Systemarchitektur-Skript

Dilyana Dimova
Mario Mancino

26. Juli 2004

Inhaltsverzeichnis

1 Einführung

1.1 Literaturhinweise

- *S.M. Müller, W.J. Paul*: Computer Architecture (Springer 2000)
- *J. Keller, W.J. Paul*: Hardware Design (vergriffen)
- *G.Even*: Skript

1.2 Definition: Rechnen

Rechnen:	(Hoffentlich sinnvolle) Manipulation von Zeichenreihen
Beispiel:	Hardware
Definition:	Schaltfunktion $f : \{0,1\}^n \longrightarrow \{0,1\}$
Ziel:	Bauen von Boxen, die Schaltfunktionen berechnen
Kalkül:	Bool'sche Algebra
Variablen:	$V = \{x_0, x_1, x_2, \ldots\}$

1.3 Bool'sche Algebra

1.3.1 $\sim: \{0,1\} \longrightarrow \{0,1\}$

x	$\sim x$
0	1
1	0

1.3.2 $\wedge, \vee, \oplus : \{0,1\}^n \longrightarrow \{0,1\}$

x_0	x_1	$x_0 \wedge x_1$	$x_0 \vee x_1$	$x_0 \oplus x_1$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

1.4 Definition: Bool'sche Ausdrücke (vollständig geklammert)

1. $B_0 = V \cup \{0, 1\}$
2. $e_1, e_2 \in B_i \;\Rightarrow\; e_1, e_2 \in B_{i+1}$
   $(\sim e_1) \in B_{i+1}$
   $e_1 \wedge e_2 \in B_{i+1}$
   $e_1 \vee e_2 \in B_{i+1}$
   $e_1 \oplus e_2 \in B_{i+1}$
   (sonst nichts in B_{i+1})
3. $BA = \bigcup_{i=0}^{\infty} B_i$

Beispiel:

$$(x_1 \wedge ((\sim x_2) \vee x_3)) \in BA$$

Beweis:

$$x_1, x_2, x_3 \in V \subseteq B_0, B_1, B_3, \ldots$$
$$(\sim x_2) \in B_1, B_2, \ldots$$
$$((\sim x_2) \vee x_3) \in B_2, \ldots$$
$$(x_1 \wedge ((\sim x_2) \vee x_3)) \in B_3, \ldots \subseteq BA$$

1.5 Exkurs: Mengen

M: Menge, $n \in \mathbb{N}$
$M_n = \{(a_1, \ldots, a_n) \,|a_i \in M\}$
Beispiel:

$$M = \{0, 1\}$$
$$M_1 = \{(0), (1)\}$$
$$M_2 = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$$
$$M_3 = \{(0, 0, 0), (0, 0, 1), \ldots\}$$

1.6 Exkurs: Induktionsbeweis

A(n): Aussage

1. Zeige A(1)
2. Zeige A(n) → A(n+1)
3. Schliesse: A(n) für alle $n \in \mathbb{N}$

1.7 Rechnen mit bool'schen Algebren

Beispiel:

$$x_1 \wedge x_2 = x_2 \wedge x_1$$
$$((x_1 \wedge x_2) \wedge x_3) = (x_1 \wedge (x_2 \wedge x_3))$$

1.7.1 Definition: Einsetzung

$\varphi : V \to \{0, 1\}$
Intuition: setze für x_i Werte $\varphi(x_i)$ ein.

Beispiel:

	φ
x_1	0
x_2	1
x_3	0

1.7.2 Definition: $\varphi(e)$ für $e \in BA$

$$\varphi(0) = 0$$
$$\varphi(1) = 1$$
$$\varphi(e_1 \wedge e_2) = \varphi(e_1) \wedge \varphi(e_2)$$
$$\varphi(e_1 \vee e_2) = \varphi(e_1) \vee \varphi(e_2)$$
$$\varphi(e_1 \oplus e_2) = \varphi(e_1) \oplus \varphi(e_2)$$
$$\varphi(\sim e) = \sim \varphi(e)$$

Beispiel:

$$e = (x_1 \wedge ((\sim x_2) \vee x_3))$$
$$\varphi(x_1) = 0$$
$$\varphi(x_2) = 1$$
$$\varphi(x_3) = 0$$

Einsetzung:

$$\begin{aligned} \varphi(\sim x_2) &= \sim \varphi(x_2) \\ &= \sim 1 \\ &= 0 \end{aligned}$$

$$\begin{aligned} \varphi((\sim x_2) \vee x_3) &= \varphi(\sim x_2) \vee \varphi(x_3) \\ &= 0 \vee 0 \\ &= 0 \end{aligned}$$

$$\begin{aligned} \varphi(e) &= \varphi(x_1) \wedge \varphi((\sim x_2) \vee x_3) \\ &= 0 \wedge 0 \\ &= 0 \end{aligned}$$

1.7.3 Definition: Identität (Allgemeingültige Rechenregel)

$e_1, e_2 \in BA$:

$$e_1 \equiv e_2 \Leftrightarrow \varphi(e_1) = \varphi(e_2) \; für \; alle \; \varphi$$

1.7.4 Satz: Kommutativität

$$x_1 \wedge x_2 = x_2 \wedge x_1$$
$$x_1 \vee x_2 = x_2 \vee x_1$$
$$x_1 \oplus x_2 = x_2 \oplus x_1$$

Beweis: trivial durch Ausprobieren aller Fälle.
Rechenregeln:

$$(x_1 \wedge x_2) \wedge x_3 = x_1 \wedge (x_2 \wedge x_3)$$
$$x_1 \wedge (x_2 \vee x_3) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3)$$

1.8 Exkurs: Funktionen($\neq$Funktionssymbole)

Seien X,Y Mengen, dann gilt:

$$X \times Y = \{(x, y) | x \in X, y \in Y\}$$

Beispiel:

$$\begin{aligned} \{0,1\}^2 &= \{0,1\} \times \{0,1\} \\ \{0,1\}^3 &\neq \{0,1\} \times \{0,1\}^2 \\ \{0,1\}^3 &= \{(0,0,0),(0,0,1),\ldots\} \\ \{0,1\} \times \{0,1\}^2 &= \{(0,(0,0)),(0,(0,1)),\ldots\} \end{aligned}$$

1.8.1 Definition: Relation

$R \subset X \times Y$: Relation
z.B.: $X = Y = R,\ R = \{(x, y) | x \geq y\}$

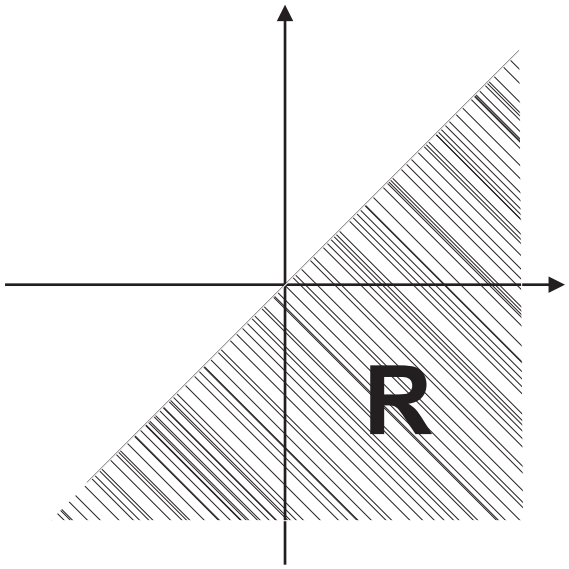


Abbildung 1: Relation

1.8.2 Definition: Funktion

Die Funktion ist eine spezielle (Rechtseindeutige) Relation:

$$(x, y_1) \in \mathbb{R}$$
$$(x, y_2) \in \mathbb{R}$$

$$\Rightarrow \quad y_1 = y_2$$

Der Funktion liegt eine Funktionstabelle zugrunde, f_i ist nur ein Name dafür.

1.9 Erweiterung zu 1.7.2

$$\varphi(f(e_1, \ldots, e_s)) =_{Def} f(\varphi(e_1), \ldots, \varphi(e_s)))$$

1.10 Konventionen

Um Schreibarbeit zu sparen, vereinbart man:
$\sim$ bindet stärker als $\wedge$
$\wedge$ bindet stärker als $\vee$

1.10.1 Abkürzungen

$\bar{e}$ ist Abkürzung für $(\sim e)$
$/e$ ist Abkürzung für $(\sim e)$
$x_i x_j$ ist Abkürzung für $x_i \wedge x_j$
$=$ ist Abkürzung für $\equiv$
$x[n:1]$ ist Abkürzung für $(x_n, \ldots, x_1)$

1.11 Definition: elementarer bool'scher Ausdruck

Ein elementarer bool'scher Ausdruck, in dem $\wedge, \vee, \sim$ verwendet werden.

$$ElBA = \{e \in BA | e\ elementar\} = alte\ Definition$$

1.12 Satz: Darstellungssatz

$$\forall\, f:\ \{0,1\}^n \to \{0,1\}\ \exists\, e\ elem.\ B.A.\ mit:$$
$$e \equiv f(x_1,\ldots,x_n)$$

1.12.1 Beweis

Beweis per Induktion über n:
n=0:

$$f : \underbrace{\{0,1\}_0}_{\phi} \to \{0,1\}$$
$$\subseteq \phi \times \{0,1\}$$

Konvention: 0,1:$\{0,1\}^0 \to \{0,1\}$
n=1:

Funktionstabelle:

x_1	f_1	f_2	f_3	f_4
0	0	1	1	0
1	0	1	0	1
e	0	1	x_1	$\overline{x_1}$

n →n+1:

$$f(x_1,\ldots,x_{n+1}) = x_{n+1} \wedge f(x_1,\ldots,x_n,1) \vee x_{n+1} \wedge f(x_1,\ldots,x_n,0)$$

1.12.2 Beispiel

x_1	x_2	x_3	$f(x_3,x_2,x_1)$	$f(0,x_2,x_1)$	$f(0,0,x_1)$
0	0	0	0	0	0
0	0	1	1	1	1
					$\overbrace{x_1}$
					$f(0,1,x_1)$
0	1	0	1	1	1
0	1	1	0	0	0
				$\overbrace{e_0 = \overline{x_2}x_1 \vee x_2\overline{x_1}}$	$\overbrace{\overline{x_1}}$
				$f(1,x_2,x_1)$	$f(1,0,x_1)$
1	0	0	1	1	1
1	0	1	1	1	1
					$\overbrace{1}$
					$f(1,1,x_1)$
1	1	0	1	1	1
1	1	1	0	0	0
				$\overbrace{e_0 = \overline{x_2} \wedge 1 \vee x_2\overline{x_1}}$	$\overbrace{\overline{x_1}}$

1.13 Lösen einer Gleichung

Einsetzung mit $\varphi(e) = \varphi(e')$ heißt *Lösen* der Gleichung.

1.13.1 Beispiel 1

Sei e ein elementarer bool'scher Ausdruck.

$$\overline{e} = 1$$

Lösung: Einsetzen von φ mit:

$$\varphi(\overline{e}) = \varphi(1) = 1$$
$$\varphi(\overline{e}) = \sim \varphi(e) = 1 \Leftrightarrow \varphi(e) = 0$$

d.h.		φ Lösung von $\overline{e} = 1$
	$\Leftrightarrow$	φ Lösung von $e = 0$

Dies wird abgekürzt als $\overline{e} = 1 \Leftrightarrow e = 0$

1.13.2 Beispiel 2

Seien e, e' elementare bool'sche Ausdrücke.
e $\wedge$ e' = 1
Sei φ Lösung:

$$\varphi(e \wedge e') = \varphi(1) = 1$$
$$\varphi(e) \wedge \varphi(e') \Leftrightarrow \varphi(e) = 1 \wedge \varphi(e') = 1$$

$\Leftrightarrow$		φ Lösung von e=1
	und	φ Lösung von e'=1

1.13.3 Beweis: $e_1 \wedge \ldots \wedge e_n = 1 \Leftrightarrow e_1 = 1 \wedge \ldots \wedge e_n = 1$

Induktion über n:
n=2:

$$e_1 = e$$
$$e_2 = e'$$

(eben in 1.13.1 und 1.13.2 gemacht) *n*→*n+1*:

$$\underbrace{e_1 \wedge \ldots \wedge e_{n-1}}_{e} \wedge \underbrace{e_n}_{e'} = 1 \Leftrightarrow \underbrace{e_1 \wedge \ldots \wedge e_{n-1} = 1}_{\Leftrightarrow e_1=1 \wedge \ldots \wedge e_{n-1}=1} \textit{ und } e_n = 1$$

1.13.4 Beweis: $e_1 \vee \ldots \vee e_n = 1 \Leftrightarrow e_1 = 1 \vee \ldots \vee e_n = 1$

Der Beweis läuft analog zu 1.13.3.

1.14 Zusammenhang zwischen Identitäten und Lösungen von Gleichungen

Seien e, e′ elementare bool'sche Ausdrücke.

$$\overbrace{\begin{array}{ll} \forall\varphi: & \varphi \text{ Lösung von } e=1 \\ \Leftrightarrow & \varphi \text{ Lösung von } e'=1 \end{array}}^{Lösung\ von\ e=1,\ e'=1}$$

$$\underbrace{\begin{array}{lll} \varphi(e)=1 & \Leftrightarrow & \varphi(e')=1 \\ \varphi(e)=0 & \Leftrightarrow & \varphi(e')=0 \end{array}}_{e\equiv e'}$$

Um e ≡ e′ zu zeigen, genügt es zu zeigen:
Gleichung e=1 und e′=1 haben gleiche Lösungen - Kurz:
e=1 ⇔ e′=1

$$\boxed{\begin{array}{c} \text{e} \equiv \text{e}' \qquad \Leftrightarrow \\ \text{e=1} \Leftrightarrow \text{e}'\text{=1} \end{array}}$$

1.15 Spezielle bool'sche Ausdrücke

1.15.1 Definition: Literal, Monom, Polynom

Seien $x_1, x_2, \ldots, x_n$ Variablen.

Ausdruck L heißt $\underline{Literal}$:
$\Leftrightarrow L = x_i$ für ein x_i

Ausdruck M heißt $\underline{Monom}$:
$\Leftrightarrow M = L_1 \wedge \ldots \wedge L_s$ für ein s und Literale $L_i \ldots L_s$

Ausdruck P heißt $\underline{Polynom}$:
$\Leftrightarrow P = M_1 \vee \ldots \vee L_t$ für ein t und Monome $M_i \ldots M_s$

1.15.2 Bemerkung

Gleichungen

$$L = 1$$
$$M = 1$$
$$P = 1$$

können wir lösen.

Notation:

$$\epsilon \in \{0, 1\}$$
$$x_i^\epsilon = \begin{cases} x_i\text{: } \epsilon = 1 \\ \overline{x_i}\text{: } \epsilon = 0 \end{cases}$$
$$\boxed{x_i^\epsilon = 1 \Leftrightarrow x_i = \epsilon}$$

1.15.3 Beispiel

Sei $a = a[n:1] \in \{0,1\}_n$, $a \mapsto M(a) =_{Def.} \underbrace{x_n^{a_n} \wedge \ldots \wedge x_1^{a_1}}_{(Monom)}$

n=3, a=101:

$$\begin{aligned} M(a) &= x_3^1 \wedge x_2^0 \wedge x_1^1 \\ &= x_3\overline{x_2}x_1 \end{aligned}$$

$$\begin{aligned} M(a) = 1 &\Leftrightarrow x_i^{a_i} = 1 \; für \; alle \; i \\ &\Leftrightarrow x_i = a_i \; für \; alle \; i \\ &\Leftrightarrow x[n:1] = a[n:1] \end{aligned}$$

$$\boxed{M(a) = 1 \Leftrightarrow x = a}$$

1.16 Vollständig disjunktive Normalform

Sei $f : \{0,1\}^n \to \{0,1\}$

$$\underbrace{T(f) = \{a \in \{0,1\}^n | f(a) = 1\}}_{(Träger \; von \; f)}$$

$$\underbrace{T(f) \mapsto P(f) = \bigvee_{a \in T(f)} M(a) \; (Polynom)}_{(Vollständig \; disjunktive \; Normalform \; von \; f)}$$

$$\begin{aligned} P(f) = 1 &\Leftrightarrow es \; gibt \; a \in T(f) : \; M(a) = 1 \\ &\Leftrightarrow es \; gibt \; a \in T(f) : \; x = a \\ &\Leftrightarrow x \in T(f) \end{aligned}$$

$$\Rightarrow \boxed{P(f) \equiv f}$$

(Neuer Beweis des Darstellungssatzes)

1.16.1 Beispiel

x_3	x_2	x_1	$f(x)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

T(f)={	000	,	011	,	100	,	101	,	111	}		
	↓		↓		↓		↓		↓			
	M(000)		M(011)		M(100)		M(101)		M(111)			
	=		=		=		=		=			
	$\overline{x_3x_2x_1}$		$\overline{x_3}x_2x_1$		$x_3\overline{x_2x_1}$		$x_3\overline{x_2}x_1$		$x_3x_2x_1$		$\equiv$	$f(x_3, x_2, x_1)$

1.17 Kosten der Darstellungssätze

C(e) = Anzahl Vorkommen von $\wedge, \vee, \sim$ in e (Kosten)
G(e) = größte auftretende Kosten für ein $f : \{0,1\}^n \to \{0,1\}$

1.17.1 Hilfssätze

#M = Anzahl der Elemente in M (Mächtigkeit von M)

$$\#(M \times N) = \#M \cdot \#N$$
$$\#(M^N) = (\#M)^N \;\Rightarrow\; \#\{0,1\}^n = (\#\{o,1\}^n = 2^n$$

1.17.2 Kosten

1. $f(x) \equiv \bigvee_{a \in T} m(a)$
   vollständige disjunktive Normalform, Kosten:

   $$G(f) \leq 2n \cdot 2^n$$

2. $f(x[n:1]) = x_n \cdot f(1, x[n-1:1]) \vee \overline{x_n} \cdot f(0, x[n-1:1])$
   Differenzengleichung:

G(1)	=	1
G(n)	=	4 + 2 $\cdots$ G(n-1)

   geraten (k=n-1):

$$\begin{aligned} G(n) &= 2^k \cdots G(n-k) + 2^{k+1} + 2^k + \ldots + 4 \\ &= 2^{n-1} \cdots G(1) + \underbrace{2^n + 2^{n-1} + \ldots + 4 + (+2+1}_{2^{n+1}-1} - 3) \\ &= 2^{n+1} + 2^{n-1} - 4 \\ &= 5 \ldots 2^{n-1} - 4 \end{aligned}$$

1.18 Teure Funktionen

Gibt es $f : \{0,1\}^n \to \{0,1\}$ für das gilt: Jedes e, das f berechnet ist teuer?
Ja:

- Jedes e berechnet eine Funktion
- Die Menge der billigen Ausdrücke ist klein

$\Rightarrow \{f | \exists$: e billig und e berechnet $f\}$ ist klein, $\{f : [0,1]^n \to \{0,1\}\}$ ist groß

1.18.1 Beweis

Sei e ein elementarer bool'scher Ausdruck.
$A = \{x_1, \ldots, x_n, (,), \wedge, \vee, \sim, \llcorner, 0, 1\}$
$\#A = n + 8$

$$\begin{aligned} 2^{2^n} &= \#\{f | f : \{0,1\}^n \to \{0,1\}\} \\ &\leq \#\{e | G(e) \leq k\} \\ &= \#(A^{5k} \\ &= (n+8)^{5k} \end{aligned}$$

$$k : \ \forall f : \{0,1\}^n \to \{0,1\} \ \exists e : \ e \equiv f(x)$$
$$G(e) \leq k$$

$$\boxed{k \geq \frac{2^n}{5 \cdot log(n+8)}}$$

1.18.2 Offene Fragen

- Wie sehen f aus, für die jedes e teuer ist?
  $\oplus$ erlauben?
  $A = \{x_1, \ldots, x_n, (,), \wedge, \vee, \sim, \llcorner, 0, 1, \oplus\}$
  $\#A = n + 9$
  $k \geq \frac{2^n}{5 \cdot log(n+9)}$

- Bool'sches Polynom p:
  Hat die Gleichung p=0 eine Lösung?
  Gibt es ein *schnelles* Rechenverfahren?
  P = NP?

2 Schaltpläne und Schaltkreise

2.1 Gatter

Eingänge: $x_1 \quad x_2$
Gatter: Box zur Berechnung von Schaltfunktionen
Ausgänge: $y_1 \ldots y_n$ (Kabel, Leitungen)

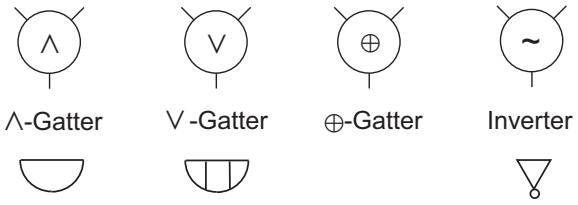


Abbildung 2: Gatter-Symbole

2.1.1 Beispiel-Gatter

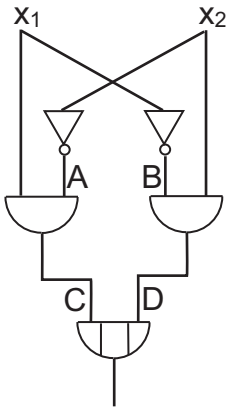


Abbildung 3: Schaltkreis - XOR Gatter

x_1	x_2	A	B	C	D	E
0	0	1	1	0	0	0
0	1	0	1	0	1	1
1	0	1	0	1	0	1
1	1	0	0	0	0	0

$$G \equiv x_1 \oplus x_2$$

2.1.2 Einsetzung an den Eingängen angelegter Werte

$$\varphi : \{x_1, \ldots, x_n\} \to \{0, 1\}$$

Z′, Z″: Eingänge des Gatters in Schaltung S
Z: Ausgang eines Gatters in Schaltung S
f: berechnete Funktion des Gatters in Schaltung S

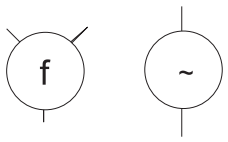


Abbildung 4: Eingänge des Gatters

$$\begin{aligned} \varphi(Z) &= f(\varphi(Z'), \varphi(Z'')) \\ &= \begin{cases} \varphi(Z') \wedge \varphi(Z'') : & f = \wedge \\ \varphi(Z') \vee \varphi(Z'') : & f = \vee \\ \varphi(Z') \oplus \varphi(Z'') : & f = \oplus \end{cases} \\ \varphi(Z) &= \sim \varphi(Z') \end{aligned}$$

2.1.3 Schwierigkeiten von Zyklen

Abbildung 5: Schaltkreis - Zyklen-Problem

$$\varphi(A) = 1 \Rightarrow \varphi(B) = 0 \Rightarrow \varphi(A) = 0\textbf{!}$$
$$\varphi(A) = 0 \Rightarrow \varphi(B) = 1 \Rightarrow \varphi(A) = 1\textbf{!}$$

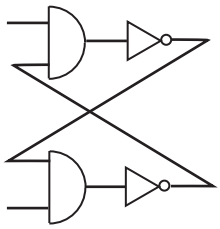


Abbildung 6: Schaltkreis - AND Flip-Flop

Bei beiden Beispielen ergeben sich Probleme, da in den jeweiligen Gattern mit den Untergattern $G_1, \ldots, G_S$ von einem Untergatter G_n zu einem vorherigen Gatter G_{n-k} $(0 < k < n)$ gesprungen wird.

2.1.4 Definition: Pfad

Der Pfad wird definiert durch:

$$G_1 \rightarrow G_2 \rightarrow \ldots \rightarrow G_S$$

Dabei ist S die Länge das Pfades.

2.2 Definition: Schaltkreis

Ein Schaltkreis ist eine Schaltung in der keine Zyklen vorkommen dürfen.

2.2.1 Hoffnung/Satz

Wir hoffen ein Schaltkreis ist für alle Z mit $Z = Eingang$ oder $Z = Ausgang$ eines Gatters $\varphi(Z)$ wohldefiniert.

2.2.2 Beweis

Induktion über die Tiefe zwischen Z und den Eingängen.
Die Tiefe von einem Gatter ist die Länge eines längsten Pfades von den Eingängen zu G:

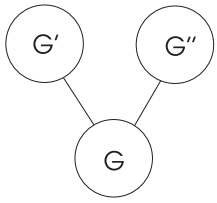


Abbildung 7: Tiefe (Formal)

$$Tiefe(G) = max\{Tiefe(G'), Tiefe(G'')\} + 1$$

2.2.3 Beispiel

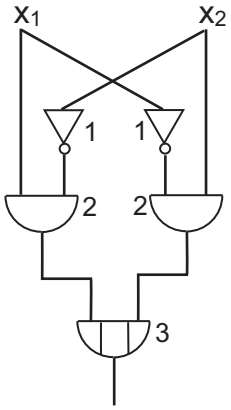


Abbildung 8: Schaltkreis - XOR-Gatters (Tiefe)

$$Tiefe(G) = max\{Tiefe(G'), Tiefe(G'')\} + 1$$

2.2.4 Hilfssatz

In einem Schaltkreis hat jedes Gatter eine Tiefe.

Beweis:
Annahme: Gatter G in Schaltkreis S hat keine Tiefe:
d.h.: $\forall\, t\ \exists$ Pfad von den Eingängen zu G mit Länger $\geq$ t
$x_1 \to G_1 \to \ldots \to G_t \to G$

Sei S endlich, $E = \#Gatter$ in $S,\ t > E$
$\Rightarrow$ Gatter wiederholt sich $\Rightarrow$ Zyklus **!**
Um zu beweisen, dass φ wohldefiniert ist, bedient man sich einer Induktion über die Tiefe(G).

2.3 Satz: Darstellungssatz (Schaltkreise)

$$\forall\, f:\ \{0,1\}^n \to \{0,1\}\ \exists\, Schaltkreis\ S:\ S\ berechnet\ f$$

2.3.1 Definitionen

$$S\ berechnet\ f,\ falls\ \exists\ Leitung\ u\ in\ S:$$
$$u\ \equiv\ f(x_1,\ldots,x_n)$$

$$K(S)\ =\ \#Gatter\ von\ S$$
$$(Kosten)$$

$$T(S)\ =\ \max\{Tiefe(G)\mid G\ ist\ Gatter\ in\ S\}$$

2.3.2 Beweis

Sei e bool'scher Ausdruck.

Behauptung:
$\exists$ Schaltkreis S
$\exists$ Leitung u in S: $u\ \equiv\ e$

Induktion über i:

$$e \in BA_i$$
$$e \in BA_0$$
$$e \in \{0,1,x_1,\ldots,x_i\}$$

0,1 erlauben wir als Eingänge von jedem Schaltkreis:

i → i+1:
$\circ \in \{\wedge, \vee, \oplus\}$
$e', e'' \in BA_i$

1. $e = e' \circ e''$

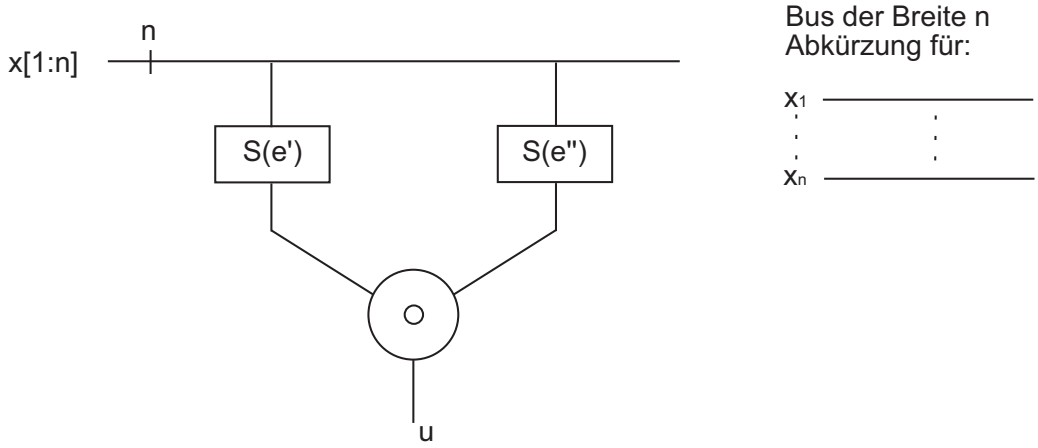


Abbildung 9: Schaltkreise - Darstellungssatz (Binäre Operationen)

2. $e = \sim e'$

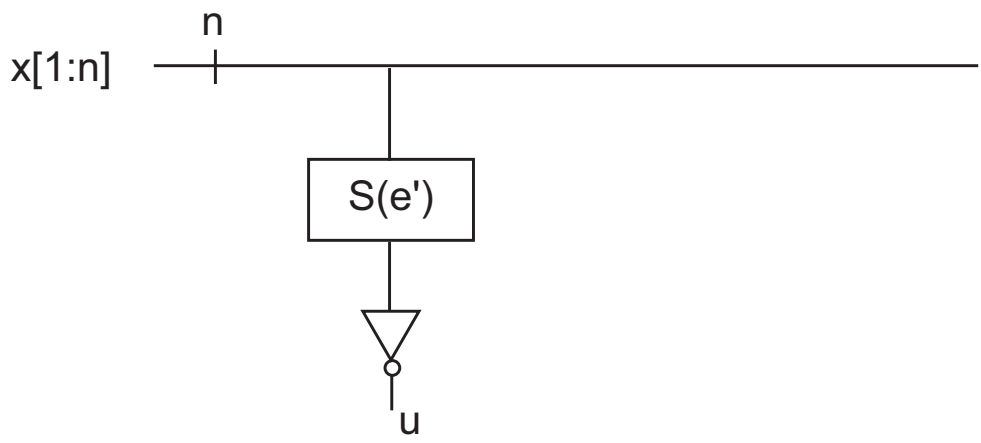


Abbildung 10: Schaltkreise - Darstellungssatz (Negation)

Bemerkung: K(S(e)) = G(e)

2.4 Polynome als Schaltkreise

Sei e Polynom:

$$e = M_1 \wedge \ldots \wedge M_t$$
$$M_i = L_{i1} \wedge \ldots \wedge M_{ij}$$
$$M_{ik} = \{x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$$

Zur Berechnung aller Literale reichen n Inverter.

