdrücken würde lauten:

1. $u = \varphi[e]$ - Werte e aus!
2. $v = \varphi[e']$ - Werte ggf. e' aus!
3. $\varphi[(\neg e)] = \neg u$ bzw. $\varphi[(e \circ e')] = u \circ v$ - Bestimme aus u und v mit Hilfe der entsprechenden Tabelle den Funktionswert!

Dieser Algorithmus benötigt lediglich die Fähigkeit, Funktionswerte aus einer Tabelle abzulesen, nicht aber, komplexe boolesche Ausdrücke berechnen zu können.

Beispiel:

$\varphi[x_1] = 1$
$\varphi[x_2] = 0$
$\varphi[(x_1 \vee (x_1 \wedge x_2))]$

$= u \vee v,$	$u = \varphi[x_1],$	$v = \varphi[(x_1 \wedge x_2)]$			(Algorithmus für φ)
$= u \vee v,$	$u = 1,$	$v = u' \wedge v',$	$u' = \varphi[x_1],$	$v' = \varphi[x_2]$	(φ, Algorithmus)
$= u \vee v,$	$u = 1,$	$v = u' \wedge v',$	$u' = 1,$	$v' = 0$	(φ)
$= u \vee v,$	$u = 1,$	$v = 0$			(Funktionstabelle $\wedge$)
$= 1$					(Funktionstabelle $\vee$)

1.2.2 Schaltfunktionen

Unser ursprüngliches Ziel war die Definition von Funktionen, die uns die Möglichkeit geben, Systeme, wie zum Beispiel einen Addierer, formal zu beschreiben. Rekapitulieren wir die Tabelle in Abschnitt 1.1 für $a = x_2, b = x_1, c = x_0$, Übertrag c' und Summenbit s, dann suchen wir $e_1, e_2 \in B$, so dass:

$$\begin{aligned} c'(x_2, x_1, x_0) &\equiv e_2 \\ s(x_2, x_1, x_0) &\equiv e_1 \end{aligned}$$

Hierbei fällt allerdings auf, dass c' und s keine booleschen Ausdrücke sind. Unsere bisherige Definition dieser ist einfach nicht ausreichend. Daher müssen wir die erweiterten booleschen Ausdrücke

$$E = \bigcup_{i=0}^{\infty} E_i$$

einführen. Es wird eine Menge von Funktionssymbolen

$$F = \{f_0, f_1, f_2, \ldots\} = \{f_i \mid i \in \mathbb{N}\}$$

hinzugefügt. Die Menge $N = \{n_0, n_1, \ldots\}$ beinhaltet die Anzahl von Funktionsargumenten n_i zur Funktion f_i. Für E_0 definieren wir einfach $E_0 = B_0$. Die Ausdrücke der Stufe $i + 1$ werden wie folgt definiert

$$\begin{aligned} E_{i+1} = E_i \cup \{f_j(e_{n_j-1}, \ldots, e_0) \mid f_j \in F, \forall k \in \{0, \ldots, j-1\} : e_k \in E_i\} \cup \{(\neg e) \mid e \in E_i\} \cup \\ \{(e_1 \circ e_2) \mid e_1, e_2 \in E_i, \circ \in \{\wedge, \vee, \oplus\}\} \end{aligned}$$

Durch diese Konstruktion können boolesche Ausdrücke mit sogenannten Schaltfunktionen vermischt werden. Die Einsetzungsfunktion wird entsprechend erweitert.

$$\varphi[f_j(e_{n_j-1}, \ldots, e_0)] = f_j(\varphi[e_{n_j-1}], \ldots, \varphi[e_0])$$

Desweiteren kann man zeigen, dass zu jeder Schaltfunktion ein entsprechender boolescher Ausdruck existiert.

Theorem 1.2 *(Darstellungssatz) Sei $f : \{0,1\}^n \longrightarrow \{0,1\}$ eine Schaltfunktion mit n Variablen, dann*

$$\exists e \in B \quad : \quad e \equiv f(x_{n-1}, \ldots, x_0),$$

sodass e f berechnet, also $\forall \varphi : \varphi[f] = \varphi[e]$.

Zum Beweis stellen wir folgende Definitionen und Lemmas auf.

Definition 1.3 *(Lösung einer Bestimmungsgleichung) Seien $e_1, e_2 \in B$ und $\varphi : V \longrightarrow \{0,1\}$ eine Einsetzung, für deren Erweiterung auf E gilt:*

$$\varphi[e_1] = \varphi[e_2],$$

dann heißt φ die Lösung der Bestimmungsgleichung

$$e_1 = e_2.$$

Zum Beispiel lautet für die Gleichung $\overline{X}_2 X_1 \overline{X}_0 = 1$ die Lösung $\varphi[X_2] = 0, \varphi[X_1] = 1, \varphi[X_0] = 0$.

Lemma 1.3 *(Äquivalenz von booleschen Ausdrücken) Gegeben sei die Identität $e \equiv e'$, dann gilt:*

$$\forall \varphi \quad : \quad \varphi[e] = 1 \Leftrightarrow \varphi[e'] = 1,$$

Es müssen zum Beweis einer Äquivalenz also nur boolesche Bestimmungsgleichungen gelöst werden. Offensichtlich gilt obiges Lemma nicht für arithmetische Ausdrücke.

Lemma 1.4 *Seien φ eine Einsetzung und $e_1, e_2 \in B$. Dann ist φ genau dann eine Lösung von $e_1 \wedge e_2 = 1$, wenn φ auch eine Lösung von $e_1 = 1$ und $e_2 = 1$ ist, also:*

$$\varphi[e_1 \wedge e_2] = \varphi[1] \quad \Leftrightarrow \quad \varphi[e_1] = 1 \text{ und } \varphi[e_2] = 1,$$

Beweis:

$$\begin{array}{lrcll} & \varphi[e_1 \wedge e_2] = 1 & = & \varphi[1] & \\ \Leftrightarrow & \varphi[e_1] \wedge \varphi[e_2] & = & 1 & \text{(Definition } \varphi\text{)} \\ \Leftrightarrow & \varphi[e_1] = 1 & \text{und} & \varphi[e_2] = 1 & \text{(Funktionstabelle } \wedge\text{)} \quad \square \end{array}$$

Per Induktion lässt sich dieses Lemma auch allgemein für $n-1$-fache Verundung beweisen:

$$\begin{array}{rcl} \varphi[e_1 \wedge e_2 \wedge \ldots \wedge e_n] = 1 & \Leftrightarrow & \varphi[e_1] = 1 \wedge \varphi[e_2] = 1 \wedge \ldots \wedge \varphi[e_n] = 1 \\ & \Leftrightarrow & \forall i : \varphi[e_i] = 1 \quad \text{(Lemma } \bigwedge\text{)} \end{array}$$

Weitere Lemmas sind:

$$\begin{array}{rcl} \varphi[\overline{e}] = 1 & \Leftrightarrow & \varphi[e] = 0 \\ \varphi[e_1 \vee e_2 \vee \ldots \vee e_n] = 1 & \Leftrightarrow & \varphi[e_1] = 1 \vee \varphi[e_2] = 1 \vee \ldots \vee \varphi[e_n] = 1 \\ & \Leftrightarrow & \exists i : \varphi[e_i] = 1 \quad \text{(Lemma } \bigvee\text{)} \end{array}$$

Für den Beweis des Darstellungssatzes benötigen wir noch eine weitere Notation.

Definition 1.4 *(Literale) Sei X eine Variable und $\varepsilon \in \{0,1\}$, dann wird ein Literal folgendermaßen definiert.*

$$X^\varepsilon \quad = \quad \begin{cases} \overline{X} & : \quad \varepsilon = 0 \\ X & : \quad \varepsilon = 1 \end{cases}$$

Hierzu können wir ein nützliches Lemma beweisen.

Lemma 1.5 *Sei φ eine Einsetzung und x eine boolesche Variable.*

$$\varphi[x^\varepsilon] = 1 \quad \Leftrightarrow \quad \varphi[x] = \varepsilon$$

Beweis:
Fall $\varepsilon = 1$:

$$\varphi[x^\varepsilon] = \varphi[x^1] = \varphi[x] = 1 \quad \Leftrightarrow \quad \varphi[x] = 1 = \varepsilon \qquad \checkmark$$

Fall $\varepsilon = 0$:

$$\varphi[x^\varepsilon] = \varphi[x^0] = \varphi[\overline{x}] = 1 \quad \Leftrightarrow \quad \varphi[x] = 0 = \varepsilon \qquad \square$$

Hinweis:
Normalerweise verzichten wir beim Rechnen auf die φ-Schreibweise. Wir ersetzen stets $\varphi[e]$ mit e. Allerdings können nur mit Hilfe der φ-Notation Lemmas wie $\overline{e} = 1 \Leftrightarrow e = 0$ oder $e \wedge e' = 1 \Leftrightarrow e = 1 \wedge e' = 1$ (Lemma 1.4) bewiesen werden.

Um boolesche Ausdrücke zu gegebenen Schaltfunktionen zu finden, benötigen wir weitere Definitionen und Lemmas.

Definition 1.5 *(Monom) Sei $a[n-1:0] \in \{0,1\}^n, X[n-1:0] \in V^n$ dann bezeichnet $m(a)$ das Monom zum Bitvektor a.*

$$m(a) \quad = \quad X_{n-1}^{a_{n-1}} \wedge X_{n-2}^{a_{n-2}} \wedge \cdots X_0^{a_0} = \bigwedge_{i=0}^{n-1} X_i^{a_i}$$

Lemma 1.6 *Für ein $a[n-1:0] \in \{0,1\}^n$ gilt:*

$$\begin{aligned} m(a) = 1 \quad &\leftrightarrow \quad X = a \\ &\leftrightarrow \quad \forall i \in \{0,\ldots,n-1\} : a_i = X_i \end{aligned}$$

Beweis:

$$\begin{aligned} m(a) = 1 \quad &\leftrightarrow \quad X_{n-1}^{a_{n-1}} \wedge X_1^{a_1} \wedge \cdots X_0^{a_0} = 1 \quad \text{(Definition Monom)} \\ &\leftrightarrow \quad \forall i : X_i^{a_i} = 1 \quad \text{(Lemma } \textstyle\bigwedge\text{)} \\ &\leftrightarrow \quad \forall i : X_i = a_i \quad \text{(Lemma 1.5)} \quad \square \end{aligned}$$

Jedes Monom $m(a)$ deckt eine Zeile in der Wertetabelle ab. Es wird 1, sobald die Variablen $X_{n-1},\ldots,X_0$ die entsprechenden Werte $a_{n-1},\ldots,a_0$ annehmen.
Beispiel:

$$\overline{X}_2 X_1 \overline{X}_0 = X_2^0 X_1^1 X_0^0 = 1 \quad \Leftrightarrow \quad X_2 X_1 X_0 = 010$$

Es lässt sich der Träger von f definieren:

Definition 1.6 *(Träger von f)*

$$T(f) = \{a \mid a \in \{0,1\}^n, f(a) = 1\}$$

Mit diesen Hilfsmitteln kann man zeigen, dass zu jeder Schaltfunktion ein boolescher Ausdruck gefunden werden kann der diese berechnet.

Theorem 1.7 *(Darstellungssatz) Für jede Schaltfunktion $f : \{0,1\}^n \longrightarrow \{0,1\}$ existiert ein boolescher Ausdruck e über $X_{n-1},\ldots,X_0$, so dass:*

$$e \quad = \quad f(X[n-1:0])$$

Zum Beweis wollen wir einen booleschen Ausdruck konstruieren, der als *vollständige disjunktive Normalform (DNF)* von f bezeichnet wird.

Definition 1.7 *(DNF) für $f : \{0,1\}^n \longrightarrow \{0,1\}$ und $a[n-1:0] \in \{0,1\}^n$*

$$d(f) = \bigvee_{a \in T(f)} m(a)$$

Dabei ist $\bigvee_{i=1}^{0} x_i = 0$, also $T(f) = \varnothing \Rightarrow d(f) = 0$. Nun wollen wir Folgendes zeigen:

$$d(f) = 1 \quad \overset{!}{\Leftrightarrow} \quad f(X_{n-1},\ldots,X_0) = 1$$

Daraus folgt dann die Identität von e $(=d(f))$ und f.
Beweis:

$$\begin{aligned} d(f) = 1 \quad &\Leftrightarrow \quad \bigvee_{a \in T(f)} m(a) = 1 \;\Leftrightarrow\; \exists a \in T(f) : m(a) = 1 \quad \text{(Lemma } \textstyle\bigvee\text{)} \\ &\Leftrightarrow \quad \exists a \in T(f) : X = a \quad \text{(Lemma 1.6)} \\ &\Leftrightarrow \quad X \in T(f) \\ &\Leftrightarrow \quad f(X) = 1 \quad \text{(Definition 1.6)} \quad \square \end{aligned}$$

Wir können demnach jede Schaltfunktion in einen booleschen Ausdruck überführen, indem wir ihre DNF konstruieren.
Beispiel: Wir betrachten ein Summenbit s eines Addierers. Der Träger von s enthält alle Eingangskombinationen, für die $s = 1$ wird. Aus der Tabelle in Abschitt 1.1 erhalten wir $T(s) = \{001, 010, 100, 111\}$. Die DNF zu s lautet dann:

$$s(x_2,x_1,x_0) \quad = \quad \overline{x}_2\overline{x}_1 x_0 \vee \overline{x}_2 x_1 \overline{x}_0 \vee x_2\overline{x}_1\overline{x}_0 \vee x_2 x_1 x_0$$

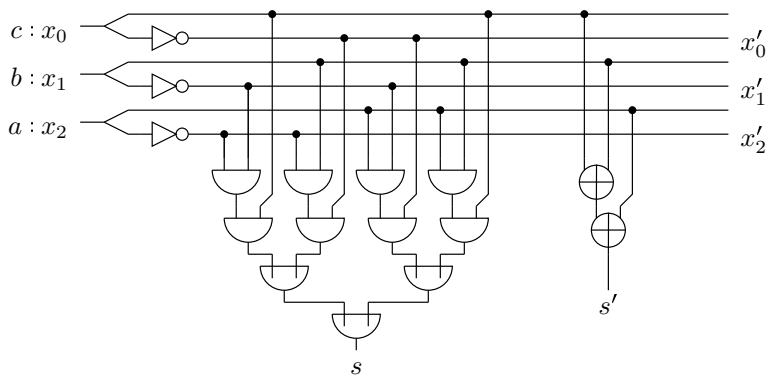


Abbildung 1.1: Summenbit-Schaltkreise

1.2.3 Schaltkreise und Straight Line Programmes

Aus verdrahteten Gattern entstehen Schaltkreise. Diese dienen der Berechnung von Schaltfunktionen. Abbildung 1.1 zeigt eine Realisierungen des Summenbits aus dem letzten Beispiel. Um die Äquivalenz von Schaltkreis und -funktion zu beweisen, benötigen wir eine algebraische Beschreibung der Verdrahtung. Dies wird durch Straight Line Programmes (SLP) ermöglicht.

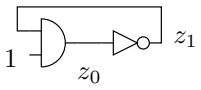
Definition 1.8 *Straight Line Program (SLP)*
Gegeben: *Eingangsvariablen* $\{X_0, X_1, \ldots\} \in V$,$\{0,1\}$
Programm: *Ein Programm* S *besteht aus einer Folge von Zeilen* $(z_0, \ldots, z_s)$. *Für alle* z_i *gilt:*

$$z_i \quad = \quad \begin{cases} \neg u & : \quad u \in V \cup \{0,1\} \vee \exists j < i : u = z_j \\ u \circ v & : \quad \circ \in \{\wedge, \vee, \oplus\} \wedge (u, v \in V \cup \{0,1\} \vee \exists j,k < i : u = z_j \vee v = z_k) \end{cases}$$

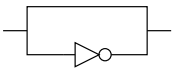
Beispiele:

$$\begin{array}{llll} z_0 = \overline{x}_0 & & z_2 = \overline{x}_2 \\ z_1 = \overline{x}_1 & & x_3 = z_2 \wedge z_1 \end{array}$$

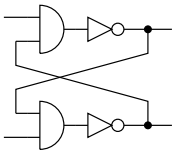
Zudem sind folgende Konstruktionen nicht mit SLPs beschreibbar:

- Oszillator 

  $z_0 = 1 \wedge z_1 \qquad z_1 = \overline{z_0}$ ✘ $j = 1 \not< 0 = i$

- Kurzschluss x  z_0

  $z_0 = \overline{x} \qquad z_0 = x$ ✘ doppelte Definition

- Latch 
Hiermit kann man Daten speichern.
Eine genauere Beschreibung befindet sich in [KP95].

Mit SLPs lässt sich ähnlich wie mit booleschen Ausdrücken rechnen. Wir erweitern die Einsetzung φ auf Straight Line Programmes. $\varphi : V \longrightarrow \{0,1\}$ legt Werte an die Eingänge eines Schaltkreises bzw. SLP an. Wir definieren weiterhin:

- $\varphi[0] = 0$, $\varphi[1] = 1$
- $z_i = x_j : \varphi[z_i] = \varphi[x_j]$
- $z_i = \neg u : \varphi[z_i] = \neg\varphi[u]$
- $z_i = u \circ v : \varphi[z_i] = \varphi[u] \circ \varphi[v]$

Zwei Schaltkreise u und v sind äquivalent, wenn gilt:

$$\begin{array}{ccc} u & \equiv & v \\ & \Leftrightarrow & \\ \forall\varphi : \varphi[u] & = & \varphi[v] \end{array}$$

So ist zum Beispiel $s' \equiv s$, denn man kann zeigen:

$$\begin{array}{lcl} s'(x[2:0]) = 1 & \Leftrightarrow & \#\{x_i \mid x_i = 1\} \text{ ist ungerade} \\ & \Leftrightarrow & s(x[2:0]) = 1 \end{array}$$

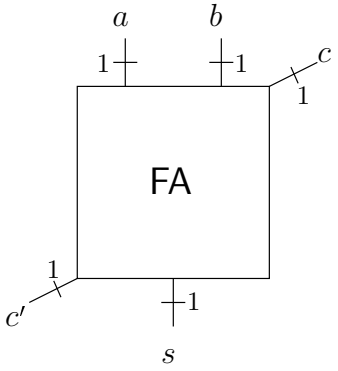
Da s und s' äquivalent sind und s' weniger aufwendig als s ist, kann man Schaltkreise, die s enthalten vereinfachen. Solange keine Zwischenergebnisse von s anderweitig verwendet werden, darf man überall s durch s' ersetzen. Wertet man s aus, so erhält man:

$$\begin{array}{lcll} s & \equiv & (x_2x_1x_0 \vee x_2\overline{x}_1\overline{x}_0) \vee (\overline{x}_2x_1\overline{x}_0 \vee \overline{x}_2\overline{x}_1x_0) & \text{(Definition } \varphi) \\ & \equiv & (\overline{x}_2\overline{x}_1x_0 \vee \overline{x}_2x_1\overline{x}_0) \vee (x_2\overline{x}_1\overline{x}_0 \vee x_2x_1x_0) & \text{(Kommutativität } \vee) \\ & \equiv & (((\overline{x}_2\overline{x}_1x_0 \vee \overline{x}_2x_1\overline{x}_0) \vee x_2\overline{x}_1\overline{x}_0) \vee x_2x_1x_0) & \text{(Assoziativität } \vee) \end{array}$$

Dies ist wiederum die ursprüngliche Definition der Schaltfunktion s. Schaltkreis und -funktion sind also äquivalent.

1.3 Addierer

Im Folgenden soll nun ein Schaltkreis entworfen werden, der Zahlen in ihrer Binärdarstellung addiert. Die kleinste Additions-Einheit, die die Bits a,b,c addiert und c' sowie s berechnet heißt Volladdierer (Full Adder).



Für die Ausgänge gilt:

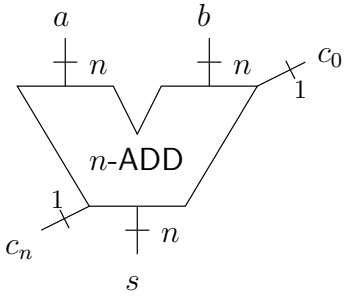
$$\langle c'\ s\rangle \quad = \quad a + b + c$$

Aus dem Volladdierer für einzelne Bits soll nun ein n-Addierer Konstruiert werden.

Definition 1.9 *(n-Addierer) Ein n-Addierer ist ein Schaltkreis mit Eingängen $a, b \in \{0,1\}^n$, $c_0 \in \{0,1\}$ und Ausgängen $s \in \{0,1\}^n, c_n \in \{0,1\}$. Für diese Signale gilt:*

$$\langle c'\ s[n-1:0]\rangle \quad = \quad \langle a[n-1:0]\rangle + \langle b[n-1:0]\rangle + c_0$$

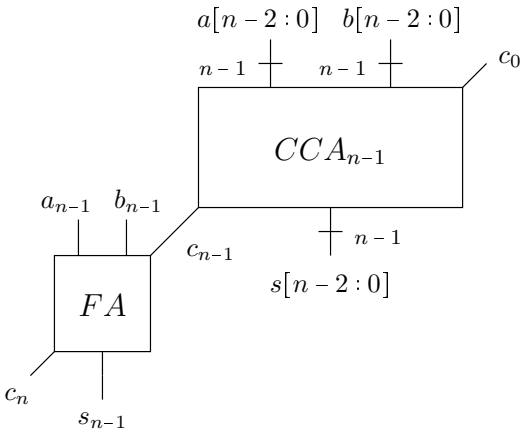
Wir verwenden für Addierer die folgende schematische Darstellung.



Dabei stellen die Verbindungen, die mit n gekennzeichnet sind, einen *Bus* dar. Ein Bus der Breite n fasst n parallele Verbindungen zusammen.

Aus dem Schul-Additionsalgorithmus ergibt sich direkt die Konstruktion des sogenannten Carry Chain Adders (CCA). In einer Kette aus Full Addern werden die Ausgangsüberträge des Vorgängers als Eingangsüberträge an den folgenden Volladdierer übergeben. Die einzelnen Summenbits s_i bilden den Summenbitstring s. Eine iterative Definition könnte wie folgt lauten.

- $CCA_1 : FA, \qquad \langle c_1\ s_0 \rangle = a_0 + b_0 + c_0 = \langle a_0 \rangle + \langle b_0 \rangle + c_0$
- $CCA_{n-1} \to CCA_n : \qquad \langle c_n\ s_{n-1} \rangle = a_{n-1} + b_{n-1} + c_{n-1}$



Da der CCA eine direkte Hardware-Umsetzung des Additionsalgorithmus ist, folgt seine Korrektheit direkt aus der Korrektheit der Schulmethode.

1.4 Kosten und Tiefe von Schaltkreisen

Um Aufwand und Geschwindigkeit eines Schaltkreises einschätzen zu können, definieren wir die folgenden Funktionen für Kosten und Tiefe eines Schaltkreises S:

Definition 1.10 *Kostenfunktion $C(S)$ (Cost)*

$$C(S) = \sum_{\text{Gatter } g_i \in S} C(g_i)$$

Die Kosten eines Schaltkreises entsprechen der Summe der Kosten aller enthaltenen Gatter.
In einem Schaltkreis S existieren Pfade von Gattern, wie in Abbildung 1.2 dargestellt, wobei ein Ausgang von Gatter g_i an einen Eingang von Gatter g_{i+1} angeschlossen ist. Betrachten wir einen Schaltkreis als gerichteten azyklischen Graphen (DAG) $S = (V_S, E_S)$ mit Gattern $g_i \in V$ als Knoten und Verbindungen $(g_i, g_j) \in E_S = V_S \times V_S$ zwischen den Gattern als Kanten, so erhalten wir folgende Definition für einen Pfad von Gattern über S.

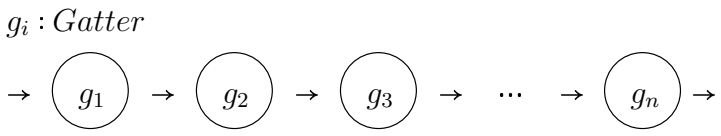


Abbildung 1.2: Pfad von Gattern in einem Schaltkreis

Definition 1.11 *(Pfad über Schaltkreis) Ein Pfad p_S über den Schaltkreis S besteht aus einer Folge von n Gattern $g_i \in V_S$*

$$p_S = (g_1, \ldots, g_n),$$

für die gilt: $\forall i \in \{0, n-1\} : (g_i, g_{i+1}) \in E_S$. *Die Menge aller Pfade über einen Schaltkreis heißt* $P(S)$.

$$P(S) = \bigcup_S p_S$$

Die Tiefe eines Pfades ergibt sich aus der Summe über die Tiefen $D(g)$ der enthaltenen Gatter.

$$D(p_S) = \sum_{i=1}^{n-1} D(g_i)$$

Nun können wir die Tiefe von Schaltkreisen wie folgt definieren.

Definition 1.12 *Tiefenfunktion $D(S)$ (Depth)*

$$D(S) = \max\{D(p) \mid p \in P(S)\}$$

Die Tiefe eines Schaltkreises entspricht dem enthaltenen Pfad mit der größten Verzögerung. Wir legen fest, dass Schaltkreise keine Zyklen enthalten dürfen. Die Kosten und Tiefen der einzelnen Gatter werden durch folgende Tabelle festgelegt.

Gatter	NOT	NAND/NOR	AND/OR	XOR/XNOR
Kosten	1	2	2	4
Tiefe	1	1	2	2

Beispiel: n-Bit Carry-Chain-Addierer $CCA(n)$ - Kosten und Tiefe belaufen sich auf

$$\begin{aligned} C(CCA_n) &= n \cdot C(FA) \quad \text{(billig)} \\ D(CCA_n) &= n \cdot D(FA) \quad \text{(langsam)} \end{aligned}$$

Da ein Eingangsübertragssignal unter Umständen die gesamte Volladdiererkette durchlaufen muss, ist die Konstruktion recht langsam. Es exisiteren allerdings schnellere Addierervarianten. Wir wollen nun betrachten, wie Kosten und Tiefe von rekursiv definierten Schaltkreisen berechnet werden können. Zunächst muss eine rekursive Formel, Rekurrenz genannt, aufgestellt werden, die die Konstruktion widerspiegelt. Im Falle der Kosten des CCA wäre das:

$$C(CCA_n) = C(CCA_{n-1}) + C(FA)$$

Diese Formel muss dann in eine geschlossene Form gebracht werden. Dazu setzt man sie wiederholt in sich selbst ein, bis man ein Schema erraten kann.

$$\begin{aligned} C(CCA_n) &= C(CCA_{n-1}) + C(FA) \\ &= (C(CCA_{n-2}) + C(FA)) + C(FA) \\ &= ((C(CCA_{n-3}) + C(FA)) + C(FA)) + C(FA) \\ &\vdots \\ &= ((\cdots(C(CCA_{n-i}) + \underbrace{C(FA)) + \cdots) + C(FA)) + C(FA)}_{i \text{ mal}} \\ &= C(CCA_{n-i}) + i \cdot C(FA) \end{aligned}$$

Dann legt man $i = n - 1$ fest, um von $C(CCA_{n-i})$ zu den Kosten eines 1-bit carry chain adders $C(CCA_1) = C(FA)$ zu gelangen. Daraus ergibt sich dann:

$$\begin{aligned} C(CCA_n) &= C(CCA_1) + (n-1) \cdot C(FA) \\ &= n \cdot C(FA) \end{aligned}$$

Diese Vermutung muss allerdings noch per Induktion über n bewiesen werden.
Beweis:
Induktionsanfang: $n = 1$

$$\begin{aligned} C(CCA_1) &= C(FA) && \text{(Definition)} \\ &= 1 \cdot C(FA) = n \cdot C(FA) && \checkmark \end{aligned}$$

Induktionsvoraussetzung: $C(CCA_{n-1}) = (n-1) \cdot C(FA)$
Induktionsschritt: $n - 1 \to n$

$$\begin{aligned} C(CCA_n) &= C(CCA_{n-1}) + C(FA) && \text{(Rekurrenz)} \\ &= (n-1) \cdot C(FA) + C(FA) && \text{(Induktionsvoraussetzung)} \\ &= n \cdot C(FA) && \square \end{aligned}$$

Eine hilfreiche Formel zum Auflösen geometrischer Reihen ist außerdem:

$$\sum_{i=0}^{n-1} q^i = \frac{q^n - 1}{q - 1}$$

Für Zweierpotenzen vereinfacht sie sich zu:

$$\begin{aligned} \sum_{i=0}^{n-1} 2^i &= \frac{2^n - 1}{2 - 1} \\ &= 2^n - 1 \end{aligned}$$

1.4.1 Multiplexer

Für die Implementierung effizienterer Addierer benötigen wir Multiplexer. Dies sind Signalweichen, mit denen ein Ausgangssignal in Abhängigkeit eines select-Signals auf eines von mehreren (hier: zwei) Eingangssignalen gesetzt wird.

Definition 1.13 *Ein n-Multiplexer (n-MUX) ist ein Schaltkreis mit Eingängen $x, y \in \{0,1\}^n$ und einem Ausgang $z \in \{0,1\}^n$, sowie einem Auswahlsignal $s \in \{0,1\}$.*

$$z = \begin{cases} x & : \quad s = 0 \\ y & : \quad s = 1 \end{cases}$$

Abbildung 1.3 zeigt das zugehörige Schaltsymbol sowie die Implementierung eines 1-Bit MUX. Es folgt eine mögliche SLP-Beschreibung des Multiplexers.

$$z_i = s \ ? \ x : y \quad : \quad s, x, y \in V \cup \{0,1\} \vee \exists j, k, l < i : x = z_j \vee y = z_k \vee s = z_l$$

Ein n-MUX wird mit n 1-Multiplexern implementiert, indem jedem Ausgangsbit ein eigener Multiplexer für die jeweiligen Eingangsbits zugeordnet wird (siehe Abbildung 1.4). Alle 1-MUXe werden mit demselben select-Signal s angesteuert. Multiplexer werden der Realität durch spezielle Gatter realisiert. Wir legen ihre Kosten auf 3 und die Tiefe auf 2 fest.
Die Kosten des n-MUX betragen $C(n - MUX) = n \cdot C(1 - MUX)$. Wegen der Parallelschaltung bleibt die Tiefe bei $D(n - MUX) = 2$.

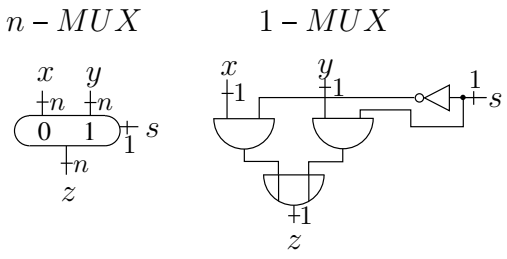


Abbildung 1.3: Multiplexer: Schaltsymbol und Implementierung

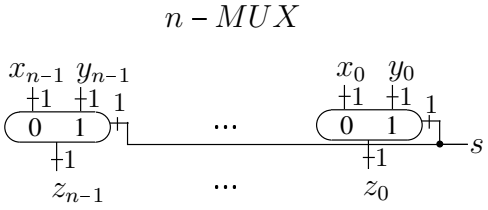


Abbildung 1.4: n-Multiplexer

1.4.2 Conditional Sum Adder

Mit Hilfe des Multiplexers lässt sich nun ein verbesserter, schnellerer Addierer bauen. Die Operanden a und b werden für $n = 2^k$ in obere und untere Bits unterteilt:

$$a_H = a[n-1:\tfrac{n}{2}] \qquad a_L = a[\tfrac{n}{2}-1:0]$$
$$b_H = b[n-1:\tfrac{n}{2}] \qquad b_L = b[\tfrac{n}{2}-1:0]$$

Die rekursive Konstruktion eines n-Bit Conditional Sum Adders ($CSA(n)$) ist in Abbildung 1.5 dargestellt.

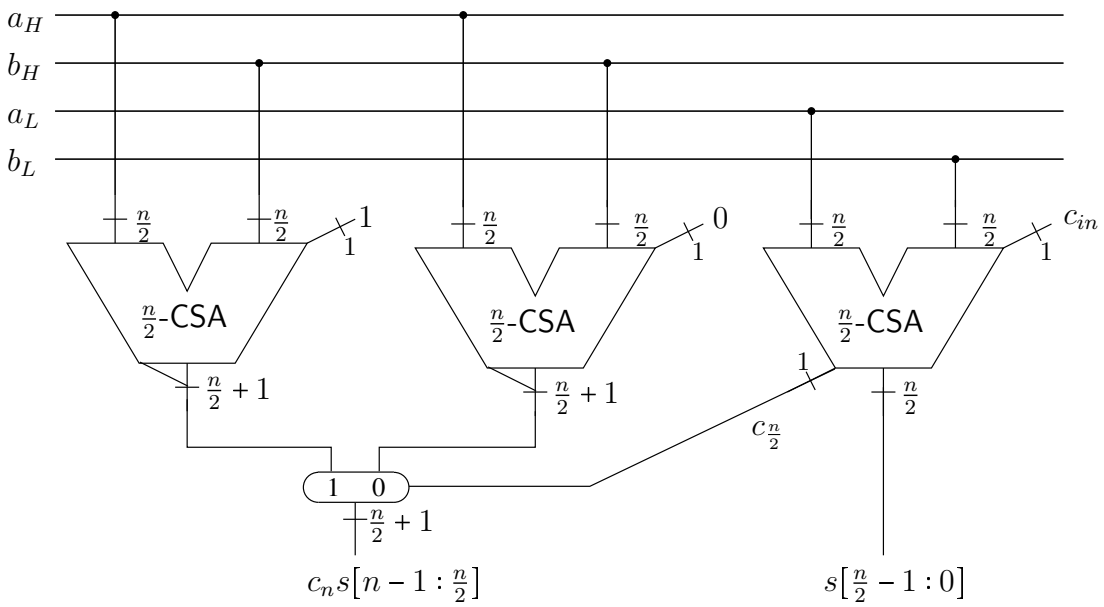


Abbildung 1.5: Conditional Sum Adder

Es werden zunächst parallel die Teilsummen für die oberen und unteren Bits mit Hilfe von $\frac{n}{2}$-CSA berechnet. Da zu diesem Zeitpunkt noch nicht klar ist, ob die unteren Bits einen Übertrag $c_{\frac{n}{2}}$ generieren,

werden für die oberen Bits beide Varianten mit carry in 1 oder 0 berechnet und dann mit einem MUX über den carry out des „unteren“ Addierers gewählt. Da rekursiv wieder Conditional Sum Adder für Signale der halben Bitbreite verwandt werden ergibt sich eine logarithmische Tiefe. 1-CSAs werden mit einem Volladdierer realisiert.

$$\begin{aligned}
CSA(1) &= FA \\
D(CSA(1)) &= D(FA) \\
D(CSA(n)) &= D(CSA(\tfrac{n}{2})) + D(MUX) = D(CSA(\tfrac{n}{2})) + 2 \\
&= D(CSA(\tfrac{n}{4})) + 2 + 2 \\
&= D(CSA(\tfrac{n}{2^3})) + \underbrace{2 + 2 + 2}_{2\cdot 3} \\
&\vdots \\
&= D(CSA(\tfrac{n}{2^k})) + 2 \cdot k \quad (n = 2^k \rightarrow k = \log_2 n) \\
&= D(CSA(1)) + 2 \cdot \log_2 n \\
&= D(FA)) + 2 \cdot \log_2 n \ \in \ \mathcal{O}(\log n)
\end{aligned}$$

Damit wäre der CSA schneller als der CCA, was durch höhere Kosten „erkauft“ wird.

Allerdings ist die obige Behauptung $D(n) = 2 \cdot \log_2 n + D(FA)$ nur eine Vermutung, die noch bewiesen werden muss.

Beweis: durch Induktion über n bzw. k mit $n = 2^k$

$$D(n) \stackrel{!}{=} 2 \cdot \log_2 n + D(FA)$$

Induktionsanfang: $n = 1 = 2^0$

$$D(1) = D(FA) \stackrel{!}{=} 2 \cdot \log_2 1 + D(FA) = 2 \cdot 0 + D(FA) = D(FA) \quad \checkmark$$

Induktionsvoraussetzung: $D(\frac{n}{2}) = 2 \cdot \log_2 \frac{n}{2} + D(FA)$
Induktionsschritt: $\frac{n}{2} \rightarrow n$, bzw. $k - 1 \rightarrow k$

$$\begin{aligned}
D(n) &= D(\tfrac{n}{2}) + \underbrace{D(MUX)}_{\quad\quad 2} \quad \text{(Konstruktion CSA)} \\
&= D(\tfrac{n}{2}) + \\
&= 2 \cdot \log_2 \frac{n}{2} + D(FA) + 2 \quad \text{(Induktionsvorraussetzung)} \\
&= 2 \cdot (\log_2 n - \log_2 2) + D(FA) + 2 \quad \text{(Logarithmusgesetze)} \\
&= 2 \cdot \log_2 n - 2 + 2 + D(FA) \quad \text{(Distributivität, Kommutativität)} \\
&= 2 \cdot \log_2 n + D(FA) \quad \square
\end{aligned}$$

Damit verfügen wir nun also über einen relativ schnellen Addierer für Binärzahlen. Um eine ALU zu konstruieren, benötigen wir jedoch auch einen Schaltkreis, der Binärzahlen voneinander subtrahieren kann.

1.5 Binäre Subtraktion

Nun wird nach einer Möglichkeit gesucht, die Subtraktion von Binärzahlen durchzuführen. Dazu wird allerdings noch Grundwissen in der Modulorechnung benötigt, welches der folgende Exkurs vermitteln soll.

1.5.1 Modulorechnung

Wir definieren eine binäre Relation $\equiv_{\mathrm{mod}\, k}$ auf den Ganzen Zahlen.

Definition 1.14 *$a, b \in \mathbb{Z}$ heißen „kongruent modulo $k \in \mathbb{N}$“, wenn $a \equiv_{\mathrm{mod}\, k} b$, das heißt:*

$$\exists x \in \mathbb{Z} : a = b + x \cdot k$$

Alternativ kann $a \equiv b \mod k$ geschrieben werden.

Lemma 1.8 *$\equiv_{\mathrm{mod}\, k}$ ist eine Äquivalenzrelation, das bedeutet, es gelten die drei Eigenschaften:*

- *Reflexivität* $\leftrightarrow$ $a \equiv_{\mathrm{mod}\, k} a$
- *Symmetrie* $\leftrightarrow$ $a \equiv_{\mathrm{mod}\, k} b \Leftrightarrow b \equiv_{\mathrm{mod}\, k} a$
- *Transitivität* $\leftrightarrow$ $(a \equiv_{\mathrm{mod}\, k} b \wedge b \equiv_{\mathrm{mod}\, k} c) \Rightarrow a \equiv_{\mathrm{mod}\, k} c$

Die Relation $\equiv_{\mathrm{mod}\, k}$ erzeugt für Zahlen $a \in \mathbb{Z}$ und $k \in \mathbb{N}$ verschiedene Äquivalenzklassen der Form

$$[a]_k = \{b \mid a \equiv_{\mathrm{mod}\, k} b\}.$$

Eine Äquivalenzklasse fasst also Elemente zusammen, die bzgl. der Äquivalenzrelation gleich sind. Zwei Zahlen $a, b \in M$ einer Menge M liegen dann in derselben Äquivalenzklasse, wenn gilt $[a] = [b]$ bzw. $[a] \cap [b] \neq \varnothing$. Statt mit Äquivalenzklassen kann man auch mit Repräsentanten dieser argumentieren. $R = \{r_1, r_2, \ldots\}$ wird dann ein Repräsentantensystem genannt, falls

- $\forall i \neq j : [r_i] \neq [r_j]$
- $\bigcup_{r \in R} [r] = M$

Für das Rechnen $\equiv_{\mathrm{mod}\, k}$ gilt

- $M = \mathbb{Z}$
- $[a]_k = \{\ldots, a - 2k, a - k, a, a + k, a + 2k, \ldots\}$

Das Standard-Repräsentantensystem für Modulorechnung ist gemeinhin

$$R_k = \{0, 1, \ldots, k - 1\}$$

Man kann zeigen, dass $\bigcup_{r \in R_k} [r] = \mathbb{Z}$ und $\forall r, r' \in R_k.\ r \neq r' \Rightarrow [r] \cap [r'] = \varnothing$ gilt.

Definition 1.15 *Die unäre Funktion $(a \bmod k) \in \{0, \ldots, k - 1\}$ weißt a den Repräsentanten seiner Äquivalenzklasse bezüglich $\equiv_{\mathrm{mod}\, k}$ zu.*

$$a \bmod k = \varepsilon\{r \mid r \in R_k \wedge a \in [r]_k\}$$

Diese Definition verwendet den Hilbert-ε-Operator. Für eine Menge $A \neq \varnothing$ liefert εA ein Element aus A zurück. Insbesondere gilt $\varepsilon\{a\} = a$.

Im Falle von $a \geq 0$ entspricht $\beta = (a \bmod k)$ dem Rest β der ganzzahligen Division von positiven a durch k. Das folgende Korollar verdeutlicht den Zusammenhang zwischen den Gebilden $(a \bmod k)$ und $\equiv_{\mathrm{mod}\, k}$.

Korollar:

$$a \equiv_{\mathrm{mod}\, k} y \wedge a \in \{0, \ldots, k - 1\} \Leftrightarrow a = (y \bmod k)$$

Desweiteren existiert ein wichtiges Lemma für Binärzahlen.

Lemma 1.9 *Sei $a \in \{0,1\}^m$, $k = 2^n$ und $m > n$, dann gilt:*

$$\langle a[m-1:0]\rangle \equiv_{mod\, 2^n} \langle a[n-1:0]\rangle$$

Beweis:

$$\begin{aligned}
\langle a[m-1:0]\rangle &= \langle a[m-1:n]\rangle \cdot 2^n + \langle a[n-1:0]\rangle \quad \text{(Zerlegungslemma 1.1)} \\
\Leftrightarrow \langle a[m-1:0]\rangle &\equiv_{\mathrm{mod}\, 2^n} \underbrace{\langle a[n-1:0]\rangle}_{\leq 2^n-1} \quad \text{(Definition } \equiv_{\mathrm{mod}\, k},\ x = \langle a[m-1:n]\rangle) \ \square \\
\Leftrightarrow (\langle a[m-1:0]\rangle \bmod 2^n) &= \langle a[n-1:0]\rangle \quad \text{(Korollar)}
\end{aligned}$$

Man kann also für Binärzahlen Modulorechnung mit einer Zweierpotenz 2^n einfach durchführen, indem man alle Bits verwirft, die größer gleich 2^n gewichtet werden, das heißt alle a_i mit $i \geq n$ ignoriert und nur die „Rest-Bits“ a_i mit $i \in \{0,\ldots,n-1\}$ betrachtet.

Weitere nützliche Lemmas sind:

Lemma 1.10 *Sei $a \in \{0,\ldots,k-1\}$ dann gilt:*

$$a = (a \bmod k)$$

Beweis: Die Behauptung folgt trivial aus der Reflexivität von $\equiv_{\mathrm{mod}\, k}$ und dem Korollar.

Lemma 1.11 *Sei $a \equiv_{\mathrm{mod}\, k} a' \ \wedge\ b \equiv_{\mathrm{mod}\, k} b'$ dann gilt:*

$$a + b \equiv_{\mathrm{mod}\, k} a' + b' \ \wedge\ a - b \equiv_{\mathrm{mod}\, k} a' - b'$$

Beweis: Wir setzen die Definition von $\equiv_{\mathrm{mod}\, k}$ ein — $a = a' + x \cdot k \quad b = b' + x \cdot k$ — und addieren/subtrahieren die Gleichungen: $a \pm b = a' \pm b' + (x \pm y) \cdot k$. Es existieren also Faktoren $(x \pm y)$ für k, so dass die Behauptung gilt.

1.5.2 Subtraktionsalgorithmus

Im folgenden wird der Subtraktionsalgorithmus für binäre Zahlen eingeführt. Dabei verwenden wir folgende abkürzende Notation mit einen Bitstring $X[n-1:0]$:

$$\bar{X} = (\bar{X}_{n-1},\ldots,\bar{X}_0)$$

Definition 1.16 *(Subtraktionsalgorithmus) Für $x, y \in \{0,1\}^n$ gilt:*

$$\langle x\rangle - \langle y\rangle \equiv_{\mathrm{mod}\, k} \langle x\rangle + \langle \bar{y}\rangle + 1$$

Im Falle von $\langle x\rangle \geq \langle y\rangle$ erhält man: $\langle x\rangle - \langle y\rangle = (\langle x\rangle + \langle \bar{y}\rangle + 1 \bmod 2^n)$

Durch Negation des Subtrahenden und Addition von 1 soll also eine Subtraktion realisiert werden, wenn man nur die letzten n Bits betrachtet. Warum dies stimmen soll, ist im ersten Moment nicht intuitiv klar, das Beispiel in Tabelle 1.2 für $n = 4$ lässt jedoch erahnen, dass der Algorithmus funktioniert. Um dies zu beweisen, muss man sich mit Two's-Complement-Zahlen beschäftigen.

	1	0	0	1	⇔		9	⇒		1	0	0	1		
−	0	1	1	1	⇔	−	7	⇒	+	1	0	0_1	0_1		
	0	0	1	0	⇔		2		1	0	0	1	0		mod 2^4
								=		0	0	1	0		✔

Tabelle 1.2: Beispiel zum Subtraktionsalgorithmus

1.5.3 Two's-Complement-Zahlen

Ein Bitstring $a \in \{0,1\}^n$ lässt sich nicht nur als eine Binärzahl $\langle a \rangle$ interpretieren sondern auch als Two's-Complement-Zahl $[a]$. Dadurch lassen sich nun auch negative Zahlen darstellen.

Definition 1.17 *(Two's-Complement-Zahlen) Sei $a \in \{0,1\}^n$, dann heißt*

$$[a[n-1:0]] = -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle$$

die Two's-Complement-Darstellung der Länge n von a.

Beispiel: $-5 \quad [1011] = -1 \cdot 2^3 + \langle 011 \rangle = -8 + 3 = -5 \quad \checkmark$

Definition 1.18 $T_n = \{-2^{n-1}, \ldots, 2^{n-1}-1\}$ *ist die Menge der mit n Bits darstellbaren Two's-Complement-Zahlen. Es gilt*

$$-2^{n-1} \le [a] \le \langle a[n-2:0] \rangle \le \langle \underbrace{1 \cdots 1}_{n-1} \rangle = \sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1$$

Der einzige Unterschied zur Binärdarstellung ist also, dass das oberste Bit negativ gewichtet wird. Dadurch verlagert sich der darstellbare Zahlenbereich um 2^{n-1} ins Negative. Weiterhin ist bemerkenswert, dass der Wertebereich asymmetrisch ist. Dies hat aber auch den Vorteil, dass die Null dadurch eindeutig definiert ist, anders als zum Beispiel bei Dezimalzahlen, wo es $+0$ und -0 gibt.

Der Korrektheitsbeweis der Subtraktion verwendet verschiedene Lemmata über Two's-Complement-Zahlen. Diese und weitere sind hier mit den zugehörigen Beweisen aufgelistet.
Dabei ist $a = a[n-1:0] \in \{0,1\}^n$.

Lemma 1.12 *(TWOC Lemma 1) sign bit*

$$[a[n-1:0]] < 0 \Leftrightarrow a[n-1] = 1$$

Beweis:
"⇒": Komplement

$$a_{n-1} = 0 \quad \Rightarrow \quad [a] = \underbrace{\langle a[n-2:0] \rangle}_{\ge 0} \ \Rightarrow \ [a] \ge 0 \qquad \checkmark$$

"⇐":

$$\begin{aligned} a_{n-1} = 1 \quad &\Rightarrow \quad [a] \le -2^{n-1} + \langle \underbrace{1 \cdots 1}_{n-1} \rangle \\ &\Rightarrow \quad [a] \le -2^{n-1} + (2^{n-1} - 1) = -1 \\ &\Rightarrow \quad [a] < 0 \qquad \square \end{aligned}$$

Das oberste, negativ gewichtete Bit wird auch *sign bit* genannt, da es über das Vorzeichen der Two's-Complement-Darstellung entscheidet. Ist es 1, so ist die zugehörige TWOC-Zahl negativ, ansonsten positiv.

Lemma 1.13 *(TWOC Lemma 2)*

$$\langle a \rangle = [0a]$$

Beweis:

$$\begin{aligned} [0a] &= -2^{n-1} \cdot 0 + \langle a \rangle \quad \text{(Definition [ ])} \\ &= \langle a \rangle \quad \square \end{aligned}$$

Binärzahlen können in TWOC-Zahlen umgewandelt werden, indem man ihnen eine Null voranstellt. Dadurch erhöht sich die Länge der Zahlendarstellung von n auf $n+1$.

Lemma 1.14 *(TWOC Lemma 3)*

$$[a] \equiv \langle a \rangle \mod 2^n$$

Beweis:

$$\begin{aligned}
[a] - \langle a \rangle &= -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle - (a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle) \quad \text{(Def. [ ] und Zerl.lemma 1.1)} \\
&= -2 \cdot a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle - \langle a[n-2:0] \rangle \\
&= -a_{n-1} \cdot 2^n \\
&\equiv_{\mathrm{mod}\, 2^n} 0 \\
\Leftrightarrow [a] &\equiv_{\mathrm{mod}\, 2^n} \langle a \rangle \quad \square
\end{aligned}$$

Dieses Lemma ist von entscheidender Bedeutung. Es sagt aus, dass es egal ist, ob ein Rechner Bitstrings als Binär- oder TWOC-Zahlen interpretiert. Solange die Rechnungen modulo 2^n durchgeführt werden, bleiben die entsprechenden Bits identisch.

Lemma 1.15 *(TWOC Lemma 4) sign extension*

$$[a_{n-1}a] = [a]$$

Beweis:

$$\begin{aligned}
[a_{n-1}a] &= -a_{n-1} \cdot 2^n + \langle a \rangle \quad \text{(Definition [ ])} \\
&= -a_{n-1} \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle \quad \text{(Zerlegungslemma 1.1)} \\
&= (-2 \cdot a_{n-1} + a_{n-1}) \cdot 2^{n-1} + \langle a[n-2:0] \rangle \\
&= -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle \\
&= [a] \quad \text{(Definition [ ])} \quad \square
\end{aligned}$$

Durch sign extension kann man die Länge eines Bitstrings vergrößern, ohne den Wert der zugehörigen Two's-Complement-Darstellung zu ändern. Man darf das sign bit beliebig oft wiederholen.

Lemma 1.16 *(TWOC Lemma 5)*

$$-[a] = [\bar{a}] + 1$$

Beweis:

$$\begin{aligned}
[\bar{a}] + 1 &= -\bar{a}_{n-1} \cdot 2^{n-1} + \langle \bar{a}[n-2:0] \rangle + 1 \quad \text{(Definition [ ])} \\
&= -\underbrace{(1 - a_{n-1})}_{\bar{a}_{n-1}} \cdot 2^{n-1} + \sum_{i=0}^{n-2} \underbrace{\bar{a}_i}_{1-a_i} \cdot 2^i + 1 \quad \text{(Definition } \langle\ \rangle \text{ und } \bar{x} = 1 - x,\ x \in \{0,1\}) \\
&= -2^{n-1} + 2^{n-1} \cdot a_{n-1} + \underbrace{\sum_{i=0}^{n-2} 2^i}_{} - \underbrace{\sum_{i=0}^{n-2} a_i \cdot 2^i}_{} + 1 \\
&= -2^{n-1} + 2^{n-1} \cdot a_{n-1} + (2^{n-1} - 1) - \langle a[n-2:0] \rangle + 1 \\
&= -(-a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle) + 2^{n-1} - 2^{n-1} - 1 + 1 \\
&= -[a] \quad \square
\end{aligned}$$

Eine TWOC-Zahl kann also negiert werden, indem man alle Bits invertiert und 1 addiert.

1.5.4 Korrektheitsbeweis

Nun haben wir alle Aussagen die wir für den Korrektheitsbeweis des Subtraktionsalgorithmus brauchen. Zu zeigen war $\langle a \rangle - \langle b \rangle = \langle a \rangle + \langle \bar{b} \rangle + 1$ mod 2^n für $a, b \in \{0,1\}^n$, $\langle a \rangle \geq \langle b \rangle$.

Beweis:

$$\begin{aligned}
\langle a\rangle - \langle b\rangle &= \langle a\rangle - [0b] \quad \text{(TWOC Lemma 2)}\\
&= \langle a\rangle + (-[0b]) \quad \text{(Konvention } x - y = x + (-y)\text{)}\\
&= \langle a\rangle + [1\bar{b}] + 1 \quad \text{(TWOC Lemma 5)}\\
&= \langle a\rangle + (-2^n \cdot 1 + \langle \bar{b}\rangle) + 1 \quad \text{(Definition [ ])}\\
&= \langle a\rangle + \langle \bar{b}\rangle + 1 + (-1)\cdot 2^n \quad \text{(Assoziativität, Kommutativität } +\text{)}\\
&\equiv_{\mathrm{mod}\, 2^n} \langle a\rangle + \langle \bar{b}\rangle + 1 \quad \text{(Definition } \equiv_{\mathrm{mod}\, k}, x = -1, k = 2^n\text{)} \quad \square
\end{aligned}$$

Für $\langle a\rangle \geq \langle b\rangle$ folgt $\langle a\rangle - \langle b\rangle \in \{0,\ldots,2^n-1\}$ und somit:

$$\langle a\rangle - \langle b\rangle = (\langle a\rangle + \langle \bar{b}\rangle + 1 \mod 2^n)$$

Es ist nun also bewiesen, dass der Subtraktionsalgorithmus funktioniert. Daraus folgt, dass man Subtraktion von Binärzahl mit Hilfe eines Addierers bewerkstelligen kann. Man muss lediglich den zweiten Operandeneingang negieren und den "carry in" auf 1 setzen.

1.6 Implementierung

Eine ALU (Arithmetic Logic Unit) führt arithmetische und logische Operationen aus. Die Realisierung der arithmetischen Operationen Addition und Subtraktion wurde bereits besprochen. Sie werden in einer Arithmetic Unit (AU) als Bestandteil der ALU implementiert.

1.6.1 Arithmetic Unit

Mit einer AU lassen sich Binärzahlen addieren und subtrahieren. Zunächst definieren wir Operatoren für binäre Addition und Subtraktion mit $a, b \in \{0,1\}^n$.

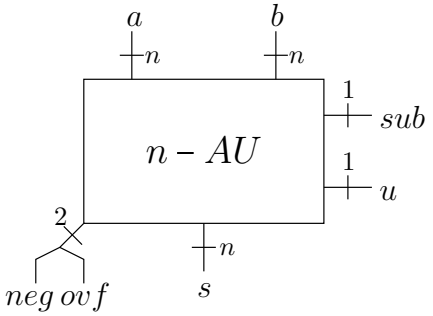
$$\begin{array}{lcl}
+_n, -_n & : & \{0,1\}^{2n} \longrightarrow \{0,1\}^n\\
+_n(a,b) & = & \mathrm{bin}_n(\langle a\rangle + \langle b\rangle \bmod 2^n)\\
-_n(a,b) & = & \mathrm{bin}_n(\langle a\rangle - \langle b\rangle \bmod 2^n)
\end{array}$$

Dabei ist $\mathrm{bin}_n(x)$ die Binärdarstellung zu einem $x \in \{0,\ldots,2^n-1\}$, die wie folgt definiert wird.

$$\mathrm{bin}_n(x) = y \quad \text{mit} \quad y \in \{a \mid a \in \{0,1\}^n, \langle a\rangle = x\}$$

Ebenso definieren wir die Binärdarstellung für TWOC-Zahlen $x \in \{-2^{n-1},\ldots,2^{n-1}-1\}$.

$$\mathrm{twoc}_n(x) = y \quad \text{mit} \quad y \in \{a \mid a \in \{0,1\}^n, [a] = x\}$$



Definition 1.19 *(Arithmetic Unit) Dies ist ein Schaltkreis mit Dateneingängen $a, b \in \{0,1\}^n$, Ausgängen $s \in \{0,1\}^n$,$ovf, neg \in \{0,1\}$ und Steuersignalen $sub, u \in \{0,1\}$. In der obigen Abbildung ist eine AU*

abgebildet. Für s gilt:

$$\begin{aligned} s \quad &= \quad \begin{cases} a +_n b & : \quad sub = 0 \\ a -_n b & : \quad sub = 1 \end{cases} \\ B_n \ni \langle s \rangle \quad &\equiv_{\mathrm{mod}\, 2^n} \quad \begin{cases} \langle a \rangle + \langle b \rangle & : \quad sub = 0 \\ \langle a \rangle - \langle b \rangle & : \quad sub = 1 \end{cases} \\ &\equiv_{\mathrm{mod}\, 2^n} \quad \begin{cases} [a] +_n [b] & : \quad sub = 0 \\ [a] -_n [b] & : \quad sub = 1 \end{cases} \\ &\equiv_{\mathrm{mod}\, 2^n} \quad [s] \in T_n \end{aligned}$$

Um die ovf- und neg-Ausgänge zu definieren betrachten wir das exakte Resultat r.

$$r \quad = \quad \begin{cases} \langle a \rangle + \langle b \rangle & : \quad (u, sub) = 10 \\ \langle a \rangle - \langle b \rangle & : \quad (u, sub) = 11 \\ [a] + [b] & : \quad (u, sub) = 00 \\ [a] - [b] & : \quad (u, sub) = 01 \end{cases}$$

Dabei steht u für unsigned*, also Operationen mit Binärzahlen (kein TWOC) und sub für Subtraktion. Für die Ausgänge ovf und neg gilt dann Folgendes.*

$$\begin{aligned} ovf = 1 \quad &\Leftrightarrow \quad u = 1 \wedge r \notin B_n \vee \qquad (B_n = \{0, \ldots, 2^n - 1\}) \\ &\qquad\quad u = 0 \wedge r \notin T_n \qquad (T_n = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}) \\ neg = 1 \quad &\Leftrightarrow \quad r < 0 \end{aligned}$$

Im Falle von $ovf = 0$ gilt $\langle s \rangle = r$ für $u = 1$ bzw. $[s] = r$ für $u = 0$.

