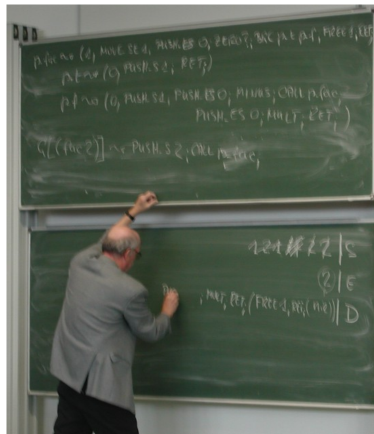
# Organisation und Architektur von Rechnern



Mitschrift von www.kuertz.name

**Hinweis:** Dies ist **kein offizielles Script**, sondern nur eine private Mitschrift. Die Mitschriften sind teweilse **unvollständig, falsch oder inaktuell**, da sie aus dem Zeitraum 2001–2005 stammen. Falls jemand einen Fehler entdeckt, so freue ich mich dennoch über einen kurzen Hinweis per E-Mail – vielen Dank!

Klaas Ole Kürtz (klaasole@kuertz.net)

# Inhaltsverzeichnis


**0 Begriffliches** **1**

**1 Organisation maschineller Berechnungsabläufe** **3**
1.1 Schichtenmodell . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.2 miniScheme SE(M)CD-Maschine . . . . . . . . . . . . . . . . 3
1.2.1 Sprachumfang . . . . . . . . . . . . . . . . . . . . . . . 3
1.2.2 SE(M)CD-Maschine . . . . . . . . . . . . . . . . . . . . 4
1.2.3 Beispiel für den Traversier-Mechanismus . . . . . . . . 5
1.2.4 Auswerte- bzw. Reduktionsregeln . . . . . . . . . . . . 5
1.2.5 Anwendung primitiver Funktionen und Prozeduren . . 7
1.3 Instruktions- und Codeausführende Scheme-Maschine . . . . 8
1.3.1 Scheme-Quellcode . . . . . . . . . . . . . . . . . . . . . 8
1.3.2 Instruktionssatz der SECDH-Maschine . . . . . . . . . 8
1.3.3 Compiler MiniScheme $\rightarrow$ SECDH-Code . . . . . . . . 9
1.3.4 Beispiel: Compilierung/Ausführung $(fac\,2)$ . . . . . . . 9
1.4 Codeausführende Scheme-Maschine, 2. Versuch . . . . . . . 10
1.4.1 Behandlung von *letrec*-Ausdrücken . . . . . . . . . . . 10
1.4.2 Modifizierte SECDH2-Maschine . . . . . . . . . . . . . 12
1.4.3 Instruktionssatz . . . . . . . . . . . . . . . . . . . . . . 14
1.4.4 Modifikation des Compilers . . . . . . . . . . . . . . . . 14
1.4.5 Übersetzungsbeispiel . . . . . . . . . . . . . . . . . . . 14

**2 Rechnerarchitekturen** **17**
2.1 Architekturklassen . . . . . . . . . . . . . . . . . . . . . . . . 17
2.2 Instruktionsklassen und Ausführungsphasen . . . . . . . . . . 18

**3 MC680X0-Architektur (CISC)** **21**
3.1 Abbildung der Laufzeitumgebung auf Speicherbereiche . . . . 21
3.2 Registerkonfiguration . . . . . . . . . . . . . . . . . . . . . . . 21
3.3 Datenformate . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.4 Adressierungsmodes . . . . . . . . . . . . . . . . . . . . . . . 23
3.5 Instruktionen/Instruktionsformate . . . . . . . . . . . . . . . 24
3.6 Organisation von Prozedur- und Funktionsaufrufen . . . . . . 25
3.7 Assemblerprogrammierung . . . . . . . . . . . . . . . . . . . . 27
3.8 Konvertierung symbolischer in numerische Adressen . . . . . . 28
3.9 Segmentierung, Linking und Paging . . . . . . . . . . . . . . . 28

**4 Systemkonfiguration des MC86000** **32**
4.1 Das Unterbrechungssystem . . . . . . . . . . . . . . . . . . . . . 32
4.1.1 Unterbrechungsvektoren . . . . . . . . . . . . . . . . . . 32
4.1.2 Programmstatus/Ausführungsmodi . . . . . . . . . . . 33
4.1.3 Unterbrechungsbehandlung . . . . . . . . . . . . . . . . . 34
4.1.4 Maschinelle Bearbeitung der externen Interrupts . . . . 34
4.2 Ein- und Ausgabe-Vorgänge . . . . . . . . . . . . . . . . . . . 36

**5 SPARC-RISC-Architektur** **39**
5.1 Registersatz . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.2 Instruktionsformate und Instruktionen . . . . . . . . . . . . . 40
5.3 Instruktionsausführung . . . . . . . . . . . . . . . . . . . . . . 40
5.4 Behandlung von Pipeline-Abhängigkeiten . . . . . . . . . . . . 40

**A Graphiken, Schemata etc.** **42**
A.1 genereller Rechneraufbau . . . . . . . . . . . . . . . . . . . . . 42
A.2 SEMCD-Regelwerk . . . . . . . . . . . . . . . . . . . . . . . . 43
A.3 Compiler MiniScheme → SECDH-Code . . . . . . . . . . . . 44
A.4 SECDH-Regelwerk . . . . . . . . . . . . . . . . . . . . . . . . 45
A.5 Architekturen . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
A.6 generelle Architektur des MC86K . . . . . . . . . . . . . . . . 47
A.7 Paralleles Interface . . . . . . . . . . . . . . . . . . . . . . . . 48
A.8 Serielles Interface . . . . . . . . . . . . . . . . . . . . . . . . . 49

# 0 Begriffliches

- **Organisation**[1] beschreibt das geordnete arbeitsteilige Zusammenwirken mehrerer Komponenten eines Systems bei der Durchführung zielgerichteter Handlungen.
- **wesentlich sind:** Disziplin bzw. Mechanismen des geordneten Zusammenwirkens
- **Agenten/Aktoren** (aktive Komponenten) führen Transaktionen und Operationen auf (Darstellungen von) Nachrichten, Dokumenten und Objekten einer realen oder virtuellen Welt aus
- **Kanäle** (passive Komponenten) dienen als Traeger von Nachrichten und Objekten beim Transport zwischen Komponenten
- **System** ist Menge aktiver oder passiver Komponenten, die zweckgebunden miteinander kooperieren oder kommunizieren, wird beschrieben durch
  - Eigenschaften der Komponenten und deren Wechselwirkungen
  - die Festlegung von Systemgrenzen gegenueber seiner Umgebung
  - die Wechselwirkungen mit seiner Umgebung bzw. die Beziehungen zwischen Systemeingabe und Systemausgabe

**Rechnerorganisation** betrifft:

- Organisation von alorithmisch/deklarativ spezifizierten Berechnungs- oder Entscheidungsvorgängen oder Prozessen, → programmausführendes System
- Aktoren sind vom System unterstützte Operatoren
- Kanäle sind Speicherbereiche, Laufzeitumgebungen zur Kommunikation von Operanden zwischen Operatoren
- Ablaufplanung bzw. Ressourcenverwaltung für das programmausführende System bzw. Abwicklung von Kommunikationsvorgängen bzw. Verwaltung großer Datenbestände, → Betriebssystem

[1] Unterhalten Sie sich nicht mit dem Studienberater. Das ist verschwendete Zeit. (...) Da wird Ihnen nur Käse erzählt. Sie müssen das bei mir versuchen und nicht bei diesen Quatschköpfen. (...) Wenn sich Leute einschreiben und verschwinden dann in der Mitte das Semesters - das merken wir uns. Da wird 'ne schwarze Liste geführt.

- Aktoren sind promäre/sekundäre Scheduler für Prozesse, Transaktionen, ...
- Kanäle sind diverse Datenstrukturen zur Realisierung von Prozess, Ressource, Systemzuständen

**Rechner(System)Architektur** betrifft:

- Gestaltung/Festlegung des Erscheinungsbildes der Schnittstellen von Rechnersystemen gegenüber dem system-nahen Benutzer → System(kern)programmierer, Implementierer von Compiler-backends/Code-Generatoren
- **Architektur** des programmausführenden Systems ist definiert durch
  - nicht privilegierten Instruktionssatz/Adressierungsmodes/Elementadatentypen (Formate)
  - direkt zugängliche Ressourcen (Funktionseinheiten, Register, Speicherbereiche, ...)
- Das **Betriebssystem** ist definiert durch
  - privilegierten Instruktionssatz (Prozessmanagement, i/o-Vorgänge, Prozess-Kommunikation)
  - Unterbrechungssystem, Prozesswechselmechanismen
- Architekturen definieren **Systemfamilien** mit identischer Funktionalität, Maschinenprogramme werden auf allen Modellen der Familie auf gleiche Weise ausgeführt und produzieren exakt gleiche Ergebnisse

Siehe auch Folie im Netz bzw. A.1

# 1 Organisation maschineller Berechnungsabläufe

## 1.1 Schichtenmodell

- spezifisch algorithmisch oder deklarativ über höhere Programmiersprachen (HLL = high level language, z.B. Scheme, Java, C, Prolog), mathematische oder algebraische Notation

- Semantische Lücke zur Maschinenebene (dort nur binäre Zeichenketten), Überbrückung:

  $$L_0 \xrightarrow{P_0} L_1 \longrightarrow ... \longrightarrow L_i \xrightarrow{P_i} L_{i+1} \longrightarrow ... \xrightarrow{P_{n-1}} L_n$$

  Ein abstrakter Prozessor $P_i$ kann

  - als Interpreter jedes Konstrukt der Sprache $L_i$ durch eine Routine in $L_{i+1}$ ausführen (wird ausgeführt durch $P_{i+1}$)
  - als Compiler jedes Programm in $L_i$ als Ganzes in Programm in $L_{i+1}$ übersetzen

Siehe Folie zur Schichtung in Rechnern.

## 1.2 miniScheme SE(M)CD-Maschine

### 1.2.1 Sprachumfang

- atoms: Variablen, Konstanten, Operationen
- `(e0 e1 ... en)` (Applikationen, Listen)
- `(lambda (u1 ... un) eb)` (Abstraktion)
- `(if e0 e1 e2)` (Conditional)
- `(define f e)` (bindet Wert von $e$ an $f$)
- `(let ((v1 e1) ... (vn en)) eb)` (bindet Variablen in $e_b$)
- `(letrec ((f1 e1) ... (fm em)) eb)` (bindet Variablen rekursiv in $e_1$ bis $e_m$ und in $e_b$)

Semantik über abstrakten Evaluator $EVAL[|e|]$:

- $EVAL[|atom|] = atom$
- $EVAL[|(e_0...e_n)|] = EVAL[|(EVAL[|e_0|])...EVAL[|e_n|]|]$
- $EVAL[|(lambda\ (u_1...u_n)\ e_b)|] = (lambda\ (u_1...u_n)\ e_b)$
- $EVAL[|(if\ e_0\ e_1\ e_2)|] = \begin{cases} EVAL[|e_1|] & \text{falls} \quad EVAL[|e_0|] = \#t \\ EVAL[|e_2|] & \text{falls} \quad EVAL[|e_0|] = \#f \\ \perp undefined & \text{sonst} \end{cases}$
- $EVAL[|(define\ f\ e)|] = f \leftarrow EVAL[|e|]$
- $EVAL[|(let\ ((v_1\ e_1)...(v_n\ e_n))\ e_b)|] = EVAL[|e_b[v_1 \leftarrow EVAL[|e_1|]...]|]$
- $EVAL[|(letrec\ ((f_1\ e_1)...(f_m\ e_m))\ e_b)|]$
  $= EVAL[|e_b[...f_i \leftarrow EVAL[|e_i[...f_i \leftarrow (letrec(...)f_i)]|]]|]$
- $EVAL[|((lambda\ (u_1...u_n)\ e_b)\ e_1...e_m)|]$
  $= \begin{cases} EVAL[|e_b[u_1 \leftarrow EVAL[|e_1|]...]|] & \text{falls} \quad n = m \\ \perp undefined & \text{sonst} \end{cases}$

Beispiele:

```
(define foo (lambda (u v w) (u (v w w) w)))
(foo + + 3) = 9
((foo + +) 3) = #<error>
((foo +) + 3) = #<error>
(foo + +) = #<error>
(define foq (lambda (u) (lambda (v) (lambda (w) (u (v w w) w)))))
(foq + + 3) = error
(((foq +) +) 3) = 9
((foq +) +) = #<procedure> (lambda (w) (u (v w w) w)) [u <- +, v <- +]
(foq +) = #<procedure> (lambda (v) (lambda (w) (u (v w w) w))) [u <- +]
```

Als *Abschluß (closure)* wird die Kombination aus Umgebung (Environment) und Ausdruck in der Umgebung bezeichnet, z.B. `[[y <- 3] (+ y 1)]`. Die Substitution der Variablen im Ausdruck durch die Werte in der Umgebung wird verschoben, bis der Ausdruck selbst weiter ausgewertet wurde.

