Mitschrift

Informatik 2 / Systemarchitektur SS07

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

Inhaltsverzeichnis

0 Notation **1**
0.1 Unärzahlen . . . 1
0.2 Algebraische Notation . . . 1

1 Rechnen in Booleschen Ausdrücken **3**
1.1 Boolesche Ausdrücke . . . 3
1.2 Schaltfunktionen . . . 4
1.3 Schaltkreise . . . 6

2 Arithmetic / Logic Units (ALUs) **9**
2.1 Addition . . . 9
2.1.1 Addierer . . . 9
2.1.2 Binärdarstellungen . . . 9
2.1.3 Additionsalgorithmus . . . 10
2.1.4 Formale Definition der Rechenoperationen . . . 12
2.1.5 Kosten und Tiefe von Schaltkreisen . . . 13
2.1.6 Multiplexer . . . 13
2.1.7 Conditional Sum Adder . . . 14
2.2 Subtraktion . . . 16
2.2.1 Modulorechnung . . . 16
2.2.2 Subtraktionsalgorithmus . . . 17
2.2.3 Two's-Complement-Zahlen . . . 17
2.2.4 Korrektheitsbeweis . . . 18
2.3 Implementierung . . . 19
2.3.1 Arithmetic Unit . . . 19
2.3.2 ovf und neg für Two's-Complement-Zahlen . . . 20
2.3.3 neg und ovf für Binärzahlen . . . 22
2.3.4 Konstruktion der ALU . . . 23

3 Prozessoren **26**
3.1 Spezifikation . . . 26
3.1.1 Instruktionen . . . 27
3.1.2 Übergangsfunktion . . . 28
3.1.3 load word & store word . . . 29
3.1.4 ALU-Instruktionen . . . 29
3.1.5 Kontrollinstruktionen . . . 30
3.2 Assemblersprache . . . 31
3.3 Hardware . . . 33
3.3.1 Register . . . 34
3.3.2 Speicher . . . 35
3.3.3 Program Counter . . . 36
3.4 Korrektheit . . . 36

3.4.1 Simulationsrelation und Simulationssatz . . . 37
3.4.2 Simulationsbeweis . . . 38
3.4.3 Register 0 . . . 48

4 Software / C0 **50**
4.1 Kontextfreie Grammatiken . . . 50
4.2 C0-Syntax . . . 54
4.3 Typdeklarationen . . . 56
4.4 C0-Semantik . . . 58
4.4.1 Typdeklarationen . . . 59
4.4.2 Variablendeklarationen . . . 62
4.4.3 Funktionsdeklarationen . . . 64
4.4.4 Konfiguration von C0-Maschinen . . . 65
4.4.5 Ausdrucksauswertung . . . 66
4.4.6 Anweisungsausführung . . . 68
4.5 Korrektheit von C0-Programmen . . . 70
4.5.1 Beispiel 1: Zuweisung und Fallunterscheidung . . . 70
4.5.2 Beispiel 2: Rekursion . . . 71
4.5.3 Beispiel 3: Dynamische Speicherzuweisung . . . 76
4.5.4 Beispiel 4: Schleife . . . 77
4.6 C0-Compiler . . . 79
4.6.1 Korrektheit . . . 79
4.6.2 Ausdrucksübersetzung . . . 84
4.6.3 Anweisungsübersetzung . . . 89

5 Betriebssystem-Kernel **93**
5.1 Betriebssytemunterstützung im Prozessor . . . 93
5.1.1 Interrupts . . . 93
5.1.2 Adressübersetzung . . . 99
5.2 CVM-Semantik . . . 102
5.2.1 Spezielle Funktionen des abstrakten Kerns . . . 104
5.2.2 Semantik von trap-Instruktionen . . . 105
5.3 CVM-Implementierung . . . 105
5.3.1 Datenstrukturen des konkreten Kerns . . . 106
5.3.2 Korrektheit . . . 107
5.3.3 Implementierungsdetails . . . 109
5.4 inline assembler code . . . 111

Zusammenfassung

Die vorliegende Mitschrift fasst den Inhalt der Vorlesung *„Systemarchitektur"*, gehalten von Prof. Dr. Wolfgang J. Paul an der Universität des Saarlandes im Sommersemester 2007, 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.

Diese Mitschrift wurde vorlesungsbegleitend von Christoph Baumann verfasst. Fragen und Hinweise werden dankend über die email-Adresse *s9chbaum@stud.uni-saarland.de* entgegengenommen.

Kapitel 0

Notation

0.1 Unärzahlen

Unärzahlen sind die primitivste Form der Zahlendarstellung. Zahlen werden zum Beispiel durch eine gewisse Anzahl von *Strichen* repräsentiert.

$$|||||$$

Definition 0.1 *Addition um 1 wird wie folgt definiert:*

$$(X + 1) = X\ |$$

(Mache 1 Strich hinter die Zahl!)

Den verschiedenen Zahlen gibt man unterschiedliche Namen, beispielsweise:

- $| \mathrel{\widehat{=}} \text{Eins}$
- $|| \mathrel{\widehat{=}} \text{Drei}$
- $||| = (3 + 1) \mathrel{\widehat{=}} \text{Zwei}$
- $|||| \mathrel{\widehat{=}} @\clubsuit\mho\#\bigstar\wp\triangle$
- $\cdots$

Das Beispiel verdeutlicht: *Namen sind „Schall und Rauch“*. Grundlagen für Zahlensysteme bilden die Peano-Axiome.

0.2 Algebraische Notation

Algebraische Notation verwendet:

- Konstanten, Variablen sowie arithmetische Ausdrücke, bestehend aus denselben, z.B.

  $$(X + 1) \cdot (y + 3) - 4$$

- Identitäten, z.B.

  $$(a + b)^2 \equiv (a^2 + 2ab + b^2)$$

- Klammerungsregeln, z.B.

$$\left.\begin{matrix} 2 - - - y \\ y = 3 \end{matrix}\right\} \rightarrow -1$$

- Abkürzungen, z.B.

$$e^2 \mathrel{\widehat{=}} e \cdot e$$

- Bestimmungsgleichungen

$$a + 1 = 2 \Rightarrow a = 1$$

„=“ ist mit mehreren Bedeutungen überlagert.

Im Folgenden werden diese Grundlagen vorrausgesetzt. Weitere Informationen zu Zahlensystemen und den Hintergründen mathematischer Notation finden sich in [KP95].

Kapitel 1

Rechnen in Booleschen Ausdrücken

1.1 Boolesche Ausdrücke

Boolesche Ausdrücke verwenden:

- Konstanten:

  $$0, 1$$

- Operationen:

  $$\sim, \wedge, \vee, \oplus$$

  Boolesche Funktionen (NOT, AND, OR, XOR)

  $$\sim : \{0,1\} \longrightarrow \{0,1\} \qquad \wedge, \vee, \oplus \; : \; \underbrace{\{(0,0),(0,1),(1,0),(1,1)\}}_{\widehat{=}\{0,1\}^2} \longrightarrow \{0,1\}$$

- Prioritäten:
  - Unäre Operatoren ($\sim$) haben Vorrang vor binären Operatoren ($\wedge, \vee$)
  - Es gilt *„Punkt vor Strich"*, wobei $\wedge$ als Punkt- und $\vee$ als Strichrechnung angesehen wird.

X	$\sim X$
0	1
1	0

Tabelle 1.1: Wertetabelle für NOT

X_1	X_2	$X_1 \wedge X_2$	$X_1 \vee X_2$	$X_1 \oplus X_2$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Tabelle 1.2: Wertetabelle für AND, OR und XOR

- Identitäten:

  Theorem 1.1 *Es gelten folgende Gesetze:*

– *Kommutativgesetze*

$$\begin{aligned} X_1 \wedge X_2 &\equiv X_2 \wedge X_1 \\ X_1 \vee X_2 &\equiv X_2 \vee X_1 \\ X_1 \oplus X_2 &\equiv X_2 \oplus X_1 \end{aligned}$$

– *Assoziativgesetze*

$$\begin{aligned} (X_1 \wedge (X_2 \wedge X_3)) &\equiv ((X_1 \wedge X_2) \wedge X_3) \\ (X_1 \vee (X_2 \vee X_3)) &\equiv ((X_1 \vee X_2) \vee X_3) \\ (X_1 \oplus (X_2 \oplus X_3)) &\equiv ((X_1 \oplus X_2) \oplus X_3) \end{aligned}$$

– *Distributivgesetz*

$$\begin{aligned} (X_1 \wedge (X_2 \vee X_3)) &\equiv ((X_1 \wedge X_2) \vee (X_1 \wedge X_3)) \\ (X_1 \vee (X_2 \wedge X_3)) &\equiv ((X_1 \vee X_2) \wedge (X_1 \vee X_3)) \end{aligned}$$

Beweis: durch Wahrheitstabellen

- Abkürzungen:

$$\begin{aligned} X \wedge Y &\mathrel{\widehat{=}} XY \\ \sim X &\mathrel{\widehat{=}} \bar{X} \end{aligned}$$

Weiterhin gelten die folgenden Identitäten und Äquivalenzen:

$$\begin{aligned} X \wedge \bar{X} &\equiv 0 \\ X \vee \bar{X} &\equiv 1 \\ \bar{X} = 1 &\Leftrightarrow X = 0 \\ \bar{X} = 0 &\Leftrightarrow X = 1 \\ X_1 \wedge X_2 = 1 &\Leftrightarrow X_1 = 1 \text{ UND } X_2 = 1 \\ X_1 \wedge X_2 \wedge \cdots \wedge X_n = 1 &\Leftrightarrow \forall i \in \{1, \ldots, n\} : \; X_i = 1 \\ X_1 \vee X_2 = 1 &\Leftrightarrow X_1 = 1 \text{ ODER } X_2 = 1 \\ X_1 \vee X_2 \vee \cdots \vee X_n = 1 &\Leftrightarrow \exists i \in \{1, \ldots, n\} : \; X_i = 1 \end{aligned}$$

1.2 Schaltfunktionen

Definition 1.1 *Schaltfunktion* f

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

Gesucht: Ausdruck e über $X_1, \ldots, X_n$, so dass

$$f(X_1, \ldots, X_n) \equiv e \Leftrightarrow (f(X_1, \ldots, X_n) = 1 \Leftrightarrow e = 1)$$

Beispiel: Addierer

- **SUM** $s \; : \; \{0,1\}^3 \longrightarrow \{0,1\}$
- **CARRY** $c' \; : \; \{0,1\}^3 \longrightarrow \{0,1\}$

	a	b	c	c'	s	
	0	0	0	0	0	
e_{s_1}	0	0	1	0	1	
$\vee e_{s_2}$	0	1	0	0	1	
$\vee e_{s_3}$	1	0	0	0	1	
	0	1	1	1	0	e_{c_1}
	1	0	1	1	0	$\vee e_{c_2}$
	1	1	0	1	0	$\vee e_{c_3}$
$\vee e_{s_4}$	1	1	1	1	1	$\vee e_{c_4}$

Tabelle 1.3: Addierer-Beispiel

Gesucht: Ausdrücke e_{sum}, e_{carry}, die SUM, CARRY *berechnen*.
Tabelle 1.3 zeigt die Wertzuweisung aller möglichen Eingangskombinationen auf die Ausgangsbits s und c. Das Summenbit s ist Eins sobald an den Eingängen a, b und c eine ungerade Anzahl Einsen anliegt. Das Übertragsbit c (*Carry-Bit*) ist Eins, wenn an den Eingängen zwei oder mehr Einsen anliegen. Für jede dieser Eingangskombinationen werden Teilausdrücke (*Monome*) e_{s_i} bzw. e_{c_i} gebildet, die genau dann Eins ergeben, wenn die jeweilige Kombination anliegt. Die Monome werden dann zum Gesamtausdruck verodert.

$$\begin{aligned}
e_{s_1} = 1 \Leftrightarrow (a=0) \;\wedge\; (b=0) \;\wedge\; (c=1) &\leftrightarrow e_{s_1} = \bar{a} \wedge \bar{b} \wedge c \\
e_{s_2} = 1 \Leftrightarrow (a=0) \;\wedge\; (b=1) \;\wedge\; (c=0) &\leftrightarrow e_{s_1} = \bar{a} \wedge b \wedge \bar{c} \\
e_{s_3} = 1 \Leftrightarrow (a=1) \;\wedge\; (b=0) \;\wedge\; (c=0) &\leftrightarrow e_{s_1} = a \wedge \bar{b} \wedge \bar{c} \\
e_{s_4} = 1 \Leftrightarrow (a=1) \;\wedge\; (b=1) \;\wedge\; (c=1) &\leftrightarrow e_{s_1} = a \wedge b \wedge c \\
e_{sum} &= e_{s_1} \vee e_{s_2} \vee e_{s_3} \vee e_{s_4}
\end{aligned}$$

$$\begin{aligned}
e_{c_1} = 1 \Leftrightarrow (a=0) \;\wedge\; (b=1) \;\wedge\; (c=1) &\leftrightarrow e_{c_1} = \bar{a} \wedge b \wedge c \\
e_{c_2} = 1 \Leftrightarrow (a=1) \;\wedge\; (b=0) \;\wedge\; (c=1) &\leftrightarrow e_{c_1} = a \wedge \bar{b} \wedge c \\
e_{c_3} = 1 \Leftrightarrow (a=1) \;\wedge\; (b=1) \;\wedge\; (c=0) &\leftrightarrow e_{c_1} = a \wedge b \wedge \bar{c} \\
e_{c_4} = 1 \Leftrightarrow (a=1) \;\wedge\; (b=1) \;\wedge\; (c=1) &\leftrightarrow e_{c_1} = a \wedge b \wedge c \\
e_{carry} &= e_{c_1} \vee e_{c_2} \vee e_{c_3} \vee e_{c_4}
\end{aligned}$$

Theorem 1.2 Darstellungssatz

$$\forall f : \exists e,\ e \text{ berechnet } f$$

Für jede Schaltfunktion existiert ein Boolescher Ausdruck, der die Funktion berechnet. Zum Beweis des Satzes werden folgende Definitionen und Lemmas benötigt.

Definition 1.2

$$\text{Sei } X \text{ eine Variable, } \varepsilon \in \{0,1\}$$

$$X^{\varepsilon} = \begin{cases} X, & \varepsilon = 1 \\ \bar{X}, & \varepsilon = 0 \end{cases}$$

Lemma 1.3 $X^{\varepsilon} = 1 \leftrightarrow X = \varepsilon$

Beweis:

$$\begin{aligned} \varepsilon = 1 \quad &: \quad X^1 \overset{\text{(def)}}{=} X = 1 \leftrightarrow X = 1 \checkmark \\ \varepsilon = 0 \quad &: \quad X^0 \overset{\text{(def)}}{=} \bar{X} = 1 \leftrightarrow X = 0 \checkmark \quad \square \end{aligned}$$

Lemma 1.4 *Sei* $a = [a_1, \ldots, a_n] \in \{0,1\}^n$*, Monom* $m(a) = X_1^{a_1} \wedge X_2^{a_2} \wedge \cdots X_n^{a_n}$

$$m(a) = 1 \leftrightarrow \forall i \in \{1, \ldots, n\} : a_i = X_i$$

Beweis:

$$\begin{aligned} m(a) = 1 \quad &\leftrightarrow \quad X_1^{a_1} \wedge X_2^{a_2} \wedge \cdots X_n^{a_n} = 1 \quad \text{(Def. Monom)} \\ &\leftrightarrow \quad \forall i : X_i^{a_i} = 1 \quad \text{(Def. AND)} \\ &\leftrightarrow \quad \forall i : X_i = a_i \quad \text{(Lemma 1.3)} \quad \square \end{aligned}$$

Jedes Monom $m(a)$ deckt eine Zeile in der Wertetabelle ab. Es wird 1, sobald die Variablen $X_1, \ldots, X_n$ die entsprechenden Werte $a_1, \ldots, a_n$ annehmen. Es lässt sich der Träger von f definieren:

Definition 1.3 *Träger von f*

$$T(f) = \{a \mid f(a) = 1\}$$

Damit erhalten wir die vollständige disjunktive Normalform (DNF) für f:

Definition 1.4 *DNF*

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

$$T(f) = \emptyset \Rightarrow d(f) = 0$$

Beweis: $d(f) = 1 \overset{!}{\leftrightarrow} f(X_1, \ldots, X_n) = 1$

$$\begin{aligned} d(f) = 1 \quad &\leftrightarrow \quad \exists a \in T(f) : m(a) = 1 \\ &\leftrightarrow \quad \exists a \in T(f) : X = a \quad \text{(Lemma 1.4)} \\ &\leftrightarrow \quad X \in T(f) \\ &\leftrightarrow \quad f(X) = 1 \quad \text{(Definition 1.3)} \quad \square \end{aligned}$$

1.3 Schaltkreise

Schaltkeise lassen sich mit Hilfe von *Straight Line Programs* algebraisch beschreiben

Definition 1.5 *Straight Line Program (SLP)*
Gegeben: *Eingangsvariablen* $\{X_1, X_2, \ldots\} \in V$
Programm: *Für eine Folge von Zuweisungen* $(Z_1, \ldots, Z_n)$ *gilt* $\forall i \in \{1, \ldots, n\}$:

- $Z_i \in V$ *oder*
- $Z_i = \bar{Z}_j$ *mit* $j < i$ *oder*
- $Z_i = Z_j * Z_k$ *mit* $j, k < i$ *und* $* \in \{\wedge, \vee, \oplus\}$

Aus solchen Programmen lassen sich logische Schaltkreise entwickeln die das jeweilige SLP repräsentieren. Abbildung 1.1 verdeutlicht die Konstruktionsvorschrift. Man kann auf diese Weise für jeden Booleschem Ausdruck einen Schaltkreis generieren.

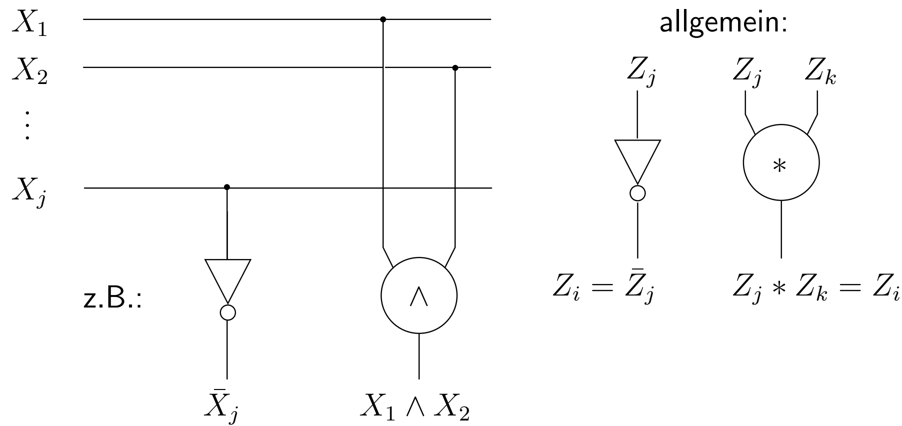


Abbildung 1.1: Konstruktionsvorschrift für Schaltkreise

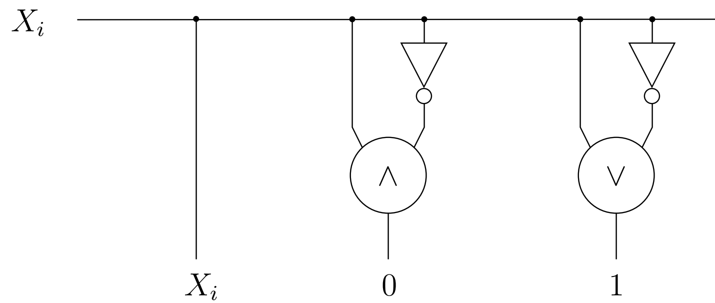


Abbildung 1.2: Schaltungen für den Induktionsanfang

Theorem 1.5 $\forall$Booleschen Ausdrücke e : $\exists$SLP/Schaltkreis $S(e) = (Z_1, \ldots, Z_n) : Z_n \equiv e$

Beweis: Sei $K(e)$ eine Funktion, die die Anzahl der logischen Operatoren ($\sim$,$\wedge$,$\vee$,$\oplus$) im Booleschen Ausdruck e, also die Komplexität des Ausdrucks, angibt. Dann erfolgt der Beweis per Induktion über $K(e)$.

Induktionsanfang: $K(e) = 0 \rightarrow e \in \{0, 1, X_i\}$

$$S(e) = \begin{cases} (\sim X_i, X_i \wedge \sim X_i) & : \quad e = 0 \\ (\sim X_i, X_i \vee \sim X_i) & : \quad e = 1 \\ (X_i) & : \quad e = X_i \end{cases}$$

Abbildung 1.2 zeigt die zugehörigen Schaltungen.

Induktionsannahme: Es exitiert ein SLP für jeden Ausdruck e' mit $K(e') = k$:

$$S(e') = (Z_1, \ldots, Z_k), \quad Z_k \equiv e'$$

Induktionsschritt: $K(e) \rightarrow K(e) + 1$ bzw. $k \rightarrow k + 1$

Fall 1: $e = \sim e', \quad K(e') = k$

$$\begin{aligned} S(e) &= (S(e'), Z_{k+1}) \\ &= (Z_1, \ldots, Z_k, Z_{k+1}) \quad \text{(Induktionsannahme)} \\ &= (Z_1, \ldots, Z_k, \sim Z_k) \; \Rightarrow \; \sim Z_k \overset{\text{(IA)}}{\equiv} \sim e' = e \quad ✓ \end{aligned}$$

Fall 2: $e = e' * e''$, $\quad * \in \{\wedge, \vee, \oplus\}$, $\quad K(e') + K(e'') = l + m = k$

$$\exists S(e') = (Z_1, \ldots, Z_l) \ : \ Z_l \equiv e', \qquad \exists S(e") = (Z'_1, \ldots, Z'_m) \ : \ Z'_m \equiv e'' \quad \text{(Induktionsannahme)}$$

$$S(e) = (S(e'), S(e''), Z_{k+1}) \quad = \quad (Z_1, \ldots, Z_l, Z'_1, \ldots, Z'_m, Z_l * Z'_m)$$

$$\Rightarrow \ Z_l * Z'_m \ \stackrel{\text{(IA)}}{\equiv} \ e' * e'' = e \ \checkmark \qquad \square$$

Für die verschiedenen Gatter in den Schaltkreisen existert eine Vielzahl von Darstellungen. Im folgenden sollen die Symbole gemäß Abbildung 1.3 verwendet werden.

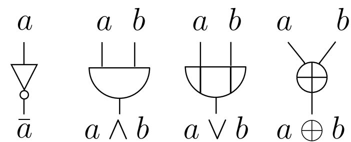


Abbildung 1.3: neue Symbole für logische Gatter

Kapitel 2

Arithmetic / Logic Units (ALUs)

2.1 Addition

2.1.1 Addierer

Definition 2.1 *Ein n-Addierer*

- *ist ein Schaltkreis.*
- *besitzt $2n+1$ Eingänge:*
  - $a \in \{0,1\}^n$ *und* $b \in \{0,1\}^n$ *(Operandeneingänge)*
  - $c_{in} \in \{0,1\}$ *(Eingangsübertrag / carry in)*
- *besitzt $n+1$ Ausgänge*
  - $s \in \{0,1\}^n$ *(Summenbits)*
  - $c_{out} \in \{0,1\}$ *(Ausgangsübertrag / carry out)*

Abbildung 2.1 zeigt die schematische Darstellung eines Addierers.

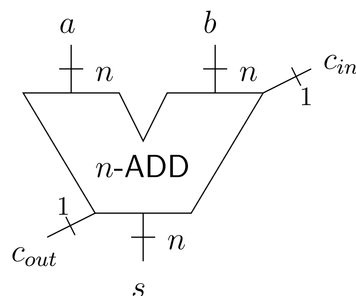


Abbildung 2.1: n-Addierer

Das gewünschte Verhalten „$a + b + c_{in} = s$“ lässt sich mit dem bisherigen Wissensstand nicht beschreiben. Zunächst muss auf Binärdarstellungen eingegangen werden.

2.1.2 Binärdarstellungen

Definition 2.2 *Ein Bitstring $a \in \{0,1\}^n$ lässt sich in der folgenden Weise als Binärzahl interpretieren:*

$$\langle a \rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

Beispiel:

$$\begin{aligned}\langle 101 \rangle &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 5\end{aligned}$$

Allgemein definiert man die Zahlendarstellung zur Basis B für Zahlen $a \in \{0, \ldots, B-1\}^n$

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

Beispiel:

$$\begin{aligned}\langle 101 \rangle_{10} &= 1 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0 \\ &= 101\end{aligned}$$

Wir identifizieren normalerweise Zahlen mit ihren Darstellungen zur Basis 10 (Anzahl der Zehen des Menschen).

Beispiel: Das folgende Beispiel verdeutlicht ein Lemma, das auch für Binärzahlen gilt:

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

Lemma 2.1 *(Zerlegungslemma) Sei $m < n$, dann gilt:*

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

Dabei wird die abkürzende Schreibweise

$$a = a_{n-1} \cdots a_0 \mathrel{\widehat{=}} a[n-1:0]$$

verwendet.
Der Beweis des Zerlegungslemmas erfolgt auf Übungsblatt 2.

2.1.3 Additionsalgorithmus

Ein verbreiteter Algorithmus zur Addition ist die Schulmethode. Wie Tabelle 2.1 zeigt, kann dieser auch auf Binärzahlen angewandt werden.

a		1	0	1	0	
b		0	1	1	1	
c	1	1	1	0	0	$\leftarrow c_{in}$
s	1	0	0	0	1	

Tabelle 2.1: Binäre Schulmethode für $c_{in} = 0$

Probe: $\langle 1010 \rangle = 10$ $\quad \langle 0111 \rangle = 7$ $\quad \langle 10001 \rangle = 17 = 10 + 7$ ✓

Definition 2.3 *(Additionsalgorithmus) Die formale Definition der Schulmethode lautet wie folgt:*

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

Für die Summenbits und den obersten Übertrag soll dann gelten:

$$\langle c_{out}\ s\rangle = \langle a\rangle + \langle b\rangle + c_{in} \quad (c_{out} = c_n)$$

Das i-te Summenbit s_i ergibt sich aus der Addition (modulo 2) der beiden Operandenbits a_i, b_i und dem Übertrag c_i des vorhergehenden Summenbits. Dabei entsteht ein weiterer Übertrag c_{i+1}. Ein Schaltkreis, der die o.g. Definitionen für ein Bit breite Signale erfüllt, heißt Volladdierer (Full Adder). Abbildung 2.2 zeigt die zugehörige schematische Darstellung.

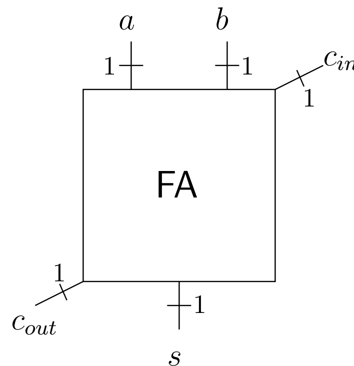


Abbildung 2.2: Full Adder

Aus dem Additionsalgorithmus ergibt sich direkt die Konstruktion des Carry Chain Adders (CCA), wie in Abbildung 2.3 dargestellt. 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.

Theorem 2.2 *Für $a[n-1:0]$, $b[n-1:0]$, $c[n:0]$ und $s[n-1:0]$ definiert nach dem Additionsalgorithmus gilt:*

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

Beweise: per Induktion über n

$$\begin{aligned} n=1 \quad : \quad \langle a_0\rangle + \langle b_0\rangle + c_0 &= a_0 \cdot 2^0 + b_0 \cdot 2^0 + c_0 \quad (\text{Def.}\langle\rangle) \\ &= a_0 + b_0 + c_0 \quad (\text{Potenzrechnung}) \\ &= \langle c_1\ s_0\rangle \quad (\text{Additionsalgorithmus}) \quad \checkmark \end{aligned}$$

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

Der Beweis zeigt, warum die Schulmethode funktioniert. Er benutzt Kommutivität und Distributivität der Addition, sowie Potenzrechnung. Letzteres wirkt insofern problematisch, als dass bisher Potenzrechnung auf der Basis der Multiplikation beziehungsweise der Addition definiert wurde. Dieser Widerspruch lässt sich durch eine separate Definition der Rechenoperationen vermeiden.