Einschub: Bitweise Vektoroperationen

Bevor wir die arithmetische Einheit implementieren können, müssen wir noch zusätzliche Notation für bitweise Vektoroperationen einführen. Gegeben seien zwei Bitvektoren $x, y \in \{0,1\}^n$, dann definieren wir die folgenden Schreibweisen.

$$\begin{aligned} x \oplus y \quad &= \quad (x_{n-1} \oplus y_{n-1}, \ldots, x_0 \oplus y_0) \\ x \wedge y \quad &= \quad (x_{n-1} \wedge y_{n-1}, \ldots, x_0 \wedge y_0) \\ x \vee y \quad &= \quad (x_{n-1} \vee y_{n-1}, \ldots, x_0 \vee y_0) \end{aligned}$$

Überdies kann man die Verknüpfung eines Vektors $x \in \{0,1\}^n$ mit einem einzelnen Bit $z \in \{0,1\}$ definieren.

$$x \oplus z \quad = \quad (x_{n-1} \oplus z, \ldots, x_0 \oplus z)$$

Wir benötigen nur den Fall für XOR, da diesem eine besondere bedeutung zukommt. Es gilt nämlich:

$$x \oplus z \quad = \quad \begin{cases} x & : \quad z = 0 \\ \overline{x} & : \quad z = 1 \end{cases}$$

Das bedeutet, dass x in Abhängigkeit von z invertiert werden kann. Außerdem führen wir folgende Abkürzende Schreibweise für die Verfielfältigung von Bits $x \in \{0,1\}$ ein.

$$\underbrace{x \ldots x}_{n \text{ mal}} \quad \triangleq \quad x^n$$

Die Notation 0^{32} würde also eine Folge von 32 Nullen beschreiben.

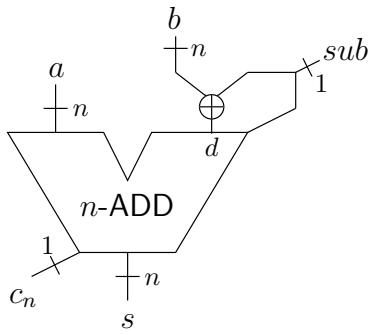


Abbildung 1.6: Implementierung der AU (ohne ovf, neg, u)

In Abbildung 1.6 ist die Implementierung der AU dargestellt. Es wird ein Addierer verwendet, bei dem der zweite Operandeneingang und der Eingangsübertrag in Abhängigkeit von sub geändert werden. Ist $sub = 0$ so wird einfach $a +_n b$ addiert. Für $sub = 1$ ändern sich die Eingänge, so dass $a +_n \overline{b} +_n 1$ also $a -_n b$ berechnet wird. Es gilt:

$$
\begin{array}{lcll}
\langle s \rangle & = & (\langle a \rangle + \langle d \rangle + sub \mod 2^n) & \text{(Konstruktion)} \\
& = & (\langle a \rangle + \langle b \oplus sub \rangle + sub \mod 2^n) & \text{(XOR)} \\
& = & \begin{cases} \langle a \rangle + \langle b \rangle \mod 2^n & : \quad sub = 0 \\ \langle a \rangle + \langle \overline{b} \rangle + 1 \mod 2^n & : \quad sub = 1 \end{cases} & (b \oplus 1 = \overline{b}) \\
& \equiv_{\mathrm{mod}\, 2^n} & \begin{cases} \langle a \rangle + \langle b \rangle & : \quad sub = 0 \\ \langle a \rangle - \langle b \rangle & : \quad sub = 1 \end{cases} & \text{(Binäre Substraktion)} \\
& = & \begin{cases} (\langle a \rangle + \langle b \rangle \mod 2^n) & : \quad sub = 0 \\ (\langle a \rangle - \langle b \rangle \mod 2^n) & : \quad sub = 1 \end{cases} & (\langle s \rangle \in B_n) \qquad \square
\end{array}
$$

neg-Bit

Das neg-Bit der AU signalisiert, dass das Ergebnis einer arithmetischen Operation negativ ist. Dies wird für Vergleichsoperationen benötigt, da für $\circ \in \{\leq, <, >, \geq, =, \neq\}$ gilt:

$$a \circ b \Leftrightarrow a - b \circ 0$$

Bei der Berechung des neg-Bits unterscheiden wir zwei Fälle:

Fall 1: $u = 1$ - unsigned (Binärzahlen)
Es muss $sub = 1$ gelten, da das Ergebnis der Addition nicht negativ werden kann. Dann erkennen wir:

$$
\begin{array}{lcll}
\langle a \rangle - \langle b \rangle & = & \langle a \rangle - [0b] & \text{(TWOC Lemma 2)} \\
& = & \langle a \rangle + [1\overline{b}] + 1 & \text{(TWOC Lemma 5)} \\
& = & \langle a \rangle - 2^n + \langle \overline{b} \rangle + 1 & \text{(Definition [ ])} \\
& = & \langle a \rangle + \langle \overline{b} \rangle + 1 - 2^n & \text{(Konstruktion)} \\
& = & \langle c_n \; s[n-1:0] \rangle - 2^n & \text{(Addierer für } sub = 1) \\
& = & c_n \cdot 2^n + \langle s[n-1:0] \rangle - 2^n & \text{(Binärdarstellung)} \\
& = & (c_n - 1) \cdot 2^n + \underbrace{\langle s[n-1:0] \rangle}_{\leq 2^n - 1} \overset{!}{<} 0 & \text{(Distributivgesetz)} \\
& & \Leftrightarrow \quad c_n = 0 & (-2^n + 2^n - 1 < 0)
\end{array}
$$

Für $u = 1$ muss also $c_n = 0$ gelten, damit das Ergebnis der Subtraktion negativ wird.

Fall 2: $u = 0$ - signed (Two's-Complement-Zahlen)
Hier kann sowohl die Summe, als auch Differenz von a und b negativ werden. Das Ergebnis ist genau dann negativ, wenn das oberste Bit (*sign bit*) 1 ist. Es kann jedoch ein overflow auftreten, so dass n Bits nicht mehr ausreichen, das korrekte Ergebnis darzustellen. Deshalb kann man nicht einfach s_{n-1} als neg-Bit wählen. Man muss vielmehr das korrekte Ergebnis $r \in T_{n+1}$ betrachten. Wir stellen die folgende Rechnung an.

$$
\begin{array}{lll}
[a] \pm [b] & = & [a] + [d] + sub \qquad \text{(Konstruktion)} \\
& = & [a_{n-1}a] + [d_{n-1}d] + sub \qquad \text{(sign extension)} \\
& \equiv_{\mathrm{mod}\, 2^{n+1}} & \langle a_{n-1}a \rangle + \langle d_{n-1}d \rangle + sub \qquad \text{(TWOC Lemma 3, Lemma 1.11)} \\
& = & \langle c_{n+1}\ s[n:0] \rangle \qquad (n+1\text{-Addierer}) \\
& \equiv_{\mathrm{mod}\, 2^{n+1}} & \langle s[n:0] \rangle \qquad \text{(Lemma 1.9)} \\
& \equiv_{\mathrm{mod}\, 2^{n+1}} & [s[n:0]] \quad \in T_{n+1} \qquad \text{(TWOC Lemma 3)}
\end{array}
$$

$[s[n:0]]$ stellt also dass korrekte Ergebnis $r \in T_{n+1}$ dar.
Mit TWOC Lemma 1 erhalten wir für das sign Bit s_n:

$$[a] \pm [b] < 0 \Leftrightarrow s_n = 1$$

s_n ist also das neg-Bit im Falle von Two's-Complement-Zahlen. Wir erhalten es in dem wir die Konstruktion des Summenbits betrachten.

$$s_n = c_n \oplus a_n \oplus d_n$$

Dabei stehen a_{n-1} und d_{n-1} aufgrund der sign extension auch an Stelle n. Insgesamt definieren wir somit das neg-Bit wie folgt.

$$neg \quad = \quad (u \wedge sub \wedge \overline{c_n}) \vee (\overline{u} \wedge (c_n \oplus a_{n-1} \oplus d_{n-1}))$$

ovf-Bit

Ein overflow tritt auf, wenn das korrekte Ergebnis der arithmetischen Operation den darstellbaren Zahlenbereich verlässt. Dann erhalten wir also nicht das korrekte Ergebnis aus der Arithmetic Unit. Um die Bedingungen für $ovf = 1$ zu finden, stellen wir auch hier eine Fallunterscheidung über u an.

Fall 1: $u = 1$ - unsigned (Binärzahlen)
Für den unsigned ovf-Ausgang einer AU gilt:

$$ovf = 1 \Leftrightarrow \begin{cases} c_n = 1 & : \quad sub = 0 \\ c_n = 0 & : \quad sub = 1 \end{cases}$$

Dies ist identisch mit der Formel:

$$(\overline{sub} \wedge c_n) \vee (sub \wedge \overline{c_n}) \quad = \quad sub \oplus c_n$$

Fall 2: $u = 0$ - signed (Two's-Complement-Zahlen)
Wir betrachten erneut das korrrekte Ergebnis der arithmetischen Operation.

$$\begin{aligned}
[a] \pm [b] &= [a] + [d] + c_{in} \quad \text{(Konstruktion)} \\
&= -a_{n-1} \cdot 2^{n-1} - d_{n-1} \cdot 2^{n-1} + \underbrace{\langle a[n-2:0]\rangle + \langle d[n-2:0]\rangle + c_{in}}_{} \quad \text{(Definition [ ])} \\
&= -a_{n-1} \cdot 2^{n-1} - d_{n-1} \cdot 2^{n-1} + \quad \langle c_{n-1}\ s[n-2:0]\rangle \quad \text{(Definition Addition)} \\
&= -a_{n-1} \cdot 2^{n-1} - d_{n-1} \cdot 2^{n-1} + \underbrace{c_{n-1} \cdot 2^{n-1} - c_{n-1} \cdot 2^{n-1}}_{=0} + \langle c_{n-1}\ s[n-2:0]\rangle \quad (\text{"}+\ 0\text{"}) \\
&= -\underbrace{(a_{n-1} + d_{n-1} + c_{n-1})}_{} \cdot 2^{n-1} + c_{n-1} \cdot 2^{n-1} + c_{n-1} \cdot 2^{n-1} + \langle s[n-2:0]\rangle \quad \text{(Zerl.lemma)} \\
&= - \quad \langle c_n\ s_{n-1}\rangle \cdot 2^{n-1} + 2 \cdot c_{n-1} \cdot 2^{n-1} + \langle s[n-2:0]\rangle \quad \text{(Definition Addition)} \\
&= -2 \cdot c_n \cdot 2^{n-1} - s_{n-1} \cdot 2^{n-1} + 2 \cdot c_{n-1} \cdot 2^{n-1} + \langle s[n-2:0]\rangle \quad \text{(Zerl.lemma)} \\
&= -c_n \cdot 2^n + c_{n-1} \cdot 2^n + [s[n-1:0]] \quad \text{(Definition [ ] , Kommutativität } +) \\
&= \underbrace{(c_{n-1} - c_n) \cdot 2^n + \underbrace{[s[n-1:0]]}_{\in T_n}}_{=r} \quad \text{(Distributivgesetz)}
\end{aligned}$$

Ob $r \in T_n$ liegt und kein overflow auftritt, hängt also nur von den beiden carry Bits c_n und c_{n-1} ab. Wir unterscheiden drei Fälle:

Fall 2.1: $c_n = c_{n-1}$

$$r = \underbrace{(c_{n-1} - c_n)}_{=0} \cdot 2^n + [s[n-1:0]] = [s[n-1:0]] \in T_n \quad \checkmark$$

In diesem Fall tritt also kein overflow auf.

Fall 2.2: $c_n = 0, c_{n-1} = 1$

$$\begin{aligned}
\Rightarrow r &= 0 \cdot 2^n + 1 \cdot 2^n + [s[n-1:0]] \\
&= \underbrace{2^n + \underbrace{[s[n-1:0]]}_{\geq -2^{n-1}}}_{\geq 2^{n-1}} \\
&\Rightarrow [a] \pm [b] \notin T_n \quad \checkmark
\end{aligned}$$

Fall 2.3: $c_n = 1, c_{n-1} = 0$

$$\begin{aligned}
\Rightarrow r &= -1 \cdot 2^n + 0 \cdot 2^n + [s[n-1:0]] \\
&= \underbrace{-2^n + \underbrace{[s[n-1:0]]}_{\leq 2^{n-1}-1}}_{\leq -2^{n-1}-1} \\
&\Rightarrow [a] \pm [b] \notin T_n \quad \checkmark
\end{aligned}$$

Aus 2.2 und 2.3 folgt:

$$\begin{aligned}
c_n \neq c_{n-1} &\Leftrightarrow [a] \pm [b] \notin T_n \\
&\Leftrightarrow ovf = 1 \quad \text{(Definition AU)} \quad \square
\end{aligned}$$

Um das ovf-Bit zu berechnen, muss man also nur die Bits c_n und c_{n-1} per XOR vergleichen. Leider stellen nicht alle Addierer den Übertrag c_{n-1} zur Verfügung. Allerdings kann es aus dem Summenbit s_{n-1} hergeleitet werden.

Für das overflow-Bit erhalten wir insgesamt:

$$ovf \quad = \quad (u \wedge (sub \oplus c_n)) \vee (\overline{u} \wedge (c_n \oplus c_{n-1}))$$

Schaltkreise, die ovf und neg effizient implementieren, sollen als Übungsaufgabe entwickelt werden.

1.6.2 Arithmetic Logic Unit

Eine ALU berechnet arithmetische und logische Funktionen für zwei Operanden $a, b \in \{0,1\}^n$. Das Ergebnis $aluop(a, b, f)$ liegt am Ausgang $alures$ an. Welche Funktion von der ALU ausgeführt werden soll, wird über den *function code* $f \in \{0,1\}^4$ gesteuert. Abbildung 1.7 zeigt die schematische Darstellung einer ALU.

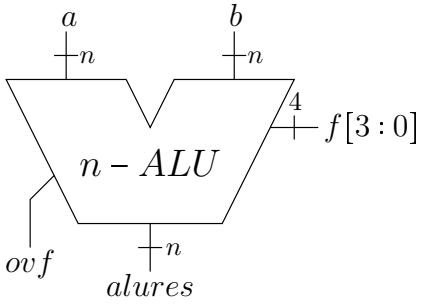


Abbildung 1.7: Arithmetic / Logic Unit

In Tabelle 1.3 sind die zum Funktionscode zugehörigen Funktionen aufgeführt. Zeilen 1-8 beinhalten die arithmetischen und booleschen Funktionen, Zeilen 9-16 die Vergleichsoperationen.
Bei Letzteren steht f_2 für <, f_1 für = und f_0 für >. Für arithmetische Operationen steuert f_0 das $unsigned$-Steuerbit der AU, also $u = \overline{f_3} f_0$. Außerdem gilt $sub = f_3 \vee f_1$, d.h. bei Vergleichsoperationen werden a und b als TWOC-Zahlen ($signed$) interpretiert.

$f[3:0]$	$aluop(a,b,f)$	ovf
0000	$a +_n b$	$ovf(u=0)$
0001	$a +_n b$	$ovf(u=1)$
0010	$a -_n b$	$ovf(u=0)$
0011	$a -_n b$	$ovf(u=1)$
0100	$a \wedge b$	0
0101	$a \vee b$	0
0110	$a \oplus b$	0
0111	$b[\frac{n}{2}-1:0]0^{\frac{n}{2}}$	0
1000	0^n	0
1001	$0^{n-1}([a] > [b])$	0
1010	$0^{n-1}([a] = [b])$	0
1011	$0^{n-1}([a] \geq [b])$	0
1100	$0^{n-1}([a] < [b])$	0
1101	$0^{n-1}([a] \neq [b])$	0
1110	$0^{n-1}([a] \leq [b])$	0
1111	$0^{n-1}1$	0

Tabelle 1.3: Funktionstabelle

In Abbildung 1.8 ist die Implementierung der ALU dargestellt. Sie besteht aus drei Blöcken:

- der Arithmetic Unit zur Berechnung von Addition und Subtraktion und zur Berechnung der Differenz für die Vergleichsoperationen (sub und u gesteuert über f),

- den booleschen Funktionen inklusive des logischen Linksshifts um $\frac{n}{2}$ Bits
- und die Logic Unit (LU), die Vergleichsoperationen realisiert. (Alle Vergleiche basieren auf der Two's-Complement-Interpretation der Operanden)

Die einzelnen Ergebnisse der verschiedenen Blöcke werden über einen Baum aus Multiplexern zum Ausgang geleitet. Die MUXe werden von den function code Bits gesteuert, so dass das Resultat der gewünschten Funktion als s am Ausgang anliegt. Es gilt:

$$alures' = \begin{cases} a +_n b & : \quad f[2:0] = 00* \\ a -_n b & : \quad f[2:0] = 01* \\ a \wedge b & : \quad f[2:0] = 100 \\ a \vee b & : \quad f[2:0] = 101 \\ a \oplus b & : \quad f[2:0] = 110 \\ b[\frac{n}{2}-1:0]0^{\frac{n}{2}} & : \quad f[2:0] = 111 \end{cases}$$

Die Ergebnisse der Vergleichsoperationen werden von der Logic Unit berechnet. Es ergibt sich der *condition code* cc, der anzeigt ob die ausgewählte Bedingung wahr ist. Der Test auf "kleiner" wird mit Hilfe des neg-Bits durchgeführt, es gilt:

$$neg = 1 \quad \Leftrightarrow \quad [a] < [b]$$

Für den Test auf Gleichheit führen wir ein Bit eq (equal) ein, für das gilt:

$$\begin{aligned} eq = 1 \quad &\Leftrightarrow \quad [a] = [b] \\ &\Leftrightarrow \quad \neg(a \neq b) \end{aligned}$$

Für dessen implementierung benutzen wir einen Gleichheitstester, wie in Abbildung 1.9 gezeigt. Dieser wird mit einem bitweisen XOR, einem $\vee$-tree (Baum aus OR-Gattern) und einem Inverter realisiert. Wir erkennen:

$$\begin{aligned} a \neq b \quad &\Leftrightarrow \quad \exists i : a_i \neq b_i \quad \Leftrightarrow \quad \exists i : a_i \oplus b_i = 1 \\ &\Leftrightarrow \quad \bigvee_i a_i \oplus b_i = 1 \quad \Leftrightarrow \quad \overline{\bigvee_i a_i \oplus b_i} = 0 \end{aligned}$$

Die Bedingung "größer" kennzeichnen wir durch das Signal pos ("Differenz streng positiv"). Dieses ist genau dann wahr, wenn a weder kleiner noch gleich b ist, also:

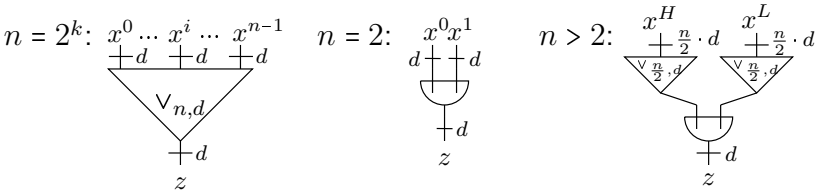
$$pos \quad = \quad \overline{neg \vee eq}$$

Den condition code erhalten wir aus der Verknüpfung der drei Signale mit den jeweiligen function code bits.

$$cc = f_2 \wedge neg \vee f_1 \wedge eq \vee f_0 \wedge pos$$

Einschub: $\vee$-Bäume

Ein $\vee_{n,d}$-tree ist ein Baum aus OR-Gattern. $n = 2^k$ Eingänge x^i der Breite d werden zu einem Resultat $z \in \{0,1\}^d$ verodert. Wir verwenden die folgende schematische Darstellung.



Die Konstruktion ist identisch zu einem vollständigen binären Baum mit d-stelligen OR-Gattern als Knoten. Auch aus anderen binären Schaltkreisen und Gattern lassen sich solche Bäume erstellen.

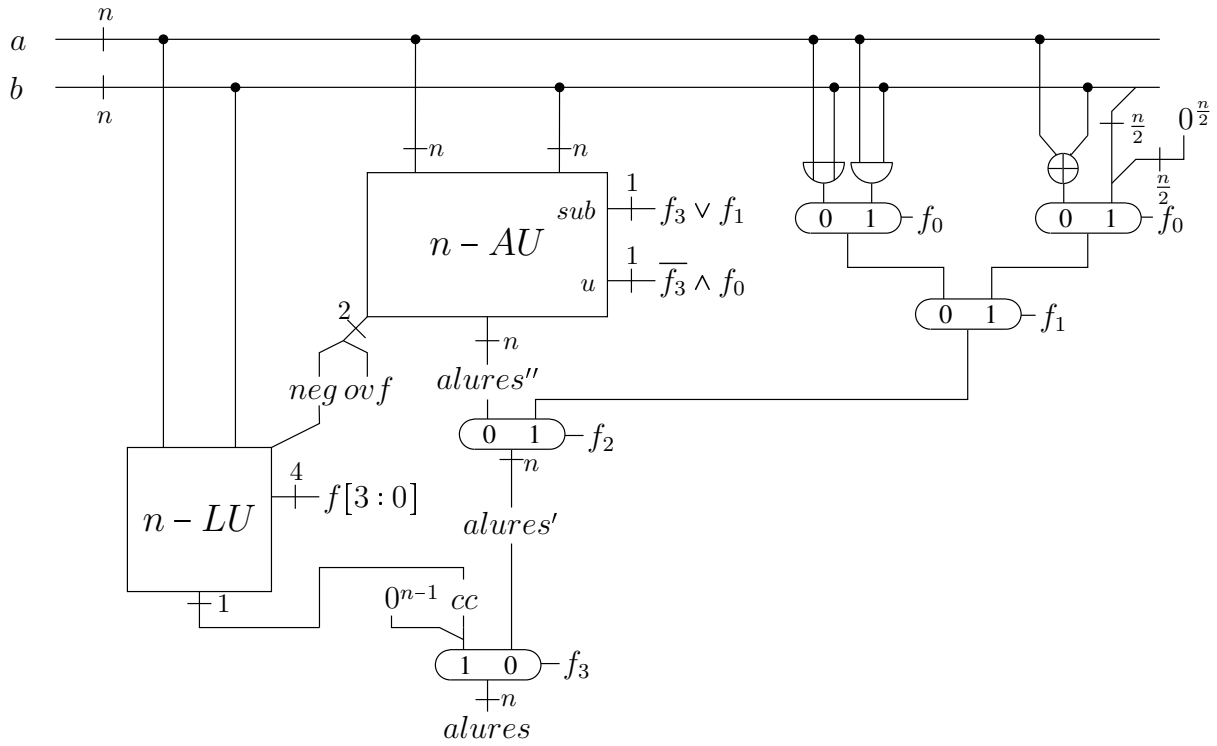


Abbildung 1.8: Implementierung der ALU

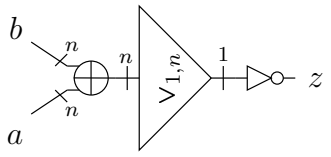


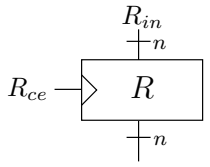
Abbildung 1.9: Gleichheitstester

Kapitel 2

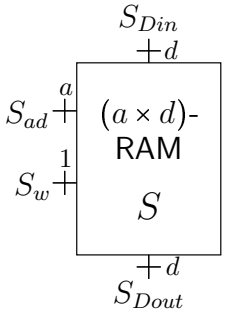
Speicherelemente und getaktete Schaltungen

Reale Hardware, wie zum Beispiel ein Prozessor, rechnet zustandsabhängig, das heißt, dass sie zu den gleichen Eingaben, nicht immer die gleichen Ergebnisse ausgibt. Die Ausgaben hängen vielmehr vom *Zustand* der Hardware ab. Um überhaupt erst einmal in einem Zustand verharren zu können, sind zwingend Speicherelemente notwendig. Solche Bauteile speichern Werte von Signalen für beliebige Zeit ab. Wir sprechen insbesondere von

- Registern R



- und RAMs S (Random Access Memory).



Der Zustand dieser Bausteine kann nur in regelmäßigen Abständen geändert werden. Zu diesem Zweck sind alle Speicherelemente mit einem Taktsignal (clock) verbunden. Die clock taktet die Schaltung, wir sprechen von *clocked circuits* oder getakteten Schaltungen. Um Schreibarbeit zu sparen lassen wir allerdings nach Möglichkeit den clock-Anschluss außer Acht. Alle Speicherelemente werden implizit von derselben clock getaktet.

Es ist anzumerken, dass wir hier *statisches RAM* betrachten (SRAM) im Gegensatz zu *dynamischem RAM* (DRAM). Erstere werden mit Registern (Flipflops) realisiert, während bei letzteren die Daten auf Kondensatoren gespeichert werden, deren Ladung regelmäßig aufgefrischt werden muss. Für die Spezifikation der Funktionsweise ist dies allerdings irrelevant.

2.1 Hardware-Konfiguration

Hardware besteht aus verschiedenen Komponenten:

- Register
- Speicher, das heißt
  - RAM (Random Access Memory)
  - oder multiport-RAM (RAM mit mehreren Ein- bzw. Ausgängen)
- Schaltkreise zwischen den Speicherelementen (glue logic)

Wir beschreiben den Zustand der Hardware im Takt t mit h^t. Dieser Zustand entspricht dabei dem Zustand der Register und RAMs.

Definition 2.1 *(Hardware-Konfiguration h) Der aktuelle Zustand der Hardware und ihrer Bestandteile wird durch eine Konfiguration beschrieben in der Form:*

$$h = (\underbrace{h.R, \ldots}_{\text{Register}}, \underbrace{h.S, \ldots}_{\text{RAMs}})$$

Die Komponenten haben dabei die Form:

- $h.R \in \{0,1\}^n$ *für jedes n-bit Register R*
- $h.S \in \{0,1\}^a \longrightarrow \{0,1\}^d$ *für jeden $(a \times d)$-RAM S*

Die Hardware-Konfiguration kann sich mit jedem Takt ändern. Die Übergangsfunktion $\delta_h(h)$ berechnet bzw. spezifiziert die Hardware h, in dem sie den Folgezustand der Konfiguration h nach einem Takt berechnet.

$$\begin{aligned} h' &= \delta_h(h) \\ h^{t+1} &= \delta_h(h^t) \end{aligned}$$

Dabei sind nicht immer alle Zustände wohldefiniert. Wir betrachten Schaltkreise F_i deren Eingangssignale aus Registern R_j oder RAMs S_k kommen. Die Schaltkreise berechnen zustandsabhängige Ausgangssignale $F_i(h)$. Diese sind in Konfiguration h wohldefiniert, falls die Inputs in die Schaltkreise wohldefiniert sind. Diese Eigenschaft ist wichtig, da eben jene outputs wieder als Eingangssignale $S_{Din}(h), R_{in}(h), R_{ce}(h), S_{ad}(h), S_w(h)$ zu den Speicherelementen zurückgeleitet werden können. Weiterhin sind die Ausgänge von Registern immer wohldefiniert. Dies ist für RAMs nicht immer der Fall.

2.2 Spezifikation

Wir wollen nun die Semantik der Speicherelemente spezifizieren. Diese ergibt sich aus der allgemeinen Definition der Folgezustände $h'.R$ bzw. $h'.S$.

2.2.1 Register

Es gilt für $h.R \in \{0,1\}^n$:

$$h'.R = \begin{cases} h.R & : \quad R_{ce}(h) = 0 \\ R_{in}(h) & : \quad \text{sonst} \end{cases}$$

Das Signal $R_{in}(h)$ beschreibt den Wert der in der Konfiguration h am Eingang von R anliegt. R_{ce} heißt *clock enable*-Signal von R und signalisiert, dass der im aktuellen Takt anliegende Wert im Register gespeichert werden soll. Falls kein neuer Wert ins Register geschrieben wird, bleibt der Inhalt unverändert.

2.2.2 Speicher

In Speichern können im Gegensatz zu einfachen Registern mehrere Werte gespeichert werden. Deshalb werden auch Adress-Signale benötigt um verschiedene Speicherbereiche auszuwählen. Ein gewöhnlicher RAM S verfügt über die folgenden Schnittstellen:

- S_{Din} - Dateneingang
- S_{Dout} - Datenausgang
- S_{ad} - Adress-Signal (**ad**dress)
- S_w - Schreib-Signal (**w**rite)

Der Speicher stellt eine Abbildung $h.S : \{0,1\}^a \longrightarrow \{0,1\}^d$ dar. Grafik 2.1 zeigt den Aufbau des Speichers in schematischer Art und Weise. Die Funktionsweise ist für ein $x \in \{0,1\}^a$ folgendermaßen definiert:

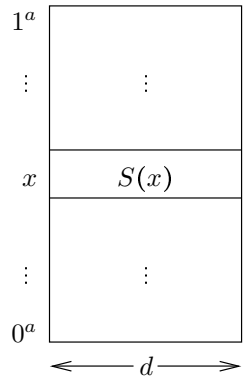


Abbildung 2.1: Aufbau des Speichers

$$
\begin{aligned}
S_{Dout}(h) &= h.S(S_{ad}(h)) \\
h'.S(x) &= \begin{cases} S_{Din}(h) & : \quad S_w(h) \wedge (S_{ad}(h) = x) \\ h.S(x) & : \quad \text{sonst} \end{cases}
\end{aligned}
$$

Die Festlegung, dass am Ausgang verzögerungsfrei die Inhalte des RAMs entsprechend dem angelegten Adress-Signal angezeigt werden, ist problematisch. Falls durch einen Designfehler zum Beispiel Datenausgang und Adresseingang kurzgeschlossen werden, so würde sich das Ausgangssignal zusammen mit dem Adress-Signal ständig ändern. Der Zustand der Hardware wäre dann nicht mehr wohldefiniert. Solche Situationen kommen allerdings nicht nur durch fehlerhaftes Schaltungsdesign zustande. Auch in der späteren Realisierung kann es durch bestimmte Komponenten- und Bus-Anordnungen im Layout zu solchen Kurzschlüssen kommen. Der Grund hierfür ist elektromagnetische Induktion, die zwischen benachbarten Leitern auftritt. Man spricht von *cross talk* zwischen den Signalen. Moderne Layout-Routinen beachten diese Phänomene und sorgen dafür, dass Address- und Datenbusse ausreichend voneinander isoliert sind.

2.3 Formale Hardware-Beweise

Mit dem gegebenen Definitionen lässt sich formal über getaktete Schaltungen argumentieren.
Beispiel: Abbildung 2.2 zeigt einen getakteten Schaltkreis. Darin wird das $reset$-Signal verwendet. Das $reset$-Signal sorgt in Hardware-Systemen dafür, dass zu Beginn einer Hardware-Rechnung ein wohldefinierter Ausgangszustand h^0 erreicht wird. Dabei bezeichnen wir Folgen von Hardware-Zuständen

$(h^{-1}, h^0, h^1, h^2, \ldots)$ als Hardware-Rechnungen, falls gilt:

$$\forall i : h^{i+1} = \delta_h(h^i)$$

Eigentlich müsste also die Übergangsfunktion, sowie alle anderen Hardwarefunktionen vom $reset$-

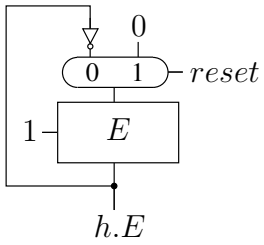


Abbildung 2.2: clocked circuit

Signal abhängen: $h' = \delta_h(h, reset)$. Im Folgenden gehen wir aber davon aus, dass jedes System zu Beginn zurückgesetzt wird, und $reset$ danach inaktiv bleibt. Wir können daher bei der Betrachtung von Takten $t > 0$ die Abhängigkeit von $reset$ implizit machen und Schreibarbeit sparen.

$$\begin{aligned} reset(h^{-1}) &= 1 \\ \forall t \geq 0 : reset(h^t) &= 0 \end{aligned}$$

Der einzige Reset findet also vor h^0 statt. Daher wird das Register E mit 0 initialisiert. In allen folgenden Takten wird der Inhalt von E fortwährend invertiert.
Behauptung:

$$\forall i \quad : \quad h^i.E = (i \bmod 2)$$

Beweis: durch vollständige Induktion über i
Induktionsanfang: $i = 0$

$$\begin{aligned} h^0.E &= E_{in}(h^{-1}) && (E_{ce} = 1) \\ &= 0 && (reset(h^{-1}) = 1) \\ &= (0 \bmod 2) \end{aligned}$$

Induktionvoraussetzung: $h^i.E = (i \bmod 2)$
Induktionsschritt: $i \to i + 1$

$$\begin{aligned} h^{i+1}.E &= \overline{E_{in}(h^i)} && (E_{ce} = 1) \\ &= \overline{h^i.E} && (reset(h^{-1}) = 0) \\ &= \overline{(i \bmod 2)} && \text{(Induktionsvoraussetzung)} \\ &= (i + 1 \bmod 2) && \square \end{aligned}$$

2.4 RAM-Implementierung

Wir betrachten nun die Konstruktion eines RAMs. Ein wichtiger Bestandteil sind dabei Decoder.

Definition 2.2 *(n-Decoder) Ein a-Decoder ist ein Schaltkreis mit Eingang $x \in \{0,1\}^a$ und Ausgang $y \in \{0,1\}^{2^a}$. Er berechnet die Unärdarstellung von $\langle x \rangle$. Es gilt:*

$$y_i = 1 \leftrightarrow \langle x \rangle = i \qquad i \in \{0, \ldots, 2^a - 1\}$$

Abbildung 2.3 zeigt die schematische Darstellung eines Decoders.

Ein Decoder wird wie folgt implementiert.

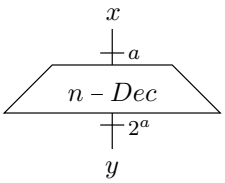
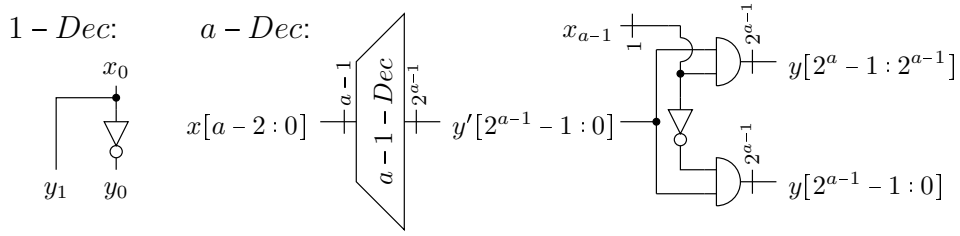


Abbildung 2.3: a-Decoder



Korrektheitsbeweis: Sei $\langle x \rangle = i \in \{0, \ldots, 2^a - 1\}$, dann soll gelten:

$$\begin{array}{ll} & y_i = 1 \\ \forall j \neq i: & y_j = 0 \end{array}$$

Für den $1 - Dec$ gilt:

$$\begin{array}{lll} x_0 = 0 & \Rightarrow & y_0 = \overline{x_0} = 1 \wedge y_1 = x_0 = 0 \quad \checkmark \\ x_0 = 1 & \Rightarrow & y_0 = \overline{x_1} = 0 \wedge y_1 = x_0 = 1 \quad \checkmark \end{array}$$

Für den $a - Dec$ unterscheiden wir zwei Fälle:
Fall 1: $x[a-1] = 0 \Rightarrow i < 2^{a-1}$

$$\begin{array}{rcll} y[2^a - 1 : 2^{a-1}] & = & 0^{2^{a-1}} & \text{(Konstruktion, } \wedge\text{)} \\ y[2^{a-1} - 1 : 0] & = & y' & \text{(Konstruktion, } \wedge\text{)} \\ \forall i \in [2^{a-1} - 1 : 0]. \quad y_i = 1 & \Leftrightarrow & y'_i = 1 & \\ & \Leftrightarrow & \langle x[a-2:0] \rangle = i & \text{(Induktionsvoraussetzung)} \\ & \Leftrightarrow & \langle 0x[a-2:0] \rangle = i & \text{(zero extension)} \\ & \Leftrightarrow & \langle x[a-1:0] \rangle = i & (x[a-1] = 0) \quad \checkmark \end{array}$$

Fall 2: $x[a-1] = 1 \Rightarrow i \geq 2^{a-1}$

$$\begin{array}{rcll} y[2^a - 1 : 2^{a-1}] & = & y' & \text{(Konstruktion, } \wedge\text{)} \\ y[2^{a-1} - 1 : 0] & = & 0^{2^{a-1}} & \text{(Konstruktion, } \wedge\text{)} \\ \forall i \in [2^a - 1 : 2^{a-1}]. \quad y_i = 1 & \Leftrightarrow & y'_{i-2^{a-1}} = 1 & \text{(Indexverschiebung)} \\ & \Leftrightarrow & \langle x[a-2:0] \rangle = i - 2^{a-1} & \text{(Induktionsvoraussetzung)} \\ & \Leftrightarrow & 2^{a-1} + \langle x[a-2:0] \rangle = i & (+2^{a-1}) \\ & \Leftrightarrow & \langle 1x[a-2:0] \rangle = i & \text{(Zerlegungslemma)} \\ & \Leftrightarrow & \langle x[a-1:0] \rangle = i & (x[a-1] = 1) \quad \square \end{array}$$

Ein RAM wird nun wie in Abbildung 2.4 gezeigt konstruiert. Das Adresssignal wird dekodiert, so dass für jedes Register ein Steuersignal existiert. Von diesen ist immer genau eines 1 und der Rest 0. Mit dem Signal wird der ce-Eingang des jeweiligen Registers für das Schreibsignal w „freigeschaltet“. Außerdem wird darüber selektiert, welcher Registerinhalt auf den Ausgang $Dout$ des RAMs geschaltet wird. Dies

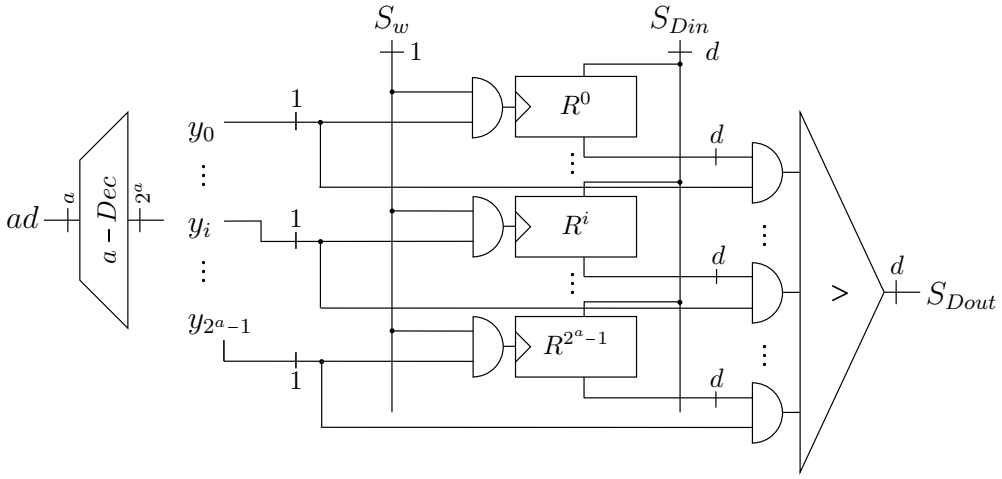


Abbildung 2.4: Implementierung eines RAMs

geschieht indem jedes Ausgangsbit mit dem entsprechenden Steuersignal verundet wird.

$$
\begin{array}{lrcl}
j \in \{0,\ldots,2^a-1\}: & Q^j & \in & \{0,1\}^d \\
& Q^j & = & R^j \wedge y_j \\
& & = & \begin{cases} R^j & : \quad y_j = 1 \\ 0^d & : \quad \text{sonst} \end{cases} \\
& & = & \begin{cases} R^{\langle ad \rangle} & : \quad j = \langle ad \rangle \\ 0^d & : \quad \text{sonst} \end{cases}
\end{array}
$$

Diese Q^j mit $j \in \{0,\ldots,2^a-1\}$ werden dann über einen d Bit breiten $\vee$-tree zum Ausgang geleitet. Dieser besteht aus d parallel geschalteten $\vee$-trees mit jeweils 2^a Eingängen.

$$
\begin{array}{rcl}
Dout_k & = & \bigvee_{j=0}^{2^a-1} Q_k^j \\
& = & R_k^{\langle ad \rangle}
\end{array}
$$

Als Registerbank der DLX benötigen wir einen *3-Port-RAM*, wie in Abbildung 2.5 dargestellt. Dieser hat im Unterschied zu einem normalen RAM drei Adresseingänge und zwei Datenausgänge und wird wie folgt definiert:

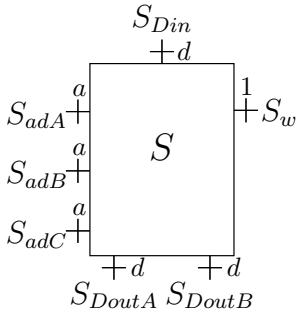


Abbildung 2.5: 3-Port-RAM

$$
\begin{aligned}
h.S &: \{0,1\}^a \longrightarrow \{0,1\}^d \\
S_{DoutA}(h) &= h.S(S_{adA}(h)) \\
S_{DoutB}(h) &= h.S(S_{adB}(h)) \\
h'.S(x) &= \begin{cases} S_{Din}(h) & : \quad S_w(h) \wedge (S_{adC}(h) = x) \\ h.S(x) & : \quad \text{sonst} \end{cases}
\end{aligned}
$$

S_{adC} bestimmt also lediglich ein Schreibziel, während S_{adA} und S_{adB} anzeigen, welche beiden Speicherbereiche gelesen werden sollen.

Kapitel 3

Bau eines einfachen Prozessors

Nun soll ein einfacher DLX-Prozessor gebaut werden. Die folgenden Schritte sind dazu notwendig.

- Spezifikation des Rechners - Was ist die gewünschte Funktionsweise?
- Konstruktion der Hardware - Wie ist der Prozessor implementiert?
- Beweis der Korrektheit - Entspricht die Hardware der Spezifikation?

3.1 Spezifikation

Die Spezifikation eines Prozessors wird auch ISA (**I**nstruction **S**et **A**rchitecture) genannt. Sie beschreibt den vorliegenden Instruktionssatz und beinhaltet:

- für Benutzer sichtbare Datenstrukturen (Hardware kann unsichtbare Komponenten enthalten)
- ausführbare Operationen und deren Format (Syntax der Instruktionen)
- Effekt der Operationen auf die benutzersichtbaren Datenstrukturen (Semantik der Instruktionen)

Die Syntax eines Maschinenbefehls lautet zum Beispiel, dass er durch einen Bitstring $\in \{0,1\}^{32}$ dargestellt wird. Die Semantik des Befehls ergibt sich aus einer Konfiguration c der für den Programmierer sichtbaren Register und RAMs und einer Übergangsfunktion $\delta(c) = c'$, die die Auswirkungen des Befehls durch den Zustand nach Ausführung der Instruktion definiert.

3.1.1 Konfiguration

Unser einfacher DLX-Prozessor (vereinfachte **MIPS**-Architektur - **M**icroprocessor without **i**nterlocked **p**ipeline **s**tages) besteht aus den Komponenten

- GPR - general purpose register file (Zusammenfassung von Registern zum Zwischenspeichern von Daten)
- RAM - random access memory (Arbeitsspeicher, der Daten und Instruktionen enthält und gelesen/geschrieben werden kann)
- PC - program counter (Befehlszähler der auf die auszuführenden Instruktionen im Speicher verweist)

Abbildung 3.1 veranschaulicht den grundsätzlichen Aufbau. Es wird ein mathematisches Modell aufgestellt, mit dem die Arbeitsweise des Prozessors spezifiziert werden kann. Dazu definiert man Konfigurationen $c = (c.pc, c.gpr, c.m)$, die einen temporären Zustand der Maschine beschreiben. Für die Komponenten gilt:

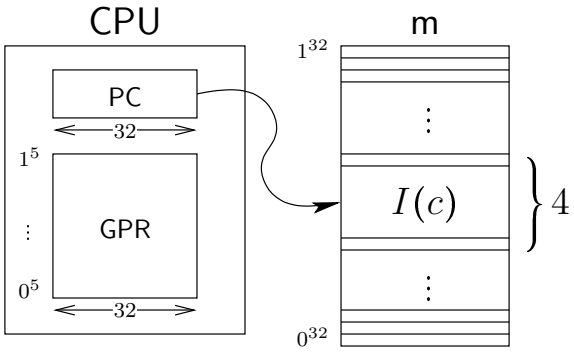


Abbildung 3.1: Grober Aufbau der DLX

- $c.pc \in \{0,1\}^{32}$ - program counter
- $c.m : \{0,1\}^{32} \to \{0,1\}^8$ - memory
  - Byte-adressierbarer Speicher
  - 32-Bit Adressen
  - $c.m(x)$ beschreibt den Inhalt der Speicherzelle mit Adresse x
- $c.gpr : \{0,1\}^5 \to \{0,1\}^{32}$ - general purpose register
  - 32 Register, jedes 32 Bit breit
  - $c.gpr(x)$ beschreibt den Inhalt von Register x

Für den Speicherzugriff verwenden wir folgende Notation, um elegant über mehrere zusammenhängende Bytes im Speicher argumentieren zu können:

$$m_d(y) = m(y +_{32} d -_{32} 1_{32}) \circ \ldots \circ m(y +_{32} 1_{32}) \circ m(y),$$

wobei $u_{32} \equiv bin_{32}(u)$ der 32 Bit Binärdarstellung von $u \in \{0, \ldots, 2^{32} - 1\}$ entspricht. So gilt zum Beispiel $4_{32} = 0^{29}100$. Abbildung 3.2 zeigt die addressierten d Bytes im Speicher.

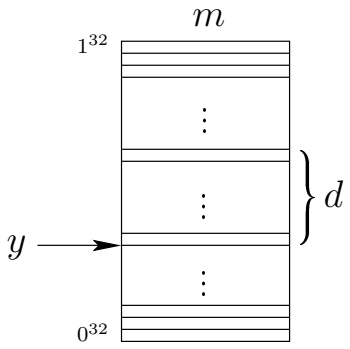


Abbildung 3.2: Adressierung von d Bytes im Speicher

3.1.2 Instruktionen

Beim Übergang von Konfiguration c nach c' wird diejenige Instruktion $I(c)$ ausgeführt deren Adresse der program counter $c.pc$ enthält

$$I(c) = m_4(c.pc)$$

Instruktionen werden also durch 32-Bit Konstanten im Speicher repräsentiert. Wir definieren zum Dekodieren dieser Bitmuster Prädikate p' über Instruktionen $i \in \{0,1\}^{32}$.

$$p' : \{0,1\}^{32} \longrightarrow \{0,1\}$$

Diese Prädikate lassen sich später durch Schaltfunktionen realisieren. Um Schreibarbeit zu sparen, definieren wir für jedes Instruktions-Prädikat p' auch ein entsprechendes Prädikat p über die Konfiguration c der DLX-Maschine nach folgendem Schema.

$$p(c) = p'(I(c))$$

Je nach Struktur unterscheidet man drei Typen von Instruktionen – *I-type, R-type und J-type*. Dabei sind die obersten 6 Bit, der sogenannte opcode $opc'(I) = I[31:26]$, von entscheidender Bedeutung. Man definiert Prädikate, die den jeweiligen Typ der Instruktion I anzeigen.

- $Rtype'(I) \leftrightarrow opc(I) = 0^6$ *(register type)*
- $Jtype'(I) \leftrightarrow opc(I) \in \{000010, 000011, 111110, 111111\}$ *(jump type)*
- $Itype'(I) \leftrightarrow \text{sonst}$ *(immediate type)*

Genauso wie bei den Prädikaten gilt für Funktionen f, die nur von der aktuellen Instruktuion abhängen, die Notation $f(c) = f'(I(c))$, also zum Beispiel $opc(c) = opc'(I(c))$.
Abbildung 3.3 stellt die Struktur der drei Instruktionenstypen dar und definiert die Bedeutung der einzelnen Abschnitte innerhalb einer Instruktion.

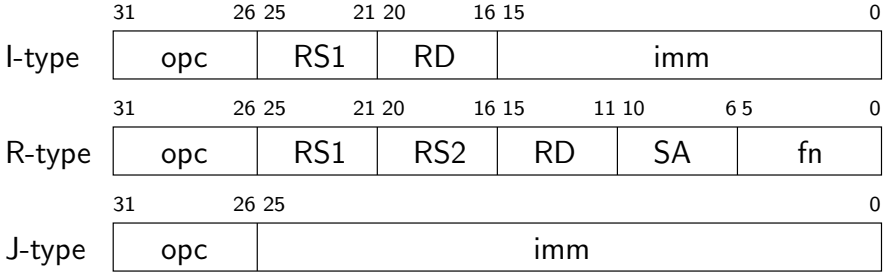


Abbildung 3.3: Typen von Instruktionen

Diese lassen sich auch durch folgende Funktionen auf Instruktionen I beschreiben:

- $RS1'(I) = I[25:21]$ - register source 1
- $RS2'(I) = I[20:16]$ - register source 2 (nur bei Rtype-Instruktionen)
- $RD'(I) = \begin{cases} I[15:11] & : \quad Rtype'(I) \\ I[20:16] & : \quad \text{sonst} \end{cases}$ - *meistens* register destination (nicht bei store word)
- $SA'(I) = I[10:6]$ - shift amount (nur bei Rtype-Instruktionen)
- $fn'(I) = I[5:0]$ - function (nur bei Rtype-Instruktionen)
- $imm'(I) = \begin{cases} I[25:0] & : \quad Jtype'(I) \\ I[15:0] & : \quad Itype'(I) \end{cases}$ - immediate-Konstante
- $sxtimm'(I) = \begin{cases} (I[25])^6 I[25:0] & : \quad Jtype'(I) \\ (I[15])^{16} I[15:0] & : \quad \text{sonst} \end{cases}$ - sign-extended immediate-Konstante

Die immediate-Konstante muss für $signed$-Berechnungen zur 32-Bit Konstante $sxtimm$ erweitert werden.

Lemma 3.1 *(sign-extended immediate-Konstante) Mit den obigen Definitionen für $imm(c)$ und $sxtimm(c)$ gilt*

$$[sxtimm(c)] = [imm(c)].$$

Hierbei wird sign extension in 32-Bit Two's-Complement-Arithmetik benutzt.

Für $unsigned$-Berechnungen wird imm mit Nullen erweitert. Die zu verwendende Konstante $co'(I)$ ergibt sich zu:

$$co'(I) = \begin{cases} sxtimm'(I) & : \quad u'(I) = 0 \\ 0^{16}imm'(I) & : \quad u'(I) = 1 \end{cases}$$

Dabei signalisiert $u'(I)$, dass eine Instruktion eine $unsigned$-Berechung durchführt.

$$u'(I) = Rtype'(I) \wedge I[3:2]I[0] = 001 \vee Itype'(I) \wedge I[29:28]I[26] = 001$$

Man beachte dass u nur für die Erzeugung des ovf-Bits und die Erweiterung einer immediate-Konstante bei Addition und Subtraktion relevant ist. Das berechnete Ergebnis der Operation ist modulo 2^{32} identisch.
Der vollständige DLX-Instruktionssatz findet sich in Anhang A.

3.1.3 Übergangsfunktion

Um den Effekt von Instruktionen zu beschreiben, muss man die Übergänge von aufeinanderfolgenden Konfigurationen betrachten. Es wird die Übergangs/"next state"-Funktion

$$\delta_D(c) = c'$$

definiert, die die Ausführung einer Instruktion auf einen Zustand c beschreibt und den Folgezustand c' liefert. Dabei muss zwischen den verschiedenen möglichen Instruktionen unterschieden werden. Wir betrachten drei Klassen von Instruktionen:

- ls - load/store-Operationen, die auf den Speicher lesen oder manipulieren
- $comp$ - compute-Operationen, die ALU-Funktionen ausführen
- $control$ - Kontrolloperationen, die den Programmfluss steuern

3.1.4 load word & store word

Bei „load word"- bzw. „store word"-Instruktionen wird auf den Speicher zugegriffen. Dieser Zugriff geschieht an einer effektiven Adresse, die sich aus dem Inhalt von $RS1$ und der imm-Konstante zusammensetzt, und umfasst 4 Bytes (= 1 Word). Im Falle einer „load word"-Instruktion wird das Prädikat $lw(c)$ wahr:

$$lw(c) \leftrightarrow opc(c) = 100011$$

Die effektive Adresse ist die Summe von $RS1$ und der sign-extended immediate-Konstante:

$$ea(c) = c.gpr(RS1(c)) +_{32} sxtimm(c)$$

Der Effekt der load-Instruktion lässt sich dann wie folgt beschreiben:

$$c'.gpr(RD(c)) = c.m_4(ea(c))$$

Der program counter springt zur nächsten Instruktion und sämtliche anderen Register sowie der Speicher bleiben unverändert:

$$\begin{aligned} c'.pc &= c.pc +_{32} 4_{32} \\ \forall x \in \{0,1\}^5, x \neq RD(c) : c'.gpr(x) &= c.gpr(x) \\ c'.m &= c.m \end{aligned}$$

Bei „store word"-Instruktionen ($sw(c) \leftrightarrow opc(c) = 101011$) hingegen wird der Speicher verändert:

$$c'.m_4(ea(c)) = c.gpr(RD(c))$$

$RD(c)$ beschreibt in diesem Fall ein Quell- und kein Zielregister. Der pc wird wieder um 4 erhöht und alle übrigen Speicherzellen und Register bleiben konstant.

$$\begin{aligned} c'.pc &= c.pc +_{32} 4_{32} \\ \forall x \in \{0,1\}^{32}, x \notin \{ea(c), \ldots, ea(c) +_{32} 3_{32}\} : c'.m(x) &= c.m(x) \\ c'.gpr &= c.gpr \end{aligned}$$

3.1.5 ALU-Instruktionen

Die meisten übrigen I-type und R-type Instruktionen stellen arithmetische oder logische Operationen dar. Hierfür wird die ALU verwendet, was durch die folgenden Prädikate signalisiert wird:

$$\begin{aligned} compimm(c) &\leftrightarrow I(c)[31:30] = 01 \\ comp(c) &\leftrightarrow Rtype(c) \wedge I(c)[5:4] = 00 \end{aligned}$$

Bei R-type ALU-Instruktionen ($comp(c)$) wird keine immediate-Konstante sondern ein zweites Register ($RS2(c)$) als rechter Operand verwendet. Linker und rechter Operand der ALU-Operation sind dann wie folgt definiert:

$$\begin{aligned} lop(c) &= c.gpr(RS1(c)) \quad \text{(left operand)} \\ rop(c) &= \begin{cases} c.gpr(RS2(c)) & : \quad Rtype(c) \\ co(c) & : \quad \text{sonst} \end{cases} \quad \text{(right operand)} \end{aligned}$$

Dazu kommt der Funktionscode für die ALU, der die zu berechnende $aluop$ bestimmt:

$$aluf(c) = \begin{cases} I(c)[3:0] & : \quad Rtype(c) \\ I(c)[29:26] & : \quad \text{sonst} \end{cases}$$

Abbildung 3.4 stellt eine ALU-Operation schematisch dar. Das Ergebnis $alures(c)$ wird in general purpose Register $RD(c)$ gespeichert. Alle anderen Komponenten bleiben unverändert.

$$\begin{aligned} c'.gpr(RD(c)) &= alures(c) = aluop(lop(c), rop(c), aluf(c)) \\ \forall x \in \{0,1\}^5, x \neq RD(c) : c'.gpr(x) &= c.gpr(x) \\ c'.m &= c.m \\ c'.pc &= c.pc +_{32} 4_{32} \end{aligned}$$

3.1.6 Kontrollinstruktionen

Die verbleibenden Instruktionen sind Kontrollinstruktionen, also branches und jumps.

$$\begin{aligned} jump(c) &= j(c) \vee jal(c) \vee jr(c) \vee jalr(c) \\ branch(c) &= beqz(c) \vee bnez(c) \end{aligned}$$

Diese ermöglichen die Kontrolle des Programmflusses, indem sie den pc durch Addieren von Registerinhalten oder immediate-Konstanten manipulieren. Bei branches muss dafür eine Bedingung an ein

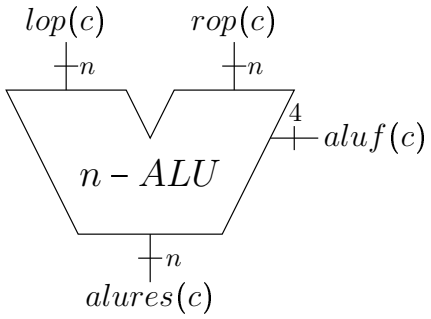


Abbildung 3.4: Verwendung der ALU durch ALU-Instruktion

angegebenes Register erfüllt sein, damit der "Sprung" im Programmcode stattfindet. Im Falle, dass in der Konfiguration c solche Instruktionen ausgeführt werden, gilt das Prädikat

$$control(c) \leftrightarrow I(c)[31:28] = 1101 \vee I(c)[31:27] = 00001.$$

Gilt es nicht, so wird der pc grundsätzlich um 4 erhöht, also auf die „nächste sequentielle Instruktion" gesetzt:

$$\overline{control(c)} \Rightarrow c'.pc = c.pc +_{32} 4_{32}$$

Der pc wird individuell verändert, wenn ein Sprung vorliegt oder ein branch ausgeführt wird:

$$\begin{aligned} jbtaken(c) &= jump(c) \vee btaken(c) \\ btaken(c) &= beqz(c) \wedge (c.gpr(RS1(c)) = 0^{32}) \vee bnez(c) \wedge (c.gpr(RS1(c)) \neq 0^{32}) \end{aligned}$$

Im Falle von $beqz$ (*branch equal zero*) muss Register $RS1(c)$ gleich Null sein, im Falle von $bnez$ (*branch not equal zero*) ungleich Null sein, damit gesprungen wird. Das Sprungziel wird dann je nach Instruktion per Addition der immediate-Konstante zum pc oder per Zuweisung eines Registerinhalts bestimmt:

$$btarget(c) = \begin{cases} c.pc +_{32} sxtimm(c) & : \quad branch(c) \vee j(c) \vee jal(c) \\ c.gpr(RS1(c)) & : \quad \text{sonst} \end{cases}$$

Im ersten Fall sprechen wir von einem *relativen*, ansonsten von einem *absoluten* Sprung.
Der Effekt der Kontrollinstruktion ($control(c)$ gilt) ist dann:

$$c'.pc = \begin{cases} btarget(c) & : \quad bjtaken(c) \\ c.pc +_{32} 4_{32} & : \quad \text{sonst} \end{cases}$$

m und gpr bleiben unverändert. Nur bei „jump and link"-Instruktionen muss der „nächste sequentielle pc" in Register 31 (= $\langle 1^5 \rangle$) gespeichert werden:

$$jal(c) \vee jalr(c) \Rightarrow c'.gpr(1^5) = c.pc +_{32} 4_{32}$$

3.2 Assemblersprache

Um das Schreiben von Programmen zu erleichtern, führen wir eine Assemblersprache ein, die eine Abfolge von Instruktionen in einem besser lesbaren Format beschreibt. Syntaktisch bestehen die Befehle aus:

- mnemonics - die schon von den Prädikaten bekannten Kürzel zu jeder Instruktion ($lw, sw, beqz, \ldots$)
- die Parameter ($RS1, RD, imm, \ldots$) im Dezimalformat in einer bestimmten Reihenfolge

Die Reihenfolge der Parameter wird teilweise entsprechend den zugehörigen Feldern in den Instruktionen der Maschinensprache definiert, also erst $RS1$, dann $RS2/RD$ und dann imm/RD. Zur besseren Lesbarkeit und einem intuitiven Verständnis verwendet man allerdings oft eine abweichende Reihenfolge, in der RD vorangestellt wird, also $mnemonic\ RD\ RS1\ RS2/imm$, so wie es auch in den Übungen gefordert wird. Die folgenden Programme sind in letzterer Schreibweise verfasst. Eine „load word"-Instruktion zum Beispiel, die den Inhalt des Speichers an der effektiven Adresse $c.gpr(5) +_{32} 20_{32}$ in Register 3 lädt, würde dann wie folgt geschrieben werden:

$$lw\ 3\ 5\ 20$$

Die den Assemblerbefehlen zugehörigen Instruktionen liegen im Speicher in einem Bereich der für Programme reserviert ist. Er beginnt bei Adresse 0^{32} und reicht bis einschließlich Byte $D - 1$ für ein $D = 4k, k \in \mathbb{N}^+$. Von hier wird auch wegen $c^0.pc = 0^{32}$ die erste Instruktion geladen. Abbildung 3.5 verdeutlicht die Speicheraufteilung.