### 1.2.2 SE(M)CD-Maschine

- SE(M)CD-Maschine von Landin (ca. 1964) als abstrakter Evaluator erfunden

- interne Darstellung von SCHEME-Ausdrücken → Konstruktorsyntax

| | | | |
|---|---|---|---|
| Atom | *atom* | $\Longrightarrow$ | $K^{(n+1)} e_0\ e_1\ ...\ e_n$ |
| Lambda | $(lambda\ (u_1\ ...\ u_n)\ e_b)$ | $\Longrightarrow$ | $\lambda^{(n+1)}\ u_1\ ...\ u_n\ e_b$ |
| Applikation | $(e_0\ e_1\ ...\ e_n)$ | $\Longrightarrow$ | $\underbrace{\overline{@}^{(n+1)}}_{Applikator}\ \underbrace{e_n\ ...\ e_1}_{Operanden}\ \underbrace{e_0}_{Operator}$ |

- 

| | Stack | Items auf dem Stack |
|---|---|---|
| S | Werte-Stack | Closure, Atoms: $[E\lambda^{(n+1)}u_1...u_n\ e_b]$ |
| E | Environment Stack | Variablen-Werte-Paare $\langle v\ e\rangle$ |
| M | Zwischenstack für Konstruktoren | $@^{(n+1)}$ (z.T. $\lambda^{(n+1)}$) |
| C | Codestack (Ausdrucksstack) | Konstruktorausdr., Ausdruckskompon. |
| D | Dump z. Speichern v. Maschinenzuständen | Maschinenzustände (E, C, D) |

- Arbeitsweise der Maschine wird beschrieben durch Transformationsfunktion $\tau$, also eine Menge von Transformationsregeln, die Zustände in Zustände überführen ($\{|guard\}$ ist dabei eine Zusatzbedingung für Anwendbarkeit der Regel):

$$\tau(S, E, M, C, D)\{|guard\} \mapsto (S', E', M', C', D')$$

### 1.2.3 Beispiel für den Traversier-Mechanismus

Traversieren von Ausdrücken von $C$ nach $S$ unter Zuhilfenahme von $M$:

$$\begin{array}{r|c} & S \\ & M \\ \kappa^{(3)}e_0e_1\kappa^{(2)}e_2e_3 & C \end{array} \mapsto \begin{array}{r|c} & S \\ \kappa^{(3|3)} & M \\ e_0e_1\kappa^{(2)}e_2e_3 & C \end{array} \mapsto \begin{array}{r|c} e_1^te_0^t & S \\ \kappa^{(3|1)} & M \\ \kappa^{(2)}e_2e_3 & C \end{array}$$

$$\mapsto \begin{array}{r|c} e_1^te_0^t & S \\ \kappa^{(2|2)}\kappa^{(3|1)} & M \\ e_2e_3 & C \end{array} \mapsto \begin{array}{r|c} e_3^te_2^te_1^te_0^t & S \\ \kappa^{(2|0)}\kappa^{(3|1)} & M \\ & C \end{array} \mapsto \begin{array}{r|c} \kappa^{(2)}e_3^te_2^te_1^te_0^t & S \\ \kappa^{(3|0)} & M \\ & C \end{array}$$

$$\mapsto \begin{array}{r|c} \kappa^{(3)}\kappa^{(2)}e_3^te_2^te_1^te_0^t & S \\ & M \\ & C \end{array}$$

wobei $e^t = e_{\text{transponiert}}$ (links-rechts-vertauscht).

### 1.2.4 Auswerte- bzw. Reduktionsregeln

(siehe auch Folie im Netz bzw. A.2):

(0) $(S, E, M, nil, (E', C', D'))$ (sog. *return configuration*)

$\Rightarrow (S, E', M, C', D')$

(1) $([E'\,\lambda^{(2)} u_n e_b] : e_a^t : S, E, \overline{@}^{(2|0)} : M, C, D)$

$\Rightarrow (S, \langle u_n e_a \rangle : E', M, e_b : nil, (E, C, D))$

(2) $([E'\,\lambda^{(n-i+2)} u_1 ... u_n e_b] : e_a^t : S, E, \overline{@}^{(j|0)} : M, C, D) \wedge (j = (n-i+2) > 2)$

$\Rightarrow ([\langle u_i e_a \rangle : E'\,\lambda^{(n-i+1)} u_{i+1} ... u_n e_b] : S, E, \overline{@}^{(j-1|0)} : M, C, D)$

(2a) $([E' \lambda^{(n)} u_1 ... u_n e_b] : e_1 : ... : e_m : S, E, \overline{@}^{(m|0)} : M, C, D) \wedge (m = n)$

$\Rightarrow (S, \langle u_n e_n \rangle : ... : \langle u_1 e_n \rangle : E', M, e_b : nil, (E, C, D))$

(2b) $([E' \lambda^{(n)} u_1 ... u_n e_b] : e_1 : ... : e_m : S, E, \overline{@}^{(m|0)} : M, C, D) \wedge (m \neq n)$

$\Rightarrow (error_m essage, -, -, -, -)$

(3) $(S, E, nil, v : C, D)$

$\Rightarrow (lookup(v, E) : S, E, nil, C, D)$

(4) $(S, E, \overline{@}^{(n|i)} : M, v : C, D) \wedge (i > 0)$

$\Rightarrow (lookup(v, E) : S, E, \overline{@}^{(n|i-1)} : M, C, D)$

(5) $(S, E, nil, \lambda^{(n+1)} u_1 ... u_n e_b : C, D)$

$\Rightarrow ([E\,\lambda^{(n+1)} u_1 ... u_n e_b] : S, E, nil, C, D)$

(6) $(S, E, \overline{@}^{(m|i)} : M, \lambda^{(n+1)} u_1 ... u_n e_b : C, D)$

$\Rightarrow ([E\,\lambda^{(n+1)} u_1 ... u_n e_b] : S, E, \overline{@}^{(m|i-1)} : M, C, D)$

(7) $(S, E, M, \kappa^{(n)} : C, D)$

$\Rightarrow (S, E, \kappa^{(n|n)} : M, C, D)$

(8a) $(S, E, nil, atom : C, D)$

$\Rightarrow (atom : S, E, nil, C, D)$

(8b) $(S, E, \kappa^{(n|i)} : M, atom : C, D) \wedge (i > 0)$

$\Rightarrow$ $(atom : S, E, \kappa^{(n|i-1)} : M, C, D)$

(9) $(S, E, \kappa^{(n|0)} : nil, C, D)$

$\Rightarrow$ $(\kappa^{(n)} : S, E, M, C, D)$

(10) $(S, E, \kappa^{(n|0)} : \kappa^{(m|i)} : M, C, D) \wedge (i > 0)$

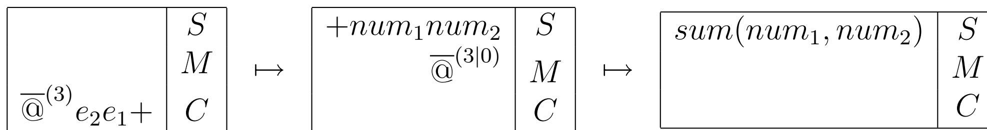
$\Rightarrow$ $(\kappa^{(n)} : S, E, \kappa^{(m|i-1)} : M, C, D)$

Zusätzlich:

- Definition Lookup: $lookup(v, E) \rightarrow \begin{cases} e & \text{falls} \quad \langle v\, e\rangle \in E \ \text{(jüngster Eintrag)} \\ \bot & \text{sonst} \end{cases}$
- Anfangszustand: $(nil, nil, nil, e : nil, nil)$
- Endzustand: $(e^r : nil, nil, nil, nil, nil)$[2]
- Beispielfolge von Auswertungen siehe Folie im Netz

### 1.2.5 Anwendung primitiver Funktionen und Prozeduren

Zum Beispiel `(+ e1 e2)`:



$(+ : num_1 : num_2 : S, E, \overline{@}^{(3|0)} : M, C, D) \Rightarrow (sum(num_1, num_2) : S, E, M, C, D)$

Schwieriger wird's bei Sonderformen wie `if`:

$$
\begin{array}{rcl}
\texttt{(if e0 e1 e2)} & \not\Rightarrow & \overline{@}^{(4)} e_2 e_1 e_0 if \\
\texttt{(if e0 e1 e2)} & \Rightarrow & \overline{@}^{(2)} e_0 \Delta^{(2)} e_1 e_2 \\
\text{bei } true & \Rightarrow & (\#t : S, E, \overline{@}^{(2|1)} : M, \Delta^{(2)} e_1 e_2 : C, D) \\
 & \Rightarrow & (S, E, M, e_1 : C, D) \\
\text{bei } false & \Rightarrow & (\#f : S, E, \overline{@}^{(2|1)} : M, \Delta^{(2)} e_1 e_2 : C, D) \\
 & \Rightarrow & (S, E, M, e_2 : C, D)
\end{array}
$$

[2] Nachdem alle das gesamte Regelwerk brav von der Tafel abgeschrieben haben, legt Prof. Kluge die Folie auf mit der Bemerkung „... das habe ich ihnen ins Netz gestellt.“

Bei der Sonderform `define`:

$$\texttt{(define f (lambda (u1 ... un) eb))} \Rightarrow \left\langle f\ \lambda^{(n)} u_1 ... u_n e_b \right\rangle : E$$

$$(S, E, M, f : C, D) | lookup(f, E) = \lambda^{(n)} = \lambda^{(n)} u_1 ... u_n e_b$$
$$\Rightarrow (S, E, M, \lambda^{(n)} u_1 ... u_n e_b : C, D)$$

Voraussetzung: eindeutige Namenswahl für die Identifikatoren $f$.

## 1.3 Instruktions- und Codeausführende Scheme-Maschine

Erweiterung der SE(M)CD um einen *Heap* $H$, d.h. $(S, E, C, D, H)$; der Heap enthält Pointer auf Code + Stelligkeit des Codes, d.h. $H[...p \rightsquigarrow (r, code)...]$.

### 1.3.1 Scheme-Quellcode

1. Funktions- bzw. Prozedur-Definitionen: ausschließlich:

   ```
   (define f (lambda (u1 ... un) eb))
   (letrec ((ff1 e1) ... (ffk ek)) e0)
   ```

   Keine Verschachtelung von `letrec` etc. Insbesondere müssen $\lambda$-Ausdrücke wie z.B.

   ```
   (lambda (u1 ... un)    eb)
   ```

   geschlossene Ausdrücke sein, d.h. in $e_b$ treten keine weiter außen gebundenen Variablen auf, nur $u_1$ bis $u_n$ bzw. in $e_b$ lokal gebundene Variablen !

2. Transformation Lambda-gebundener Variablen in Bindungsindizes (Deklarationsposition)

   $$\texttt{((lambda (u v w) (u (v (u w) w))) a1 a2 a3)}$$
   $$\rightarrow \texttt{(}\lambda^{(3)}\texttt{(\#2 (\#1 (\#2 \#0) \#0)))}$$

   Beispielaufruf: $\texttt{((}\lambda^{(3)}\ \texttt{(\#2 (\#1 (\#2 \#0) \#0)))}\ a_1\, a_2\, a_3\texttt{)}$

   zugehöriges Environment: $E = \underbrace{a_3}_{\#0}\ \underbrace{a_2}_{\#1}\ \underbrace{a_1}_{\#2}$

### 1.3.2 Instruktionssatz der SECDH-Maschine

Siehe Folie im Anhang (A.4) bzw. online

### 1.3.3 Compiler MiniScheme → SECDH-Code

$e :=$ `atom|(if` $e_0\, e_1\, e_2$`)|(`$e_0\, e_1 \ldots e_n$`)|(`$\lambda^{(n)}\ \#e_b$`)|(define` $f f\, e$`)`

Übersetzungsschema: $C[e : es] \rightarrow code[e]; C[es]$ (mit $e \equiv$ Ausdruck, der übersetzt wird und $es \equiv$ Rest von Ausdrücken)

- $C[atom : es]$ $(atom = const \vee pp \vee primf)$

$\Rightarrow$ $PUSH_S\, atom;\ C[es]$

- $C[\#i : es]$

$\Rightarrow$ $PUSH_ES\, i;\ C[es]$

- $C[($`if` $e_0\, e_1\, e_2)]$

$\Rightarrow$ $C[e_0]; BRC\, p_t\, p_f; C[es]$ und $p_t \rightsquigarrow \mathcal{F}[e_1]$, $p_f \rightsquigarrow \mathcal{F}[e_2]$

- $C[(e_0\, e_1 \ldots e_{n-1}\, e_n) : es]$

$\Rightarrow$ $C[e_n]; C[e_{n-1} : \ldots : e_0 : ap^{(n)} : es]$

- $C[\lambda^{(r)} : e_b : ap^{(r)} : es]$

$\Rightarrow$ $CALL\, p_{ff};\ C[es]$ und $p_{ff} \rightsquigarrow \mathcal{F}[\lambda^{(r)} e_b]$

- $C[\lambda^{(r)} e_b : es]$

$\Rightarrow$ $PUSH_S\, p_{ff};\ C[es]$ und $p_{ff} \rightsquigarrow \mathcal{F}[\lambda^{(r)} e_b]$

- $C[($`define` $f f\, e) : es]$

$\Rightarrow$ $C[es]$ und $p_{ff} \rightsquigarrow \mathcal{F}[e]$

- $\mathcal{F}[e] = \begin{cases} (r, MOVE_SE\, r;\ C[e_b];\ FREE\, r;\ RET;) & \text{falls} \quad e = \lambda^{(r)} e_b \\ (0,\ C[e];\ RET;) & \text{sonst} \end{cases}$

### 1.3.4 Beispiel: Compilierung/Ausführung $(fac\, 2)$

- `(define fac (lambda (n) (if (zero? n) 1 (* n (fac (- n 1))))))`

$\Rightarrow$ $C[$`(define fac (`$\lambda^{(1)}$ `(if (zero? #0) 1 (* #0 (fac (- #0 1))))))`$]$

- $p_{fac} \rightsquigarrow \mathcal{F}[\lambda^{(1)}$`(if (...))`$]$

- $p_{fac} \rightsquigarrow (1, MOVE_SE\, 1;\ C[$`(if ...)`$];\ FREE\, 1;\ RET;)$

- $C$[`(if (zero? #0) 1 (* #0 (fac (- #0 1))))`]

$\Rightarrow$ $C$[`(zero? #0)`]; $BRC\, p_t\, p_f$;

- $p_t \rightsquigarrow (0, PUSH_S\,1;\ RET;)$
- $p_f \rightsquigarrow (0, C$[`(* #0 (fac (- #0 1)))`]$)$

$\Rightarrow$ $(0, C$[`(fac (- #0 1))`]; $C[\#0 : * : ap^{(2)}];\ RET;)$

$\Rightarrow$ $(0, C$[`(- #0 1)`]$ : C$[`fac`$ap^{(1)}$]; $C[\#0 : * : ap^{(p)}];\ RET;)$

$\Rightarrow$ $(0, PUSH_S\,1;\ PUSH_ES\,0;\ MINUS;\ PUSH_S\,p_{fac};$
$AP1;\ PUSH_ES\,0;\ MULT;\ RET;)$