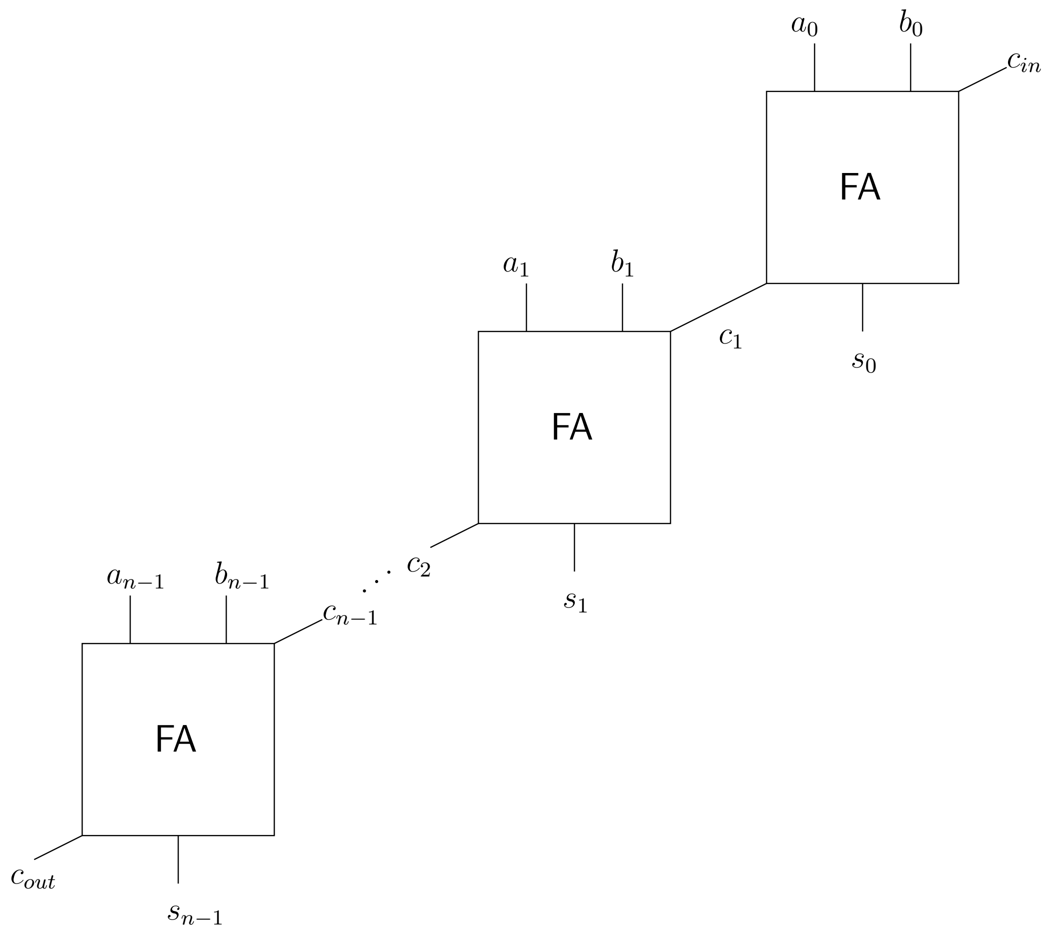


Abbildung 2.3: Carry Chain Adder

2.1.4 Formale Definition der Rechenoperationen

In den Peano-Axiomen ist die Nachfolgerfunktion $N(x)$ für natürliche Zahlen $x \in \mathbb{N}$ definiert. Sei der Nachfolger von 0 definiert als $N(0) \stackrel{\text{def}}{=} 1$. Dann gelten die folgenden rekursiven Definitionen für die verschieden Rechenarten:

Definition 2.4 $+$ *(Addieren)*

$$\begin{aligned} x + 0 &= 0 \\ x + N(y) &= N(x + y) \end{aligned}$$

Definition 2.5 $\cdot$ *(Multiplizieren)*

$$\begin{aligned} x \cdot 0 &= 0 \\ x \cdot N(y) &= x \cdot y + x \end{aligned}$$

Definition 2.6 $\uparrow$ *(Potenzieren)*

$$\begin{aligned} x^0 &= 1 \\ x^{N(y)} &= x^y \cdot x \end{aligned}$$

Hiermit lassen sich die entsprechenden Assoziatitivitäts-, Kommutativitäts- und Distributivitätsgesetze per Induktion beweisen (siehe Übungsblatt 2, Aufgabe 5).

2.1.5 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 2.7 *Kostenfunktion $C(S)$ (Cost)*

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

Die Kosten eines Schaltkreises entsprechen der Anzahl der enthaltenen Gatter.
In einem Schaltkreis S existieren Pfade von Gattern, wie in Abbildung 2.4 dargestellt, wobei ein

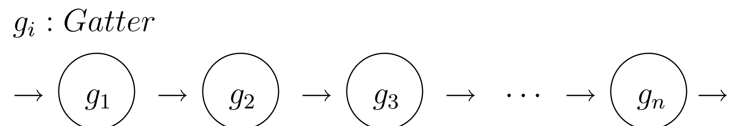


Abbildung 2.4: Pfad von Gattern in einem Schaltkreis

Ausgang von Gatter g_i an einen Eingang von Gatter g_{i+1} angeschlossen ist.

Definition 2.8 *Tiefenfunktion $D(S)$ (Depth)*

$$D(S) = \max\{\text{Länge von } P \mid P \text{ Pfad in } S\}$$

Die Tiefe eines Schaltkreises entspricht dem längsten enthaltenen Pfad. Wir legen fest, dass Schaltkreise keine Zyklen enthalten dürfen.
Beispiel: n-Bit Carry-Chain-Addierer $CCA(n)$

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

Da ein Eingangsübertragssignal unter Umständen die gesamte Volladdiererkette durchlaufen muss, ist die Konstruktion recht langsam. Es stellt sich die Frage, ob man den Weg des Carry nicht abkürzen kann, um die Tiefe des Addierers zu verringern. Dazu müssen wir zunächst ein neues Bauteil einführen.

2.1.6 Multiplexer

Multiplexer sind Signalweichen, mit denen ein Ausgangssignal in Abhängigkeit eines select-Signals auf eines von mehreren (hier: zwei) Eingangssignalen gesetzt wird.

Definition 2.9 *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 2.5 zeigt das zugehörige Schaltsymbol sowie die Implementierung eines 1-Bit MUX. Ein n-MUX wird mit n 1-Multiplexern implementiert, indem jedem Ausgangsbit ein eigener Multiplexer für die jeweiligen Eingangsbits zugeordnet wird (siehe Abbildung 2.6). Alle 1-MUXe werden mit demselben select-Signal s angesteuert.
Die Kosten des n-MUX betragen $C(n - MUX) = n \cdot C(1 - MUX)$. Wegen der Parallelschaltung bleibt die Tiefe bei $D(n - MUX) = 3$.

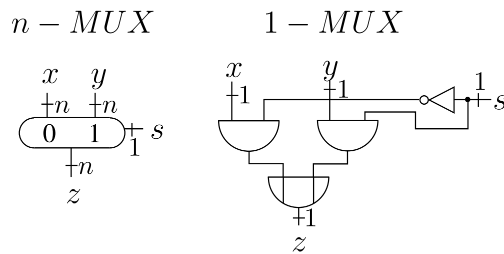


Abbildung 2.5: Multiplexer: Schaltsymbol und Implementierung

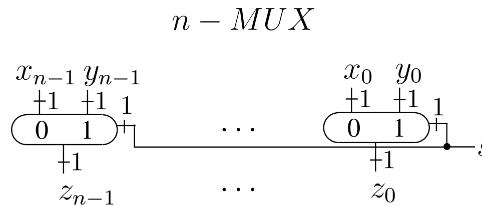


Abbildung 2.6: n-Multiplexer

2.1.7 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:

$$\begin{aligned} a_H &= a[n-1 : \frac{n}{2}] \\ a_L &= a[\frac{n}{2} - 1 : 0] \\ b_H &= b[n-1 : \frac{n}{2}] \\ b_L &= b[\frac{n}{2} - 1 : 0] \end{aligned}$$

Die rekursive Konstruktion eines n-Bit Conditional Sum Adders ($CSA(n)$) ist in Abbildung 2.7 dargestellt.

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

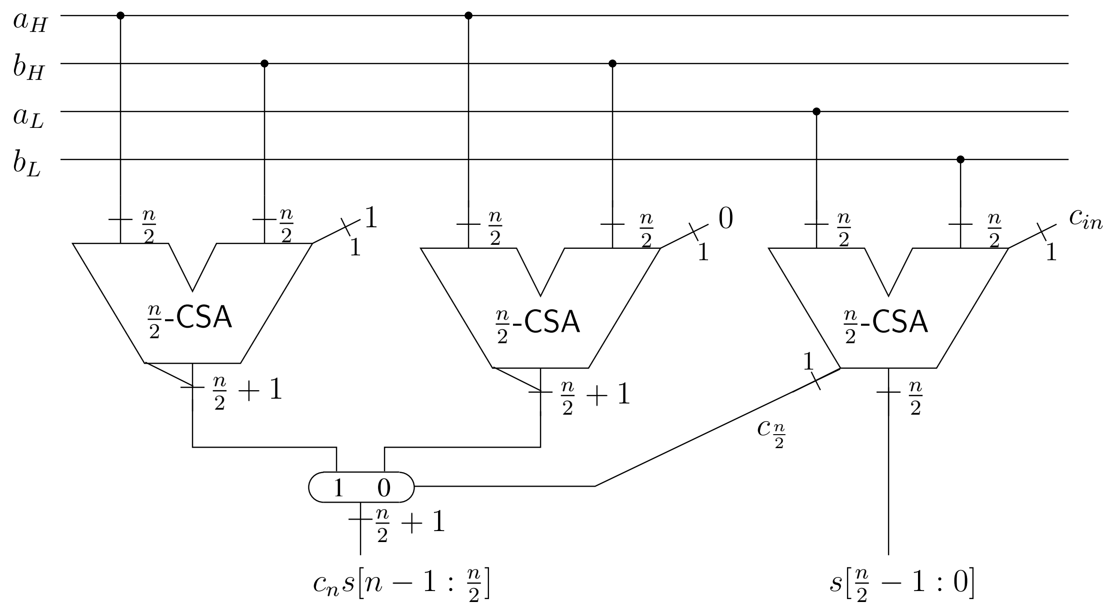


Abbildung 2.7: Conditional Sum Adder

mit einem Volladdierer realisiert.

$$\begin{aligned}
CSA(1) &= FA \\
D(CSA(1)) &= D(FA) \\
D(CSA(n)) &= D(CSA(\frac{n}{2})) + 3 \\
&= D(CSA(\frac{n}{4})) + 3 + 3 \\
&= D(CSA(\frac{n}{8})) + \underbrace{3+3+3}_{3\cdot 3} \\
&\vdots \\
&= D(CSA(\frac{n}{2^k})) + 3\cdot k \quad (n = 2^k \rightarrow k = \log_2 n) \\
&= D(CSA(1)) + 3\cdot \log_2 n \\
&= D(FA)) + 3\cdot \log_2 n \;\in\; \mathcal{O}(\log n)
\end{aligned}$$

Damit wäre der CSA schneller als der CCA, was durch höhere Kosten „erkauft“ wird. Allerdings ist die obige Behauptung $D(n) = 3\cdot\log_2 n + D(FA)$ nur eine Vermutung, die noch bewiesen werden muss.
Beweis: durch Induktion über n bzw. k mit $n = 2^k$

$$D(n) \stackrel{!}{=} 3\cdot\log_2 n + D(FA)$$

Induktionsanfang: $n = 1 = 2^0$

$$D(1) = D(FA) \stackrel{!}{=} 3\cdot\log_2 1 + D(FA) = 3\cdot 0 + D(FA) = D(FA) \quad ✓$$

Induktionsvoraussetzung: $D(\frac{n}{2}) = 3\cdot\log_2\frac{n}{2} + D(FA)$
Induktionsschritt: $\frac{n}{2} \rightarrow n$, bzw. $k-1 \rightarrow k$

$$\begin{aligned}
D(n) &= D(\frac{n}{2}) + \underbrace{D(MUX)} \quad (Konstruktion\ CSA) \\
&= D(\frac{n}{2}) + \quad\quad 3 \\
&= 3 \cdot \log_2 \frac{n}{2} + D(FA) + 3 \quad (Induktionsvorraussetzung) \\
&= 3 \cdot (\log_2 n - \log_2 2) + D(FA) + 3 \\
&= 3 \cdot \log_2 n - 3 + 3 + D(FA) \\
&= 3 \cdot \log_2 n + D(FA) \quad \square
\end{aligned}$$

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.

2.2 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.

2.2.1 Modulorechnung

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

Definition 2.10 *$a, b \in \mathbb{Z}$ heißen „$\mathrm{kongruent\ modulo}\ k \in \mathbb{N}$“, wenn $a \equiv b \mod k$, das heißt:*

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

Lemma 2.3 *$\equiv_{\mathrm{mod}\,k}$ ist eine Äquivalenzrelation, das bedeutet, es gelten die drei Eigenschaften:*

- *Reflexivität* $\quad \leftrightarrow \quad x \equiv_{\mathrm{mod}\,k} x$
- *Symmetrie* $\quad \leftrightarrow \quad x \equiv_{\mathrm{mod}\,k} y \;\Leftrightarrow\; y \equiv_{\mathrm{mod}\,k} x$
- *Transitivität* $\quad \leftrightarrow \quad (x \equiv_{\mathrm{mod}\,k} y \;\wedge\; y \equiv_{\mathrm{mod}\,k} z) \;\Rightarrow\; x \equiv_{\mathrm{mod}\,k} z$

Die Relation $\equiv_{\mathrm{mod}\,k}$ erzeugt k verschiedene Äquivalenzklassen.

Definition 2.11 *$x \bmod k \in \{0, \ldots, k-1\}$ ist ein Repräsentant der Äquivalenzklasse von x bezüglich $\equiv_{\mathrm{mod}\,k}$.*

Beispiel: $\equiv_{\mathrm{mod}\,3}$ erzeugt die folgenden Äquivalenzklassen:

$$\begin{aligned}
[0] &= \{0, 3, 6, \ldots\} \\
[1] &= \{1, 4, 7, \ldots\} \\
[2] &= \{2, 5, 8, \ldots\}
\end{aligned}$$

Im Gegensatz zur Kongruenz $a \equiv b \mod k$ bezeichnet $\beta = x \bmod k$ den Rest β der ganzzahligen Division von x mit k. Das folgende Korollar verdeutlicht den Zusammenhang zwischen beiden Notationen.
Korollar:

$$(x \equiv y \mod k \;\wedge\; x \in \{0, \ldots, k-1\}) \quad \Rightarrow \quad x = y \bmod k$$

Desweiteren existiert ein wichtiges Lemma für Binärzahlen.

Lemma 2.4 *Sei $a \in \{0,1\}^m$, $k = 2^n$ und $m > n$, dann gilt:*

$$\langle a[m-1:0]\rangle \bmod 2^n = \langle a[n-1:0]\rangle$$

Beweis:

$$\begin{aligned}
\langle a[m-1:0]\rangle &= \langle a[m-1:n]\rangle \cdot 2^n + \langle a[n-1:0]\rangle \quad (Zerlegungslemma\ 2.1)\\
\Leftrightarrow \langle a[m-1:0]\rangle - \langle a[n-1:0]\rangle &= \langle a[m-1:n]\rangle \cdot 2^n\\
\Leftrightarrow \langle a[m-1:0]\rangle &\equiv \underbrace{\langle a[n-1:0]\rangle}_{\leq 2^n-1} \quad \bmod 2^n \quad (Definition\ \equiv_{\bmod k})\\
\Leftrightarrow \langle a[m-1:0]\rangle \bmod 2^n &= \langle a[n-1:0]\rangle \quad (Korollar) \quad \square
\end{aligned}$$

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.

2.2.2 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 2.12 *(Subtraktionsalgorithmus) Für $x, y \in \{0,1\}^n$ und $\langle x\rangle \geq \langle y\rangle$ gilt:*

$$\langle x\rangle - \langle y\rangle = \langle x\rangle + \langle \bar{y}\rangle + 1 \bmod 2^n$$

Durch Negation des Subtrahenden und Addition von 1 soll also eine Subtraktion realisiert werden, wenn nur die letzten n Bits betrachtet. Warum dies stimmen soll, ist im ersten Moment nicht intuitiv klar, das Beispiel in Tabelle 2.2 für $n = 4$ lässt jedoch erahnen, dass der Algorithmus funktioniert. Um dies zu beweisen, muss man sich mit Two's-Complement-Zahlen beschäftigen.

$$\begin{array}{rcccc c rr c rccccc l}
 & 1 & 1 & 0 & 1 & \Leftrightarrow & & 13 & \Rightarrow & & & 1 & 1 & 0 & 1 & \\
- & 0 & 1 & 1 & 0 & \Leftrightarrow & - & 6 & \Rightarrow & + & & 1 & 0 & 0_1 & 1_1 & \\
\hline
 & 0 & 1 & 1 & 1 & \Leftrightarrow & & 7 & & & 1 & 0 & 1 & 1 & 1 & \bmod 2^4\\
 & & & & & & & & = & & & 0 & 1 & 1 & 1 & ✔
\end{array}$$

Tabelle 2.2: Beispiel zum Subtraktionsalgorithmus

2.2.3 Two's-Complement-Zahlen

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

Definition 2.13 *(Two's-Complement-Zahlen) Sei $x \in \{0,1\}^n$, dann heißt*

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

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

Definition 2.14 *$T_n = \{-2^{n-1}, \ldots, 2^{n-1}-1\}$ ist die Menge der mit n Bits darstellbaren Two's-Complement-Zahlen. Es gilt:*

$$-2^{n-1} \leq [x] \leq \langle x[n-2:0]\rangle \leq 2^{n-1} - 1$$

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

2.2.4 Korrektheitsbeweis

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 2.5 *(TWOC Lemma 1)*

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

Beweis:

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

Lemma 2.6 *(TWOC Lemma 2)*

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

Beweis:

$$\begin{aligned} [a] &= -2^{n-1} \cdot a_{n-1} + \langle a[n-2:0] \rangle \quad \text{(Definition [ ])} \\ \Leftrightarrow [a] - \langle a[n-2:0] \rangle &= -a_{n-1} \cdot 2^{n-1} \\ \Leftrightarrow [a] &\equiv \langle a[n-2:0] \rangle \mod 2^{n-1} \quad (\text{Definition } \equiv_{\mathrm{mod\,k}}) \quad \square \end{aligned}$$

Lemma 2.7 *(TWOC Lemma 3)*

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

Beweis:

$$\begin{aligned} [a] - \langle a \rangle &= -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle - (a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle) \quad \text{(Def. [ ] und Zerl.lemma 2.1)} \\ &= -2 \cdot a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle - \langle a[n-2:0] \rangle \\ &= -a_{n-1} \cdot 2^n \\ &\equiv 0 \mod 2^n \\ \Leftrightarrow [a] &\equiv \langle a \rangle \mod 2^n \quad \square \end{aligned}$$

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 2.8 *(TWOC Lemma 4) sign extension*

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

Beweis:

$$\begin{aligned} [a_{n-1}a] &= -a_{n-1} \cdot 2^n + \langle a \rangle \quad \text{(Definition [ ])} \\ &= -a_{n-1} \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle \quad (Zerlegungslemma\ 2.1) \\ &= (-2 \cdot a_{n-1} + a_{n-1}) \cdot 2^{n-1} + \langle a[n-2:0] \rangle \\ &= -a_{n-1}) \cdot 2^{n-1} + \langle a[n-2:0] \rangle \\ &= [a] \quad \text{(Definition [ ])} \quad \square \end{aligned}$$

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 2.9 *(TWOC Lemma 5)*

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

Beweis:

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

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 $a, b \in \{0,1\}^n$.
Beweis:

$$\begin{aligned}
\langle a \rangle - \langle b \rangle &= \langle a \rangle - [0b] \quad \text{(TWOC Lemma 1)} \\
&= \langle a \rangle + [1\bar{b}] + 1 \quad \text{(TWOC Lemma 5)} \\
&= \langle a \rangle - 2^n \cdot 1 + \langle \bar{b} \rangle + 1 \quad \text{(Definition [ ])} \\
&\equiv \langle a \rangle + \langle \bar{b} \rangle + 1 \mod 2^n \quad (\text{Definition } \equiv_{\text{mod k}}) \\
&\Rightarrow \langle a \rangle - \langle b \rangle \in \{0, \ldots, 2^n - 1\} \quad (\text{für } \langle a \rangle \geq \langle b \rangle) \\
\Rightarrow \langle a \rangle - \langle b \rangle &= \langle a \rangle + \langle \bar{b} \rangle + 1 \mod 2^n \quad \square
\end{aligned}$$

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.

2.3 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.

2.3.1 Arithmetic Unit

Mit einer AU lassen sich Binärzahlen addieren und subtrahieren.

Definition 2.15 *(Arithmetic Unit) In Abbildung 2.8 ist eine AU abgebildet. Für s gilt:*

$$\langle s \rangle = \begin{cases} \langle a \rangle + \langle b \rangle \mod 2^n & : \quad sub = 0 \\ \langle a \rangle - \langle b \rangle \mod 2^n & : \quad sub = 1 \end{cases}$$

$$[s] = \begin{cases} [a] + [b] \mod 2^n & : \quad sub = 0 \\ [a] - [b] \mod 2^n & : \quad sub = 1 \end{cases}$$

außerdem: $neg = 1 \quad \Leftrightarrow \quad [a] + (-1)^{sub} \cdot [b] < 0$

$ovf = 1 \quad \Leftrightarrow \quad [a] + (-1)^{sub} \cdot [b] \notin T_n = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$

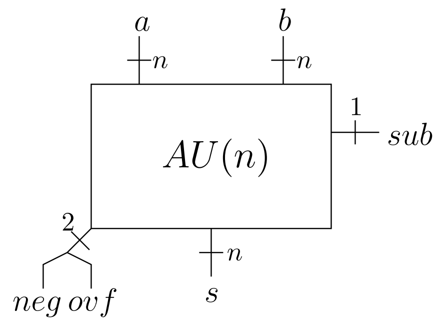


Abbildung 2.8: Arithmetic Unit

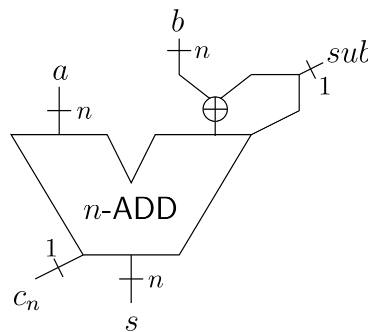


Abbildung 2.9: Implementierung der AU (ohne ovf, neg)

In Abbildung 2.9 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. Dabei werden zwei Eigenschaften des XORs verwendet die sich leicht mit Hilfe einer Wertetabelle beweisen lassen.

$$\begin{aligned} x \oplus 0 &= x \\ x \oplus 1 &= \bar{x} \end{aligned}$$

2.3.2 ovf und neg für Two's-Complement-Zahlen

Zur Berechnung des overflows lässt sich folgendes Lemma beweisen:

Lemma 2.10 *(TWOC overflow-Berechnung) Für den ovf-Ausgang der AU gilt bei der Berechnung mit Two's-Complement-Zahlen:*

1. $ovf = 1 \Leftrightarrow c_n \neq c_{n-1}$
2. $ovf = 0 \Leftrightarrow [s] = [a] + (-1)^{sub} \cdot [b]$

Im Falle dass kein overflow auftritt, erhalten wir also das korrekte Ergebnis der arithmetischen Operation. Ein overflow tritt immer genau dann auf, wenn die obersten beiden carry bits ungleich sind.

Beweis: Sei $b' = b \oplus sub$, dann gilt für das Ergebnis des Addierers:

$$\begin{aligned}
[a] + [b'] + c_{in} &= -a_{n-1} \cdot 2^{n-1} - b'_{n-1} \cdot 2^{n-1} + \underbrace{\langle a[n-2:0]\rangle + \langle b'[n-2:0]\rangle + c_{in}} \quad \text{(Definition [ ])} \\
&= -a_{n-1} \cdot 2^{n-1} - b'_{n-1} \cdot 2^{n-1} + \quad \langle c_{n-1}\ s[n-2:0]\rangle \quad \text{(Definition Addition)} \\
&= -a_{n-1} \cdot 2^{n-1} - b'_{n-1} \cdot 2^{n-1} + \underbrace{c_{n-1} \cdot 2^{n-1} - c_{n-1} \cdot 2^{n-1}}_{=0} + \langle c_{n-1}\ s[n-2:0]\rangle \quad ("+0") \\
&= -\underbrace{(a_{n-1} + b'_{n-1} + c_{n-1})} \cdot 2^{n-1} + 2 \cdot c_{n-1} \cdot 2^{n-1} + \langle s[n-2:0]\rangle \quad \text{(Zerl.lemma 2.1)} \\
&= - \quad \langle c_n\ s_{n-1}\rangle \cdot 2^{n-1} + 2 \cdot c_{n-1} \cdot 2^{n-1} + \langle s[n-2:0]\rangle \quad \text{(Definition Addition)} \\
&= -2 \cdot c_n \cdot 2^{n-1} - s_{n-1} \cdot 2^{n-1} + 2 \cdot c_{n-1} \cdot 2^{n-1} + \langle s[n-2:0]\rangle \quad \text{(Zerl.lemma 2.1)} \\
&= -c_n \cdot 2^n + c_{n-1} \cdot 2^n + [s[n-1:0]] \quad \text{(Definition [ ])}
\end{aligned}$$

1. Fall $c_n = c_{n-1}$

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

Hieraus folgt Teil 2. unter der Annahme das Teil 1. gilt, also $ovf = 0 \Leftrightarrow c_n = c_{n-1}$.
2. Fall $c_n \neq c_{n-1}$, es gilt, dass $s \in T_n = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$:

1. $c_n = 1 \wedge c_{n-1} = 0$

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

2. $c_n = 0 \wedge c_{n-1} = 1$

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

Aus 1. und 2. folgt:

$$\begin{aligned}
c_n \neq c_{n-1} \quad &\Leftrightarrow \quad [a] + [b'] + c_{in} \notin T_n \\
&\Leftrightarrow \quad [a] + (-1)^{sub} \cdot [b] \notin T_n \quad \text{(Subtraktionsalgorithmus, Konstruktion AU)} \\
&\Leftrightarrow \quad ovf = 1 \quad \text{(Definition AU)} \quad \square
\end{aligned}$$

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. Dies kann man mit Hilfe der Identität

$$c_{n-1} \equiv a_{n-1} \oplus b'_{n-1} \oplus s_{n-1}$$

beheben, wobei hier die Definition $s_{n-1} = a_{n-1} \oplus b'_{n-1} \oplus c_{n-1}$ und Eigenschaften des XORs verwendet werden. Abbildung 2.10 zeigt die Implementierung der ovf-Berechnung.

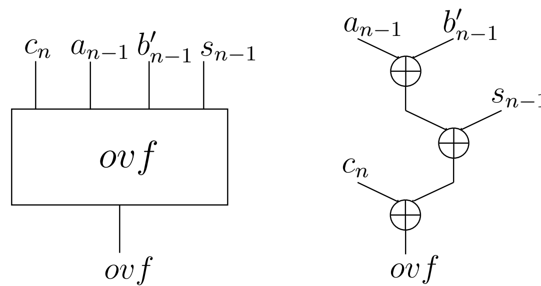


Abbildung 2.10: ovf-Berechnung

Das neg-Bit der AU signalisiert, dass das Ergebnis einer arithmetischen Operation negativ ist. Dies wird für Vergleichsoperationen benötigt, da

$$a \gtrless b \leftrightarrow a - b \gtrless 0$$

gilt.
Im Falle von Two's-Complement-Zahlen ist ein Ergebnis genau dann negativ, wenn das oberste Bit (*sign bit*) 1 ist. Es kann jedoch ein overflow auftreten, woraufhin n Bits nicht mehr ausreichen, das korrekte Ergebnis darzustellen. Deshalb kann man nicht einfach s_{n-1} als neg-Bit wählen.

Lemma 2.11 *(TWOC neg-Berechnung) Für den neg-Ausgang der AU gilt bei der Berechnung mit Two's-Complement-Zahlen:*

$$neg = c_n \oplus a_{n-1} \oplus b'_{n-1}$$

Beweisidee: Das neg-Bit ist das sign bit res_n des exakten Ergebnisses

$$res = [a_{n-1}a] - [b'_{n-1}b'] \in T_{n+1}.$$

Man beachte die *sign extension* der beiden Operanden a und b.

Ein Schaltkreis zur Berechnung des neg-Bits soll in Aufgabe 2 des Übungsblattes 3 konstruiert werden.

2.3.3 neg und ovf für Binärzahlen

Soll die AU nur mit positiven Binärzahlen (*unsigned integers*) rechnen, so vereinfacht sich die Realisierung von neg und ovf. Ein Ergebnis kann nur noch bei einer Subtraktion negativ werden und zwar genau dann, wenn der zweite Operand größer als der erste ist.

Lemma 2.12 *(unsigned neg-berechnung) Für einen unsigned neg-Ausgang einer AU gilt:*

$$\begin{aligned} \langle a \rangle - \langle b \rangle < 0 \quad &\Leftrightarrow \quad c_n = 0 \\ &\Rightarrow \quad neg = \bar{c}_n \end{aligned}$$

Beweisidee: Bei der Subtraktion zweier positiver Binärzahlen $a, b \in \{0,1\}^n$ gilt $\forall i \in \{1, \ldots, n\}$:

(1) $a_{i-1} > b_{i-1} \quad \Rightarrow \quad c_i = 1$
(2) $a_{i-1} < b_{i-1} \quad \Rightarrow \quad c_i = 0$
(3) $a_{i-1} = b_{i-1} \quad \Rightarrow \quad c_i = c_{i-1}$

Dies wird auch von Tabelle 2.3 verdeutlicht.
Wenn $a_{i-1} = b_{i-1}$ ist, wird der carry propagiert (weitergeleitet), ansonsten wird ein neuer carry generiert (erzeugt) falls $a_{i-1} > b_{n-1}$. Der Beweis des Lemmas folgt dann per Induktion über n unter Ausnutzung dieser Erkenntnisse.

