Vorlesungsskript Systemarchitektur SS05

Prof. Dr. Wolfgang J. Paul
Dipl.-Ing. Dominik Rester

Universität Saarbrücken
Computer Science Department

27. Juli 2005

Inhaltsverzeichnis

Kapitel 1

Prozessoren

1.1 Boole'sche Ausdrücke

1.2 Schaltkreise

1.3 Elementare Rechnerarithmetik

1.4 ALU - arithmetic logic unit

1.5 Spezifikation eines einfachen Prozessors

In diesem Kapitel definieren wir einen Instruktionssatz und führen mathematische Maschinen ein. Eine mathematische Maschine ist ein Tupel (C, δ, c_0).

C: Menge von Konfigurationen (Konfiguration: Schnappschuß der Maschine nach einem Schritt)
$c_0 \in C$
δ $C \to C$, $\delta(c) = c'$, c' entsteht aus c durch einen Schritt der Maschine

Eine Rechnung ist eine (endliche oder unendliche) Folge von Konfigurationen:

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

Wobei c^0 die Startkonfiguration ist und es gilt $\forall i \geq 0 : c^{i+1} = \delta(c^i)$

1.5.1 Spezifikation des DLX Prozessors (vereinfacht)

Konfiguration: C = (c.pc, c.gpr, c.m)
pc: Program Counter
gpr: General Purpose Register
m : Memory (Speicher), byte addressiert

c.pc $\in \{0,1\}^{32}$
c.gpr: $\{0,1\}^5 \to \{0,1\}^{32}$, gpr(x) = aktueller Inhalt von gpr Nummer x.
c.m $\in \{0,1\}^{32} \to \{0,1\}^8$, m(y) = aktueller Inhalt von Speicherzelle y, y: Adresse

Speicherzugriff

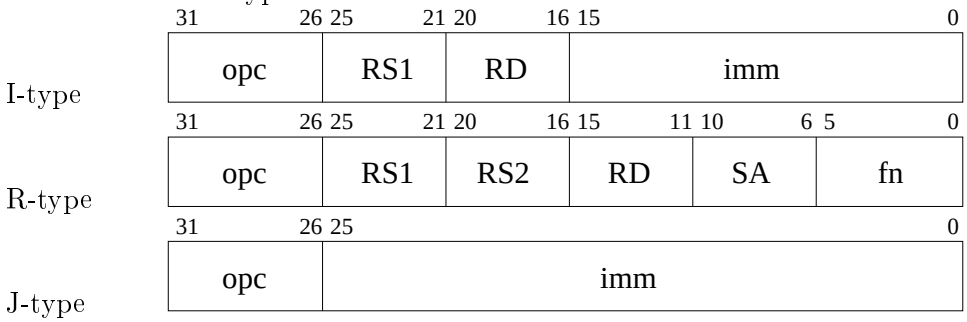
Wir benutzen die 'little endian' Codierung, d.h. bei einem Speicherzugriff der Breite d > 1 wird das niedrigste Byte adressiert:

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

$1_{32} = 0 \ldots 01 = 0^{31}1$
$+_{32} : \{0,1\}^{32} \times \{0,1\}^{32} \to \{0,1\}^{32}$ (Addition modulo 2^{32})
$-_{32} : \{0,1\}^{32} \times \{0,1\}^{32} \to \{0,1\}^{32}$
$x_{32} : y \in \{0,1\}^{32}$ mit $< y >= x$

Instruktionssatz

Instruktion $\in \{0,1\}^{32}$
Wir unterscheiden 3 Typen von Instruktionen:



Abkürzungen:

$I(c)$: Instruktion, die in Konfiguration c ausgeführt wird: $I(c) = c.m_4(c.pc)$
opc : Op-Code: opc(c) = I(c)[31:26]

$R - Type(c) \leftrightarrow opc = 0^6$
$J - Type(c) \leftrightarrow \overline{opc \in \{000010, 000011, 111110, 111111\}}$
$I - Type(c) \leftrightarrow \overline{(R - Type(c) \vee J - Type(c))}$

imm : immediate Konstante :

$$imm(c) = \begin{cases} I(c)[25:0] & : \ J - Type = 1 \\ I(c)[15:0] & : \ sonst \end{cases}$$

sxtimm(c): sign-extended immediate Konstante:

$$sxtimm(c) = \begin{cases} I(c)_{25}^{6} I(c)[25:0] & : \ J - Type = 1 \\ I(c)_{15}^{16} I(c)[15:0] & : \ sonst \end{cases}$$

Lemma: $[sxtimm(c)] = [imm(c)]$

$RS1$: Register Source 1: RS1(c) = I(c)[25:21]

$RS2$: Register Source 2: RS2(c) = I(c)[20:16]
RD : Register Destination:

$$RD(c) = \begin{cases} I(c)[15:11] & : \ R-Type = 1 \\ I(c)[20:16] & : \ sonst \end{cases}$$

SA : (Shift Amount) oder special purpose register address
f_u : Function Code : $f_u(c)$ = I(c)[5:0]
$aluf$: ALU funktion, steuert ALU Operation :

$$aluf(c) = \begin{cases} f_u(c)[3:0] & : \ R-Type = 1 \\ I(c)[29:26] & : \ sonst \end{cases}$$

In der folgenden Übersicht aller Instruktionen schreiben wir nicht jedesmal die Abhängigkeit von der Konfiguration c und anstatt $gpr(RD) = x$, schreiben wir $RD = x$.

I-type Instruktionen

I[31:26]	mnemonic	Effekt
load/store		
100011	lw	RD = m_4(RS1 + sxtimm)
101011	sw	m_4(RS1 + sxtimm) = RD
01****	comp.imm	RD = aluop(RS1, sxtimm, I[29:26])
control		
110100	beqz	PC = PC + (RS1 = 0 ? sxtimm:4)
110101	bnez	PC = PC + (RS1 $\neq$ 0 ? sxtimm:4)
110110	jr	PC = RS1
110111	jalr	R31 = PC+4 ; PC = RS1

R-type Instruktionen

I[31:26]	I[5:0]	mnemonic	Effekt
000000	00****	compute	RD = aluop(RS1, RS2, f_u)
special moves			
000000	010000	movs2i	RD = SA (SPR[SA] $\rightarrow$ GPR[RS1])
000000	010001	movi2s	SA = RS1 (GPR[RD] $\rightarrow$ SPR[SA])

J-type Instruktionen

I[31:26]	mnemonic	Effekt
control		
000010	j	PC = PC + sxtimm
000011	jal	R31=PC+4 ; PC = PC + sxtimm
111110	trap	ESR = SR, ECA = 5, EPC = PC + 4, EDATA = sxtimm, PC = 0
111111	rfe	SR = ESR, PC = EPC, MODE = 1

I-type comp.imm

I[31:26]	mnemonic	Effekt
arrithmetische und logische Operationen		
01 0***	comp.imm	RD = aluop(RS1, sxtimm, I[29:26])
01 0000	addio	RD = RS1 + sxtimm
01 0001	addi	RD = RS1 + sxtimm (no overflow)
01 0010	subio	RD = RS1 - sxtimm
01 0011	subi	RD = RS1 - sxtimm (no overflow)
01 0100	andi	RD = RS1 $\wedge$ sxtimm
01 0101	ori	RD = RS1 $\vee$ sxtimm
01 0110	xori	RD = RS1 $\oplus$ sxtimm
01 0111	lhgi	RD = imm 0^{16}
Test und Set Operationen		
01 1***	Test Set	RD = xcc(RS1, sxtimm, I[29:26)
01 1000	clri	RD = 0^{32};
01 1001	sgri	RD = (RS1 > sxtimm ? $0^{31}1$: 0^{32});
01 1010	seqi	RD = (RS1 = sxtimm ? $0^{31}1$: 0^{32});
01 1011	sgei	RD = (RS1 $\geq$ sxtimm ? $0^{31}1$: 0^{32});
01 1100	slsi	RD = (RS1 < sxtimm ? $0^{31}1$: 0^{32});
01 1101	snei	RD = (RS1 $\neq$ sxtimm ? $0^{31}1$: 0^{32});
01 1110	slei	RD = (RS1 $\leq$ sxtimm ? $0^{31}1$: 0^{32});
01 1111	seti	RD = 1^{32};

R-type compute

I[31:26]	I[5:0]	mnemonic	Effekt
000000	00 ****	compute	RD = aluop(RS1, RS2, f_u)
arithmetische und logische Operationen			
000000	00 0000	addo	RD = RS1 + RS2
000000	00 0001	add	RD = RS1 + RS2 (no overflow)
000000	00 0010	subo	RD = RS1 - RS2
000000	00 0011	sub	RD = RS1 - RS2 (no overflow)
000000	00 0100	and	RD = RS1 $\wedge$ RS2
000000	00 0101	or	RD = RS1 $\vee$ RS2
000000	00 0110	xor	RD = RS1 $\oplus$ RS2
000000	00 0111	lhg	RD = RS2[15:0] 0^{16}
Test Set Operationen			
000000	00 1000	clr	RD = 0^{32};
000000	00 1001	sgr	RD = (RS1 > RS2 ? $0^{31}1$: 0^{32});
000000	00 1010	seq	RD = (RS1 = RS2 ? $0^{31}1$: 0^{32});
000000	00 1011	sge	RD = (RS1 $\geq$ RS2 ? $0^{31}1$: 0^{32});
000000	00 1100	sls	RD = (RS1 < RS2 ? $0^{31}1$: 0^{32});
000000	00 1101	sne	RD = (RS1 $\neq$ RS2 ? $0^{31}1$: 0^{32});
000000	00 1110	sle	RD = (RS1 $\leq$ RS2 ? $0^{31}1$: 0^{32});
000000	00 1111	set	RD = 1^{32};

Prädikate

$lw(c) = 1 \iff opc(c) = 100011$
BA: lw(c) = $I_{31}(c) \wedge \overline{I_{30}(c)} \wedge \overline{I_{29}(c)} \wedge \overline{I_{28}(c)} \wedge I_{27}(c) \wedge I_{26}(c)$
...
$compimm(c) = 1 \iff I[31:30](c) = 01$

Sei $\delta(c) = c'$ und effective address $ea(c) = c.gpr(RS1(c)) +_{32} sxtimm(c)$
Speichersemantik:

$$lw(c) \Longrightarrow c'.gpr(x) = \begin{cases} c.m_4(ea(c)) & : x = RD(c) \wedge x \neq 0^5 \\ c.gpr(x) & : sonst \end{cases}$$

$\boxed{c.gpr(0^5) = 32}$

$$c'.pc = c.pc +_{32} 4_{32}$$

$\overline{sw(c)} \Longrightarrow c'.m = c.m$ (Inhalt des Speichers kann nur durch storeword Instruktionen geändert werden)

$$sw(c) \Longrightarrow c'.gpr = c.gpr \wedge \\ c'.m_4(ea(c)) = c.gpr(RD(c)) \wedge \\ c'.m(x) = c.m(x)\ x \notin \{ea(c), ea(c) +_{32} 1_{32}, \ldots, ea(c) +_{32} 3_{32}\}$$

$compimm(c) = 1 \Longleftrightarrow I(c)[31:30] = 01$
$comp(c) = 1 \Longleftrightarrow R - type(c) \wedge I(c)[5:4] = 00$

Linker Operand:

$$lop(c) = c.gpr(RS1(c))$$

Rechter Operand:

$$rop(c) = \begin{cases} c.gpr(RS2(c)) & : R - type(c) \\ sxtimm(c) & : sonst \end{cases}$$

ALU Funktion:

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

$comp(c) \vee compimm(c) \Longrightarrow$

$$c.gpr(x) = \begin{cases} aluop(lop(c), rop(c), aluf(c)) & : x = RD(c) \wedge x \neq 0^5 \\ c.gpr(x) & : sonst \end{cases}$$

Kontroll Operationen:

beqz	branch equal zero
bnez	branch not equal zero
jr	jump register
jalr	jump and link register
j	jump
jal	jump and link

$control(c) = 1 \Longleftrightarrow I(c)[31:28] = 1101 \vee I(c)[31:27] = 00001$

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

$jump(c) = jr(c) \vee jalr(c) \vee j(c) \vee jal(c)$
$branch(c) = beqz(c) \vee bnez(c)$

Genommener Branch : branch taken:

$$btaken(c) = beqz(c) \wedge c.gpr(RS1(c)) = 0^{32} \vee bnez(c) \wedge c.gpr(RS1(c)) \neq 0^{32}$$

$Aeqz(c) = 1 \Longleftrightarrow c.gpr(RS1(c)) = 0^{32}$

$$btaken(c) = branch(c) \wedge (\overline{I_{26}(c)} \wedge Aeqz(c) \vee I_{26}(c) \wedge \overline{Aeqz(c)}) = branch(c)(\wedge I_{26}(c) \oplus Aeqz)$$

$bjtaken(c) = jump(c) \vee btaken(c)$

Branch (oder Jump) Taget:

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

$control(c) \Longrightarrow$

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

Für Funktionsaufrufe wird die Rücksprungadresse (pc +4) in Register 31 gesichert:

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

Spezifikation der Maschine

Speicher:

$c'.m_4(ea(c)) = c.gpr(\overline{RD(c)}) : sw(c)$
$c'.m(x) = c.m(x) : \overline{sw(c)} \vee sw(c) \wedge x \notin \{ea(c), \ldots, ea(c) +_{32} 3_{32}\}$

PC:

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

General Purpose Register:

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

1.6 Konstruktion eines einfachen Prozessors

In diesem Kapitel konstruieren wir einen einfachen Prozessor und zeigen, daß er den spezifizierten Prozessor simulieren kann. Zunächst benötigen wir einige weitere Bauteile, die Bestandteil der Hardware Konfiguration sein werden.
Hardware Konfiguration h = (...,h.R ,..., h.S, ...)

Register

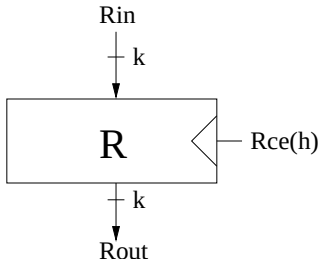


Abbildung 1: k-Bit Register

Rce(h): clock enable von Register R, wird durch einen Schaltkreis berechnet.
Rout(h) = h.R

$$h'.R = \begin{cases} Rin(h) & : Rce(h) = 1 \\ h.R & : sonst \end{cases}$$

Schaltkreis zur Berechnung der ce Signale

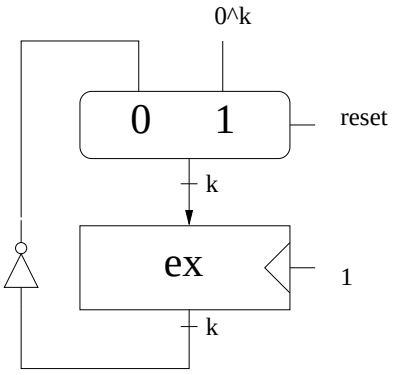


Abbildung 2: Berechnung des ex Signals

X:Signal
X^t: Wert von X im Takt t
Für den Prozessor soll gelten: $reset^{-1} = 1$ und $reset^t = 0$ für t = 0, 1, 2, 3, ...

Sei: $h' = \delta_h(h)$ und $h^{t+1} = \delta_h(h^t)$
Es gilt:
$h^0.ex = 0$

$h^1.ex = 1$
$h^2.ex = 0$
und offensichtlich:

$$\boxed{h^t.ex = t \; mod \; 2 \text{ für } t \geq 0}$$

RAM

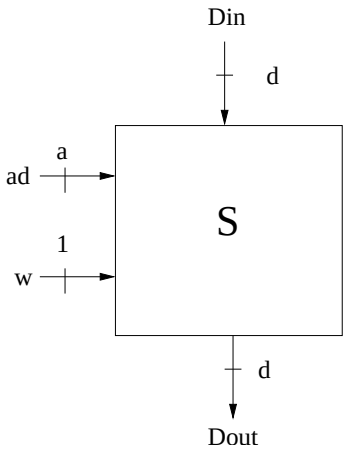


Abbildung 3: $2^a \times d$ RAM

Dout(h) = h.S(ad(h)), für w(h) = 0 (Lesezyklus)

$$h'.S(x) = \begin{cases} Din(h) & : \; w(h) = 1 \wedge x = ad(h)(Schreibzyklus) \\ h.S(x) & : \; sonst \end{cases}$$

3-port RAM

Bei einem 3-port RAM kann man gleichzeitig zwei Adressen auslesen und eine dritte beschreiben. Die Leseadressen bezeichnen wir mit adA und adB, die Schreibadresse mit adC.

DoutA(h) = h.S(adA(h)), für $w(h) = 0 \vee w(h) = 1 \wedge adA(h) \neq adC(h)$
DoutB(h) = h.S(adB(h)), für $w(h) = 0 \vee w(h) = 1 \wedge adB(h) \neq adC(h)$

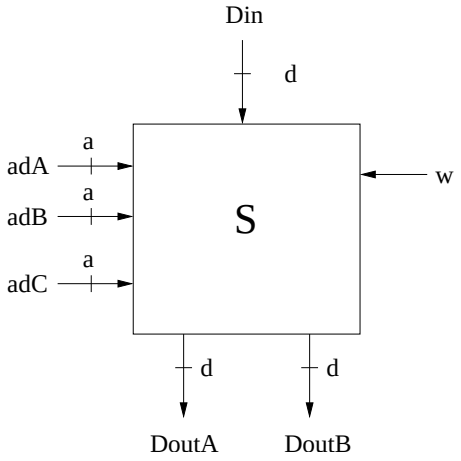


Abbildung 4: $2^a \times d$ 3-port RAM

$$h'.S(x) = \begin{cases} Din(h) & : \ w(h) = 1 \wedge x = adC(h)(Schreibzyklus) \\ h.S(x) & : \ sonst \end{cases}$$

Übersicht einfacher Prozessor

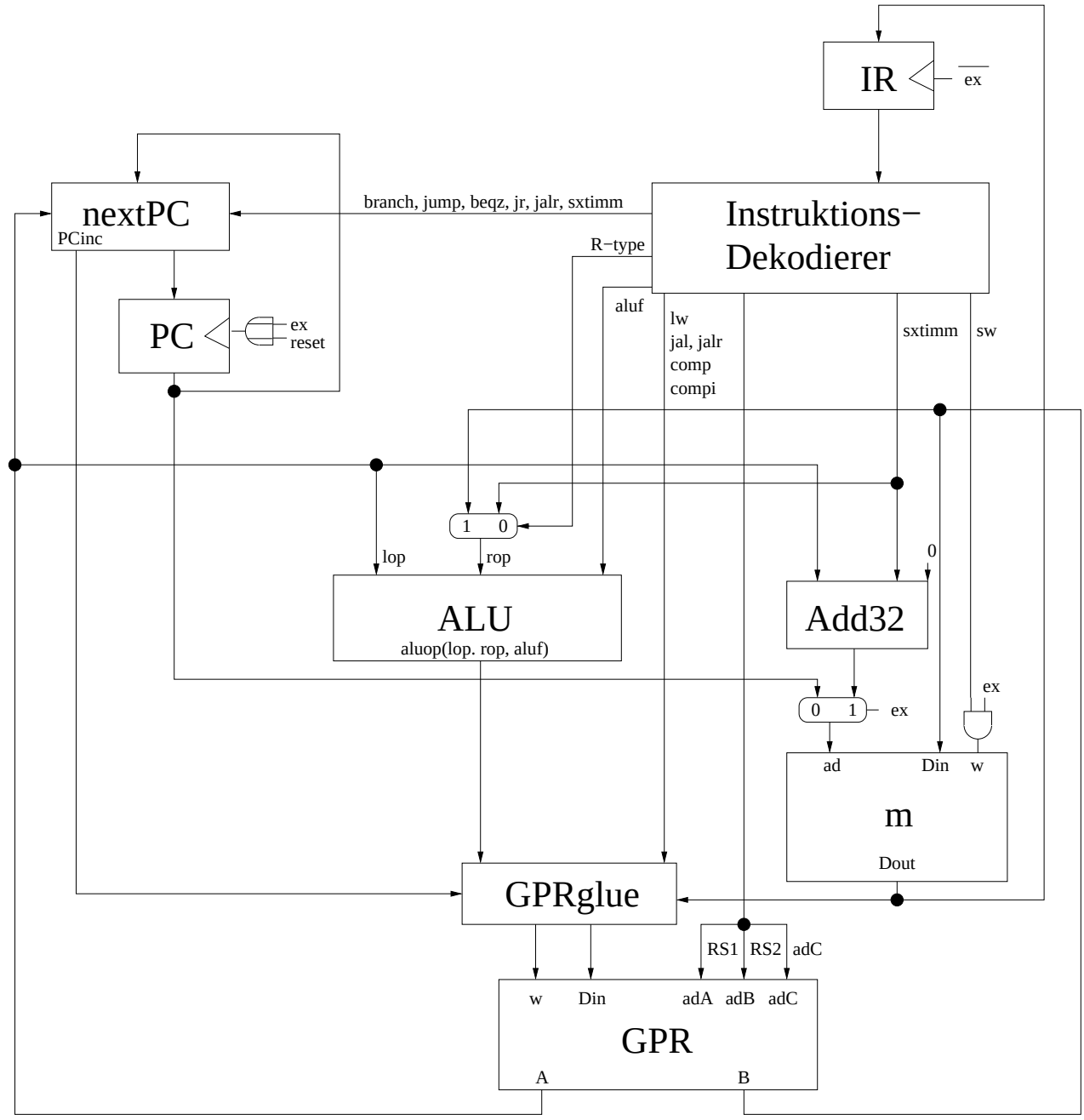


Abbildung 5: Funktionseinheiten und Datenpfade des DLX Prozessors

1.6.1 Hardware Konfiguration

h = (h.IR, h.GPR, h.m, h.PC, h.ex)

h.IR = Instruction Register
h.GPR = $2^5 \times 32$ 3-port RAM mit GPR(0^5) = 0^{32}
h.m : $\{0,1\}^{30} \longrightarrow \{0,1\}^{32}$ Achtung: unterscheide c.m:$\{0,1\}^{32} \longrightarrow \{0,1\}^8$

Wir setzen voraus, daß alle Adressen durch 4 teilbar sind. Diese Forderung nennen wir Alignment. Später (TODO Interrupt Kapitel) werden wir ein Konzept vorstellen, daß diese Eigenschaft sicherstellt.

$$< ea(c) > = 0 \bmod 4 \iff \text{ea(c)[1:0]} = 00$$
$$< c.pc > = 0 \bmod 4 \iff \text{c.pc[1:0]} = 00$$

1.6.2 Simulationssatz

Ziel: Zeige, daß die Hardware die spezifizierte Maschine simuliert.