- $p_{fac} \rightsquigarrow (1, MOVE_SE\,1;\ PUSH_ES\,0;\ ZERO?;\ BRC\,p_t\,p_f;\ FREE\,1; RET;)$
- $p_t \rightsquigarrow (0, PUSH_S\,1;\ RET;)$
- $p_f \rightsquigarrow (0, PUSH_S\,1;\ PUSH_ES\,0; MINUS;$
  $CALL\,p_{fac};\ PUSH_ES\,0;\ MULT;\ RET;)$
- $C$[`(fac 2)`] $\rightsquigarrow PUSH_S, 2;\ CALL\,p_{fac};$

Beispielaufruf von `(fac 2)`[3]:

| | |
|---|---|
| $2\ \not2\ \not1\ \not1\ \not1\ \not0\ \not0\ \not1\ \not1\ \not\#f\ \not1\ \not1\ \not2\ \not1\ \not\#f\ \not2\ \not2$ | $S$ |
| $\not0\ \not1\ \not2$ | $E$ |
| $((((((free1, ret), pushes0, mul, ret), free1, ret), pushes0, mult), ret, free1, ret), nil)$ | $D$ |

## 1.4 Codeausführende Scheme-Maschine, 2. Versuch

Erweiterung um geschachtelte Funktionsdefinitionen mit (relativ) freien Variablen → `letrec`-Konstrukte

### 1.4.1 Behandlung von *letrec*-Ausdrücken

modinizierte Syntax: `(letrec (... (`$f_i\, e_i$`) ...)` $e_0$`)` wird geschrieben als `letrec ...` $f_i = e_i$ `... in` $e_0$ (wobei $e_0$ der Rumpf des *letrec*s ist und $e_i$ der Rumpf von $f_i$ ist). Beispiel[4]:

[3] „Sie können's auch für 200 ausprobieren..."
[4] „... um das nicht zu sehr zu überladen..."

```
letrec f = lambda u1 u2
|             letrec g = lambda v1 v2
|             |             letrec h1 = lambda w1 w2 (... (h2 w1 v2) (f w3 u1)
|             |             |       h2 = lambda z1 z2 (... (h1 u1 z2) (g u2 v1)
|             |             in (... (h1 u1 v1) ... (g u1 v2) ... (f u2 v1) ...)
|             in (... (g u2 u1) ... (f u1 u2) ...)
in (... (f 2 3) ...)
```

Abläufe auf dem **Environment**-Stack:

- nach Aufruf von $f$ im Rumpf des definierenden `letrec`:

$$| \langle u_2, 3\rangle \langle u_1, 2\rangle |nil|E$$

- nach Aufruf von $h$, in $g$ in $f$:

$$| \underbrace{\langle w_2, 3\rangle \langle w_1, 2\rangle}_{h_1} | \underbrace{\langle v_2, 2\rangle \langle v_1, 3\rangle}_{g} | \underbrace{\langle u_2, 3\rangle \langle u_1, 2\rangle}_{f} |nil|E$$

- nach mehreren wechselseitigen Aufrufen von $h_1$ und $h_2$:

$$| \underbrace{\langle z_2, ...\rangle \langle z_1, ...\rangle}_{h_2} |...| \underbrace{\langle z_2, 3\rangle \langle z_1, 2\rangle}_{h_2} | \underbrace{\langle w_2, 3\rangle \langle w_1, 2\rangle}_{h_1} | \underbrace{\langle ...\rangle}_{g} | \underbrace{\langle ...\rangle}_{f} |nil|E$$

- nach mehreren wechselseitigen Aufrufen von $h_1$ und $h_2$ sowie von $g$ und $f$, Environment für $h_2$:

$$| \underbrace{h_2} |h_1|...|h_2|h_1| \underbrace{g} |...|g| \underbrace{f} |...|f|nil|E$$

(für ein Environment müssen jeweils die jüngsten Frames verkoppelt werden)

- nach mehreren Aufrufen von $h_1, h_2, g$ und $f$ sowie Aufruf von $g$ in $h_2$, Environment für neues $g$:

$$| \underbrace{g} |h_2|...|h_1|g|...|g| \underbrace{f} |...|f|nil|E$$

- nach mehreren Aufrufen von $h_1, h_2, g$ und $f$ sowie Aufruf von $g$ in $h_2$, Environment für neues $f$:

$$| \underbrace{f} |g|h_2|...|h_1|g|...|g|f|...|f| \underbrace{nil} |E$$

- **Problem**: Finden der richtigen Frames, die zu einem entsprechenden Environment zusammengefaßt werden müssen. !

Ersetzung $\lambda$-gebundener Variablen durch **Indextupel** $[\#i\,\#j]$ wobei $\#i$ die Entfernung zum bindenden $\lambda$ angibt und $\#j$ die Deklarationsposition im bindenden $\lambda$ spezifiziert.

$$\lambda_i\ u_n...u_j...u_1\ \texttt{in}\ (...\ \lambda\ v_m...v_1\ \texttt{in}\ (...u_j...))\ \text{ mit } i > 0$$

$$\Lambda^n...\ \texttt{in}\ (...\ \Lambda^m...\ \texttt{in}\ (...[\#i\,\#j]...))$$

Ersetzen von Funktionsnamen durch **Indextupel**: Gegeben: `letrec` ... $ff = \lambda\ u_1...u_n\ (...)$ ... `in` $e_0$. Ersetze Vorkommen von $ff$ durch $[\#i\,ff]$ ($\#i$ *nesting level* für Funktion $ff$)

- $[\# - 1\,ff]$ falls $ff$ im Rumpf des definierenden `letrec` vorkommt.
- $[\#0\,ff]$ falls $ff$ frei im Rumpf der Funktionsdefinition vorkommt oder im Rumpf einer auf gleicher Ebene definierten Funktion.
- $[\#i\,ff]$ falls zwischen den Vorkommen von $ff$ und dem $\lambda\ u_1...u_n$ des definierenden Vorkommens $i$ $\lambda$-Binder liegen.

**Auswertung des Beispiels**:

```
letrec f = lambda u1 u2
|         in letrec g = lambda v1 v2
|            |         in letrec h1 = lambda w1 w2 (... (h2 w1 v2) (f w3 u1)
|            |            |      h2 = lambda z1 z2 (... (h1 u1 z2) (g u2 v1)
|            |            in (... (h1 u1 v1) ... (g u1 v2) ... (f u2 v1) ...)
|            in (... (g u2 u1) ... (f u1 u2) ...)
in (... (f 2 3) ...)
```

```
letrec f =  ΛΛ
|           in  letrec g =  ΛΛ
|               |           in  letrec  h1 = ΛΛ(...([#0 h][#0 #1][#1 #0])...
|               |               |                   ([#2 f][#0 #0][#2 #1])...)
|               |               |       h2 = ΛΛ(...([#0 h1][#2 #1][#0 #0])...
|               |               |                   ([#1 g][#2 #0][#1 #1])...)
|               |               in(...([# − 1 h1][#1 #1][#0 #1]
|               |                                 ...([#0 g]...)...([#1 f]...)...))
|               in(...([# − 1 g][#0 #0][#0 #1])...([#0 f][#0 #1][#0 #0])...)
in(...([# − 1 #f]23...)...)
```

### 1.4.2 Modifizierte SECDH2-Maschine

- $E$, $C$ und $H$ werden eigenständige Heap-Bereiche
- $E[ppf \rightsquigarrow frame]$, $C[pf \rightsquigarrow code]$ und $H[ph \rightsquigarrow misc]$.
- Anfügen eines Frames: $E \rightarrow ppf \rightsquigarrow frame : E$

- Löschen/Entfernen eines Frames: $ppf \rightsquigarrow frame : E \rightarrow E$
- für alle Pointer $pp$ gilt: $pp \in \underbrace{\{0...r_E\}}_{E} \mid \underbrace{\{0..r_C\}}_{C} \mid \underbrace{\{0...r_H\}}_{H}$
- Maschinenzustand:

$$(S, pe, E[pe \rightsquigarrow frame], pc, C[pc \rightsquigarrow code], D, H)$$

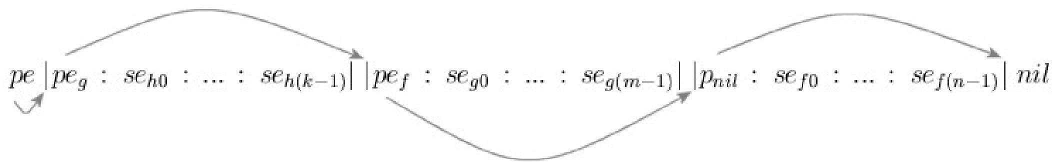
- Pointer auf den Code sollen numerierte Instruktionen ansprechen, wenn $pc \rightsquigarrow instr_0$ soll $pc + i \rightsquigarrow inst_i$.

### Beispiel für Aufruffolge

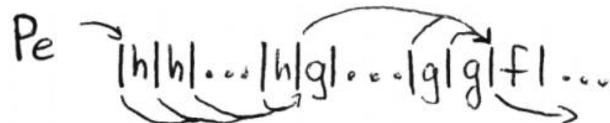
```
instr(0)
...
instr(i-1)
CALL pff ·-------> ARGS n
                    ...
                    instr(j-1)
                    BRC pt pf ·-------> instr'(0)
                                         ...
                                         instr'(k)
                    instr(j+1) <------· RET
                    ...
                    instr(m)
instr(i+1) <------· RET
...
instr(n)
```

### Darstellung des Environment

Annahme: Funktion $f$ ist $n$-stellig, Funktion $g$ ist $m$-stellig, $h$ ist $k$-stellig. Aufruffolge: $f \rightsquigarrow g \rightsquigarrow h$




Bei mehreren Frames pro Funktion:



### 1.4.3 Instruktionssatz

- $PUSH_S$ ein Atom (Zahl etc.) auf den S-Stack schieben
- $PUSH_ES$ eine Variable auf den S-Stack schieben aus dem E-Stack
- $CALL\ p_f$ ruft Programm an Stelle $p_f$ auf
- $RET$ kehrt danach wieder zurück
- $FREE\ n$
- $MAKEFRAME\ n$ erstellt ein neues Environment Frame mit $n$ Einträgen vom $S$-Stack
- $LINK\ h$ derefernziert den aktuellen Environment-Pointer $(h+1)$-mal und legt den so erhaltenen Zeiger auf den $S$-Stack
- $ARGS\ n$ dient ausschließlich dazu, die Stelligkeit einer Funktion nachzuhalten, wird ansonsten ignoriert (Pseudo-Instruktion)
- $CREATE_CLOS\ n\ p_{ff}$ erzeugt eine Closure im Heapund dereferenziert dabei das aktuelle Environment $(h+1)$-mal; Problem: der Pointer auf das Environment kann später ins Leere zeigen, falls das Environment, aus dem es aufgerufen wurde, schon wieder abgebaut worden ist. Daher müssen die Environment-Pointer in der Closure „dafür sorgen“, daß die entsprechenden Einträge im Environment verfügbar bleiben[5]!
- $AP\ r$ löst eine Closure auf und sorgt für die Ausführung der Funktion im korrekten Environment

Regelwerk[6] siehe Folie online

### 1.4.4 Modifikation des Compilers

### 1.4.5 Übersetzungsbeispiel

```
letrec f = lambda u, v
           letrec g = lambda w z
                      (if (> w u)
                          (g (- 1 u) z)
                          (f v (+ 1 w)))
```

[5]Möglichkeiten: Environment mit in die Closure kopieren, Garbage Collection (nur Envorinments abbauen, die nicht mehr referenziert sind) etc.

[6]„Das können Sie eigentlich alles alleine machen...“

```
            in (g v u)
in (f 1 2)
```

Auswertung per Hand ergibt:

$$\texttt{(f 1 2)} \rightarrow \texttt{(g}^{(1,2)}\texttt{ 2 1)} \rightarrow \texttt{(g}^{(1,2)}\texttt{ 0 1)}$$

$$\rightarrow \texttt{(f 2 1)} \rightarrow \texttt{(g}^{(2,1)}\texttt{ 1 2)} \rightarrow \texttt{(f 1 2)}$$

Umsetzung in Nummerndarstellung:

$$\begin{array}{lll} \texttt{letrec} & f = \Lambda^{(2)}\,\texttt{letrec} & g = \Lambda^{(2)}(\texttt{if}\,(>[\#0,\#1]\,[\#1,\#1]) \\ & & ([\#0,g]\,(-1\,[\#1,\#1])\,[\#0,\#0]) \\ & & ([\#1,f]\,[\#1,\#0]\,(+1[\#0,\#1]))) \\ & & \texttt{in}\,([\#-1,g]\,[\#0,\#0]\,[\#0,\#1]) \\ & \texttt{in}\,([\#-1,f]\,1\,2) & \end{array}$$

Compilerschritte:

- $\mathcal{C}\left[\left|\texttt{letrec}\,f = \Lambda^{(2)}...\texttt{in}\,([\#-1,f]\,1\,2)\right|\right]$

$\Rightarrow$ $\mathcal{C}\,[|([\#-1,f]\,1\,2)|]$ und $p_f \rightsquigarrow \mathcal{F}\left[\left|\Lambda^{(2)}...\right|\right]$

$\Rightarrow$ $\mathcal{C}\left[\left|2:1:[\#-1,f]:\overline{@}^{(2)}\right|\right]$

$\Rightarrow$ `PUSH_S 2, PUSH_S 1, LINK -1, CALL p_f`

- $\mathcal{F}\left[\left|\Lambda^{(2)}\texttt{letrec}\,g = ...\texttt{in}\,([\#-1,g]\,[...]\,[...])\right|\right]$

$\Rightarrow$ `ARGS 2, MAKEFRAME 2,` $\mathcal{C}\,[|letrec g = ...|]$ `, FREE 2, RET`

- $\mathcal{C}\left[\left|\texttt{letrec}\,g = \Lambda^{(2)}(\texttt{if}\,...)\texttt{in}\,([\#-1,g]\,[\#0,\#0]\,[\#0,\#1])\right|\right]$