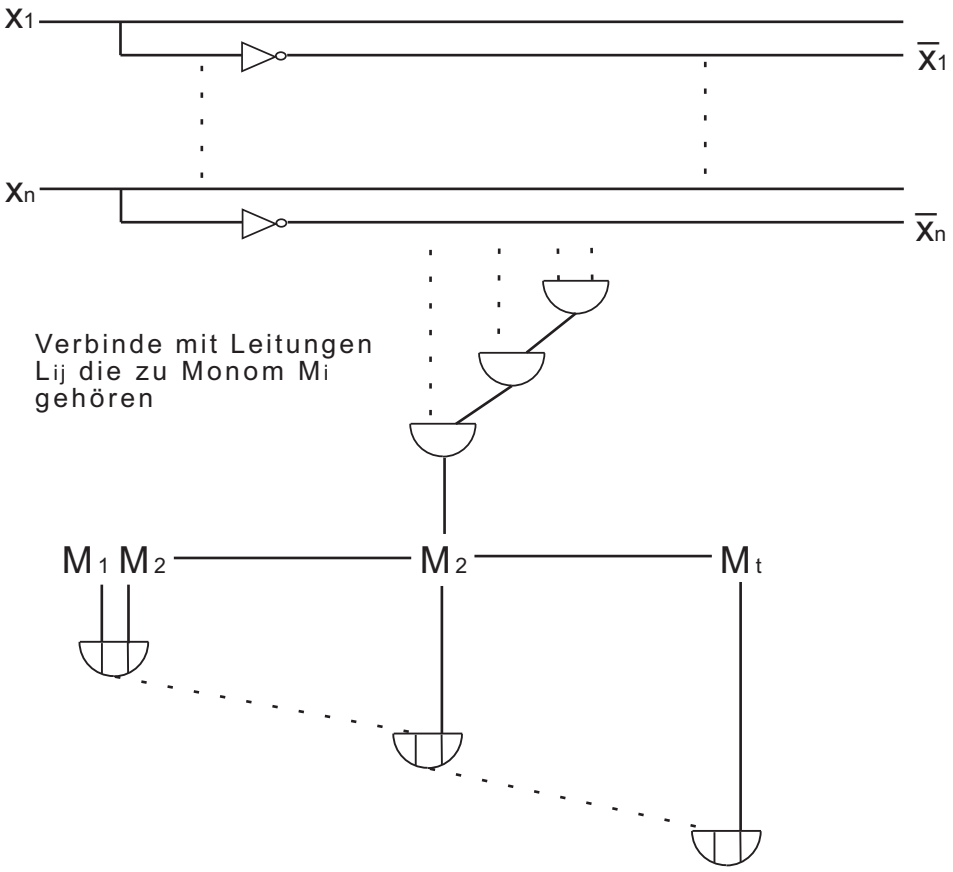


Abbildung 11: Schaltkreise - Polynom

$$\begin{aligned} Tiefe \quad leq \quad & 1 + \max\{j, i\} + t - 2 \\ leq \quad & 1 + +n + 2^n - 2 \end{aligned}$$

langsam!

2.5 Flache $\wedge$ / $\vee$ Bäume

2.5.1 Definition: Schaltkreise $\wedge$-n

n=1:

$$x_0$$
$$T(\wedge_1) = 0$$

n=2:

Abbildung 12: Schaltkreise - $\wedge - 2$ Baum

$$T(\wedge_2) = 1$$

$\frac{n}{2} \rightarrow n$*:*

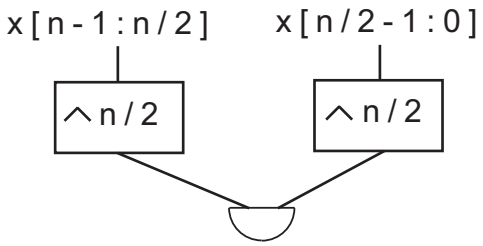


Abbildung 13: Schaltkreise - $\wedge - n$ Baum

$$T(\wedge_n) \;=\; T(\wedge_{n/2}) + 1 \;=\; \log_2 n$$

Es gilt $u \;\equiv\; v$ wegen Assoziativität von $\wedge$.

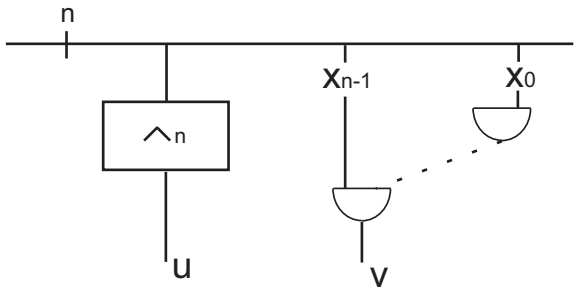


Abbildung 14: Vergleich - $\wedge-$Reihe / $\wedge - n$ Baum

2.5.2 n keine Zweierpotenz

Falls n keine Zweierpotenz ist:

$$\begin{aligned} n &= n' + k \\ \\ \frac{n}{2} &\leq n' \\ &= \max\{2^i | 2^i \leq n\} \\ k &= n - n' \\ &\leq \frac{n}{2} \end{aligned}$$

In diesem Fall sieht $\wedge_n$ so aus:

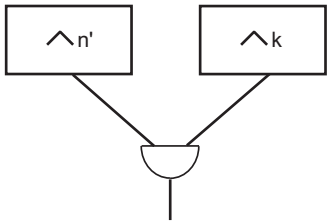
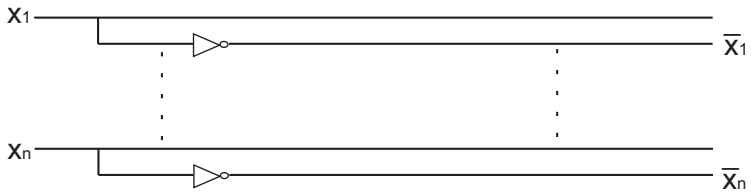


Abbildung 15: Schaltkreise - $\wedge - n$ Baum, falls n keine Zweierpotenz ist

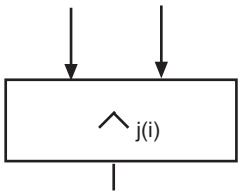
2.5.3 Neue Berechnung von Polynomen

Tiefe $\leq$



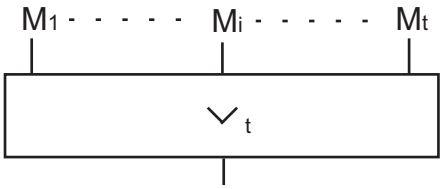
1

+



$\lceil \log_2 n \rceil$

+



$\lceil \log_2 n \rceil$

$$t \leq 2^n \leq n + \log_2 n + 2$$

2.5.4 PLA: Programmed Logic Array

Verwendung zur Realisierung sogenannter „random logic“:
Es werden keine Regelmäßigkeiten der Funktionstabelle ausgenutzt (Zufall = Abwesenheit von Regelmäßigkeiten)

3 Zahlendarstellung, Addieren (für ganze Zahlen)

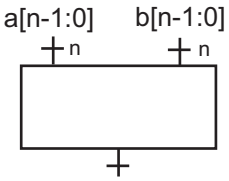


Abbildung 16: Addierer

Kodierung der Eingänge?
Addierer-Schaltkreis?
Kodierung der Summe?

3.1 Binärdarstellung

Binärdarstellungen:

$$a \in \{0,1\}^n \;=\; \{0,1\} \vee \{0,1\}^2 \vee \ldots$$

Dargestellte Zahl:

$$a \in \{0,1\}^n$$
$$a \;=\; a[n-1:0] \;=\; a_{n-1} \ldots a_0$$

Definition:

$$\boxed{< a > = \sum_{i=0}^{n-1} a_i \cdot 2^i}$$

$\Rightarrow$

$$\begin{aligned} < a > &= \sum_{i=0}^{n-1} a_i \cdot 2^i \\ &\leq \sum_{i=0}^{n-1} 2^i \\ &= 2^n - 1 \end{aligned}$$

3.1.1 Beispiel

$$\begin{aligned} < 101 > &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 4 + 0 + 1 \\ &= 5 \end{aligned}$$

3.1.2 Größe der Summe

Sei $S = C^0 + C^1 + \ldots + C^{n-1}$ $(C = 2)$:

$$\begin{aligned} S \cdot C &= C^1 + \ldots + C^{n-1} + C^n \\ S \cdot (C-1) &= C^n - C^0 \\ &= C^n - 1 \\ S &= \frac{C^n - 1}{C - 1} \\ &= 2^n - 1 (falls\ c = 2) \\ \Rightarrow < a > + < b > &\leq (2^n - 1) \cdot 2 \\ &< 2^{n+1} - 1 \end{aligned}$$

Hoffnung:

$$\exists\, S \in \{0,1\}^{n+1}$$
$$mit\ < S > = < a > + < b >$$

3.2 Satz: Eindeutigkeit der Binärdarstellung

$< >$: $\{0,1\} \rightarrow \{0, \ldots, 2^n - 1\}$ ist bijektiv (Beweis: Übungsblatt, Hinweis: $\vee$-Baum $\rightarrow$ $\vee$-Gatter-Platzierung)
D.h. jede Zahl aus $\{0, \ldots, 2^n - 1\}$ hat eine eindeutige Binärdarstellung der Länge n.

3.3 Definition: n-bit-Addierer

Schaltkreis:

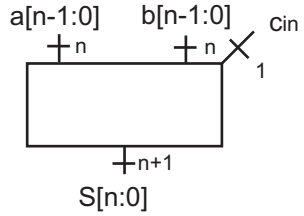


Abbildung 17: n-bit Addierer

mit:

$$< s > = < a > \ + \ < b > \ + c_{in}$$

3.4 1-bit Addierer / Volladdierer / Full Adder (FA)

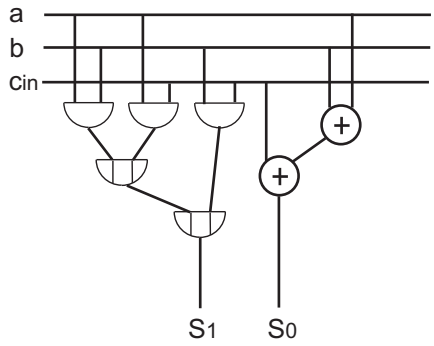


Abbildung 18: Schaltkreis - 1-bit Addierer

3.4.1 Funktionstabelle

a	b	c_{in}	S_1	S_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

3.5 Peano Axiome

$$Nachfolgerfunktion\ N : \mathbb{N}_0 \to \mathbb{N}_0$$

$$0 \in \mathbb{N}_0$$

0 ist eine (natürliche) Zahl.

$$\forall\, x \in \mathbb{N}_0\ \exists\ < \in\ \mathbb{N}_0 :\ y\ =\ N(x)$$

Jede natürliche Zahl hat einen Nachfolger.

$$\neg\ \exists\, y :\ 0 = N(j)$$

0 ist nicht Nachfolger einer Zahl.

$$x \neq y\ \Leftrightarrow\ N(x) \neq N(y)$$

Sind zwei natürliche Zahlen verschieden dann haben sie verschiedene Nachfolger, d.h aus $x \neq y$ folgt $x + 1 \neq y + 1$.

Definition:

$$\begin{aligned} N(x) &= x+1 \\ \\ N(0) &= 1 \\ N(1) &= 2 \\ \\ N(N(N(0))) &= 2\cdot 1+1 \\ &= N(N(0))+N(0) \\ &= (1+1)+1 \end{aligned}$$

$$\begin{array}{lrcl} \text{Defintion} \cdot & x\ldots 0 & = & 0 \\ & x\cdot(y+1) & = & x\cdot y+x \\ & x\cdot(0+1) & = & \underbrace{x\cdot 0}_{0}+x \\ \text{Defintion} + & \| & & \\ & x\cdot 1 & = & x \\ \hline & (1+1)1 & = & 1{+}1 \end{array}$$

3.6 Zerlegung von Zahlendarstellungen

$$a \in \{0,\ldots,B-1\}^n \ (z.B.\ B=10)$$

$$\begin{aligned} <a>_B &= \sum_{i=0}^{n-1} a_i B^i \\ <101>_{10} &= 1\cdot 10^2+0\cdot 10^1+1\cdot 10^0 \\ &= (1+1)+1 \\ &= 101???? \end{aligned}$$

Konvention:
Ziffernfolge $\in \{0,\ldots,9\}$ - Standardinterpretation: $a = a_{10}$

$$\begin{aligned} a &= a_{n-1}\cdot\ldots\cdot a_k\cdot a_{k-1}\cdot\ldots\cdot a_0 \\ <a[n-1:0]>_B &= <a[n-1:k]>_B\cdot B^k+<a[k-1:0]>_B \end{aligned}$$

$B=10,\ k=3$:

$$24197 = 24\cdot 10^3+197$$

3.7 Additionsalgorithmus

Summanden:

$$\begin{aligned} a &= a[n-1:0] \in \{0,1\}^n \\ b &= b[n-1:0] \in \{0,1\}^n \\ c &= c_{-1} \end{aligned}$$

	$(c_1 = 1)$	$(c_0 = 1)$	$(c_{in} = 0)$	c	
	1	0	1	a	
	1	1	1	b	c_i: Übertrag von Stelle i nach Stelle i+1
1	1	0	0	S	

$$\boxed{\begin{array}{ccc} c_{-1} & = & c_{in} \\ < c_1, S_i > & = & a_i + b_i + c_{i+1} \\ S_n & = & c_{n-1} \end{array}}$$

Satz:

$$\boxed{\forall\, i \;< a[i:0] > \;+\; < b[i:0] > \;+ c_{in} \;=< c_i, S[i:0] >}$$

3.7.1 Beweis: Induktion über i

$i = 0$*:*

$$\begin{aligned} < a_0 > + < b_0 > + c_{in} &= \underbrace{a_0 + b_0}_{Def.: <>} + \underbrace{c_{-1}}_{Def.: c_{in}} \\ &= \underbrace{< c_0, S_0 >}_{Def.: <c_0, S_0>} \\ c &= c_{-1} \end{aligned}$$

$i-1 \to i$:

$$
\begin{aligned}
< a[i:0] > + < b[i:0] > + c_{in} &= \underbrace{a_i \cdot 2^i + < a[i-1:0] > + b_i \cdot 2^i + < b[i-1:0] >}_{Zerlegung} + c_{in} \\
&= a_i \cdot 2^i + b_i \cdot 2^i + \underbrace{< c_{i-1}, S[i-1:0] >}_{Induktionsvorraussetzung} \\
&= a_i \cdot 2^i + b_i \cdot 2^i + \underbrace{c_{i-1} \cdot 2^i + < S[i-1:0] >}_{Zerlegung\ n=k=i} \\
&= \underbrace{(a_i + b_i + c_{i-1}) \cdot 2^i}_{Distributivgesetzt} + < S[i-1:0] > \\
&= \underbrace{< c_i, s_i > \cdot 2^i}_{Def.:\ Additionsschritt} + < S[i-1:0] > \\
&= \underbrace{< c_i, s_i, S[i-1:0] >}_{Zerlegung\ n=i+1,\ k=i} \\
&= < c_i, S[i:0] >
\end{aligned}
$$

3.8 Carry-Chain-Adder

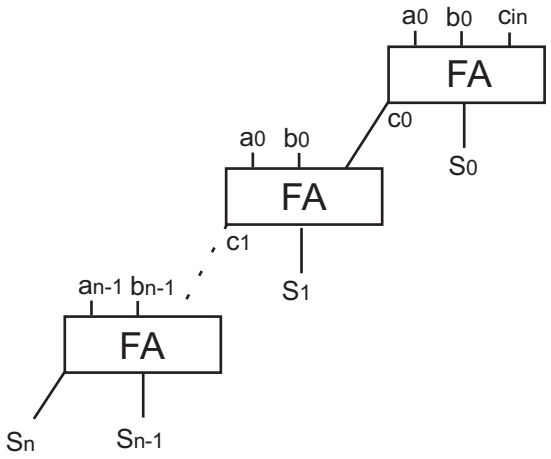


Abbildung 19: Carry-Chain-Adder

Kosten $= n \cdot K(FA)$
Tiefe $\leq n \cdot T(FA)$
Für alle n-bit Addierer:
Tiefe $\geq \log(2n)$

3.9 Definition: Multiplexer (Mux)

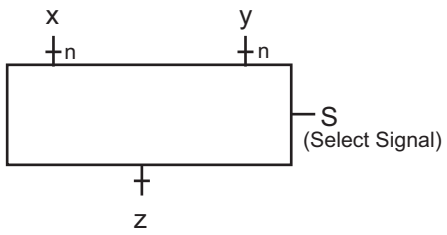


Abbildung 20: Multiplexer

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

3.9.1 1-bit Multiplexer Schaltkreis

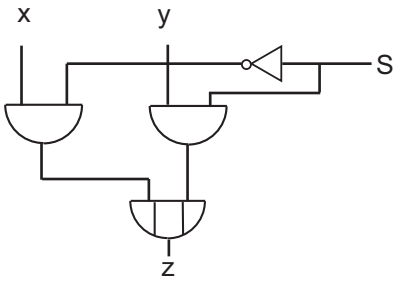


Abbildung 21: 1-bit Multiplexer Schaltkreis

$$z = x\overline{s} \wedge ys$$

3.9.2 n-Mux

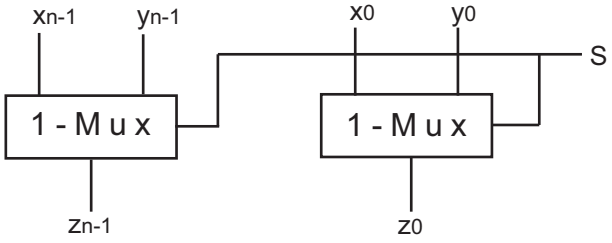


Abbildung 22: n-bit Multiplexer

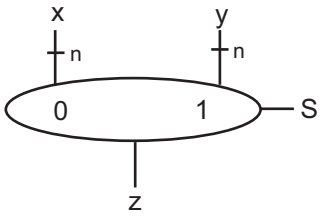


Abbildung 23: Multiplexer Symbol

3.10 Conditional Carry Adder

Definition von Schaltkreis A_n ($A_1 = FA$):

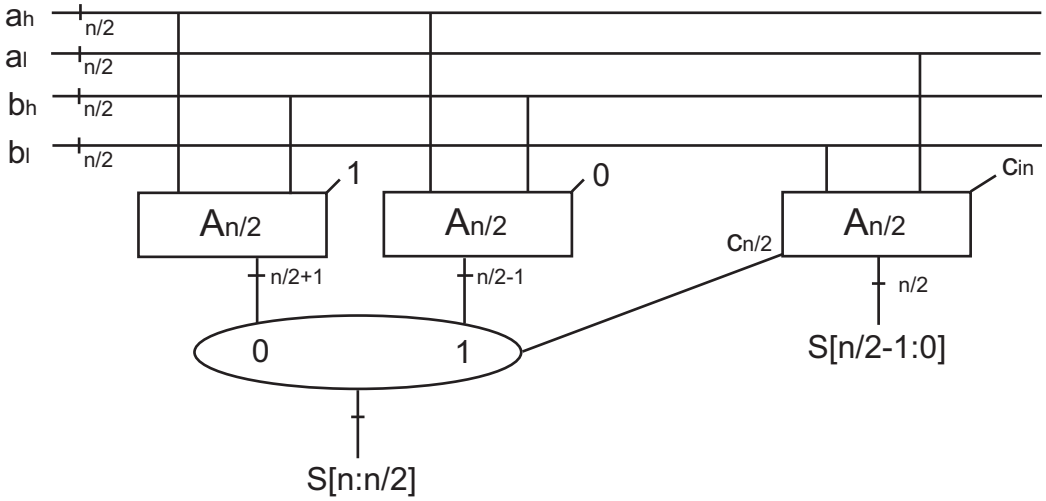


Abbildung 24: Conditional Carry Adder

Tiefe:

$$\begin{aligned} T(A_1) &= T(FA) \\ T(A_n) &= T(A_{n/2}) + \underbrace{T(Mux)}_{3} \end{aligned}$$

$$\boxed{\Rightarrow \; T(A_n) \; = \; O(\log n)}$$

3.11 Exkurs: Modulo

$$(u \; mod \; v \;\; = \;\; Rest \; bei \; Division \; durch \; v \; \in \; \{0, \ldots, v-1\}$$

$$u \; \equiv \; u' \; mod \; v \;\; \Leftrightarrow \;\; (u \; mod \; v) \; = \; (u' \; mod \; v)$$

Division mit Rest bei u < 0?

$$u = a \cdot v + b \; ; \; a \in \mathbb{Z}, \; b \in \{0, \ldots, v-1\}$$

3.12 Subtraktion

Notation:

$$x[n- : 0]$$

$$\overline{x} = (\overline{x_{n-1}}, \ldots, \overline{x_0})$$

Subtraktionsalgorithmus: $x, y \; \in \; \{0,1\}^n$

$$\boxed{< x > - < y > = \; (< x > + < \overline{y} > +1) \; mod \; 2^n}$$

3.12.1 Beispiel

n=4:

$$\begin{array}{rcccc} & 1 & 1 & 0 & 1 \\ - & 0 & 1 & 1 & 0 \end{array} \quad \overset{Regel}{\rightarrow} \quad \begin{array}{rcccc} & 1 & 1 & 0 & 1 \\ + & 1 & 0 & 0 & 1 \\ + & & & & 1 \\ \hline 1 & 0 & 1 & 1 & 1 \end{array} \quad mod\, 2$$

$$0 \quad 1 \quad 1 \quad 1$$

3.13 Two's Complement Zahlen

$x \in \{0,1\}^n$
Definition:

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

$[x]$ heißt two's complement Darstellung von x der Länge n

$$T_n = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$$

$$[\,] : 0,1^n \to T_n$$

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

3.13.1 Beispiel

$$\begin{aligned} [1011] &= -1 \cdot 2^3 + < 11 > \\ &= -8 + 3 \\ &= -5 \end{aligned}$$

3.14 Beweis: Subtraktion

Eigenschaften:

$$a = a[n-1:0] \in \{0,1\}^n$$

3.14.1 Lemma 1

$$< a > = [0a]$$

Beweis:

$$\begin{aligned} [0a] &= -0 \cdot 2^n + < a > \ (Definitionvon[\,]) \\ &= < a > \end{aligned}$$

3.14.2 Lemma 2

$$[a] = < a[n-2:0] > \ mod\ 2^{n-1}$$

Beweis:

$$\begin{aligned} [a] &= -a_{n-1} \cdot 2^{n-1} + < a[n-2:0] > \\ &= < a[n-2:0] > \ mod\ 2^{n-1} \end{aligned}$$

3.14.3 Lemma 3

$$[a] = < a > \ mod\ 2^n$$

Beweis:

$$\begin{aligned} [a] - < a > &= -a_{n-1} \cdot 2^{n-1} + < a[n-1:0] > -(a_{n-1} \cdot 2^{n-1} - < a[n-1:0] >) \\ &= -2 \cdot a_{n-1} \cdot 2^{n-1} \\ &= 0\ mod\ 2^n \end{aligned}$$

Sind Adressen in Rechnern:

- Binärzahlen?
- Two's Complement Zahlen=

Nach Lemma 3: egal, da $mod\ 2^{32}$

3.14.4 Lemma 4

$$[a_{n-1}a] = [a] \qquad \text{(Sign Extension)}$$

Beweis:

$$\begin{aligned} [a_{n-1}a] &= -a_{n-1} \cdot 2^n + < a > \\ &= -a_{n-1} \cdot 2^n + a_{n-1} \cdot 2^{n-1} + < a[n-2:0] > \\ &= -a_{n-1} \cdot 2^{n-1} + < a[n-2:0] > \\ &= [a] \end{aligned}$$

Beispiel:

$$\begin{aligned} [1011] &= [11011] \\ &= [111011] \end{aligned}$$

Definition:

$a \in \{0,1\}^n$ interpretiert als two's complement Zahl:
a_{n-1}: sign-bit

3.14.5 Lemma 5

$$[a] = [\overline{a}] + 1$$

Beweis:

$$\begin{aligned}
[\overline{a}] &= -[\overline{a_{n-1}}] \cdot 2^{n-1} + < [\overline{a}][n-2:0] > \\
&= -[\overline{a_{n-1}}] \cdot 2^{n-1} + \sum_{i=0}^{n-2} \overline{a_i} \cdot 2^i \\
&= -[(1-a_{n-1}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (1-a_i) \cdot 2^i \\
&= x_{n-1} \cdot 2^{n-2} - \sum_{i=0}^{n-2} x_i 2^i - 2^{n-1} + \sum_{i=0}^{n-2} 2^i \\
&= -[x] - 2^{n-1} + 2^{n-1} - 1 \\
&= -[x] - 1
\end{aligned}$$

3.14.6 Beweis der Subtraktion

$$\begin{aligned}
< a > - < b > &= < a > -[0b] && \text{(Lemma 1)} \\
&= < a > +[1\overline{b}] + 1 && \text{(Lemma 5)} \\
&= < a > + < \overline{b} > +1 \; mod \; 2^n && \text{(Lemma 2)}
\end{aligned}$$

3.15 Definition: n-bit-twoc-Addierer

Siehe hierzu auch Abbildung 17 (Seite 22).

$$[S] \; = \; [a] + [b] + c_{in} \; mod \; 2^n$$

Sei A_n ein n-bit-Addierer:

$$\begin{aligned}
< S[n-1:0] > &= < a > + < b > + c_{in} \\
&= -[\overline{a_{n-1}}] \cdot 2^{n-1} + \sum_{i=0}^{n-2} \overline{a_i} \cdot 2^i \\
&= -[(1-a_{n-1}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (1-a_i) \cdot 2^i \\
&= x_{n-1} \cdot 2^{n-2} - \sum_{i=0}^{n-2} x_i 2^i - 2^{n-1} + \sum_{i=0}^{n-2} 2^i \\
&= -[x] - 2^{n-1} + 2^{n-1} - 1 \\
&= -[x] - 1
\end{aligned}$$

$$\begin{aligned}
< S[n-1:0] > &= < a > + < b > + c_{in} \\
&= [a] + [b] + c_{in} \; mod \; 2^n && \text{(Lemma 3)} \\
&= [S[n-1:0]] \; mod \; 2^n && \text{(Lemma 3)}
\end{aligned}$$

3.16 Definition: Arithmetic Unit (AU)

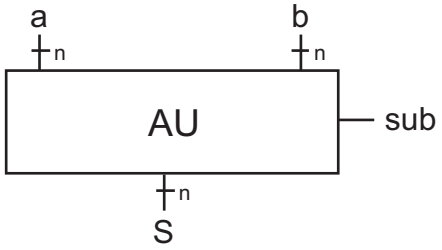


Abbildung 25: Arithmetic Unit

$$[S] = \begin{cases} [a] + [b] \; mod \; 2^n & (sub = 0) \\ [a] + [\overline{b}] \; mod \; 2^n & (sub = 1) \end{cases}$$

Nach Lemma 5:

$$[S] = \begin{cases} [a] + [b] + 0 \; mod \; 2^n & (sub = 0) \\ [a] + [\overline{b}] + 1 \; mod \; 2^n & (sub = 1) \end{cases}$$

3.16.1 Erweiterung der Notation

$$\begin{aligned} a \in \{0,1\}^n ; \; x \in \{0,1\} \\ \circ : \{0,1\}^n \to \{0,1\} \\ a \circ x = (a_{n-1} \circ x, \ldots, a_0 \circ x) \\ x \circ a = (x \circ a_{n-1}, \ldots, x \circ a_0) \end{aligned}$$

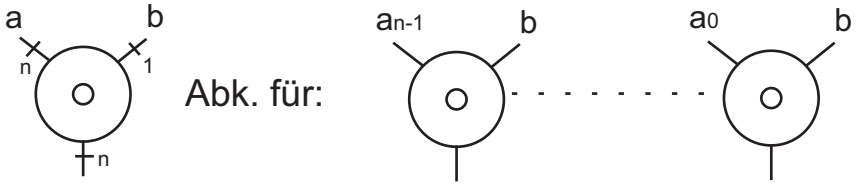


Abbildung 26: Erweiterung der Notation

Beispiel:

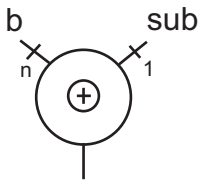


Abbildung 27: Beispiel: Erweiterung der Notation

$$\begin{aligned} b \oplus sub &= (b_{n-1} \oplus sub, \ldots, b_0 \oplus sub) \\ &= \begin{cases} b & (sub = 0) \\ \overline{b} & (sub = 1) \end{cases} \end{aligned}$$

3.17 n-bit-twoc AU (auch für Binärzahlen)

$$[S] \;=\; [a] + [b \oplus sub] + sub \; mod \; 2^n$$

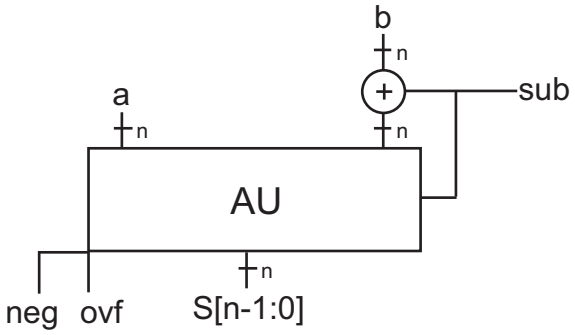


Abbildung 28: n-bit-twoc AU (auch für Binärzahlen)

Behauptung:

addiert/subtrahiert auch Binärzahlen

Frage:

Wann ist $[a] + [b] + c_{in} \;\in\; T_n = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$?