Voraussetzung:

$reset^{-1} = 1, \forall t \geq 0 : reset^t = 0$
$c^0.pc = 0^{32}$
$c^0.gpr = h^0.GPR$
$\forall x \in \{0,1\}^{30} : c^0.m_4(x00) = h^0.m(x)$

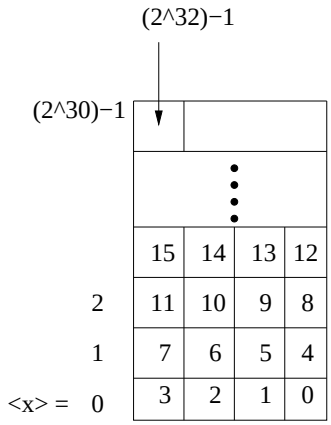


Abbildung 6: Zusammenhang zwischen *byte* und *word* Adressierung

Behauptung: $\forall i \geq 0$:

h^{2i} codiert c^i (Instruktion benötigt 2 Takte)

$c^i.pc = h^{2i}.PC$	a.)
$c^i.gpr = h^{2i}.GPR$	b.)
$c^i.m_4(x00) = h^{2i}.m(x)$	c.)

Instruktions Register IR

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

Beweis: (Induktion über i)

$h^{2i}.PC$	=	$c^i.pc$	(Induktions Voraussetzung)
$h^{2i}.ex$	=	0	(da 2i mod 2 = 0)
$ad(h^{2i})$	=	$h^{2i}.PC[31:2]$	(ex = 0)
	=	$c^i.pc[31:2]$	
$Dout(h^{2i})$	=	$h^{2i}.m(c^i.pc[31:2])$	(Konstruktion)
	=	$c^i.m_4(c^i.pc[31:2]00)$	
	=	$c^i.m_4(c^i.pc)$	(Alignment)
	=	$I(c^i)$	
$h^{2i+1}.IR$	=	$Dout^{2i}(h)$	(Konstruktion)
	=	$I(c^i)$	q.e.d.

Instruktions Decoder

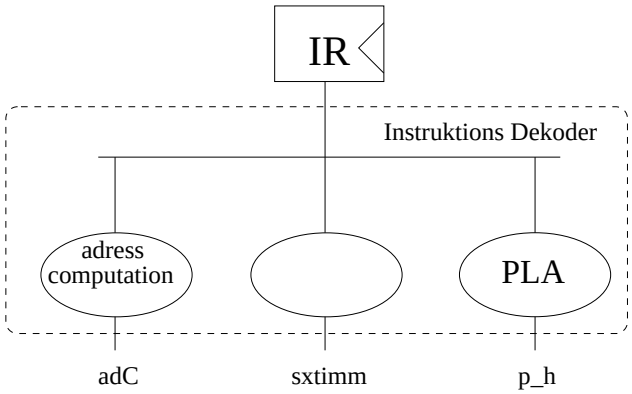


Abbildung 7: Instruktionsdekoder

p_h : Hardware Prädikate erhält man, indem man in den Prädikaten der Spezifikation I(c) durch h.IR ersetzt.
p_f : Hardware Funktionen ebenso.

Lemma:
$p_h(h^{2i+1}) = p(c^i)$
$p_f(h^{2i+1}) = f(c^i)$

Address Computation:

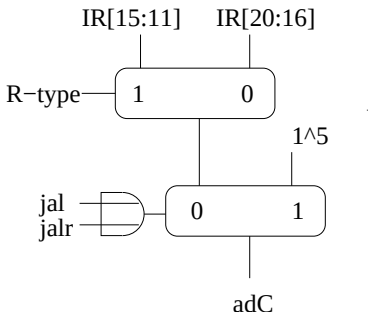


Abbildung 8: Berechnung der Schreibadresse für das GPR

$\underline{jal(c^i) \vee jalr(c^i)} \Longrightarrow adC(h^{2i+1}) = 1^5$
$jal(c^i) \vee jalr(c^i) \Longrightarrow adC(h^{2i+1}) = RD(c^i)$

ALU Korrektheit:

$$\begin{array}{rcll} lop(h^{2i+1}) & = & A(h^{2i+1}) & \\ & = & h^{2i+1} \cdot gpr(RS1_h(h^{2i+1})) & \\ & = & h^{2i} \cdot gpr(RS1(c^i)) & (1) \\ & = & c^i.gpr(RS1(c^i)) & \\ & = & lop(c^i) & \end{array}$$

(1) gilt, da:

$$\begin{array}{rcl} GPRw & = & ex \wedge \ldots \\ \Longrightarrow \quad GPRw(h^{2i}) & = & (\text{2i mod 2}) \wedge \ldots \\ & = & 0 \\ \Longrightarrow \quad h^{2i+1}.GPR & = & h^{2i}.GPR \end{array}$$

$B(h^{2i+1}) == c^i.gpr(RS2(c^i))$

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

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

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

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

GPRglue:

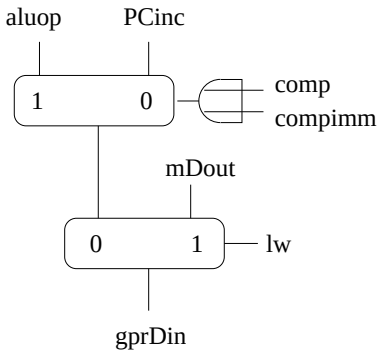


Abbildung 9: Selektierung der Eingangsdaten für das GPR

$$gprDin(h^{2i+1}) = \begin{cases} aluop(h^{2i+1}) & : comp(h^{2i+1}) \vee compimm(h^{2i+1}) \\ PCinc(h^{2i+1}) & : jal(h^{2i+1}) \vee jalr(h^{2i+1}) \\ mDout(h^{2i+1}) & : lw(h^{2i+1}) \end{cases}$$

$$= \begin{cases} aluop(c^i) & : comp(c^i) \vee compimm(c^i) \\ PCinc(c^i) & : jal(c^i) \vee jalr(c^i) \\ mDout(c^i) & : lw(c^i) \end{cases}$$

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

$comp(c^i) \vee compimm(c^i) \Longrightarrow GPRw(h^{2i+1}) = 1 \Longrightarrow$

$$h^{2(i+1)}.GPR(x) = \begin{cases} GPRDin(h^{2i+1}) & : x = Cad(h^{2i+1}), x \neq 0^5 \\ h^{2i+1}.GPR(x) & : \end{cases}$$

$$= \begin{cases} aluop(lop(c^i), rop(c^i), aluf(c^i)) & : x = RD(c^i) \quad (Konstruktion) \\ h^{2i+1}.GPR(x) & : \quad (GPRw(h^{2i}) = 0 \end{cases}$$

$$= \begin{cases} \ldots & : \\ c^i.gpr(c) & : \text{Induktionsvoraussetzung b.)} \end{cases}$$

$jal(c^i) \vee jalr(c^i) \Longrightarrow$ ... ebenso, benutze $PCinc(h^{2i+1}) = c^(i).PC +_{32} 4_{32}$

$lw(c^i) \Longrightarrow ...mDout = c^i.m_4(ea(c^i)[31:2]00) = c^i.m_4(ea(c^i))(alignment)$

$\overline{(comp(c^i) \vee compimm(c^i) \vee jal(c^i) \vee jalr(c^i) \vee lw(c^i))}$:

$\Longrightarrow$	$GPRw(h^{2i+1}) = 0$	(Konstruktion GPRw)
$\Longrightarrow$	$h^{2i+2}.GPR = h^{2i+1}.GPR$	(Konstruktion GPRw)
$=$	$h^{2i}.GPR$	$(ex^{2i} = 0)$
$=$	$c^i.gpr$	(Ind. Vor.)

nextPC

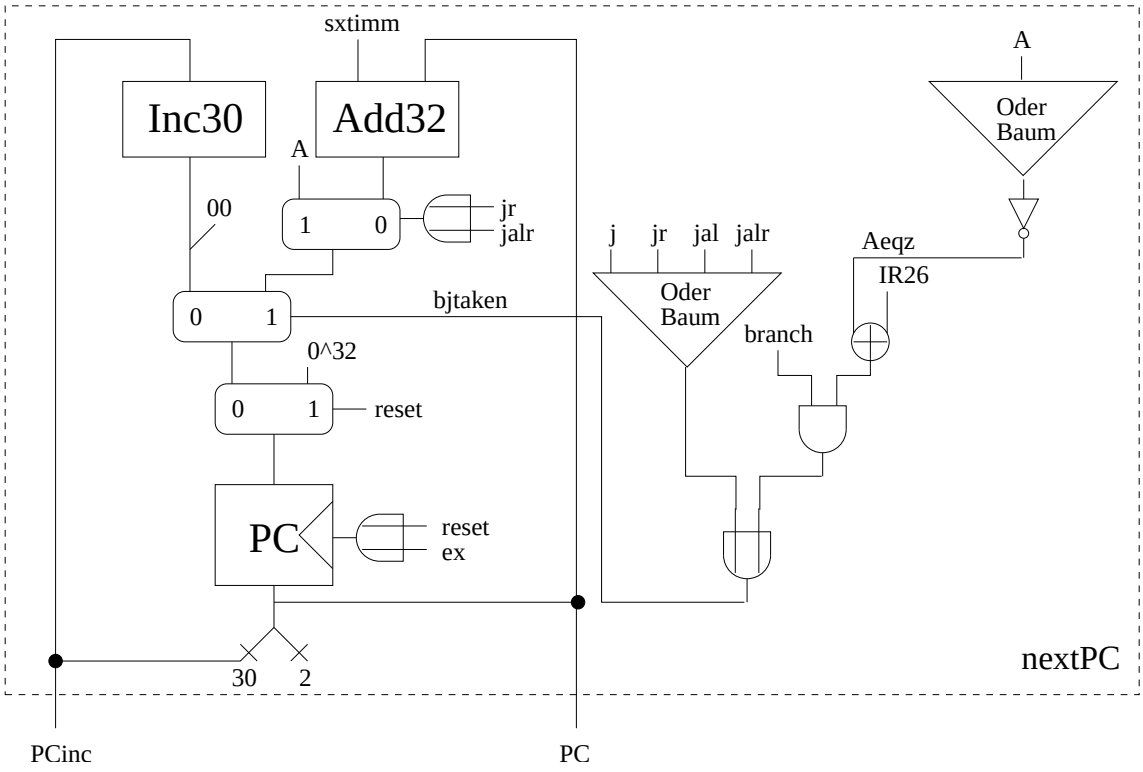


Abbildung 10: nextPC Berechnung

$bjtaken_h = j_h \vee jr_h \vee jal_h \vee jalr_h \vee branch_h \wedge (IR_{26} \oplus Aeqz_h)$
$Aeqz(h^{2i+1}) = 1 \iff c^i.gpr(RS1(c^i)) = 0$

Äquivalent zum General Purpose Register zeigt man:
$h^{2i+2}.PC = c^i.pc$ (Induktionsschritt i+1 für a.))

Speicher

Memory Write Signal: $mw = sw \wedge ex$

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

$$
\begin{array}{rrcll}
sw(c^i) \Longrightarrow & mDin(h^{2i+1}) & = & c^i.gpr(RD(c^i)) & \\
& = & & & \\
& B(h^{2i+1}) & = & h^{2i+1}.GPR(RS2_h(h^{2i+1})) & \\
& & = & h^{2i+1}.GPR(h^{2i+1}.IR[20:16]) & \\
& & = & h^{2i+1}.GPR(I(c^i[20:16])) & \\
& & = & h^{2i+1}.GPR(RD(c^i)) & \text{(wegen : sw} \Longrightarrow \text{I-Type)}
\end{array}
$$

Kapitel 2

C0 Compiler

Übersicht:

1. Grammatiken
2. Syntax von C0
3. Semantik von C0
4. Compiler C0 → DLX

2.1 Grammatiken

2.1.1 Kontextfreie Grammatiken

Definition: CFG (context free grammar)
Eine kontextfreie Grammatik G besteht aus G = (T, N, S, P)

- T: Terminal Alphabet
- N: Nichtterminal Alphabet
- S ∈ N: Startsymbol
- P $\subset N \times (N \cup T)^*$: Produktionensystem

Wobei Terminal und Nichtterminal Alphabet endliche Mengen sind.

Erklärung: X Menge → $X^* = \bigcup_{i=0}^{\infty} X^i$ (endliche Folgen aus X)
Beispiel: {0,1}* = {ε, 0, 1, 00, ..., 11, 000, ..., 111, 0000, ...}

L(G) = {w | S $\longrightarrow_G^* w \wedge w \in T^*$} ist die von G erzeugte kontextfreie Sprache.

Schreibweise:
$n \longrightarrow w_1 \mid \ldots \mid w_s$ Abkürzung für:
$n \longrightarrow w_1, \ldots n \longrightarrow w_s$
bzw:
(n, w_1) ∈ P , ..., (n,w_s) ∈ P

(n, w) $\in$ P bedeutet:
aus n kann man in einem Schritt Zeichenreihe w ableiten

Definition: Ableitungsbäume zu G

- S Ableitungsbaum mit Wurzel s, Blattwort s
- Q Ableitunhsbaum mit Wurzel m, Blattwort w = $w_1 n w_2$, n $\in \mathbb{N}$

Sei n $\longrightarrow$ w' $\in$ P $\Longrightarrow \exists$ Ableitungsbaum mit Wurzel m und Blattwort $w_1 w' w_2$

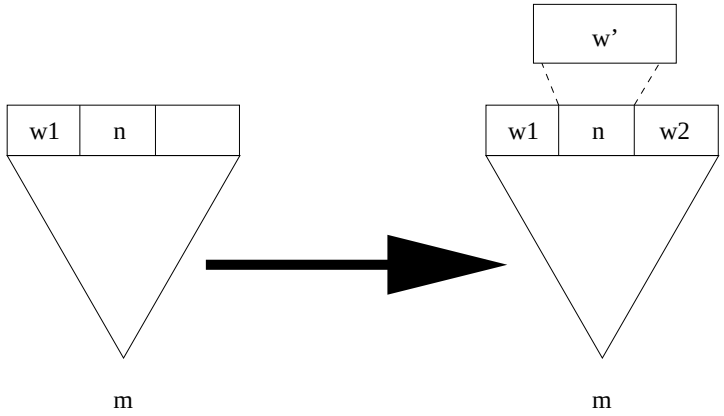


Abbildung 11: Ableitungsbaum

Beispiel:
N = {V, C, <CF>}
T = {X, 0, 1}
S = V
Erzeugen eines Ableitungsbaumes für das Wort X01:

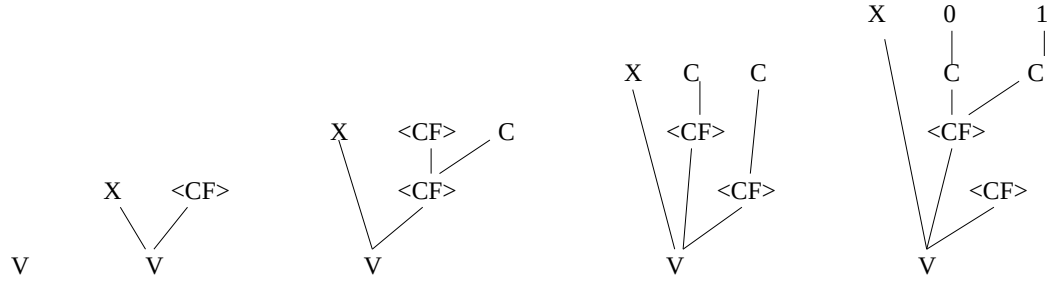


Abbildung 12: Erzeugen eines Ableitungsbaums für ein Wort aus einer gegebenen Grammatik

V ⟶ X | X<CF>
<CF> ⟶ C | <CF>C
C ⟶ 0 | 1

Schreibweise: n $\longrightarrow^*_G$ w:
Es gibt zu G einen Ableitungsbaum mit Wurzel n und Blattwort w, bzw. w ist aus n ableitbar, bzw. w ist zu n reduzierbar.

Beispiel:
Boole'sche Ausdrücke BA
Variable oder Konstante VC
Konstantenfolge CF

$$\begin{aligned} BA &\longrightarrow VC \mid (\sim BA) \mid (BA \wedge BA) \mid (BA \vee BA) \mid (BA \oplus BA) \\ C &\longrightarrow 0 \mid 1 \\ < CF > &\longrightarrow C \mid < CF > C \\ V &\longrightarrow X \mid X < CF > \\ VC &\longrightarrow V \mid C \end{aligned}$$

S = <BA>
T = {0 , 1, X, (,), $\wedge$, $\vee$, $\sim$, $\oplus$}
NT = {C , V , <CF>, <BA>
S = <BA>

2.1.2 Mehrdeutige Grammatiken

Eine Grammatik ist mehrdeutig, falls es zu einem Wort der durch die Grammatik definierten Sprache mehrere Ableitungsbäume gibt.