Die Definition des ovf-Bits für Binärzahlen ist trivial. Ein Overflow entsteht dann, wenn das Ergebnis einer arithmetischen Operation den darstellbaren Zahlenbereich verlässt. Das heißt in diesem Fall, dass

a_{i-1}	b_{i-1}	$\bar{b}_{i-1}$	c_i	
0	0	1	c_{i-1}	$\leftarrow$ (3)
0	1	0	0	$\leftarrow$ (2)
1	0	1	1	$\leftarrow$ (1)
1	1	0	c_{i-1}	$\leftarrow$ (3)

Tabelle 2.3: Beweisidee zur neg-Berechnung

- die Summe zweier Binärzahlen zu groß wird ($\langle a \rangle + \langle b \rangle \geq 2^n$) oder
- die Differenz zweier Binärzahlen negativ wird ($\langle a \rangle - \langle b \rangle < 0$).

Lemma 2.13 *(unsigned ovf-Berechnung) Für einen unsigned ovf-Ausgang einer AU gilt:*

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

2.3.4 Konstruktion der ALU

Eine ALU berechnet arithmetische und logische Funktionen für zwei Operanden $a, b \in \{0,1\}^n$. Das Ergebnis liegt am Ausgang s an. Welche Funktion von der ALU ausgeführt werden soll, wird über den *function code* $f \in \{0,1\}^4$ gesteuert. Abbildung 2.11 zeigt die schematische Darstellung einer ALU.

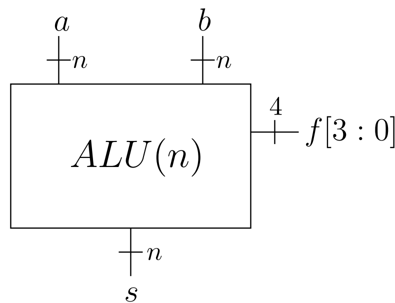


Abbildung 2.11: Arithmetic / Logic Unit

In Tabelle 2.4 sind die zum Funktionscode zugehörigen Funktionen aufgeführt. Zeilen 1-8 sind die arithmetischen, Zeilen 9-16 die logischen Funktionen. Die hier verwendete AU erzeugt ovf- und neg-Signale für TWOC-Zahlen.
In Abbildung 2.12 ist die Implementierung der ALU dargestellt. Sie besteht aus drei Blöcken:

- der Arithmetic Unit zur Berechnung von Addition und Subtraktion (sub gesteuert über f),
- den Booleschen Funktionen inklusive des logischen Linksshifts um $\frac{n}{2}$ Bits
- und die Logic Unit (LU), die Vergleichsoperationen realisiert.

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:

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

$f[3:0]$	s	ovf
0000	$a +_u b$	ovf
0001	$a +_u b$	0
0010	$a -_u b$	ovf
0011	$a -_u b$	0
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	0
1001	$a > b$	0
1010	$a = b$	0
1011	$a \geq b$	0
1100	$a < b$	0
1101	$a \neq b$	0
1110	$a \leq b$	0
1111	1	0

Tabelle 2.4: Funktionstabelle

Die LU wird wie in Abbildung 2.13 gezeigt implementiert. Es ergibt sich der *condition code*

$$x_{cc} = ((\overline{\bigvee_i a_i \oplus b_i}) \wedge f_1) \vee (neg \wedge f_2) \vee ((\bigvee_i a_i \oplus b_i) \wedge \overline{neg} \wedge f_0).$$

Dabei steht der erste Term für $a \stackrel{?}{=} b$, der zweite Term für $a \stackrel{?}{<} b$ und der dritte Term für $a \stackrel{?}{>} b$. Die einzelnen Test werden über die function code Bits kombiniert. Der erste Term $\overline{\bigvee_i a_i \oplus b_i}$ beschreibt einen Gleichheitstester. Dieser wird mit einem bitweisen XOR, einem ORTREE (Baum aus OR-Gattern) und einem Inverter realisiert. Es gilt:

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

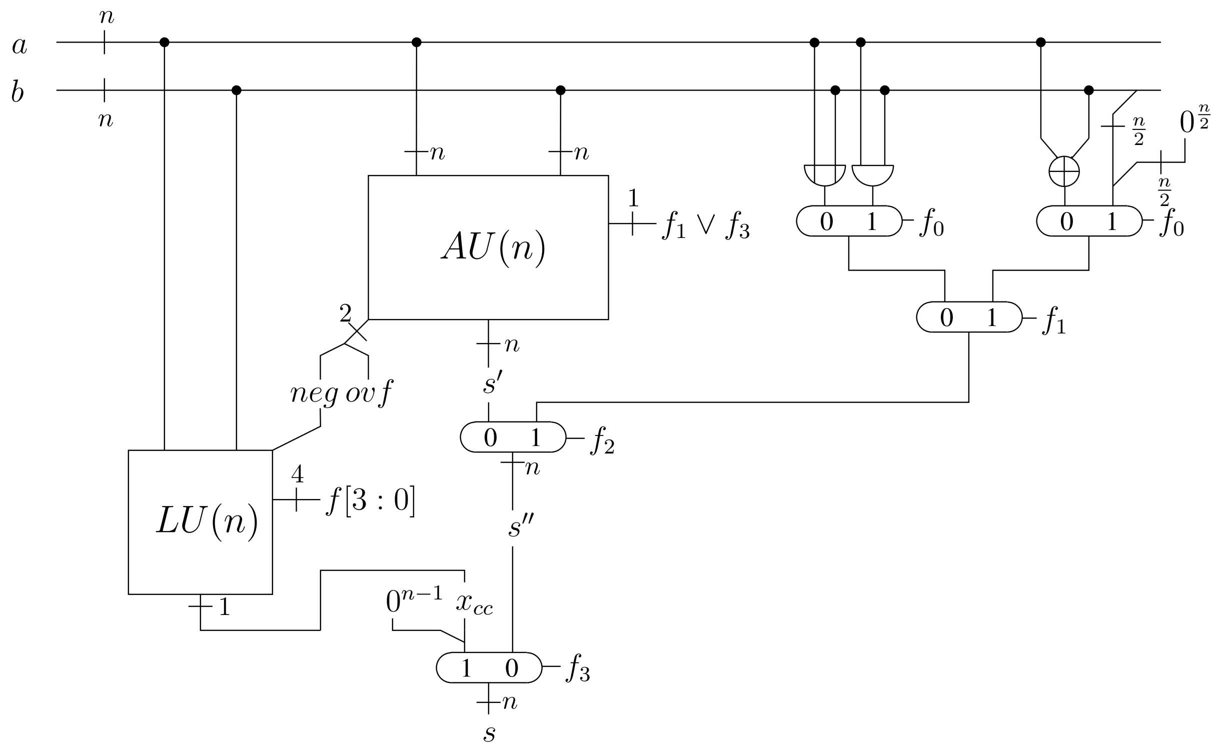


Abbildung 2.12: Implementierung der ALU

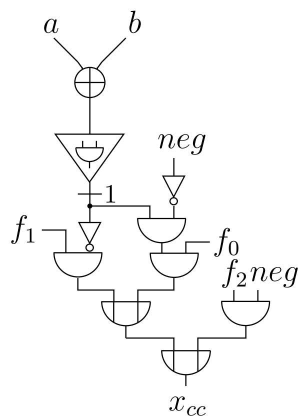


Abbildung 2.13: Logic Unit

Kapitel 3

Prozessoren

Nun soll ein einfacher DLX-Prozessor konstruiert werden. Der vollständige Instruktionssatz und die Datenpfade der DLX finden sich auf der Vorlesungswebsite. Im Folgenden werden die Spezifikation und eine Hardware-Implementierung betrachtet und der Beweis geführt, dass das eine System das andere simuliert.

3.1 Spezifikation

Die Spezifikation eines Prozessors beinhaltet:

- für den Benutzer sichtbare Datenstrukturen
- ausführbare Operationen, insbesondere deren
  - Format / Syntax
  - Effekt / Semantik

Unser einfacher DLX-Prozessor (**M**icroprocessor without **i**nterlocked **p**ipeline **s**tages - **MIPS**) 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 sich Konfigurationen $c = (c.pc, c.gpr, c.m)$, die einen temporären Zustand der Maschine beschreiben. Für die Komponenten gilt:

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

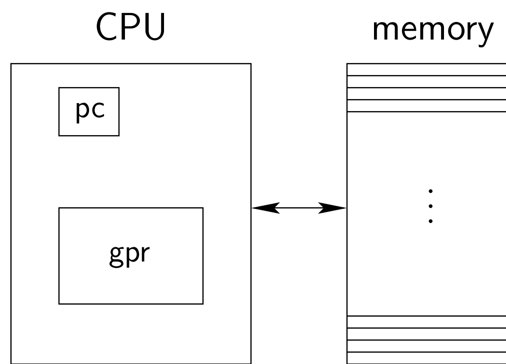


Abbildung 3.1: Grober Aufbau der DLX

- 32 Register, jedes 32 Bit breit
- $c.gpr(x)$ beschreibt den Inhalt von Register x

Für den Speicherzugriff verwenden wir folgende Notation:

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

wobei $u_{32} \mathrel{\widehat{=}} bin_{32}(u)$ der 32 Bit Binärdarstellung von $u \in \{0, \ldots, 2^{32} - 1\}$ entspricht. Abbildung 3.2 zeigt die addressierten d Bytes im Speicher.

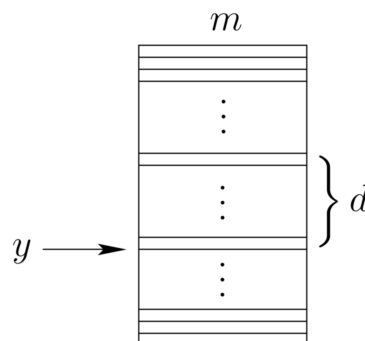


Abbildung 3.2: Adressierung von d Bytes im Speicher

3.1.1 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. 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 $opc(c) = I(c)[31 : 26]$, von entscheidender Bedeutung. Man definiert Prädikate, die den jeweiligen Typ der Instruktion $I(c)$ anzeigen.

- $Rtype(c) \leftrightarrow opc(c) = 0^6$
- $Jtype(c) \leftrightarrow opc(c) \in \{000010, 000011, 111110, 111111\}$
- $Itype(c) \leftrightarrow \text{sonst}$

Abbildung 3.3 stellt die Struktur der Instruktionenstypen dar und definiert die Bedeutung der einzelnen Abschnitte innerhalb einer Instruktion. Diese lassen sich auch durch folgende Funktionen beschreiben:

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

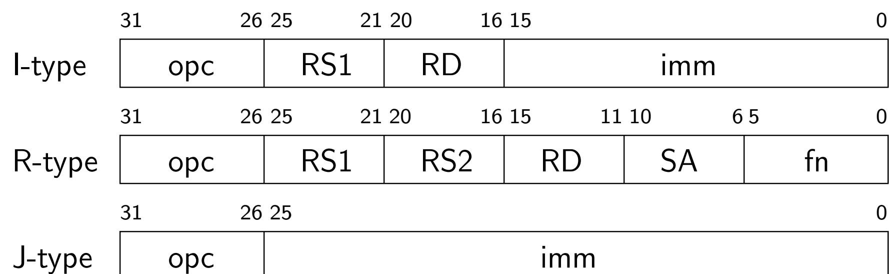


Abbildung 3.3: Typen von Instruktionen

Die immediate-Konstante muss zur weiteren Berechnung zur 32-Bit Konstannte $sxtimm$ 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.

3.1.2 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.

3.1.3 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 für $RD(c) \neq 0^5$ 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:

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

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.

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

3.1.4 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:

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

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:

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

Dazu kommt der Funktionscode für die ALU:

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

Abbildung 3.4 stellt eine ALU-Operation schematisch dar.

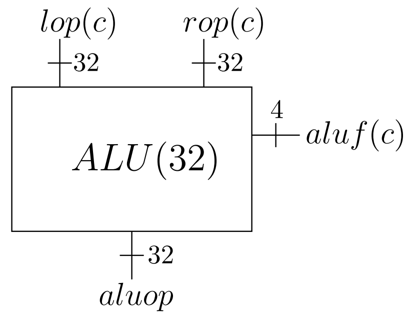


Abbildung 3.4: Verwendung der ALU durch ALU-Instruktion

Deren Semantik lautet für $RD(c) \neq 0^5$:

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

3.1.5 Kontrollinstruktionen

Die verbleibenden Instruktionen sind Kontrollinstruktionen, also branches und jumps. Diese ermöglichen die Kontrolle des Programmflusses, indem sie den pc manipulieren. Im Falle, dass 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 (genommen) wird:

$$\begin{aligned} bjtaken(c) &= btaken(c) \vee jump(c) \\ btaken(c) &= beqz(c) \wedge (c.gpr(RS1(c)) = 0^{32}) \vee bnez(c) \wedge (c.gpr(RS1(c)) \neq 0^{32}) \\ jump(c) &= j(c) \vee jal(c) \vee jr(c) \vee jalr(c) \\ branch(c) &= beqz(c) \vee bnez(c) \end{aligned}$$

Das Sprungziel wird dann je nach Instruktion per Addition der immediate-Konstante zum pc oder per Zuweisung eines Registerinhalts bestimmt:

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

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

In der Vorlesung wurde die Reihenfolge der Parameter 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.

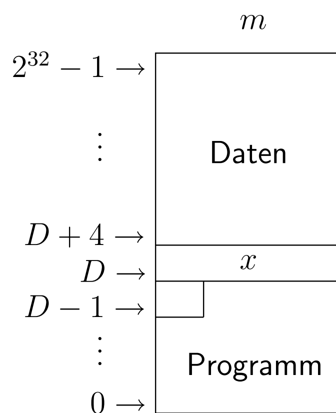


Abbildung 3.5: Speicheraufteilung (memory map) zwischen Daten und Instruktionen

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.

```
0: lw   1 0 D      // ea=GPR(R0)+imm
4: addi 1 1 1      // GPR(R1)=GPR(R1)+1
8: 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 $y = m_4(x)$.

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

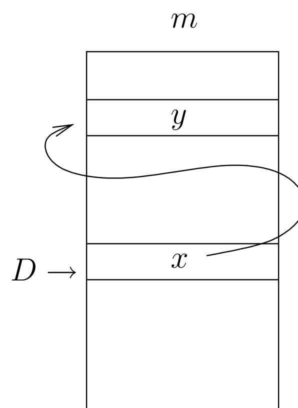


Abbildung 3.6: Erläuterung zum zweiten Beispielprogramm

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 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.

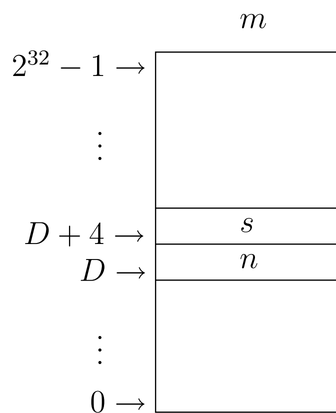


Abbildung 3.7: Speicherbelegung des Beispielprogramms

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:

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

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