$\Rightarrow$ $\mathcal{C}\,[|([\#-1,f]\,[\#0,\#0]\,[\#0,\#1])|]$ und $p_g \rightsquigarrow \mathcal{F}\left[\left|\Lambda^{(2)}(\texttt{if}\,...)\right|\right]$

$\Rightarrow$ `PUSH_ES 0 1, PUSH_ES 0 0, LINK -1, CALL p_g`

- $\mathcal{F}\left[\left|\Lambda^{(2)}(\texttt{if}\,...)\right|\right]$

$\Rightarrow$ `ARGS 2, MAKEFRAME 2,` $\mathcal{C}\,[|(\texttt{if}\,...)|]$ `, FREE 2, RET`

- $\mathcal{C}\,[|(\texttt{if}\,p\,t\,f)|]$

$\Rightarrow$ $\mathcal{C}\,[|p|]$ `, BRC p_t p_f` und $p_t \rightsquigarrow \mathcal{C}\,[|t|]$ `, RET` sowie $p_f \rightsquigarrow \mathcal{C}\,[|f|]$ `, RET`

- $\mathcal{C}\,[|(>\,[\#0,\#1]\,[\#1,\#1])|]$

$\Rightarrow$ `PUSH_ES 1 1, PUSH_ES 0 1, GT`

- $\mathcal{C}\,[|([\#0, g]\,(-\,1\,[\#1, \#1])\,[\#0, \#0])|]$

$\Rightarrow$ `PUSH_ES 0 0, PUSH_ES 1 1, PUSH_S 1, MINUS, LINK 0, CALL p_g`

- $\mathcal{C}\,[|([\#1, f]\,[\#1, \#0]\,(+\,1\,[\#0, \#1]))|]$

$\Rightarrow$ `PUSH_ES 0 1, PUSH_S 1, PLUS, PUSH_ES 1 0, LINK 1, CALL p_f`

# 2 Rechnerarchitekturen

## 2.1 Architekturklassen

- **CISC**-Architektur (Complex Instruction Set Computer)
  - Ebene dicht unterhalb von Hochsprachen bzw. dicht unterhalb von SECDH-Code
  - verschiedene Formate, aufwendige Adresssierungsmodelle etc., damit auch eine hohe Codedichte
  - Schichtung in CISC-Architekturen:

| Software | | | | |
|---|---|---|---|---|
| Software | | | **Hochsprache** | HLL |
| | *Compilation* | ⇒ | **Zwischensprache** | IML |
| | *Compilation* | ⇒ | **Assembler** | ASL |
| | *Assemblierung* | ⇒ | **Maschinencode** | ML |
| Firmware | *Interpretation* | ⇒ | **Mikroprogrammsprache** | MPL |
| | *Interpretation* | ⇒ | **Nanoprogrammsprache** | NPL |
| Hardware | *Steuerung* | ⇒ | **Register-Transfer-Struktur** | |

  - aufwendige Interpretation, „platzkonsumierendes“ Steuerwerk

- **RISC**-Architektur (Reduced Instruction Set Computer)
  - gerinerer Instruktionsvorat (Reduktion von über 200 auf ca. 40), einheitliches Format für alle Instruktionen
  - alle rechnenden Instruktionen finden nur *Register* → *Register* statt, Laden bzw. Entladen von Registerinhalten ausschließlich über *load*- oder *store*-Instruktionen (*L/S-Architektur*)
  - extrem einfache Instruktionsinterpretation (fast kein Steuerwerk)
  - Pipeline-Verarbeitung
  - geringe Codedichte, bei gleichem Hochsprachenprogramm deutlich länger als mit CISC-Instruktionssatz
  - Schichtung in RISC-Architekturen:

| Software | | Hochsprache |
|---|---|---|
| | *Compilation* ⇒ | **Zwischensprache** |
| | | **abstrakte Maschine** |
| | *Compilation* ⇒ | Programmiersprache **C** |
| | *Compilation* ⇒ | Zwischensprache **C** |
| maschinen- | *Compilation* ⇒ | **Assembler** |
| spezifisch | *Assemblierung* ⇒ | **Maschinencode** |
| Hardware | | (Nanoprogrammierung) |
| | *Steuerung* ⇒ | **Register-Transfer-Struktur** |

- weitere Architekturen/Übersicht: siehe Anhang (A.5)
  - $VN$: von-Neumann-architecture (generelle Verarbeitung von Instruktionen)
  - $MPC$: micro-programm-controlled
  - $HWC$: hardware-controlled
  - $CISC$: complex-instruction-set-computer
  - $RISC$: reduced-instruction-set-computer
  - $OOU, OOC$: Instruktionen:

    $$(ope)rator \quad \{\underbrace{(ope)rand}_{adresses} \mid \underbrace{label}_{adresses} \mid value\}^k \wedge k \in \{0,1,2,3\}$$

    *Operator-Operand-Uncoupled* (Operator legt Adressierungsmodus für Operanden nicht fest) oder *Operator-Operand-Coupled* (der Operator legt Adressierungsmodus schon fest)
  - $WM$: well-mapped (one instruction at a time)
  - $PPL$: pipelined

## 2.2 Instruktionsklassen und Ausführungsphasen

**Instruktionsklassen**:

- **transportierende**„Haben Sie 'ne Vorstellung, wie viele Kilometer Sie schon angeschrieben haben? Wenn man so alles auseinanderzieht... Wie kann man denn das ausrechnen? ... pro DINA4-Seite ca. 4,5 Meter... auf mehr als 3.000 bis 5.000 Meter kommen sie nicht...“ Instruktionen
  - Speicher → Register (load, move)
  - Register → Speicher (store, move)

  - Speicher → Speicher (move)
  - Register → Register (move)
  - Stack → Register/Speicher (S → E)
  - Register/Speicher → Stack (E → S)
- **rechnende** Instruktionen
  - arithmetische
  - logische
  - vergleichende Instruktionen
- **steuernde** Instruktionen
  - unbedingte Sprünge (branch always)
  - bedingte Sprünge (branch condition)
  - Subroutinen-Aufrufe (branch to subroutine)
  - Supervisor-Calls (Betriebssystemaufrufe)

**Instruktionsaufführungsphasen**:

- für rechnende und transportierende Instruktionen:
  - $IF$: instruction fetch (from memory)
  - $ID$: instruction decode (Beginn der Interpretation)
  - $AG$: adress generation (for operands)
  - $DF$: data (operand) fetch (from memory
  - $EX$: operator execution
  - $WB$: write back (store result in memory/register)
- für steuernde Instruktionen:
  - $IF$: instruction fetch (from memory)
  - $ID$: instruction decode (Beginn der Interpretation)
  - (z.T. $ECC$: evaluate branch condition)
  - $AG$: adress generation (for branch target)
  - $TIF$: fetch target instruction
  - $ECC$: evaluate branch condition

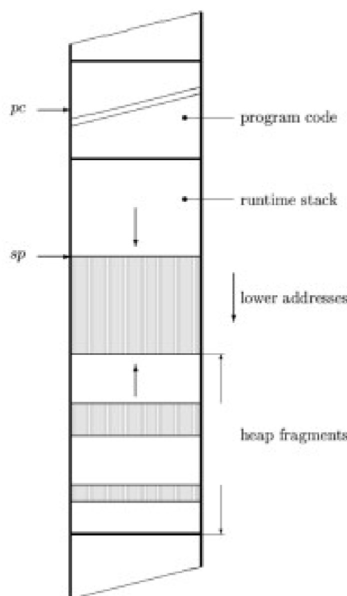
- alle rechnenden/transportierenden Instruktionen setzen ein Werte-Tupel (Condition Code) $(N, Z, C, V) \in \mathbb{B}^4$ mit
  - $N$: non-zero-Resultat
  - $Z$: zero-Resultat
  - $C$: Carry (Übertrag)
  - $V$: Overflow (des Resultats)

  Zustand wird abgefragt und rückgesetzt von nachfolgenden Branch-Instruktionen

# 3 MC680X0-Architektur (CISC)

## 3.1 Abbildung der Laufzeitumgebung auf Speicherbereiche

Abbildung der Laufzeitumgebung für Programmausführung auf Bereich hinreichender Größe mit forlaufenden Adressen: Region




## 3.2 Registerkonfiguration

- Datentypen:

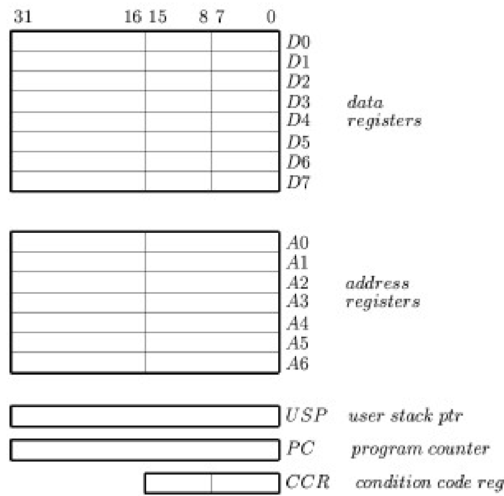
| Datenlänge | MC860X0 | üblich |
|---|---|---|
| $\mathbb{B}^8$ | byte | byte |
| $\mathbb{B}^{16}$ | word | halfword |
| $\mathbb{B}^{32}$ | long word | word |

- Adressierung:

$$MM[\underbrace{0 \mathinner{..} 2^m - 1}_{adress} \mid \underbrace{7 \mathinner{..} 0}_{byte}]$$

wobei $m_{virtuell} = 32$ ist, d.h. man kann $2^{32} \approx 4 \cdot 10^9$ Byte generieren; wobei $m_{real} < 32$ ist.

- sichbare Ressourcen:



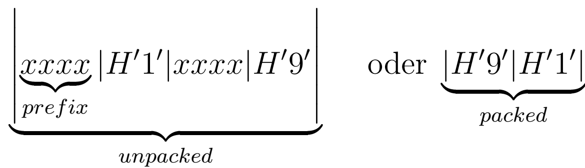

$$R \in \{A_0, ..., A_6, D_0, ..., D_7, USP \equiv A_7, PC, CCR, ...\}$$

- Für den *supervisor mode* reservierte Register:

## 3.3 Datenformate

- Hexadezimaldarstellung für Dezimalzahlen:

| Hexadezimal | Binär |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| ⋮ | ⋮ |
| 9 | 1001 |
| A | 1010 |
| B | 1011 |
| ⋮ | ⋮ |
| F | 1111 |



$$\underbrace{\left|\underbrace{xxxx}_{prefix}|H'1'|xxxx|H'9'\right|}_{unpacked} \quad \text{oder} \quad \underbrace{|H'9'|H'1'|}_{packed}$$

## 3.4 Adressierungsmodes

- Struktur: $(ope)rator\ addr_spec_1\ addr_spec_2$
- Möglichkeiten:
  1. direkte Angabe eines Registers
  2. Berechnung einer effektiven Adresse (Speicher) aus mehreren Komponenten
  3. Implizite Benutzung spezieller Register ($PC$, $SP$)
- Sei $R \in \{A_0, ..., A_n, D_0, ..., D_7, SR, CCR, ...\}$, dann bezeichnet $R$ den Inhalt des Registers $R$.
- Sei $MM[0 .. 2^m - 1]$ der Speicher mit Adressbereich $[0 .. 2^m - 1]$ (wobei $m \leq 32$).
- Sei $addr \in [0 .. 2^m - 1]$ eine Adresse, dann bezeichnet $MM[addr]$ den Inhalt der Speicherzelle mit Adresse $addr$.
- $MM[A_n]$ ist Inhalt der Speicherzelle, die durch den Inhalt von $A_n$ adressiert wird
- Adressierungsmöglichkeiten:

| Modus | Syntax | Effekt |
|---|---|---|
| register_direct | $D_n$<br>$A_n$ | $rand \rightleftarrows D_n$<br>$addr \rightleftarrows A_n$ |
| register_indirect | $(A_n)$ | $rand \rightleftarrows MM[A_n]$ |
| register_indirect mit pre_decrement | $-(A_n)$ | $A_n := A_n - size;$<br>$rand \rightleftarrows MM[A_n]$ |
| register_indirect mit post_increment | $(A_n)+$ | $rand \rightleftarrows MM[A_n]$<br>$A_n := A_n + size;$ |
| register_indirect mit displacement | $d_{16}(A_n)$<br>$d_{16} \in [-2^{15} .. 0 .. 2^{15} - 1]$ | $rand \rightleftarrows MM[A_n + d_{16}]$ |
| register_indirect indexed | $d_8(A_n, Xm.s_{size} \cdot s_{scale})$<br>$d_8 \in [-2^7 .. 0 .. 2^7 - 1]$ | $rand \rightleftarrows MM[addr]$ mit<br>$addr = A_n + d_8 + Xm \cdot s_{size} \cdot s_{scale}$ |
| register_indirect indexed mit base_displacement | $db(A_n, Xm.s_{size} \cdot s_{scale})$<br>$bd \in [-2^{31} .. 0 .. 2^{31} - 1]$ | $rand \rightleftarrows MM[addr]$ mit<br>$addr = A_n + db + Xm \cdot s_{size} \cdot s_{scale}$ |