Beispiel: (arithmetische Ausdrücke)

$$\begin{aligned} < AA > \longrightarrow\ & < VC > | < AA > + < AA > | < AA > -_2 < AA > | < AA > \cdot < AA > | \\ & < AA > / < AA > | -_1 < AA > \end{aligned}$$

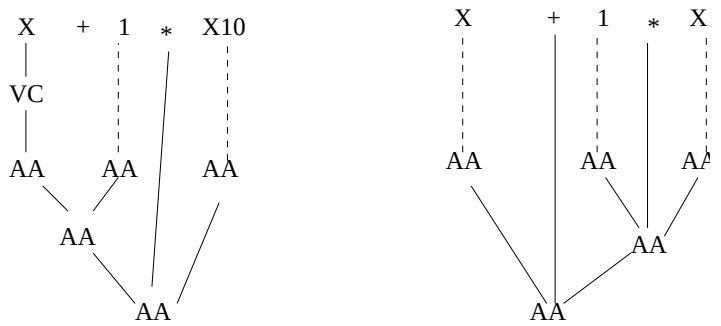


Abbildung 13: Verschiedene Ableitungsbäume zu einem Wort aus einer mehrdeutigen Grammatik

Zu dem gegebenen Wort existieren zwei Ableitungsbäume, somit ist die Grammatik nicht eindeutig

Definition:
Eine Grammatik G ist eindeutig, falls $\forall w \in L(G)$ gilt: $\exists^1$ Ableitungsbaum mit Wurzel S und Blattwort w.

Für arithmetische Ausdrücke ist die folgende Grammatik eindeutig (Satz von Loeckx, Mehlhorn, Wilhelm '86):

$< F >$	$\longrightarrow$	$< VC > \mid -_1 < F > \mid (< A >)$	Faktor
$< T >$	$\longrightarrow$	$< F > \mid < T > \cdot < F > \mid < T > / < F >$	Term
$< A >$	$\longrightarrow$	$< T > \mid < A > + < T > \mid < A > -_2 < T >$	Ausdruck

Für Boole'sche Ausdrücke können wir ebenfalls eine eindeutige Grammatik angeben:

$< BF >$	$\longrightarrow$	$< Atom > \mid \sim < BF > \mid (< BA >)$	Boole'scher Faktor
$< BT >$	$\longrightarrow$	$< BF > \mid < TT > \wedge < BF >$	Boole'scher Term
$< BA >$	$\longrightarrow$	$< BT > \mid < BA > \vee < BT >$	Boole'scher Ausdruck
$< Atom >$	$\longrightarrow$	$< A > = < A > \mid < A > \neq < A > \mid .. \geq .. \mid .. > .. \mid .. \leq .. \mid .. < .. \mid 0 \mid 1$	

Grammatik für Anweisungen:
$< An >$: Anweisung
$< id >$: identifier

$< An >$	$\longrightarrow$	$< id > = < A > \mid$	Zuweisung
	$\longrightarrow$	$< id > = < BA > \mid$	Boole'sche Zuweisung
	$\longrightarrow$	$if < BA > then < AnF > \mid$	
	$\longrightarrow$	$if < BA > then < AnF > else < AnF > \mid$	bedingte Anweisung
	$\longrightarrow$	$id = Na(AnF) \mid$	Funktionsaufruf
	$\longrightarrow$	$returnA \mid$	
	$\longrightarrow$	$< AnF >$	
$< AnF >$	$\longrightarrow$	$< An > \mid < AnF > ; < An >$	

Diese Grammatik ist ebenfalls nicht eindeutig, da sich bei bedingten Anweisungen, der 'else Teil' nicht eindeutig einem 'if' zuordnen läßt.

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

Diese Mehrdeutigkeit läßt sich leicht beseitigen, indem man in der betreffenden Regel Klammmern um die Anweisung setzt: $< An > \quad \longrightarrow \quad if < BA > then\{< AnF >\}else\{< AnF >\}$

2.2 C0 Syntax

C0: ∼PASCAL mit C-Syntax

2.2.1 C0 Grammatik

Wir definieren die komplette C0 Grammatik:
Startsymbol: <programm>, Nonterminale stehen in eckigen Klammern

<Zi>	→	0 | ... | 9	Ziffer
<ZiF>	→	<Zi> | <ZiF><Zi>	Ziffernfolge
<Bu>	→	a | ... | z | A | ... | Z	Buchstabe
<BuZi>	→	<Bu> | <Zi>	alphanumerisches Zeichen
<BuZiF>	→	<BuZi> | <BuZiF><BuZi>	alphanumerische Zeichenfolge
<Na>	→	<Bu> | <Bu><BuZiF>	Name
<C>	→	<ZiF>	Konstante
<id>	→	<Na> | <C> | <id>.<Na> |	Identifier (für Var. u. Konst.)
		<id>[<A>] | <id>* | &<id>	
<F>	→	<id> | $-_1$<F> | (<A>)	Faktor
<T>	→	<F> | <T> * <F> | <T> / <F>	Term
<A>	→	<T> | <A> + <T> | <A> $-_2$ <T>	(algebr.) Ausdruck
<Atom>	→	<A> > <A> | <A> ≥ <A> |	
		<A> < <A> | <A> ≤ <A> |	
		<A> == <A> | <A> ≠ <A> |	
		0 | 1	"Boole'sche Variable"
<BF>	→	<Atom> | ∼<BF> | (<BA>)	Boole'scher Faktor
<BT>	→	<BF> | <BT> ∧ <BF>	Boole'scher Term
<BA>	→	<BT> | <BA> ∨ <BT>	Boole'scher Ausdruck
<An>	→	<id> = <A> | <id> = <BA> |	Zuweisung
		if <BA> then {<AnF>} else {<AnF>} |	bedingte Anweisung
		if <BA> then {<AnF>} |	
		while <BA> do {<AnF>} |	Schleife
		<id> = <Na>(PF) |	Funktionsaufruf
		<id> = <Na>() |	Funktionsaufruf
		<Na> = new <Typ>*	Speicher anfordern
<PF>	→	<A> | <PF>,<A> |	Parameterfolge
<AnF>	→	<An> | <AnF>;<An>	Anweisungsfolge
<program>	→	<DF>; <AnF>	C_0-Programm
<DF>	→	<Dekl> | <DF>; <Dekl>	Deklarationsfolge
<Dekl>	→	<VaD> | <FuD> | <TypD>	Deklaration
<VaD>	→	<Typ> <Na>	Variablen-Deklaration
<VaDF>	→	<VaD> | <VaDF>; <VaD>	Variablen-Deklarationsfolge
<Typ>	→	<elTyp> | elTyp[<ZiF>] | elTyp*|	el. Typen, Array-Typ, Pointer
		<Na> | <Na>[<ZiF>] | <Na>* |	selbstdefinierte Typen
		struct {<KompDF>}	struct-Typ
<elTyp>	→	int | bool | char | unsigned	elementare Typen
<KompDF>	→	<KompD> | <KompDF>,<KompD>	Komponentendeklarationsfolge
<KompD>	→	<Na>:elTyp | <Na>:<Na>	Komponentendeklaration
<FuD>	→	<Typ> <Na>(<ParDF>){<VaDF>;<rumpf>} |	Funktionsdeklaration
		<Typ> <Na>(<ParDF>){<rumpf>} |	ohne lokalen Variablen
		<Typ> <Na>(){<VaDF>; <rumpf>} |	ohne Parameter
		<Typ> <Na>(){<rumpf>}	ohne lokalen Var. und P.
<ParDF>	→	<ParD> | <ParDF>,<ParD>	Parameterdeklarationsfolge
<ParD>	→	<Typ> <Na>	Parameterdeklaration
<rumpf>	→	<AnF>; return <A>| return <A>	Funktionsrumpf
<TypD>	→	typedef <Typ> <Na>	Typdeklaration

Beispiele zur C0 Grammatik

$< ZiF > \longrightarrow^* 0113908$

$< Na > \longrightarrow^* a13$

$< ZiF > \nrightarrow^* 0$ keine führenden Ziffern

$< A > \longrightarrow^* < id > + < id > * < id >$

$< A > \longrightarrow^* 2/2/2$

$< Atom > \longrightarrow^* 2 > X + 3$

$< BA > \longrightarrow^* 2 > X + 3 \vee 1$

$< An > \longrightarrow^* if < BA > then\{\underbrace{if < BA > then\{< AnF >\} else \{< AnF >\}}_{<AnF>}\}$

$< An > \longrightarrow^* if < BA > then\{\underbrace{if < BA > then\{< AnF >\}}_{<AnF>}\} else \{< AnF >\}$

$< An > \longrightarrow^* < id >=< Na > (< PF >) | < id >=< Na > ()$

$< VaD > \longrightarrow^* < typ >< Na >$

2.2.2 Typdeklarationen

Die C0 Grammatik erlaubt folgende Typen:

- Elementare Typen: int, bool, char, unsigned
- Array Typen: int[15], x7[3]
- Pointer Typen: int*, $\underbrace{x7*}_{Pointer\ auf\ Variable\ vom\ Typ\ x7}$
  Typ x7 wird definiert durch Typdeklaration:
  $< TypD > \longrightarrow^* typedef\ int[14]\ x7$
  Beispiel: mehrdimensionales Array:
  typedef int[7] row;
  typedef row[7] matrix;
  matrix A;
- Struct Typen: typedef LEL* u;
  typedef struct{content: int, next:u} LEL; (Listenelement)
- voll rekursive Datentypen:
  typedef LEL[17] P3; (Array von structs)
  typedef struct{N1:int, N2:P3} U2
  typedef U2[4] V7;

Kontextbedingung:
Komplexe Typen (Arrays und Structs) können nur deklariert werden, wenn die Typen ihrer Komponenten bereits deklariert sind:

- i. Typdeklaration: typedef T[<ZiF>] name ⇒ T muß in j. Typdeklaration deklariert worden sein, mit j<i

- i. Typdeklaration: typedef struct{na_1.typ_1, ... , na_s:typ_s} name $\Rightarrow \forall k$: typ_k muß in j_k. Typdeklaration deklariert worden sein, mit j_k <i

Diese Einschränkung gilt nicht für Pointer.
i. Typdeklaration: typedef typ* name: typ darf in j. Typdeklaration, mit j>i, deklariert werden.

Pointer dereferenzieren:
typedef LEL* u;
u p;

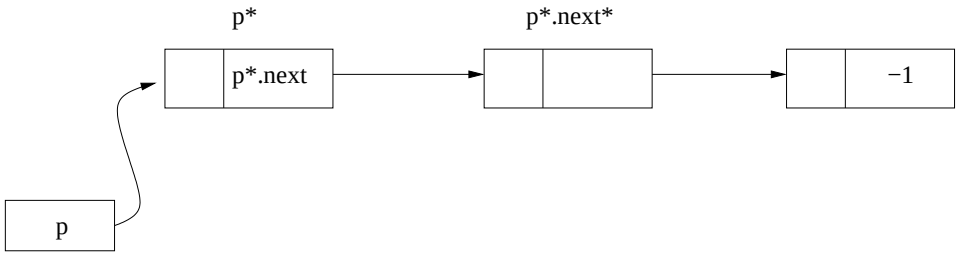


Abbildung 14: einfach verkettete Liste

Wir unterscheiden Variablen mit Namen, diese werden durch globale oder lokale Variablendeklarationen deklariert, und Variablen ohne Namen, diese werden durch die *new* Anweisung angelegt.
$< An > \quad \longrightarrow^* \quad < Na >= new < typ > *$
Die Anweisung name = new typ* erzeugt eine neue namenlose Variable vom Typ *typ* und weist der Variable *name* einen Pointer auf die neue Variable zu.

Beispiel: p = new LEL*;
q = new LEL*;
q*.next = p;

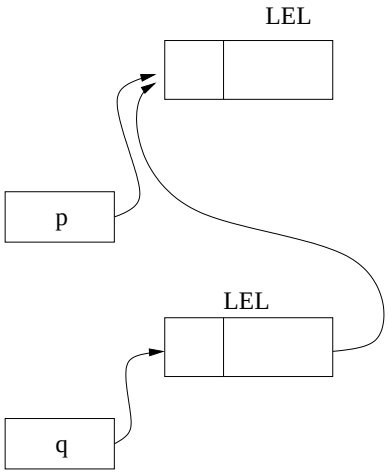


Abbildung 15: Zuweisungen von Pointern

$< id >$ $\longrightarrow^*$ $\& < id >$ Address of operator

Beispiel:

```
typedef LEL* u;
typedef u* v;
v r;
r = &(p*.next);
```

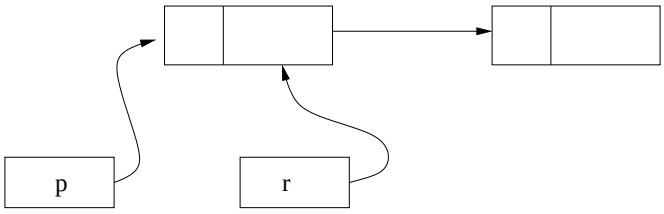


Abbildung 16: Anwendung des *Addressof* Operators

Definition: Ein S-Ausdruck ist eine Folge $s_1 \dots s_r$ mit $s_i \in \{*, [< A >], . < Na >\}$
Die s_i heißen Selektoren.

$< programm >$ $\longrightarrow$ $< DF >; < AnF >$ in DF werden Typen, Funkionen, Variablen deklariert
$< FuD >$ $\longrightarrow$ $< Typ >< Na > (< ParDF >)\{< VaDF >; < rumpf >\}$
Typ: Typ des Rückgabewerts
Name: Name der deklarierten Funktion

ParDF: Deklaration der Parameter der Funktion
VaDF: Deklaration der lokalen Variablen
rumpf: Rumpf der Funktion

2.3 C0 Semantik

$$\delta_{C0}: \underbrace{K_p \rightarrow K_p}_{Konfigurationen\ zu\ Programmp} \qquad L(g) = \{p \in T^* \mid < program > \rightarrow p\}$$

$\delta_{C0}(c) = c'$ c' geht aus c durch Ausführen einer Anweisung hervor
$\delta_{DLX}(c) = c'$ c' geht aus c durch Ausführen einer Instruktion hervor
$\delta_h(c) = c'$ c' geht aus c durch Ausführen eines Taktes hervor

K_p durch Deklarationen in p festgelegt
$K_p \ni$ c = (c.pr, c.rd, c.lms, c.hm, c.rbs)

2.3.1 C0 Konfiguration

Eine C0 Konfiguration c besteht aus:

- c.pr: Programmrest: Folge von C0 Instruktionen, die noch auszuführen sind.
- c.rd: Rekursionstiefe: Anzahl der Funktionsaufrufe, bei denen return noch nicht erfolgt ist
- c.lms: [0:c.rd] $\rightarrow$ {m | m Memory} local memory stack
  - c.lms(0) enthält die globalen Variablen
  - c.lms(i): (i>0) enthält Parameter und lokale Variablen der Funktion mit Rekursionstiefe i, wird später bei Aufruf einer Funktion mit Rekursionstiefe i erzeugt.
    p: Anzahl der Parameter, l: Anzahl der lokalen Variablen, es gilt n = p + l

$$lms(i).name(j) = \begin{cases} \text{Name des j. deklarierten Parameters,} & : j<p \\ \text{Name der (j-p). deklarierten Variable,} & : j \geq p \end{cases}$$

- c.hm: Heap, enthält namenlose Variablen
- rbs: Return Binding Stack
  c.rbs(i) bezeichnet die Subvariable, in die das Ergebnis der Funktion mit Rekursionstiefe i gespeichert werden.

Zusätzlich benötigen wir noch folgende Informationen, die sich während der Laufzeit nicht ändern und deshalb nicht Teil der Konfiguration sind:

nt : number of types

nf : number of functions = Anzahl der deklarierten Funktionen

tt : Type table, kodiert Typdeklarationen.

- tt = (tt.name, tt.tc)
- tt:[0:nt-1] → {Namen}
- tt.name(i) = Name des i. Typs in tt = Name des (i-3). deklarierten Typs (da die 4 elementaren Typen immer vorhanden sind)
- tt.tc: [0:nt-1] → {Type Deskriptoren}
- tt.tc(i) = el für i ∈ $\{0, \ldots 3\}$
- Type code für i≥4 hängt ab von (i-3). Typdeklaration

i	name	tc
0	int	el
1	bool	el
2	char	el
3	unsigned	el
4	1. dekl. Typ	

ft Function Table

- ft(i) = (ft(i).name, ft(i).typ, ft(i).body)
- ft:[1:nf] → Funktionsdescriptor
- ft(i).name : Name der i. deklarierten Funktion
- ft(i).typ : Typ des Rückgabewertes der i. deklarierten Funktion
- ft(i).body : Rumpf der i. deklarierten Funktion

st : Symboltabelle einer Funktion

- st(i) = (st(i).n, st(i).name, st(i).typ)
- st(0): Symboltabelle des Hauptprogramms, enthält die globalen Variablen.
- st(i+1), i≥0: Symboltabelle der i. deklarierten Funktion
- st(i+1).n, i≥0: Anzahl der Variablen und Parameter der i. Funktion
- st(0).n: Anzahl globaler Variablen
- st(i).typ: [0:st(i).n - 1] → [0:nt - 1]
- st(i).name: [0:st(i).n - 1] → {Namen}

sy Symboltabelle eines Memory
sy(top(c')) = st(i) für irgendein i

2.3.2 Anlegen von neuen Typen mittels typedef

Fallunterscheidung über Format der (i-3). Typdeklaration.

1. typedef name1[n] name2
   Erzeugt einen neuen arrayTyp mit Namen *name*2, wobei die Elemente vom Typ *name*1 sind.
   Falls gilt: $\exists j < i$: c.tt.name(j) = name1

$$\Longrightarrow \begin{cases} \text{c.tt.name(i)} = \text{name2} \\ \text{c.tt.tc(i)} = \text{arr(n,j)} \end{cases}$$

Beispiel:
typedef int[5] x;
typedef x[5] y;

erweitert die Type Table um:

i	name	tc
...	...	...
4	x	arr(5,0)
5	y	arr(5,4)

2. typedef name1* name2
   Erzeugt einen neuen Pointertyp mit Namen *name*2
   Falls gilt: $\exists j \in \{0 \dots nt-1\}$: c.tt.name(j) = name1 (j>i möglich)

$$\Longrightarrow \begin{cases} \text{c.tt.name(i)} = \text{name2} \\ \text{c.tt.tc(i)} = \text{ptr(j)} \end{cases}$$

   Beispiel: typedef LEL* u

   erweitert die Type Table um:

i	name	tc
...	...	...
6	u	ptr(7)

   Anmerkung: Der Index von ptr() kann im allgemeinen erst eingesetzt werden, wenn der Typ *name*1 angelegt wird.

3. typedef struct$\{n_1 : t_1, \dots, n_s : t_s\}$ name wobei n_i Komponentennamen und t_i die entsprechenden Typen bezeichnen
   Falls $\forall$ k $\in \{1, \dots, s\}$ gilt: $t_k = c.tt.name(i_k) \wedge i_k < i$

$$\Longrightarrow \begin{cases} c.tt.name(i) = name \\ c.tt.tc(i) = struct(n_1 : i_1, \dots n_s : i_s) \end{cases}$$

   Beispiel: typedef struct{content: int, next: u} LEL

   erweitert die Type Table um:

i	name	tc
...	...	...
7	LEL	struct(content:0, $n_s : i_s$)

2.3.3 Größe von Typen

Wir unterscheiden einfache und komplexe Typen:
t einfach (elementarer Typ oder Pointer Typ) : size(t) = 1
t = t'[n] : size(t) = n * size(t')
t = struct$\{n_1 : t_1, \dots n_s : t_s\}$: size(t) = $\sum_{i=1}^{s} size(t_i)$
Die Größe jedes Typs läßt sich somit induktiv berechnen.