```
 0: lw   1 0 D
 4: sw   1 0 D+4
 8: lw   1 0 D
12: beqz 1   28  // goto *
16: subi 1 1 1
20: sw   1 0 D
24: lw   2 0 D+4
28: add  3 1 2
32: sw   3 0 D+4
36: j        -28 // loop
40: ...          // *
```

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 Hardware

Nun soll eine Hardware-Implementierung entworfen werden, die die vorgegebene Spezifikation erfüllt. Hardware besteht aus verschiedenen Komponenten:

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

Der vollständige Datenpfad der vereinfachten DLX findet sich auf der Vorlesungswebseite. Im Folgenden sollen nun die einzelnen Komponenten betrachtet werden. Der aktuelle Zustand der Hardware und ihrer Bestandteile wir durch eine Konfiguration beschrieben in der Form:

$$h = \langle \ldots, h.R, \ldots, h.S \rangle$$

Dabei beschreibt

- $h.R$ den aktuellen Inhalt von Register R
- $h.S$ den aktuellen Inhalt von Speicher S
- $h' = \delta_H(h)$ den neuen Zustand der Hardware nach einem Takt

$\delta_H(h)$ ist die Übergangsfunktion auf Hardwareseite, die der Spezifikationsvariante $\delta_D(c)$ entspricht.

3.3.1 Register

Abbildung 3.8 zeigt ein n-Bit Register. Dies ist ein getakteter Schaltkreis, der Werte speichern kann.

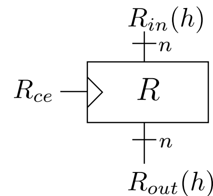


Abbildung 3.8: n-Bit Register

Es gilt für $h.R \in \{0,1\}^n$:

$$\begin{aligned} R_{out}(h) &= h.R \\ h'.R &= \begin{cases} R_{in}(h) & : \quad R_{ce}(h) = 1 \\ h.R & : \quad \text{sonst} \end{cases} \end{aligned}$$

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. Als Beispiel zur Verwendung von Registern soll jetzt das EX-Register der Hardware betrachtet werden. Es speichert, ob sich die DLX im *execute*- oder im *fetch*-Modus befindet.
Beispiel: Abbildung 3.9 stellt die Implementierung der Komponente $h.ex$ dar. Darin wird das $reset$-

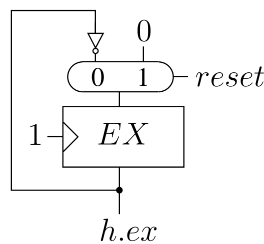


Abbildung 3.9: Implementierung von $h.ex$

Signal verwendet. Im Folgenden gehen wir davon aus, dass das System zu Beginn zurückgesetzt wird, und $reset$ danach inaktiv bleibt.

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

Der einzige Reset findet also vor h^0 statt. Daher wird EX mit 0 initialisiert. In allen folgenden Takten wird der Inhalt von ex fortwährend invertiert.

$$\begin{aligned} EX_{in}(h^{-1}) &= 0 \\ h^0.ex &= 0 \\ EX_{in}(h^0) &= \overline{h^0.ex} = 1 \\ h^1.ex &= 1 \\ &\vdots \\ h^i.ex &= \begin{cases} 0 & : & i \text{ gerade} \\ 1 & : & i \text{ ungerade} \end{cases} \end{aligned}$$

3.3.2 Speicher

In Speichern können im Gegensatz zu einfachen Registern mehrere Werte gespeichert werden. Deshalb werden auch Adress-Signale benötigt um verschiedene Speicherbereiche auszuwählen. Abbildung 3.10 zeigt einen gewöhnlichen RAM S. Er verfügt über die folgenden Schnittstellen:

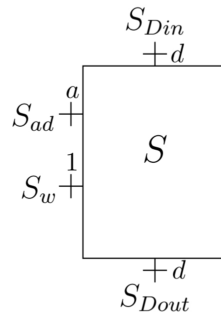


Abbildung 3.10: RAM

- 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. Die Funktionsweise ist für ein $x \in \{0,1\}^a$ folgendermaßen definiert:

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

Als Registerbank der DLX benötigen wir einen *3-Port-RAM*, wie in Abbildung 3.11 dargestellt. Dieser hat im Unterschied zu einem normalen RAM drei Adresseingänge und zwei Datenausgänge und wird wie folgt definiert:

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

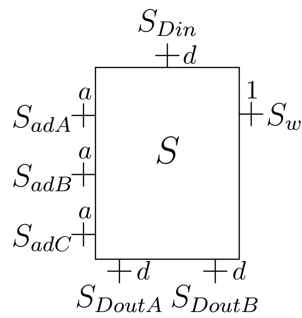


Abbildung 3.11: 3-Port-RAM

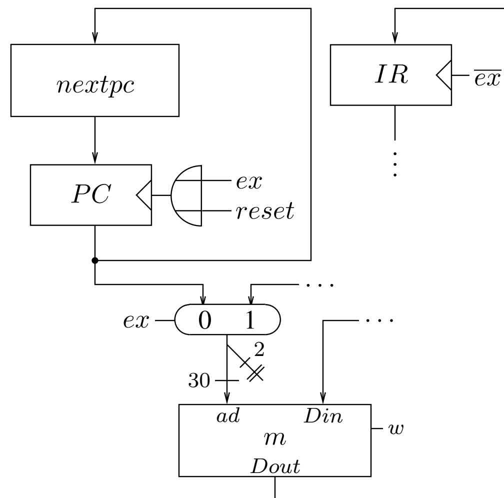


Abbildung 3.12: Implementierung des fetch-Modus

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

3.3.3 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. Abbildung 3.12 zeigt den relevanten Ausschnitt aus dem Datenpfad der DLX. Es gilt:

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

3.4 Korrektheit

Nachdem nun die Spezifikation und Implementierung der DLX angegeben wurde, soll im Folgenden bewiesen werden, dass beide einander entsprechen. Dies geschieht mit Hilfe eines Simulationsbeweises, der die Zustände beider Maschinen betrachtet und argumentiert, dass sie nach jedem Ausführungsschritt äquivalent sind. Zur besseren Übersicht sollen die Definitionen von Spezifikation und Hardware hier

nochmals aufgelistet werden. Für die Spezifikation gilt:

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

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

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

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

$$c'.gpr(x) = \begin{cases} 0^{32} & : x = 0^5 \\ c.m_4(ea(c)) & : lw(c) \wedge (x = RD(c)) \wedge (x \neq 0) \\ aluop(lop(c), rop(c), aluf(c)) & : (comp(c) \vee compimm(c)) \\ & \quad \wedge (x = RD(c)) \wedge (x \neq 0) \\ c.pc +_{32} 4_{32} & : (jal(c) \vee jalr(c)) \wedge (x = 1^5) \\ c.gpr(x) & : sonst \end{cases} \tag{3.5}$$

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. 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 Hardwarekomponenten zu einem bestimmten Zeitpunkt. Dabei ist:*

- $h = (h.ex, h.pc, h.IR, h.gpr, h.m)$ - *Hardwarekonfiguration*
- $h.ex \in \{0,1\}$ - *1-Bit Register für aktuellen Modus (fetch/execute, siehe Abbildung 3.9)*
- $h.pc \in \{0,1\}^{32}$ - *32-Bit Register für den Programm Counter (siehe Abbildung 3.12)*
- $h.IR \in \{0,1\}^{32}$ - *32-Bit Register für die aktuelle Instruktion*
- $h.gpr : \{0,1\}^5 \longrightarrow \{0,1\}^{32}$ - *Multiport Register File (siehe Abbildung 3.11)*
- $h.m : \{0,1\}^{30} \longrightarrow \{0,1\}^{32}$ - *Speicher (RAM, siehe Abbildung 3.10)*

Man beachte, dass der Speicher $h.m$ der Hardware im Gegensatz zu $c.m$ der DLX *word-adressed* ist. Das bedeutet, dass nur ganze Wörter aus dem Speicher gelesen werden können. Abbildung 3.13 zeigt die Aufteilung des memory.

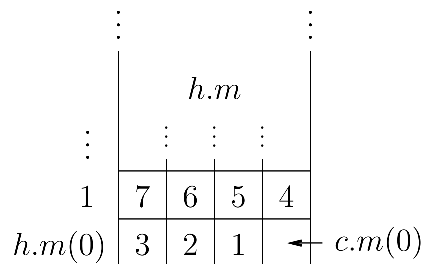


Abbildung 3.13: wortadressierter Speicher der Hardware

3.4.1 Simulationsrelation und Simulationssatz

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:*

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

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.*

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

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 3.14). Dann kann die Konsistenz der Speicherinhalte gesichert werden. Im Folgenden gehen wir davon aus, dass die Startbedingungen für den Simulationssatz erfüllt sind.

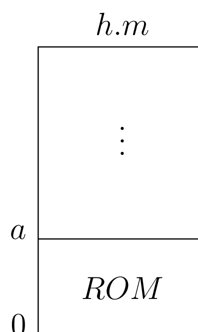


Abbildung 3.14: ROM-Bereich im Hardware-Speicher

3.4.2 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. Beim Schritt von $i \rightarrow i+1$ gilt die

Induktionsvorraussetzung.
IV: Es gilt $sim(c^i, h^{2i})$, das heißt:

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

Nun muss gezeigt werden, dass pc, gpr und m für alle möglichen Schritte der DLX konsistent bleiben, 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 3.12.

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:

$$\begin{aligned} m_{ad}(h^{2i}) &= h^{2i}.pc[31:2] \quad (\text{MUX, } h^{2i}.ex = 0) \\ &= c^i.pc[31:2] \quad (sim(c^i, h^{2i})) \\ m_{Dout}(h^{2i}) &= h^{2i}.m(m_{ad}(h^{2i})) \quad (\text{Spezifikation RAM}) \\ &= h^{2i}.m(c^i.pc[31:2]) \\ &= c^i.m(c^i.pc[31:2]00) \quad (sim(c^i, h^{2i})) \\ &= c^i.m(c^i.pc) \quad (\text{alignment}) \\ &= I(c^i) \\ h^{2i+1}.IR &= m_{Dout}(h^{2i}) \quad (IR_{ce}(h^{2i}) = \overline{h^{2i}.ex} = 1) \\ &= I(c^i) \quad \square \end{aligned}$$

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 3.15 dargestellt ist. Er dekodiert Prädikate, Funktionen und berechnet die Adresse des Zielregisters adC.
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 wahr sind.*

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

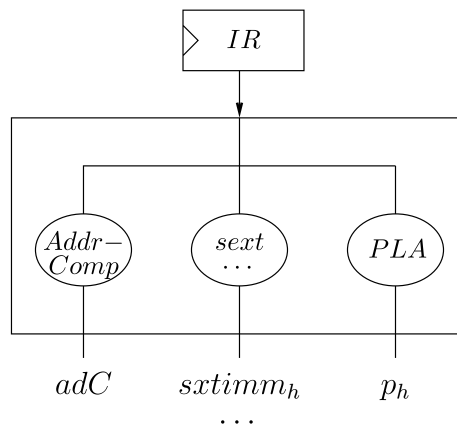


Abbildung 3.15: Instruktionsdekodierer

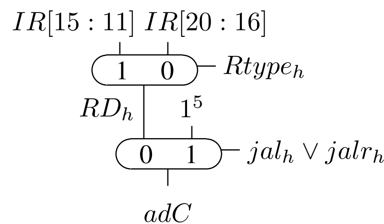


Abbildung 3.16: Schaltkreis Addr-Comp

Ebenso 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 besitzen den gleichen Wert wie ihre zugehörigen Funktionen in der DLX.*

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

Der Beweis muss separat für jede spezielle Funktion erbracht werden.
Beweis für $f = RS1$:

$$\begin{aligned} RS1(h^{2i+1}) &= h^{2i+1}.IR[25:21] \\ &= I(c^i)[25:11] \\ &= RS1(c^i) \end{aligned}$$

Dabei wird stehts Lemma 3.3 benutzt. Der Beweis von $sxtimm(h^{2i+1}) = sxtimm(c^i)$ erfolgt in Aufgabe 4 des fünften Übungsblattes.

Der Dekodierer bestimmt auch aus der Instruktion welches GPR-Register adC geschrieben werden soll. Abbildung 3.16 zeigt den zugehörigen Schaltkreis $Addr - Comp$.

Lemma 3.6 *Das GPR-Zielregister in der Hardware entspricht dem der DLX:*

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

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

Nun wollen wir ALU-Operationen betrachten. Abbildung 3.17 zeigt den relevanten Ausschnitt des

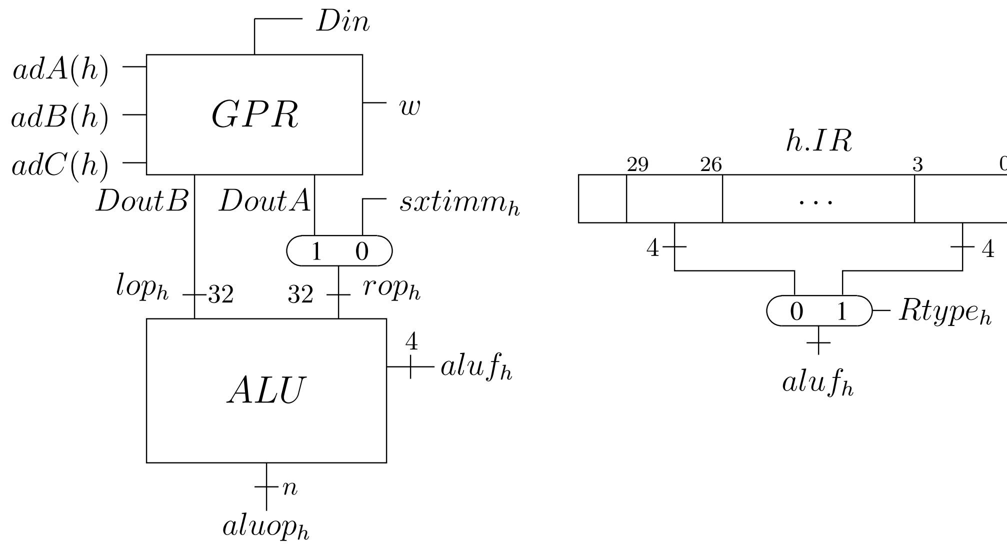


Abbildung 3.17: ALU-Instruktionen

Datenpfades der DLX sowie die Implementierung von $aluf_h(h)$. Es gilt nach Konstruktion von $aluf_h(h)$ und Definition der DLX:

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

$$aluf(c^i) = \begin{cases} I(c^i)[29:26] & : \overline{Rtype(c^i)} \\ I(c^i)[3:0] & : Rtype(c^i) \end{cases}$$

Lemma 3.7 *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:

$$\begin{aligned} adA(h) &= h.IR[25:21] \\ adB(h) &= h.IR[20:16] \end{aligned}$$

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 $RS2$ 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 3.8

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

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 3.9

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

Beweis:

$$\begin{aligned} A(h^{2i+1}) &= h^{2i+1}.gpr_{DoutA} \quad \text{(Konstruktion)} \\ &= h^{2i+1}.gpr(\underbrace{adA(h^{2i+1})}_{RS1(c^i)}) \quad \text{(Register File Spezifikation)} \\ &= c^i.gpr(RS1(c^i)) \quad \text{(Lemma 3.8)} \quad \checkmark \\ B(h^{2i+1}) &= h^{2i+1}.gpr_{DoutB} \quad \text{(Konstruktion)} \\ &= h^{2i+1}.gpr(\underbrace{adB(h^{2i+1})}_{RS2(c^i)}) \quad \text{(Register File Spezifikation)} \\ &= c^i.gpr(RS2(c^i)) \quad \text{(Lemma 3.8)} \quad \checkmark \quad \square \end{aligned}$$

Der linke Operand der ALU ergibt sich dann zu:

$$\begin{aligned} lop_h(h^{2i+1}) &= A(h^{2i+1}) \quad \text{(Konstruktion)} \\ &= c^i.gpr(RS1(c^i)) \quad \text{(Lemma 3.9)} \\ &= lop(c^i) \end{aligned}$$

Für den rechten Operanden erhalten wir:

$$\begin{aligned} rop_h(h^{2i+1}) &= \begin{cases} B(h^{2i+1}) & : Rtype_h(h^{2i+1}) \\ sxtimm_h(h^{2i+1}) & : \text{sonst} \end{cases} \quad \text{(Konstruktion, MUX)} \\ &= \begin{cases} B(h^{2i+1}) & : Rtype(c^i) \\ sxtimm(c^i) & : \text{sonst} \end{cases} \quad \text{(Lemma 3.4, 3.5)} \\ &= \begin{cases} c^i.gpr(RS2(c^i)) & : Rtype(c^i) \\ sxtimm(c^i) & : \text{sonst} \end{cases} \quad \text{(Lemma 3.9)} \\ &= rop(c^i) \end{aligned}$$

Zusammendfassend gilt also

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

und somit durch die Korrektheit der ALU:

Lemma 3.10 *(aluop) Das Ergebnis der ALU-Operationen auf beiden Maschinen ist identisch.*

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

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

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

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

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

Lemma 3.11

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

Nach dem fetch sind die Speicherinhalte der Hardware identisch zur DLX. Nun soll ein Wert aus Speicher gelesen werden. Abbildung 3.18 zeigt die relevanten Datenpfade. Es gilt:

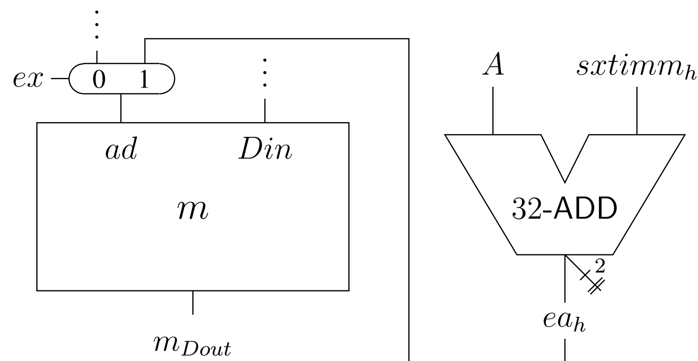


Abbildung 3.18: Datenpfad für load word-Instruktion

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:

$$\begin{aligned}
m_{Dout}(h^{2i+1}) &= h^{2i+1}.m(m_{ad}(h^{2i+1})) \quad \text{(Spezifikation RAM)} \\
&= h^{2i+1}.m(ea_h(h^{2i+1})[31:2]) \quad (\text{MUX, } h^{2i+1}.ex = 1) \\
&= h^{2i+1}.m((A(h^{2i+1}) +_{32} sxtimm(h^{2i+1}))[31:2]) \quad \text{(Addierer)} \\
&= h^{2i+1}.m((c^i.gpr(RS1(c^i)) +_{32} sxtimm(c^i))[31:2]) \quad \text{(Lemmas 3.9, 3.5)} \\
&= h^{2i+1}.m(ea(c^i)[31:2]) \quad (\text{Definition } ea(c)) \\
&= c^i.m(ea(c^i)[31:2]00) \quad \text{(Lemma 3.11)} \\
&= c^i.m(ea(c^i)) \quad \text{(alignment)} \quad \square
\end{aligned}$$

Nun wollen wir uns ansehen, wie die Ergebnisse in das GPR geschrieben werden. Abbildung 3.19 zeigt einen teil der GPR „glue logic“, die dessen Eingänge ansteuert. Die Korrektheit der Signale $aluop_h$,

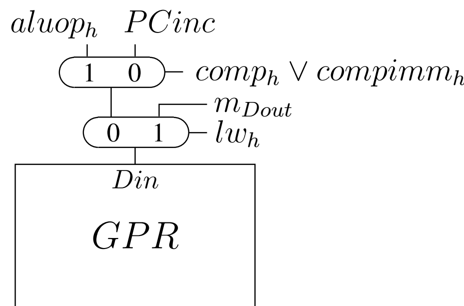


Abbildung 3.19: GPR glue logic für gpr_{Din}

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 3.20 ist der Ausschnitt von $nextPC$ dargestellt, der den PC inkrementiert.

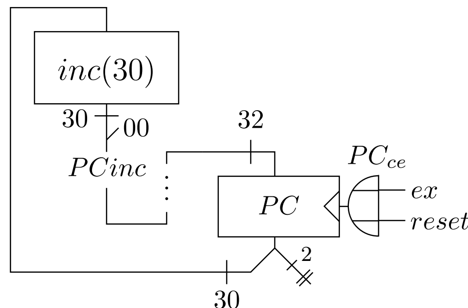


Abbildung 3.20: Implementierung von $PCinc$

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

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

Beweis:

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

Ein Incrementer soll in Aufgabe 2 des fünften Übungsblattes entworfen werden.

Nun kann endlich ein 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 $x \in \{0,1\}^{30}$:

$$\begin{aligned} h^{2i+2}.gpr(x) &= \begin{cases} gpr_{Din}(h^{2i+1}) & : (x = adC(h^{2i+1})) \wedge gpr_w(h^{2i+1}) \\ h^{2i+1}.gpr(x) & : \text{sonst} \end{cases} \quad \text{(Def. RAM)} \\ &= \begin{cases} aluop_h(h^{2i+1}) & : (comp_h(h^{2i+1}) \vee compimm_h(h^{2i+1})) \\ & \quad \wedge (x = adC(h^{2i+1})) \\ m_{Dout}(h^{2i+1}) & : lw_h(h^{2i+1})) \wedge (x = adC(h^{2i+1})) \\ PCinc(h^{2i+1}) & : (jal_h(h^{2i+1}) \vee jalr_h(h^{2i+1})) \\ & \quad \wedge (x = 1^5) \\ h^{2i+1}.gpr(x) & : \text{sonst} \end{cases} \quad \text{(Konstr. 3.19)} \\ &= \begin{cases} aluop(lop(c^i), rop(c^i), aluf(c^i)) & : (comp(c^i) \vee compimm(c^i)) \wedge (x = RD(c^i)) \\ c^i.m_4(ea(c^i)) & : lw_h(c^i) \wedge (x = RD(c^i)) \\ c^i.pc +_{32} 4_{32} & : (jal(c^i) \vee jalr(c^i)) \wedge (x = 1^5) \\ h^{2i+1}.gpr(x) & : \text{sonst} \end{cases} \\ & \quad \text{Dies folgt aus den Lemmas 3.10, 3.4, 3.6, 3.12, 3.13 und 3.8.} \\ &= c^i.gpr(x) \quad (\text{für } x \neq 0^5) \end{aligned}$$

Ein Vergleich mit $c^i.gpr(x)$ (3.5) zeigt, dass die Korrektheit der Hardware für $x = 0^5$ noch nicht gewährleistet ist, da bisher nicht gesichert ist, dass $\forall i : h^{2i}.gpr(0^5) = 0^{32}$. Dies muss später bei der Konstruktion des RAMs erzwungen werden. Dann gilt auch $\forall x \in \{0,1\}^5$:

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

Jetzt soll die Korrektheit des Speichers gezeigt werden, für die store word-Instruktionen entscheidend sind. Abbildung 3.21 zeigt den relevanten Datenpfad für store word-Speicherzugriffe. Es lässt sich dann

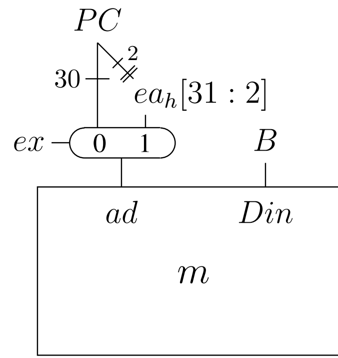


Abbildung 3.21: Datenpfad für store word

die Simulationsbedingung für den Speicher beweisen

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

Beweis:

$$\begin{aligned}
m_{ad}(h^{2i+1}) &= ea_h(h^{2i+1}[31:2]) \quad (\text{MUX, } h^{2i+1}.ex = 1) \\
&= ea(c^i) \quad (\text{siehe Beweis Lemma 3.12}) \\
m_{Din}(h^{2i+1}) &= B(h^{2i+1}) \\
&= h^{2i+1}.gpr(adB(h^{2i+1})) \quad (\text{Def. RAM})) \\
&= h^{2i+1}.gpr(h^{2i+1}.IR[20:16]) \quad (\text{Dekodierer}) \\
&= h^{2i+1}.gpr(RD_h(h^{2i+1})) \quad (Itype_h, \text{ insbesondere } sw_h) \\
&= c^i.gpr(RD(c^i)) \quad (\text{für } sw(c^i), \text{Lemma 3.8 und 3.5})
\end{aligned}$$

Im Falle von $sw(c^i) \wedge (x = ea(c^i)[31:2])$ gilt dann:

$$\begin{aligned}
h^{2i+2}.m(x) &= h^{2i+1}.m(ea(c^i)[31:2]) \\
&= h^{2i+1}.m(m_{ad}(h^{2i+1})) \\
&= m_{Din}(h^{2i+1}) \quad (\text{Definition RAM, } m_w(h^{2i+1}) = 1) \\
&= c^i.gpr(RD(c^i)) \\
&= c^{i+1}.m_4(x00) \quad (\text{Spezifikation, 3.2})
\end{aligned}$$

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

$$\begin{aligned}
h^{2i+2}.m(x) &= h^{2i+1}.m(x) \quad (\text{Definition RAM, } m_w(h^{2i+1}) = 0 \text{ oder } x \neq ea_h(h^{2i+1})) \\
&= h^{2i}.m(x) \quad (\text{Lemma 3.11}) \\
&= c^i.m(x00) \quad (sim(c^i, h^{2i})) \\
&= c^{i+1}.m(x00) \quad (\text{Spezifikation, 3.3}) \quad \square
\end{aligned}$$

Die Hardware simuliert also auch den Speicher der DLX korrekt. Bleibt nur noch die Korrektheit des Programmzählers zu beweisen. Hierbei sind die Instruktionen zur Kontrolle des Programmflusses (branch, jump etc.) zu betrachten. Abbildung 3.22 stellt die Implementierung des nextPC-Blockes dar. Aus der Konstruktion lässt sich ableiten:

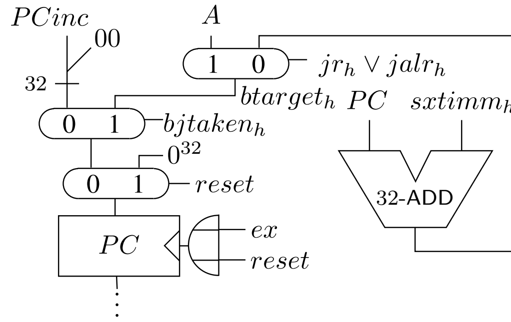


Abbildung 3.22: nextPC

$$
\begin{aligned}
btarget_h(h^{2i+1}) &= \begin{cases} A(h^{2i+1}) & : jr_h(h^{2i+1}) \vee jalr_h(h^{2i+1}) \\ sxtimm_h(h^{2i+1}) +_{32} h^{2i+1}.pc & : \text{sonst} \end{cases} \\
&= \begin{cases} c^i.gpr(RS1(c^i)) & : jr(c^i) \vee jalr(c^i) \\ sxtimm(c^i) +_{32} h^{2i+1}.pc & : \text{sonst} \end{cases} \quad \text{(Lemmas 3.9, 3.4, 3.5)} \\
&= \begin{cases} c^i.gpr(RS1(c^i)) & : jr(c^i) \vee jalr(c^i) \\ sxtimm(c^i) +_{32} h^{2i}.pc & : \text{sonst} \end{cases} \quad (PC_{ce}(h^{2i}) = 0) \\
&= \begin{cases} c^i.gpr(RS1(c^i)) & : jr(c^i) \vee jalr(c^i) \\ sxtimm(c^i) +_{32} c^i.pc & : \text{sonst} \end{cases} \quad (sim(c^i, h^{2i})) \\
&= btarget(c^i)
\end{aligned}
$$

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

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

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 3.23 dargestellt mit einem *zero tester* implementiert. Wegen $A(h^{2i+1}) = c^i.gpr(RS1(c^i))$ (Lemma 3.9) gilt:

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

Dann definiert man

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

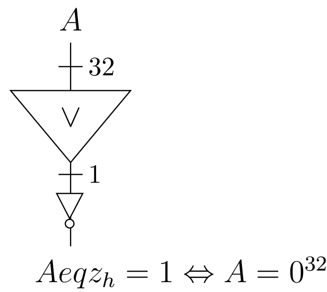


Abbildung 3.23: Implementierung von $Aeqz_h$

und es gilt:

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

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

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

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:

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

3.4.3 Register 0

Im Beweis der GPR-Korrektheit wurde ein wichtiger Fakt ausgelassen. Die Spezifikation der DLX gibt für das Register 0 des GPR vor:

$$h.gpr(0^5) = 0^{32}$$

Dies blieb bisher in der Hardware unberücksichtigt und soll nun gesichert werden. Dazu muss man sich die Konstruktion eines RAMs ansehen. Ein wichtiger Bestandteil sind dabei Decoder.

Definition 3.4 *(n-Decoder) Ein n-Decoder ist ein Schaltkreis mit Eingang $x \in \{0,1\}^n$ und Ausgang $y \in \{0,1\}^{2^n}$. Er berechnet die Unärdarstellung von $\langle x \rangle$. Es gilt:*

$$y[i] = 1 \leftrightarrow \langle x \rangle = i \qquad i \in \{0, \ldots, 2^n - 1\}$$

Abbildung 3.24 zeigt die schematische Darstellung eines Decoders.

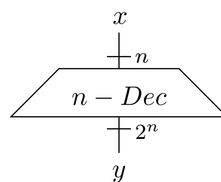


Abbildung 3.24: n-Decoder

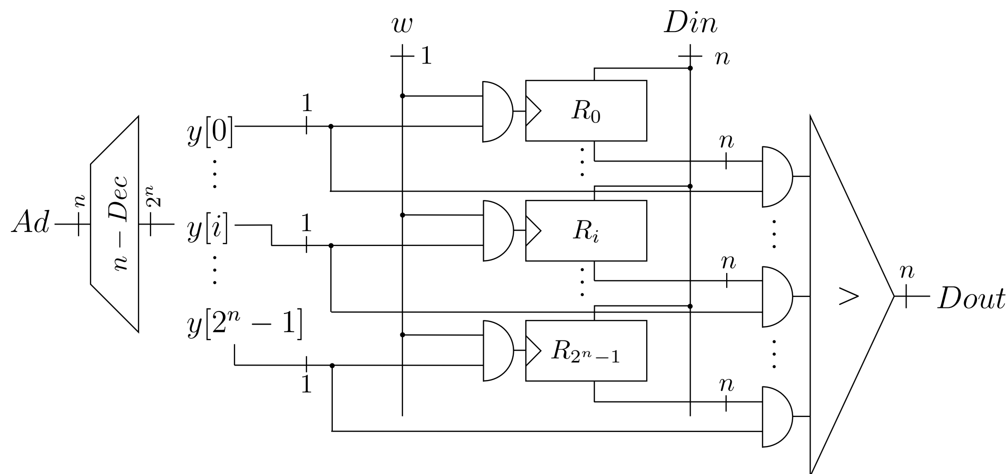


Abbildung 3.25: Implementierung eines RAMs

In Aufgabe 3 des fünften Übungsblattes soll ein Decoder konstruiert werden.

Ein RAM wird nun wie in Abbildung 3.25 gezeigt konstruiert. Das Adresssignal wird dekodiert, so das 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. Will man nun sicherstellen, dass in $R0$ stets nur 0^{32} steht, muss $R0$ einfach nur durch 0^{32} ersetzt werden. Schreibversuche mit $Ad = 0^5$ laufen dann ins Leere.

Kapitel 4

Software / C0

Das Programmieren in Assembler ist äußerst primitiv und mühsam. Es wäre wünschenswert, Programme komfortabler und übersichtlicher schreiben zu können, daher beschäftigt sich dieses Kapitel mit der Entwicklung einer Hochsprache für die DLX namens C_0. Diese vereint die bekannte *C*-Syntax mit der wohldefinierten *Pascal*-Semantik. Im Folgenden sollen Syntax und Semantik von C_0-Programmen definiert werden. Anschließend wird ein Compiler für DLX-Assembler konstruiert und dessen Korrektheit bewiesen, zunächst ist allerdings zur Definition der C_0-Syntax eine Einführung in kontextfreie Grammatiken nötig.

4.1 Kontextfreie Grammatiken

Grammatiken sind mathematische Modelle, die dazu dienen, formale Sprachen zu definieren. 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*
- S *- 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.

Die Definition verwendet die abkürzende Schreibweise

$$M^* = \bigcup_{i=0}^{\infty} M^i,$$

wobei

$$M^i = \{(a_1, \ldots, a_i) \mid \forall j \in [1 : i] : a_j \in M\}$$

endliche Folgen von i Elementen aus M darstellt. Desweiteren führen wir 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 $(n, \omega^1), \dots, (n, \omega^t) \in P$ so schreibt man:

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

Es folgt ein Beispiel für eine einfache cfg:

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

Dabei stehen V für eine Variable, C für eine binäre Ziffer und $\langle CF \rangle$ für eine binäre Ziffernfolge. Es lässt sich erahnen, dass die obige Grammatik Wörter erzeugt, die mit einem X beginnen, gefolgt von einer beliebig langen Folge aus 0 und 1. Um die erzeugte Sprache einer Grammatik $L(G) \in T^*$ näher zu definieren müssen Ableitungsbäume betrachtet werden.

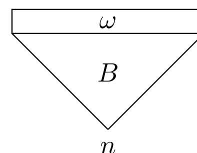


Abbildung 4.1: Ableitungsbaum

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.*

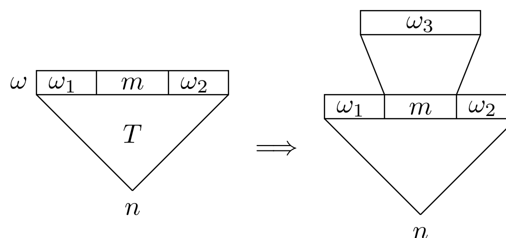


Abbildung 4.2: weitere Ableitung eines Ableitungsbaumes

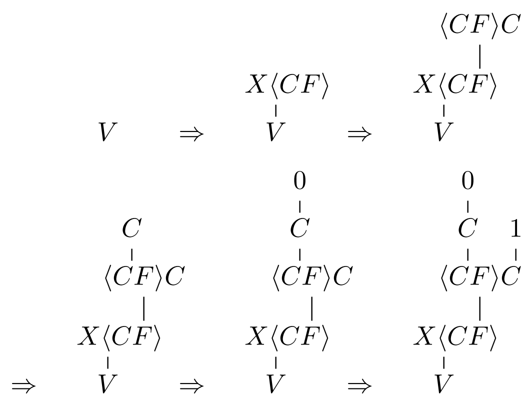


Abbildung 4.3: Ableitungsbeispiel

Abbildung 4.3 zeigt als Beispiel die schrittweise Ableitung des Blattwortes $X01$ aus der Wurzel V mit der gegebenen Beispielgrammatik.

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:*

$$n \longrightarrow_G^* \omega$$

Die Sprache die durch G erzeugt wird ist dann:

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

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

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

Beispiel: Vollständig geklammerte Boolesche Ausdrücke:

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

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

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

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:

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

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

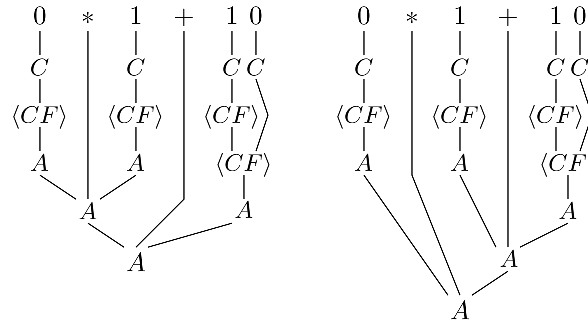


Abbildung 4.4: Ableitungsbäume für uneindeutige Grammatik

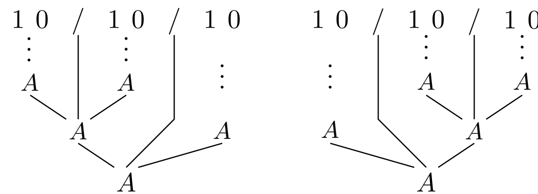


Abbildung 4.5: Ableitungsbäume für uneindeutige Grammatik

Wurzel.

Definition 4.4 *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:

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

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

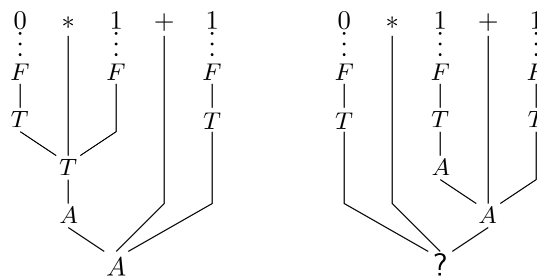


Abbildung 4.6: Ableitung mit eindeutiger Grammatik

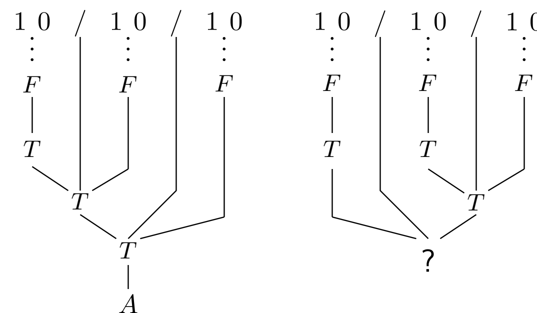


Abbildung 4.7: Ableitung mit eindeutiger Grammatik

„Sackgasse“. Es gibt keine zweite Möglichkeit, einen Ausdruck auf das Startsymbol zurückzuführen. 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).

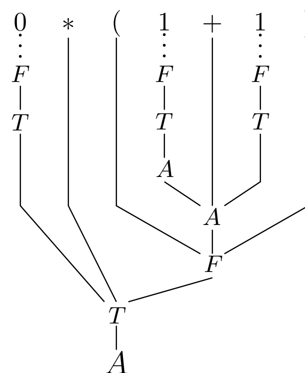


Abbildung 4.8: eindeutige Ableitung mit Klammersetzung

Nun soll eine Grammatik für die Programmiersprache C_0 erstellt werden.

4.2 C0-Syntax

Tabelle 4.1 zeigt die Produktionsregeln der kontextfreien Grammatik von C_0. Sämtliche Symbole in eckigen Klammern stellen Nichtterminale dar. Die Terminale von C_0 sind Buchstaben und Zahlen sowie Schlüsselworte (if, $then$, $while$, $typedef$,...), Operatoren (+, & , ∨, *, ...) und weitere Sonderzeichen ({,],;,=,...). Das Startsymbol ist $\langle programm \rangle$. Daraus kann eine Folge von Variablen-, Funktions-,

$\langle Zi\rangle$	$\longrightarrow$	$0 \mid \ldots \mid 9$	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 (Variablen und Konstanten)
$\langle F\rangle$	$\longrightarrow$	$\langle id\rangle \mid -_1\langle F\rangle \mid (\langle A\rangle) \mid \langle C\rangle$	Faktor
$\langle T\rangle$	$\longrightarrow$	$\langle F\rangle \mid \langle T\rangle * \langle F\rangle \mid \langle T\rangle/\langle F\rangle$	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
$\langle Atom\rangle$	$\longrightarrow$	$\langle A\rangle > \langle A\rangle \mid \langle A\rangle \geq \langle A\rangle \mid \langle A\rangle < \langle A\rangle \mid \langle A\rangle \leq \langle A\rangle \mid$	„Boolesche Variable“
		$\langle A\rangle == \langle A\rangle \mid \langle A\rangle \neq \langle A\rangle \mid 0 \mid 1$	
$\langle BF\rangle$	$\longrightarrow$	$\langle Atom\rangle \mid\sim\langle BF\rangle \mid (\langle BA\rangle)$	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
$\langle An\rangle$	$\longrightarrow$	$\langle id\rangle = \langle A\rangle \mid \langle id\rangle = \langle BA\rangle \mid$	„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
		$\langle id\rangle = \langle Na\rangle(\langle PF\rangle) \mid$	Funktionsaufruf mit Parametern
		$\langle id\rangle = \langle Na\rangle() \mid$	Funktionsaufruf ohne Parameter
		$\langle id\rangle = new\ \langle Typ\rangle^*$	Speicher allozieren
$\langle PF\rangle$	$\longrightarrow$	$\langle A\rangle \mid \langle PF\rangle, \langle A\rangle$	Parameterfolge
$\langle AnF\rangle$	$\longrightarrow$	$\langle An\rangle \mid \langle AnF\rangle; \langle An\rangle$	Anweisungsfolge
$\langle program\rangle$	$\longrightarrow$	$\langle DF\rangle$	C_0-Programm
$\langle DF\rangle$	$\longrightarrow$	$\langle Dekl\rangle \mid \langle DF\rangle; \langle Dekl\rangle$	Deklarationsfolge
$\langle Dekl\rangle$	$\longrightarrow$	$\langle VaD\rangle \mid \langle FuD\rangle \mid \langle TypD\rangle$	Deklaration
$\langle VaD\rangle$	$\longrightarrow$	$\langle Typ\rangle\ \langle Na\rangle$	Variablendeklaration
$\langle VaDF\rangle$	$\longrightarrow$	$\langle VaD\rangle \mid \langle VaDF\rangle; \langle VaD\rangle$	Variablendeklarationsfolge
$\langle Typ\rangle$	$\longrightarrow$	$\langle elTyp\rangle \mid \langle elTyp\rangle[\langle ZiF\rangle] \mid \langle elTyp\rangle^* \mid$	elementare Typen, Array-Typ, Pointer
		$\langle Na\rangle \mid \langle Na\rangle[\langle ZiF\rangle] \mid \langle Na\rangle^* \mid$	selbstdefinierte Typen
		$struct\{\langle KompDF\rangle\}$	struct-Typ
$\langle elTyp\rangle$	$\longrightarrow$	$int \mid bool \mid char \mid unsigned$	elementare Typen
$\langle KompDF\rangle$	$\longrightarrow$	$\langle KompD\rangle \mid \langle KompDF\rangle; \langle KompD\rangle$	Komponentendeklarationsfolge
$\langle KompD\rangle$	$\longrightarrow$	$\langle Typ\rangle\ \langle Na\rangle$	Komponentendeklaration
$\langle FuD\rangle$	$\longrightarrow$	$\langle Typ\rangle\ \langle Na\rangle(\langle ParDF\rangle)\{\langle VaDF\rangle; \langle rumpf\rangle\} \mid$	Funktionsdeklaration
		$\langle Typ\rangle\ \langle Na\rangle(\langle ParDF\rangle)\{\langle rumpf\rangle\} \mid$	ohne lokale Variablen
		$\langle Typ\rangle\ \langle Na\rangle()\{\langle VaDF\rangle; \langle rumpf\rangle\} \mid$	ohne Parameter
		$\langle Typ\rangle\ \langle Na\rangle()\{\langle rumpf\rangle\}$	ohne lokale Variablen und Parameter
$\langle ParDF\rangle$	$\longrightarrow$	$\langle ParD\rangle \mid \langle ParDF\rangle, \langle ParD\rangle$	Parameterdeklarationfolge
$\langle ParD\rangle$	$\longrightarrow$	$\langle Typ\rangle\ \langle Na\rangle$	Parameterdeklaration
$\langle rumpf\rangle$	$\longrightarrow$	$\langle AnF\rangle; return\ \langle A\rangle \mid return\ \langle A\rangle$	Funktionsrumpf
$\langle TypD\rangle$	$\longrightarrow$	$typedef\ \langle Typ\rangle\ \langle Na\rangle$	Typdeklaration

Tabelle 4.1: Grammatik von C_0

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 C_0-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

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

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
}
```

Im nächsten Abschnitt soll näher auf die Typdeklarationen in C_0 eingegangen werden.