## 3.5 Instruktionen/Instruktionsformate

- $MOVE.x$ *source*, *destination* bewegt *source* $\rightarrow$ *destination*, Länge ist $x$
- $ADD.x$ *source*, *destination* bewegt *source*+*destination* $\rightarrow$ *destination*
- $MOVEA.x$ $ea, A_n$ schreibt die effektive Adresse $ea$ nach $A_n$, Länge ist $x$
- $MOVE$ $CCR, ea$ schreibt *ConditionCodeRegister* nach $ea$
- $MOVE$ $ea, SR$ schreibt $ea$ nach *StaticRegister*
- $MOVEM.x$ $ea, A_0 - A_3/A_6/D_2/D_5$ etc. verschiebt mehrere (move multiple) Einträge, beginnend bei $ea$ auf die Register $A_0, A_1, A_2, A_3, A_6, D_2, D_5$, Länge spezifiziert durch $x$
- $MOVEM.x$ $A_0 - A_4, ea$ etc. verschiebt mehrere Einträge, z.B. $A_0, A_1, A_2, A_3, A_4$ wird auf $ea$ und folgende Adressen gespeichert, Länge spezifiziert durch $x$
- $CMPI.x$ *#data*, $ea$ vergleicht (compare immediate)
- $LINK.xA_6$, *#displacement* setzt einen Link auf ein älteres Frame auf dem Stack, d.h.
  - $SP \leftarrow SP - 4$ (leere Zelle für Link schaffen)
  - $(SP) \leftarrow A_6$ (Link zum alten $A_6$ schaffen)
  - $A_6 \leftarrow SP$ ($A_6$ korrigieren auf den neuen Stackpointer)
  - $SP \leftarrow SP - \#displacement$ (Stackpointer weiterschieben)
- $UNLK$ $A_6$
  - $A_6 \rightarrow SP$ (Stackpointer auf den Link setzen)
  - $(SP) \rightarrow A_6$ ($A_6$ auf den Wert des Links setzen)
  - $SP \leftarrow SP + 4$ (Stackpointer „hinter“ den Link setzen)
- $LEA$ $ea, A_n$ lädt von der Adresse nach $A_n$ ($ea \rightarrow A_n$)
- $JMP$ $ea$ springt zur Adresse $ea$ mit dem ProgramCounter ($ea \rightarrow PC$)
- $JSR$ $ea$ Jump to Subroutine: $SP \leftarrow SP - 4; PC \rightarrow (SP); ea \rightarrow PC;$

- $BSR\, d(PC)$ Branch to Subroutine $SP \leftarrow SP - 4; PC \rightarrow (SP); PC \leftarrow PC + d$
- $BCC\ d(PC)$ if $(CC = \#true)$ then $PC \leftarrow PC + d$; else $PC$ mit $CC = EQ|GE|GT|...|LE|NE|...$
- $RTS$ return from subroutine $(SP) \rightarrow PC; SP \leftarrow SP + 4$
- $SUB.x\ ea, D_n$ Substraktion
- $SUBA.x\ ea, A_n$ Substraktion von Adressen
- $MULS.x\ ea, D_n$ Multiplikation mit Vorzeichen
- $MULU.x\ ea, D_n$ Multiplikation ohne Vorzeichen
- $CMP.x\ ea, D_n$ Vergleich von $MM[ea]$ mit $D_n$
- $CMPA.x\ ea, A_n$ Vergleich von Adressen

## 3.6 Organisation von Prozedur- und Funktionsaufrufen

- $f\, u_1 \dots u_m = \dots(g\, a_1 \dots a_n)\dots$
- $g\, v_1 \dots v_n = \dots$
- Abbildung der Strukturen $E, S, D$ der $SECDH_1$ auf einen Laufzeitstack $(SP)$
- Aktivierungsrecords bestehen aus:
  - $argumentframe$ für Parameter der aufzurufenden Funktion
  - $PC$ als Rücksprungsadresse in die aufrufende Funktion
  - $SR$ Statusregister der aufrufenden Funktion
  - Register, die von $f$ belegt, aber von $g$ verwendet werden
  - Workspace für temporäre bzw. für lokale Variablen von $g$, soweit nicht in den Registern $D_0$ bis $D_7$ bzw. $A_0$ bis $A_6$ unterzubringen
- aufrufender Code:

```
        ...                               * Bedingungen der Argumente
        MOVE.x arg_source_1, -(SP)        * \
        ...                               *  | Aufbau des arg-frames auf den Stack
        MOVE.x arg_source_n, -(SP)        * /
        BSR LABEL                         * label -> displacement(PC)
                                          * Einsprung in den aufrufenden Code
        ADDA.x #size_of_arg_frame, SP     * loescht das arg-frame
```

- aufgerufener Code:

```
LABEL:  MOVE SR,-(SP)                     * rettet Statusregister
        MOVEM.L reg_list, -(SP)           * rettet Register
        SUBA #size_of_workspace, SP       * schafft Platz fuer den Workspace
        LINK A6,#0                        * setzt den Link Pointer
        ...                               * \ Code fuer Berechnungen
        ADD.x d1(A6),d2(A6)               *  |
        ...                               * /
        MOVE.x result_adress,D0           * belegt D0 mit Funktionswert
        UNLK A6;                          * \
        ADDA #size_of_workspace, SP       *  | loescht den Stack
        MOVEM (SP)+,reg_list              *  | bis zum PC
        MOVE (SP)+,SR                     * /
        RST;                              * bewirkt return zum aufrufenden Code
```

- weiteres Beispiel:

```
letrec f = lambda vars
|          in letrec g = lambda vars
|             |          in letrec h1 = lambda vars
|             |             |           in (... (h1 ...) ... (h2 ...) ...)
|             |             |       h2 = lambda vars
|             |             |           in (... (f  ...) ... (h2 ...) ...)
|             |             in (... (g ...) ... (f ...) ...)
|             in (... (g ...) ...)
in (f ...)
```

- Aufruffolge: `f calls g calls h1 calls h2 calls f`
- Seien $p, q$ Funktionen/Prozeduren, $p$ rufe $q$ auf. Fallunterscheidung:
  - $q$ ist lokal zu $p$ definiert: Der environment pointer in $q$ muß auf den environment pointer in $p$ gesetzt werden
  - $q$ und $p$ sind auf gleichem Niveau definiert: $q$ übernimmt den environment pointer von $p$
  - $q$ ist $k$ Level über $p$ definiert: Der environment pointer von $q$ wird durch $k$-fache Dereferenzierung des environment pointers von $p$ erhalten.
- noch einmal Beispielcode:

```
        MOVEA    A6,-(SP)              * saves A6 on stack
        MOVEA    (A6),A6               * k times as required by the nesting
                                       * levels of calling and called func.
        MOVE.x   source_1,-(SP)        * \
        MOVE.x   ...                   *  | pushes arguments on stack
        MOVE.x   source_n,-(SP)        * /
        JSR/BSR  LABEL                 * branches to subroutine
        ADDA     #size_of_arg_frage,SP * deletes argument frame
        MOVEA    (SP)+,A6              * restores A6 of calling function

LABEL:  MOVEM    reg_list,-(SP)        * save registers
        SUBA     #size_of_workspace,SP * get workspace
        LINK     A6,#0                 * set pointer to environment
        ...                            * \
        ADD      16(A6),D2             *  | run function, access to args
        SUB      8(A6),D3              *  | and workspace by #(A6)
        ...                            * /
        MOVE.x   result,D0             * passing result through D0
        ADDA     #4,SP;                * delete environment link
        ADDA     #size_of_workspace,SP * delete workspace
        MOVEM    (SP)+,reglist         * restore registers
        RTS                            * return
```

## 3.7 Assemblerprogrammierung

Bestandteile von Assemblerprogrammen (Maschinenprogramme in symbolischer bzw. mnemonischer Darstellung):

1. Maschineninstruktionen, siehe oben
2. Assembler-Directives: Pesudoinstruktionen, d.h. Anweisungen an den Assembler, bestreffen Code-Generierung. Beispielsweise:
   - `{label} ORG {expr}` (mit z.B. $expr = 0$), Assembler benutzt einen location counter $LC$, der die Distanz von Instruktionen bzw. Daten in Anzahl der Bytes relativ zum Origin ermittelt
   - `{label} EQU {expr}`, z.B. `offset EQU 8`, Beispieleinsatz: `ADD.x offset(A6),D3;`
   - `{symbol/label} REG {`$D_n - D_m, A_n - A_m$`}` definiert Registersätze, die bei Unterprogrammaufrufen gerettet werden müssen (reglists)
   - `{label} DC.{size} {item, item, ...}` (mit $size = B, W, L, S$) wird benutzt, um unter `label` einen String von Items abzuspeichern
   - `{label} DCB.{size} {length},{value}` (mit $size = B, W, L, S$) legt unter `label` einen Block der Länge `length` vom Format `size` ab, der mit `value` initialisiert wird; fehlendes Value bedeutet keine Initialisierung
   - `END` am Ende

## 3.8 Konvertierung symbolischer in numerische Adressen

- Beispiel: Testprogramm für Fehlerfreiheit eines Peicherbereiches

```
S LC LABEL   INSTR   VALUES              * COMMENT                     ADDRESS
- -- ------- ------- ------------------- * --------------------------- --------
             ORG     #0                  *
2  0 BASE    DC.W    #base_address       * of memory segment to test
2  2 SIZE    DC.W    #4096               * size of segment in bytes
2  4 TESTPAT DC.W    #AAAA               * test pattern
4  6 ENTRY   MOVEM   D1-D2/A1,-(SP)      * save registers
2 10         MOVEA   BASE,A1             * \  loads regs A1, D1, D2     -12(PC)
2 12         MOVE.W  SIZE,D1             *  | with base adress, size    -12(PC)
2 14         MOVE.W  TESTPAT,D2          * /  and test pattern          -12(PC)
2 16 LOOP_1  MOVE.W  D2,(A1)+            * save test pattern
2 18         SUB.W   #1,D1               * decrement counter
4 20         CMPI    #0,D1               * test if done
2 24         BGT     LOOP_1              * if not: loop                 -10(PC)
2 26         MOVEA   BASE,A1             * \  reloads regs A1, D1 with  -28(PC)
2 28         MOVE.W  SIZE,D1             * /  base adress and size      -28(PC)
2 30 LOOP_2  CMP     (A1)+,D2            * read and compare
2 32         BNZ     FAILURE             * look for errors
2 34         SUB.W   #1,D1               * decrement counter
4 36         COMPI   #0,D1               * test if done
2 40         BGT     LOOP_2              * if not: loop                 -12(PC)
4 42         MOVEM   (SP)+,D1-D2/A1      * restore saved registers
2 46         RTS;                        * return
  48 END
```

- LocationCounter legt eine Symboltabelle an:

| Label | *LC* |
|---|---|
| BASE | 0 |
| SIZE | 2 |
| TESTPAT | 4 |
| ENTRY | 6 |
| LOOP_1 | 16 |
| LOOP_2 | 30 |

- Symboltabelle für OP-Codes:

| opc | binary |
|---|---|
| ADD | 1011 |
| ⋮ | ⋮ |

## 3.9 Segmentierung, Linking und Paging

- ein *Segment* ist eine lineare Abfolge von Objekten der Form

```
label   instruction
label   data
```

wobei $SN$ die Menge der Objekte im Segment ist und $n$ die Anzahl der Bytes zur Darstellung der Objekte von $SN$ angibt; die Ordnungsrelation unter den Objekten sei $<$

- Der *LocationCounter* des Assemblers erzeugt eine Abbildung $\alpha^S : SN \to [0\mathinner{..}n-1]$ mit der Eigenschaft

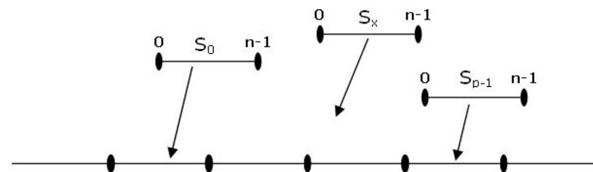
$$(u < v) \Rightarrow (\alpha^S(u) < \alpha^S(v)) \;\forall\; u, v \in SN$$

  Somit ist $\alpha^S$ eine Abbildung in einen logischen Adreßraum.

- im Allgemeinen besteht ein Assemblerprogramm aus mehreren Segmenten $S_0, ..., S_{p-1}$, jedes Segment wird individuell assembliert mit

$$\alpha_r^S : SN_r \to [0\mathinner{..}n_r - 1]$$

- in einem *virtuellen Adreßraum* der Größe $\left(\sum_{i=0}^{p-1} n_i\right)$ liegen alle $p$ einzelnen Adreßräume hintereinander




  Damit ist die Abbildung in einen zusammenhängen virtuellen Adreßraum $V$ (*Linking*) gegeben durch:

$$\alpha_r^V(u) = \alpha_r^S(u) + \left(\sum_{i=0}^{r-1} n_i\right) - 1 \;\forall\; r \in [0\mathinner{..}p-1], u \in SN_r$$

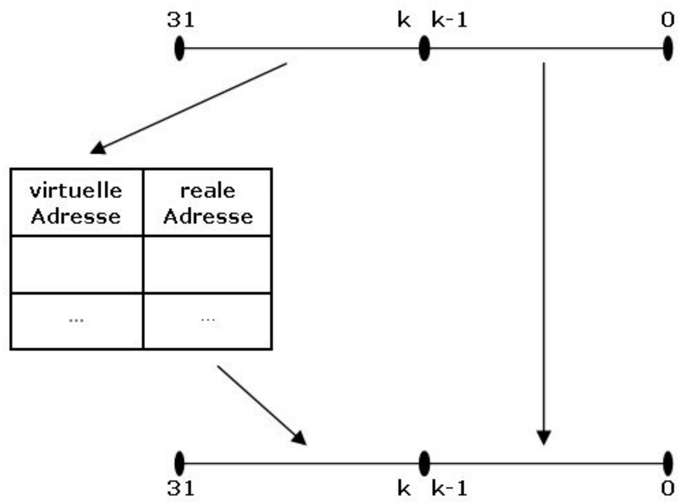
- Linking-Mechanismus:
  - `IMPORT` $\texttt{label}_1$ sorgt dafür, daß $\texttt{label}_1$ im Segment benutzt werden kann, ist außerhalb definiert
  - `EXPORT` $\texttt{label}_2$: im Segment ist $\texttt{label}_2$ definiert
  - `PUBLIC` $\texttt{label}_3$: wir im Semgmengt *und* außerhalb benutzt
  - Bei einer Situation wie der folgenden werden zunächst die Imports gesucht und dann die entsprechenden Exports zugeordnet, um Adreßbezüge herzustellen

```
        BSR     label       * in Si
label:  MOVE    ...         * in Sj
```

- Ladevorgang:
  - Lösung 1: statische Zuweisung eines realen Adreßbereiches; bei mehreren Segmenten noch die Adreßbezüge zwischen den Segmenten korrigieren etc.; Start der Programmausführung durch setzen $PC \leftarrow abs$; damit durchläuft der $PC$ reale Adresse im Speicher
  - Lösung 2: Unterteilung des realen Adreßraums in Seitenrahmen (Kacheln); die Segmente des virtuellen Adreßraums werden in einzelne Kacheln zugeordnet, wobei die Zuordnung nicht linear sein muß.
    * realer Adreßraum: $R = [0 .. 2^m - 1]$ mit $20 \leq m \leq 32$ (d.h. zwischen 1M und 4G)
    * realer Seitenrahmen: $RP = [0 .. 2^k - 1]$ mit $10 \leq k \leq 14$
    * Indizierung realer Seitenrahmen: $IR = [0 .. 2^{m-k} - 1]$
    * Reale Adresse: $|\underbrace{m-1 \ldots k}_{\in IR} | \underbrace{k-1 \ldots 0}_{\text{offset}}|$
    * virtueller Adreßraum: $V = [0 .. M - 1]$
    * virtueller Seitenrahmen: $RV = [0 .. 2^k - 1]$
    * Indizierung virtueller Seitenrahmen: $IV = \left[0 .. \left\lceil \frac{M-1}{2^k} \right\rceil\right]$
    * gesuchte Funktion:

      $$\alpha^P : IV' \rightarrow IR \text{ mit } IV' \subseteq IC$$

    * Umsetzung virtueller in reale Seitenrahmenindizes:
    * Der $PC$ erzeugt virtuelle Adressen, die über eine Seitenzuordnungstabelle in reale Adressen umgesetzt werden. Fehlende Einträge für virtuelle Seitenindizes füren zu einem $page_miss$, d.h. die Seite muß aus dem Hintergrundspeicher nachgeladen werden.