Definition:

$$\begin{aligned} ovf(a, b, c_{in}) \quad &\Leftrightarrow \quad [a] + [b] + c_{in} \;\notin\; T_n \; (overflow) \\ neg(a, b, c_{in}) \quad &\Leftrightarrow \quad [a] + [b] + c_{in} \;<\; 0 \end{aligned}$$

3.17.1 Satz: Overflow

$$ovf(a,b,c_{in}) = 1 \;\Leftrightarrow\; c_{n-1} + c_{n-2}$$

3.17.1.1 Beweis

$$\begin{aligned}
[a]+[b]+c_{in} &= -a_{n-1}\cdot 2^{n-1} - b_{n-1}\cdot 2^{n-1} + < a[n-2:0] > + < b[n-2:0] > \\
&= -a_{n-1}\cdot 2^{n-1} - b_{n-1}\cdot 2^{n-1} + \underbrace{< c_{n-2}, S[n-2:0] >}_{=2^{n-1}\cdot c_{n-2} + < S[n-2:0] >} \\
&\quad (Korrektheitsbeweis\ d.\ Addierers,\ Induktion\ bei\ i = n-2) \\
&= -2^{n-1}(a_{n-1} + b_{n-1} + c_{n-1} - 2c_{n-2}) + < S[n-2:0] > \\
&= -2^{n-1}(\underbrace{< c_{n-1}, S_{n-1} >}_{2\cdot(c_{n-1}+S_{n-1}-2c_{n-2})} + < S[n-2:0] > \\
&= -2^{n}(c_{n-1} - c_{n-2}) + [S[n-1:0]] \\
&= \delta
\end{aligned}$$

Fall $c_{n-1} = c_{n-2}$:

$$\Rightarrow\; \delta = [S[n-1:0]] \;\in\; T_n$$

Fall $c_{n-1} = 1;\; c_{n-2} = 0$:

$$\Rightarrow\; \delta = -2^n + [S] \leq -2^n + 2^{n-1} - 1 \;\notin\; T_n$$

Fall $c_{n-1} = 0;\; c_{n-2} = 1 \;\equiv$ Fall$c_{n-1} = 1;\; c_{n-2} = 0$

3.17.2 Satz: Negation

$$neg(a,b,c_{in}) = 1 \;\Leftrightarrow\; [a]+[b]+c_{in} \;<\; 0 \;\in\; T_{n+1} \backslash T_n$$

$$neg = \boxed{a_{n-1} \oplus b_{n-1} \oplus c_{n-1}}$$

3.18 Definition: Arithmetic Logic Unit (ALU)

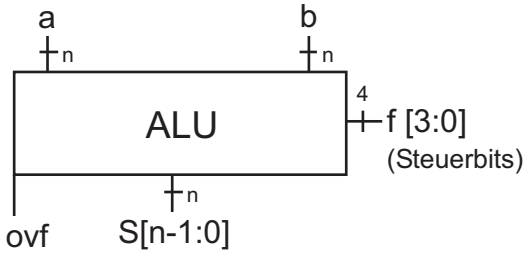


Abbildung 29: Arithmetic Logic Unit

3.18.1 Steuerbit-Tabelle

$f[3:0]$				S	overflow akt./ign.
0	0	0	0	$a +_n b$	akt.
0	0	0	1	$a +_n b$	ign.
0	0	1	0	$a -_n b$	akt.
0	0	1	1	$a -_n b$	ign.
0	1	0	0	$a \wedge b$	
0	1	0	1	$a \vee b$	
0	1	1	0	$a \oplus b$	
0	1	1	1	$b[n/2:0]\ 0^{n/2}$	

$f[3:0]$				$xcc(1,b,f)$	$\equiv$
$<$	$=$	$>$			
1	0	0	0	0	
1	0	0	1	$a > b$	$a - b > 0$
1	0	1	0	$a = b$	$a - b = 0$
1	0	1	1	$a \geq b$	$a - b \geq 0$
1	1	0	0	$a < b$	$a - b < 0$
1	1	0	1	$a \neq b$	$a - b \neq 0$
1	1	1	0	$a \neq b$	$a - b \neq 0$
1	1	1	1	1	

3.18.2 Schaltkreis

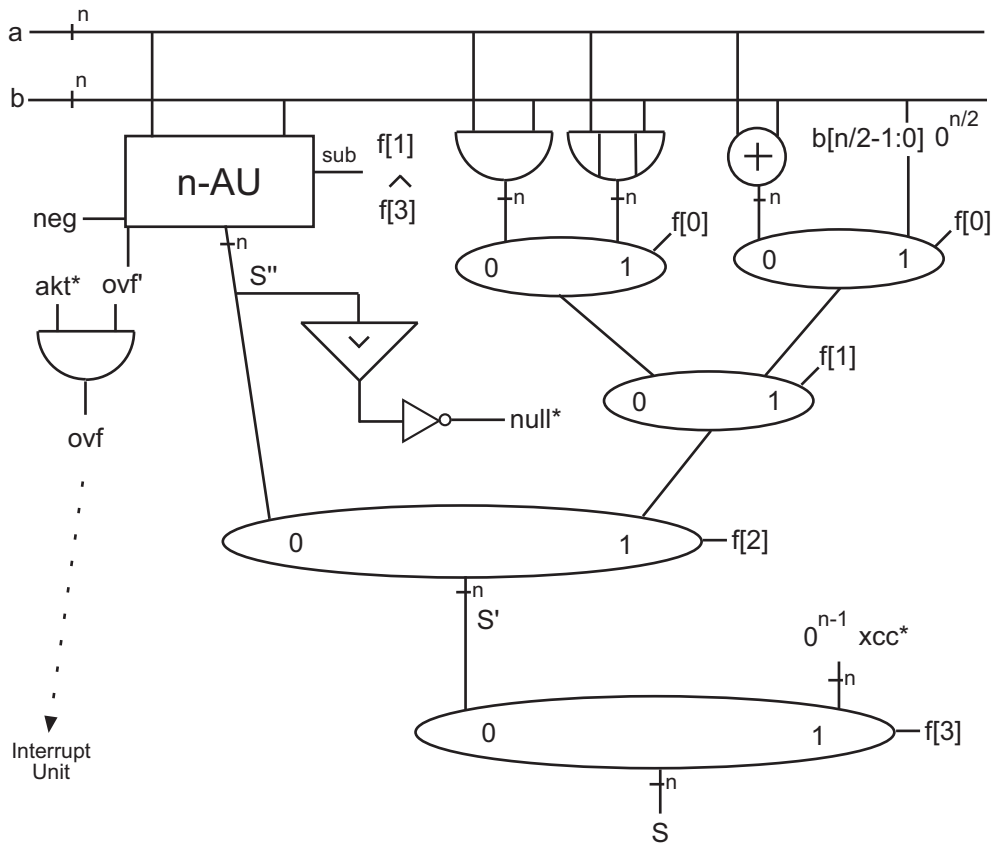


Abbildung 30: Schaltkreis - Arithmetic Logic Unit

3.18.3 akt: Overflow aktivieren/ignorieren

$$\boxed{akt \;=\; \overline{f_3}\;\overline{f_2}\;\overline{f_0}}$$

3.18.4 xcc: Fixed Point Condition Code

$$\begin{aligned} neg(a,b,f) &= 1 \quad \Leftrightarrow \quad a-b < 1 \\ null(a,b,f) &= 1 \quad \Leftrightarrow \quad a-b = 1 \\ pos(a,b,f) &= 1 \quad \Leftrightarrow \quad a-b > 1 \end{aligned}$$

$$pos \;=\; \overline{neg} \wedge \overline{null}$$

$$\boxed{xcc(a,b,f) \;=\; f_2 neg \vee f_1 null \vee f_0 pos}$$

4 Prozessorbau

Prozessor: rechnet in Schritten
Hardware: rechnet *besser* auch in Schritten

1. Mathematische Maschinen
2. Register, RAM, multiport-RAM'S (Hardware-Erweiterungen)
3. DLX_0: vereinfachter DLX-Instruktionssatz (DLX $\approx$ MIPS)
4. Bau von DLX_0-Prozessor

4.1 Mathematische Maschinen

Aus Programmier-Sprachen $\underbrace{x := x + 1}_{nicht\ Gleichheit!}$

Definition von x^t:
x vor/während Schritt t
$\rightarrow \ x^{t+1} = x^t + 1$

x: Register, RAM, ... : Hardware

x: für Benutzer sichtbare Register : Assembler-Programmierung

x: Variablen eines Programms : Programmier-Sprachen

x: für Benutzer sichtbare Datenstrukturen : Betriebssystem

Definition: mathematische Maschinen

$$\begin{aligned} M &= (\tau, \delta, c_0) \\ \tau &: Menge\ von\ Konfigurationen \\ &\quad (Zustand\ d.\ Maschine\ in\ einem\ Schritt) \\ \delta &: \tau \rightarrow \tau \\ &\quad (\ddot{U}bergangsfunktion) \\ c_0 \in \tau &: Startkonfiguration \end{aligned}$$

Rechnung:

$$Folge(C^0, c^1, c^2, \ldots) : \ c^{i+i} = f(c_i)\ \forall i$$

4.2 Speicherelemente

4.2.1 Register (n-Bit)

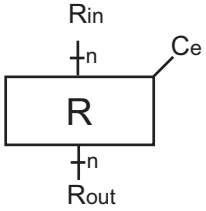


Abbildung 31: Register

$$R \in \{0,1\}^n \ (in\ Register\ gesp.\ bits)$$
$$c_e \in \{0,1\} \ (clock\ enable)$$

Arbeitsweise:

$$R := R_{in}\ falls\ c_e = 1$$
$$R = \begin{cases} R_{in}^t & : c_e^t = 1 \\ R^t & : sonst \end{cases}$$

Beispiel:

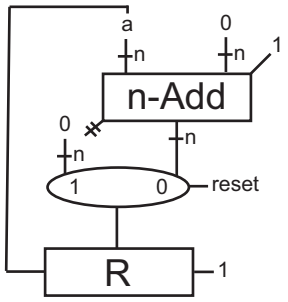


Abbildung 32: Register (Beispiel)

$$\begin{aligned} reset^0 &= 1 \\ \underbrace{reset^t}_{(t>0)} &= 0 \end{aligned}$$

$$\begin{aligned} R_{in}^0 &= 0^n \\ R_{in}^1 &= 0^n \\ R_{in}^t &= R^t +_n \underbrace{1_n}_{1_n = bin_n(1)} \\ R^{t+1} &= R^t +_n 1 \end{aligned}$$

Lemma:

$$R^t \ = \ bin_n(t)$$

4.2.2 Random Access Memory (RAM)

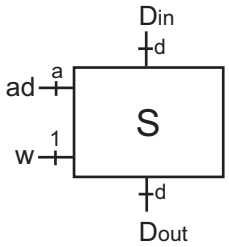


Abbildung 33: $2^a \times \ d$ RAM

4.2.2.1 $2^a \times \ d$ RAM

$$S: \ \{0,1\}^a \rightarrow \{0,1\}^d$$

$$s^{t+1}(x) \ = \ \begin{cases} D_{in}^t & : \ x = ad^t \wedge w^t = 1 \\ S^t(x) & : \ sonst \end{cases}$$

$$D_{out}^t \ = S^t(ad^t) \ (falls \ w^t = 0)$$

Allgemeine Regeln zur Wohldefiniertheit: S.M. Müller, W. Paul:

The Complexity of Simple Computer Architectures, Springer 1995.

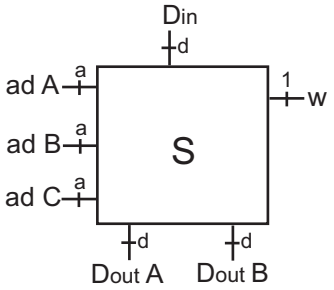


Abbildung 34: $3 - Port$ RAM

4.2.2.2 $3 - Port$ RAM

$$S: \ \{0,1\}^a \rightarrow \{0,1\}^d$$

$$S^{t+1}(x) \ = \ \begin{cases} D_{in}C^t & : \ x = adC^t \wedge w^t = 1 \\ S^t(x) & : \ sonst \end{cases}$$

$$D_{out}A^t \ = \ S^t(adA^t): \ (adA \neq adC \wedge w^t = 1) \vee w^t = 0$$

$$D_{out}B^t \ = \ S^t(adB^t): \ (adB \neq adC \wedge w^t = 1) \vee w^t = 0$$

4.3 Instruktionssatz des DLX_0-Prozessors

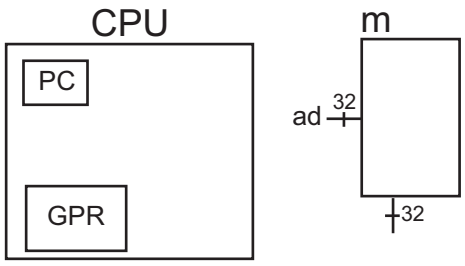


Abbildung 35: DLX_0 Instruktionssatz

Central Processing Unit:
Memory: Wort-Adressiert
Adressen $\in \{0,1\}^{32}$
$m: \{0,1\}^{32} \rightarrow \{0,1\}^{32}$
$m^t(ad)$: Inhalt von Speicherzelle ad zur Zeit t
Program Counter (PC) $\in \{0,1\}^{32}$
General Purpose Register File (GPR): $\{0,1\}^5 \rightarrow \{0,1\}^{32}$
32 Adressen
32 Register
GPR[i] $\in \{0,1\}^{32}$; $i \in \{0,1\}^5$

4.3.1 Konfiguration

$$c = (c.PC, c.GPR, c.m)$$

$$\begin{aligned} c.PC &\in \{0,1\}^{32} \\ c.GPR &: \{0,1\}^5 \rightarrow \{0,1\}^{32} \\ c.m &: \{0,1\}^{32} \rightarrow \{0,1\}^{32} \end{aligned}$$

4.3.2 Rechnung

Rechnung: $(c^0, c^1, c^2, \ldots)$

$$c^{i+1} = \delta(c^i)$$

I_i = die i-te ausgeführte Instruktion

$$c^0 \xrightarrow{I_0} c^1 \xrightarrow{I_1} c^2 \ldots c^i \xrightarrow{I_i} c^{i+1}$$

$$I(c^t) = c^t.m[c^t.PC] \in \{0,1\}^{32}$$

$$Notation: \; I(c^t) \equiv I_t$$

I_t wird meistens auch im Instruction Register (IR) zwischengespeichert. Effekt von I_t formal spezifiziert durch:

$$c^{t+1} = (c^{t+1}.PC, c^{t+1}.GPR, c^{t+1}.m) = \delta(c^t)$$

4.3.3 3 Instruktionsformate

I-Type:

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OPC	RS1	RD	imm

R-Type:

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11	10 9 8 7 6	5 4 3 2 1 0
OPC	RS1	RS2	RD	(SA)	fu

J-Type:

31 30 29 28 27 26	25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OPC	imm

Der Typ der Instruktion wird im OpCode festgelegt:

$$\underbrace{opc(c)}_{c=DLX_0-Konfiguration} = I(c)[31:26]$$

Prädikate:

$$\begin{aligned} R-Type(c) &\Leftrightarrow opc(c) = 0^6 \\ J-Type(c) &\Leftrightarrow opc(c) \in \{000010, 000011, 111110, 111111\} \\ I-Type(c) &\Leftrightarrow /(R-Type(c) \vee J-Type(c)) \end{aligned}$$

Felder definieren:

$$\begin{aligned} RS1(c) &= I(c)[25:21] \\ RS2(c) &= I(c)[20:16] \\ RD(c) &= \begin{cases} I(c)[20:16] & \text{: I-Type(c)} \\ I(c)[15:11] & \text{sonst} \end{cases} \\ fu^1(c) &= I(c)[5:0] \\ imm(c) &= \begin{cases} I(c)[15:0] & \text{: I-Type(c)} \\ I(c)[25:0] & \text{sonst} \end{cases} \end{aligned}$$

4.3.4 Definition: Signed Extension

Sei $a \in \{0,1\}^b$ (interpretiert als twoc):

$$sxt(a) = a[b-1]_{b-1}^{c}\, a \in \{0,1\}^{b+c}$$

Lemma:

$$[imm] = [sxt(imm)]$$

4.3.5 $GPR[0^5]$

Für das GPR des DLX_0 soll immer gelten:

$$\boxed{c.GPR[00000] = 0^{32}}$$

4.3.6 Instruktionen

4.3.6.1 Load/Store

load word (lw):
$lw(c) = 1 \Leftrightarrow op(c) = 100011$
effective address:

$$ea(c) = \underbrace{c.GPR[\underbrace{RS1(c)}_{\{0,1\}^5}]}_{\{0,1\}^{32}} +_{32} sxt^2(\underbrace{imm(c)}_{\{0,1\}^{16}})$$

$$\boxed{c^{t+1}.GPR\,[RD\,(c^t)] = c^t.m\,[ea(c^t)]}$$

store word (sw):
$sw(c) = 1 \Leftrightarrow op(c) = 101011$

$$c^{t+1}.m[x] = \begin{cases} c^t.m\,[ea(c^t)] & : sw(c) \wedge x = ea(c^t) \\ c^t.m\,[x] & : sonst \end{cases}$$

[2] sxt: signed Extension

4.3.6.2 Compute / Compute Immediate

compute immedate (comp.imm:)
$comp.imm(c) = 1 \Leftrightarrow I(c)[31:30] = 01$
Vergleiche hierzu Kapitel 3.18. Wir benötigen hierzu eine 32-bit ALU.

$$c^{t+1}.GPR[x] = \begin{cases} aluop(c.GPR[RS1(c)], sxt(imm(c)), I(c)[29:26]) & : x = RD(c) \wedge x \neq 0^5 \\ c^t.GPR[x] & : sonst \end{cases}$$

compute (comp):
$comp(c) = 1 \Leftrightarrow R - Type \wedge I(c)[5:4] = 00$

$$c^{t+1}.GPR[x] = \begin{cases} aluop(c.GPR[RS1(c)], c.GPR[RS2(c)], I(c)[3:0]) & : x = RD(c) \wedge x \neq 0^5 \\ c^t.GPR[x] & : sonst \end{cases}$$

Zusammenfassung:

$$lop = c.GPR[RS1(c)]$$

$$rop = \begin{cases} sxt(imm(c)) & : comp.imm(c) \\ c.GPR[RS2(c)] & : sonst \end{cases}$$

$$lop = \begin{cases} I(c)[29:26] & : comp.imm(c) \\ I(c)[3:0] & : sonst \end{cases}$$

4.3.6.3 Branch or Jump taken

$bjtaken(c) = 1 \Leftrightarrow btaken(c) \wedge jump(c)$
$branch(c) = 1 \Leftrightarrow I(c)[31:27] = 11010$
$AeqZ(c) = 1 \Leftrightarrow c.GPR[RS1(c)] = 0$

$$jump(c) = j(c) \wedge jal(c) \wedge jr(c) \wedge jalr(c)$$

$$\begin{aligned} beqz(c) &= branch(c) \wedge I(c)[26] = 0 \\ bnez(c) &= branch(c) \wedge I(c)[26] = 1 \end{aligned}$$

$$\begin{aligned} btaken(c) &= \underbrace{c.GPR[RS1(c)] = 0 \wedge beqz(c) \vee c.GPR[RS1(c)] \neq 0 \wedge bnez(c)}_{Definition} \\ &= branch(c) \wedge [c.GPR[RS1(c)] = 0 \wedge I[26] = 0 \vee c.GPR[RS1(c)] \neq 0 \wedge I[26] = 1)] \end{aligned}$$

$\Rightarrow$ Lemma:

$$btaken(c) \;=\; branch(c) \wedge (AeqZ(c) \oplus I[26])$$

$$c^{t+1}.PC = \begin{cases} c^t.PC(c^t) \;+_{32}\; sxt(imm(c^t)) & : btaken(c^t) \wedge j(c^t) \wedge jal(c^t)^* \\ c^t.GPR[RS1(c^t)] & : jr(c^t) \wedge jalr(c^t) \\ c.PC(c^t) \;+_{32}\; 1_{32} & : sonst \end{cases}$$

**Achtung*:

$$imm(c) \;\in\; \{0,1\}^{16} \; oder \; \{0,1\}^{26}$$

$$sxt(imm) = \begin{cases} imm[15]^{16} imm & : imm(c) \;\in\; \{0,1\}^{16} \\ imm[25]^{6} imm & : imm(c) \;\in\; \{0,1\}^{26} \end{cases}$$

$$c^{t+1}.GPR[x] = \begin{cases} 0^{32} & : x = 0^5 \\ c^t.m[c^t] & : lw(c^t) \;\wedge\; x = RD(c^t) \;\wedge\; x \neq 0^5 \\ aluop(lop(c^t), rop(c^t), fcode(c^t)) & : (comp.imm(c^t) \;\vee\; comp(c^t)) \;\wedge\; x = RD(c^t) \\ \underbrace{c.PC \;+_{32}\; 1_{32}}_{Rücksprungadresse} & : (jal(c^t) \;\vee\; jalr(c^t)) \;\wedge\; x = 1^5 \\ c^t.GPR[x] & : sonst \end{cases}$$

4.3.7 Zusammenfassung: Instruktionen

$$c^{t+1}.m[x] = \begin{cases} c^t.m\,[ea(c^t)] & : sw(c) \wedge x = ea(c^t) \\ c^t.m\,[x] & : sonst \end{cases}$$

$$c^{t+1}.PC = \begin{cases} c^t.PC(c^t) \;+_{32}\; sxt(imm(c^t)) & : btaken(c^t) \wedge j(c^t) \wedge jal(c^t)^* \\ c^t.GPR[RS1(c^t)] & : jr(c^t) \wedge jalr(c^t) \\ c.PC(c^t) \;+_{32}\; 1_{32} & : sonst \end{cases}$$

$$c^{t+1}.GPR[x] = \begin{cases} 0^{32} & : x = 0^5 \\ c^t.m[c^t] & : lw(c^t) \;\wedge\; x = RD(c^t) \;\wedge\; x \neq 0^5 \\ aluop(lop(c^t), rop(c^t), fcode(c^t)) & : (comp.imm(c^t) \;\vee\; comp(c^t)) \;\wedge\; x = RD(c^t) \\ \underbrace{c.PC \;+_{32}\; 1_{32}}_{Rücksprungadresse} & : (jal(c^t) \;\vee\; jalr(c^t)) \;\wedge\; x = 1^5 \\ c^t.GPR[x] & : sonst \end{cases}$$

4.4 Hardware-Konfiguration

$$h \;=\; (h.PC, \quad \underbrace{h.GPR}_{modifiziertes\ 3-Port\ RAM}, h.m)$$

$$
\begin{aligned}
h.GPR &: \{0,1\}^5 \rightarrow \{0,1\}^{32} \\
S^{t+1}[x] &= \begin{cases} 0^{32} & : x = 0^5 \\ D_{in}^t & : w^t \wedge x \neq 0^5 \wedge x = adC^t \\ S^t[x] & : sonst \end{cases} \\
h.PC &\in \{0,1\}^{32} \\
h.m &: \{0,1\}^{32} \rightarrow \{0,1\}^{32}
\end{aligned}
$$

Ziel ist es, zu beweisen, dass die beiden mathematischen Maschinen (Hardware und DLX_0) gleich funktionieren.

4.4.1 Unterteilung der Instruktionen in Stufen

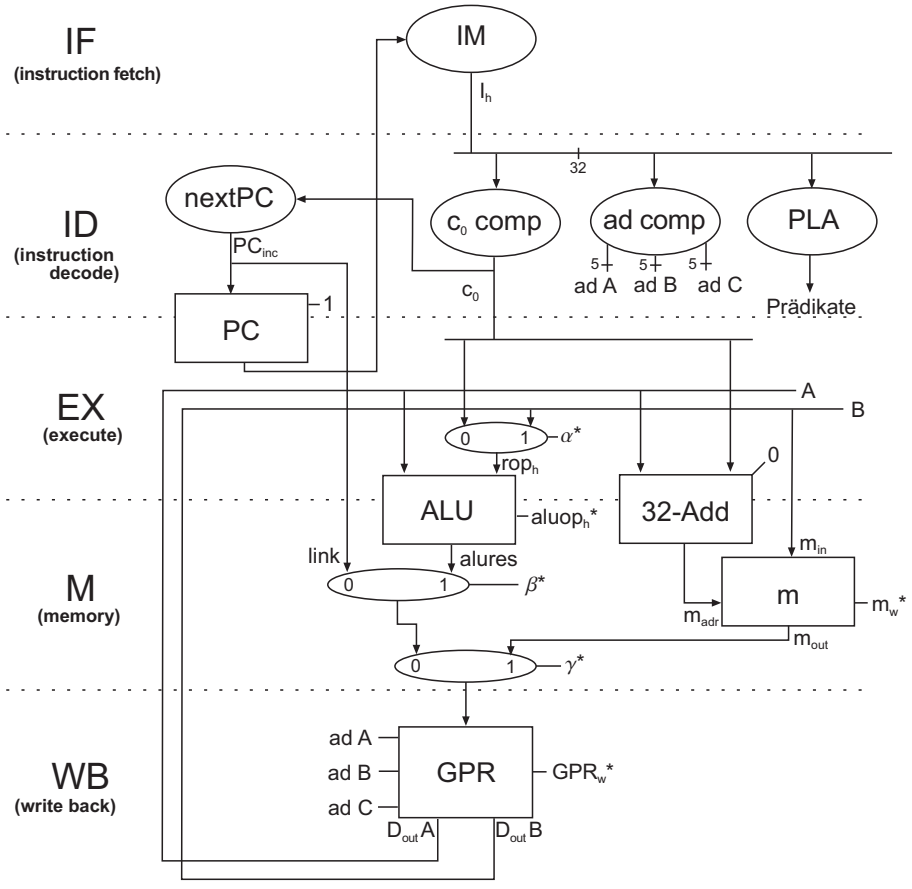


Abbildung 36: DLX_0-Hardware Konfiguration in Stufen

Noch zu definieren:
$\alpha,\ \beta,\ \gamma\ GPR_w,\ m_w,\ c_0comp,\ adcomp,\ nextPC,\ aluop_h,\ PC_{inc}$

4.5 Beweis der Gleichheit zwischen DLX_0- und Hardware-Konfiguration

$$\underbrace{c^0, c^1, c^2, \ldots c^{i+1} \;=\; \delta(c^i)}_{DLX_0\ Rechnung}$$

$$\underbrace{h^0, h^1, h^2, \ldots h^{i+1} \;=\; \delta(h^i)}_{Hardware\ Rechnung}$$

Zu zeigen $\forall\, i$:

$$\boxed{\begin{array}{rcl} h^i.PC & = & c^i.PC \\ h^i.GPR & = & c^i.GPR \\ h^i.m & = & c^i.m \\ \text{(Induktionsbehauptung)} & & \end{array}}$$

Vorraussetzung:

- Induktionsbehauptung(0) gilt
- $c^0.m[x] \;=\; IM[x]$ für alle $x \;\in\; codesection$
- kein selbsmodifizierender Code
  $\forall\, i$:
  $sw(c^i) \;\Rightarrow\; ea(c^i) \;\notin\; codesection$
  $c^i.PC \;\in\; codesection$

Induktionsschluss $i \rightarrow i+1$:

$$c^i.PC \;=\; h^i.PC$$

$$\begin{array}{rcll} I_h(h^i) & = & IM[h^i.PC] & \text{(Konstruktion)} \\ & = & IM[c^i.PC] & \text{(Induktions-Vorraussetzung)} \\ & = & m[c^0.PC] & \\ & = & m^i[c^i.PC] & \text{(keine Selbsmodifikation)} \\ & = & I[c^i] & \end{array}$$

4.5.1 $c_0 comp$

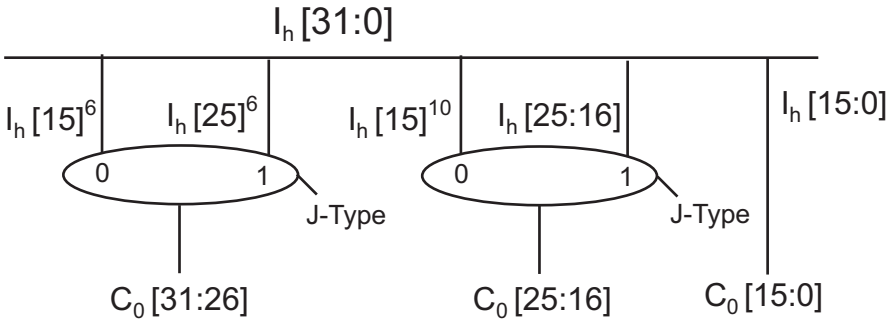


Abbildung 37: DLX_0-Beweis ($c_0 comp$)