4.3 Typdeklarationen

C_0 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
```

wobei ein array ein eindimensionales Feld von Variablen des jeweiligen Typs im Speicher darstellt. Im obigen Beispiel würde der neudefinierte Typ xyz einen Vektor aus 14 int-Elementen 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

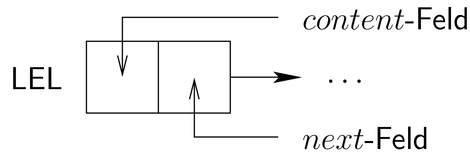


Abbildung 4.9: Listenelement

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 C_0 wäre dies:

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

Das führt jedoch auf ein Problem, da die Definitionen rekursiv verschränkt sind. Es wäre wünschenswert, dass LEL deklariert ist bevor man einen pointer darauf setzt, bzw LEL^* als u deklariert wird, bevor man es in der Strukturdeklaration verwendet. Beides führt zu Widersprüchen, da man nun einmal mit einer Deklaration beginnen muss. C_0 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.
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 C_0 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 $j > i$.

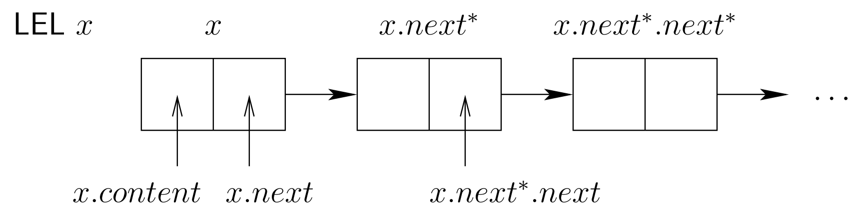


Abbildung 4.10: einfach verkettete Liste

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 $*$-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:

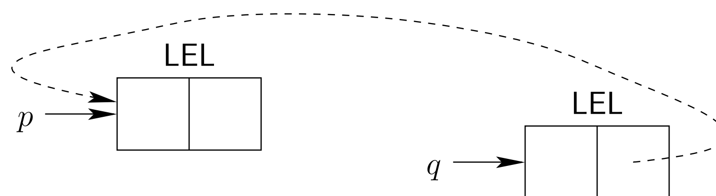


Abbildung 4.11: Konstruktion einer Liste mit new

```
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 C_0-Maschine darstellt. Offensichtlich hängt K_p nur von den Deklarationen in p ab. Analog zu δ_D und δ_H definieren die Übergangsfunktion für C_0-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 im weiteren Verlauf betrachten, wie die Typ-, Variablen- und Funktionsdeklarationen auf die Konfigurationen abgebildet werden. Desweiteren wird Ausdrucksauswertung und Anweisungsausführung behandelt werden. Am Ende des Semantikabschnitts soll die Korrektheit einiger Beispielprogramme bewiesen werden.

4.4.1 Typdeklarationen

Zur Verwaltung der deklarierten Typen legen wir eine Typtabelle tt an. Dorthin werden alle Typdeklarationen abgebildet.

$$tt = (tt.n, tt.name, tt.tc)$$

Dabei ist:

- $tt.n$ - die Anzahl der Typen (#Typdeklarationen + 4)
- $tt.name : [0 : tt.n - 1] \longrightarrow \{\text{Namen}\}$ - die zugehörigen Namen der Typen
  - $tt.name(0) = int$
  - $tt.name(1) = bool$
  - $tt.name(2) = char$
  - $tt.name(3) = unsigned$
  - $tt.name(3 + i) = \text{Name in } i\text{−ter Typdeklaration}$
- $tt.tc : [0 : tt.n - 1] \longrightarrow \{\text{Typdeskriptoren}\}$ - Beschreibung der Typen (**t**ype **c**ontent)

Um zu definieren, wie Zeile $3 + i$ der Typtabelle ausgefüllt werden soll, muss eine Fallunterscheidung über die i-te Typdeklaration getroffen werden:

- $typedef\ name_1[n]\ name_2$ (array-Typen)
  - $tt.name(3 + i) = name_2$
  - $\exists j < 3 + i : tt.name(j) = name_1 \Rightarrow tt.tc(3 + i) = arr(j, n)$
- $typedef\ struct\{t_1\ n_1; \ldots ; t_s\ n_s\}\ name_1$ (struct-Typen)
  - $tt.name(3 + i) = name_1$
  - $\forall j \in [1 : s] \exists i_j < 3 + i : tt.name(i_j) = t_j \Rightarrow tt.tc(3 + i) = struct\{i_1\ n_1, \ldots , i_s\ n_s\}$
- $typedef\ name1^*\ name2$ (pointer-Typen)
  - $tt.name(3 + i) = name2$
  - $\exists j \in [0 : tt.n - 1] : tt.name(j) = name1 \Rightarrow tt.tc(3 + i) = ptr(j)$

Für die pointer-Typen ist explizit erlaubt, dass $j \geq 3 + i$ 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
```

i	$tt.name(i)$	$tt.tc(i)$	$size(i)$
0	int	elementar	1
1	$bool$	elementar	1
2	$char$	elementar	1
3	$unsigned$	elementar	1
4	x	arr(0,5)	5
5	row	arr(4,5)	25
6	u	ptr(7)	1
7	LEL	struct{0 content, 6 next}	2

Tabelle 4.2: Typtabelle für Beispieldeklaration und Größe der Typen

Die Größe der Typen ist ebenfalls angegeben. Wie man sie 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. Für den deklarierten Typen t ergibt sich:

$$size(t) = \begin{cases} 1 & : \quad t \text{ einfach} \\ n \cdot size(t') & : \quad t = t'[n] \\ \sum_{i=1}^{s} size(t_i) & : \quad t = struct\{t_1\ n_1, \ldots, t_s\ n_s\} \end{cases}$$

Ausdrücke wie

$$t = \begin{cases} t'* \\ t'[n] \\ struct\{t_1\ n_1, \ldots, t_s\ n_s\} \end{cases}$$

sind dabei eine Kurzschreibweise dafür, dass t mittels

$$typedef \left\{ \begin{matrix} t'* \\ t'[n] \\ struct\{t_1\ n_1, \ldots, t_s\ n_s\} \end{matrix} \right\} t$$

definiert wurde. Allerdings ist diese Definition für die Größe von Typen etwas zu nachlässig, in dem Sinne, dass der Bezug zur Typtabelle fehlt. Eine formale Definition für $size(i)$ soll als Übung erstellt werden. Man kann zudem zeigen, dass die Typdeklarationen aus der Typtabelle zurückgewonnen werden können, weshalb die obige Form trotzdem ihre Berechtigung behält.

Es stellt sich die Frage, wie man auf Komponenten komplexer Typen zugreifen kann. Dazu betrachtet man zunächst den Wertebereich $\text{Range}(t)$, kurz $Ra(t)$, eines Typs. Eine informelle Beschreibung könnte lauten:

$$\text{Range}(t) = \{\omega \mid \text{Variablen vom Typ } t \text{ können den Wert } \omega \text{ annehmen}\}$$

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.

$$\begin{aligned} Ra(int) &= T_{32} \\ Ra(unsigned) &= \{0, \ldots, 2^{32} - 1\} \\ Ra(bool) &= \{0, 1\} \\ Ra(char) &= \{0, \ldots, 2^{8} - 1\} \end{aligned}$$

Der Wert von pointern ist noch undefiniert, jedoch wird sich später zeigen, dass pointer Variablen als Wert besitzen. Dies bedeutet insbesondere, dass in C_0 keine pointer-Arithmetik möglich ist. Für komplexe t kann man $Ra(t)$ als eine Folge der Länge $size(t)$ von „einfachen Werten“ darstellen.

$$Ra(t) \ni f : [0 : size(t) - 1] \longrightarrow \{\omega \mid \omega \text{ ist einfacher Wert}\}$$

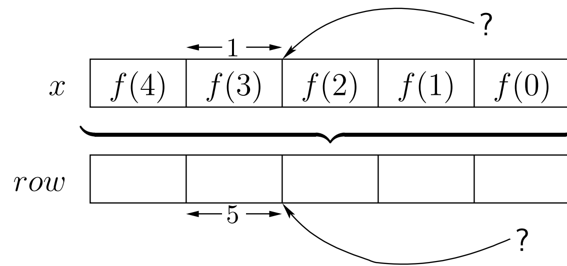


Abbildung 4.12: Zugriff auf Komponenten von Typen

Ein einfacher Wert ist der Wert einer Variable einfachen Typs. Abbildung 4.12 verdeutlicht die Fragestellung für die Beispieldeklaration

```
typedef int[5] x;
typedef x[5] row;
x y;
row z
```

wenn man zum Beispiel auf die Komponenten $y[3]$ oder $z[3]$ zugreifen möchte.

Wir definieren nun $Ra(t)$ formal für komplexe Typen:

$$Ra(t) = \{f \mid f : [0 : size(t) - 1] \longrightarrow \bigcup_{t' \text{ einfach}} Ra(t')\}$$

Mit solchen Funktionen f ließe sich also auf Teilstrukturen eines Typs zugreifen, indem man f auf einen entsprechenden Index anwendet. Man muss lediglich die Größe der gesuchten Komponente i kennen und wissen, an welcher Stelle innerhalb des komplexen Typs t sie sich befindet. Diese Positionsangabe bezeichnen wir als *displacement* (Distanz, Versatz, Verschiebung) von i in t:

$$displ(i, t)$$

Wir unterscheiden zwischen den zwei möglichen komplexen Typen für t:

- $t = t'[n]$ - t ist ein array. Abbildung 4.13 verdeutlicht die Details.

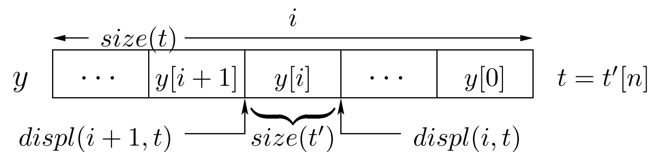


Abbildung 4.13: Zugriff auf array-Komponenten

Es gilt für ein $f \in Ra(t)$:

$$\begin{aligned} y[i] &= f_{size(t')}(displ(i, t)) \in Ra(t) \\ displ(i, t) &= i \cdot size(t') \end{aligned}$$

- $t = struct\{t_1\ n_1; \ldots; t_s\ n_s\}$ - t ist ein struct. Abbildung 4.14 zeigt die Einzelheiten.
  Es gilt für ein $f \in Ra(t)$:

$$\begin{aligned} y.n_i &= f_{size(t_i)}(displ(n_i, t)) \in Ra(t_i) \\ displ(n_i, t) &= \sum_{j<i} size(t_j) \end{aligned}$$

Wir haben also die komplexen Typen auf simple Vektoren aus einfachen Typen heruntergebrochen, die mal leicht mit der gewohnten Speichernotation behandeln kann.

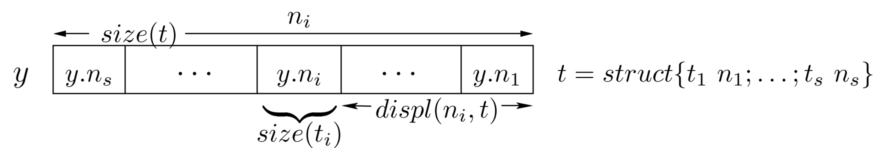


Abbildung 4.14: Zugriff auf struct-Komponenten

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. Wir definieren eine formale Beschreibung des Speichers, den ein Programm nutzt, ähnlich zur Typtabelle eines Programms.

Definition 4.5 *(memory) Ein memory m ist ein 4-Tupel $m = (m.n, m.name, m.typ, m.ct)$. Dabei sind die einzelnen Bestandteile:*

- $m.n \in \mathbb{N}$ - *die Anzahl der Variablen im Speicher*
- $m.name : [0 : m.n - 1] \longrightarrow \{Namen\} \cup \{\Omega\}$ - *die Namen der Variablen im Speicher (Ω für namenlose Variablen)*
- $m.typ : [0 : m.n - 1] \longrightarrow [0 : tt.n - 1]$ - *Zeile in der Typtabelle tt, in der der Typ der Variable deklariert wurde*
- $m.ct : [0 : size(m) - 1] \longrightarrow \{\omega \mid \omega \text{ ist ein einfacher Wert}\}$ - *der Wert der Variable (**cont**ent)*

Die Größe des Speichers berechnet sich zu

$$size(m) = \sum_{i=0}^{m.n-1} size(m.typ(i))$$

Man beachte, dass der Inhalt der Variablen als Folge einfacher Werte dargestellt wird. Dies entspricht der Vorgehensweise bei komplexen Typen. Tabelle 4.3 zeigt die Speicherbelegung für das folgende Beispiel. Dabei werden die Typdeklarationen aus vorangegangenen Beispielen verwendet. Eine solche Tabelle, die die Zuordnung von Namen und Typen zu Variablen in einem Speicher m enthält, wird als Symboltabelle $sy(m)$ bezeichnet. 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 C_0.

```
int y;
matrix z;
u p;
```

i	$m.name(i)$	$m.typ(i)$	$m.ct$
0	y	0	undefiniert
1	z	5	undefiniert
2	p	6	undefiniert

Tabelle 4.3: Symboltabelle $sy(m)$ und Inhalt $m.ct$ für Beispieldeklarationen

Der Speicher wird wie ein struct mit $m.n$ Komponenten betrachtet. $m.ct$ ist dann der Wert eines struct-Elements. Nun sollen Variablen und Subvariablen formal definiert werden.

Definition 4.6 *(Variable) Die i-te Variable in Speicher m wird dargestellt durch ein Paar (m, i). Dabei ist*

- $m = (m.n, m.name, m.typ, m.ct)$ *ein memory*
- $i \in [0 : m.n - 1]$ *der Index der Variable in* m

Definition 4.7 *(Subvariable) Zunächst definieren wir Selektoren* s_j*. Diese sind entweder*

- ε *(leer),*
- $[i]$ *mit* $i \in \mathbb{N}$ *oder*
- $.na$ *mit einem Namen* na*.*

Dann ist eine Subvariable eine Variable gefolgt von einer Anzahl von Selektoren.

$$(m, i)s_1 \cdots s_j$$

Zum Beispiel würde die Subvariable $(m, i)[17].gpr[5]$ mit $m.name(i) = PCB$ eine Komponente eines process control blocks darstellen. Die Frage ist nun, welche Subvariablen in einem Speicher m existieren, welche Typen sie haben und wie man gegebenfalls auf sie zugreift.

Definition 4.8 *(Subvariablen von* m*) Man definiert die Subvariablen und ihre Typen in einem Speicher* m *durch folgende Fallunterscheidung:*

- (m, i) *ist eine Subvariable* ($s_1 = \varepsilon$)*, das heißt, Variablen sind auch Subvariablen*
- u *ist eine Subvariable mit* $typ(u) = t'[n]$*, dann folgt*
  - $\forall j \in \mathbb{N}, j < n : u[j]$ ist eine Subvariable
  - $typ(u[j]) = t'$
- u *ist eine Subvariable mit* $typ(u) = struct\{t_1\ n_1; \ldots; t_s\ n_s\}$*, dann folgt*
  - $\forall j \in [1 : s] : u.n_j$ ist eine Subvariable
  - $typ(u.n_j) = t_j$

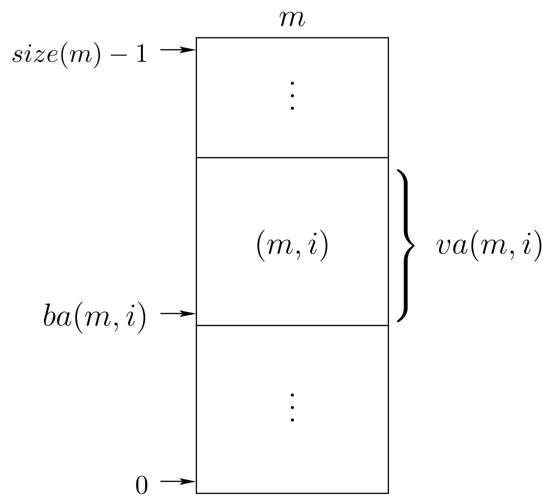


Abbildung 4.15: $m.ct$, $ba(m, i)$ und $va(m, i)$

Abbildung 4.15 zeigt $m.ct$, den Inhalt des Speichers, bestehend aus $size(m)$ einfachen Werten. Komplexe Variablen (m, i) erstrecken sich über mehrere einfache Werte und beginnen an ihrer Basisadresse $ba(m, i)$.

$$ba(m, i) = \sum_{j<i} size(m.typ(j))$$

Diese ist also über die Gesamtgröße aller vorhergehenden Variablen in m definiert. Die Größe einer Variablen beschreiben wir auch kurz mit

$$size(m.typ(j)) \mathrel{\widehat{=}} size(m, j).$$

Dann ist der Inhalt einer Variable (m, i) die Folge aller zugehörigen Werte aus dem Speicher, beginnend ab der Basisadresse.

$$va(m, i) = m.ct_{size(m,i)}(ba(m, i))$$

Bei Subvariablen muss man noch zusätzlich die Position innerhalb der Variable betrachten.

Definition 4.9 *(Basisadresse, Größe und Wert von Subvariablen) Sei u eine Subvariable in memory m. Dann unterscheiden wir die Fälle:*

- $u = (m, i)$, *u ist eine Variable, dann gilt:*

$$\begin{aligned} ba(m, u) &= ba(m, i) \\ size(m, u) &= size(m, i) \\ va(m, u) &= va(m, i) \end{aligned}$$

- $typ(u) = t'[n]$, *u ist ein array, dann gilt* $\forall j \in \mathbb{N}, j < n$:

$$\begin{aligned} ba(m, u[j]) &= ba(m, u) + displ(j, t) \\ size(m, u[j]) &= size(t') \\ va(m, u[j]) &= m.ct_{size(t')}(ba(m, u[j])) \end{aligned}$$

- $typ(u) = struct\{t_1\ n_1; \ldots; t_s\ n_s\} = t = t$, *u ist ein struct, dann gilt* $\forall j \in [1 : s]$

$$\begin{aligned} ba(m, u.n_j) &= ba(m, u) + displ(n_j, t) \\ size(m, u.n_j) &= size(t_j) \\ va(m, u.n_j) &= m.ct_{size(t_j)}(ba(m, u.n_j)) \end{aligned}$$

Man beachte dass die Funktionen für Basisadresse, Größe und Wert durch die Varianten $ba(m, u)$, $size(m, u)$ und $va(m, u)$ für Subvariablen u überladen wurden. Abbildung 4.16 veranschaulicht den Zugriff auf Subvariablen im Speicher am Beispiel von array-Komponenten. Mit den obigen Definitionen folgt allgemein für den Wert von Subvariablen u:

$$va(m, u) = m.ct_{size(u)}(ba(m, u))$$

4.4.3 Funktionsdeklarationen

Zuletzt werden die Deklarationen von Funktionen betrachtet. FN bildet die Menge der Funktionsnamen in einem C_0-Programm:

$$FN = \{\text{deklarierte Funktionsnamen}\} \cup \{main\}$$

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.

$$ft(f) = (ft(f).typ, ft(f).body, ft(f).np, ft(f).nl, ft(f).st)$$

Das wären im Einzelnen:

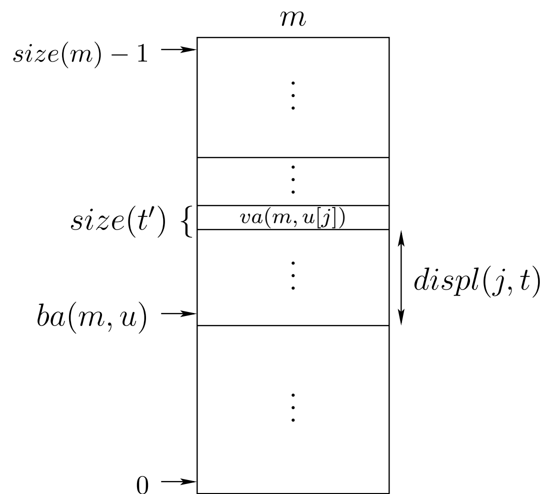


Abbildung 4.16: Zugriff auf array-Komponenten

- $ft(f).typ \in [0 : tt.n - 1]$ - Typ des Rückgabewertes von f
- $ft(f).body$ - Rumpf der Funktion f (abgeleitet von $\langle rumpf \rangle$)
- $ft(f).np$ - Anzahl der Parameter von f
- $ft(f).nl$ - Anzahl der lokalen Variablen von f
- $ft(f).st$ - Symboltabelle für Funktion f:
  - $ft(f).st.n$ - Größe der Symboltabelle von f
    Es gilt:
    $$ft(f).st.n = ft(f).np + ft(f).nl$$
  - $ft(f).st.name(i)$ - Name des Parameters oder der lokalen Variablen i von Funktion f
    Seien $p_0, \ldots, p_{ft(f).np-1}$ die Namen der Parameter und $l_0, \ldots, l_{ft(f).nl-1}$ die Namen der lokalen Variablen von f, dann gilt:
    $$ft(f).st.name(i) = \begin{cases} p_i & : \quad i \in [0 : ft(f).np - 1] \\ l_{i-ft(f).np} & : \quad i \in [ft(f).np : ft(f).st.n - 1] \end{cases}$$
  - $ft(f).st.typ(i)$ - Typ des Parameters oder der lokalen Variablen i von Funktion f
    Es gilt:
    $$ft(f).st.typ(i) = typ(ft(f).st.name(i)) \in [0 : tt.n - 1]$$

Man beachte, dass die Symboltabelle $ft(f).st$ strukturgleich zu der Symboltabelle $sy(m)$ 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.

4.4.4 Konfiguration von C0-Maschinen

Nun, da die Rahmenbedingungen durch die Typ-, Variablen- und Funktionsdeklarationen aufgestellt wurden, wollen wir die Ausführung von C_0-Programmen untersuchen. Dazu muss zunächst eine C_0-Konfiguration c definiert werden. Sie besteht aus:

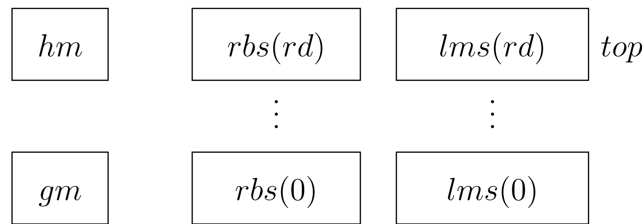


Abbildung 4.17: Komponenten der C_0-Konfiguration

- $c.pr$ (**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 der $return$-Anweisungen in $c.pr$ entspricht $c.rd$.
- $c.lms : [0 : c.rd] \longrightarrow \{\text{memories}\}$ (**l**ocal **m**emory **s**tack) - Menge von Speichern (*function frames*), die bei einem Funktionsaufruf angelegt werden
  $c.lms(i)$ stellt den i-ten function frame 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
  In der Symboltabelle schreibt man dann $c.hm.name(i) = \Omega$.
- $c.rbs : [0 : c.rd] \longrightarrow \{\text{Variablen}\}$ (**r**eturn **b**inding **s**tack) - Menge von Rückgabebindungen, die bei einem Funktionsaufruf angelegt werden
  In $c.rbs(i)$ wird das Ergebnis der zu Rekursionstiefe i gehörigen Funktion gespeichert.

Wird nun ein Programm auf der C_0-Maschine ausgeführt, so ändert sich deren Zustand und damit die Konfiguration c mit jeder Anweisungsausführung. Abbildung 4.17 stellt die Komponenten der C_0-Konfiguration schematisch dar. Der oberste local memory stack wird als *top frame* $top(c)$ bezeichnet:

$$top(c) = lms(c.rd)$$

4.4.5 Ausdrucksauswertung

Bei der Ausführung von C_0-Anweisungen müssen stets Ausdrücke (von $\langle A \rangle$ abgeleitet) ausgewertet werden, um den Effekt des Befehls zu berechnen. Die Ausdrücke sind sozusagen die Argumente der Anweisungen. 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 $(m, i)s$, also eine Subvariable (left value), der der Wert des rechten Ausdrucks e' zugewiesen werden soll. Dieser liegt im Wertebereich $Ra((m, i)s)$ (right value). Bindungen von Ausdrücken e, also L-values, werden im Zustand c durch die Funktion $bind(c, e)$ gefunden. Entsprechend gibt die Funktion $va(c, e)$ den R-value von e in der Konfiguration c zurück. Man beachte, dass die Funktion va schon als Variante $va(m, i)$ existiert und den Wert von Variable i in Speicher m liefert. Die Notation für va ist also überladen.
Die Auswertung der Ausdrücke findet rekursiv statt. Man unterscheidet für einen Ausdruck e die Fälle:

- $e \equiv e' \circ e''$ Boolescher oder arithmetischer Ausdruck ($\langle BA \rangle \longrightarrow^*_{C_0} e$ oder $\langle A \rangle \longrightarrow^*_{C_0} e$) mit $\circ \in \{<, >, \leq, \geq, ==, \neq, +, -_2, *, /\}$:

$$\begin{aligned} bind(c,e) & \quad \text{nicht definiert} \\ va(c,e) &= va(c,e') \circ va(c,e'') \end{aligned}$$

- $e \equiv id$ Identifier ($\langle id \rangle \longrightarrow^*_{C_0} id$)
  Identifier sind zumeist an Subvariablen u im Speicher gebunden:

  $$bind(c,id) = u$$

  Um den Wert des Ausdrucks zu bestimmen muss man dann den Wert der zugehörigen Subvariable mit $va(m,i)$ ermitteln:

  $$va(c,id) = va(u)$$

  Weierhin muss man die folgenden Fallunterscheidungen machen:

  - $id \in \{0, \ldots, 9\}^L$ Konstante (L ist die Größe der Hardware-Arithmetik)

    $$\begin{aligned} bind(c,id) & \quad \text{nicht definiert} \\ va(c,id) &= \sum_{i=0}^{L-1} id_i \cdot 10^i \end{aligned}$$

  - $id \equiv X$ Variablenname X

    $$\begin{aligned} bind(c,X) &= \begin{cases} (top(c), j) & : \quad top(c).name(j) = X \\ (c.gm, j) & : \quad (\nexists i : top(c).name(i) = X) \wedge c.gm.name(j) = X \end{cases} \\ va(c,X) &= va(bind(c,id)) \end{aligned}$$

  - $id \equiv e'[e'']$ Arrayzugriff

    $$\begin{aligned} bind(c,id) &= bind(c,e')[va(c,e'')] \\ va(c,id) &= va(bind(c,id)) \end{aligned}$$

  - $id \equiv e'.n$ struct-Komponenten-Zugriff

    $$\begin{aligned} bind(c,id) &= bind(c,e').n \\ va(c,id) &= va(bind(c,id)) \end{aligned}$$

  - $id \equiv e'*$ Pointer e' dereferenzieren

    $$\begin{aligned} bind(c,id) &= va(c,e') \\ va(c,id) &= va(bind(c,id)) \\ &= va(va(c,e')) \end{aligned}$$

  - $id \equiv e'\&$ Adresse von Identifier e' abfragen

    $$\begin{aligned} bind(c,id) & \quad \text{nicht definiert} \\ va(c,id) &= bind(c,e') \end{aligned}$$

Mit den Funktionen $bind(c,e)$ und $va(c,e)$ kann man also alle Ausdrücke auswerten, die in C_0 auftreten.

4.4.6 Anweisungsausführung

Nun können wir die Abarbeitung eines C_0-Programms definieren. Dabei betrachten wir die Folge der Anweisungen abhängig vom Zustand der C_0-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.
- $c^0.pr = \text{mainAnF}$ - Es wird der Rumpf von $main$ ausgeführt, jedoch ohne die $return$-Anweisung. Für den Rumpf von $main$ gilt:

  $$\langle rumpf\rangle \longrightarrow^*_{C_0} \text{mainAnF}; return\ \text{X} = ft(main).body \in T^*_{C_0}$$

- $sy(top(c^0)) = sy(lms(c^0)) = ft(main).st$ - Zu Beginn liegt einzig der local memory von $main$ auf dem stack.
- $sy(c^0.gm) = \dots$ - Der Startzustand des global memory soll unter Berücksichtigung der Variablendeklarationen in einer Übung bestimmt werden.
- $c^0.hm.n = 0$ - Der heap memory ist anfangs leer.
- $c^0.rbs(0)$ nicht definiert - $return$ wird für $main$ nicht ausgeführt.

Wird nun eine Anweisung auf die Konfiguration c ausgeführt, so erhalten wir die resultierende Konfiguration c' durch die C_0-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:

  $$\begin{aligned} va(c', id) &= va(c, e) \\ c'.pr &= r' \end{aligned}$$

  Sonst soll sich nichts ändern. Um c' zu konstruieren, untersuchen wir die Bindung von id.

  $$\begin{aligned} bind(c, id) &= (m, i)s = u \\ va(c', id) &= va(\underbrace{(m, i)s}_{u}) \\ &= m.ct_{size(u)}ba(u) \end{aligned}$$

  Dies ist der alte Wert der Subvariable u, an die id gebunden ist. Außerdem ist m eigentlich konfigurationsabhängig ($m = m(c) \in \{c.gm, c.hm, c.lms(j)\}$). Wir werten nun e aus und weisen u den neuen Wert zu.

  $$m(c').ct_{size(u)}ba(u) = va(c, e)$$

  Dabei ist es wichtig, zu bemerken, dass

  $$bind(c', id) = bind(c, id) = u,$$

  also, dass sich die Bindung des Identifiers nicht während der Ausführung der Zuweisung ändert. Dies ist bei Funktionsaufrufen zum Beispiel nicht garantiert.

- $an: \quad if\ e\ then\ \{a\}\ else\ \{b\}$ (Fallunterscheidung)

$$c'.pr = \begin{cases} a; r' & : \quad va(c,e) = true \\ b; r' & : \quad \text{sonst} \end{cases}$$

- $an: \quad while\ e\ do\ \{a\}$ (Schleife)

$$c'.pr = \begin{cases} a; while\ e\ do\ \{a\}; r' & : \quad va(c,e) = true \\ r' & : \quad \text{sonst} \end{cases}$$

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.

- $an: \quad p = new\ t*, \quad t$ ist ein Typ (new-Anweisung)

$$\begin{aligned} c'.hm.n &= c.hm.n + 1 \\ \exists i \in [0 : tt.n - 1] : tt.name(i) = t & \\ \Rightarrow c'.hm.typ(\underbrace{c'.hm.n - 1}_{c.hm.n}) &= i \\ va(c', p) &= (hm, c'.hm.n - 1) \\ &= (hm, c.hm.n) \\ c'.pr &= r' \end{aligned}$$

Auf dem heap wird eine neue Variable vom Typ t angelegt. Deren Bindung wird p zugewiesen (siehe Abbildung 4.18).

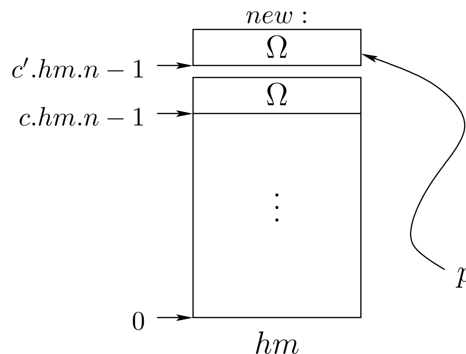


Abbildung 4.18: Ausfuhrung einer new-Anweisung

- $an: \quad id = f(e_0, \ldots, e_p - 1)$ mit $p = ft(f).np$ (Funktionsaufruf)
  Zuerst erhöhen wir die Rekursionstiefe und legen ein neues local memory für f an.

$$\begin{aligned} c'.rd &= c.rd + 1 \\ top(c') &= c'.lms(c'.rd) \\ sy(top(c')) &= ft(f).st \end{aligned}$$

Im local memory von f stehen die lokalen Variablen und die Parameter der Funktion. Die Parameter müssen noch ausgewertet und übergeben werden.

$$\begin{aligned} &\forall i \in [0 : ft(f).np - 1] : \\ &\qquad va(c', (top(c'), i)) = va(c, e_i) \end{aligned}$$

Die Bindung der Subvariable, der der Rückgabewert von f übergeben werden soll, wird im return binding stack gespeichert.

$$c'.rbs(c'.rd) = bind(c, id)$$

Als nächstes wird dann der Rumpf von f ausgeführt.

$$c'.pr = ft(f).body; r'$$

- $an: \quad return\ e$ (Rückkehr von Funktionsaufruf)
  Sei $c.rbs(c.rd) = (m, k)s$, dann bewirkt die $return$-Anweisung:

$$\begin{aligned} va((m(c'), k)s) &= va(c, e) \\ c'.rd &= c.rd - 1 \\ c'.pr &= r' \end{aligned}$$

Damit haben wir die Semantik von C_0 vollständig beschrieben. Im Folgenden soll versucht werden, auf dieser Grundlage die Korrektheit von C_0-Programmen zu beweisen.

4.5 Korrektheit von C0-Programmen

Da wir zuvor die Syntax und Semantik von C_0-Programmen definiert haben, können wir diese nun dazu benutzen formale Beweise über die Korrektheit der Ausführung u 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:

$$\begin{aligned} c^0.pr &= x = 3; if\ (x == 0)\ then\ \{x = 1\}\ else\ \{x = 2\} \\ c^0.rd &= 0 \\ c^0.rbs(0) &: \text{beliebig} \\ c^0.lms &: \{0 \mapsto fmain\} \\ c^0.hm.n &= 0 \Rightarrow c^0.hm \text{ undefiniert/leer} \\ c^0.gm.n &= 1 \\ c^0.gm.name &: \{0 \mapsto x\} \\ c^0.gm.typ &: \{0 \mapsto 0\} \\ c^0.gm.ct &: \text{undefiniert} \end{aligned}$$

Wegen der Abwesenheit von Typdefinitionen besteht die Typtabelle nur aus den elementaren Typen. Die Funktionstabelle enthält nur $main$.

$$\begin{aligned} ft(main).typ &= 0 \\ ft(main).body &= x = 3; if\ (x == 0)\ then\ \{x = 1\}\ else\{x = 2\} \\ ft(main).np &= 0 \\ ft(main).nl &= 0 \\ ft(main).st &= (n = 0, name : \emptyset, typ : \emptyset) \end{aligned}$$

Offensichtlich sollte x nach Ausführung des Programms den Wert 2 besitzen.

$$\exists t : va(c^t, x) \stackrel{!}{=} 2$$

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 C_0-Semantik der einzelnen Anweisungen verwendet.

Schrittzahl	$c.pr$	Wert
0	$x = 3; if \ldots$	$va(c, x)$: undefiniert
1	$if\ (x == 0)$	$va(c, x) = 3$
	$then\ \{x = 1\}$	
	$else\{x = 2\}$	$va(c, x == 0) = 0$
2	$x = 2$	
3	ε	$va(c, x) = 2$

Tabelle 4.4: Beispiel 1

Für $t = 3$ gilt demnach:

$$va(c^t, x) = 2 \qquad \square$$

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 f2;
  if (n<2) then {res=n} else
  {
    f1=fib(n-2);
```

```
    f2=fib(n-1);
    res=f1+f2;
  };
  return res
};
int main()
{
x=fib(2);
return -1
}
```