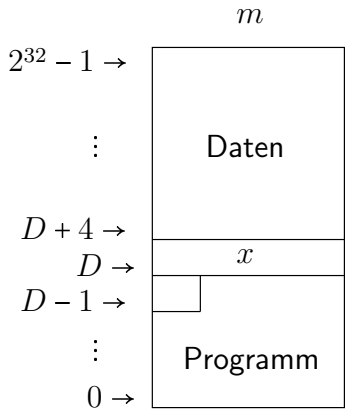


Abbildung 3.5: Speicheraufteilung (memory map) zwischen Daten und Instruktionen

Beispiel: Es soll ein Programm geschrieben werden, so dass für $m_4(D) = x$ gilt: $m_4(D + 4) = x + 1$. Dabei ist D die Startadresse des Datenbereichs. Wir müssen zunächst Register 0 des gpr auf Null setzten damit wir auf diese nützliche Konstante zurückgreifen können.

```
 0: xor  0 0 0     // GPR(R0)=GPR(R0) XOR GPR(R0)=0
 4: lw   1 0 D     // ea=GPR(R0)+imm=imm
 8: addi 1 1 1     // GPR(R1)=GPR(R1)+1
12: sw   1 0 D+4   // m_4(D+4)=GPR(R1)
```

Für die immediate-Konstante D gilt $D \in T_{16}$ und damit $D \leq 2^{15} - 1$. Daraus folgt, dass die Größe von Programmen beschränkt ist. Es können nicht beliebig viele Instruktionen verwendet werden. Ein weiteres Programm verdeutlicht den Vorteil effektiver Adressierung.

Beispiel: Das geforderte Verhalten ist in Abbildung 3.6 dargestellt. Es soll nach Ausführung des Programms gelten: $gpr(2) = y$ mit $y = m_4(x)$. Wir dereferenzieren also einen Pointer an Adresse D der auf den Wert y an Adresse x verweist.

```
0: xor  0 0 0     // GPR(R0)=GPR(R0) XOR GPR(R0)=0
4: lw   1 0 D     // GPR(R1)=m_4(0+D)=x
8: lw   2 1 0     // GPR(R2)=m_4(GPR(R1)+0)=m_4(x)=y
```

Wir sehen, warum man von „general purpose" Registern spricht. In ihnen werden sowohl Daten, als auch Adressen gespeichert (Indexregister). Das bedeutet aber auch, dass Daten und Adressen im selben Speicher gehalten werden. In diesem Fall spricht man von einer *von Neumann*-Architektur. Existieren

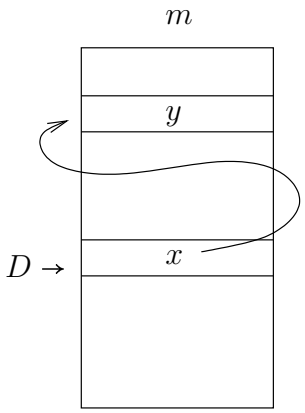


Abbildung 3.6: Erläuterung zum zweiten Beispielprogramm

verschiedene Speicher für Daten und Adressen, so liegt eine *Harvard*-Archtiektur vor. Frühe Maschinen kannten keine Indexregister und hatten so keine Möglichkeit, in Abhängigkeit der Daten auf den Speicher zuzugreifen. Das oben beschriebene Problem war somit nur mit Hilfe von selbstmodifizierendem Code zu lösen. Das Fehlen von Indexregistern kann simuliert werden, indem man bei der effektiven Adressierung stets R0 als RS1 wählt.

Es folgt ein weiteres Beispiel für eine Rechnung auf der DLX-Maschine.
Beispiel: Es soll die Summe $s = \sum_{i=1}^{n} i$ berechnet werden. Abbildung 3.7 zeigt die Belegung des Speichers.

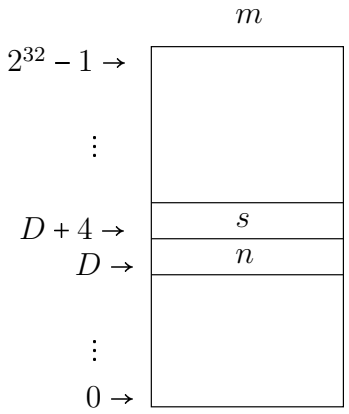


Abbildung 3.7: Speicherbelegung des Beispielprogramms

Die Summe soll in umgekehrter Reihenfolge $n + (n-1) + \ldots + 1$ berechnet werden. Das Programm in Pseudocode lautet:

```
s:=n;
do
n=0 ? goto *
n:=n-1;
s:=s+n;
loop;
*: end
```

Nach dem i-ten Durchlauf der Schleife gilt:

$$s = n + \underbrace{(n-1) + \ldots + (n-i)}_{i \text{ Durchlaufe}}$$
$$m_4(D) = n - i$$

Dabei rührt der Anfangswert n von der Initialisierung von s her. Eine mögliches Assemblerprogramm mit der geforderten Funktion kann dann so aussehen:

```
 0: xor  0 0 0     // GPR(R0)=GPR(R0) XOR GPR(R0)=0
 4: lw   1 0 D
 8: sw   1 0 D+4
12: lw   1 0 D
16: beqz 1   28  // goto *
20: subi 1 1 1
24: sw   1 0 D
28: lw   2 0 D+4
32: add  3 1 2
36: sw   3 0 D+4
40: j      -28 // loop
44: ...          // *
```

Hier müssen natürlich die entsprechenden immediate-Konstanten für D und $D+4$ eingesetzt werden. Weiterhin lassen sich für das Beispiel die aufeinander folgenden Zustände, die die DLX-Maschine anhand der Instruktionsfolge durchläuft, mit Hilfe der Übergangsfunktion formalisieren.

- c^0 - Der Anfangszustand (allgemein)
- $c^{j+1} = \delta_D(c^j)$ - Der nachfolgende Zustand von c^j (allgemein)
- c^2 - Hier der Zustand nach dem 0-ten Durchlauf der Schleife (noch kein Durchlauf)
- $c^{2+7 \cdot i}$ - Hier der Zustand nach dem i-ten Durchlauf der Schleife

$(c^0, c^1, c^2, \ldots)$ mit $c^{j+1} = \delta_D(c^j)$ nennt man eine *Rechnung*.

3.3 Implementierung

Nun wollen wir die DLX implementieren. Parallel dazu beweisen wir die Korrektheit unserer Konstruktion.

3.3.1 Hardware-Konfiguration

Die Hardware soll den DLX-Instruktionssatz realisieren, deshalb werden die Konfigurationen wie folgt definiert.

Definition 3.1 *(Hardware-Konfiguration) Eine Konfiguration h beschreibt den Inhalt der DLX-Hardwarekomponenten zu einem bestimmten Zeitpunkt. Dabei ist:*

- $h = (h.ex, h.pc, h.IR, h.gpr, h.m)$ - *Hardwarekonfiguration der DLX*
- $h.ex \in \{0,1\}$ - *1-Bit Register für aktuellen Modus (fetch/execute, siehe Abbildung 2.2)*
- $h.pc \in \{0,1\}^{32}$ - *32-Bit Register für den Programm Counter (siehe Abbildung 3.10)*
- $h.IR \in \{0,1\}^{32}$ - *32-Bit Register für die aktuelle Instruktion*
- $h.gpr : \{0,1\}^5 \longrightarrow \{0,1\}^{32}$ - *Multiport Register File (siehe Übung, 3-Port-RAM)*

- $h.m : \{0,1\}^{30} \longrightarrow \{0,1\}^{32}$ - *Speicher (RAM, siehe Abbildung 2.1)*

Man beachte, dass der Speicher $h.m$ der Hardware im Gegensatz zu $c.m$ der DLX *word-addressed* ist. Das bedeutet, dass nur ganze Wörter aus dem Speicher gelesen werden können. Abbildung 3.8 zeigt die Aufteilung des memory.

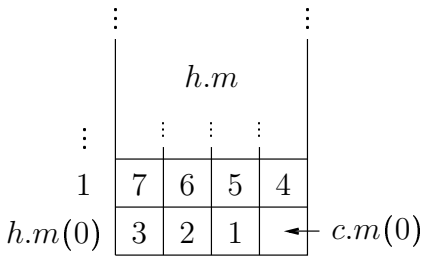


Abbildung 3.8: wortadressierter Speicher der Hardware

Ein Wort wird in vier Bytes unterteilt, die in aufsteigender Reihenfolge im Speicher angeordnet werden. Diese Anordnung nennt man *little endian byte order*. Die Anordnung der Bytes im Speicher spielt vor allem dann eine Rolle, wenn Instruktionen auf Objekte im Speicher zugreifen, deren Bitbreite kleiner ist als die eines Wortes (*halfwords, bytes*). Dann werden auch in der Implementierung zusätzliche Schaltkreise (z.B. Shifter) benötigt, um Daten in Register- und Speicherworte einzubetten.

3.3.2 Korrektheit, Simulationsrelation und Simulationssatz

Begleitend zur Implementierung der DLX, soll im Folgenden bewiesen werden, dass diese der Spezifikation entspricht. Dies geschieht mit Hilfe eines Simulationsbeweises, der die Zustände beider Maschinen betrachtet und argumentiert, dass sie nach jedem Ausführungsschritt äquivalent sind. Zur besseren Übersicht soll die Definitionen der Spezifikation hier nochmals aufgelistet werden. Es gilt:

$$\delta_D(c) = c' \tag{3.1}$$

$$c'.m_4(ea(c)) = c.gpr(RD(c)), \quad \text{wenn } sw(c) \tag{3.2}$$

$$c'.m(x) = c.m(x), \quad \text{wenn } \overline{sw(c)} \vee (sw(c) \wedge (x \notin \{ea(c), \ldots, ea(c) +_{32} 3_{32}\})) \tag{3.3}$$

$$c'.pc = \begin{cases} btarget(c) & : bjtaken(c) \\ c.pc +_{32} 4_{32} & : sonst \end{cases} \tag{3.4}$$

$$c'.gpr(x) = \begin{cases} c.m_4(ea(c)) & : lw(c) \wedge (x = RD(c)) \\ alures(c) & : (comp(c) \vee compimm(c)) \wedge (x = RD(c)) \\ c.pc +_{32} 4_{32} & : (jal(c) \vee jalr(c)) \wedge (x = 1^5) \\ c.gpr(x) & : sonst \end{cases} \tag{3.5}$$

Dabei wird mit $c^{i+1} = \delta_D(c^i)$ der Folgezustand für c^i in einer DLX-Rechnung $c^0, c^1, c^2, \ldots$ beschrieben. Analog gibt es eine Übergangsfunktion $\delta_H(h)$ für Hardware-Rechnungen mit den Konfigurationen $h^0, h^1, h^2, \ldots$, wobei $h^{i+1} = \delta_H(h^i)$ ist.

Zur Beschreibung der Äquivalenz zweier Zustände von DLX und Hardware dient die Simulationsrelation $sim(c,h)$, die aussagt, dass Zustand c der Spezifiktionsmaschine vom Zustand h der Implementierung simuliert wird.

Definition 3.2 *(Simulationsrelation) Wenn Zustand c von h simuliert wird, so gilt:*

$$sim(c,h) \Leftrightarrow \begin{cases} c.pc = h.pc \\ c.gpr = h.gpr \\ \forall x \in \{0,1\}^{30} : c.m_4(x00) = h.m(x) \end{cases}$$

Es müssen also der Programm Counter, das GPR und der Speicher übereinstimmen, dann sind die beiden Konfigurationen äquivalent. Der unterschiedlichen Adressierung der Speicher wird mittels *alignment* Rechnung getragen.

Definition 3.3 *(alignment) An die untersten beiden Bits von Speicheradressen in der DLX wird folgende Software-Bedingung gestellt:*

$$\forall c^i : \begin{cases} c^i.pc[1:0] = 00 \\ ea(c^i)[1:0] = 00 \quad : \quad lw(c^i) \vee sw(c^i) \end{cases}$$

Die Wörter werden im DLX-Speicher also nicht über zwei „Zeilen“ verteilt, sondern sind genauso angeordnet wie in der Hardware. Nun kann man den Simulationssatz zwischen DLX und Hardware aufstellen.

Theorem 3.2 *(Simulationssatz) Ist der Startzustand von Spezifikation und Implementierung äquivalent, so wird c^i von h^{2i} für alle i simuliert.*

$$\left.\begin{array}{c} c^0.pc = h^0.pc = 0^{32} \\ c^0.gpr = h^0.gpr \\ \forall x \in \{0,1\}^{30} : c^0.m_4(x00) = h^0.m(x) \end{array}\right\} \Rightarrow \forall i : sim(c^i, h^{2i})$$

Die Hardware benötigt also für jeden Schritt der DLX zwei Takte. Aus physikalischer Sicht ist es höchst unwahrscheinlich und nicht realistisch, dass nach dem Einschalten alle Register und Speicherelemente in der Implementierung die gleichen Daten wie in der Spezifikation enthalten. Die Anfangszustände sind zufällig und somit kann die Vorraussetzung für den Simulationssatz nicht ohne Weiteres realisiert werden. Jedoch kann die Vereinbarung getroffen werden, dass $\forall x \in \{0,1\}^5, y \in \{0,1\}^{30} : gpr(x)$ und $m(y)$ nur gelesen wird nachdem sie geschrieben wurden und dass ein boot-Programm zur Initialisierung in einem ROM-Bereich (Read Only Memory) im Speicher liegt (siehe Abbildung 3.9). In diesen Speicherbereich kann nicht geschrieben werden. Benutzerprogramme sollten also u.a. folgende Softwarebedingung beachten.

$$\forall i \geq 0 : sw(c^i) \Rightarrow \langle ea(c^i) \rangle \geq a$$

Mit den obigen Vereinbarungen kann die Konsistenz der Speicherinhalte gesichert werden. Im Folgenden gehen wir davon aus, dass die Start- und Softwarebedingungen für den Simulationssatz erfüllt sind. Wir setzen c^0 formal wie folgt:

$$c^0.pc = SISR \qquad c^0.gpr = h^0.gpr \qquad c^0.m_4(x00) = \begin{cases} h^0.m(x) & : \quad \langle x \rangle \geq a \\ ROM(x) & : \quad \langle x \rangle < a \end{cases}$$

Dabei bezeichnet $SISR$ die Startadresse der *Interrupt Service Routine*, welche Interrupts, wie z.B. $reset$, behandelt. Für die DLX gilt $SISR = 0^{32}$.

3.3.3 Konstruktion und Simulationsbeweis

Der Beweis der Korrektheit erfolgt per Induktion über alle Zustände der DLX. Der Induktionsanfang für $i = 0$ folgt direkt aus der Definition der Simulationsrelation und von c^0. Beim Schritt von $i \to i+1$ gilt die Induktionsvorraussetzung.
IV: Es gilt $sim(c^i, h^{2i})$, das heißt:

$$\begin{array}{rrcl} & c^i.pc & = & h^{2i}.pc \\ & c^i.gpr & = & h^{2i}.gpr \\ \forall x \in \{0,1\}^{30} : & c^i.m_4(x00) & = & h^{2i}.m(x) \end{array}$$

Nun muss gezeigt werden, dass pc, gpr und m für alle möglichen Schritte der DLX konsistent bleiben,

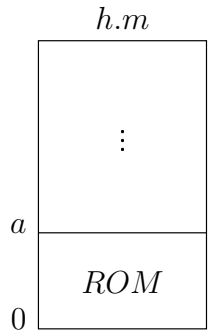


Abbildung 3.9: ROM-Bereich im Hardware-Speicher

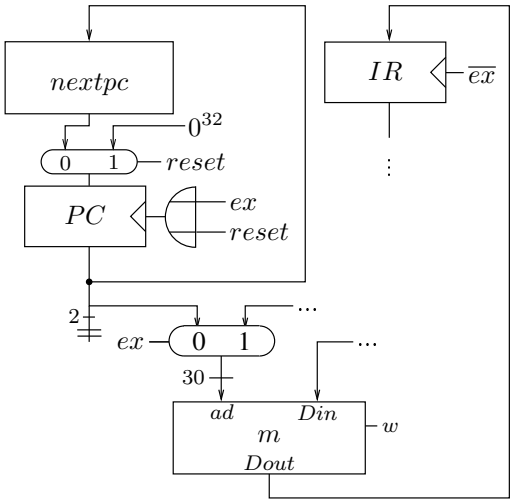


Abbildung 3.10: Implementierung des fetch-Modus

also dass $sim(c^{i+1}, h^{2i+2})$ gilt. Dabei muss zwischen den verschiedenen Instruktionarten unterschieden werden ($comp$, $compimm$, lw, sw, $control$).

Beweis: (Simulationssatz) Zunächst benötigen wir mehrere Lemmas, die es ermöglichen über die Äquivalenz von Spezifikation und Implementierung zu argumentieren. Als erstes wäre es wünschenswert, dass beide Maschinen die gleiche Instruktion verarbeiten. Dass dies der Fall ist, zeigt nachfolgendes Lemma. Dazu betrachten wir die Implementierung des fetch-Modus in Abbildung 3.10.

Program Counter

Zum Laden der nächsten Instruktion besitzt die DLX den Programmzähler $h.pc$, der im Register PC gespeichert wird. Hier steht die Adresse der Instruktion die im nächsten (geraden) $fetch$-Takt aus dem Speicher in das Instruktionsregister $h.IR$ gelesen werden soll. Der entsprechende Schaltkreis der den Takt in register $h.ex$ erzeugt, wurde bereits in Sektion 2.3 eingeführt. Es gilt:

- $h.ex = 0$: Instruktion an Adresse $h.pc$ wird aus dem Speicher m geholt (fetch).
- $h.ex = 1$: Instruktion wird ausgeführt (execute).

Abbildung 3.10 zeigt den relevanten Ausschnitt aus dem Datenpfad der DLX.

Lemma 3.3 *Das Instruktionsregister enthält in Takt $2i+1$ die Instruktion, die die DLX in Konfiguration i ausführt.*

$$h^{2i+1}.IR = I(c^i)$$

Beweis: Wir sehen uns an, welche Instruktion beim fetch aus dem Speicher geladen wird:

$$\begin{array}{lcll} I(c^i) & = & c^i.m_4(c^i.pc) & \text{(Definition } I(c)) \\ & = & c^i.m_4(c^i.pc[31:2]00) & \text{(alignment)} \\ & = & c^i.m_4(h^{2i}.pc[31:2]00) & \text{(IV : } sim(c^i, h^{2i})) \\ & = & h^{2i}.m(h^{2i}.pc[31:2]) & \text{(IV : } sim(c^i, h^{2i})) \\ & = & m_{Dout}(h^{2i}) & (h^{2i}.ex = 0,\ m_{ad} = h^{2i}.pc[31:2], \text{ RAM-Definition}) \\ & = & h^{2i+1}.IR & (\overline{h^{2i}.ex} = 1, \text{ Register-Definition}) \end{array}$$

Wurde die Instruktion in das Instruktionsregister geschrieben, so können nun die benötigten Informationen zur Ausführung dekodiert werden. Dies geschieht in der Hardware mit Hilfe des Instruktionsdekodierers, dessen Aufbau in Abbildung 3.11 dargestellt ist. Er dekodiert Prädikate, Funktionen und berechnet die Adressen der Zielregister Aad, Bad und Cad.

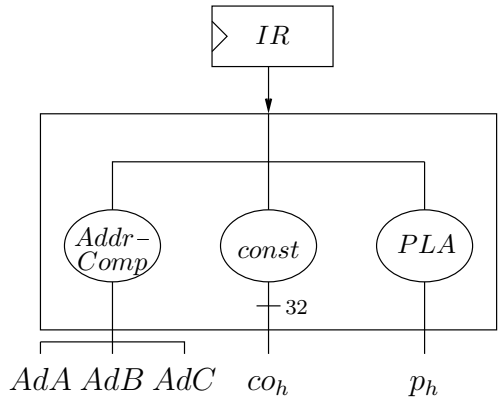


Abbildung 3.11: Instruktionsdekodierer

Sei p ein Prädikat in der DLX, dann beschreibe p_h das entsprechende Hardware-Prädikat. Zum Beispiel gilt

$$lw(c) \Leftrightarrow opc(c) = I(c)[31:26] = 100011.$$

Das zugehörige Hardwareprädikat ist dann

$$lw_h(h) = h.IR[31] \wedge \overline{h.IR[30]} \wedge \overline{h.IR[29]} \wedge \overline{h.IR[28]} \wedge h.IR[27] \wedge h.IR[26].$$

Da alle Prädikate auf der aktuellen Instruktion beruhen, lassen sich prinzipiell auch alle Hardwareprädikate durch solche Booleschen Polynome berechnen. Dies wird im Dekodierer von einer PLA (Programmable Logic Array) erledigt. Aus der Äquivalenz von $I(c)$ und $h.IR$ (vgl. Lemma 3.3) folgt direkt das folgende Lemma.

Lemma 3.4 *Alle Hardware-Prädikate sind genau dann wahr, wenn ihre zugehörigen Prädikate in der DLX-Spezifikation wahr sind.*

$$p_h(h^{2i+1}) = p(c^i)$$

Beweis:

$$\begin{array}{lcll} p_h(h^{2i+1}) & = & p'(h^{2i+1}.IR) & \text{(Definition } p_h) \\ & = & p'(I(c^i)) & \text{(Lemma 3.3)} \\ & = & p(c^i) & \square \end{array}$$

Bei der Implementierung der Prädikate via PLA ist aus Kostengründen darauf zu achten, dass nur diejenigen Instruktionsbits betrachtet werden, von denen $p'(I)$ auch tatsächlich abhängt.

Parallel zu Lemma 3.4 kann man ein Lemma für korrespondierende Hardware-Funktionen ($f_h(h)$ entspricht $f(c)$) wie z.B. $RS1_h(h) = h.IR[25:21]$ aufstellen.

Lemma 3.5 *Alle Hardware-Funktionen, die allein vom Zustand des Instruktionsregisters abhägen besitzen den gleichen Wert wie ihre zugehörigen Funktionen in der DLX-Spezifikation.*

$$f_h(h^{2i+1}) = f(c^i)$$

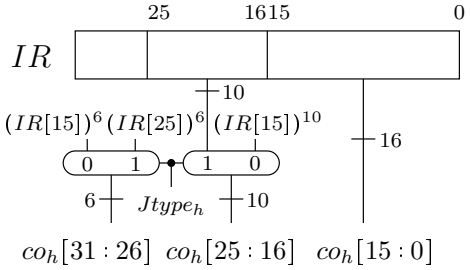
Der Beweis verwendet entsprechende Funktionen $f'(I)$, die auf 32-bit Instruktionen I definiert sind.
Beweis

$$\begin{array}{rcll} f_h(h^{2i+1}) & = & f'(h^{2i+1}.IR) & \text{(Definition } f_h\text{)} \\ & = & f'(I(c^i)) & \text{(Lemma 3.3)} \\ & = & f(c^i) & \square \end{array}$$

Für die erweiterte immediate-Konstante der Hardware $co(h)$ erhalten wir:

$$\begin{array}{rcl} co(h) & = & co'(h.IR) \\ & = & \begin{cases} (h.IR[25])^6 h.IR[25:0] & : \overline{u'(h.IR)} \wedge Jtype'(h.IR) \\ (h.IR[15])^{16} h.IR[15:0] & : \overline{u'(h.IR)} \wedge Itype'(h.IR) \\ 0^{16} h.IR[15:0] & : u'(h.IR) \end{cases} \end{array}$$

Die sign-extension wird mit MUXen implementiert.



Über den Aufbau kann man zeigen:

$$co_h(h^{2i+1}) = co(c^i)$$

Nun wollen wir ALU-Operationen betrachten. Abbildung 3.12 zeigt den relevanten Ausschnitt des Datenpfades der DLX sowie die Implementierung von $aluf_h(h)$. Es gilt nach Konstruktion von $aluf_h(h)$ und Definition der DLX:

$$\begin{array}{rcl} aluf_h(h^{2i+1}) & = & \begin{cases} h^{2i+1}.IR[29:26] & : \overline{Rtype_h(h^{2i+1})} \\ h^{2i+1}.IR[3:0] & : Rtype_h(h^{2i+1}) \end{cases} \\ aluf(c^i) & = & \begin{cases} I(c^i)[29:26] & : \overline{Rtype(c^i)} \\ I(c^i)[3:0] & : Rtype(c^i) \end{cases} \end{array}$$

Lemma 3.6 *Spezifikation und Implementierung berechnen dieselben ALU-Funktionen.*

$$aluf_h(h^{2i+1}) = aluf(c^i)$$

Dies folgt ebenfalls direkt aus den Lemmas 3.4 und 3.5.
Die beiden Leseadressen des GPR sind belegt mit:

$$\begin{array}{rcl} AdA(h) & = & h.IR[25:21] \\ AdB(h) & = & h.IR[20:16] \end{array}$$

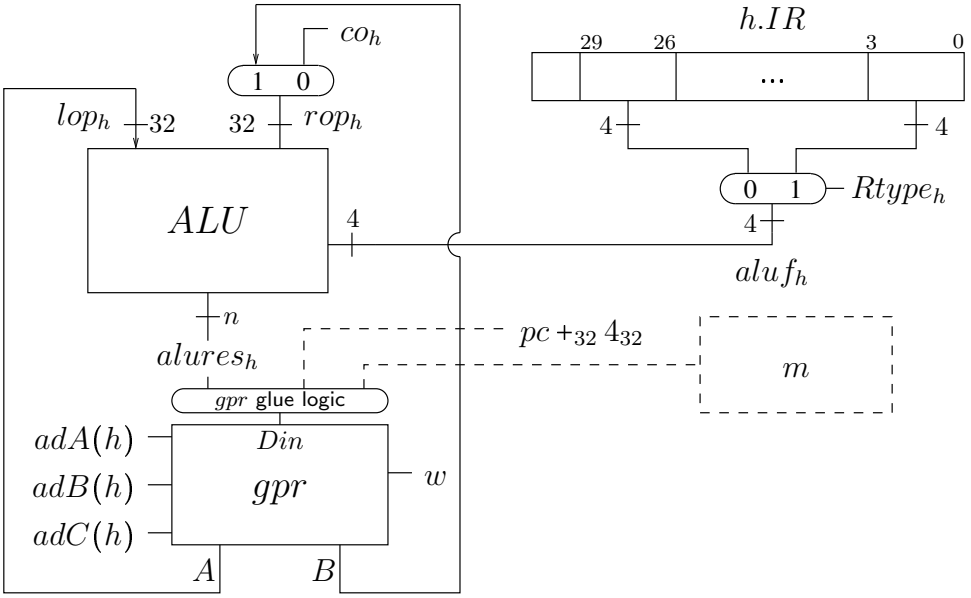


Abbildung 3.12: ALU-Instruktionen

Dann folgt:

$$\begin{aligned} AdA(h^{2i+1}) &= h^{2i+1}.IR[25:21] \\ &= I(c^i)[25:21] \quad \text{(Lemma 3.3)} \\ &= RS1(c^i) \\ AdB(h^{2i+1}) &= h^{2i+1}.IR[20:16] \\ &= I(c^i)[20:16] \quad \text{(Lemma 3.3)} \\ &= RS2(c^i) \quad (Rtype(c^i)) \end{aligned}$$

In DLX und Hardware werden also dieselben Register für $RS1$ und $RS2$ (im Falle von $Rtype$-Instruktionen) aus dem GPR gelesen.
Das Schreibsignal des GPR ist wie folgt definiert:

$$gpr_w = ex \wedge (comp_h \vee compimm_h \vee lw_h \vee jal_h \vee jalr_h)$$

Dann wird in fetch-Takten das GPR wegen $h^{2i}.ex = 0$ nie geschreiben:

$$\begin{aligned} gpr_w(h^{2i}) &= h^{2i}.ex \wedge (comp_h(h^{2i}) \vee compimm_h(h^{2i}) \vee lw_h(h^{2i}) \vee jal_h(h^{2i}) \vee jalr_h(h^{2i})) \\ &= 0 \wedge (comp_h(h^{2i}) \vee compimm_h(h^{2i}) \vee lw_h(h^{2i}) \vee jal_h(h^{2i}) \vee jalr_h(h^{2i})) \\ &= 0 \end{aligned}$$

Daraus ergibt sich eine wichtige Feststellung:

Lemma 3.7

$$h^{2i+1}.gpr = c^i.gpr$$

Nach dem fetch sind die Registerinhalte der Hardware identisch zur DLX.

Beweis:

$$\begin{aligned} h^{2i+1}.gpr &= h^{2i}.gpr \quad (gpr_w(h^{2i}) = 0) \\ &= c^i.gpr \quad (sim(c^i, h^{2i})) \quad \square \end{aligned}$$

Dies können wir nutzen um die Ausgänge des GPR näher zu betrachten:

Lemma 3.8

$$\begin{aligned} A(h^{2i+1}) &= c^i.gpr(RS1(c^i)) \\ B(h^{2i+1}) &= c^i.gpr(RS2(c^i)) \qquad \textit{für } Rtype_h(h^{2i+1}) \end{aligned}$$

Beweis:

$$\begin{aligned} A(h^{2i+1}) &= h^{2i+1}.gpr_{DoutA} && \text{(Konstruktion)} \\ &= h^{2i+1}.gpr(\underbrace{AdA(h^{2i+1})}_{RS1(c^i)}) && \text{(Register File Spezifikation)} \\ &= c^i.gpr(RS1(c^i)) && \text{(Lemma 3.7)} \quad \checkmark \\ B(h^{2i+1}) &= h^{2i+1}.gpr_{DoutB} && \text{(Konstruktion)} \\ &= h^{2i+1}.gpr(\underbrace{AdB(h^{2i+1})}_{RS2(c^i)}) && \text{(Register File Spezifikation)} \\ &= c^i.gpr(RS2(c^i)) && \text{(Lemma 3.7)} \quad \square \end{aligned}$$

Der linke Operand der ALU ergibt sich dann zu:

$$\begin{aligned} lop_h(h^{2i+1}) &= A(h^{2i+1}) && \text{(Konstruktion)} \\ &= c^i.gpr(RS1(c^i)) && \text{(Lemma 3.8)} \\ &= lop(c^i) \end{aligned}$$

Für den rechten Operanden erhalten wir:

$$\begin{aligned} rop_h(h^{2i+1}) &= \begin{cases} B(h^{2i+1}) & : \quad Rtype_h(h^{2i+1}) \\ co_h(h^{2i+1}) & : \quad \text{sonst} \end{cases} && \text{(Konstruktion, MUX)} \\ &= \begin{cases} B(h^{2i+1}) & : \quad Rtype(c^i) \\ co(c^i) & : \quad \text{sonst} \end{cases} && \text{(Lemma 3.4, 3.5)} \\ &= \begin{cases} c^i.gpr(RS2(c^i)) & : \quad Rtype(c^i) \\ co(c^i) & : \quad \text{sonst} \end{cases} && \text{(Lemma 3.8)} \\ &= rop(c^i) \end{aligned}$$

Zusammendfassend gilt also:

$$\begin{aligned} lop_h(h^{2i+1}) &= lop(c^i) \\ rop_h(h^{2i+1}) &= rop(c^i) \\ aluf_h(h^{2i+1}) &= aluf(c^i) \end{aligned}$$

Aus der Korrektheit der ALU folgt, dass für die angelegten Eingänge das korrekte Ergebnis berechnet wird. Wir erhalten folgendes Lemma.

Lemma 3.9 *(aluop) Das Ergebnis der ALU-Operationen auf beiden Maschinen ist identisch.*

$$alures_h(h^{2i+1}) = aluop(lop(c^i), rop(c^i), aluf(c^i)) = alures(c^i)$$

Dieses Ergebnis würde nun ins GPR geschrieben werden, zuvor sollen aber noch load und store word-Instruktionen behandelt werden.

Auch der Speicher $h.m$ wird wegen $h^{2i}.ex = 0$ in der fetch-Phase nicht verändert:

$$\begin{aligned} m_w(h^{2i}) &= h^{2i}.ex \wedge sw_h \\ &= 0 \wedge sw_h \\ &= 0 \\ \forall x \in \{0,1\}^{30}.\ h^{2i+1}.m(x) &= h^{2i}.m(x) \quad (m_w(h^{2i}) = 0) \end{aligned}$$

Daher gilt aufgrund der Induktionsvorraussetzung $sim(c^i, h^{2i})$:

Lemma 3.10

$$h^{2i+1}.m(x) = c^i.m_4(x00)$$

Nach dem fetch sind die Speicherinhalte der Hardware identisch zur DLX. Nun soll ein Wert aus Speicher gelesen oder in denselben geschrieben werden. Abbildung 3.13 zeigt die relevanten Datenpfade. Es gilt:

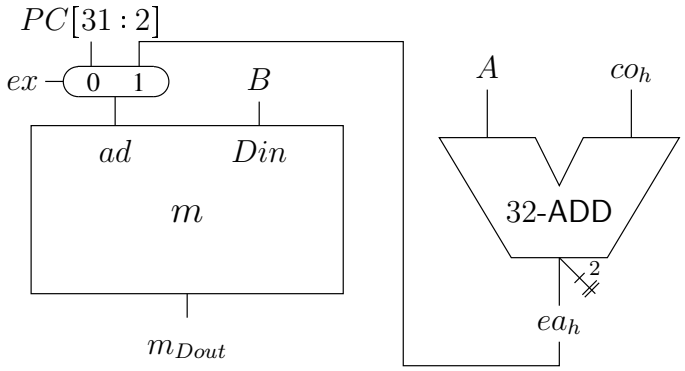


Abbildung 3.13: Datenpfad für load und store word-Instruktionen

Lemma 3.11 *(memory write) Der Speicher ändert sich nur für store word-Instruktionen.*

$$\overline{sw(c^i)} \quad \Rightarrow \quad h^{2i+2}.m = h^{2i}.m$$

Beweis: Für das Schreibsignal im execute-Takt gilt:

$$m_w(h^{2i+1}) = 1 \wedge sw(c^i) = sw(c^i)$$

Daraus erhalten wir mit Hilfe von Lemma 3.10:

$$\begin{array}{lll} \overline{sw(c^i)} & \Rightarrow & m_w(h^{2i+1}) = 0 \\ & \Rightarrow & h^{2i+2}.m = h^{2i+1}.m = h^{2i}.m \end{array}$$

Für die Adresse, die am Speicher während der execute-Phase anliegt erkennen wir:

$$\begin{array}{lcll} m_{ad}(h^{2i+1}) & = & ea_h(h^{2i+1})[31:2]) & (\text{MUX, } h^{2i+1}.ex = 1) \\ & = & (A(h^{2i+1}) +_{32} co_h(h^{2i+1}))[31:2] & (\text{Addierer}) \\ & = & (c^i.gpr(RS1(c^i)) +_{32} sxtimm(c^i))[31:2] & (\text{Lemmas 3.8, 3.5}) \\ & = & ea(c^i)[31:2] & (\text{Definition } ea(c)) \end{array}$$

An m_{Din} liegt B an. lw und sw sind $Itype$-Instruktionen, für diesen Fall ist B wie folgt definiert.

$$\begin{array}{lcll} AdB(h^{2i+1}) & = & h^{2i+1}.IR[20:16] & \\ & = & I(c^i)[20:16] & \\ & = & RD(c^i) \qquad (lw(c^i) \vee sw(c^i) \Rightarrow Itype(c^i)) & \\ B(h^{2i+1}) & = & c^i.gpr(RD(c^i)) & \end{array}$$

Da $sw(c^i) \Rightarrow m_w(h^{2i+1}) = 1$ gilt, wird der Speicher an der angelegten Adresse mit dem Wert am Dateneingang beschrieben. Es gilt für den Speicher im nächsten Takt:

$$h^{2i+2}.m(x) \quad = \quad \begin{cases} c^i.gpr(RD(c^i)) & : \quad x = ea(c^i)[31:2] \\ h^{2i+1}.m(x) & : \quad \text{sonst} \end{cases}$$

Daraus lässt sich die dritte Zeile des Simulationssatzes folgern.
Im Falle von $sw(c^i) \wedge (x = ea(c^i)[31:2])$ gilt:

$$\begin{aligned} h^{2i+2}.m(x) &= c^i.gpr(RD(c^i)) \\ &= c^{i+1}.m_4(x00) \quad \text{(Spezifikation, 3.2)} \end{aligned}$$

Liegt keine store word Anweisung vor oder ist $x \neq ea(c^i)[31:2]$ so bleibt der Speicher unverändert.

$$\begin{aligned} h^{2i+2}.m(x) &= h^{2i+1}.m(x) \quad \text{(Definition RAM, } m_w(h^{2i+1}) = 0 \text{ oder } x \neq ea_h(h^{2i+1})) \\ &= h^{2i}.m(x) \quad \text{(Lemma 3.10)} \\ &= c^i.m_4(x00) \quad (sim(c^i, h^{2i})) \\ &= c^{i+1}.m_4(x00) \quad \text{(Spezifikation, 3.3)} \end{aligned}$$

Die Hardware simuliert also auch den Speicher der DLX korrekt.

$$h^{2i+2}.m(x) = c^{i+1}.m_4(x00) \qquad \checkmark$$

Wir wollen nun load word-Instruktionen betrachten. Diese lesen den Speicher und schreiben die Daten ins gpr. Mit den bisherigen Definitionen können wir bereits das Ergebnis des load word-Zugriffs bestimmen.

Lemma 3.12 *(load word) Eine load word-Speicherzugriff liefert für DLX und Hardware das gleiche Ergebnis:*

$$m_{Dout}(h^{2i+1}) = c^i.m_4(ea(c^i))$$

Beweis:

$$\begin{aligned} m_{Dout}(h^{2i+1}) &= h^{2i+1}.m(m_{ad}(h^{2i+1})) \quad \text{(Spezifikation RAM)} \\ &= h^{2i+1}.m(ea(c^i)[31:2]) \quad \text{(Konstruktion } m_{ad}(h)) \\ &= c^i.m_4(ea(c^i)[31:2]00) \quad \text{(Lemma 3.10)} \\ &= c^i.m_4(ea(c^i)) \quad \text{(alignment)} \quad \square \end{aligned}$$

Nun wollen wir uns ansehen, wie die Ergebnisse in das GPR geschrieben werden. Abbildung 3.14 zeigt einen teil der GPR „glue logic“, die dessen Eingänge ansteuert. Die Korrektheit der Signale $aluop_h$,

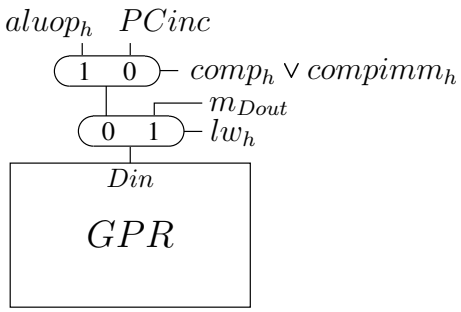


Abbildung 3.14: GPR glue logic für gpr_{Din}

m_{Dout} sowie die der Prädikate wurde bereits bewiesen. Es fehlt noch $PCinc$, das die zu sichernde Rücksprungadresse für jump and link-Anweisungen liefert. In Abbildung 3.15 ist der Ausschnitt von $nextPC$ dargestellt, der den PC inkrementiert. Ein Incrementer erhöht den Wert des Eingangssignals um 1. Im vorliegenden Schaltkreis kann dies durch einen 32-bit Addierer simuliert werden, der $h.pc$ um 4 erhöht. In den Übungen soll ein eigenständiger Incrementer konstruiert werden.

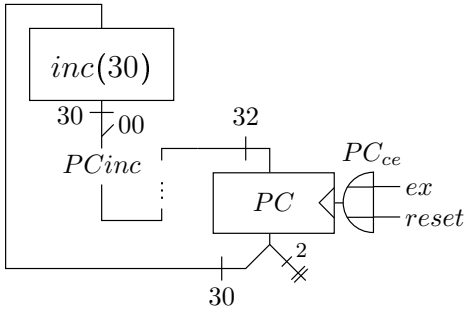


Abbildung 3.15: Implementierung von $PCinc$

Lemma 3.13 *Der inkrementierte Hardware-PC entspricht dem inkrementierten Programmzähler der DLX:*

$$PCinc(h^{2i+1}) = c^i.pc +_{32} 4_{32}$$

Beweis:

$$\begin{aligned}
PCinc(h^{2i+1}) &= (h^{2i+1}.pc[31:2] +_{30} 1_{30})00 \quad (\text{Konstruktion, Definition Incrementer}) \\
&= h^{2i+1}.pc[31:2]00 +_{32} 4_{32} \quad (\text{Zerlegungslemma}) \\
&= h^{2i}.pc[31:2]00 +_{32} 4_{32} \quad (h^{2i}.ex = 0 \Rightarrow PC_{ce}(h^{2i}) = 0) \\
&= c^i.pc[31:2]00 +_{32} 4_{32} \quad (sim(c^i, h^{2i})) \\
&= c^i.pc +_{32} 4_{32} \quad (\text{alignment}) \quad \square
\end{aligned}$$

Nun kann ein weiterer Teil des Simulationssatzes, die Bedingung an das GPR, bewiesen werden. Es soll gelten:

$$c^{i+1}.gpr \stackrel{!}{=} h^{2i+2}.gpr$$

Beweis: Man muss vier verschiedene Fälle unterscheiden, in denen das GPR eventuell modifiziert wird:

- $comp(c^i) \vee compimm(c^i)$ - das Ergebnis einer ALU-Operation wird in ein Register geschrieben
- $lw(c^i)$ - ein Wort aus dem Speicher wird in ein Register geschrieben
- $jal(c^i) \vee jalr(c^i)$ - die Rücksprungadresse wird in $R31$ gesichert
- sonst - das GPR bleibt unverändert

Für das GPR der Hardware nach dem execute-Takt gilt mit $x \in \{0,1\}^5$:

$$h^{2i+2}.gpr(x) = \begin{cases} gpr_{Din}(h^{2i+1}) & : \quad (x = AdC(h^{2i+1})) \wedge gpr_w(h^{2i+1}) \\ h^{2i+1}.gpr(x) & : \quad \text{sonst} \end{cases} \qquad (\text{Def. RAM})$$

AdC bestimmt das Register des gpr, das unter Umständen beschrieben wird. Abbildung 3.16 zeigt den zugehörigen Teil des $Addr\text{-}Comp$-Schaltkreises.

Lemma 3.14 *Das GPR-Zielregister in der Hardware entspricht dem der DLX:*

$$AdC(h^{2i+1}) = \begin{cases} 1^5 & : jal(c^i) \vee jalr(c^i) \\ RD(c^i) & : \text{sonst} \end{cases}$$

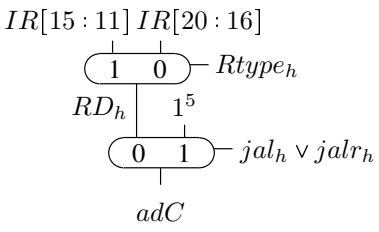


Abbildung 3.16: Teil des $Addr\text{-}Comp$ Schaltkreises zur Bestimmung von AdC

Dies folgt direkt aus der Konstruktion von $Addr\text{-}Comp$ und den Lemmas 3.4 und 3.5.

Für den Dateneingang des gpr untersuchen wir die Konstruktion der gpr glue logic.

$$gpr_{Din}(h^{2i+1}) = \begin{cases} aluop_h(h^{2i+1}) & : \quad (comp_h(h^{2i+1}) \vee compimm_h(h^{2i+1})) \\ m_{Dout}(h^{2i+1}) & : \quad lw_h(h^{2i+1}) \\ PCinc(h^{2i+1}) & : \quad \text{sonst} \end{cases} \qquad \text{(Konstr. 3.14)}$$

In den oben genannten Fällen wird das gpr dann mit den angelegten Werten beschrieben.

$$h^{2i+2}.gpr(x) = \begin{cases} aluop_h(h^{2i+1}) & : \quad (comp_h(h^{2i+1}) \vee compimm_h(h^{2i+1})) \\ & \quad \wedge (x = AdC(h^{2i+1})) \\ m_{Dout}(h^{2i+1}) & : \quad lw_h(h^{2i+1})) \wedge (x = AdC(h^{2i+1})) \\ PCinc(h^{2i+1}) & : \quad (jal_h(h^{2i+1}) \vee jalr_h(h^{2i+1})) \\ & \quad \wedge (x = 1^5) \\ h^{2i+1}.gpr(x) & : \quad \text{sonst} \end{cases}$$

$$= \begin{cases} aluop(lop(c^i), rop(c^i), aluf(c^i)) & : \quad (comp(c^i) \vee compimm(c^i)) \wedge (x = RD(c^i)) \\ c^i.m_4(ea(c^i)) & : \quad lw(c^i) \wedge (x = RD(c^i)) \\ c^i.pc +_{32} 4_{32} & : \quad (jal(c^i) \vee jalr(c^i)) \wedge (x = 1^5) \\ c^i.gpr(x) & : \quad \text{sonst} \end{cases}$$

Dies folgt aus den Lemmas 3.9, 3.4, 3.14, 3.12, 3.13, 3.7 und der Induktionsvorraussetzung $sim(c^i, h^{2i})$. Mit der Spezifikation des gpr (3.5) erhalten wir:

$$h^{2i+2}.gpr(x) = c^{i+1}.gpr(x) \qquad \square$$

Die Hardware simuliert also auch den Speicher das general purpose register file korrekt. Bleibt nur noch die Korrektheit des Programmzählers zu beweisen. Hierbei sind die Instruktionen zur Kontrolle des Programmflusses (branch, jump etc.) zu betrachten. Abbildung 3.17 stellt die Implementierung des nextPC-Blockes dar. Aus der Konstruktion lässt sich ableiten:

$$btarget_h(h^{2i+1}) = \begin{cases} A(h^{2i+1}) & : jr_h(h^{2i+1}) \vee jalr_h(h^{2i+1}) \\ sxtimm_h(h^{2i+1}) +_{32} h^{2i+1}.pc & : \text{sonst} \end{cases}$$

$$= \begin{cases} c^i.gpr(RS1(c^i)) & : jr(c^i) \vee jalr(c^i) \\ sxtimm(c^i) +_{32} h^{2i+1}.pc & : \text{sonst} \end{cases} \qquad \text{(Lemmas 3.8, 3.4, 3.5)}$$

$$= \begin{cases} c^i.gpr(RS1(c^i)) & : jr(c^i) \vee jalr(c^i) \\ sxtimm(c^i) +_{32} h^{2i}.pc & : \text{sonst} \end{cases} \qquad (PC_{ce}(h^{2i}) = 0)$$

$$= \begin{cases} c^i.gpr(RS1(c^i)) & : jr(c^i) \vee jalr(c^i) \\ sxtimm(c^i) +_{32} c^i.pc & : \text{sonst} \end{cases} \qquad (sim(c^i, h^{2i}))$$

$$= btarget(c^i)$$

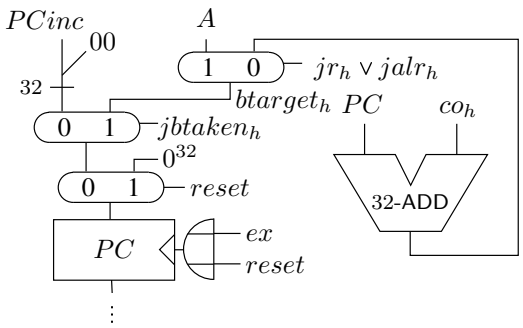


Abbildung 3.17: nextPC

Das bedeutet, dass das Sprungziel $btarget$ dem der DLX entspricht. Auch die anderen Varianten für den neuen PC, $PCinc$ und 0^{32}, sind in beiden Maschinen identisch. Daher genügt es für die Korrektheit des program counter zu zeigen, dass

$$jbtaken_h(h^{2i+1}) \stackrel{!}{=} jbtaken(c^i).$$

Dazu definieren wir ein weiteres DLX-Prädikat:

$$Aeqz(c) = (c.gpr(RS1(c)) = 0^{32})$$

Damit lässt sich $btaken(c)$ auch anders schreiben:

$$btaken(c) = beqz(c) \wedge Aeqz(c) \vee bnez(c) \wedge \overline{Aeqz(c)}$$

Mit $branch(c) = beqz(c) \vee bnez(c)$ und dem opcode für $beqz$ bzw. $bnez$ (110100 bzw. 110101) ergibt sich:

$$\begin{aligned} btaken(c) &= branch(c) \wedge (Aeqz(c) \wedge \overline{I(c)[26]} \vee \overline{Aeqz(c)} \wedge I(c)[26]) \\ &= branch(c) \wedge (Aeqz(c) \oplus I(c)[26]) \end{aligned}$$

Das korrespondierende Hardware-Prädikat $Aeqz_h(h)$ wird wie in Abbildung 3.18 dargestellt mit einem *zero tester* implementiert. Wegen $A(h^{2i+1}) = c^i.gpr(RS1(c^i))$ (Lemma 3.8) gilt:

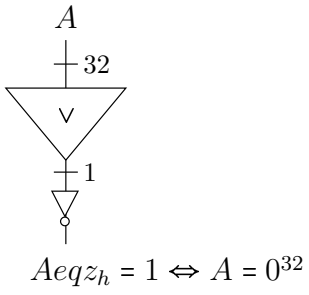


Abbildung 3.18: Implementierung von $Aeqz_h$

$$Aeqz_h(h^{2i+1}) = Aeqz(c^i)$$

Dann konstruiert man

$$jbtaken_h = jump_h \vee (branch_h \wedge (Aeqz_h \oplus IR[26]))$$

und es gilt:

$$\begin{aligned} jbtaken_h(h^{2i+1}) &= jump_h(h^{2i+1}) \vee (branch_h(h^{2i+1}) \wedge (Aeqz_h(h^{2i+1}) \oplus (h^{2i+1}).IR[26])) \\ &= jump(c^i) \vee (branch(c^i) \wedge (Aeqz(c^i) \oplus I(c^i)[26])) \quad \text{(Lemmas 3.4, 3.3)} \\ &= jbtaken(c^i) \quad \square \end{aligned}$$

Nun lässt sich die Simulationsbedingung für $h.pc$ beweisen:

$$\begin{aligned} h^{2i+2}.pc &= PC_{in}(h^{2i+1}) \quad (PC_{ce}(h^{2i+1}) = 1) \\ &= \begin{cases} 0^{32} & : reset \\ btarget_h(h^{2i+1}) & : \overline{reset} \wedge jbtaken_h(h^{2i+1}) \\ PCinc(h^{2i+1}) & : \text{sonst} \end{cases} \\ &= \begin{cases} 0^{32} & : reset \\ btarget(c^i) & : \overline{reset} \wedge jbtaken(c^i) \quad \text{(Lemmas 3.4, 3.3, 3.13)} \\ c^i.pc +_{32} 4_{32} & : \text{sonst} \end{cases} \\ &= c^{i+1}.pc \quad \text{(Spezifikation, 3.4 für } \overline{reset}) \end{aligned}$$

Damit sind alle nötigen Beweise erbracht. Es gilt

$$\begin{aligned} c^{i+1}.pc &= h^{2i+2}.pc \\ c^{i+1}.gpr &= h^{2i+2}.gpr \\ \forall x \in \{0,1\}^{30}: \quad c^{i+1}.m_4(x00) &= h^{2i+2}.m(x) \end{aligned}$$

und somit:

$$sim(c^{i+1}, h^{2i+2}) \quad \square$$

Wir haben also die Korrektheit der Hardware-Implementierung der DLX bewiesen. Entscheidend für die fehlerfreie und sichere Ausführung von Programmen ist aber nicht nur die Korrektheit der zugrundeliegenden Hardware, sondern auch die des Betriebssystems. Daher wollen wir uns in den foglenden Kapiteln mit der Spezifikation und Verifikation von Betriebssystemkernen beschäftigen.

Kapitel 4

Software / C0

Wie schon zuvor angeklungen, wollen wir verstehen, wie Betriebssystemkerne funktionieren und über ihre Korrektheit argumentieren. Prinzipiell ist dies auf Assembler-Ebene möglich, allerdings ist das Programmieren und Verifizieren in Assembler ist äußerst primitiv und mühsam, so dass nur eine geringe Produktivität erreicht würde. Um Programme komfortabler und übersichtlicher schreiben zu können, wurden Hochsprachen wie C, Java, SML etc. entwickelt. Viele Betriebssystemkerne jedoch sind zwar in C geschrieben aber verwenden trotzdem inline assembler code. Dies ist notwendig, da Teile der Hardware, wie zum Beispiel die Speicherorganisation, Register, der pc und Geräte von den Hochsprachen vor dem Programmierer verborgen werden. Stattdessen arbeiten die Programme auf selbst definierten Variablen und und Datenstrukturen.

Bei der Programmierung von Betriebsystemen oder Treibern muss man aber zwangsläufig auf oben genannte Hardwarekomponenten zugreifen. Daher enthalten Betriebssystemkerne inhärent auch assembler code zum Beispiel beim process save and restore. Dabei werden sowohl Registerinhalte in C-Variablen als auch die Inhalte von C-Variablen in Register geschrieben. Die Frage ist, wie beide Welten miteinander verbunden sind. Der Assembler müsste die Adressen der C-Variablen im Speicher kennen. Diese werden von der *allocation function* des Compilers festgelegt. Zum präzisen Argumentieren über Betriebssystemkerne, muss man also auch den Compiler spezifizieren.

Nun ist es schwierig, sich auf eine allgemein anerkannte Semantik von C zu einigen. Daher beschäftigt sich dieses Kapitel mit der Entwicklung einer Hochsprache für die DLX namens $C0$. Diese vereint die bekannte *C*-Syntax mit der wohldefinierten *Pascal*-Semantik und inline assembler code. Im Folgenden sollen Syntax und Semantik von $C0$-Programmen definiert werden. Anschließend wird ein Compiler für DLX-Assembler konstruiert und dessen Korrektheit bewiesen. Zunächst ist allerdings zur Definition der $C0$-Syntax eine Einführung in kontextfreie Grammatiken nötig.