Lemma:

$$J - Type(h^i) \;=\; J - Type(c^i)$$
$$(aus\ Darstellungssatz!)$$

Lemma:

$$c_0(h^i) \;=\; sxt(imm(c^i))$$

4.5.2 *adcomp*

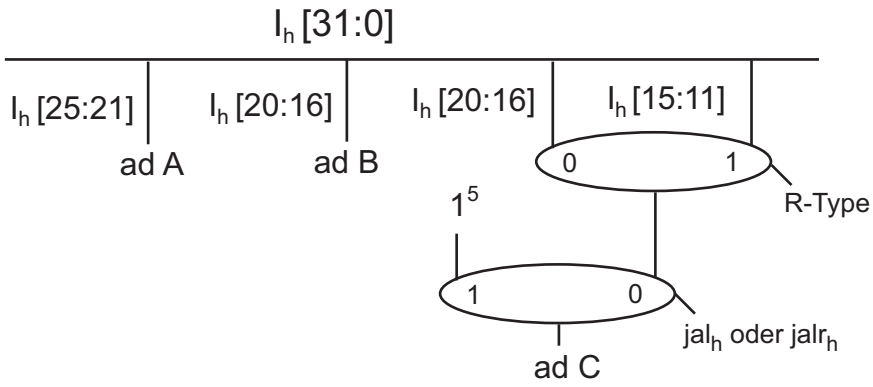


Abbildung 38: DLX_0-Beweis (*adcomp*)

Lemma:

$$
\begin{aligned}
ad\,A(h^i) &= RS1(c^i) && : \text{I-Type} \vee \text{R-Type} \\
ad\,B(h^i) &= \begin{cases} RS1(c^i) & : \text{R-Type} \\ RD(c^i) & : \text{I-Type} \end{cases} \\
ad\,C(h^i) &= \begin{cases} RD(c^i) & : \text{I-Type} \wedge \text{R-Type} \\ 1^5 & : jal(c^i) \ \wedge \ jalr(c^i) \end{cases} \\
A(h^i) &= \underbrace{h^i.GPR}_{c^i.GPR\ (ind.\ Vor.)} \ [\ \underbrace{ad\,A(h^i)}_{RS1(c^i)\ (eben\ gezeigt)} \ ] \\
\Rightarrow A(h^i) &= lop(c^i)\ (falls\ I-Type(c^i) \vee R-Type(c^i))
\end{aligned}
$$

$$
B(h^i) = \left\{ \begin{array}{ll} c^i.GPR[RS2(c^i)] & : R-Type \\ c^i.GPR[RD(c^i)] & : I-Type \end{array} \right\}
$$

$$
\alpha = R-Type_h
$$

$$
\begin{aligned}
rop_h(h^i) &= \left\{ \begin{array}{ll} c^i.GPR[RS2(c^i)] & : R-Type_h(h^i) = 1 \\ c^i.GPR[RD(c^i)] & : R-Type_h(h^i) = 0 \end{array} \right\} \\
&= \left\{ \begin{array}{ll} c^i.GPR[RS2(c^i)] & : R-Type(c^i) = 1 \\ c^i.GPR[RD(c^i)] & : R-Type(c^i) = 0 \end{array} \right\} PLA-Lemma*
\end{aligned}
$$

*Lemma:
$\forall$ Prädikate P: $P(c^i)\ P_h(h^i)$

4.5.3 $aluop_h$

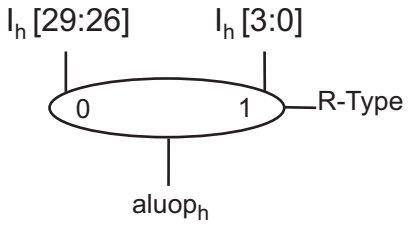


Abbildung 39: DLX_0-Beweis ($aluop_h$)

Mit ALU-Korrektheit:

$$\Rightarrow alures(h^i) = aluop(lop(c^i), rop(c^i), aluop(c^i))$$

4.5.4 m_w

Mit Addierer-Korrektheit:

$$\begin{aligned} \Rightarrow m_{ad}(h^i) &= A(h^i) +_{32} c_0(h^i) \\ &= c^i.GPR[RS1(c^i)] +_{32} sxt(imm(c^i)) \\ &= ea(c^i) \end{aligned}$$

$$m_{in}(h^i) = c^i.GPR[RD(c^i)]$$

Definition:

$$\begin{aligned} m_w &= sw_h \\ &= sw(c^i)\ (PLA - Lemma) \end{aligned}$$

4.5.5 $c^i.m \;=\; h^i.m$

$$\begin{aligned}
\underbrace{h^{i+1}.m[x]}_{neuer\ HW-Speicher} &= \left\{ \begin{array}{ll} m_{in} & : m_w(h^i) = 1 \ \wedge \ x = m_{ad}(c^i) \\ h^i.m[x] & : sonst \end{array} \right\} Def.:\ HW - Speicher \\
&= \left\{ \begin{array}{ll} c^i.GPR[RD(c^i)] & : sw(c^i) = 1 \ \wedge \ x = ea(c^i) \\ \underbrace{h^i.m[x]}_{c^i.m[x]\ (Ind.Vor.} & : sonst \end{array} \right. \\
&= c^{i+1}.m
\end{aligned}$$

$$\rightarrow \boxed{\text{Bewiesen:} \quad c^i.m \;=\; h^i.m}$$

4.5.6 $GPR.w$

$$GPR.D_{in}(h^i) \;=\; \left\{ \begin{array}{ll} aluup(lop(c^i), rop(c^i), aluop(c^i)) & : \alpha(h^i) = 0 \ \wedge \ \beta(h^i) = 0 \\ m_{out}(h^i) & : \gamma(h^i) = 1 \\ PC^i(h^i) & : \beta(h^i) = 1 \ \wedge \ \gamma(h^i) = 0 \end{array} \right.$$

Definition:

$$\beta \;=\; jal_h \vee jalr_h \ (\gamma = lw_h)$$

$$\begin{aligned}
\gamma(h^i) &= lw_h(h^i) \\
&= lw(c^i)
\end{aligned}$$

$$\begin{aligned}
m_{out}(h^i) &= \underbrace{h^i.m}_{c^i.m\ (Ind.\ Vor)} \ [\ \underbrace{m_{ad}(h^i)}_{ea(c^i)\ (eben)}\ ] \\
&= c^i m[ea(c^i)]
\end{aligned}$$

Bewiesen mit Lemma *nextPC*:

$$PC'(h^i) \;=\; h^i.PC \;+_{32}\; 1_{32}$$

Definition:

$$GPR.w \;=\; lw_h \;\vee\; compute_h \;\vee\; compute.imm_h \;\vee\; jal_h \;\vee\; jalr_h$$

$$h^{i+1}.GPR[x] = \begin{cases} o^{32} & : x = 0^5 \text{ (Def. HW-Multiport-RAM)} \\ aluup(lop(c^i), rop(c^i), aluop(c^i)) & : comp(c^i) \vee comp.imm(c^i) \wedge x = RD(c^i) \\ c^i.m(ea(c^i)) & : x = RD(c^i) \wedge sw(c^i) \\ \underbrace{h^i.PC +_{32} 1_{32}}_{c^{i+1}.GPR[x]} & : jal \vee jalr \wedge x = 1^5 \\ \underbrace{h^i.GPR[x]}_{c^i.PC\ (Ind.Vor.)} & : sonst \end{cases}$$

4.5.7 PC_{inc}

$$\begin{aligned} I_h(h^i) &= I(c^i) \\ c^{i+1} &= h^{i+1}.GPR \\ c^{i+1}.m &= h^{i+1}.m \\ PC_{inc}(h^i) &= c^i.PC +_{32} 1_{32} \end{aligned}$$

4.5.8 Spezifikation: n-Inc

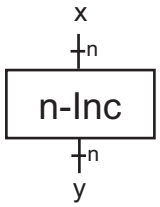


Abbildung 40: Spezifikation - n-Inc

$$\boxed{y = x +_n 1_n}$$

Nach Induktionsvorraussetzung:

$$\begin{aligned} PC_{inc}(h^i) &= h^i.PC +_{32} 1_{32} \\ &= c^i.PC +_{32} 1_{32} \end{aligned}$$

4.5.9 $nextPC$

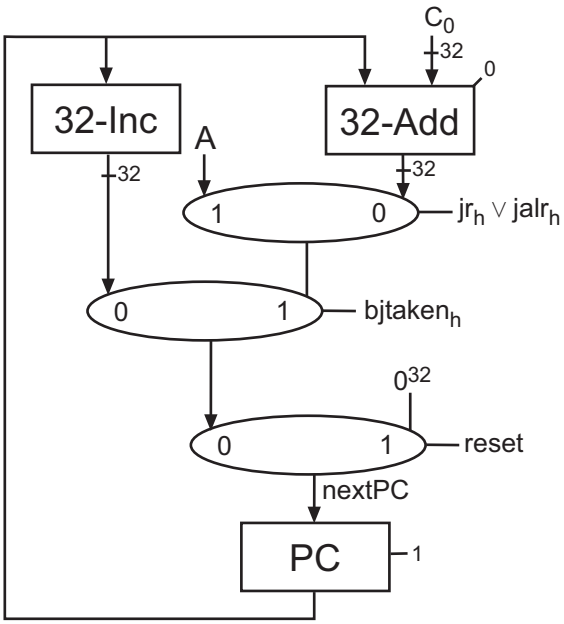


Abbildung 41: $nextPC$

PLA-Lemma:

$$
\begin{aligned}
(jr_h \vee jal_h)(h^i) &= jr(c^i) \vee jalr(c^i) \\
bjtaken(c^i) &= jump(c^i) \vee branch(c^i) \wedge AeqZ(c^i) \oplus I(c^i)[26] \\
jump_h &= j_h \vee jal_h \vee jr_h \vee jalr_h \\
\Rightarrow \quad jump_h(h^i) &= jump(c^i)
\end{aligned}
$$

Hardware:

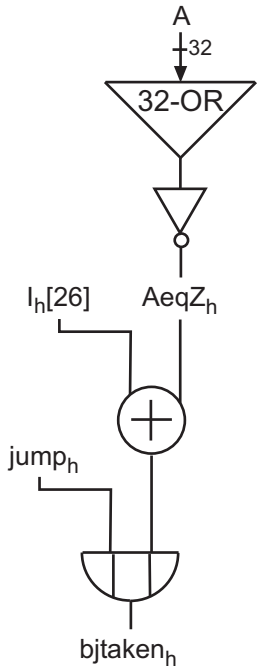


Abbildung 42: $bjtaken_h$

früher:

$$\begin{array}{rcl} A(h^i) & = & c^i.GPR[RS1(c^i)] \\ AeqZ_h(h^i) = 1 & \Leftrightarrow & \forall j : A(h^i)[j] = 0 \\ & \Leftrightarrow & A(h^i) = 0^{32} \\ & & \\ & \Rightarrow & AeqZ(h^i) = AeqZ(c^i) \\ & & \\ bjtaken_h(h^i) & = & bjtaken_h(c^i) \end{array}$$

für $reset = 0$:

$$h^{i+1}.PC = nextPC_h(h^i) = \begin{cases} c^i.GPR[RS1(c^i)] & : jr(c^i) \vee jalr(c^i) \\ c^i.PC +_{32} sxt(imm(c^i)) & : bjtaken(c^i) \wedge \sim (jr(c^i) \vee jalr(c^i)) \\ c^i.PC +_{32} 1_{32} & : sonst \end{cases}$$

5 Compiler für C_0 (PASCAL mit C-Syntax)

Vorgehensweise:

1. Definition der Syntax von C_0
   (entspricht den bool'schen Ausdrücken, DLX_0-Instruktionsformate)

2. Definition der Sematik von C_0: Definition einer abstrakten C_0-Maschine
   (entspricht $\delta(h)$, $\delta(c)$)

3. Compiler: $C_0 \rightarrow DLX_0$

4. Simulationssatz:
   p: C_0-Programm $\overset{Compiler}{\rightarrow}$ $Code(p)$: DLX_0-Programm
   Zu zeigen:
   DLX_0 mit $Code(p)$ simuliert eine C_0-Maschine mit Programm p

5.1 Kontextfreie Grammatiken

5.1.1 Beispiel: Kontextfreie Grammatik

VC Variable oder Konstante

$$\underbrace{B \rightarrow VC|(\sim B)|B \wedge B)|(B \vee B)|(B \oplus B)}_{Produktionensystem}$$

Abkürzung für:

$$\{\underbrace{B \rightarrow VC}_{Produktion}, B \rightarrow \sim B, \ldots, B \rightarrow B \oplus B\}$$

N	=	$\{B\}$	Nichtterminalalphabet
T	=	$\{(,), VC, \wedge, \vee, \oplus, \sim\}$	Terminalalphabet
S	=	B	Startsymbol

5.1.2 Definition: Kontextfreie Grammatik

$$\underbrace{Grammatik\ G = (T, N, P, S)}_{Kontextfreie\ Grammatik\ (cfg)}$$

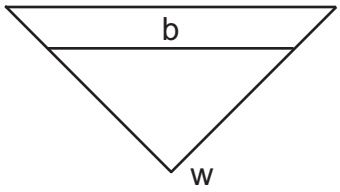
T	:	endlich, Terminalalphabet	
N	:	endlich, Nichtterminalalphabet	
S	=	Startsymbol	
P	$\subset$	$(N \times (N \cup T)*)$	(Produktionensystem)
		$(n, w) \in P$	(Produktion)

Schreibweise:

$$\underbrace{n \rightarrow w}_{Aus\ n\ kann\ w\ abgeleitet\ werden}$$

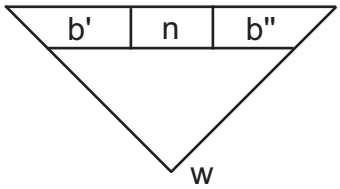
5.1.3 Arbeitsweise von Grammatiken

Definition: induktiv zu G
Q ist Ableitungsbaum zu G mit Wurzel w und Blattwort b:

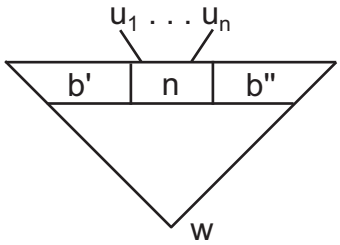


S ist ein Ableitungs-Baum mit Wurzel S und Blattwort S:

Sei Q Ableitungs-Baum mit Wurzel w und Blattwort $b = b'\ n\ b''\ \ n \in N$:



Sei $n \rightarrow u \in P$
$\Rightarrow u = u_1, \ldots, u_n$
Ableitungsbaum mit Wurzel w und Blattwort $b = b'\ u\ b''$:



Beispiel:

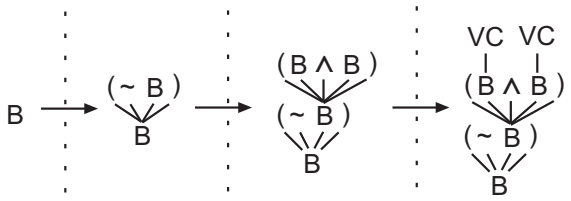


Abbildung 43: Beispiel - Arbeitsweise von Grammatiken

Blattwort: $(\sim (VC \wedge VC))$

5.1.4 Weitere Definitionen

Zi	$\rightarrow$	$0\|1\|2\|3\|4\|5\|6\|7\|8\|9$	(Ziffer)
ZiF	$\rightarrow$	$Zi\|ZiF\ Zi$	(Ziffernfolge)
V	$\rightarrow$	$X\,ZiF$	(Variable)
VC	$\rightarrow$	$ZiF\|V$	(Variable oder Konstante)
A	$\rightarrow$	$VC\|-_1 A\|A+A\|A-_2 A\|A*A\|A/A\|(A)$	(Ausdruck)

Beispiele:

$$VC \xrightarrow{*} X \underbrace{\underbrace{\underbrace{\underbrace{\underbrace{1}_{Zi}\ \underbrace{0}_{Zi}}_{ZiF}}_{ZiF}}_{V}}_{VC}$$

$$A \xrightarrow{*} \underbrace{\underbrace{\underbrace{X1}_{VC}}_{A} + \underbrace{\underbrace{X2}_{VC} * \underbrace{X3}_{VC}}_{A}}_{A}$$

Dies ist fast ein Schaltkreis (kann als Schaltkreis dargestellt werden).
Da Schaltkreise eine Semantik besitzen, besteht auch die berechtigte Hoffnung, die Semantik ves Ausdrucks ableiten zu können.

5.1.5 Problem: Alternativer Ableitungsbaum

An dem vorigen Beispiel kann man ebenfalls zeigen, dass es verschiedene Ableitungs-Bäume zum gleichen Blattwort gibt:

$$\underbrace{\underbrace{\underbrace{X1}_{VC} + \underbrace{X2}_{VC}}_{A} * \underbrace{X3}_{VC}}_{A}}_{A}$$

5.1.6 Definition: Eindeutigkeit von Grammatiken

$$G \text{ heißt eindeutig} \quad \Leftrightarrow \quad \begin{array}{l} \forall\, w \in L(G) \\ \exists \text{ genau 1 Ableitungs-Baum} \\ \text{mit Wurzel } s \text{ und Blattwort } w \end{array}$$

Dazu beschränken wir A:

$$\begin{array}{lcll} F & \rightarrow & VC|-_1 F|(F) & \text{(Faktor)} \\ T & \rightarrow & F|T*F|T/F & \text{(Term)} \\ A & \rightarrow & T|A+T|A-T & \text{(Ausdruck)} \end{array}$$

5.1.7 Satz: Grammatik G ist eindeutig

Der Beweis ist zum Beispiel in *Loexx,Mehlhorn,Wilhem: Programmiersprachen* zu finden.

5.2 Aufbohren der Grammatik

5.2.1 Bool'sche Ausdrücke

Beispiel:

$$if \quad \underbrace{2 > x + 3}_{bool'scher Ausdruck} \quad \ldots$$

Um dies darzustellen definieren wir:

$$\begin{array}{lcll} Atom & \rightarrow & A > A|A \geq A|A < A|A \leq A|A == A|A \neq A|0|1 & \text{(wie bool'sche Variable)} \\ BF & \rightarrow & Atom|\ BF|(BA) & \text{(bool'scher Faktor)} \\ BT & \rightarrow & BF|BT \wedge BF & \text{(bool'scher Term)} \\ BA & \rightarrow & BT|BA \vee BT & \text{(bool'scher Ausdruck)} \end{array}$$

5.2.2 Komplexe Datentypen

struct-Komponenten:

$$(\underbrace{\ldots\ldots\ldots\ldots\ldots\ldots\ldots}_{identifier\ des\ Typs\ struct}).Na$$

array-Komponenten:

$$(\underbrace{\ldots\ldots\ldots\ldots\ldots\ldots\ldots}_{identifier\ des\ Typs\ array})[e]$$

pointer-Dereferenzierer:

$$*(\underbrace{\ldots\ldots\ldots\ldots\ldots\ldots\ldots}_{identifier\ des\ Typs\ pointer})$$

adress of-Operator:

$$\&(\underbrace{\ldots\ldots\ldots\ldots\ldots\ldots\ldots}_{identifier\ des\ Typs\ adress\ of})$$

Um dies darzustellen erweitern wir:

$$id \quad \rightarrow \quad Na|C|id.Na|id[A]|*id|\&id$$

5.2.3 Anweisungen

An	$\rightarrow$	$id = A$	(Zuweisung)
		$id = BA$	(Zuweisung für bool'sche Variablen)
		$if\ BA\ then\ An\ else\ An$	(Bedingte Anweisung)
		$if\ BA\ then\ An$	(verkürzte bedignte Anweisung)
		$while\ BA\ do\ An$	(Schleife)
		$\{AnF\}$	(Anweisungs-Block)
		$Na(A-Folge)$	(Funktionsaufruf)
		$return\ A$	(Rückgabewert)
		$return\ BA$	(Rückgabewert für bool'sche Variablen)
$A-Folge$	$\rightarrow$	$A \mid A-Folge,\ A$	(Ausdrucks-Folge)
AnF	$\rightarrow$	$An \mid AnF;\ An$	(Anweisungs-Folge)

An einem Beispiel zeigt sich wieder, dass diese Grammatik nicht eindeutig ist:
1.

$$\underbrace{if\ BA\ then\ \underbrace{if\ BA\ then\ An}_{An}\ else\ An}_{An}$$

2.

$$\underbrace{if\ BA\ then\ \underbrace{if\ BA\ then\ An\ else\ An}_{An}}_{An}$$

5.2.4 Programm

Das Programm besteht aus einem Deklarations-Teil (zur Deklaration von Variablen, Funktionen und Typen) und einem Anwendungsteil (einer Anweisungs-Folge):

$$programm \quad \rightarrow \quad DeklF; AnF \quad \text{(Programm)}$$

5.2.5 Deklarationen

$Dekl$	$\rightarrow$	$VaD \mid FnD \mid TypD$	(Deklaration)
$DeklF$	$\rightarrow$	$DeklF \mid DeklF;\ Dekl$	(Deklarations-Folge)
NaF	$\rightarrow$	$Na \mid NaF,\ Na$	(Namens-Folge)
VaD	$\rightarrow$	$typNaF;$	(Variablen-Deklaration)
Par	$\rightarrow$	$TypNa$	(Parameter)
$ParF$	$\rightarrow$	$Par \mid ParF,\ Par$	(Parameter-Folge)
$TypD$	$\rightarrow$	$typedeftypNa;$	(Typen-Definition)
$eltyp$	$\rightarrow$	$int \mid bool \mid char \mid floag \mid \ldots$	(elementare Typen)
typ	$\rightarrow$	$eltyp \mid eltyp[ZiF] \mid Na[ZiF] \mid structNa\{KompDF\} \mid Na*$	(Typen)
$KompD$	$\rightarrow$	$Na\ :\ eltyp \mid Na\ :\ Na$	(Komponenten-Definition)
$KompDF$	$\rightarrow$	$KompD \mid KompDF$	(Komponenten-Folge)

5.3 Semantik von C_0

Für die Sematik von C_0 definieren wir wie schon bei DLX_0 eine abstrakte Maschine:

- Deklarations-Teil:

  Typen-Definition
  Variablen-Deklaration
  Funktions-Deklaration
  } $c.env(\underbrace{\ldots}_{Typ},\underbrace{\ldots}_{Var},\underbrace{\ldots}_{Fn})$

- Anweisungs-Teil:

  Programmrest
  (noch auszuführende Anweisungen)
  } $c.prog$

5.4 Deklarations-Teil

5.4.1 Typen-Definition

Als Beispiel für die Typen-Definitionen gelten folgende Deklarationen:

```
typedef x int[5];
typedef y x[5];
struct {wert:int,next:LEL*} LEL;
```

5.4.1.1 Type Table

$c.nt$ = Number of Types (Anzahl der Bennutzten Typen)
$c.tt$ = Type Table
= $(\underbrace{c.tt.name}_{Type-Name}, \underbrace{c.tt.tc}_{Type-Code})$

Type Table:
$c.tt.name$: $[0:nt-1] \rightarrow \underbrace{Namen}_{=L(Na)}$
$c.tt.tc$: $[\ ] \rightarrow TypeCodes$

Type Codes:
$c.tt.tc[i]$ = el
falls $name[i]$ elementar
$c.tt.tc[i]$: $arr(n,j)$
typedef (c.tt.name[j])[n] c.tt.name[i]
* i-te Type-Definition (inkl. elem. Typen))
* $j < i$
$c.tt.tc[i]$: $ptr(j)$
typedef (c.tt.name[j])* c.tt.name[i]
* $j > i$ ist erlaubt
$c.tt.tc[i]$: $struct(n_1,i_1;\ldots;n_s:i_s)$
$typedef\ struct(n_1,t_1;\ldots;n_s:t_s) * c.tt.name[i]$
* $t_j = name(i_j)$
* $i_j < i$

Eine Tabelle zu den Beispiel-Deklarationen könnte dann wie folgt aussehen:

i	$name$	tc
0	int	el
1	$bool$	el
$\vdots$	$\vdots$	$\vdots$
4	x	$arr(5,0)$
5	y	$arr(5,4)$
6	p	$ptr(7)$
7	LEL	$strwert,0;next,6$

5.4.1.2 Größe von Typen

- $t \underbrace{elementar\ oder\ pointer-typ}_{simple\ value}$:
  $size(t) \ = \ 1$

- $t \ = \ t'[n]$:
  $size(t) \ = \ n \cdot size(t')$

- $t \ = \ struct\{n_1 : t_1; \ldots ; n_s : t_s\}$:
  $size(t) \ = \ \sum_{i=0}^{s-1} size(t_i)$

Bemerktung (in 'Sprache' der Type Table):

- falls $c.tt.tc[i] \ = \ arr(n,j)$:
  $size(i) \ = \ n \cdot size(j)$

- falls $c.tt.tc[i] \ = \ struct\{n_1 : t_1; \ldots ; n_s : t_s\}$:
  $size(i) \ = \ \sum_{j=0}^{s-1} size(i_j)$

- $size(i)$ kann aus $c.tt$ in Reihenfolge $i = 0, 1, \ldots , c.nt - 1$ berechnet werden
  $size(t) \ = \ n \cdot size(t')$

5.4.1.3 Range

Festzulegen $\forall$ deklarierten Typen t:

$$\underbrace{Ra(t)}_{Range} = \{w | w \text{ Wert, Variablen vom Typ } t \text{ können } w \text{ als Wert annehmen } \}$$

t elementar:

$$\begin{aligned} Ra(int) &\neq \mathbb{Z} \\ Ra(int) &= \{-2^{31}, \ldots, 2^{31}-1\} \\ Ra(char) &= \{0, \ldots, 2^8 - 1\} \\ &\vdots \end{aligned}$$

Notation:
$f_s(i) = f[i+s-1 : i]$[3]

$$\begin{aligned} & Ra(t[n]) \ni f[\underbrace{size(t[n]) - 1}_{n \cdot size(t)} : 0] \\ \Leftrightarrow \quad & \forall i \in \{0, \ldots, n-1\} : f_{size(t)}(i.size(t)) \in Ra(t) \end{aligned}$$

$$\begin{aligned} & Ra(struct\{n_1 : t_1; \ldots; n_s : t_s\}) \ni f \\ \Leftrightarrow \quad & \forall i \in \{1, \ldots, s\} : f_{size(t_i)}(d_i) \in Ra(t_i) \end{aligned}$$

Achtung: Wir stellen Werte von Variablen mit (komplexen) Typ t als Folgen von simple values $\in sv$ [4] der Länge $size(t)$ dar.

[3] Teilfolge der Länge s beginnend bei Element i

[4] sv: Werte die simple Typen annehmen können (Pointer später!)

Typ $int[5]\ x$:

$\in R(int)$	$\in R(int)$	$\in R(int)$	$\in R(int)$	$\in R(int)$	$\in R(x)$

Typ $x[5]\ y$:

$\in R(x)$	$\in R(x)$	$\in R(x)$	$\in R(x)$	$\in R(x)$	$\in R(y)$

Typ $struct\{n_1 : t_1; \ldots; n_i : t_i; \ldots; n_s : t_s\}\ z$:

$size(t) - 1$					
$\in R(t_s)$	$\ldots$	$\in R(t_i)$	$\ldots$	$\in R(t_1)$	$\in R(t)$
		d_i*		$size(t_1)$	0

**Displacement*: $d_i = \sum_{j<i} size(t_j)$

Bemerkung:
Werte sind (bei uns) flach (wegen dem Adress-Of Operator &)

5.4.2 Variablen-Deklaration

- mit Namen ausserhalb von Funktionen (global)
- mit Namen innerhalb von Funktionen (lokal)
- *neu:* ohne Namen, Zugriff nur über Pointer

Änderung der Grammatik (!):

$$An \quad \rightarrow \quad Na = newNa* \quad \text{(Allokieren von Speicher)}$$

Beispiel:

```
LEL* q;
q = new LEL*;
```

5.4.2.1 Symbol-Tabelle von Speichern

Erweiterung der Konfiguration:

$$c = (nt, tt, gm, hm, lms, \ldots)$$

gm	=	global memory
hm	=	heap memory
lms	=	local memory stack

$$\text{memories } m = (\underbrace{m.n,\ m.name,\ m.typ}_{st(m)},\ m.ct)$$

$m.n$	:	number (Anz. Variablen) $\in \mathbb{N}_0$
$m.name$	:	$[0 : m.nv-1] \Rightarrow$ Namen (Variablenname)
$m.typ$	:	$[0 : m.nv-1] \Rightarrow \underbrace{[0 : c.nt-1]}_{Typ-Nr.\ aus\ c.tt}$
$\underbrace{m.ct}_{Wert}$	:	$\underbrace{[0 : \sum_{i=0}^{m\cdot n-1} size(m.typ(i)) - 1]}_{Anz.\ simple\ values} \Rightarrow sv$
$size(m)$	=	$\sum_{i=0}^{c.nv-1} size(m.typ[i])$
$st(gm)$	:	durch globale $VarDF$ festgelegt

Eine Tabelle zu den Beispiel-Deklarationen könnte dann wie folgt aussehen (Typen-Nummern entsprechen wieder dem Beispiel aus der Type Table 5.4.1.1):

i	$name$	tc	ct
0	i	0	} 1
⋮	⋮	⋮	
	u	4	} 5
⋮	⋮	⋮	
	v	5	} 25
⋮	⋮	⋮	

5.4.2.2 Definition: Variable (≠Identifier)

V wird spezifiziert durch ein paar $(m,\ i)$:

$$\begin{array}{rcl} m & : & memory \\ i & \in & \{0, \ldots, m.nv-1\} \end{array}$$

5.4.2.3 Definition: Wert von Variable (m, i)

$$\begin{aligned} \underbrace{va(m,i)}_{Value} &= m.ct_{size(m.typ[i])}\ \underbrace{ba(m,i)}_{Basis-Adr.} \\ ba(m,i) &= \sum_{j<i} size(m.typ[j]) \end{aligned}$$

Problem:

```
v[1][2] = 7;
```

Ändert 3 *g-Variablen*:

- Matrix v
- Vektor $v[1]$
- einzelnes Element $v[1][2]$

5.4.2.4 Definition: g(eneralized)-Variablen

- alle Variablen sind g-Variablen
- $(m, i)s$[5]
  - $typ(m, i)s \ = \ t[n]$
    $\Rightarrow (m, i)s[j]$ ist g-Variable
    $typ\,((m, i)\,s[j]) \ = \ t$
    (für $0 \leq j \leq n - 1$)
  - $typ((m, i)s \ = \ structn_1 : t_1; \ldots; n_s : t_s$
    $\Rightarrow (m, i)s.n_j$ ist g-Variable
    $typ((m, i)s.n_j) \ = \ t_i$
    (für $1 \leq j \leq i$)

Diesen g-Variablen können auch Basis-Adressen und Werte zugeordnet werden:

- $typ(m, i)s \ = \ t[n]$
  $ba((m, i)s[j]) \ = \ ba((m, i)s) + j \cdot size(t)$
- $typ((m, i)s \ = \ structn_1 : t_1; \ldots; n_s : t_s$
  $ba((m, i)s.n_j) \ = \ ba((m, i)s) + \sum_{k<j} size(t_k)$
- $va((m, i)s) \ = \ m.ct_{size(typ((m,i)s))}(ba((m, i)s))$

5.4.2.5 Definition: Werte von Pointern

$$\begin{aligned} pointer &= (m : a) \\ m &: memory \\ a &\in \{0,\ m.nv - 1\} \end{aligned}$$

[5] s: selector

5.4.2.6 Korrektur: Grammatik

$$FuD \quad \rightarrow \quad Typ \underbrace{Na}_{f} \underbrace{\underbrace{(ParF)}_{Parameter} \underbrace{VarDF}_{lok.\ Var.}}_{st(lm_f)}; rumpf \qquad \text{(Funktions-Deklaration)}$$

$$rumpf \quad \rightarrow \quad An;\ return(A) \qquad \text{(Funktions-Rumpf)}$$

lm_f ist der local memory von f (Variablen sind die Parameter und die lokalen Variablen)

Ausserdem wird in der Grammatik *return* aus den Anweisungen entfernt.