Beispiel:
size(int) = 1
size(x) = 5 * size(int) = 5
size(y) = 5 * size(x) = 25
size(u) = 1

size(LEL) = size(int) + size(u) = 2

t Typ : size(t) = Größe von Werten vom Typ t
Range : Ra(t) = {w | Variablen vom Typ t können Wert w annehmen} (keine Definition !)
Ra(int) $\neq \mathbb{Z}$ da wir später keine unendliche großen Werte in der Hardware speichern können
Ra(int) = $T_{32} = \{-2^{31}, \dots 2^{31} - 1\}$
Ra(char) = $\{0, \dots, 2^8 - 1\}$
Ra(t'*) = {u | u $\underbrace{Subvariable}_{noch\ nicht\ definiert}$ vom Typ t' }

Ra(t), mit t: zusammengesetzt (array oder struct), ist eine Folge von einfachen Werten der Länge size(t)

Ra(t'[n]) $\ni$ f:{0, ... , size(t'[n] - 1)} $\rightarrow$ { $\underbrace{w \mid w\ einfach}_{noch\ nicht\ definiert}$ }

Ra(struct{$n_1 : t_1, \dots n_s : t_s$}) $\ni$ f:{$0, \dots, \sum_{i=1}^{s} size(t_i) - 1$} $\rightarrow$ {w | w einfach}

2.3.4 Position von struct und array Elementen im Wert

Beispiel:

typedef int[5] x;
x y;

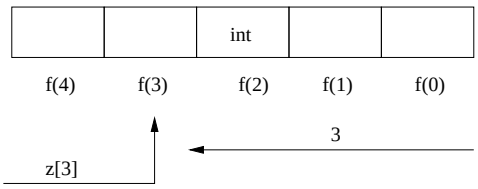


Abbildung 17: Displacement einer Arraykomponente

typedef x[5] y;
y zz;

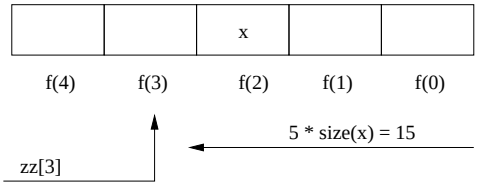


Abbildung 18: Displacement einer Komponente im mehrdimensionalen Array

Allgemeine Berechnung des Displacements einer Komponente:

- Displacement im Array
  $\boxed{\text{t = t'[n]} \wedge \text{j<n} \Rightarrow \text{displ(j,t)} = \text{j * size(t')}}$

  $\forall i : f_{size(t')}(displ(i,t)) \in Ra(t)$

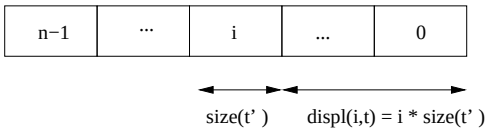


Abbildung 19: Berechnung des Displacements im Array

- Displacement im Struct
  $\boxed{\text{t = struct}\{n_1 : t_1, \ldots, n_s : t_s\} \Rightarrow \text{displ}(n_i, \text{t}) = \sum_{j<i} size(t_j)}$

  $\forall i : f_{size(t_i)}(displ(i,t)) \in Ra(t_i)$

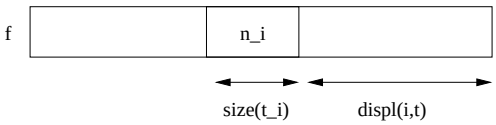


Abbildung 20: Berechnung des Displacements im Struct

Bemerkung: size(t) ist definiert, Ra(t) noch nicht, da Subvariablen noch nicht definiert sind.

2.3.5 Memory

Ein Memory ist ein Tupel m = (m.n, m.name, m.typ, m.ct)

- m.n $\in \mathbb{N}_0$, Anzahl der Variablen in m
  Wir bezeichnen die i.-te Variable in Memory i mit (m, i) wobei i $\in$ {0, ..., m.n - 1}
- m.name $\{0, \ldots, m.n-1\} \to \{Namen\}$ (global, lokal, Parameter, $\underbrace{nicht\ vorhanden}_{hm}$)
- m.typ $\{0, \ldots, m.n-1\} \to \{0, \ldots, nt-1\}$ Abbildung auf Nummern der Typen in Type Table.
- m.ct {0 ... , size(m) - 1} $\to$ {w | w einfacher Wert}, Content = Inhalt des Speichers, size(m)= $\sum_{i=0}^{m.n-1} size(m.typ(i))$

2.3.6 Subvariablen

Definition: Subvariablen Vs
V=(m, i) : Variable, s: Selektor

- Variablen sind Subvariablen (s=ε)
- Sei u Subvariable vom Typ t = t'[n]
  j<n $\Rightarrow$ u[j] Subvariable vom Typ t'
- Sei u Subvariable vom Typ t = struct$\{n_1 : t_1, \ldots, n_s : t_s\}$
  1$\leq$j$\leq$s $\Rightarrow$ u.n_j Subvariable vom Typ t_j

  Bemerkung: Dadurch sind Ra(t'*) und Ra(t) definiert

2.3.7 Basisadresse und Wert von Variablen

ba(m, i) = $\sum_{j=0}^{i-1} \underbrace{size(m.typ(j))}_{size(m,j)}$
size(m, i) = size(m.typ(i))
typ(m, i) = m.typ(i)
$m.ct_{size(m,i)}(ba(m, i)) \in Ra(typ(m, i))$

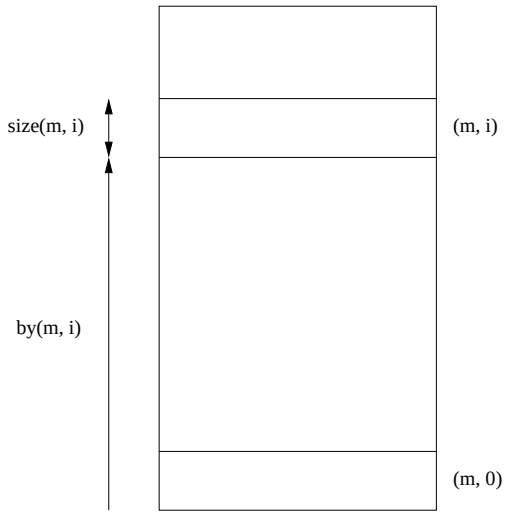


Abbildung 21: Basisadresse der i. Variable im Memory m

Wert von Variablen in Konfiguration c
va(c, (m, i)) = m.ct$_{size(m,i)}$(ba(m, i))

2.3.8 Basisadresse und Wert von Subvariablen

u = (m, i)$s_1 s_2$...

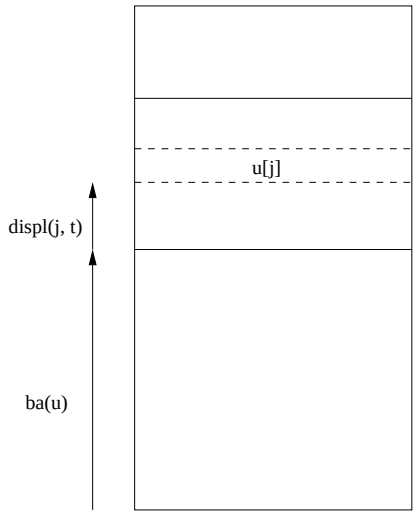


Abbildung 22: Basisadresse einer Subvariabl

- Sei u eine Subvariable vom Typ t'[n] in Memory m
  ba(u[j]) = ba(u) + displ(j, t)
  va(c, u[j]) = m.ct$_{size(u[j])}$(ba(u[j]))
  size(u[j]) = size(typ(u[j]))

- Sei u eine Subvariable vom Typ t = struct$\{n_1 : t_1, \ldots, n_s : t_s\}$
  ba(u.n$_j$) = ba(u) + displ(j, t)
  va(c, u.n$_j$) = m.ct$_{size(u.n_j)}$(ba(u.n$_j$))

Beispiel:

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

m = lms(0), size=1 + 5 + 25= 31, m.n = 3

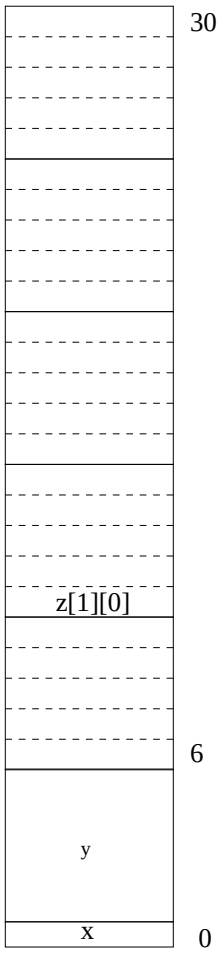


Abbildung 23: Beispiel zur Berechnung der Basisadresse einer Subvariablen

Mit top(c) bezeichnen wir den obersten local memory stack in Konfiguration c.
TODO Adressrichtung umdrehen

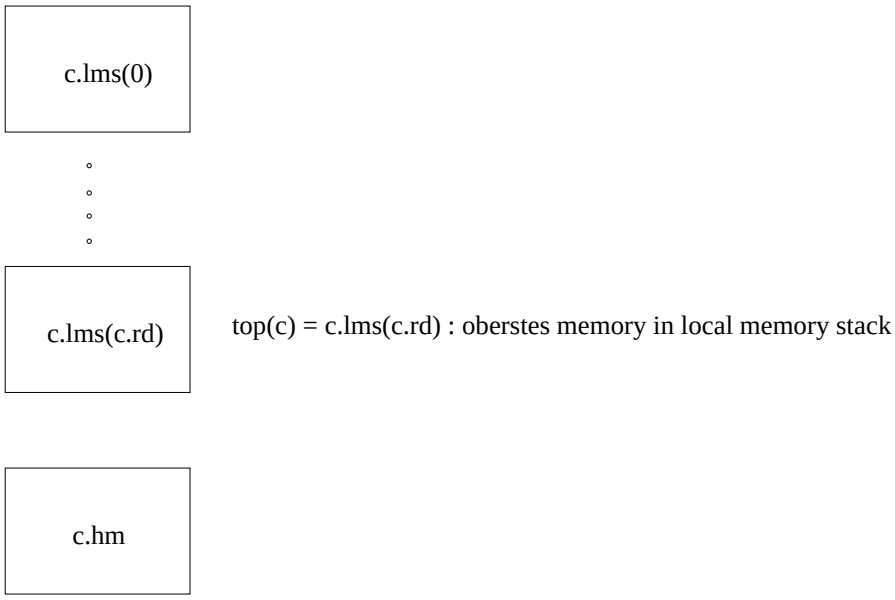


Abbildung 24: Anordnung der Memories

2.3.9 Sichtbarkeitsregeln

c: Konfiguration, x: Vorkommen einer Variablen oder eines Parameters

$$bind(c,x) = \begin{cases} (top(c), j) & : ((x \; lokal \; oder \; Parameter) \wedge top(c).name(j) = x) = \ominus \\ (lms(0), j) & : (\sim \ominus) \wedge lms(0).name(j) = x \end{cases}$$

Das Vorkommen einer Variablen oder eines Parameters ist entweder an eine Variable im obersten lms gebunden oder an eine globale Variable.

2.3.10 Ausdrucksauswertung

Seien: c: Konfiguration, e Vorkommen eines Ausdrucks

Wert: va(c, e)
Bindung: bind(c ,e) = Subvariable (falls e identifier)

Fallunterscheidung über e:

Konstanten: e Ziffernfolge
va(c, e) = $< e >_{10} = \sum e_i \cdot 10^i$
bind(c, e) nicht definiert

Variablen: e = X

$$bind(c,X) = \begin{cases} (top(c), j) & : ((X \; lokal \; oder \; Parameter) \wedge top(c).name(j) = X) = \ominus \\ (lms(0), j) & : (\sim \ominus) \wedge lms(0).name(j) = X \end{cases}$$

va(c, X) = va(bind(c, X))

Array Zugriff: e = e'[e"], e identifier, e" Ausdruck
bind(c, e) = bind(c, e')[va(c, e")]

Struct Zugriff: e = e'.n
bind(c, e) = bind(c, e').n

Pointer dereferenzieren: e = e'*, e' Pointer Variable: Wert Subvariable
bind(c, e) = va(c, e')
(va(c, e) = va(va(c, e'))) Value von Variable nimmt nur 1 Argument

Address-of: e = &e', e' identifier wird pointer zugewiesen
va(c, e) = bind(c, e')
bind(c, e) nicht definiert

Ausdrücke: z.B. e = e' + e"
bind(c, e) nicht definiert
va(c, e) = va(c, e') + va(c, e")

Beispiel: (&e')*
bind(c, (&e')*) = va(c, &e') = bind(c, e')
va(c, (&e')*) = va(bind(c, e'))

2.3.11 Ausführen von Anweisungen

c^0 : Startkonfiguration
$c^0.rd = 0$
$c^0.rbs(0)$ nicht definiert
programm: d; a
d: Deklarationsfolge
a: Anweisungsfolge
$c^0.pr$ = a
Fallunterscheidung über den Kopf des Programmrests

Zuweisung c.pr: id = e;r'
Sei u = (m, i)s

$$\underbrace{va\,\underbrace{(va\,\underbrace{(va(c', id))}_{bind(c', id)})}_{bind(c, id)}}_{m.ct_{size(u)}(ba(u))} = va(c, e)$$

c'.pr = r'

If then Else c.pr: if e then {a} else {b}; r'

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

While Schleife c.pr: while e do {a}; r'

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

Funktionsaufruf c.pr: id = f($e_0, \ldots, e_{p-1}$); r'
Sei f = ft(i).name

1. Rekursionstiefe erhöhen
   c'.rd = c.rd + 1
2. Neues lokales Memory
   sy(lms(c'.rd)) = st(i)
3. Subvariable, die Return Wert aufnimmt speichern
   rbs(c'.rd) = bind(c,id)
4. Parameter übergeben
   $\forall j \in \{0, \ldots, p-1\}$ mit typ(j) : einfach $\Rightarrow$ top(c').ct(j) = va(c, e_j)
5. Rumpf ausführen
   c.pr = ft(i).body; r'

Return c.pr: return e; r'
Kontextbedingung: keine Pointer auf lokale Subvariablen!
($\Rightarrow$ Lemma: bind(c, id) = (m ,k)s für m $\in$ {lms(0), lms(c.rd), hm})

1. Rückgabewert zuweisen
   c.rbs(c.rd) = (m, k)s
   va(c', (m, k)s) = va(c, e)
2. Rekursionstiefe veringern
   c'.rd = c.rd - 1
3. Programmrest verringern
   c'.pr = r'

New c.pr: p = new(T*); r'

1. neue namenlose Variable vom Typ T auf Heap anlegen
   V = (hm, c.hm.n)
   c'.hm.n = c.hm.n + 1
   c'.hm.typ(c.hm.n) = t mit tt.name(t)=T
2. Pointer p auf die neue Variable zeigen lassen
   va(c', p) = V
3. Programmrest verringern
   c'.pr = r'

2.4 Korrektheitsbeweise

2.4.1 DLX

Wir betrachten folgendes DLX Programm:

```
1  lw RS1 = 0, RD = 1, imm = a
2  add RS1 = 0, RD = 2, imm = 0
3  beqz RS1 = 1, imm = 16
4  add RS1 = 2, RS1 = 1, RD = 2
5  subi RS1 = 1, RD = 1, imm = 1
6  j imm = -12
7  sw RS1 = 0, imm = 1, RD = 1
```

informal: nach dem j. Durchlauf der Schleife:
$c^{2+4j}.gpr(1) = h - j$
$c^{2+4j}.gpr(2) = \sum_{k=n-j+1}^{n} k$

2.4.2 C0

1. Beispiel: Wir betrachten folgendes C0 Code Fragment und beweisen mithilfe der Semantik der C0 Anweisungen daß der Wert der Variable x nach Ausfühhren der if Anweisung zwei ist.
   int x;
   x = 3;
   if x = 0 then {x = 1} else {x = 2}

Programmreste	Werte
$c^0.pr$: x=3; if x=0 then {x=1} else {x=2}	va(c^0, x)=undef.
$c^1.pr$: if x=0 then {x=1} else {x=2}	va(c^1, x) = 3
$c^2.pr$: x=2	va(c^2, x=0)=false
$c^3.pr$: ε	va(c^3, x)=2

2. Beispiel: Wir betrachten folgendes C0 Code Fragment:
   int n;
   int result;
   result = 0;
   while n>0 do {result = result + n; n=n-1}
   Anfangs gilt va(c^0, n) = N

   Falscher Ansatz:
   Induktionsvoraussetzung:
   Nach j. Durchlauf der Schleife gilt:
   va(c^{1+3j}, n) = N - j
   va(c^{1+3j}, result) = $\sum_{k=N-j+1}^{N} k$
   Induktion: j →j + 1: (Induktionsschritt über den nächsten Durchlauf, aber while Definitionn selbst induktiv über den ersten Durchlauf definiert)
   c^{1+3j+1}.pr: result=result+n; n=n-1; while n>0 do {...}

va(c^{1+3j+3}, n) = N - j - 1
va(c^{1+3j+3}, result) = $\sum_{k=N-j}^{N} k$
c^{1+3j+3}.pr: while n>0 do {...} Problem: Schleife noch da! c.pr sollte nach Beweis ε sein.

Neuer Ansatz:
c.pr: while e do {a}
Anfangs gilt va(c, e) = true
c'.pr: $\underbrace{a}_{1.\ Durchlauf}$; $\underbrace{while\ e\ do\ \{a\}}_{N-1\ letzte\ Durchlufe}$
Induktionsbehauptung:
Sei va(c^{α}, result) = K
va(c^{α}, n) = M
c^{α}.pr: while n>0 do {...}
$\Rightarrow c^{\alpha+3M}$ nach m weiteren Durchläufen

- va($c^{\alpha+3M}$, result) = K + M + M-1 + ... + 1
- va($c^{\alpha+3M}$, n) = 0
- $c^{\alpha+3M}$.pr: while n>0 do {...}

Beweis per Induktion über M

2.5 Compiler C0 → DLX

2.5.1 Funktionsweise

p = d; a
p: C0 Programm
d: Deklarationsteil: definiert Typetable (tt), Symboltable (st) und Functiontable (ft)
a: Anweisungsteil

Der Compiler erzeugt aus a code(a). Code(a) ist eine Folge von DLX Instruktionen, die a simulieren sollte.
Code(a) wird rekursiv über Ableitungsbäume von a und den Funktionsrümpfen ft(i).rumpf definiert.

Ein Compiler besteht aus:

- Parser: Erzeugen der Ableitungsbäume und Checken der Kontextbedingungen
- Optimierer (nicht unbedingt benötigt und hier nicht betrachtet)
- Code Erzeuger: erzeugt aus C0 Statements passende DLX Instruktionen

2.5.2 Simulationssatz

Ziel: Schritt für Schritt Simulation:
Wir werden zeigen, daß zu jeder Rechnung der C-Maschine $(c^0, c^1, c^2, \ldots)$ eine Rechnung der DLX-Maschine $(d^0, d^1, d^2, \ldots)$, sowie die zugehörigen Schrittzahlen $(S(0), S(1), S(2), \ldots)$ und Funktionen aba^t $(aba^0, aba^1, aba^2, \ldots)$ existieren, sodaß für alle Schritte t gilt:

$d^{S(t)}$ codiert c^t mittels aba^t.

In Worten: t Schritte der C-Maschine werden durch S(t) Schritte der DLX-Maschine simuliert. Hierfür definieren wir ein Prädikat consis($c^t, aba^t, d^{S(t)}$), daß die Gültigkeit obiger Aussage ausdrückt. Wir zeigen, daß nach jedem Schritt consis($c^t, aba^t, d^{S(t)}$) gilt.

2.5.3 Function Frames in der DLX Maschine

Äquivalent zu den lms(i) in der C-Maschine definieren wir Function frames F(i) in der DLX Maschine. F(i) codiert lms(i) und rbs(i). Zusätzlich benötigen wir in den F(i) zwei zusätzliche Worte für Verwaltungsinformationen. Die Funktion asize(x) berechnet die Größe einer Variable x in der DLX Maschine. Es gilt asize(x) = 4 * size(x).