4.1 Kontextfreie Grammatiken

Grammatiken sind mathematische Modelle, die dazu dienen, formale Sprachen zu definieren. Formale Sprachen werden über ein Alphabet A gebildet, das eine Menge von Zeichen a_i ist.

$$A^n = \{a_1 \cdots a_n \mid \forall i : a_i \in A\}$$

ist die Menge aller Zeichenreihen der Länge n über das Alphabet A. Die Zeichenreihe der Länge Null ist das leere Wort ε. Das leere Wort ist kein Zeichen und somit in keinem Alphabet enthalten.

$$A^0 = \{\varepsilon\}, \qquad \forall A.\ \varepsilon \notin A$$

Wir definieren die Menge aller Zeichenreihen beliebiger Länge über A wie folgt.

$$A^* = \bigcup_{i \geq 0} A^i$$

Für das leere Wort erhalten wir folgende Eigenschaft.

$$\forall a \in A^*.\ a\varepsilon = a = \varepsilon a$$

Soll das leere Wort ausgeschlossen sein so verwenden wir A^+.

$$A^+ = \bigcup_{i>0} A^i$$

Eine formale Sprache L kann dann wie folgt allgemein definiert werden.

$$L \subseteq A^*$$

Dies betrachtet zunächst nur die Syntax der Sprache, die Semantik kann später hinzugefügt werden. Um genau zu spezifizieren welche Zeichenreihen zu L gehören, verwendet man Grammatiken. Die einfachsten Grammatiken sind kontextfreie Grammatiken (auch cfg - **c**ontext **f**ree **g**rammar), auf die wir uns hier beschränken wollen. Formal werden solche Grammatiken wie folgt beschrieben.

Definition 4.1 *(cfg) Eine Kontextfreie Grammatik G ist ein Mengentupel*

$$G = (T, N, S, P)$$

bestehend aus:

- T *- endliche Menge, Terminalalphabet*
- N *- endliche Menge, Nichtterminalalphabet,* $T \cap N = \varnothing$
- $S \in N$ *- Startsymbol*
- $P \subset N \times (N \cup T)^*$ *- Produktionensystem*

Aus Nichtterminalen können mit Hilfe der Produktionsregeln neue Symbole abgeleitet werden. Man beachte, dass die Regeln immer nur von einem Nichtterminal ausgehen, der Kontext dieser, also die angrenzenden Zeichen, werden ignoriert. Daher spricht man von Kontextfreiheit. Die Sprache, die durch die Grammatik beschrieben wird, besteht allein aus Terminalen. Jene können nicht weiter umgeformt werden. Jede Folge von Ableitungen entspringt im Startsymbol S.

Wir führen eine vereinfachende Schreibweise für Ableitungen nach den Produktionsregeln ein. Für die Regel $(n, \omega) \in P$ schreibt man auch kurz:

$$n \longrightarrow \omega$$

Das heißt, aus n kann ω abgeleitet werden. Falls mehrere alternative Ableitungen für n in P existieren, also $(n, \omega^1), \ldots, (n, \omega^t) \in P$ so schreibt man:

$$n \longrightarrow \omega^1 \mid \cdots \mid \omega^t$$

Es folgt ein Beispiel für eine einfache cfg:

$$\begin{array}{rcl} T & = & \{X, 0, 1, \ldots, 9\} \\ N & = & \{V, C, \langle CF\rangle\} \\ S & = & V \\ P & : & \begin{array}{rcl} C & \longrightarrow & 0 \mid 1 \mid \cdots \mid 9 \\ \langle CF\rangle & \longrightarrow & C \mid \langle CF\rangle C \\ V & \longrightarrow & X \mid X\langle CF\rangle \end{array} \end{array}$$

Dabei stehen V für eine Variable, C für eine Ziffer und $\langle CF\rangle$ für eine Ziffernfolge. Es lässt sich erahnen, dass die obige Grammatik Wörter erzeugt, die mit einem X beginnen, gefolgt von einer beliebig langen Folge aus Ziffern von 0 bis 9. Um die erzeugte Sprache einer Grammatik $L(G) \in T^*$ näher zu definieren müssen Ableitungsbäume betrachtet werden.

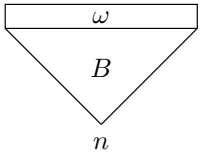


Abbildung 4.1: Ableitungsbaum

Definition 4.2 *(Ableitungsbäume) Zur einer Grammatik G existieren Ableitungsbäume B mit Wurzel $n \in N$ und Blattwort ω (siehe Abbildung 4.1). Für die Bäume gelten folgende Regeln:*

1. *$n = \omega$, dann existiert ein trivialer Ableitungsbaum mit Wurzel n und Blattwort n*
2. *Es existiert ein Ableitungsbaum T mit Wurzel $n \in N$ und Blattwort ω mit $\omega = \omega_1 m \omega_2$ und $m \longrightarrow \omega_3 \in P$, dann existiert ein Ableitungsbaum mit Wurzel n und Blattwort $\omega_1\omega_3\omega_2$ (siehe Abbildung 4.2)*
3. *Weitere Möglichkeiten, einen Ableitungsbaum zu entwickeln, gibt es nicht.*

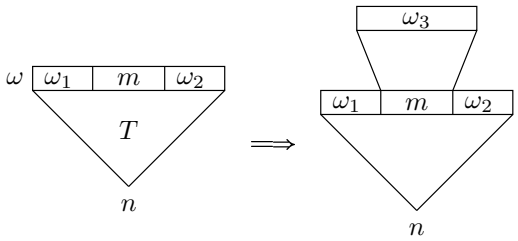


Abbildung 4.2: weitere Ableitung eines Ableitungsbaumes

Diese Definition ist informal, allerdings ist eine Umwandlung in formale Notation möglich. Man beachte, dass die Ableitungsbäume keine gewöhnlichen Bäume aus der mathematischen Theorie darstellen, da hier die Reihenfolge der Blätter betrachtet wird. Abbildung 4.3 zeigt als Beispiel für obige Definition die schrittweise Ableitung des Blattwortes $X01$ aus der Wurzel V mit der gegebenen Beispielgrammatik.

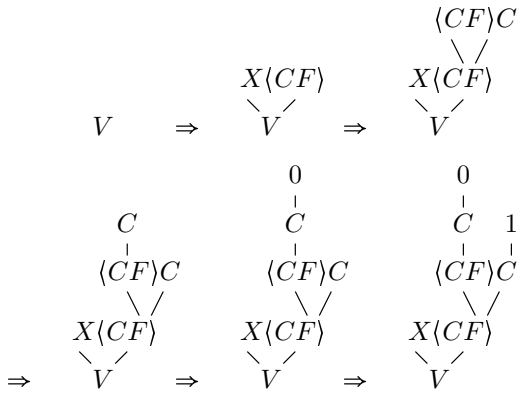


Abbildung 4.3: Ableitungsbeispiel

Definition 4.3 *Sei $n \in N$, $\omega \in \{N \cup T\}^*$, dann kann ω aus n (in mehrereren Schritten) mit der Grammatik G abgeleitet werden, falls ein Ableitungsbaum mit Wurzel n und Blattwort ω existiert. Man schreibt:*

$$n \longrightarrow_G^* \omega$$

Wir können mit Hilfe dieser Notation auch die Ableitung eines Blattwortes aus einem anderen gemäß der Konstruktion der Ableitungsbäume formalisieren.

Definition 4.4 *Seien $\omega_1, \omega_2, \omega_3 \in (N \cup T)^*$ und $m \in N$. Gilt $m \longrightarrow_G \omega_3$ und lässt sich $n \longrightarrow_G^* \omega_1 m \omega_2$ ableiten, so existiert auch die Ableitung*

$$n \longrightarrow_G^* \omega_1 \omega_3 \omega_2.$$

Dies folgt aus der Konstruktion der Ableitungsbäume. Die Relation $\Rightarrow_G \subseteq (N \cup T)^* \times (N \cup T)^*$ kodiert diesen Sachverhalt. Für das obige Beispiel gilt:

$$\omega_1 m \omega_2 \Rightarrow_G \omega_1 \omega_3 \omega_2$$

Definition 4.5 *Die Sprache $L \subseteq T^*$, die durch G erzeugt wird ist:*

$$L(G) = \{\omega \mid S \longrightarrow_G^* \omega,\ \omega \in T^*\},$$

also alle Folgen von Terminalen die aus dem Startsymbol abgeleitet werden können.

Die obige Beispielgrammatik würde beispielsweise die Sprache $L(G) = \{Xu \mid u \in \{0,1,\ldots,9\}^*\}$ realisieren.

Beispiel: Vollständig geklammerte Boolesche Ausdrücke:

$$\begin{array}{lcll}
T & = & \{X, 0, 1, \ldots, 9, \wedge, \vee, \sim, \oplus, (,)\} & \\
N & = & \{C, \langle CF\rangle, V, \langle BA\rangle\} & \\
S & = & \langle BA\rangle & \\
P & : & C & \longrightarrow 0 \mid 1 \mid \cdots \mid 9 \\
 & & \langle CF\rangle & \longrightarrow C \mid \langle CF\rangle C \\
 & & V & \longrightarrow X \mid X\langle CF\rangle \\
 & & \langle BA\rangle & \longrightarrow V \mid 0 \mid 1 \mid (\sim\langle BA\rangle) \mid (\langle BA\rangle \wedge \langle BA\rangle) \mid (\langle BA\rangle \vee \langle BA\rangle) \mid (\langle BA\rangle \oplus \langle BA\rangle)
\end{array}$$

Ebenso lässt sich eine cfg für arithmetische Ausdrücke angeben.
Beispiel: Vollständig geklammerte arithmetische Ausdrücke:

$$\begin{array}{lcll}
T & = & \{X, 0, 1, \ldots 9, +, -_2, \cdot, /, -_1, (,)\} & \\
N & = & \{C, \langle CF\rangle, V, A\} & \\
S & = & A & \\
P & : & C & \longrightarrow 0 \mid 1 \mid \cdots \mid 9 \\
 & & \langle CF\rangle & \longrightarrow C \mid \langle CF\rangle C \\
 & & V & \longrightarrow X \mid X\langle CF\rangle \\
 & & A & \longrightarrow V \mid \langle CF\rangle \mid (-_1 A) \mid (A + A) \mid (A -_2 A) \mid (A \cdot A) \mid (A/A)
\end{array}$$

Dabei steht $-_1$ für das unäre und $-_2$ für das binäre Minus. Man muss die diese beiden verschiedenen Minusoperatoren einführen da man beide nicht unterscheiden kann, ohne den Kontext zu betrachten. Die vielen Klamern in den Ausdrücken erscheinen unbequem und wir versuchen daher eine Grammatik für unvollständig geklammerte arithmetische Ausdrücke zu finden. Dazu ersetzen wir die letzte Zeile der Produktionsregeln durch:

$$A \longrightarrow V \mid \langle CF\rangle \mid -_1 A \mid A + A \mid A -_2 A \mid A \cdot A \mid A/A \mid (A)$$

Wie die Abbildungen 4.4 und 4.5 zeigen, ergeben sich mit dieser Grammatik allerdings Eindeutigkeitsprobleme. Für das gleiche Blattwort finden sich zwei syntaxkonforme Ableitungsbäume mit der gleichen Wurzel.

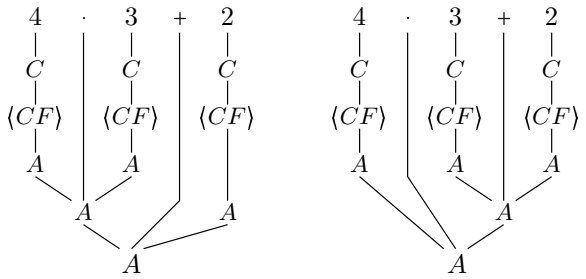


Abbildung 4.4: Ableitungsbäume für uneindeutige Grammatik

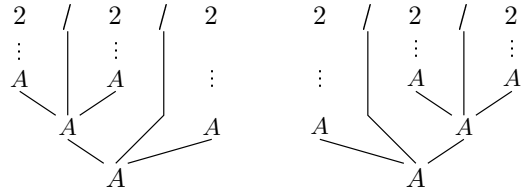


Abbildung 4.5: Ableitungsbäume für uneindeutige Grammatik

Definition 4.6 *Eine Grammatik G heißt eindeutig, falls:*

$$\forall \omega \in L(G) : \overset{1}{\exists} \text{ Ableitungsbaum } T \text{ mit Wurzel } S \text{ und Blattwort } \omega$$

Die vorliegende Grammatik für unvollständig geklammerte Ausdrücke ist offensichtlich uneindeutig. Es existiert jedoch eine eindeutige Variante davon. Man muss zunächst weiter Nichtterminale $\langle F \rangle$ für Faktoren und $\langle T \rangle$ für Terme einführen. Dann definiert man folgendes Produktionensystem:

$$\begin{array}{lcl} \langle F \rangle & \longrightarrow & V \mid \langle CF \rangle \mid -_1\langle F \rangle \mid (\langle A \rangle) \\ \langle T \rangle & \longrightarrow & \langle F \rangle \mid \langle T \rangle \cdot \langle F \rangle \mid \langle T \rangle / \langle F \rangle \\ \langle A \rangle & \longrightarrow & \langle T \rangle \mid \langle A \rangle + \langle T \rangle \mid \langle A \rangle -_2 \langle T \rangle \end{array}$$

Theorem 4.1 *Diese Grammatik ist eindeutig*

Beweis: Der Beweis dafür findet sich in [LMW86] auf Seite 111.

Die Abbildungen 4.6 und 4.7 verdeutlichen, dass nun keine zweideutigen Ableitungen wie in den Beispielen zuvor möglich sind. Versuche, einen alternativen Ableitungsbaum zu konstruieren, enden in einer „Sackgasse“. Es gibt keine zweite Möglichkeit, einen Ausdruck auf das Startsymbol zurückzuführen.

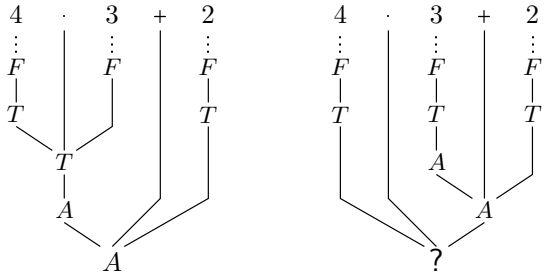


Abbildung 4.6: Ableitung mit eindeutiger Grammatik

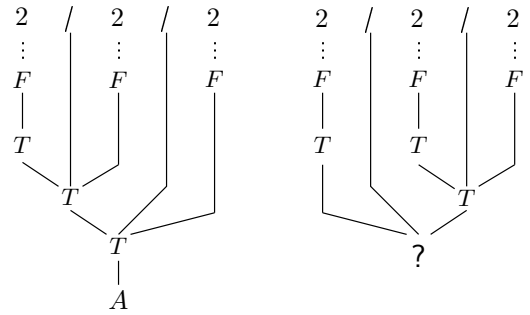


Abbildung 4.7: Ableitung mit eindeutiger Grammatik

Die vorliegende eindeutige Grammatik wertet wie gewohnt von links nach rechts und nach der „Punkt-vor-Strich“-Regel aus. Wenn man nun die Addition der Multiplikation vorziehen will, so muss man Klammern setzen (siehe Abbildung 4.8).

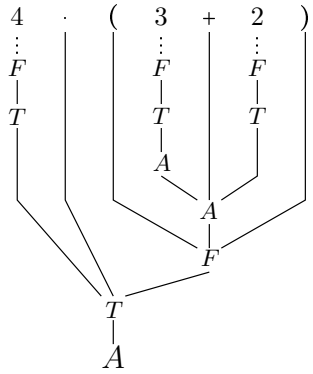


Abbildung 4.8: eindeutige Ableitung mit Klammersetzung

Nun soll eine Grammatik für die Programmiersprache $C0$ erstellt werden.

4.2 C0-Syntax

Tabelle 4.1 zeigt die Produktionsregeln der kontextfreien Grammatik von $C0$. Sämtliche Symbole in

$\langle Zi \rangle$	$\longrightarrow$	$0 \mid \ldots \mid 9$	Ziffer
$\langle ZiF \rangle$	$\longrightarrow$	$\langle Zi \rangle \mid \langle ZiF \rangle \langle Zi \rangle$	Ziffernfolge
$\langle Bu \rangle$	$\longrightarrow$	$a \mid \ldots \mid z \mid A \mid \ldots \mid Z$	Buchstabe
$\langle BuZi \rangle$	$\longrightarrow$	$\langle Bu \rangle \mid \langle Zi \rangle$	alphanumerisches Zeichen
$\langle BuZiF \rangle$	$\longrightarrow$	$\langle BuZi \rangle \mid \langle BuZiF \rangle \langle BuZi \rangle$	alphanumerische Zeichenfolge
$\langle Na \rangle$	$\longrightarrow$	$\langle Bu \rangle \mid \langle Bu \rangle \langle BuZiF \rangle$	Name
$\langle C \rangle$	$\longrightarrow$	$\langle ZiF \rangle$	Konstante
$\langle id \rangle$	$\longrightarrow$	$\langle Na \rangle \mid \langle id \rangle.\langle Na \rangle \mid \langle id \rangle[\langle A \rangle] \mid \langle id \rangle* \mid \langle id \rangle\&$	Identifier ((Sub-)Variablenbezeichner)
$\langle F \rangle$	$\longrightarrow$	$\langle id \rangle \mid -_1\langle F \rangle \mid (\langle A \rangle) \mid \langle C \rangle$	Faktor
$\langle T \rangle$	$\longrightarrow$	$\langle F \rangle \mid \langle T \rangle \cdot \langle F \rangle \mid \langle T \rangle / \langle F \rangle$	Term
$\langle A \rangle$	$\longrightarrow$	$\langle T \rangle \mid \langle A \rangle + \langle T \rangle \mid \langle A \rangle -_2 \langle T \rangle$	(algebraischer) Ausdruck
$\langle Atom \rangle$	$\longrightarrow$	$\langle A \rangle > \langle A \rangle \mid \langle A \rangle \geq \langle A \rangle \mid \langle A \rangle < \langle A \rangle \mid \langle A \rangle \leq \langle A \rangle \mid$ $\langle A \rangle == \langle A \rangle \mid \langle A \rangle \neq \langle A \rangle \mid 0 \mid 1$	"Boolesche Variable"
$\langle BF \rangle$	$\longrightarrow$	$\langle id \rangle \mid \langle Atom \rangle \mid \sim\langle BF \rangle \mid (\langle BA \rangle)$	Boolescher Faktor
$\langle BT \rangle$	$\longrightarrow$	$\langle BF \rangle \mid \langle BT \rangle \wedge \langle BF \rangle$	Boolescher Term
$\langle BA \rangle$	$\longrightarrow$	$\langle BT \rangle \mid \langle BA \rangle \vee \langle BT \rangle$	Boolescher Ausdruck
$\langle An \rangle$	$\longrightarrow$	$\langle id \rangle = \langle A \rangle \mid \langle id \rangle = \langle BA \rangle \mid \langle id \rangle = '\langle BuZi \rangle' \mid$	Zuweisung
		$if\ \langle BA \rangle\ then\ \{\langle AnF \rangle\}\ else\ \{\langle AnF \rangle\} \mid$	bedingte Anweisung
		$if\ \langle BA \rangle\ then\ \{\langle AnF \rangle\} \mid$	
		$while\ \langle BA \rangle\ do\ \{\langle AnF \rangle\} \mid$	Schleife
		$\langle id \rangle = \langle Na \rangle(\langle PaF \rangle) \mid$	Funktionsaufruf mit Parametern
		$\langle id \rangle = new\ \langle Na \rangle*$	Speicher allozieren
$\langle PaF \rangle$	$\longrightarrow$	$\varepsilon \mid \langle ParF \rangle$	Parameterfolge
$\langle ParF \rangle$	$\longrightarrow$	$\langle A \rangle \mid \langle ParF \rangle, \langle A \rangle$	nicht-leere Parameterfolge
$\langle AnF \rangle$	$\longrightarrow$	$\langle An \rangle \mid \langle AnF \rangle; \langle An \rangle$	Anweisungsfolge
$\langle program \rangle$	$\longrightarrow$	$\langle TyDF \rangle \langle VaDF \rangle \langle FunDF \rangle$	$C0$-Programm
$\langle TyDF \rangle$	$\longrightarrow$	$\varepsilon \mid \langle TypD \rangle; \langle TyDF \rangle$	Typdeklarationsfolge
$\langle TypD \rangle$	$\longrightarrow$	$typedef\ \langle Typ \rangle\ \langle Na \rangle$	Typdeklaration
$\langle Typ \rangle$	$\longrightarrow$	$\langle Na \rangle \mid \langle Na \rangle[\langle ZiF \rangle] \mid \langle Na \rangle* \mid$	Typnamen, Array-Typ, Pointer
		$struct\{\langle VarDF \rangle\}$	struct-Typ
$\langle VaDF \rangle$	$\longrightarrow$	$\varepsilon \mid \langle VarD \rangle; \langle VaDF \rangle$	Variablendeklarationsfolge
$\langle VarDF \rangle$	$\longrightarrow$	$\langle VarD \rangle \mid \langle VarD \rangle; \langle VarDF \rangle$	nicht-leere Variablendeklarationsfolge
$\langle VarD \rangle$	$\longrightarrow$	$\langle Na \rangle\ \langle Na \rangle$	Variablendeklaration
$\langle FunDF \rangle$	$\longrightarrow$	$\langle FunD \rangle \mid \langle FunDF \rangle; \langle FunD \rangle$	nicht-leere Funktionsdeklarationsfolge
$\langle FunD \rangle$	$\longrightarrow$	$\langle Na \rangle\ \langle Na \rangle(\langle PaDF \rangle)\{\langle VaDF \rangle \langle rumpf \rangle\}$	Funktionsdeklaration
$\langle PaDF \rangle$	$\longrightarrow$	$\varepsilon \mid \langle ParDF \rangle$	Parameterdeklarationfolge
$\langle ParDF \rangle$	$\longrightarrow$	$\langle VarD \rangle \mid \langle VarD \rangle, \langle PaDF \rangle$	nicht-leere Parameterdeklarationfolge
$\langle rumpf \rangle$	$\longrightarrow$	$\langle AnF \rangle; return\ \langle A \rangle \mid return\ \langle A \rangle$	Funktionsrumpf

Tabelle 4.1: Grammatik von $C0$

eckigen Klammern stellen Nichtterminale dar. Die Terminale von $C0$ sind Buchstaben und Zahlen sowie daraus bestehende Schlüsselworte (if, $then$, $while$, $typedef$,...), Operatoren (+, & , ∨, ∗, ...) und weitere Sonderzeichen ({,},;,=,...). Das Startsymbol ist $\langle program \rangle$. Daraus kann eine Folge von Variablen-, Funktions-, und Typdeklarationen abgeleitet werden. Um die Ähnlichkeit zu C zu gewährleisten, wird die Vereinbarung getroffen, dass das Hauptprogramm in einer Funktion namens *main* definiert wird. Diese Funktion wird also in allen $C0$-Programmen deklariert. Funktionsdeklarationen beginnen allgemein mit dem Typ des Rückgabewertes. Danach folgen der Funktionsname, die Deklarationsfolge der zu übergebenden Parameter, die Deklarationsfolge für lokale Variablen und der Rumpf. Funktionsrümpfe enthalten Anweisungen, wie Zuweisungen, Schleifen, Funktionsaufrufe und if-

then-else-Befehle. Bei letzteren sind die geschweiften Klammern um die Anweisungsfolgen zu beachten. Würde man die Klammern weglassen, so könnte man Ausdrücke wie

$$if\ \langle BA\rangle\ then\ if\ \langle BA\rangle\ then\ \langle An\rangle\ else\ \langle An\rangle$$

nicht eindeutig auswerten. Man wüsste nicht welcher if-Anweisung der else-Teil zugeordnet werden sollte. Stattdessen schreibt man

$$if\ \langle BA\rangle\ then\ \{\ if\ \langle BA\rangle\ then\ \{\langle An\rangle\}\ else\ \{\langle An\rangle\}\ \}$$

beziehungsweise

$$if\ \langle BA\rangle\ then\ \{\ if\ \langle BA\rangle\ then\ \{\langle An\rangle\}\ \}\ else\ \{\langle An\rangle\}.$$

Es fällt auf, dass boolesche Ausdrücke in dieser Grammatik auch einfache identifier, also boolesche Variablen sein können. Dies spart Schreibweisen wie z.B. `if x=1 then ...`. Stattdessen darf direkt `if x then ...` geschrieben werden. Solche Konstrukte die nur dem komfortableren Programmieren dienen, aber keine zusätzliche Funktionalität erzielen, bezeichnet man als *syntaktischen Zucker*. Unglücklicherweise macht dieser zusätzliche Komfort die Grammatik uneindeutig. Da Identifier sowohl arithmetische als auch boolesche Ausdrücke darstellen können, exisitieren für Zuweisungen stehts zwei Ableitungsbäume. Man muss zusätzliche Kontextbedingungen und Typ-Überprüfungen einführen, um den passenden Baum zu bestimmen und Eindeutigkeit zu erzielen.

Als Beispiel für eine Funktionsdeklaration sei der folgende Code gegeben, der die Fakultät von x berechnet:

```
int fak(int x)
{
  int y;
  if x==1 then {y=x} else
  {
    y=fak(x-1);
    y=x*y
  };
  return y
}
```

Man beachte, dass Funktionsaufrufe nur in der Form $\langle id\rangle$ = $\langle Na\rangle(\langle PF\rangle)$ möglich sind. Es ist nicht erlaubt, Ausdrücke zu konstruieren, die Funktionsaufrufe enthalten. Dies hat den Grund, dass Funktionen nicht nur auf lokale Variablen zugreifen, sondern auch globale Variablen verändern können. Ist y eine globale Variable, die von $F()$ verändert zurückgegeben wird, so würde der Effekt der Zuweisung $Z = F() * y$ davon abhängen, in welcher Reihenfolge der Ausdruck auf der rechten Seite ausgewertet wird. In $C0$ sollen Teilausdrücke aber keine Seiteneffekte haben, so dass die Reihenfolge der Ausdrucksauswertung beliebig ist. Deshalb sind solche Konstrukte schon durch die Syntax nicht gestattet.

Eine weitere Auffälligkeit ist das Fehlen einer $delete$-Anweisung zum Löschen von Datenstrukturen auf dem heap. Um Speicherlecks zu vermeiden nehmen wir an, dass der Compiler einen *garbage collection*-Algorithmus in alle Programme integriert, der dafür sorgt, das nicht mehr referenzierte heap-Variablen entfernt werden.

Im nächsten Abschnitt soll näher auf die Typdeklarationen in $C0$ eingegangen werden.

4.3 Typdeklarationen

$C0$ verwendet das wohldefinierte Typensystem von *Pascal*. Es gibt die folgenden Arten von Typen:

- elementare Typen: int, $bool$, $char$, $unsigned$
- array-Typen: z.B. $int[15]$, $xyz[3]$
- pointer-Typen: z.B. $int*$
- struct-Typen: siehe unten

Eigene Typen lassen sich per $typedef$ konstruieren. Man versieht dabei eine struct/array/pointer-Kombination von vorhanden Typen mit einem Alias, beispielsweise

```
typedef int[14] xyz;
xyz a
```

wobei ein array ein eindimensionales Feld von Variablen des jeweiligen Typs im Speicher darstellt. Im obigen Beispiel würde die Variable `a` vom neu definierten Typ `xyz` einen Vektor aus 14 int-Elementen `a[0]`,...,`a[13]` repräsentieren. Es folgt die Deklaration einer 7x7-Matrix ganzer Zahlen A:

```
typedef int[7] row;
typedef row[7] matrix;
matrix A
```

Struct-Typen, auch Record-Typen genannt, bilden eine Datenstruktur aus bereits bekannten Typen. Eine typische Anwendung für solche Strukturen sind zum Beispiel einfach verkettete Listen, wie sie in Abbildung 4.9 und 4.10 zu sehen sind. Dabei enthält jedes Listenelement ein $content$-Feld, dass den

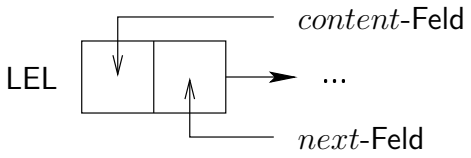


Abbildung 4.9: Listenelement

Inhalt des Elements speichert, sowie einen Zeiger $next$ auf das nächste Listenelement. Dazu muss man zwei Typen deklarieren:

- LEL - das Listenelement
- $LEL*$ - ein Zeiger (Pointer) auf ein Listenelement

In $C0$ wäre man verleitet, dies wie folgt umzusetzen:

```
typedef struct{int content; u next} LEL;
typedef LEL* u
```

Das führt jedoch auf ein Problem, da die Definitionen rekursiv verschränkt sind. Es wäre wünschenswert, dass $LEL*$ als u deklariert wird, bevor man es in der Strukturdeklaration verwendet, oder aber LEL deklariert ist, bevor man einen Pointer darauf setzt. Beides führt zu Widersprüchen, da man nun einmal mit einer Deklaration beginnen muss. $C0$ erlaubt es nun, dass man Pointer auf Typen setzen darf, die noch garnicht deklariert wurden. Im vorliegenden Beispiel müsste man also schreiben:

```
typedef LEL* u;
typedef struct{int content; u next} LEL
```

Bei der Deklaration von u ist LEL noch nicht bekannt. Trotzdem darf man bereits einen Pointer darauf deklarieren. Ansonsten dürfen neue Typen nur aus bereits bekannten Typen konstruiert werden. Außerdem sind Variablendeklarationen nur mit deklarierten Typen möglich. Es gibt also keine anonymen Typen in einem $C0$-Programm.
Es ist zu beachten, dass arrays und structs beliebig gemischt werden können. Man spricht von *vollrekursiven Datentypen*. Später wird zur Verwaltung von Benutzerprozessen beispielsweise ein array von PCBs (**P**rocess **C**ontrol **B**lock) angelegt werden. Diese sind wiederum structs zur Speicherung der CPU-Konfiguration und enthalten Komponenten, wie $.pc$ oder auch $.gpr$, welches ein array von int ist.

Das Beispiel der verketteten Liste zeigt, dass zur Definition von $C0$ die kontextfreie Grammatik allein nicht ausreichend ist. Man benötigt zusätzliche Kontextbedingungen, wie:

- Variablen müssen deklariert werden, bevor auf sie zugegriffen wird
- bei der i-ten Typdeklaration unterscheidet man die Fälle:
  - $\left.\begin{array}{c} typedef\ T\ \langle Na\rangle \\ typedef\ T[\langle ZiF\rangle]\ \langle Na\rangle \\ typedef\ struct\{\ldots;T\ \langle Na\rangle;\ldots\}\ \langle Na\rangle \end{array}\right\} \Rightarrow$ T ist elementar oder in j-ter Typdeklaration definiert mit $j < i$.
  - $typedef\ T*\ \langle Na\rangle \Rightarrow T$ darf in j-ter Typdeklaration definiert werden auch für $j \geq i$.

Das Beispiel der verketteten Liste in Abbildung 4.10 bietet Anwendungsmöglichkeiten für weitere identifier. Mit $x.content$ und $x.next$ greift man auf die einzelnen Komponenten der LEL-Struktur zu. Der

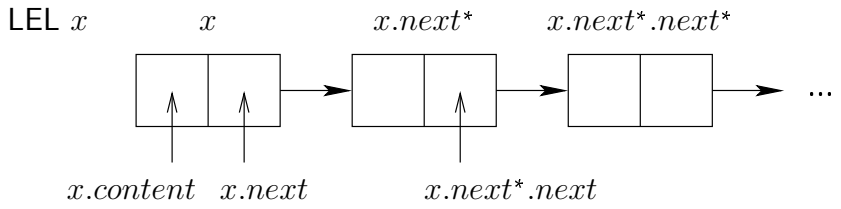


Abbildung 4.10: einfach verkettete Liste

$*$-Operator dereferenziert Pointer und gibt den Inhalt der Variable, auf die gezeigt wird, zurück. So ist $x.next*$ das Listenelement, auf das der $next$-Pointer von x zeigt. Um weitere Listenelemente zu erzeugen, kann der new-Befehl genutzt werden. Man deklariert:

```
LEL* p;
p=new LEL*
```

Dabei erzeugt new eine neue namenlose Variable vom Typ LEL und weist p einen Pointer auf diese Variable zu. Will man nun $p*$ mit einem weiteren Listenelement $q*$ wie in Abbildung 4.11 gezeigt verketten, so setzt man:

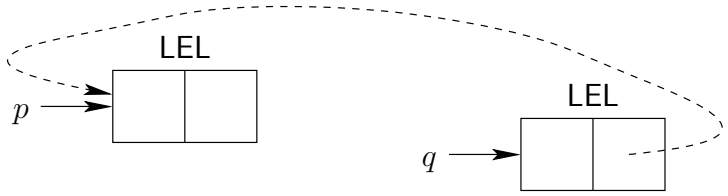


Abbildung 4.11: Konstruktion einer Liste mit new

```
LEL* q;
q=new LEL*;
q*.next=p
```

4.4 C0-Semantik

In Lehrbüchern finden sich viele verschiedene Varianten, die Semantik von Programmiersprachen zu definieren. Man unterscheidet dabei:

- *big step*-Semantik
- *small step*-Semantik

big step-Semantik beschreibt die Wirkung von Programmen, die induktiv über die Ableitungsbäume abgeleitet werden. Im Allgemeinen lassen sich Beweise über die Semantik in der big step-Vorgehensweise leichter und eleganter führen. Daher wechselt man zur detaillierteren small step-Semantik nur, wenn es unbedingt nötig ist (Behandlung von I/O, interleaving, etc.). Dies geht in Ordnung, da Konsistenzbeweise für beide Varianten zueinander existieren.
Im Folgenden betrachten wir die Abbildung $p \mapsto K_p$, wobei p ein Programm und K_p die zugehörige Menge der Konfigurationen einer abstrakten $C0$-Maschine darstellt. Offensichtlich hängt K_p nur von den Deklarationen in p ab. Analog zu δ_D und δ_H definieren die Übergangsfunktion für $C0$-Konfigurationen:

$$\begin{aligned} \delta_C &: K_p \longrightarrow K_p \\ c' &= \delta_C(c) \end{aligned}$$

c' ist die Konfiguration, die aus c durch Ausführen einer Anweisung entsteht. Wir werden nun zunächst den statischen Teil der Konfiguration betrachten, der zur Compilezeit feststeht, also aus Typ-, Variablen- und Funktionsdeklarationen hervorgeht. Desweiteren wird im weiteren Verlauf der dynamische Teil der $C0$-Konfiguration eingeführt und Ausdrucksauswertung sowie Anweisungsausführung behandelt werden. Am Ende des Semantikabschnitts soll die Korrektheit einiger Beispielprogramme bewiesen werden.

4.4.1 Typdeklarationen

Die Menge der Namen aller deklarierten Typen inklusive int, $bool$, $char$ und $unsigned$ wird mit TN bezeichnet. Zur Verwaltung der Typen legen wir eine Typtabelle tt an. Dorthin werden alle Typdeklarationen abgebildet.

$$tt = (tt.o, tt.tc)$$

Dabei ist:

- $tt.o : TN \longrightarrow \mathbb{N}$ - die Reihenfolge der Typdeklarationen. $tt.o(t) = i + 3$ für $i \in \mathbb{N}^+$ bedeutet, dass Typ t in der i-ten Typdeklaration definiert wird.
  - $tt.o(int) = 0$
  - $tt.o(bool) = 1$
  - $tt.o(char) = 2$
  - $tt.o(unsigned) = 3$
  - $tt.o(\text{Name in } i\text{–ter Typdeklaration}) = i + 3$
- $tt.tc : TN \longrightarrow TD$, $TD = \{\text{Typdeskriptoren}\}$ - Beschreibung der Typen (**t**ype **c**ontent)

Um zu definieren, wie die Typtabelle ausgefüllt werden soll, muss eine Fallunterscheidung über die i-te Typdeklaration getroffen werden:

- $typedef\ t'[n]\ t$ (array-Typen) mit $t' \in TN$
  - $tt.o(t) = i + 3$
  - $tt.tc(t) = t'[n],\ tt.o(t') < tt.o(t)$

- $typedef\ struct\{t_1\ n_1; \ldots; t_s\ n_s\}\ t$ (struct-Typen), so dass $\forall i \in [1:s].\ t_i \in TN$
  - $tt.o(t) = i + 3$
  - $tt.tc(t) = struct\{t_1\ n_1; \ldots; t_s\ n_s\},\ \forall j \in [1:s].\ tt.o(t_j) < tt.o(t)$
- $typedef\ t' *\ t$ (pointer-Typen) mit $t' \in TN$
  - $tt.o(t) = i + 3$
  - $tt.tc(t) = t'*,\ tt.o(t') > tt.o(t)$ erlaubt

Für die pointer-Typen ist explizit erlaubt, dass $tt.o(t') > i + 3$ sein darf. Deshalb muss die Konstruktion der Typtabelle in zwei Pässen erfolgen. Im ersten Pass werden die Typdeskriptoren der pointer-Typen freigelassen und erst im zweiten Pass, wenn alle nachfolgenden Typen eingetragen wurden, eingesetzt. Desweiteren implizieren die obigen Definitionen eine weitere Kontextbedingung, nämlich, dass Typen aus verschiedenen Typdeklarationen verschiedene Namen tragen müssen. Ansonsten könnte man die Tabelle nicht eindeutig aufstellen.

Tabelle 4.2 zeigt die Typtabelle für folgende Beispieldeklarationen:

```
typedef int[5] x;
typedef x[5] row;
typedef LEL* u;
typedef struct{int content; u next} LEL
```

$tt.o(t)$	$t \in TN$	$tt.tc(t)$
0	int	int
1	$bool$	$bool$
2	$char$	$char$
3	$unsigned$	$unsigned$
4	x	$int[5]$
5	row	$x[5]$
6	u	$LEL*$
7	LEL	$struct\{int\ content, u\ next\}$

Tabelle 4.2: Typtabelle für Beispieldeklaration und Größe der Typen

Es stellt sich nun die Frage, wie man auf Komponenten komplexer Typen zugreifen kann. Dazu betrachtet man zunächst den Wertebereich $\mathrm{Range}(t)$, kurz $Ra(t)$, eines Typs. Eine informelle Beschreibung könnte lauten:

$$\mathrm{Range}(t) = \{\omega \mid \text{Variablen vom Typ } t \text{ koennen den Wert } \omega \text{ annehmen}\}$$

Dies ist jedoch keine korrekte Definition, da weder die Begriffe „Variable“ noch „Wert“ bisher definiert wurden. Für die elementaren Typen lässt sich $Ra(t)$ noch leicht festlegen. Um näher an der Hardware-Zahlendarstellung zu sein, definieren wir die Werte als Bitstrings.

$$\begin{aligned} Ra(int) = Ra(unsigned) &= \{0,1\}^{32} \\ Ra(bool) &= \{0^{32}, 0^{31}1\} \\ Ra(char) &= \{0^{32}, \ldots, 0^{24}1^8\} \end{aligned}$$

Dies hat auch den Vorteil, dass wir bei der Ausdrucksauswertung nicht die Typen der Variablen betrachten müssen. Der Wert von Pointern ist noch undefiniert, jedoch wird sich später zeigen, dass Pointer Variablen als Wert besitzen. Dies bedeutet insbesondere, dass in $C0$ keine pointer-Arithmetik möglich ist. Für komplexe t unterscheiden wir die zwei Fälle:

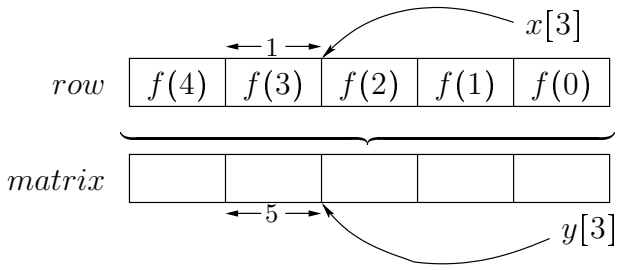


Abbildung 4.12: Zugriff auf Komponenten von Typen

- $tt.tc(t) = t'[n]$ - Der Wertebereich von t entspricht allen möglichen Belegungen für die Elemente des Arrays. Die Belegungen werden durch Funktionen f repräsentiert, die jedem Arrayelement den entsprechenden Wert zuordnen.

$$Ra(t) = \{f : [0 : n-1] \longrightarrow Ra(t')\}$$

- $tt.tc(t) = struct\{t_1\ n_1; \ldots; t_s\ n_s\}$ - Hier wählen wir für die Darstellung der Belegungen der struct-Komponenten ebenfalls Funktionen f, die die Namen der Komponenten auf ihren Wertebereich abbilden.

$$Ra(t) = \{f : \{n_1, \ldots, n_s\} \longrightarrow \bigcup_{i=1}^{s} Ra(t_i) \mid \forall i \in [1:s].\ f(n_i) \in Ra(t_i)\}$$

Durch diese elegante Definition lassen sich Komponentenzugriffe auf die Auswertung von Funktionen reduzieren. Abbildung 4.12 verdeutlicht die Vorgehensweise für die folgende Beispieldeklaration, wenn man zum Beispiel auf die Komponenten $y[3]$ oder $z[3]$ zugreifen möchte.

```
typedef int[5] row;
typedef row[5] matrix;
row y;
matrix z;
```

Mit den Funktionen f lässt sich also auf Teilstrukturen eines Typs zugreifen, indem man f auf einen entsprechenden Index anwendet.

4.4.2 Variablendeklarationen

Dieser Abschnitt beschäftigt sich mit dem Speicher und der Repräsentation von Variablen bzw. Subvariablen darin. Variablen bezeichnen dabei Plätze im Speicher an denen Werte stehen können. Die Menge aller Variablennamen wird mit VN bezeichnet. Wir definieren eine formale Beschreibung von Speicher, den ein Programm nutzt, ähnlich zur Typtabelle eines Programms.

Definition 4.7 *(memory) Ein memory m besteht aus Symboltabelle $m.st$ und den enthaltenen Werten $m.va$.*

$$m \quad = \quad (m.st, m.va)$$

Eine Symboltabelle $st = (st.na, st.typ, st.f)$ enthält die folgenden Bestandteile.

- $st.na \subset VN$ *- eine endliche Menge von Variablen die in st aufgeführt sind*
- $st.typ : VN \longrightarrow TN$ *- weist allen $x \in st.na$ den Namen des Typs von x zu*
- $st.f \in FN \cup \{gm, hm\}$ *- der Name der $C0$-Funktion, deren Variablen und Parameter in m gespeichert werden, FN bezeichnet die Menge aller Funktionsnamen, gm und hm werden als Sonderfälle für das globale und das heap memory verwendet.*

Die Speicherinhalte liegen im Wertebereich der jeweiligen Variablen $x \in m.st.na$.

$$m.va(x) \in Ra(m.st.typ(x))$$

Tabelle 4.3 zeigt die den resultierenden Speicher für die folgende globale Beispieldeklaration. Dabei werden die Typdeklarationen aus vorangegangenen Beispielen verwendet. Es ist zu beachten, dass nur Variablen von solchen Typen definiert werden dürfen, die in der Typtabelle stehen, also vorher definiert wurden. Dies ist eine weitere Kontextbedingung an $C0$.

```
int y;
matrix z;
u p;
```

$x \in m.st.na$	$m.st.typ(x)$	$m.va(x)$
y	int	$i \in \{0,1\}^{32}$ beliebig
z	$matrix$	$f \in \{g : [0:6] \longrightarrow Ra(row)\}$ beliebig
p	u	(m,x) beliebig

Tabelle 4.3: Symboltabelle $m.st$ und Inhalt $ct.m$ für Beispieldeklarationen

Da die Variablen nocht nicht initialisiert sind, haben sie zunächst beliebige Werte im jeweiligen Wertebereich. Nun sollen Variablen und Subvariablen formal definiert werden.

Definition 4.8 *(Variable) Das Paar* (m,x) *repräsentiert die Variable mit Namen* x *in Speicher* m. *Dabei ist*

- m *der Name eines memory*
- $x \in m.st.na$ *ein Name aus der Symboltabelle von* m

Definition 4.9 *(Typ von Variablen) Der Typ* $typ(v)$ *einer Variable* $v = (m,x)$ *wird wie folgt definiert.*

$$typ(v) = m.st.typ(x)$$

Wir wollen außerdem über Subvariablen argumentieren.

Definition 4.10 *(Selektoren) Zunächst definieren wir Selektoren* s_j. *Diese sind entweder*

- $[k]$ *mit* $k \in \mathbb{N}$ *oder*
- $.n_k$ *mit einem Namen* n_k.

Dann ist eine Subvariable eine Variable gefolgt von $t \geq 0$ von Selektoren s_j:

$$(m,x)s_1 \cdots s_t$$

oder kurz $(m,x)s$ mit $s = s_1 \cdots s_t$.
Zum Beispiel würde die Subvariable $(m,PCB)[17].gpr[5]$ eine Komponente eines process control blocks darstellen. Die Anzahl der Selektoren kann auch Null sein, also $s = \varepsilon$, das bedeutet, dass Variablen auch Subvariablen sind.

Definition 4.11 *(Subvariablen von* m*) Das Prädikat* $sub(v',v)$ *codiert die Beziehung "*v' *ist Subvariable von* v*".Wir unterscheiden die folgenden Fälle:*

- *Es gilt* $sub((m,x),(m,x))$ *- Variablen sind Subvariablen von sich selbst*
- *Es gelte* $sub(v',v)$.

– *Falls* $typ(v') = t'[n]$*, dann gilt für alle* $k \in [0 : n-1]$*:*

$$sub(v'[k], v) \qquad typ(v'[k]) = t'$$

– *Falls* $typ(v') = struct\{t_1\ n_1; \ldots; t_s\ n_s\}$*, dann gilt für alle* $i \in [1 : s]$*:*

$$sub(v'.n_i, v) \qquad typ(v'.n_i) = t_i$$

Wir können nun die Werte von Pointervariablen definieren. Pointer zeigen auf Subvariablen. Für $t = t'*$ gilt:

$$Ra(t) = \{(m,x)s \mid sub((m,x)s, (m,x)) \wedge typ((m,x)s) = t'\} \cup \{null\}$$

Dabei ist m ein Speichername und s eine Folge von Selektoren. $null$ bezeichnet einen Pointer ohne Ziel. Man beachte, dass aus dieser Definition folgt, dass in $C0$ keine Pointerarithmetik möglich ist, da Pointer auf Subvariablen zeigen und nicht auf Adressen, zu denen man offsets addieren kann.

4.4.3 Funktionsdeklarationen

Zuletzt werden die Deklarationen von Funktionen betrachtet. FN bildet die Menge der Funktionsnamen in einem $C0$-Programm:

$$FN = \{\text{deklarierte Funktionsnamen}\}$$

Über diese Funktionen wird in einer Funktionstabelle Buch geführt:

$$ft : FN \longrightarrow FD$$

Dabei ist FD die Menge der Funktionsdeskriptoren. Diese enthalten alle Informationen, die zur Beschreibung einer $C0$-Funktion nötig sind.

$$ft(f) = (ft(f).st, ft(f).body, ft(f).np, ft(f).par)$$

Das wären im Einzelnen:

- $ft(f).st$ - Symboltabelle für Funktion f:
  - $ft(f).st.na = \{rds, p_1, \ldots, p_{ft(f).np}, x_1, \ldots, x_l\}$ - Namen der Parameter und der lokalen Variablen von Funktion f,
    rds (return destination) steht für die Zielvariable, in der der Rückgabewert von f gespeichert werden soll, $p_1, \ldots, p_{ft(f).np}$ für die Parameter und $x_1, \ldots, x_l$ für die lokalen Variablen.
  - $ft(f).st.typ$ - Typen der Parameter und der lokalen Variablen,
    $tt.tc(ft(f).st.typ(rds)) = rtyp(f)*$, wobei $rtyp(f)$ den Rückgabetyp der Funktion f beschreibt und über den Ableitungsbaum der Funktionsdeklaration definiert ist. Der Typ $rtyp(f)*$ muss dementsprechend in der Typtabelle aufgeführt sein.
  - $ft(f).st.f = f$ - Name der Funktion
- $ft(f).body$ - Rumpf der Funktion f (Anweisungsfolge abgeleitet von $\langle rumpf \rangle$)
- $ft(f).np$ - Anzahl der Parameter von f
- $ft(f).par : [1 : ft(f).np] \longrightarrow \underbrace{\{p_1, \ldots, p_{ft(f).np}\}}_{\subset ft(f).st.na}$ - Reihenfolge der Parameter von f

Man beachte, dass die Symboltabelle $ft(f).st$ strukturgleich zu der Symboltabelle $m.st$ eines Speichers m ist. Mit den obigen Informationen, lässt sich eine Funktion vollständig beschreiben. Das Ergebnis hängt dann nur noch von den übergebenen Parametern und globalen Variablen ab.

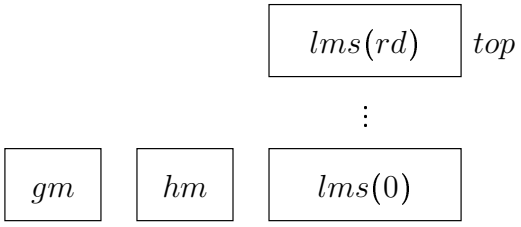


Abbildung 4.13: Komponenten der $C0$-Konfiguration

4.4.4 Konfiguration von C0-Maschinen

Nun, da die statischen Rahmenbedingungen durch die Typ-, Variablen- und Funktionsdeklarationen aufgestellt wurden, wollen wir die Ausführung von $C0$-Programmen und somit die dynamischen Aspekte der $C0$-Semantik untersuchen. Dazu muss zunächst der dynamische Teil einer $C0$-Konfiguration c definiert werden. Sie besteht aus:

- $c.pr$ (**p**rogram **r**est) - verbleibende Anweisungsfolge, Programmrest
- $c.rd$ (**r**ecursion **d**epth) - Rekursionstiefe, die Anzahl der Funktionsaufrufe, die noch nicht mit $return$ beendet wurden; Die Anzahl local memories auf dem Stack entspricht $c.rd + 1$.
- $c.lms : [0 : c.rd] \longrightarrow M$ (**l**ocal **m**emory **s**tack) - eine Stack von Speichern (*function frames*), die jeweils bei einem Funktionsaufruf angelegt und bei return abgebaut werden; M ist die Menge aller Speicher; $c.lms(i)$ stellt den Speicher des function frame i dar.
- $c.gm$ (**g**lobal **m**emory) - der Speicher für globale Variablen
- $c.hm$ (**h**eap **m**emory) - der Speicher für namenlose Variablen, die durch new-Anweisungen erzeugt werden

Man beachte, dass die Speicher mit der Symboltabelle auch statische Komponenten besitzen. Dies rührt daher, dass die Symboltabelle des heap dynamisch ist und local memories, die ebenfalls dynamisch auf dem stack verwaltet werden, über ihre Symboltabelle mit der jeweils zugehörigen Funktion identifiziert werden. Alle anderen statischen Komponenten wie Typtabelle und Funktionstabelle können als implizite Parameter für die $C0$-Ausführung betrachtet werden. Außerdem ist zu beachten, dass die Variablen des heaps namenlos sind. Wir geben ihnen künstliche Namen $hm.st.na \subset \mathbb{N}_0$ und nummerieren sie nach der Reihenfolge ihrer Deklaration durch new.

Abbildung 4.13 stellt die Komponenten der $C0$-Konfiguration schematisch dar. Der oberste local memory im stack wird als *top frame* $top(c)$ bezeichnet:

$$top(c) = lms(c.rd)$$

Zu Beginn gilt $c^0.rd = 0$ und es liegt nur der local memory von `main` auf dem Stack. Wird nun ein Programm auf der $C0$-Maschine ausgeführt, so ändert sich deren Zustand und damit die Konfiguration c mit jeder Anweisungsausführung.

4.4.5 Ausdrucksauswertung

Bei der Ausführung von $C0$-Anweisungen müssen Ausdrücke (von $\langle A \rangle$ abgeleitet) und Identifier (von $\langle id \rangle$ abgeleitet) ausgewertet werden, um den Effekt des Befehls zu berechnen. Während die Identifier Subvariablen in gm, hm oder $lms(c.rd)$ identifizieren sind Ausdrücke sind sozusagen die Argumente der Anweisungen und können wiederum Identifier enthalten. Bei der Bestimmung des Werts eines Ausdrucks unterscheiden wir zwischen

- R-value und

- L-value.

R und *L* stehen dabei für **r**ight und **l**eft, was sich erschließt, wenn man die Ausführung einer Zuweisung $e = e'$ betrachtet. Der Wert des linken Ausdrucks e ist dabei eine Bindung der Form $(m, x)s$ mit $s = s_1 \cdots s_j$, also eine Subvariable (left value), der der Wert des rechten Ausdrucks e' zugewiesen werden soll. Dieser liegt im Wertebereich $Ra(typ((m, x)s))$ (right value). Bindungen von Ausdrücken e, also L-values, werden im Zustand c durch die Funktion $lv(e, c)$ gefunden. Entsprechend gibt die Funktion $va(e, c)$ den R-value von e in der Konfiguration c zurück.

Um den Wert von Subvariablen $(m, x)s_1 \cdots s_t$ zu bestimmen, führen wir die gesonderte Funktion vv (*variable value*) ein. Sie greift auf die va-Komponente des memory m zu und wertet die Indizes sukzessive aus durch einfaches Anwenden der Definition von $Ra(t)$ für komplexe Datentypen.

$$vv((m, x)s_1 \cdots s_t, c) = c.m.va(x)(s_1) \cdots (s_t)$$

Die Auswertung von $C0$-Ausdrücken findet rekursiv statt und muss zwischen der Interpretation als right value und left value unterscheiden. Für einen Ausdruck e gibt es die folgenden Fälle.

- $e \equiv e' \circ e''$ Boolescher oder arithmetischer Ausdruck ($\langle A \rangle \longrightarrow^*_{C0} e$) mit $\circ \in \{+, -_2, \cdot, /, \wedge, \vee\}$:

$$\begin{aligned} lv(e, c) &\quad \text{nicht definiert} \\ va(e, c) &= \begin{cases} va(e', c) \circ va(e'', c) & : \quad \circ \in \{\wedge, \vee\} \\ va(e', c) \pm_{32} va(e'', c) & : \quad \circ \in \{+, -_2\} \\ twoc_{32}([va(e', c)] \circ [va(e'', c)]) & : \quad \circ \in \{*, /\} \end{cases} \end{aligned}$$

  Man beachte, dass das Ergebnis der Division von Bitstrings im Gegensatz zu Addition, Subtraktion und Multiplikation davon abhängt, ob man die Strings als Binär- oder TWOC-Zahlen interpretiert. Wir legen uns hier auf die TWOC-Interpretation fest, damit wir auch mit negativen Zahlen rechnen können.

- $e \equiv e' \circ e''$ Vergleichsoperationen ($\langle BA \rangle \longrightarrow^*_{C0} e$) mit $\circ \in \{<, >, \leq, \geq, ==, \neq\}$:

$$\begin{aligned} lv(e, c) &\quad \text{nicht definiert} \\ va(e, c) &= 0^{31}([va(e', c)] \circ [va(e'', c)]) \end{aligned}$$

  Auch hier kommt es auf den Typ der Teilausdrücke an, welches Ergebnis die Ausdrucksauswertung liefert. Wir legen uns auf die TWOC-Interpretation fest, da die vorliegende DLX-ISA keine $unsigned$-Vergleiche unterstützt. Diese Unterstützung kann als Übungsaufgabe nachträglich integriert werden.

- $e \equiv \circ e'$ Boolescher oder arithmetischer Ausdruck mit unärem Operator $\circ \in \{-_1, \sim\}$:

$$\begin{aligned} lv(e, c) &\quad \text{nicht definiert} \\ va(e, c) &= \begin{cases} \sim va(e', c) & : \quad \circ = \sim \\ 0_{32} -_{32} va(e', c) & : \quad \circ = -_1 \end{cases} \end{aligned}$$

- $e \equiv '\chi'$ Character ($\langle BuZi \rangle \longrightarrow_{C0} \chi$)
  Char-Konstanten (Buchstaben und Zahlen) werden durch den entsprechenden 8-bit ASCII-Code des Zeichens mit 24 führenden Nullen repräsentiert.

$$\begin{aligned} lv(e, c) &\quad \text{nicht definiert} \\ va(e, c) &= 0^{24} ascii(\chi) \end{aligned}$$

- $e \equiv id$ Identifier ($\langle id \rangle \longrightarrow^*_{C0} id$)
  Identifier sind zumeist an Subvariablen xs im Speicher gebunden. Um die Bindung des Namen x der Variable mit dem Speicher zu verknüpfen, führen wir die Bindungsfunktion

  $$bind : VN \times K_p \longrightarrow \{(m, x) \mid m \in M, x \in VN\}$$

  ein. Hier zeigen sich auch die $C0$-Sichtbarkeitsregeln für Variablen.

  $$bind(x, c) = \begin{cases} (top(c), x) & : \quad x \in top(c).st.na \\ (c.gm, x) & : \quad x \notin top(c).st.na \wedge x \in c.gm.st.na \end{cases}$$

  Falls eine lokale Variable namens x im top level stack frame existiert, so bezeichnet der Identifier x diese Variable, ansonsten identifiziert x eine globale Variable.

  Für die verschiedenen Identifier muss man die folgende Fallunterscheidung machen:

  - $id \in \{0, \ldots, 9\}^L$ Konstante (L ist die Länge der Dezimaldarstellung)

    $$\begin{aligned} lv(id, c) & \quad \text{nicht definiert} \\ va(id, c) & = \text{bin}_{32}(\langle id[L-1:0] \rangle_Z) \qquad (Z = N(9)) \end{aligned}$$

  - $id \equiv X$ Variablenname X

    $$\begin{aligned} lv(id, c) & = bind(id, c) \\ va(id, c) & = vv(lv(id, c), c) = vv(bind(id, c), c) \end{aligned}$$

  - $id \equiv e'[e'']$ Arrayzugriff

    $$\begin{aligned} lv(id, c) & = lv(e', c)[\langle va(e'', c) \rangle] \\ va(id, c) & = vv(lv(id, c), c) = vv(lv(e', c)[\langle va(e'', c) \rangle], c) \end{aligned}$$

  - $id \equiv e'.n$ struct-Komponenten-Zugriff

    $$\begin{aligned} lv(id, c) & = lv(e', c).n \\ va(id, c) & = vv(lv(id, c), c) = vv(lv(e', c).n, c) \end{aligned}$$

  - $id \equiv e'*$ Pointer e' dereferenzieren

    $$\begin{aligned} lv(id, c) & = va(e', c) \\ va(id, c) & = vv(lv(id, c), c) = vv(va(e', c), c) \end{aligned}$$

    Der left value eines Pointers ist gerade die Subvariable, auf die er zeigt, also der Wert des Pointers. Der right value entspricht dem Wert der referenzierten Subvariable.