Es zu zeigen, dass:

$$\exists t : va(c^t, x) = 1 \wedge c^t.pr = \varepsilon$$

Startkonfiguration

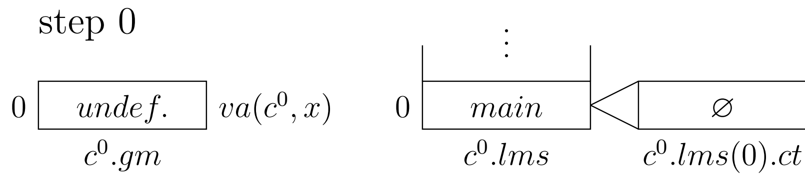
$$\begin{aligned}
c^0.pr &= x = fib(2) \\
c^0.rd &= 0 \\
c^0.rbs(0) &: \text{beliebig} \\
c^0.lms &: \{0 \mapsto fmain\} \\
c^0.hm.n &= 0 \Rightarrow c^0.hm \text{ undefiniert/leer} \\
c^0.gm.n &= 1 \\
c^0.gm.name &: \{0 \mapsto x\} \\
c^0.gm.typ &: \{0 \mapsto 0\} \\
c^0.gm.ct &: \text{undefiniert}
\end{aligned}$$

In der Typtabelle stehem nur die elementaren Typen. Die Funktionstabelle enthält die Funktionen $main$ und fib.

$$\begin{aligned}
ft(main).typ &= 0 \\
ft(main).body &= x = fib(2) \\
ft(main).np &= 0 \\
ft(main).nl &= 0 \\
ft(main).st &= (n = 0, name : \emptyset, typ : \emptyset). \\
ft(fib).typ &= 0 \\
ft(fib).body &= if\ (n < 2)\ then\ \{res = n\}\ else\ \{\ldots\}; return\ res \\
ft(fib).np &= 1 \\
ft(fib).nl &= 3 \\
ft(fib).st.n &= 4 \\
ft(fib).st.name &= \{0 \mapsto n, 1 \mapsto res, 2 \mapsto f_1, 3 \mapsto f_2\} \\
ft(fib).st.typ &= \{0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0, 3 \mapsto 0\}
\end{aligned}$$

Der local memory stack enthält nur den lokalen Speicher für $main$, der allerdings leer ist.

$$\begin{aligned}
sy(top(c^0)) &= sy(c^0.lms(c^0.rd)) \\
&= = sy(c^0.lms(0)) \\
&= ft(main).st \\
c^0.lms(0).n &= 0
\end{aligned}$$



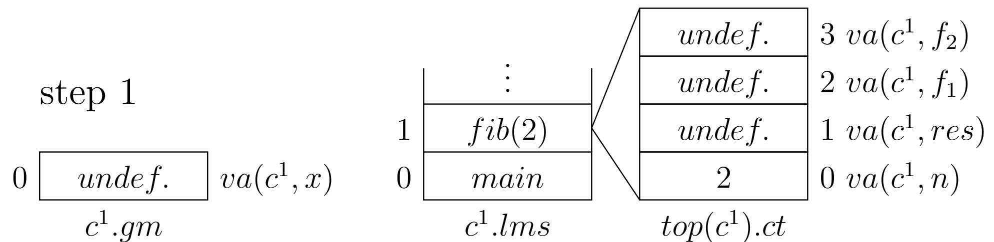
Schritt 1
$x = fib(2)$ wird ausgeführt. Mit der C_0-Semantik für Funktionsaufrufe ergibt sich:

$$\begin{aligned} c^1.pr &= if\ (n < 2)\ then\ \{res = n\}\ else\ \{\ldots\}; return\ res \\ c^1.rd &= 1 \\ c^1.rbs(1) &= bind(c^0, x) \\ &= (c.gm, 0) \end{aligned}$$

Ein neuer lokaler Speicher wird für fib auf dem stack angelegt.

$$\begin{aligned} sy(top(c^1)) &= sy(c^1.lms(c^1.rd)) \\ &= sy(c^1.lms(1)) \\ &= ft(fib).st \end{aligned}$$

In dem Speicher wird Platz für die Parameter (n) und lokalen Variablen (res, f_1, f_2) von fib reserviert. Der Wert für n wird durch die Parameterübergabe gesetzt.



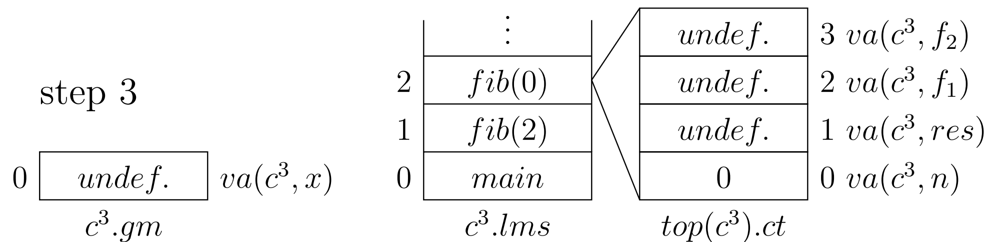
Schritt 2
Die Fallunterscheidung wird ausgeführt. Wegen $va(c^1, n) = 2 \not< 2$ ergibt sich für den Programmrest:

$$c^2.pr \;=\; f_1 = fib(n-2); f_2 = fib(n-1); res = f_1 + f_2; return\ res$$

Schritt 3
Ein weiterer Funktionsaufruf von fib, diesmal mit $va(c^2, n) = 0$ als Parameter:

$$\begin{aligned} c^3.pr &= if\ (n < 2)\ then\ \{res = n\}\ else\ \{\ldots\}; return\ res; \\ &\quad\ f_2 = fib(n-1); res = f_1 + f_2; return\ res \\ c^3.rd &= 2 \\ c^3.rbs(2) &= bind(c^2, f_1) \\ &= (c.lms(1), 2) \end{aligned}$$

Der neue top-frame ist $c^3.lms(2)$ und es gilt $sy(c^3.lms(2)) = ft(fib).st$.



Schritt 4
Wieder wird eine if-$then$-$else$-Anweisung ausgewertet. Diesmal wird jedoch durch $va(c^3, n) = 0 < 2$ der $then$-Teil zum Programmrest hinzugefügt.

$$c^4.pr = res = n; return\ res; f_2 = fib(n-1); res = f_1 + f_2; return\ res$$

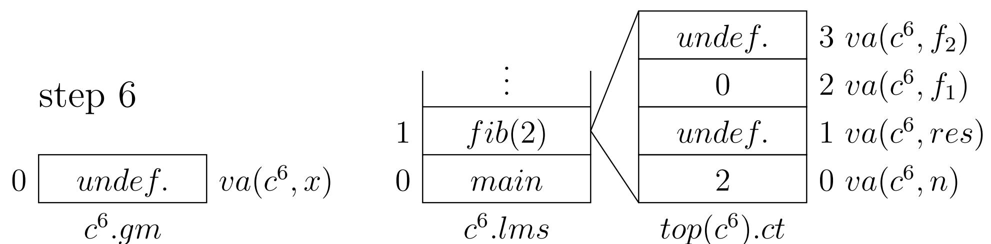
Schritt 5
Aufgrund der Zuweisung $res = n$ gilt:

$$\begin{aligned} va(c^5, res) &= va(c^5.lms(2), 0) \\ &= va(c^4, n) \\ &= 0 \\ c^5.pr &= return\ res; f_2 = fib(n-1); res = f_1 + f_2; return\ res \end{aligned}$$

Schritt 6
Das Ergebnis von $fib(0)$ wird durch die $return$-Anweisung an die aufrufende Funktion $fib(2)$ zurückgegeben.

$$\begin{aligned} c^5.rbs(c^5.rd) &= c^5.rbs(2) \\ &= (c.lms(1), 2) \\ va(c^6.lms(1), 2) &= va(c^5, res) \\ &= 0 \\ c^6.rd &= 1 \\ c^6.rbs(c^6.rd) &= c^6.rbs(1) \\ &= (c.gm, 0) \\ c^6.pr &= f_2 = fib(n-1); res = f_1 + f_2; return\ res \end{aligned}$$

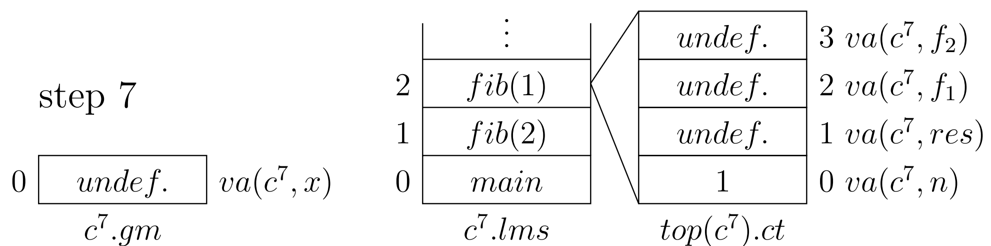
$fib(2)$ liegt nun wieder oben auf dem stack.



Schritt 7
Noch ein Funktionsaufruf von fib, dieses Mal mit $va(c^6, n) = 1$ als Parameter:

$$\begin{aligned} c^7.pr &= if\ (n < 2)\ then\ \{res = n\}\ else\ \{\ldots\}; res = f_1 + f_2; return\ res; \\ c^7.rd &= 2 \\ c^7.rbs(2) &= bind(c^6, f_2) \\ &= (c.lms(1), 3) \end{aligned}$$

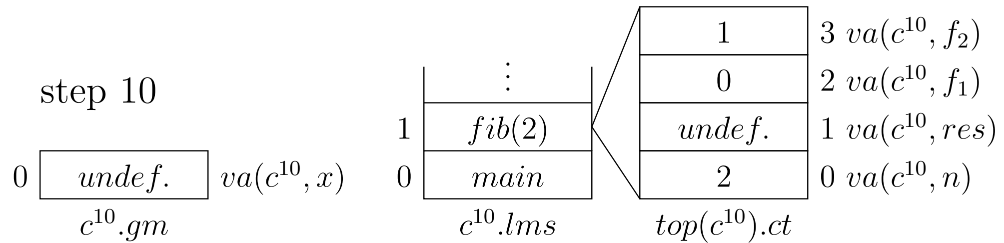
Der oberste local memory ist nun der zu $fib(1)$ gehörende $c^7.lms(2)$.



Schritte 8-10
Diese Schritte sind analog zu den Schritten 4-6. Es resultieren die folgenden Veränderungen:

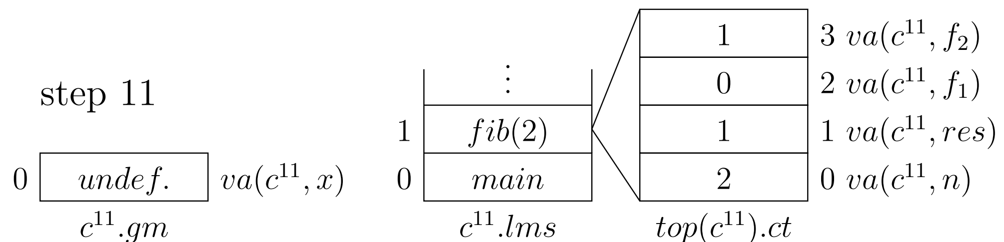
$$\begin{aligned} c^{10}.pr &= res = f_1 + f_2; return\ res; \\ c^{10}.rd &= 1 \\ c^{10}.rbs(1) &= (c.gm, 0) \\ va(c^{10}.lms(1), 3) &= va(c^9, res) \\ &= 1 \end{aligned}$$

Der top-frame gehört wieder zu $fib(2)$.



Schritt 11
res wird der Wert des Ausdrucks $f_1 + f_2$ zugewiesen:

$$\begin{aligned} va(c^{11}, res) &= va(c^{10}, f_1 + f_2) \\ &= va(c^{10}, f_1) + va(c^{10}, f_2) \\ &= 0 + 1 \\ &= 1 \\ c^{11}.pr &= return\ res \end{aligned}$$

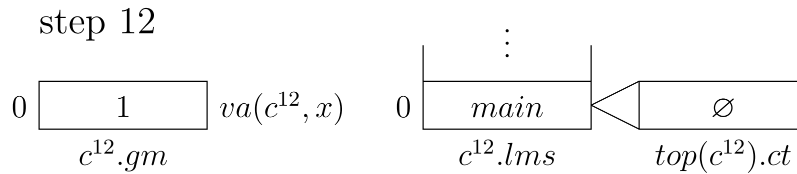


Schritt 12
Die Funktion $fib(2)$ terminiert und das Ergebnis wird an $main$ zurückgegeben.

$$\begin{aligned} c^{11}.rbs(c^{11}.rd) &= (c.gm, 0) \\ va(c^{12}.gm, 0) &= va(c^{11}, res) \\ &= va(c^{11}.lms(1), 1) \\ &= 1 \\ c^{12}.rd &= 0 \\ c^{12}.pr &= \varepsilon \end{aligned}$$

Für $t = 12$ gilt also:

$$\begin{aligned} va(c^t, x) &= 1 \\ c^t.pr &= \varepsilon \qquad \square \end{aligned}$$



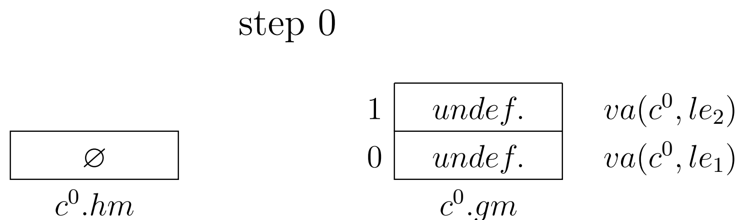
Die vorliegende Beweisführung wirkt in gewisser Weise „mechanisch“, da man lediglich nach einem festen Schema die C_0-Semantik Schritt für Schritt auf das gegebene Programm anwendet. Die Vermutung liegt nahe, dass dies auch automatisch von einem computergestützten Beweissystem ausgeführt werden kann. Das Verisoft-Projekt beschäftigt 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 C_0-Programm gegeben:

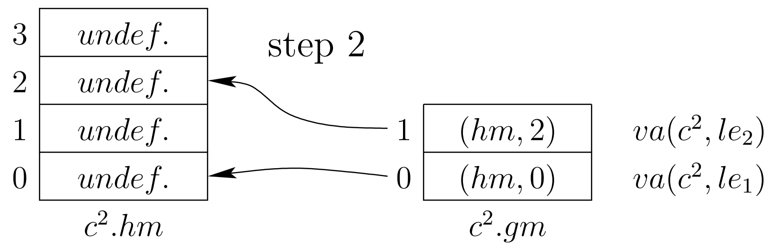
```
typedef LEL* u;
typedef struct{int content, u next} LEL;
u le1;
u le2;
int main()
{
  le1=new LEL*;
  le2=new LEL*;
  le1*.content=1;
  le1*.next=le2;
  le2*.content=2;
  le2*.next=null;
  return -1
}
```

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.

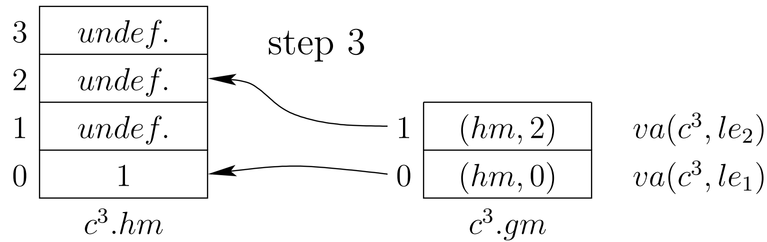


Nach den ersten beiden Schritten wurden zwei Variablen vom Typ LEL auf dem heap erzeugt und le_1 bzw. le_2 zugewiesen. Der zugehörige Speicherplatz auf dem heap ist nun reserviert und der heap ist gewachsen.

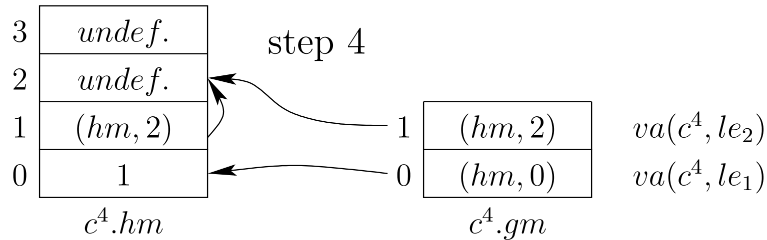
$$\begin{aligned}
c^2.hm.n &= c^1.hm.n + 1 \\
&= c^0.hm.n + 1 + 1 \\
&= 2 \\
size(c^2.hm) &= size(c^2.hm.typ(0)) + size(c^2.hm.typ(1)) \\
&= size(LEL) + size(LEL) \\
&= 4
\end{aligned}$$



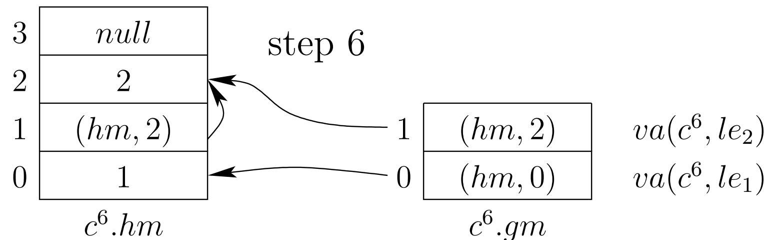
Die einzelnen Komponenten erhalten nun Werte, z.B. $le_1.content = 1$.



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.

$$\begin{aligned} \exists t : va(c^t, result) &= \sum_{k=1}^{N} k \\ c^t.pr &= \varepsilon \end{aligned}$$

Erste Idee: Man führt eine Induktion über die Schleifendurchgänge durch.
Induktionsbehauptung: nach dem j-ten Durchlauf der Schleife gilt:

$$\begin{aligned} va(c^{1+3j}, n) &= N - j \\ va(c^{1+3j}, result) &= \sum_{k=N-j+1}^{N} k \end{aligned}$$

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:

$$\begin{aligned} c^{1+3j+1}.pr &= result = result + n; n = n - 1; while\ (\ldots)\ do\ \{\ldots\} \\ va(c^{1+3j+3}, n) &= N - j - 1 \quad ✓ \\ va(c^{1+3j+3}, result) &= \sum_{k=N-j+1}^{N} k + (N - j) \\ &= \sum_{k=N-j}^{N} k \quad ✓ \\ c^{1+3j+3}.pr &= while\ (\ldots)\ do\ \{\ldots\} \neq \varepsilon \quad ✗ \end{aligned}$$

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 über einzelne Durchläufe nicht zielführend. Man muss dagegen über die Anzahl der verbleibenden Schleifendurchläufe argumentieren.
Korrekter 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.

$$c'.pr = \underbrace{a}_{\text{1. Durchlauf}} ; \underbrace{while\ (e)\ do\ \{a\}}_{N-1\ \text{Durchläufe}}$$

Induktionsbehauptung: Die Induktion wird nun über die Anzahl der verbleibenden Schleifendurchgänge M durchgeführt. Sei

$$\begin{aligned} va(c^\alpha, result) &= K \\ va(c^\alpha, n) &= M \\ c^\alpha.pr &= while\ (\ldots)\ do\ \{\ldots\}, \end{aligned}$$

dann folgt für die Konfiguration nach M Durchläufen $c^{\alpha+3M}$:

$$\begin{aligned} va(c^{\alpha+3M}, result) &= K + \underbrace{M + M - 1 + \ldots + 1}_{\sum_{i=1}^{M} i} \\ va(c^{\alpha+3M}, n) &= 0 \\ c^{\alpha+3M}.pr &= while\ (\ldots)\ do\ \{\ldots\} \end{aligned}$$

Der Beweis mit Induktionsanfang für $M = 0$ und Induktionsschritt $M \rightarrow M + 1$ ist nun leicht zu bewältigen und soll in Aufgabe 3 des achten Übungsblattes gezeigt werden. Weil n jetzt den Wert 0 hat, wird die Schleife mit dem nächsten Ausführungsschritt auf $c^{\alpha+3M}.pr = while\ (n > 0)\ do\ \{\ldots\}$ terminieren. Dann gilt

$$c^{\alpha+3M+1}.pr = \varepsilon \tag{4.1}$$

und die Korrektheitsbehauptung gilt für $t = \alpha + 3M + 1$.

4.6 C0-Compiler

Um C_0-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 $\delta_C(c^i) = c^{i+1}$ - C_0-Rechnung
- $d^0, d^1, d^2, \ldots, d^{s(i)}$ mit $\delta_D(d^i) = d^{i+1}$ - DLX-Rechnung

i C_0-Anweisungen werden in $s(i)$ Schritten der DLX simuliert. Der erzeugte Assemblercode zu einem C_0-Programm p wird mit $code(p)$ bezeichnet. Seine Erzeugung erfolgt größtenteils gemäß einer induktiven Definition über die Ableitungsbäume der Funktionsrümpfe.

4.6.1 Korrektheit

Natürlich möchte man erreichen, dass ein C_0-Programm genau das aus der C_0-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 C_0- 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 C_0-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 ebenfalls zustandsabhängig. Nun kann man den Simulationssatz aufstellen.

Theorem 4.2 *(Simulationssatz)* $\exists aba^i, \exists s(i)$:

$$\forall i : consis(c^i, aba^i, d^{s(i)})$$

Die Simulation erfolgt dann Schritt für Schritt und man muss die Umsetzung der Rechnung in der DLX untersuchen. Der Speicher der DLX wird dabei in verschiedene Bereiche unterteilt. Diese entsprechen den Komponenten der C_0-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 $i \in [0 : c.rd]$.

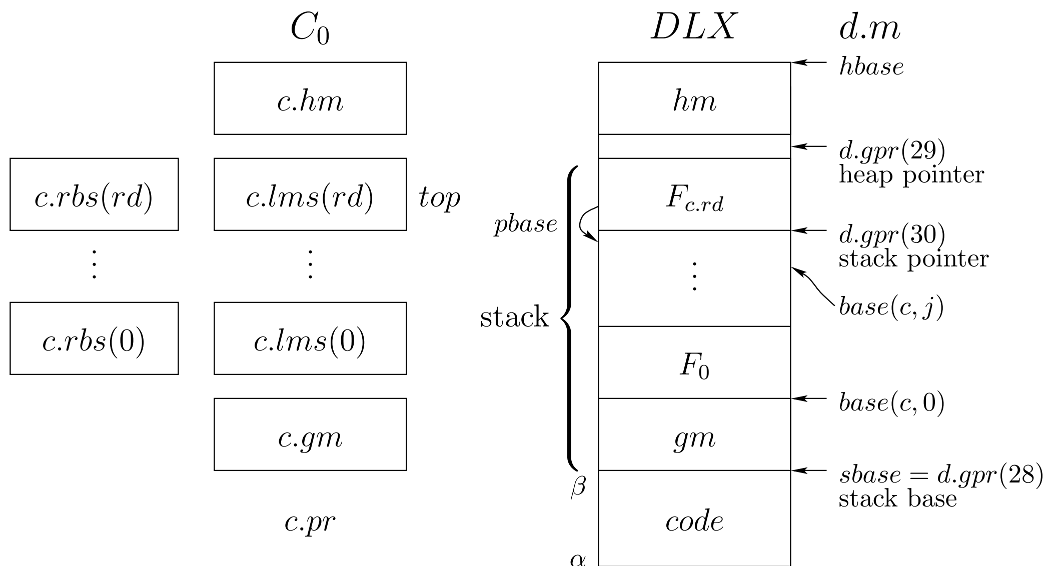


Abbildung 4.19: DLX simuliert C_0-Konfiguration

- hm - Dieser Speicherabschnitt repräsentiert den heap memory.

Abbildung 4.19 stellt die Speicheraufteilung schematisch dar und stellt sie der C_0-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 $d.gpr(28)$ gespeichert wird. Der heap befindet sich am „oberen Rand" des Speichers und wächst „nach unten" auf den stack zu. Der *heap pointer* in $d.gpr(29)$ zeigt stets auf die erste (oberste) freie Adresse unterhalb des heap memory. Der stack wächst hingegen „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.