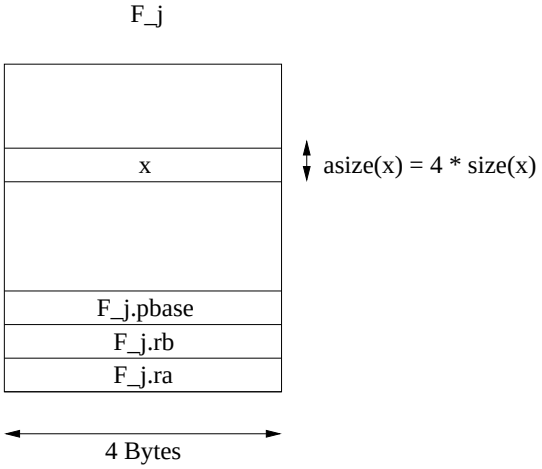


Abbildung 25: Function Frame in der *DLX* Maschine

F_j.pbase : predecessorbase, zeigt auf die erste Adresse von F_{j-1}.

F_j.rb : return binding, Adresse an die Ergebnis der Funktion gespeichert wird.

F_j.ra : return address: Rücksprungadresse, zeigt auf Adresse der Instruktion, die nach Ende der Funktion ausgeführt werden soll.

Anmerkung: Im untersten Frame F_0 sind die drei Werte undefiniert.

Zusätzlich verwenden wir 3 Register

GPR28 Global Memory Pointer: zeigt auf die erste Adresse in F_0.

GPR29 Heappointer: zeigt auf die erste freie Adresse im Heap

GPR30 Stackpointer: zeigt auf die Adresse des obersten Function frames.

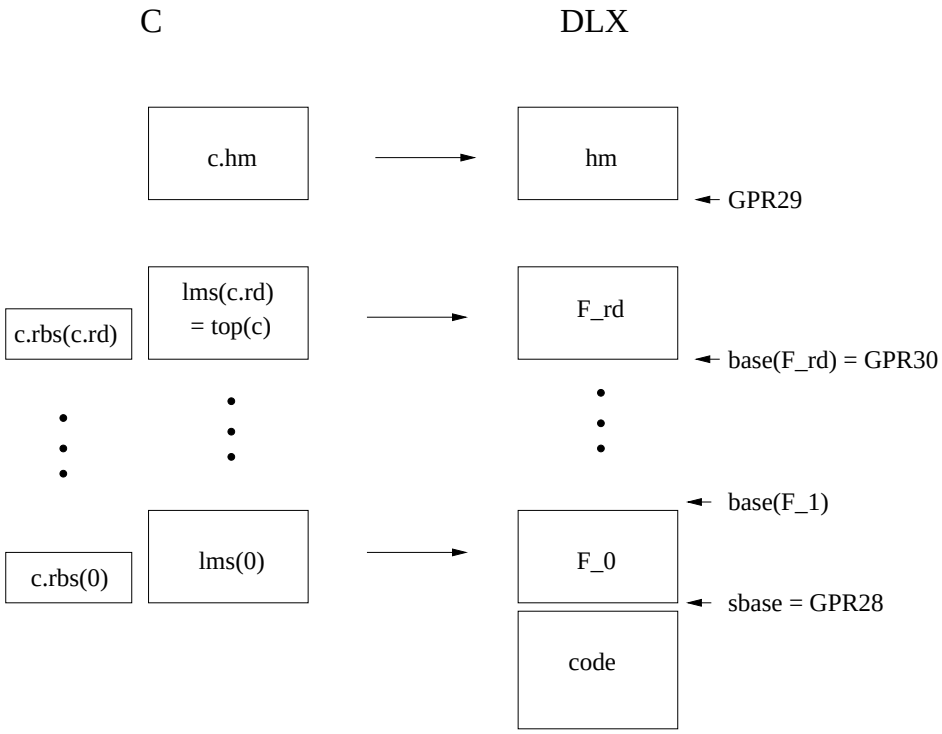


Abbildung 26: Abbildung der C Maschine auf die DLX Maschine

2.5.4 Position der C Variablen im Function Frame

Sei:

f = ft(i).name

x = st(i).name

Dann definieren wir, ähnlich dem Displacement in Structs, das Displacement einer Variable im Function Frame:

$$displ(x, f) = 12 + 4 \cdot \sum_{k<j} size(st(i).typ(k))$$

Achtung: Vorlesung $displ(x, f) = \sum_{k<j} size(st(i).typ(k))$

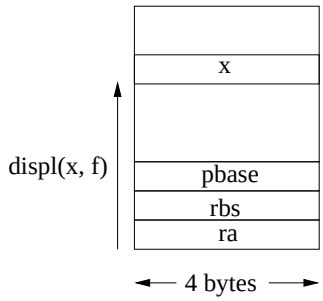


Abbildung 27: Displacement der C Variablen im Function Frame der DLX Maschine

2.5.5 Position der C-Variablen (m, i) ín DLX Maschine

$$\text{aba(c, (m, i))} = \underbrace{base(F_j)}_{} + \underbrace{\sum_{k<j} 12 + asize(c.lms(k))}_{\sum 12+4*size(c.lms(k))} + 3 * 4 + 4 * \text{ba(m, i)}$$

Sei X: Vorkommen von Variablennamen in Funktion f oder Hauptprogramm

$$\begin{aligned} \text{aba(c, X)} &= \text{aba(c, bind(c, X))} \\ &= \text{aba(c, (m, i))} && \text{, m} \in \{\text{top(c), c.lms(0)}\} \\ &= \text{base}(F_j) + 12 + 4 * \text{ba(m, i)} && \text{, m} \in \{\text{top(c), c.lms(0)}\} \end{aligned}$$

aba(c, (m, i)) = base(c, j) + 12 + 4 * ba(m, i)
base(c, j) = $\sum_{k<j}(12 + 4 * size(c.lms(k)))$

aba von Array- und Structkomponenten:
aba(c, u[k]) = aba(c, u) + 4 * k * size(t')
aba(c, u.n_k) = aba(c, u) + $\sum_{i<k}$ 4 * size(t')

e: identifier
aba(c, e) = aba(c, bind(c, e))
definiert für e = Xs, mit s=$s_1 s_2$..., $s_i \in \{[k], .n_k, \varepsilon\}$

2.5.6 Konsistenzbedingungen

Um den Simulationssatz zeigen zu können müssen wir die Konsistenzbedingungen definieren.

Definition: consis(c, aba, d) $\Leftrightarrow$

e-consis(c, aba, d) elementare Konsistenz
$\forall$ Subvariablen U mit typ(U) = elementar:
va(c, U)= d.m_4(aba(c, U))

p-consis(c, aba, d) Pointer Konsistenz
Sei p Pointervariable in C-Maschine, zeigt auf Variable y = va(c, p):
d.m_4(aba(c, p)) = aba(c, y) ,falls p erreichbar

r-consis(c, aba, d) Rekursion

• c.gpr(28) = sbase	F_0
• c.gpr(29) > base(c, c.rd) + 12 + 4 * size(top(c))	Heap darf Stack nicht überlappen
• c.gpr(30) = base(c, c.rd)	F_{rd}
• $\forall$j: 0<j≤c.rd d.m$_4$($\underbrace{base(c,j)+8}_{F_j.pbase}$) = base(c, j - 1)	pbase
• d.m$_4$(base(c, j) + 4) = aba(c, c.rbs(j))	rb
• c.pr = r; return x; r' , r enthält kein return head(r'): 1. Statement von r' code(head(r')): Vom Compiler erzeugter Code start(r'): Begin von Code(head(r')) in d.m d.m$_4$(base(c, j) = start(r')	ra

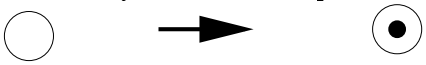
c-consis(c, aba, d) Control
Sei c.pr = r
d.pc = start(r)) : Anfangsadresse von code(head(r))

keine Selbstzerstörung p: Programm, code(p) liegt (unverändert) unterhalb von sbase in d.m zwischen Adressen α, β mit $\alpha + \beta < sbase$
d.m$_{\beta-\alpha}(\alpha)$ = code(p)

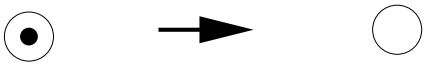
2.5.7 Aho - Ullmann

Spiel auf Graphen (engl. pebble game):
möglichhe Spielzüge:

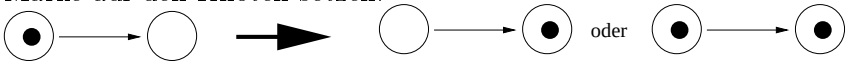
- Auf die Quelle eines Graphen kann eine Marke gesetzt werden



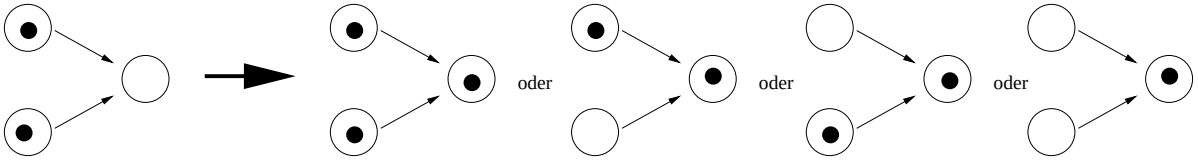
- Von einem markierten Knoten kann die Marke genommen werden



- Wenn der Vorgänger eines Knotens markiert ist, kann der Knoten ebenfalls markiert werden. Hierzu kann man die Marke des Vorgängers auf den Knoten verschieben oder eine zusätzliche Marke auf den Knoten setzen.



- Hat ein Knoten mehrere Vorgänger und sind alle markiert, kann der Knoten ebenfalls markiert werden. Hierzu kann eine der Marken der Vorgänger verschoben oder eine zusätzliche Marke gesetzt werden. Die Marken der anderen Vorgänger können gelöscht werden oder verbleiben.



Definition: T(n) = min{x | Jeder Baum mit n inneren Knoten und ingrad 2 kann mit x Marken markiert werden}

Satz(Aho, Ullmann):
Ein Baum mit n inneren Knoten und ingrad 2 kann immer mit $\lceil \log n \rceil + 2$ Marken markiert werden:
$T(n) \leq \lceil log\ n \rceil + 2$
Bemerkung: innerer Knoten $\neq$ Blatt

Beweis: Induktion über n
Induktionsanfang n=1: 2 Möglichkeiten

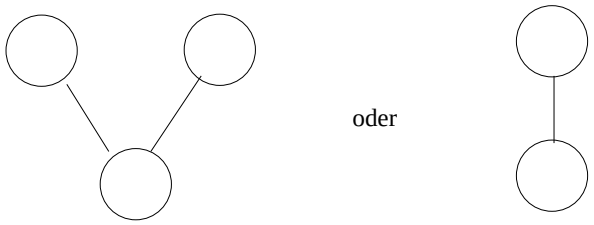


Abbildung 28: Bäume mit einem inneren Knoten

$T(1) = \lceil \log 1 \rceil + 2 = 0 + 2$
Beide Bäume können mit 2 Marken markiert werden.

Induktionsschritt n - 1 → n:

1. Fall:

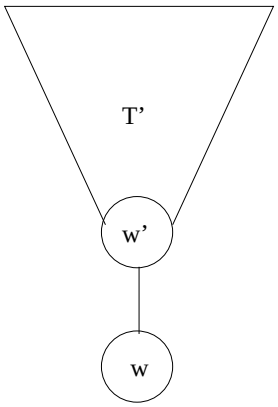


Abbildung 29: Induktionsschritt 1.Fall

(a) markiere T' mit $\leq$ T(n-1) Marken

(b) lösche alle Marken außer der auf w', markiere w

$$\begin{aligned} \#\text{Marken} &\leq \max\{\text{T(n-1)}, 2\} \\ &\leq \max\{\lceil log\ n - 1 \rceil + 2, 2\} \\ &= \lceil log\ n - 1 \rceil + 2 \\ &\leq \lceil log\ n \rceil + 2 \end{aligned}$$

2. Fall:

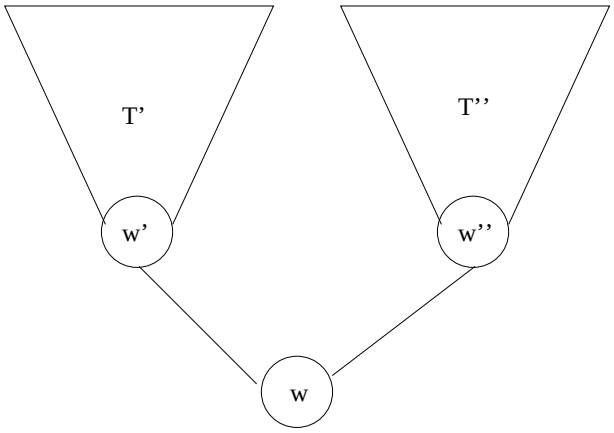


Abbildung 30: Induktionsschritt 2.Fall

oBdA: T' hat $\geq$ innere Knoten als T" $\Rightarrow$ # innere Knoten T" $\leq$ n/2

(a) Markiere T' mit $\leq$ T(n-1) Knoten

(b) lösche alle Marken außer der auf w'

(c) Markiere T" $\leq$ T(n/2)

(d) Nur Marke auf w' und w" lassen, markiere w

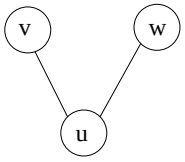
$$\begin{aligned} \#\text{Marken} &\leq \max\{\text{T(n-1)}, 1 + \text{T(n/2)}, 2\} \\ &\leq \max\{\lceil log\ n - 1 \rceil + 2, 1 + \lceil log\ n/2 \rceil + 2, 2\} \\ &= 1 + \lceil log\ n \rceil - 1 + 2 \\ &= \lceil log\ n \rceil + 2 \end{aligned}$$

In unserem Fall entsprechen die Marken den Registern des Prozessors. Die Register 0 und 28 bis 31 sind bereits reserviert, somit verbleiben für den Compiler 27 Register für die Ausdrucksauswertung.

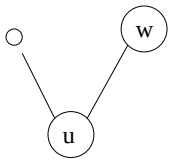
Beispiel: Kann ein Ausdruck mit 1 Million Operanden ausgewertet werden?
Ja, da $\lceil log\ 1000000 \rceil + 2 = 22 \leq 27$

2.5.8 Leftvalue und Rightvalue

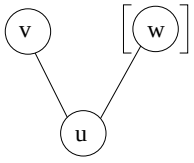
Wenn wir einen Ausdruck e auswerten, müssen wir unterscheiden, ob wir den Wert oder die Bindung des Ausdrucks benötigen. Abhängig davon, wo der Ausdruck vorkommt, sagen wir es handelt sich um einen *Leftvalue*, bzw. um einen *Rightvalue*. Handelt es sich um einen *Leftvalue*, benötigen wir stehts die Bindung, bei einem *Rightvalue* den Wert von e.
Wir definieren ein Prädikat R(e), das genau dann wahr ist, wenn e ein *Rightvalue* ist.

u = v ∘ w : 

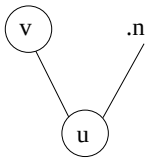
R(v) = R(w) = 1, ∘ ∈ {+, -, ...}

u =∘w : 

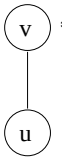
R(w) = 1, ∘ ∈ {∼, $-_1$}

u = v[w]: 

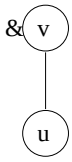
R(v) = 0, R(w) = 1

u = v.w : 

R(v) = 0, R(w) = 1

u = v* : 

R(v) = 1

u = &v : 

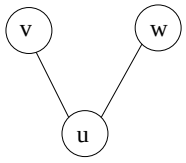
R(v) = 0

2.5.9 Codeerzeugung

Die Operanden werden in Register $gpr(v)$ und $gpr(w)$ geladen. Abhänging davon, ob der auszuwertende Ausdruck e ein right value oder left value ist, wird in Register u folgendes berechnet:

$$gpr(u) = \begin{cases} \text{va(c, e)} & : \text{R(e)} = 1 \\ \text{'bind(c ,e)'} & : \text{R(e)} = 0 \end{cases}$$

mit 'bind' = aba(c, bind(c, e))



Compiler:
e $\mapsto ecode(1)ecode(2)\dots ecode(T)$
$Ecode(t) = ecode(1)\dots ecode(t)$
$d(0) \rightarrow^*_{Ecode(t)} d(t)$

$d \rightarrow^*_p d'$ bedeutet, daß d durch Ausführen von p in d' überführt wird.

Beweis per Induktion über t:
Seien m(1), m(2), ... Züge der Markierungsstrategie, die Marken setzen oder verschieben.
m(t): Zug t $\rightarrow$ ecode(t) Folge von DLX Operationen
Vor: d(0) DLX Startkonfiguration für Auswertung des Ausdrucks
$\forall$ Züge t:
Zug m(t) setzt Marke j auf Knoten (bzw. Teilausdruck) u
d(t) : DLX Konfiguration nach Ausführung von ecode(1) ... ecode(t)

$$d(t).gpr(j) = \begin{cases} \text{va(c, u)} & : \text{R(u) = 1, u kein Pointer} \\ \text{aba(bind(c ,u))} & : \text{R(u) = 0} \\ \text{aba(va(c ,u))} & : \text{u Pointer, R(u) = 1} \end{cases}$$

Voraussetzungen:

1. auszuwertender Ausdruck e kommt im Rumpf von Funktion f vor
2. Es gilt: sy(top(c)) = st(i) und ft(i).name = f
3. consis(c, aba, d(0))

Codeerzeugung:

1. u = x Variable
   Unterscheide ob x lokale oder globale Variable ist:
   (a) x: lokal:
   $$gpr(j) = \underbrace{base(top(c))}_{gpr(30)} + displ(x, f)$$
   erzeugter Code:
   `addi RD = j, RS1 = 30, imm = displ(x, f)`
   (b) x: global
   $$gpr(j) = \underbrace{sbase}_{gpr(28)} + displ(x, f)$$

Lemma: d' Konfiguration nach Ecode(t-1) + 1 Instruktion

$$d' = \delta_{DLX}(d(t-1))$$

Behauptung: d'(t).gpr(j) = aba(bind(c, x))
Beweis:

- m=lms(y)
- $bind(c,x) = \begin{cases} (\text{top(c), j}) & : \text{x lokal} \wedge \text{top(c).name(j)=x} \\ (\text{lms(0), j}) & : \text{x global} \wedge \text{lms(0).name(j)=x} \end{cases}$
- aba(m, j) = $12 + base(c,y) + 4 \cdot \sum_{k<j} size(m.typ(k))$

Wir machen eine Fallunterscheidung über die Bindung von x:

(a) x lokal

$$\begin{aligned} aba(top(c),j) &= \underbrace{base(c,c.rd)}_{gpr(30)(r-consis)} + 12 + 4 \cdot \textstyle\sum_{k<j} size(\underbrace{top(c).typ(k)}_{st(i).typ(k)(Vor.2)}) \\ &= \text{d.gpr(30) + displ(x, f)} \end{aligned}$$

(b) x global

$$\begin{aligned} aba(lms(0),j) &= \underbrace{sbase}_{gpr(28)(r-consis)} + 12 + 4 \cdot \textstyle\sum_{k<j} size(\underbrace{lms(0).typ(k)}_{st(0).typ(k)(Vor.2)}) \\ &= \text{d(0).gpr(28) + displ(x, main)} \end{aligned}$$

Zusätzlich benötigen wir eine Fallunterscheidung über das Vorkommmen von x:

(a) R(u)=0: fertig
erzeugter Code: $d(t).gpr(j) = aba(bind(c,x))$

(b) R(u)=1: zusätzlich dereferenzieren
$gpr(j) = m_4(gpr(j))$
lw RS1=j, RD=j, imm=0

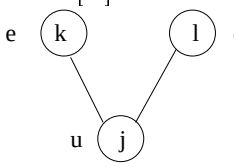
i. typ(x) = elementar
Sei u = bind(c, x)
$m_4(aba(c,u)) = va(c,u) = va(c,bind(c,x)) = va(c,x)$ (e-consis)
Sei d' DLX Konfiguration vor Ausführen der erzeugten Instruktion zum dereferenzieren.
$d'.gpr(j) = aba(bind(c,x)) = aba(u)$ (siehe Fall R(u)=0)
$d(t).gpr(j) = d'.m_4(d'.gpr(j)) = va(c,x)$

ii. typ(x) = ptr(z), gleicher Code
$d(t).gpr(j) = d'.m_4(\underbrace{d'.gpr(j)}_{aba(c,u)}) = aba(c, va(c,x))$ (p-consis)

2. Konstanten
u = c, c $\in \{0,\ldots,9\}^\alpha$, mit $0 \leq \langle c \rangle_{10} \leq 2^{31}-1$
Sei $\langle b \rangle_2 = \langle c \rangle_{10}$
d(t).gpr(j) = b = va(c, u)
erzeugter Code:

```
lhgi RD = j, imm = b[31:16]⊕b[15]
xori RD = j, RS1 = j, imm = b[15:0]
```

3. u = e[e']



Induktionsvoraussetzung(t-1):

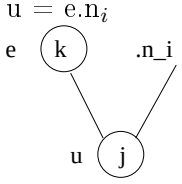
- d(t-1).gpr(l) = va(c,e')
- d(t-1).gpr(k) = aba(c, bind(c, e))

typ(e) = t'[n] = t

(a) R(u)=0
$gpr(j) = gpr(k) + gpr(l) \cdot 4size(t')$
Achtung: Die Multiplikation erfordert mehrere Assemblerinstruktionen.