  - $id \equiv e'\&$ Adresse von Identifier e' abfragen (Address-of-Operator)

    $$\begin{aligned} lv(id, c) & \quad \text{nicht definiert} \\ va(id, c) & = lv(e', c) \end{aligned}$$

    Der Address-of-Operator liefert einen Wert zurück, der nur einem Pointer zugewiesen werden kann. Folglich entspricht dieser Wert der Subvariable, die mit e' identifiziert wird.

Die obigen Definitionen decken alle Ausdrücke der Grammatik ab. Mit den Funktionen $lv(e, c)$ und $va(e, c)$ kann man also rekursiv alle Ausdrücke auswerten, die in $C0$ auftreten.

An dieser Stelle schränken wir den Wertebereich von Pointertypen $t = t'*$ ein. Der Address-of-Operator nicht auf lokale Variablen angewendet werden. Dies hat den Zweck, das vermieden werden soll, dass Pointer auf Subvariablen im lms zeigen, die es nicht mehr gibt, weil in der Zwischenzeit der Stack abgebaut worden ist. Pointer (bis auf rds) dürfen nur noch auf Objekte im global und heap memory zeigen. Diese Regelung stellt jedoch keine große Einschränkung dar, da im schlimmsten Fall alle Berechnungen auf globalen oder heap-Variablen durchgeführt werden können.

4.4.6 Anweisungsausführung

Nun können wir die Abarbeitung eines $C0$-Programms definieren. Dabei betrachten wir die Folge der Anweisungen abhängig vom Zustand der $C0$-Maschine und geben für jeden Typ von Anweisung an, wie welche Auswirkungen dieser auf die aktuelle Konfiguration hat. Zu Beginn müssen wir jedoch die Startkonfiguration c^0 definieren.

- $c^0.rd = 0$ - Die Rekursionstiefe ist zu Anfang 0 (entspricht der Anzahl von Funktionsaufrufen, die noch nicht zurückgekehrt sind).
- $c^0.pr = ft(main).body$ - Es wird der Rumpf von $main$ ausgeführt. Die letzte return-Anweisung definiert einen Rückgabewert an das Betriebssystem, das das kompilierte $C0$-Programm ausführt.
- $top(c^0).st = c^0.lms(c^0.rd).st = c^0.lms(0).st = ft(main).st$ - Zu Beginn liegt einzig der local memory von $main$ auf dem stack.
- $c^0.gm$ - Der Startzustand ergibt sich aus der Variablendeklarationsfolge des $C0$-Programms. Die Speicherinhalte sollen zu Anfang mit Standardwerten für die jeweiligen Typen gefüllt sein. Wir können eine Funktion

  $$default : tt.na \longrightarrow \bigcup_{t \epsilon tt.na} Ra(t)$$

  für diesen Zweck definieren. Namen und zugehörige Typen der globalen Deklarationen werden entsprechend in die Symboltabelle eingetragen.
- $c^0.hm.st.na = \varnothing$ - Der heap memory ist anfangs leer. Für den heap gilt die Invariante

  $$c^i.hm.st.na = [0 : H(c^i) - 1],$$

  wobei $H(c) = \#c.hm.st.na$ die Anzahl der in Konfiguration c auf dem heap allokierten Variablen bezeichnet. Folglich gilt $H(c^0) = 0$.

Wird nun eine Anweisung in Konfiguration c ausgeführt, so erhalten wir die resultierende Konfiguration c' durch die $C0$-Übergangsfunktion

$$\delta_C(c) = c',$$

die es nun zu definieren gilt. Dabei führen wir eine Fallunterscheidung über den Programmrest $c.pr = an; r'$, beziehungsweise die auszuführende Anweisung an, durch.

- $an: \quad id = e$ (Zuweisung)
  Anschließend soll gelten:

  $$\begin{aligned} va(id, c') &= va(e, c) \\ c'.pr &= r' \end{aligned}$$

  Sonst soll sich nichts ändern. Um c' zu konstruieren, benötigen wir den left value von id.

  $$\begin{aligned} lv(id, c) &= (m, x)s_1 \cdots s_t \\ va(id, c') &= vv(lv(id, c'), c') \\ &= vv(lv(id, c), c') \\ &= vv((m, x)s_1 \cdots s_t, c') \\ &= c'.m.va(x)(s_1) \cdots (s_t) \end{aligned}$$

  Dies ist der Wert der Subvariable, an die id gebunden ist, nach der Zuweisung. Dieser soll dem zugewiesenen Wert entsprechen.

  $$c'.m.va(x)(s_1) \cdots (s_t) = va(e, c)$$

Dabei ist es wichtig, zu bemerken, dass

$$lv(id, c') = lv(id, c)$$

gilt, also dass sich die Bindung des Identifiers nicht während der Ausführung der Zuweisung ändert. Dies ist bei Funktionsaufrufen und return zum Beispiel nicht garantiert. Zudem müssen wir hier die schon angesprochen Kontextbedingung für die Zuweisungen an Pointer beachten. Bezeichnet id eine pointer-Subvariable, so dürfen dieser keine lokale Subvariablen $(lms(j), k)s$ zugewiesen werden. Ansonsten treten undefinierte Bindungen auf, sobald der local memory stack frame k abgebaut wird.

- $an:\quad if\ e\ then\ \{a\}\ else\ \{b\}$ (Fallunterscheidung)

$$c'.pr = \begin{cases} a; r' & : \quad va(e,c) \neq 0^{32} \\ b; r' & : \quad \text{sonst} \end{cases}$$

- $an:\quad while\ e\ do\ \{a\}$ (Schleife)

$$c'.pr = \begin{cases} a; while\ e\ do\ \{a\}; r' & : \quad va(e,c) \neq 0^{32} \\ r' & : \quad \text{sonst} \end{cases}$$

Die while-Schleife fügt sich selbst also erneut unverändert zum Programmrest hinzu, so lange die Abbruchbedingung noch nicht erfüllt ist. Diese induktive Definition über den *ersten* Schleifendurchlauf führt später zu einer unintuitiven Beweisführung der Korrektheit.

- $an:\quad p = new\ t^*, \quad t \in TN$ (new-Anweisung) Sei $R = H(c) - 1$

$$\begin{aligned} c'.hm.st.na &= c.hm.st.na \cup \{R+1\} \\ c'.hm.st.typ(x) &= \begin{cases} t & : \quad x = R+1 \\ c.hm.st.typ(x) & : \quad \text{sonst} \end{cases} \end{aligned}$$

Auf dem heap wird eine neue Variable vom Typ t angelegt. Deren Bindung wird p zugewiesen (siehe Abbildung 4.14). Sei $lv(p,c) = (m,x)s_1 \cdots s_t$ mit $tt.tc(typ(lv(p,c))) = t*$ Dann ergibt sich für den dynamischen Teil der $C0$-Kofiguration:

$$\begin{aligned} c'.m.va(x)(s_1)\cdots(s_t) &= (c'.hm, R+1) \\ c'.pr &= r' \end{aligned}$$

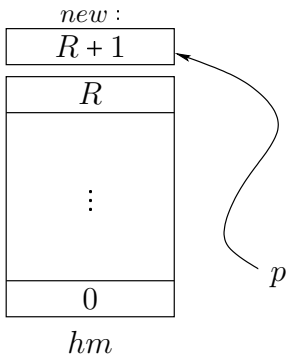


Abbildung 4.14: Ausfuhrung einer new-Anweisung

- $an:\quad id = f(e_1, \ldots, e_{ft(f).np})$ (Funktionsaufruf)
  Zuerst erhöhen wir die Rekursionstiefe und legen ein neues local memory für f an.

  $$\begin{aligned} c'.rd &= c.rd + 1 \\ top(c').st &= ft(f).st \end{aligned}$$

  Im local memory von f stehen die lokalen Variablen und die Parameter der Funktion. Die Parameter müssen noch ausgewertet und übergeben werden.

  $$\begin{aligned} &\forall i \in [1 : ft(f).np] : \\ top(c').va(ft(f).par(i)) &= va(e_i, c) \end{aligned}$$

  Die Bindung der Subvariable, der der Rückgabewert von f übergeben werden soll, wird in der return destination Variable gespeichert.

  $$top(c').va(rds) = lv(id, c)$$

  Als nächstes wird dann der Rumpf von f ausgeführt.

  $$c'.pr = ft(f).body; r'$$

- $an:\quad return\ e$ (Rückkehr von Funktionsaufruf)
  Die Semantik der $return$-Anweisung soll als einfache Übung selbst definiert werden.

Damit haben wir die Semantik von $C0$ vollständig beschrieben. Im Folgenden soll versucht werden, auf dieser Grundlage einen Compiler zu konstruieren und diesen zu verifizieren. Zunächst erfolgt aber ein Einschub zur formalen Verifikation von $C0$-Programmen mit Hilfe der formalen Semantik.

4.5 Korrektheit von C0-Programmen

Da wir zuvor die Syntax und Semantik von $C0$-Programmen definiert haben, können wir diese nun dazu benutzen formale Beweise über die Korrektheit der Ausführung zu führen. Dies soll an vier exemplarischen Beispielen verdeutlicht werden.

4.5.1 Beispiel 1: Zuweisung und Fallunterscheidung

Das folgende Programm ist gegeben:

```
int x;
int main()
{
  x=3;
  if (x==0) then {x=1} else {x=2};
  return -1
}
```

Es soll nun ausgeführt werden und wir betrachten die Startkonfiguration c^0:

$$\begin{aligned} c^0.pr &= x = 3; if\ (x == 0)\ then\ \{x = 1\}\ else\ \{x = 2\}; return\ -1 \\ c^0.rd &= 0 \\ c^0.lms(0).st &= ft(main).st = (\{rds\}, \{rds \mapsto t\}, main), \quad tt.tc(t) = int* \\ c^0.lms(0).va &: \{rds \mapsto \text{undefiniert}\} \\ c^0.hm.st &= (\varnothing, \text{undefiniert}, hm) \\ c^0.hm.va &: \text{undefiniert} \\ c^0.gm.st &= (\{x\}, \{x \mapsto int\}, gm) \\ c^0.gm.va &: \{x \mapsto default(int)\} \end{aligned}$$

Der local memory stack enthält den lokalen Speicher der $main$-funktion. Dieser ist allerdings leer und $ft(main).st.na$ enthält nur den return destination Pointer rds. Wegen der Abwesenheit von Typdefinitionen besteht die Typtabelle nur aus den elementaren Typen sowie einem Typen t für rds von $main$ mit $tt.tc(t) = rtyp(main)*$. Der Wert von rds ist undefiniert, da die Rekursionstiefe noch 0 beträgt. In diesem Fall stellt der Rückgabewert eine Nachricht an das Betriebssystem nach Aufruf und Ausführung der Funktion dar. Die Funktionstabelle enthält nur $main$.

$$\begin{aligned} rtyp(main) &= int \\ ft(main).body &= x = 3; if\ (x == 0)\ then\ \{x = 1\}\ else\{x = 2\}; return\ -1 \\ ft(main).np &= 0 \\ ft(main).par &: \text{undefiniert} \\ ft(main).st &= (\{rds\}, \{rds \mapsto t\}, main), \quad tt.tc(t) = int* \end{aligned}$$

Offensichtlich sollte x nach Ausführung des Programms bis zum letzten $return$ den Wert 2 besitzen.

$$\exists t : va(x, c^t) \stackrel{!}{=} 2_{32}$$

Dabei ist $c^t = \delta_C(c^{t-1}) = \underbrace{\delta_C(\delta_C(\cdots\delta_C}_{t \text{ mal}}(c^0)\cdots))$.

Tabelle 4.4 verfolgt den Wert von x während der Ausführung des Programms. Dazu wird die $C0$-Semantik der einzelnen Anweisungen verwendet.

Schrittzahl	$c.pr$	Wert
0	$x = 3; if \ldots$	$va(x, c) = default(int)_{32}$
1	$if\ (x == 0)$	$va(x, c) = 3_{32}$
	$then\ \{x = 1\}$	
	$else\{x = 2\}; \ldots$	$va(x == 0, c) = 0^{32}$
2	x = 2;return -1	
3	$return\ -1$	$va(x, c) = 2_{32}$

Tabelle 4.4: Beispiel 1

Für $t = 3$ gilt demnach:

$$va(x, c^t) = 2_{32} \qquad \square$$

Natürlich wurde hier der Einfachheit halber viel Formalismus weggelassen. Das folgenden Beispiel soll genauer nachvollzogen werden.

4.5.2 Beispiel 2: Rekursion

Beim nächsten Beispiel stehen (rekursive) Funktionsaufrufe im Mittelpunkt. Wir betrachten die Funktion $fib(n)$, die die n-te Fibonacci-Zahl liefert.

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n) &= fib(n-2) + fib(n-1) \end{aligned}$$

Dazu sei folgender Code gegeben:

```
int x;
int fib(int n)
{
  int res;
  int f1;
```

```
int main()
{
  x=fib(2);
  return -1
}
```

```
  int f2;
  if (n<2) then {res=n} else
  {
    f1=fib(n-2);
    f2=fib(n-1);
    res=f1+f2;
  };
  return res
};
```

Es zu zeigen, dass:

$$\exists t : va(x, c^t) = 1_{32} \wedge c^t.pr = return\ -1$$

Startkonfiguration

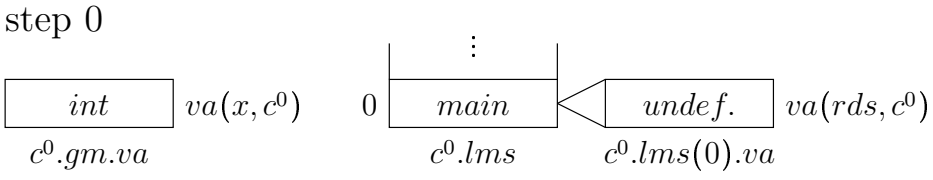
$$\begin{aligned}
c^0.pr &= x = fib(2); return\ -1 \\
c^0.rd &= 0 \\
c^0.lms(0).st &= ft(main).st = (\{rds\}, \{rds \mapsto t\}, main), \quad tt.tc(t) = int* \\
c^0.lms(0).va &: \{rds \mapsto \text{undefiniert}\} \\
c^0.hm.st &= (\varnothing, \text{undefiniert}, hm) \\
c^0.hm.va &: \text{undefiniert} \\
c^0.gm.st &= (\{x\}, \{x \mapsto int\}, gm) \\
c^0.gm.va &: \{x \mapsto default(int)\}
\end{aligned}$$

In der Typtabelle stehem nur die elementaren Typen und t mit $tt.tc(t) = int*$ für rds von $main$ und fib. Die Funktionstabelle enthält die Funktionen $main$ und fib.

$$\begin{aligned}
rtyp(main) &= int \\
ft(main).body &= x = fib(2); return\ -1 \\
ft(main).np &= 0 \\
ft(main).par &: \text{undefiniert} \\
ft(main).st &= (\{rds\}, \{rds \mapsto t\}, main), \quad tt.tc(t) = int* \\
rtyp(fib) &= int \\
ft(fib).body &= if\ (n < 2)\ then\ \{res = n\}\ else\ \{\ldots\}; return\ res \\
ft(fib).np &= 1 \\
ft(fib).par &: \{1 \mapsto n\} \\
ft(fib).st &= (\{rds, n, res, f1, f2\}, \{rds \mapsto t, n \mapsto int, res \mapsto int, f1 \mapsto int, f2 \mapsto int\}, fib)
\end{aligned}$$

Der local memory stack enthält nur den lokalen Speicher für $main$, der allerdings bis auf rds leer ist.

$$\begin{aligned}
top(c^0).st &= c^0.lms(c^0.rd).st \\
&= c^0.lms(0).st \\
&= ft(main).st \\
c^0.lms(0).st.na &= \{rds\}
\end{aligned}$$



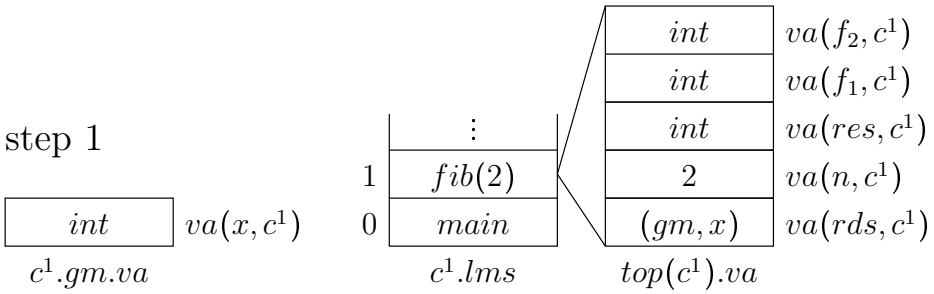
Schritt 1
$x = fib(2)$ wird ausgeführt. Mit der $C0$-Semantik für Funktionsaufrufe ergibt sich:

$$\begin{aligned} c^1.pr &= if\ (n < 2)\ then\ \{res = n\}\ else\ \{\ldots\}; return\ res; return\ -1 \\ c^1.rd &= 1 \\ va(rds, c^1) &= lv(x, c^0) \\ &= (gm, x) \end{aligned}$$

Ein neuer lokaler Speicher wird für fib auf dem stack angelegt.

$$\begin{aligned} top(c^1).st &= c^1.lms(c^1.rd).st \\ &= c^1.lms(1).st \\ &= ft(fib).st \end{aligned}$$

In dem Speicher wird Platz für rds, die Parameter (n) und lokalen Variablen (res, f_1, f_2) von fib reserviert. Der Wert für n wird durch die Parameterübergabe gesetzt.



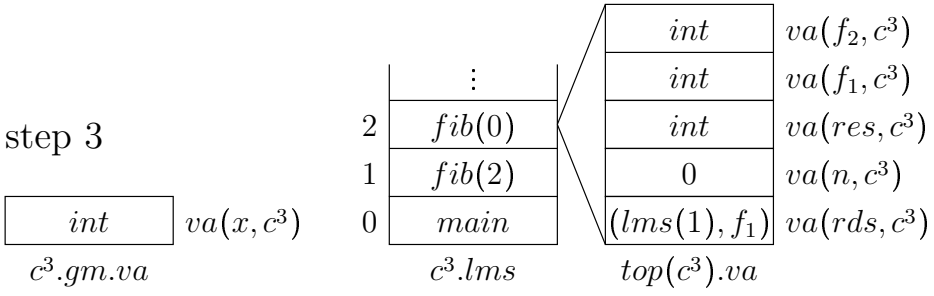
Schritt 2
Die Fallunterscheidung wird ausgeführt. Wegen $[va(n, c^1)] = 2 \not< 2$ ergibt sich für den Programmrest:

$$c^2.pr = f_1 = fib(n-2); f_2 = fib(n-1); res = f_1 + f_2; return\ res; return\ -1$$

Schritt 3
Ein weiterer Funktionsaufruf von fib, diesmal mit $va(n, c^2) = 0_{32}$ als Parameter:

$$\begin{aligned} c^3.pr &= if\ (n < 2)\ then\ \{res = n\}\ else\ \{\ldots\}; return\ res; \\ &\quad f_2 = fib(n-1); res = f_1 + f_2; return\ res; return\ -1 \\ c^3.rd &= 2 \\ va(rds, c^3) &= bind(f_1, c^2) \\ &= (lms(1), f_1) \end{aligned}$$

Der neue top-frame ist $lms(2)$ und es gilt $c^3.lms(2).st = ft(fib).st$.



Schritt 4
Wieder wird eine if-$then$-$else$-Anweisung ausgewertet. Diesmal wird jedoch durch $[va(n, c^3)] = 0 < 2$ der $then$-Teil zum Programmrest hinzugefügt.

$$c^4.pr = res = n; return\ res; f_2 = fib(n-1); res = f_1 + f_2; return\ res; return\ -1$$

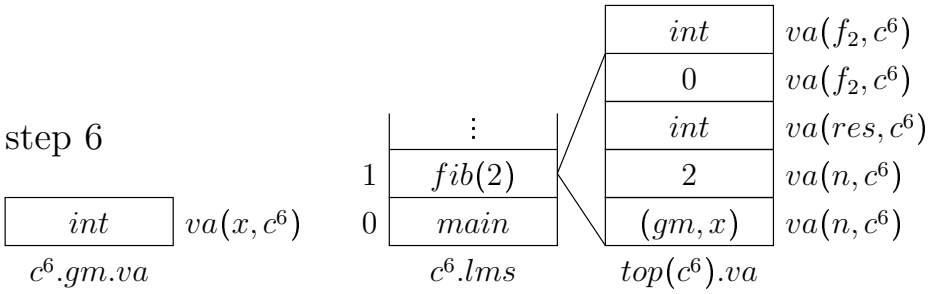
Schritt 5
Aufgrund der Zuweisung $res = n$ gilt:

$$\begin{aligned} va(res, c^5) &= vv((lms(2), res), c^5) \\ &= va(n, c^4) \\ &= 0 \\ c^5.pr &= return\ res; f_2 = fib(n-1); res = f_1 + f_2; return\ res; return\ -1 \end{aligned}$$

Schritt 6
Das Ergebnis von $fib(0)$ wird durch die $return$-Anweisung an die aufrufende Funktion $fib(2)$ zurückgegeben.

$$\begin{aligned} va(rds, c^5) &= vv((lms(2), rds), c^5) \\ &= vv((lms(2), rds), c^3) \\ &= (lms(1), f_1) \\ va((lms(1), f_1), c^6) &= va(res, c^5) \\ &= 0 \\ c^6.rd &= 1 \\ va(rds, c^6) &= vv((lms(1)rds), c^6) \\ &= (gm, x) \\ c^6.pr &= f_2 = fib(n-1); res = f_1 + f_2; return\ res; return\ -1 \end{aligned}$$

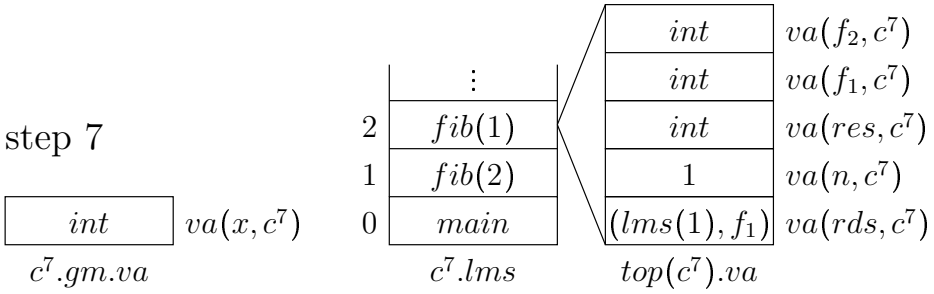
$fib(2)$ liegt nun wieder oben auf dem stack.



Schritt 7
Noch ein Funktionsaufruf von fib, dieses Mal mit $va(n, c^6) = 1_{32}$ als Parameter:

$$\begin{aligned} c^7.pr &= if\ (n < 2)\ then\ \{res = n\}\ else\ \{\ldots\}; res = f_1 + f_2; return\ res; return\ -1 \\ c^7.rd &= 2 \\ va(rds, c^7) &= bind(f_2, c^6) \\ &= (lms(1), f_2) \end{aligned}$$

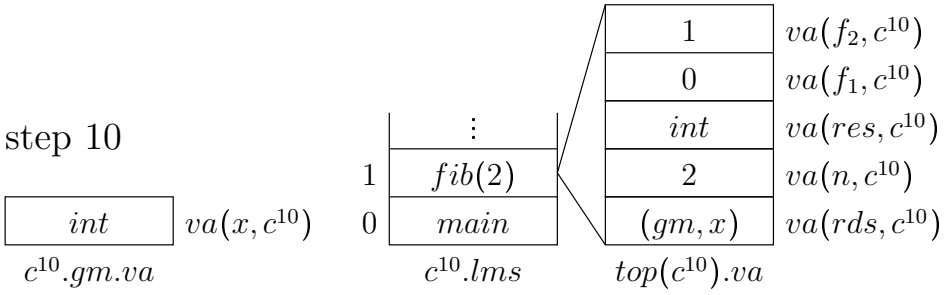
Der oberste local memory ist nun der zu $fib(1)$ gehörende $lms(2)$.



Schritte 8-10
Diese Schritte sind analog zu den Schritten 4-6. Es resultieren die folgenden Veränderungen:

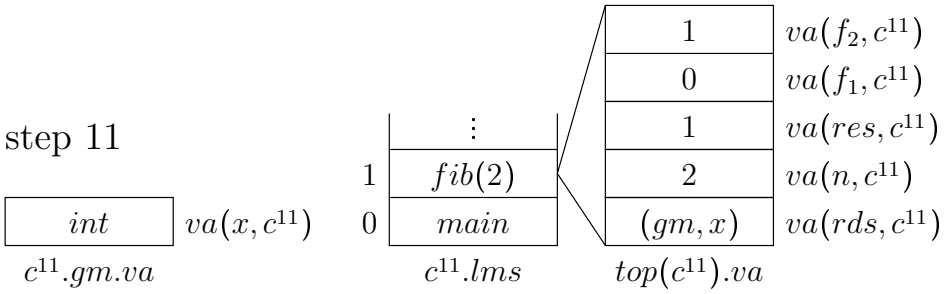
$$\begin{aligned} c^{10}.pr &= res = f_1 + f_2; return\ res; return\ -1 \\ c^{10}.rd &= 1 \\ va(rds, c^{10}) &= (gm, 0) \\ vv((lms(1), f_1), c^{10}) &= va(res, c^9) \\ &= 1 \end{aligned}$$

Der top-frame gehört wieder zu $fib(2)$.



Schritt 11
res wird der Wert des Ausdrucks $f_1 + f_2$ zugewiesen:

$$\begin{aligned} va(res, c^{11}) &= va(f_1 + f_2, c^{10}) \\ &= va(f_1, c^{10}) +_{32} va(f_2, c^{10}) \\ &= 0_{32} +_{32} 1_{32} \\ &= 1_{32} \\ c^{11}.pr &= return\ res; return\ -1 \end{aligned}$$

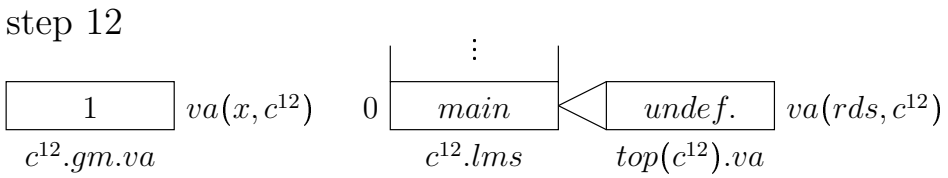


Schritt 12
Die Funktion $fib(2)$ terminiert und das Ergebnis wird an $main$ zurückgegeben.

$$\begin{aligned} va(rds, c^{11}) &= (gm, x) \\ vv((gm, x), c^{12}) &= va(res, c^{11}) \\ &= vv((lms(1), res), c^{11}) \\ &= 1_{32} \\ c^{12}.rd &= 0 \\ c^{12}.pr &= return\ -1 \end{aligned}$$

Für $t = 12$ gilt also:

$$\begin{aligned} va(x, c^t) &= 1_{32} \\ c^t.pr &= return\ -1 \end{aligned} \quad \square$$



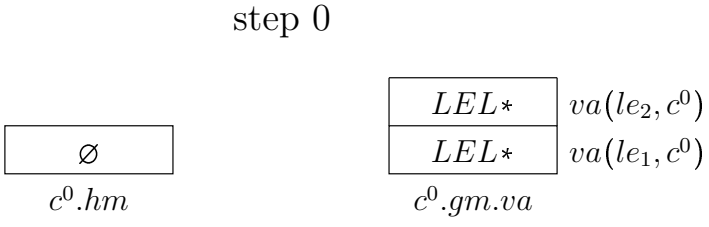
Die vorliegende Beweisführung wirkt in gewisser Weise „mechanisch“, da man lediglich nach einem festen Schema die $C0$-Semantik Schritt für Schritt auf das gegebene Programm anwendet. Die Vermutung liegt nahe, dass dies auch automatisch von einem computergestützten Beweissystem ausgeführt werden kann. Das Verisoft-Projekt beschäftigte sich mit dieser Thematik.

4.5.3 Beispiel 3: Dynamische Speicherzuweisung

Im Folgenden soll anhand einer einfachen verketteten Liste, die Allozierung von Speicher auf dem heap durch den new-Befehl untersucht werden. Es ist dieses $C0$-Programm gegeben:

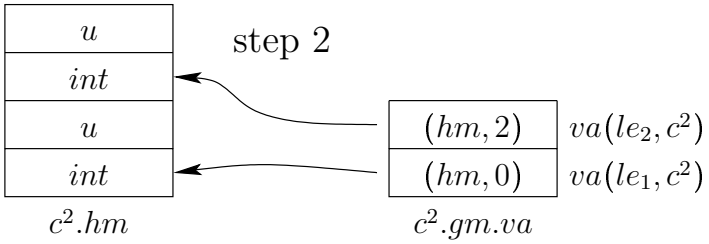
```
typedef LEL* u;
typedef struct{int content, u next} LEL;
u le1;
u le2;
int main()
{
  le1=new LEL*;
  le2=new LEL*;
  le1*.content=1;
  le1*.next=le2;
  le2*.content=2;
  le2*.next=null;
  return -1
}
```

Die einzelnen Programmschritte sollen hier nicht in aller Ausführlichkeit betrachtet werden, vielmehr sollen die Auswirkungen auf gm beziehungsweise hm im Mittelpunkt stehen. Im Startzustand ist der heap leer. Im global memory liegen le_1 und le_2, deren Werte noch undefiniert sind.

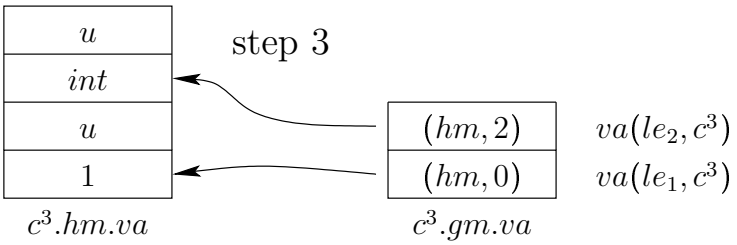


Nach den ersten beiden Schritten wurden zwei Variablen vom Typ LEL auf dem heap erzeugt und le_1 bzw. le_2 zugewiesen. Der zugehörige Speicherplatz auf dem heap ist nun reserviert und der heap ist gewachsen.

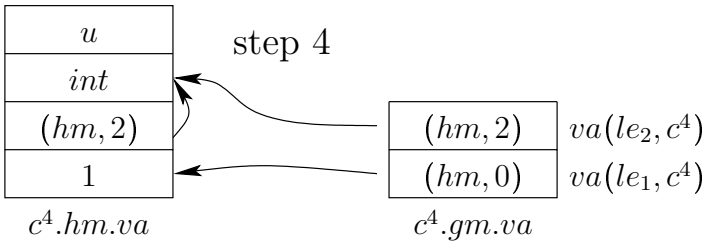
$$\begin{aligned} H(c^2) &= H(c^1)+1 \\ &= H(c^0)+1+1 \\ &= 2 \\ c^2.hm.st &= (\{0,1\},\{0 \mapsto LEL, 1 \mapsto LEL\}, hm) \end{aligned}$$



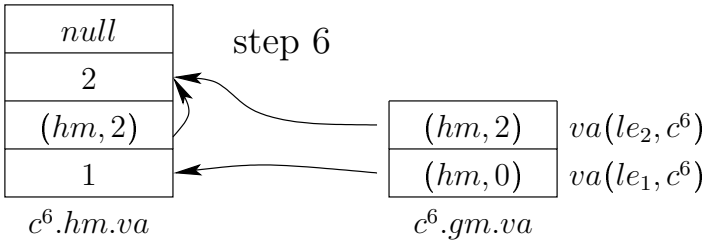
Die einzelnen Komponenten erhalten nun Werte, z.B. $le_1.content = 1$.



Nach Schritt 4 zeigt der next-Pointer von le_1 auf das gleiche Listenelement wie le_2.



Um einen Pointer, der auf keine Variable zeigen soll, zu initialisieren, kann man ihm den Nullpointer $null$ zuweisen.



4.5.4 Beispiel 4: Schleife

Bei der Beschreibung der Semantik von $while\ (\ldots)\ do\ \{\ldots\}$ fiel bereits die induktive Definition über den ersten Schleifendurchlauf auf. Wie sich diese auf Korrektheitsbeweise auswirkt, soll nun mit Hilfe des folgenden Beispiels genauer betrachtet werden.

```
int n;
int result;
int main()
{
  result=0;
  while (n>0) do
  {
    result=result+n;
    n=n-1;
  }
  return -1
}
```

Nach der while-Schleife soll $result$ den Wert der Summe über die natürlichen Zahlen bis $N = va(c^0, n)$ besitzen.

$$\begin{aligned} \exists t : va(result, c^t) &= \sum_{k=1}^{N} k \\ c^t.pr &= return\ -1 \end{aligned}$$

Erste Idee: Man führt eine Induktion über die Schleifendurchgänge durch.
Induktionsbehauptung: nach dem j-ten Durchlauf der Schleife gilt:

$$\begin{aligned} va(n, c^{1+3j}) &= N - j \\ va(result, c^{1+3j}) &= \sum_{k=N-j+1}^{N} k \end{aligned}$$

Der Summand 1 im Konfigurationsindex $1 + 3j$ rührt dabei von der Initialisierung von $result$ mit 0 her. Der Faktor 3 ist die Anzahl der Instruktionen pro Schleifendurchlauf im vorliegenden Beispiel.
Induktionsschritt: $j \to j + 1$
Erzeugt man die Konfiguration $c^{1+3(j+1)}$ durch Ausführen der 3 Instruktionen eines Durchlaufs auf die Konfiguration c^{1+3j} (Induktionsvorraussetzung), so erhält man:

$$\begin{aligned} c^{1+3j+1}.pr &= result = result + n; n = n - 1; while\ (\ldots)\ do\ \{\ldots\}; return\ -1 \\ va(n, c^{1+3j+3}) &= N - j - 1 \qquad \checkmark \\ va(result, c^{1+3j+3}) &= \sum_{k=N-j+1}^{N} k + (N - j) \\ &= \sum_{k=N-j}^{N} k \qquad \checkmark \\ c^{1+3j+3}.pr &= while\ (\ldots)\ do\ \{\ldots\} \neq return\ -1 \qquad \text{✗} \end{aligned}$$

Man kann offenbar mit dieser Herangehensweise keine Aussage über die noch verbleibenden Durchläufe treffen. Durch die Definition, dass sich die Schleife nach jedem Durchlauf erneut zum Programmrest hinzufügt, ist eine Induktion auf herkömmliche Weise über einzelne Durchläufe nicht zielführend. Man sollte dagegen über die Anzahl der verbleibenden Schleifendurchläufe argumentieren.
Eleganterer Ansatz: Man führt die Induktion der Definition entsprechend über den ersten Schleifendurchlauf aus und schließt aus der Induktionsvoraussetzung, dass die restlichen Durchläufe korrekt terminieren werden.

$$c'.pr = \underbrace{a}_{1.\ \text{Durchlauf}}; \underbrace{while\ (e)\ do\ \{a\}}_{N-1\ \text{Durchlaufe}}; return\ -1$$

Induktionsbehauptung: Die Induktion wird nun über die Anzahl der verbleibenden Schleifendurchgänge M durchgeführt. Sei

$$\begin{aligned} va(result, c^{\alpha}) &= K \\ va(n, c^{\alpha}) &= M \\ c^{\alpha}.pr &= while\ (\ldots)\ do\ \{\ldots\}; return\ -1, \end{aligned}$$

dann folgt für die Konfiguration nach M Durchläufen $c^{\alpha+3M}$:

$$\begin{aligned} va(result, c^{\alpha+3M}) &= K + \underbrace{M + M - 1 + \ldots + 1}_{\sum_{i=1}^{M} i} \\ va(n, c^{\alpha+3M}) &= 0 \\ c^{\alpha+3M}.pr &= while\ (\ldots)\ do\ \{\ldots\}; return\ -1 \end{aligned}$$

Der Beweis mit Induktionsanfang für $M = 0$ und Induktionsschritt $M \to M + 1$ ist nun leicht zu bewältigen. Weil n jetzt den Wert 0 hat, wird die Schleife mit dem nächsten Ausführungsschritt auf $c^{\alpha+3M}.pr = while\ (n > 0)\ do\ \{\ldots\}; return\ -1$ terminieren. Dann gilt

$$c^{\alpha+3M+1}.pr = return\ -1$$

und die Korrektheitsbehauptung gilt für $t = \alpha + 3M + 1$.

4.6 C0-Compiler

Um $C0$-Programme auf der DLX ausführen zu können, müssen diese erst in DLX-Assembler übersetzt werden. Dies wird von einem Compiler übernommen, der hier definiert werden soll. Man betrachtet dabei zwei Rechnungen:

- $c^0, c^1, c^2, \ldots, c^i$ mit $\delta_C(c^i) = c^{i+1}$ - $C0$-Rechnung
- $d^0, d^1, d^2, \ldots, d^{s(i)}$ mit $\delta_D(d^i) = d^{i+1}$ - DLX-Rechnung

i $C0$-Anweisungen werden in $s(i)$ Schritten der DLX simuliert. Der erzeugte Assemblercode zu einem $C0$-Programm p wird mit $code(p)$ bezeichnet. Seine Erzeugung erfolgt größtenteils gemäß einer induktiven Definition über die Ableitungsbäume der Funktionsrümpfe.

4.6.1 Spezifikation und Korrektheit

Natürlich möchte man erreichen, dass ein $C0$-Programm genau das aus der $C0$-Semantik heraus erwartete Ergebnis liefert, wenn man es auf der DLX ausführt. Um die Korrektheit des erzeugten Codes zu überprüfen stellt man eine Konsistenzrelation zwischen $C0$- und DLX-Rechnung auf.

$$consis(c^i, aba^i, d^{s(i)})$$

besagt, dass c^i von $d^{s(i)}$ kodiert wird. Dabei bezeichnet aba^i die **a**llocated **b**ase **a**ddress von $C0$-Variablen in der DLX-Maschine. aba liefert also zu jedem Variablennamen die zugehörige Basisadresse im Speicher der DLX. Da sich diese Orte während der Rechnung verändern können, ist aba zustandsabhängig. Weil aba implizit jedoch auch Konstanten wie $sbase$ und $hbase$ festlegt, betrachten wir es als Parameter für die Kompilierung. Nun kann man den Simulationssatz für den Compiler aufstellen.

Theorem 4.2 *(Simulationssatz)* $\exists aba^i, \exists s(i)$:

$$\forall i : consis(c^i, aba^i, d^{s(i)})$$

Die Simulation erfolgt Schritt für Schritt und man muss die Umsetzung der Rechnung in der DLX untersuchen.

Speicheraufteilung

Der Speicher der DLX wird in verschiedene Bereiche unterteilt. Diese entsprechen den Komponenten der $C0$-Konfiguration. Es gibt

- $code$ - Hier werden die erzeugten Instruktionen $code(p)$ abgelegt.
- gm - Dieser Speicherabschnitt repräsentiert den global memory.
- F_i - Dies sind function frames für den i-ten rekursiven Funktionsaufruf mit $i \in [0 : c.rd - 1]$.
- hm - Dieser Speicherabschnitt repräsentiert den heap memory.

Abbildung 4.15 stellt die Speicheraufteilung schematisch dar und stellt sie der $C0$-Konfiguration gegenüber. Der $code$-Bereich erstreckt sich von einer Adresse α bis nach β.

$$code(p) = d.m_{\beta-\alpha+1}(\alpha)$$

Dieser Bereich darf nicht während der Ausführung überschrieben werden. Die aufeinanderfolgenden Bereiche für gm und die function frames werden zusammenfassend als *stack* bezeichnet. Er beginnt an der Adresse $sbase > \beta$ (*stack base*), die immer in $d.gpr(28_5)$ gespeichert wird. Der stack wächst „nach oben“ auf den heap zu. Es ist zu sichern, dass heap und stack niemals kollidieren. In einem solchen Fall müsste das aktuelle Programm wegen Speichermangel abgebrochen werden. Der heap beginnt an Adresse $hbase$ und wächst ebenfalls „nach oben“ auf den „oberen Rand“ des Speichers $hmax$ zu.

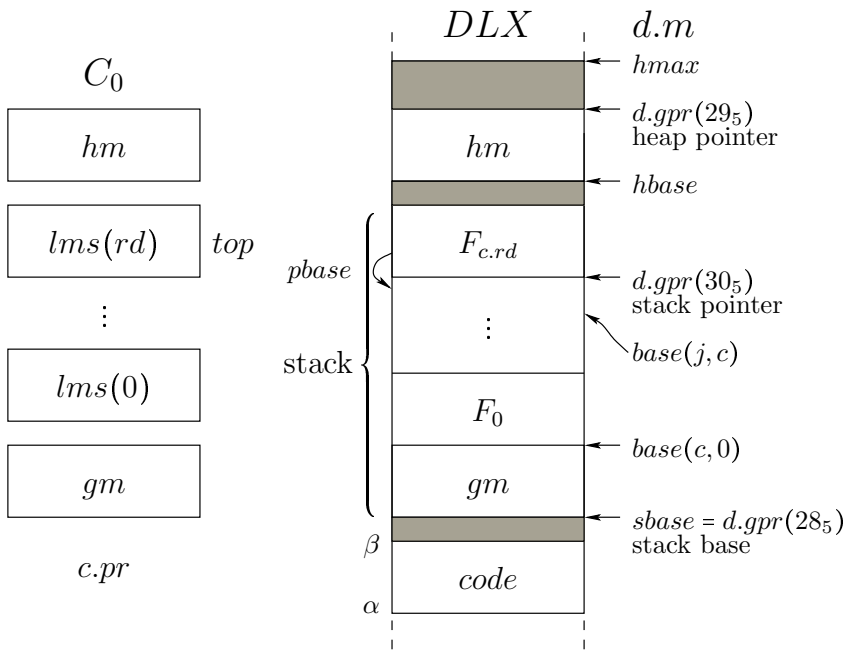


Abbildung 4.15: DLX simuliert $C0$-Konfiguration

Der *heap pointer* in $d.gpr(29_5)$ zeigt stets auf die erste (oberste) freie Adresse oberhalb des heap memory. Die Wachstumsrichtungen von heap und stack wurden so gewählt, um möglichst angenehme Definitionen in der Formalisierung des Compilers zu erhalten. Eine Optimierung wäre, den heap von "oben" auf den stack zuwachsen zu lassen. Dann würde mehr Speicher für Programme zur Verfügung stehen, die hohe Rekursionstiefen erreichen, aber nur wenig Speicher auf dem heap allokieren.

Aufbau der Function Frames

Die stack frames F_i enthalten nicht nur die Parameter und lokalen Variablen von Funktionsaufrufen, sondern auch weitere Daten, wie Rücksprung- oder Rückgabeadressen. $base(c, j)$ beschreibt die Basisadresse von function frame F_j und auf den obersten frame (top frame) verweist der *stack pointer* in $d.gpr(30_5)$. So ist immer bekannt, wo der frame der Funktion, die gerade ausgeführt wird, im Speicher liegt. Lokale Variablen, Parameter und andere Komponenten im top frame können relativ zum stack pointer addressiert werden. Um nach einer $return$-Anweisung den vorhergehenden Funktionsaufruf wiederzufinden, wird die frame-Basisadresse $base(c, j)$ für $j < c.rd$ in F_{j+1} gespeichert. Das genaue Format eines funktion frame F_j ist in Abbildung 4.16 dargestellt.
Angenommen, F_j entstand durch den Aufruf der Funktion f im $C0$-Programm, dann kodiert F_j den local memory $lms(j)$ einer entsprechenden Konfiguration c mit

$$ft(f).st = c.lms(j).st \qquad c.lms(j).st.f = f$$

Außerdem enthält ein function frame, wie aus der Grafik ersichtlich ist, noch drei weitere Wörter, die den Parametern und lokalen Variablen vorangestellt sind, nämlich:

- $F_j.rd$ (**r**eturn **d**estination) - Dies ist das Pendant zu $rds \in c.lms(j).st.na$. Es speichert die allocated base address der Variable, an die das Ergebnis von f zurückgegeben werden soll.
- $F_j.pbase$ (**p**redecessor **base**) - Hier wird $base(j-1, c)$ gespeichert.
- $F_j.ra$ (**r**eturn **a**ddress) - Hier wird gespeichert, zu welcher Stelle in $code$ nach der Ausführung der Funktion zurückgesprungen werden soll.

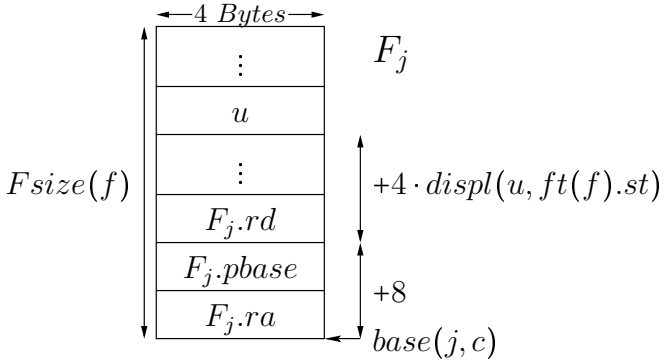


Abbildung 4.16: function frame F_j

Größe von Typen und Speichern

Damit der Compiler Code für den Aufbau der function frames erzeugen kann muss er deren benötigte Größe im Speicher kennen. Außerdem muss auch über die Größe des global und heapmemory argumentiert werden können. Diese Größen hängen von den Symboltabellen und der dem Platz, den Variablen der verschiedenen Typen im Speicher benötigen, ab. Wie man diese Werte berechnet, soll nun betrachtet werden.
Man unterscheidet einfache und komplexe Typen. Einfache Typen sind entweder elementare Typen (int, $bool$, $char$, $unsigned$) oder pointer-Typen, während komplexe Typen arrays und structs beinhalten. Wir definieren eine Funktion $size : TN \longrightarrow \mathbb{N}$, die jedem Eintrag der Typtabelle die Größe des entsprechenden Typs in Bytes zuordnet. Für $t \in \{int, bool, char, unsigned\}$ gilt:

$$size(t) = 4$$

Für diese Typen wird also ein Word im Speicher reserviert. Für die deklarierten Typen ergibt sich mit $tt.o(t) \geq 4$:

$$size(t) = \begin{cases} n \cdot size(t') & : \quad tt.tc(t) = t'[n] \\ \sum_{i=1}^{s} size(t_i) & : \quad tt.tc(t) = struct\{t_1\ n_1; \ldots; t_s\ n_s\} \\ 4 & : \quad tt.tc(t) = ptr(t') \end{cases}$$

Die Größe von Speichern hängt nur von der jeweiligen Symboltabelle ab. Der zu reservierende Speicher für eine Symboltabelle berechnet sich wie folgt.

$$size(st) = 8 + \sum_{x \in st.na} size(st.typ(x))$$

Der zusätzliche Summand 8 repräsentiert den benötigten Speicher für ra und $pbase$ im function frame. Um eine Fallunterscheidung in der Definition von $displ$ zwischen lms, gm und hm zu vermeiden, reservieren wir diese 8 Byte auch im globalen und heap memory, obwohl sie nicht genutzt werden. Die Größe des $Fsize(f)$ des function frame F_j zu Funktion f beträgt dann:

$$Fsize(f) = size(ft(f).st)$$

Für die Größe von Speichern m haben wir:

$$size(m) = size(m.st)$$

Adresszuweisung

Die Basisadresse des j-ten function frame F_j berechnet sich rekursiv wie folgt.

$$base(0, c) = \langle sbase \rangle + size(c.gm.st) \qquad base(j+1, c) = base(j, c) + Fsize(c.lms(j).st.f)$$

Nun interressieren uns, wo die Variablen und Parameter der Funktionen auf dem stack gespeichert werden. Dazu führen wir für einen Parameter und lokale Variablen x von f mit ein sogenanntes *displacement* ein. Dieses bestimmt den offset von x im function frame und hängt nur von der Symboltabelle $ft(f).st$ ab. Das displacement ist also unabhängig von j und dem Zustand des local memory stack, insbesondere davon, wo im lms der zum Funktionsaufruf zugehörige lokale Speicher liegt. Es gilt für eine beliebige Symboltabelle st:

$$displ(x, st) = 8 + \sum_{y \in st.na}^{y<x} size(st.typ(y))$$

Dabei verwenden wir eine Ordnung von Variablen und Parametern, die der Deklarationsreihenfolge im Ableitungsbaum von $st.f$ entspricht. Für alle Parameter p und lokale Variablen x gilt dabei $p < x$ und außerdem $rds < p$.

Desweiteren benötigen wir auch das displacement von Subvariablen. Wir unterscheiden zwischen den zwei möglichen komplexen Typen der Subvariable xs:

- $typ(xs) = t'[n]$ - xs ist ein array.

$$displ(xs[k], st) \quad = \quad displ(xs, st) + k \cdot size(t')$$

- $typ(xs) = struct\{t_1\ n_1; \ldots; t_s\ n_s\}$ - xs ist ein struct

$$displ(xs.n_k, st) \quad = \quad displ(xs, st) + \sum_{i=1}^{k-1} size(t_i)$$

Die displacements sind also unabhängig von j und zur Compilezeit bekannt. Deshalb kennt man die Basisadressen lokaler Variablen schon zur Compilezeit und kann jene beim Erzeugen des Codes direkt adressieren. Gleiches gilt sich für globale Variablen in gm, da die Symboltabelle des globalen Speichers zur Compilezeit feststeht.

Wir betrachten nun $(lms(j), x)s$, eine lokale Subvariable der Funktion $f = c.lms(j).st.f$. Wegen $c.lms(j).st = ft(f).st$ hängt der Aufbau des local memory und somit das displacement von x nur von der Funktion f ab und nicht etwa von j.

$$displ(xs, f) = displ(xs, ft(f).st)$$

Dann kann die allocated base address von $(lms(j), x)s$ folgendermaßen ermittelt werden.

Definition 4.12 *aba berechnet sich für lokale Variablen und Parameter $(lms(j), x)s$ zu:*

$$aba((lms(j), x)s, c) = base(j, c)_{32} +_{32} displ(xs, f)_{32}$$

Dabei verwenden wir die abkürzende Notation $x_{32} = bin_{32}(x)$ für beliebige Terme x.

Definition 4.13 *Ist $(gm, x)s$ eine globale Subvariable, so gilt:*

$$aba((gm, x)s, c) = sbase +_{32} displ(xs, gm.st)_{32}$$

Definition 4.14 *Ist $x \in [0 : H(c) - 1]$ eine heap-Variable und s ein gültiger Selektor für x, so gilt:*

$$aba((hm, x)s, c) = hbase +_{32} displ(xs, c.hm.st)_{32}$$

aba bestimmt also für jede Subvariable, an welcher Adresse sie vom Compiler im Speicher der DLX allokiert wird, abhängig davon ob es sich um eine lokale, globale oder Heapvariable handelt. An dieser Stelle muss genug Speicher für den Inhalt der Subvariable reserviert werden. Wir sprechen daher auch von der "Allokierungsfunktion" des Compilers.

Konsistenzbedingungen

Nun, da aba definiert ist, können die Konsistenzbedingungen an den Compiler formal aufgeschrieben werden. Wir definieren die Konsistenzrelation $consis(c, aba, d)$ zwischen der $C0$-Konfiguration c und DLX-Konfiguration d bezüglich der Allokierungsfunktion aba.

Definition 4.15 *(consis) Die Konsistenzrelation* $consis(c, aba, d)$ *beinhaltet folgende Teilbedingungen:*

- $\mathrm{e-consis}$ - *Konsistenz der* **e***lementaren Subvariablen*
- $\mathrm{p-consis}$ - **p***ointer-Konsistenz*
- $\mathrm{c-consis}$ - **c***ontrol-Konsistenz*
- $\mathrm{r-consis}$ - **r***ecursion-Konsistenz (Konsistenz des stack)*
- $\mathrm{h-consis}$ - **h***eap-Konsistenz (Konsistenz des heap memory)*
- $\mathrm{Programmintegrit\ddot{a}t}$ - $d.m_{\beta-\alpha+1}(\alpha)$ *enthält Programmcode und wird vom Programm nicht verändert (kein selbstmodifizierender Code wird generiert)*

Definition 4.16 *(e-consis)* $\mathrm{e-consis}$ *liegt vor, wenn für alle elementaren Subvariablen* $(m, x)s$ *von* c *gilt:*

$$vv((m, x)s, c) = d.m_4(aba((m, x)s, c))$$

Das bedeutet, dass alle elementaren Werte und somit auch alle Werte komplexer Variablen (bis auf pointer-Komponenten) in DLX- und $C0$*-Maschine identisch sind.*

Definition 4.17 *(p-consis)* $\mathrm{p-consis}$ *liegt vor, wenn für alle pointer-Subvariablen* $(m, x)s$ *in* c *gilt:*

$$d.m_4(aba((m, x)s, c)) = aba(vv((m, x)s, c), c)$$

Das bedeutet, dass alle Pointer in DLX- und $C0$*-Maschine auf die entsprechenden selben Variablen zeigen. Abbildung 4.17 veranschaulicht den Sachverhalt für* $vv((m, x)s, c) = (m', u)$.

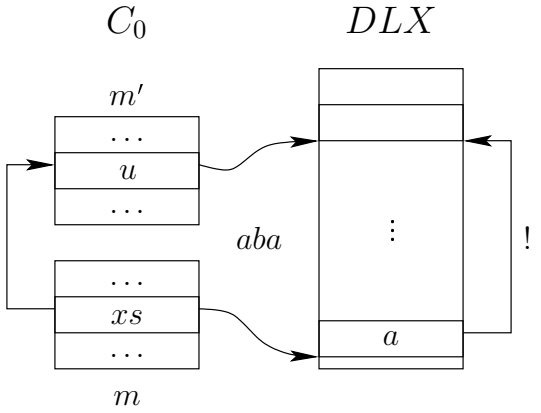


Abbildung 4.17: $\mathrm{p-consis}$

Sei a eine $C0$-Anweisungsfolge

$$a = an; a'$$

und an sei eine Anweisung. Diese lässt sich aus der Anweisungsfolge mit Hilfe der $head$-Funktion extrahieren.

$$head(a) = an$$

Sie liefert also die erste Anweisung einer Anweisungsfolge zurück. Der compilierte Code für an - $code(an)$ - wird an einer bestimmten Stelle im Speicher abgelegt, die wir mit $start(code(an))$ kennzeichnen wollen. Dabei gilt:

$$\alpha \leq start(code(an)), start(code(an)) + 4, \ldots \leq \beta$$

Definition 4.18 *(Programmintegrität) Wir fordern, dass dieser Bereich durch die Programmausführung nicht verändert wird.*

$$d.m_{\langle\beta\rangle - \langle\alpha\rangle + 1}(\alpha) = d^0.m_{\langle\beta\rangle - \langle\alpha\rangle + 1}(\alpha)$$

Die Konsistenz der Programmflusskontrolle lässt sich mit der Definition von $start$ und $head$ ebenso leicht definieren.

Definition 4.19 *(c-consis)* $\mathrm{c-consis}$ *liegt vor, wenn für den Programmrest* $c.pr = an; r'$ *gilt:*

$$d.pc = start(code(head(c.pr)))$$

Das bedeutet, dass in DLX- und $C0$-Maschine dieselbe Anweisung ausgeführt wird. $d.pc$ zeigt auf die Startadresse des compilierten Codes der ersten Anweisung des aktuellen Programmrests. Bei der Verifikation der korrekten Codeerzeugung wird diese intuitiv richtige Definition allerdings einen unerwartet komplexen Beweis erfordern.

Definition 4.20 *(r-consis)* $\mathrm{r-consis}$ *beschreibt die Konsistenz der Stackverwaltung. Dies beinhaltet im Einzelnen:*

- $d.gpr(28_5) = sbase > \beta$ *(stack base)*
- $d.gpr(30_5) = base(c.rd, c)_{32}$ *(stack pointer)*
- $base(c.rd, c) + Fsize(top(c).st.f) \leq \langle hbase \rangle$ *(stack überlappt nicht mit heap)*
- *Sei* $F_i.pbase = d.m_4((base(i, c) + 4)_{32})$, $\forall i \in [1 : c.rd].\ F_i.pbase = base(i-1, c)_{32}$ *(pbase)*
- $sbase[1:0] = hbase[1:0] = 00 \wedge \forall i \leq c.rd.\ F_i.pbase[1:0] = 00$ *(alignment)*
- *Sei* $F_i.rd = d.m_4((base(i, c) + 8)_{32})$, $\forall i \in [1 : c.rd].\ F_i.rd = aba(vv((c.lms(i), rds), c), c)$ *(rd)*
- *Sei* $F_i.ra = d.m_4(base(i, c)_{32})$. *Die Rücksprungadresse muss auf die Instruktion nach dem generierten Code für die* $C0$*-Anweisung* s *zeigen, welches den Frame* F_i *erzeugt hat. Die formale Spezifikation findet sich in Sektion 4.6.4.* *(ra)*

Für den heap memory erhalten wir folgende Bedingungen

Definition 4.21 *(h-consis)* $\mathrm{h-consis}$ *beschreibt die Konsistenz des heap memory.*

- $d.gpr(29_5) = hbase +_{32} size(c.hm.st), \langle d.gpr(29_5) \rangle < hmax$ *(heap pointer)*
- $hbase[1:0] = 00$ *(alignment)*

Sind diese Bedingungen erfüllt, so sind die DLX- und $C0$-Konfigurationen zueinander konsistent. Wir wollen nun die Konstruktion eines Compilers skizzieren, der den obigen Konsistenzbedingungen genügt. Wir beginnen mit der Übersetzung von $C0$-Ausdrücken.