Neu:

- Deklaration lokaler Variablen (vorher vergessen)
- Nur eine return-Anweisung am Ende des Rumpfs

5.4.3 Funktions-Deklaration

Erweiterung der Konfiguration:

$$c \ = \ (nt, tt, gm, hm, lms, nf, ft, \ldots)$$

$c.nf$: $[0 : c.nf - 1] \rightarrow \{fd | fd : function\ descriptor\}$
(number of functions $\in \mathbb{N}_0$)
$c.ft$ = function table

fd = $(fd.name,\ fd.typ\ , fd.st\ fd.rumpf)$
$fd.name$: Name d. dekl. Funktion
$fd.typ$: Typ d. Funktion
$fd.st$: Symbol-Table d. Funktion

$fd.typ$ = $(t_0, \ldots, t_n - 1, t_n)$
t_i $\in$ $\underbrace{[0 : c.nt - 1]}_{Typen\ d.\ TT}$
$0 \leq i \leq n-1$: Typ des i-ten Parameters
$i = n$: Typ des return-Werts

$c.nf$: $[0 : c.nf - 1] \rightarrow \{fd | fd : function\ descriptor\}$
(number of functions $\in \mathbb{N}_0$)
$c.ft$ = function table

$$
\begin{array}{rcl}
lm_f.nv & = & \#Parameter + \#\, lokalerVariablen \\
lm_f.name[i] & = & \begin{cases} \text{aus Fn.Def(ParDF)} & : & i < \#Parameter \\ \text{aus VarDF für lok. Var.} & : & sonst \end{cases} \\
lm_f.typ[i] & = & \begin{cases} \text{aus Fn.Def(ParDF)} & : & i < \#Parameter \\ \text{aus VarDF für lok. Var.} & : & sonst \end{cases}
\end{array}
$$

$\Rightarrow$ $size(lm_f)$ dadurch festgelegt

5.4.3.1 Rekursion, Rekursionstiefe

Erweiterung der Konfiguration:

$$c = (nt, tt, gm, hm, lms, nf, ft, rd, lms, rds, \ldots)$$

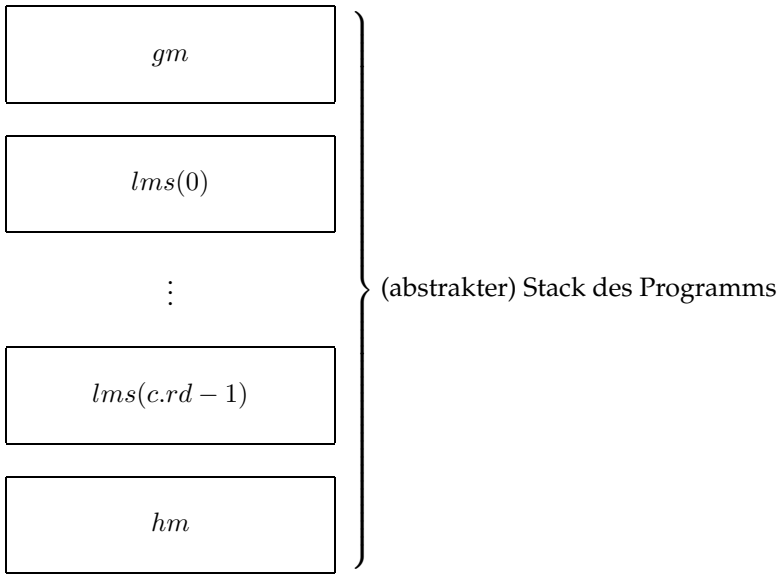
$c.rd$	:	$\#call's$ die noch nicht zurückgekehrt sind
$c.lms$	:	$[0 : c.rd - 1] \to \{m \mid m$ ist lok. mem. lm_f für ein $f\}$ (local memory stack)
$c.rds$	:	return destinations
$c.rds[i]$	:	$(m : a)\ m \in gm, lm(i)$ (Basisadresse, an dem return-Wert von $rd = i$ gespeichert wird

5.4.3.2 Heap Memory

$$c = (nt, tt, gm, hm, lms, nf, ft, rd, lms, rds, hm)$$

$c.hm$	:	heap memory speichert Variablen (und g-Variablen) ohne Namen

5.4.4 Speicher der C_0-Maschine



5.5 Zusammenfassung der Grammatik

$< Zi >$	$\rightarrow$	$0\|1\|2\|3\|4\|5\|6\|7\|8\|9$	(Ziffer)
$< ZiF >$	$\rightarrow$	$< Zi > \| < ZiF >< Zi >$	(Ziffernfolge)
$< Bu >$	$\rightarrow$	$a\|b\|c\|\ldots\|x\|y\|z$	(Buchstabe)
$< BuF >$	$\rightarrow$	$< Bu > \| < BuF >< Bu >$	(Buchstabenfolge)
$< BuZi >$	$\rightarrow$	$< Bu > \| < Zi >$	(Buchstabe/Ziffer)
$< BuZiF >$	$\rightarrow$	$< BuZi > \| < BuZiF >< BuZi >$	(Buchstaben-/Ziffernfolge)
$< Na >$	$\rightarrow$	$< Bu > \| < Bu >< BuZiF >$	(Name)
$< NaF >$	$\rightarrow$	$< Na \| < NaF >, < Na >$	(Namens-Folge)
$< C >$	$\rightarrow$	$< ZiF >$	(Konstante)
$< id >$	$\rightarrow$	$< Na > \| < C > \| < id > . < Na > \|$	
$< id >$	$\rightarrow$	$< id > [< A >] \| * < id > \| \& < id >$	(Identifier - früher Variable oder Konstante)
$< F >$	$\rightarrow$	$< VC > \|-_1 < F > \|(< F >)$	(Faktor)
$< T >$	$\rightarrow$	$< F > \| < T > * < F > \| < T > / < F >$	(Term)
$< A >$	$\rightarrow$	$< T > \| < A > + < T > \| < A > - < T >$	(Ausdruck)
$< A - Folge >$	$\rightarrow$	$< A > \| < A - Folge >, < A >$	(Ausdrucks-Folge)
$< Atom >$	$\rightarrow$	$< A > > < A > \| < A > \geq < A > \|$	
		$< A > < < A > \| < A > \leq < A > \|$	
		$< A > == < A > \| < A > \neq < A > \|$	
		$0 \mid 1$	(wie bool'sche Variable)
$< BF >$	$\rightarrow$	$< Atom > \| < BF > \|(< BA >)$	(bool'scher Faktor)
$< BT >$	$\rightarrow$	$< BF > \| < BT > \wedge < BF >$	(bool'scher Term)
$< BA$	$\rightarrow$	$< BT > \| < BA > \vee < BT >$	(bool'scher Ausdruck)
$< An >$	$\rightarrow$	$< id >=< A >$	(Zuweisung)
		$< id >=< BA >$	(Zuweisung für bool'sche Variablen)
		$if\ < BA >\ then\ < An >\ else\ < An >$	(Bedingte Anweisung)
		$if\ < BA >\ then\ < An >$	(verkürzte bedignte Anweisung)
		$while\ < BA >\ do\ < An >$	(Schleife)
		$\{< AnF >\}$	(Anweisungs-Block)
		$< Na > (< A - Folge >)$	(Funktionsaufruf)
		$< id >=< Na > (< A - Folge >)$	(Funktionsaufruf mit Rückgabewert)
		$< Na >=\ new < Na > *$	(Allokieren von Speicher)
$< AnF >$	$\rightarrow$	$< An > \| < AnF >; < An >$	(Anweisungs-Folge)
$< Dekl >$	$\rightarrow$	$< VaD > \| < FnD > \| < TypD >$	(Deklaration)
$< DeklF >$	$\rightarrow$	$< DeklF > \| < DeklF >; < Dekl >$	(Deklarations-Folge)
$< VaD >$	$\rightarrow$	$< typ >< NaF >;$	(Variablen-Deklaration)
$< Par >$	$\rightarrow$	$< Typ >< Na >$	(Parameter)
$< ParF >$	$\rightarrow$	$< Par > \| < ParF >, < Par >$	(Parameter-Folge)
$< TypD >$	$\rightarrow$	$typedef < typ >< Na >;$	(Typen-Definition)
$< FuD >$	$\rightarrow$	$< Typ > < Na > (< ParF >)\ < VarDF >;$	
		$< rumpf >$	(Funktions-Deklaration)
$< rumpf >$	$\rightarrow$	$< An >; return(< A >)$	(Funktions-Rumpf)
$< eltyp >$	$\rightarrow$	$int\|bool\|char\|floag\|\ldots$	(elementare Typen)
$< typ >$	$\rightarrow$	$< eltyp > \| < eltyp > [< ZiF >] \|$	
		$< Na > [< ZiF >] \|$	
		$struct\{< KompDF >\} < Na > \| < Na > *$	(Typen)
$< KompD >$	$\rightarrow$	$< Na > : < eltyp > \| < Na > : < Na >$	(Komponenten-Definition)
$< KompDF >$	$\rightarrow$	$< KompD > \| < KompDF >$	(Komponenten-Folge)
$< programm >$	$\rightarrow$	$< DeklF >; < AnF >$	(Programm)

5.6 Ausdrucksauswertung

3 Funktionen:

- addr (Adresse eines Ausdrucks bestimmen):
  $c,\ Ausdruck\ \rightarrow\ (m : a)$

- typ (Typ eines Ausdrucks bestimmen):
  $c,\ Ausdruck\ \rightarrow\ [0 : nt - 1]$

- va (Wert eines Ausdrucks bestimmen):
  $c,\ Ausdruck\ \rightarrow\ Ra(typ)$

5.6.1 Definition: Bindungsfunktion($c, Na\ \rightarrow\ (m, i)$)

$$bind(c, n)\ =\ \begin{cases} (c.lms[c.rd - 1],\ i) : & \text{falls } \exists i \text{ mit} \\ & c.lms[c.rd - 1].st.name[i]\ =\ n \\ (gm,\ i) : & \text{falls } \exists i \text{ mit} \\ & c.gm.st.name[i]\ =\ n \end{cases}$$

5.6.2 Auswertung

1. **Konstante**: $e = < C >$
   $addr(c,e) = undefiniert$
   $typ(c,e) = c.env.tt[x] \quad (x \mathrel{\widehat{=}} int)$
   $va(c,e) = < e >_{10}$

2. **Variable**: $e = < Na >$
   Sei $bind(c, < Na >) = (m : i)$

   $addr(c,e) = (m : ba(m,i))$
   $typ(c,e) = (m,i)$
   $va(c,e) = va(m,i)$

3. **Arrayzugriff**: $e = < id > [< A >]$
   $addr(c,e) = (m : (a + j \cdot size(t)))$
   $typ(c,e) = t$
   $va(c,e) = va(< id >)_{size(t)}(j \cdot size(t))$

4. **Strukturen**: $e = < id > . < Na >$
   Sei $addr(c, < id >) = (m : a)$
   $typ(c, < id >) = struct\{n_1 : t_1, \ldots, n_s : t_s\}$
   $< Na > = n_i$ (Wir greifen auf die i-te Komponente zu

   $addr(c,e) = (m : (a + \sum_{j=1}^{i-1} size(t_j)))$
   $typ(c,e) = t_i$
   $va(c,e) = va(< id >)_{size(t)}(\sum_{j=1}^{i-1} size(t_j))$

5. **Dereferenzierung**: $e = * < id >$
   Sei $va(c, < id >)[0] = (m : a)$
   $typ(c, < id >) = ptr(t)$

   $addr(c,e) = (m : a)$
   $typ(c,e) = t$
   $va(c,e) = m.ct_{size(t)}(a)$

6. **adress-of Operator**: $e = \& < id >$
   Sei $addr(c, < id >) = (m : a)$
   $typ(c, < id >) = t$

   $addr(c,e) = undefiniert$
   $typ(c,e) = ptr(t)$
   $va(c,e) = (m : a)$

7. **arithmetische/boole'sche Ausdrücke**: $e = < A > + < T >$
   $addr(c,e) = undefiniert$
   $typ(c,e) = typ(c, < A >)$ (Achtung bei bool'schen Vergleichsoperatoren!)
   $va(c,e) = va(c, < A <)[0] + va(c, < T >)[0] \ mod \ 2^{32}$

5.7 Ausführung von Anweisungen

Definition: $next - state$ Funktion ($delta: Konf \to Konf$)
Berechnet Nachfolgekonfigurationen, $delta$ führt die erste Anweisung vom Programmrest aus:
Fallunterscheidung nach der ersten Anweisung in $c.prog$, r sei der Rest des Programms:

1. **Zuweisung**: $c.prog = < id >=< A >;\ r$ / $c.prog = < id >=< BA >;\ r$

$$\begin{array}{rcl} \text{Sei} \quad typ(c, < id >) & = & typ(c, < A >) \\ & = & t \in sv \\ & \Rightarrow & size(t) = 1 \\ addr(c, < id >) & = & (m : a) \end{array}$$

$$\begin{array}{rcl} delta(c).m.ct[i] & = & \begin{cases} va(c, < A >)[0]: & \text{falls } i = a \\ c.m.ct[i]: & \text{falls } i \neq a \end{cases} \\ delta(c).prog & = & r \end{array}$$

2. **if-Anweisung**: $c.prog = if\ < BA > \ then\ < An_1 > \ else\ < An_2 >;\ r$

$$delta(c).prog = \begin{cases} < An_1 >;\ r & \text{falls } va(c, < BA >) = 1 \\ < An_2 >;\ r & \text{falls } va(c, < BA >) = 0 \end{cases}$$

3. **while-Schleife**: $c.prog = while < BA > \ do\ < An >;\ r$

$$delta(c).prog = \begin{cases} < An >;\ while < BA > \ do\ < An >;\ r & \text{falls } va(c, < BA >) = 1 \\ r & \text{falls } va(c, < BA >) = 0 \end{cases}$$

4. **Speicherallokierung**: $c.prog = < Na >= new * < Typ >;\ r$

$$\begin{array}{rcl} \text{Sei} \quad t & : & \text{Typnummer von } < Typ > \\ typ(c, < Na >) & = & ptr(t) \\ addr(c, < Na >) & = & (m : a) \end{array}$$

$$\begin{array}{rcl} delta(c).hm.n & = & c.hm.n + 1 \\ delta(c).hm.typ[c.hm.n] & = & t \\ delta(c).m.ct[i] & = & \begin{cases} (c.hm : size(c.hm)) & \text{falls } i = a \\ c.m.ct & \text{falls } i \neq a \end{cases} \\ delta(c).prog & = & r \end{array}$$

5. **Funktionsaufrufe**: $c.prog = < id >=< Na > (< AF >);\ r$

$$\begin{array}{rcl} \text{Sei} \quad p_i & : & i\text{-ter Parameter des Funktionsaufrufs} \\ addr(c, < id >) & = & (m : a) \\ c.ft[j].name & = & < Na > \\ typ(c, p_i) & = & c.ft[j].typ[i] \\ typ(c, < id >) & = & \underbrace{c.ft[j].typ[n]}_{Anz.\ d.\ Param.} \end{array}$$

$$\begin{array}{rcl} delta(c).rd & = & c.rd + 1 \\ delta(c).lms[c.rd].st & = & c.ft[j].st \end{array}$$

$\forall\ k < n:$

$$\begin{array}{rcl} delta(c).lms[c.rd]_{size(typ(p_k))}(\underbrace{ba(c.lms[c.rd], k)}_{(m,i)}) & = & va(c, p_k) \\ delta(c).prog & = & c.ft[j].body;\ r \\ & & (j\text{-te Fn. der Funktions-Tabelle}) \\ delta(c).rds[c.rd - 1] & = & (m : a) \end{array}$$

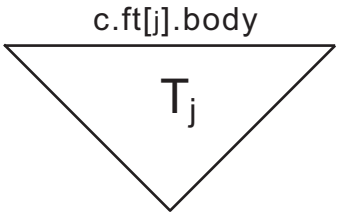
5.8 Compilieren

C_0-Maschine:
Konfiguration: $c = (nt, tt, gm, hm, lms, nf, ft, rd, lms, rds, hm, prog)$
insbesondere: $c.prog$ Programmrest ($< AnF >$)

Startkonfiguration von C_0:
$c.prog = c.ft[0].body$

Funktionen:
Anzahl: $c.nf$
spezifiziert in: $c.ft[0 : c.nf - 1]$
body's: $c.ft[j].body \quad j = 0, \ldots, c.nf - 1$

Ableitungsbäume:



„Wald": $T_0, \ldots, T_{c.nf-1} = \vec{T}$ von Ableitungsbäumen.

5.8.1 Compiler

Der Compiler erzeugt aus $\vec{T}$ ein DLX_0-Programm: $code(\vec{T})$:

Vorgehensweise (vor dem Compilieren):

1. Erzeugen der Symboltabellen für die Funktionen (aus Deklarations-Teil und Parameter-Listen)

2. Erzeugen der Typtable (aus dem Typ-Deklarations-Teil)

3. Erzeugen der Funktionstabelle (aus dem Funktions-Deklarations-Teil)

Vorgehensweise (Compilieren in 2 Phasen):

1. Bäume T_j getrennt übersetzen $\rightarrow$ $code(T_j)$
   allerdings sind einige Sprungweiten noch offen:

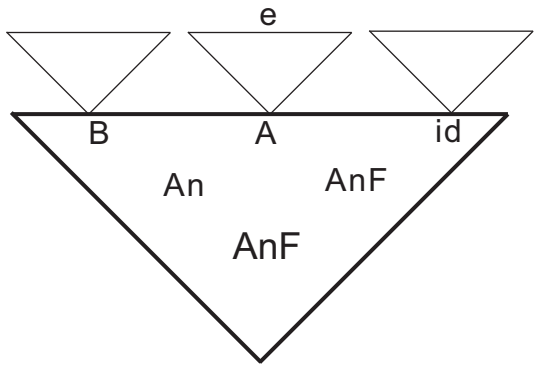
$code(T_0)$	$\ldots$	$code(T_i)$	$\ldots$	$code(T_j)$	$\ldots$	$code(T_{c.nf-1})$

   Sei in $code(T_i)$ ein Aufruf von $c.ft[j]$:
   Rücksprung: geht mit jal
   Vorsprung: es fehlt im Allgemeinen die Distanz (wenn die Funktion nach der gerade übersetzten Funktion

2. nicht aufgelöste Distanzen einsetzen

5.8.2 Übersetzen der Bäume

Ebenfalls in 2 Phasen:



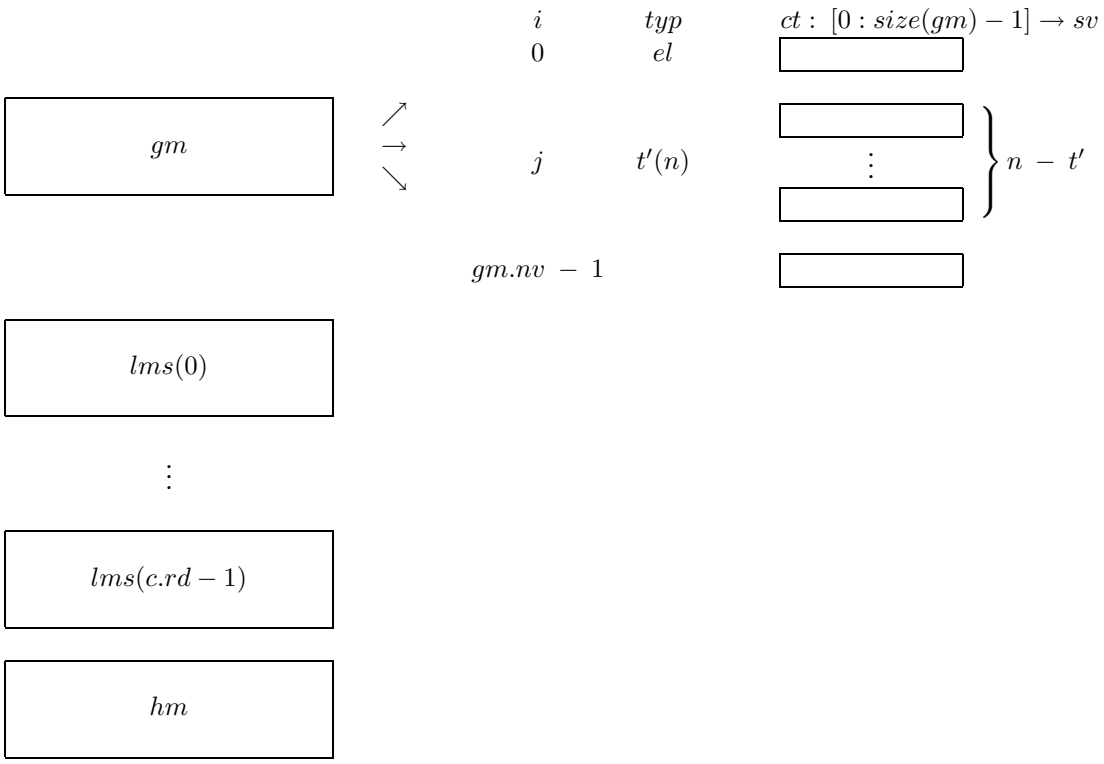
1. $code(e)$ erzeugen, für e:
   Ausdruck A
   Bedingung B
   Identifier id

2. $code(q)$ erzeugen, für alle q:
   An, AnF

Ausdrücke:
Beispiel: $e = x$
x: Name (einer Variable)
steht irgendwo im Speicher $d.m$ einer DLX_0-Maschine
in der Konfiguration $d = (d.PC, d.GPR, d.m)$

Ansatz:
Wir codieren (auf ziemlich einfache Weise) die memories $c.gm$, $c.lms[i]$ und $c.hm$ im Speicher der DLX_0-Maschine.

C_0-Maschine:



- $size(gm) = \sum_{i=0}^{gm.nv\ -\ 1} size(g.typ[i])$
- Der Compiler kann $size(gm)$ berechnent
- Der Compiler kann $size(lm_f)$ (für alle deklarierten Funktionen) berechnen

5.9 Codierung von C_0-Konfigurationen in DLX_0-Konfigurationen

Ausdrücke:

$$\underbrace{F_1 \;=\; c.ft[i]}_{i-te\ deklarierte\ Fn.}$$

Funktions-Stack:

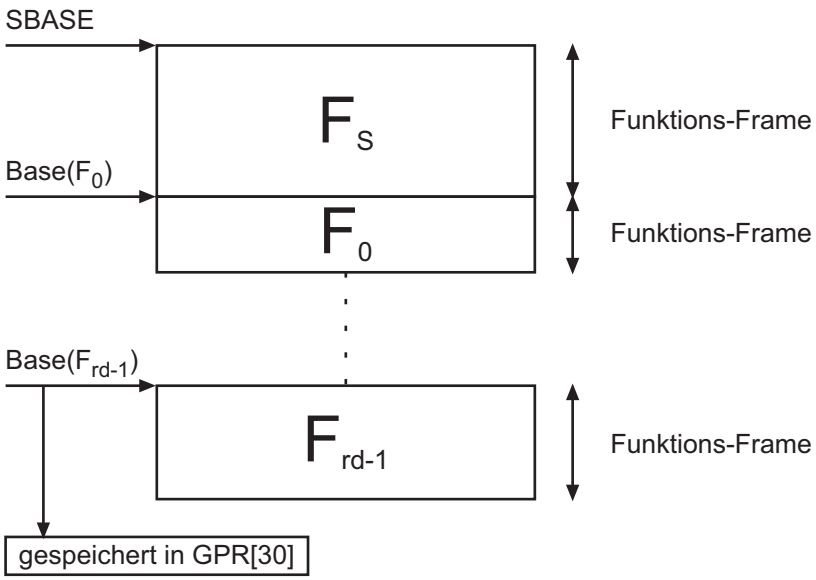


Abbildung 44: Funktions-Stack

F_i codiert $lms[i].ct$ und 3 Wörter mit Zusatzinformationen:

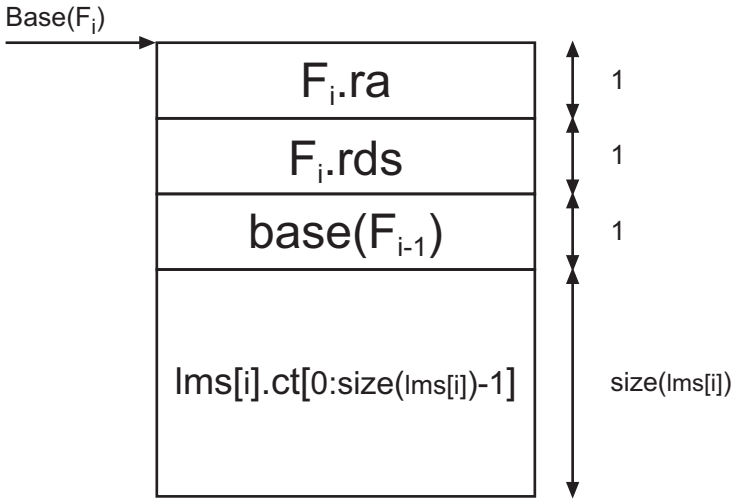


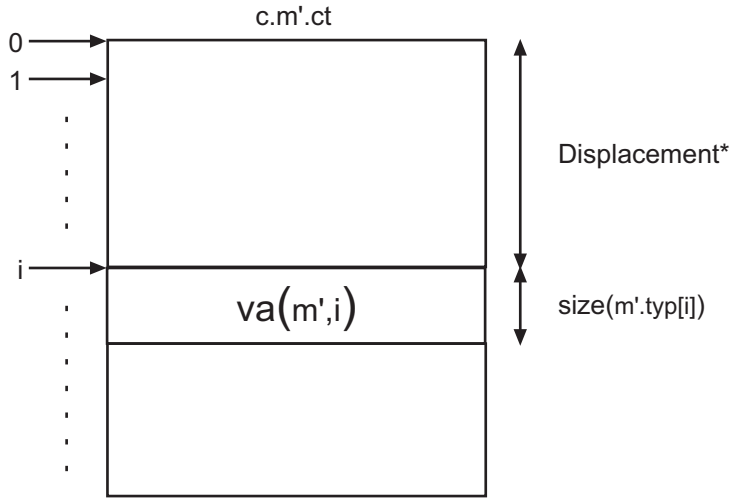
Abbildung 45: Funktions-Frame

$Base(F_i)$	Basisadresse von Frame i im DLX_0-Speicher $d.m$ (i.A. erst zur Laufzeit bekannt)
$F_i.ra$	Rücksprungadresse (beim Aufruf mit jal berechnet und hier gespeichert)
$F_i.rds$	Ziel für Rückgabewert (hier wird das Ergebnis der Funktion gespeichert)
$Base(F_{i-1})$	Zeigt auf F_{i-1} (wird bei Rücksprung aus F_i in $GPR[30]$ geschrieben)

Wenn es gelingt diese Planung während der Laufzeit des übersetzten Programms einzuhalten (Induktionsbewes über die Laufzeit), dann kann man zu Laufzeit auf Werte von Variablen im gm (relativ zu $SBASE$) und im $lms[rd-1]$ zugreifen (relativ zu $GPR[30]$)
Detail:

$(m', i) = x$ Variable der C_0-Maschine
$m' \in gm, \underbrace{lms[rd-1]}_{ltop}$

Wo steht der Wert von $va(c, x)$ in $c.m'.ct$?



**Displacement:*

$$\begin{aligned} displ(i,\ m') &= \sum_{j<i} size(m'.typ[j]) \\ &= displ(i,\ f) \end{aligned}$$

Sei m' = $lms[a]$ zu Funktion f
m'' = $lms[b]$ zu Funktion f
$\Rightarrow$ $displ(i, m')$ = $dipl(i, m'')$

Das Displacement $displ(i, f)$ (zur Compile-Zeit bekannt!) hängt nur davon ab, zu welcher Funktion f das $lms[a]$ gehört.