(b) R(u)=1 zusätzlich dereferenzieren

4. u = e.n_i



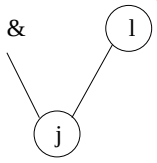
$typ(e) = struct\{n_1 : t_1, \ldots, n_s : t_s\} = t$

(a) R(u)=0
$gpr(j) = gpr(k) + 4 \cdot displ(i, t)$

(b) R(u)=1 zusätzlich dereferenzieren

5. u = e* derefernzieren

6. u=&e $\Rightarrow$ R(e)=0 $\wedge$ R(u)=1



$gpr(j) = gpr(k)$

Sinnvollerweise nutzt man kein neues Register (also k=j) und erzeugt somit keinen Code.

2.5.10 Übersetzen von Anweisungen

Definitionen:

- l(code(x)) = Länge von code(x) in bytes
- Sei f = ft(k).name:
  $msize(f) = \sum_{l<st(k).nt} size(st(k).typ(l))$
- $fsize(f) = 4 \cdot (3 + msize(f))$

Wir machen eine Fallunterscheidung über den Typ der Anweisung:

1. e = e' Zuweisung, R(e)=0, R(e')=1
   code(e) speichert das Ergebnis in Marke j
   code(e') speichert das Ergebnis in Marke k, Marke j darf zur Auswertung nicht verwendet werden.

   d' Konfiguration nach Ausführung von code(e)code(e')
   $d'.gpr(j) = aba(c, bind(c, e))$
   $d'.gpr(k) = \begin{cases} \text{va(c, e')} & : \text{A} \\ \text{aba(c, va(c, e')} & : \text{B} \end{cases}$
   $m_4(d'.gpr(j)) = d'.gpr(k)$
   erzeugter Code:
   `sw RS1=j, RD=k, imm=0`
   Zu zeigen: Im Fall A bleibt e-consis, im Fall B p-consis erhalten.

2. if e then {a} else {b}
   code(e), speichert Ergebnis in Register k
   `beqz RS1=k, imm=l(code(a))+8`
   code(a)
   `j imm=l(code(b))+4`
   code(b)

3. while e do {a}
   code(e), speichert Ergebnis in Register k
   `beqz Rs1=k, imm=l(code(a))+8`
   code(a)
   `j imm=-(l(code(a)) + 4 + l(code(e)))`

4. $id = f(e_1, \ldots, e_p)$ im Rumpf von f'
   typ(e_i)=einfach, f'=ft(k).name

   (a) Abbruch falls Erzeugen des neuen Stackframes eine Überschneidung mit dem Heap bedeuten würde.
   erzeugter Code:

   ```
   addi RS1=30, RD=1, imm=fsize(f')+fsize(f)
   sub RS1=1, RS2=29, RD=1
   sgri RS1=1, RD=1, imm=0
   beqz RS1=1, imm=8
   Code für Abbruch
   ```

   Code für Abbruch wird eine *trap* Instruktion sein, die wir in Kapitel 3 vorstellen werden.

   (b) Übergeben der Parameter: pcode(1)...pcode(p)
   wobei pcode(i) den i. Parameter übergibt.
   Erzeugter Code:
   ecode(e_i) (wertet e_i aus, speichert Wert in Register k)
   `sw RD=k, RS1=30, imm=fsize(f') + 4·3 + 4·(i-1)`

   (c) *pbase* im neuen Stackframe setzen
   `sw RD=30, RS1=30, imm=fsize(f') + 8`

(d) Adresse für den Return Wert von f in *rb* speichern
ecode(id) (speichert Bindung von id in Register l)
`sw RD=l, RS1=30, imm=fsize(f') + 4`

(e) Stackpointer auf neuen Stackframe setzen
`addi RS1=30, RD=30, imm=fsize(f')`

(f) Sprung zu Code(f)
Sei a(f) Startadresse von code(f).
gpr(j) := a(f) (2 DLX Instruktionen)
Problem: Falls Funktion f erst nach Funktion f' übersetzt wird ist a(f) noch nicht bekannt. Daher macht der Compiler 2 Durchläufe:

i. nicht alle a(f_i) bekannt, lasse Sprungweiten offen.

ii. Alle a(f_i) bekannt, fülle Sprungweiten ein

`jalr RS1=j`
Gepipelinete Maschinen besitzen die Eigenart, daß nach einem Sprung stets die nächste regulär folgende Instruktion noch ausgeführt wird. Daher könnte man in einer solchen das Sichern von *ra* auch im Funktionsaufruf noch unterbringen, eben direkt nach obigem *jalr*. In unserem einfachen Prozessor funktioniert das allerdings nicht so, daher müssen wir das Sichern am Anfang von Code(f) unterbringen.

(g) Anfang von code(f): Speichere *ra* in Stackframe
`sw RS1=30, RD=31, imm=0`

Kapitel 3

Betriebssystemkerne

3.1 Übersicht

1. Betriebssystemunterstützung in Prozessoren
   - Interrupts und Special Purpose Register
   - Memory Management Unit (MMU) und Adressübersetzung
   - Input/Output (I/O)
2. Communicating Virtual Maschines: CVM
   - abstraktes Modell für einen in $C0$ geschriebenen abstrakten Microkern und durch DLX-Maschinen simulierte Benutzerprozesse
3. Implementierung
   - Implementierung des abstrakten Kerns k durch konkreten Kern K
   - Hierzu benötigen wir eine Erweiterung der Sprace $C0$ zu $C0_A$. In $C0_A$ können zusätzlich zu $C0$ Anweisungen auch DLX-Assembler Instruktionen verwendet werden. Diese können direkt im $C0$ Anweisungsteil stehen. Daher der Name Inline Assembler.

3.2 Betriebssystemunterstützung in Prozessoren

3.2.1 Interrupts

Definition: Ein Interrupt ist ein Signal, das den Prozessor dazu bringt die normale Programausführung zu unterbrechen und, je nach Interrupt, eine bestimmte Aktion auszuführen. Hierzu wird, je nach Interrupt, ein bestimmter Handler aufgerufen. Formal stellt dies eine Art Funktionsaufruf dar.

Wir unterscheiden externe und interne Interrupts

- iev(j) internes Interrupt event Signal, nur definiert für j $\in$ {1,2,3,4,5,6}
- eev(j) externes Interrupt event Signal, nur definert für j=0 oder j $\geq$ 7

Ein externer Interrupt entsteht nicht im Prozessor, sondern wird von außen signalisiert, wohingegen ein interner Interrupt durch die Berechnungen im Prozessor entsteht.

Daher erweitern wir unsere Übergangsfunktion um das Argument *eev*. Beachte, daß *iev* bereits in der Konfiguration enthalten ist.

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

3.2.2 Übersicht über die Interrupts

j	Name	resume type	maskierbar	extern
0	reset	abort	nein	ja
1	ill	abort	nein	nein
2	mal	abort	nein	nein
3	pff	repeat	nein	nein
4	pfls	repeat	nein	nein
5	trap	continue	nein	nein
6	ovf	continue	ja	nein
≥ 7	I/O	continue	ja	ja

Maskierbar bedeutet, daß das Auftreten eines Interrupts unter bestimmten Bedingungen ignoriert werden kann. Der resume type eines Interrupts gibt an, wie ein durch einen Interrupt unterbrochenes Programm weiterausgeführt wird: *abort* bricht das Programm ab, da der Grund des Interrupts nicht behoben werden kann, *repeat* wiederholt die unterbrochene Instruktion, nachdem die Ursache für den Interrupt beseitigt wurde, *continue* führt das Programm normal weiter aus, nachdem der Interrupt behandelt wurde.

Beschreibung der Interrupts:

0. Ein *Reset* Interrupt tritt auf, wenn der Rechner neugestartet wird.
1. *Illegal* Interrupt bedeutet, daß der Inhalt des Instruktionsregisters vom Instruktionsdekoder nicht als gültige DLX Assembler Instruktion erkannnt wurde.
2. Ein *Misalignment* Interrupt bedeutet, daß die Adresse, mit der auf den Speicher zugegriffen wurde nicht durch 4 teilbar, also nicht aligned, ist.
3. Ein *Page Fault Fetch* tritt auf, wenn der Speicherbereich, aus dem die nächste auszuführende Instruktion geladen werden soll sich momentan nicht im RAM, sondern in der Auslagerungsdatei auf der Festplatte befindet.
4. Ein *Page Fault on load/store* tritt auf, wenn der Speicherbereich, auf den die effektive Adresse einer *lw* oder *sw* Instruktion zeigt gerade ausgelagert ist.
5. Ein *trap* Interrupt wird durch die gleichnamige DLX Assembler Instruktion erzeugt.
6. Ein *Overflow* Interrupt signalisiert, daß bei einer arithmetischen Operation das Ergebnis nicht mehr im darstellbaren Bereich liegt.
7. Ein *I/O* Interrupt tritt auf, sobald Daten von einem externen Gerät für den Prozessor bereitstehen, z.B. wenn eine Taste auf dem Keyboard gedrückt wird.

3.2.3 Special Purpose Register und zusätzliche Signale

Um die Behandlung von Interrupts zu ermöglichen erweitern wir unseren Prozessor um ein sogenanntes *Special Purpose Register* File. Daher erweitern wir die Konfiguration um eine Komponente

$$c.SPR : \{0,1\}^5 \rightarrow \{0,1\}^{32}$$

i	SPR(i)	Name
0	SR	Status Register
1	ESR	Exception Status Register
2	ECA	Exception Cause Register
3	EPC	Exception PC
4	EDATA	Exception Data

Im folgenden schreiben wir c.SR anstatt c.SPR(0), usw.

Wir definieren uns das *Cause* Signal $\in \{0,1\}^{32}$, so daß das i. Bit 1 ist, genau dann wenn Interrupt i aktiv ist:

$$CA(c,eev)[j] = \begin{cases} \text{eev[j]} & : \text{j} \in \{0,\ 7,\ 8,\ \ldots\} \\ \text{iev(c)[j]} & : \text{j} \in \{1,\ \ldots,\ 6\} \end{cases}$$

Das Status Register speichert die Maske, die angibt welche Interrupts maskiert sind. Daher definieren wir das *Masked Cause* Signal $\in \{0,1\}^{32}$ in Abhängigkeit von CA und SR:

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

Es gilt: MCA[i] = 1 $\Leftrightarrow$ Interrupt i ist aktiv und nicht maskiert.

Wir definieren uns ein Signal *Jump to Interrupt Service Routine* $\in$ {0,1}, das genau dann aktiv ist, wenn ein nicht maskierter Interrupt aktiv ist.

$$JISR = \bigvee_{j \in \{0,\ldots,31\}} MCA(c,eev)[j]$$

Allgemein können mehrere Interrupts gleichzeitig auftreten (darunter jeweils nur ein Interner). Daher definieren wir noch den *interrupt level* $\in \{0,\ldots 31\}$ der den Index des Interrupts mit der höchsten Priorität unter den gerade aktiven, nicht maskierten Interrupts angibt. Beachte, daß kleinerer Index gleichbedeutend mit höherer Priorität ist.

$$il(c,eev) = min\{j \mid MCA(c,eev)[j] = 1\}$$

3.2.4 Erweiterung des DLX Instruktionssatzes

Wir erweitern den Instruktionssatz um folgende Instruktionen:
J-Type:

I[31:26]	mnemonic	Effekt
111110	trap	iev(c)[5] = 1
111111	rfe	SR = ESR, PC = EPC

R-Type:

I[5:0]	mnemonic	Effekt	
010000	movs2i	RD = SA	(SPR[SA] → GPR[RS1])
010001	movi2s	SA = RS1	(GPR[RD] → SPR[SA])

- Die *trap* Instruktion bietet die Möglichkeit gezielt einen speziellen Interrupthandler aufzurufen. Der Index des gewünschten Handlers wird als immediate Konstante angegeben und im $EDATA$ Register gespeichert. Beachte, daß im allgemeinen die Menge der durch *trap* aufrufbaren Interrupthandler disjunkt zu der Menge der Handler für die anderen Interrupts ist.
- Die rfe Instruktion dient zum Beenden eines Interrupt Handlers. Sie lädt gleichzeitig den PC und das *Status Register* des durch den Interrupt unterbrochenen Programs.
- Die $movs2i$ und $movi2s$ Instruktion kopieren Werte vom SPR ins GPR, bzw. umgekehrt.

3.2.5 Effekt eines Interrupts

Sobald das Signal $JISR$ aktiv wird werden die Special Purpose Register und der PC gleichzeitig und wie folgt geupdated:

Setze das Statusregister auf 0, damit die Interruptbehandlung möglichst nicht unterbrechbar ist:

$c'.SR = 0^{32}$

Speichere den PC, an dem das Programm nach der Interruptbehandlung evtl. fortgesetzt wird:

$$c'.EPC = \begin{cases} c.PC & : \text{il(c)} \in \{3, 4\} \\ \delta(c).PC & : \text{sonst} \end{cases}$$

Speichere den Grund des Interrupts, anhand dessen die Interrupt Service Routine den entsprechenden Interrupt Handler auswählt:

$c'.ECA = MCA(c, eev)$

Speichere die Interruptmaske um sie nach Behandlung des Interrupts wiederherstellen zu können. Beachte: Wenn die durch den Interrupt unterbrochene Instruktion gerade das *Status Register* verändert hätte und der Interrupt vom Typ *continue* ist, wird der neue Wert gespeichert:

$$c'.ESR = \begin{cases} c.gpr(RS1(c)) & : movi2s(c) \wedge SA(c) = 0^5 \wedge il(c, eev) \geq 5 \\ c.SR & : \text{sonst} \end{cases}$$

Das $EDATA$ Register speichert im Falle eines *trap* Interrupts den Index des aufzurufenden Traphandlers und im Falle eines Page Faults on load/store die effektive Adresse:

$$c'.EDATA = \begin{cases} sxtimm(c) & : il(c, eev) = 5 \\ ea(c) & : \text{sonst} \end{cases}$$

Der PC wird auf die Startadresse der Interrupt Service Routine gesetzt. Bei uns steht die Interrupt Service Routine am Anfang des Speichers:

$c'.PC = SISR = 0^{32}$ (Startaddress of Interrupt Service Routine)

3.2.6 Änderungen an der Hardware

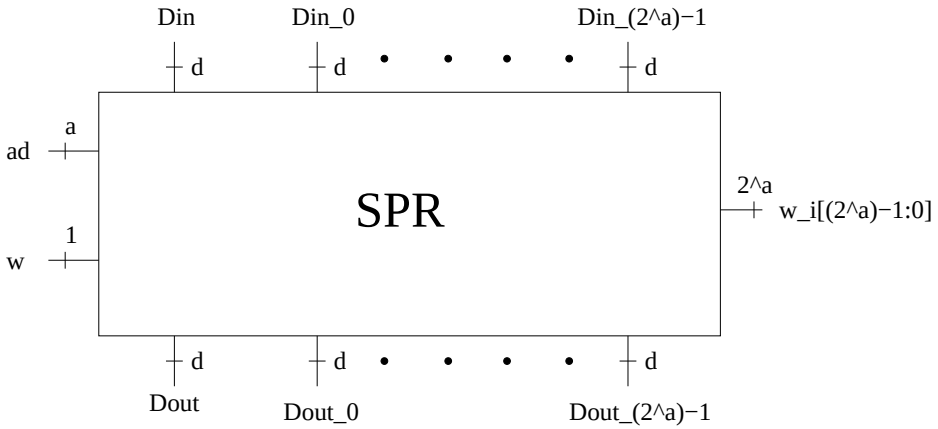


Abbildung 31: Spezifikation eines 2^a x d SPR

Konstruktion des SPR:

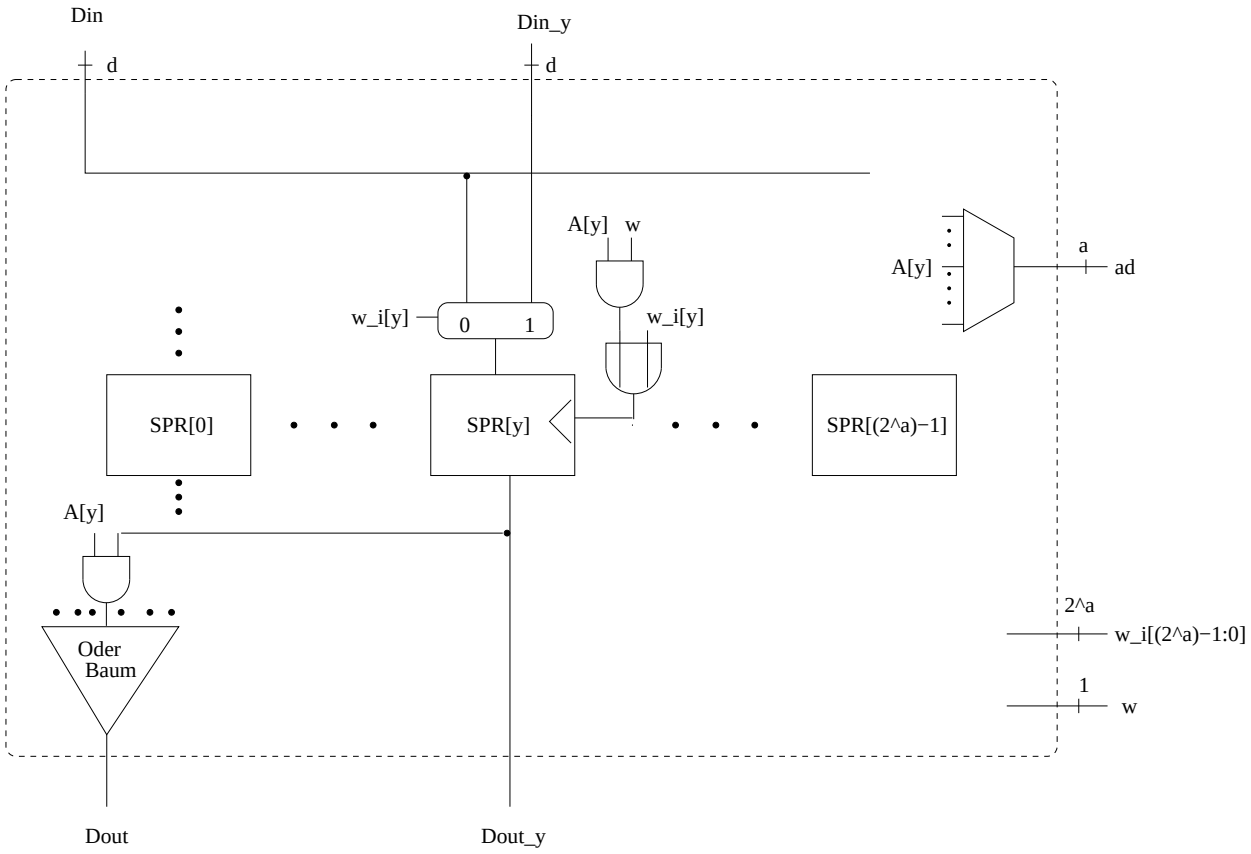


Abbildung 32: Konstruktion des SPR. Anmerkung: Der Übersicht halber, wurde die genaue Verdrahtung nur für ein $y \in \{0, \ldots, 2^a - 1\}$ gezeichnet