* (aktive) Teile der Seitentabelle sind in der *Memory-Management-Unit* (MMU) als sog. *look-aside-Buffer* untergebracht
* Segmentierung: das Binden der Segmente in einem wirtuellen Adreßraum wird unteralssen; logische Adressen der Segmente werden über eine Segmenttabelle direkt in reale Adressen umgesetzt; Adressierung im Segment erfolgt mit logischen Adressen als offset relativ zur Segment-Basis-Adresse:

# 4 Systemkonfiguration des MC86000

Generelle Konfiguration siehe Anhang (A.6)

## 4.1 Das Unterbrechungssystem

### 4.1.1 Unterbrechungsvektoren

Liste von Adressen, die die Anfangsadressen der Unterrechungsbehandlungsroutinen enthält; jedem Unterbrechungstyp ist ein eindeutiger Eintrag zugeordnet, dabei ergibt sich die Adresse des jeweiligen Eintrags aus der Vektor-Basis-Adresse (*vbr*) und einem der Vektor-Nummer $v_n$ entsprechenden Offset von $4 \cdot v_n$:

$vbr \rightarrow$
$v_n \downarrow$

| Eintrag | $v_n$ |
|---|---|
| `reset SP` | 0 |
| `reset PC` | 1 |
| `bus err` | 2 |
| `addr err` | 3 |
| `ill instr` | 4 |
| `div 0` | 5 |
| `chk` | 6 |
| `trapv` | 7 |
| `priv instr` | 8 |
| `trace` | 9 |
| ⋮ | ⋮ |
| `level 0` | 24 |
| ⋮ | ⋮ |
| `level 7` | 31 |
| `trap 0` | 32 |
| ⋮ | ⋮ |
| `trap 15` | 47 |
| ⋮ | ⋮ |
| ... | 64 |
| ⋮ | ⋮ |
| ... | 255 |

Einzelne $v_n$:

- 0 bis 23: *special prupose exceptions*
  - `reset` wird extern (sys-start/reset) oder intern (reset-Anweisung) ausgelöst
  - `bus err` wird extern bei Memory-Access-Problemen ausgelöst
  - `add err` wird intern ausgelöst bei Word- oder Longword-Zugriff auf eine ungerade Adresse
  - `ill instr` wird intern bei nicht zu interpretierendem OP-Code ausgelöst

- `priv instr` (priveliged instruction): intern, Ausführung eriner Supervisor-Operation im Normal-Modus

- 24 bis 31: *external interrupts / autovector-interrupts*
  - Unterbrechungen, die außerhalb erzeugt werden: Timer, I/O-Geräte, Kommunikation
  - asynchron, können maskiert werden und anhängig bleiben, müssen aber bedient werden
  - gestaffelte Priorität

- 32 bis 47: *traps*
  - Unterbrechungen, die vom Prozessor bzw. laufenden Programm erzeugt werden
  - synchron, müssen unmittelbar bedient werden ($TRAP$-Instruktion: Betriebssystemaufrufe aus dem laufenden Programm)

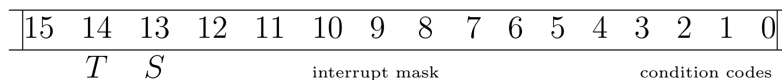
- 64 bis 255: *vector interrupts*
  - entstehen durch externe Interrupts mittels spezieller Hardware-Unterstützung

Prioritäten:

- höchste Priorität: `reset`, `bus err`, `addr err`
- mittlere Priorität: Interrupts, Trace, `ill instr`, privelidge violation
- niedrige Priorität: `traps`, `trapv`, `chk`, `div 0`

### 4.1.2 Programmstatus/Ausführungsmodi

Status-Register:



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $T$ | $S$ | | | interrupt mask | | | | | | | condition codes | | | |

- $T$: Trace-Modus: löst nach jedem Befehl eine Trace-Exception aus
- $S$: Supervisor-Modus (privilegierter Modus, $S$) (S = 1) oder User-Modus (Normalmodus, $N$) (S = 0)
  - getrennte Laufzeitstacks für $S$- und $N$-Modus

– privilegierte Befehle im $S$-Modus:
  * $STOP$: hält Programmsausführung an
  * $RESET$: führt zu System-Reset
  * $RTE$: im Prinzip $RTS$ von einer Unterbrechung
  * $MOVE$, $AND$, $OR$, $ECR$, $MOVE$ mit $SR$ oder $USP$ als Ziel

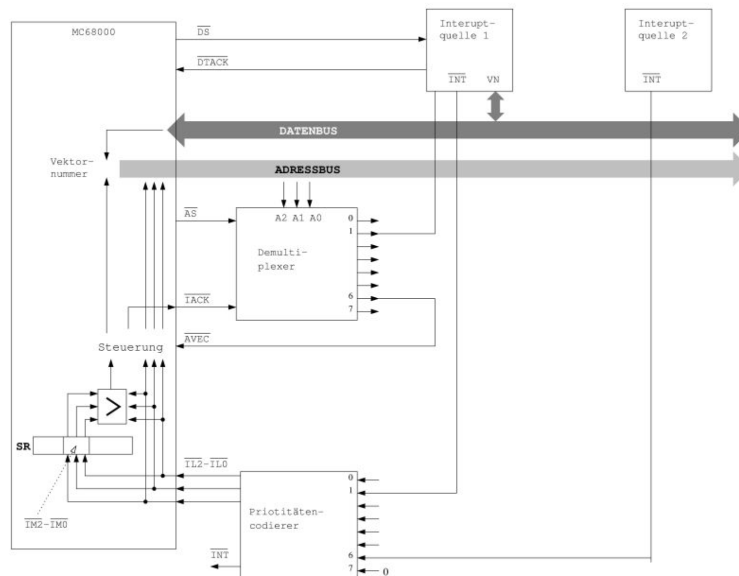
- Interrupt Mask: Legt fest, von welchen Level Interrupts akzeptiert werden, z.B. 111, um alle (bis Level 7 = 111) zu akzeptieren

### 4.1.3 Unterbrechungsbehandlung

- Reset-Behandlung:
  1. Initialisierung des $SR$ mit `0010 0111 0000 0000`
  2. Initialisierung des $SSP$ mit $v_0$
  3. Initialisierung des $PC$ mit $v_1$
  4. Start der Programmausführung
- Generelle Unterbrechungsbehandlung:
  1. Retten des $SR$ in ein spezielles Datenregister
  2. Setzes des $S$-Bits $\Rightarrow$ Supervisor-Modus
  3. Löschen des $T$-Bits (kein Tracen der Exception-Behandlung)
  4. Retten des $PC$ auf dem Supervisor-Stack
  5. Retten des Datenregisters aus 1. auf dem Supervisor-Stack
  6. (ggf. zusätzliche Operationen)
  7. Laden des $PC$ mit $vbr + 4 \cdot v_n$
  8. Start der Programmausführung

### 4.1.4 Maschinelle Bearbeitung der externen Interrupts

1. Priorisierung der Interrupts
   - Schaltung:




- Interrupts bleiben generell in den Interrupts-Quellen anhängig, bis sie von der CPU akzeptiert wurden (über ein $\overline{IACK}$-Signal von der CPU zu den Interrupt-Quellen), siehe unten Autovector-Interrupts
- Der Prioritätendekoder läßt immer den Interrupt höchster Priorität durch, er übersetzt den (die) vorgefundenen Interrupt-Request(s) in ein 3-Bit-Inputerrupt-Level
- Das Interrupts-Level wird mit der Interrupt-Maske verglichen, daraus resultiert die Entscheidung über die Akzeptanz des Interrupts.
- Während der Behandlung des Interrupts wird die Interrupt-Maske auf das entsprechende Level gesetzt, um keine Interrupts niedrigerer Priorität zuzulassen

2. Autovektor- ↔ Vektor-Interrupts

   - Autovektor-Interrupts: nach der Bestätigung durch die CPU (per $\overline{IACK}$) legt die Interruptquelle $\overline{AVEC}$ auf 0; Prozessor benutzt Autovektor-Eintrag ($v_{25}$ bis $v_{32}$).
   - Vektor-Interrupts: nach der Bestätigung (per $\overline{IACK}$) ist $\overline{AVEC}$ auf 1 und die Interruptquelle legt die gewünschen Vektornummer auf den Datenbus, quittiert mit $\overline{DTACK}$ und der Prozessor verwendet den entsprechenden Vektoreintrag ($v_{64}$ bis $v_{255}$) als Interrupt-Vektor.

3. Unterteilung der Interruptebenen

- zusätzlicher Prioritätendekoder, der den Subinterrupt in eine Vektoradresse umsetzt, und zusätzliche Logik, die den Handshake mit dem Prozessor regelt, d.h. sie bedient $\overline{AVEC}$, $\overline{DB}$, $\overline{DTACK}$

4. Daisy-Chain

   - Verkettung der einzelnen Subinterrupt-Quellen durch ein wired-or (Tristate-Ausgänge an den einzelnen Subinterrupt-Quellen), genau ein Interrupt-Level wird bei entsprechender Maskierung ausgelöst
   - das $\overline{IACK}$ wird durch die Interruptquellen hindurchgereicht, bis es zur ersten in der Prioritätenliste gelangt, die einen Interrupt ausgelöst bzw. angefordert hat.
   - Vorteile: verhältnismäßig wenig Hardwareaufwand, trotzdem Priorisierung (entlang der Kette) möglich

5. Interruptquellen-Polling

   - Verkettung der Interrupt-Requests mit wired-or, die Unterbrechungsroutine überprüft durch Nachfrage (Polling), welche Interruptquelle ausgelöst hat und reagiert entsprechend
   - Vorteil: minmaler Hardwareaufwand; Nachteil: Polling kostet Laufzeit

## 4.2 Ein- und Ausgabe-Vorgänge

- Ein- und Ausgabe:
  - Starten/Terminieren peripherer Geräte
  - unterschiedliche Taktraten von Prozessor und Peripherie: aufwendige Synchronisationsmechanismen
  - ggf. Umwandlung Byte-serieller in Bit-serielle Datentranporte
  - Interfaces bzw. Schnittstellen (Ports) zwischen Systembus/Prozessor und Periperhie:
    * Daten(puffer)register $DR$
    * Steuerregister (Control) $CR$
    * Statusregister $SR$
- Synchronisationsmechanismen:

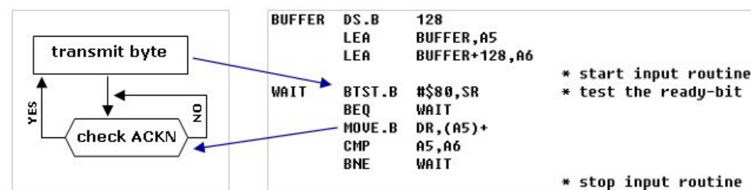
– *busy-wait* mit *polling*: peripheres device sehr viel langsamer als Prozessor, Prozessor ist belastet mit Untätigkeit während der Poll-Zyklen

– *Synchronisation per Interrupt:* Peripherie sehr viel langsamer als Prozessor

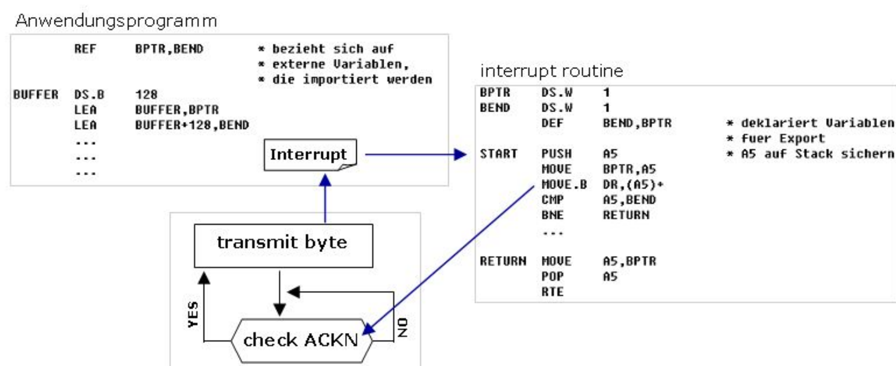
– *Handshaking:* Austausch von Synchronisationssignalen zwischen Prozessor ($\overline{ACKN}$) und Peripherie ($READY$); bei etwa gleichen Taktraten