4.6.2 Ausdrucksübersetzung

Wie bereits beschrieben, enthalten $C0$-Anweisungen Ausdrücke, die die Parameter für die Anweisungsausführung darstellen. Die Ausdrücke können Boolsche oder arithmetische Ausdrücke sein und Identifier enthalten. Daher müssen sie zunächst ausgewertet werden, bevor die Anweisung ausgeführt werden kann. Dazu wird für jeden Ausdruck ein individueller Code erzeugt, der dann schrittweise abgearbeitet wird bis die Ergebnisse vorliegen. Dabei müssen jedoch auch Zwischenergebnisse in den Registern abgespeichert werden und, da die Anzahl der general-purpose-Register begrenzt ist, stellt sich die Frage, wie groß Ausdrücke werden dürfen, so dass man sie noch auswerten kann. Die $C0$-Grammatik stellt keine Begrenzung an die Größe der Ausdrücke und es wäre theoretisch möglich, Konstrukte mit Millionen von Operatoren abzuleiten. Glücklicherweise stellt es sich heraus, dass man einer geschickten Herangehensweise folgend, Ausdrücke mit einer gewissen Anzahl von Operatoren mit Hilfe einer logarithmischen Anzahl von Registern auswerten kann. Diese Strategie beschreibt der "Aho-Ullmann-Algorithmus".

Aho-Ullman-Algorithmus

Der Aho-Ullmann-Algorithmus wird bei einem Spiel auf Graphen angewendet. Im speziellen Fall der Ausdrucksauswertung beschränken wir ins auf binäre Bäume, nämlich gerade die Ableitungsbäume unserer Ausdrücke. Im Spiel tätigt man Züge, indem man Marken auf die Knoten der Ableitungsbäume setzt, beziehungsweise sie wieder entfernt. Dies beschreibt die Benutzung eines Registers zur Speicherung des Ergebnisses des Teilausdruckes, der zu dem jeweiligen Knoten gehört. Es gelten die folgenden Regeln:

1. Das Blatt eines Baumes darf jederzeit markiert werden (Konstanten und Subvariablen lassen sich direkt in ein Register auswerten).
2. Die Markierung von Knoten kann jederzeit wieder aufgehoben werden (Nicht mehr benötigte Zwischenergebnisse können verworfen und die Register freigegeben werden).
3. Ein Knoten darf markiert werden, sobald alle seine Kinder markiert sind (Ausdrücke können erst ausgewertet werden wenn alle Operanden vorliegen).

Der Einfachheit halber ist es auch erlaubt, die Marke eines Kindes direkt auf dessen Vater weiterzuschieben, sofern dieser markiert werden darf. Man benötigt dann keine weitere Marke bzw. kein weiteres Register (Quellregister dürfen auch Zielregister von Instruktionen sein). Abbildung 4.18 erklärt die Regeln in schematischer Art und Weise. Ziel des Spieles ist, die Wurzel zu markieren, beziehungsweise

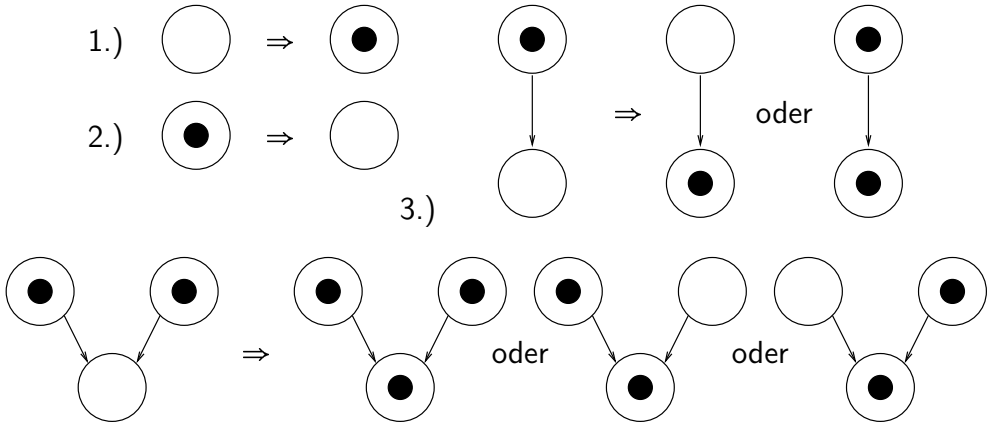
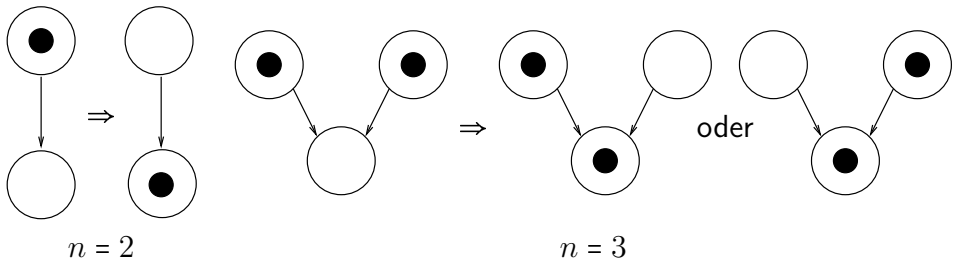


Abbildung 4.18: Marken-Spiel zum Aho-Ullman-Algorithmus

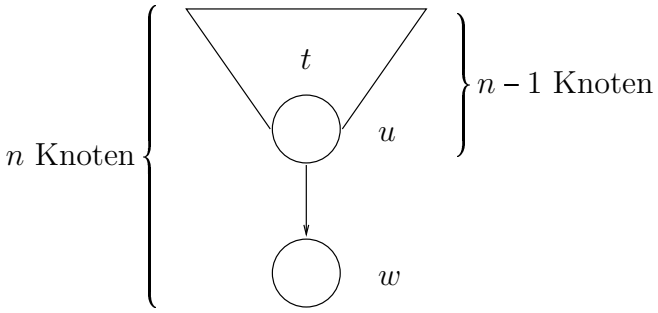
für gerichtete azyklische Graphen, ausgehend von den Quellen, alle Senken zu markieren.

Theorem 4.3 *Man kann jeden binären Baum, der $n > 1$ Knoten besitzt, mit $\lceil \log_2 n \rceil$ Marken markieren.*

Beweis: Sei $T(n)$ die maximale Anzahl von Marken, die für einen Baum mit n Knoten benötigt werden. Der Beweis erfolgt dann durch Induktion über die Knotenanzahl n. Wegen Regel 1 gilt $T(1) = 1$. Graphen mit $n = 2$ oder $n = 3$ Knoten lassen nach den Regeln 1 und 3 markieren.



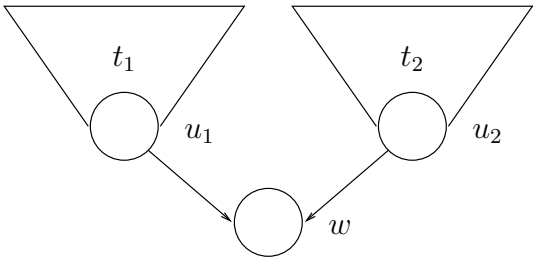
Es gilt, $T(2) = 1 \leq \lceil 1 \rceil = \lceil \log_2 2 \rceil$ und $T(3) = 2 = \lceil 1.58496 \rceil \leq \lceil \log_2 3 \rceil$. Die Behauptung stimmt also für den Induktionsanfang. Beim Induktionsschritt $n-1 \to n$ müssen zwei Fälle unterschieden werden.
Fall 1: Die Wurzel hat nur ein Kind.



In diesem Fall benötigt man, sobald das Kind markiert ist, nach Regel 3 keine weiteren Marken mehr, um die Wurzel zu markieren. Man schiebt die Marke einfach von u nach w und es gilt:

$$T(n) = T(n-1)$$

Fall 2: Die Wurzel hat zwei Kinder.



Hier wenden wir eine spezielle Strategie an, den Aho-Ullmann-Algorithmus. Zuerst wird der größere Teilbaum markiert. Ohne Beschränkung der Allgemeinheit soll dies Teilbaum t_1 sein. Weil t_2 höchstens halb so groß wie der gesamte Baum sein darf, um nicht größer als t_1 zu werden, muss dann gelten:

$$\begin{array}{ll} \text{Größe von } t_1 & \leq \quad n-2 \\ \text{Größe von } t_2 & \leq \quad \frac{n}{2} \end{array}$$

Die Strategie lautet dann:

- Markiere u_1 mit höchstens $T(n-2)$ Marken!
- Behalte nur die Marke auf u_1 und entferne alle anderen Marken!
- Markiere zusätzlich u_2 mit insgesamt maximal $T(\frac{n}{2}) + 1$ Marken!

- Halte nur die Marken auf u_1 und u_2 und schiebe dann eine Marke auf w! Dies benötigt nicht mehr als 2 Marken.

Im Induktionsschritt ist dann zu zeigen, dass:

$$T(n) = \max\{T(n-2), T(\frac{n}{2}) + 1, 2\} \overset{!}{\leq} \lceil \log_2 n \rceil$$

IV:

$$\forall j, 1 < j < n : T(j) \leq \lceil \log_2 n \rceil$$

IS: Für den ersten Fall (nur ein Kind) gilt:

$$\begin{aligned} T(n) &= T(n-1) \\ &\leq \lceil \log_2(n-1) \rceil \quad (\text{Induktionsvoraussetzung}) \\ &\leq \lceil \log_2 n \rceil \quad (\text{Logarithmus ist monoton wachsend}) \end{aligned}$$

Im zweiten Fall muss man die drei Fälle für das Maximum unterscheiden.

1. $\max\{T(n-2), T(\frac{n}{2}) + 1, 2\} = T(n-2)$, dann gilt analog zu Fall 1:

$$\begin{aligned} T(n) &= T(n-2) \\ &\leq \lceil \log_2 n \rceil \quad \checkmark \end{aligned}$$

2. $\max\{T(n-2), T(\frac{n}{2}) + 1, 2\} = T(\frac{n}{2}) + 1$, dann gilt:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 1 \\ &\leq \lceil \log_2(\frac{n}{2}) \rceil + 1 \quad (\text{Induktionsvoraussetzung}) \\ &= \lceil \log_2 n - \log_2 2 \rceil + 1 \quad (\text{Logarithmengesetze}) \\ &= \lceil \log_2 n - 1 \rceil + 1 \\ &= \lceil \log_2 n \rceil - 1 + 1 \quad (\forall a \in \mathbb{Z} : \lceil x \pm a \rceil = \lceil x \rceil \pm a) \\ &= \lceil \log_2 n \rceil \quad \checkmark \end{aligned}$$

3. $\max\{T(n-2), T(\frac{n}{2}) + 1, 2\} = 2$, dann gilt für $n \geq 3$:

$$\begin{aligned} T(n) &= 2 \\ &= \lceil \log_2 3 \rceil \\ &\leq \lceil \log_2 n \rceil \quad \checkmark \end{aligned}$$

Damit ist für alle Fälle gezeigt, dass:

$$T(n) \leq \lceil \log_2 n \rceil \qquad \square$$

Nun kann man die Ableitungsbäume nach dem Aho-Ullmann-Algorithmus auswerten. Dabei markiert man Knoten v mit Markierungen $R(v) \in \{0, 1\}$. Dies ist nötig, da man zwischen R- und L-values unterscheiden muss (vgl. 4.4.5). $R(v) = 1$ bedeutet dass man den Zahlenwert eines Ausdrucks berechnet. Bei $R(v) = 0$ hingegen ermittelt man die Bindung eines Identifiers. So würde bei einer Zuweisung $e = e'$ zum Beispiel gelten dass $R(e) = 0$ und $R(e') = 1$. Hat man jedoch einen Ausdruck der Form $e \circ e'$ mit $\circ \in \{+, -_2, \cdot, /, \wedge, \ldots\}$ dann ist $R(e) = R(e') = 1$, denn man benötigt den Zahlenwert der Ausdrücke, um die arithmetische/Boolesche Operation auswerten zu können.

Code-Erzeugung

Die Züge im Markenspiel müssen letztendlich in Assembler-Instruktionen zur Auswertung eines $C0$-Programms umgesetzt werden. Dabei werden Register in Anspruch genommen um die Zwischenergebnisse zu speichern. Dies entspricht dem Markieren eine Knotens im Markenspiel. Liegt Marke $j \in \{0,1\}^5$ auf Knoten v, so bedeutet dies, dass das zugehörige Ergebnis in $gpr(j)$ gespeichert wird. Dabei unterschieden wir, wie zuvor angesprochen, die Auswertung als L- oder R-value.
Angenommen, es werden die Züge $1, 2, \ldots, t, \ldots, T$ zur Auswertung eines Ausdrucks benötigt, dann steht $ecode(t)$ für den durch Zug t erzeugten Code. Der bisher für die Ausdrucksauswertung erzeugte Code ist:

$$Ecode(t) = ecode(0) \circ \cdots \circ ecode(t)$$

Um die Auswirkung von Code auf die DLX zu beschreiben, verwenden wir die folgende Notation.

$$d \longrightarrow^*_{Code} d'$$

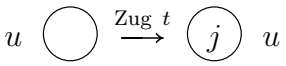
Seien zwei Konfigurationen c, d gegeben, für die anfangs $consis(c, aba, d)$ und $d = d_0$ gilt. Das heißt also, die beiden Konfigurationen entsprechen einander bezüglich der Konsistenzrelation für den Compiler. Nun soll ein Ausdruck e ausgewertet werden. In c geschieht dies gemäß der $C0$-Semantik, in d durch Ausführen von $Ecode(t)$.

$$d_0 \longrightarrow^*_{Ecode(t)} d_t$$

Unter diesen Vorraussetzungen lässt sich die folgende Behauptung für die Auswertung von e aufstellen (Induktion über Züge t).
Induktionsbehauptung: Für alle Knoten u soll in Zug t Knoten u mit Marke j markiert werden (Der Wert von Ausdruck u wird in Register j geschrieben).
Dann gilt:

$$d_t.gpr(j) = \begin{cases} va(c,u) & : \quad R(u) = 1 \wedge u \text{ ist kein Pointer} \\ aba(lv(u,c),c) & : \quad R(u) = 0 \quad (u \text{ ist ein Identifier}) \\ aba(va(c,u),c) & : \quad R(u) = 1 \wedge u \text{ ist ein Pointer} \end{cases}$$

Für Pointer Beziehungsweise Identifier wird also die Basisadresse der zugehörigen Variablen gespeichert. Die Behauptung beschreibt das gewünschte Ergebnis der Ausdrucksauswertung.



Um dies zu erreichen, muss eine Fallunterscheidung über $ecode(t)$, das heißt über den in Zug t auszuwertenden $C0$-Ausdruck u durchgeführt werden. Zur Illustration der Korrektheit wollen wir auch obige Behauptung für manche Fälle beweisen.

- $u = X$, X ist der Name einer Variablen (Identifier)

  1. X ist lokale Variable der Funktion f, dann muss die Basisadresse über den stack pointer und das displacement berechnet werden. Es wird die folgende Instruktion erzeugt.
     - `addi` j `30` $displ(f, X)$

     Der Effekt nach Zug t auf Register j wird folgendermaßen formuliert.

$$\begin{array}{lcll} d_t.gpr(j) & = & d_{t-1}.gpr(30_5) +_{32} displ(f,X)_{32} & \text{(DLX-Semantik)} \\ & = & base(c.rd, c) +_{32} displ(f,X)_{32} & (r\text{–}consis) \\ & = & aba((top(c), X), c) & \text{(Definition } aba) \\ & = & aba(bind(X, c), c) & (X \text{ lokale Variable}) \\ & = & aba(lv(u, c), c) & \text{(Definition } lv) \quad \checkmark \end{array}$$

2. X ist eine globale Variable, dann muss die Basisadresse über die stack base und das displacement berechnet werden.

   $$d_t.gpr(j) = d_{t-1}.gpr(28_5) +_{32} displ(X, gm.st)_{32}$$

   Es wird die folgende Instruktion erzeugt.

   – `addi  j  28  displ(X, gm.st)`

Mit der DLX-Semantik folgt:

$$\begin{array}{lcll} d_t.gpr(j) & = & d_{t-1}.gpr(28_5) +_{32} displ(X, gm.st)_{32} & \text{(DLX-Semantik)} \\ & = & sbase +_{32} displ(X, gm.st)_{32} & \text{(DLX-Semantik)} \\ & = & aba((gm, X), c) & \text{(Definition } aba\text{)} \\ & = & aba(lv(u, c), c) & \text{(Definition } lv \text{ und } bind\text{)} \quad \checkmark \end{array}$$

Ist $R(u) = 0$ so genügt dies und die allocated base address von u wird gespeichert. Ansonsten ist $R(u) = 1$ und der Wert der Variable soll abgefragt werden. Man spricht von *dereferenzieren*. Dies ist jedoch nur bei einfachen Variablen möglich, da in Registern nur einfache Werte gespeichert werden können. Die entsprechende zusätzliche Instruktion lautet:

– `lw  j  j  0`

Sei $d^x.gpr(j) = aba(lv(u, c), c)$. Für die Korrektheit des Dereferenzierens ergibt sich:

$$\begin{array}{lcll} d^{x+1}.gpr(j) & = & d^x.m_4(d^x.gpr(j) +_{32} 0_{32}) & \text{(DLX-Semantik)} \\ & = & d^x.m_4(aba(lv(u, c), c)) & \text{(Vorraussetzung)} \\ & = & \begin{cases} vv(lv(u, c), c) & : \ u \text{ ist kein Pointer} \\ aba(vv(lv(u, c), c), c) & : \ u \text{ ist ein Pointer} \end{cases} & (e\text{–}consis) \ (p\text{–}consis) \\ & = & \begin{cases} va(u, c), c) & : \ u \text{ ist kein Pointer} \\ aba(va(u, c), c) & : \ u \text{ ist ein Pointer} \end{cases} & \text{(Definition } va\text{)} \quad \checkmark \end{array}$$

- $u = c_0$, c_0 ist eine Konstante in Dezimaldarstellung. Sei b mit $\langle b \rangle_2 = \langle c_0 \rangle_{10}$ die Binärdarstellng von $\langle c_0 \rangle_{10}$, dann soll b in Register j gespeichert werden.

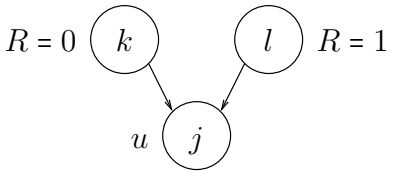
  $$d_t.gpr(j) = b$$

  In der DLX benötigt dies 2 Instruktionen, da die Konstante bis zu 32 Bit in Anspruch nimmt und immediate-Konstanten lediglich 16 Bit fassen können. Der Code lautet:

  – `lhgi  j  b[31 : 16] ⊕ b[15]`
  – `xori  j  j  b[15 : 0]`

  Wir kürzen diesen Code mit dem Makro $geti(j, b)$ ab. Die oberen Bits von b lassen sich nicht ohne weiteres mit `lhgi` laden, da die unteren Bits wegen der sign extension eventuell mit Einsen aufgefüllt werden, falls $b[15]$ Eins ist. Dann würde das `xori` die oberen Bits in Register j invertieren. Deshalb invertiert man sie schon beim Hereinladen in Abhängigkeit von $b[15]$. Dies wird alles vom Compiler erledigt, der den Wert und die Bitdarstellung der Konstante „kennt“.

- $u = e[e']$, u ist eine array-Komponente und $typ(e) = t'[n] = t$ für ein $n \in \mathbb{Z}$



Zunächst werden die Bindung von e und der Wert von e' mit $R(e) = 0$ bzw. $R(e') = 1$ ermittelt und in den Registern k und l gespeichert.

$$\begin{aligned} d^x.gpr(k) &= aba(lv(e,c),c) \\ d^x.gpr(l) &= va(e',c) \end{aligned}$$

Dann kann die Bindung von $e[e']$ für $R(u) = 0$ bestimmt werden.

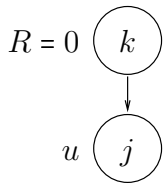
$$d^{x'+1}.gpr(j) = d^{x'}.gpr(k) +_{32} (\langle d_x.gpr(l)\rangle \cdot size(t'))_{32}$$

Man beachte die Multiplikation von $size(t')$ mit $d_x.gpr(l)$ im obigen Ausdruck. Während der Compiler zu Compilezeit bekannte Konstanten natürlich direkt multiplizieren kann, muss die obige Multiplikation zur Laufzeit ausgeführt werden. In Ermangelung eines DLX-Befehls zur Multiplikation werden solche Multiplikationen standartmäßig vom Compiler als Software-Multiplikation durch sukzessives Addieren in der Hardware implementiert, was sehr langsam ist. Es lässt sich jedoch ein optimierter Algorithmus finden, der die Multiplikation in logarithmischer Laufzeit abwickelt. Nachdem die Assembly-Multiplikationsroutine den benötigten Wert in Register l zum Zeitpunkt x' berechnet hat, kann der L-value von u mittels einer `add`-Instruktion in Register j erzeugt werden.

$$\begin{aligned}
d^{x'+1}.gpr(j) &= d^{x'}.gpr(k) +_{32} d^{x'}.gpr(l) && \text{(DLX-Semantik)} \\
&= d^{x}.gpr(k) +_{32} (\langle d^{x}.gpr(l)\rangle \cdot size(t'))_{32} && \text{(SW-Multiplikation)} \\
&= aba(lv(e,c),c)) +_{32} (va(e',c) \cdot size(t'))_{32} && \text{(Vorraussetzung)} \\
\text{1. Fall } e \text{ lokal}: &= base(j,c)_{32} +_{32} displ(lv(e,c),f)_{32} +_{32} (va(e',c) \cdot size(t'))_{32} && \text{(Definition } aba\text{)} \\
&= base(c.rd,c)_{32} +_{32} (displ(lv(e,c),f) + (va(e',c) \cdot size(t'))_{32} && \text{(Lemma 1.11)} \\
&= base(c.rd,c)_{32} +_{32} displ(lv(e,c)[va(e',c)],f)_{32} && \text{(Definition } displ(xs[k],st)\text{)} \\
&= aba(lv(e,c)[va(e',c)],c) && \text{(Definition } aba\text{)} \\
&= aba(lv(e[va(e',c)],c),c) && \text{(Definition } lv\text{)} \\
&= aba(lv(u,c),c) && \checkmark \\
\text{2. Fall } e \text{ global}: & \quad \text{analog}
\end{aligned}$$

Ist $R(u) = 1$, so muss die Adresse in Register j noch zusätzlich dereferenziert werden. Der zu erzeugende Code ergibt sich entsprechend der vorhergehenden Fälle.

- $u = e.n_i$, u ist eine struct-Komponente mit $typ(e) = t = struct\{t_1\ n_1; \ldots; t_s\ n_s\}$



Angenommen e wurde schon in Register k ausgewertet ($R(e) = 0$). Dann liegt die Basisadresse der Subvariable schon vor und es muss nur noch der displacement offset der i-ten Komponente aufaddiert werden.

$$\begin{aligned} d_{t-1}.gpr(k) &= aba(lv(e,c),c) \\ d_t.gpr(j) &\overset{!}{=} aba(lv(e.n_i,c),c) \end{aligned}$$

Man kann $k = j$ wählen (Marke schieben) und kein weiteres Register wird benötigt.

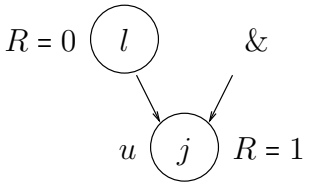
- $u = e*$, e ist ein Pointer der dereferenziert werden soll.
  Liegt der Wert des Pointers e ($R(e) = 1$) in Register j vor und ist $R(u) = 0$, so muss kein weiterer Code erzeugt werden. Der L-value für u liegt dann schon in Register j bereit.

$$\begin{aligned} d_{t-1}.gpr(j) &= aba(va(e,c),c) \quad \text{(Vorraussetzung)} \\ &= aba(lv(e*,c),c) \quad \text{(Definition } lv\text{)} \\ &= aba(lv(u,c),c) \quad \checkmark \end{aligned}$$

  Für $R(u) = 1$ muss man wie gewohnt dereferenzieren.

  - `lw  j  j  0`

- $u = e\&$, u ist die Adresse von Subvariable e (Pointer auf e, $R(u) = 1$)

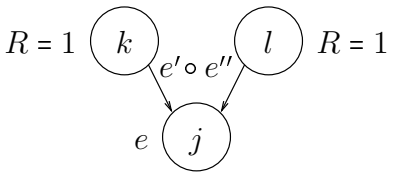


  Ist der Wert von Identifier e ($R(e) = 0$) in Register l vorhanden, so hat man auch hier bereits das gesuchte Ergebnis vorliegen, und man kann es direkt an Register j übergeben. Wählt man $l = j$ (Marke schieben), so muss gar kein Code erzeugt werden.

$$\begin{aligned} d_{t-1}.gpr(l) &= aba(lv(e,c),c) \quad \text{(Vorraussetzung)} \\ &= aba(va(e\&,c),c) \quad \text{(Definition } va\text{)} \\ &= aba(va(u,c),c) \quad \checkmark \end{aligned}$$

Es kann nun Code generiert werden, um alle möglichen $C0$-Identifier auszuwerten. Die Auswertung von $C0$-Ausdrücken für arithmetische und boolesche Ausdrücke sowie Vergleichsoperationen geschieht dann über die entsprechenden DLX-Instruktionen für die jeweiligen Operationen.

- $e = e' \circ e''$, e' und e'' sind Ausdrücke, $\circ \in \{+, -_2, \cdot, /, >, \geq, <, \leq, ==, \neq\}$



  Für die verschiedenen Operatoren werden folgende Instruktionen generiert.

$+$	`add`	j	k	l
$-_2$	`sub`	j	k	l
$\cdot$	`jal`	*swmult*		
$/$	`jal`	*swdiv*		
$==$	`seq`	j	k	l

$>$	`sgr`	j	k	l
$\geq$	`sge`	j	k	l
$<$	`sls`	j	k	l
$\leq$	`sle`	j	k	l
$\neq$	`sne`	j	k	l

  Da die hier vorgestellte DLX keine Multiplikations- und Divisionsinstruktionen zur Verfügung stellt, müssen diese Operationen durch Assembly-Routinen im Programm emuliert werden. Zur Berechnung einer entsprechenden Operation springt die Ausführung zu der jeweiligen Routine und kehrt später mit dem korrekten Ergebnis zurück. j, k und l müssen den Anforderungen der Routinen an die Ein- und Ausgangsregister entsprechend gewählt werden.

Ist e'' eine Konstante, so könnte ein optimierter Compiler die entsprechenden $Itype$-Instruktionen wählen. Ist e' eine Konstante, so muss diese entweder in ein Register geladen werden oder es können eventuell die Operanden vertauscht werden. Dabei sind gegebenenfalls weitere Anpassungen und zusätzliche Instruktionen nötig. Sind sowohl e' als auch e'' Kontanten, so kann der Compiler den Ausdruck direkt auswerten. Nach Ausführung des Codes muss gelten:

$$
\begin{aligned}
d_t.gpr(k) &= d_{t-1}.gpr(k) \circ_{32} d_{t-1}.gpr(l) \\
&= \begin{cases}
va(e',c) \circ va(e'',c) & : \quad \circ \in \{\wedge, \vee\} \\
va(e',c) \pm_{32} va(e'',c) & : \quad \circ \in \{+, -_2\} \\
twoc_{32}([va(e',c)] \circ [va(e'',c)]) & : \quad \circ \in \{*, /\} \\
0^{31}([va(e',c)] \circ [va(e'',c)]) & : \quad \circ \in \{<, >, \leq, \geq, ==, \neq\}
\end{cases}
\end{aligned}
$$

- $e = \circ e'$, e' ist ein Ausdruck, $\circ \in \{\sim, -_1\}$ und $R(e) = 1$
  Für die Negation nutzen wir die Eigenschaft $x \oplus 1 = \sim x$. Es stehe der Wert von e' ($R(e') = 1$) in Register j. Wir negieren nur das unterste Bit, damit die übrigen Bits 0 bleiben.
  - `xori` j j `1`

  Zur Implemenentierung des unären Minus ziehen wir den Wert von j von 0 ab.
  - `sub` j k j

  Dazu ist allerdings ein Register k nötig, das 0 enthält. Alternativ kann auch die Eigenschaft $-[x] = [\overline{x}] + 1$ genutzt werden.
  - `xori` j j $2^{16} - 1$
  - `addi` j j `1`

Nach Ausführung von $code(e) = Ecode(t)$ in t Schritten ist die Wurzel des Ableitungsbaumes markiert. Das heißt, der Wert des gesamten Ausdrucks steht in einem Register. Darauf baut dann die Übersetzung von Anweisungen auf.

4.6.3 Anweisungsübersetzung

Die Übersetzung der Anweisungen hängt offensichtlich vom Typ der Anweisung ab, die der Compiler übersetzen soll. Im Folgenden wird davon ausgegangen, dass auftretende Ausdrücke e durch $code(e) = Ecode(T)$ in mit T Instruktionen (T Züge) übersetzt werden. Das Ergebnis steht dann in einem Register j (Wurzel des Ableitungsbaumes mit Marke j markiert). Ist $a = an; a'$ eine Anweisungsfolge mit Anweisung an dann ist der zugehörige Assemblercode eine Aneinanderreihung der Code-Stücke für die einzelnen Anweisungen.

$$code(a) = code(an)\ code(a')$$

Für $code(an)$ stellen wir eine Fallunterscheidung über an an.

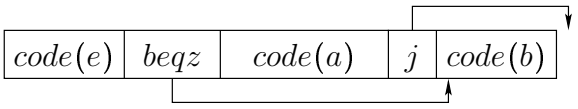
- $an:\quad e = e'$ (Zuweisung)
  Der Typ von e bzw. e' sei hierbei *einfach*, dann gilt $R(e) = 0$ und $R(e') = 1$. $code(e)$ und $code(e')$ müssen zuvor ausgeführt werden und wir speichern die Adresse von e und den Wert von e' in Register j bzw. $k \neq j$. Bei der Auswertung von e' darf dann selbstverständlich das Register j nicht verwendet werden. Die Konfiguration d' nach Ausführung von $code(e)\ code(e')$ lautet:

  $$
  \begin{aligned}
  d'.gpr(j) &= aba(lv(e,c),c) \\
  d'.gpr(k) &= \begin{cases}
  va(e',c) & : \quad e' \text{ ist kein Pointer} \\
  aba(va(e',c),c) & : \quad e' \text{ ist ein Pointer}
  \end{cases}
  \end{aligned}
  $$

  Dann muss nur noch der Wert von e' der Adresse von e zugewiesen werden.

  - `sw` k j `0`

- $an:$ $if\ e\ then\ \{a\}\ else\ \{b\}$ (Fallunterscheidung)
  Zuerst wird e mit $code(e)$ in Register k ausgewertet. Je nachdem wird danach zu $code(a)$ oder $code(b)$ gesprungen. Nach der Ausführung vom $then$-Teil muss der Code für den $else$-Fall übersprungen werden. Die folgende Abbildung zeigt die Struktur des zu erzeugenden Codes.



  Gebe $|(code(A)|$ die Länge des für Anweisungsfolge A erzeugten Codes an, dann wird für die if-Anweisung der folgende Code produziert.

  - $code(e)$
  - `beqz` k $|code(a)| + 8$
  - $code(a)$
  - `j` $|code(b)| + 4$
  - $code(b)$

- $an:$ $while\ e\ do\ \{a\}$ (Schleife)
  Die Übersetzung der $while$-Schleife soll als Übung durchgeführt werden.

- $an:$ $id = f(e_1, \ldots, e_p)$ (Funktionsaufruf in Rumpf von Funktion f')
  Ein Funktionsaufruf ist die aufwendigste Anweisung, denn es muss ein neuer function frame auf dem stack erzeugt werden und der Programmfluss zu einer neuen Funktion umgelenkt werden. Die folgenden Teilaufgaben sind zu erledigen.

  1. heap und stack dürfen nicht aufeinander treffen. Es gilt:

     $$\langle d.gpr(30_5)\rangle + Fsize(f') + Fsize(f) \overset{!}{<} \langle hbase\rangle$$

     Dazu muss die Distanz zwischen heap und stack nach dem geplanten Aufruf ausgewertet werden. Der Test-Code lautet:

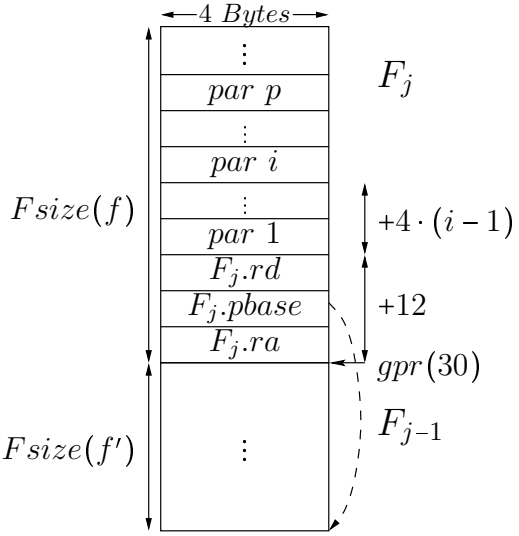
– `addi 1 30` $Fsize(f') + Fsize(f)$;	R1=neuer stkptr
– $geti(2, hbase)$;	R2=$hbase$
– `sls 4 1 2` ;	R4=0^{31}([stkptr] < [$hbase$]?)
– `xor 3 1 2` ;	R3[i]=(stkptr[i] ≠ $hbase[i]$)
– `slsi 3 3 0` ;	R3<0⇔R3[31]=1⇔stkptr[31]≠$hbase[31]$ R3=0^{31}(stkptr[31] ≠ $hbase[31]$?)
– `xor 4 4 3` ;	R4=(R3=1 ? R4[31 : 1]$\overline{\text{R4[0]}}$: R4) =0^{31}(⟨stkptr⟩ < ⟨$hbase$⟩?)
– `bnez 1 8` ;	kein Abbruch wenn ⟨stkptr⟩ < ⟨$hbase$⟩
– $code(Abbruch)$;	Fehlermeldung an Betriebssystem

     Da nur Vergleichsoperationen für Two's-Complement-Zahlen zur Verfügung stehen, die vorliegenden Werte aber $unsigned$ sind, ist dieser etwas aufwendiger. Der Abbruch-Code ist eine $trap$-Instruktion, die den Betriebssystemkern aufruft. Dieser kann dann zum Beispiel dem Programm mehr Speicher zuweisen.

  2. Die Parameter müssen übergeben werden. Dabei ist zu beachten, dass nur Parameter mit einfachen Typen übergeben werden können. Da in der Parameterliste Ausdrücke stehen, müssen diese zuerst zu $pcode(1) \cdots pcode(p)$ übersetzt und jeweils in Register k_i ausgewertet werden. Es gilt $R(e_i) = 1$, dass heißt, das bei $C0$-Funktionsaufrufen die Werte von Variablen übergeben werden und nicht Referenzen darauf (*call by value*). Der Code $pcode(i)$ zur Übergabe von Parameter i lautet dann:

- $code(e_i)$
- `sw` k_i `30` $Fsize(f') + 4 \cdot (3 + (i-1)))$

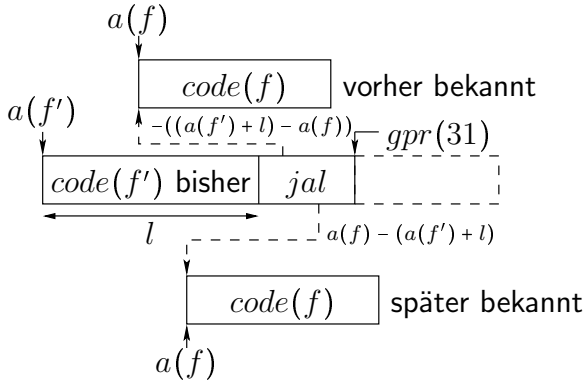
3. Der alte stack pointer muss in $pbase$ des neuen function frame gespeichert werden, damit die Programmausführung nach dem Beenden von f zu f' zurückkehren kann. Dazu wird folgende Instruktion erstellt.
   - `sw` `30` `30` $Fsize(f') + 4$



4. Der Rückgabewert der Funktion soll in id gespeichert werden. Dazu muss die Adresse von id in $F_j.rd$ geschrieben werden. Wir werten zunächst id mit $R(id) = 0$ in Register k aus.
   - $code(id)$
   - `sw` k `30` $Fsize(f') + 8$
5. Nun muss noch der stack pointer erhöht werden, wodurch der neu angelegte function frame zum top frame wird.
   - `addi` `30` `30` $Fsize(f')$
6. Zuletzt soll zum Rumpf von f gesprungen werden. Dazu definieren wir die Funktion

   $$a(f) = start(code(f)),$$

   die die Startadresse von $code(f)$, dem generierten Code für Funktion f, angibt. Dabei gibt es zwei Möglichkeiten.



Fall 1: $code(f)$ wurde bereits übersetzt, dann ist $a(f)$ bekannt und es kann dorthin gesprungen werden. Es ist auch $f = f'$ möglich. Da die Sprungweite für jal relativ angegeben werden muss, wird die Länge l des bisher für f' erzeugten Codes benötigt.

– `jal` $-(a(f') + l - a(f))$

Fall 2: $code(f)$ wird erst später übersetzt, dann ist $a(f)$ noch nicht bekannt. Um dieses problem zu beheben, werden Programme vom Compiler in zwei Durchgängen übersetzt. Im ersten Durchlauf wird alles soweit wie möglich übersetzt, bis auf die Sprungweiten. Diese werden in der zweiten Phase eingesetzt, wenn die Positionen der Funktionsrümpfe im Code bekannt sind. Die Sprunginstruktion lautet dann ebenfalls:

– `jal` $a(f) - (a(f') + l)$

Für lange Sprünge kann die immediate Konstante eventuell nicht ausreichen und man muss die Sprungweite zunächst mit $geti(j, a(f))$ durch zwei Instruktionen in ein Register laden. Dafür kann man das Sprungziel absolut angeben.

$$gpr(j) = bin_{32}(a(f))$$

Der Sprungbefehl ist daher:

– `jalr` j

Es werden stets jump-and-link Instruktionen benutzt, da diese den nächsten sequentiellen PC für die Rücksprungadresse sichern.

7. Am Anfang von $code(f)$ der aufgerufenen Funktion muss sofort $gpr(31_5)$ nach $F_j.ra$ gerettet werden. In dieses Register hatte die $jal/jalr$-Instruktion zuvor die Rücksprungadresse gesichert.

   – `sw 31 30 0`

   Dementsprechend ist $code(f)$ =`sw 31 31 0`, $code(ft(f).body)$.

- $an:$ $id = new\ t^*$ (Speicher auf heap allozieren)
  Die Übersetzung der new-Anweisung soll als Übungsaufgabe gestellt werden.
- $an:$ $return\ e$ (Rücksprung)
  Auch die $return$-Anweisung soll selbständig übersetzt werden.

Der Bau des Compilers ist nun abgeschlossen und $C0$-Programme können nach der Übersetzung in Assembler auf der DLX ausgeführt werden. Sie laufen korrekt ab, solange der Compiler die Konsistenzbedingungen beachtet.

4.6.4 Korrektheitsbeweis

Im Folgenden soll anhand eines nichttrivialen Teils der Konsistenzrelation gezeigt werden, dass der Compiler korrekten Code erzeugt. Wir betrachten die control-Konsistenz $\mathrm{c-consis}$, die wie folgt definiert wurde.

$$d.pc = start(code(head(c.pr)))$$

Der program counter muss vor Ausführung eines $C0$-statements auf den Anfang des generierten Codes zeigen, damit sich $C0$- und DLX-Programm in konsistenten Zuständen befinden. Damit diese Bedingung für alle Zustände des $C0$-Programms gilt, muss der program counter insbesondere nach Ausführung des generierten Codes für eine $C0$-Anweisung auf die Start-Adresse der nächsten Anweisung im Programmrest zeigen. Dies ist aber im Allgemeinen nicht der Fall, da Anweisungen in $while$-Schleifen oder $if\text{-}then\text{-}else$-Fallunterscheidungen geschachtelt sein können. Im generierten Code gelangt man dann erst nach einer Anzahl von Sprüngen zum Start des Codes für die nächste Anweisung im Programmrest.

Es gelte $consis(c, aba, d^p)$ und $head(c.pr)$ sei eine Anweisung a. Im Allgemeinen ergibt sich folgende hardware-Rechnung der DLX.

$$d^p \longrightarrow^*_{code(a)} d^q \longrightarrow_j \cdots \longrightarrow_j d^r$$

Nun soll $consis(c', aba, d^r)$ gelten. Es lässt leicht zeigen, dass $d^q.pc = end(code(a)) + 4$ ist, was man für $\mathrm{c-consis}$ benötigt ist allerdings

$$d^r.pc = start(code(head(c'.pr))).$$

Konsistenzbedingung an Rücksprungadressen

Wir nummerieren die Anweisungen im Programmrest in der Form $c.pr(i)$ durch und bezeichnen mit $r(j,c) \in \mathbb{N}$ die $return$-Anweisung im Programmrest von c, die den j-ten Rekursionschritt abschließt.

$$\begin{array}{lcl} r(c.rd,c) & = & \min\{i \mid c.pr(i) \Leftarrow_{C0} return\ \langle A\rangle\} \\ r(j-1,c) & = & \min\{i \mid i > r(j,c) \wedge c.pr(i) \Leftarrow_{C0} return\ \langle A\rangle\} \end{array}$$

Das return-statement in $main$ wird zum Beispiel durch $r(0,c)$ identifiziert, das nächste return-statement in $c.pr$ hingegen durch $r(c.rd,c)$. Außerdem gilt die folgende Invariante.

$$\#\{i \mid c.pr(i) = return\ \langle A\rangle\} = c.rd + 1;$$

Bei der Definition von $\mathrm{r-consis}$ blieben wir eine Formalisierung der Korrektheitsbedingung an $F_i.ra$ schuldig. Ein naiver Versuch einer Definition könnte lauten:

$$\forall i \in [1 : c.rd].\ F_i.ra = start(code(c.pr(r(i,c)+1)))$$

Dies stimmt aber nicht für Funktionsaufrufe, die in while- oder if-$then$-$else$-Anweisungen geschachtelt sind, aus dem selben Grund, der oben bereits veranschaulicht wurde. Es muss viel mehr gefordert werden, dass zur Instruktion zurückgesprungen wird, die dem generierten Code für das *call statement* von F_i folgt. Das call statement von F_i ist dabei derjenige $C0$-Funktionsaufruf, der F_i erzeugt hat.

Nun wird in der $C0$-Semantik der Programmrest lediglich Schritt für Schritt abgearbeitet und nicht darüber Buch geführt, welche Anweisungen im Ableitungsbaum des Programms welche function frames erzeugt haben. Für die Spezifikation und den Beweis des Compilers führen wir eine solche nicht für die Funktionsweise benötigte Komponente für c ein. Man spricht hier auch von *ghost*-Komponenten eines Systems.

$$c.calls\ :\ [1 : c.rd] \longrightarrow \{s \mid s \Leftarrow_{C0} \langle id\rangle = \langle Na\rangle(\langle PaF\rangle) \text{ ist call statement im Ableitungsbaum}\}$$

Für Funktionsaufrufe $s\ :\ id = f(e_1,\ldots,e_p)$ setzen wir

$$c'.calls(c'.rd) = s.$$

Ansonsten lassen wir $c.calls$ unverändert. Nun kann die Konsistenzbedingung an die Rücksprungadressen formuliert werden.

$$\forall i \in [1 : c.rd].\ F_i.ra = end(code(c.calls(i))) + 4$$

Für den Beweis der Kontrollkonsistenz müssen zudem weiterer Formalismus eingeführt werden, um über die beliebige Verschachtelung von while- und if-$then$-$else$-Anweisungen, sowie die resultierenden Sprünge am Ende von eingebetteten Anweisungsfolgen argumentieren zu können.

Definitionen und Lemmata

Wir betrachten $C0$-Funktionsrümpfe $a_1;a_2;\ldots;a_{j-1};a_j;return\ \langle A\rangle \Leftarrow_{C0} \langle rumpf\rangle$ im Ableitungsbaum eines Programms. Innerhalb der Anweisungsfolge $a_1;a_2;\ldots;a_{j-1};a_j$ lässt sich die direkte Nachfolgeanweisung $dsucc$ definieren. So ist zum Beispiel $dsucc(a_1) = a_2$ und $dsucc(a_{j-1}) = a_j$. Auf der letzten Anweisung a_j ist $dsucc$ nicht definiert. Es gilt allgemein:

$$dsucc(a_x) = a_{x+1} \qquad dsucc(a_j) = \perp$$

Sei $s\ :\ if\ e\ then\ \{a_1;\ldots;a_j\}\ else\ \{b_1;\ldots;b_k\}$ eine $C0$-Statement. Die letzten $C0$-Anweisungen im if-Teil beziehungsweise $else$-Teil der Fallunterscheidung werden durch die Prädikate $lastif$ und $lastelse$ identifiziert. Für s gilt $lastif(a_j)$ und $lastelse(b_k)$. Wir definieren für diese Anweisungen eine Vateranweisung mit Hilfe der Funktion fa, die hier wie folgt definiert ist.

$$fa(a_j) = fa(b_k) = s$$

Für $while$-Schleifen $s : while\ e\ do\ \{a_1; \ldots; a_j\}$ definieren wir analog ein Prädikat $lastwhile(a_j)$ und setzen $fa(a_j) = s$. Um darüber sprechen zu können, welche Statements s im Ableitungsbaum "folgen", definieren wir die Funktion $succ$.

$$succ(s) = \begin{cases} dsucc(s) & : \quad dsucc(s) \neq \bot \\ fa(s) & : \quad lastwhile(s) \\ succ(fa(s)) & : \quad lastif(s) \vee lastelse(s) \end{cases}$$

Ist s Teil einer Anweisungsfolge, so entspricht der Nachfolger der direkt nachfolgenden Anweisung. Für die letzten Anweisungen in if-, $else$- oder $while$-Statements ist der Nachfolger entweder das der if-$then$-$else$-Anweisung folgende Statement oder die $while$-Schleife selbst, da diese bei der Ausführung stets noch einmal dem Programmrest hinzugefügt wird.

Wir benötigen zwei weitere technische Definitionen. Die Funktion $depth$ liefert die Schachtelungstiefe der letzten Anweisung in einer Anweisungsfolge.

$$depth(s) = \begin{cases} 0 & : \quad dsucc(s) \neq \bot \\ depth(fa(s)) + 1 & : \quad \text{sonst} \end{cases}$$

Der Beweis der Kontrollkonsistenz nach Ausführung eines Statements s wird später durch Induktion über $depth(s)$ geführt werden. Darüber hinaus definieren wir die Sprungdistanz $jdis$ für ein Statement s, die der Anzahl von generierten Sprunginstruktionen entspricht, die benötigt werden um in der DLX von der Adresse $end(code(s)) + 4$ zur Adresse $start(succ(d))$ zu gelangen.

$$jdis(s) = \begin{cases} 0 & : \quad dsucc(s) \neq \bot \\ 1 & : \quad lastwhile(s) \\ jdis(fa(s)) & : \quad lastelse(s) \\ 1 + jdis(fa(s)) & : \quad lastif(s) \end{cases}$$

Für Anweisungen inmitten einer Anweisungsfolge sind keine Sprünge notwendig. Die Anweisung am Ende einer $while$-Schleife springt zurück zum Anfang des $while$-Statements, welches gerade $succ(s)$ ist. if-$then$-$else$-Anweisungen können beliebig geschachtelt werden, daher entspricht die Sprungdistanz für die letzte Anweisung eines $else$-Blockes der der Vater-Anweisung. Für if-Blöcke kommt zusätzlich der Sprung über den $else$-Teil hinzu.

Lemma 4.4 *(jump distance) Sei d^t die DLX-Konfiguration nach Ausführung des generierten Codes für ein $C0$-Statement s. Dann gilt:*

$$d^t.pc = end(code(s)) + 4 \ \Rightarrow\ d^{t+jdis(s)}.pc = start(code(succ(s)))$$

Beweis: per Induktion über $D = depth(s)$

Induktionsanfang: $D = depth(s) = 0$, dann folgt nach Definition $dsucc(s) \neq \bot$ und $jdis(s) = 0$. Folglich gilt der allgemeine Fall einer Anweisung inmitten einer Anweisungsfolge und es gilt $end(code(s)) + 4 = start(code(succ(s)))$. Somit:

$$\begin{aligned} d^t.pc = end(code(s)) + 4 \quad &\Leftrightarrow \quad d^t.pc = start(code(succ(s))) \\ &\Leftrightarrow \quad d^{t+0}.pc = start(code(succ(s))) \\ &\Leftrightarrow \quad d^{t+jdis(s)}.pc = start(code(succ(s))) \quad \checkmark \end{aligned}$$

Induktionsschritt: $D - 1 \to D$, wir müssen drei Fälle betrachten.

1. $lastwhile(s)$: s ist die letzte Anweisung in einer $while$-Schleife.

$$\begin{aligned} d^t.pc = end(code(s)) + 4 \quad &\Rightarrow \quad d^{t+1}.pc = start(code(fa(s))) && \text{(jump, Codeerzeugung } while) \\ &\Rightarrow \quad d^{t+jdis(s)}.pc = start(code(fa(s))) && \text{(Definition } jdis) \\ &\Rightarrow \quad d^{t+jdis(s)}.pc = start(code(succ(s))) && \text{(Definition } succ) \quad \checkmark \end{aligned}$$

2. $lastif(s)$: s ist die letzte Anweisung in einem if-Block.

$$d^t.pc = end(code(s)) + 4$$

$\Rightarrow$	$d^{t+1}.pc = end(code(fa(s))) + 4$	(jump über $else$, Codeerzeugung if)
$\Rightarrow$	$d^{t'}.pc = end(code(s')) + 4$	(Substitution $t' = t+1$, $s' = fa(s)$)
$\Rightarrow$	$d^{t'+jdis(s')}.pc = start(code(succ(s')))$	(IV, $depth(s') = D-1$)
$\Rightarrow$	$d^{t+(1+jdis(fa(s)))}.pc = start(code(succ(fa(s))))$	(Rücksubstitution, Assoz. $+$)
$\Rightarrow$	$d^{t+jdis(s)}.pc = start(code(succ(s)))$	(Definitionen $jdis$, $succ$) ✓

3. $lastelse(s)$: s ist die letzte Anweisung eines $else$-Blocks. Aus der Codeerzeugung für if-$then$-$else$ folgt $end(code(s)) = end(code(fa(s)))$. Somit gilt:

$$d^t.pc = end(code(s)) + 4 = end(code(fa(s))) + 4$$

$\Rightarrow$	$d^t.pc = end(code(s')) + 4$	(Substitution $s' = fa(s)$)
$\Rightarrow$	$d^{t+jdis(s')}.pc = start(code(succ(s')))$	(IV, $depth(s') = D-1$)
$\Rightarrow$	$d^{t+jdis(fa(s)))}.pc = start(code(succ(fa(s))))$	(Rücksubstitution)
$\Rightarrow$	$d^{t+jdis(s)}.pc = start(code(succ(s)))$	(Definitionen $jdis$, $succ$) ✓

Aus der Korrektheit aller drei Fälle für $jdis(s) > 0$ folgt die Behauptung. □

Bei der Anwendung des Lemmas im Beweis der Kontrollkonsistenz muss unterschieden werden, ob es sich bei s um einen Funktionsaufruf handelt oder nicht. Im Falle eines Funktionsaufrufes gilt die Bedingung $d^t.pc = end(code(s)) + 4$ nur unter der Vorraussetzung, dass die return-Anweisung der aufgerufenen Funktion zur richtigen Adresse zurückspringt. Dies folgt aus der Gültigkeit der zuvor angesprochenen $\mathrm{r-consis}$-Bedingung an die Rücksprungadressen im Stack.

$$\forall i \in [1 : c.rd].\ d.m_4(base(i,c)_{32}) = end(code(c.calls(i))) + 4$$

Deren Korrektheit kann gezeigt werden durch Argumentation über das Zusammenspiel von $jalr$ bei der Codeerzeugung für Funktionsaufrufe und das Erzeugen der sw-Instruktion zum Speichern der return address als erste Instruktion von $code(f)$. Wir benötigen zudem für den Beweis der Kontrollkonsistenz ein weiteres Lemma, welches den Zusammenhang zwischen $succ$ und $c.pr$ herstellt.

Lemma 4.5 *(successor) Sei $c.pr(i)$ das i-te Statement des Programmrestes und $i+1 < r(0,c)$. Dann gilt für das $i+1$-te Statement:*

$$c.pr(i+1) = \begin{cases} succ(c.calls(j)) & : \quad i = r(j,c) \\ succ(c.pr(i)) & : \quad \text{sonst} \end{cases}$$

Die oben angesprochene Fallunterscheidung wird hier also über die $return$-Statements geführt.

Beweis: Wir betrachten die Ausführung eines $C0$-Programms und die zugehörige Folge von Konfigurationen.

$$c^0, c^1, \ldots, c^x$$

Der Beweis des Lemmas erfolgt durch Induktion über x

Induktionsanfang: $x = 0$,

$$\begin{aligned} c^0.pr &= ft(main).body = a_1; \ldots; a_j; return\ e \\ c^0.pr(i+1) &= a_{i+1} = dsucc(a_i) = succ(a_i) = succ(c.pr(i)) \quad ✓ \end{aligned}$$

Induktionsschritt: $x \to x+1$, wir müssen eine Fallunterscheidung über $c^x.pr(1)$ durchführen.

1. Die nächste Anweisung ist eine Zuweisung, new-Anweisung, $return$-Anweisung, eine $while$-Schleife mit ungültiger Schleifenbedingung oder eine $if\text{-}then$-Anweisung mit ungültiger Eingangsbedingung.

   $$\begin{array}{lcll} c^{x+1}.pr & = & tail(c^x.pr) = c^x.pr(2,3,4,\ldots) & (C0\text{-Semantik}) \\ c^{x+1}.pr(i) & = & c^x.pr(i-1) & \checkmark \end{array}$$

   Die Korrektheit folgt aus der Induktionsvorraussetzung für c^x.

2. $c^x.pr = \underbrace{while\ e\ do\ \{a_1;\ldots;a_j\}}_{s};c_1;c_2;\ldots$ und $va(e,c^x) \neq 0^{32}$

   $$c^{x+1}.pr = a_1;\ldots;a_j;s;c_1;c_2;\ldots \quad (C0\text{-Semantik } while)$$

   a_1 bis a_j sind direkte Nachfolger und die Behauptung gilt für $i \in [1:j-1]$ analog zum Beweis des Induktionsanfangs. Für $i > j$ gilt die Induktionsvorraussetzung analog zu Fall 1 mit angepasster Indexverschiebung. Nur der Fall $i = j$ ist noch interessant.

   $$c^{x+1}.pr(j+1) = s = fa(a_j) = succ(a_j) = succ(c^x.pr(j)) \quad \checkmark$$

3. $c^x.pr = \underbrace{if\ e\ then\ \{a_1;\ldots;a_j\}\ else\ \{b_1;\ldots;b_k\}}_{s};c_1;c_2;\ldots$ und $va(e,c^x) = 0^{32}$

   $$c^{x+1}.pr = b_1;\ldots;b_k;c_1;c_2;\ldots \quad (C0\text{-Semantik } if\text{-}then\text{-}else)$$

   Für b_1 bis b_k kann wieder der Ansatz des Induktionsanfangs gewählt werden. Für $i > k+1$ gilt die Induktionsvorraussetzung analog zu Fall 1. Es bleibt der Fall $i = k$

   $$\begin{array}{lcll} c^{x+1}.pr(k+1) & = & c_1 & (C0\text{-Semantik}) \\ & = & succ(s) & (\text{Induktionsvorraussetzung}) \\ & = & succ(fa(b_k)) & (\text{Definition } fa,\ lastelse(b_k)) \\ & = & succ(b_k) & (\text{Definition } succ) \\ & = & succ(c^{x+1}.pr(i)) & (C0\text{-Semantik}) \quad \checkmark \end{array}$$

   Der Fall $va(e,c^x) \neq 0^{32}$ kann analog mit $s = fa(a_j)$ gezeigt werden.

4. $c^x.pr = \underbrace{id = f(e_1,\ldots,e_p)}_{s};c_1;c_2;\ldots$ und $ft(f).body = a_1;\ldots;a_j;return\ e$

   $$c^{x+1}.pr = a_1;\ldots;a_j;return\ e;c_1;c_2;\ldots \quad (C0\text{-Semantik Funktionsaufruf})$$

   Die Behauptung folgt für a_1 bis $return\ e$ analog zum Beweis des Induktionsanfangs. Für $c_1;c_2;\ldots$ kann die Induktionsvorraussetzung analog zu Fall 1 verwendet werden. Für $i = j+1$ gilt $i = r(c^{x+1}.rd,c)$ und wir müssen den $return$-Fall der Behauptung betrachten.

   $$\begin{array}{lcll} c^{x+1}.pr(i+1) & = & c_1 & (C0\text{-Semantik}) \\ & = & c^x.pr(2) & (\text{Voraussetzung}) \\ & = & succ(c^x.pr(1)) & (\text{Induktionsvorraussetzung}) \\ & = & succ(c^{x+1}.calls(c^{x+1}.rd)) & (C0\text{-Semantik Funktionsaufruf}) \quad \checkmark \end{array}$$

Aus der Korrektheit aller Fälle von $c^{x+1}.pr(1)$ folgern wir die Behauptung. □

Beweis der Kontrollkonsistenz

Für den Beweis von $\mathrm{c-consis}$ folgen wir größtenteils der Fallunterscheidung aus dem Beweis von 4.5. Zu zeigen ist folgendes Theorem.