Berechnung der Eingangsignale des SPR:

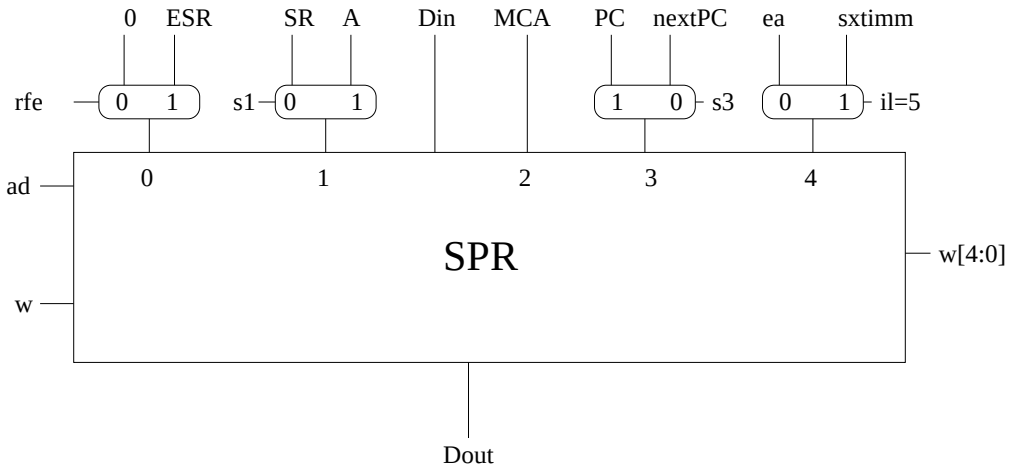


Abbildung 33: Selektierung der Eingangssignale für die individuellen Eingänge

$$\begin{aligned} s_3 &= il = 3 \vee il = 4 \\ &= (MCA[3] \vee MCA[4]) \wedge \overline{\bigvee_{j \leq 2} MCA[j]} \\ s_1 &= (il \geq 5) \wedge movi2s \wedge SA = 0^5 \end{aligned}$$

Für die PC Berechnung ergeben sich folgende Änderungen:

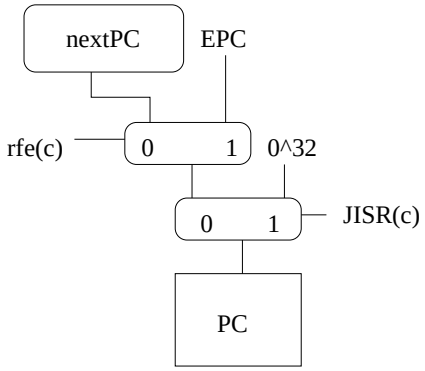


Abbildung 34: Änderungen an der nextPC Berechnung

Außerdem müssen wir das SPR mit dem GPR verdrahten:

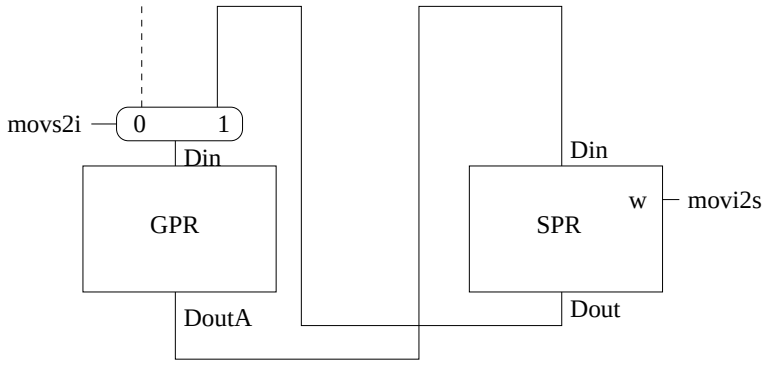


Abbildung 35: Verdrahtung des SPR mit dem GPR

Zusätzlich ändern wir das Schreibsignal für das GPR:

$$gprw = gprw_{alt} \vee movs2i \wedge \overline{JISR \wedge il \in \{1,3,4\}}$$

Das Schreibsignal für den Speicher ändern wir in:

$$mw = mw_{alt} \wedge \overline{JISR \wedge il \in \{2,3,4\}}$$

3.2.7 Physikalische und Virtuelle Maschinen

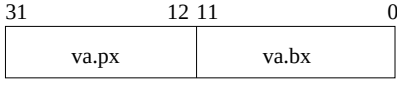
Ab sofort unterscheiden wir physikalische und virtuelle DLX Maschinen:

virtuelle DLX Maschine

Die bisherige DLX Maschine nennen wir virtuelle Maschine. Wir werden einen Mechanismus vorstellen, der jedem Benutzer seine eigene virtuelle Maschine vorgaukelt. Ab sofort nennen wir die Adressen, die von Benutzern ($\neq$ *Betriebssystemkern*) benutzt werden virtuelle Adressen va. Mit diesen greifen die Benutzer auf virtuellen Speicher zu.

Hierzu unterteilen wir den gesamten Speicher in sogenannte Pages (dt. Seiten) à 4 Kilobytes. Die Pages sind aufeinanderfolgend durchnummeriert, beginnend mit der 0. Page an Adresse 0.

Somit können wir virtuelle Adressen va unterteilen in:



- den Pageindex: va.px $\in \{0,1\}^{20}$, also den Index der Page im virtuellen Speicher.
- den Byteindex: va.bx $\in \{0,1\}^{12}$, also den Offset innerhalb einer Page.

physikalische DLX Maschine

Die physikalische Maschine ist nur für den Betriebssystemkern sichtbar. Zusätzlich zur virtuellen Maschine existieren noch 3 weitere SPRs:

i	Name	Beschreibung
5	PTO	Page Table Origin
6	PTL	Page Table Length
7	MODE	

Wir benutzen c.MODE als Abkürzunng für c.SPR(7)[0], da uns nur das letzte Bit interessiert und die anderen stets 0 sind: $c.MODE[31:0] \in \{0^{31}0, 0^{31}1\}$

- Anhand des *MODE* Registers unterscheiden wir, ob der Prozessor im *system mode* (MODE=0) oder im *user mode* (MODE=1) läuft.
  Hiervon hängt auch die Semantik der Befehle ab:

  c.MODE=0 wie bisher

  c.MODE=1 die Assembler Instruktionen *movi2s* und *movs2i* sind nicht erlaubt und führen zu einem *Illegal Instruktion* Interrupt.
  Virtuelle Adressen werden mithilfe der sogenannten *Pagetable* übersetzt.

- *PTO* ist die Startadresse der Pagetable im physikalischen Speicher

- *PTL* ist die Länge der Pagetable in Wörtern, also die Anzahl der Einträge.

3.2.8 Pagetable

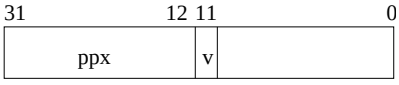
Wir definieren die *pagetable entry address*, also die physikalische Adresse des Pagetable Eintrages, der benötigt wird um eine gegebene virtuelle Adresse *va* zu übersetzen:

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

Wir definieren den zugehörigen *page table entry*:

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

Diesen unterteilen wir in:



- den *physical page index*, also den Index der physikalischen Page, an die der Speicherzugriff des Benutzers umgeleitet wird:

$$ppx(c, va) = pte(c, va)[31:12]$$

- das *valid* Bit, das angibt ob die angeforderte virtuelle Adresse zur Zeit im physikalischen Speicher bereitsteht (valid=1) oder in der Auslagerungsdatei gespeichert ist (valid=0).

$$v(c, va) = pte(c, va)[11]$$

- die restlichen Bits eines *pte* werden für unsere Zwecke nicht benötigt

Schließlich definieren wir zu einer gegebenen virtuellen Adresse va die *physical memory address*, also die physikalische Adresse an die die virtuelle Adresse umgeleitet wird:

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

Der Intuition mag hierfür vielleicht das folgende Bild helfen. Allerdings enthält es keine genaue Semantik und dient nur der Veranschaulichung:

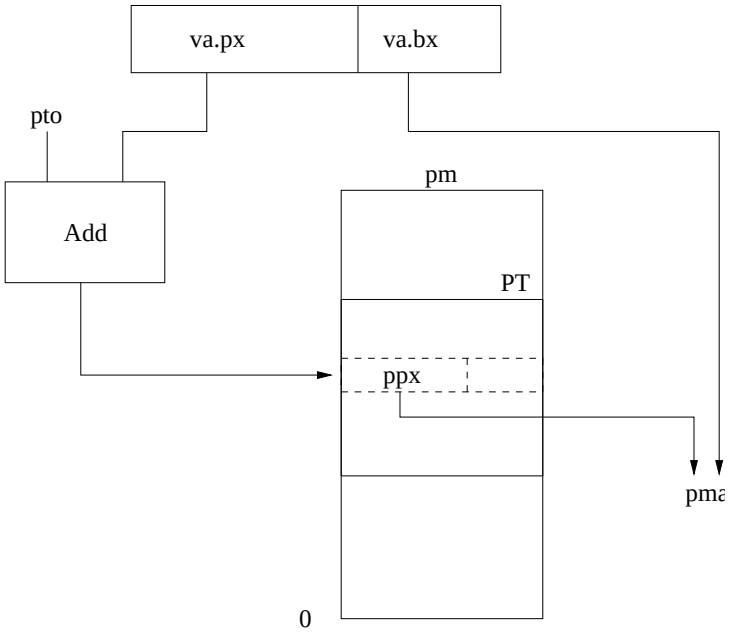


Abbildung 36: Intuitive Adressübersetzung

3.2.9 Page Fault Interrupts

Ein *Page Fault Interrupt* wird generiert, falls:

- beim Übersetzen einer virtuellen Adresse *va* die zugehörige Page nicht gültig, also v(c, va)=0 ist
- ein Benutzer versucht auf eine virtuelle Adresse zuzugreifen, für die es keinen Pagetable Eintrag gibt. Dies nennen wir *pagetable length exception*, *ptle*. Es gilt $ptle(c) = ptlef(c) \vee ptlels(c)$

  Eine *ptle* on fetch liegt vor, falls:

$$ptlef(c) = \langle c.PC.px \rangle \geq \langle c.PTL \rangle$$

  Eine *ptle* on load/store liegt vor, falls:

$$ptlels(c) = \langle ea(c).px \rangle \geq \langle c.PTL \rangle$$

Wir definieren das page fault on fetch Signal:

$$iev(c)[3] = pff(c) = c.MODE \wedge ptlef(c)(\vee\overline{ptlef(c)} \wedge \overline{v(c, c.pc)})$$

Und das page fault on load/store Signal:

$$iev(c)[4] = pfls(c) = c.MODE \wedge (lw(c) \vee sw(c)) \wedge [ptlels(c) \vee \overline{ptlels(c)} \wedge \overline{v(c, ea(c))}]$$

3.2.10 Konstruktion einer MMU

Wir konstruieren eine *Memory Management Unit* die alle Speicherzugriffe im *usermode* des Prozessors mithilfe der Pagetable übersetzt.
Da wir hierzu 2 Zugriffe auf den physikalischen Speicher benötigen um einen Zugriff auf den virtuellen Speicher zu simulieren, unterteilen wir die bisherigen Takte $fetch$ und *execute* jeweils in eine Phase0 und eine Phase1. Dies werden wir anhand eines Signals *phase* unterscheiden. Und zwar wird *phase* aktiv sein, genau während Phase1.

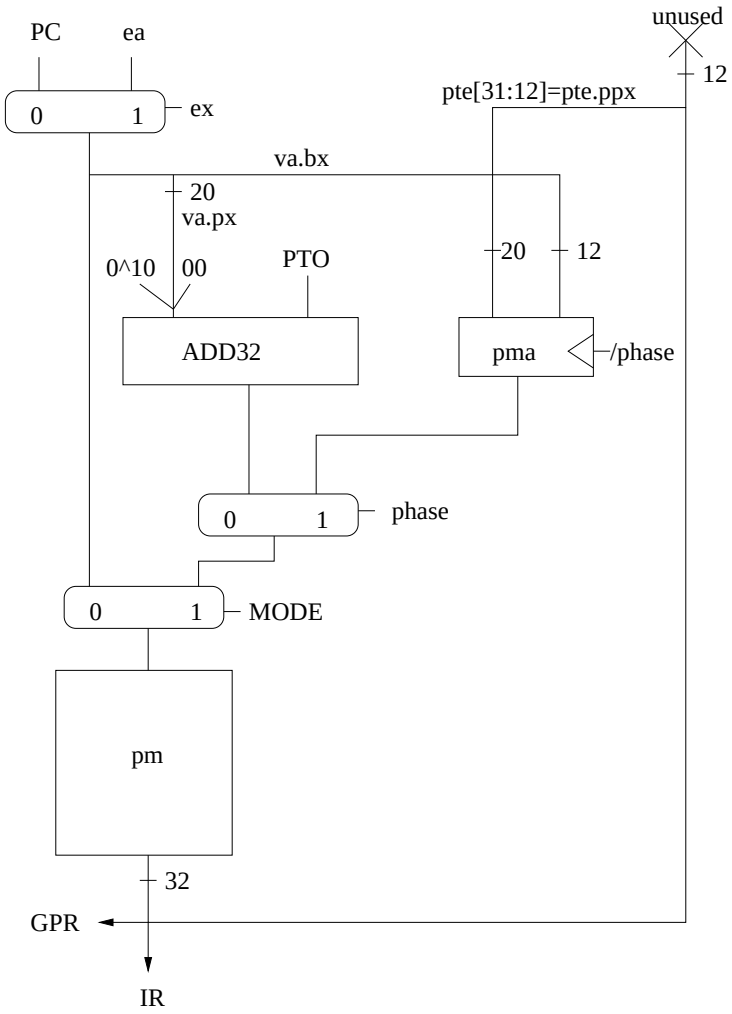


Abbildung 37: Integration der Memory Management Unit

Wir geben die Änderungen am Schaltkreis zur Berechnung des ex Signals, sowie den Schaltkreis zur Berechnung des $phase$ Signals an.

$$exin = \overline{reset} \wedge [\overline{MODE} \wedge \overline{ex} \vee MODE \wedge ((\overline{ex} \wedge phase) \vee (ex \wedge \overline{phase}))]$$

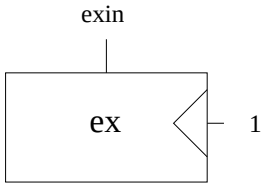


Abbildung 38:

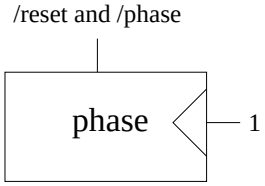


Abbildung 39:

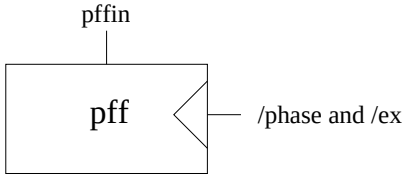


Abbildung 40:

$$pffin = MODE \wedge [ptlef \vee \overline{pte[11]}]$$

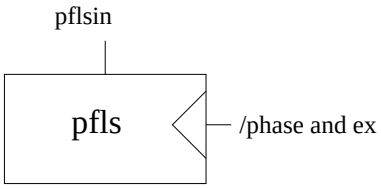


Abbildung 41:

$pflsin = MODE \wedge [ptlels \vee \overline{pte[11]}]$

3.3 CVM Spezifikation - abstrakter Kern k

Wir definieren ein neues Maschinenmodell: Communicating Virtual Maschines (CVM). Es besteht aus:

- k, dem abstrakten Kern, modelliert durch eine C0 Maschine, läuft im *system mode* des Prozessors
- vm(i), den Benutzern, modelliert durch DLX Maschinen, laufen im *user mode* des Prozessors
- D(i), den Devices, also externe Geräte, die an den Prozessor angeschlossen sind, z.B. Festplatte, Keyboard, Netzwerkinterface, Grafikkarte

Eine CVM Konfiguration besteht aus:

c C0 Konfiguration zu k

vm(i) virtuelle DLX Maschine von Benutzer i, i $\in \{1, \ldots, p\}, p$:Anzahl Benutzer

ptl(i) Page Table Length von vm(i), es gilt $ptl(i) \cdot 4096$ = Speichergröße von Benutzer i
$cvm.vm(i).vm : \{a \in \{0,1\}^{32} \mid \langle a \rangle < 4096 \cdot cvm.ptl(i)\} \rightarrow \{0,1\}^8$

cp current Process $\in \{0, \ldots p\}$
cvm.cp = 0 $\rightarrow$ Kern läuft
cvm.cp = i>0 $\rightarrow$ Benutzer i läuft

d(i) Konfiguration von Device d(i)

3.3.1 Next State Funktion des CVM Modells

$\delta(cvm) = cvm'$

- cvm.cp = u > 0, JISR(cvm.vm(u)=0)
  - cvm'.vm(u) = δ_{DLX}(cvm.vm(u))
  - $\forall i \in \{1, \ldots, p\} : i \neq u$: cvm'.vm(i) = cvm.vm(i)

  - cvm'.c = cvm.c
  - cvm'.cp = cvm.cp

- cvm.cp = 0, reset=0, cvm.c.pr=an;r und *an* keine spezielle Funktion
  - cvm.c = δ_{C0}(cvm.c)
  - $\forall i \in \{1, \ldots p\}$: cvm'.vm(i) = cvm.vm(i)
  - cvm'.cp = 0

3.3.2 Spezielle Funktionen des abstrakten Kerns k

cvm.cp = 0, cvm.c.pr = an;r

- an=startcp: startet Benutzerprozess
  in k gibt es eine C0 Variable CP, deren Wert durch den sogenannten Scheduler verwaltet wird. Der Scheduler ist ein Algorithmus, der bestimmt, welcher Benutzerprozess als nächstes läuft. Gewöhnlich ist er als C Funktion des Kerns implementiert.
  $cvm'.cp = va(cvm.c, CP)$

- an = copy(s, r, s1, s2, l): kopiert von s nach r
  s: sender, r: receiver $\in \{1, ..., p\}$
  s_i: Startadressen in virtuellem Speicher des jeweiligen Benutzers. Es muß gelten:
  $s_1 + l < 4096 \cdot$ cvm.ptl(s)
  $s_2 + l < 4096 \cdot$ cvm.ptl(r)
  $cvm'.vm(va(cvm.c, r)).vm_{va(cvm.c,l)}(va(cvm.c, s_2)) = cvm.vm(va(cvm.c, s)).vm_{va(cvm.c,l)}(va(cvm.c, s_1))$
  $cvm'.cp = 0$
  $cvm'.c.pr = r$

- an=alloc(u, x): x neue Pages für Benutzer u
  Restriktionen:
  - $0 \leq \sum_{i=1}^{p} cvm.ptl(i) \leq \frac{4GB}{4KB} = 2^{20}$ um den Speicherplatz, der für die Pagetables benötigt wird, zu beschränken. Die obere Grenze für den virtuellen Speicher aller Benutzerprozesse bezeichnen wir mit TVM (Total Virtual Memory).
  - Zusätzlich verlangen wir, daß x ein Vielfaches von 1MB ist um keinen Speicherplatz auf den Swappages zu verschenken.

  $cvm'.ptl(u) = cvm.ptl(u) + x$

- an=free(u, x): entfernt die x letzten Pages von Benutzer u
  Es gelten dieselben Restriktionen wie bei *alloc*.
  $cvm'.ptl(u) = cvm.ptl(u) - x$

3.3.3 Benuterinterface des Kerns

Mittels der *trap* Instruktion können Benutzer bestimmte Funktionen des Kerns aufrufen. Dies stellt quasi eine Art remote function call von der Benutzermaschine zur Kernmaschine dar.