– Programmbeispiel: Dateneingabe von 128 Bytes vom peripheren Gerät

  * busy-wait with handshaking:




  * handshaking with interrupts:




- Paralleles Interface (Schaltung siehe A.7)

  – *Datenmodus:* port wird benutzt für Datentransport von und zur Periperherie; *Steuermodus:* port wird benutzt zum Austuasch von Sensor- und Steuerimpulsen mit einem peripheren Prozess

  – Register $DDR$ (*data direction register*) gibt Benutztungsrichtung von Datenleitungen an

  – Register $IMR$ (*interrupt mask register*)

  – wesentliche Bestandteile der Steuerregister $CRX$:

    - $CRX[1]$: Daten- oder Steuermodus
    - $CRX[4]$: Handshaking
    - $CRX[5]$: Interrupt-Maskierung ($\overline{INT}$)
    - $CRX[7]$: Interrupt anhängig oder Statusbit $READY$

- Serielles Interface (Schaltung siehe A.8)
  - DUART[7]-Baustein, siehe Übung
  - Serialisierung eines Bytes geschieht in der Schaltung
  - wesentliche Signale:
    - $\overline{CTS}$: clear to send data
    - $\overline{DCD}$: data carrier detect
    - $\overline{RTS}$: request to send

[7] Dualport universal asynchronous receiver transmitter

# 5 SPARC-RISC-Architektur

SPARC bedeutet *Scalable Processor ARCitecture*

## 5.1 Registersatz

Alle Register sind 32 Bit lang, d.h. Wortformat. Sichtbar in jedem Zustand der Programmausführung sind

- acht globale Register $G_0, ..., G_7$
- 24 Window-Register $R_8, ..., R_{31}$, die wie folgt aufgeteilt sind:
  - $R_{31} \cdots R_{24} = I_7 \cdots I_0$: *in*-Register
  - $R_{23} \cdots R_{16} = L_7 \cdots L_0$: *locals*-Register
  - $R_{15} \cdots R_8 = O_7 \cdots O_0$: *out*-Register

Beispielaufruffolge $f \rightarrow g \rightarrow h$:

| | | | | | | |
|---|---|---|---|---|---|---|
| $f$ | in | locals | out | | | |
| $f$ | $R_{31} \cdots R_{24}$ | $R_{23} \cdots R_{16}$ | $R_{15} \cdots R_8$ | | | |
| $g$ | | | in | locals | out | |
| $g$ | | | $R_{31} \cdots R_{24}$ | $R_{23} \cdots R_{16}$ | $R_{15} \cdots R_8$ | |
| $h$ | | | | | in | locals | out |
| $h$ | | | | | $R_{31} \cdots R_{24}$ | $R_{23} \cdots R_{16}$ | $R_{15} \cdots R_8$ |

Steuerung:

- $CWP[0 .. n-1]$: current window pointer mit insgesamt $k < 2^n$ verfügbaren Windows
- $cansave[0 .. n-1]$: Anzahl der noch freien Windows
- $canrestore[0 .. n-1]$: Anzahl der belegten Windows
- $otherwindow[0 .. n-1]$: für Nutzung durch Betriebssystem (Trap-Handler)

Gleichungen/Bedingungen:

- $cansave + canrestore + otherwindow = k - 2$
- Anlegen eines neuen Windows:

```
if (CANSAVE > 1) then { CANSAVE--; CANRESTORE++; CWP++; } else { trap }
```

- Rückgabe eines belegten Windows:

```
if (CANRESTORE > 1) then { CANSAVE++; CANRESTORE--; CWP--; } else { trap }
```

Umsetzung von Funktionen mit relativ freien Variablen in „flache“ Funktionen:

(`def` $f$ $\lambda(x_1...x_n)$ $(...x_1...x_n...u_1...u_n...)$) $\Rightarrow$ (`def` $f$ $\lambda(x_1...x_n u_1...u_n)$ $(...)$)

## 5.2 Instruktionsformate und Instruktionen

Beispiel: Berechnung der Fakultätsfunktion:

```
fac n = if n >= 1 then (n*fac(n-1)) else 1
```

Ausgabe des Compilers (ohne Optimierung):

```
fac:    SAVE    SP,-120,SP       creates a new stack frame & register window
        ST_W    fp,68,I0         \ pass parameter n from calling
        LD_W    fp,68,O0         / to called procedure
        CMP     O0,1             \ computes (n <= 1)
        BR_LE   LL3              / and braches if true
        NOP                      wait for branch to proceed through pipeline
        LD_W    fp,68,O0
        ADD_W   O0,-1,O1         computes (n-1)
        MOVE_W  O1,O0            sets up parameter for passenge to next call
        CALL    fac
        NOP                      wait for branch to proceed through pipeline
        MOVE_W  O0,O1            \
        LD_W    fp,68,O0          | load fac(n-1) and n and calls subroutine
        CALL    mulu              | to multiply them
        NOP                      /
        ST_W    fp,-20,O0
        BRA_A   LL4
        NOP
LL3:    MOVE    1,O0             \ computes else-clause
        ST_W    fp,-20,O0        / with fac(1) = 1
LL4:    LD_W    fp,-20,I0
        BRA_A   LL2
        NOP
LL2:    RET
        RESTORE
```

## 5.3 Instruktionsausführung

Phasen:

1. $IF/IA$: instruction fetch bzw. instruction adressing
2. $ID/SRA$: instruction decode bzw. source register accessing
3. $EX/AG$: instruction execution bzw. adress generation
4. $DF/DS$: data fetch bzw. data store
5. $WR$: write back (result to register)

## 5.4 Behandlung von Pipeline-Abhängigkeiten

1. Datenabhängigkeiten: z.B.

```
ADD R16, ..., ...
SUB ..., R16, ...
```

Korrekturmöglichkeiten:

- Korrektur durch Compilation/Codeerzeugung (kein Hardwareaufwand):

  ```
  ADD R16, ..., ...
  NOP
  NOP
  NOP
  SUB ..., R16, ...
  ```

- Korrektur durch *stalling*: Falls die Bedingung

  $$(ir.sr_1 \vee ir.sr_2) = (ID/SRA.dr \vee EX/AG.dr \vee DF/DS.dr)$$

  erfüllt ist, muß in der Pipeline in der $IF/IA$- und $ID/SRA$-Phase abgewartet werden (*stalling*), wenig Hardwareaufwand

- Korrektur durch *result forwarding*: Falls obige Bedingung erfüllt, wird das Ergebnis der entsprechenden Stufe abgegriffen und direkt an die ALU-Eingangs-Datenregister weitergeleitet. Damit entstehen keine `NOP`s, optimale Lösung für die Laufzeit, aber am meisten Hardwareaufwand.

2. *branch*-Instruktionen:

```
ADD   ..., ..., ...
BR_cc label
```

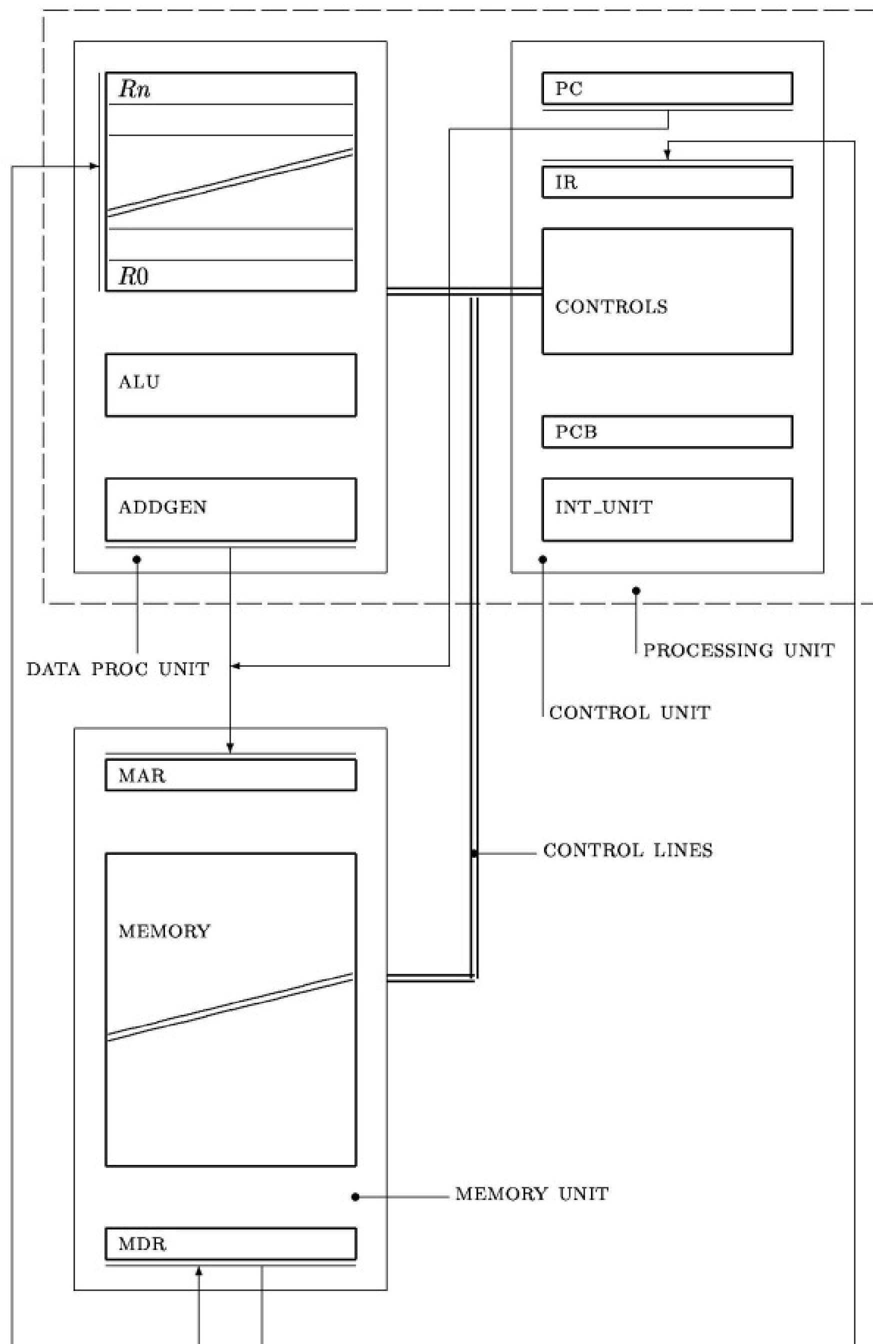
Korrekturmöglichkeiten:

- Bei schematischer Ausführung: Instruktionen in $IF/IA$, $ID/SRA$ und $EX/AG$, die `BR_CC` unmittebar folgen, werden ungültig, falls ein Sprung durchgeführt wird. Es entstehen also drei Bubbles, die der Compiler durch `NOP`s ausgleichen müßte.

- *branch shortcut*: In der $ID/SR$-Phase wird durch einen Addierer der $PC$ und das Displacement schon addiert (d.h. die branch-Adresse wird schon berechnet) und mit einer Kontroll-Logik werden die Statusregister der vorhergehenden Phase kombiniert abgefragt, so daß ggf. der neue $PC$ gesetzt werden kann mit nur einem Bubble.

# A Graphiken, Schemata etc.

## A.1 genereller Rechneraufbau



## A.2 SEMCD-Regelwerk

Returning from procedure calls

(0) $(S,\ E,\ M,\ nil,\ (E',\ C',\ D')) \rightarrow (S,\ E',\ M,\ C',\ D')$

Evaluating applications of closures

(1) $([\ E'\ \lambda^{(2)} u\ e_b\ ] : {e_a}^t : S,\ E,\ \overline{@}^{(2|0)} : M,\ C,\ D)$
$\rightarrow (S,\ < u\ e_a >: E', M,\ e_b : nil,\ (E,\ C,\ ; D))$

(2) $([\ E'\ \lambda^{(n-i+2)} u_i \dots u_n\ e_b\ ] : e_a^t : S,\ E,\ \overline{@}^{(j)} : M,\ C,\ D) \mid j = (n-i+2) > 2$
$\rightarrow (\ [ < u_i\ e_a\ >: E'\ \lambda^{(n-i+1)} u_{i+1} \dots u_n\ e_b\ ]\ :\ S,\ E,\ \overline{@}^{(j-1|0)} : M,\ C,\ D\ )$

(2a) $(\ [\ E'\ \lambda^{(n+1)} u_1 \dots u_n\ e_b\ ] : e_1^t : \dots : e_m^t : S,\ E,\ \overline{@}^{(m+1|0)} : M,\ C,\ D\ ) \mid n = m$
$\rightarrow (\ S,\ < u_n\ e_n >: \dots :< u_1\ e_1 >: E',\ M,\ e_b : nil,\ (\ E,\ C,\ D\ )\ )$

(2b) $(\ [\ E'\ \lambda^{(n+1)} u_1 \dots u_n\ e_b\ ] : e_1^t : \dots : e_m^t : S,\ E,\ \overline{@}^{(m+1|0)} : M,\ C,\ D\ ) \mid n \neq m$
$\rightarrow (\ error_message,\ nil,\ nil,\ nil,\ nil\ )$

Substituting bound variables while traversing expressions from $C$ to $S$

(3) $(S,\ E,\ nil,\ v : C,\ D)$
$\rightarrow (lookup(\ v,\ E\ ) : S,\ E,\ nil,\ C,\ D)$

(4) $(S,\ E,\ \overline{@}^{(n|i)} : M,\ v : C,\ D) \mid (i > 0)$
$\rightarrow (lookup(\ v,\ E\ ) : S,\ E,\ \overline{@}^{(n|i-1)} : M,\ C,\ D)$

Creating closures on $S$ for abstractions on $E$

(5) $(S,\ E,\ nil,\ \lambda^{(n+1)} u_1 \dots u_n\ e_b : C,\ D)$
$\rightarrow ([\ E\ \lambda^{(n+1)} u_1 \dots u_n\ e_b\ ] : S,\ E,\ nil,\ C,\ D)$