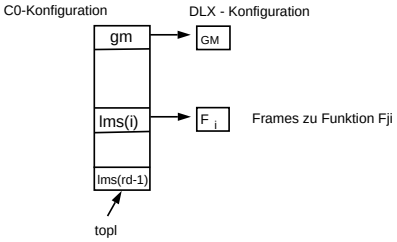
Sei
y : Name
m : memory zu f
$adr(c, y)$ = $(m : \underbrace{ba(m, i)}_{displ(i,f)})$
$typ(c, y)$: simple
$va(c, y)$: va(m,i)

} C_0-Semantik

DLX_0-Code lädt $va(m, i) = va(c, y)$ in $GPR[u]$
lokal (aus rd) $GPR[u] = m[GPR[30] + displ(i, f) + 3]$
global (aus gm) $GPR[u] = m[SBASE + displ(i, f)\,]$

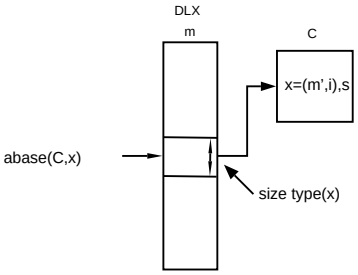
5.9.1 Definition: Compiler - Korrektheit

Der Compiler ist korrekt falls:
$\forall$ Rechnungen $C^1, C^2, ...,$ mit Programm P
$\forall$ Rechnungen $d^1, d^2, ..., d^{s(i)}, ...,$ mit $code(P)$
$\exists S : \mathbb{N} \to \mathbb{N}$ Schrittzahlen : $\forall i : d^{s(i)}$ kodiert C^i.



C_0 - Konfiguration

x=(m,i) Variable von C $\longrightarrow$ Wo steht $va(C, x)$ in $d.m$
x=(m,i) g - Variable von C



Gewünscht: $\forall x : typ(x)$ - elementar:
$va(C, x) = d.m(abase(C, x))$ e-Konsistenz

$e - consis(C, abase, d)$

5.9.2 Definition: abase

Frames

$$\begin{aligned} base(C, GM) &= sbase \\ base(C, F_0) &= sbase + size(GM) \\ base(C, F_{i+1}) &= base(C, F_i) + \underbrace{size(F_i)}_{3+size(f_{ji})} \end{aligned}$$

Bemerkung: f_{ji} erst zum Compilerzeit bekannt.

Variablen (m', j)
$abase(C, (m', j)) = base(C, \widetilde{m}) + \underbrace{displ(j, m')}_{base(j,m')} + 3$

$$\widetilde{m} = \begin{cases} F_i & m' = lms(i) \\ GM & m' = gm \end{cases}$$

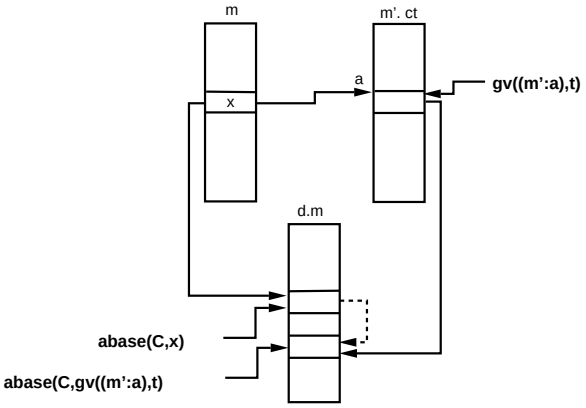
Namen n
$abase(C,n) = abase(C, \underbrace{bind(C,n)}_{\text{Variable}})$

$$bind(C,n) = \begin{cases} (C.topl, i) & (1) \quad C.topl.name(i) = n \\ (C.gm, i) & \sim (1) \wedge gm.name(i) = n \end{cases}$$

g-Variable und Identifier
$e = (m,i)s$

$$\begin{aligned} abase(C, e[i]) &= abase(C,e) + i.size(typ(C,e)) \\ abase(C, e.n) &= abase(C,e) + \sum_{j<i}(typ(C,t_j)) \\ abase(C, C.e^*) & \quad \text{später bei } code(u = new \quad t^*) \text{ definieren .} \end{aligned}$$

$typ(C,x) = t^*$
$va(C,x) = (m' : a)$



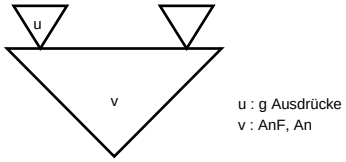
$(m' : a)$ Basisadrese einer g-Variable $(gv((m : a), t))$

d - Speicher
$abase(C,x)$
$abase(C, gv((m' : a), t))$
$(m' : a) = va(C,x)$

$p - consis(C, abase, d) : \forall x : typ(C,x) = t* : d.m(abase(C,x)) = abase(C, gv(va(C,x), t))$

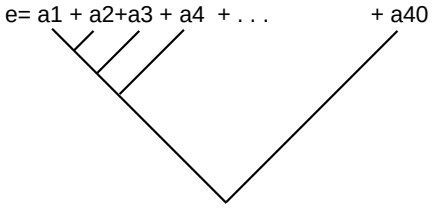
d kodiert C
$consis(C, abase, d) \leftrightarrow$
$e - consis(\quad \cdots \quad) \cdots \wedge$
$p - consis(\quad \cdots \quad) \cdots \wedge$
$r \cdots\cdots \wedge$
$c \cdots\cdots$
Notation: g - Ausdruck : A, B, id

Ableitungsbaum von f, Funktion in P



- g - Ausdrücke compilieren.
- Auswertung: entlang den Knoten des Baums, von Blättern zur Wurzel.

5.9.3 Beispiel



- Wahl bei Reihenfolge
- beeinflüßt die Anzahl der benötigten Register.

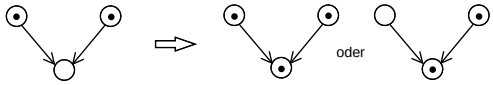
5.9.4 Exkurs: Aho-Ullmann-Algorithmus

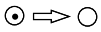
Spiel auf DAG's (Directed Acyclic Graphs)
hier: Bäume

Spielzüge:
Marken (engl. pebbles) auf Knoten setzen oder entfernen.

Regeln:

1. u Quelle

2. 

3. 

Ziel:

Jede Senke hat irgendwann eine Marke. Minimiere die Anzahl der Marken gleichzeitig auf dem Graphen.

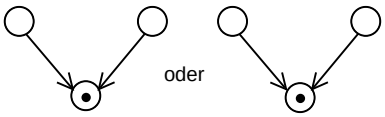
$$\tau(n) = \min\{ \quad x| \text{ jeder Baum mit n inneren Knoten und mit } indegree = 2 \text{ kann mit x Marken markiert werden}\}$$

5.9.5 Satz: Ahu-Ullmann

$\tau(n) \leq \lceil \log n \rceil + 2$

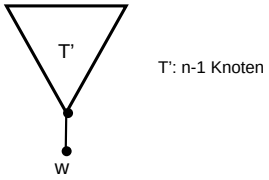
Beweis:

Induktion über n.
I.A $n = 1$



I.S $n - 1 \rightarrow n$
T Baum mit n Knoten, w Wurzel.

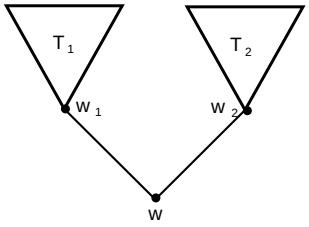
Fall 1:



- Markiere T'
- Nur die Marke auf w′ halten
- Markiere w

$$\begin{aligned}\tau(n) &\leq \max\{\tau(n-1), 2\} \\ &\leq \max\{\lceil\log(n-1)\rceil + 2, 2\} \\ &\leq \lceil\log(n-1)\rceil + 2 \\ &\leq \lceil\log n\rceil + 2\end{aligned}$$

Fall 2



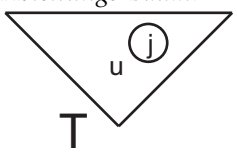
OBdA : $2 \leq T_2 \leq \frac{n}{2}$

- Markiere T_1
- Lasse Marke auf w_1
- Markiere T_2
- Lasse Marke auf w_2
- Markiere w

$$\begin{aligned}\tau(n) &\leq \max\{\tau(n-1), 1 + \tau(\frac{n}{2}), 2\} \\ &\leq \max\{\lceil\log(n-1)\rceil + 2, 1 + \lceil\log(\frac{n}{2})\rceil + 2, 2\} \\ &\leq \max\{\lceil\log(n-1)\rceil + 2, \lceil\log n\rceil - 1 + 1 + 2, 2\} \\ &\leq \lceil logn\rceil\end{aligned}$$

5.9.6 g-Ausdrucksübersetzung

Ableitungs-Baum:



Es gelte:

$$e - consis(c,\ abase,\ d)$$
$$p - consis(c,\ abase,\ d)$$
$$(r - consis(\ldots))$$

$\forall\ i$: Aus Move m_i generiere $ecode(i)$, so dass:

- starte DLX_0 in Konfiguration d
- sei $d(i)$ Konfiguration, die nach Abarbeitung von $E(i) = code(e_1) \ldots code(c_i)$ erreicht wird
- $d \rightarrow^*_{E(i)} d(i)$

$\forall\ j$ (Marken), $\forall\ u$ (Knoten, g-Ausdrücke in T), so dass nach Zug m_i Knoten u Marke j hat.
Induktionsbehauptung:

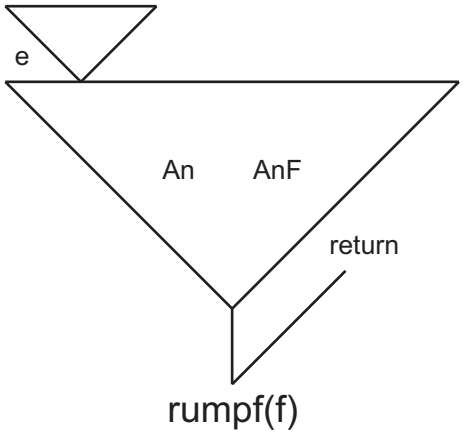
$$d(i).GPR[j] = \begin{cases} va(c,u) & : R(u) = 1,\ typ(c,m) \neq ptr(t) \\ abase(c,u) & : R(u) = 0 \\ abase(c, gv(va(c,u),t) & : R(u) = 1,\ typ(c,m) = ptr(t) \end{cases}$$

5.9.6.1 Annotation des Ableitungs-Baums T mit Prädikat R

Left: $\underbrace{x}_{abase}$ = *Right:* $\underbrace{x + z + 2}_{va}$

$R(u) = 1$ = berechne $va(\ldots)$
$R(u) = 0$ = berechne $abase(\ldots)$

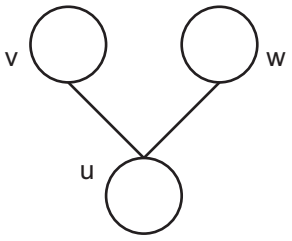
Definiere rekursiv von der Wurzel zu den Blättern:



Vorkommen von g-Ausdrücken:

R(e)	Typ	
0	Zuweisung (links)	$e = \ldots$
1	Zuweisung (rechts)	$\ldots = e$
1	Parameter	$f(\ldots, e, \ldots)$
0	Funktionsaufruf (links)	$e = f(\ldots)$
1	Bedingung	$if\ e\ \ldots$
		$while\ e\ \ldots$

Induktionsschritt: Wurzel → Blätter:



$u = v \circ w$	$\circ \in +, -_2, \cdot, /, \wedge, \vee$
	$R(v) = R(w) = 1$
$u = v \circ w$	$\circ \in -_1, \sim$
$u = v[w]$	$R(w) = 1$
	$R(v) = 0$
$u = v.n$	$R(v) = 0$
	$R(v) = R(u)\ (?)$
$u = \&v$	$R(v) = 0$

$$ecode(u_i) \quad : \quad \begin{cases} remove(j) & \text{: kein code} \\ m_i \text{ setzt Marke } k \text{ auf Knoten } u & \text{: sonst} \end{cases}$$

5.9.6.2 Ausdrücksübersetzung Fälle:

1. u Konstante $u \in \{0,1\}^{32}$
   $ecode(m_i)$: 2 DLX_0-Instruktionen

   $\boxed{GPR[k] := u}$ (Übung!)

2. u Name

$$b_i(c,u) = \begin{cases} (gm,j) & : u \text{ nicht lokal} \rightarrow abase(c,u) = SBASE + displ(j,gm) \\ (topl,j) & : u \text{ lokal} \rightarrow abase(c,u) = \underbrace{base(F_{rd-1})}_{GPR[30]} + displ(j,gm) \end{cases}$$

   (f: Funktion deren Rumpf gerade übersetzt wird)
   $ecode(m_i)$: $R(u) = 0$
   u nicht lokal:
   u nicht lokal:

$$\boxed{GPR[k] := \underbrace{SBASE + displ(j,m)}_{2\ Instruktionen}}$$

   u lokal:

$$\boxed{GPR[k] := \underbrace{GPR[30] + displ(j,f)}_{1 \times add.imm}}$$

   $R(u) = 1$: zusätzlich $GPR[k]$ dereferenzieren

$$\boxed{GPR[k] := \underbrace{m[GPR[k] + 0]}_{load}}$$

Korrektheit:

$R(0)$: u lokal

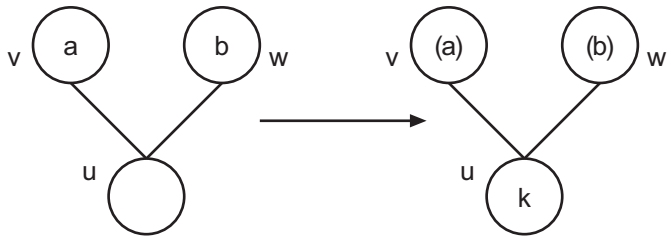
$$\begin{aligned} d(i).GPR[k] &= c.GPR[30] + displ(j,f) \\ &\text{(Lemma: } c.GPR[30] = d(i).GPR[30]) \\ &= base(F_{c.rd-1}) + displ(j,f) \\ &(r - consis(c, abase, d)) \\ &= base(bind(c,u)) \\ &\text{(Def.: } bind) \\ &= abase(c,u) \\ &\text{(Def.: } abase(c,Name)) \end{aligned}$$

(triviales Nachrechnen)

$R(1)$: $d'(i)$ nach einem weiteren Schritt

$$\begin{aligned} d'(i).GPR[k] &= d.m[d(i).GPR[k]] \\ &= d.m[abase(c,m)] \\ &= va(c,m) \quad (e - consis(\ldots)) \end{aligned}$$

3. $u = v \circ w$



Induktions-Vorraussetzung $(i-1)$:

$$\begin{aligned} d(i-1).GPR[a] &= va(c,v) \\ d(i-1).GPR[b] &= va(c,w) \end{aligned}$$

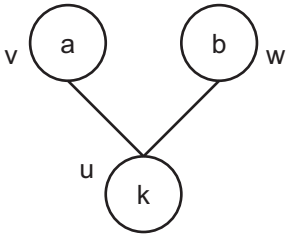
Code:

$$GPR[k] := GPR[a] \circ GPR[b]$$

Korrektheit:

$$\begin{aligned} d(i).GPR[k] &= d(i-1).GPR[a] \circ d(i-1).GPR[b] \\ &\quad ((DLX_0\text{-Semantik}) \\ &= va(c,v) \circ va(c,w) \\ &= va(c, v \circ w) \\ &\quad ((C_0\text{-Semantik}) \end{aligned}$$

4. $u = v[w]$



Induktions-Vorraussetzung $(i-1)$:

$$\begin{aligned} d(i-1).GPR[a] &= abase(c,v) \\ d(i-1).GPR[b] &= va(c,w) \\ typ(c,v) &= t \\ abase(c,v[w]) &= abase(c,v) + va(c,w) \cdot size(typ(c,v)) \end{aligned}$$

Code:

$$\boxed{\begin{array}{lll} GPR[28] & := & \underbrace{GPR[b] \cdot size(typ(c,v))}_{buing:\ Mult.-Alg.} \\ \\ GPR[k] & := & GPR[a] + GPR[28] \end{array}}$$

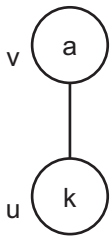
$$\begin{aligned} R(u) = 0 &: fertig \\ R(u) = 1 &: dereferenzieren \end{aligned}$$

Korrektheit: (*Übung*)

5. $u = v.n$

   (Übung!)

6. $u = v*$



   $R(u) = 0$:

   $\boxed{GPR[k] := GPR[a]}$

   $R(u) = 0$:

   zusätzlich dereferenzieren

7. $u = \&v$

   $\boxed{GPR[k] := GPR[a] \ (?)}$

5.9.7 r-Konsistenz

Zur Veranschaulichung, siehe Funktions-Stack (Abb. 44, Seite 75) und Funktions-Frame (Abb. 45, Seite 76).

$d: Funktions - Stack$
$r - consis(c, \quad, d)$

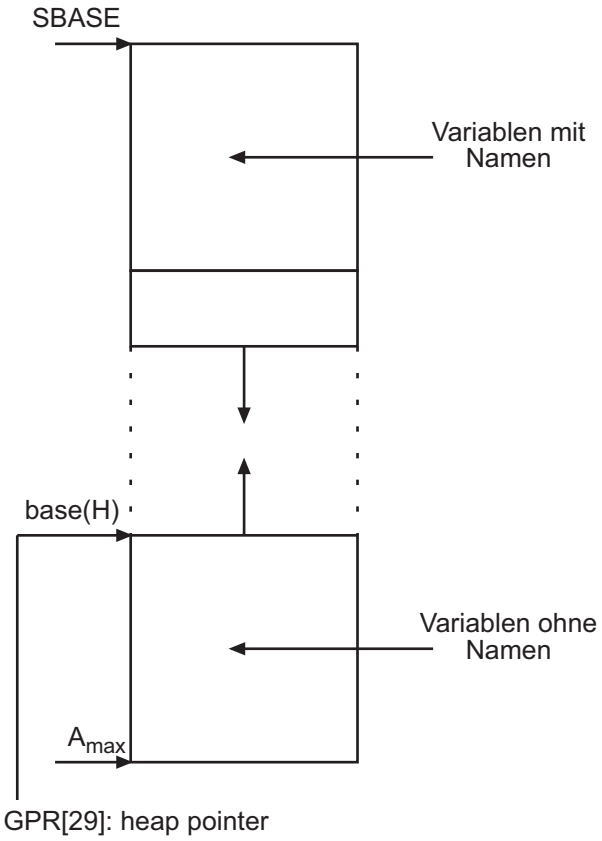


Abbildung 46: Heap und Stack im DLX_0-Memory

$$base(H) \quad = \quad min\{ \underbrace{abase(hm, j)}_{C_0\text{-Variable im Heap}} \mid j < c.hm.nv\}$$

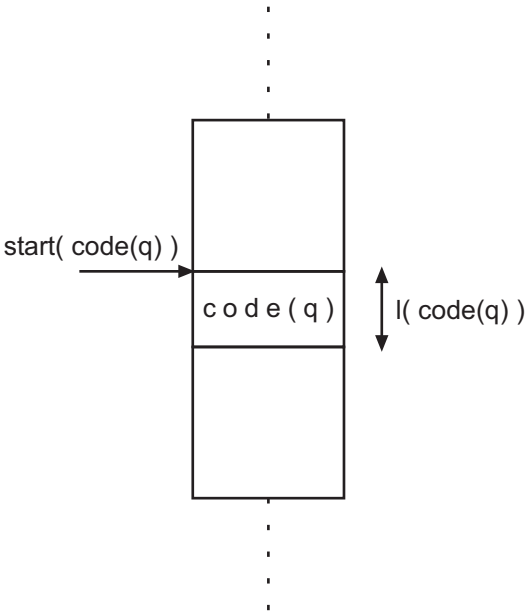
$$\begin{array}{|lcl|} \hline d.GPR[30] & = & base(c, F_{c.rd-1}) \\ d.GPR[29] & = & base(c.H) \\ \hline \end{array}$$

5.9.8 c-Konsistenz

$d: Funktions - Stack$
$c - consis(c, d)$

q : An, AnF in Funktion f
$\downarrow$
$code(f)$: Folge von DLX_0-Anweisungen

Sei s Folge von DLX_0-Anweisungen
$start(s)$: 1. Adresse, die von $code(s)$ belegt ist



$$first(q) = \begin{cases} q & : \text{falls } q < An > \\ \text{erste Anweisung von } q & : \text{falls } q < AnF > \end{cases}$$

$c.pr$: Folge der noch auszuführenden C_0-Anweisungen

$first(c.pr)$: erste dieser Anweisungen
$\downarrow$
$code(first(c.pr))$: übersetzte erste Anweisung

start(code(first(c.pr))) = d.PC

5.9.9 Satz

Zu C_0-Rechnung $c^1, c^2, \ldots$

$\exists$ DLX_0-Rechnung	$d^1, d^2, \ldots$
$\exists$ Funktionen	$alloc^1, alloc^2, \ldots$
$\exists$ Schrittzahlen	$s(1), s(2), \ldots$

so daß $\forall$ Schritte i der C_0-Maschine

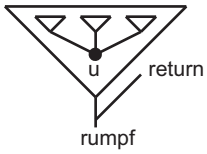
$$consis(c^i, alloc^i, d^{s(i)}$$

Beweis:

Mit Ausnahme von $c - consis(\ldots)$ einfaches Ausrechnen

5.9.10 Pre-Order-Traversal

Reihenfolge der Erzeugung der $code(u)$: *Pre-Order-Traversal*

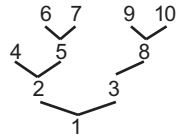


T	:	Baum
$POT(T)$	:	pre-order-traversal von T (Permutationen der Knoten von T)

Definition (rekursiv):

1. $T = u$: $POT(T) = u$
2. $T = T_1 \ldots T_s = u$: $POT(T) = POT(T_1), \ldots, POT(T_s), u$

Beispiel:



$$POT(T) = 4, 6, 7, 5, 2, 9, 10, 8, 3, 1$$

5.9.11 Code-Generierung

Fälle:

1. **Zuweisung** $q : e = e'$

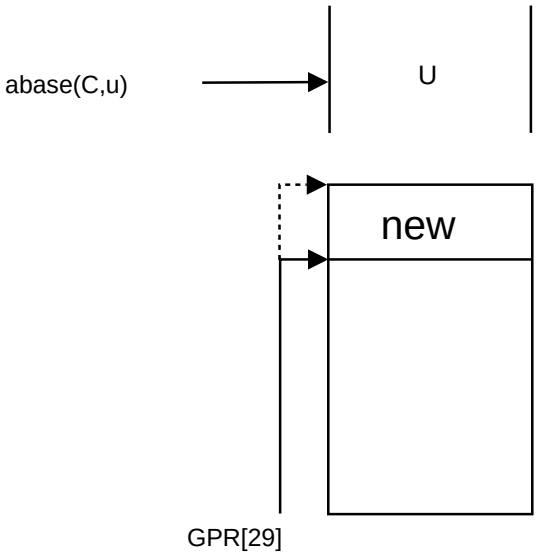
   $\boxed{\begin{array}{c} code(e) \\ \hline code(e') \end{array}}$

   - $R(e) = 0$
   - Ergebnis in $GPR[i]$
   - Marke j nicht verwenden $\Leftrightarrow$ $GPR[j]$ nicht überschieben
   - Ergebnis in $GPR[k]$

   $\boxed{m(GPR[j] + 0) = GPR[k]}$ $store$
   Korrektheit: leicht (mit Ausname von $c - consis$

   $$\begin{array}{rcl} e & - & consis \\ p & - & consis \\ r & - & consis \\ c & - & consis \end{array}$$

2. **Speicherallozierung** $q : new \quad t^*$



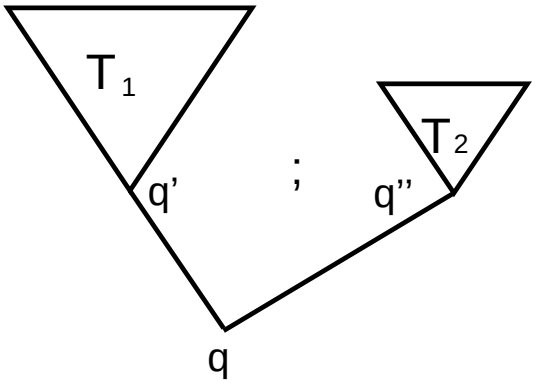
   $\boxed{GPR[29] := GPR[29] - size(t)}$ Heap größer

   $\boxed{code(u)}$

   - $R(u) = 0$
   - Ergebnis in $GPR[j]$

- Adresse des Pointers in $GPR[j]$
- Pointer setzen:
  $\boxed{m(GPR[j] := GPR[29]}$

3. **Anweisungsfolge** $q : AnF; An$



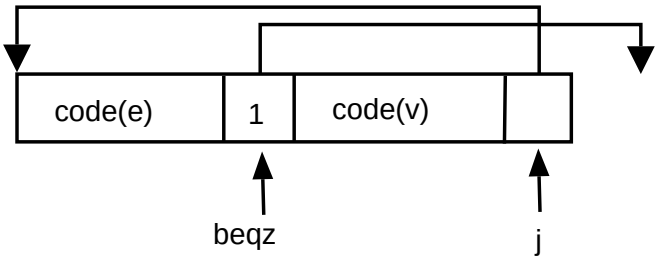
$$\begin{aligned} q &= q'; q'' \\ q' &: AnF \\ q'' &: An \end{aligned}$$

- $code(q) = code(q'); code(q'')$ - automatisch bei POT.

4. **If - Anweisung** $q : if$ a $then$ b1 $else$ b2

   Übung

5. **While - Schleife** $q : while$ e do u



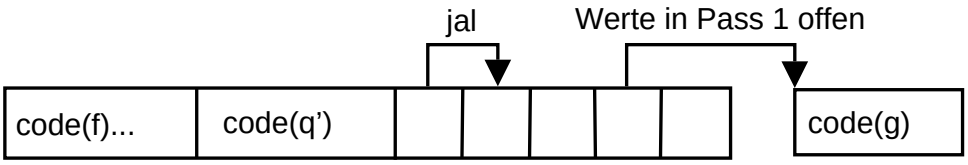
$\boxed{code(e)}$

- Ergebnis in $GPR[k]$
- $beqz$ $\boxed{PC := PC + GPR[k]?l(code(u) + 2) : 1}$
- $l(code(u))$ - bekannt (POT)

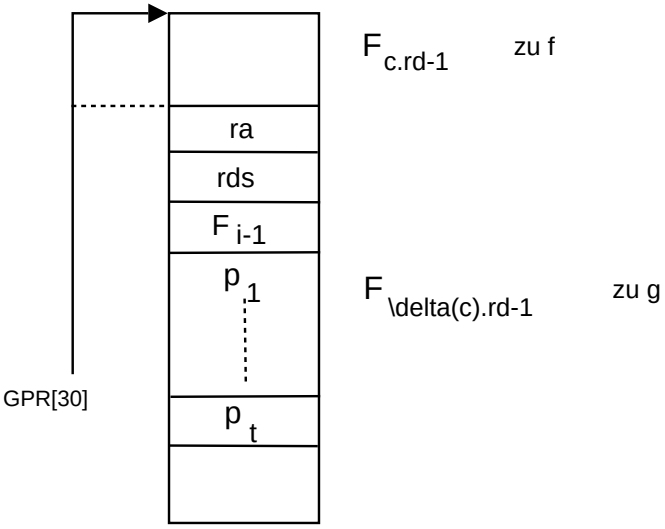
$\boxed{code(u)}$

- j $\boxed{PC := PC - (l(code(u)) + l(code(e)) + 1)}$

6. **call** $q : e_0 = g(e_1, \cdots, e_k)$ in Rumpf(f)



$p_1, \cdots, p_t$ - Parameter von g



$p(code(i))$: übergibt Parameter (i)
$p(code(1)), \cdots, p(code(t))$
$p(code(i))$ $\boxed{ecode(R_i))}$

- $R(e_i) = 1$
- Ergebnis in $GPR[ji]$
- $\boxed{m(GPR[30] + (size\, lm_f + 3) + 3 + i - 1) := GPR[ji]}$

rds $\boxed{ecode(e_0)}$

- $R(e_0) = 0$
- Ergebnis in $GPR[j]$
- $\boxed{m(GPR[30] + (size(\quad) + 3 + 1) := GPR[j]}$

  altes GPR retten

- $\boxed{m(GPR[30] + (size(\quad) + 3 + 2) := GPR[30]}$

  stack Pointer auf neuen Top Frame.

- $\boxed{m(GPR[30] := GPR[30] + size(\quad) + 3}$

- bis hierher : $code(q')$
- $\boxed{m(GPR[1] := start(code(g))}$

  bekannt nach Pass 1

jal:

- $PC := PC + 1, GPR[31] := PC$
- $GPR[31] := GPR[31] + 6$
- $\boxed{m(GPR[30] := GPR[31]}$ - Rückgabeadresse
- $\boxed{GPR[1] := start(code(g))}$ - Pass 2
- $PC := GPR[1]$ - Sprung in g

7. **return** Übung

6 Erweiterungen zur DLX_0

6.1 DLX_0 mit Interrupts

6.1.1 Definition: Interrupt

Ereignis, das den „normalen“ Programmablauf unterbricht, stattdessen: Aufruf einer Interrupt Service Routine (ISR).