Die 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 $d.gpr(30)$. So ist immer bekannt, welche Funktion gerade ausgeführt wird. 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 4.20 dargestellt.
Angenommen, F_j entstand durch den Aufruf der Funktion f im C_0-Programm, dann kodiert F_j den local memory $c.lms(j)$ mit

$$ft(f).st = sy(c.lms(j))$$

Außerdem enthält ein function frame, wie aus der Grafik ersichtlich ist, noch drei weitere Wörter, die den Parametern und lokalen Variablen vorangestellt sind, nämlich:

- $F_j.pbase$ (**p**redecessor **base**) - Hier wird $base(c, j-1)$ gespeichert.
- $F_j.rb$ (**r**eturn **b**inding) - Dies ist das Pendant zu $c.rbs(j)$. Es speichert die allocated base address der Variable, an die das Ergebnis von f zurückgegebenwerden soll.
- $F_j.ra$ (**r**eturn **a**ddress) - Hier wird gespeichert, zu welcher Stelle in $code$ nach der Ausführung der Funktion zurückgesprungen werden soll.

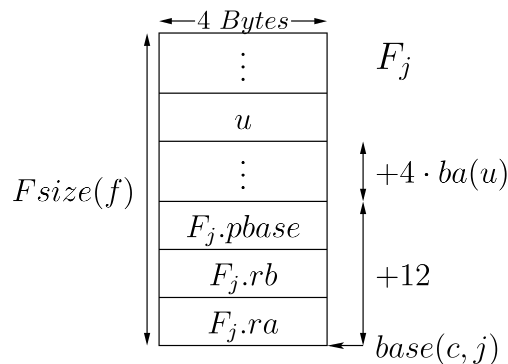


Abbildung 4.20: function frame F_j

Die Größe des $Fsize(f)$ des function frame F_j zu Funktion f beträgt dann:

$$Fsize(f) = 4 \cdot (3 + size(c.lms(j)))$$

Sei $(lms(j), i)$ ein Parameter oder eine lokale Variable von f mit dem Namen $X = lms(j).name(i)$. Dann hängt die Basisadresse $ba(lms(j), i)$ nur von der Symboltabelle $ft(f).st$ ab. Sie ist unabhängig von j und dem Zustand des local memory stack, insbesondere davon, wo im lms der zum Funktionsaufruf zugehörige lokale Speicher liegt. Es gilt:

$$ba(c.lms(j), i) = \sum_{k<i} size(c.lms(j).typ(k))$$

Ferner lässt sich das displacement von Parametern bzw. Variablen $X = ft(f).name(i)$ in Funktionen f wie folgt definieren.

$$displ(f, X) = \sum_{k<i} size(ft(f).st.typ(k))$$

Man beachte, dass diese Definition, im Gegensatz zu der von ba nicht auf dem Speicher, sondern allein auf der Funktionstabelle beruht. Dann kann man zeigen:

$$ba(lms(j), i) = displ(f, X)$$

Dies folgt direkt aus der Definition $sy(lms(j)) = ft(f).st$ und es gilt insbesondere:

$$\begin{aligned} lms(j).name &= ft(f).st.name \\ lms(j).typ &= ft(f).st.typ \end{aligned}$$

Diese 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 lässt sich für globale Variablen in gm zeigen. Man muss lediglich $displ(gm, X)$ definieren, der Rest des Beweises verläuft analog zur obigen Vorgehensweise. Dies soll ausführlicher in Aufgabe 4 des neunten Übungsblattes nachvollzogen werden.

Wir betrachten nun $u = (lms(j), i)s$, eine lokale Subvariable der Funktion f. Wegen $sy(c.lms(j)) = ft(f).st$ hängt der Aufbau des local memory und somit die Basisadresse $ba(u)$ nur von der Funktion f ab und nicht etwa von j. Es kommt nur auf die Definition von f an, die in den Funktionsdeklarationen des C_0-Programms festgelegt wurde. Dann kann die allocated base address von u folgendermaßen ermittelt werden.

Definition 4.10 *aba berechnet sich für lokale Variablen und Parameter u zu:*

$$aba(c, u) = base(c, j) + 4 \cdot (3 + ba(u))$$

Dabei ist die Basisadresse des j-ten function frame:

$$base(c,j) = 4 \cdot (size(gm) + \sum_{i<j}(3 + size(c.lms(i))))$$

Definition 4.11 *Ist $u = (gm,i)s$ eine globale Subvariable, so gilt:*

$$aba(c,u) = sbase + 4 \cdot ba(u)$$

Definition 4.12 *Ist id ein Identifier, so gilt:*

$$aba(c,id) = aba(c,bind(c,id))$$

Die Notation von aba ist für Identifier und Variablen überladen.

Dann gilt für einen Identifier X mit $bind(c,X) = u$ (lokal oder global) und $sy(lms(c.rd)) = ft(f).st$ allgemein:

$$\begin{aligned} aba(c,X) &= aba(c,bind(c,X)) \quad \text{(Definition 4.12)} \\ &= \begin{cases} base(c,c.rd) + 4 \cdot (3 + displ(f,X)) & : \quad X \text{ ist lokal} \\ sbase + 4 \cdot displ(gm,X) & : \quad \text{sonst} \end{cases} \\ &= \begin{cases} base(c,c.rd) + 4 \cdot (3 + ba(u)) & : \quad u \text{ ist lokal} \quad (ba(lms(j),i) = displ(f,X)) \\ sbase + 4 \cdot ba(u) & : \quad \text{sonst} \quad (ba(gm,i) = displ(gm,X)) \end{cases} \\ &= aba(c,u) \quad \text{(Definition 4.10, 4.11)} \end{aligned}$$

Die Faktoren 4 ergeben sich dabei aus aus der unterschiedlichen Addressierung von C_0- und DLX-Speicher. Nun, da aba definiert ist, können die Konsistenzbedingungen an den Compiler formal aufgeschrieben werden.

Definition 4.13 *(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)*
- Programmintegrität - $d.m_{\beta-\alpha+1}(\alpha) = code(p)$

Definition 4.14 *(e-consis) $\mathrm{e-consis}$ liegt vor, wenn für alle Identifier id mit $typ(c,id)$ elementar gilt:*

$$va(c,id) = d.m_4(aba(c,id))$$

Das bedeutet, dass alle elementaren Werte und somit auch alle Werte komplexer Variablen in DLX- und C_0-Maschine identisch sind.

Definition 4.15 *(p-consis) $\mathrm{p-consis}$ liegt vor, wenn für alle Identifier p mit $typ(c,p)$ pointer, also mit $va(c,p) = u$ (u ist eine Subvariable), gilt:*

$$d.m_4(aba(c,p)) = aba(c,u)$$

Das bedeutet, dass alle pointer in DLX- und C_0-Maschine auf die entsprechenden selben Variablen zeigen. Abbildung 4.21 veranschaulicht den Sachverhalt.

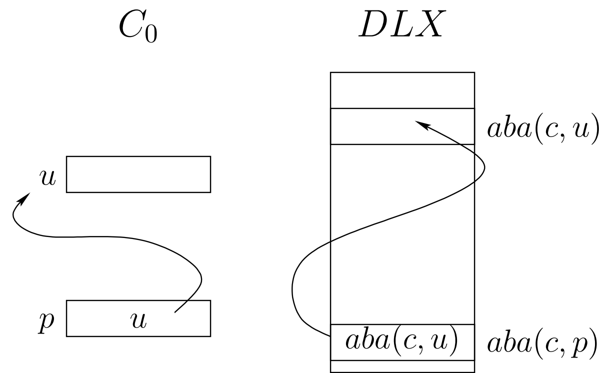


Abbildung 4.21: $\mathrm{p-consis}$

Sei a eine C_0-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 $start(an)$ kennzeichnen wollen. Dabei gilt:

$$\alpha \leq start(an), start(an) + 4, \ldots \leq \beta$$

Die Konsistenz der Programmflusskontrolle lässt sich dann leicht definierten.

Definition 4.16 *(c-consis)* $\mathrm{c-consis}$ *liegt vor, wenn für den Programmrest* $c.pr = an; r'$ *gilt:*

$$d.pc = start(head(c.pr))$$

Das bedeutet, dass in DLX- und C_0*-Maschine dieselbe Anweisung ausgeführt wird.* $d.pc$ *zeigt auf die Startadresse des compilierten Codes der ersten Anweisung des aktuellen Programmrests.*

Definition 4.17 *(r-consis)* $\mathrm{r-consis}$ *beschreibt die Konsistenz der Stackverwaltung. Dies beinhaltet im Einzelnen:*

- $d.gpr(28) = sbase > \beta$ *(stack base)*
- $d.gpr(29) > base(c, c.rd) + 4 \cdot (3 + size(top(c)))$ *(heap pointer) - heap und stack überlappen sich nicht.*
- $d.gpr(30) = base(c, c.rd)$ *(stack pointer)*
- $\forall j > 0 : d.m_4(base(c, j) + 8) = base(c, j - 1)$ *(pbase)*
- $\forall j > 0 : d.m_4(base(c, j) + 4) = aba(c, c.rbs(j))$ *(rb)*
- *Falls* $c.pr = a; return\ e; r'$ *und* $return \notin a$*, dann ist* $d.m_4(base(c, j)) = start(head(r'))$ *(ra)*

Sind diese Bedingungen erfüllt, so sind die DLX- und C_0-Konfigurationen zueinander konsistent.

4.6.2 Ausdrucksübersetzung

Wie bereits beschrieben, enthalten C_0-Anweisungen Ausdrücke, die die Parameter für deren Ausführung 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 es stellt sich die Frage, wie groß Ausdrücke werden dürfen, so dass man sie noch auswerten kann. Die C_0-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 mit Hilfe einer geschickten Herangehensweise Ausdrücke mit einer gewissen Anzahl von Operatoren unter Benutzung einer logarithmischen Anzahl von Registern auswerten kann. Dazu führen wir den Aho-Ullmann-Algorithmus ein.

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 4.22 erklärt die Regeln in schematischer Art und Weise. Ziel des Spieles ist, die Wurzel zu markieren, beziehungsweise

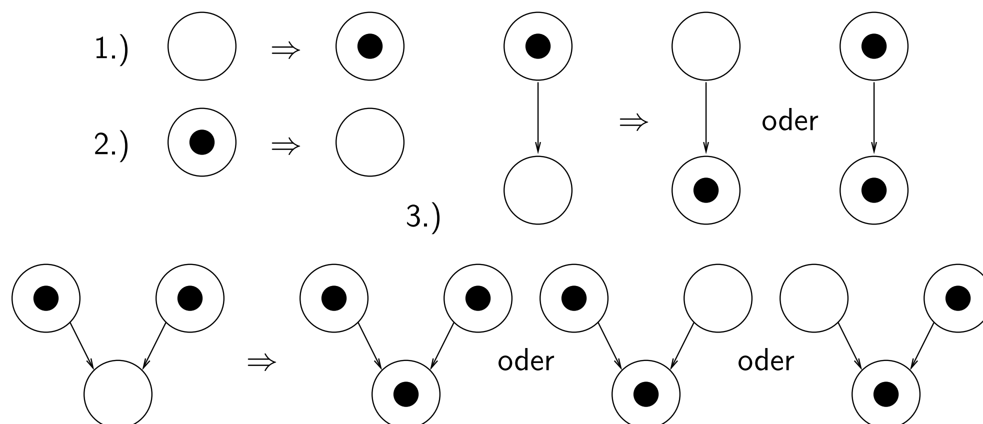
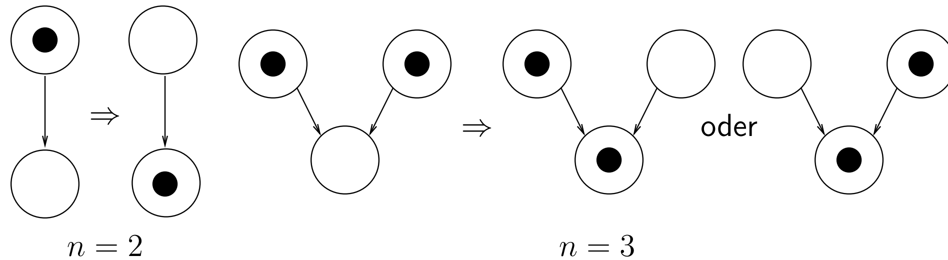


Abbildung 4.22: Marken-Spiel zum Aho-Ullman-Algorithmus

für gerichtete azyklische Graphen, ausgehend von den Quellen, alle Senken zu markieren.

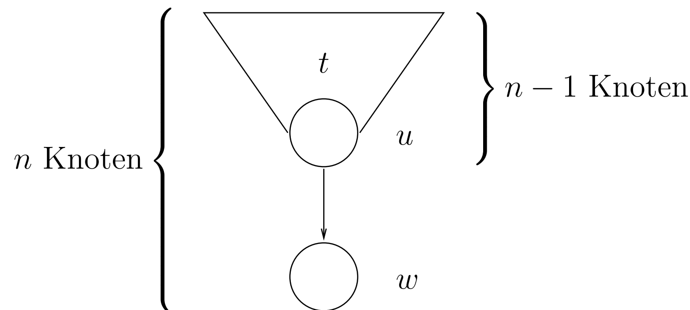
Theorem 4.3 *Man kann jeden binären Baum, der $n > 1$ Knoten besitzt, mit $\lceil \log_2 n \rceil$ Marken markieren.*

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 $T(3) = 2 \leq \lceil 1.58496 \rceil \leq \lceil \log_2 3 \rceil$. Die Behauptung stimmt also für den Induktionsanfang. Beim Induktionsschritt $n - 1 \rightarrow n$ müssen zwei Fälle unterschieden werden.

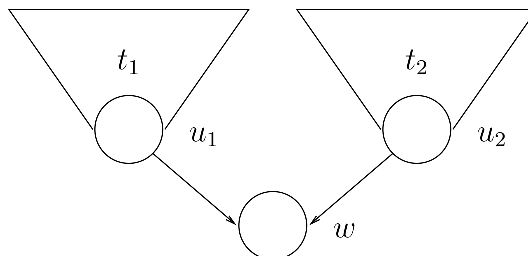
Fall 1: Die Wurzel hat nur ein Kind.



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:

$$T(n) \leq T(n-1)$$

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:

$$\begin{aligned} \text{Größe von } t_1 &\leq n - 2 \\ \text{Größe von } t_2 &\leq \frac{n}{2} \end{aligned}$$

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 u_2 mit 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:

$$T(n) \leq \max\{T(n-2), T(\frac{n}{2}) + 1, 2\} \overset{!}{\leq} \lceil \log_2 n \rceil$$

IV:

$$\forall j, 1 < j < n : T(j) \leq \lceil \log_2 n \rceil$$

IS: Für den ersten Fall (nur ein Kind) gilt:

$$\begin{aligned} T(n) &\leq T(n-1) \\ &\leq \lceil \log_2(n-1) \rceil \quad \text{(Induktionsvoraussetzung)} \\ &\leq \lceil \log_2 n \rceil \quad \text{(Logarithmus ist monoton wachsend)} \end{aligned}$$

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:

$$\begin{aligned} T(n) &\leq T(n-2) \\ &\leq \lceil \log_2 n \rceil \quad \checkmark \end{aligned}$$

2. $\max\{T(n-2), T(\frac{n}{2}) + 1, 2\} = T(\frac{n}{2}) + 1$, dann gilt:

$$\begin{aligned} T(n) &\leq T(\frac{n}{2}) + 1 \\ &\leq \lceil \log_2(\frac{n}{2}) \rceil + 1 \quad \text{(Induktionsvoraussetzung)} \\ &\leq \lceil \log_2 n - \log_2 2 \rceil + 1 \quad \text{(Logarithmengesetze)} \\ &\leq \lceil \log_2 n - 1 \rceil + 1 \\ &\leq \lceil \log_2 n \rceil - 1 + 1 \quad (\forall a \in \mathbb{Z} : \lceil x - a \rceil = \lceil x \rceil - a) \\ &\leq \lceil \log_2 n \rceil \quad \checkmark \end{aligned}$$

3. $\max\{T(n-2), T(\frac{n}{2}) + 1, 2\} = 2$, dann gilt für $n \geq 3$:

$$\begin{aligned} T(n) &\leq 2 \\ &\leq \lceil \log_2 3 \rceil \\ &\leq \lceil \log_2 n \rceil \quad \checkmark \end{aligned}$$

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 $\circ \in \{+, -_2, *, /, \wedge, \ldots\}$ 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 C_0-Programms umgesetzt werden. Dabei werden Register in Anspruch genommen um die Zwischenergebnisse zu speichern. Dies entspricht dem Markieren eine Knotens im Markenspiel. Liegt Marke $j \in \{1, 2, \ldots\}$ auf Knoten v, so bedeutet das, dass das zugehörige Ergebnis in $gpr(j)$ gespeichert wird.
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. Nun soll ein Ausdruck e ausgewertet werden. In c geschieht dies gemäß der C_0-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:

$$d_t.gpr(j) = \begin{cases} va(c, u) & : \quad R(u) = 1 \wedge u \text{ ist kein pointer} \\ aba(bind(c, u)) & : \quad R(u) = 0 \quad (u \text{ ist ein Identifier}) \\ aba(va(c, u)) & : \quad R(u) = 1 \wedge u \text{ ist ein pointer} \end{cases}$$

Für Pointer Beziehungsweise Identifier wird also die Basisadresse der zugehörigen Variablen gespeichert. Die Behauptung beschreibt das gewünschte Ergebnis der Ausdrucksauswertung.

$$u \bigcirc \xrightarrow{\text{Zug } t} \textcircled{j} \; u$$

Um dies zu erreichen, muss eine Fallunterscheidung über $ecode(t)$, das heißt über den in Zug t auszuwertenden C_0-Ausdruck u durchgeführt werden.

- $u = X$, X ist der Name einer Variablen
  1. X ist lokale Variable der Funktion f, dann muss die Basisadresse über den stack pointer und das displacement berechnet werden.

     $$d_t.gpr(j) = d_t.gpr(30) + 4 \cdot displ(f, X) + 12$$

     Es wird die folgende Instruktion erzeugt.
     - `addi` j `30` $4 \cdot displ(f, X) + 12$
  2. X ist eine globale Variable, dann muss die Basisadresse über die stack base und das displacement berechnet werden.

     $$d_t.gpr(j) = d_t.gpr(28) + 4 \cdot displ(gm, X)$$

     Es wird die folgende Instruktion erzeugt.

– `addi` j 28 $4 \cdot displ(gm, X)$

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.

$$d_t.gpr(j) = d_{t-1}.m_4(gpr(j))$$

Die entsprechende zusätzliche Instruktion lautet:

– `lw` j j 0

- $u = c_0$, 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.

$$d_t.gpr(j) = b$$

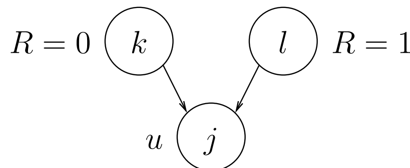
In der DLX benötigt dies 2 Instruktionen, da die Konstante bis zu 32 Bit in Anspruch nimmt und immediate-Konstanten lediglich 16 Bit fassen können. Der Code lautet:

– `lhgi` j $b[31:16] \oplus b[15]$

– `xori` j j $b[15:0]$

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.

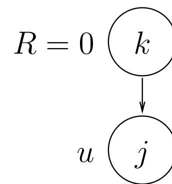
$$\begin{aligned} d_{t-1}.gpr(k) &= aba(c, bind(c, e')) \\ d_{t-1}.gpr(l) &= va(c, e') \end{aligned}$$

Dann kann die Bindung von $e[e']$ für $R(u) = 0$ bestimmt werden.

$$d_t.gpr(j) = d_{t-1}.gpr(k) + 4 \cdot d_{t-1}.gpr(l) \cdot size(t')$$

Man beachte die beiden beiden Multiplikationen in dem obigen Ausdruck. Diese werden 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. Dies ist Thema von Aufgabe 4 des zehnten Übungsblatts.
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 das displacement der i-ten Komponente aufaddiert werden.

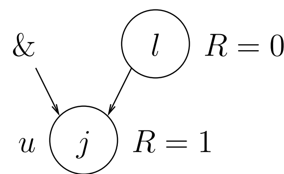
$$\begin{aligned} d_{t-1}.gpr(k) &= aba(c, bind(c, e)) \\ d_t.gpr(j) &= d_{t-1}.gpr(k) + 4 \cdot displ(i, t) \end{aligned}$$

Man kann $k = j$ wählen (Marke schieben) und keine weiteres Register wird benötigt. Die Multiplikation mit vier wird hier im Code durch zweimaliges Verdoppeln realisiert. Gegebenenfalls ($R(u) = 1$) muss man die Adresse noch dereferenzieren.

- $u = e*$, e ist ein Pointer der dereferenziert werden soll.
  Liegt der Wert des Indentifiers e ($R(e) = 0$) in Register j vor, so kann 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 bereits das gesuchte Ergebnis vorliegen, und man kann es direkt an Register j übergeben. Wählt man $l = j$ (Marke schieben), so muss garkein Code erzeugt werden.

Es kann nun Code generiert werden, um alle möglichen C_0-Identifier auszuwerten. Die C_0-Ausdrucksauswertung für arithmetische und Boolesche Ausdrücke geschieht dann trivial über die entsprechenden DLX-Instruktionen für die jeweiligen Operationen. Am Ende 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 $code(e) = ecode(e) = Ecode(T)$ 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.

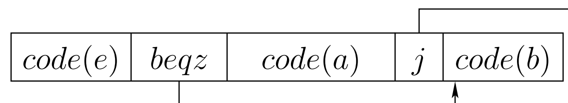
- $an: \quad e = e'$ (Zuweisung)
Es gilt $R(e) = 0$ und $R(e') = 1$. $code(e)$ und $code(e')$ müssen zuvor ausgeführt werden und speichern die Adresse von e und den Wert von e' in Register j bzw. $k \neq j$. Bei der Auswertung von e' darf dann selbstverständlich das Register j nicht verwendet werden. Die Konfiguration d' nach Ausführung von $code(e)$ $code(e')$ lautet dann:

$$\begin{aligned} d'.gpr(j) &= aba(c,e) \\ d'.gpr(k) &= \begin{cases} va(c,e') & : \quad e' \text{ ist kein pointer} \\ aba(c, va(c,e')) & : \quad e' \text{ ist ein pointer} \end{cases} \end{aligned}$$

Dann muss nur noch der Wert von e' der Adresse von e zugewiesen werden.

  – `sw` $k \quad j \quad 0$

- $an: \quad if\ e\ then\ \{a\}\ else\ \{b\}$ (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 $l(code(A))$ die Länge des für Anweisungsfolge A erzeugten Codes an, dann wird für die if-Anweisung der folgende Code produziert.

  – $code(e)$
  – `beqz` $k \quad l(code(a)) + 8$
  – $code(a)$
  – `j` $l(code(b)) + 4$
  – $code(b)$

- $an: \quad while\ e\ do\ \{a\}$ (Schleife)
Die Übersetzung der $while$-Schleife soll in Aufgabe 3 des zehnten Übungsblatts durchgeführt werden.

- $an: \quad id = f(e_1, \ldots, e_p)$ (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:

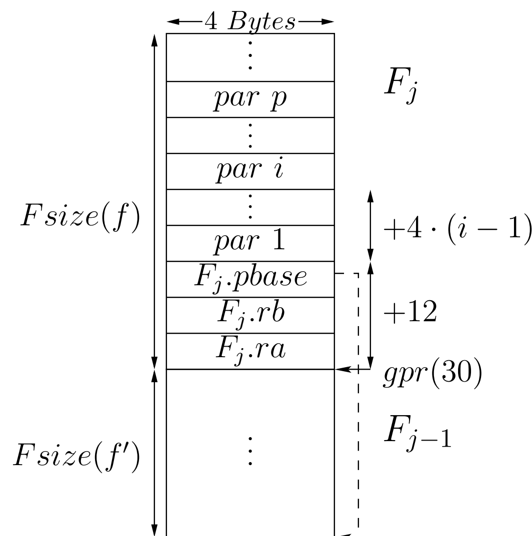
  $$gpr(30) + Fsize(f') + Fsize(f) \overset{!}{<} gpr(29)$$

  Dazu muss die Distanz zwischen heap und stack nach dem geplanten Aufruf ausgewertet werden. Der Test-Code lautet:

     – `addi 1 30` $Fsize(f') + Fsize(f)$
     – `subi 1 29 1`
     – `sgri 1 1 0`
     – `bnez 1 8`
     – $code(Abbruch)$

Der Abbruch-Code ist eine $trap$-Instruktion, die den Betriebssystemkern aufruft. Dieser kann dann zum Beispiel dem Programm mehr Speicher zuweisen.

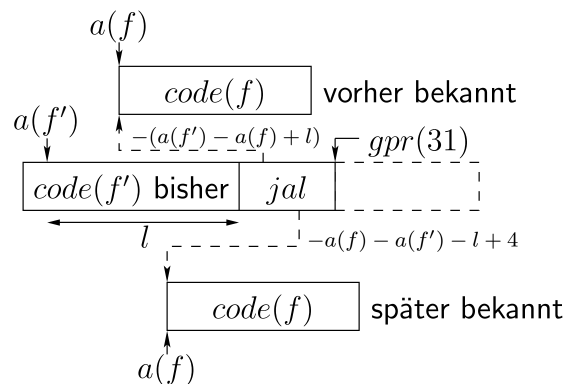
2. Die Parameter müssen übergeben werden. Dabei ist zu beachten, dass nur Parameter mit einfachen Typen übergeben werden können. Da in der Parameterliste Ausdrücke stehen, müssen diese zuerst zu $pcode(1) \cdots pcode(p)$ übersetzt und jeweils in Register k_i ausgewertet werden. Es gilt $R(e_i) = 1$, dass heißt, das bei C_0-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:
   - $ecode(e_i)$
   - `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` $Fsize(f') + 8$



4. Die Adresse der Subvariable id, die den Rückgabewert erhalten soll, muss in rb gespeichert werden ($F_j.rb = rbs(j)$). Dazu wird zuerst durch $code(id)$ die allocated base address von id bestimmt ($R(id) = 0$) und in Register k gespeichert. Der generierte Code lautet:
   - $code(id)$
   - `sw` k `30` $Fsize(f') + 4$
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

   $$a(f) = start(code(ft(f).body)),$$

   die die Startadresse von $code(f) = code(ft(f).body)$ 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` $-(a(f') - a(f) + l)$

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 Pässen übersetzt. Im ersten Pass wird alles soweit wie möglich übersetzt, bis auf die Sprungweiten. Diese werden im zweiten Pass eingesetzt, wenn die Positionen der Funktionsrümpfe im Code bekannt ist. Die Sprunginstruktion lautet dann:

– `jal` $a(f) - a(f') - l + 4$

Für lange Sprünge kann die immediate Konstante eventuell nicht ausreichen und man muss die Sprungweite zunächst mit 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 $gpr(31)$ nach ra gerettet werden. In dieses Register hatte die $jal/jalr$-Instruktion zuvor die Rücksprungadresse gesichert.

   – `sw 31 30 0`

- $an:$ $id = new\ t*$ (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 C_0-Programme können nach der Übersetzung in Assembler auf der DLX ausgeführt werden. Sie laufen korrekt ab, solange der Compiler die Konsistenzbedingungen beachtet.

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 Benutzern vorgegaukelt wird, er würde für sich allein auf einer eigenen (virtuellen) DLX rechnen. Dabei werden alle Programme 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 C_0A, einer Erweiterung von C_0 um *inline assembler code*. Dieser ist nötig, da in reinem C_0 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
- virtual memory inklusive system/user mode

Wir beginnen mit der Einführung des Interrupt-Mechanismus.

5.1.1 Interrupts

Interrupts oder *exceptions* unterbrechen die Ausführung der aktuellen Instruktion und führen zur Ausführung eines *Interrupt handlers*, der auf den jeweiligen Interrupt reagiert und entsprechende Instruktionen ausführt. 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.

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

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.
- ill - Eine illegale Instruktion sollte ausgeführt werden.
- 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.
- 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 $c.spr : \{0,1\}^5 \longrightarrow \{0,1\}^{32}$.

$$c = (c.pc, c.gpr, c.m, c.spr)$$

Darin dienen die ersten fünf Register der Interruptbehandlung. Sie sind in Tabelle 5.2 aufgeführt und haben die folgenden Funktionen:

i	$spr(i)$	name
0	SR	**S**tatus **R**egister
1	ESR	**e**xception **Ca**use
2	ECA	**e**xception **S**tatus
3	EPC	**e**xception **PC**
4	$EDATA$	**e**xception **data**

Tabelle 5.2: special purpose Register zur Interrupt-Behandlung

- 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. Dazu 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 den aktuellen programm counter um später wieder zur Ausgangsinstruktion zurückkehren zu können.
- $EDATA$ - 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.