(6) $(S,\ E,\ \overline{@}^{(m|i)} : M,\ \lambda^{(n+1)} u_1 \dots u_n\ e_b : C,\ D) \mid (i > 0)$
$\rightarrow ([\ E\ \lambda^{(n+1)} u_1 \dots u_n\ e_b\ ] : S,\ E,\ \overline{@}^{(m|i-1)} : M,\ C,\ D)$

Traversing expressions from $C$ to $S$

(7) $(S,\ E,\ M,\ \kappa^{(n)} : C,\ D) \rightarrow\ (S,\ E,\ \kappa^{(n|n)} : M,\ C,\ D)$

(8a) $(S,\ E,\ nil,\ atom : C,\ D) \mid (i > 0) \rightarrow\ (atom : S,\ E,\ nil,\ C,\ D)$

(8b) $(S,\ E,\ \kappa^{(n|i)} : M,\ atom : C,\ D) \mid (i > 0)$
$\rightarrow\ (atom : S,\ E,\ ap^{(n|i-1)} : M,\ C,\ D)$

(9) $(S,\ E,\ \kappa^{(n|0)} : nil,\ C,\ D) \rightarrow\ (\kappa^{(n)} : S,\ E,\ nil,\ C,\ D)$

(10) $(S,\ E,\ \kappa^{(n|0)} : \kappa^{(m|i)} : M,\ C,\ D) \mid (i > 0)$
$\rightarrow\ (\kappa^{(n)} : S,\ E,\ \kappa^{(m|i-1)} : M,\ C,\ D)$

## A.3 Compiler MiniScheme → SECDH-Code

$\mathcal{C}[\, atom : es \,] \;\rightarrow\;$ PUSH_S $atom$; $\mathcal{C}[\, es \,]$;

$\mathcal{C}[\, \#i : es \,] \;\rightarrow\;$ PUSH_ES $i$; $\mathcal{C}[\, es \,]$;

$\mathcal{C}[\, (\,$ if $e_0\, e_1\, e_2\,) : es \,]$
$\rightarrow\; \mathcal{C}[\, e_0 \,]$; BRC $p_t\; p_f$; $\mathcal{C}[\, es \,]$;
$p_t \rightsquigarrow \mathcal{F}[\, e_1 \,]; \quad p_f \rightsquigarrow \mathcal{F}[\, e_2 \,];$

$\mathcal{C}[\, (\, e_0 \ldots e_{(n-1)}\,) : es \,] \;\rightarrow \mathcal{C}[\, e_n \,]; \mathcal{C}[\, e_{(n-1)} : \ldots : e_0 : ap(n) : es \,]$
$\mathcal{C}[\, (\, +\, e_1\, e_2\,) : es \,] \;\rightarrow \mathcal{C}[\, e_2 \,];\; \mathcal{C}[\, e_1 \,];$ PLUS; $\mathcal{C}[\, es \,]$

$\mathcal{C}[\, (\, -\, e_1\, e_2\,) : es \,] \;\rightarrow \mathcal{C}[\, e_2 \,];\; \mathcal{C}[\, e_1 \,];$ MINUS; $\mathcal{C}[\, es \,]$

$\mathcal{C}[\, \Lambda^{(r)} e_b : ap^{(r)} : es \,] \;\rightarrow$ CALL $p_{ff}; \mathcal{C}[\, es \,]$
$p_ff \;\rightsquigarrow\; \mathcal{F}[\, \Lambda^{(r)} e_b \,]\;;$

$\mathcal{C}[\, \Lambda^{(r)} e_b : es \,] \;\rightarrow$ PUSH_S $p_ff; \mathcal{C}[\, es \,]$
$p_ff \;\rightsquigarrow\; \mathcal{F}[\, \Lambda^{(r)} e_b \,];$

$\mathcal{C}[\, (\,$ define $ff\; e\,) : es \,] \;\rightarrow\; \mathcal{C}[\, es \,]; \quad p_ff \;\rightsquigarrow\; \mathcal{F}[\, e \,];$

$\mathcal{C}[\, (\,$ letrec $((\, ff_1\, e_1\,) \ldots (\, ff_k\, e_k\,))\; e_b\,) : es \,] \;\rightarrow\; \mathcal{C}[\, e_0 \,];\; \mathcal{C}[\, es \,];$
$p_ff_1 \;\rightsquigarrow\; \mathcal{F}[\, e_1 \,]; \ldots;\; p_ff_k \;\rightsquigarrow\; \mathcal{F}[\, e_k \,];$

$$\mathcal{F}[\, e \,] \;\rightarrow\; \begin{cases} (\, r,\; \texttt{MOVE_SE}\; r;\; \mathcal{C}[\, e_b \,];\; \texttt{FREE}\; r;\; \texttt{RET};\;) & \text{if } e = \Lambda^{(r)} e_b \\ (\, 0,\; \mathcal{C}[\, e \,];\; \texttt{RET};\;) & \text{otherwise} \end{cases}$$

## A.4 SECDH-Regelwerk

$$( S,\ E,\ \texttt{PUSH_S}\ atom : C,\ D,\ H\ ) \rightarrow (\ atom : S,\ E,\ C,\ D,\ H\ )$$

$$( S,\ se_1 : \ldots se_i : E,\ \texttt{PUSH_ES}\ i : C,\ D,\ H\ ) \\ \rightarrow (\ se_i : S,\ se_1 : \ldots : se_i : E,\ C,\ D,\ H\ )$$

$$(\ se_1 : \ldots se_n : S,\ E,\ \texttt{MOVE_SE}\ n : C,\ D,\ H\ ) \\ \rightarrow (\ S,\ se_n : \ldots : se_1 : E,\ C,\ D,\ H\ )$$

$$( S,\ se_1 : \ldots se_n : E,\ \texttt{FREE}\ n : C,\ D,\ H\ ) \rightarrow (\ S,\ E,\ C,\ D,\ H\ )$$

$$(\ num_1 : num_2 : S,\ E,\ \texttt{PLUS} : C,\ D,\ H\ ) \\ \rightarrow (\ sum(num_1, sum_2) : S,\ E,\ C,\ D,\ H\ )$$

$$(\ \#t : S,\ E,\ \texttt{BRC}\ p_t\ p_f : C,\ D,\ H[\ p_t \rightsquigarrow code_t\ ]\ ) \\ \rightarrow (\ S,\ E,\ code_t : nil,\ (\ C,\ D\ ),\ H\ )$$

$$(\ \#f : S,\ E,\ \texttt{BRC}\ p_t\ p_f : C,\ D,\ H[\ p_f \rightsquigarrow code_f\ ]\ ) \\ \rightarrow (\ S,\ E,\ code_f : nil,\ (\ C,\ D\ ),\ H\ )$$

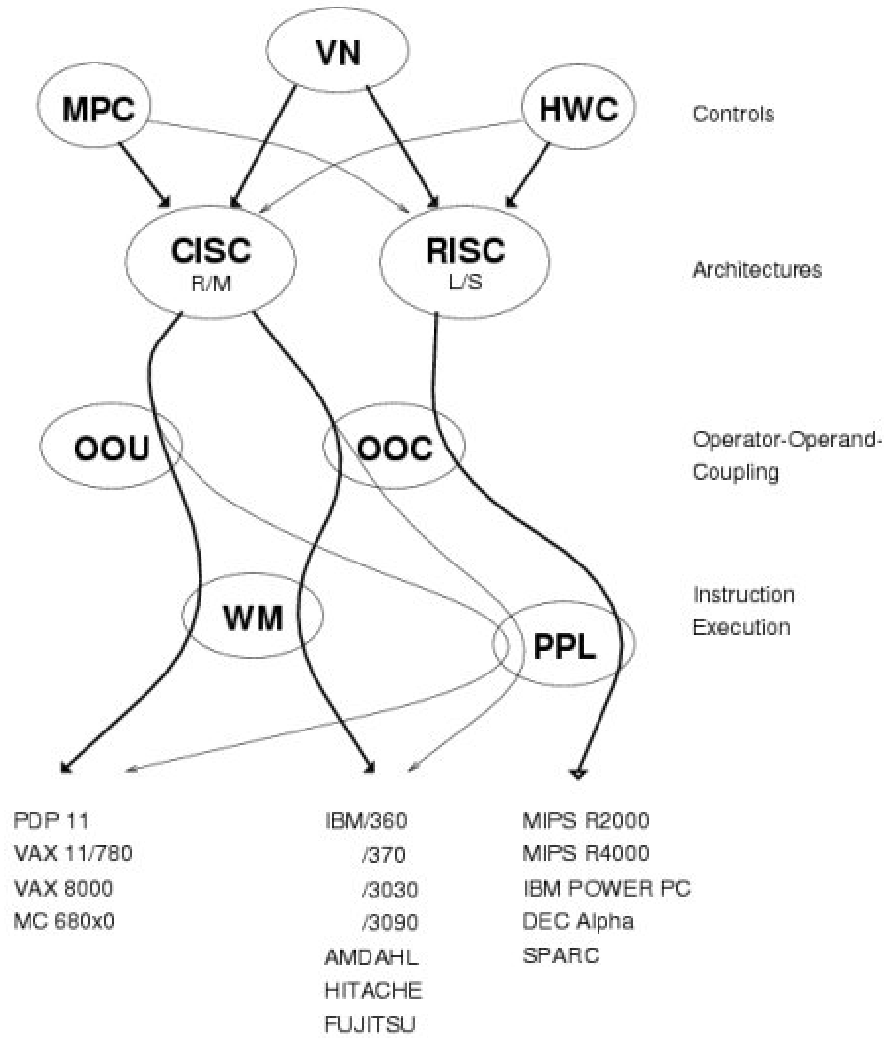
$$(\ S,\ E,\ \texttt{CALL}\ p_ff : C,\ D,\ H[\ p_ff \rightsquigarrow (\ r,\ code_ff\ )\ ]\ ) \\ \rightarrow (\ S,\ E,\ code_ff : nil,\ (\ C,\ D\ ),\ H\ )$$

$$(\ p_ff : S,\ E,\ \texttt{AP}\ r : C,\ D,\ H[\ p_ff \rightsquigarrow (\ r,\ code_ff\ )\ ]\ ) \\ \rightarrow (\ S,\ E,\ code_ff : nil,\ (\ C,\ D\ ),\ H\ )$$

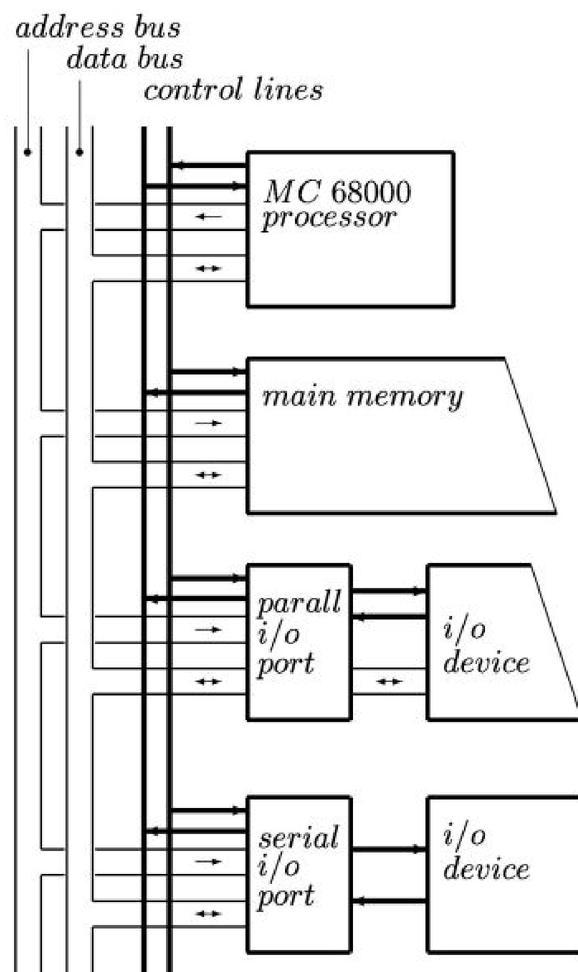
$$(\ prim_func^{(n)} : e_1 : \ldots e_n : S,\ E,\ \texttt{AP}\ n : C,\ D,\ H\ ) \\ \rightarrow (\ value : S,\ E,\ C,\ D\ ,\ H\ )$$

$$(\ S,\ E,\ \texttt{RET} : nil,\ (\ C,\ D\ ),\ H\ ) \rightarrow (\ S,\ E,\ C,\ D,\ H\ )$$

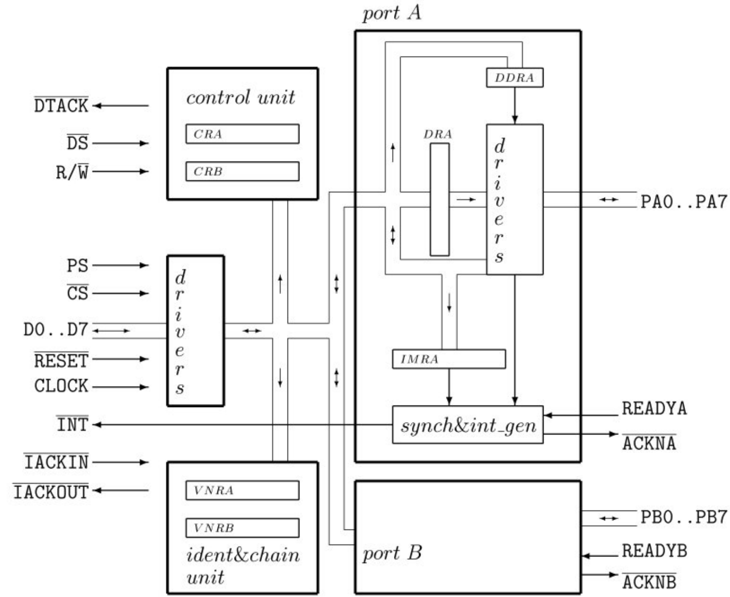
## A.5 Architekturen



## A.6 generelle Architektur des MC86K



## A.7 Paralleles Interface



## A.8 Serielles Interface