6.1.2 Grober Ablauf

1. Interrupt wird ausgelöst
2. CPU speichert:
   - Teile des aktuellen Programmzustandes (v.a. PC)
   - Quelle/Grund des Interrupts
3. Behandlung des Interrupts: ISR
4. Meistens: Wiederherstellung des Zustands vor/bei Interrupts (teils Hardware, teils Software)

6.1.3 Klassifikation

Unterscheidungsmerkmale von Interrupts:

1. interner/externer Interrupt
   - interner Interrupt (exception, trap)
     Ausgelöst durch Prozessor oder Memory.
   - externe Interrupt
     Quelle ausßerhalb von CPU oder Memory (z.B. ein Gerätertreiber).
     Nicht deterministisch - Zeitpunkt des Auftretens ist nicht vorhersehbar.
2. (nicht) maskierbar
   maskierbare Interrupts können per Software ignoriert bzw. abgeschaltet werden.

3. Return Type
   Sei I die unterbrochene Funktion:

   - „repeat“:
     nach ISR wird I ausgeführt
   - „continue“:
     nach ISW wird I' (die Instruktionsfolge von I, die ohne Interrupt aufgetreten wäre)
   - „abort“:
     weder mit I noch mit I' weitermachen

4. Priorität
   Falls zwei Interrupts zeitgleich auftreten wird der Interrupt mit der höchsten Dringlichkeit zuerst behandelt

6.1.4 Interrupts des DLX_0

Identifiziere Interrupts mit Nummer $j \in \{0, \ldots, 31\}$

$j \rightarrow Prioritt$ 0: dringend
$\vdots$
31: nicht so dringend

j	Abkürzung	Name	Return-Type	Maskierbar	Ext/Int
0	$reset$	Reset/Power-Up	abort	nein	ext
1	ill	Illegal Instruction	abort	nein	int
2	$\rightarrow$ RA1				
3	pff	Page Fault Fetch	repeat	nein	int (Mem)
4	$pfls$	Page Fault Load/Store	repeat	nein	int (Mem)
5	$trap$	Trap (SW-Interrupt)	continue	nein	int
	($\rightarrow$ leitet System-Call-routine des OS ein)				
6	ovf	arithmetic overflow	continue/abort	ja	int
$\underbrace{6+i}_{i\in\{1,\ldots,25\}}$	ex_i	External I/O	continue	ja	ext

6.1.5 Konfiguration / Register

$$c = (pc, spr, gpr, mem)$$

$$\begin{aligned} c.pc &: \{0,1\}^{32} \\ c.spr &: \{0,1\}^5 \rightarrow \{0,1\}^{32} \\ c.gpr &: \{0,1\}^5 \rightarrow \{0,1\}^{32} \\ c.mem &: \{0,1\}^{32} \rightarrow \{0,1\}^{32} \end{aligned}$$

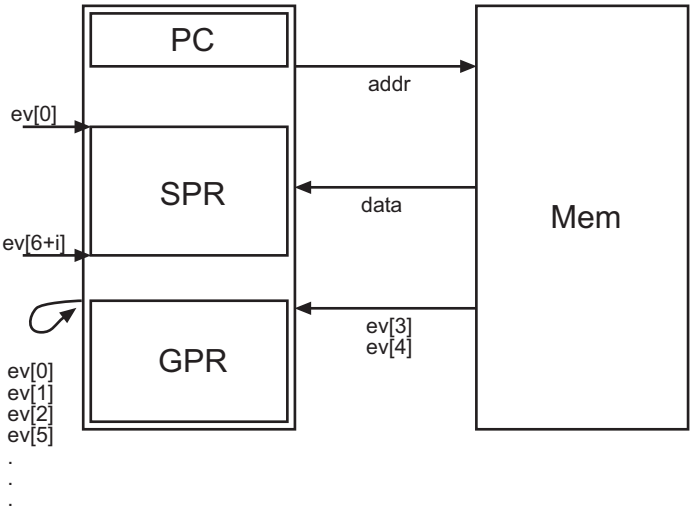


Abbildung 47: Special Purpose Register

Derzeit werden nur $SPR's$ benutzt:

$SPR[0]$	$\sim$	SR	(Status Register)
$SPR[1]$	$\sim$	ESR	(Exception Status Register)
$SPR[2]$	$\sim$		(Exception Cause Register)
$SPR[3]$	$\sim$	EPR	(Exception Program Counter)
$(4$	$\rightarrow$	$RA1)$	
$SPR[5]$	$\sim$	$Edata$	(Exception Data)

$Berechnung$:

$$C^0 \xrightarrow{I_1} C^1 \xrightarrow{I_2} C^2 \xrightarrow{I_3} \dots$$

$$C^0\ Startkonfiguration$$

$$\delta\ Schrittfunktion$$

$$(!) statt\ \delta(fr\ DLX_0)\ schreibe\ nun\ \delta_n$$

6.1.6 Definition: δ für DLX_0 mit Interrupts

$$\delta(c) = \begin{cases} \delta_n(c) \text{ (plus einige neue Instruktionen)} & \text{falls kein Interrupt} \\ C' \text{ (neu - siehe unten)} & \text{sonst} \end{cases}$$

6.1.7 Neue Instruktionen

Zur Erinnerung:
$RS1 = I[25:20]$
$RD = I[15:11]$
$SA = I[10:6]$

Typ	$I[31:26]$	$I[5:0]$	Mnemonic	Effekt
R	000000	010000	movsli	$GPR[RD] = SPR[SA]$
R	000000	010001	movi2s	$SPR[SA] = GPR[RD]$
R	000000	000000	add	wie $addu$
R	000000	000010	sub	wie $subu$
I	010000	%	addi	wie $addiu$
I	010010	%	subi	wie $subiu$
I	111110	%	trap	löst $trap$ aus
I	111111	%	rfe	$SR = ESR$, $PC = EPC$

6.1.8 Interrupts

Eventsignale für interne Interrupts:

$ev[1] = 1 \quad \Leftrightarrow \quad I = mem(PC)$
I keine bekannte Instruktion
$ev[1] = \underbrace{(add(I) \vee sub(I) \vee \ldots)}_{fr\ alle\ Instruktionen}$
$ev[3] = 1$ $nchste\ Vorlesung$!
$ev[4] = 1$ $nchste\ Vorlesung$!
$ev[5] = 1 \quad \Leftrightarrow \quad trap(I)$
$ev[6] = 1 \quad \Leftrightarrow \quad (add(I) \vee sub(I) \vee addi(I) \vee subi(I)) \wedge ovf$ (Overflow-Signal aus ALU)

Wie bereits erwähnt: $ev[1]$ und $ev[6+i]$ extern!

6.1.9 Definition: nicht sichtbar

Masked Cause Register (MCR):

$$MCA[j] = \begin{cases} ev[j] & \text{falls } j \text{ nicht maskierbar ist} \\ SR[j] \wedge ev[j] & \text{sonst} \end{cases}$$

(!) $SR[j]$ bestimmt ob maskierbarer Interrupt auftreten kann.

6.1.10 Definition: JISR-Signal (Jump ISR)

$$JISR = \bigvee_j MCA[j]$$

6.1.11 Definition: Interrupt Level

$$il = min\{j | MCA[j] = 1\}$$

6.1.12 Delta-Funktion

$$
\begin{array}{rcl}
\delta(c) & = & c' \text{ falls } JISR = 1, \text{ wobei für } c' \text{ gilt:} \\
 & & \\
ESR & = & SR \\
ECR & = & MCR \\
EPC & = & \begin{cases} PC & \text{falls Interrupt il vom Typ „repeat“} \\ PC^u * & \text{sonst} \end{cases} \\
Edata & = & \begin{cases} imm & \text{falls } trap(I) \\ EA & \text{falls } lw(I) \text{ oder } sw(I) \end{cases} \\
SR & = & 0^{32} \\
PC & = & SISR \text{ Start der ISR (Konstante, z.B. 512)}
\end{array}
$$

6.1.13 Aufbau der ISR

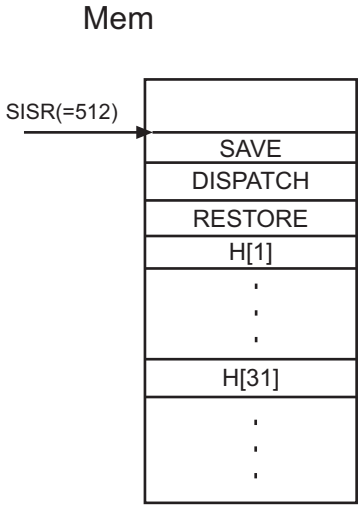


Abbildung 48: Interrupt Service Routine (Aufbau)

Ablauf:

$SAVE$	-	sichere Register
$DISPATCH$	-	ermittle il aus ECR (find first one comp)
$H[il]$	-	Handler für Interrupt il
$RESTORE$	-	Register wiederherstellen, rfe zum Beenden der ISR benutzen
$H[0]$	-	Sonderbehandlug für $reset$

6.1.14 Devices & I/O (Geräte & E/A)

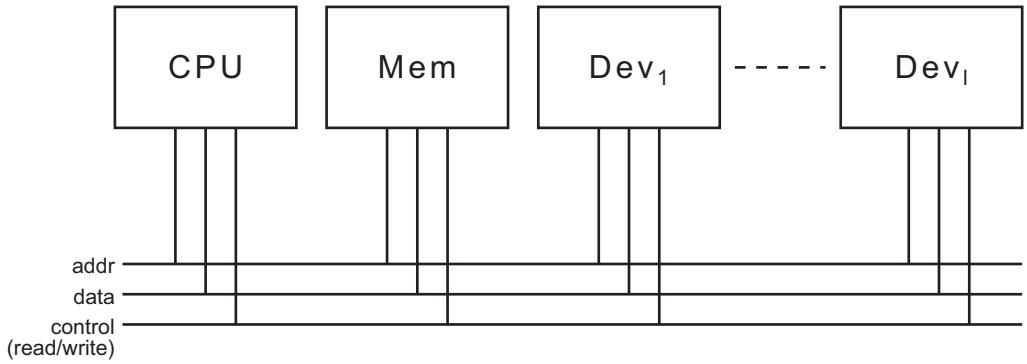


Abbildung 49: Bus-System

CPU :
„Bus Master“, add und $control$ werden **nur** durch die CPU gesteuert.

$data$:
Kann von allen benutzt werden, sollte aber nur von höchstens einer Komponente gleichzeitig benutzt werden.

Seien: $k \in \mathbb{N},\ 2^{k-1} < l < 2^k$
$m \in \mathbb{N},\ m + k < 2^k$

Idee: Gerät $1 \leq u \leq l, < i[k-l\ :\ 0] >= i$

Gerät über Adressen $a[31:0]$ ansprechen mit:
$a[31\ :\ 32-m] = 1^m$
$a[31-m\ :\ 32-m-k] = i[k+1\ :\ 0]$

6.1.15 Beispiel: Hard Disk (HD)

Festplatte mit Geräteadresse $< i[k-1 \,:\, 0] >= i$:

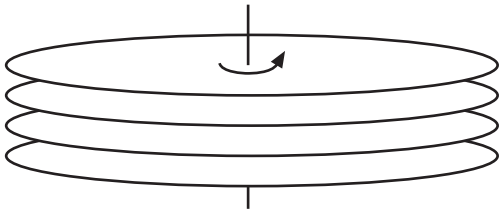


Abbildung 50: Festplatte

- Daten werden abgelekt in Sektoren (zu je 512 Bytes)
- Sektoren haben Koordinaten:

  „c“ylinder: Abstand vom Zentrum

  „h“ead: „Etage“ (einzelne Platten)

  „s“ector: Sektor des durch (c, h) bestimmten Kreises

  Die Festplatte rechnet dies um in die lba (linear block adress):
  $lba \in \{0,1\}^{28}$: Festplattengröße $\leq\ 2^9 \times 2^{28} bytes\ =\ 128\ GB$

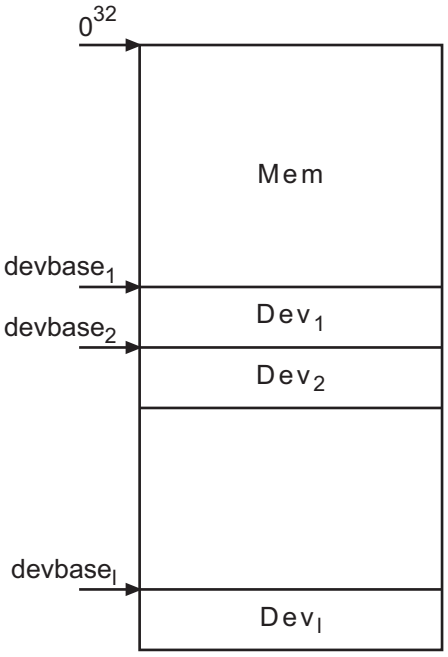


Abbildung 51: Devbase

Ab Adresse $devbase_i = 1^k\, i[k-1:0]\, 0^{32-m-k}$ blendet die Festplatte Register ein:

busy	=	$0^{31}1$	(HD beschäftigt)
error	=	$0^{31}1$	(letztes Kommando lieferte Fehler)
cmd	=	0^{32}	(READ)
	=	$0^{31}1$	(WRITE)
lba	=		
count	=	$\#Sektor$	
ien	=		(Festplatte löst Interrupt aus)
data	=		(32-bit Daten, Read/Write)

6.1.15.1 Lesezugriffe

1. CPU setzt (sw'), lba, $count$
2. CPU setzt $cmd = 0^{32}$
3. HD setzt $busy \neq 0^{32}$ solange sie liest
4. CPU wartet bis $busy = 0^{32}$ (*)
5. CPU prüft $error \neq 0^{32}$ → Fehler, Abbruch
6. CPU liest 128 Wörter (≙ 512 Bytes) vom Data Register (**)
7. Schon $count$ Sektoren gelesen? (Nein: → wieder zu Schritt 3)

(*) Polling
(**) Kopieren
} „$stupide$“ $Arbeit\ für\ die\ CPU$

Um die CPU von dieser Arbeit zu entlasten gibt es folgende Möglichkeiten:
(*): ien$\neq 0^{32}$ → HD löst externen Interrupt ex_i aus, falls non-busy
(**): DMA(direct memory access) → HD kann selbst von/in Memory lesen/schreiben

6.2 Virtueller Speicher(VM)

6.2.1 Motivation

RAM ist teuer, dagegen sind Festplatten billig. Es stellt sich also die Frage, ob es möglich ist, mit wenig RAM und einer Festplatte viel RAM zu simulieren (eine voll ausgestatteter DLX_0 kann $2^{32} \times 4\ Bytes = 16\ GB$ adressieren!).

Ziel zunächst:
Simulation für ein Programm (user-program)

Lösung: „Demand Paging“

- lade Teile des Speichers von HD nach, wenn gebraucht
- Page: 4KB zusammenhängender Speicher (1024 Worte)
- Paging: Pages auf Festplatte speichern bzw. von Festplatte laden

6.2.2 Vergleich zweier Maschinen

(die alte) DLX_0:

- keine Interrupts
- keine Devices
- keine SPR's
- memsize $\in \{0,1\}^{32}$: Größe von Mem, nur Zugriffe auf Adressen $a \in \{0,1\}^{32}$ mit $< a > < < memsize >$ „erlaubt"

(die neue) DLX_0:

- mit Interrupts
- mit Devices
- mit SPR's
- memsize
- Modes & Adress Translation

6.2.3 DLX_0 mit Interrupts & Adress Translation

Konfiguration:

$$\text{c = (pc, spr, gpr, pm}^6\text{, sm}^7\text{: } \{0,1\}^{32} \rightarrow \{0,1\}^{32})$$

neue SPR's:

9	pto	page-table origin
10	ptl	page-table last
11	Emode	exception mode
16	Mode	mode-flag: $0^{31}1: \ user-mode$ $0^{32}: \ system-mode$

Im user-mode (aufgrund einer Page-Table im PM):

- leite Speicherzugriff auf Adresse $va \in \{0,1\}^{32}$ um auf Adresse $pa \in \{0,1\}^{32}$ im PM

ODER

- löse Interrupt aus (Page Fault - ev[3]/ev[4]), → ISR, Page Fault Handler wird ausgeführt

[6] physical memory (altes Mem)
[7] swap memory(zusammenhängender Bereich auf HD)

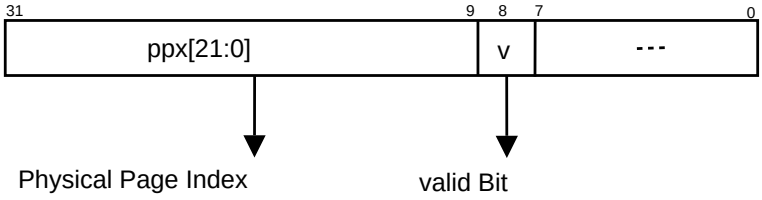
6.2.4 Page Table Lookup

$va[31:0] = (px[21:0], wx[9:0])$

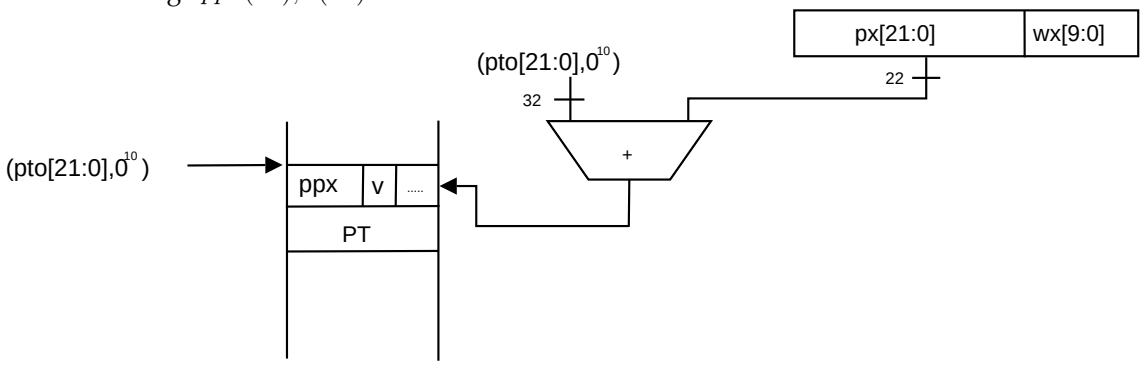
$wx \quad \rightsquigarrow$ word index
$px \quad \rightsquigarrow$ page index

$$\underbrace{PT(px)}_{\text{Page Table Entry}} = PM \quad \underbrace{[ptea]}_{\text{Page Table Address}}$$

$$\langle ptea \rangle \quad = \quad \langle pto[21:0] \rangle .1024 + \langle px \rangle \quad (\mod 2^{32})$$



Abkürzung : $ppx(va), v(va)$



6.2.5 Page Fault Exception

$$pf(va) \quad = \quad (\langle px(va) \rangle > \langle ptl \rangle)$$
$$\vee \quad (v(va) = 0)$$

$$pma(va) = (\underbrace{ppx(va)}_{\text{22 Bit}}, \underbrace{wx(va)}_{\text{10 Bit}}) \in \{0,1\}^{32}$$

6.2.6 Instruktionsausführung

1. $mode = 0$: alles wie vorher

2. $mode = 1$: "$usermode$"

   (a) $pf(PC) = 1? \rightarrow$ löse pff Interrupt aus ($ev[3]$)

   (b) $I = PM[pma(PC)]$

   - $\sim (lw(I) \vee sw(I))$ - wie vorher
   - $pfa(EA) = 1? \rightarrow$ löse $pfls$ Interrupt aus ($ev[4]$)
   - lade /speichere von $pma(EA)$

Was passiert bei Interrupt?

JISR = 1	ESR	=	SR	(Masken)
	ECA	=	MCA	(Quelle)
	EPC	=	$\begin{cases} \text{PC} & \text{typ } -repeat \\ \text{PC}^{\,n} & \text{typ } -continue \end{cases}$	
	EData	=	PC ^{n}= PC vor $\delta_u(C)$ $\begin{cases} imm & \text{bei Trap} \\ \text{EA} & \text{bei } load/store \end{cases}$	
	PC	=	SISR	(start ISR)
	SR	=	0^{32}	
	MODE	=	0^{32}	EMODE=MODE

6.2.7 Simulation

$C_v^0 \xrightarrow{I_v^1} C_v^1 \xrightarrow{I_v^2} \cdots$ Berechnung der DLX_0 ohne Interrupts
v steht für virtuell

$$\begin{array}{lccccccccccccccc} mode & 0 & \cdots & 0 & & \cdots & & 0 & & 1 & & \cdots & & 1 & & 0 & \cdots \\ & C^0 & \rightarrow & C^1 & \rightarrow & \cdots & \rightarrow & C^{\alpha-1} & \rightarrow & C^{\alpha} & \rightarrow & \cdots & \rightarrow & C^{\beta-1} & \rightarrow & C^{\beta} & \cdots \end{array}$$

$C^0 \rightarrow \cdots \rightarrow C^{\alpha-1}$ Intializierung, Power-up
$C^{\alpha} \rightarrow \cdots \rightarrow C^{\beta-1}$ User Programm
$C^{\beta} \rightarrow \cdots \rightarrow C^{\gamma-1}$ Page Fault Handler

User Programm und PFH wechseln sich ab.

$C_v = (pc_v, gpr_v, mem)$
$C = (pc, spr, gpr, pm, sm)$

$\pi : C \rightarrow C_v$
π bildet C-Konfiguration auf C_v-Konfiguration ab.

$(mode = 1)$: $gpr_v(i) = gpr(i)$
$pc_v = pc$

$$mem(va) = \begin{cases} PM[pma(va)] & \text{falls } pf = 0 \\ SM[sma(va)] & \text{sonst} \end{cases}$$

6.2.8 Theorem

i - Schritt in der Berechnung von DLX_0 mit Interrupts.

$\sim (pff^i \vee pfls^i)$ - kein *page fault fetch,*
kein *page fault lode/store*
$\sim$(JISR)

$mode^i = 0^{31}1$

j - Schritt in der Berechnung von DLX_v.

$$\pi(C^i) = C_v^j \Rightarrow \pi(C^{i+1}) = C_v^{j+1}$$

7 Betriebssystem-Kern

- *abstraktes Modell:*
  CVM (communicating virtual machines)
  k abstrakter Kern (syntaktisches C_0-Programm)

- *Implementierung:*
  konkreter Kern (zwangsläufig nicht C_0):
  Benutzerprozesse, Prozessorregister } müssen sichtbar sein
  C_{0A}: C_0 + Assembler-Makros

- *Simulationssatz:*
  K + Benutzerprogramme auf physikalischer Maschine simulieren
  k + Benutzerprogramme vom CVM-Modell

7.1 CVM: Syntax und Semantik

Konfigurationen:

$$c \;=\; (c.np,\; c.C,\; c.vm(p),\; c.cp)$$

$c.np$: number of processes $(c.np = 1024)$
($c.np \in \mathbb{N}$)

prozess 1: abstrakter Kern, C_0-Maschine
$c.C$: Konfiguration der C_0-Maschine

prozess 2: Benutzer-Prozesse (virtuelle DLX_0-Maschinen)
$c.vm(p)$: Konfiguration der virtuellen DLX_0-Maschinen
($p \;=\; 1, \ldots, c.np - 1$)

$c.cp$: current process
($p \in \{0, \ldots, c.np - 1\}$)

tvm : total virtual memory [Bytes] $(4\,GB)$

$c.vm(p).ptl$: # Pages (je 4K) von der virtuellen Maschine $p > 0$

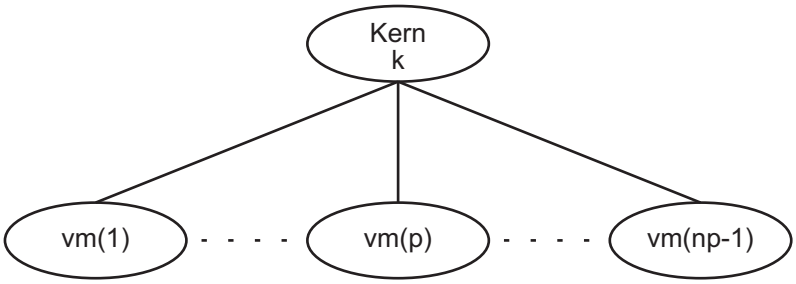


Abbildung 52: Communicating Virtual Machine

Forderung:

$$4K \times \sum_{p=1}^{c.np-1} c.vm(p).ptl \leq tvm$$

7.1.1 Exkurs: Syntax von k

Funktionen die k aufrufen darf:

$alloc(p, x)$	:	neue Seiten für $vm(p)$
$free(p, x)$	:	Seiten von $vm(p)$ entfernen

Semantik:

Sei	$c.cp$	=	0	(k läuft)
	$c.C.pr$	=	$alloc(p, x);\ r$	(Aufruf von $alloc(\ldots)$)
	$\delta(c)$	=	c'	(nächste CVM-Konfiguration)
$\Rightarrow$	$c'.C.pr$	=	r	(abstrakt: $alloc(\ldots)$ wird in einem Schritt abgearbeitet)
	$c'.vm(p).ptl$	=	$c.vm(p).ptl\ +\ x$	

7.1.2 Semantik von cp

$$\begin{array}{rcll} c.cp & = & p > 0 & \text{Benutzerprozess } p \text{ läuft} \\ JISR(c.vm(p)) & = & 0 & \text{kein Interrupt einer virtuellen Maschine } (vm \text{ sieht keine PageFaults!}) \end{array}$$

δ_{vm}: Nachfolge-Konfiguration von vm:

$$\left.\begin{array}{rcl} c.vmp(p) & = & \delta_{vm}(c.vmp(p)) \\ c'.vm(x) & = & c.vm(x) \;\; (x \neq p) \\ c'C & = & c.C \\ c'.cp & = & p \end{array}\right| \begin{array}{c} \text{(!) Simulation kann Handling} \\ \text{von Page-Faults efordern} \\ \text{(Interrupt auf physikalischer Maschine)} \end{array}$$

$c.cp \;=\; 0$ (k läuft)
$c.C.pr \;\neq\; s;\; r$ (s: normales C_0-Statement)

$start$: Statement von k (startet UserProzess 'act')
act: int-Variable von k

$$\left.\begin{array}{l} c'.C \;=\; \delta_c(c.C) \\ \delta_c\text{: Nachfolge-Konfiguration für } C_0\text{-Maschinen} \end{array}\right\} \text{Kern läuft lokal}$$

$c'.vm(p) \;=\; c.vm(p)$

7.1.2.1 Kontrolle von k an User

$$\begin{array}{lrcl} \text{Sei} & c.C.pr & = & startu;\; r \\ & c'.C.pr & = & r \\ & p & = & va(c.C,\; act) \\ \\ \Rightarrow & c'.cp & = & p \end{array}$$

7.1.2.2 Kontrolle von $vm(p)$ an k

Dies ist nur durch Interrupts (traps!) möglich.

7.1.3 Semantik von Interrupts

Speziell:

traps = kernel calls (= Prozedur-Aufrufe)

$$\begin{array}{lrcll} \text{Sei} & c.cp & = & p \;>\; 0 & \\ & JISR(c.vm(p)) & = & 1 & \\ & trap(c.vm(p)) & = & 1 & \text{(Prädikat wie } add.load\text{)} \\ & < imm(c.vm(p)) > & = & i & \text{(Parameter von } trap \;=\; i\text{)} \end{array}$$

$\Rightarrow$ Aufruf von kernel call i durch abstrakten Kern k

7.1.4 Formalismus zum Spefizieren der Handler von Kernel Calls

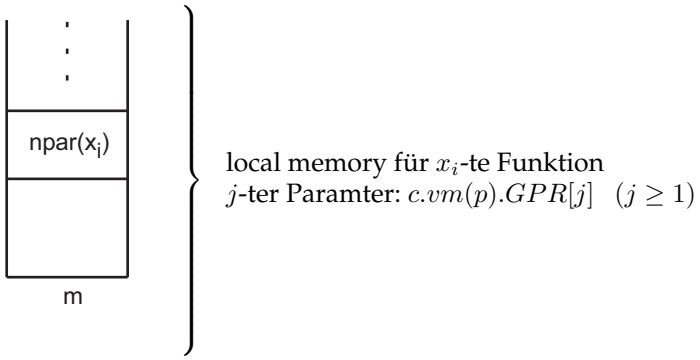
Analog zur Funktions-Definition in C_0-Konfigurationen:

$c.C.fd$	:	$[0 : nf - 1] \rightarrow FD$	(gibt es, function declaration)
nf	=	# C-Funktionen des Kerns	
Sei $npar(x)$	=	# Parameter der x-ten Funktion von k	
$c.nkc$	$\in$	$\mathbb{N}$	(number of kernel calls)
$c.kcd$	:	$[0 : c.nkc - 1] \rightarrow [0 : c.C.nf - 1]$	(kernel call definition)

7.1.4.1 Definition der Wirkung von $trap(i)$

$c'.cp$	=	0	(Kern k läuft)
$c'.C.rd$	=	$c.C.rd + 1$	(Funktionsaufruf)

rufe auf: x_i = $c.kcd[i]$-te Funktion von k (hat $npar(x_i)$ Parameter $\leq$ 20)



Für kernel calls:
Semantik von $return(e)$ nach rds zurückgeben.