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 $\delta_D(c, eev) = c'$ unter Berücksichtigung der Interrupt event signals.
Tritt ein Interrupt auf, so steht immer ein event signal dahinter. Wir definieren Grund (**ca**use) eines Interrupts als:

$$ca(c, eev)[j] = \begin{cases} eev[j] & : \quad j \in \{0, 7, 8, \dots, 31\} \quad \text{(extern)} \\ iev(c)[j] & : \quad \text{sonst} \quad \text{(intern)} \end{cases}$$

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.

$$mca(c, eev)[j] = \begin{cases} ca(c, eev)[j] & : \quad j \leq 5 \quad \text{(nicht maskierbar)} \\ ca(c, eev)[j] \wedge c.SR[j] & : \quad \text{sonst} \end{cases}$$

Dann wird die ISR aufgerufen, sobald eines der Bits im masked cause 1 ist, also ein event signal aktiviert worden ist, dass nicht maskiert wurde. Das Signal was die ISR aufruft heißt $JISR(c, eev)$ (**J**ump to **I**nterrupt **S**ervice **R**outine).

$$JISR(c, eev) = \bigvee_j mca(c, eev)[j]$$

Da mehrere Interrupts gleichzeitig auftreten können, definieren wir den **i**nterrupt **l**evel $il(c, eev)$, der den aufgetretenen Interrupt mit höchsten Priorität (niedrigste Ordnungsnummer) zurückgibt.

$$\exists j \in [0:31] : j = 1 \quad \Rightarrow \quad il(c, eev) = \min\{j \mid mca(c, eev)[j] = 1\}$$

Nun sollen die Auswirkungen eines Interrupts betrachtet werden. Offensichtlich gilt, dass, wenn kein Interrupt auftritt, die DLX wie gewohnt rechnet.

$$JISR(c, eev) = 0 \quad \Rightarrow \quad \delta_D(c, eev) = \delta_{D_{\text{alt}}}(c)$$

Ist $JISR(c, eev) = 1$, so treten die folgenden Effekte ein.

$$\begin{aligned}
c'.SR &= 0^{32} \\
c'.EPC &= \begin{cases} c.pc & : \quad il(c, eev) \in \{3, 4\} \quad (\text{repeat}) \\ \delta_{D_{alt}}(c).pc & : \quad \text{sonst} \end{cases} \\
c'.ECA &= mca(c, eev) \\
c'.ESR &= \begin{cases} c.gpr(RS1(c)) & : \quad il(c, eev) \geq 5 \wedge movi2s(c) \wedge SA(c) = 0^5 \\ c.SR & : \quad \text{sonst} \end{cases} \\
c'.EDATA &= \begin{cases} sxtimm(c) & : \quad il(c, eev) = 5 \quad (trap) \\ ea(c) & : \quad il(c, eev) = 4 \quad (pfls) \end{cases} \\
c'.PC &= SISR
\end{aligned}$$

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 ($SA(c) = 0^5$). 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. 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	$I(c)[5:0]$	mnemonic	name	effect
	010000	$movs2i$	**move s**pecial **to i**nteger	$c'.gpr(RD(c)) = c.spr(SA(c))$
	010001	$movi2s$	**move i**nteger **to s**pecial	$c'.spr(SA(c)) = c.gpr(RS1(c))$

Außerdem kommen noch zwei neue J-type-Instruktionen dazu.

J-type	$I(c)[5:0]$	mnemonic	name	effect
	111110	$trap$	trap	$iev(c)[5] = 1$
	111111	rfe	**r**eturn **f**rom **e**xception	$c'.SR = c.ESR, c'.PC = c.EPC,$ $c'.mode = 1$

$trap$ 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.

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.

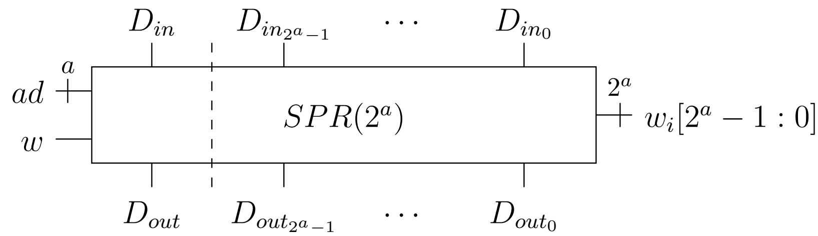


Abbildung 5.1: special purpose register file SPR

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:

$$s_1(c) = movi2s(c) \wedge SA(c) = 0^5 \wedge il(c, eev) \geq 5$$

Im Falle einer mit resume type continue unterbrochenen move-Instruktion die das Statusregister aktualisieren sollte, wird demnach wie spezifiziert A, also $h.gpr(RS1_h)$, statt dem alten SR in ESR gesichert. s_{2_h} ist die Hardware-Variante von

$$s_2(c) = (il(c, eev) = 5)$$

und signalisiert lediglich, dass ein $trap$-Interrupt ausgelöst wurde, weshalb $sxtimm_h$ in $EDATA$ zu speichern ist. s_{3_h} implementiert das select-Signal

$$\begin{aligned} s_3(c) &= (il(c, eev) \in \{3, 4\}) \\ &= (mca(c, eev)[3] \vee mca(c, eev)[4]) \wedge (\overline{\bigvee_{j \leq 2} mca(c, eev)[j]}), \end{aligned}$$

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. Dadurch kann der Benutzer den 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.

$$\begin{aligned} mw(c) &= mw_{alt}(c) \wedge (\overline{JISR \wedge (il(c, eev) \in \{3, 4\})}) \\ gpr_w(c) &= (gpr_{w_{alt}}(c) \vee movs2i(c)) \wedge \wedge(\overline{JISR \wedge (il(c, eev) \in \{3, 4\})}) \end{aligned}$$

Das GPR wird nun auch im Falle von $movs2i(c)$ geschrieben. Zuletzt muss noch die Logik zur Berechnung des neuen program counter aktualisiert werden. Die neuen Datenpfade sind in Abbildung 5.5 dargestellt.

Abbildung 5.2: Implementierung des SPR

Abbildung 5.3: Verdrahtung des SPR

Abbildung 5.4: Datenpfade zwischen GPR und SPR

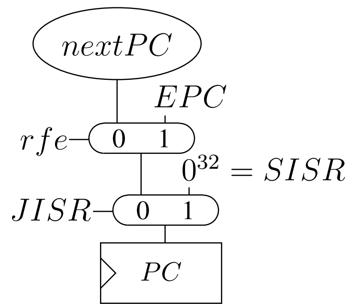


Abbildung 5.5: Erweiterung der $nextPC$-Umgebung

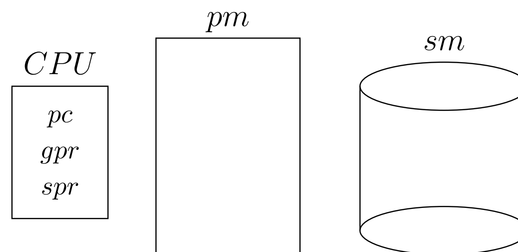


Abbildung 5.6: Die physikalische Maschine

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 dann Seiten zugewiesen und in einer page table verzeichnet. Die virtuellen Adressen verweisen dann auf Einträge in der page table mit deren Hilfe 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 drei neue special purpose register eingeführt:

i	$spr(i)$	name
5	PTO	**p**age **t**able **o**rigin
6	PTL	**p**age **t**able **l**ength
7	$MODE$	(bit 0)

Dies sind im Einzelnen:

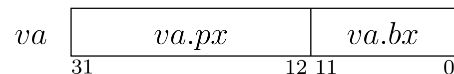
- $c.pto = c.spr[5]$ - Die Startadresse der page table des aktuellen Benutzers
- $c.ptl = c.spr[6]$ - Die Länge der page table des aktuellen Benutzers
- $c.mode = c.spr[7][0]$ - $mode$-Bit

Das $mode$-Bit gibt an in welchem Modus sich der Prozessor im Moment befindet (system/user mode).

$$c.mode = \begin{cases} 0 & : \quad \text{system mode} \\ 1 & : \quad \text{user mode} \end{cases}$$

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 $pte(c, va)$ (**p**age **t**able **e**ntry) in der page table. Die genaue Adresse des Eintrages $ptea(c, va)$ (**p**age **t**able **e**ntry **a**ddress) berechnet si zu:

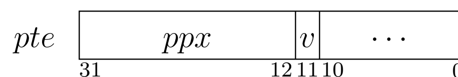
$$ptea(c, va) = c.pto +_{32} 0^{10}va.px00$$

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

$$pte(c, va) = c.pm_4(ptea(c, va))$$

Der page table entry besteht aus:

- $ppx(c, va) = pte(c, va)[31:12]$ - **p**hysical **p**age **i**ndex
- $v(c, va) = pte(c, va)[11]$ - **v**alid bit
- $pte(c, va)[10:0]$ - 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, wird ein page fault ausgelöst und die Seite muss aus dem swap memory nachgeladen werden. Ist $v(c, va) = 1$, so liegt die gewünschte Seite an der **p**hysical **m**emory **a**ddress $pma(c, va)$:

$$pma(c, va) = ppx(c, va) \circ va.bx$$

Dieser Speicherzugriff kann jedoch auch scheitern, falls der page index die Länge der page table überschreitet, also auf eine nicht zuordbarbare 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.

$$\begin{aligned} ptlef(c) &= (\langle c.pc.px \rangle \geq \langle c.ptl \rangle) \\ ptlels(c) &= (\langle ea(c).px \rangle \geq \langle c.ptl \rangle) \end{aligned}$$

Die Interrupt event signals für page faults werden dann wie folgt definiert.

$$\begin{aligned} iev[3] &= pff(c) \\ &= c.mode \wedge (ptlef(c) \vee (\overline{ptlef(c)} \wedge \bar{v}(c, c.pc))) \\ iev[4] &= pfls(c) \\ &= c.mode \wedge (lw(c) \vee sw(c)) \wedge (ptlels(c) \vee (\overline{ptlels(c)} \wedge \bar{v}(c, ea(c)))) \end{aligned}$$

Im system mode können keine page faults auftreten, da der Kern keine Adressübersetzung nutzt. Die veränderte fetch-Semantik ergibt sich zu:

$$I(c) = \begin{cases} c.pm_4(c.pc) & : \quad c.mode = 0 \quad \text{(system mode)} \\ c.pm_4(pma(c, c.pc)) & : \quad c.mode = 1 \quad \text{(user mode)} \end{cases}$$

Die neue Semantik von load/store soll selbständig definiert werden. Abbildung 5.7 veranschaulicht die Adressübersetzung.

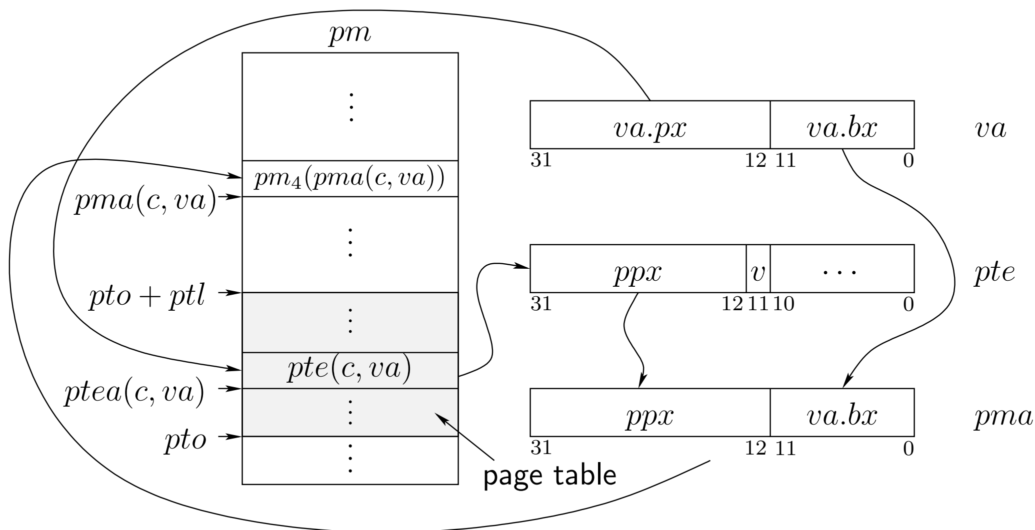
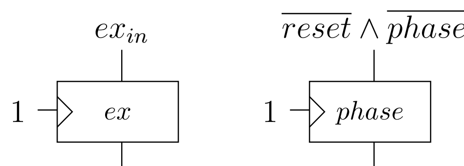


Abbildung 5.7: Adressübersetzung

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.



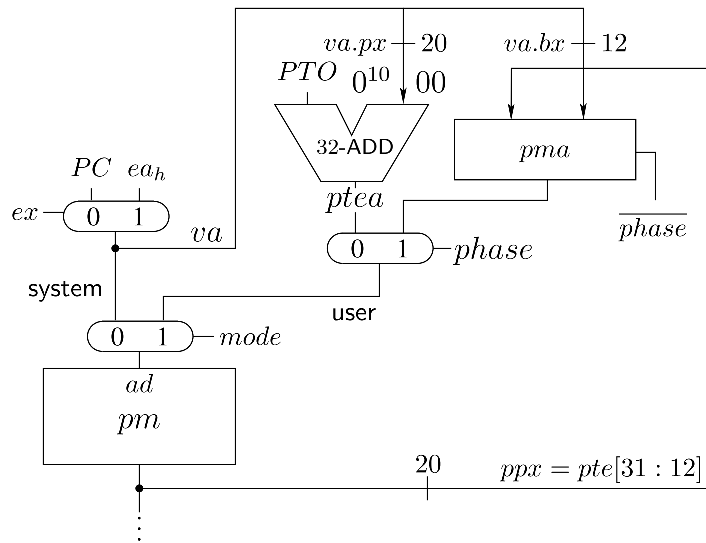
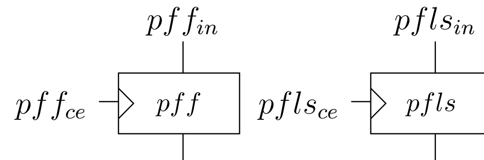


Abbildung 5.8: Datenpfade der memory management unit

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.

$$ex_{in} = \overline{reset} \wedge (\overline{MODE} \wedge \overline{ex} \vee MODE \wedge (\overline{ex} \wedge phase \vee ex \wedge \overline{phase}))$$

Auftretende page faults werden in speziellen Registern gespeichert.



Die Eingangs- und clock enable- Signale werden mit $\bar{v} = pm_{Dout}[11]$ folgendermaßen definiert

$$\begin{aligned} pff_{in} &= MODE \wedge (ptlef_h \vee \bar{v}) \\ pfls_{in} &= MODE \wedge (ptlels_h \vee \bar{v}) \\ pff_{ce} &= \overline{ex} \wedge \overline{phase} \\ pfls_{ce} &= (lw_h \vee sw_h) \wedge ex \wedge \overline{phase} \end{aligned}$$

Die Datenpfade der MMU sind in Abbildung 5.8 dargestellt. Als letzte Erweiterung muss dafür gesorgt werden, dass im Fall eines Interrupts in den system mode gewechselt wird.

$$JISR(c) \quad \Rightarrow \quad c'.mode = 0$$

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 Spezifikation ist eine Art „Formalisierung des Benutzerhandbuchs des Kerns“ und kommt ohne Assembler aus, nur bei der Implementierung muss inline assembler verwendet werden. Wir sprechen vom abstrakten Kern k, der ein C_0 Programm ist. Dazu kommen:

- der i-te Benutzerprozess $vm(i)$
- die i-te I/O-Device $D(i)$

Formal enthält die CVM-Konfiguration cvm:

- $cvm.c$ - C_0-Konfiguration des abstrakten Kerns
- $cvm.vm(i)$ - virtuelle DLX-Konfiguration des Benutzers i
- $cvm.ptl(i)$ - Anzahl der pages für Benutzer i
- $cvm.vm(i).vm : \{a \mid a \in \{0,1\}^{32}, \langle a \rangle \leq 4K \cdot cvm.ptl(i)\} \longrightarrow \{0,1\}^8$ - virtueller Speicher des Benutzers i
- $cvm.cp \in \{0,\ldots,p\}$ - **c**urrent **p**rocess, aktuell laufender Prozess (p ist die Anzahl der Benutzerprozesse)

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

$$\delta(cvm) = cvm'$$

Hierbei gibt es zwei triviale Fälle:

- $cvm.cp = u \land u > 0 \land JISR(c.vm.vm(u)) = 0$, das heißt Benutzer u läuft und wird nicht von einem Interrupt unterbrochen, dann gilt:

  $$cvm'.vm(u) = \delta_D(cvm.vm(u))$$

  Die virtuelle Maschine des Benutzers u ändert sich also gemäß der DLX-Übergangsfunktion. Alle anderen Komponenten bleiben unverändert.

  $$\begin{aligned} cvm'.cp &= cvm.cp \\ cvm'.c &= cvm.c \\ \forall i \neq u, i \in \{1,\ldots,p\} : cvm'.vm(i) &= cvm.vm(i) \\ \forall i \neq u, i \in \{1,\ldots,p\} : cvm'.ptl(i) &= cvm.ptl(i) \end{aligned}$$

- $cvm.cp = 0$, 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 C_0-Semantik ausgeführt. Die C_0-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 ($trap$)

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. Die essentiellen Vertreter sollen hier aufgeführt werden. Zuvor definieren wir noch diese Kurzschreibweise für den Wert von C_0-Variablen und Ausdrücke X des Kerns k.

$$\widetilde{X} \quad \widehat{=} \quad va(cvm.c, X)$$

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:

- $an: \quad start\ CP$ - 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 nennt sie **scheduler**.

- $an: \quad copy(s, r, s_1, s_2, l)$ - Kopiert einen Bereich der Länge l aus dem Speicher von Benutzer s in den Speicher von Benutzer r. Dabei ist.
  - s - der Sender (**s**ender)
  - r - der Empfänger (**r**eceiver)
  - 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,

  $$\begin{aligned} \widetilde{s_1} + \widetilde{l} &< 4K \cdot cvm.ptl(\widetilde{s}) \\ \widetilde{s_2} + \widetilde{l} &< 4K \cdot cvm.ptl(\widetilde{r}) \end{aligned}$$

  dann bewirkt $copy$:

  $$cvm'.vm(\widetilde{r}).vm_{\widetilde{l}}(\widetilde{s_2}) = cvm.vm(\widetilde{s}).vm_{\widetilde{l}}(\widetilde{s_1})$$

  Der Rest der Konfiguration bleibt unverändert. Man beachte wie hier der Formalismus des DLX-Speichers und der C_0-Ausdrücke vermischt wird.

- $an: \quad alloc(u, x)$ - Weist Benutzer $\widetilde{u}$ $\widetilde{x}$ neue pages zu, die mit 0 initialisiert werden müssen. Ansonsten wäre es Prozessen eventl. möglich auf die Daten anderer Prozesse unberechtigt zuzugreifen.

  $$\begin{aligned} cvm'.ptl(\widetilde{u}) &= cvm.ptl(\widetilde{u}) + \widetilde{x} \\ \forall a \in \{0,1\}^{32} : \langle a \rangle \geq 4 \cdot cvm.ptl(\widetilde{u}) &\Rightarrow cvm'.vm(\widetilde{u}).vm(a) = 0^8 \end{aligned}$$

  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 genaue Effekt soll selbständig definiert werden.

5.2.2 Semantik von trap-Instruktionen

Nun sol der Fall betrachtet werden, dass ein Benutzerprozess läuft ($cvm.cmp = u > 0$) 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

$$kcd : [0 : nk - 1] \longrightarrow \{\text{Namen}\}$$

mit der Anzahl nk der verschiedenen kernel calls, die das Betriebssystem zur Verfügung stellt. Dann ist $f = kcd(j)$ der Name der funktion die mit der $trap$-Instruktion aufgerufen werden sollte. Als die $ft(f).np$ 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 C_0-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 $return_from_trap(x)$ 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 $sy(top(cvm'.c)) = ft(f).st$ - Ein neuer function frame für f wird erzeugt.
- $\forall par \in [0 : ft(f).np-1] : top(cvm'.c).ct(par) = cvm'.vm(u).gpr(par+1)$ - Die Registerinhalte von u werden als Parameter übergeben.
- $cvm'.c.pr = ft(f).body; cvm.c.pr$ - Die Funktion wird ausgeführt und mit $return_from_trap(x)$ abgeschlossen. Das System wechselt danach wieder zu Benutzer u in den user mode.

Anmerkung des Verfassers: Alternativ zur oben vorgestellten Herangehensweise, bei der man die CVM-Primitive mittels eines Pseudo-Funktionsaufrufs startet, gibt es auch die Möglichkeit, die Kernel-Funktionen direkt als C0-Funktionen aufzurufen. Man deklariert dazu die globalen Variablen des Kerns $result$ und $dummy$, die die Rückgabewerte der speziellen Funktionen auffangen sollen. Wir benötigen ebenfalls eine spezielle Funktion $rft(u)$, die nach dem Aufruf das Resultat und die Kontrolle an den aufrufenden Prozess u zurückgibt. Ein Aufruf der Kernel-Funktion f durch Prozess u hat dann die folgenden Auswirkungen.

- $cvm'.cp = 0$ - Die Maschine wechselt in den system mode und der Kern läuft.
- $cvm'.c.pr = result{=}f(\langle par_1 \rangle, \ldots, \langle par_{ft(f).np} \rangle); dummy{=}rft(u); cvm.c.pr$ mit den Registerinhalten $par_i = cvm'.vm(u).gpr(i)$ von Prozess u, die als Parameter an f übergeben werden.

Dieser Ansatz erscheint eleganter, da die C_0-Semantik ausgenutzt und nicht in gewisser Weise umgangen wird. Auf der anderen Seite benötigt es mehr Instruktionen, zwei komplette Funktionsaufrufe durchzuführen. Daher ist diese Variante langsamer als die obige.

ACHTUNG: *Diese Anmerkung ist als eine zusätzliche Information und als Denkanstoß gedacht. Die Gültigkeit der Definition aus der Vorlesung bleibt dadurch unangetastet.*

5.3 CVM-Implementierung

Nun soll CVM implementiert werden. Dazu betrachten wir den *konkreten Kern* K, bestehend aus:

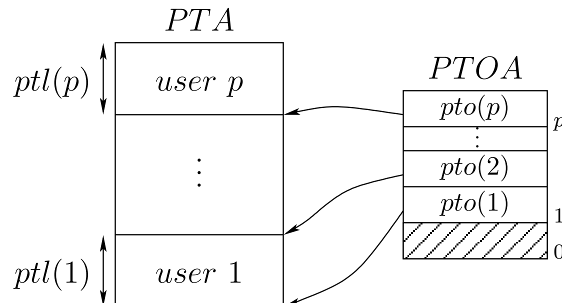
- dem abstrakten Kern k

- zusätzlichen Datenstrukturen und Funktionen

Formal gesehen benötigt man noch eine „Theorie des Linkens“ von C_0-Programmen, die unter anderm die Disjunktheit verwendeter Namen behandelt, wir wollen uns hier aber auf die Datenstrukuren des konkreten Kerns konzentrieren.

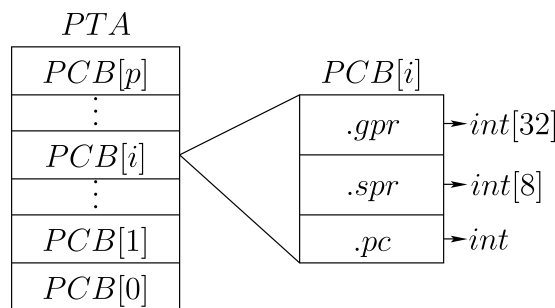
5.3.1 Datenstrukturen des konkreten Kerns

Zur Verwaltung der page tables existiert im Speicher ein **p**age **t**able **a**rray PTA, das alle page tables der Benutzer $1, \ldots, p$ fasst. Dazu gibt es ein **p**age **t**able **o**rigin **a**rray $PTOA$, in dem die Startadressen der page tables für jeden Benutzer aufgeführt werden.



Das $PTOA$ kann entweder in einem extra array gespeichert werden, oder man berechnet es stets aus den process control blocks der Benutzer über $PCB.spr[5]$. Die page table von Benutzer i befindet sich im Bereich $PTA[pto[i+1] - 1 : pto[i]]$. 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.

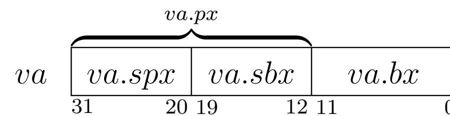
Weiterhin gibt es ein array von **p**rocess **c**ontrol **b**locks PCB im Speicher.



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. Man kann PCB als C_0-Datenstruktur deklarieren:

```
typedef int[32] u;
typedef int[8] v;
typedef struct{u gpr, v spr, int pc} pcb;
typedef pcb[p+1] pcba;
pcba PCB;
```

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 einfahcn 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 Speicher 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 (C_0A)
- 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

- kernel allocated base adresses $k\text{-}aba^0, k\text{-}aba^1, \ldots$ aus der Kernel-Implementierung

existieren, so dass eine Simulationsrelation zwischen CVM, Kernel und DLX gilt.
Für globale Variablen X gilt $aba^0(X) = aba^1(X) = \ldots = aba(X)$, da sich die zugeordneten Basisadressen von globalen Variablen im Programmverlauf nicht ändern. Die Datenstrukturen des konkreten Kerns liegen daher im Kern. 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, gilt:

$$va(d,e) = d.pm_4(aba(e))$$

Es werden hier erneut die Notationen aus C_0-Semantik, DLX-Konfiguration und dem Compiler zusammengeführt. Mit Hilfe der Datenstrukturen von k kann man nun die Adressübersetzung für mehrere Benutzer definieren.

$$\begin{aligned}
pto(c,u) &= va(d, PCB[u].pto)\\
ptea(d,u,va) &= pto(d,u) + 4 \cdot va.px\\
pte(d,u,va) &= d.pm_4(ptea(d,u,va))\\
ppx(d,u,va) &= pte(d,u,va)[31:12]\\
v(d,u,va) &= pte(d,u,va)[11]\\
pma(d,u,va) &= ppx(d,u,va) + 4 \circ va.bx
\end{aligned}$$

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 $sim(cvm^i, k^{s(i)}, d^{t(i)}, aba^i, k\text{-}aba^i)$ 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. $sim(cvm^i, k^{s(i)}, d^{t(i)}, aba^i, k\text{-}aba^i)$ 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 $\mathrm{k-consis}(k^{s(i)}, k\text{-}aba^i, cvm^i.c)$ 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:*

$$cvm.vm(u).R = \begin{cases} d.R & : \quad cvm.cp = u \quad (u \text{ läuft})\\ va(d, PCB[u].R) & : \quad \text{sonst} \end{cases}$$

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 $\mathrm{k-consis}(k, k\text{-}aba, c)$ benötigt. Sie verwendet die kernel allocated base address function

$$k\text{-}aba : \{\text{(Sub−)Variablen von } c\} \longrightarrow \{\text{(Sub−)Variablen von } k\},$$

die Verbindung zwischen den Variablen in beiden Kernen herstellt. $k\text{-}aba(x) = x'$ bedeutet dann in etwa, dass x' die Variable von k ist, die x simuliert.

Definition 5.2 *(k-consis) Gilt die k-consis-Relation* $\mathrm{k-consis}(k, k\text{-}aba, c)$*, so wird* c *von* k *simuliert und es gilt* $\mathrm{k-econsis}$ *und* $\mathrm{k-pconsis}$ *für einfache Variablen* x.

- $\mathrm{k-econsis}$ *(elementare Werte):*

$$\begin{aligned} va(c,x) &= va(k,x') \\ &= va(k, k\text{-}aba(x)) \end{aligned}$$

- $\mathrm{k-pconsis}$ *(pointer):*

$$\begin{aligned} va(c,x) &= y \\ \Rightarrow va(k, k\text{-}aba(x)) &= k\text{-}aba(y) \end{aligned}$$

Eine Implementierung, für die diese Konsistenzrelationen gelten, führt dann das CVM-Betriebssystem korrekt aus. Dies lässt sich mit Hilfe der C_0-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.

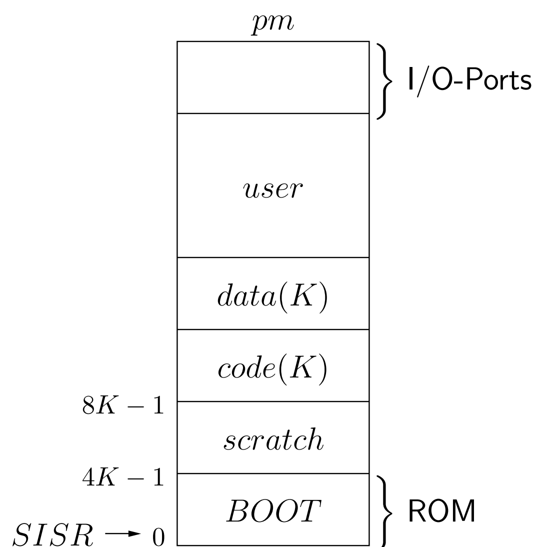


Abbildung 5.9: memory map des physikalischen Speichers

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 C_0-Subvariable $PCB[u].ECA$ gesichert wurde, kann die Berechnung von il durch ein einfaches C_0-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 C_0 kann man niemals auf bestimmte Register, Speicheradressen, I/O-Geräte etc. zugreifen. Es wird daher eine Möglichkeit benötigt, Assembler-Code in gewöhnlichen C_0-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 C_0A-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 C_0-Variable der Wert eines Registers zugewiesen werden. Für $x = PCB[u]$ und $sw(d^i)$ gilt dann:

$$\begin{aligned} ea(d^i) &= aba(k^i, x) \\ va(k^{i+1}, x) &= d^i.gpr(RD(d^i)) \end{aligned}$$

Man beachte, wie erneut DLX- und C_0-Semantik elegant und problemlos miteinander vermischt werden.

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.