Die Funktionen des Kerns sind durch die Funktionstabelle spezifiziert. Die Nummer der aufzurufenden Funktion zu einer gegebenen *trap* Instruktion mit immediate Konstante i wird durch die kernel call definition Funktion bestimmt.

$$kcd : [0 : nk - 1] \rightarrow [0 : nf - 1]$$

Wobei nk die Anzahl der für Benutzer zur Verfügung stehenden Funktionen, sogenannte kernel calls, und nf die Anzahl der Funktionen des Kerns sind.

Binäres Interface

Seien
$cvm.cp = u$
$JISR(cvm.vm(u)) = 1$
$il((cvm.vm(u)) = 5$ (trap)
$\langle sxtimm(cvm.vm(u))\rangle = i$
f = FT(kcd(i)).name
n = Anzahl Parameter von f, n ≤ 10

Effekt:
f aufrufen, mit Parametern GPR[1:10] von Benutzer u
$cvm'.c.rd = cvm.c.rd + 1$
$\forall l : 0 \leq l < n \mid top(cvm'.c).ct[l] = cvm.vm(u).gpr(l + 1)$
$rbs(cvm'.c.rd) = cvm.vm(u).gpr(11)$
$cvm'.c.pr = FT(kcd(i).body)\ cvm.c.pr$
Der Rückgabewert der Funktion wird in Register 11 übergeben.

C Interface

TODO

3.4 CVM Implementierung - konkreter Kern K

Wir erweitern $C0$ zu $C0_A$, d.h. wir erlauben Inline Assembler Code innerhalb unserer $C0$ Programme. Hierzu erweitern wir die $C0$ Syntax um einen zusätzlichen Anweisungstyp $asm(u)$, wobei u eine Folge von DLX Assembler Instruktionen sein muß.
Die Codeerzeugung ist einfach:

$$code(asm(u)) = u$$

Die Semantik werden wir später behandeln.

Die Implementierung des abstrakten Kerns k nennen wir den konkreten Kern K. Dieser wird die Datenstrukturen und Funktionen von k, sowie zusätzliche Datenstrukturen und die Implementierung der speziellen Funktionen von k enthalten. Die speziellen Funktionen werden $C0_A$ Programme sein. Eine Konfiguration von K nennen wir *cc*.

3.4.1 Speicheraufteilung in der physikalischen Maschine

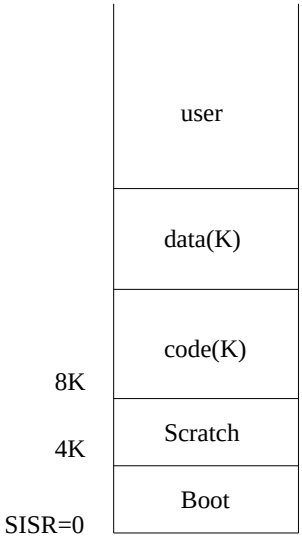


Abbildung 42:

- ROM: bezeichnet einen nichtflüchtigen Speicher (read-only memory). Hierin werden wir einen Teil der Interrupt Service Routine, sowie den Bootloader unterbringen.
- Der Bootloader ist ein Assembler Programm, welches dazu dient beim Anschalten des Rechners (wenn der Speicher bis auf das ROM leer ist) den Kern in den Speicher zu laden.
- code(K) und data(K) bezeichnen Code, sowie die Variablen des Kerns.
- user bezeichnet den Speicherbereich, für den virtuellen Speicher der Benutzer.

3.4.2 Neue Datenstrukturen von K

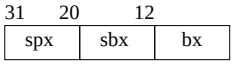
- Pagetable Array
  Der Kern speichert die Pagetables aller Prozesse in einem Array. Die Größe dieses Arrays ist durch TVM eingeschränkt: Die Größe läßt sich als Größe eines *pte* multipliziert mit der (durch TVM bestimmten) maximalen Anzahl der virtuellen Pages berechnen.

$$\underbrace{size(pte)}_{4} \cdot \underbrace{\#pages}_{TVM/PAGESIZE} = 4 \cdot TVM/4K = 4MB$$

  TODO Bild einfügen

- Swap Memory Page Table
  Analog zur Unterteilung des physikalischen Speichers in Pages unterteilen wir den Swap Speicher in swap pages der Größe 1 MB. Zur Adressierung der Swap Pages unterteilen wir den

Pageindex einer virtuellen Adresse weiter in einen swap page index und einen swap byte index.

va = 

Analog zur Berechnung des physikalischen Pageindex aus dem Pageindex einer Adresse mittels der Pagetable, benutzen wir die Swap Pagetable um zu einem gegebenen Swap Pageindex den physikalischen Swap Pageindex zu bestimmen.

- Usedlist: verlinkte Liste der benutzen Pages.

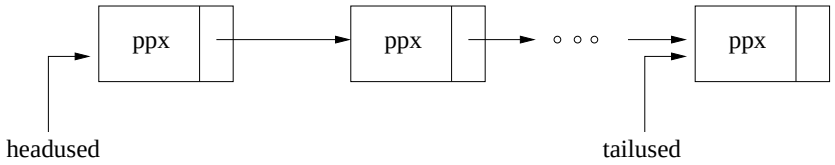


Abbildung 43: Die usedlist wird als *FIFO* verwendet, d.h. Elemente werden vor *headused* eingefügt und bei *tailused* entnommen

- Freelist: verlinkte Liste der freien Pages.
- SwapFreelist: verlinkte Liste der freien Swap Pages.
- SwapUsedlist:verlinkte Liste der benutzten Swap Pages.
- Process Controll Blocks
  Der Kern speichert für jeden Benutzer einen sogenannten Process Control Block. Hierin werden die Werte der Prozessorregister gespeichert wenn der Process gerade nicht aktiv ist. Zusätzlich wird noch die Anfangsadresse der Swap Page Table, sowie deren Länge gespeichert. Die Process Control Blocks werden in einem Array gespeichert, welches über die Benutzernummer indiziert wird.

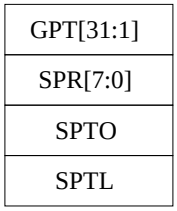


Abbildung 44:

3.4.3 Konfiguration der physikalischen Maschine

Eine Konfiguration der physikalischen Maschine c_p kodiert die Konfigurationen mehrerer virtueller Maschinen.
Hierzu definieren wir für globale Variablen e:

$$va(c_p, e) = c_p.pm_4(aba^0(cvm.c, e)) = c_p.pm_4(aba(e))$$

da sich der Speicherort von globalen Variablen nicht ändert ist die Adresse unabhängig von aba.

Unser Ziel ist eine Schritt für Schritt Simulation:
$cvm^0, cvm^1, \ldots$ Rechnung von cvm Maschine
$aba^0, aba^1, \ldots$ Allokationsfunktionen
$c_p^0, c_p^1, \ldots$ Rechnung der physikalischen DLX Maschine
$S(0), S(1), \ldots$ Schrittzahlenn

so daß $\forall i : c_p^{S(i)}$ *kodiert* cvm^i

- $\bigwedge_u c_p^{S(i)}$ *kodiert* $cvm^i.vm(u)$
- $consis(c^i, aba^i, c_p^{S(i)})$

Adressübersetzung für Benutzer u

alte Def.	neue Def.
$d.PTO$	$pto(c_p, u)$
$d.PTL$	$ptl(c_p, u)$

Wir wiederholen die Definitionen zur Adressübersetzung:
$ptea(c_p, u, va) = pto(c_p, u) + 4 \cdot va.px$
$pte(c_p, u, va) = c_p.pm_4(ptea(c_p, u, va))$
$ppx(c_p, u, va) = pte(c_p, u, va)[31:12]$
$v(c_p, u, va) = pte(c_p, u, va)[11]$
$pma(c_p, u, va) = ppx(c_p, u, va) \circ va.bx$

Äquivalent definieren wir:
$sptea(c_p, u, va) = spto(c_p, u) + 4 \cdot va.spx$
$spte(c_p, u, va) = c_p.pm_4(sptea(c_p, u, va))$
$pspx(c_p, u, va) = spte(c_p, u, va)[31:20]$
$sma(c_p, u, va) = pspx(c_p, u, va) \circ va.sbx \circ va.bx$

3.4.4 B-Relation

Wir definieren die sogenannte B-Relation:

$$B(c_p, cvm, u) \equiv c_p \textit{ kodiert } cvm.vm(u)$$

1. Register R von cvm.vm(u) werden richtig kodiert:
   R ∈ {gpr(i), spr(j)}
   $$cvm.vm(u).R = \begin{cases} c_p.R & : cvm.cp = u \\ va(c_p, pcb[u].R) & : \text{sonst} \end{cases}$$

2. Speicher wird richtig kodiert:
$$cvm.vm(u).vm(va) = \begin{cases} c_p.pm_4(pma(c_p,u,va)) & : v(c_p,u,va) \\ c_p.sm(sma(c_p,u,va)) & : \text{sonst} \end{cases}$$

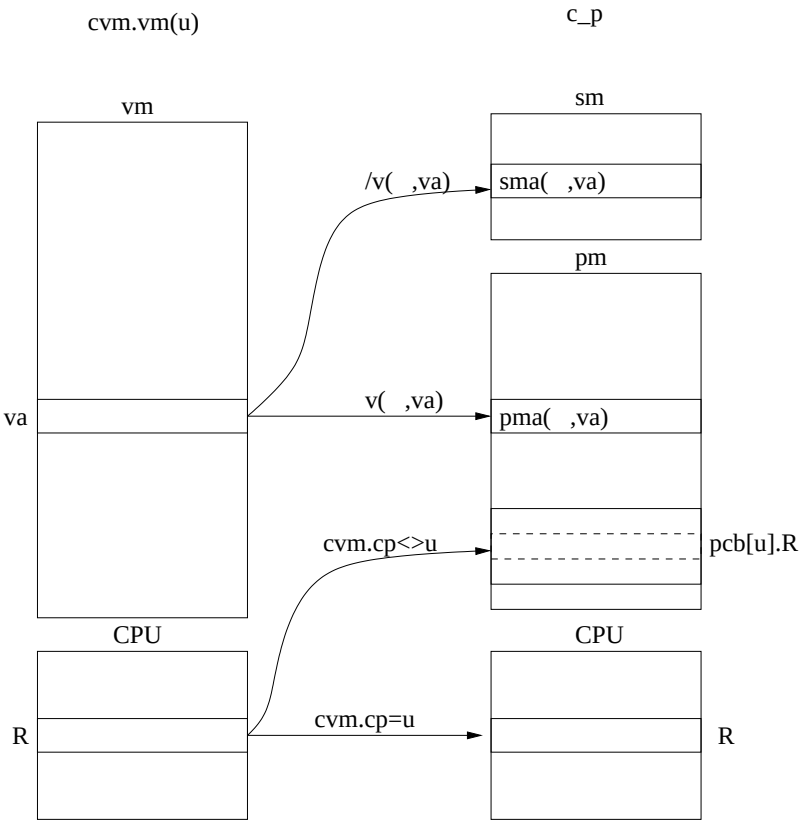


Abbildung 45: B-Relation

3.4.5 Simulation von cvm

Wir zeigen die Simulation eines Schrittes von cvm.vm(u)

1. Fall: Kern aktiv, keine spezielle Funktion des Kerns aktiv. Die Korrektheit der Simulation folgt aus der Compiler Korrektheit.

2. Fall: Benutzer u aktiv, kein Interrupt($cvm.cp = u, JISR(cvm.vm(u) = 0)$):
physikalische Maschine rechnet in usermode: $c_p.MODE = 1$
MMU übersetzt va in $pma(c_p, u, va)$
Die physikalische und die virtuelle Maschine machen je einen Schritt. Es gilt: $B(c_p', cvm', u)$
Die Korrektheit der Simulation folgt aus der Korrektheit der MMU

3. Fall Benutzer u aktiv, Interrupt($cvm.cp = u, JISR(cvm.vm(u) = 1)$):

   - physikalische Maschine wechselt in systemmode und springt zur Startadresse der Interrupt Service Routine ($c_p.MODE = 0, c_p.pc = 0$). CVM Maschine wechselt den *current process* auf den Kern($cvm.cp = 0, cvm'.c.pr =$ 'ISR' $; cvm.c.pr$).
   - code0:
     Zu Beginn der ISR steht der sogenannte *code*0, der überprüft, ob der Grund des Interrupts *reset* ist. Falls ja, wird als nächstes der *Bootloader* aufgerufen, der den Kern von der Festplatte in den Speicher lädt. Andernfalls wird *process save* aufgerufen.
     Zu Beginn der ISR stehen noch die Werte des Benutzers, bei dem der Interrupt aufgetreten ist, in den Registern und müssen in den passenden PCB gesichert werden. Um die Adresse des passenden PCB zu berechnen werden allerdings Register benötigt. Deshalb werden die Werte zweier Register in die sogenannte *Scratch* Page gesichert. Diese Register stehen der ISR nun für Berechnungen zur Verfügung:

     ```
     sw RD=1, RS1=0, imm=4096
     sw RD=2, RS1=0, imm=4100
     movs2i RD=1, SA=ECA
     ```

     Anmerkung: 4096 und 4100 sind Adressen innerhalb der Scratch page.

     teste il=0 (Reset)

     $$c_p.pc = \begin{cases} start(Bootloader) & : \text{il} = 0 \\ start(process\ save) & : \text{sonst} \end{cases}$$

   - process save:
     Sichere die Werte der Benutzerregister in den entsprechenden PCB.
     (a) $\forall R \in \{gpr(i), spr(j)\}, i \neq 1, 2 : va(cc, PCB[u].R) = cvm.vm(u).R$
     bzw. für die physikalische Maschine:
     $\forall R \in \{gpr(i), spr(j)\}, i \neq 1, 2 : c_p.pm_4(aba^0(PCB[u].R) = c_p.R$
     wobei gilt: $u = va(cc, CP)$, bzw. $u = va(c_p, aba(CP))$ Nummer des zuletzt gestarteten Prozesses.
     (b) Kopiere die Werte der zwei zwischengespeicherten Register aus der Scratch Page in den PCB:
     $PCB[u].gpr(1) = pm_4(4096)$
     $PCB[u].gpr(2) = pm_4(4100)$

   - lade einige Register des Kerns aus PCB[0]:
     $c_p.gpr(28) = va(c_p, PCB[0].gpr(28))$
     $c_p.gpr(29) = va(c_p, PCB[0].gpr(29))$
     $c_p.gpr(30) = va(c_p, PCB[0].gpr(30))$
     Im Fall eines unterbrechbaren Kerns müßten alle Register des Kerns restauriert werden. Da unser Kern nicht unterbrechbar ist ($\forall i : cvm^i.c.SR = 0^{32}$) müssen wir nur *Stack*− und *Heappointer* wiederherstellen.
   - Dispatcher:
     Der Dispatcher entscheidet anhand des Interrupt Levels des Prozesses, der als letztes

aktiv war, welche Funktion des Kerns aufgerufen wird. Hierzu berechnet er den Interrupt Level aus $PCB[CP].ECA$:

$$il = min\{k \mid PCB[CP].ECA[k] = 1\}$$

und ruft dann die entsprechende Funktion auf.

- $il = 0$(Reset): Wird bereits in $code0$ abgefangen.
- $il \in \{1, 2\}$(Misalignment, Illegal Instruction): Abbruch des Prozesses.
- $il = 3$(Page Fault Fetch):
  Bestimme virtuelle Adresse va, die den Page Fault erzeugt hat: $va = PCB[CP].EPC$.
  Rufe Pagefaulthandler mit va als Argument auf und starte danach den Prozess, der den Pagefault erzeugt hat erneut.

  $$cvm'.c.pr = pfh(va); startCP; cvm.c.pr$$

- $il = 4$(Page Fault Load/Store):
  Analog zu pff, allerdings ist $va = PCB[CP].EDATA$

  $$cvm'.c.pr = pfh(va); startCP; cvm.c.pr$$

- $il = 5$(Trap):
  Die aufzurufende Funktion wird in der $trap$ Instruktion als immediate Konstante kodiert und steht im $EDATA$ Register zur Verfügung. Seien:
  $i = PCB[CP].EDATA$
  $f = FT(kcd(i)).name$
  $n =$ Anzahl der Parameter von f

  $$cvm'.c.pr = f(PCB[CP].gpr(1), \ldots PCB[CP].gpr(n)); startCP; cvm.c.pr$$

- $il = 6$(Overflow): Starte Funktion, die mit einem Softwarealgorithmus höherer Genauigkeit die Berechnung ohne Overflow durchführt (Hier nicht behandelt). Anschließend starte wieder Benutzerprozeß.
- $il > 6$(Externer Interrupt): Rufe entsprechenden Device Driver auf. Anschließend starte wieder Benutzerprozeß.

3.4.6 Implementierung

startCP ($C0_A$ *Funktion*)

- Kopiere *Stack−* und *Heappointer* des Kerns aus $c_p.gpr$ in $PCB[0]$.
- Lade Register von $vm(CP)$ aus $PCB[CP]$ in physikalische Register.

Pagefault Handler

Die Funktion $pfh(unsigned\ va, unsigned\ i)$ nimmt als Parameter die virtuelle Adresse va und die Nummer des Benutzers i, der den Page Fault verursacht hat.

- Falls keine freie Page im physikalischen Speicher existiert:

  - Bestimme Page, die ausgeswappt wird um Platz im Speicher zu schaffen: $VICTIM = tailused.ppx$
  - Kopiere den Inhalt der $VICTIM$ Page ins Swap Memory.
  - Markiere den Pagetable Eintrag, der auf die $VICTIM$ Page gezeigt hat als invalid.
  - Verschiebe $VICTIM$ in die $freelist$

- Bestimme eine freie Page: $x = headfree.ppx$
- Lade den Inhalt der gewünschten Page aus dem Swap Memory in Page x.
- Setze den zugehörigen Pagetable Eintrag auf valid.
- verschiebe x in die $usedlist$

Aus der $FIFO$ Implementation der $usedlist$ folgt:

- zuletzt geladene Seite ist nie $VICTIM$
- nach 2 Pagefaults derselben Instruktion sind beide benötigten Pages im Speicher
- spätestens beim 3. Start tritt kein Pagefault mehr auf

3.4.7 Semantik von $C0_A$

Es gilt $consis(cc, aba, d^i)$:

$$\delta_{C0_A}(cc, d) = \begin{cases} \ldots & : cc.pr = asm(u); r \\ \delta_{C0}(cc) & : sonst \end{cases}$$

Im Fall einer Inline Assembler Instruktion muß man folgende Fallunterscheidung machen. Die Konfiguration cc^i der $C0$ Maschine kann nur von sw Assembler Instruktionen geändert werden. Wobei als Einschränkung gilt, daß nur globale Variablen geändert werden dürfen.

- $\overline{sw(d^i)} \Rightarrow cc^{i+1} = cc^i$
- $sw(d^i) \wedge ea(d^i) = aba(cc, id) \Rightarrow va(cc^{i+1}, id) = d^i.gpr(RD(d^i))$
- $sw(d^i) \wedge \overline{ea(d^i) = aba(cc, id)} \Rightarrow cc^{i+1} = cc^i$

3.4.8 I/O Devices

Ein I/O Device besteht aus Sicht des Prozessors aus einer Menge konsekutiver Speicheradressen, sogenannten I/O Ports. Nachfolgend steht eine Übersicht über die I/O Ports der Festplatte:

- Datenpuffer: für Austausch von Pages zwischen CPU und Device.
- Command Register: Für Befehle an Device: z.B. Schreibe Inhalt des Puffers an die Swap Memory Adresse sma=y, oder lies eine Page von sma=y in den Puffer.
- Status Register: zum Anzeigen, daß die Festplatte gerade damit beschäftigt ist, in den Puffer zu schreiben oder daraus zu lesen. Alternativ hierzu kann die Festplatte auch einen externen Interrupt auslösen, sobald sie fertig ist