Theorem 4.6 *(Kontrollkonsistenz) Sei c^i eine $C0$-Konfiguration mit $c^i.pr = s; c_1, c_2; \ldots$ und $s(i)$ eine Anzahl von DLX-Schritten, so dass für einen geeigneten Parameter aba gilt:*

$$consis(c^i, aba, d^{s(i)})$$

Sei $t(i)$ nun die Anzahl von Instruktionen, die für die Ausführung von $code(s)$ auf der DLX benötigt werden.

$$d^{s(i)} \longrightarrow^*_{code(s)} d^{s(i)+t(i)}$$

Dann lässt sich für alle i eine Schrittzahl $s(i+1)$ finden, so dass die Konsistenzbedingung $\mathrm{c-consis}$ erfüllt ist.

$$d^{s(i+1)}.pc = start(code(c^{i+1}.pr(1)))$$

Beweis: Wir führen eine Fallunterscheidung über s durch.

1. s ist eine Zuweisung, new-Anweisung, eine $while$-Schleife mit ungültiger Schleifenbedingung oder eine $if\text{-}then$-Anweisung mit ungültiger Eingangsbedingung. Aus der Korrektheit von $code(s)$ erhalten wir:

   $$d^{s(i)+t(i)}.pc = end(code(s)) + 4$$

   Wir schlussfolgern:

   $$\begin{array}{rcll} d^{s(i)+t(i)+jdis(s)}.pc & = & start(code(succ(s))) & \text{(Lemma 4.4)} \\ & = & start(code(a_1)) & \text{(Lemma 4.5)} \\ & = & start(code(c^{i+1}.pr(1))) & (C0\text{-Semantik}) \quad \checkmark \end{array}$$

   Daraus folgt, dass in diesem Fall $s(i+1) = s(i) + t(i) + jdis(s)$ gesetzt werden muss.

2. $s : if\ e\ then\ \{a_1; \ldots; a_j\}\ else\ \{b_1; \ldots; b_k\}$ und sei $va(e, c^i) = 0^{32}$. Auf der DLX wird für $code(s)$ e ausgewertet und der if-Teil mit $beqz$ übersprungen. Mit der korrekten Sprungweite laut Codeerzeugung gilt:

   $$d^{s(i)+t(i)}.pc = start(code(b_1)) = start(code(c^{i+1}.pr(1))) \quad \checkmark$$

   Hier setzen wir also $s(i+1) = s(i) + t(i)$. Für den Fall $va(e, c^i) \neq 0^{32}$ verfahren wir analog. Der Branch wird hier nicht genommen und die DLX springt ein Instruktionswort weiter zur Adresse $start(code(a_1))$, die sich laut Codeerzeugung direkt nach $beqz$ befindet.

3. $s : while\ e\ do\ \{a_1; \ldots; a_j\}$ und sei $va(e, c^i) \neq 0^{32}$. Dieser Fall lässt sich genauso wie der vorherige beweisen und wir setzen $s(i+1) = s(i) + t(i)$.

4. $s : id = f(e_1, \ldots, e_p)$ und $ft(f).body = a_1; \ldots; a_j; return\ e$. Der auf der DLX ausgeführte umfasst $code(s)$ sowie die sw-Instruktion zur Sicherung der return address zu Beginn der aufgerufenen Funktion.

   $$\begin{array}{rcll} d^{s(i)+t(i)}.pc & = & start(code(f)) & (code(s), jalr) \\ d^{s(i)+t(i)+1}.pc & = & start(code(a_1)) & (F_{c^{i+1}.rd}.ra \text{ speichern}) \\ & = & start(code(c^{i+1}.pr(1))) & (C0\text{-Semantik}) \quad \checkmark \end{array}$$

   Folglich legen wir $s(i+1) = s(i) + t(i) + 1$ fest.

5. $s : return\ e$. Der erzeugte Code für $return$ schreibt den Rückgabewert, baut den stack frame der zugehörigen Funktion ab und springt zuletzt zur gespeicherten return address $F_{c^i.rd}.ra$ zurück. Laut $\mathrm{r-consis}$ gilt:

$$F_{c^i.rd}.ra = d^{s(i)}.m_4(base(c^i.rd, c^i)_{32}) = end(code(c^i.calls(c^i.rd))) + 4$$

Sei $s' = c^i.calls(c^i.rd)$, der Funktionsaufruf, der den vorliegenden stack frame erzeugt hat. Nach der Ausführung von $return$ erhalten wir:

$d^{s(i)+t(i)}.pc$	=	$end(code(s')) + 4$	($\mathrm{r-consis}$, $code(s)$)
$d^{s(i)+t(i)+jdis(s')}.pc$	=	$start(code(succ(s')))$	(Lemma 4.4)
	=	$start(code(succ(c^i.calls(c^i.rd))))$	(Rücksubstitution)
	=	$start(code(c^{i+1}.pr(1)))$	(Lemma 4.5, $1 = r(c^i.rd, c^i)$) ✓

Hier setzen wir also $s(i+1) = s(i) + t(i) + jdis(c^i.calls(c^i.rd))$.

Somit ist die Korrektheit für alle Fälle von s gezeigt und die Gültigkeit von $\mathrm{c-consis}$ für unseren Compiler bewiesen. □

Kapitel 5

Betriebssystem-Kernel

Nun, da wir in der Lage sind, die DLX in einer Hochsprache zu programmieren, können komplexe Programme entworfen werden. Im Folgenden soll ein Betriebssystemkern konstruiert werden, der es mehreren DLX-Benutzerprogrammen erlaubt, die CPU parallel zu nutzen. Man spricht von einem CVM-System (**C**ommunicating **V**irtual **M**achines), da jedem Benutzer vorgegaukelt wird, er würde für sich allein auf einer eigenen (virtuellen) DLX rechnen. Dabei werden alle Benutzerprogramme in einem eingeschränkten Modus, dem sogenannten *user mode*, ausgeführt. Dieser erlaubt es aber auch, mit anderen Prozessen, bzw. dem Kernel über die $trap$-Funktion zu kommunizieren. Der Kern selbst arbeitet im *system mode* und hat die volle Kontrolle über den Prozessor. Er verwaltet die Benutzerprozesse und stellt spezielle Funktionen zur Verfügung, die die Benutzer aufrufen können. Außerdem werden I/O-Geräte unterstützt.
Um dies alles zu ermöglich muss zunächst die DLX für ein Betriebssystem tauglich gemacht werden. Dazu werden Interrupts und Adressübersetzung sowie die beiden verschiedenen Benutzermodi eingeführt. Danach soll CVM spezifiziert und implementiert werden. Letzteres geschieht in $C0A$, einer Erweiterung von $C0$ um *inline assembler code*. Dieser ist nötig, da in reinem $C0$ sowohl Prozessorregister als auch I/O-Geräte unsichtbar sind.

5.1 Betriebssytemunterstützung im Prozessor

Wie bereits beschrieben, müssen Maßnahmen getroffen werden, damit ein Betriebssystem auf der DLX ausgeführt werden kann. Im Wesentlichen müssen die folgenden Funktionalitäten implementiert werden.

- Interrupts inklusive system/user mode
- virtual memory

Wir beginnen mit der Einführung des Interrupt-Mechanismus. Um die DLX- von den $C0$-konfigurationen abgrenzen zu können, bezeichnen wir erstere im Folgenden mit d.

5.1.1 Interrupts

Interrupts oder *exceptions* unterbrechen die Ausführung der aktuellen Hardware-Instruktion und führen zur Ausführung eines *Interrupt handlers*, der auf den jeweiligen Interrupt reagiert und eine entsprechende Behandlungsroutine ausführt. Diese ist in Assemblersprache geschrieben, kann aber auch ein $C0$-Program, z.B. den Betriebssystemkern, starten. Es gibt interne und externe Interrupts.

- Interne Interrupts werden durch das Ausführen von Code hervorgerufen und signalisieren Fehler im Programmablauf. Sie können im Falle der $trap$-Instruktion allerdings auch gewünscht und von einem Benutzerprogramm hervorgerufen worden sein.
- Externe Interrupts kommen von außerhalb des Prozessors, zum Beispiel von I/O-Geräten oder von einem reset. Sie stellen sozusagen externe Eingaben für den Prozessor dar.

Im Folgenden soll nun zunächst der Effekt von Interrupts auf die DLX spezifiziert und dann implementiert werden.

Spezifikation

Auftretende Interrupts werden durch *interrupt event signals* dargestellt. Es gibt

- $iev[j]$ - **i**nternal **ev**ent signal mit $j \in [1:6]$
- $eev[j]$ - **e**xternal **ev**ent signal mit $j \in \{0, 7, 8, \ldots, 31\}$

Um den externen Interrupts Rechnung zu tragen, wird die Übergangsfunktion der DLX entsprechend erweitert.

$$\delta_D(d, eev) = d'$$

Tabelle 5.1 zeigt eine Auflistung der möglichen Interrupts und ihrer Attribute.

j	mnemonic	name	resume type	maskierbar	extern
0	$reset$	reset	abort	nein	ja
1	ill	illegal	abort	nein	nein
2	mal	misalignment	abort	nein	nein
3	pff	page fault on fetch	repeat	nein	nein
4	$pfls$	page fault on load/store	repeat	nein	nein
5	$trap$	trap	continue	nein	nein
6	ovf	overflow	continue	ja	nein
7	I/O	I/O-Geräte	...	ja	ja
⋮	⋮	⋮	⋮	⋮	⋮

Tabelle 5.1: Interrupts der DLX

Die Interrupts bedeuten im Einzelnen:

- $reset$ - Die CPU wird neu gestartet. Dieser Interrupt entspricht dem $reset$-Signal der DLX ohne Interruptunterstützung.
- ill - Eine illegale Instruktion sollte ausgeführt werden. Entweder konnte keine gültige Instruktion dekodiert werden oder ein Benutzer wollte eine system-mode-Instruktion ausführen.
- mal - Ein Speicherzugriff verletzte die alignment-Bedingung.
- $pff/pfls$ - page faults treten nur im user mode (virtuelle Adressierung) auf. Sie signalisieren dass eine Seite nicht in der page table vorhanden war, bzw. dass ein Zugriff außerhalb der page table stattfinden sollte.
- $trap$ - Aufruf einer Kernelfunktion
- ovf - Eine ALU-Berechnung hatte einen overflow zur Folge. Der Benutzer kann diesen Interrupt maskieren (deaktivieren).
- I/O - zur Kommunikation mit externen I/O-Geräten ($j \geq 7$)

Jedem Interrupt wird ein *resume type* zugeordnet. Dieser bestimmt, ob die unterbrochene Instruktion nach Abschluss des Interrupt handlers wiederholt oder verworfen werden soll, oder ob die Programmausführung komplett unterbrochen werden soll. Einige Interrupts sind auch maskierbar, das heißt sie können je nach Einstellung von der DLX ignoriert werden, falls sie auftreten sollten. Zur Abwicklung der Interrupts wird eine neue Komponente zur Verfügung gestellt. Es handelt sich um das **s**pecial **p**urpose **r**egister file $d.spr : \{0,1\}^5 \longrightarrow \{0,1\}^{32}$.

$$d = (d.pc, d.gpr, d.m, d.spr)$$

$\langle i \rangle$	$spr(i)$	name
0	SR	**S**tatus **R**egister
1	ESR	**e**xception **SR**
2	ECA	**e**xception **Ca**use
3	EPC	**e**xception **PC**
4	$EDATA$	**e**xception **data**
7	$MODE$	system/user **mode**

Tabelle 5.2: special purpose Register zur Interrupt-Behandlung

Darin dienen die ersten fünf Register der Interruptbehandlung. Sie sind in Tabelle 5.2 aufgeführt und haben die folgenden Funktionen:

- SR - Speichert eine Bitmaske, die aussagt, welche Interrupts - sofern möglich - maskiert werden sollen. $SR[i]$ steht dabei für den Interrupt i. Eine 0 maskiert den Interrupt, eine 1 macht ihn für die DLX sichtbar.
- ESR - Falls der Interrupt handler gestartet wird, werden alle maskierbaren Interrupts maskiert, damit die Interrupt Service Routine ungestört arbeiten kann. Dafür wird $SR = 0^{32}$. Nun können nur noch $reset$ oder Programmierfehler im Interrupt handler die ordnungsgemäße Behandlung des Interrupts durchkreuzen. Danach soll das ursprüngliche Statusregister wieder hergestellt werden. Dazu wird es beim Aufruf der Interrupt Service Routine (ISR) in ESR gesichert.
- ECA - Speichert die maskierten event signals um dem Interrupt handler den Typ der aufgetretenen Interrupts mitzuteilen.
- EPC - Speichert die Adresse der Instruktion, zu der nach Abschluss der Interrupt Service Routine zurückgekehrt werden soll.
- $EDATA$ - Speichert Parameter für den Interrupt handler. Im Falle einer $trap$ Instruktion steht hier die immediate-Konstante. Tritt ein page fault on load/store auf, enthält $EDATA$ die effektive Adresse des Speicherzugriffs.
- $MODE$ - Signalisiert in welchem Modus sich die CPU befindet (system/user mode). Dabei ist nur das niedrigste Bit $mode(d) = d.spr(7_5)[0]$ von Bedeutung.

$$mode(d) = \begin{cases} 0 & : \text{ system mode} \\ 1 & : \text{ user mode} \end{cases}$$

Das Maskieren aller maskierbaren Interrupts während der Ausführung der ISR hat zur Folge, dass eventuell auftretende externe Interrupts von I/O-Geräten in dieser Zeit nicht wahrgenommen werden. Daher ist eine Konvention notwendig, dass alle Geräte ihre Anfragen an den CPU solange aktiviert lassen müssen, bis sie eine Bestätigung des Interrupts von diesem erhalten.

Das Erkennen von Interrupts soll nun formalisiert werden. Wir definieren die neue Konfiguration $\delta_D(d, eev) = d'$ unter Berücksichtigung der Interrupt event signals.
Tritt ein Interrupt auf, so muss mindestens ein event signal aktiviert worden sein. Wir definieren den Grund (**ca**use) eines Interrupts als:

$$ca(d, eev)[j] = \begin{cases} iev(d)[j] & : \quad j \in [1:6] \quad (\text{intern}) \\ eev[j] & : \quad \text{sonst} \quad (\text{intern}) \end{cases}$$

Man beachte, dass die internen event signals von der aktuellen Konfiguration c abhängen also daraus berechenbar sind. Nur die externen event signals sind zusätzliche inputs. Da gewisse Interrupts maskiert

werden können müssen wir einen **m**asked **ca**use einführen.

$$mca(d,eev)[j] = \begin{cases} ca(d,eev)[j] & : \quad j \leq 5 \quad (\text{nicht maskierbar}) \\ ca(d,eev)[j] \wedge d.SR[j] & : \quad \text{sonst} \end{cases}$$

Dann wird die ISR aufgerufen, sobald eines der Bits im masked cause 1 ist, also ein event signal aktiviert worden ist, das nicht maskiert wurde. Das Signal was die ISR aufruft heißt $JISR(d,eev)$ (**J**ump to **I**nterrupt **S**ervice **R**outine).

$$JISR(d,eev) = \bigvee_j mca(d,eev)[j]$$

Da mehrere Interrupts gleichzeitig auftreten können, definieren wir den **i**nterrupt **l**evel $il(d,eev)$, der den aufgetretenen Interrupt mit der höchsten Priorität (niedrigste Ordnungsnummer) zurückgibt.

$$\exists j \in [0:31].\ mca(d,eev)[j] = 1 \quad \Rightarrow \quad il(d,eev) = \min\{j \mid mca(d,eev)[j] = 1\}$$

Nun sollen die Auswirkungen eines Interrupts betrachtet werden. Offensichtlich gilt, dass die DLX wie gewohnt rechnet, wenn kein Interrupt auftritt.

$$JISR(d,eev) = 0 \quad \Rightarrow \quad \delta_D(d,eev) = \delta_{D_{\text{alt}}}(d,reset)$$

Ist $JISR(d,eev) = 1$, so treten die folgenden Effekte ein.

$$\begin{aligned} d'.SR &= 0^{32} \\ d'.EPC &= \begin{cases} d.pc & : \quad il(d,eev) \in \{3,4\} \quad (\text{repeat}) \\ \delta_{D_{alt}}(d,reset).pc & : \quad \text{sonst} \end{cases} \\ d'.ECA &= mca(d,eev) \\ d'.ESR &= \begin{cases} d.gpr(RS1(d)) & : \quad il(d,eev) \geq 5 \wedge movi2s(d) \wedge SA(d) = 0^5 \\ d.SR & : \quad \text{sonst} \end{cases} \\ d'.EDATA &= \begin{cases} sxtimm(d) & : \quad il(d,eev) = 5 \quad (trap) \\ ea(d) & : \quad il(d,eev) = 4 \quad (pfls) \end{cases} \\ d'.PC &= SISR \end{aligned}$$

Die Definition von EPC implementiert den resume type des Interrupts. Nur bei repeat wird die unterbrochene Instruktion wiederholt. Die erste Zeile in der Fallunterscheidung für ESR spiegelt die Situation wider, dass ein Interrupt mit resume type continue auftritt, während der Benutzer versucht, mit einer $movi2s$-Instruktion das Statusregister zu manipulieren ($SA(d) = 0^5$). Nach der Behandlung des Interrupts soll der Anschein erweckt werden, der Programmablauf sei nie unterbrochen wurden. Deshalb muss das SR auf den Wert zurückgesetzt werden, den es ohne eine Unterbrechung nach Ausführung der $movi2s$-Instruktion gehabt hätte. Darum wird der entsprechende Wert aus dem gpr in ESR gesichert. Dies bedeutet insbesodere, dass der neue Wert für SR noch nicht für die Maskierung von während der Ausführung von $movi2s$ auftretenden Interrupts benutzt wird. Die Adresse $SISR$ (**S**tart of **I**nterrupt **S**ervice **R**outine) markiert die Startadresse des Interrupt handlers. Es gilt $SISR = 0^{32}$, was bedeutet, dass das bisherige $reset$-Signal nur ein Sonderfall der ISR gewesen ist.

Wie schon angedeutet, bringt die Erweiterung der Konfiguration auch neue Instruktionen mit sich. zum einen hat man zwei neue R-type-Instruktionen, die dazu dienen Daten zwischen dem gpr und dem spr auszutauschen. Man spricht daher von den *move*-Intruktionen.

R-type	$I(d)[5:0]$	mnemonic	name	effect
	010000	$movs2i$	**mov**e **s**pecial **to** **i**nteger	$d'.gpr(RD(d)) = d.spr(SA(d))$
	010001	$movi2s$	**mov**e **i**nteger **to** **s**pecial	$d'.spr(SA(d)) = d.gpr(RS1(d))$

Außerdem kommen noch zwei neue J-type-Instruktionen dazu.

J-type	$I(d)[5:0]$	mnemonic	name	effect
	111110	$trap$	trap	$iev(d)[5] = 1$
	111111	rfe	**r**eturn **f**rom **e**xception	$d'.SR = d.ESR, d'.PC = d.EPC,$ $d'.mode = 1$

$trap$ löst einen Interrupt aus, der den Interrupt handler eine angeforderte Kernelfunktion ausführen lässt. So können Benutzerprogramme mit dem Betriebssystem und anderen Benutzern über solcherlei Funktionsaufrufe kommunizieren. rfe wird zum Ende jedes Interrupts ausgeführt. Es stellt die ordnungsgemäße Konfiguration je nach resume type wieder her.

Es ist zu beachten, dass rfe nur im system mode ausgeführt werden darf. Dazu dürfen Benutzer mit $movi2s$ und $movs2i$ auch nur auf das Statusregister zugreifen und lediglich das Maskenbit für ovf modifizieren. Verstöße gegen diese Regeln führen zu einer ill-exception.

Implementierung

Auch die Hardware der DLX muss für die Interruptbehandlung erweitert werden. Die offensichtlichste Veränderung ist das zusätzliche special purpose register file SPR. Dieses ist eine Registerbank mit den üblichen Ein- und Ausgängen ad, w, D_{in} und D_{out}. Diese stellen das Interface für die move-Instruktionen dar. Bei Interrupts müssen mehrere Register gleichzeitig geschrieben und gelesen werden, daher stellt das SPR zusätzlich private Ein- und Ausgangssignale D_{in_j}, D_{out_j} und $w_i[j]$ für jedes Register j zur Verfügung. Abbildung 5.1 zeigt die schematische Darstellung des SPR.

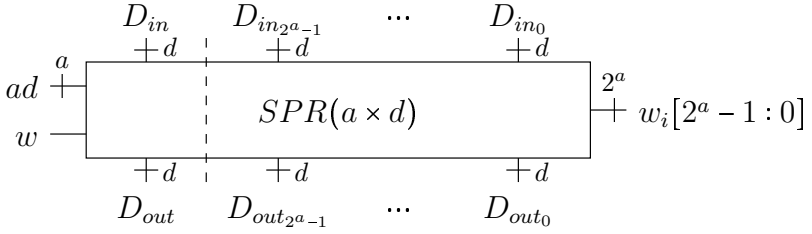


Abbildung 5.1: special purpose register file SPR

Die individuellen Schreibzugriffe über die w_i haben Vorrang gegenüber den globalen über w und ad. Die Implementierung ähnelt einem gewöhnlichem RAM und ist in Abbildung 5.2 dargestellt.
Nun muss das SPR noch mit dem Rest der DLX verdrahtet werden. Wie dies geschieht zeigt Abbildung 5.3. Das select-Signal s_{1_h} ist die Hardware-Version von:

$$s_1(d) = movi2s(d) \wedge SA(d) = 0^5 \wedge il(d, eev) \geq 5$$

Im Falle einer mit resume type continue unterbrochenen move-Instruktion die das Statusregister aktualisieren sollte, wird der Spezifikation entsprechend demnach $A = h.gpr(RS1_h)$ statt dem alten SR in ESR gesichert. s_{2_h} ist die Hardware-Variante von

$$s_2(d) = (il(d, eev) = 5)$$

und signalisiert lediglich, dass ein $trap$-Interrupt ausgelöst wurde, weshalb $sxtimm_h$ statt ea_h in $EDATA$ zu speichern ist. s_{3_h} implementiert das select-Signal

$$\begin{aligned} s_3(d) &= (il(d, eev) \in \{3, 4\}) \\ &= (mca(d, eev)[3] \vee mca(d, eev)[4]) \wedge (\overline{\bigvee_{j \leq 2} mca(d, eev)[j]}), \end{aligned}$$

welches anzeigt, dass ein Interrupt mit resume type repeat zu behandeln ist. Dementsprechend wird der alte PC in EPC gespeichert und nicht $PCinc$. Die Belegung der Schreib- und Adresssignale ist als selbständige Übung durchzuführen. In Abbildung 5.4 sehen wir die Datenpfade zwischen GPR und SPR. Mit den move-Instruktionen können Daten zwischen den Registern ausgetauscht werden.

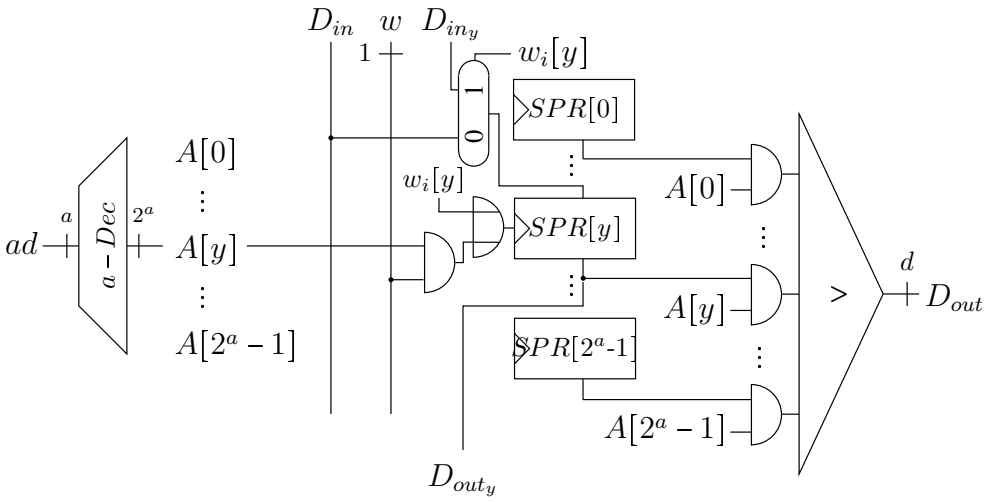


Abbildung 5.2: Implementierung des SPR

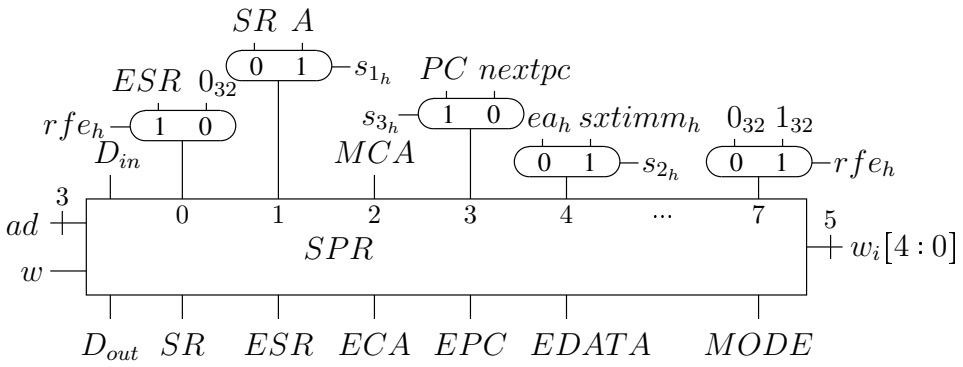


Abbildung 5.3: Verdrahtung des SPR

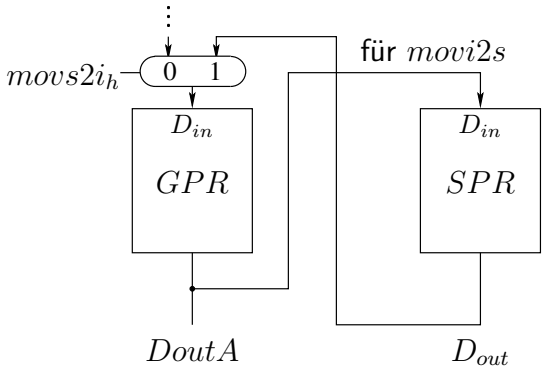


Abbildung 5.4: Datenpfade zwischen GPR und SPR

Dadurch kann der Programmierer Inhalt der SPR-Register abfragen und gegebenenfalls ändern.
Bei einem auftretenden Interrupt könnten die Speicherkomponenten mit potentiell fehlerhaften Daten aktualisiert werden, daher müssen dann die entsprechenden Schreibsignale deaktiviert werden.

$$\begin{aligned} mw(d) &= mw_{alt}(d) \wedge (\overline{JISR \wedge (il(d, eev) \in \{3,4\})}) \\ gpr_w(d) &= (gpr_{w_{alt}}(d) \vee movs2i(d)) \wedge \wedge (\overline{JISR \wedge (il(d, eev) \in \{3,4\})}) \end{aligned}$$

Das GPR wird nun auch im Falle von $movs2i(d)$ geschrieben. Zuletzt muss noch die Logik zur Berechnung des neuen program counter aktualisiert werden. Die neuen Datenpfade sind in Abbildung 5.5 dargestellt.

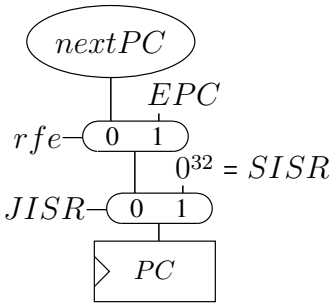


Abbildung 5.5: Erweiterung der $nextPC$-Umgebung

5.1.2 Adressübersetzung

Den verschiedenen Benutzerprogrammen auf der DLX soll vorgespielt werden, sie wären allein auf der DLX und hätten ungeteilten Zugriff auf deren Resourcen. In Wirklichkeit rechnet jeder Benutzerprozess u auf einer virtuellen Maschine $vm(u)$ und greift mit virtuellen Adressen auf einen virtuellen Speicher zu. Lediglich über $trap$-Instruktionen kann ein Benutzer mit anderen Benutzern und I/O-Geräten kommunizieren. Der Betriebssystemkern muss die virtuellen Adressen in physikalische Adressen übersetzen und die Speicherzugriffe zu dem Speicherbereich weiterleiten, der dem Benutzer zugewiesen wurde. Um die verschiedenen Bereiche zu verwalten, wird der Speicher in Seiten (pages) von je $4K = 2^{12}$ Bytes unterteilt. Jedem Benutzer werden Seiten zugewiesen und in einer page table verzeichnet. Die virtuellen Adressen verweisen dann auf Einträge in der page table mit deren Hilfe sich die physikalische Adresse des adressierten Bytes berechnen lässt.

Spezifikation

Die virtuellen Maschinen rechnen auf einer eingeschränkten DLX und benutzen diese Adressen, als ob es physische Adressen wären. Die reale physische Maschine muss die Adressen übersetzen. Dazu passen wir die Komponenten der Konfiguration wie in Abbildung 5.6 gezeigt an. Es gibt nun zum einen den physikalischen Speicher pm, der den gewohnten RAM darstellt, und neu dazu den swap memory sm. Dieser Auslagerungsspeicher kann zum Beispiel auf einer Festplatte liegen. In den swap memory können aktuell nicht benötigte Seiten ausgelagert werden, was den verfügbaren Speicher der DLX vergrößert. Es existieren Treiber, die Daten zwischen pm und sm hin- und herkopieren können. Die Spezifikation selbiger ist nicht schwierig, wenn man einfach die Zustände „vorher“ und „nachher“ definiert. Korrektheitsbeweise über Treiber sind hingegen kompliziert und noch kaum erforscht. Das Problem bei der Treiberkorrektheit ist, zu zeigen, dass außer den spezifizierten keine weiteren Veränderungen eintreten. Dies erweist sich als schwierig, da weitere Prozesse und Geräte im Hintergrund über *side channels* auf sm zugreifen und die Treiber durch Interrupts unterbrochen werden können. Wir gehen im folgenden davon aus, dass die Treiber korrekt arbeiten und blenden diese Problematik aus.
Außer dem sm werden zwei neue special purpose register eingeführt:

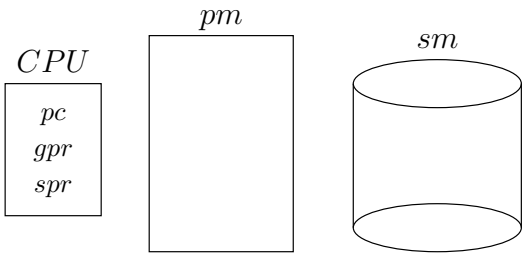


Abbildung 5.6: Die physikalische Maschine

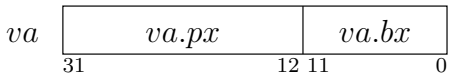
i	$spr(i)$	name
5	PTO	**p**age **t**able **o**rigin
6	PTL	**p**age **t**able **l**ength

Dies sind im Einzelnen:

- $d.pto = d.spr[5]$ - Die Startadresse der page table des aktuellen Benutzers
- $d.ptl = d.spr[6]$ - Die Länge der page table des aktuellen Benutzers

Nun soll die Übersetzung von virtuellen Adressen mit Hilfe der page table definiert werden. Eine virtuelle Adresse $va \in \{0,1\}^{32}$ unterteilt sich in:

- $va.px = va[31:12]$ - page index
- $va.bx = va[11:0]$ - byte index



Der page index verweist auf einen Eintrag $pte(d, va)$ (**p**age **t**able **e**ntry) in der page table. Die genaue Adresse des Eintrages $ptea(d, va)$ (**p**age **t**able **e**ntry **a**ddress) berechnet sich zu:

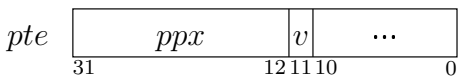
$$ptea(d, va) = d.pto +_{32} 0^{10} va.px 00$$

Das bedeutet, dass der page index als relative Word-Adresse für die page table entries dient. Der jeweilige Eintrag findet sich dann im physischen Speicher

$$pte(d, va) = d.pm_4(ptea(d, va))$$

Der page table entry besteht aus:

- $ppx(d, va) = pte(d, va)[31:12]$ - **p**hysical **p**age **i**ndex
- $v(d, va) = pte(d, va)[11]$ - **v**alid bit
- $pte(d, va)[10:0]$ - noch ungenutzt



Das valid bit sagt aus, ob die zugehörige Seite im pm vorhanden ist. Ist dies nicht der Fall und auf die Seite wird zugegriffen, so wird ein page fault ausgelöst und die Seite muss aus dem swap memory nachgeladen werden. Ist $v(d, va) = 1$, so beginnt die gewünschte Seite an der physikalischen Adresse, die durch den physical page index spezifiziert wird. Die **p**hysical **m**emory **a**ddress $pma(d, va)$, die

sich aus physical page index und byte index zusammen setzt, repräsentiert dann die für va gesuchte physikalische Byte-Adresse innerhalb der Seite.

$$pma(d, va) = ppx(d, va) \circ va.bx$$

Dieser Speicherzugriff kann jedoch auch scheitern, falls der page index die Länge der page table überschreitet, also auf eine nicht zuordbare Seite zugegriffen werden soll. Man spricht von einer **p**age **t**able **l**ength **e**xception on **f**etch bzw. on **l**oad/**s**tore.

$$\begin{aligned} ptlef(d) &= (\langle d.pd.px \rangle \geq \langle d.ptl \rangle) \\ ptlels(d) &= (\langle ea(d).px \rangle \geq \langle d.ptl \rangle) \end{aligned}$$

Die Interrupt event signals für page faults werden dann wie folgt definiert.

$$\begin{aligned} iev[3] &= pff(d) \\ &= mode(d) \wedge (ptlef(d) \vee \bar{v}(d, d.pc)) \\ iev[4] &= pfls(d) \\ &= mode(d) \wedge (lw(d) \vee sw(d)) \wedge (ptlels(d) \vee \bar{v}(d, ea(d))) \end{aligned}$$

Im system mode können keine page faults auftreten, da der Kern keine Adressübersetzung nutzt. Die veränderte fetch-Semantik ergibt sich zu:

$$I(d) = \begin{cases} d.pm_4(d.pc) & : \quad mode(d) = 0 \quad \text{(system mode)} \\ d.pm_4(pma(d, d.pc)) & : \quad mode(d) = 1 \quad \text{(user mode)} \end{cases}$$

Die neue Semantik von load/store lautet wie folgt.

$$lw(d): \quad d'.gpr(RS1(d)) = \begin{cases} d.pm_4(ea(d)) & : \quad mode(d) = 0 \\ d.pm_4(pma(d, ea(d))) & : \quad mode(d) = 1 \end{cases}$$

$$sw(d): \quad \left.\begin{array}{ll} d'.pm_4(ea(d)) & : \quad mode(d) = 0 \\ d'.pm_4(pma(d, ea(d))) & : \quad mode(d) = 1 \end{array}\right\} = d.gpr(RD(d))$$

Abbildung 5.7 veranschaulicht die Adressübersetzung.

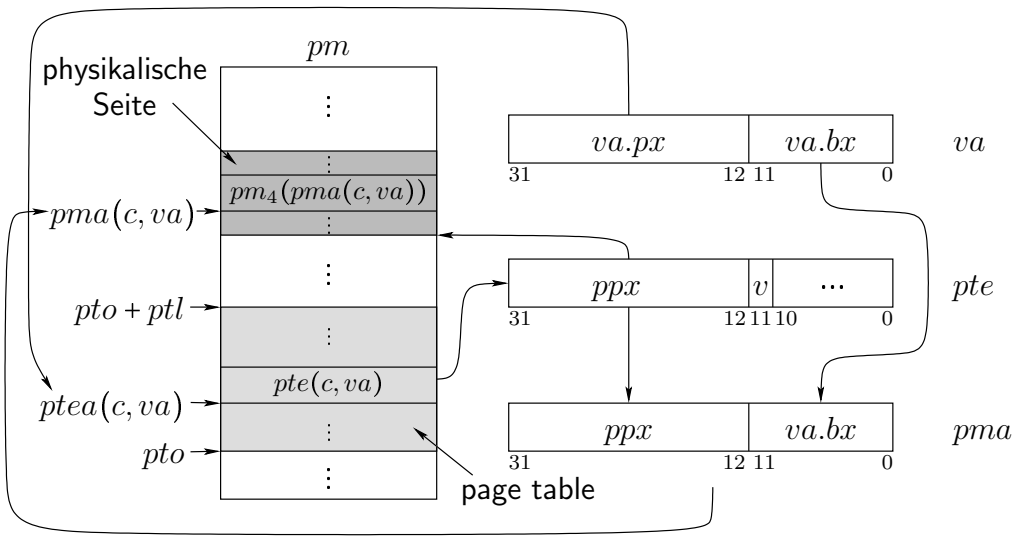
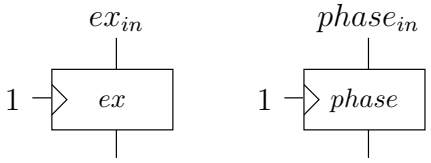


Abbildung 5.7: Adressübersetzung

Implementierung

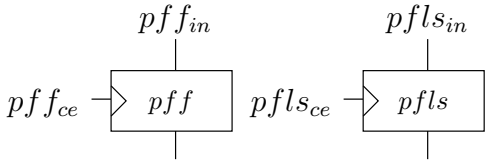
Zur Implementierung der Adressübersetzung muss die Hardware um eine MMU (**m**emory **m**anagement **u**nit) erweitert werden. Da die Übersetzung in zwei Schritten erfolgt, muss auch der Kontrolltakt $c.ex$ angepasst werden. Wir führen einen weiteren Zähler $phase$ ein, der stetig zwischen 0 und 1 wechselt, wie ex zuvor.



ex zählt im system mode unverändert, im user mode jedoch mit halber geschwindigkeit, da jeder Speicherzugriff nun doppelt so lang dauert. In der ersten Phase wird stets $pte(c, va)$ aus dem Speicher geladen und $pma(c, va)$ berechnet. In der zweiten phase kann dann auf $pm_4(pma(c, va))$ zugegriffen werden, sofern kein page fault eingetreten ist. Die Eingänge der Register ex und $phase$ werden entsprechend beschaltet. Hier müssen für $phase$ auch die Übergänge zwischen system und user mode durch $JISR$ und rfe berücksichtigt werden. Sei $mode_h = h.spr(0_5)[0]$:

$$
\begin{aligned}
ex_{in} &= \overline{reset} \wedge (\overline{mode_h} \wedge \overline{ex} \vee mode_h \wedge (\underbrace{\overline{ex} \wedge phase \vee ex \wedge \overline{phase}}_{ex \oplus phase})) \\
phase_{in} &= \overline{reset} \wedge (\overline{mode_h} \wedge \overline{ex \wedge rfe_h} \vee mode_h \wedge (\overline{phase} \vee ex \wedge JISR_h))
\end{aligned}
$$

Auftretende page faults werden in speziellen Registern gespeichert.



Die Eingangs- und clock enable- Signale werden mit $v = pm_{Dout}[11]$ folgendermaßen definiert

$$
\begin{aligned}
pff_{in} &= mode_h \wedge (ptlef_h \vee \bar{v}) \\
pfls_{in} &= mode_h \wedge (ptlels_h \vee \bar{v}) \\
pff_{ce} &= \overline{ex} \wedge \overline{phase} \\
pfls_{ce} &= (lw_h \vee sw_h) \wedge ex \wedge \overline{phase}
\end{aligned}
$$

Die Datenpfade der MMU sind in Abbildung 5.8 dargestellt.

5.2 CVM-Semantik

CVM bedeutet **c**ommunicating **v**irtual **m**achines und ist ein paralleles Berechnungsmodell zur Beschreibung

- des Betriebssystemkerns
- und der Benutzerprozesse.

Die Funktion des Kerns besteht darin, eine Anzahl von virtuellen Benutzerprozessen auf einer physikalischen Maschine zu simulieren. Der Kern lässt sich dabei in einen maschinenspezifischen Teil und einen Teil, der Handler für Interrupts und Service Calls enthält, aufteilen. Die Spezifikation des maschinenunabhängigen Teils ist eine Art „Formalisierung des Benutzerhandbuchs des Kerns“ und kommt ohne Assembler aus, nur bei der Implementierung muss inline assembler (maschinenabhängig) verwendet werden. Wir sprechen vom abstrakten Kern k, der ein $C0$ Programm mit *speziellen Funktionen* ist.

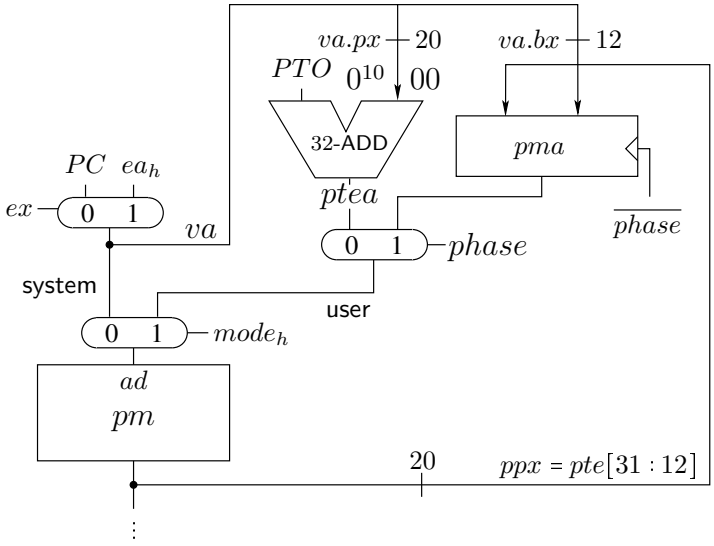


Abbildung 5.8: Datenpfade der memory management unit

Zum abstrakten Kern hinzu kommen:

- der i-te Benutzerprozess $vm(i)$ (DLX-Maschinen in user mode)
- die i-te I/O-Device $D(i)$ (wird hier nicht weiter betrachtet)

Formal enthält CVM-Konfiguration cvm mit p Benutzerprozessen:

- $cvm.c$ - $C0$-Konfiguration des abstrakten Kerns
- $cvm.vm(i)$ - virtuelle DLX-Konfiguration des Benutzers $i \in [1 : p]$ mit virtuellem Speicher $cvm.vm(i).m : A(cvm, i) \longrightarrow \{0,1\}^8$ des Benutzers i.

$$A(cvm, i) = \{a \mid a \in \{0,1\}^{32}, \langle a \rangle \leq 4K \cdot \langle cvm.ptl(i) \rangle\}$$

- $cvm.ptl(i) \in \mathbb{N}$ - Anzahl der pages für Benutzer i
- $cvm.cp \in \{0, \ldots, cvm.p\}$ - **c**urrent **p**rocess, aktuell laufender Prozess

Für $cvm.cp$ gilt:

$$cvm.cp = \begin{cases} 0 & : \text{ Kern läuft im system mode} \\ i & : i > 0,\ vm(i) \text{ läuft im user mode} \end{cases}$$

Im Folgenden soll die Arbeitsweise des Kerns beschrieben werden. Dazu definieren wir

$$\delta_{cvm}(cvm, eev) = cvm'$$

Hierbei gibt es zwei triviale Fälle:

- $cvm.cp = u \wedge u > 0 \wedge JISR(c.vm(u), eev) = 0$, das heißt Benutzer u läuft und wird nicht von einem Interrupt unterbrochen, dann gilt:

$$cvm'.vm(u) = \delta_D(cvm.vm(u), eev)$$

Die virtuelle Maschine des Benutzers u ändert sich also gemäß der DLX-Übergangsfunktion. Alle anderen Komponenten bleiben unverändert.

$$\begin{aligned} cvm'.cp &= cvm.cp \\ cvm'.c &= cvm.c \\ \forall i \neq u, i \in \{1,\ldots,p\} : cvm'.vm(i) &= cvm.vm(i) \\ \forall i \neq u, i \in \{1,\ldots,p\} : cvm'.ptl(i) &= cvm.ptl(i) \end{aligned}$$

Es ist zu beachten, dass unter diesen Fall auch page faults des Benutzerprogramms fallen. Da im cvm-Modell lediglich der virtuelle Speicher der Benutzer modelliert wird, ist vollkommen unsichtbar, welche Seiten des virtuellen Speichers tatsächlich im physikalsichen Speicher vorhanden sind. Der konkrete Kern muss die Illusion des virtuellen Speichers entsprechend implementieren. Lediglich page table length exceptions, also Zugriffe außerhalb des virtuellen Speichers eines Benutzers sind auf Ebene des abstrakten Kerns und der Benutzermaschinen sichtbar. Das Betriebssystem könnte dann als möglichen Effekt zum Beispiel die Ausführung des Benutzerprogramms unterbrechen. Es wäre auch denkbar, dass in einem gewissen Rahmen zusätzlicher Speicher alloziiert wird, also $cvm.ptl(u)$ erhöht wird.

- $cvm.cp = 0$, das heißt, der Kern läuft im system mode. Ist $cvm.c.pr = an; r$ und an keine *spezielle Funktion* des Kerns, dann wird einfach an gemäß der $C0$-Semantik ausgeführt. Die $C0$-Maschine macht einen Schritt.

  $$cvm'.c = \delta_C(cvm.c)$$

  Die anderen Komponenten ändern sich nicht.

  $$\begin{aligned} cvm'.cp &= 0 \\ \forall i \in \{1,\ldots,p\} : cvm'.vm(i) &= cvm.vm(i) \\ \forall i \in \{1,\ldots,p\} : cvm'.ptl(i) &= cvm.ptl(i) \end{aligned}$$

In den verbleibenden Fällen interagieren Kern und Benutzer miteinander. Dabei sind die folgenden Interaktionen möglich.

- Kern $\longrightarrow$ Benutzer (spezielle Funktionen)
- Benutzer $\longrightarrow$ Kern ($JISR$, insbesondere $trap$)

5.2.1 Spezielle Funktionen des abstrakten Kerns

Der Betriebssystemkernel k verfügt über spezielle Funktionen, die die Benutzerverwaltung ermöglichen. Diese werden auch CVM-Primitive genannt und werden zum Implementieren die sogenannten *system calls* verwendet. Die essentiellen Vertreter sollen hier aufgeführt werden. Zuvor definieren wir noch diese Kurzschreibweise für den Wert von $C0$-Variablen und Ausdrücke X des Kerns k.

$$\widetilde{X} \;\widehat{=}\; \langle va(X, cvm.c)\rangle = \langle vv(lv(X, cvm.c), cvm.c)\rangle$$

Es soll gelten, dass $cvm.cp = 0$ und $cvm.c.pr = an; r$, an ist eine spezielle Funktion. Dann führen wir eine Fallunterscheidung über an durch:

- $an:$ $start(CP)$ - Startet einen neuen Benutzerprozess. $start$ hat folgenden Effekt:

  $$\begin{aligned} cvm'.c.pr &= r \\ cvm'cp &= \widetilde{CP} \in [1:p] \end{aligned}$$

  In k gibt es eine Funktion, die bestimmt, welcher Prozess als nächstes gestartet werden soll, und die die neuen Werte für CP berechnet. Man bezeichnet diese Funktion als *scheduler*.

- $an:\quad copy(u_1, s_1, u_2, s_2, l)$ - Kopiert einen Bereich der Länge $\widetilde{l}$ aus dem Speicher von Benutzer $\widetilde{u_1}$ in den Speicher von Benutzer $\widetilde{u_2}$. Dabei ist.
  - u_1 - der Sender
  - u_2 - der Empfänger
  - s_1 - die Startadresse des Speicherbereichs beim Sender
  - s_2 - die Startadresse des Speicherbereichs beim Empfänger
  - l - die Länge des zu sendenden Bereichs

  Durch diesen Befehl ist eine Kommunikation der Prozesse untereinander über den Speicher möglich. Wenn die jeweiligen Speicherbereiche vorhanden sind,

  $$\begin{aligned} \widetilde{s_1} + \widetilde{l} &< 4K \cdot cvm.ptl(\widetilde{u_1}) \\ \widetilde{s_2} + \widetilde{l} &< 4K \cdot cvm.ptl(\widetilde{u_2}) \end{aligned}$$

  dann bewirkt $copy$:

  $$cvm'.vm(\widetilde{u_2}).m_{\widetilde{l}}((\widetilde{s_2})_{32}) = cvm.vm(\widetilde{u_1}).m_{\widetilde{l}}((\widetilde{s_1})_{32})$$

  Der Rest der Konfiguration bleibt unverändert. Man beachte wie hier der Formalismus des DLX-Speichers und der $C0$-Ausdrücke vermischt wird. Zum einen argumentiert man über die Speicher von DLX-Maschinen, andererseits sind die Parameter dazu die Werte von $C0$-Ausdrücken und -Variablen.

  Der konkrete Kern muss zur Implementierung des $copy$-Befehls auf den Benutzerspeicher zugreifen. Da die Adressparameter virtuell sind, wird eine Adressübersetzung benötigt. Dies kann zum Einen in Software simuliert werden. Alternativ dazu kann sich der konkrete Kern eine spezielle page table für den Speicherzugriff anlegen und die Hardwareübersetzung nutzen. Der Kern liefe dann für kurze Zeit im user mode mit maskierten externen Interrupts.

- $an:\quad alloc(u, x)$ - Weist Benutzer $\widetilde{u}$ eine Anzahl von $\widetilde{x}$ neuen Seiten zu, die mit 0 initialisiert werden müssen. Ansonsten wäre es Prozessen eventl. möglich auf die Daten anderer Prozesse unberechtigt zuzugreifen.

  $$\begin{aligned} cvm'.ptl(\widetilde{u}) &= cvm.ptl(\widetilde{u}) + \widetilde{x} \\ \forall a \in \{0,1\}^{32} : \langle a \rangle \geq 4K \cdot cvm.ptl(\widetilde{u}) &\Rightarrow cvm'.vm(\widetilde{u}).m(a) = 0^8 \end{aligned}$$

  Mit diesem Befehl kann der zugewiesene Speicher eines Benutzerprogramms vergrößert werden, wenn z.B. der stack mit dem heap kollidiert ist, und das entsprechende Programm den Abbruchcode ausgeführt hat.

- $an:\quad free(u, x)$ - Gibt bei Benutzer $\widetilde{u}$ die obersten $\widetilde{x}$ pages frei. Der Effekt ist, dass die page table length verringert wird.

  $$cvm'.ptl(\widetilde{u}) = cvm.ptl(\widetilde{u}) - \widetilde{x}$$

  Natürlich dürfen nicht mehr Seiten freigegeben werden, als dem Prozess zugeordnet sind.

5.2.2 Semantik von trap-Instruktionen

Nun sol der Fall betrachtet werden, dass ein Benutzerprozess läuft ($cvm.cp = u > 0$) und dieser eine $trap$-Instruktion ausführt. Sei $j = imm(cvm.vm(u))$ die immediate-Kontante der $trap$-Instruktion. Diese kodiert eine Kernel-Funktion, die ausgeführt werden soll. Es existiert eine Abbildung

$$kcd : \{0,1\}^{26} \longrightarrow FN,$$

die die verschiedenen kernel calls codiert, die das Betriebssystem dem Benutzer zur Verfügung stellt (API - Application Programming Interface). Dann ist $f = kcd(j)$ der Name der Funktion die mit der $trap$-Instruktion aufgerufen werden sollte. Als die $np = ft(f).np$ Parameter dieser Funktion werden die Registerinhalte $vm(u).gpr(1) \cdots vm(u).gpr(np)$ als integer übergeben. $trap$ ist also nichts als ein Funktionsaufruf innerhalb der $C0$-Semantik. Das Ergebnis des Funktionsaufrufs wird in der globalen Kern-Variable res gespeichert. Als letzter Befehl der Funktion wird statt $return\ x$ eine spezielle Funktion $return\_from\_trap()$ des Kerns gestartet werden, die die Ergebnisse der Kernelfunktion an $vm(u)$ zurückgibt und den Fuktionsaufruf beendet. Die genaue Semantik von *return from trap* kann selbständig erarbeitet werden. Formal hat $trap$ mit $cvm.cp = u$ die folgenden Auswirkungen:

- $cvm'.cp = 0$ - Die Maschine wechselt in den system mode und der Kern läuft.
- $cvm'.c.rd = cvm.c.rd + 1$ - Die Rekursionstiefe wird erhöht.
- $top(cvm'.c) = cvm'.c.lms(c.rd)$ und $top(cvm'.c).st = ft(f).st$ - Ein neuer function frame für f wird erzeugt.
- $\forall i \in [1 : ft(f).np].\ top(c').va(ft(f).par(i)) = cvm'.vm(cvm.cp).gpr(i)$ - Die Registerinhalte von u werden als Parameter übergeben.
- $cvm'.c.pr = ft(f).body; return\_from\_trap(); cvm.c.pr$ - Die Funktion wird ausgeführt und mit der speziellen Funktion $return\_from\_trap()$ abgeschlossen. Das System wechselt danach je nach scheduling wieder zu einem Benutzerprozess in den user mode.
- Dazu muss jedoch gespeichert werden, welcher Prozess den kernel call ausgeführt hat und welcher Prozess als nächstes geschedulet wird. Dafür nutzen wir die globalen Variable CU und setzen $top(c').va(rds) = lv(CU, cvm.c)$. Außerdem gilt $va(CU, cvm'.c) = cvm.cp$.

Der Kern kann dem Benutzer Erfolgs- und Fehlermeldungen mit Hilfe der speziellen Funktion $setreg(u, i, X)$ in Register zurückgeben. Der Effekt von $setreg$ ist wie folgt definiert.

$$cvm'.vm(\widetilde{u}).gpr(\widetilde{i_5}) = \widetilde{X}_{32}$$

Man beachte, dass ein Programm, solange es nicht mit dem Kern oder anderen Programmen kommuniziert, genau so ausgeführt wird als wäre es allein auf der Hardware. Ein solches Programm ist also sicher und kann nicht von anderen Programmen gehackt werden. Dazu muss natürlich der Kern korrekt implementiert sein.

5.3 CVM-Implementierung

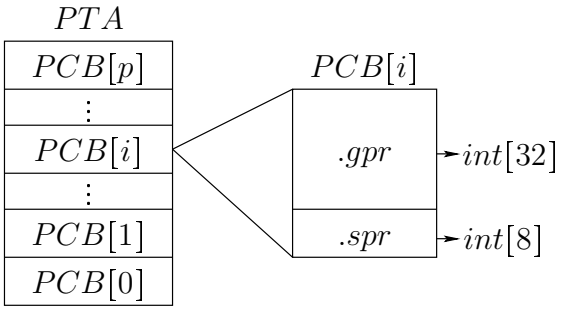
Nun soll CVM implementiert werden. Dazu betrachten wir den *konkreten Kern* K, bestehend aus:

- dem abstrakten Kern k
- zusätzlichen Datenstrukturen und Funktionen

Die zusätzlichen Komponenten werden zum Abstrakten Kern hinzugelinkt. Formal gesehen benötigt man dazu eine „Theorie des Linkens“ von $C0$-Programmen, die unter anderem die Disjunktheit verwendeter Namen behandelt. Zudem werden im konkreten Kern die speziellen Funktionen mit Hilfe von inline assembler code implementiert, wir wollen uns hier aber auf die Datenstrukuren des konkreten Kerns konzentrieren.

5.3.1 Datenstrukturen des konkreten Kerns

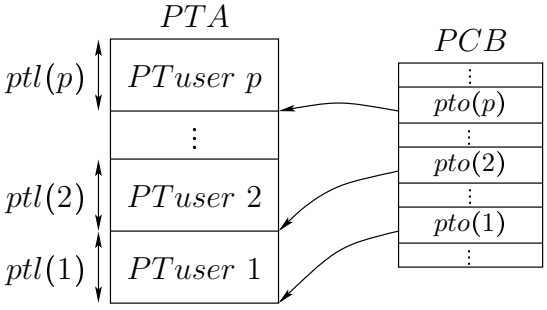
Zur Verwaltung der page tables existiert im Speicher ein array von **p**rocess **c**ontrol **b**locks PCB im Speicher.



Dies ist ein struct, in dem die aktuelle Konfiguration eines Benutzerprogramms gespeichert wird. $PCB[0]$ wird hier nicht benutzt, da unser Kern außer durch $reset$ nicht unterbrochen werden kann. Der PCB enthält keine separate Komponente zur Speicherung des pc, denn dieser wird im Falle eines Interrupts in im special purpose register EPC gesichert und befindet sich somit schon im PCB. Man kann PCB als $C0$-Datenstruktur deklarieren:

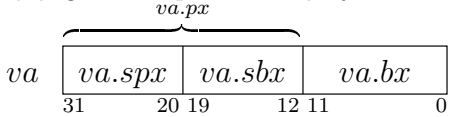
```
typedef int[32] u;
typedef int[8] v;
typedef struct{u gpr, v spr} pcb;
typedef pcb[p+1] pcba;
pcba PCB;
```

Weiterhin gibt es ein **p**age **t**able **a**rray PTA, das alle page tables der Benutzer $1, \ldots, p$ fasst. Die Startadressen der page tables für jeden Benutzer können aus den process control blocks der Benutzer über $pto[i] = PCB[i].spr[5]$ berechnet werden. Dabei befindet sich die page table von Benutzer i im Bereich $PTA[pto[i+1]-1 : pto[i]] = d.m_{4 \cdot ptl(i)}(pto[i])$.