$c'.C.rds(c'.C.rd)$	=	p	(aufrufenden Prozess merken)
$return(e)$	:	Rückgabewert an $GPR[1]$ von p	

typ der Parameter : int
typ des Rückgabewerts : int

7.2 CVM mit I/O-Devices

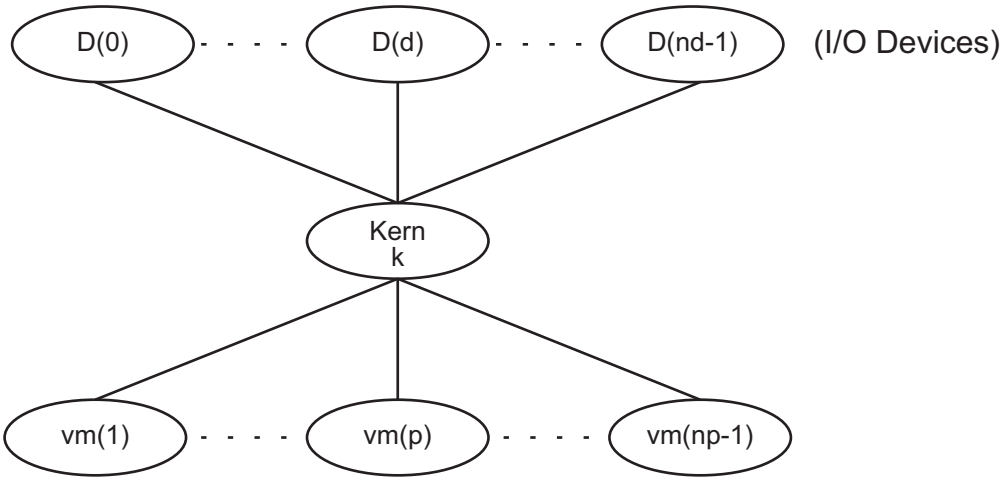


Abbildung 53: Communicating Virtual Machine mit I/O Device

7.2.1 Wiederholung: Interrupts

$0 = reset$] Boot-Routine

$1 = ill$
$2 = mal$

$3 = pff$
$4 = pfls$
] werden vom konkreten Kern behandelt

$5 = trap$] k-Funktion

$6 = XOVF$
$7 = OVF$
$8 = UNF$
$9 = INX$
$10 = DBZ$
$11 = INV$
] floating point] „$goto$“ im Benutzer-Programm

$12 = timer$] Scheduler

$13 = timer$
$\vdots = timer$
] ext_i von $D(d)$ (User Prozesse)

$IS = [6:11]$ (IEEE - FP+XOVF)
$ES = [13:31]$ (extern)
$SP = \{1, 2, 12\}$ „special“ (Handler von Kern-Funktionen)

7.2.2 $return$

klassisch (C: C_0-Konfiguration)			*aus kernel call*		
	$C.pr$	$= return(e); r$			
	$C'.rd$	$= C.rd - 1$	$c'.C.rd$	$= c.C.rd - 1$	
Sei	$C.rds(rd)$	$= (\underbrace{m : a}_{rds})\ m \in \{gm, hm, lms(c.rd-1)\}$	$c.C.rds(c.C.rd)$	$= p$	
	$c'.m.ct[a]$	$= va(C.e)$	$c'.vm(p).GRP[1]$	$= \underbrace{va(c.C.e)}_{(!)\ typ(c.C.e)=int}$	

7.2.3 Kernel Calls

	$c.cp$	$=$	$p > 0$	(Benutzerprozess)
	$trap(c.vm(p))$	$\vee$	$il(c.vm(p)) = 5$	
	$< imm(c.vm(p)) >$	$=$	i	($\boxed{trap \mid bin_{26}(i)}$)
	$c.kcd(i)$	$=$	j	(k-Funktion mit Restriktor $c.C.fd(j)$ behandelt trap i)
Sei	n	$=$	$c.C.fd(j)$	
$\Rightarrow$	$\bullet\ c'.cp$	$=$	0	k läuft
Sei	n	$=$	$k.nparam$	(# Parameter von k)
	$\bullet\ \underbrace{c'.C.rd}_{rd'}$	$=$	$c.C.rd + 1$	(neues lm und rds)
	$tlm(c')$	$=$	$c.C.lms(rd')$	(neues top local memory)
	$\bullet\ tlm(c').st$	$=$	$h.st$	(Symbol table übernehmen)
	$\bullet\ tlm(c').ct[\underbrace{ba(h,i)}_{i}]$	$=$	$c.vm(p).GPR[i]$	
	$\bullet\ c'.C.rds(rd)$	$=$	p	(Rückgabe an Prozess p)
	$\bullet\ c'.C.pr$	$=$	$n.body;\ c.C.pr$	

7.2.3.1 $i = il(c.vm(p)) \in SP$ **(„special“)**

$c.shd : SP \rightarrow [0 : c.C.nf - 1]$ (special handler definition)

Sei $j = c.shd(j)$

$j = c.shd(j)$

$\vdots$

Rest weiter wie trap

7.2.3.2 $i = il(c.vm(p)) \in IS$

$c.ihd : IS \rightarrow \{0,1\}^{32}$

$c'.vm.PC = c.ihd[i]$ (schlichtes goto)

7.2.3.3 $i \in ES$

$$c.hd \quad : \quad ES \rightarrow \{ \underbrace{1, \ldots, np-1}_{aufrufender\ Prozess} \}$$

$$p' \quad = \quad c'.cp\ (= c.hd[i]) \qquad \text{(Aufruf des Prozesses)}$$

$$c.dbase \quad : \quad [0 : np-1] \rightarrow \mathbb{N}$$

$c.dbase(p)$: Start der data section

$$c'.v_{,10}\,[c.dbase(p')] \quad = \quad c.vm(p).GPR[10:1]$$

$$c'.vm(p').PC \quad = \quad 0^{32}$$

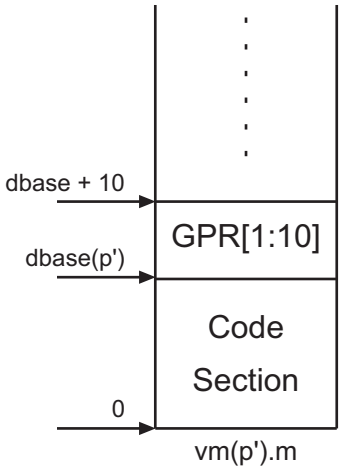


Abbildung 54: $vm(p').m$

7.2.4 „User-Functions“

k: C_0-Programm + $startu$ + $\underbrace{\text{vorderfinierte Funktionen}}_{\text{z.B. } alloc, free}$

- $c.cp \;=\; 0$
  k läuft

- $c.C.pr \;= setmasks(p, e);\; r$
  $c'.C.pr \;=\; r$
  $c'.vm(va(c.C.p)).SR \;=\; va(c.C, e)$

  Notation: $\widetilde{x} \;=\; va(c.C, x)$ (x: g-Ausdruck)
  $\Rightarrow\;\; c'.vm(\widetilde{p}).SR \;=\; \widetilde{e}$ (Status-Register SR speichert Masken)

- $copy(p_1, p_2, s_1, s_2, l)$
  von p_1 (ab s_1) nach p_2 (ab s_2) l Daten kopieren:
  $c'.vm(\widetilde{p_2})_{\widetilde{l}}(\widetilde{s_2}) \;=\; c.vm(\widetilde{p_1})_{\widetilde{l}}(\widetilde{s_1})$

- Handler-Definitionen ($sethd(i, j)$ / $setihd(i, j)$)
  $c'.hd(\widetilde{i}) \;=\; \widetilde{j}$

- $reset(p)$
  $ptl(p) \;=\; 0 \ldots$ (einfachste Primitive?)

- $fork(p, q)$
  q Kopie von p (einfachste Primitive?)

7.2.5 Devices

$c.nd$: number of devices
$c.dsize$: $[0 : c.nd - 1 \rightarrow \mathbb{N}$ (# Ports)

Definition: I/O-Port:
Adresse die von I/O-Devices belegt wird.

$c.iop[p]$: $\underbrace{[0 : c.dsize - 1]}_{Adressen} \rightarrow \underbrace{\{0,1\}^{32}}_{Inhalt}$ (Speicher-iop)

iop verhält sich manchmal wie ein Speicher
$c.alk[d] \in \{0,1\}$ (1: aktiv)

$c'.akt[d]$
testen von $c'.akt[d]$ } Device-spezifisch

typisch:

- externer Interrupt von Device $d \Rightarrow akt[d] = 0$
- Kommando an Device $d \Rightarrow akt[d] = 1$

Nur bei $c.akt[d] = 0$ (!):

- $outcopy(p, d, s_1, s_2, l)$
  $c.iop(\widetilde{d})_{\widetilde{l}}(\widetilde{s_2}) = c.vm(\widetilde{p})_{\widetilde{l}}(\widetilde{s_1})$
  $c'.iop[\widetilde{d}](\widetilde{i}) = c.iop[\widetilde{d}](\widetilde{i}) \quad i \in [\widetilde{s_2} + \widetilde{i} - 1 : \widetilde{s_2}]$

- $incopy(d, p, s_1, s_2, l)$
  $c'.vm(\widetilde{p})_{\widetilde{l}}(\widetilde{s_2}) = c.iop(\widetilde{d_{\widetilde{l}}}(\widetilde{s_1})$
  $c.akt[d] = 0 \wedge c.C.pr \neq outcopty(\ldots)$
  $\underbrace{c'.iop[d](i) = c.iop[d](i)}_{Speichersemantik}$

7.3 Definition: C_{0A} (C_0 mit Assembler-Code)

Konkreter Kern $K \in C_{0A}$ wird um den abstrakten Kern k herumprogrammiert:

- Syntax: neues Statement: $asm(s)$ (Asembler-Marko, s : Folge von DLX_0-Instruktionen)
- Compilieren: $code(asm(s)) = s$
- (!) Semantik von C_{0A}-Programmen (nutze $abase$ vom Compiler)
- (!) $kconsis(k, kbase, K)$: K codiert k
- Konstruktion von K
- Simulationssatz

7.4 Konkreter Kern K

$K \in C_{0A}(= C_0 + asm(s)^8)$

7.4.1 Semantik

$$\delta_A(cc^9) \rightarrow cc' \mid cc.pr = asm(s);\ r \quad (DLX_0\text{-Konfuguration } d \text{ sichtbar für } K)$$

(! Geht nicht !)

$$\underbrace{\delta_A(cc, d) = cc'}_{\text{Eingabe der } C_{0A}\text{-Maschine}}$$

$code(K)$ läuft auf *physikalischer* DLX_0-Maschine. Die Konfiguration der physikalischen Maschine nennen wir d.

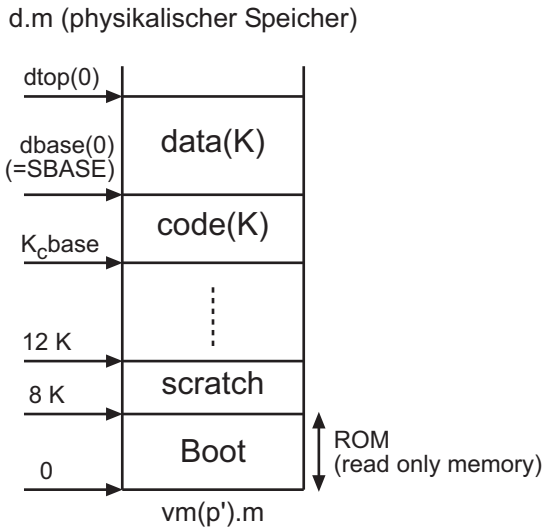


Abbildung 55: Memory Map

[8] Folge von DLX_0-Instruktionen
[9] C_0-Konfiguration

7.4.2 Restriktionen an $asm(s)$

s liest/schreibt aus/in $data(K)$ in Konfiguration d ($ea(d) \in [dtop(0) : abase(0)]$)
$\Rightarrow \exists\, x = (gm, j)s$ (globale g-Variable)

$$ea(d) = abase(\not{d}, x)$$

(bei globalen Variablen unabhängig von d)

Definition:

$$abase(x) = abase(d^0, x)$$

$cc.pr = asm(s);\ r$
$\delta_A(cc, d) = cc'$
Es gelte $d \rightarrow_s^* d'$, d.h.:

$$\begin{array}{lccccccccc} \exists\, T\text{:} & d & = & d_0 & \rightarrow & d_1 & \rightarrow & \ldots & \rightarrow & d_T \\ & & & \downarrow & & \downarrow & & & & \downarrow \\ \text{definiere:} & & & cc_0 & & cc_1 & & & & cc_T \end{array} \quad = \quad d \quad = \quad cc'$$

Fall 1: $store(d_i) \wedge ea(d_i) = abase(c)$
(x gibt es durch die Restriktion) falls $ea(d_i) \in Data(K)$
$\Rightarrow va(cc^{i+1}, x) = d.GPR[RS1(d_i)]$

Fall 2: sonst
$cci + 1 = cc^i$

$cc.pr = m;\ r$ (im Assembler-Makro)
$\Rightarrow cc'.pr = body(m);\ r$
($body(m)$ aus Makro, Definition: Syntax con C_{0A})

7.4.3 Datenstrukturen des konkreten Kerns K

- alle vom abstrakten Kern k
- globale Variable act (aktueller Prozess)
- Array von processed control Blöcken
  $pcb(p), p \in [1 : np - 1]$

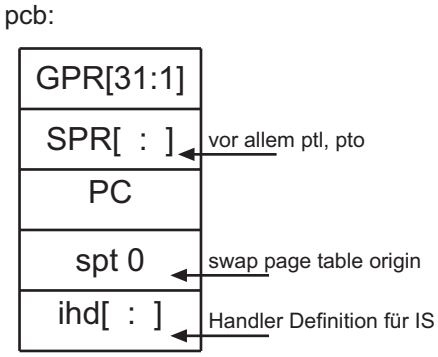


Abbildung 56: Processed Control Block

- array $hd[\ :\ ]$: Handler-Definitionen für $il \in ES$
- page-table-array
  (insgesamt: $4 \times tvm \,/\, 4K = 4 \times 4\,GB \,/\, 4\,k = 4\,MB$)

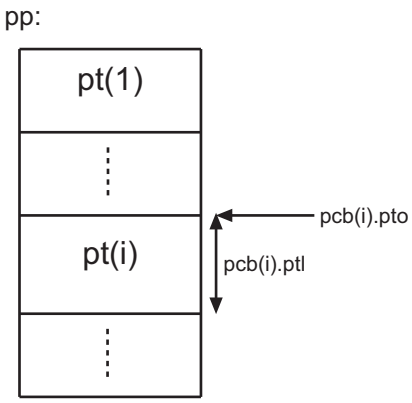


Abbildung 57: Page Table Array

- swap memory page-table:
  sm-pages: 1 MB Größe = $256 \times 4Kpages$
  $va = (\underbrace{spx(va)}_{12\ bit}, \underbrace{sbx(va)}_{20\ bit})$

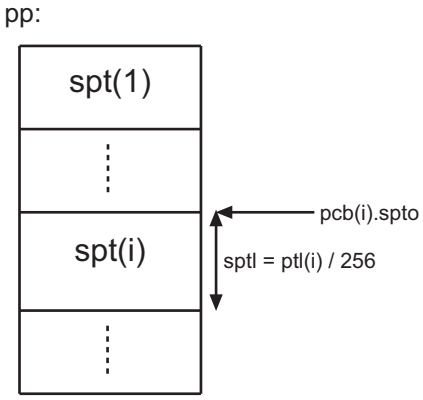


Abbildung 58: Swap Page

- free list: verlinkte Liste der freien sm-pages

x: g-Variable, $va(c, x)$, c: C_0-Konfiguration
$val(e, x)$ e : physikalische DLX_0-Konfiguration

x global, elementar:

$$val(e, x) = e.b[abase(x)]$$

$e \rightarrow vm(e, p)$: die von e codierte p-te user-Konfiguration

Komponenten: $vm(e, p).R$ CPU-Register (GPR, SPR, PC)
$vm(e, p).m$ *virtueller* Speicher

$$vm(e,p).R \quad \begin{cases} e.r: & p \text{ „läuft“ in Konfiguration } e \\ va(e,\ pcb(p).R): & \text{sonst} \end{cases}$$

p läuft in Konfiguration $e \leftrightarrow emode = 1 \wedge va(e, act) = p$

$$vm(e,p).m(va) \quad \begin{cases} e.m[pma(e,p,a)]: & valid(e,p,va) \\ e.sm[sma(e,p,va)]: & \text{sonst} \end{cases}$$

physikalischer Speicher $e.m$ ist Cache für swap memory sm.

$$\begin{aligned} pma(e,p,va) &= (ppx(e,p,va),\ bx(va)) \\ ppx(e,p,va) &= pte(e,p,va)[31:12] \\ pte(e,p,va) &= va(e, PP[val(e, pcb(p).pto] + ppx(va)) \\ valid(e,p,va) & \\ sma(e,p,va) & \end{aligned}$$ } Klausur-Aufgabe

7.5 Simulationssatz

gegeben: C_0: Startkonfiguration von CVM
$in^1,\ in^2, \ldots$ (Inputs für physikalische Maschine)

Behauptung:
$\exists\, c^0,\ c^1,\ c^2, \ldots$ CVM-Rechnung
$\exists\, cc^0,\ cc^1,\ cc^2, \ldots$ Folge von C_0-Konfiguration für K
$\exists\, d^0,\ d^1,\ d^2, \ldots$ DLX_0-Rechnung (physikalische Maschine)
$\exists\, s(0),\ s(1),\ s(2), \ldots$ } Schrittzahlen
$\exists\, t(0),\ t(1),\ t(2), \ldots$ }
$\exists\, abase^0,\ abase^1,\ abase^2, \ldots$ (bekannt!) } $allocation$
$\exists\, kbase^0,\ kbase^1,\ kbase^2, \ldots$ (bisher unbekannt!) }

$\forall\, i:\ \underbrace{kconsis}_{\text{ebenfalls neu!}}\ (c^i.C,\ kbase^i,\ cc^{s(i)})$
nach $s(i)$ Schritten von K sind i Schritte von CVM bezüglich k simuliert

$\forall\, i : consis(cc^j,\ abase^j,\ d^{t(j)})$
nach $t(j)$ Schritten von DLX_0 sind j Schritte von K simuliert

(noch zu definieren: $kconsis,\ kbase$)

$\forall\, i,\ \forall\, p:\ c^i.vm(p)\ =\ vm(d^{t(s(i))},\ p)$

$\forall\, j:\ cc^{j+1}\ =\ \delta_A(cc^j,\ d^{t(j)})$

„Kindereien“: $c^i.ptl(p)\ =\ va(d^{t(s(i))},\ pcb(p).ptl)$
ihd } Bonus-Aufgabe, Klausur
ha }

7.5.1 Definition: $konsis(cc,\ kbase,\ c)$

$$konsis(cc,\ kbase,\ c) \quad \leftrightarrow \quad \begin{array}{l} e-konsis(cc,\ kbase,\ c) \\ \wedge\ p-konsis(cc,\ kbase,\ c) \\ \wedge\ c-konsis(cc,\ c) \end{array}$$

$$Var(c)\ =\ \left\{ x \,\middle|\, X\ \underbrace{\text{Variable}}_{(m,i)}\ \text{von Konfiguration } c \right\}$$

$kbase$: $Var(c)\ \rightarrow\ Var(cc)$
$kbase(m,i)\ =\ (m,i)$ (gleiches m)

(m,j) von cc kodiert, (m,i) von c

7.5.2 Definition: $e-konsis(c,\ kbase,\ cc)$

Sei $(m,i)s$ elementare g-Variable von c:

$$\Rightarrow\ \boxed{va(c,\ (m,i)s)\ =\ va(cc,\ (kbase(m,i))s)}$$

7.5.3 Definition: $p - konsis(c,\ kbase,\ cc)$

Der heap von c ist isomorth zu einem subgraph von cc:

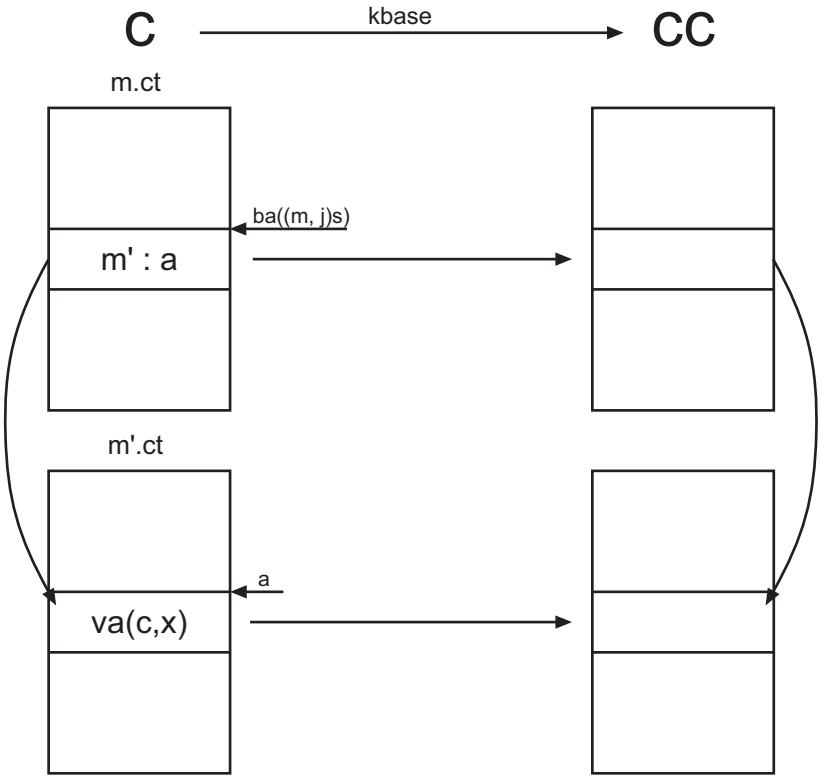


Abbildung 59: $p - konsis(c,\ kbase,\ cc)$

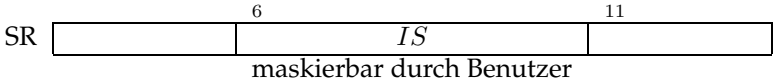
$$
\begin{array}{rcl}
\underbrace{type((m,j)s)}_{c} & = & t* \\
va(c,\ (m,j)s) & = & m' : a \\
x & = & gv(c,\ m' : a,\ t) \\
\\
va(cc,\ (kalloc(m,j))s) & = & ba(cc, x) \\
 & = & ba(cc,\ gv(c,\ m' : a,\ t)) \\
\boxed{va(cc,\ (kalloc(m,j))s)} & = & \boxed{ba(cc,\ gv(c,(m,j)s),\ t)}
\end{array}
$$

7.5.4 Definition: $c - konsis(c,\ cc)$

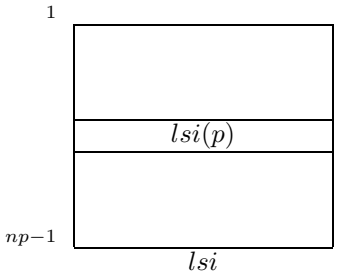
später!

7.6 Korrekturen

1.
c^0 c^1 c^2 | CVM
cc^0 cc^1 cc^2 | K
d^0 d^1 d^2 | physikalische DLX_0
$konsis(c^i.C,\ kbase^i,\ cc^{s(i)})$
$c - konsis(c^i.C,\ abase^i,\ d^{t(i)})$

2.


CVM-Funktion in K: $SR[11:6] = \{0,1\}^6$

3.
C_0-Datenstruktur für K


$lsi(p)$: last page swapped in for process p

page fault handler ($pff,\ pfls$) swappen $lsi(p)$ *nicht* raus

Grund: Terminierung bei zwei Page-Faults in einer Intruktion

4.
$c \rightarrow c'$: $c.cp = p$
$JISR(c.vm(p))$
$il = (c.vm(p)) \in ES$

handler: $prozess\ p' = c'.cp = c.hd(il)$ (wie bisher)
$c.vm(p').R = \delta_d(c.vm(p)).R$
$R \in EHR = \{EPC,\ EDATA,\ ECAUSE\}$ (error handling register)

7.6.1 Notation: $body(\ldots,\ldots)$

$body(main, P)$: body von Funktion $main$ in Programm P

7.7 $body(main,\ k)$

1. $init_k$ (initialisieren):
   - Scheduler-DS (Datenstrukturen)
   - create $root$ $\underbrace{\text{(Haupt-Bestriebssystem-Prozess)}}_{\text{Prozess 1}}$

     code, data (laden)

     platz, zeit (zuweisen)
2. $act\ =\ 1$
3. $\underbrace{\text{while 1 do } startnextu}_{loop_a}$

7.8 $body(main,\ K)$

1. $init_K$ (initialisieren):
   - Datenstrukturen für memory management
   - leere Benutzerprozesse
2. $init_k$
3. $act\ =\ 1$
4. $\underbrace{\text{while 1 do } \underbrace{\{startnextu,\ dispatch\}}_{\text{Makro's}}}_{loop_c}$

7.9 Definition: $c - consis(c,\ cc)$

$$c - consis(c,\ cc) \quad \leftrightarrow \quad \underbrace{c.pr\ =\ apr(cc.pr)}_{\text{aktrakter } pr}$$

$cc.pr$	$apr(cc.pr)$
r; $loop_c$ ($disp$, $startnextu$ nicht in r)	r; $loop_a$
$startnextu$; $disp$; $loop_c$ $disp$; $loop_c$	$startnextu$; $loop_a$ $loop_a$

Bemerkung: apr streicht $disp$

7.10 Bootstrap

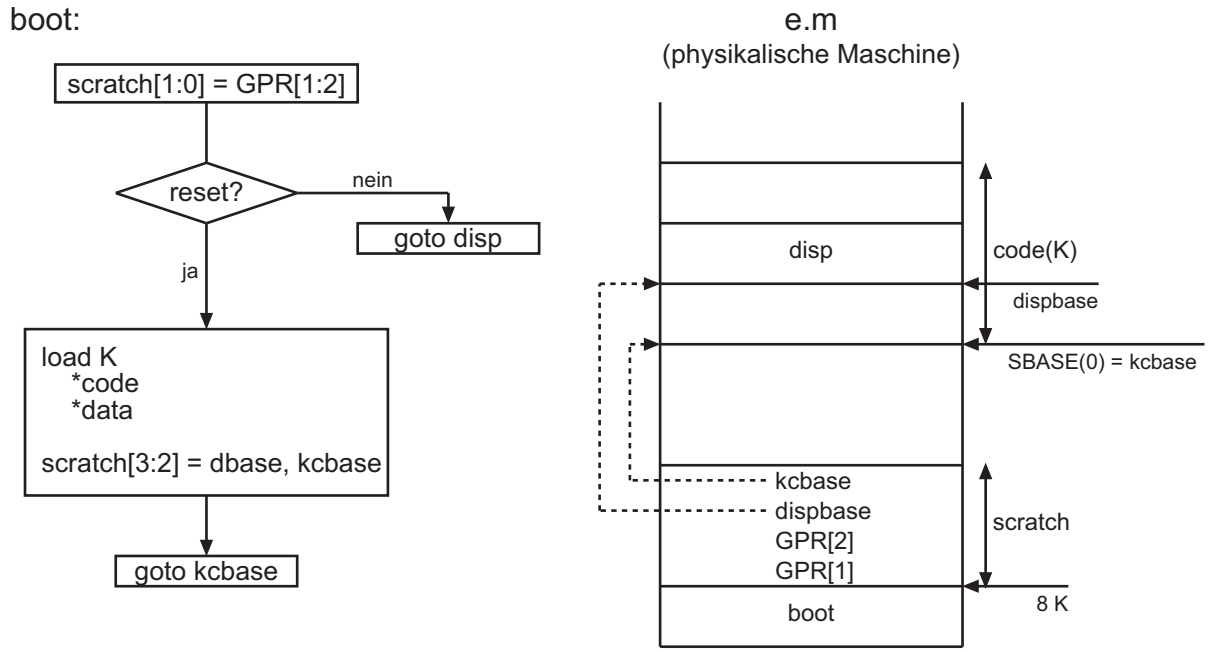


Abbildung 60: Bootstrap

Bemerkung:
Nach $boot$ und $init_k$ gilt der Simulationssatz für C_0 (Induktions-Anfang)

startnextu:
$pcb(act).R$ (CPU-Register restaurieren, $R \neq PC$)
$e.EPC = pcb(act).EPC$
rfe

disp:
1) CPU-Register in $pcb(act)$ retten:
$GPR[2:1]$ aus $scratch[1:0]$
alle anderen aus CPU
} ($asm(\ldots)$)
2) il berechnen:
$min\ \{pcb(act).ECA[j] = 1\}$ $(il \neq 0)$

Fälle:

a) $il = 5\ (trap)$
$i =< pcb(act).EDATA >\ \ (kcall\ i)$

Sei $n = c.C.fd(kcall(i)).name$
(Name d. C_0-Funktion von k, Handler von trap i)

$npar = (\quad).npar \leq 10\ (\#Parameter)$

Aufruf:
$n(pcb(act),\ GPR[npar : 1])$

(!) $JISR$ ist implementiert als Funktionsaufruf

b) $il = 3\ (pff)$
$pfh(act,\ EPC)$ (page fault handler)

c) $il = 4\ (pfls)$
$pfh(act,\ EDATA)$

d) $il = 12\ (timer)$
call scheduler
Bemerkung:

- brauche I/O Port um aktuelle Zeit auszulesen
- ändert oft act

e) $il \in ES$ Nächster Prozess: $hd(il)$
$\underbrace{pcb(hd(il)).R}_{R \in EHR} = pcb(act).R$
(Korrektur 4, Paramterübergabe)
$R \neq EPC$

$pcb(hd(il)).EPC = 0^{32}$ (Anfang von Code des Prozesses $hd(il)$)
$act = hd(il)$

f) $il \in IS = [11 : 6]$
$pcb(act).EPC = ihd(act,\ il)$

Abbildungsverzeichnis