Die direkte Anordnung aller page tables „übereinander“ ist offensichtlich nicht sehr effizient. Falls $ptl(i)$ erhöht wird, müssen alle page tables darüber nach oben verschoben werden. Stattdessen reservieren wir für jede page table $ptlmax$ viele Speicherworte im page table array.

In der physikalischen Maschine hatten wir den swap memory sm eingeführt (vgl. Abbildung 5.6). Dieser soll im konkreten Kern genutzt werden, um den physikalischen Speicher zu vergrößern. Ähnlich wie bei der virtuellen Adressübersetzung unterteilt man den swap memory in große Seiten von $1M$ und führt eine zweite Adressübersetzung mit Hilfe von swap page tables ein. Der page index $va.px$ einer virtuellen Adresse va wird in swap page index spx und swap byte index sbx getrennt.



Genauso wie für die gewöhnlichen page tables werden ein **s**wap **p**age **t**able **a**rray $SPTA$ und ein **s**wap **p**age **t**able **o**rigin array angelegt in denen die **s**wap **p**age **t**able $spt(i)$ und deren Startadressen $spto(i)$ für jeden Benutzer i abgespeichert sind.

Die page tables werden für viele Benutzer schnell groß und nehmen viel Platz ein. Daher ist es ratsam,

die Größe des PTA zu begrenzen. Wir legen die maximale Größe auf $4M$ fest. Es stellt sich dann die Frage, wieviel **t**otal **v**irtual **m**emory TVM zur Verfügung steht. Dies lässt sich mit einer einfachen Rechnung beantworten.

$$\begin{aligned} size(PTA) &= \underbrace{size(pte)}_{4} \cdot \underbrace{\#pages}_{TVM/pagesize} \\ &= 4 \cdot \frac{TVM}{4K} \leq 4M \\ TVM &\leq 4G \end{aligned}$$

Es stehen mit der Begrenzung also maximal 4GB virtueller Speicher für alle Benutzer zur Verfügung.

Desweiteren verfügt K über vier einfach verkettete Listen zur Seitenverwaltung. Diese geben Aufschluss über:

- freie Seiten (4K) im pm
- freie Seiten (1M) im sm
- besetzte Seiten (4K) im pm
- besetzte Seiten (1M) im sm

Um diese Listen auf dem Laufenden zu halten, gibt es viele Strategien. Eine davon wäre, wiederholt neue Elemente einzufügen und alte zu streichen. Die würde aber dazu führen dass viele Elemente auf dem heap liegen bleiben, die nicht mehr zugänglich sind. Man spricht von *garbage* und es werden gemeinhin garbage collector eingesetzt, um diesen zu reduzieren, jedoch sind solche Routinen langsam und bisher noch nicht während einer Rechnung im Hintergrund (on the fly) durchführbar. Daher verzichten wir auf solche Maßnahmen und wählen lieber einen besseren Ansatz.
Es fällt auf, dass die Längen der free- und used-Listen zusammenaddiert die Anzahl der Seiten ergeben. Daher empfielt es sich Elemente zwischen den Listen einfach nur „umzuhängen“ anstatt neue Elemente auf dem heap zu erstellen. Man kann die Listen auch direkt als array implementieren. Dies kann als eigenständige Übung durchgeführt werden.

5.3.2 Korrektheit

Um über die Korrektheit des Betriebssystems zu argumentieren, betrachten wir die CVM-Rechnung $cvm^0, cvm^1, \ldots$ (abstrakter Kern $cvm^i.c$ und Benutzer $cvm^i.vm(u)$). Die Behauptung ist dann, dass

- Konfigurationen $d^0, d^1, d^2, \ldots$ der *physikalischen* DLX-Maschine
- Rechnungen $k^0, k^1, k^2, \ldots$ des *konkreten* Kerns ($C0A$)
- Schrittzahlen $s(0), s(1), s(2), \ldots$ des konkreten Kerns
- Schrittzahlen $t(0), t(1), t(2), \ldots$ der DLX
- allocated base adresses $aba^0, aba^1, \ldots$ aus der Compiler-Korrektheit und
- allocated sub-variable $asv^0, asv^1, \ldots$ aus der Kernel-Implementierung

existieren, so dass eine Simulationsrelation zwischen CVM, Kernel und DLX gilt.
Für globale Variablen X gilt $aba^0(lv(X,c),c) = aba^1(lv(X,c),c) = \ldots = aba(X)$, da sich die zugeordneten Basisadressen von globalen Variablen im Programmverlauf nicht ändern. Die Datenstrukturen des konkreten Kerns liegen daher in globalen Variablen. Daraus folgt, dass die free- und used-Listen garnicht auf dem heap angelegt werden können, sondern als array im gm implementiert werden müssen.
Für Identifier e, die auf globale Subvariablen einfachen Typs verweisen, definieren wir:

$$va(e,d) = d.pm_4(aba(e))$$

Es werden hier erneut die Notationen aus $C0$-Semantik, DLX-Konfiguration und dem Compiler zusammengeführt. Mit Hilfe der Datenstrukturen von k kann man nun die Adressübersetzung für mehrere Benutzer definieren.

$$\begin{aligned} pto(d,u) &= va(d, PCB[u].pto) \\ ptea(d,u,va) &= pto(d,u) + 4 \cdot va.px \\ pte(d,u,va) &= d.pm_4(ptea(d,u,va)) \\ ppx(d,u,va) &= pte(d,u,va)[31:12] \\ v(d,u,va) &= pte(d,u,va)[11] \\ pma(d,u,va) &= ppx(d,u,va) + 4 \circ va.bx \end{aligned}$$

Es gilt für die process control blocks:

$$\begin{aligned} PCB[0].pto &= aba(PCB) \\ \forall u > 0 : PCB[u].pto &= \sum_{i<u} PCB[i].ptl + PCB[0].pto \end{aligned}$$

Analog wird die Adressübersetzung für swap-Adressen definiert.

Nun kann man die Simulationsrelation $sim(cvm^i, k^{s(i)}, d^{t(i)}, aba^i, asv^i)$ aufstellen, die besagt:

- $d^{t(i)}$ kodiert cvm^i

Allerdings fehlt hier der bezug zu aba und ein direkter Beweis ist schwierig. Daher spaltet man die Simulation in Teilschritte auf. $sim(cvm^i, k^{s(i)}, d^{t(i)}, aba^i, asv^i)$ ist dann äquivalent zu:

1. $d^{t(i)}$ kodiert $k^{s(i)}$
2. $k^{s(i)}$ kodiert $cvm^i.c$
3. $d^{t(i)}$ kodiert alle $cvm^i.vm(u)$

Die erste Konsistenzbedingung wurde schon mit $consis(d^{t(i)}, aba^i, k^{s(i)})$ zur Compilerkorrektheit eingeführt. Für die zweite Bedingung führen wir eine ähnliche Relation $\mathrm{k-consis}(k^{s(i)}, asv^i, cvm^i.c)$ ein. Die dritte Bedingung spiegelt sich in der *B-Relation* $\forall u$ wider.

Definition 5.1 *(B-Relation) Gilt die B-Relation $B(cvm, u, d)$, so wird $cvm.vm(u)$ von d kodiert. Für alle Register R des Benutzers u gilt dann:*

$$cvm.vm(u).R = \begin{cases} d.R & : \quad cvm.cp = u \quad (u \text{ lauft}) \\ va(d, PCB[u].R) & : \quad \text{sonst} \end{cases}$$

Für den Speicher von Benutzer u gilt:

$$cvm.vm(u).m(va) = \begin{cases} d.pm(pma(d,u,va)) & : \quad v(d,u,va) = 1 \quad (\text{im RAM}) \\ d.sm(sma(d,u,va)) & : \quad v(d,u,va) = 0 \quad (\text{auf der Festplatte}) \end{cases}$$

Für die Konstistenz zwischen abstraktem und konkretem Kern wird die Relation $\mathrm{k-consis}(k, asv, c)$ benötigt. Sie verwendet die allocated sub-variable function

$$asv : \{(\text{Sub–})\text{Variablen von } c\} \longrightarrow \{(\text{Sub–})\text{Variablen von } k\},$$

die die Verbindung zwischen den Variablen in beiden Kernen herstellt. $asv(x) = x'$ bedeutet dann in etwa, dass x' die Variable von k ist, die x simuliert.

Definition 5.2 *(k-consis) Gilt die k-consis-Relation $\mathrm{k-consis}(k, asv, c)$, so wird c von k simuliert und es gilt $\mathrm{k-econsis}$ und $\mathrm{k-pconsis}$ für einfache Variablen x.*

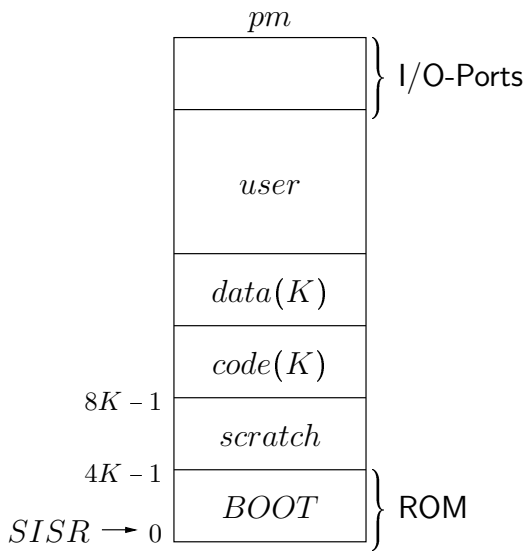


Abbildung 5.9: memory map des physikalischen Speichers

- k – econsis *(elementare Werte):*

$$\begin{aligned} va(c,x) &= va(k,x') \\ &= va(k,asv(x)) \end{aligned}$$

- k – pconsis *(pointer):*

$$\begin{aligned} va(c,x) &= y \\ \Rightarrow va(k,asv(x)) &= asv(y) \end{aligned}$$

Eine Implementierung, für die diese Konsistenzrelationen gelten, führt dann das CVM-Betriebssystem korrekt aus. Dies lässt sich mit Hilfe der $C0$-Semantik und Computer-gestützten Beweissystemen überprüfen.

5.3.3 Implementierungsdetails

Nun soll ein Einblick in die Implementierung des Kerns gegeben werden. Dabei werden wir uns auf wenige Problembeispiele beschränken.

memory map

Zunächst muss man einen Plan erstellen, wie der Speicher der physikalischen Maschine aufgeteilt werden soll. Dazu legt man eine memory map fest, wie sie in Abbildung 5.9 dargestellt ist. Wir unterscheiden die folgenden Speicherbereiche:

- $BOOT$ - ein **r**ead **o**nly **m**emory (ROM) in dem der *bootloader* residiert. Dieser testet, ob ein reset vorlag, und läd gegebenenfalls den Betriebssystemkern von der Festplatte.
- $scratch$ - Dieser Bereich ist eine Art „Schmierzettel“ für den Betriebssystemprogrammierer auf dem temporäre Daten zwischengelagert werden können
- $code(K)$ - Der Code des konkreten Kerns K
- $data(K)$ - Die Daten des konkreten Kerns K

- $user$ - Die Benutzerprogramme und -daten
- I/O-Ports - Der Zugriff auf I/O-Geräte erfolgt durch Lese- und Schreiboperationen auf gewissen Adressen im Speicher. Diese werden als I/O-Ports bezeichnet. Man unterscheidet:
  - data ports
  - control ports (status, command)

Es soll nun angenommen werden, user u läuft gerade auf der Maschine. Sofern kein Interrupt auftritt, wird das Benutzerprogramm von u korrekt abgearbeitet, solange die B-Relation hält. Dies ist durch die Korrektheit der memory management unit zu gewährleisten. Interessanter sind die Fälle, in denen Interrupts auftreten.

Interrupts

Ist $JISR(c) = 1$ so springt das Programm zur Adresse $SISR = 0^{32}$ nach $BOOT$ und der bootloader wird im system mode ausgeführt. Dieser testet auf einen reset:

$$ECA[0] \stackrel{?}{=} 1$$

Dafür muss jedoch zuerst der Inhalt des ECA special purpose Register in ein general purpose Register geladen werden mittels $movs2i$:

$$gpr(x) = ECA$$

Falls jedoch kein reset vorlag, hätte man nun den vorherigen Wert von $gpr(x)$ unwiederbringlich überschrieben, deshalb rettet man ihn vorher mit einem sw nach $scratch$. Dieses zunächst triviale erscheinende Beispiel zeigt schon die Notwendigkeit des $scratch$ auf.
Wenn kein reset den Interrupt auslöste, übergibt der bootloader an den konkreten Kern der die interrupt handler bereithält. Zuvor müssen jedoch die Register R von Prozess u in einen process control block exportiert werden.

$$PCB[u].R = vm(u).R$$

Dabei ist zu beachten, dass sich $vm(u).gpr(x)$ mittlerweile im $scratch$ befindet und von dort gerettet werden muss. Diese gesamte Prozedur wird *process save* genannt und muss mit Hilfe von inline assembler Code programmiert werden. Im weiteren Verlauf wird der bootloader verlassen und der Kern muss den interrupt level il berechnen.

$$il = \min\{i \mid vm(u).ECA[i] = 1\}$$

Da das exception cause register durch process save in die $C0$-Subvariable $PCB[u].ECA$ gesichert wurde, kann die Berechnung von il durch ein einfaches $C0$-Programm (z.B. geschachtelte Fallunterscheidungen) ausgeführt werden, so dass kein inline assembler benötigt wird.
Angenommen ein page fault tritt auf, dann gilt:

- $pff \Leftrightarrow il = 3$
- $pfls \Leftrightarrow il = 4$

Die Adresse, die die exception auslöste steht dann in $PCB[u].EPC$ bzw. $PCB[u].EDATA$ und der page fault handler wird gestartet.

page fault handler

Bei einem page fault muss eine benötigte Seite aus dem Speicher nachgeladen werden. Ist die *free list* des pm noch nicht leer, das heißt, sind noch freie Plätze im physikalischen Speicher für eine weitere Seite verfügbar, dann muss nur die benötigte Seite in pm aus dem swap memory mit Hilfe der entsprechenden Treiber geladen werden und die page tables aktualisiert werden.
Ist kein Platz mehr frei, so muss eine Seite ausgewählt werden, die nach sm ausgelagert wird (*victim selection*). Diese Seite darf auf keinen Fall diejenige sein, die als letztes in den pm geladen wurde. Der Grund dafür ist einfach der, dass pro Instruktion immer zwei page faults auftreten können, nämlich pff und $pfls$, und diese in beiden Reihenfolgen. Wenn man nun stets die Seite auslagert, die der vorherige page fault eingelagert hat, so gerät man in einen Teufelskreis aus page faults, der zu einem *deadlock* führt. Beachtet man die oben genannte Vorschrift, so kann jede Instruktion mit höchstens zwei page faults ausgeführt werden.

Nach Abschluss des interrupt handler müssen die Register des unterbrochenen Prozesses u wiederhergestellt werden. Man spricht von *process restore*. Danach kann die Kontrolle wieder an u übergeben werden und dieser rechnet weiter, als ob nie ein Interrupt aufgetreten wäre.

5.4 inline assembler code

Bisher wurde immer von inline assembler code gesprochen, ohne dessen genaue Bedeutung zu klären. Zum Abschluss soll deswegen die Syntax und Semantik von inline assembler code definiert werden. Wir benötigen inline assembler in der Implementierung des Konkreten Kerns, denn ausschließlich mit $C0$ kann man niemals auf bestimmte Register, Speicheradressen, I/O-Geräte etc. zugreifen. Es wird daher eine Möglichkeit benötigt, Assembler-Code in gewöhnlichen $C0$-Code einzubetten. Dies geschieht mit dem Befehl $asm(p)$, wobei p ein Programm in DLX-Assembler ist. Die Übersetzung von asm ist denkbar trivial. Es wird lediglich p in den generierten Code eingefügt.

$$code(asm(p)) = p$$

Bei der Semantik von inline assembler müssen stets zwei Rechnungen gleichzeitig betrachtet werden:

- $k^0, k^1, \ldots, k^i, k^{i+1}, \ldots$ - eine $C0A$-Rechnung
- $d^0, d^1, \underbrace{\ldots, d^i, d^{i+1}, \ldots}_{\text{inline asm}}$ - eine DLX-Rechnung

Dabei sind vor allem load/store-Anweisungen interessant. Beim process save muss zum Beispiel einer $C0$-Variable der Wert eines Registers zugewiesen werden. Für $x = PCB[u]$ und $sw(d^i)$ gilt dann:

$$\begin{aligned} ea(d^i) &= aba(lv(x, k^i), k^i) \\ va(x, k^{i+1}) &= d^i.gpr(RD(d^i)) \end{aligned}$$

Man beachte, wie erneut DLX- und $C0$-Semantik elegant und problemlos miteinander vermischt werden

Anhang A

Instruktionssatz der DLX

Die folgenden Abbildungen zeigen die Syntax und Semantik der DLX-Architektur.

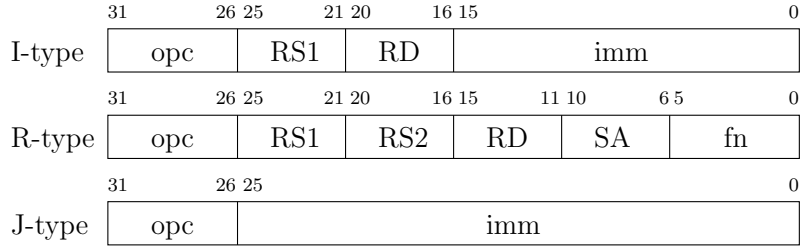


Abbildung A.1: Typen von Instruktionen

$I[28{:}26]$ / $I[31{:}29]$	000	001	010	011	100	101	110	111
000	*Rtype*	—	j	jal	—	—	—	—
001	—	—	—	—	—	—	—	—
010	addis	addi	subis	subi	andi	ori	xori	lhgi
011	clri	sgri	seqi	sgei	slsi	snei	slei	seti
100	—	—	—	lw	—	—	—	—
101	—	—	—	sw	—	—	—	—
110	—	—	—	—	beqz	bnez	jr	jal
111	—	—	—	—	—	—	trap	rfe

Tabelle A.1: Übersicht über die Kodierung der $Itype$- und $Jtype$- Instruktionen

$I[2{:}0]$ / $I[5{:}3]$	000	001	010	011	100	101	110	111
000	adds	add	subs	sub	and	or	xor	lhg
001	clr	sgr	seq	sge	sls	sne	sle	set
010	movs2i	movi2s	—	—	—	—	—	—
011	—	—	—	—	—	—	—	—
100	—	—	—	—	—	—	—	—
101	—	—	—	—	—	—	—	—
110	—	—	—	—	—	—	—	—
111	—	—	—	—	—	—	—	—

Tabelle A.2: Übersicht über die Kodierung der $Rtype$-Instruktionen, $I[31:26]$ = 000000

load & store		move	
lw	RD = m_4(RS1 + sxtimm)	movs2i	RD = SA(SPR[SA] → GPR[RS1])
sw	m_4(RS1 + sxtimm) = RD	movi2s	SA = RS1(GPR[RD] → SPR[SA])
I-Type control			
beqz	PC = PC + (RS1 = 0 ? sxtimm : 4)	jr	PC = RS1
bnez	PC = PC + (RS1 ≠ 0 ? sxtimm : 4)	jalr	R31 = PC + 4; PC = RS1
I-type arithmetic & logic		R-type arithmetic & logic	
addis	RD = RS1 + sxtimm (signed overflow)	adds	RD = RS1 + RS2 (signed overflow)
addi	RD = RS1 + 0^{16}imm (unsigned overflow)	add	RD = RS1 + RS2 (unsigned overflow)
subis	RD = RS1 – sxtimm (signed overflow)	subs	RD = RS1 – RS2 (signed overflow)
subi	RD = RS1 – 0^{16}imm (unsigned overflow)	sub	RD = RS1 – RS2 (unsigned overflow)
andi	RD = RS1 ∧ sxtimm	and	RD = RS1 ∧ RS2
ori	RD = RS1 ∨ sxtimm	or	RD = RS1 ∨ RS2
xori	RD = RS1 ⊕ sxtimm	xor	RD = RS1 ⊕ RS2
lhgi	RD = imm 0^{16}	lhg	RD = RS2$[15:0]$ 0^{16}
I-type test & set		R-type test & set	
clri	RD = 0^{32}	clr	RD = 0^{32}
sgri	RD = (RS1 > sxtimm ? $0^{31}1 : 0^{32}$)	sgr	RD = (RS1 > RS2 ? $0^{31}1 : 0^{32}$)
seqi	RD = (RS1 = sxtimm ? $0^{31}1 : 0^{32}$)	seq	RD = (RS1 = RS2 ? $0^{31}1 : 0^{32}$)
sgei	RD = (RS1 ≥ sxtimm ? $0^{31}1 : 0^{32}$)	sge	RD = (RS1 ≥ RS2 ? $0^{31}1 : 0^{32}$)
slsi	RD = (RS1 < sxtimm ? $0^{31}1 : 0^{32}$)	sls	RD = (RS1 < RS2 ? $0^{31}1 : 0^{32}$)
snei	RD = (RS1 ≠ sxtimm ? $0^{31}1 : 0^{32}$)	sne	RD = (RS1 ≠ RS2 ? $0^{31}1 : 0^{32}$)
slei	RD = (RS1 ≤ sxtimm ? $0^{31}1 : 0^{32}$)	sle	RD = (RS1 ≤ RS2 ? $0^{31}1 : 0^{32}$)
seti	RD = $0^{31}1$	set	RD = $0^{31}1$
J-Type			
j	PC = PC + sxtimm	jal	R31 = PC + 4; PC = PC + sxtimm
trap	trap-Interrupt	rfe	return from exception

Tabelle A.3: Semantik der DLX-Instruktionen

Anhang B

Change Log

Hier werden die Veränderungen an der Mitschrift von Version zu Version gegenchronologisch aufgelistet.

Revision 60

- **Aktualisierung**: $C0$-Grammatik: Erklärung angepasst, Konstanten sind keine Identifier, $\langle eltyp \rangle$ komplett entfernt, Begründung: int, $bool$, $char$ und $unsigned$ aus $\langle Na \rangle$ ableitbar, Kontextbedingung: Typdefinitionen dürfen nur bereits definierte Typen verwenden und die elementaren Typen sind standartmäßig in der Typtabelle enthalten.

Revision 59

- **Inkonsistenz**: Sektion 4.6.3: Übersetzung Funktionsaufruf, testcode-Kommentar, $\langle stkptr \rangle < \langle hbase \rangle$ statt stkptr < heapptr
- **Ergänzung**: Sektion 4.6.4 hinzugefügt, Korrektheitsbeweis der Kontrollkonsistenzbedingung für den Compiler
- **Inkonsistenz**: Definition von $\mathrm{r-consis}$ für ra in 4.6.4 formalisiert, Einführung von $c.calls(i)$
- **Inkonsistenz**: $C0$-Grammatik: $\langle eltyp \rangle$ noch in Definition von $\langle Typ \rangle$, $\langle Typ \rangle$ aus new-Anweisung und Funktionsdeklaration entfernt, somit keine anonymen Typen mehr möglich, Kontextbedingung erzwingt das Benutzen definierter Typen

Revision 58

- **Inkonsistenz**: Spezifikation Logic Unit: pos statt g.
- **Inkonsistenz**: Sektion 4.4.2: Beispiel für Subvariable $(m, PCB)[17].gpr[5]$ anstatt alter Notation.
- **Inkonsistenz**: Sektion 4.4.5: $Ra(typ((m,x)s))$ anstatt $Ra(typ((m,i)s))$
- **Inkonsistenz**: Sektion 4.6.2: `sw k 30` $Fsize(f')+8$ anstatt `sw k 30` $Fsize(f')+4$ bei Funktionsaufruf, Speichern von Rückgabeadresse in $F_j.rd$
- **Inkonsistenz**: Abbildung 4.17 an Definition von $\mathrm{p-consis}$ angepasst
- **Inkonsistenz**: diverse Stellen: $m.st$ anstatt $st.m$
- **Inkonsistenz**: C0-grammatik: $ParF$ statt PF
- **Aktualisierung**: Kapitel 5: an neue Notation angepasst, teilweise Formulierungen überarbeitet, Typos gefixt

- **Fehler**: Sektion 5.1.1: Definition von il, $mca(d, eev)[j] = 1$ statt $j = 1$
- **Ergänzung**: Sektion 5.1.1: Klarstellung zu den verbotenen User-Instruktionen hinzugefügt.
- **Aktualisierung**: Abbildung 5.7: überarbeitet
- **Fehler**: Definitionen von ex_{in} und $phase_{in}$ berichtigt. Berücksichtigung von $mode$-Wechseln
- **Fehler**: Definitionen von v bei page faults berichtigt.
- **Inkonsistenz**: $mode(d)$ statt $d.mode$
- **Ergänzung**: Sektion 5.2: Klarstellung zur Sichtbarkeit von page faults in cvm hinzugefügt
- **Aktualisierung**: $\widetilde{X}$ muss Zahlenwert statt Binärdarstellung für Ausdruck X berechnen.
- **Inkonsistenz**: $copy$-Primitiv, $\widetilde{u_1}$ und $\widetilde{u_2}$ statt $\widetilde{s}$ und $\widetilde{r}$ in Vorbedingung
- **Inkonsistenz**: $copy$-Primitiv, benötige $(\widetilde{s_1})_{32}$ für Adressen
- **Ergänzung**: $copy$-Primitiv, Anmerkung zum Zugriff des Kerns auf user memory hinzugefügt.
- **Aktualisierung**: $alloc$ und $free$, an neue Notation angepasst, $cvm.ptl(i) \in \mathbb{N}$
- **Fehler**: $alloc$-Primitiv, $4K$ statt Faktor 4 in zweiter Zeile
- **Aktualisierung**: 5.2.2, Syntax von kcd und $trap$-Semantik an neue Notation angepasst.
- **Inkonsistenz**: Semantik von Funktionsaufrufen und $trap$, $top(c').va(ft(f).par(i))$ anstelle von $va(ft(f).par(i), c')$ bei Parameterübergabe
- **Ergänzung**: spezielle Funktion $setreg$ zu $trap$-Semantik hinzugefügt.
- **Aktualisierung**: Sektion **??**, Umstrukturierung, Notation angepasst.
- **Aktualisierung**: Sektion **??**, Notation der sw-Semantik angepasst

Revision 57

- **Inkonsistenz**: Sektion 4.6.2: Bedingungen an $d.gpr(j)$ und $d.gpr(k)$ in Code für Zuweisung an neue Notation angepasst.
- **Fehler**: neue DLX-Spezifikation und -Implementierung: Spec fordert für `addi` und `subi` dass mit $sxtimm(c)$ gerechnet weren muss, brauche stattdessen $0^{16}imm(c)$.

Revision 56

- **Inkonsistenz**: Definition 4.4.6: $va(e, c)$ statt $va(c, e)$ an diversen Stellen
- **Aktualisierung**: DLX-Instruktionssatz im Anhang: `add(i)o` und `sub(i)o` (overflow signalisiert) ist jetzt `add(i)s` und `sub(i)s` (signed overflow). Ebenso `add(i)` und `sub(i)` jetzt unsigned overflow anstatt no overflow. Das alles wegen Änderung in der ALU. f_0 bestimmt u Bit anstatt der Overflowmaskierung
- **Inkonsistenz**: Ausdrucks-Coderzeugung Arrayzugriff: x und x' vertauscht, Fallunterscheidung über e, $displ(lv(e, c), f)$ anstatt $displ(xs, f)$
- **Inkonsistenz**: $C0$-Grammatik, $\cdot$ statt $*$ für Multiplikation, weniger Verwechslungsgefahr mit Dereferenzierungsoperator $*$. Ebenfalls in vorhergehenden Grammatiken angepasst.
- **Inkonsistenz**: $h\text{–}consis$: $\langle d.gpr(29_5)\rangle < hmax$, muss Zahlenwert vom heap pointer vergleichen

- **Fehler**: Ausdrucksübersetzung Pointer dereferenzieren: $R(e) = 1$ für $u = e*$
- **Inkonsistenz**: Sektion 4.6.2: Marken $j, k, l \in \{0,1\}^5$
- **Aktualisierung**: Sektion 4.6.3: Anweisungsübersetzung an neue Notation angepasst.
- **Aktualisierung**: Sektion 4.6.3: Funktionsaufruf, heap wächst nun nach oben, testcode verändert
- **Fehler**: in der Vorlesung, stack pointer ist $gpr(30_5)$ nicht $gpr(29_5)$
- **Ergänzung**: Sektion 4.6.3: Funktionsaufruf, mehr Kommentare zu testcode
- **Ergänzung**: Makro $geti(j, b)$ zum Laden von Konstanten b in Register j definiert.
- **Inkonsistenz**: $code(f)$ =`sw 31 31 0`, $code(ft(f).body)$, nicht $code(ft(f).body)$

Revision 55

- **Inkonsistenz**: Definition 4.20: $base(i, c)$ statt $base(c, j)$ bei rds und ra, $i \geq 1$ für $pbase$
- **Fehler**: Definition 4.20: $F_i.ra = start(code(c.pr(r(i, c)+1)))$ anstatt $F_i.ra = start(code(head(r(i, c)+1)))$
- **Inkonsistenz**: Abbildung 4.16: jetzt $F_i.rd$ statt $F_i.rb$
- **Aktualisierung**: Sektion 4.6.2: Beweis des Aho-Ullman-Algorithmus, $\leq$ durch $=$ ersetzt wo $<$ nicht gelten kann
- **Aktualisierung**: Sektion 4.6.2: An neue Notation für Ausdrucksauswertung angepasst, Arithmetische Operationen, pointer und Address-of fehlen noch
- **Aktualisierung**: Sektion 4.6: $displ(xs, f)$ eingeführt
- **Inkonsistenz**: Sektion 4.4.2: $m.st$ statt $st.m$
- **Inkonsistenz**: Sektion 4.5: Symboltabellen haben drei Elemente ($st.f$ vergessen) und sind Tupel, keine Mengen

Revision 54

- **Aktualisierung**: Sektion 4.6: aba nicht mehr überladen wie im alten Skript, zusätzliche Erklärung hinter Definitionen hinzugefügt.
- **Inkonsistenz**: Sektion 4.4.5: Einschränkung von $Ra(t)$ für Pointertypen t mit $\exists t' \in tt.na.\, tt.tc(t) = t'*$ abgeschwächt, pointer auf local memories grundsätzlich erlaubt (für rds), allerdings Address-of auf lokalen variablen verboten, $null$ in $Ra(t)$ aufgenommen
- **Inkonsistenz**: Sektion 4.4.6: new-Anweisung, $R = \max\{c.hm.st.na\}$ undefiniert für R^0, ersetzt durch $R = H(c) - 1$
- **Ergänzung**: Sektion 4.4.6: $default$-Funktion deklariert,
- **Ergänzung**: Sektion 4.6: Definition der Programmintegrität hinzugefügt
- **Aktualisierung**: Sektion 4.6: $start(code(an))$ anstelle von $start(an)$, Hinweis auf komplexen Beweis von $c{-}consis$ hinzugefügt, $r{-}consis$ überarbeitet, alignment hinzugefügt, $h{-}consis$ ausgelagert aus $r{-}consis$

Revision 53

- **Inkonsistenz**: Einführung von $size$ erst in 4.6, außerdem hat $size$ nun die Einheit $bytes$ statt $words$.
- **Inkonsistenz**: Definition von $displ$ angepasst

Revision 52

- **Fehler**: Sektion 4.6: Faktor 4 in Definition von $displ(x, st)$ hinzugefügt
- **Inkonsistenz**: Sektion 4.4.5: Ausdrucksauswertung, $lv(e'[e''], c) = lv(e', c)[\langle va(e'', c)\rangle]$ anstatt $lv(e', c)[va(e'', c)]$
- **Ergänzung**: Sektion 4.4.5: Ausdrucksauswertung, $va(id, c)$ für verschiedene Identifier ausgeschrieben.
- **Inkonsistenz**: Sektion 4.6: Definition $base(0, c)$ benötigt zusätzlichen Summand 8 für $pbase$ und ra im gm
- **Aktualisierung**: Sektion 4.5: Programmausführung auf Stand der Vorlesung gebracht

Revision 51

- **Aktualisierung**: Sektion 4.4.6: Anweisungsausführung auf Stand der Vorlesung gebracht
- **Inkonsistenz**: Funktionsdeklarationen: Typ $rtyp(f)*$ muss in der Typtabelle aufgeführt sein. $tt.tc(ft(f).st.typ(rds) = rtyp(f)$
- **Ergänzung**: Sektion 4.6: Erklärung zum Wesen von aba hinzugefügt
- **Aktualisierung**: Abbildung 4.15, heap und stack wachsen nun nach oben
- **Ergänzung**: Sektion 4.6: Erklärungen zum memory layout und zu den stack frames erweitert.
- **Aktualisierung**: Sektion 4.6: Invariante $c.lms(j).st.f = f$ hinzugefügt
- **Aktualisierung**: Sektion 4.6: $base$ und $displ$ an neue Definitionen angepasst.
- **Fehler**: Displacements benötigen Faktor 4 im Vergleich zur Vorlesung
- **Aktualisierung**: Sektion 4.6: aba an neue Definitionen angepasst.

Revision 50

- **Inkonsistenz**: Erklärung zu Lsitenbeispiel und Abbildung 4.9: `u` statt `LEL*` in erstem Beispielcode
- **Aktualisierung**: Ausdrucksauswertung: binäre arithmetische und boolesche Ausdrücke getrennt, zusätzliche Erklärung
- **Fehler**: $C0$-Grammatik: $struct$-Zeile von Typ verrutscht, Anpassung an neue Definition von $VaDF$ (keine leeren structs)
- **Aktualisierung**: $C0$-Grammatik: Deklarationsfolgen umbenannt, $TypDF \rightarrow TyDF$, $FuDF \rightarrow FunDF$, $VarDF$ eingeführt, längerer Name impliziert dass die Folge nichtleer ist (vgl. $VaDF$, $VarDF$), $PaDF$ anstatt $ParDF$, Funktionsdeklaration auf einen Fall reduziert (enthält alle Möglichkeiten für Anzahl der Parameter/lokale Variablen), ebenso Funktionsaufruf nur noch ein Fall, erkauft durch ε in Grammatik

Revision 49

- **Fehler**: Erklärung $Ra(t)$ für structs: f nicht s-stellig
- **Aktualisierung**: Sektion 4.4.2: neue Definitionen für Speicher und Variablen
- **Aktualisierung**: Definition 4.11: Subvariablen überarbeitet
- **Aktualisierung**: Definition 4.4 überarbeitet
- **Aktualisierung**: Konfiguration von $C0$-Maschinen vereinfacht
- **Ergänzung**: Sektion 4.2: Hinweis zu garbage collection hinzugefügt
- **Inkonsistenz**: Typtabelle: $tt.tc : TN \longrightarrow TD$ anstelle von $tt.tc : [0 : tt.n - 1] \longrightarrow TD$
- **Aktualisierung**: Ausdrucksauswertung angepasst, Notation lv, va, vv, Sonderfälle bei $e \circ e'$ für Multiplikation, Division und Vergleichsoperationen
- **Ergänzung**: Erklärung zu Pointerauswertung hinzugefügt
- **Ergänzung**: Sektion 1.6.1: Definition for $twoc_n(x)$ hinzugefügt
- **Ergänzung**: Einschränkung von Pointerwerten

Revision 48

- **Ergänzung**: Definition 4.1: Bedingungen an N und S hinzugefügt.
- **Typo**: Referenz [LMW86] repariert.
- **Typo**: Sektion 4.2: "auch globale" statt "aglobale"
- **Ergänzung**: Klarstellung zum leeren Wort ε, kein Alphabet enthält das leere Wort.
- **Aktualisierung**: type table angepasst an neue Notation
- **Aktualisierung**: $Ra(t)$ umdefiniert, elementare typen als Bitstrings, komplexe Typen mit Fallunterscheidung und Funktionen als Wert

Revision 47

- **Ergänzung**: Definition von c^0 und Erklärung zu Vorraussetzungen des Simulationssatzes hinzugefügt.
- **Ergänzung**: Abbildung 3.10: MUX am PC hinzugefügt (aus nextpc-Schaltung herausgezogen)
- **Fehler**: Beweis Lemma 3.3: Im letzten Schritt wird $\overline{h^{2i}.ex} = 1$ benötigt anstatt $h^{2i+1}.ex = 1$.
- **Ergänzung**: Beweise Lemmata 3.4 und 3.5 erweitert, Bemerkung zu Kosten/Bezug der Prädikate hinzugefügt.
- **Ergänzung**: Lemma 3.9: Bedeutung der ALU-Korrektheit hervorgehoben.
- **Fehler**: Beweis $h^{2i+2}.gpr(x) = c^{i+1}.gpr(x)$: $lw(c^i)$ anstatt $lw_h(c^i)$
- **Aktualisierung**: Umbennenung von $bjtaken$ in $jbtaken$ in Konstruktion und Korrektheitsbeweis der DLX

Revision 46

- **Aktualisierung**: C0-Grammatik angepasst, Programm besteht nun aus strikter Folge von Typ-, Variablen- und Funktionsdeklarationen, Variablendeklarationen dürfen nur deklarierte Typen benutzen.

Revision 45

- **Aktualisierung**: Sektion 1.6.2: v in $alures$ umbenannt. $alures'$ und $alures''$ in ALU-Implementierung vertauscht
- **Fehler**: Sektion 1.6.1: Herleitung neg-Bit, $u = 0$, Beweis $[a] \pm [b] \equiv_{\mathrm{mod}\, 2^{n+1}} [s[n:0]]$ berichtigt
- **Aktualisierung**: 1.6.2: z wurde in eq umbenannt

Revision 44

- **Fehler**: Implementierung $sxtimm'$: $Jtype'(h.IR)$ statt $Jtype'(I)$
- **Ergänzung**: Kapitel 2.2.2: 3-Port-RAM hinzugefügt
- **Aktualisierung**: Einleitung zu Kapitel 3 überarbeitet
- **Fehler**: Abbildung 3.2: Adresse y zeigt auf niedrigstes Byte
- **Fehler**: Sektion 1.5.1: Definitionen der Reflexivität und Symmetrie von $\equiv_{\mathrm{mod}\, k}$ berichtgt
- **Ergänzung**: Anhang mit Instruktionssatz hinzugefügt
- **Aktualisierung**: Sektionen 1.6.2 und 3.1.5: Unterscheidung $aluop(a, b, f)$ und $alures(c)$ geklärt.
- **Aktualisierung**: Sektion 3.1.6: Abschnitt zu Kontrollinstruktionen überarbeitet, weitere Erklärungen hinzugefügt, $bjtaken$ in $jbtaken$ umbenannt

Revision 43

- **Ergänzung**: Kapitel 2.2.2: Anmerkung zu SRAM hinzugefügt
- **Ergänzung**: Grafiken 3.1 und 3.2 erweitert
- **Fehler**: Sektion 1.2.3: Straight Line Program Beispiele, z_1 statt z_0
- **Ergänzung**: Sektion 2.3: Anmerkung zum $reset$-Signal
- **Ergänzung**: Sektion 2.4: Decoder-Beweis üeerarbeitet und vervollständigt

Revision 42

- **Fehler**: Eingänge von Schaltkreis-Grafik vor Sektion 1.2.1 vertauscht
- **Inkonsistenz**: Sektion 1.5.1: Notation verenheitlicht, a, b, c statt x, y, z
- **Ergänzung**: Beweis Subtraktionsalgorithmus ausführlicher
- **Ergänzung**: ∨-tree auf Breite d verallgemeinert

Revision 41

- **Inkonsistenz**: Sektion 1.4: Tiefe von MUX in Tiefenberechung des CSA berichtigt, Tiefe 2 statt 3, Beweis ausführlicher
- **Aktualisierung**: Vergleiche in Tabelle 1.7 sind $signed$
- **Aktualisierung**: Vergleichsbits z und g in eq und pos umbenannt

Revision 40

- **Aktualisierung**: Definition Subtraktionsalgorithmus überarbeitet
- **Aktualisierung**: Sektion 1.6.1 überarbeitet, Modulo-Schreibweise angepasst, Typos berichtigt
- **Fehler**: c_{n+1} statt c_n, T_{n+1} statt T_n und a_n, d_n statt a_{n+1}, d_{n+1} in Herleitung neg-Bit für $u = 0$

Revision 39

- **Aktualisierung**: Modulo-Exkurs an Vorlesung angepasst, genauere Abgrenzung von unärem und binärem Modulo
- **Aktualisierung**: Lemma 1.9 überarbeitet
- **Ergänzung**: Lemmata 1.10 und 1.11 hinzugefügt

Revision 38

- **Fehler**: Sektion 1.2.2: Beispiel für Lösung von Bestimmungsgleichung, $\overline{X}_2 X_1 \overline{X}_0 = 1$ anstatt $\overline{X}_2 \overline{X}_1 \overline{X}_0 = 1$
- **Ergänzung**: Lemmata $\bigwedge$ und $\bigvee$ gekennzeichnet
- **Aktualisierung**: Fußzeilen aktualisiert, jetzt mit Revisionsnummer

Revision 37

- **Ergänzung**: unsauberer Schritt im Beweis des Zerlegungslemmas ausgebessert
- **Aktualisierung**: Sektion 1.2.3: Klammerschreibweise von φ angepasst

Revision 36

- **Aktualisierung**: Sektion 1.2.2: Klammerschreibweise von φ angepasst
- **Ergänzung**: Beweis DNF detallierter
- **Ergänzung**: Definition der "Lösung von Bestimmungsgleichungen" konkretisiert, wir wenden die Erweiterung von φ auf boolesche Ausdrücke an, sind aber nur an der Lösung φ, d.h. den Werten für die Variablen, interessiert

Revision 35

- **Ergänzung**: Definition der Summenschreibweise verallgemeinert, jetzt auch Summen definiert, die nicht bei 0 beginnen
- **Formatierung**: Beweise für Zerlegungslemma und Additionsalgorithmus besser lesbar
- **Ergänzung**: Einschub Bitfolgen hinzugefügt
- **Aktualisierung**: Sektion 1.2.1: Definition von φ bereinigt

Revision 34

- **Fehler**: Beweis Lemma 0.5 berichtigt

Revision 33

- **Aktualisierung**: Lemma 0.5 benötigt keine Induktion, folgt direkt aus Definition der Addition
- **Ergänzung**: Einschub Summennotation

Revision 32

- **Aktualisierung**: Portierung des Skripts nach SS10 aus SS08-Skript
- **Fehler**: Die Unendlichkeit der natürlichen Zahlen beruht auf den ersten 4 Peano-Axiomen

Revision 31

- **Fehler**: Beispiel 1 C0-Programmausführung, $ft(main).body$ enthält return
- **Fehler**: Interrupt Special Purpose Registers, Erklärung von EPC fehlerhaft

Revision 30

- **Ergänzung**: zusätzliche Passagen aus Mitschrift07, CSA, memory correctness proof

Revision 29

- **Fehler**: Tabelle 5.2, ECA und ESR vertauscht
- **Fehler**: DLX-Simulationssatz, $gpr_{Din}(h^{2i+1})$, Bedingung ($x = AdC(h^{2i+1})$) überflüssig
- **Inkonsistenz**: cvm-Konfiguration und -Übergangsfunktion überarbeitet
- **Ergänzung**: Ausdrucksauswertung für char-Konstanten hinzugefügt.

Revision 28

- **Inkonsistenz**: Definition einfache Werte, Pointer dürfen doch auf Subvariablen zeigen (Wir ignorieren hier den garbage collector)
- **Inkonsistenz**: Kapitel 4, p^* anstat $p*$ für pointer und pointer-Typen
- **Inkonsistenz**: Anweisungsausfügrung, Beispiel while-Schleife, prinzipiell ist Induktion über Schleifendurchgänge möglich, aber komplizierter, Formulierung abgemildert
- **Inkonsistenz**: Abb. 3.10 und 3.13, MUX unterschiedlich breit
- **Inkonsistenz**: Definition $base(c, j)$, $sbase$ kommt als Bezugspunkt hinzu
- **Inkonsistenz**: Hinweise auf Übungsaufgaben entfernt
- **Fehler**: $free(u, x)$, $-_{32}$ statt $+_{32}$
- **Ergänzung**: Vorlesung vom 16.07.08 hinzugefügt

Revision 27

- **Typo**: findet ich auf der Vorlesungswebsite
- **Inkonsistenz**: DLX-Konfiguration, $c.gpr(0^5)$ nicht mehr immer 0^{32}
- **Inkonsistenz**: C_0-Konfiguration, $c.hm.name(i) = \Omega$
- **Inkonsistenz**: Ausdrucksauswertung Identifier, $st.top(c)$ und $c.st.gm$ statt $top(c)$ und $c.gm$
- **Inkonsistenz**: Anweisungsausführung, $rbs(c.rd - 1)$
- **Inkonsistenz**: Compiler-Korrektheit, function frames F_i mit $i \in [0 : c.rd - 1]$
- **Ergänzung**: Implementierung der Adressübersetzung hinzugefügt
- **Ergänzung**: Vorlesung vom 14.07.08 hinzugefügt

Revision 26

- **Inkonsistenz**: DLX-Konfigurationen heißen d ab Kapitel 5
- **Ergänzung**: Interrupt-Implementierung hinzugefügt
- **Ergänzung**: Vorlesung vom 09.07.08 hinzugefügt

Revision 25

- **Inkonsistenz**: Definition von Subvariablen überarbeitet
- **Inkonsistenz**: Definition 4.11, j überflüssig
- **Inkonsistenz**: Funktionsdeklarationen, main wird explizit deklariert
- **Inkonsistenz**: dynamische Semantik von C_0, $ba(u)$, $size(u)$ und $typ(u)$ ebenfalls c-abhängig
- **Inkonsistenz**: Anweisungsausführung entsprechend angepasst
- **Inkonsistenz**: C_0-Konfiguration in 4.4.4 und Beispielprogramme Abschnitt 4.5, Abb. 4.16, Überreste aus alter Mitschrift korrigiert ($c.rd$ etc.)
- **Fehler**: Definition Typtabelle, structs, $\varepsilon\{k < 3 + i \mid tt.name(k) = t_j\}$ statt $\varepsilon\{k \mid tt.name(k) = t_k\} < 3 + i$
- **Ergänzung**: Definition $typ(v)$ hinzugefügt
- **Ergänzung**: Definition Menge der einfachen Werte hinzugefügt
- **Ergänzung**: Vorlesung vom 07.07.08 hinzugefügt

Revision 24

- **Typo**: beudeutet,
- **Inkonsistenz**: Definition $ba(v[j])$, Typ t undefiniert
- **Inkonsistenz**: Definition $ba(v[k])$ und $ba(v.n_k)$, j überflüssig
- **Inkonsistenz**: Definition C_0-Konfiguration, $c.ct.lms(i)$ sind Folgen einfacher Werte, $c.rbs$ enthält Subvariablen
- **Inkonsistenz**: Ausführung von Funktionsaufrufen, $top(c') = lms(c'.rd - 1)$
- **Fehler**: Softwaremultiplikation bei Übersetzung von Ausdrücken, Passage überarbeitet
- **Ergänzung**: Ausführung von Zuweisungen, Kontextbedingung für globale pointer hinzugefügt

Revision 23

- **Inkonsistenz**: C_0-Konfiguration, $c.rd$ entspricht nicht #returns im Programmrest, sonder #Frames auf dem lms, pr wird mit body von $main$ ohne $return$ initialisiert
- **Inkonsistenz**: C_0-Compiler, Anpassungen an neue Notation, $c^0.rd = 1$, $c.st$ etc.
- **Inkonsistenz**: $ithreturn(c, i)$, es muss $i \in [1 : c.rd - 1]$ gelten, deshalb gilt die Korrektheit von ra für die Frames $c.rd - i$
- **Fehler**: Abb. 4.15, $c.rd - 1$ statt $c.rd$.
- **Ergänzung**: Vorlesung vom 02.07.08 hinzugefügt

Revision 22

- **Inkonsistenz**: Definition Ausdrucksauswertung, $va(c, X) = va(c, bind(c, X))$
- **Ergänzung**: $size(c, m)$ hinzugefügt.
- **Ergänzung**: Vorlesung vom 30.06.08 hinzugefügt
- **Ergänzung**: C_0-Programmverifikation hinzugefügt, angepasst an neue memory-Notation

Revision 21

- **Inkonsistenz**: alignment von ea nur für load und store word
- **Inkonsistenz**: Typtabelle, pointer auf sich selbst, also $tt(i).tc = ptr(i)$, sind nicht verboten (aber sinnlos)
- **Inkonsistenz**: neue Notation für Speicher, Variablen und ihren Wert zur Laufzeit eingeführt
- **Ergänzung**: Ableitungsbäume, Ableitungsrelation "⇒" hinzugefügt.
- **Ergänzung**: Vorlesung vom 25.06.08 hinzugefügt

Revision 20

- **Inkonsistenz**: Speicherdefinition, Symboltabelle nun $st(m)$ statt $sy(m)$
- **Inkonsistenz**: Grammatik für vollst. gekl. arith. Ausdrücke, enthält nun Ziffern von 0 bis 9
- **Inkonsistenz**: Ableitungsbeispiel in Abbildung 4.3, jetzt ein Knoten pro (Nicht-)Terminal
- **Ergänzung**: Vorlesung vom 23.06.08 hinzugefügt

Revision 19

- **Typo**: rechten seite
- **Inkonsistenz**: Definition $size(j)$, $size(i)$ statt $size(t')$
- **Inkonsistenz**: Grammatik für vollst. gekl. boolesche Ausdrücke, Variablennamen enthalten Ziffern von 0 bis 9
- **Ergänzung**: rekursive Definition von Subvariablen hinzugefügt.
- **Ergänzung**: Vorlesung vom 18.06.08 hinzugefügt

Revision 18

- **Fehler**: Definition $comp(c)$, $I(c)[5:4] = 00$ statt 01
- **Inkonsistenz**: Modulorechnung: Abschnitt überarbeitet, Hilbert-Operator eingeführt
- **Ergänzung**: Anmerkungen zur C_0-Grammatik, boolesche Variablen und Funktionsaufrufe.
- **Ergänzung**: Vorlesung vom 16.06.08 hinzugefügt

Revision 17

- **Fehler**: signed ovf Fall 2.1, $c_{n-1} - c_n$ statt $c_n - c_{n-1}$
- **Fehler**: Lemma 3.11, h^{2i+2} statt h^{2i+1}
- **Inkonsistenz**: C_0-Grammatik: $KompDF$ redundant zu $VaDF$, deswegen $KompDF$ entfernt und in $struct$ durch $VaDF$ ersetzt.
- **Ergänzung**: Zuweisung von chars und boolesche Variablen zur C_0-Grammatik hinzugefügt.
- **Ergänzung**: Vorlesungen vom 09.06.08 und 11.06.08 hinzugefügt

Revision 16

- **Typo**: Standart
- **Fehler**: AU, Semantik von $\langle s \rangle$, $\mod 2^n$ in der letzten Zeile hinzugefügt
- **Inkonsistenz**: Spezifikation gpr, 3.5, Register 0 darf beschrieben werden
- **Ergänzung**: Vorlesung vom 04.06.08 hinzugefügt

Revision 15

- **Typo**: arithmetisch, binäre Baum, wir durch eine, nach zu Beginn,
- **Fehler**: Beweis ex-Schaltkreis, $i \to i+1$ statt $i = 0$
- **Fehler**: Beweis des Subtraktionsalgorithmus, TWOC Lemma 2 statt 1
- **Fehler**: Einschub Vektoroperationen, $x \in \{0,1\}^n$ anstatt $x \in \{0,1\}$
- **Fehler**: Definition φ für SLPs, x_j statt x_y
- **Fehler**: Definition AU, $s \in \{0,1\}^n$ statt $s \in \{0,1\}$
- **Inkonsistenz**: ovf Fall 2, d statt b'
- **Inkonsistenz**: Registeradressierung, ab sofort heißen die Adressen AdA, AdB und AdC
- **Ergänzung**: RAM-Implementierung, Q^j mit $j \in \{0, \ldots, 2^a - 1\}$
- **Ergänzung**: Vorlesung vom 02.06.08 hinzugefügt

Revision 14

- **Inkonsistenz**: $gpr(0^5)$ kann jetzt beliebige Werte besitzen. $gpr(0^5) = 0^{32}$ darf nicht mehr angenommen werden.
- **Inkonsistenz**: striktere Unterscheidung zwischen Prädikaten/Funktionen $p'(I)$,$f'(I)$ über Instruktionen und $p(c)$,$f(c)$ über Konfigurationen.
- **Ergänzung**: Vorlesung vom 26.05.08 hinzugefügt
- **Ergänzung**: Vollständiges Kapitel zur Assemblersprache hinzugefügt
- **Ergänzung**: Vorlesung vom 28.05.08 hinzugefügt

Revision 13

- **Ergänzung**: Vorlesung vom 21.05.08 hinzugefügt

Revision 12

- **Typo**: Abbildung 1.1 anstatt Abbildung 1.2.3, foglende
- **Ergänzung**: MUX hinzugefügt
- **Ergänzung**: Schaltkreise interpretiert als Graphen zur formalen Definition die Tiefe
- **Ergänzung**: Vorlesung vom 19.05.08 hinzugefügt

Revision 11

- **Typo**: Fall 2.2, $AU(n)$
- **Fehler**: Definition AU, $b \oplus 1 = \overline{b}$ anstatt $b \oplus 1 = \overline{1}$

Revision 10

- **Fehler**: Definition ovf, $T_n = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$ anstatt $T_n = \{-2^{n-1}, \ldots, 2^{n-1} - 2\}$
- **Ergänzung**: x^n Notation hinzugefügt
- **Ergänzung**: AU überarbeitet
- **Ergänzung**: Vorlesung vom 14.05.08 hinzugefügt

Revision 9

- **Fehler**: Definition Monom, $X \in V^n$ anstatt $X \in \{0,1\}^n$
- **Fehler**: Beweis Lemma 1.6 Zeile 3, Lemma 1.5 anstatt 1.4
- **Fehler**: Beweis Lemma 1.6,Indizes $n-1 \ldots 0$ statt $1 \ldots n$

Revision 8

- **Ergänzung**: Geometrische Reihe hinzugefügt

Revision 7

- **Fehler**: Lemma 1.4, $\varphi(e_1 \wedge e_2) = 1$ statt $\varphi(e_1 \wedge e_2 = 1)$
- **Ergänzung**: Kosten und Tiefe von Schaltkreisen hinzugefügt
- **Ergänzung**: Vorlesung vom 07.05.08 hinzugefügt

Revision 6

- **Ergänzung**: Vorlesung vom 05.05.08 hinzugefügt

Revision 5

- **Typo**: Schreiweise, So würde die Nachfolgerfunktion unär wird wie folgt definiert, reelle Welt, Sollen, booleshcen, bekannt Schema
- **Fehler**: Lemma $\varphi(e_1 \vee e_2 \vee \ldots \vee e_n)$, $\vee$ statt $\wedge$
- **Inkonsistenz**: Definition X^{ε}, X statt x
- **Ergänzung**: konventionelle Definition von $\varphi(e \circ e')$
- **Ergänzung**: Vorlesung vom 30.04.08 hinzugefügt

Revision 4

- **Fehler**: „Kleines 1+1“, $Z = 1 \cdot Z + 0$ statt $Z = 1 \cdot Z + 1$
- **Ergänzung**: Vorlesung vom 28.04.08 hinzugefügt

Revision 3

- **Typo**: Boolesche
- **Fehler**: „Einschub Minus“, $x \dot{-} 1 = V(x)$ statt $x \dot{-} = V(x)$
- **Fehler**: Tabelle binärer Additionsalgorithmus, c' ist nicht immer 0
- **Fehler**: Tabelle boolesche Funktionen, $\neg x$ statt $\neg a$
- **Ergänzung**: Abschnitt 1.2.1, $\oplus$ zu Prioritäten hinzugefügt

Revision 2

- **Ergänzung**: Dieses Change Log hinzugefügt
- **Typo**: Schlatkreise, menge
- **Fehler**: in Definition von Lemma 1.1 (Zerlegungslemma): „... dann lässt sich $a[n-1:0]$ in...“statt „... dann lässt sich $a[n-1]$ in...“
- **Inkonsistenz**: $\langle c_{i+1}\ s_i \rangle$ und $\langle c_n, s[n-1:0] \rangle$, ab sofort wird Konkatenation von bit strings ohne spezielles Zeichen ausgedrückt, also $\langle c_n\ s[n-1:0] \rangle$, es sei denn die Schreibweise ist nicht eindeutig. Dann kann “,” oder “$\circ$” verwendet werden.
- **Ergänzung**: Abschnitt 0.5, Unterscheidung von Z und 10, Identität von einstelligen Zahlen und ihren Darstellungen hinzugefügt
- **Ergänzung**: Vorlesung vom 23.04.08 hinzugefügt

Revision 1

- **Fehler**: im Beispiel für Definition 0.5: Basis 2 statt 4 für $\langle 101 \rangle_4$

Revision 0

- **Ergänzung**: Vorlesungen vom 16.04.08 und 21.04.08 hinzugefügt

Literaturverzeichnis

[KP95] Jörg Keller and Wolfgang J. Paul. *Hardware Design*. Teubner, 1995.

[LMW86] Jacques Loeckx, Kurt Mehlhorn, and Reinhard Wilhelm. *Grundlagen der Programmiersprachen*. Teubner, Stuttgart, 1986.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.